Chapitre 6

Conception globale d'un programme

Dans les chapitres précédents nous nous sommes initiés à la programmation, plus particulièrement à la **programmation structurée**, à l'aide d'un langage de programmation particulier, à savoir le langage C. Le **jeu d'instructions** que nous avons introduit est suffisant pour programmer tout ce qui est programmable. La justification de cette affirmation est du ressort de ce que l'on appelle la *théorie de la calculabilité*, et plus spécifiquement de la *thèse de Church-Turing*. Toute nouvelle instruction que nous ajouterons désormais peut être émulée dans le mini langage C tel que nous l'avons défini jusqu'ici ; il suffit de le vérifier au cas par cas, ce que nous ferons de temps en temps. Il est donc temps de faire une pause. Nous allons commencer par réfléchir à la meilleure façon de concevoir les programmes.

6.1 Pragmatique

Jusqu'ici, nous avons présenté la syntaxe et la sémantique des instructions que nous avons introduites, et ceci au fur et à mesure. Nous allons maintenant nous intéresser à un troisième aspect, dit **pragmatique** (ou **méthodologie** de la programmation, ou **style de programmation**). Cet aspect pratique de la programmation a pour but d'indiquer comment écrire des programmes lisibles par le programmeur et ceux qui lisent le programme. Les mauvaises langues disent qu'on place en pragmatique tout ce que l'on ne peut pas placer en syntaxe ou en sémantique, c'est-à-dire tous les aspects de la programmation qui ne peuvent pas être décrits dans un cadre logique bien défini.

Présentons l'aspect pragmatique de ce que nous avons vu jusqu'à maintenant. Nous le continuerons au fur et à mesure de l'introduction de nouvelles notions de langage C, en particulier par ce que l'on appelle le *raffinement successif de la conception d'un programme.*

6.1.1 Présentation d'un programme

6.1.1.1 Passage à la ligne

Introduction.- D'un point de vue purement syntaxique, un programme en langage C peut être écrit sur une seule ligne. En fait on a la possibilité d'aller à la ligne quand on veut (ou presque ; l'exception étant qu'on ne peut pas couper les identificateurs et les constantes de type chaîne de caractères, même au niveau d'un espace) et de faire suivre tout espace par autant d'espaces que l'on veut (sauf, toujours, pour les constantes de type chaîne de caractères). Ces possibilités ont évidemment pour but de permettre d'écrire des programmes lisibles.

Exemple 1.- Comparer le programme suivant :

```
#include <stdio.h> void main(void) { printf("Bonjour");
printf("007");}
```

et le même réécrit de la façon suivante :

```
#include <stdio.h>
void main(void)
     {
     printf("Bonjour");
     printf("007");
     }
```

6.1.1.2 Indentation

Introduction.- Une des erreurs de syntaxe les plus fréquentes est de ne pas mettre autant d'accolades fermantes '}' que d'accolades ouvrantes '{' ou bien de mal les apparier. Pour éviter ceci, on a intérêt à **indenter** les programmes (c'est-à-dire à ne pas commencer chaque ligne en début de ligne) et à placer l'accolade '}' correspondant à l'accolade '{' en-dessous de celle-ci, pour obtenir une structure de la forme suivante :

```
{
---
      {
      ---
            {
            ---
```

```
            }
        ---
        }
    ---
    }
```

Exemple 2.- Comparer le programme suivant :

```
#include <stdio.h> void main(void) { float X,Y;printf("x = ");
scanf("%f",&X);While (X != 1000) {Y = (sin(X) + log(X))
/(exp(X) + 2);printf("f(%f) = %f",X,Y);printf("x
= ");scanf("%f",
&X);} }
```

et le même réécrit de la façon suivante :

```
#include <stdio.h>
void main(void)
     {
     float X, Y;
     printf("x = ");
     scanf("%f",&X);
     while (X != 1000)
           {
           Y = (sin(X) + log(X))/(exp(X) + 2);
           printf("f(%f) = %f",X,Y);
           printf("x = ");
           scanf("%f",&X);
           }
     }
```

6.1.2 Utilisation des commentaires

6.1.2.1 Notion

Lorsqu'on écrit un petit programme, et surtout au moment où on l'a écrit, ce qu'il fait est clair pour nous, ainsi que le rôle de chacune des instructions et de chacune des variables auxiliaires. Ceci devient nettement moins évident lorsqu'on le relit six mois plus tard. Cela peut ne pas être clair du tout lorsqu'il est lu par quelqu'un d'autre. On a donc intérêt à insérer des **commentaires**, c'est-à-dire des petits textes servant à expliquer ce que l'on fait mais qui ne sont pas pris en compte par le compilateur.

