Méthodes et Outils pour la Programmation

Grégory Bise

10 janvier 2015

Table des matières

1 Introduction **4**
1.1 Problème 4
1.2 Algorithme 4
1.3 Programme 5

2 Pseudo-Code **6**
2.1 Syntaxe 6
2.2 Structure 6
2.3 Exemple simple 7
2.4 Boucles 9
2.5 Fonctions 10
2.6 Conclusion 14

3 Programmation **15**
3.1 Pourquoi le langage C ? 15
3.2 Environnement de développement 16
3.2.1 Compilateur - GCC 16
3.2.2 IDE - Eclipse 17
3.3 Premier Programme 18
3.3.1 Écriture, compilation et exécution 18
3.3.2 Analyse 20
3.4 Variables 21
3.4.1 Noms de variable 21
3.4.2 Types de variables 21
3.5 Entrée / Sortie 22
3.6 Conditions 24
3.7 Commentaires 30
3.8 Boucles 31
3.8.1 While 31

3.8.2 Do ... While . . . 32
3.8.3 For . . . 33
3.9 Tableaux . . . 33
3.9.1 Déclarer un tableau . . . 34
3.9.2 Affectation et manipulation des éléments d'un tableau 35
3.9.3 Deux dimensions . . . 36
3.10 Fonctions . . . 37
3.10.1 Déclaration . . . 37
3.10.2 Appel . . . 38
3.10.3 Portée des variables . . . 39
3.10.4 Modularité . . . 41
3.11 Doxygen . . . 42
3.11.1 Syntaxe . . . 42
3.11.2 Génération . . . 43
3.12 Récursivité . . . 44
3.12.1 Analyse . . . 46
3.12.2 Conseils . . . 47
3.13 Pointeurs . . . 47
3.13.1 Déclaration . . . 48
3.13.2 Affectation et accès . . . 48
3.13.3 Tableaux et pointeurs . . . 49
3.13.4 Fonctions et pointeurs . . . 50
3.13.5 Allocation et libération de mémoire . . . 52
3.13.6 Conseils . . . 53
3.14 Structures . . . 53
3.14.1 Nouveau type . . . 53
3.14.2 Structures . . . 54
3.14.3 Chaînes de caractères . . . 58
3.14.4 Énumérations . . . 60
3.14.5 Nouveaux types struct et enum . . . 62
3.15 Listes chaînées . . . 65
3.15.1 Définition . . . 65
3.15.2 Manipulation . . . 68
3.16 Conclusion . . . 70

4 Programmation Orientée Objet 71
4.1 Quelques notions de C++ . . . 71
4.1.1 Entrée / Sortie . . . 72
4.1.2 Références . . . 73
4.2 Les objets . . . 73

4.3 Les classes 74
4.4 Déclaration d'une classe et d'un objet 75
4.4.1 Déclaration d'une classe 75
4.4.2 Déclaration d'un objet 78
4.4.3 Encapsulation 79
4.4.4 Accès aux données et méthodes d'un objet 79
4.4.5 Utilisation des objets 80
4.4.6 Tableaux d'objets 80
4.4.7 Allocation dynamique d'objets 81
4.5 Héritage 81
4.5.1 Accessibilité des membres d'une classe 84
4.5.2 Polymorphisme 87
4.6 Méthodes virtuelles 88
4.6.1 Utilisation 88
4.6.2 Méthode virtuelle pure 90

5 Gestion de version 93
5.1 Intérêt 93
5.2 Subversion 94
5.2.1 Fonctionnement 94
5.2.2 Commandes de base 95
5.3 La base en pratique 96
5.3.1 Serveur 96
5.3.2 Client 99
5.3.3 Plus loin dans la pratique 103
5.3.4 Clients graphiques 104

Chapitre 1

Introduction

Dans ce cours, Méthodes et Outils pour la Programmation, nous allons aborder différents moyens et outils afin de résoudre un problème via la programmation logicielle. Ce cours ne demande aucun prérequis si ce n'est quelques notions d'Anglais.

Si vous n'avez aucune notion d'Anglais je vous conseille vivement de visiter le site `http://www.wordreference.com/` *pour connaître la définition des mots anglais de ce cours.*

1.1 Problème

Un problème en informatique est constitué de données sous une certaine forme et d'une question portant sur ces données comme par exemple les problèmes suivants :

— 250gr de farine, 0.5L de lait, 2 oeufs, 2gr de sel.
  Comment faire une pâte à crêpes ?

— Une voiture accidentée.
  Est-elle réparable ?

— Un échiquier avec une partie entamée et $n \in \mathbb{N}$.

Pour chacun des problèmes cités précédemment il existe une ou plusieurs instances, c'est-à-dire solutions.

1.2 Algorithme

Une fois le problème défini, l'étape suivante consiste à le résoudre. Un algorithme est une méthode indiquant sans ambiguïté des actions mécaniques

à effectuer pour trouver la réponse à un problème.
Précisons certains termes de cette définition : *sans ambiguïté* exprime le fait que tout le monde comprend la méthode de la même façon. Pour répondre au problème de la pâte à crêpes, on n'autorisera donc pas d'instructions telles que "saler à votre convenance" car cela ne désigne pas la même quantité de sel suivant à qui on s'adresse. Le terme *mécanique* signifie qu'il ne fait pas appel à l'intelligence, à la réflexion.

1.3 Programme

Un programme désigne alors la traduction d'un algorithme dans un langage de programmation. On nomme instruction la traduction d'une action mécanique. Il existe de nombreux langages de programmation, le langage C sera utilisé dans un premier temps comme support de ce cours pour les chapitres à venir.
Avant de se lancer dans la programmation en tant que telle sur machine, ce qui implique l'apprentissage d'un langage et la configuration d'un environnement de développement, nous allons commencer par apprendre les bases d'une manière simple pour écrire des algorithmes, le pseudo-code.

Chapitre 2

Pseudo-Code

Ce chapitre est une introduction rapide au pseudo-code, syntaxe qui peut vous être utile lors de la résolution de problèmes. Toutes les notions abordées ici seront vu en profondeur dans les chapitres suivants dans l'apprentissage d'un langage de programmation.

Le pseudo-code est une façon de décrire un algorithme sans référence à un langage de programmation en particulier. De cette manière on ne se concentre que sur l'algorithme de résolution du problème en décrivant son fonctionnement dans un vocabulaire simple. Puis, une fois le pseudo-code écrit, on peut le traduire et le structurer dans un langage de programmation adapté.

2.1 Syntaxe

Il n'existe pas de standard pour l'écriture de pseudo-code, toutefois la majorité d'algorithmes écrits en pseudo-code utilisent tous le même formalisme.

2.2 Structure

Les éléments de base d'un pseudo-code sont le nom de l'algorithme, suivi des instructions de l'algorithme encadrées par les balises *Begin* et *End*. Par exemple, l'algorithme 1 est la représentation en pseudo-code du fameux programme *HelloWorld* utilisé comme base d'apprentissage de nombreux langages.

L'algorithme 1 est assez rudimentaire et ne possède pas de logique. Voyons maintenant comment écrire des algorithmes plus complexes qui demandent

Algorithm 1 Afficher "Hello World !"

```
Begin
    afficher "Hello World !"
End
```

des conditions, des boucles d'itérations et l'utilisation de données variables.

2.3 Exemple simple

Imaginons que nous voulions écrire un algorithme permettant de donner le maximum de deux valeurs entières. Les valeurs entières sont données en tant que valeurs d'entrée de l'algorithme, prenons les lettres a et b pour les représenter. L'algorithme 2 est une solution possible à notre problème.

Algorithm 2 getMaximum

Inputs : a, b as Integers
Output : Integer, valeur maximum entre a et b.

```
Begin
    result = 0

    if a > b then
        result = a
    else
        result = b
    end if

    return result
End
```

Malgré la simplicité de cet algorithme, il fait appel à quelques notions citées précédemment comme les conditions, et l'utilisation de variables. On remarque que les données en entrée sont définies en début de l'algorithme via le mot clé *Inputs*, de même, dans le cas où l'algorithme renvoie une valeur on peut utiliser le mot clé *Output* pour définir ce que l'algorithme retourne. On donne un nom pour chaque donnée d'entrée ainsi que son type. Dans notre exemple, a et b sont du même type, entier.

Variables Une variable est une zone de stockage pour un résultat, une valeur. Pour utiliser une variable il faut la déclarer, c'est-à-dire lui donner

un nom unique et une valeur. Dans notre exemple, *result* est une variable que l'on initialise (affecte) à la valeur 0. Le nom de la variable est important pour la compréhension de l'algorithme, il doit être explicite de façon à indiquer l'utilité de la variable. Ici *result* servira à stocker le résultat de l'algorithme, soit la valeur maximale entre a et b.

Conditions Déterminer la plus grande valeur entre deux entiers consiste à comparer leur valeur en utilisant les opérateurs de comparaison classiques, $>$, $<$, $>=$, $<=$, ou encore $==$ pour tester l'égalité. Si a est strictement plus grand que b, alors le résultat est a, sinon le résultat est b. Cette phrase est la traduction des instructions de *if* à *end if* (c.f. algorithme 2). Le mot clé *end if* marque la fin de l'exécution des instructions incluses dans la condition *if* ou *else*, cette dernière n'étant exécutée uniquement si la condition *if* n'est pas satisfaite.
Il est possible d'utiliser des conditions multiples avec les mots clés *&&* pour *et*, *//* pour *ou*, *!* pour la négation. L'exemple 3 montre plusieurs conditions.

Algorithm 3 Conditions multiples

```
1: if a > b && b > 0 then
2: end if
3: if !(a > b) then
4: end if
5: if a == 0 || b == 2 then
6: end if
7: if a > 0 || (a == 0 && b == 0) then
8: end if
```

Retour Pour finir on retourne le résultat en utilisant le mot clé *return* suivi d'une variable contenant une valeur (ou directement d'une valeur). Le mot clé *return* marque la fin de l'algorithme, aucune instruction après *return* ne sera exécutée. Ainsi l'algorithme 2 peut être simplifié comme le montre l'algorithme 4.
Un algorithme ne retourne pas systématiquement un résultat, par exemple l'algorithme 1 ne fait qu'afficher un message sans retour.

Ce que nous avons vu jusqu'à maintenant permet de résoudre de nombreux problèmes mais on peut rapidement être confronté à devoir répéter des instructions identiques plusieurs fois ou tant qu'une condition est satisfaite. Ce genre de traitement est réalisable en utilisant des boucles d'itérations.

Algorithm 4 getMaximum

Inputs : a, b as Integers
Output : Integer, valeur maximum entre a et b.

```
Begin
    if a > b then
        return a
    end if

    return b
End
```

2.4 Boucles

On appelle boucle d'itérations le mécanisme permettant de répéter une plusieurs instructions. Il existe plusieurs mot clés afin d'effectuer des boucles, en fonction du besoin de l'algorithme, certains sont plus adaptés que d'autres.

Prenons pour exemple l'algorithme 5 qui reçoit en entrée un entier positif et qui affiche cet entier autant de fois que sa valeur. Première étape, utilisons une variable a pour stocker la valeur d'entrée, soit la valeur de x. Cette nouvelle variable va servir de compteur, tant que sa valeur est supérieur à 0, nous allons afficher x et décrémenter a. Le mot clé *while* permet de réaliser ce traitement, toutes les instructions comprises entre *while* et *end while* sont exécutées tant que la condition est satisfaite.

Algorithm 5 printManyTimes

Inputs : x as Integer

```
Begin
    a = x
    while  a > 0  do
        afficher x
        a = a - 1
    end while
    afficher "fin"
End
```

Le meilleur moyen pour comprendre un algorithme est de le tester en situation. Faisons l'exercice en prenant 2 comme valeur d'entrée.

$a = x$ soit $a = 2$

itération 1 $a = 2$, la condition $a > 0$ est satisfaite, on affiche la valeur de x, et on décrémente a, soit $a = 1$. On atteint l'instruction *end while*, comme nous sommes dans une boucle, on retourne à l'instruction *while* pour vérifier de nouveau la condition $a > 0$.

itération 2 $a = 1$, la condition $a > 0$ est satisfaite, on affiche la valeur de x, et on décrémente a, soit $a = 0$.

itération 3 $a = 0$, la condition $a > 0$ n'est pas satisfaite, on n'exécute pas les instructions du bloc *while*. On passe aux instructions après *end while*, soit afficher le message *fin*.

Il existe d'autres moyens pour effectuer des boucles, comme par exemple l'instruction *for*, à la syntaxe particulière. La syntaxe se compose en trois parties séparées par un point virgule. La première partie représente l'initialisation, la suivante la condition et enfin le pas de l'itération. L'algorithme 6 reprend l'algorithme 5 en utilisant la boucle *for*.

Algorithm 6 printManyTimes-for

Inputs : x as Integer

```
1: Begin
2:     for a = x ; a > 0 ; a = a - 1 do
3:         afficher x
4:     end for
5:     afficher "fin"
6: End
```

Dans notre algorithme 6, l'instruction *for* peut se lire, *pour "a = x", tant que "a est strictement supérieur à 0", à pas de "a = a - 1" faire "afficher x"*. Si ce n'est l'utilisation d'instructions différents pour la boucle, les algorithmes 5 et 6 ont un comportement identique.

2.5 Fonctions

Une fonction est un ensemble d'instructions, tous les algorithmes présentés jusqu'à maintenant peuvent être considérés comme des fonctions.

On identifie une fonction par un nom, ses paramètres d'entrée si elle en a, et un type de valeur de retour si la fonction retourne une valeur. Lorsque l'on écrit un programme, il devient vite utile de découper le programme principal en fonctions (sous-programmes) pour gagner en clarté et lisibilité. De plus une fonction peut très bien servir à plusieurs programmes, voire à d'autres fonctions.

