Une autre approche du C++

Philippe Dunski

Apprendre le développement en C++

Apprendre le développement en C++

Remerciements

Table des matières

Liste des illustrations

Liste des tableaux

Liste des exemples

Liste des équations

Partie I. Introduction

Dans cette partie, j'aborderai un cerain nombre de points qui n'ont en définitive qu'un rapport lointain avec la programmation en générale, et encore plus avec la programmation en C++ en particulier.

J'y aborderai en effet des notions comme les grands noms qui ont permis à l'informatique de devenir ce qu'elle est, les différents types et générations de langages ou les outils dont vous pourriez avoir besoin.

Le rôle de cette partie est donc essentiellement de vous permettre d'« élargir vos horizons », d'améliorer votre compréhension générale de ce qu'est un ordinateur et des problèmes particuliers que peut poser ce genre de machine.

Il n'est pas nécessaire de lire cette partie avant de lire les autres, mais il sera y sans doute très régulièrement fait référence par la suite. Vous pourrez donc lire les chapitres qu'elle contient au moment où les informations vous seront utiles.

Vous y trouverez notemment XXX pages, correspondant à YYY chapitres, répartis en neuf parties distinctes et contenant un nombre incalculable de références croisées.

Voici en quelques mots le sommaire de l'ouvrage que vous tenez entre les mains.

Chapitre 1. Demandez le programme

Si le but premier de cet ouvrage est de permettre au lecteur d'apprendre le C++ selon une approche moderne, je souhaitais en effet lui faire atteindre un niveau de connaissance qui lui permette d'atteindre une compréhension *correcte* – quoi que parfois basique – du langage et d'évoluer par la suite vers une maîtrise complete du langage.

Mais je voulais aussi limiter au maximum les prérequis nécessaires à cet apprentissage. J'ai donc du partir du principe que le lecteur ne connaissait *absolument rien*.

Or, quel que soit le langage de programmation utilisé, l'écriture du code peut à bien des égards être considéré comme un « long et fastidieux travail de dactylographie »; le vrai travail de développement se faisant normalement avant même d'écrire la première ligne de code.

Et c'est particulièrement vrai en ce qui concerne le C++, car il s'agit d'un langage très permissif, qui vous autorise très régulièrement à charger un fusil et à le pointer, doigt sur la gâchette, ne serait-ce que métaphoriquement parlant. Au risque, bien sûr... de vous tirer une balle dans le pied.

La connaissance et la mise en œuvre de principes communs à la plupart des langages et aux différents paradigmes (en plus de quelques spécificités propres au C++) s'avère indispensable en C++, si l'on souhaite obtenir un résultat de qualité.

Et, bien sûr, je ne voualis en aucune manière ne serait-ce qu'envisager l'idée de ne pas tout faire pour permettre au lecteur d'atteindre un niveau de qualité « correct » (« correct » pouvant ici être traduit par « excellent », car c'est le seul niveau de qualité que j'accepte personnellement).

Il m'a donc parru tout ce qu'il y a de plus logique de fournir au lecteur – y compris au débutant qui n'a aucune connaissance préalable – tout ce qui lui permettrait d'atteindre un tel niveau qualité dans ses développements.

Cet ouvrage est donc divisé en neuf parties, que l'on pourrait résumer comme suit:

- La première partie (Introduction, que vous lisez actuellement) aborde une série d'aspects et de notions destinés à votre « édification personnelle »;
- La deuxième partie (Principes communs) aborde les principes de base communs à n'importe quel langage qualifié de « procédural »;
- la troisième partie (La syntaxe de base) présente la syntaxe de base de C++ ainsi que la mise en œuvre des principes de base dans ce langage;
- La quatrième partie (Les principes de l'orienté objet) aborde le paradigme orienté objet du point de vue théorique, ce qui en fait une introduction cohérente à tout langage supportant ce paradigme;
- la cinquième partie (C++ et l'orienté objet) présente la manière dont C++ met le paradigme orienté objet en œuvre;
- la sixième partie (Le paradigme générique) présente le paradigme générique ainsi que l'approche « fonctionnelle » du point de vue théorique;

- la septième partie (C++ et généricité)présente la manière dont C++ met le paradigme générique en œuvre, ainsi que la manière dont il permet le cas échéant d'avoir une approche fonctionnelle.
- la huitième partie (La bibliothèque standard) présente quelques fonctionnalités issues de la bibliothèque standard qui n'ont pas été abordées ailleurs;
- la neuvième partie (Annexe) contient un certain nombre d'informations utiles qui n'auront pas trouvé place ailleurs

Que lire, et dans quelle ordre?

Cet ouvrage pourrait à juste titre être considéré comme un recueil de « fiches thématiques » dédié au développement en général et au développement en C++ en particulier.

Il serait donc sans doute vain d'essayer de le lire d'une traite, de la première à la dernière page, bien qu'il soit – malgré tout – organisé de manière à permettre à qui le souhaiterait de le lire de la sorte .

La meilleure manière de lire ce livre, c'est de s'en servir comme d'une référence « absolue » en ce qui concerne le domaine de la programmation, et donc, d'aller « piocher » dans les différentes sections ce qui vous intéresse au moment où vous en avez besoin.

Les sections suivantes visent néanmoins à vous proposer un ordre de lecture cohérent en fonction de votre niveau et de vos objectifs.

Je pars de rien

Si votre objectif est d'apprendre le C++ alors que vous n'avez strictement aucune connaissance en programmation, je vous conseille de commencer par les quatre premières sections du chapitre Les outils du développeur, puis de lire les parties :

- 2 :Principes communs,
- 3 :La syntaxe de base,
- 4 (Les principes de l'orienté objet)
- , 5 (C++ et l'orienté objet),
- 6 (Le paradigme générique) ,
- 7 (C++ et généricité)
- et 8 (La bibliothèque standard)

Quant aux première (Introduction) et neuvième (Annexe) parties, vous y passerez focément si vous suivez les références croisées que vous trouvez, mais, si vous avez « du temps à perdre » et que vous voulez améliorer votre culture personnelle, vous pouvez les lire à n'importe quel moment.

Je veux améliorer ma culture personnelle

Si votre seul objectif – en dehors de disposer d'une référence en français s'entend – en aquérant cet ouvrage est votre édification personnelle, je vous conseillerais de lire en priorité les première (Introduction) et neuvième (Annexe).

Elles traitent de sujets divers et variés qui tournent autour de la notion de « développement informatique ».

Je connais d'autres langages

Si vous connaissez déjà d'autres langages, certaines parties purement théoriques n'auront sans doute qu'un intérêt très limité pour vous:

- la deuxième partie ne vous apprendra sans doute rien de nouveau;
- la quatrième partie (Les principes de l'orienté objet) ne vous apprendra sans doute quelque chose que si vous n'avez jamais travaillé avec un langage orienté objet

Par contre, la sixième partie, bien que purement théorique, sera sans doute nécessaire pour vous permettre de comprendre la septième. En effet, même les generics de java sont très loin de proposer l'ensemble des possibilité offertes par le paradigme générique.

Pour le reste, l'ordre défini plus haut devrait également vous convenir.

Mon projet prend de l'ampleur

Nous atteindrons rapidement un point auquel, même les petits programmes « jetables » que je vous présenterai nécessiteront de faire appel à plusieurs fichiers.

Il sera alors temps de s'intéresser aux sections 5(Automatiser la compilation) et 6 (Automatiser la configuration d'un projet) du chapitre Les outils du développeur.

Dans le même temps, vous serez sans doute tentés par le fait de développer vos propres projets qui commenceront à prendre de l'ampleur.

Peut-être travaillerez-vous en équipe sur un projet, que vous l'ayez initialisé ou non. La section 7 (Gérer différentes versions) du chapitre Les outils du développeur vous viendra surement bien en aide.

Enfin, l'étape finale consistera sans doute à diffuser votre projet, et vous serez donc des plus intéressés par la section 8 (Les outils d'intégration continue) du même chapitre.

Des outils plus performants

On pourra dire ce que l'on veut, les outils que je vous proposent ont beau être performants, certains ont beau être hautement configurables, le fait que tous les outils soient "disséminés un peu partout" est très souvent perçu comme « un problème ».

C'est, bien sur, une pure question de gouts: certains développeurs aiment autant utiliser des outils très spécifiques, quitte à « passer de l'un à l'autre » en permanence, et d'autres préfèrent avoir des outils « centralisés », de préférences, susceptibles d'être utilisés en grosse partie avec la souris.

Je me garderai bien d'émettre la moindre opinion sur le sujet, car je n'ai aucune raison de le faire. Mais j'en suis bien conscient: certains lecteurs en auront rapidement marre de jongler avec des outils disséminés un peu partout et souhaiteront quelque chose de peu plus performant.

Il sera alors temps pour eux, quelque soit le moment où cela peut leur arriver, de s'intéresser à la section 9 (Les environnements de développement intégré) du chapitre Les outils du développeur.

Chapitre 2. L'algèbre de Boole

Selon wikipedia, ce que l'on désigne sous le terme d'*algèbre de Boole* représente

> L'algèbre de Boole, ou *calcul booléen*, est la partie des mathématiques qui s'intýresse à une approche algébrique de la logique, vue en termes de variables, d'opérateurs et de fonctions sur les variables logiques, ce qui permet d'utiliser des techniques algýbriques pour traiter les expressions à deux valeurs du calcul des propositions.
>
> —sources wikipedia

Cette approche fut lancée en 1854 par le mathématicien, logicien et philosophe George Boole, dont elle porte le nom en son honneur

Deux valeurs

L'algèbre de Boole s'intéresse à un ensemble (au sens mathématique du terme) appelé B et ne contenant que deux valeurs: la valeur « vrai » et la valeur « faux ».

Chapitre 3. Merci monsieur Von newman

Chapitre 4. C++? KESSAKO?

Si cet ouvrage a atterri dans vos mains, c'est parce que vous voulez apprendre ou vous perfectionner en C++. Mais j'estime personnellement qu'il est impossible de comprendre un sujet, d'arriver à le maîtriser, si l'on n'a pas déjà une vision -- serait-elle générale -- du contexte dans lequel le sujet en question évolue.

Un homme sans histoire est un homme sans avenir car il se condamne à reproduire les erreurs du passé dit-on souvent. C++ a -- comme bien d'autres langages de programmation -- tiré les leçons du passé pour se forger sa propre philosophie.

Si vous voulez apprendre C++, vous devez *ad minima* savoir ce que c'est. Si vous espérez un jour atteindre un niveau élevé de maîtrise de C++, vous devrez en comprendre la philosophie propre. Et vous ne pourrez obtenir cette compréhension qu'en comprenant le contexte dans lequel il a vu le jour et les défis qu'il a décidé de relever.

Mettons donc directement fin au suspense. C++ est ce que l'on appelle un « langage de programmation »« compilé », « multi paradigme » dit « de troisième génération ».

Ce chapitre revient en détail sur chacun des termes utilisés afin de vous permettre de comprendre exactement de quoi il retourne.

Un langage de programmation

Vous l'aurez compris, C++ n'est très certainement pas une marque de voiture ni le titre d'un livre de recettes de cuisine. C'est ce que l'on appelle un *langage de programmation*. Cela implique, d'abord et avant tout qu'il s'agit d'un langage.

Un langage n'est rien de plus qu'un ensemble de conventions qui permettent à deux interlocuteurs de se comprendre lorsqu'ils tentent de communiquer entre eux.

Ces conventions peuvent prendre plusieurs formes allant du terme utilisé pour représenter quelque chose à la syntaxe utilisée, en passant par l'ordre dans lequel les différents termes sont "mis en phrase". C'est ainsi que si vous voulez une cuillère, vous parlerez de *lepel* en nérlandais ou de *spoon* en anglais et que vous devrez placer les différents termes qui correspondent au sujet, au verbe et aux attributs dans un ordre bien déterminé afin de faire une phrase qui respecte les conventions propre à la langue que vous parlez.

La seule différence entre n'importe quelle langue comme le français, l'espagnol ou l'anglais et C++, c'est que l'un des interlocuteur est une machine. Toutes les conventions que nous utiliserons aurons donc pour objectif de nous faire comprendre par "quelque chose d'aussi bête qu'un ordinateur".

Un langage compilé

Nous le verrons plus loin, un ordinateur n'est capable de comprendre qu'un nombre limité d'instructions et uniquement lorsqu'elles prennent la forme d'une succession de 1 et de 0.

Il est possible de distinguer deux grandes catégories de langages de programmation en fonction de la manière dont les différentes instructions sont converties en un succession de 1 et de 0 compréhensible par le processeur: Les langages interpretés d'une part et les langages compilés de l'autre.

Les langages interprétés sont des langages dont chaque exécution provoquera la conversion « à la volée » de l'ensemble du code en instructions compréhensibles par le processeur, alors que le code écrit dans un langage compilé sera converti une bonne fois pour toutes en un ensemble d'instructions et sauvegardé sous la forme d'un fichier exécutable.

L'application qui permet d'interpréter du code écrit dans un langage interprété s'appelle, très logiquement, un interpréteur alors que l'application permettant de créer un exécutable à partir d'un code écrit dans un langage compilé s'appelle -- tout aussi logiquement -- un compilateur.

Les langages "semi compilés"

Certains langages estiment, à plus ou moins juste titre, que ces deux catégories souffrent d'inconvénients majeurs et tentent de concilier ces deux aspects antagonistes afin d'en ressortir le meilleur des deux côtés.

L'idée générale est de convertir le code source écrit par le développeur en un langage intermédiaire, nommé *byte code* qui sera interprété à l'exécution par ce que l'on appelle alors une *machine virtuelle*. Java et le framework *.net* sous windows sont basés sur de telles machines virtuelles.

On parle alors de langage *semi compilé* -- ou de langage *semi interprété* selon le point de vue -- pour indiquer que « la moitié du travail » a été effectuée avant l'exéctution mais que l'autre moité devra être faite plus tard.

Des avantages et des inconvénients

Ces catégories présentent toutes les deux des avantages et des inconvénients.

Le pire de l'histoire étant qu'un avantage de l'une correspond souvent à un inconvénient de l'autre et que, chacun voyant midi à sa porte, les avantages cités par certains peuvent très facilement être considérés comme des inconvénients par d'autres.

Les langages interprétés

Le propre des langages interprétés est qu'il n'est pas nécessaire de générer le code binaire exécutable pour être en mesure d'utiliser les programmes qu'ils décrivent. Cela se traduit par deux avantages principaux:

- Le code est directement utilisable sur n'importe quelle architecture, sur n'importe quel système d'exploitation aussi longtemps que le système sur lequel il est exécuté dispose de l'interpréteur adapté au langage utilisé.
- Le développement et les corrections sont souvent plus rapides car il n'est pas nécessaire d'attendre que l'ensemble du processus de compilation n'ait été exécuté.

Par contre, les langages interprétés ne sont malgré tout pas exempts d'inconvénients.

Aussi bizare que cela puisse parraître, la raison est exactement la même que pour les avantages : il n'est pas nécessaire de générer le code binaire exécutable afin d'être en mesure d'utiliser les programmes qu'ils décrivent. Cela se traduit alors par plusieurs inconvénients majeurs :

- L'impossibilité d'exécuter le programme sans recourir à un interpréteur ou à une machine virtuelle. Cette restriction est particulièrement importante si vous devez développer un programme qui, pour une raison ou une autre, ne peut pas attendre

que la machine virtuelle ou l'interpréteur soit disponible (tout ce qui a trait au "noyau" du système d'exploitation, par exemple) ;

- Le « passage obligé » par l'interpréteur afin de fournir au processeurs les instructions qu'il doit exécuter. Cette iterprétation du code fait que les instructions seront rarement optimisées pour une architecture particulière et prend généralement du temps. Il n'est donc pas rare que les langages interprétés soient « relativement plus lents » à l'exécution que les langages compilés.
- L'interpréteur ou la machine virtuels sont indispensables pour l'exécution des programmes.

Les langages compilés

Ce ne sera sans doute une surprise pour personne, le propre des langages compilés est qu'il impliquent de générer le code binaire exécutable avant de pouvoir utiliser les programmes qu'ils décrivent.

Cela se traduit par plusieurs avantages majeurs :

- Le programme est exécutable sans qu'il soit besoin d'installer « autre chose » que le programme lui-même et ses éventuelles dépendances
- Comme le programme est directement composé des instructions que le processeur devra exécuter, les programmes issus de langages compilés sont globalement plus rapides à l'exécution que ceux issus des langages interprétés.
- Les compilateurs sont souvent en mesure de fournir un code binaire exécutable utilisant des instructions processeur spécifiques à l'architecture ciblée. L'« optimisation » qu'implique l'utilisation d'instructions spécifiques tend, là aussi, à rendre le programme globalement plus rapide.

Mais ces avantages ne doivent pas occulter les inconvénients pour autant.

La principale raison de ceux-ci sera, une fois encore, strictement la même : les programmes décrits par les langages compilés nécessitent la génération du code binaire exécutable avant de pouvoir être utilisés.

Cela se traduit par deux inconvénients majeurs:

- Pour être en mesure d'utiliser un programme sur une architecture ou avec un système d'exploitation donné, il faut généralement que le programme ait été compilé spécifiquement pour l'architecture et / ou pour le système d'exploitation visé: un programme compilé pour être utilisé sous windows ne pourra – par exemple – en aucun cas être utilisé sous linux sans passer par l'étape de compilation au préalable. L'inverse est d'ailleurs tout aussi vraie.
- La génération du code binaire exéctutable est un processus qui prend du temps et qui doit impérativement être exécuté dés que l'on a effectué le moindre changement dans le code. L'apport de correction au code peut donc demander d'avantage de temps si plusieurs modifications de code s'avèrent nécessaires avant d'arriver à apporter la correction adéquate.

Un langage multiparadigme

Selon wikipedia un paradigme est

> une représentation du monde, une manière de voir les choses, un modèle cohérent de vision du monde qui repose sur une base définie (matrice disciplinaire, modèle théorique ou courant de pensée). C'est

> une forme de rail de la pensée dont les lois ne doivent pas être confondues avec celles d'un autre paradigme et qui, le cas échéant, peuvent aussi faire obstacle à l'introduction de nouvelles solutions mieux adaptées.
>
> —wikipedia

Si cette définition s'applique parfaitement à des langages comme java ou C# qui ont décidé de respecter -- parfois en dehors de toute logique -- intégralement le paradigme orienté objet, elle mériterait amplement d'être quelque peu adoussie lorsqu'il est question de C++.

Avec une définition tellement restrictive, il serait en effet assez difficile d'envisager la possibilité de disposer de plusieurs paradigmes au sein d'un seul et unique langage car, si on en croit la définition, le passage d'un paradigme à l'autre n'irait jamais sans poser un certain nombre de problèmes.