6.1.2.2 Syntaxe en langage C

Syntaxe.- Les deux groupes de symboles '`/*`' et '`*/`' sont réservés en C pour indiquer respectivement le début et la fin d'un *commentaire*, c'est-à-dire que tout ce qui se trouve entre ces deux symboles (ces symboles y compris) est ignoré par le compilateur.

On peut mettre n'importe quel signe entre les délimiteurs, à l'exception du délimiteur de fin de commentaire ('`*/`'). Les commentaires peuvent s'étaler sur plusieurs lignes.

Exemple.- Comparer le programme vu précédemment et le même réécrit de la façon suivante :

```
/* Calcul de f(x) = (sin(x)+ln(x))/(exp(x)+2),
pour plusieurs valeurs de x, differentes de 1000,
introduites au clavier.
On termine en proposant la valeur 1000. */

#include <stdio.h>
#include <math.h>
void main(void)
     {
     float x, y;
     printf("x = ");
     scanf("%f", &x);
     while (x != 1000.0)
           {
           y = (sin(x) + log(x))/(exp(x) + 2);
           printf("f(%f) = %f", x, y);
           printf("\nx = ");
           scanf("%f", &x);
           }
     }
```

6.1.3 Étapes de conception d'un programme

Introduction.- Concevoir un programme n'est pas en général une tâche facile. Il n'existe pas de procédé mécanique pour cela. La conception d'un programme est un véritable acte de créativité. Il existe cependant un plan à suivre.

Phases de conception d'un programme.- Le processus de conception d'un programme peut être divisé en deux phases : la **phase de conception de l'algorithme** et la **phase d'implémentation**. L'**algorithme** est un programme dans un pseudo-langage écrit en français dont on définit soi-même le jeu des ordres élémentaires et le jeu de combinaisons de tels ordres. Pour produire un programme dans un langage de programmation tel que le langage C, il faut traduire cet algorithme dans ce langage, ce qui s'appelle l'**implémentation** ou **codage**.

Définition du problème.- La première étape, à la fois de la phase de résolution et de la conception d'un programme, est d'être certain que la tâche à accomplir est **spécifiée** de façon complète et précise. Il ne faut pas considérer cette étape à la légère. On doit en particulier spécifier le nom et la nature des données et des résultats. C'est l'étape de l'établissement du **cahier des charges**.

Phase de résolution.- Beaucoup de programmeurs néophytes ne comprennent pas la nécessité de concevoir un algorithme avant d'écrire un programme. Ils essaient de court-circuiter la phase de résolution, ou tout au moins de la réduire à la seule définition du problème. Ceci semble raisonnable *a priori* puisqu'on a l'impression de faire deux fois la même chose et de gagner ainsi du temps. Le problème est que l'expérience a montré que l'on ne gagne pas de temps en procédant ainsi : la **conception en deux phases** produit beaucoup plus vite un programme qui fonctionne. Elle simplifie la phase de conception de l'algorithme en n'ayant pas à prendre soin des détails inhérents à chaque langage.

Phase d'implémentation.- Cette phase n'est pas totalement triviale. Il faut être soigneux. Il faut quelquefois émuler des instructions qui nous plairaient bien mais qui n'existent pas dans ce langage de programmation. Mais en fait plus on devient familier avec un langage plus cette étape devient une routine.

Tests.- Les tests existent dans chacune des deux phases. Si des défauts sont trouvés, il faut revoir la conception (partielle ou globale) du programme. La première phase de test se fait mentalement en exécutant les étapes de tête. Pour de grands algorithmes on peut se servir de papier et de crayon. Le programme en langage C, lui, est testé en le compilant et en le faisant tourner sur des données. Le compilateur donnera des messages d'erreurs pour certaines sortes d'erreurs, appelées en général **erreurs de syntaxe** : il manque des points-virgules, les accolades `{` et les `}` sont mal appariées, on essaie d'attribuer une valeur réelle à une variable déclarée de type entier... Lorsqu'on fait tourner le programme pour un jeu de données, il faut comparer les résultats donnés par l'ordinateur et ceux que vous connaissez pour vous apercevoir que le programme ne fait peut-être pas exactement ce qui est attendu ce qu'on appelle des **erreurs de sémantique**).