Prenons pour exemple un programme de jeu de plus ou moins. Le but du jeu est de choisir une valeur mystère, puis de faire deviner la valeur à l'utilisateur en lui demandant un chiffre et en lui indiquant si la valeur mystère est plus petite ou plus grande que son chiffre. Enfin le programme se termine lorsque l'utilisateur trouve la valeur mystère. L'algorithme 7 est une version simple du programme de plus ou moins.

Algorithm 7 Plus ou moins

```
Begin
    valmystere = 0
    valuser = 0

    afficher "Bienvenue au jeu du plus ou moins"

    afficher "Entrez la valeur mystère :"
    saisir valmystere

    while valmystere != valuser do
        afficher "Entrez un nombre :"
        saisir valuser
        if valuser > valmystere then
            afficher "c'est moins !"
        else
            if valuser < valmystere then
                afficher "c'est plus !"
            end if
        end if
    end while

    afficher "Bravo !"
End
```

On peut aisément découper un tel programme en fonctions utiles. Je décide de créer deux fonctions, une première permettant de retourner un entier saisie, 8, et une seconde retournant le résultat d'une comparaison de deux entiers, 9.

Algorithm 8 getInteger

Inputs : message as String
Output : Integer, valeur saisie.

```
Begin
    value = 0

    afficher message
    saisie value

    return value
End
```

Algorithm 9 cmpIntegers

Inputs : a, b as Integers
Output : Integer, résultat de la comparaison :
0 : a == b
-1 : a < b
1 : a > b

```
Begin
    if a > b then
        return 1
    else
        if a < b then
            return -1
        end if
    end if

    return 0
End
```

Grâce à ces nouvelles fonctions nous pouvons réécrire le programme 7 comme le montre le programme 10.

Algorithm 10 Plus ou moins - amélioré

```
Begin
    result = -1
    valmystere = 0
    valuser = 0

    afficher "Bienvenue au jeu du plus ou moins"

    valmystere = getInteger "Entrez la valeur mystère :"

    while result != 0 do
        valuser = getInteger "Entrez un nombre :"
        result = cmpIntegers valmystere, valuser
        if result == -1 then
            afficher "c'est moins !"
        else
            if result == 1 then
                afficher "c'est plus !"
            end if
        end if
    end while

    afficher "Bravo !"
End
```

Utiliser des fonctions est essentiel pour plusieurs choses. En terme de maintenance, si vous n'utilisez pas de fonctions vous allez vite répéter et dupliquer des instructions. Si vous faites une erreur vous devrez corriger l'erreur à plusieurs endroits, au risque d'en oublier. En utilisant des fonctions, vous regroupez les instructions ce qui, en cas d'erreur, ne fait une modification que dans la fonction, donc un seul endroit. En terme de lisibilité, utilisez des noms de fonctions explicites de sorte que lorsque vous parcourez un programme vous soyez capable de comprendre le but de l'appel à une fonction sans aller voir ce que fait réellement cette fonction.

2.6 Conclusion

Nous avons abordé dans ce chapitre plusieurs notions de base de la programmation, les variables, les conditions, les boucles et les fonctions. Abstraire le langage de programmation au bénéfice de l'algorithme peut faciliter sa résolution. Lorsque vous êtes bloqué en programmation, essayez de résoudre le problème via le pseudo-code. Passons désormais à la pratique sur ordinateur.

Chapitre 3

Programmation

Dans ce chapitre nous allons voir comment traduire des algorithmes en langage machine afin de les exécuter. Pour ce faire nous utiliserons le langage C.

3.1 Pourquoi le langage C ?

Il existe de nombreux langages de programmation et il n'existe pas de langage *ultime* qui surpasse tous les autres mais on peut toujours trouver le langage le plus adapté au besoin. Le C est un langage considéré par certains de bas niveau car il est proche du système et permet une gestion de la mémoire que beaucoup d'autres langages (comme le Java et Python) abstraient.

Néanmoins, c'est parce que le C est proche de la machine et permet la gestion de la mémoire que l'on se doit d'adopter une certaine rigueur dans la programmation pour éviter tout débordement. Bien employé il permet d'obtenir des programmes performants, optimisés, et portables avec un minimum de changements.

Pour programmer, dans l'absolu un simple éditeur de texte est suffisant, mais ce n'est pas confortable lorsque l'on a affaire à des projets de grande envergure où un éditeur de texte plus complet et muni d'outils peut-être très productif.

3.2 Environnement de développement

Par environnement de développement on entend une suite d'outils indispensables pour augmenter la productivité des programmeurs. Bien que ce cours n'aborde que les concepts de base de la programmation il est tout de même important d'être bien outillé et de prendre de bonnes habitudes. Nous allons donc configurer notre système pour pouvoir programmer en C et pour cela nous avons besoin d'un compilateur et d'un bon IDE (Integrated Development Environment).

3.2.1 Compilateur - GCC

Le code que nous allons écrire (fichier source) n'est pas directement compréhensible par la machine et doit être *traduit* en instructions binaires. La compilation consiste à traduire notre fichier source en langage machine, le fichier source devient un fichier objet contenant du code objet mais non exécutable. Pour créer un exécutable il faut faire une liaison entre les différents fichiers objets que le programme utilise.

Naturellement il existe plusieurs compilateurs, nous allons utiliser GCC qui, dans le logiciel libre, est l'un des plus utilisé. L'installation du compilateur diffère en fonction du système d'exploitation que vous utilisez.

Linux Ouvrez un terminal et tapez la commande *gcc*. Si vous voyez le message *no input files* s'afficher c'est que votre compilateur est installé et vous pouvez passer à la section suivante. Sinon utilisez le gestionnaire de paquet de votre distribution pour installer le paquet *gcc* (pour Fedora, un *sudo yum install gcc* devrait faire l'affaire).

Windows À moins que vous n'ayez déjà installé GCC, ce dernier n'est pas présent par défaut dans Windows. Allez à l'adresse `http://sourceforge.net/projects/mingw/files/Installer/`, téléchargez et installez *mingw-get-setup.exe* en suivant les instructions (choisissez c : comme répertoire de destination). Ensuite, pour que le compilateur soit accessible depuis une console, il faut ajouter le répertoire d'installation dans le *path*, voici la procédure pour Win7 :

- Appuyez sur les touches Windows et pause.
- Dans la fenêtre qui vient de s'ouvrir, cliquez sur "Paramètres système avancés" sur la gauche.

— Dans la nouvelle fenêtre, cliquez sur le bouton "Variables d'environnement" en bas.
— Nouvelle fenêtre, dans la partie basse "Variables système", faites défiler jusqu'à trouver le mot "Path" et cliquez deux fois dessus pour l'éditer.
— Ne supprimez rien, ajouter en fin un point virgule puis le répertoire "bin" d'installation de Mingw. Si vous avez installé Mingw dans c :, le chemin sera C :\MinGW\bin.
— Cliquer sur "OK" dans toutes les fenêtres ouvertes.

3.2.2 IDE - Eclipse

La fonctionnalité essentielle que doit proposer un éditeur pour écrire du code est la colorisation syntaxique. En reconnaissant la syntaxe du langage de programmation, l'éditeur colorie les mot-clés ce qui rend plus lisible et compréhensible le code.

Eclipse n'est pas l'IDE le plus léger qu'il soit mais il est l'un des plus complets en outillage intégré et il est libre. Eclipse peut être téléchargé à l'adresse `https://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/lunasr1`. De même que pour le compilateur, l'installation diffère en fonction du système d'exploitation que vous utilisez.

Linux Avant de télécharger une version depuis le site web de Eclipse, vérifiez s'il n'y a pas une version proposée dans votre gestionnaire de paquet (pour Fedora tapez *sudo yum search eclipse*). Si aucune version n'est proposée par votre gestionnaire de paquet :
— votre système possède Java 7, rendez vous à l'adresse indiquée dans le paragraphe précédent.
— votre système ne possède pas Java 7, rendez-vous à l'adresse `https://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/junosr2`.

Téléchargez la version correspondante à votre système d'exploitation sur la droite du site. Décompressez l'archive tar.gz où vous désirez, pour lancer Eclipse il suffit de cliquer deux fois sur le programme nommé *eclipse*.

Windows Rendez vous à l'adresse indiquée dans le paragraphe précédent, et téléchargez la version correspondante à votre système d'exploitation sur la droite du site. Décompressez l'archive zip où vous désirez, pour lancer Eclipse il suffit de cliquer deux fois sur le programme nommé *eclipse*.

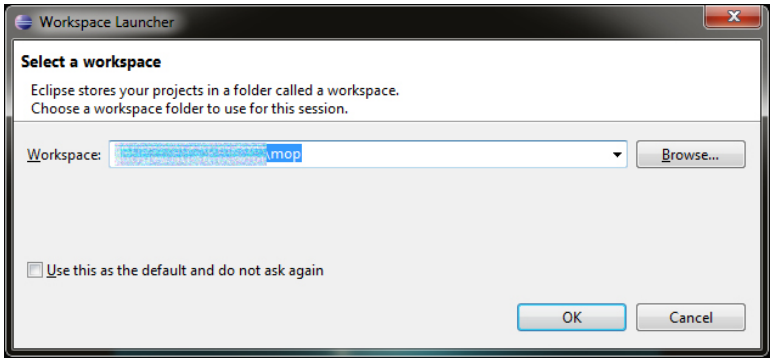


FIGURE 3.1 – Eclipse - workspace

Une fois l'installation terminée, lancez Eclipse. Une fenêtre s'ouvre et demande de choisir un répertoire *workspace*. Ce sera votre espace de travail, je vous conseille de créer un répertoire *mop* à l'emplacement que vous souhaitez et de choisir ce répertoire comme *workspace* (c.f. 3.1).

Vous êtes désormais sur la page d'accueil de votre workspace, fermez l'onglet *Welcome*, vous verrez alors votre espace de travail et ses différentes parties (c.f. 3.2). Vous êtes prêt pour écrire votre premier programme.

3.3 Premier Programme

Pour ce premier programme nous allons reprendre le classique "Hello World !", déjà vu en pseudo code.

3.3.1 Écriture, compilation et exécution

Dans Eclipse, faites un clic droit dans la partie gauche, *Project Explorer*, choisissez *New C Project*. Dans le champ *Project name*, donnez un nom à votre projet, par exemple, *HelloWorld*. Ne vous préoccupez pas de la suite, cliquez sur *Next* deux fois et enfin *Finish*. Vous avez désormais un project nommé *HelloWorld* dans votre espace de projets.

Ajoutons un fichier source à notre projet. Faites un clic droit sur le nom du projet puis sur *New* et *Source File*. Donnez un nom au fichier source, par exemple *helloworld.c* (évitez d'utiliser des espaces dans les noms de fichiers).

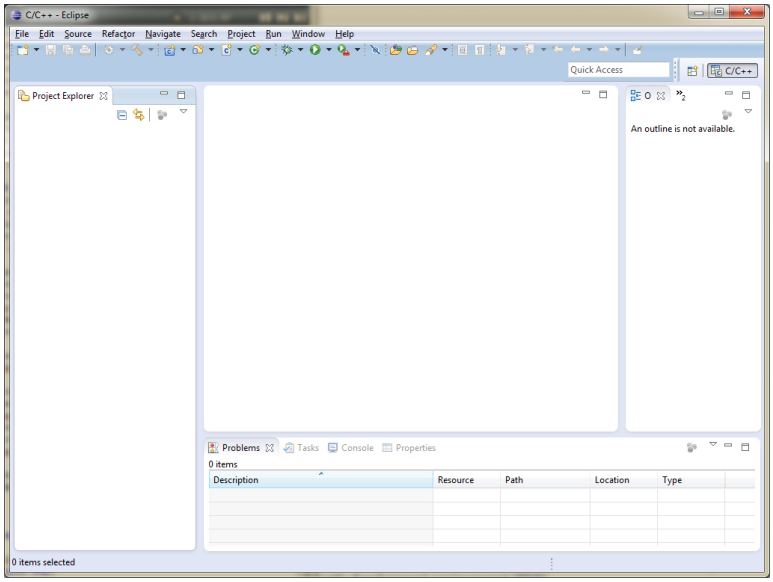


FIGURE 3.2 – Eclipse - fenêtre principale

Cliquez deux fois sur le nouveau fichier créé pour l'ouvrir dans la fenêtre centrale de l'éditeur. Recopiez le code 3.1.

```
1 #include <stdio.h>
2
3 int main( void )
4 {
5    printf( "Hello World !\n" );
6    return 0;
7 }
```

Listing 3.1 – Programme - Hello World !

Avant d'analyser le code, nous allons le compiler et l'exécuter. Ouvrez un terminal (console sous Windows), placez-vous dans le répertoire où se trouve votre fichier helloworld.c et tapez la commande suivante :

```
gcc helloworld.c -o helloworld
```

Si tout s'est bien passé vous devriez avoir un exécutable nommé *helloworld*. Exécutez le, normalement le message *Hello World !* s'affiche dans votre terminal, analysons le code pour en comprendre le fonctionnement.

3.3.2 Analyse

Tout programme informatique possède un point d'entrée, c'est ici que tout commence et tout se termine. Le point d'entrée d'un programme est une fonction appelée *main*, c'est cette fonction qui sera exécutée en premier lors du lancement du programme, et donc toutes les instructions comprises entre les accolades et . Les paramètres d'entrée sont donnés entre parenthèses après le nom de la fonction, ici le mot clé *void* veut dire qu'il n'y a pas de paramètres d'entrée. Le type de retour de la fonction est donné avant le nom de la fonction, ici la fonction main retourne une valeur de type entier. Dans la cas où la fonction ne retourne pas de valeur, on doit employer le mot clé *void*. Remarquez que chaque instruction se termine par un point virgule. Nous verrons plus loin dans ce cours comment les fonctions sont construites et utilisées.
La première instruction exécutée est :

```
printf( "Hello World!\n" );
```

Qu'est-ce que *printf* ? Ce n'est pas un mot clé, c'est une fonction standard disponible en C, son but est d'envoyer sur la sortie standard ce qu'on lui donne en argument. Dans notre exemple l'argument est une chaîne de caractères mais nous verrons plus tard que *printf* permet d'afficher en sortie des formats bien plus complexes.