De deux choses l'une. Ou bien cette définition est fausse, du moins en partie, car trop restrictive, ou bien C++ doit être considéré plutôt comme un langage « multi points de vue » que comme un langage multiparadigme.

Cette réflexion ne vient pas de moi, je dois l'avouer. Je l'ai allègrement adaptée de Bjarne Stroutrup (le créateur originel de C++) qui semble ne réellement pas apprécier que l'on parle de « son » langage comme d'un langage multiparadigme.

Et j'ai personnellement tendance à me rallier à son point de vue.

Mais bon, en attendant de trouver une meilleure formule, continuons donc à parler de C++ comme d'un langage multiparadigme, en adoucissant la définition du terme sous une forme qui parler d'un « schéma de pensée, d'une manière d'aborder le monde qui nous entoure ».

Car, si l'on admet une telle définition, C++ entre bel et bien dans la catégorie des langages dits « multiparadigme » (si tant est qu'il n'en soit pas le seul représentant). Voici ceux auxquels il donne acces.

Le paradigme impératif

Le paradigme impératif est sans doute le paradigme le plus ancien à avoir été mis en œuvre en informatique.

Il considère qu'un programme n'est en définitive qu'une suite d'instructions qu'il faut effectuer dans un ordre bien précis afin d'obtenir un résultat reproductible et clairement identifié.

Il faut bien sûr s'entendre sur le sens que l'on donne au terme instruction, car une instruction peut en réalité s'avérer être un bloc d'instructions exécutées dans un ordre précis qui sera rerpésenté sous la forme d'une fonction, par exemple.

Chaque instruction correspond plus ou moins à une manipulation effectuée sur une donnée particulière.

Ce paradigme fera l'objet d'une étude approfondie dans la partie Principes communs.

Le paradigme orienté objet

Le paradigme orienté objet est en grande partie basée sur le paradigme impératif avec malgré tout une différence majeure : au lieu de réfléchir en termes de données manipulées, nous réfléchirons en termes de services attendus de la part des données que l'on manipule, bien que ce ne soit pas -- très loin de là -- sa caractéristique majeure.

Ce paradigme s'intéresse en effet beaucoup plus aux fonctions que l'on peut appeler depuis une donnée qu'à la manière dont les données sont réellement manipulées. Il fera l'objet d'une étude approfondie dans la partie Les principes de l'orienté objet.

Le paradigme générique

Enfin, C++ est un langage qui permet l'utiisation du paradigme générique. Ce paradigme respecte un schéma de pensée totalement différent dans le sens où il est possible de ne pas savoir quel type de donnée nous manipulerons réellement, mais de savoir par contre parfaitement la manière dont ces données seront manipulées.

Cette approche permet d'éviter de nombreuses copies inutiles (et sources d'erreurs) de code. Elle fera l'objet d'une étude aprofondie dans la partie Le paradigme générique.

Une intégration parfaite

Si j'ai tendance à être d'accord avec Stroutroup quant à l'usage du vocable « langage multiparadigme » vis à vis de C++, c'est essentiellement parce que, contrairement à ce que laisse entendre la définition fournie par wikipedia, il est tout à fait possible d'intégrer dans un seul langage différents paradigmes en faisant en sorte qu'il soit facile de passer de l'un à l'autre et même d'utiliser une fonction ou une structure de données issue de l'un dans un contexte dans lequel on en utilise clairement un autre.

C++ a réussi cet exercice avec brio, car il est tout aussi facile d'invoquer une fonction générique depuis une fonction libre (issue du paradigme procédural) que depuis la fonction membre d'une classe (issue du paradigme Orienté Objet). A vrai dire, il est tout à fait possible d'utiliser n'importe quelle structure de donnée ou fonction issue de n'importe quel paradigme dans le contexte d'utilisation de n'importe quel autre paradigme.

C'est sans doute ce qui rend le C++ si complexe, mais c'est aussi ce qui le rend si puissant. Contrairement à ce que l'on pourrait croire, l'intégration de ces trois paradigmes fait effectivement de C++ un des langages parmis les plus complexes, mais vous ne manquerez pas de le constater au travers de cet ouvrage, cela n'en fait pas forcément un langage compliqué pour la cause.

les générations de langage

La communication entre l'homme et la machine a toujours posé problème, car nous ne sommes – décidément – vraiment pas « faits pareils »: l'homme est « quelque chose » de complexe, capable de comprendre un grand nombre de mots, et de subtilités diverses, alors que la machine, elle, elle ne fonctionne que grâce au courant électrique, et elle ne connaît finalement que deux états: le courant passe, ou il ne passe pas.

Il n'y a aucune nuance dans ces deux états, car il est impossible de faire passer « mais pas trop » le courant dans les circuits de la machine.

Pour nous faire comprendre par la machine, nous avons donc du mettre au point des langages que la machine serait en mesure de comprendre, malgré son manque de nuances.

Bien sur, les langages mis au point ont évolué avec le temps, et l'on peut désormais les classer en trois catégories distinctes que voici:

Le langage machine

L'informatique telle que nous la connaissons aujourd'hui a pris véritablement son essors lorsque l'idée de Von Newman [1] a été mise en œuvre.

Les choses sont devenues « plus faciles » que ce qu'elles étaient avant car ce que l'on pouvait désormais appeler « ordinateurs » (bien qu'il semble que le terme ne soit apparu que plus tard) disposaient désormais d'un certain nombre d'instructions auxquelles nous pouvions faire appel.

Selon l'idée de Von Newman, chaque instruction est représentée par une valeur numérique qui lui est propre, mais qui ne peut – bien évidemment – être comprise par l'ordinateur que si elle est représentée sous une forme binaire.

Il était désormais possible d'écrire des programmes sous une forme proche de

Exemple 4.1. un programme en langage machine

```
0010001000000100
0010010000000100
0001011001000010
0011011000000011
1111000000100101
0000000000000101
0000000000000110
0000000000000000
```

Dans ce genre de programme, chaque ligne de code est composée de seize bits pouvant soit représenter une instructions, soit une valeur simple.

Dans cet exemple, les trois dernières lignes représentent respectivement les valeurs 5, 6 et 0, sous une notation Ã seize bits avec complément Ã deux. Les cinq première lignes de code représentent quant Ã elle les instructions qui devront être exécutées.

Pour pouvoir interpréter ce code, l'opérateur (et l'ordinateur) devait interpréter les quatre premiers bits de chaque ligne comme un *opcode* représentant l'instruction Ã effectuer, les douze derniers bits contenant les valeurs les valeurs associée aux opérandes de l'opération.

Ces opérandes pouvaient correspondre soit Ã un des registres utilisé par la machine pour manipuler les données, soit Ã une adresse mémoire Ã laquelle la machine pourrait trouver une donnée Ã traiter.

Ainsi, la première ligne de ce code `0010001000000100` contient un *opcode*(`0010`) et deux opérandes.

L'*opcode* correspond Ã l'instruction « copie le contenu de l'adresse mémoire indiquée dans le registre de travail indiqué ». Pour savoir de quel registre de travail nous parlons, il fallait observer les trois bits suivants (`001`). Les neufs derniers bits représentant l'adresse mémoire dont il fallait copier le contenu dans le registre de travail.

Et, bien sur – où aurait été le plaisir autrement? – même si tous les ordinateurs disposent sans doute d'instructions identiques (comme « ajouter une valeur à ce qui se trouve dans le registre A »), le code correspondant à chaque instruction peut tout à fait varier d'un ordinateur à l'autre.

[1] voir: merci Mr Von Newman

Le langage machine est depuis considéré comme la première **génération de langage** .

les mnemoniques

Vous l'aurez compris, le langage machine a apporté « une certaine facilité » à la programmation, mais il reste malgré tout très sujet à l'introduction d'erreur: il suffit en effet de se tromper et de mettre un `1` là où il aurait fallu un `0` (ou l'inverse) pour que, dans le meilleur des cas, une valeur incorrecte soit utilisée ou, dans le pire des cas, pour que ce soit carrément la mauvaise instruction qui soit utilisée.

Les développeurs de l'époque mirent donc au point des outils capables de comprendre un certains nombre de symboles et de les traduire en langage machine.

Nous pouvions désormais utiliser des instructions comme `ADDA <valeur>` en éyant la certitude que ce serait traduit en langage machine sous la forme de l'instruction correspondant à « ajouter telle valeur au registre A ».

Ils nommèrent *mnémonique* les instructions comme `ADDA` , et il en existait – vous vous en seriez douté – un grand nombre d'autres.

Ces mnémoniques, que l'on regroupe volontiers dans le vocable de *langage assembleur* représente ce qu'il est convenu d'appeler un *langage de deuxième génération*.

Quant à l'outil qu'ils mirent au point pour automatiser cette « traduction », il le nommèrent *assembleur*.

les langages de « proches du langage humain »

Le langage assembleur avait « ouvert la voie »: grâce à lui, il devenait possible de « s'abstraire » de certains détails propres à la machine sur laquelle on travaille (comme l'*opcode* spécifique à une instruction particulière), mais reste malgré tout très proche de ce que ce que l'on peut désormais appeler un ordinateur est capable de comprendre.

L'évolution suivante a consisté à créer ce que l'on appelle couremment des langage proches du langage humain; c'est à dire des langages qui s'éloignent encore un peu plus (on dit généralement « qui font un peu plus abstraction ») de la représentation des données et des instructions telles qu'elles sont comprise par l'ordinteur pour se rapporcher d'une représentation « facilement compréhensible », qui peut facilement être lue par un humain.

Cette nouvelle évolution a donné naissance aux langages dont on dit généralement qu'ils font partie des *langages de troisième génération*.

Plusieurs approches

Des divergences d'opinion sont très vite apparues, à partir du moment où il devenait possible d'avoir des langages de programmation « proches du langage humain », quant à l'idée que se faisaient les gens de ce que devait permetre un tel langage.

Cretaines personnes estimaient en effet que le seul langage réellement universel est le langage des mathématiques, et qu'un langage de programmation doit donc permettre d'exprimer les choses sous la forme de fonction mathématiques.

D'autres estimaient qu'un langage de programmation devait pouvoir exprimer les différentes étapes par lesquelles il est nécessaire de passer pour obtenir un résultat prévisible et reproductible.

Les premiers ont donc mis au point des langages dont on dit désormais qu'ils favorisent la *programmation fonctionnelle*, alors que les second mettaient au point des langages dont on dit désormais qu'il favorient la *programmation Séquentielle*.

Il va de soi que ces deux approches ont occasionné de nombreuses « guerres de clochers » et qu'il a longtemps été considéré comme impensable de les concilier.

Cependant, chaque approche a fini par montrer ces limites, si bien que l'on en arrive de plus en plus souvent à observer l'arrivée « sur le marcher » de langages qui tente de concilier ces deux approches de manière à « profiter du meilleur des deux mondes ».

Le langage C et le C++ ont clairement fait le choix de favoriser la programmation impérative. Cependant, les évolutions récentes de C++ tendent de plus en plus à intégrer la capacité d'utiliser la programmation fonctionnelle.

Chapter 5. Les outils du développeur

Du papier et un bic

Un cerveau

Un éditeur de texte plat

Une chaine de compilation

Automatiser la compilation

Automatiser la configuration d'un projet

Gérer différentes versions

Les outils intégration continue

Les environnements de développement intégré

Chapter 6. Les bases de calcul

Lorsqu'il s'agit de représenter des valeurs numériques, nous sommes particulièrement habitués à les représenter à l'aide de dix “graphes” que nous appelons généralement “les chiffres” qui vont – comme tout le monde le sait – de 0 à 9 inclus.

Ce qu'il faut bien comprendre, c'est que la manière de représenter une valeur numérique à l'aide de ces “glyphes” n'est jamais qu'une convention “parmis tant d'autres” qui puisse être utilisée pour la représentation des valeurs numérique.

Nous pourrions, en effet, définir un ensemble composé d'un nombre quelconque de “glyphes”qui seraient utilisés selon des règles identiques mais qui permettrait la représentation de n'importe quelle valeur.

Pensez, par exemple, au système horraire, dans lequel une heure est composée de … 60 minutes, et dans lequel une minute est composée de … 60 secondes.

Au lieu d'utiliser la notation décimale qui nous est bien connue, nous pourrions parfaitement définir un ensemble de … 60 glyphes différents qui nous permettrait de représenter les minutes et les seconde à l'aide d'un seul glyphe, mais avec lequel nous pourrions néanmoins utiliser la représentation `¢% minutes` par exemple serait parfaitement capable de représenter, ¢ heures et %minutes.

La notation décimale

Ce que l'on appelle couramment la *notation décimale* (ou, plus simplement, la *base dix*) correspond aux conventions que nous apprenons à utiliser depuis notre plus jeune âge pour représenter des valeurs numériques.

Le terme *décimal* – et surtout le terme *dix* venant du fait que l'on dispose de … dix “symboles” – couramment appelés des “chiffres” – pour représenter “n'importe quelle valeur numérique”.

Pour représenter des valeurs plus grandes que neuf, nous allons “tout simplement” placer les différents chiffres l'un à coté de l'autre et “adapter” la valeur de chaque chiffre en fonction de sa position dans le nombre représenté.

Ainsi, si nous voulons représenter la valeur numérque quatre-cent nonente sept, nous la représenterons sous la forme du nombre 497, et nous suivrons une logique proche de

- `4` est à la troisième position, sa valeur est donc de $4 * 10^2$
- `9` est à la deuxième position, sa valeur est donc de $9 * 10^1$
- `7` est à la première position, sa valeur est donc de $7 * 10^0$
- la somme correspond donc à $4 * 10^2 + 9 * 10^1 + 7 * 10^0$

A vrai dire, nous sommes tellement habitué à travailler de la sorte que nous le faisons pour ainsi dire “naturellement”, sans même avoir besoin d'y penser.

Mais, si nous voulions exprimer le raisonnement dans son intégralité, nous pourrions dire que

> La valeur d'un chiffre est égal à la multiplication de sa valeur naturelle par la base utilisée élevée à la puissance représentée par sa position moins un

D'une manière plus mathématique, nous pourrions exprimer ce même raisonnement sous une forme proche de

Equation 6.1. Calculer une valeur dans différentes bases

$V = S * B^{P-1}$

avec

- V : valeur réelle
- S : valeur du symbole
- B : base utilisée
- P : position du chiffre dans le nombre

Note

Nous verrons plus loin que la valeur 10 n'as absolument pas été choisie au hasard.

La notation binaire

Depuis “toujours”, lorsque l'on travaille avec de l'électricité, nous n'avons finalement que peu de choix: soit le courant passe “quelque part”, soit il ne passe pas.

Pour représenter n'importe quelle valeur numérique à l'intérieur d'un ordinateur, nous ne pouvons donc disposer que de deux “symboles”, qui seront par convention le `0` pour représenter le fait que le courant ne passe pas et le `1` pour représenter le fait que le courant passe.

Je parlerai souvent – en abusant du terme – de *bit* pour désigner les symboles utilisés en notation binaire, car il s'agit du terme généralement utilisé pour désigner, au niveau de l'ordinateur, l'élément qui permet de se rendre compte que le courant passe ou non.

Mais alors, vous demanderez vous peut-être, comment est il possible de représenter des valeurs tellement importantes?

La réponse à cette question est toute simple: il suffit de respecter exactement la même équation que celle que l'on suit en base dix. La seule différence, c'est que, au lieu de multiplier la valeur du symbole par 10^{P-1}, nous allons la multiplier par 2^{P-1}.

Si bien que nous pouvons tout à fait représenter la valeur quatre-cent nonente sept en binaire sous la forme de `1 1111 0001`.

En effet, si nous suivons le même raisonnement, nous nous rendons compte que

- le `1` qui se trouve à la neuvième position correspond à une valeur égale à $1 * 2^8$, soit `256` en décimal
- le `1` qui se trouve à la huitième position correspond à une valeur égale à $1 * 2^7$, soit `128` en décimal
- le `1` qui se trouve à la septième position correspond à une valeur égale à $1 * 2^6$, soit `64` en décimal

- le `1` qui se trouve à la sixième position correspond à une valeur égale à `1 * 2`^{5}, soit `32` en décimal
- le `1` qui se trouve à la cinquième position correspond à une valeur égale à `1 * 2`^{4}, soit `16` en décimal
- le `1` qui se trouve à la cinquième position correspond à une valeur égale à `1 * 2`, soit `1` en décimal
- `256 + 128 + 64 +32 +16 + 1 = 497`

La difficulté du binaire

Le gros problème, lorsqu'il s'agit de représenter une valeur numérique, c'est que quel que soit la base utilisée, le risque de faire une erreur lors de la retranscription de cette valeur augmente énormément avec le nombre de symboles nécessaire à cette représentation.

Il n'y a en effet pas grand chose à faire: si vous voulez utiliser trois symboles (décimaux) pour représenter une valeur numérique comprise dans l'intervalle autorisé par ces trois symboles (typiquement l'intervalle [0,999[), vous avez logiquement le choix entre... mille valeurs possibles.

Mais sur ces mille valeurs possibles, il n'y en aura jamais qu'une seule qui correspondra effectivement à la valeur que vous voulez représenter. Ce qui laisse neuf-cent nonente neuf valeurs erronées.

Et même si on décide qu'il n'y a qu'une alternative dans le sens où soit, nous utilisons le bon symbole à la bonne place, soit nous utilisons un symbole incorrect à une place particulière, le fait d'utiliser trois symboles nous donne trois possibilités de faire une erreur.

Mais il faut également prendre en compte le fait que, chaque fois que nous écrivons un symbole, que ce soit le bon ou non ne nous empêchera absolument pas de faire une erreur sur les symboles qu'il reste à retranscrire.

Si bien que, pour représenter la valeur quatre-cent nonente sept de la section précédante sous une forme décimale, nous nous retrouvons avec `3 + 2 +1 = 7` risques de faire une erreur en le retranscrivant.

Et comme cette même valeur numérique nécessite 9 bits pour pouvoir être représentée, le risque de faire une erreur augmente de manière significative, car nous nous retrouvons `9 + 8 + 7 + 6 +5 +4 + 3 +2 + 1 = 45` risques de faire une erreur de retranscription.

Je vous ferai grâce du calcul, mais songez que, pour une valeur numérique représentée (en binaire) à l'aide de 32 bits, il est possible de représenter 4 294 967 296 valeurs différentes, ce qui fait que, pour une valeur correcte, il y en a plus de quatre milliards qui seront erronnées.

Mais, indépendamment de la valeur représentée lors de la retranscription, si l'on doit représenter une valeur sur tente deux bits, nous avons cinq-cent soixante et une “chances” de faire une erreur contre une seule chance de ne pas en faire.

Et, bien sur, le risque de faire une erreur se répète forcément... pour chaque valeur numérique que nous voudrons représenter.

La notation hexadécimale

Le risque d'erreur lors de l'utilisation de la notation binaire a très vite été flagrant, même lorsque les ordinateurs étaient très loin de comprendre “d'une seule traite” trente-deux bits consécutifs.