Interaction de toutes les étapes entre elles.- La façon de faire décrite ci-dessus est idéalisée. C'est la figure de base que vous devez avoir en mémoire, mais la réalité est un peu plus compliquée. En fait des erreurs sont découvertes à n'importe quel moment et vous devez revenir en arrière. Par exemple le test d'un algorithme peut révéler que la définition du problème est incomplète.

6.2 Thèse de Church-Turing

Nous avons vu que l'ordinateur est un outil d'aide aux *calculs*, autrement de mise en place de la pensée algorithmique. Nous avons donné des exemples de *calculs* mais nous n'avons pas donné de définition générale de ce qu'est un **calcul**.

Maintenant que nous sommes initiés à la programmation (ou, plus exactement, à la **programmation structurée**) à travers un langage de programmation donné (à savoir le langage C), nous allons pouvoir répondre à la question de savoir ce qu'est un calcul.

6.2.1 Restriction aux calculs sur les entiers naturels

Nous avons abordé divers types de calculs, en particulier des calculs sur les entiers naturels, sur les entiers relatifs, sur les rationnels, sur les réels, sur les mots... Nous avons déjà vu également que, pour tout ce que l'on pouvait vraiment appeler *calcul effectif*, on pouvait, *via* un codage très simple, se ramener à un calcul soit sur les entiers naturels, soit sur les mots, à notre convenance.

Pour pouvoir donner une définition générale de ce qu'est un calcul, on a intérêt à ne considérer que les calculs sur un type donné d'entités. La définition des calculs sur les entiers naturels est un bon choix.

6.2.2 Définition

Introduction.- En ce qui concerne les entiers naturels, nous avons vu comment saisir un entier, comment afficher un entier, comment effectuer certaines opérations élémentaires sur les entiers (à savoir l'addition, la multiplication et la division euclidienne) grâce à une *affectation*. Tout ceci correspond à des *instructions élémentaires*. Nous avons vu aussi comment les entiers jouent le rôle de *booléen* (en utilisant la convention selon laquelle 0 représente le faux et tous les autres entiers le vrai), d'où la manipulation de conditions. Nous avons enfin vu comment combiner ces instructions simples pour obtenir des instructions plus complexes (et donc effectuer des calculs plus compliqués) grâce aux *structures de contrôle* : le *séquencement*, les *itérations* et les *conditionnelles*.

Vous avez vu en exercice que ceci permet de programmer beaucoup de calculs. Peut-être, à un certain moment, n'avez-vous pas vu comment programmer tel calcul que l'on vous demandez mais, avec un peu d'aide, vous y êtes parvenus.

En fait les programmeurs s'aperçoivent que, pour les calculs sur les entiers naturels, il est inutile de disposer d'autre chose. Ceci conduit à la *thèse de Church-Turing*.

Thèse (**Thèse de Church-Turing**, 1936)

Tout calcul sur les entiers naturels peut être effectué en utilisant l'addition, la multiplication, la division euclidienne, l'affectation, le séquencement, les itérations et les tests.

Remarques.- 1°) Cette notion de calcul doit être claire désormais bien que l'on n'ait pas formalisé ce que l'on entend par *peut être effectué en utilisant l'addition, la multiplication, la division euclidienne, l'affectation, le séquenement, les itérations et les tests* en donnant une syntaxe et une sémantique bien précise. Ceci sera vu dans le cours dit de *calculabilité*.

2°) On appelle cet énoncé une **thèse**, et non une *définition* ou un *théorème*. En effet nous avons, d'une part, une notion intuitive de ce qu'est un calcul et, d'autre part, une définition précise d'un certain nombre de calculs (même si nous ne l'avons pas donnée explicitement ici, comme nous venons de le remarquer). Cet énoncé identifie ces deux notions. Cela va donc au-delà d'une simple définition. Ce n'est pas un théorème car la première notion est intuitive.