D'où provient *printf* ? Cette fonction est connue de votre programme grâce à la première ligne de votre fichier .c :

```
#include <stdio.h>
```

Le fichier *stdio.h* contient les définitions de nombreuses fonctions standards traitant les entrées et sorties. Remarquez que le fichier possède l'extension .h pour *header*. Les fichiers *header* ne contiennent que des définitions, fonctions, variables, constantes, etc. Par la directive *#include* on permet au compilateur, lors de la compilation du fichier .c, de connaître la définition de la fonction *printf*. Supprimez cette ligne de votre code et compilez à nouveau votre fichier, le compilateur devrait sortir en erreur et vous informer qu'il ne trouve pas la définition de *printf*.

Vous venez d'écrire et de compiler votre premier programme, après une analyse sommaire de celui-ci nous allons voir désormais en détail les bases de la programmation C.

3.4 Variables

Une variable est pour l'ordinateur une zone mémoire réservée et modifiable par le programme. Pour définir une variable on lui donne un nom et un type, en fonction de son type, la variable prendra une place plus ou moins importante en mémoire.

3.4.1 Noms de variable

Les noms de variables peuvent être aussi long que l'on désire, toutefois un nom court et explicite et bien plus efficace qu'une longue description. Les noms doivent respecter les règles suivantes :

— le nom doit commencer par une lettre (capitale ou minuscule) ou un _.
— le nom peut comporter des lettres, chiffres et le caractère _ mais pas d'espaces.
— le nom ne peut pas être un mot clé réservé du langage (if, else, while, etc.)

Le nom ne suffit pas à définir la variable, il faut obligatoirement lui associer un type, qui précède le nom.

3.4.2 Types de variables

On peut identifier cinq types :

— Entiers
— Réels
— Caractères
— Chaînes de caractères
— Vide

Les types entiers peuvent être signés (*signed*) ou non signés (*unsigned*). Par défaut, si ce n'est pas spécifié, le type est considéré comme signé.

Les entiers

Un entier sert à stocker un nombre entier, on le représente par les mots clés *int*, *char* ou *short*. Sa taille en mémoire varie en fonction du type d'entier, le tableau ci-dessous donne la taille occupée en mémoire ainsi que les bornes des valeurs possible.

Déclaration	Taille (octet)	Plage de valeurs
unsigned char	1	[0, 255]
char	1	[-128, 127]
unsigned short	2	[0, 65 535]
short	2	[-32 768, 32 767]
unsigned int	4 [1]	[0, 4 294 967 295]
int	4	[-2 147 483 648, 2 147 483 647]
unsigned long	8	[0, 18 446 744 073 509 551 615]
long	8	[-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]

Les réels

Un réel sert à stocker un nombre réel, on le représente par les mots clés *float* ou *double*.

Déclaration	Taille (octet)
float	4
double	8
long double	12

Les caractères

Les caractères sont stockés dans le type entier *char*. Pour indiquer qu'on souhaite stocker un caractère, on le met entre quote, par exemple :

```
char valeur = 'a';
```

Les chaînes de caractères

Il n'existe pas de type spécifique aux chaînes de caractères en C, cependant on peut voir les chaînes de caractères comme un tableau de caractères déclaré comme suit :

```
char msg[] = "message";
```

Vide

Le type vide est représenter par *void*.

3.5 Entrée / Sortie

Il est possible d'afficher et de saisir des informations via l'entrée et la sortie standard de l'ordinateur. Par entrée et sortie standard il faut comprendre

1. Pour les processeurs 64 bits, le type int est codé sur 8 octets et non 4.

le clavier et l'écran (terminal d'exécution). Pour cela on utilise essentiellement deux fonctions, *scanf* pour saisir des valeurs au clavier, et *printf* pour afficher des valeurs sur le terminal.

Les deux fonctions fonctionnent suivant le même principe, on donne une chaîne de caractères qui contient un ou plusieurs format (introduit par le caractère %), puis la ou les variables concernées.

Entrée

Pour saisir une valeur il faut tout d'abord déclarer une variable qui servira à stocker la valeur saisie, ensuite appeler la fonction *scanf* en lui donnant le type que l'on attend et la variable. Comme par exemple, pour saisir un entier signé :

```
int value = 0;
scanf( "%d", &value );
```

Remarquez que dans l'appel de la fonction la variable est précédent du symbol &, nous reviendrons sur cette spécificité plus tard dans le cours. Ici le format utilisé est %d pour saisir une valeur de type entière.

En fonction du type de variable, et donc du type de valeur attendue à la saisie, il faut donner à scanf le bon format, la table 3.1 donne une liste non exhaustive des formats acceptés par *scanf*.

Sortie

Nous avons déjà vu la fonction *printf* pour afficher du texte seulement, nous allons voir maintenant comment afficher le contenu d'une variable.

printf, tout comme *scanf*, utilise un format afin de spécifier le type de la valeur de sortie. Par exemple, si vous souhaitez afficher un entier sous sa forme décimale vous devez utiliser le formateur %d, mais vous pouvez vouloir afficher l'entier sous forme hexadécimale et ceci est possible avec *printf*. De plus le nombre de variables que l'on peut afficher n'est pas limité, c'est-à-dire que dans un appel de *printf* vous pouvez afficher autant de variables que vous utilisez de formats, par exemple :

```
printf( "texte : %s\n decimal : %d\n hex : %02X\n float : %f", "bonjour",
15, 255, 1.5 );
```

Type	Format
int	%d
long	%ld
float	%f
double	%lf
char	%c
string	%s
hexadécimal	%x

Table 3.1 – Formats de printf et scanf

Cet appel va afficher sur le terminal les lignes suivantes :

```
texte : bonjour
decimal : 15
hex : FF
float : 1.5
```

Remarquez que pour chaque format donné dans le premier argument, qui est une chaîne de caractères, il y a une variable associée par la suite. Ainsi nous avons les associations suivantes :

— %s - "bonjour"
— %d - 15
— %02X - 255
— %f - 1.5

La table 3.1 donne une liste non exhaustive des formats acceptés par *printf*.

3.6 Conditions

Nous allons voir comment demander à l'ordinateur de n'effectuer des instructions que si certaines conditions sont remplies.

Les instructions conditionnelles permettent à un bloc d'instructions (un suite d'instructions encadrées par des accolades et) de n'être exécuté que si une condition prédéterminée est réalisée. Dans le cas contraire, les instructions sont ignorées et la séquence d'exécution continue à partir de l'instruction qui suit immédiatement la fin du bloc. Prenons pour exemple le

programme 3.2, en fonction de la valeur donnée à n, le programme exécutera les instructions du bloc *if{ ... }* sinon celles du bloc *else{ ... }*.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int n = 0;
6
7     printf( "Entrez un nombre :\n" );
8     scanf( "%d", &n );
9
10    if( 10 < n )
11    {
12       printf( "n est plus grand que 10\n" );
13    }
14    else
15    {
16       printf( "n est plus petit que 10 (ou egal)\n" );
17    }
18
19    return 0;
20 }
```

Listing 3.2 – Programme - Exemple conditions

Les conditions sont des expressions booléennes, elles sont dites satisfaites si leur évaluation est vraie.

Expressions booléennes

Il existe plusieurs opérateurs pour réaliser des expressions booléennes, les opérateurs binaires *&&* pour *et*, // pour *ou*, l'opérateur unaire de négation, *!* et les opérateurs binaires de comparaison >, <, ==, *!*=, >=, <=. Tous ces opérateurs ont un résultat booléen, on peut ainsi former de nombreuses expressions booléennes.

Instruction if

Avec des si on refait le monde, et on peut aussi refaire tout un programme. C'est l'instruction de base des expressions conditionnelles, on l'utilise comme le montre l'exemple 3.3 où *condition* est une expression booléenne et *liste 1* et *liste 2* des suites d'instructions.

```
1  if( condition )
2  {
3     liste 1
```

```
4 }
5 else
6 {
7    liste 2
8 }
```

Listing 3.3 – Instruction if

Lors de l'exécution, si la condition est satisfaite, les instructions de *liste 1* seront exécutées et celles de *liste 2* ignorées. Si la condition est fausse alors les instruction de *liste 2* seront exécutées et celles de *liste 1* ignorées.

Lorsque l'on se trouve dans le bloc d'instructions de *if* ou de *else*, on peut imbriquer de nouvelles conditions comme le montre le programme 3.4.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int n = 0;
6
7     printf( "Entrez un nombre :\n" );
8     scanf( "%d", &n );
9
10    if( 10 < n )
11    {
12       printf( "n est plus grand que 10\n" );
13    }
14    else
15    {
16       if( 10 == n )
17       {
18          printf( "n est egal a 10\n" );
19       }
20       else
21       {
22          printf( "n est plus petit que 10\n" );
23       }
24    }
25
26    return 0;
27 }
```

Listing 3.4 – Programme - Exemple conditions imbriquées

De plus, *else* n'est pas obligatoire, comme le montre l'exemple 3.5. Si le nombre entré au clavier est pair, le message "le nombre est pair" s'affiche puis le programme quitte grâce à l'instruction *return*. Si le nombre est impair, les instructions du bloc *if* ne sont pas exécutées, on saute directement à l'affichage du message "le nombre est impair" et le programme quitte.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int n = 0;
6
7     printf( "Entrez un nombre :\n" );
8     scanf( "%d", &n );
9
10    if( 0 == (n%2) )
11    {
12       printf( "le nombre est pair\n" );
13       return 0;
14    }
15
16    printf( "le nombre est impair\n" );
17
18    return 0;
19 }
```

Listing 3.5 – Programme - Exemple if seul

Toutefois les instructions *if* et *else* ont leurs limites, en effet l'imbrication peut s'avérer illisible ou du moins peu élégant (c.f 3.6). Dans ce cas l'instruction *switch* prend tout son sens.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int n = 0;
6
7     printf( "Entrez une valeur de carte :\n" );
8     scanf( "%d", &n );
9
10    if( 10 > n && 1 < n)
11    {
12       printf( "%d\n", n );
13    }
14    else
15    {
```

```
16        if( 10 == n )
17        {
18           printf( "valet\n" );
19        }
20        else if( 11 == n )
21        {
22           printf( "dame\n" );
23        }
24        else if( 12 == n )
25        {
26           printf( "roi\n" );
27        }
28        else if( 1 == n )
29        {
30           printf( "as\n" );
31        }
32        else
33        {
34           printf( "la valeur n'est pas une carte valide\n" );
35        }
36     }
37
38     return 0;
39  }
```

Listing 3.6 – Programme - conditions imbriquées sans switch

Instruction switch

Lorsque l'on doit agir en fonction de la valeur d'une variable, il existe une solution plus élégante que l'imbrication d'instructions *if* et *else*, l'instruction *switch*.

Le programme 3.7 est la version *switch* du programme 3.6. *switch* reçoit un paramètre qui va être évaluer, en fonction de sa valeur, le programme va exécuter les instructions associées au *case* correspond à la valeur évaluée. Si aucun cas ne correspond à la valeur évaluée, le programme exécutera les instructions correspondant à *default*. Remarquez que chaque *case* se termine par un *break*, instruction qui va faire sortir le programme du bloc *switch*. Afin de comprendre l'utilité d'ajouter des *break*, reprenez le programme 3.7, supprimez tous les *break*, et testez le en entrant une valeur de 10. Vous devriez voir sur votre terminal tous les messages car lorsque le programme a exécuté l'instruction *printf("valet\n");*, n'ayant pas de *break* après, il a exécuté les instructions du *case* suivant jusqu'à sortir du bloc *switch*.

```c
#include <stdio.h>

int main( void )
{
   int n = 0;

   printf( "Entrez une valeur de carte :\n" );
   scanf( "%d", &n );

   if( 10 > n && 1 < n)
   {
      printf( "%d\n", n );
      return 0;
   }

   switch( n )
   {
      case 10:
         printf( "valet\n" );
         break;

      case 11:
         printf( "dame\n" );
         break;

      case 12:
         printf( "roi\n" );
         break;

      case 1:
         printf( "as\n" );
         break;

      default:
         printf( "la valeur n'est pas une carte valide\n" );
         break;
   }

   return 0;
}
```

Listing 3.7 – Programme - switch

3.7 Commentaires

Nous avons vu assez d'éléments de programmation pour que les programmes deviennent complexes en terme de compréhension. Pour faciliter le travail de lecture et de compréhension, il est possible d'ajouter des commentaires dans le code mais attention à ne pas trop en mettre ce qui a pour effet de rendre le code illisible.

Pour ajouter un commentaire il suffit de l'écrire entre les balises /* et */ sur une ou plusieurs lignes.

```
/* voici un commentaire */
```

Les programmes 3.9 et 3.8 sont un exemple et contre exemple de code commenté.

```
1 #include <stdio.h>
2
3 int main( void )
4 {
5    /* affiche le message "Hello World !"
6     * sur la sortie standard. */
7    printf( "Hello World !\n" );
8    return 0;
9 }
```

Listing 3.8 – Exemple commentaire inutile

```
1 #include <stdio.h>
2
3 int main( void )
4 {
5    int n = 0, fact = 1;
6
7    printf( "Entrez une valeur :\n" );
8    scanf( "%d", &n );
9
10   /* calcul de la factorielle de n */
11   for( i = 1; i <= n; i++ )
12   {
13      fact = fact * i;
14   }
15
16   printf( "resultat : %d\n", fact );
17
18   return 0;
19 }
```

Listing 3.9 – Exemple commentaire utile

3.8 Boucles

Nous allons voir comment demander à l'ordinateur d'exécuter plusieurs fois une ou plusieurs instructions sans avoir besoin de réécrire ces instructions. Les boucles sont utiles pour répéter des traitements tant qu'une condition est satisfaite. Pour ce faire nous allons voir les instructions *while*, *do ... while* et *for*.