Pour leur malheur, les premiers à devoir programmer ce que l'on appelait à l'époque des “super calculateurs” n'avaient pas vraiment le choix, car il devaient littérallement brancher un fils entre la sortie d'un bit particulier et l'entrée de l'élélemnt suivant pour le bit correspondant.

L'idée de Von Neuman permettra, une fois qu'un système d'entrée sera mis au point de remédier à ce problème.

Vous l'avez remarqué, j'ai écrit la valeur binaire en regroupant les bits par groupe de quatre bits consécutifs dans la section précédante. Cela n'était absolument pas un hasard.

Les gens se sont en effet rendus compte, s'ils prennaient un groupe de quatre bits consécutifls , que ce groupe leur permettrait de représenter un ensemble de valeurs compris dans l'intervalle `[0,15[` (en notation décimale), et qu'il ne manquerait alors que six symboles pour pouvoir représenter les valeurs comprises dans l'intervalle `[10,15]`.

Or, ces six symboles étaient faciles à trouver : il “suffisait” de prendre les première lettres de l'alphabet. La *notation hexadécimale* (ou en *base seize*) était née.

Une même cause ayant toujours les mêmes effet, en respectant l'équation donnée pour le calcul en base dix, mais en multipliant la valeur du symbole par 16^{p-1}, ils étaient donc en mesure de représenter n'importe quelle valeur sous une forme hexadécimale.

Ainsi, la valeur quatre-cent nonente sept que j'utilisais comme exemple plus haut peut-elle être représentée sous la forme hexadécimale de 1F1. Et nous pouvons en apporter la preuve:

- Le 1 se trouvant à la position 3 correspond à la valeur $1 * 16^2$, soit `256` en décimale
- Le F se trouvant à la position 2 correspond à la valeur $15 * 16^1$ soit `240` en décimale
- Le 1 se trouvant à la position 1 correspond à la valeur $1 * 16^0$, soit `1` en décimale
- `256 + 240 + 1` (en décimal) est bien égal, jusqu'à preuve du contraire, à `497` (en décimal, toujours)

La notation octale

Avant même que la notation hexadécimale n'apparaisse, les gens avaient bien pris conscience des problèmes liés à l'utilisation du binaire.

Leur première idée fut donc de créer des groupes de trois bits, car ceux-ci leur permettaient de représenter des valeurs comprises dans l'intervalle `[0,8[`.

La *notation octale*, que l'on peut également appeler la notation en *base huit* était née.

Une même cause ayant toujours les mêmes effets, en respectant l'équation donnée pour le calcul en base dix, mais en multipliant la valeur du symbole par 8^{p-1}, ils étaient donc en mesure de représenter n'importe quelle valeur sous une forme octale.

Ainsi, la valeur quatre-cent nonente sept que j'utilisais comme exemple plus haut peut-elle être représentée sous la forme octale de `761`. Et nous pouvons en apporter la preuve:

- le `7` qui se trouve à la troisième position correspond à une valeur égale à $7 * 8^2$, soit `64 * 7 = 448` en décimal

- le `6` qui se trouve à la deuxième position correspond à une valeur égale à $7 * 8^1$, soit `48` en décimal
- le `1` qui se trouve à la première position correspond à une valeur égale à $1 * 8^0$, soit `1` en décimal
- `448 + 48 + 1` (en décimal) est bien égal, jusqu'à preuve du contraire, à `497`.

Ces groupes de trois bits consécutifs présentaient des avantages indéniables, parmis lesquels nous pouvons entre autre citer:

- le fait qu'ils permettaient de réduire drastiquement le risque d'erreur lors de la retranscription;
- le fait que le nombre de symboles nécessaires à la représentation d'une valeur numérique particulière était fort proche du nombre de symboles nécessaire pour la représentation de la même valeur en décimale;
- nous disposions déjà de tous les symboles pour représenter cet intervalle par l'intermédiaire des chiffres arabes;

Pourquoi dans cet ordre?

D'aucuns pourront – avec raison – se demander pourquoi j'ai décidé de présenter la notation octale **après** la notation hexadécimale.

En effet, que nous considérions l'ordre “logique” `2 < 8 < 16` ou l'ordre “historique” de l'apparition des différentes bases de calcul, tout aurait du m'inciter à parler de la notation octale **avant** la notation hexadécimale.

Cependant, la très grosse majorité des ordinateurs auxquels vous serez confrontés sont capables de travailler avec des ensembles de bits consécutifs qui sont des multiples de quatre.

En effet, les termes d'“*architecture 32bits*” et d'“*architecture 64bits*” représentent “tout simplement” des ordinateurs dont “toute la mécanique interne” travaille avec des groupes de … 32 ou de 64 bits consécutifs.

Pas de bol, 32 et 64 sont bel et bien des multiples de quatre, mais ne sont pas des multiples de trois, si bien que vous serez sans doute bien plus souvent confrontés à une notation hexadécimale qu'à une notation octale.

Une équivalence parfaite

L'existence des différentes bases nous mets exactement dans la même situation qui met en parallèle le système métrique (milimètres, metres, kilomètres) et le système anglo-saxon de mesure (pouce, pied, mile):

Comme les deux systèmes permettent de représenter exactement la même chose, il est tout à fait possible d'effectuer une conversion de manière à représenter **la même valeur** dans l'un ou l'autre système.

De la même manière, les notations binaire, octale, décimale et hexadécimale ne sont jamais que des *conventions* utilisées pour faciliter la représentation de choses strictement identiques.

Il est donc possible de convertir une valeur représentée dans l'une de ces notations pour la fournir dans une autre, ce qui fait que l'équivalence est parfaite.

Il est d'ailleurs possible de placer cette équivalence dans un tableau

Table 6.1. équivalence des différentes base

binaire	octal	décimal	hexadécimal
0000	00	00	00
0001	01	01	01
0010	02	02	02
0011	03	03	03
0100	04	04	04
0101	05	05	05
0110	06	06	06
0111	07	07	07
1000	10	08	08
1001	11	09	09
1010	12	10	0A
1011	13	11	0B
1100	14	12	0C
1101	15	13	0D
1110	16	14	0E
1111	17	15	0F

Chapitre 7. Les grand noms de l'informatique

Dans ce chapitre, je rends hommage aux « grand noms » de l'informatique, à toutes ces personnes sans lesquelles l'informatique ne serait sans doute pas ce qu'elle est devenue.

John Von neumann

Né en 1903 à Budapest et mort en 1957 à Washington, Jhon Von Neuman peut être considéré à juste titre comme le père de l'informatique moderne.

Il fut en effet le premier, avec ses collaborateurs, à proposer une structure de stockage unique pour conserver à la fois les instructions et les données demandées ou produite par un calcul.

Cette structure est toujours utilisées dans les ordinateurs modernes. [1]

Bjarn Stroutrup

Bjarne Stroutrup est véritablement le père du C++. Sans lui, le C++ – et a fortiori, cet ouvrage – n'aurait jamais vu le jour.

Il est l'auteur de plusieurs ouvrages pouvant être considérés comme des références en matière de C++, bien que certains soient devenus plus ou moins obsolètes à cause de leur âge.

Parmi ceux-ci, nous pouvons citer:

- The C++ Programming Language: un ouvrage de référence sur le C++, régulièrement remis à jour en ce qui concerne les normes récentes
- The Annotated C++ Reference Manual : Bien que non remis à jour, cet ouvrage peut à juste titre servir de référence pour les fonctionnalités les plus ancienne du C++

[1] Voir aussi merci Mr Von neumann

Partie II. Les principes communs

Dans cette partie, j'aborderai les principes qui sont finalement communs à l'ensemble des langages dits « procéduraux ».

Cela inclut:

- les notions de base, comme celle de fonction, de tests, de boucles et de récursivité, ainsi que certaines structures typiques, comme la notion de tableau, de pile, de file, ou d'arbre binaire;
- une présentation de certaines représentation algorithmique « agnostiques »;
- le principe de la responsabilité unique;
- la notion d'encapsulation;

Chapitre 8. Pourquoi cette partie?

Beaucoup trop de gens croient qu'il suffit d'avoir une idée de programme et de se jeter sur son clavier et de commencer à « vomir » du code pour obtenir un résultat.

Il ne pourraient faire plus grave erreur. Car l'écriture du code – quel que soit le langage – n'est en définitive « qu'une étape parmi d'autres » du processus qui permet d'obtenir une application.

De leur côté, beaucoup de cours dédiés à l'apprentissage d'un langage particulier accentuent cette idée mettant directement certains besoins rencontrés lors de la création d'une application en relation avec la manière de répondre à ce besoin par le code.

En écrivant cet ouvrage, j'ai – très clairement – pris la décision de rompre avec les « pratiques encestrales », et de proposer une approche plus cohérente – et j'espère avec une coube d'apprentissage plus facile – du problème.

Car tous les *bons* développeurs vous le diront: un langage de programmation n'est jamais que le moyen dont dispose le développeur pour se faire comprendre par « quelque chose d'aussi bête qu'un ordinateur ».

Les meilleurs d'entre eux iront même plus loin, en disant que l'écriture du code n'est – en définitive – qu'un « long et fastidieux travail de dactylographie ». Et je reconnais sans vergogne que je suis à peu près de leur avis, bien que j'adore véritablement écrire du code.

Les développeurs qui s'exclameront de la sorte se sont rendus compte que tous les langages qui proposent un paradigme particulier respectent systématiquement les mêmes règles, les mêmes principes, utilisent des concepts équivalents, dont la seule différence est parfois le nom donnés à ces concepts.

Si cette partie existe et qu'elle se trouve en première position dans le livre, c'est tout simplement parce que je suis entièrement d'accord avec eux.

C'est la raison pour laquelle l'approche que je suivrai dans cet ouvrage consistera d'abord à vous apprendre les règles, les principes et les concepts d'un point de vue purement théorique, sans me soumettre au « carcan » d'un quelconque langage avant de vous expliquer précisément comment mettre ces règles, principes et concepts en œuvre avec un langage comme le C++.

A mon humble avis, cette approche rendra non seulement votre apprentissage du C++ plus facile, mais elle facilitera aussi l'apprentissage de n'importe auquel langage auquel vous serez tentés de vous intéresser par la suite .

Et, de fait, si la première partie de cet ouvrage s'intitule « les principes communs », c'es parce que tous les langages se rapportant à la programmation impérative respectent un ensemble de règles et utilisent un ensemble de concepts commun.

Cette partie n'ayant finalement pour seul but que de vous les présenter.

Chapitre 9. Réfléchir à la logique

Les gens qui croient qu'il suffit de se jeter sur son clavier et de se mettre à « vomir » du code pour créer une application ne pourraient pas faire plus lourde erreur.

Il faut en effet prendre conscience du fait que, lorsque nous décidons de créer un programme, nous le faisons toujours dans un but similaire: manipuler des données afin d'atteindre un objectif spécifique.

Il ne sert donc à rien de nous jeter sur son clavier si nous n'avons pas une idée un tout petit peu précise de l'objectif à atteindre, des données qu'il faudra manipuler et de la manière dont ces données doivent être manipulées pour atteindre l'objectif en question.

De plus, il ne faut pas oublier que le langage – quel qu'il soit – que nous choisirons ne sera finalement que la convention qui nous permet de nous adresser à l'interpréteur ou au compilateur qui utilise le langage; ni que l'interpréteur ou le compilateur ne sera là que ... pour servir d'interpête entre nous et l'ordinateur.

Il va de soi que nous devrons nous veille à respecter les différentes conventions utilisées par le langage que nous aurons choisi pour créer notre application.

Mais, avant que le langage choisi n'interfère dans notre travail, nous avons bien du pain sur la planche.

Chapitre 10. Quatre questions de base

De manière générale, quel que puisse être notre objectif lors du développement d'une application ou d'une fonctionnalité quelconque, nous aurons toujours besoin de répondre à quatre questions de base.

Tant que nous n'aurons pas une réponse satisfaisante à ces quatre questions, il ne servira finalement pas à grand-chose de vouloir écrire le code.

Pour être tout à fait honnête, les trois premières ont, surtout, pour objectif de « débrousailler les chemins tortueux » de la réflexion afin de nous mener à la dernière, dont le but sera de nous permettre de coder « relativement facilement » les fonctionnalités dont on a besoin.

Voici donc les quatre question que nous devrons nous poser.

Que doit faire ma fonctionnalité?

Vouloir développer une fonctionnalité quelconque – quelle que soit la taille de cette fonctionnalité – sans en avoir au préalable définis les objectifs n'a absolument aucun sens :

Vous ne pourrez en effet jamais déterminer si la fonctionnalité que vous avez développée répond à vos attentes si ... vous ne savez pas, à la base, quelles sont réellement ces attentes.

Vous aurez bien souvent déjà répondu en partie à cette question, car, le plus souvent, lorsque vous décidez de développer quelque chose, c'est sans doute parce que vous avez eu – ou que quelqu'un d'autre a eu – une idée proche de

> Tiens, ce serait pas mal si on avait une fonctionnalité qui <fait telle ou telle chose>

ou proche de

> Tiens, j'ai besoin d'une fonctionnalité qui <fait telle ou telle chose>

De telles idées sont largement suffisantes pour décider de lancer le développement de quelque chose.

Il faudra cependant prendre en compte le fait que ces idées risquent d'évoluer énormément au fil du temps.

De quelles données aurai-je besoin?

Je vous l'ai dit plus haut: le but premier de n'importe quelle application, de n'importe quelle fonctionnalité que vous pourrez développer restera **toujours** de... manipuler des données.

Il est donc très intéressant de vouloir « une fonctionnalité qui <fait quelque chose> », mais, si l'on ne détermine pas préciséments les données qui seront manipulée pour « faire cette chose », nous n'arriverons sans doue jamais au résultat souhaîté.

Nous verrons plus loin que cette notion de « données manipulées » peut s'avérer particulièrement complexe à définir.

Mais nous verrons aussi quelques règles qui peuvent très largement faciliter le travail lorsqu'il est question de répondre à cette question.

Ou vais-je trouver ces données?

Au risque de faire un sillogisme, il faut bien se rendre compte que les données que notre fonctionnalité devra manipuler ne « tomberont pas du ciel », que nous ne les obtiendrons pas « de l'air du temps », et qu'il est donc très important de définir clairement quelle en est l'origine.

Nous nous rendrons en effet compe au cours de cette section qu'une fonctionnalité peut manipuler des données de différentes origines:

- Certaines données font « partie intégrante » de la fonctionnalité que l'on souhaite développées
- D'autres données peuvent être fournies par « ce qui fait appel » (au niveau d'une application, par exemple) à la fonctionnalité
- D'autres données, enfin, peuvent venir « de l'extérieur ».

Chacune de ces origines sera étudiée plus en détail par la suite, et nous nous rendrons rapidement compte que l'origine d'une donnée va influer énormément sur la manière dont elle devra être traitée.

Comment dois-je les manipuler?

Une fois que nous avons pu déterminer de quelles données nous avons besoin ainsi que leurs origines respectives, il ne reste finalement plus qu'une question à se poser, mais elle est primordiale : comment dois-je manipuler ces données?

Le but de cette question est de nous inciter à réfléchir correctement à l'ensemble de la logique qui nous permettra d'obtenir le résulat attendu.

L'expérience m'a en effet prouvé que, si nous sommes capables d'exprimer **très clairement** cette logique, en prenant tous les aspects du problème en compte, l'écriture du code devient un vrai jeu d'enfant:

une fois que nous avons corrigé les inévitables erreurs de syntaxe et que nous avons été en mesure de reproduire fidèlement la logique exprimée – et pour autant, bien sur, que cette logique soit correcte – nous pouvons quasiment avoir la certitude que notre fonctionnalité fera exactement ce que l'on attend d'elle.

Bien sûr, la grosse difficulté réside – justement – dans le fait d'exprimer cette logique de manière précise, et à s'assurer qu'elle est correcte.

Mais tout cela sera abordé dans les chapitres suivants.

Chapitre 11. Glossaire de base

> Blissingr signifie le feu. C'est le feu.
>
> Le nom, c'est la chose. Connait le nom, et tu maîtrise la chose
>
> —Eragon 2006

Avez vous remarqué que tous les termes que nous pouvons utiliser dans notre vie de tous les jours représente soit « une sorte de chose », soit « une chose (ou une personne) particulière », soit « une action »?

C'est parfaitement normal, me direz vous, car c'est ce qui permet à deux personnes (ou plus) de communiquer entre elles.

Et, de fait, c'est parce qu'il existe des conventions qui permettent à chacun de savoir ce que signifie les termes « tasse », « tournevis » ou « cuisiner » qu'il est possible de se faire comprendre.

Mais j'ai bien parler de conventions au pluriel, car il en existe – pour faire simple – autant que de langue parlées. Ainsi, vous devriez utiliser les termes « *cup* », « *screwdriver* » ou « *to cook* » pour vous faire comprendre d'une personne parlant anglais en désignant exactement les mêmes notions.

De plus, chaque domaine d'activité utilise des termes qui ont une signification spécifique à ce domaine particulier.

Il arrive que ces termes ne soient utilisés que dans ce domaine particulier, mais il arrive aussi qu'un même terme soit utilisé dans plusieurs domaines avec des significations différentes.

Le terme « fouet » en est un parfait exemple, car il aura une signification tout à fait différente pour le cuisinier de celle qu'il aura pour le marchand d'esclaves.

Avant de commencer, il me semble donc important de vous expliquer quelques termes « de base » (car d'autres viendront par la suite) utilisés par les développeurs afin que nous puissions nous comprendre.

Glossaire

Le jargon de base

une donnée	une *donnée* va représenter une information – quelle qu'elle soit – qui sera utilisée par l'ordinateur. On peut faire la distinction entre trois « notions » de données bien particulier: les données dites *constantes*, ce que l'on appelle couremment des *variables* et ce que l'on apppelle des *littérales*.
Les constantes	Le terme de constante désigne une donnée dont la valeur n'est pas desitnée à évoluer au fil du temps.
Les variables	Le terme *variable* désigne une donnée dont tout ou partie de la valeur est susceptible d'évoluer au fil du temps.

Les littérales	Le terme *littérale* représente une valeur qui est fournie directement « telle quelle » dans le code. Ces valeurs sont – forcément – *constantes*, vu qu'il n'y a aucune raison d'aller les modifier en cours d'exécution du programme.
Expression	Une *expression*représente un élément susceptible d'être compris par le langage que l'on utilise
Instruction	Une *instruction* représente une action quelconque que nous voulons pouvoir exécuter
Une fonction	une *fonction* regroupe un ensemble d'instructions qui doivent être exécutées dans un ordre bien particulier
Procédure	Le terme de *procédure* a une signifcation très semblable au terme de fonction. Bien que certains langages fassent une différence entre les deux termes, et que d'autres utiliseront de préférence un terme plutôt que l'autre, il n'est pas incorrect de les considérer comme de parfaits synonymes lorsque l'on parle des « langages de programmation » de manière générale.
Type de donnée	Le terme de *type de donnée* représente la « nature » de la donnée que l'on veut manipuler et, par conséquent, l'usage qui peut en être fait, de la même manière qu'un couteau sera utilisé à des fins différentes qu'un tournevis.