6.2.3 Les ordinateurs existent-ils ?

Introduction.- Nous avons dit, dans un chapitre antérieur, qu'un *ordinateur* est un outil capable d'effectuer tous les calculs. Nous avons accepté, pour commencer, de considérer toute machine présentée comme étant un ordinateur, qu'il en est bien ainsi. Maintenant que nous avons une notion précise de ce qu'est un calcul, nous pouvons nous interroger à nouveau.

Discussion.- Un ordinateur doit pouvoir effectuer des affectations, des séquencements, des itérations et des tests. Il n'y a pas de problème.

Mais un ordinateur doit aussi pouvoir manipuler les entiers naturels, *tous* les entiers naturels, aussi grands qu'ils puissent être. Et, là, nous avons vu qu'il y a un problème, tout au moins avec les ordinateurs que nous avons pu approcher. Une machine possède une mémoire finie, aussi grande soit-elle. Elle ne peut donc pas manipuler des entiers aussi grands que l'on veut. D'où la conclusion :

Les ordinateurs n'existent pas.

Les machines présentées comme *ordinateurs* en sont seulement des ébauches, plus ou moins convenables.

6.2.4 Opérations non calculables

Nous avons une notion intuitive d'opérations calculables (sur les entiers naturels). D'autre part nous avons donné, en Mathématiques, une définition générale de ce qu'est une opération sur un ensemble A, par exemple une opération binaire est tout simplement une application de $A \times A$ dans A. La question qui se pose alors est la suivante :

Toute opération sur les entiers naturels est-elle calculable ?

La réponse, due à TURING en 1936, est NON. La démonstration se fait soit *via* un argument de *cardinalité*, soit en exhibant une opération (le *problème de la halte*) dont on montre qu'elle n'est pas calculable. Ceci dépasse le niveau d'un cours d'initiation à la programmation (bien qu'il est bon d'en retenir le résultat) et sera vu dans le cours de *calculabilité*.

6.2.5 Autres opérations calculables : exemple du graphisme

Nous avons défini formellement ce qu'est une opération calculable. Nous n'avons donc, en théorie, rien besoin de plus pour programmer. Cependant rien n'empêche d'ajouter des aides à la programmation, même si elles ne sont pas indispensables en théorie. C'est ce que nous verrons dans la suite.

Les néophytes ont quand même l'impression que l'on se moque d'eux et qu'il n'est pas possible de tout programmer, en particulier tout ce qui est *graphisme*. Et pourtant le graphisme est obtenu en envoyant (sur la bonne sortie, certes) un flot de 0 et de 1, ou d'entiers si l'on préfère, interprétés par le moniteur.

Par contre, pour simplifier la tâche des êtres humains (et non des ordinateurs), des instructions de haut niveau plus compréhensibles par un être humain, seront créées. Par contre, revers de la médaille, il faut que quelqu'un les traduise en instructions de bas niveau.

6.3 Exercices

Nous avons maintenant suffisamment de matériel pour faire beaucoup de programmes (en théorie d'ailleurs tout le matériel, ce que nous dirons plus tard ne servira qu'à simplifier la mise en oeuvre).

Exercice 1.- (**Triplets pythagoriciens**)

Un **triplet** d'entiers naturels (x, y, z) est dit **pythagoricien** si, et seulement si, on a :

$$x^2 + y^2 = z^2.$$

Écrire un programme C qui demande un entier naturel n et qui affiche alors tous les triplets pythagoriciens (x, y, z) tels que : $x, y \leq n$.

Exercice 2.- (**Polynôme d'Euler**)

Euler a affirmé que la valeur du polynôme $P(n) = n^2 - n + 41$ est un nombre premier pour les entiers naturels n compris entre 0 et 39.

Écrire un programme C qui demande un entier naturel m et qui, pour tous les entiers n compris entre 0 et m, affiche la valeur de n, la valeur de $P(n)$, dit si $P(n)$ est premier ou indique son plus petit diviseur non trivial sinon.

Exercice 3.- (**Nombres parfaits**)

Un entier naturel n est un **nombre parfait** si, et seulement si, il est égal à la somme de ses diviseurs stricts. Par exemple 6 et 28 sont des nombres parfaits.

Écrire un programme C qui demande un entier naturel m et affiche les nombres parfaits inférieurs à m.

Exercice 4.- (**PGCD**)

Écrire un programme C qui demande deux entiers naturels a et b et affiche leur plus grand commun diviseur $pgcd(a, b)$.