3.8.1 While

L'instruction *while* prend en paramètre une condition, comme pour l'instruction *if*. Cependant, pour *while*, tant que la condition est satisfaite les instructions du bloc seront exécutées. Par exemple le programme 3.10 demande à l'utilisateur d'entrer une valeur et l'affiche autant de fois que sa valeur. Comment feriez-vous ce programme sans utiliser de boucle ? C'est impossible puisqu'on ne sait pas par avance combien de fois il faudra répéter les instructions.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int n = 0, a = 0;
6
7     printf( "Entrez une valeur :\n" );
8     scanf( "%d", &n );
9
10    a = n;
11
12    while( 0 < a )
13    {
14       pintf( "%d\n", n );
15       a = a - 1;
16    }
17
18    return 0;
19 }
```

Listing 3.10 – Exemple de boucle while

Si nous analysons le programme 3.10, une fois n valorisée par l'utilisateur, a prend la valeur de n car celui-ci va servir de compteur. En effet on remarque dans la boucle que la dernière instruction du bloc est $a = a$ - *1* ce qui a pour effet de décrémenter a. La condition de répétition étant *0* $< a$ lorsque a

vaudra *0* le programme arrêtera de répéter les instructions du bloc *while* et passera aux instructions suivantes, dans notre exemple *return 0*.

Attention aux conditions utilisées dans les boucles, il se peut que le programme ne s'arrête jamais si par mégarde vous avez crée une boucle infinie. Il est très facile d'illustrer ce cas, reprenez le programme 3.10 et supprimez (ou commentez) la ligne $a = a$ - *1*. La valeur de a est constante et la condition $0 < a$ sera toujours satisfaite, le programme tournera en boucle sur l'affichage de la valeur de n. Ceci est vrai pour toutes les boucles.

3.8.2 Do ... While

Cette instruction est une variante de *while* dans le sens où les instructions du bloc sont exécutées une première fois avant la vérification de la condition, et la répétition des instructions du bloc tant que la condition est satisfaite. Le programme 3.11 est la version *do ... while* du programme 3.10. Notez que cette fois ci il y a un point virgule à la fin de *while*.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int n = 0, a = 0;
6
7     printf( "Entrez une valeur :\n" );
8     scanf( "%d", &n );
9
10    a = n;
11
12    if( 0 < a )
13    {
14       do
15       {
16          pintf( "%d\n", n );
17          a = a - 1;
18       }
19       while( 0 < a );
20    }
21    return 0;
22 }
```

Listing 3.11 – Exemple de boucle do ... while

3.8.3 For

L'instruction *for* possède une syntaxe plus évoluée que *while*, on ne précise pas seulement une condition, mais aussi un compteur et son incrément. Le programme 3.12 est la version for des programmes 3.10 et 3.11.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int n = 0, a = 0;
6
7     printf( "Entrez une valeur :\n" );
8     scanf( "%d", &n );
9
10    for( a = n; 0 < a; a-- )
11    {
12       printf( "%d\n", n );
13    }
14
15    return 0;
16 }
```

Listing 3.12 – Exemple de boucle for

Il y a donc trois parties dans la syntaxe de l'instruction *for*, les parties étant séparées par un point virgule. La première partie sert d'initialisation de variables, d'où $a = n$. La seconde partie est la condition de sortie de la boucle, on retrouve donc $0 < a$. Enfin la troisième partie est l'incrément ou le pas de l'itération, ici a est décrémenté pour que la condition $0 < a$ soit un jour satisfaite.

Pour décrémenter a j'ai utilisé une syntaxe particulière qui est a--. Cette syntaxe est équivalente à $a = a - 1$, on peut écrire également a++ pour incrémenter.

3.9 Tableaux

Jusqu'à maintenant nous avons manipulé des types simples, nous allons voir dans cette partie comment manipuler des tableaux permettant de stocker une ou plusieurs valeurs de même type.

FIGURE 3.3 – Déclaration tableau en mémoire

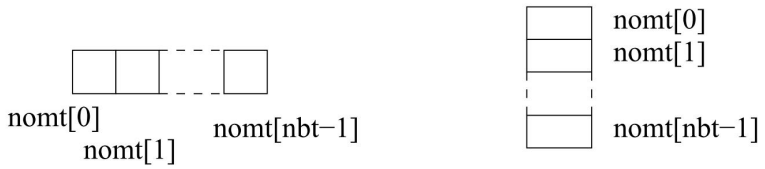


FIGURE 3.4 – Accès tableau en mémoire

3.9.1 Déclarer un tableau

La déclaration d'un tableau est de la forme :

```
typet nomt [nbr];
```

Où *typet* est le type des éléments qui seront stockés dans le tableau, *nomt* est le nom donné au tableau et *nbr* est le nombre d'éléments. Par exemple :

```
int tab[5]; /* tableau de 5 entiers */
float tmoy[2]; /* tableau de 5 réels */
```

À la déclaration l'ordinateur va réserver *nbr* cases consécutives en mémoire de la taille de *typet*. En déclarant un tableau de cinq entiers, l'ordinateur va réserver un espace mémoire de vingt octets (5 fois la taille d'un entier). La figure 3.3 montre la représentation en mémoire d'un tableau.

On accède à un élément du tableau par son indice, le premier élément est noté *nomt[0]*, le suivant *nomt[1]*, et ainsi de suite. Le nombre donné entre crochet est donc l'indice d'un élément du tableau variant de 0 à nbr - 1. La figure 3.4 représente l'accès en mémoire d'un élément du tableau.

3.9.2 Affectation et manipulation des éléments d'un tableau

En accédant à un élément du tableau via son indice on va pouvoir écrire une valeur dans son emplacement en mémoire. Par exemple pour affecter une valeur au deuxième élément du tableau on écrit (l'indice 0 étant le premier élément) :

```
nomt[1] = valeur;
```

Le programme 3.13 manipule un tableau de trois entiers, en fin d'exécution le tableau possèdera les valeurs, 1, 2 et 4.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int tab[3];
6
7     tab[0] = 1;
8     tab[1] = 2 * tab[0];
9     tab[2] = 2 * tab[1];
10
11    return 0;
12 }
```

Listing 3.13 – Tableaux - initialisation

Les éléments d'un tableau étant accessible par indice, on peut facilement utiliser une boucle for pour traiter tous ses éléments comme le montre le programme 3.14. On parcours le tableau en utilisant une boucle *for* allant de 0 à 2 et on affiche chaque élément.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int tab[3], i;
6
7     tab[0] = 1;
8     tab[1] = 2 * tab[0];
9     tab[2] = 2 * tab[1];
10
11    for( i = 0; i < 3; i++ )
12    {
13       printf( "%d\n", tab[i] );
14    }
15
```

```
16     return 0;
17 }
```

Listing 3.14 – Tableaux - parcours

On peut également initialiser un tableau lors de sa déclaration, le programme 3.15 présente trois méthodes courantes lors de la déclaration de tableaux.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     /* tous les element valent 0 */
6     int tab1[3] = { 0, };
7     /* les cinq elements sont initialises
8      * a 1 2 3 4 et 5. */
9     int tab2[5] = { 1, 2, 3, 4, 5 };
10    /* la taille du tableau sera de 2 */
11    int tab3[]  = { 10, 20 };
12
13    return 0;
14 }
```

Listing 3.15 – Tableaux - initialisation à la déclaration

3.9.3 Deux dimensions

On l'utilise rarement mais les tableaux peuvent aussi être de deux dimensions, et vus comme une matrice.

```
int matrix[2][5];
```

matrix est un tableau d'entiers à deux dimensions de 2 lignes et 5 colonnes. Le programme 3.16 donne un exemple d'utilisation de tableaux à deux dimensions.

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5     int matrix[2][5] = { { 1, 2, 3, 4, 5 },
6                          { 6, 7, 8, 9, 10 } };
7     int l, c;
8
9     for( l = 0; l < 2; l++ )
```

```
10    {
11       for( c = 0; c < 5; c++ )
12       {
13          printf( "%d\n", matrix[l][c] );
14       }
15    }
16
17    return 0;
18 }
```

Listing 3.16 – Tableaux à deux dimensions

3.10 Fonctions

Terminé la programmation où tout le code se situe dans la fonction *main* du programme. Nous allons voir dans cette partie comment écrire des fonctions et ainsi rendre nos programmes plus lisibles et modulaires.

Rendre le code lisible est important pour la maintenance de celui-ci, mais aussi pour ses collègues lorsque l'on travaille en équipe et que l'on n'est pas le seul intervenant dans un projet. La modularité est essentielle en programmation, bien découper le programme en fonctionnalités (lire fonctions) va d'une part alléger le programme principal et d'autre part rendre votre code (en particulier les fonctions) **réutilisable**.

Pour commencer nous allons voir comment déclarer des fonctions, puis comment les utiliser, et enfin comment les rendre publiques (accessibles par d'autres programmes) ou privées (uniquement accessibles par le programme interne).

3.10.1 Déclaration

Pour commencer une fonction porte un nom, qui comme pour les variables, doit être assez explicite pour comprendre ce que fait la fonction. Ensuite on doit trouver la liste des paramètres d'entrées (arguments) si elle en possède, et le type de retour de la fonction. On appelle cet ensemble le prototype (ou la signature) de la fonction. Par exemple :

```
void printHelloWord( void );
```

Est une fonction qui ne prend aucun paramètre et ne retourne aucune valeur, d'où l'emploi du type *void* en entrée et en sortie. Alors que :

```
float calculMoyenne( float somme, int n );
```

Prend en entrée un réel et un entier, et retourne une valeur réelle.

3.10.2 Appel

Lorsque l'on veut utiliser une fonction on l'appel via son nom et en précisant les arguments (ou non). Le programme 3.17 montre l'appel aux deux fonctions introduites précédemment.

```
1  #include <stdio.h>
2
3
4  static void printHelloWorld( void )
5  {
6     printf( "Hello World !\n" );
7  }
8
9  static float calculMoyenne( float somme, int n )
10 {
11    if( 0 >= n ) { return 0; }
12
13    return somme / n;
14 }
15
16 int main( void )
17 {
18    printHelloWorld();
19
20    printf( "Moyenne : %f\n", calculMoyenne( 15, 2 ) );
21
22    return 0;
23 }
```

Listing 3.17 – Appel de fonctions

Les fonctions sont définies avant la fonction *main* car pour qu'une fonction soit appelée, elle doit être connue. Il est possible de déclarer le prototype des fonctions en tête de fichier, puis le *main* puis le corps des fonctions comme le montre le programme 3.18.

```
1  #include <stdio.h>
2
3
4  static void printHelloWorld( void );
5  static float calculMoyenne( float somme, int n );
6
7
```

```
 8 int main( void )
 9 {
10    float somme = 15;
11
12    printHelloWorld();
13
14    printf( "Moyenne : %f\n", calculMoyenne( somme, 2 ) );
15
16    return 0;
17 }
18
19 static void printHelloWorld( void )
20 {
21    printf( "Hello World !\n" );
22 }
23
24 static float calculMoyenne( float somme, int n )
25 {
26    if( 0 >= n ) { return 0; }
27
28    return somme / n;
29 }
```

Listing 3.18 – Appel de fonctions

La fonction *main* est le point d'entrée du programme, lorsque l'ordinateur rencontre un appel à une fonction, il va *sauter* à la première instruction donnée par la fonction, exécuter les instructions qui s'y trouve et revenir au *main* lorsque la fonction se termine.

Si la fonction attend des arguments, ceux-ci doivent être données entre parenthèses et doivent correspondre aux types attendus. La fonction *calculMoyenne* attend un réel et un entier, on doit donc l'appeler avec des variables (ou directement des valeurs) de même type. Il en va de même pour le retour de la fonction, son contenu doit être stocké dans une variable de même type.

3.10.3 Portée des variables

La question ne se posait pas vraiment jusqu'à maintenant puisque toute variable était déclarée dans *main*. Regardez dans le programme 3.18, la variable *somme* est déclarée dans *main* et le même nom de variable est utilisé dans la fonction *calculMoyenne*, ce sont deux variables différentes. Une variable n'existe que la où elle est déclarée, sa zone d'effet et délimitée par les accolades { et }, on parle alors de variable locale. Ainsi la variable *somme*

de *main* est différente de la variable *somme* de la fonction *calculMoyenne*. Et donc toute modification de valeur de *somme* dans *calculMoyenne* ne sera pas visible dans *main*.

On peut étendre la portée d'une variable en la rendant globale, pour ce faire il faut la déclarer en dehors d'une fonction, elle sera accessible en lecture et écriture par n'importe quelle fonction du fichier source. Le programme 3.19 donne un exemple d'utilisation de variable globale au fichier source, son exécution donne la sortie :

```
0
1
6
```

```
1 #include <stdio.h>
2
3
4 static int valeur = 0;
5
6
7 static void incrementeValeur( void )
8 {
9    valeur++;
10 }
11
12 static void printValeur( void )
13 {
14    printf( "%d\n", valeur );
15 }
16
17
18 int main( void )
19 {
20    printValeur();
21    incrementeValeur();
22    printValeur();
23    valeur = 6;
24    printValeur();
25 }
```

Listing 3.19 – Variable globale

J'ai utilisé le mot clé *static* devant la déclaration des fonctions et de la variable globale. En déclarant une fonction, ou une variable, en tant que *static* dans un fichier source, on limite sa disponibilité dans le fichier source seul. On peut dire que les fonctions, ou variables, sont privées et donc non accessibles

aux autres fichiers sources (contrairement aux déclarations publiques). Ceci est très important pour bien découper votre programme en fonctionnalités et le rendre modulaire.

3.10.4 Modularité

Nous allons voir un exemple de programme avec plusieurs fichiers sources afin de comprendre l'intérêt de la modularité.

Téléchargez le fichier *module.zip* disponible sur la page d'accueil du cours. Décompressez l'archive, ouvrez le fichier *main.c*, remarquez qu'il n'y a pas de déclarations de fonctions, les fonctions utilisées sont déclarées et implémentées dans des fichiers différents. Le nom des fichiers permet de catégoriser les fonctions, par exemple *mathutils.c* contiendra des fonctions mathématiques utiles, quand à *trace.c* des fonctions pour afficher des traces d'exécution. Vous pouvez compiler le programme en utilisant la commande :