Chapitre 12. Exprimer la logique

Nous pouvons très facilement comparer le code d'un programme ou d'une fonctionnalité quelconque avec une recette de cuisine.

En effet, le programme que nous écrivons a pour objectif de décrire très clairement à l'ordinateur les étapes par lesquelles il doit passer pour obtenir le résultat souhaité, exactement comme une recette de cuisine décrit les étapes à respecter pour nous permettre d'obtenir le plas souhaité.

Mais il faut bien se rendre compte que le langage de programmation que nous utilisons n'a comme objectif que de... nous permettre de faire comprendre à un programme (le *compilateur* ou l'*interpréteur*) les étapes qui doivent être effectuées.

Si bien que nous pouvons parfaitement faire le parallèle entre les langues comme le français, l'anglais ou l'espagnol et les langages de programmation.

Tout comme il est tout à fait possible de traduire une recette de cuisine dans différentes langues, il est aussi possible de traduire les différentes étapes par lesquelles l'ordinateur devra passer pour obtenir un résultat donné dans les différents langages de programmation.

Cela ne va modifier ni la recette de cuisine ni la logique de passer d'une langue ou d'un langage à un(e) autre, et le résultat sera toujours le même.

Seulement, pour être en mesure de « traduire » les différentes étapes que l'ordinateur devra suivre dans un langage de programmation quelconque, nous devrons – à tout le moins – ... savoir par quelles étapes il doit passer.

Nous donnerons généralement à cette suite d'étapes par lesquelles l'ordinateur doit passer, à la logique qu'il doit mettre en œuvre pour obtenir le résultat souhaité le nom d'*algorithme*.

Cependant, il est souvent intéressant de « garder une trace » de la logique que l'on prévoit de mettre en œuvre:

Que ce soit pour être en mesure de l'améliorer, pour pouvoir s'assurer qu'elle est correcte ou – tout simplement – pour nous donner l'occasion de « garder bien au chaud » pour « plus tard », nous pouvons trouver mille et une raisons de nous approprier le fameux aphorisme

> Les paroles s'envolent, les écrits restent.

Et nous pouvons donc trouver tout autant de raisons pour vouloir représenter cette logique « sur papier ».

Il existe énormément de méthodes qui nous permettent de représenter cette logique sur papier.

J'en parlais récemment avec mon père, qui a eu « le bonheur » de travailler dans l'informatique à une époque où « tout était à faire », et il me disait que, à l'époque, nous pouvions trouver presque autant de méthodes pour représenter un algorithme qu'il n'y avait de développeurs:

- Certains utilisaient le *flowchart*;
- d'autres utilisaient le *pseudocode*;

- d'autres encore utilisaient le *Nassi-Shneidermann* ou le *Jackson*;
- Certains d'entre eux exprimaient tout simplement la logique sous forme de phrases simples décrivant ce qui devait être fait;
- J'oublie forcément quelques « catégories ».

Note

Bien que j'aborderai les différentes méthodes algorithmiques de manière séparée, il est important de comprendre que ce ne sont jamais que des conventions différentes nous permettant d'exprimer, avec plus ou moins de facilité, de manière plus ou moins compréhensible, des notions qui seront en définitive toujours les mêmes.

Nous pourrons donc toujours envisager de convertir un algorithme exprimé dans une représentation particulière en une expression du même algorithme dans une représentation différente.

Une logique correcte

Le besoin de décrire correctement la logique que l'ordinateur doit appliquer et, surtout, de pouvoir la tester a toujours été très présent en informatique.

Car, dés qu'il a été question de demander à « quelque chose d'aussi bête qu'une machine » de faire quelque chose pour nous, il devenait indispensable non seulement de pouvoir lui indiquer comment le faire, mais aussi – et surtout – de s'assurer que la machine soit cappable de faire **correctement** ce que l'on attendait de sa part.

Aux débuts de l'informatique, le temps passé à programmer la machine et à attendre la fin de l'exécution de sa tâche était proprement phénoménal.

Si bien que les premiers développeurs avaient l'habitude de dire une phrase proche de

> *You'll have only one try a day. Make it count*

que nous pourrions traduire sous la forme de

> Tu n'auras droit qu'à un essais par jour. Fais qu'il compte.

Bien sur, l'évolution des techniques a fait qu'il est non seulement bien plus facile de programmer la machine, mais aussi que l'exécution d'un programme « simple » devient de plus en plus rapide. Si bien que cette expression tombe tout doucement en désuétude.

Et pourtant, certains domaines particuliers de la recherche sont toujours confrontés au fait que certains algorithmes demandent toujours un temps d'exécution énorme.

Le temps passé à coder (dans un langage de programmation particulier) ces algorithmes et – le cas échéant – à compiler le code qui en résulte n'est alors qu'une « infime goutte d'eau » que représente la durée d'exécution.

D'aucuns pourront cependant prétendre que les chances pour qu'ils se retrouvent confrontés au fait d'avoir à mettre de tels algorithmes en œuvres sont sans doute encore plus faibles que leurs chances de gagner au lotto ou à l'euro-million. Et je serais bien obligé d'admettre la véracité de leur argument.

Même si ces personnes ne l'expriment pas forcément de manière aussi franche, elles estimeront sans doute que « c'est une perte de temps » que de se « poser » cinq minutes afin de réfléchir à la logique à mettre en œuvre et, surtout, de s'assurer que cette logique est correcte.

Ces personnes n'ont sans doute simplement pas conscience du temps qu'elle peuvent perdre à ne pas le faire. Car les occasions de perdre du temps sont alors nombreuses, il est même possible d'en citer quelques unes:

- Si le résultat obtenu ne correspond pas au résultat attendu, il faudra quand même « tout refaire » ou, au minimum, commencer à investiguer pour savoir pourquoi;
- il arrive que l'algorithme mis en place fonctionne « presque toujours ». Mais cela implique qu'il y ait des cas... où l'algorithme ne fonctionne pas;
- Même si l'on obtient toujours le résultat attendu, certains algorithmes sont moins efficaces que d'autre – parfois en fonction des circonstances – et l'on en vient à se satisfaire d'un algorithme qui **correctement** « fait le travail » en cinq minutes, alors qu'un autre aurait pu le faire... en cinq secondes.

Bien sur, le fait de se poser pour réfléchir à la logique et pour s'assurer qu'elle est correcte n'offre strictement aucune garantie quant au fait que nous ne serons **jamais** confrontés à de telles occasions de perdre du temps.

Cela permet cependant d'en réduire considérablement le risque.

> Le développement informatique est comme la gestion de capitaux:
>
> Les bonnes décision feront gagner du temps et (éventuellement) de l'argent.
>
> De mauvais choix occasionneront une « dette technique » dont les intérêts ne feront qu'augmenter et que nous payerons de plus en plus cher (en temps et en argent) au fil du temps.

Chapitre 13. Le pseudocode

Je sais, je me répète, et je le ferai encore souvent. Mais il faut comprendre que n'importe quel langage de programmation n'est qu'un ensemble de conventions qui nous permettent de nous faire comprendre par un programme (le *compilateur* ou l'*interpréteur*).

Mais, comme il ne s'agit finalement que d'une convention, nous pourrions tout aussi bien envisager d'avoir un « langage » qui ne sera utilisé que pour nous permettre de transmettre l'algorithme non pas à un programme qui se chargera de le compiler ou de l'interpréter mais à une « tierce personne » qui pourra l'analyser, et / ou l'utiliser pour le « traduire » dans un langage de programmation particulier.

Ce langage aurait toutes les caractéristiques que l'on retrouve dans les langages de programmation, dont le formalisme indispensable si on veut se faire comprendre par un programme.

Le résultat de cette représentation aurait donc « toutes les caractéristiques » d'un code que nous aurions écrit dans un langage de programmation particulier.

La seule différence, c'est sans doute qu'il n'existe aucun interpréteur ni aucun compilateur qui soit en mesure de manipuler un tel code. Et c'est bien sur tout à fait normal, vu que ce n'est pas le but de ce dernier.

C'est la raison pour laquelle nous parlerons généralement de *pseudocode* pour désigner une telle représentation de la logique.

La seule représentation « textuelle »

Vous ne manquerez pas de le constater par vous-même : la grosse majorité des méthodes de représentation d'algorithmes se fait à base de pictogrammes ou de « dessins » qui ne sont jamais très compliqués à comprendre.

L'utilisation de dessins et pictogrammes est particulièrement facile lorsqu'il s'agit de transmettre un algorithme dans un « format papier »(ou équivalent).

Par contre, cette utilisation de « petits dessins » n'est par exemple absolument pas pratique lorsqu'il s'agit de transmettre un algorithme sous la forme de texte, lors d'une discussion sur un forum, par exemple

Je vous avouerai en outre faire partie de ces gens qui considèrent que, quitte à écrire du « faux code », nous avons sans doute bien plus intérêt à en écrire « directement du vrai » .

Et pourtant, en dehors des explications propres aux différentes méthodes que je vais donner, je vous présenterai la très grosse majorité des algorithmes sous forme de pseudocode, par facilité, dans cet ouvrage.

Note

Je dois cependant reconnaître que, si nous commençons à écrire du vrai code, nous sommes tenus de respecter les règles spécifiques prpores au langage utilisé, et que nous perdons dés lors tout l'intérêt d'une éventuelle représentation « indépendante du langage utilisé » de notre algorithme.

Chapitre 14. Le « Flowchart »

L'une des première méthodes utilisées pour représenter un algorithme a été la méthode appelée *flowchart*. Ce terme n'est rien d'autre que la contraction du mot anglais « *flow* » qui signifie « déroulement » et du mot anglais « *chart* » qui signifie « diagramme ».

Nous pouvons donc assez facilement déduire de ces deux traductions que le terme *flowchart* représente un... diagramme de déroulement, et donc qu'il permet de représenter la manière dont les choses doivent se dérouler.

Cependant, le terme en lui-même peut aussi être traduit par le terme d'« organigramme (de programmation) », que vous rencontrerez sans doute de manière récurrente sur les forums francophones dédiés à la programmation.

L'idée de base

L'idée de base de cette représentation est d'utiliser des pictogrammes dont le sens ne fera aucun doute pour celui qui analyse l'algorithme afin de représenter les différentes étapes de cet algorithme.

Chaque étape étant reliée à la suivante par une flèche indiquant clairement le sens dans lequel l'exécution se déroule.

Un début et un fin

Je vous l'ai déjà répété bien des fois, le but d'un algorithme est de fournir la logique qui devra être mise en œuvre (sans doute par un ordinateur) pour obtenir un résultat donné.

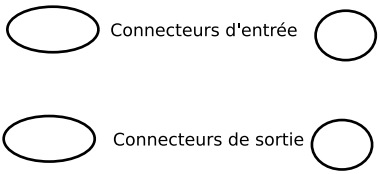
Nous pourrions tout à fait exprimer les choses sous un angle différent et dire qu'un algorithme représente la logique à mettre en œuvre pour permettre à l'oridinateur, au départ d'une situation donnée, d'arriver à une situation clairement définie.

Cette deuxième manière d'exprimer les choses présente l'énorme avantage de mettre en évidence une caractéristique essentielle à tout algorithme, qui est le fait que chaque algorithme démarre systématiquement avec une « situation de départ » et tente d'atteindre une « situation d'arrivée ».

Autrement dit, il y a systématiquement un début et une fin à un algorithme. En flowchart, nous représenterons généralement ces deux points particuliers sous la forme de cercles ou d'éllipses, respectivement appelés *connecteur d'entrée* et *connecteur de sortie*.

Ces deux points (entrée et sortie) définissent ce que l'on appelle généralement une « procédure »

Figure 14.1. Un début et une fin



En pseudo code, nous pourrions représenter cela par les termes « Début Procédure » et « Fin Procédure » ou,:

Exemple 14.1. Le début et la fin en pseudocode

```
Début Procédure
...
Fin Procédure
```

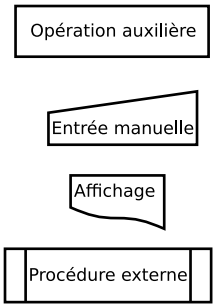
Les actions simples

Pouvoir définir le début et la fin d'un algorithme est particulièrement important. Mais si nous ne serions guère avancé si nous étions incapable de représenter ce qui se passe entre ces deux points.

Le flowchart définit de très nombreux symboles destinés à représenter certaines actions courantes. Parmis ceux-ci, nous pouvons au minimum citer

- le symbole représentant une action devant être exécutée – une multiplication ou une addition par exemple – représenté par un rectangle ou un carré;
- Une entrée effectuée par l'utilisateur sera représentée par un « rectangle » dont le coté du dessus n'est pas parallèlle;
- L'affichage sera représenté par un rectangle dont le coté du dessous prend la forme d'une doucine
- l'appel à une procédure définie par ailleurs – que nous appellerons *fonction (ou procédure) externe* représenté par un carré ou un rectangle avec un « bord vertical »

Figure 14.2. Les actions de base



Note

Pour être tout à fait précis, le symbole que j'utilise pour l'affichage est celui qui était utilisé pour les bandes relais.

Le flowchart dispose d'un tas de symboles représentant des cartes perforées, des bandes relais ou des bandes perforées, mais... aucun sybmole pour l'écran.

Note

Il existe une quantité énorme de pictogrammes associés au flowchart, dont certains sont adaptés à des situations bien précises.

Il n'entre pas dans les intentions de cet ouvrage de les passer tous en revue. Si vous souhaitez en avoir un apperçu exhaustif, effectuez une recherche sur « flowchart symbols » avec votre moteur de recherche préféré

Les actions simples en pseudocode

Les opérations auxilières, comme les additions et autres seront tout simplement représentées par... l'opération qui doit être effectuées

les entrées et sorties seront, quant à elles, sans doute représentées par des verbes comme « Afficher », « Lire », et autres

Enfin, l'appel à une procédure externe sera sans doute représenté par l'utilisation de termes comme « Appeler »

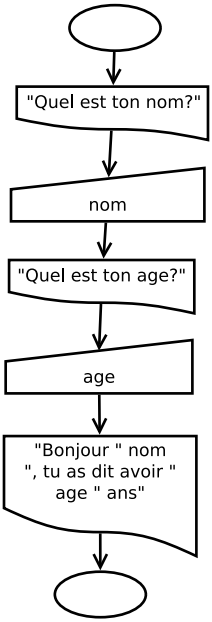
Un premier algorithme

A l'aide de ces quelques symboles, nous pourrions déjà créer ce qu'il est bel et bien convenu d'appeler un « premier algorithme ». Il ne nous manque, pour le flowchart, finalement qu'une seule information: comment relier les différents éléments entre eux.

Et la réponse est on ne peut plus simple: il suffit de tracer une flèche qui part d'un élément du diagramme et qui en rejoint un autre; le sens de la flèche permettant d'indiquer quand quel ordre les instructions seront exécutées.

Ainsi, si nous voulons créer une procédure qui demande son nom et son age à l'utilisateur avant d'afficher le tout, nous pourrions le faire sous la forme de

Figure 14.3. Un premier flowchart



C'est simple, et c'est efficace.

Exemple 14.2. Premier algorithme en pseudocode

Nous aurions pu créer exactement le même algorithme en pseudocode, qui aurait alors sans dout pris la forme de

```
Début <Nom de Procédure>
    Afficher "Quel est ton nom?"
    Attendre nom
    Afficher "Quel est ton age?"
    Attendre age
    Afficher "Bonjour " nom
             "Tu as dit avoir " age " ans"
Fin <Nom de Procédure>
```

Où < Nom de procédure > correspondra à un nom permettant d'identifier la procédure dont il est question de manière strictement unique et non ambiguë.

Enfin des données à traiter

Bon, le premier exemple était particulièrement simple. Il ne faut cependant pas négliger l'intérêt didactique de cet exemple, car il met bien en évidence le fait que tout algorithme va – a priori – **manipuler des données**.

Vous ne l'avez peut-être pas remarqué, mais cette algorithme manipule énormément de données, la preuve:

- "Quel est votre nom?" peut être considéré comme une donnée de type « chaine de caractère » (*)
- `nom` est une donnée qui sera aussi considérée comme une donnée de type « chaine de caractères » (étant donné que l'on s'attend d'avantage à ce que l'utilisateur introduise « jean-francois » que `3.1415926`)
- "Quel est ton age?" peut également être considéré comme une donnée de type « chaine de caractères »
- `age` est une donnée que l'on peut considérer comme étant « de type entier » (étant donné que l'on s'attend d'avanage à ce que l'utilisateur introduise `21` que « hello world »)
- "Bonjour ", " tu as dit avoir " et "ans" peuvent enfin – elles aussi – être considérées comme des données de type « chaine de caractères ».

Note

(*)Nous pourrions même préciser que ces données sont des *constantes* et des *littérales*, car il est évident qu'elles ne changeront absolument jamais et que, de plus, leur valeur est fournie « telle quelle » dans le code.

Nous pouvons mettre cette notion de *donnée constante* en opposition avec les données `nom` et `age`, qui correspondent à ce que l'on appelle des *variables*.

La liste des variables

Il est souvent intéressant, lorsque l'on définit un algorithme, d'y adjoindre un récapitulatifs des données qui seront utilisées par celui-ci. Cela permet, en cas de besoin, à celui qui devra l'implémenter dans un langage quelconque, de savoir à quoi il a réellement affaire.

Le plus facile pour fournir cette liste est souvent de créer un tableau qui reprenne l'ensemble des caractéristiques de celles-ci, sous une forme qui pourrait être proche de

Tableau 14.1. les données du premier algorithme

Nom / Valeur	Type	usage	remarques
"Quel est ton nom?"	chaine de caractères	affichage	constante littérale
"Quel est ton age?"	chaine de caractères	affichage	constante littérale
"Bonjour "	chaine de caractères	affichage	constante littérale
" tu as dit avoir "	chaine de caractères	affichage	constante littérale

Nom / Valeur	Type	usage	remarques
nom	chaine de caractères	le nom de la personne	introduit par l'utilisateur
age	entier	l'âge de la personne	introduit par l'utilisateur

Un tel tableau n'est pas forcément indispensable, mais il permet d'éviter toute ambiguïté quant à l'usage qui sera fait des données que l'on manipule.

Glossaire

Algorithmes : notions de base

Point d'entrée

Le point d'entrée correspond au moment où l'on décide de faire appel à un algorithme particulier.