```
gcc *.c -o test
```

A chaque fichier source est associé un fichier d'en-tête (.h) du même nom (ce n'est pas une obligation d'utiliser le même nom mais c'est plus pratique) contenant la déclaration, pas l'implémentation, des fonctions qui seront publiques, donc visibles et disponibles à d'autres fichiers sources. L'implémentation est toujours dans le fichier source. En procédant ainsi on crée des modules fonctionnels unitaires qui peuvent éventuellement être compilés indépendamment du programme principal sous forme de librairies. C'est grâce aux fichiers d'en-tête que le programme principal connait les fonctions appelées dans *main*. Regardez dans *main.c* au début du fichier j'ai ajouté les lignes :

```
#include "trace.h"
#include "mathutils.h"
```

Je n'ai pas mis le nom du fichier entre chevrons $<$ et $>$ mais entre guillemets car les fichiers sont au même endroit que *main.c*. En utilisant des guillemets, le compilateur cherchera le fichier d'en-tête au même emplacement que le fichier source compilé.

Regardez le fichier *trace.c*, il contient une fonction déclarée en *static* car elle n'est pas destinée à être utilisée par d'autres modules mais seulement en interne dans le fichier *trace.c*, on limite donc sa portée au fichier seul. De ce

fait dans le fichier d'en-tête *trace.h* seul le prototype de la fonction publique y est déclaré. C'est là la base de modularité, se poser la question de ce qui doit être rendu publique ou privé.

Vous aurez sans doute remarqué que j'ai mis des commentaires avant la déclaration de chaque fonction, dans le fichier d'en-tête pour les fonction publiques et dans le fichier source pour les fonctions statiques. J'ai utilisé un formalisme propre à *Doxygen*, un générateur de documentation.

3.11 Doxygen

Doxygen est un générateur de documentation à partir de codes sources commentés dans une syntaxe qui lui est propre. Le principal intérêt de Doxygen est d'utiliser le code comme documentation ce qui a pour avantage de les synchroniser (pour peu que le développeur se donne la peine de commenter ou de mettre à jour les commentaires).

Les commentaires Doxygen vont surtout être utilisés pour décrire les fonctions publiques afin de pouvoir produire un document décrivant l'API (Application Programming Interface) de notre programme. Ce genre de documentation est fort utile lorsque vous développez une bibliothèque de fonctions.

Nous allons voir dans cette courte partie comment écrire des commentaires de fonctions de manière à ce que Doxygen puisse les interpréter pour générer la documentation.

3.11.1 Syntaxe

Si vous utilisez Eclipse comme IDE, vous pouvez le configurer pour interpréter les commentaires Doxygen. Pour ce faire allez dans le menu Window, Preferences, puis C/C++, Editor. Enfin dans l'encadrement Documentation Tool Comments choisissez Doxygen.

Les commentaires Doxygen sont encadrés par les balises /** et */. Ils se placent avant la déclaration de la fonction. A l'intérieur des commentaires nous allons pouvoir, entre autre, donner une description de la fonction, de ces paramètres et indiquer ce que la fonction retourne. Pour décrire les paramètres et la valeur de retour il faut utiliser des balises spéciales, soit

@param et *@return*. Le programme 3.20 donne un exemple de commentaires au format Doxygen.

```
1  #include <stdio.h>
2
3  /**
4   * @file
5   */
6
7  /**
8   * Display the message "Hello World !".
9   */
10 static void printHelloWorld( void )
11 {
12    printf( "Hello World !\n" );
13 }
14
15 /**
16  * Compute the sum of two integers.
17  *
18  * @param a   First integer.
19  * @param b   Second interger.
20  *
21  * @return Sum of a and b.
22  */
23 static int sum( int a, int b )
24 {
25    return a + b;
26 }
27
28
29 int main( void )
30 {
31    printHelloWorld();
32
33    printf( "9 + 8 : %d\n", sum( 9, 8 ) );
34
35    return 0;
36 }
```

Listing 3.20 – Commentaires Doxygen

3.11.2 Génération

Doxygen est capable de générer la documentation sous plusieurs formats comme le HTML, le PDF ou encore LaTeX. Le programme est disponible à l'adresse `http://www.stack.nl/~dimitri/doxygen/`. Une fois installé sur

votre machine, n'oubliez pas de modifier votre variable d'environnement PATH pour que l'exécutable *doxygen* soit connu dans un terminal.

Recopiez le source 3.20 et ouvrez un terminal là où se trouve le fichier. Avant de générer la documentation il faut générer un fichier de configuration pour Doxygen en tapant la commande :

```
doxygen -g
```

Normalement vous devriez avoir un nouveau fichier *Doxyfile* contenant la configuration par défaut qui sera utilisée pour générer votre documentation. Ouvrez le fichier avec un éditeur de texte pour voir son contenu et changez quelques valeurs comme suit :

PROJECT_NAME qui sera le nom de votre projet.

OUTPUT_DIRECTORY qui contiendra tous les fichiers générés par Doxygen. Par exemple vous pouvez le valoriser à *out*.

OPTIMIZE_OUTPUT_FOR_C que vous pouvez valoriser à YES car nous sommes en langage C.

GENERATE_LATEX valorisez le à NO pour ne pas générer la documentation dans ce format. Nous nous contenterons du format HTML qui est activé par défaut également.

Désormais pour générer la documentation il faut taper la commande :

```
doxygen
```

Doxygen va automatiquement parcourir tous les fichiers sources qu'il reconnait et appliquer les paramètres du fichier de configuration créé précédemment. En sortie vous devriez avoir un dossier *out* qui contient un dossier *html* où le fichier *index.html* est la page principale de la documentation. En cliquant sur le bouton "Files" puis sur votre fichier source vous verrez sa documentation.

Si vous avez bien suivi les explications les fonctions commentées ne devrait pas apparaître dans la documentation, tout simplement car elles sont statiques et considérées privées. Par défaut Doxygen ne génère que ce qui est publique. Supprimez les mots clés *static* et générer de nouveau la documentation.

3.12 Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même directement comme dans le programme 3.21 soit indirectement, c'est-à-dire par

l'intermédiaire d'autres fonctions, comme le programme 3.22.

```
1  #include <stdio.h>
2
3
4  static int factoriel( int n )
5  {
6     if( 0 == n )
7     {
8        return 1;
9     }
10
11    return n * factoriel( n - 1 );
12 }
13
14 int main( void )
15 {
16    printf( "3! = %d\n", factoriel( 3 ) );
17
18    return 0;
19 }
```

Listing 3.21 – Récursivité

```
1  #include <stdio.h>
2
3
4  static int pair( int n );
5  static int impair( int n );
6
7  static int pair( int n )
8  {
9     if( n == 0 )
10    {
11       return 1;
12    }
13
14    return impair( n - 1 );
15 }
16
17 static int impair( int n )
18 {
19    if( n == 0 )
20    {
21       return 0;
22    }
23
24    return pair( n - 1 );
25 }
```

```
26
27
28 int main( void )
29 {
30    if( 1 == pair( 4 ) )
31    {
32       printf( "le nombre 4 est pair\n" );
33    }
34    else
35    {
36       printf( "le nombre 4 est impair\n" );
37    }
38
39    return 0;
40 }
```

Listing 3.22 – Récursivité croisée

Dans le programme 3.22 les deux fonctions *pair* et *impair* sont déclarées en tête de fichier avant leur implémentation puisqu'elle dépendent l'une de l'autre. En effet si on implémente *pair* en premier le compilateur ne trouvera pas *impair*, et vice versa. La déclaration des prototypes avant l'implémentation permet de pallier à ce problème.

Nous allons analyser comment se déroule l'exécution du programme 3.21.

3.12.1 Analyse

Le point important quand une fonction récursive est exécutée est que le premier appel à cette fonction n'est pas fini lorsqu'elle est appelée à nouveau. La mémoire réservée pour le premier appel pour stocker les arguments et variables locales est conservée, les autres appels auront leur propre mémoire. Prenons le programme 3.21 pour exemple, ce qui donne l'exécution suivante :

```
factoriel( 3 )
   3 != 0
   3 * factoriel( 3 - 1 )
       factoriel( 2 )
       2 != 0
       2 * factoriel( 2 - 1 )
           factoriel( 1 )
           1 != 0
           1 * factoriel( 1 - 1 )
```

```
            factoriel( 0 )
            1
        1 * 1
        1
    2 * 1
    2
3 * 2
6
```

J'ai représenté les appels en cascade jusqu'à *factoriel(0)* qui est le premier appel où la fonction retourne une valeur. Tant que la fonction ne retournait pas de valeur on empilait les appels (d'où l'indentation). A partir de ce moment la fonction retourne la valeur 1 donc je dépile les appels pour revenir à l'appel initial et pouvoir donner le résultat final, soit 6.

3.12.2 Conseils

Tout comme les boucles on peut rapidement faire une fonction récursive infinie si la condition d'arrêt n'est jamais satisfaite (le lien suivant donne une bonne représentation de ce cas `http://thecodinglove.com/post/84806012400/my-first-recursive-function`). Ainsi il est conseillé de commencer par écrire la condition d'arrêt avant tout traitement ou appel récursif. Pour des fonctions simples vérifier le fonctionnement sur plusieurs valeurs, soit en utilisant un programme de test ou sur papier en représentant l'exécution en cascade comme précédemment.

3.13 Pointeurs

Avec le langage C nous avons la possibilité d'accéder directement à la mémoire de la machine, et ce en utilisant des pointeurs. Autrement dit, les pointeurs sont des variables qui pointent sur une adresse mémoire, généralement l'adresse d'une variable.

Nous allons voir dans les parties suivantes quel est l'intérêt de travailler avec des pointeurs. Pour manipuler les pointeurs on utilise deux opérateurs qui sont l'étoile * et le et logique &.

3.13.1 Déclaration

On peut déclarer un pointeur sur n'importe quel type, pour ce faire on ajoute une étoile entre le type et le nom de la variable à la déclaration. Par exemple :

```
int *a; /* pointeur sur un entier */
int *a, *b, *c; /* trois pointeurs sur un entier */
```

Lorsque que l'on déclare un pointeur on doit lui donner pour valeur une adresse.

3.13.2 Affectation et accès

On va vouloir faire pointer notre pointeur sur l'adresse d'une variable, l'opérateur & permet de récupérer l'adresse d'une variable, par exemple :

```
int a = 2;
int *ptr = &a; /* le pointeur pointe sur l'adresse de a */
```

Suite aux deux instructions précédentes, *ptr* contient l'adresse de *a*, on peut donc dire que *ptr* == *&a*. Étant donné que *ptr* pointe sur l'adresse de *a*, on peut accéder au contenu de la mémoire, c'est-à-dire à la valeur contenu à l'adresse que l'on pointe. Pour cela on utilise l'étoile comme ceci **ptr*. L'étoile sert donc à la déclaration et pour accéder à la valeur pointée. La figure 3.5 donne une représentation mémoire des instructions précédentes, le programme 3.23 montre un exemple de déclaration et d'utilisation des pointeurs. Notez que l'on peut afficher une adresse en utilisant *printf* et le format *%p*.

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4
5  int main( void )
6  {
7     int a = 2;
8     int *ptr = NULL;
9
10    printf( "adresses de a et ptr\n" );
11    printf( "@a   : %p\n", &a );
12    printf( "@ptr : %p\n", ptr );
13    printf( "\n" );
14
15    /* pointe sur l'adresse de a */
16    ptr = &a;
```

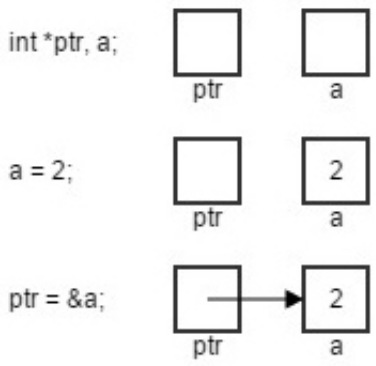


FIGURE 3.5 – Pointeurs - représentation mémoire

```
17     printf( "adresses de a et ptr\n" );
18     printf( "@a   : %p\n", &a );
19     printf( "@ptr : %p\n", ptr );
20     printf( "valeurs aux adresses de a et ptr\n" );
21     printf( "a    = %d\n", a );
22     printf( "*ptr = %d\n", *ptr );
23     printf( "\n" );
24
25     /* change la valeur pointee par ptr */
26     *ptr = 52;
27     printf( "valeurs aux adresses de a et ptr\n" );
28     printf( "a    = %d\n", a );
29     printf( "*ptr = %d\n", *ptr );
30     printf( "\n" );
31
32     return 0;
33 }
```

Listing 3.23 – Exemple pointeurs

Dans le programme 3.23 on remarque que lorsque l'on change la valeur pointée par *ptr*, la valeur de *a* change également, ce qui est tout à fait logique puisque l'on écrit directement à l'adresse de *a*.