Il correspond généralement à une situation particulière qui justifie que l'on ait recours à l'algorithme

Point de sortie

Le *point de sortie* correspond à la fin de l'algorithme; au moment où l'on estime avoir atteint l'objectif fixé, à moins bien sur que l'algorithme ne se soit rendu compte qu'il ne pourrait jamais l'atteindre.

Opération auxilière

Une opération auxilière correspond aux différentes opérations de base susceptibles d'être effectuées directement par l'algorithme, sans forcément nécessiter le recours à un « algorithme secondaire »

Entrée / Sortie

Les *entrées* et les *sorties* correspondent au moyen par lequel l'ordinateur sera capable d'interagir avec des « éléments extérieurs ».

Ces termes sont définis sont définis du point de *vue de l'ordinateur*, si bien que les *entrées* correspondent *à tout ce qui peut arriver* à l'ordinateur et qui pourra être utilisé par lui alors que les *sorties* correspondent à *tout ce qui part* de l'ordinateur afin d'être utilisé par les « éléments extérieurs ».

Procédure externe

Une « procédure externe » correspond à un algorithme qui a été développé « par ailleurs » et auquel il nous est possible de faire appel en cas de besoin.

Chapitre 15. Le principe de responsabilité unique

Première partie

L'un des principes les plus importants en programmation, auquel il est important de ne déroger sous aucun prétexte est très certainement le principe appelé le *principe de responsabilité unique*; dont vous croiserez souvent l'acronyme anglais SRP mis pour*Single Responsability Principle*.

Ce principe s'exprime de manière très similaire pour les algorithmes que nous voudrons créer que pour les données que l'on envisage de manipuler. Voici comment il s'exprime:

> **Le principe de la responsabilité unique**
>
> Toute donnée ne doit être utilisée que dans un seul et unique but, clairement déterminé;
>
> Toute algorithme ne doit viser qu'un seul et unique objectif.

L'un des problèmes auquel nous sommes régulièrement confrontés – surtout lorsque le développeur n'a pas pris le temps de se « poser » trente secondes afin de réfléchir à ce qu'il doit faire – est que de nombreux algorithmes essayent de résoudre tous les problèmes en une fois, ce qui se traduit souvent par un algorithme prenant une forme proche de

Exemple 15.1. Un algorithme qui en fait trop

Imaginons que nous ayons pour but de programmer un robot afin qu'il s'occupe des tâches ménagères

```
Début tachesMenageresDeLaJournee
    Remarque : je veux sortir le chien
    ... suivent XX actions à entreprendre pour sortir le chien
    Remarque : je veux préparer le café
    ... suivent YY actions pour préparer le café
    Remarque : je veux faire la vaisselle
    ... suivent ZZ actions pour faire la vaisselle
    Remarque : je veux faire la lessive
    ... suivent XY actions pour faire la lessive
    Remarque : je veux préparer le repas
    ... suivent XZ action pour préparer le repas
    Remarque : il y a tant de choses à faire dans une maison
    Remarque : nous pourrions rajouter tant de choses
fin tachesMenageresDeLaJournee
```

A partir du moment où chaque « tache ménagères » est correctement effectuée, il est très difficile de considérer qu'un tel algorithme soit faux.

Cependant, même si l'on peut considérer que cet algorithme ne poursuit qu'un seul et unique objectif (celui d'effectuer les tâches ménagères de la journée), nous sommes bien obligés de convenir qu'il en poursuit en réalité bien plus que cela, à savoir:

- sortir le chien (qui est, effectivement une tâche ménagère);
- préparer le café (idem);
- faire la vaisselle (idem);
- faire la lessive (idem);
- préparer le repas (idem)
- ... faire toutes les tâches ménagères que j'aurais pu vouloir ajouter.

Certains d'entre vous pourraient estimer que ce n'est pas vraiment un problème car, comme je l'ai indiqué, cet algorithme ne poursuit qu'un seul et unique objectif : celui d'effectuer les tâches ménagères de la journée. Soit.

Mais réfléchissons un tout petit peu plus loin, et envisageons ne serait-ce que trente secondes que nous livrions notre robot à deux (ou à plusieurs) clients. Que va-t-il se passer, selon vous?

La réponse est simple à deviner:

- Certains clients seront enchantés de l'ordre dans lequel les tâches ménagères sont effectuées;
- d'autres clients n'ont pas de chien, et ne souhaitent pas que le robot « perde du temps » à sortir un chien ... qui n'existe pas;
- d'autres encore préféreront avoir le café avant que le chien ne soit sorti;

En un mot comme en cent, nous trouverons toujours bien un client qui n'est pas pleinement satisfait des tâches effectuées par notre robot ou de l'ordre dans lequel il les effectue.

Et comme notre objectif est quand-même – comme tout bon commerçant – de satisfaire au mieux notre clientèle afin de vendre le plus de robots possibles, nous en viendrions très certainement à prévoir plusieurs algorithmes pour que notre robot puisse faire les mêmes choses, mais dans un ordre différent. Par exemple:

Exemple 15.2. Un algorithme qui en fait toujours trop

Cet algorithmes reprend les mêmes tâches ménagères, mais dans un ordre différent

```
Début tachesMenageresDeLaJourneeVersion2
    Remarque : je veux préparer le café
    ... suivent YY actions pour préparer le café
    Remarque : je veux sortir le chien
    ... suivent XX actions à entreprendre pour sortir le chien
    Remarque : je veux préparer le repas
    ... suivent XZ action pour préparer le repas
    Remarque : je veux faire la vaisselle
    ... suivent ZZ actions pour faire la vaisselle
```

```
    Remarque : je veux faire la lessive
    ... suivent XY actions pour faire la lessive
    Remarque : il y a tant de choses à faire dans une maison
    Remarque : nous pourrions rajouter tant de choses
fin tachesMenageresDeLaJourneeVersion2
```

Et vous aurez remarqué que je n'ai toujours pas pris en compte le cas où le client n'aurait pas de chien...

Or, vu que c'est un autre algorithme, il sera implémenté ... dans une autre fonction. La meilleure preuve en est que le nom de la fonction a été modifié.

Cela occasionnera d'ailleurs des problèmes du au non respect d'un principe d'Xtrem Programming connu sous l'acronyme de DRY (pour *Don't Repeat Yourself* ou, si vous préférez en français: *Ne vous répétez pas*), dont je parlerai plus tard.

Mais vous l'aurez compris: si nous vendons notre robot à quinze clients différents, nous risquons très fort de devoir fournir... quinze fonctions permettant d'effectuer les tâches ménagères de la journée dans un ordre différent; ce qui se traduirait par... quinze algorithmes différents.

Le plus frustrant de l'histoire, c'est que l'algorithme rattaché à chacune des tâches ménagères reste fondamentalement le même: que l'on sorte le chien avant ou après avoir préparé le café n'a absolument aucune incidence sur les actions qui doivent être entreprises pour sortir le chien (ou pour préparer le café).

Et c'est donc là que le SRP prend tout son sens, car nous pourrions tout à fait éviter d'avoir à recréer sans cesse des algorithmes identiques en créant des algorithmes spécifiques qui ne s'occupent que d'une seule et unique chose.

Exemple 15.3. plusieurs algorithmes secondaires

Ainsi, cet algorithme pourrait tout à fait être « subdivisé » en plusieurs « algorithmes secondaires » sous une forme qui serait proche de

```
début sortirLeChien
    ... les XX actions à entreprendre pour sortir le chien
fin sortirLeChien
début preparerLeCafe
    ... les YY actions à entreprendre pour préparer le café
fin preparerLeCafe
début faireLaVaisselle
    ... les ZZ actions pour faire la vaisselle
fin faireLaVaisselle
début faireLaLessive
    ... les XY actions pour faire la lessive
fin faireLaLessive
début preparerLeRepas
    ... les XZ action pour préparer le repas
fin preparerLeRepas
... toutes les autres taches ménagères que l'on peut envisager
```

Une telle séparation des responsabilités présente énormément d'avantages.

Le tout premier est qu'il est bien plus facile de « s'y retrouver dans le code » lorsque nous sommes confrontés à une fonction qui ne fait que dix ou quinze lignes que lorsque nous sommes confronté à une fonction qui en fait cent ou deux-cents.

Le deuxième avantage est qu'il devient possible de tester chaque algorithme séparément: Si un client nous appelle – par exemple – en nous disant qu'« il y a un problème lorsque le robot prépare le café », il ne sert à rien de s'inquiéter de ce que fait le robot lorsqu'il sort le chien ou lorsqu'il fait la vaisselle.

Enfin, comme une « journée type » de notre robot consistera à ... effectuer différentes tâches ménagères, il nous devient très facile de définir plusieurs algorithmes de journée type comme étant le fait de... faire appel aux différents algorithmes que nous avons créés.

Exemple 15.4. Utiliser les algorithmes « secondaires »

Ainsi, nous pourrions définir un premier algorithme de journée type sous la forme de

```
Début premiereJourneeType
    Appeler sortirLeChien
    Appeler preparerLeCafe
    Appeler faireLaVaisselle
    Appeler faireLaLessive
    Appeler preparerLeRepas
Fin premièreJourneeType
```

Mais nous pourrions tout aussi bien créer une deuxième journée type « dans la foulée », sans rien avoir à rajouter. Elle pourrait prendre la forme de

```
Début deuxiemeJourneeType
    Appeler preparerLeCafe
    Appeler sortirLeChien
    Appeler preparerLeRepas
    Appeler faireLaVaisselle
    Appeler faireLaLessive
fin deuxiemeJourneeType
```

Et, le fin du fin, c'est que nous pourrions fournir au client une liste des tâches ménagères que le robot est capable d'entreprendre, et le laisser décider de quelles tâches il souhaite que le robot s'occupe et dans quel ordre.

Chapitre 16. Les structures décisionnelles

Pour être tout à fait honnête, ce que j'ai présenté comme un « algorithme » dans le chapitre précédent tient d'avantage d'une « liste d'instructions » que d'un véritable algorithme.

En effet, la notion d'algorithme ne peut être considérée comme complète que si nous lui donnons la capacité de déterminer l'opportunité d'entreprendre ou non une action particlulière en fonctions des circonstances rencontrées.

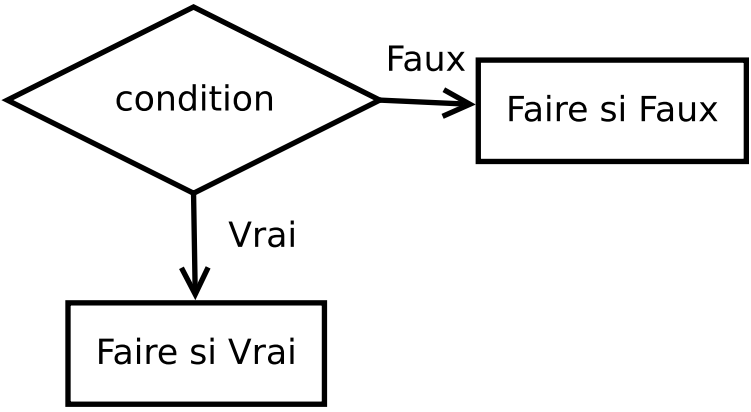
Les circonstances prises en compte sont communément appelées *conditions* et consiste à une expression dont l'évaluation ne poura aboutir qu'à deux résultat possibles:

- *Vrai*, si la condition est remplie ou
- *Faux*, si la condition **n'est pas** remplie.

Le flowchart nous permet de représenter les décisions au travers d'un losange dans lequel nous écrirons la condition devant être testée.

Deux flèches partirons de ce losange: l'une menant à ce qui doit être exécuté si la condition est remplie, une autre menant à ce qui doit être exécuté si la condition n'est *pas* remplie, et qui ressemble à quelque chose comme

Figure 16.1. Un test simple



Note

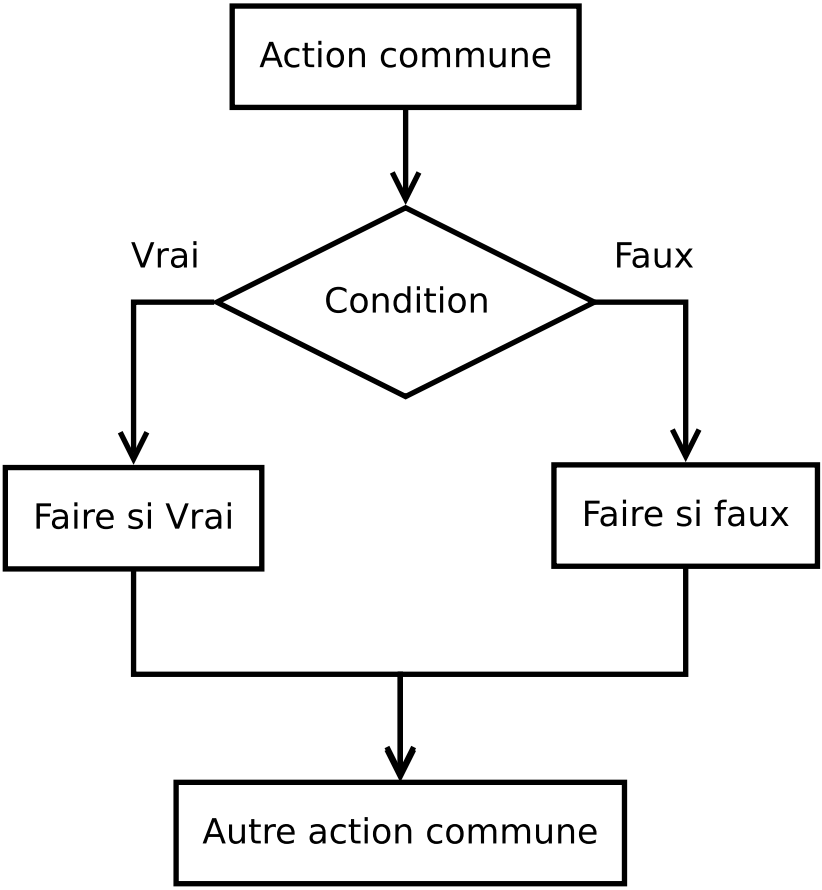
Il arrive très régulièrement qu'il n'y ait des actions à entreprendre de dans un seul cas. la flèche correspondant au cas dans lequel il « n'y a rien à faire » mènera alors à la première action à devoir être effectuée de manière indépendante du test.

Avertissement

Par convention, nous essayerons *généralment* [1]d'exprimer les conditions de manière à ce que le « gros du travail » se trouve du coté de la branche « vrai » du test

Il va de soi que « tout à une fin » et que, une fois que l'algorithme a effectué les actions qui étaient « soumises à condition », il devra reprendre son exécution normale. Si bien que nous croiserons souvent des flowchart prenant d'avantage la forme de

Figure 16.2. Un test simple sous sa forme complète



[1]Nous verrons plus tard qu'il y a des circonstances dans lesquelles il est préférable de faire l'inverse

Exemple 16.1. Les tests Vrai / Faux en pseudocode

Il va de soi qu'il est possible de représenter ce genre de test en pseudo code. Nous utiliserons généralement les termes de « Si <condition à vérifier> » pour introduire les actions devant être effectuées si une condition est vérifiées, de « Sinon » pour introduire les actions devant être effetuées dans le cas contraire et de « Fin Si » pour indiquer la fin des actions soumises à condition.

La forme « classique » d'un test « vrai / faux » en pseudocode serait donc proche de

```
Début <nom de procédure>
    ... Tout ce qui doit être fait AVANT le test
    Si <condition à respecter>
        ... Ce qui doit être fait si la condition est remplie
    Sinon
        ... Ce qui doit être fait si la condition N'EST PAS remplie
    Fin Si
    ... Ce qui doit être fait APRES, sans être soumis à la condition
Fin <nom de procédure>
```

Les années bissextiles

Grâce aux structures décisionnelles, nous pouvons « enfin » envisager de donner un « semblant d'intelligence » à notre algorithme.

Par exemple, nous pouvons définir un algorithme qui

1. demande en quelle année nous sommes et

2. affiche l'année suivie de " est bissextile" ou " n'est pas bissextile" en fonction de la situation.

Nous sommes tellement habitués à manipuler les années bissextiles que nous n'y prétons pour ainsi dire plus d'importance. Pourtant, les règles qui régissent le fait qu'une année soit bissextiles sont particulièrement précises, et ont une influence bien plus importante que ce que nous ne pourrions croire de prime abord.

Pensez donc: de ces règles dépendra le fait que la date du 29 février puisse être considérée comme une date valide.

La règle s'exprime à peu près dans ces termes:

> Une année est bissextile si le nombre qu'elle représente est divisible par quatre, sauf si ce nombre est divisible par 100, à moins qu'il ne soit également divisible par 400.

C'est ainsi que nous en arrivons à une situation dans laquelle

- 1996 était une année bissextile, parce que 1996 est divisible par 4

- 1900 n'était pas une année bissextile, parce que, bien que 1900 soit divisible par quatre (comme toutes les cenaines), ce n'est pas divisible par 400 et

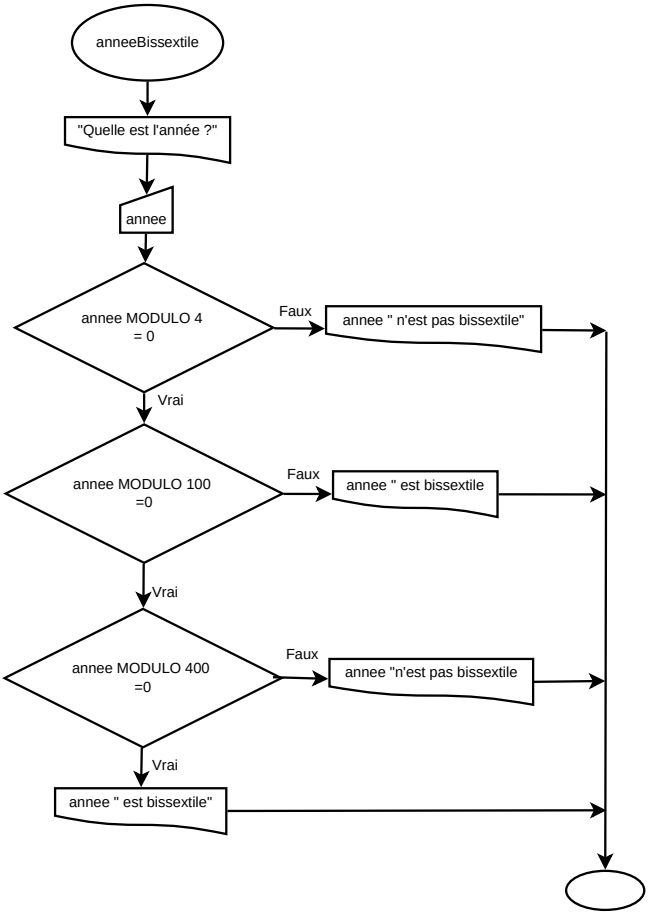
- 2000 était une année bissextile, parce que 2000 est non seulement divisible par cent (et est donc forcément divisilble par 4), mais est également divisible par 400

Le moyen le plus simple de définir si une valeur est divisible par une autre est, finalement, de travailler comme on l'a appris à l'école primaire au tout début du temps où l'on apprenait le calcul.

A l'époque, nous n'avions pas encore vu qu'il était possible de travailler avec des chiffres apèrs la virgule, et les divisions que l'on faisait ne s'occupaient que de valeurs entières et prenaient « ce qu'il restait » éventuellement de la division en compte. Ainsi, `7 / 3` était il égal à `2 reste 1`.

En informatique (et en mathématique, de manière générale), nous appelons le reste de la division de deux entiers le *modulo*. Si le reste de la division d'un entier par un autre est égal à 0, c'est que le premier nombre est divisible par le deuxième, tout simplement.

Figure 16.3. Un premier exemple de test



Exemple 16.2. Un exemple de test en pseudocode

En psudocode, le même algorithme prendrait la forme de

```
Début anneeBissextile
    Afficher "Quelle est l'année?"
    Attendre annee
    Si annee MODULO 4 = 0
```

```
        Si annee MODULO 100 = 0
            Si annee MODULO 400 = 0
                Afficher annee " est bissextile"
            Sinon
                Afficher annee " n'est pas bissextile"
            Fin Si
        Sinon
            Afficher annee " est bissextile"
        Fin Si
    Sinon
        Afficher annee " n'est pas bissextile"
    Fin Si
Fin anneeBissextile
```

Note

L'algorithme que je vous présente ici est tout à fait **correct**, dans le sens où il nous permet d'obtenir **exactement** le résultat que l'on attend de sa part.

Nous verrons cependant plus tard qu'il n'est peut-être pas aussi efficace que ce que nous pourrions souhaiter.

Glossaire

Les structures conditionnelles

Condition	Une *condition* est un ensemble de circonstances pouvant être analysées de manière à prendre une décision.
Décision	Une *décision* est le mécanisme qui consiste à déterminer s'il est opportun d'effectuer une action (ou un ensemble d'actions) particulière(s) en fonction d'une condition
Alternative	Une *alternative* permet de définir une action (ou un ensemble d'actions) particulière(s) à effectuer – s'il y en a – dans le cas où une condition ne serait pas remplie.

Chapitre 17. Les boucles

Nous pourrions qualifier les différents algorithmes que je vous ai présentés jusqu'ici de « *one shot* », dans le sens où la logique qu'ils mettent en évidence est destinée à n'être exécutée qu'une seule et unique fois: si nous voulions que cette logique soit exécutée plusieurs fois, nous devrions faire appel à l'algorithme le nombre de fois souhaité.

Mahleureusement pour nous, la logique d'un grand nombre d'algorithme nécessite d'être exécutée « un certain nombre de fois ».

Exemple 17.1. La répétition de base

Bien sur, si nous voulions faire sortir le chien trois fois d'affilée, nous pourrions tout à fait envisager de créer un algorithme proche de

```
début sortirTroisFoisLeChien
    Appeler sortirLeChien
    Appeler sortirLeChien
    Appeler sortirLeChien
fin sortirTroisFoisLeChien
```

Il serait particulièrement malvenu de dire qu'un tel algorithme est faux, à partir du moment où il fait exactement ce que l'on attend de sa part, à savoir... sortir le chien trois fois.

Mais imaginez maintenant que nous voulions entreprendre une action particulière non pas trois ou quatre fois, mais bien... une bonne centaine ou un bon millier de fois (voire d'avantage). Essayez d'imaginer à quoi pourrait ressembler un tel algorithme sous cette forme!

Ensuite, imaginez les problèmes que cela pourrait occasionner si, au lieu d'effectuer l'action 100 fois comme il est nécessaire de le faire, nous venions à « mal compter » le nombre d'appels que l'on place dans cet algorithme, et que nous finission par y faire appel seulement 97 fois, ou, à l 'inverse 103 fois.

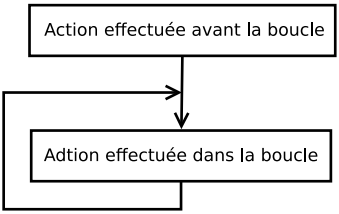
Pire enocre: il se peut que le nombre de fois qu'une action devra être entreprise dépende exclusivement de circonstances qui ne pourront être connues que... lors de l'exécution de l'algorithme.

Nous avons donc besoin de la notion que l'on appelle une *boucle*.

En flowchart, cette notion est représentée par une ligne qui rejoint un endroit de l'algorithme par lequel le « flux d'exécution normal » est déjà passé.

De manière générale, elle sera donc représentée sous une forme proche de

Figure 17.1. Une boucle de base en flowchart



Bien sur, vous l'aurez surement compris rien qu'en regardant cette image, un algorithme utilisant une boucle « telle quelle » sera dans l'impossibilité de sortir de cette boucle, ce qui le rend « incorrect » aux yeux de certains développeur.

Car, en effet, s'il n'est pas possible de sortir de la boucle, cela signifie qu'il est impossible d'atteindre... Le point auquel il est possible de considérer que l'objectif de l'algorithme est atteint, ni même d'atteindre le point où l'on pourrait considérer qu'il est impossible d'atteindre l'objectif fixé.

Si bien qu'une condition permettant de déterminer s'il faut entrer dans la boucle ou non est nécessaire.

Selon la position de cette condition par rapport à l'action (ou aux actions) qui devra (devront) être exécutée(s) dans la boucle, nous pouvons dores et déjà citer deux grandes « quatégories » de boucles:

- D'une part, il y a des boucles dont la conditon pour savoir s'il faut entrer dans la boucle est évaluée **avant** d'exécuter la boucle et

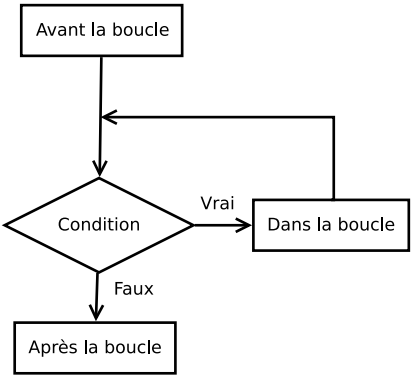
- d'autre part, il y a des boucles dont la condition permettant de déterminer s'il faut faire « un nouveau passage » dans la boucle est évaluée **après** que la boucle ait été exécutée.

Les boucles « tant que »

On désigne les boucles pour lesquelles la condition d'entrée dans la boucle est évaluée avant d'exécuter les actions de celle-ci par le terme de *boucle « tant que »*, car nous exécuterons les actions en question « tant que » la condition sera remplie.

En flowchart, une telle boucle prendra généralement une forme proche de

Figure 17.2. Une boucle « Tant Que » en flowchart



Exemple 17.2. Une boucle tant que en pseudocode

En pseudocode, nous indiquerons généralement ce genre de boucle au travèrs des termes Tant que, ce qui pourrait donner quelque chose ressemblant à

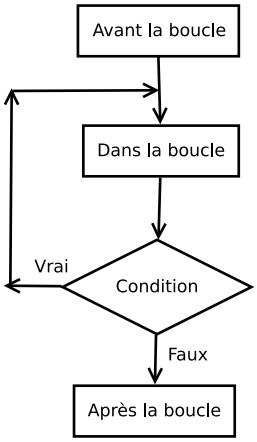
```
Début <nom de procécure>
    ... Avant la boucle
    Tant que <condition à remplire>
        ... ce qui doit être fait
    Fin tant
    ... Après la boucle
Fin <nom de procédure>
```

Les boucles « Jusque »

On désigne les boucles pour lesquelles nous vérifions après avoir exécuté ce qui doit l'être dans la boucle s'il faut « retourner une fois de plus » dans la boucle par le terme de *boucle « jusque »*.

En flowchart, une telle boucle sera représentée sous une forme proche de

Figure 17.3. Une boucle « Jusque » en flowchart



Exemple 17.3. Une boucle tant que en pseudocode

En pseudocode, nous indiquerons généralement ce genre de boucle au travèrs des termes Tant que, ce qui pourrait donner quelque chose ressemblant à

```
Début <nom de procécure>
    ... Avant la boucle
    Faire
        ... ce qui doit être fait
    Jusque <condition à remplire>
    ... Après la boucle
Fin <nom de procédure>
```

Avertissement

De manière assez contre intuitive, nous aurons généralement à définir la condition dans une telle boucle comme étant la condition de **sortie** de la boucle (comprenez: la condition qui fait que l'on passe à l'étape qui suit la boucle) à cause du terme « jusque ».

Or la plupart des langages considèrent cette condition comme la condition de **rentrée** dans la boucle (comprenez: celle qui fera que la boucle est exécutée « une fois de plus »)

Les boucle « pour »

Il existe une dernière notion de boucle, qui est – en définitive – fort proche de la notion de boucle « tant que », mais qui est sans doute bien plus souvent utilisée: la notion de *boucle « pour »*.

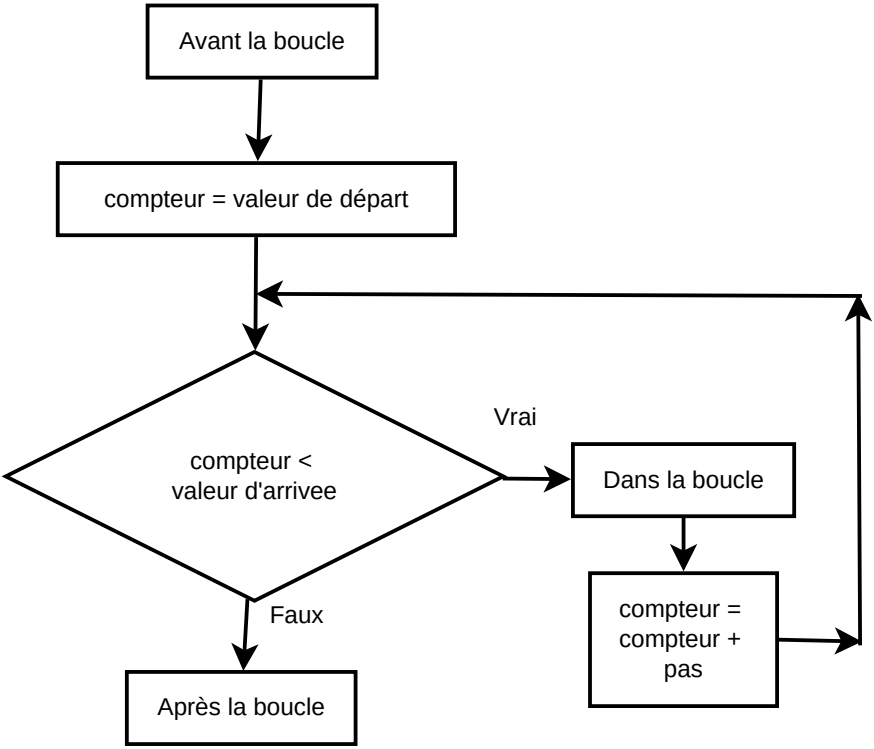
Nous nous rendrons en effet rapidement compte qu'il est souvent intéressant – voire indispensable – de disposer d'un « compteur » qui sera incrémenté à chaque passage dans la boucle.

Ce compteur peut avoir mille et un usages différents, il serait donc vaint que j'essaye de tous les détailler.

Contentons nous donc de savoir, dans un premier temps, que nous avons besoin d'un compteur soit pour « garder en mémoire » le nombre de fois qu'une boucle s'est effectuée, soit pour pouvoir parcourir un ensemble bien particulier d'éléments.

La forme « classique » d'une telle boucle en flowchart ressemblera sans doute à quelque chose comme

Figure 17.4. Une boucle « pour » en flowchart



Exemple 17.4. Une boucle pour que en pseudocode

En pseudocode, nous indiquerons généralement ce genre de boucle au travèrs des termes Pour , ce qui pourrait donner quelque chose ressemblant à

```
Début <nom de procécure>
    ... Avant la boucle
    Pour compteur = <ValeurDeDepart> à <ValeurD'arrivee>
               (par pas de <PasAAppliquer>)
        ... ce qui doit être fait
    Fin pour
    ... Après la boucle
Fin <nom de procédure>
```

où PasAAppliquer représente la manière dont la valeur du compteur doit évoluer entre chaque exécution de la boucle.

On considère en effet par défaut que cette valeur va être incrémentée de 1 entre chaque passage dans la boucle, mais il se peut que l'on souhaite que la valeur du compteur évolue d'une manière différente:

- parce que nous voulons la décrémenter au lieu de l'incrémenter;
- parce que nous voulons l'augmenter de 3;
- parce que nous voulons la multiplier par deux;

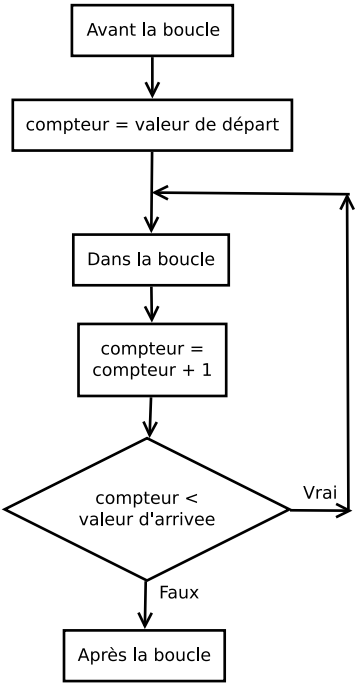
Bref, en un mot comme en cent, l'incrémentation « classique » du compteur ne correspond pas à nos besoins, et que nous sohaitons donc faire évoluer sa valeur de manière différente.

Les boucles sont « interchangeables »

Dans bien des cas, les différentes sortes de boucles sont plus ou moins interchangeables, dans le sens où, en modifiant un peu la logique, il est possible d'utiliser une boucle « tant que » à la place d'une boucle « jusque » et inversement.

Je vous ai d'ailleurs fait remarquer plus haut qu'une boule « pour » n'était qu'une adaptation de la boucle « tant que », mais nous pourrions tout à fait obtenir un résultat similaire avec une boucle « jusque », la preuve:

Figure 17.5. Une autre boucle « pour » en flowchart



La très grosse différence avec la première tenant dans le fait que celle-ci sera exécutée au moins une fois quoi qu'il arrive.

Note

La très grosse différence entre une boucle « Tant que » et une boucle « Jusque » tient essentiellement dans le fait que les actions qui doivent être (éventuellement) répétées dans la boucle seront effectivement exécutées une première fois.

Ainsi, les actions devant être répétées à l'intérieur d'une boucle « Tant que »pourront ne jamais être exécutées du tout si la condition permettant de rentrer dans la boucle ne sont jamais remplies, alors qu'elles seraient – en théorie – systématiquement exécutées au moins une fois dans le cadre d'une boucle « Jusque ».

Avertissement

« En théorie », les actions se trouvant dans une boucle « Jusque » seront effectuées une première fois « quoi qu'il arrive ». Mais il va de soi que nous pouvons également modifier ce comportement en ajoutant – par exemple – une structure conditionnelle destinée à « empêcher » les actions de la boucle d'être exécutée.

Avertissement

Il est donc tout à fait correct de dire que les différents types de boucles sont « interchangeables ».

Il est cependant très important de se rendre compte que le choix du type de la boucle que nous pourrons faire va très fortement influer sur la qui est mise en place dans la boucle.

Si, à partir du moment où cette logique est correcte, il est difficile de considérer qu'un algorithme est erroné, il faut cependant garder en tête le fait que, tout correct qu'il puisse être, l'algorithme risque de ne pas être « aussi efficace » que ce que l'on aurait pu souhaiter.

Le choix adéquat du type de boucle pourra donc avoir un impacte majeure sur l'efficacité générale de l'algorithme.

Glossaire

Les boucles

Boucle	Le terme de **boucle** représente toute situation dans laquelle un algorithme effectuera une (ou plusieurs) action(s) de manière répétée
Boucle « Tant Que »	Une **boucle « Tant que »** représente une situation dans laquelle l'algorithme vérifiera une condition avant d'exécuter les instructions qui se trouvent dans une boucle. Si la condition n'est pas vérifiée, la boucle n'est pas exécutée.
Boucle « Jusque »	Une **boucle** « **Jusque** » est une boucle dont on peut partir du principe que la logique sera exécutée au minimum une fois.
Boucle « Pour »	Une *boucle « Pour »* est une boucle qui sera exécutée un nombre bien particulier de fois et dont la logique pourra en permanence disposer d'une donnée représentant spécifiquement le nombre de fois que la boucle a déjà été exécutée.
Condition d'entrée	Le terme de *condition d'entrée* est utilisé pour désigner la condition qui sera vérifiée pour déterminer si l'algorithme doit passer « une fois de plus » par les instructions contenue dans la boucle. La différence majeure qui existe entre une boucle « Jusque » et une boucle « Tant que » résidant dans le moment où cette conditon est vérifiée dans le flux d'exécution normal.

Chapitre 18. Les limites du flowchart

Le flowchart est un moyen très élégant de représenter des algorithmes, et présente – tres certainement – l'énorme intérêt d'être relativement facile à comprendre vu que chaque pyctogramme représente une action bien particulière, un élément bien particulier de l'algorithme.

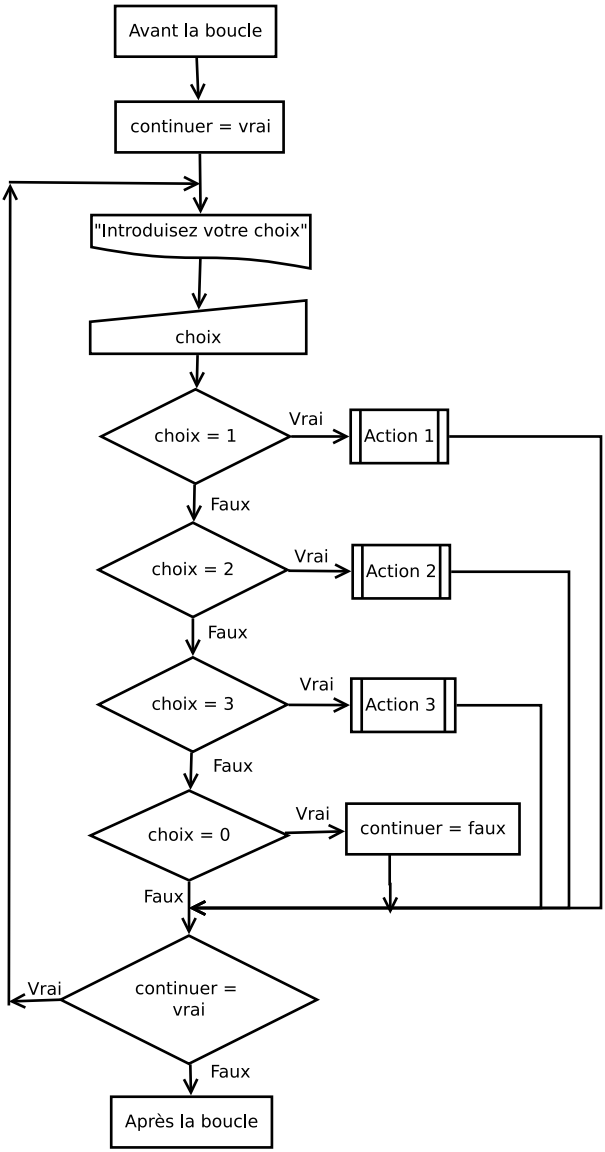
Un « plat de spagetti »

Mais il souffre également de deux problèmes majeurs:

Le premier est du à la manière dont les différentes parties de l'algorithmes sont reliées entre elles: même en y portant toute notre attention, il est fréquent de se retrouver avec tellement de flèches dans tous les sens que l'algorithme fini par ressembler à un « gros plat de spagetti ».