3.13.3 Tableaux et pointeurs

On peut également initialiser un pointeur avec l'adresse d'un tableau, du moment que les types correspondent comme le montre les instructions ci-après :

```
int tab[5], *p;
p = tab;
```

A la suite des deux instructions, p va pointer sur la première case du tableau d'entiers mais on peut directement initialiser le pointeur sur l'adresse d'une autre case.

```
p = &tab[0]; /* pointe sur le premier element (p = tab) */
p = &tab[2]; /* pointe sur le troisieme element */
...
```

Lorsque que l'on pointe sur un tableau, on peut accéder aux valeurs de deux manières. Soit de la même manière qu'un tableau, en précisant l'index entre crochets, soit en utilisant l'étoile et en incrémentant l'adresse pointée. En effet si p pointe à l'adresse de *tab*, alors $p + i$ est l'adresse de la case i. Le programme 3.24 donne un exemple d'utilisation des pointeurs sur un tableau.

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4
5  int main( void )
6  {
7     int tab[] = { 1, 2, 3, 4, 5 };
8     int *p = tab;
9
10    printf( "tab[1]    : %d\n", tab[1] );
11    printf( "p[1]      : %d\n", p[1] );
12    printf( "*(p + 1) : %d\n", *(p + 1) );
13
14    return 0;
15 }
```

Listing 3.24 – Exemple pointeurs

3.13.4 Fonctions et pointeurs

Bien entendu on peut passer en paramètres de fonctions des pointeurs, et ce pour principalement deux raisons. La première raison, est qu'en donnant l'adresse de la variable, toute modification dans la fonction est reportée sur la variable puisqu'on écrit directement à son emplacement mémoire. Une seconde raison est de *soulager* la mémoire utilisée. En effet lorsque l'on donne des arguments à une fonction, un espace mémoire est réservé pour les stocker

mais en donnant l'adresse on utilise un espace mémoire déjà réservé pour la variable. Le programme 3.25 en est l'exemple.

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4
5  static void changeValue( int* a )
6  {
7     if( NULL != a )
8     {
9        *a = 42;
10    }
11 }
12
13 static void printTab( const int* tab, int size )
14 {
15    int i = 0;
16
17    if( NULL != tab && 0 < size )
18    {
19       while( i < size )
20       {
21          printf( "%d ", tab[i] );
22          i++;
23       }
24       printf( "\n" );
25    }
26 }
27
28
29 int main( void )
30 {
31    int a = 2;
32    int tab[] = { 12, 15, 1, 8, 9, 10 };
33
34    printf( "a : %d\n", a );
35
36    changeValue( &a );
37    printf( "a : %d\n", a );
38
39    printTab( tab, sizeof( tab ) / sizeof( tab[0] ) );
40
41    return 0;
42 }
```

Listing 3.25 – Pointeurs et fonctions

La fonction *changeValue* reçoit en argument l'adresse d'un entier et change sa valeur. Attention à bien utiliser l'étoile pour modifier la valeur pointée et non l'adresse. Quand à la fonction *printTab* elle affiche un tableau d'entiers. Remarquez que dans ce cas on ne change aucune valeur du tableau, on souhaite seulement le lire, on peut donc utiliser le mot clé *const* pour garantir qu'aucune écriture n'aura lieu. Ajoutez l'instruction *tab[1] = 2;* en fin de la fonction *printTab* et recompilez le fichier source. Le compilateur devrait signaler une erreur puisque grâce à *const*, *tab* est accessible en lecture seulement. L'utilisation de *const* n'est pas obligatoire mais c'est une bonne pratique.

3.13.5 Allocation et libération de mémoire

Dans les exemples que nous avons vus pour l'instant, les pointeurs sont initialisés avec des adresses de variables existantes. Cependant on peut allouer dynamiquement de la mémoire pour le pointeur de sorte qu'il ait sa propre zone mémoire de stockage. Tant que le pointeur n'est pas alloué il est préférable de l'initialiser à *NULL* ce qui permet de savoir qu'il ne pointe nulle part.

Pour allouer de la mémoire il faut utiliser la fonction *malloc* en lui précisant la quantité de mémoire à allouer (soit le nombre d'octets) comme ceci :

```
int *p = NULL;
/* alloue de la memoire pour 5 types int */
p = malloc( 5 * sizeof( int ) );
```

Ainsi p va pointer sur une zone mémoire réservée de 5 fois la taille d'un entier. Si toutefois l'allocation échoue, manque de mémoire, *malloc* retourne la valeur *NULL*.

Il est très important de libérer la mémoire que l'on a alloué dynamiquement. A chaque allocation la mémoire est réservée, lorsque l'on n'a plus besoin de cet espace, ou que l'on termine le programme, il faut libérer la mémoire réservée de sorte qu'elle ne soit pas réservée pour rien. De même que pour l'allocation, il existe une fonction très simple jouant ce rôle, *free*.

```
free( p );
p = NULL;
```

L'appel à *free* libère la mémoire mais ne change pas l'adresse de *p*, il est donc judicieux de réinitialiser le pointeur à *NULL* pour savoir que l'espace qu'il pointait n'est plus disponible.

Pourquoi vouloir allouer de la mémoire dynamiquement ? Et bien tout simplement pour ne pas gaspiller de la mémoire en déclarant statiquement des tableaux trop volumineux car on ne sait pas toujours par avance la taille réelle qui sera nécessaire lors de l'exécution du programme.

3.13.6 Conseils

Une mauvaise utilisation des pointeurs peut rendre votre programme très instable. Voici quelques conseils qui peuvent améliorer la qualité de votre code :

- Toujours initialiser les pointeurs à NULL, ou à l'adresse d'une variable. Ne jamais les laisser dans un état inconnu.
- Avant d'accéder à la valeur pointer par un pointeur, vérifier que le pointeur en question pointe sur une adresse et ne vaut pas NULL.
- Pour chaque allocation dynamique, il doit y avoir une libération de la mémoire. Si la mémoire n'est pas libérée on parle alors de fuite mémoire.

3.14 Structures

Grâce aux tableaux on peut stocker plusieurs éléments du même type. Nous allons voir maintenant comment créer de nouveaux types, ainsi que des structures et des énumérations.

3.14.1 Nouveau type

On peut avoir besoin de redéfinir un type standard avec un nouveau nom comme par exemple :

```
typedef int entier;
```

L'exemple précédent définit un nouveau type *entier* qui est un *int*. De cette manière les déclarations suivantes sont équivalentes :

```
int a = 0;
entier a = 0;
```

Le type standard *int* reste accessible et utilisable, l'instruction *typedef* déclare un type supplémentaire. La définition d'un nouveau type s'écrit de la façon suivante :

```
typedef type nom
```

Où *type* est le type sur lequel on se base, et *nom*, le nom du nouveau type.

On peut donc déclarer des variables de ce nouveau type et s'en servir comme du type initial.

On peut également utiliser des déclarations de types sur des tableaux comme le montre le programme 3.26.

```
1
2 #include <stdio.h>
3
4 /** declare un type nomme tabdeux qui est un
5  * tableau de deux entiers. */
6 typedef int tabdeux[2];
7
8 int main( void )
9 {
10    /* pas besoin des [], tabdeux est deja un tableau */
11    tabdeux t = { 1, 2 };
12
13    printf( "%d %d\n", t[0], t[1] );
14
15    return 0;
16 }
```

Listing 3.26 – Définition de type sur tableaux

La redéfinition d'un type sert surtout pour simplifier l'écriture et la lisibilité du code. Mais également pour définir des types un peu plus complexe comme les structures.

3.14.2 Structures

Les structures permettent de grouper plusieurs variables de différents types sous un seul nom, et à un seul emplacement mémoire.

Imaginons que nous voulions écrire un programme qui traite des personnes, on admet qu'une personne est spécifiée par un nom, un prénom, et une date de naissance. On pourrait définir une variable pour chaque caractéristique, ce qui deviendra vite fastidieux quand on voudra gérer plus d'une personne. Le plus simple est créer un nouveau type regroupant toutes ces caractéristiques, soit une structure.

Déclaration

Une structure se déclare avec le mot clé *struct* suivi du nom de la structure, puis des variables qui la composent entre accolades. Pour reprendre l'exemple précédent, on pourrait définir une personne via la structure suivante :

```
struct personne {
   char nom[32];
   char prenom[32];
   unsigned short annee;
   unsigned short mois;
   unsigned short jour;
};
```

Pour le nom et le prénom de la personne, principalement afin de simplifier l'exemple, je fais le choix d'un tableau de taille fixe mais on pourrait utiliser deux pointeurs, allouer la mémoire lors de l'utilisation et bien entendu libérer la mémoire en fin de programme.

Utilisation

Une fois la structure définie, on peut déclarer une variable de ce type comme suit :

```
struct personne p1;
```

Dés cet instant, en mémoire, est réservée la place nécessaire pour contenir chacun des champs de la structure. On va pouvoir accéder à chacun des champs qui composent la structure en utilisant le nom de la variable, suivi d'un point et du nom du champ. Par exemple pour affecter une valeur à l'année de notre structure déclarée précédemment, on écrit :

```
p1.annee = 1975;
```

Une structure peut également contenir d'autres structures, par exemple on peut généraliser l'exemple précédent comme suit :

```
struct date {
   unsigned short annee;
   unsigned short mois;
   unsigned short jour;
}
```

```
struct personne {
   char nom[32];
   char prenom[32];
   struct date naissance;
};
```

Cette fois pour accéder à l'année, il faut passer par *p1* le nom de la variable de type *personne*, puis par *naissance*, la variable déclarée dans *personne*. Ce qui donne l'appel suivant :

```
p1.naissance.annee = 1975;
```

Le programme 3.27 donne un exemple d'utilisation de la structure *personne*. Le programme initialise la structure puis l'affiche, le tout en utilisant des fonctions. De cette manière vous pouvez remarquez que, comme pour les variables de type standard, lorsque l'on veut que la structure soit initialisée dans une fonction, il faut utiliser un pointeur sur celle-ci. Sinon les modifications dans la fonction même de la structure n'auront aucun effet en sortie de la fonction. Notez que lorsque l'on veut accéder à un champ d'un pointeur sur structure, on n'utilise plus le point mais une flèche.

```
1  #include <stdio.h>
2  #include <stddef.h>
3  #include <string.h>
4
5
6  /**
7   * Structure that represents a date.
8   */
9  struct date
10 {
11    unsigned short annee;
12    unsigned short mois;
13    unsigned short jour;
14 };
15
16 /**
17  * Structure that represents a person.
18  */
19 struct personne
20 {
21    char nom[32];
22    char prenom[32];
23    struct date naissance;
24 };
25
```

```
/**
 * Display a date, format DD/MM/YYYY.
 *
 * @param[in] d   Date to display.
 */
static void afficheDate( struct date d )
{
   printf( "%d/%d/%d\n", d.jour, d.mois, d.annee );
}

/**
 * Display the last name, first name, and the birthday
 * of a person.
 *
 * @param[in] p   Person to display.
 */
static void affichePersonne( struct personne p )
{
   printf( "%s %s\n", p.prenom, p.nom );
   afficheDate( p.naissance );
}

/**
 * Initialize a personne structure with the parameters
 * given as arguments.
 *
 * @param[out] p1       Structure of type personne to initialize.
 * @param[in]  nom      First name of the person.
 * @param[in]  prenom   Last name of the person.
 * @param[in]  annee    Year of birth.
 * @param[in]  mois     Month of birth.
 * @param[in]  jour     Day of birth.
 *
 * @return  0 - no errors.
 *         -1 - null pointers given.
 */
static int setPersonne(
      struct personne* p1,
      const char* nom,
      const char* prenom,
      unsigned short annee,
      unsigned short mois,
      unsigned short jour )
{
   if( NULL == nom ||
       NULL == prenom )
   {
      return -1;
   }
```

```
75
76    strcpy( p1->nom, nom );
77    strcpy( p1->prenom, prenom );
78    p1->naissance.annee = annee;
79    p1->naissance.mois = mois;
80    p1->naissance.jour = jour;
81
82    return 0;
83 }
84
85 int main( void )
86 {
87    struct personne p1;
88
89    if( 0 == setPersonne( &p1, "Golade", "Larry", 1978, 11, 22 ) )
90    {
91       affichePersonne( p1 );
92    }
93
94    return 0;
95 }
```

Listing 3.27 – Exemple d'utilisation des structures

Vous aurez sans doute remarqué également que dans l'exemple j'initialise les chaînes de caractères en utilisant la fonction *strcpy*, une fonction standard disponible en incluant le fichier *string.h*. Une des principales fonctions utilisées pour les chaînes de caractères, mais pas forcément la plus adaptée.

3.14.3 Chaînes de caractères

Les chaînes de caractères sont de simples tableaux de caractères mais il existe des fonctions standards qui permettent de réaliser des traitements classiques comme la copie d'une chaîne dans une autre ou récupérer la longueur d'une chaîne. Lorsque l'on travaille avec ces fonctions, on utilise des pointeurs et on considère qu'une chaîne de caractères se termine toujours par le caractère spécial \0.

C'est en rencontrant ce caractère que les fonctions de traitement des chaînes de caractères s'arrêtent et comprennent que la chaîne est terminée. Donc pour utiliser les fonctions standards pour les chaînes de caractères, il faut toujours garder un emplacement du tableau pour le caractère de fin de chaîne \0.

Pour copier une chaîne de caractères dans une autre on utilise *strcpy*, pour *string copy*, de cette manière :

```
char msg1[16];
char *msg2 = "bonjour";

strcpy( msg1, msg2 );
```

On doit donner, dans l'ordre, la destination et la source. Le caractère de fin de chaîne est également copié. De ce fait *msg1* aura le même contenu que *msg2*. Cependant que se passe-t-il si *msg2* est plus long que *msg1* ? Eh bien on va provoquer un *buffer overflow*, on va déborder en mémoire dans une zone qui n'est pas réservée pour *msg1* ce qui peut avoir des conséquences étranges, dont le résultat est le plantage du programme.