Imaginons que nous voulons demander un choix « en boucle » à l'utilisateur, et que nous voulions effectuer différentes actions en fonction de ses choix . Voici à quoi pourrait ressembler le flowchart:

Figure 18.1. Quand le flowchart atteint ses limites



Vous remarquerez que je suis encore bien gentil, et que je me suis limité à une boucle et à quatre choix, dont la plupart fait appel à un algorithme externe. Je vous laisse imaginer à quoi il aurait pu ressembler si l'algorithme avait été un peu plus compliqué...

Le poids de son histoire

Le deuxième problème dont souffre le flowchart est – tout simplement – le poids de son histoire.

En effet, il suffit regarder les différents pyctogrammes utilisés pour se rendre compte que cette méthode a été dévloppée au tout début de l'informatique, à l'époque où l'on programmait encore les ordinateurs en langage machine ou – dans le meilleur des cas – en assembleur.

Vous l'aurez d'ailleurs deviné lorsque je vous ai signalé que j'utilisais en réalité le pyctogramme réservé aux bandes relais pour représenter l'affichage.

Mais, observez bien ce diagramme...

L'exemple que je vous présente correspond finalement la notion de boucle « jusque ». Mais j'aurais tout aussi bien pu ajouter un compteur qui aurait été incrémenté à chaque passage de la boucle et créer ce qui serait devenu une boucle « pour » qui aurait été exécutée dix fois. Le diagramme serait resté sensiblement identique.

Le fait est que, lorsque l'on crée une boucle dans n'importe quel langage de programmation, et que l'ordinateur décide d'exécuter une fois de plus les instructions qu'elle contient, il va tout simplement ... retourner à l'adresse mémoire à laquelle se trouve la première instruction de cette boucle.

C'est d'ailleurs ce que représente la flèche qui part de la branche « vrai » du test `continuer = vrai` et qui semble se « balader à contre courant »: retourne à l'endroit de ton code où la prochaine instruction que tu exécutera sera `afficher "Introduisez est votre choix?"`

Le fait, disais-je donc, est que l'utilisation du flowchart est apparue dans les tous premiers temps de l'informatique, à une période ou la notion de « programmation structurée » n'était pas encore apparue.

C'est la raison pour laquelle les différents types de boucles sont représentées sensisblement de la même manière. Et c'est aussi l'une des principale faiblesses de la technique: à moins de suivre spécifiquement « la bonne ligne », il est tout à fait possible de « louper » le fait que nous sommes effectivement face à une boucle.

Pire encore: presque tous les langages fournissent une instruction qui permet de « sauter » à une ligne particulière (ou, le plus souvent, à un repère bien particuier dans le code). Bien que ce mode de fonctionnement corresponde exactement à ce que l'ordinateur fera, la technique en elle-même est une véritable calamité, ne serait-ce que parce que cela « brise » le flux d'exécution « logique » de notre algorithme.

Si cela ne tenait qu'à moi, je vous avouerai que cette instruction (dont je ne ferai plus mention par la suite) aurait déjà été supprimée depuis bien longtemps de **tous** les langages de programmation.

Pour être tout à fait honnête, l'un de mes profs menaçait de nous jeter par la fenêtre (du deuxième étage!) si nous utilisions cette instruction... Cela aide énormément à apprendre à s'en passer!

Chapitre 19. Le Nassi-Shneidermann

Une manière différente de représenter un algorithme

Je vous ai fait part à peine plus haut des limites dont souffrait le flowchart. Voici une méthode qui lève la plupart d'entre elle. Elle est appelée *Nassi-Shneidermann*, du nom des inventeurs de la technique. On y fait également régulièrement référence au travers des termes *structurogramme* ou *graphe NSD*.

Cette technique est très intéressante dans le cadre de la *programmation structurée* parce qu'elle permet de représenter bien plus clairement certains concepts (comme les boucles), dont nous avons vu qu'elles atteignaient les limites du flowchart.

Il faut cependant reconnaître que la « perte » des pyctogrammes du flowchart rend peut-être les choses « moins faciles » à appréheder quand on n'en a pas l'habitude.

Qu'est-ce qui change?

Soyons clairs, en dehors de l'objectif poursuivi par le flowchart et le Nassi-shneidermann, il **n'y a aucun point commun** entre les deux techniques:

Là où le **flowchart** utilise des pictogrammes spécifiques pour représenter les différentes actions, le **Nassi-Shneidermann** ne fait aucune différence entre une opération auxilière, l'appel d'un autre algorithme, l'affichage d'information ou le fait de récupérer des données introduites par l'utilisateur.

Là où le **flowchart** va relier les différentes instructions entre elles grâce à des flèches, les instructions seront exécutées strictement dans l'ordre dans lequel elles apparaissent dans un **Nassi-Sheidermann** (en partant du haut et en se dirigeant vers le bas du diagramme).

Là où le **flowchart** utilise un pictogramme pour représenter le début de l'algorithme et un autre pour en représenter la fin, alors que ces deux pyctogrammes peuvent se retrouver finalement « n'importe où » dans l'algorithme, en **Nassi-Shneidermann**, le début de l'algorithme est forcément « en haut » du diagramme, et sa fin est focément ... en bas du diagramme.

Alors que les boucles sont « relativement » difficiles à identifier dans un *flowchart*, il est facile de les identifier, non seulement en se disant « tiens, voilà une boucle », mais, surtout en se disant « tiens, voilà une boucle de <tel type> » en *Nassi-Shneidermann*.

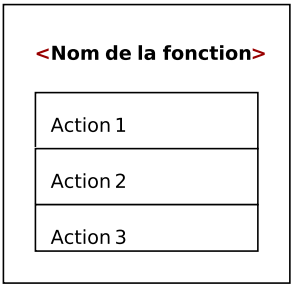
Au vu de toutes ces différences, nous pouvons presque nous demander si nous sommes bel et bien en face de deux techniques poursuivant le même but. Mais je peux pourtant vous assurer que tel est bien le cas.

Un carré dans un carré

En Nassi-Sheiderman, la fonction est représentée par un carré qui contient un autre carré dans laquelle la logique prendra place « dans l'ordre du flux d'exécution ».

Ainsi, un Nassi-Shneidermann classique pourrait ressembler à quelque chose comme

Figure 19.1. Un Nassi-Shneidermann classique



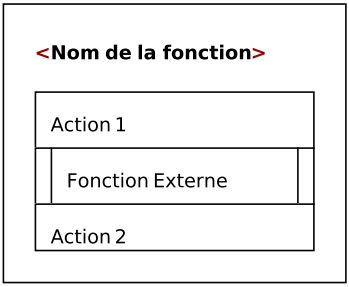
Les appels externes

Voici sans doute le seul point commun que nous pourrions trouver entre le flowchart et le Nassi-Shneidermann: l'appel de fonctions externes.

En effet, l'appel d'une fonction externe est clairement mis en évidence dans le Nassi-Shneidermann par l'adjonction de « côtés » de part et d'autre de l'action en elle-même, tout comme c'est le cas en flowchart.

Ainsi, l'appel d'une fonction (ou d'une procédure) externe sera-t-il représenté sous la forme de

Figure 19.2. Appel de fonction externe en Nassi-Shneiderman

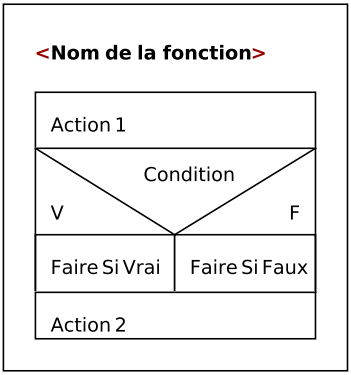


Les tests

Les tests « vrai / faux » seront représenté par un triangle divisant littéralement le diagramme en deux, ce qui permet d'avoir, sur une même ligne, les actions à entreprendre si la condition est vérifiée et l'alternative.

Pour savoir ce qu'il faut faire à partir du résultat du test, il « suffit » donc de choisir la colonne qui y correspond

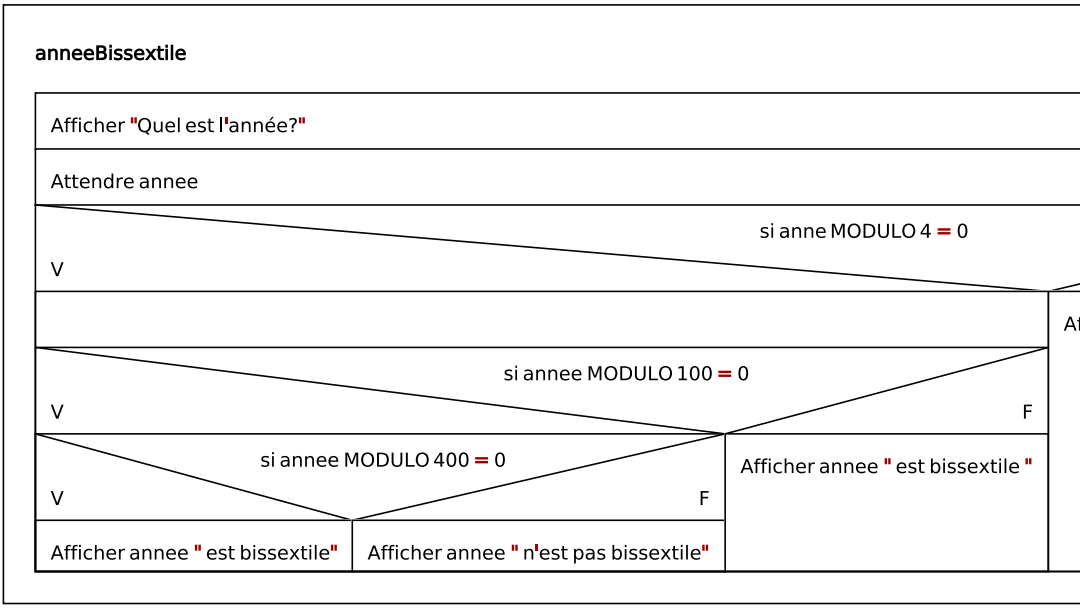
Figure 19.3. Un test en Nassi-Shneidermann



Comme je vous l'ai déjà fait remarquer, il est tout à fait possible de représenter n'importe quel algorithme sous n'importe quelle forme.

L'algorithme que je vous présentais alors pour définir si une année est bissextile pourrait donc sans aucun problème être représenté sous la forme d'un Nassi-Shneiderman. Il prendrait alors une forme proche de

Figure 19.4. Un exemple de test simple en Nassi-Shneidermann



Note

Je n'ai absolument modifié d'acune manière l'algorithme précédent. Je n'ai fait que le représenter d'une manière différente.

Il souffre donc exactement du même défaut que son prédécesseur, à savoir: le fait qu'il n'est peut-être pas aussi efficace que ce que nous pourrions souhaiter.

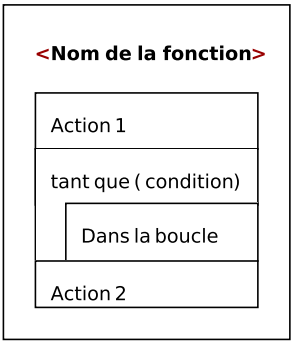
Mais nous reviendrons sur ce point plus tard.

Les boucles « Tant que »

Les boucles « Tant que » sont représenté sous la forme d'un L placé à l'horizontale regroupant l'ensemble des actions à entreprendre dans la boucle.

Elles prennent donc la forme de

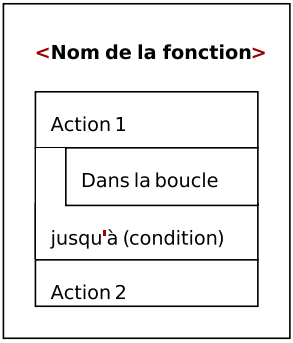
Figure 19.5. Une boucle « Tant que » en Nassi-Shneidermann



Les boucle « Jusque »

Les boucle « jusque » ne font en définitive qu'inverser les actions et les conditions par rapport à la boucle « tant », en Nassi-Shneidermann. Si bien qu'elles ressemblent à quelque chose comme

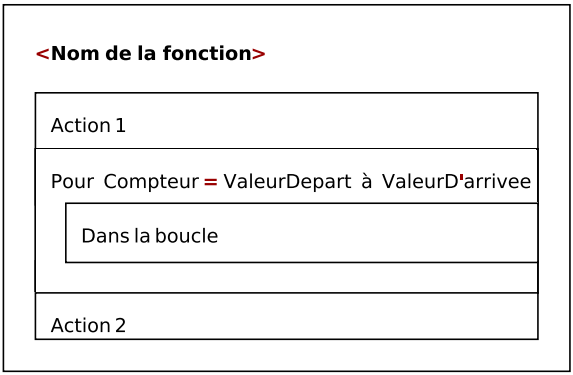
Figure 19.6. Une boucle « Jusque » en Nassi-Shneidermann



Les boucles « Pour »

Enfin, les boucles « Pour » prennent la forme d'une pince, ce qui les fait ressembler à

Figure 19.7. Une boucle « Pour » en Nassi-Shneidermann



Chapitre 20. Paramètres et retours de fonction

Je tentais plus haut de vous convaincre de l'importance de respecter le principe dit « de la responsabilité unique ».

Peut-être ne l'avez vous pas remarqué, mais l'algorithme que je vous présentais plus haut pour déterminer si une année est bissextile **ne respecte pas** ce principe.

Avant de vous démontrer ce point particulier, j'implore votre indulgence, car j'ai une sainte horreur de ceux qui jouent à « faites ce que je dis, pas ce que je fais ». Or, c'est justement ce que j'ai fait dans le cas présent.

La seule raison valable qui puisse m'avoir poussé à agir de la sorte, c'est que, au moment où je vous présentais cet algorithme, nous ne disposions pas de deux notions indispensables pour pouvoir appliquer correctement ce principe.

Il est donc plus que temps d'aborder ces deux notions et de vous démontrer que j'ai effectivement péché.

Si l'on observe attentivement l'algorithme de anneeBissextile que je vous ai donné, on se rend compte qu'il prend en charge un grand nombre de responsabilités. Jugez par vous même:

1. Affichier "Quelle est l'annee";

2. récupérer l'année introduite par l'utilisateur;

3. Tester l'année pour savoir si elle est divisible par quatre, par cent et par quatre-cent;

4. afficher le résultat du test.

On peut donc tourner les choses comme on le veut, mais cette fonction prend au final quatre responsabilités distinctes, ce qui en fait donc trois de trop par rapport au principe de la responsabilité unique.

Certains d'entre vous essayeront peut-être de se convaincre que « ce n'est quand même pas si grave », que « il est complêtement maboule, ce mec! ».

Si je ne peux pas forcément leur donner tort pour ce qui est de la deuxième phrase, pour ce qui est de la première, je dois de m'imposer en disant que « *c'est pire que grave: c'est catastrophique* ».

Mon discours a, en effet – pour ceux qui ne s'en seraient pas encore apperçus – toujours été qu'un algorithme est indépendant du langage qui sera utilisé. Et je ne reviendrai évidemment pas sur ce discours qui reste pleinement d'actualité.

Mais mon discours n'a pas encore été tout à fait complet sur le sujet. Car, en plus d'être totalement indépendant du langage utilisé, un algorithme doit – sauf cas particuliers – être totalement indépendant des circonstances dans lesquelles il est utilisé. Mais cela mérite bien sur quelques explications.

Quand je vous ai présenté cet algorithme, je vous ai dit

Grâce aux structures décisionnelles, nous pouvons « enfin » envisager de donner un « semblant d'intelligence » à notre algorithme.

Par exemple, nous pouvons définir un algorithme qui

1. **demande** en quelle année nous sommes et

2. **affiche** l'année suivie de " est bissextile" ou " n'est pas bissextile" en fonction de la situation.

En utilisant explicitement le terme « *demande* »,je m'enfermais moi même dans une logique dans laquelle il semble évident que l'année qui sera utilisée par l'algorithme serait introduite par l'utilisateur.

De la même manière, en utilisant le terme « *affiche* », je m'enfermais moi-même dans une logique dans laquelle le résultat de l'algorithme ne pourrait être utilisé que pour... provoquer un affichage.

Oui, mais, si on y réfléchit un tout petit peu, on se fout pas mal d'où la valeur de l'année peut bien venir!!! Qu'elle ait été introduite par l'utilisateur, qu'elle ait été récupérée dans le BIOS de l'ordinateur ou qu'elle ait été récupérée au travers du réseau (internet), ce qui importe, c'est la valeur qui représente l'année en question.

Et, de la même manière, on se fout pas mal de savoir si le résultat de l'algorithme sera affiché ou – à l'inverse – s'il sera utilisé, par exemple, pour déterminer si le 29 avril peut être considéré comme une date valide ou non.

Cela sort complètement du cadre de l'algorithme dont le seul but est... de déterminer si une année donnée est bissextile ou non, et cela, quelle que soit l'origine de la valeur utilisée pour l'année et quel que soit l'utilisation que l'on fera du résultat obtenu.

Et cela nous ramène donc aux deux notions qui nous manquent.

Un peu plus tôt, je vous ai indiqué que l'une des questions essentielles qu'il fallait se poser était « Où vais-je trouver les données (dont l'algorithme a besoin)? ».

L'une des réponses possible à cette question est – tout simplement – « Elle sera fournie par l'élément qui fait appel à l'algorithme ».

L'énorme avantage, c'est que l'on n'a plus vraiment besoin de s'inquiéter de l'origine effective, « réelle », de la donnée: nous savons que l'algorithme aura besoin d'une donnée particulière, qu'il est incapable d'en définir la valeur « par lui-même », mais qu'il a besoin de cette donnée pour pouvoir « faire son job » et donc que « l'élément qui fera appel » à l'algorithme devra... lui fournir cette donnée d'une manière ou d'une autre.

Selon le point de vue d'où l'on observe les choses, de telles données seront désignées sous le terme de *paramètre* (attendu par l'algorithme), ou sous le terme d'*argument* (fourni à l'algorithme).

D'un autre coté, un algorithme poursuit un objectif bien particulier. Deux situations peuvent dés lors se présenter, à partir du moment où l'on se rend compte qu'un algorithme ne vaut que par l'utilisation qui en est faite:

Soit l'algorithme se « suffit à lui-même », dans le sens où « l'élément qui fait appel » à l'algorithme ne s'attend pas à obtenir une réponse de la part de l'algorithme; soit l'algorithme devra « fournir une réponse » à l'élément qui y a fait appel.

Pour que l'algorithme soit en mesure de fournir une réponse à l'élément qui y a fait appel, nous définirons ce que l'on appelle couremment une *valeur de retour*. Comme cette valeur de retour sera disponible pour une utilisation bien particulière,

son utilisation sera généralement définie en indiquant le *type* de la valeur que l'algorithme doit renvoyer, que l'on désigne généralement sous le terme de *type de retour*.

L'algorithme permettant de déterminer si une année est bissextile ou non a besoin de ces deux notions pour pouvoir fonctionner correctement.

En effet, l'année à prendre en compte est totalement indépendante de l'algorithme, et l'algorithme n'a – a priori – aucun moyen d'en déterminer la valeur par lui-même; surtout si nous acceptons l'idée que cette valeur peut venir de « n'importe quelle source ».

De la même manière, la seule chose que nous attendons de la part de cet algorithme est de nous dire si une année donnée est bissextile ou non. Peu importe l'origine de l'année en question, peu importe ce que l'on fera de la réponse obtenue: ces deux informations sont du seul recours de... l'élément qui fera appel à l'algorithme.

Paramètres d'entrée et / ou de sortie

De manière générale, on peut distinguer deux grandes catégories de paramètres.

D'une part, il y a les paramètres qui ne seront pas modifiés par la fonction – ou, du moins, dont les modifications ne seront pas répercutées sur la donnée qui a été transmise en argument dans la fonction appelante – et, de l'autre, il y a les paramètres dont les modifications appportées par la fonction seront effectivement répercutées sur la donnée qui a été transmise comme argument par la fonction appelée.

Nous désignerons souvent la première catégorie sous le terme de *paramètre d'entrée* alors que nous désignerons la seconde catégorie sous le terme de *paramètre de sortie*.

La grosse différence qui existe entre ces deux catégories est le fait que les *paramètes d'entrée* sont clairement indispensables au bon fonctionnement de la fonction, ne serait-ce que parce qu'ils représentent des données auxquelles la fonction n'a aucun moyen d'avoir accès; alors que les *paramètres de sorties* ne sont généralement fournis à la fonction *que pour lui permettre d'y apporter des modifications*.

A priori, lorsqu'un paramètre est désigné comme étant un *paramètre de sortie*, la fonction n'utilisera pas la valeur de ce paramètre dans sa logique décisionnelle; si bien qu'il serait envisageable de transformer ce paramètre en une variable locale destinée à être renvoyée en tant que retour de fonction.

Ceci dit, nous sommes régulièrement confrontés à des paramètres qu'il s'agirait de classer dans la catégorie des *paramètres de sorties* – parce que les modifications qui y seront apportées par la fonction appelée seront répercutées au niveau de la variable qui a servi d'argument dans la fonction appelante – qui interviennent néanmoins dans la logique décisionnelle de la fonction, ce qui les ferait *de facto* entrer dans la catégorie des *paramètres d'entrée*.

Nous désignerons « logiquement » ce genre de paramètre comme étant des *paramètres d'entrée / sortie*.

Les effet de bord

Nous essayerons généralement autant que possible d'éviter l'utilisation de **paramètre de sorties** dans les fonctions, car ils occasionnent ce qu'il convient d'appeler un **effet de bord**.

Nous disons qu'une fonction présente un effet de bord lorsqu'elle va avoir un impact au niveau de la valeur d'une donnée qui existe en dehors de cette fonction.

De manière générale, nous essayerons autant que faire se peut d'éviter ce genre de situation, ne serait-ce que parce que les effets de bord peuvent créer une certaine confusion dans l'esprit de celui qui lira le code de la fonction appelante et qui risque – s'il n'a pas pleinement conscience du fait que la fonction appelée génère un effet de bord – de s'étonner que la valeur d'une donnée avant l'appel de la fonction est différente de sa valeur après que la fonction ait été appelée.

Nous verrons d'ailleurs plus tard qu'il existe de nombreuses situations dans lesquelles un tel effet de bord occasionne énormément de risques du point de vue de la logique elle-même.

Cependant, nous verrons également plus tard qu'il existe de nombreux cas dans lesquels nous n'avons finalement pas d'autre solution que de provoquer un effet de bord; dans lesquels nous n'avons en définitive que le choix entre provoquer un effet de bord et celui de ne pas être en mesure de respecter le SRP.

Dans de telles situations, nous avons généralement intérêt à avoir recours à une fonction qui occasionne un effet de bord plutôt que de décider de déroger à un des principes majeurs de développement.

Représenter les paramètres

Lorsque l'on crée un algorithme, il est primordial de faire en sorte que la personne qui l'aura devant les yeux qui qui devra le manipuler soit en mesure de le comprendre le plus facilement possible.

Après tout, il faut se souvenir que, au final, notre algorithme devra être compris par quelque chose d'aussi bête qu'un ordinateur, c'est à dire, par quelque chose qui ne connaît que deux valeurs (1 et 0) et qui ne dispose que d'un nombre particulièrement instructions.

Mais il faut assi se méfier de la personne qui s'occupera de traduire notre algorithme dans un langage de programmation quelconque, car, si on laisse « traîner » la moindre ambiguïté quant à l'usage qui sera fait d'une donnée, nous courrons le risque que la personne qui se chargera de traduire l'algorithme dans un langage donné commence à « interpréter » ce qui se passe dans les « zone d'ombres » laissées par l'algorithme.

Et le fait que la personne qui traduira l'algorithme dans un langage de programmation particulier sera très souvent la personne qui a créé l'algorithme n'aura en réalité aucun impact sur ce genre de problème.

Pour que la personne qui devra traduire notre algorithme dans un langage de programmation quelconque – dont il est toujours préférable de se dire que ce sera quelqu'un d'autre, et qu'il risque d'être distrait en le faisant – ne puisse avoir aucun doute quant à l'utilité d'une donnée qui est attendue sous forme d'un paramètre par notre algorithme, il est important de lui transmettre deux informations simples que sont le « quoi » et le « pour quoi ».

Le « quoi » doit permettre à la personne de connaître non seulement le nom qui sera donné à la fonction dans notre algorithme, mais aussi le type de cette donnée, pour qu'il puisse l'indiquer dans son code; alors que le « pour quoi » devra lui permettre de déterminer s'il a affaire à un paramètre d'entrée, à un paramètre de sortie ou à un paramètre d'entrée / sortie.

L'idéal est donc, pour chaque paramètre requis par un algorithme, de l'indiquer clairement sous une forme proche de `IN : TypeDeDonnée nomParamètre` dans laquelle nous utiliserons bien sur IN pour les *paramètres d'entrée*, OUT pour les *paramètres de sortie* et IN/OUT pour les *paramètres d'entrée / sortie*.

La valeur de retour

Il semble évident que, lorsque l'on fait appel à un algorithme quelconque, notre but est forcément d'obtenir un résultat prédictible et reproductible.

Ce qui peut parraître moins évident, c'est le fait le fait que le résultat que nous obtiendrons en faisant appel à un algorithme particulier n'a réellement de l'intérêt qu'au travers ... de l'utilisation qui pourra être faite de ce résulat par la suite.

En effet, aucune des choses que nous pouvons mettre en place dans une application n'a d'intérêt si elle n'est pas utilisée:

Nous pouvons créer le meilleur algorithme du monde, s'il n'est pas utilisé, nous avons simplement « perdu notre temps » à le mettre au point.

De la même manière, si nous ne pouvons pas utiliser le résultat d'un algorithme quelconque, cela nous pousse à nous poser une question embarassante, qui risque d'en fâcher plus d'un : pourquoi avoir « perdu notre temps » à calculer se résultat?

Dés lors, pour qu'un algorithme puisse avoir une utilité réelle, il est souvent nécessaire qu'il puisse ... renvoyer (à celui qui y a fait appel) une valeur qui correspondra au résultat du traitement effectué.

Le plus souvent, ce qui nous intéressera sera le type de la donnée qui correspond au résultat obtenu, car, du fait des boucles et des structures décisionnelles qui peuvent jalonner un algorithme, la valeur en elle-même du résultat peut être « incertaine »; du moins, aussi longtemps que l'on ne dispose pas de l'ensemble des données qui seront transmises comme paramètre à notre algorithme.

C'est la raison pour laquelle nous désignerons généralement la donnée qui sera renvoyée par un algorithme à celui qui y a fait appel sous le terme de *type de retour* ou, par facilité, sous le terme de *valeur de retour*.

Généralement, nous introduirons le type de la valeur renvoyée par un algorithme en faisant suivre le nom de l'algorithme de deux points `:` et du type de la valeur renvoyée, sous une forme qui sera donc fort proche de `nomDeLalgorithme : typeDeRetour`

Année bissextile seconde

Nous disposons maintenant de toutes les informations qui nous permettront de définir un algorithme efficace permettant de vérifier si une année donnée est bissextile ou non.

En pseudocode, cet algorithme prendrait une forme proche de

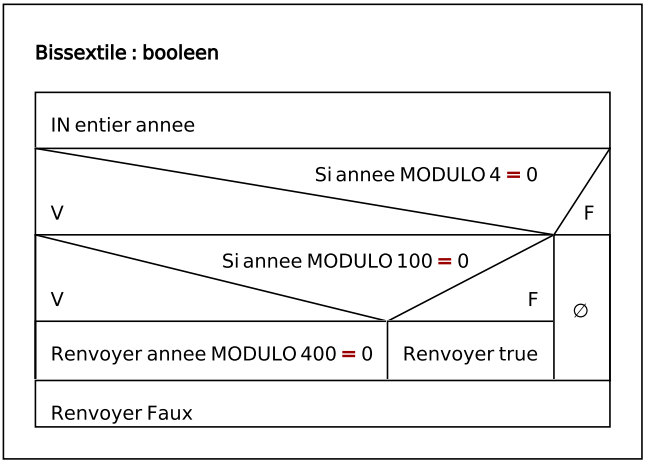
Exemple 20.1. Annee bissextile seconde

```
bissextile : booleen
```

```
in Entier annee
Si(annee MODULO 4 = 0)
    Si(annee MODULO 100 = 0)
       renvoyer annee MODULO 400 = 0
    Fin Si
    renvoyer Vrai
Fin Si
Renvoyer Faux
Fin  bissextile
```

En Nassi-Shneidermann, cet algorithme pourrait prendre une forme proche de

Figure 20.1. Une boucle « Tant que » en Nassi-Shneidermann



Utiliser les branches else ... ou non

Vous l'aurez remarqué dans les deux formes que je viens de présenter pour cet algorithme que j'ai fait en sorte que l'instruction « Renvoyer Faux » se trouve complètement en dehors de la structure décisionnelle constituée par le test `si annee MODULO 4 = 0`.

Peut être aurez vous été surpris de cette décision, et peut-être même pourriez vous croire que cet algorithme est incorrect; que j'aurais du mettre ce retour de valeur dans la branche « Faux » du test.

S'il est vrai que j'aurais *pu* mettre ce retour dans la branche « Faux » du test, je peux néanmoins vous assurer que cet algorithme est fournira toujours un résultat résultat parfaitement correct.

En effet, tous les cas qui pourraient survenir si la condition `annee MODULO 4 = 0` est vérifiée vont provoquer la « fin prématurée » de l'algorithme, soit à cause de l'instruction « Renvoyer Vrai » si l'année considérée est divisible par quatre mais qu'elle n'est pas divisible par 100, soit en renvoyant le résultat du test annee MODULO 400 = 0 si l'année est divisible par 100.

Les instructions qui arrivent en dehors de la structure conditionnelle initiée par `annee MODULO 4 = 0` ne seront donc atteintes que si cette condition **n'est pas** vérifiée; exactement comme cela aurait été le cas de la branche « Faux » du test. La logique est donc tout à fait correcte.

La seule chose, c'est que je m'évite ainsi de devoir préciser que je crée une alternative, ce qui rend finalement l'algorithme « plus simple » (surtout dans le cadre d'une représentation sous la forme d'un pseudo code).

En outre, l'exprérience démontre que les outils utilisés par certains langages ont – sous certaines conditions – tendance à se plaindre lorsqu'ils sont confrontés à une fonction qui est sensée renvoyer une fonction mais que la dernière instruction dans le « flux d'exécution normale » n'est pas une instruction provoquant le renvoi d'une valeur.

Or, si l'on venait à placer l'instruction `Renvoyer Faux` dans la branche « Faux » du test (ce qui serait tout à fait correct, d'un point de vue purement logique), nous n'aurions plus aucun chemin d'exécution qui ne mène pas tout droit à « une fin prématurée » de l'algorithme.

Et, partant de là, le fait de rajouter une instruction de renvois de valeur (dont nous serions déjà en peine de définir la « meilleur valeur possible », vu que toutes les valeurs sont déjà prises en compte) dans le flux « normal » d'exécution de l'algorithme (comprenez : en dehors de tout test) ne ferait que... rajouter une instruction qui n'aurait aucune chance d'être atteinte, quelles que soit les circonstances, quelle que soit la valeur que nous puissions associer au paramètre `annee`.

Cela ne veut bien sûr absolument pas dire qu'il ne faudra **jamais** utiliser la possibilité qui nous est donnée de définir des alternatives. Cela veut seulement dire que, parfois, quand les circonstances sont favorables, il n'est pas **forcément** nécessaire d'y recourir.

Glossaire

Les données exeternes

Paramètre	Un *paramètre* représente une donnée que l'utilisateur d'un algorithme doit lui fournir pour que l'algorithme soit en mesure d'effectuer le travail que l'on attend de sa part.
Argument	Un argument représente la donnée qui sera utilisée comme paramètre pour un « algorithme secondaire » au niveau de l'algorithme qui y fait appel
Paramètre d'entrée	Un *paramètre d'entrée* représente un paramètre qui sera transmis à un algorithme afin qu'il puisse l'utiliser afin de prendre différentes décision et dont les éventuelles altérations de la valeur que le paramètre pourrait subir ne seront pas répercutées, au niveau de l'algorithme appelant, sur la donnée ayant servi d'argument.
Parmaètre de sortie	Un *paramètre de sortie* représente un paramètre qui sera fourni à l'algorithme dans le seul but que ce dernier puisse en altérer la valeur et que les altérations apportées soient répercutée, au niveau de l'algorithme appelant, sur la donnée ayant servi d'argument. A priori, la valeur un paramètre de sortie ne devrait pas être utilisé par l'algorithme dans sa logique décisionnelle.
Paramètre d'entrée / sortie	Un *paramètre d'entrée / sortie* est un paramète qui est clairement utilisé pour servir à la fois de paramètre d'entrée – dont la valeur sera utilisée par l'algorithme dans sa logique décisionnelle – et de paramètre de sortie (dont les modification seront répecutée au niveau de la donnée ayant servi d'agrument dans l'alogirhme appelant)
Effet de bord	Un effet de bord représente les modifications subies par la donnée ayant servi d'argument à un algorithme lorsque l'argument est utilisé en tant que paramètre de sortie (voire, en tant que paramètre d'entrée / sortie).
Valeur de retour	La valeur de retour représente le type de la donnée qu'un algorithme est capable de renvoyer à celui qui y a fait appel afin de lui faire connaître le résultat de l'exécution.

Partie III. La syntaxe de base

Dans cette partie, j'aborderai la syntaxe de base propre au C++ ainsi que la manière de mettre les différentes notions abordées d'un point de vue théorique dans la section précédante en œuvre.

Chapitre 21. La fonciton principale

Partie IV. Les principes de l'orienté objet

Dans cette partie, j'aborderai la conception orientée objet d'un point de vue purement théorique. Parmi les aspects étudiés, on peut citer:

- Penser en termes de services
- la loi de Déméter
- les principes SOLID non vu précédemment
- Sémantique de valeur Vs Sémantique d'entité
- Les patrons de conception

Chapitre 22. Organiser son projet

Partie V. C++ et l'orienté objet

Dans cette partie, j'aborderai les techniques qui permettent au C++ de tirer le meilleur parti du paradigme orienté objet. On peut citer parmis celles-ci:

- les moyens d'encapsulation
- l'amitié
- Assurer la sémantique d'entité
- l'héritage public
- l'héritage privé
- l'héritage multiple
- Manipuler des hiérarchies de classes

Chapitre 23. Assurer l'encapsulation

Partie VI. Le paradigme générique

Dans cette partie, nous aborderons une approche très particulière de la programmation appelée *programmation générique*

Chapitre 24. Définition

Nous pourrions résumer la programmation générique en une simple phrase proche de

> La programmation générique permet, à défaut de savoir quel type de donnée sera manipulé, de définir très clairement la manière dont ces données seront manipulées.

Partie VII. C++ et le paradigme générique

Pour être tout à fait honnête,

Chapitre 25. Comment ca fonctionne?

Partie VIII. La bibliothèque standard

Comme de nombreux langages de programmation, C++ vient avec une série de fonctionnalités fournie par ce qu'il convient d'appeler la *bibliothèque standard* propre au langage.

Il s'agit le plus souvent de types de données, de fonctions et de variables globales destinés à faciliter la vie du développeur dans son « travail de tous les jours ».

Contrairement à d'autres langages (comme C# ou java) qui viennent avec une bibliothèque standard particulièrement bien fournie au point – à mon sens – de « partir un peu dans tous les sens », la bibliothèque standard de C++ n'essaye de répondre qu'à un nombre « relativement restreint » de problèmes.

Chapitre 26.

Partie IX. Annexe

Dans cette partie, j'aborderai une série de thèmes qui n'avaient décidément par leur place ailleurs. Parmi ceux-ci, nous pouvons citer:

- une introduction à la manière d'organiser son projet
- une rapide introduction à CMake
- une introduction rapide du processus appelé compilation

Chapitre 27. Organiser son projet

Sources et lectures

[PLPAP] *Programming Languages Principles And Practices*. Kenneth C Louden et Kenneth Lambert. Course Technology.

[TAOCP] *The Arrt Of Computer Programming*. Donald Knuth.

[TC++PL] *The C++ Programming Language*. Bjarne Stroutrup.

[TAC++RM] *The Annotated C++ Reference Manual*. Bjarne Stroutrup et Margaret A. Ellis.

[TDEC++] *The Design and Evolution of C++*. Bjarne Stroutrup.

Index

L

M

N

O

P

S

T

V