Pour éviter le *buffer overflow*, il est conseillé d'utiliser *strncpy* pour copier des chaînes de caractères. La fonctionnalité est identique mais il y a un paramètre en plus, le nombre maximum d'octets à copier. Par exemple, considérons le code suivant :

```
char msg1[4];
char *msg2 = "bonjour";

strcpy( msg1, msg2 );
```

La taille de *msg1* est trop petite pour contenir *msg2*, on va provoquer un *buffer overflow*. Par contre, avec le code suivant :

```
char msg1[4];
char *msg2 = "bonjour";

strncpy( msg1, msg2, sizeof( msg1 ) / sizeof( char ) );
```

En exécutant le code précédent, on évite le *buffer overflow*, mais le caractère de fin de chaîne n'est pas copié. Dans ce cas, nous pouvons le faire nous-même, en connaissant le nombre d'octets copiés et en plaçant \0 en dernier. Sinon utiliser une méthode un peu plus sécurisée, la fonction *strncat* qui permet de concaténer deux chaînes de caractères et ajoute automatiquement le caractère de fin de chaîne.

```
char msg1[4];
char *msg2 = "bonjour";

/* chaine vide puisque le caractere de fin de chaine
 * est au premier emplacement */
msg1[0] = '\0';
/* on enleve 1 au nombre maximum de caracteres que peut
 * contenir msg1 pour laisser une place pour \0 */
strncat( msg1, msg2, sizeof( msg1 ) / sizeof( char ) - 1 );
```

Le code précédent est une bonne alternative pour copier une chaîne dans une autre, y compris le caractère de fin de chaîne. Cette méthode est plus sécurisée que *strcpy* ou *strncpy*. Le résultat d'exécution fait que *msg1* vaut :

```
char msg1 = { 'b', 'o', 'n', '\0' };
```

Pour information, les pages de manuel des fonctions sont disponibles aux adresses suivantes : `http://linux.die.net/man/3/strcpy`, `http://linux.die.net/man/3/strlen` et `http://linux.die.net/man/3/strncat`.

3.14.4 Énumérations

Les énumérations sont un ensemble de noms ayant chacun une valeur différente. On s'en sert essentiellement pour rendre le code plus lisible, et lorsque la valeur en elle même n'est pas importante. Par exemple une fonction pouvant renvoyer différents codes d'erreurs, peu importe les valeurs, le but étant d'avoir pour chaque code une valeur différente. Dans ce cas on peut utiliser une énumération comme dans le programme 3.28.

Une énumération est déclarée via le mot clé *enum* suivi d'un nom. Ensuite on donne la liste des valeurs nommées entre accolades, chaque valeur étant séparée par une virgule (c.f. programme 3.28). On peut attribuer une valeur numérique à un nom puis chaque valeur suivante sera incrémentée de 1. Ainsi dans l'exemple la valeur numérique de départ est 0.

Dans la fonction de l'exemple, on peut tout aussi bien utiliser directement des valeurs numériques, 0 pour ok, -1 si le pointeur est null, etc. Mais il est plus facile d'abstraire la valeur et d'utiliser un nom, plus explicite. De plus le retour de la fonction donne le nom de l'énumération, en allant voir

sa définition on sait tout de suite le genre d'erreurs que peut retourner la fonction.

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4
5  enum err_fct {
6     no_error = 0,
7     null_pointer,
8     bad_value,
9  };
10
11 /**
12  * Accept positive number only.
13  */
14 static enum err_fct function1( int* arg )
15 {
16    if( NULL == arg )
17    {
18       return null_pointer;
19    }
20
21    if( 0 >= arg )
22    {
23       return bad_value;
24    }
25
26    return no_error;
27 }
28
29
30 int main( void )
31 {
32    int a = 0;
33    enum err_fct error;
34
35    error = function1( &a );
36    if( no_error != error )
37    {
38       printf( "function1 error : %d\n", error );
39    }
40
41    return 0;
42 }
```

Listing 3.28 – Exemple d'utilisation des énumérations

3.14.5 Nouveaux types struct et enum

Comme pour les types standards, il est possible de créer de nouveaux types à partir de structures ou d'énumérations, pour éviter d'avoir à répéter *struct* ou *enum* lors de la déclaration de variables. Les programmes 3.29 et 3.30 reprennent les programmes précédents en utilisant des nouveaux types.

```
1  #include <stdio.h>
2  #include <stddef.h>
3  #include <string.h>
4
5
6  /**
7   * Structure that represents a date.
8   */
9  typedef struct _date_
10 {
11    unsigned short annee;
12    unsigned short mois;
13    unsigned short jour;
14 } date;
15
16 /**
17  * Structure that represents a person.
18  */
19 typedef struct _personne_
20 {
21    char nom[32];
22    char prenom[32];
23    date naissance;
24 } personne;
25
26 /**
27  * Display a date, format DD/MM/YYYY.
28  *
29  * @param[in] d   Date to display.
30  */
31 static void afficheDate( date d )
32 {
33    printf( "%d/%d/%d\n", d.jour, d.mois, d.annee );
34 }
35
36 /**
37  * Display the last name, first name, and the birthday
38  * of a person.
39  *
40  * @param[in] p   Person to display.
41  */
42 static void affichePersonne( personne p )
```

```
43 {
44    printf( "%s %s\n", p.prenom, p.nom );
45    afficheDate( p.naissance );
46 }
47
48 /**
49  * Initialize a personne structure with the parameters
50  * given as arguments.
51  *
52  * @param[out] p1        Structure of type personne to initialize.
53  * @param[in]  nom       First name of the person.
54  * @param[in]  prenom    Last name of the person.
55  * @param[in]  annee     Year of birth.
56  * @param[in]  mois      Month of birth.
57  * @param[in]  jour      Day of birth.
58  *
59  * @return  0 - no errors.
60  *         -1 - null pointers given.
61  */
62 static int setPersonne(
63       personne* p1,
64       const char* nom,
65       const char* prenom,
66       unsigned short annee,
67       unsigned short mois,
68       unsigned short jour )
69 {
70    if( NULL == nom ||
71        NULL == prenom )
72    {
73       return -1;
74    }
75
76    strcpy( p1->nom, nom );
77    strcpy( p1->prenom, prenom );
78    p1->naissance.annee = annee;
79    p1->naissance.mois = mois;
80    p1->naissance.jour = jour;
81
82    return 0;
83 }
84
85 int main( void )
86 {
87    personne p1;
88
89    if( 0 == setPersonne( &p1, "Golade", "Larry", 1978, 11, 22 ) )
90    {
91       affichePersonne( p1 );
```

```
92    }
93
94    return 0;
95 }
```

Listing 3.29 – Exemple nouveau type structure

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4
5  typedef enum _err_fct_ {
6     no_error = 0,
7     null_pointer,
8     bad_value,
9  } err_fct;
10
11 /**
12  * Accept positive number only.
13  */
14 static err_fct function1( int* arg )
15 {
16    if( NULL == arg )
17    {
18       return null_pointer;
19    }
20
21    if( 0 >= arg )
22    {
23       return bad_value;
24    }
25
26    return no_error;
27 }
28
29
30 int main( void )
31 {
32    int a = 0;
33    err_fct error;
34
35    error = function1( &a );
36    if( no_error != error )
37    {
38       printf( "function1 error : %d\n", error );
39    }
40
41    return 0;
42 }
```

Listing 3.30 – Exemple nouveau type énumérations

3.15 Listes chaînées

Dans cette partie il est nécessaire d'avoir compris ce que sont les pointeurs et les structures.

Comme nous l'avons vu dans un chapitre précédent, les tableaux permettent de stocker des variables de même type et d'y accéder de façon très simple. Ils ont cependant des inconvénients : ils réservent un grand espace mémoire continu et ne sont pas adaptés si vous avez un nombre de variables à stocker qui varie avec une amplitude importante au cours de l'exécution du programme.

Pour remédier à ces inconvénients, on utilise des listes chaînées. Leur implémentation est un peu complexe, aussi beaucoup de langages proposent une librairie pour les utiliser. Ce chapitre explique comment les listes chaînées fonctionnent.

3.15.1 Définition

Les listes sont un objet mathématique récursif dont une définition est la suivante, une liste est :

1. soit la liste vide notée $\emptyset$.
2. soit un couple formé d'un élément et d'une liste.

D'après la définition précédente, on peut représenter une liste chaînée d'entiers par la structure suivante :

```
struct element {
   int valeur;
   struct element* next;
};
```

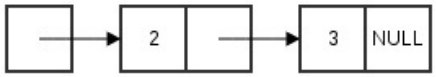


FIGURE 3.6 – Listes chaînées - représentation mémoire

L'idée est la suivante :

1. un élément est constitué de deux parties, la valeur qu'il stocke, et l'adresse de l'élément suivant.
2. une liste pointe sur le premier élément.

Prenons pour exemple une liste de deux éléments, soit les valeurs 2 et 3. En mémoire la liste peut-être représentée par la figure 3.6. Le premier carré est la liste, soit l'adresse du premier élément. Ensuite le premier élément qui est la valeur entière 2 puis l'adresse du prochain élément qui est la valeur 3 et l'adresse NULL. Étant le dernier élément l'adresse de l'élément suivant est NULL.

Comme on travaille avec des adresses, on peut *facilement* ajouter de nouveaux éléments en tête, fin ou encore milieu de liste. Pour ajouter un élément en début de liste on crée le nouvel élément puis on change l'adresse de la liste pour qu'elle pointe sur ce nouvel élément, enfin dans le nouvel élément on fait pointer l'adresse de l'élément suivant sur le reste de la liste (soit la liste précédente). La figure 3.7 donne un exemple d'ajout en tête de liste.

Pour ajouter un élément en fin de liste, on doit tout d'abord se placer sur le dernier élément, c'est-à-dire l'élément qui pointe sur une adresse NULL. Ensuite on change l'adresse pour le nouvel élément crée, comme le montre la figure 3.8.

Ajouter un élément en milieu requiert tout d'abord d'itérer sur la liste jusqu'à trouver la position, ou la valeur après laquelle, l'on veut l'insérer.

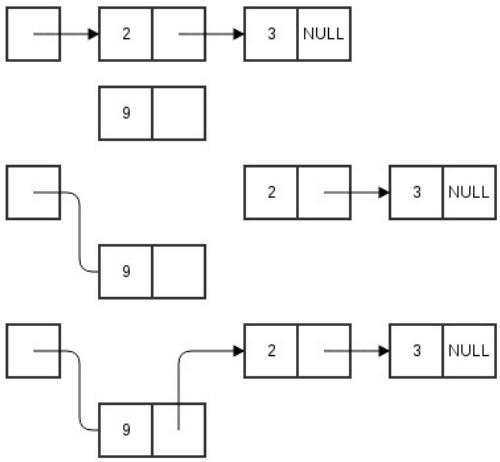


FIGURE 3.7 – Listes chaînées - ajout en tête

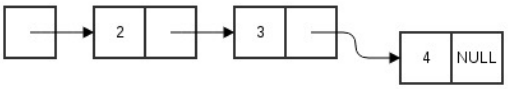


FIGURE 3.8 – Listes chaînées - ajout en queue

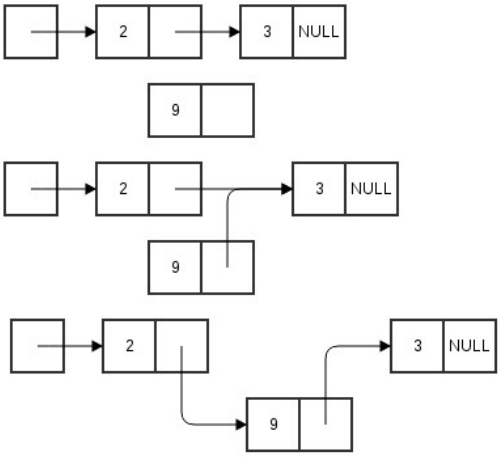


FIGURE 3.9 – Listes chaînées - ajout en milieu

Par exemple sur la figure 3.9, on souhaite ajouter la valeur 9 après la valeur 2. Il faut tout d'abord parcourir la liste jusqu'à trouver la valeur 2. Ensuite regarder où pointe le prochain élément de la valeur 2 et faire pointer le prochain élément de la valeur 9 au même endroit. Finalement on change le prochain élément de la valeur 2 pour qu'il pointe sur la valeur 9.

Les listes chaînées demande une certaine gymnastique des pointeurs, il est donc intéressant d'avoir une base pour les manipuler.

3.15.2 Manipulation

L'archive *chainedlist.zip* contient un exemple d'implémentation des fonctions de base pour manipuler des listes chaînées, à savoir ajouter un élément en tête ou fin de liste, récupérer la valeur de tête ou de fin, et détruire une liste. Par détruire la liste on entend libérer la mémoire réservée pour la liste et ses composantes.

Le fichier *int_chained_list.h* contient les fonctions publiques et le fichier *int_chained_list.c* leur implémentation. Remarquez en début de ce fichier la définition de la structure *linked_list* qui représente une liste chaînée d'entier, suivi de quelques déclarations statiques, puis de l'implémentation des fonctions publiques. Dans les déclarations statiques on voit la déclaration de la variable *list*, unique instance de la liste qui sera utilisée par les fonctions de ce module. De ce fait ce module ne gère qu'une seule liste chaînées à l'exécution. Il y a également deux fonctions statiques, donc utilisées que dans ce fichier source, *get_latest_item* qui permet de se placer sur le dernier élément de la liste, et *free_list_given* que nous verrons un peu plus loin.

Analysons la fonction *add_to_head* qui permet d'ajouter un élément en tête de liste. Première étape, est-ce que la liste est vide ? si oui on crée le nouvel élément directement en utilisant la variable *list*. Si non, on crée un nouvel élément nommé *head*, on initialise sa valeur avec celle donnée en paramètre et son élément suivant est *list*, puis on met à jour *list* pour son adresse soit celle de *head*.

Pour ajouter un élément en fin de liste, on procède de la même manière pour la première étape, à savoir si la liste est vide, l'élément crée devient la liste. Sinon on crée un nouvel élément nommé *tail*, on se place sur le dernier élément de la liste *list* et on met à jour cet élément pour que *next* pointe sur *tail*.

Pour libérer la mémoire, la fonction publique *free_list* appelle une fonction récursive privée *free_list_given*. Cette dernière reçoit en paramètre une liste, tant que la liste n'est pas le dernier élément, on appelle de nouveau la fonction avec l'élément suivant (soit le reste de la liste). De cette manière lorsque l'on arrive sur le dernier élément, on appelle *free* sur celui-ci pour le supprimer, et par dépilage supprimer tous les éléments précédents jusqu'au premier.

Dans l'exemple fourni avec ce cours on utilise des entiers comme valeurs, mais on peut utiliser n'importe quel type, standard, ou complexe, voire même stocker des adresses. Pour bien comprendre les mécanismes d'ajouts et de libération vous pouvez ajouter des traces au programme en utilisant *printf* et le format *%p* pour afficher des adresses. Par exemple, dans la fonction *free_list_given* ajouter la ligne ci-dessous juste avant l'instruction *free* et vous verrez les adresses qui seront libérées.

```
printf( "%p\n", thelist );
```

3.16 Conclusion

Nous avons vu dans ce chapitre les bases de la programmation et du langage C, les variables, boucles d'itérations, comment construire un programme en fonctions et de façon modulaire, et avons abordé des concepts plus complexes et spécifiques au langage, les pointeurs, pour terminer par les listes chaînées.

La programmation modulaire nous a permis d'architecturer notre programme en rassemblant dans des fichiers séparés des fonctionnalités communes. De sorte qu'un module possède un ensemble de fonctions ayant une base commune, fonctions mathématiques, fonctions traitant des listes chaînées, etc.

La programmation modulaire est une base, du moins une bonne vision, de ce que peut être la programmation orientée objet.

Chapitre 4

Programmation Orientée Objet

Dans ce chapitre nous allons voir les bases de la programmation orientée objet, à savoir les notions de classe, d'objet et d'encapsulation.

Les deux modélisations principales en informatique sont la perspective algorithmique et la perspective orientée objet. Le principe de l'approche algorithmique consiste à décomposer de gros algorithmes en fonctions plus petites. Les logiciels centrés sur les algorithmes étant difficiles à mettre à jour, le développement logiciel s'appuie de plus en plus sur une perspective orientée objet dont les briques de base sont les objets.

Le langage de programmation C vu jusqu'à présent, ne permet la programmation orientée objet, nous allons donc utiliser le C++, grand frère du C, dans ce chapitre. Par convention lorsque l'on écrit du code C++ l'extension du fichier est .cpp et non plus .c. De même pour compiler du C++ il faut appeler la commande *g++* et non plus *gcc*. Si vous compilez vos programmes en utilisant un IDE, celui-ci devrait s'en sortir pour comprendre que vous faites du C++.

4.1 Quelques notions de C++

En C++ il y a quelques méthodes standards qui changent par rapport au langage C, comme les entrées sortie et la notion de référence. Je vais donner dans cette partie ces quelques différences qui nous servirons plus tard dans l'écriture du code. Sachez toutefois que les fonctions standard du C peuvent être utilisées en C++.

4.1.1 Entrée / Sortie

Nous avons vu en C les fonctions *printf* et *scanf* pour afficher et saisir des données. En C++ les équivalents sont respectivement *cout* et *cin*. Par exemple les instructions suivantes sont équivalentes :

```
int a = 2;
float b = 0.5;
const char* message = "Bonjour";

/* en C */
printf( "%d - %f - %s\n", a, b, message );

/* en C++ */
cout << a << " - " << b << " - " << message << endl;
```

Avec *cout* on peut directement écrire les variables à leur emplacement dans le message final, chaque variable est séparée par deux chevrons inférieurs, *endl* est l'équivalent
n pour le saut de ligne.

De même pour la saisie, les instructions suivantes sont équivalentes :

```
int n = 0;

/* en C */
scanf( "%d", &n );
printf( "valeur de n : %d\n", n );

/* en C++ */
cin >> n;
cout << "valeur de n : " << n << endl;
```

Avec *cin* il n'y a pas besoin de mettre un & devant la variable car la fonction utilise la notion de référence que nous allons voir peu après. Tout comme *cout* qui utilise des chevrons inférieurs, *cin* utilise des chevrons supérieurs.

L'utilisation *cout* et *cin* nécessite d'ajouter les lignes suivantes en début de fichier :

```
#include <iostream>
using namespace std;
```

4.1.2 Références

Pour comprendre les références, il faut avoir compris ce que sont les pointeurs en C. On peut dire qu'une référence à une variable est en quelque sorte son adresse, on utilise cette fonctionnalité essentiellement pour les arguments de fonctions dont la valeur change dans la fonction. En C on utilise un pointeur, en C++ on peut utiliser une référence.

```
/* en C */
static void fonction_c( int* ptr )
{
   (*ptr) = 4;
}

/* en C++ */
static void fonction_cpp( int& p )
{
   p = 4;
}

int a = 0;

fonction_c( &a );
fonction_cpp( a );
```

On utilise le symbole & pour représenter la référence en C++ (l'adresse en C), lorsque l'argument d'une fonction est une référence, il n'y a pas besoin de répéter le symbole & lors de l'appel de la fonction. De même dans la fonction, il n'y a pas besoin d'utiliser un caractère spécial pour indiquer que l'on souhaite changer la valeur de p, c'est automatique.

4.2 Les objets

Le but de la modélisation par objets est de d'écrire des objets. Par exemple, Laurent Dupond, Luky Luke, la page 34 et la fenêtre du haut sont des objets. Un objet est simplement quelque chose qui a un sens dans le contexte de l'application.

Un objet est un concept, une abstraction ou une chose ayant des limites très claires et un sens précis dans le contexte du problème étudié. Les ob-

jets servent un double objectif : ils améliorent la compréhension du monde réel et proposent une base pratique pour l'implémentation informatique. La décomposition d'un problème en objets dépend à la fois du jugement et de la nature du problème. Il n'existe pas une seule et unique représentation correcte.

Chaque objet a une identité et peut être distingué des autres. Deux pommes ayant les mêmes couleur, forme et texture demeurent des pommes individuelles , une personne peut manger l'une, puis l'autre. Le terme identité signifie que les objets peuvent être distingués grâce à leur existence inhérente et non grâce à la description des propriétés qu'ils peuvent avoir.

4.3 Les classes

Une classe d'objets d'écrit un groupe d'objets ayant des propriétés similaires (données membres) et un comportement commun (fonctions membres ou méthodes). Personne, Personnage de bande dessinée, Page et Fenêtre sont des classes d'objets. Chaque Personne a un âge, un QI et peut travailler. Les objets et les classes d'objets apparaissent souvent comme des noms dans les descriptions de problèmes.

Le terme classe est souvent utilisé à la place de classe d'objets. On parle de l'instance d'une classe pour désigner un objet d'une classe. Les objets d'une classe ont le même type de comportement et les mêmes propriétés. L'unicité de la plupart de ces objets provient souvent des différences dans les valeurs des données membres. Pourtant, il est possible d'avoir des objets ayant les mêmes valeurs de données membres.

Les objets d'une classe partagent un objectif sémantique commun, plus encore que la spécification de propriétés et de comportements communs. Bien qu'une étable et un cheval aient un prix et un âge, ils peuvent appartenir à des classes différentes. Si l'étable et le cheval ne sont considérés que comme des actifs financiers, ils pourront appartenir à la même classe. Mais si le développeur prend en compte qu'un homme peint une étable et qu'il nourrit un cheval, l'étable et le cheval seront modélisés comme des classes différentes. L'interprétation de la sémantique dépend du but de chaque application et est fonction du jugement individuel.

Chaque objet connaît sa classe. La classe d'un objet est une propriété implicite de l'objet.

Le fait de regrouper les objets en classe permet d'abstraire un problème. L'abstraction donne à la modélisation sa puissance et sa capacité à généraliser. Les définitions communes (tels que les noms des données membres) sont stockés une fois par classe plutôt qu'une fois par instance. Les méthodes sont écrites une fois par classe et tous les objets de la classe bénéficient de la réutilisation du code.

4.4 Déclaration d'une classe et d'un objet

Déclarer une classe correspond plus ou moins à définir un nouveau type. Une fois déclarée, on utilise une classe pour déclarer des objets de cette classe. C'est analogue à déclarer des variables d'un nouveau type.

La déclaration d'une classe permet de définir les données membres qui constituent les objets de cette classe ainsi que les méthodes que l'on pourra appeler sur ces objets.

4.4.1 Déclaration d'une classe

La syntaxe de déclaration de classe est donnée par le listing 4.1, contenu d'un fichier d'en-tête.

```
1
2  #ifndef CLASS_H_
3  #define CLASS_H_
4
5
6  class nomClass
7  {
8  public:
9          int donnee1;
10         double donnee2;
11
12         void methode1( int a );
13         void methode2( void );
14
15 private:
16         int autremethode( void );
17
18         char donnee3;
```

```
19 };
20
21
22 #endif
```

Listing 4.1 – Déclaration d'une classe

Une classe possède un nom, et l'on définit ses attributs (variables) et méthodes (fonctions) entre accolades. Notez qu'il faut obligatoirement terminer la déclaration par un point virgule après la dernière accolade.

Remarques les mots clés *public* et *private*, ils définissent des zones, tous ce qui est déclaré après *public* sera considéré publique et accessible à l'extérieur de la classe, de même, tous ce qui est déclaré *private* sera considéré privé à la classe et donc non accessible en dehors de la classe. On retrouve ici une notion déjà vue lors de la programmation modulaire pour restreindre l'accessibilité aux variables ou fonctions d'un module. Il est courant que les attributs d'une classe soient privés.

Les listings 4.2 et 4.3 montrent la déclaration d'une classe *Emplacement* et son implémentation. La déclaration de la classe s'effectue dans un fichier d'en-tête et son implémentation dans un fichier source, comme nous sommes en C++ l'extension du fichier source n'est plus .c mais .cpp pour C plus plus.

La classe *Emplacement* possède deux attributs, x et y représentant des coordonnées privées donc seule la classe peut y avoir accès. Dans le fichier source, notez la manière dont sont implémentées les méthodes, on donne le type, le nom de la classe suivie de deux fois :, puis le nom de la fonction et ses arguments. Bien entendu pour que le fichier source connaissent la classe il faut inclure le fichier d'en-tête via *#include*, ici le fichier d'en-tête se nomme *emplacement.h*.

```
1
2 #ifndef EMPLACEMENT_H_
3 #define EMPLACEMENT_H_
4
5
6 class Emplacement
7 {
8 public:
9         Emplacement( void );
10        ~Emplacement( void );
```

```
11
12          void init( float abscisse, float ordonnee );
13          void deplace( float dx, float dy );
14          void affiche( void );
15
16 private:
17          float x;
18          float y;
19 };
20
21
22 #endif /* EMPLACEMENT_H_ */
```

Listing 4.2 – emplacement.h

```
1
2 #include <stdio.h>
3 #include "emplacement.h"
4
5
6 Emplacement::Emplacement( void )
7 {
8          x = 0;
9          y = 0;
10 }
11
12 Emplacement::~Emplacement( void )
13 {
14 }
15
16 void Emplacement::init( float abscisse, float ordonnee )
17 {
18          x = abscisse;
19          y = ordonnee;
20 }
21
22 void Emplacement::deplace( float dx, float dy )
23 {
24          x += dx;
25          y += dy;
26 }
27
28 void Emplacement::affiche( void )
29 {
30          printf( "%f %f\n", x, y );
31 }
```

Listing 4.3 – emplacement.cpp

4.4.2 Déclaration d'un objet

Une fois la classe *Emplacement* implémentée, on peut déclarer un objet par exemple *point* comme le montre le code suivant :

```
#include <stdio.h>
#include "emplacement.h"

int main( void )
{
   Emplacement point;

   return 0;
}
```

Á la déclaration de l'objet, celui-ci sera automatiquement initialisé par son constructeur. Le constructeur d'une classe est une méthode portant le même nom que la classe, et sans type de retour, dans notre exemple 4.2 le constructeur est *Emplacement*(*void*) et son implémentation dans le fichier source initialise les attributs x et y à 0. On peut faire ce que l'on veut dans le constructeur, généralement on se contente d'initialiser l'objet. Une classe peut avoir autant de constructeur que l'on souhaite, du moment que leur prototype est différent. En effet on peut donner au constructeur des arguments, par exemple on peut créer un constructeur qui prend en paramètre l'abscisse et l'ordonnée de notre emplacement pour qu'on puisse les donner directement à la déclaration de l'objet :

```
/* dans l'implémentation de Emplacement */
void Emplacement::Emplacement( float abscisse, float ordonnee )
{
   /* on appelle notre methode qui initialise x et y */
   init( abscisse, ordonnee );
}

/* dans le programme principal */
/* constructeur par defaut x et y valent 0 */
Emplacement point;
/* constructeur autre, x vaudra 1 et y vaudra 2 */
Emplacement autre_point( 1, 2 );
```

Lorsque l'objet est détruit, c'est le destructeur qui est appelé, comme le constructeur il porte le nom de la classe mais avec le symbole tilde devant, $\sim Emplacement(void)$. Dans notre cas il n'y a rien à faire lorsque l'objet sera détruit, nous verrons plus tard son utilité.

4.4.3 Encapsulation

L'association d'attributs et de méthodes au sein d'une classe, avec la possibilité de rendre privés certains d'entre eux, s'appelle l'encapsulation des données.

4.4.4 Accès aux données et méthodes d'un objet

On accède aux données et méthodes d'un objet en écrivant le nom de l'objet suivi d'un point puis du nom de la donnée ou de la méthode. On ne peut accéder qu'aux données et méthodes publiques d'un objet en dehors de la déclaration de sa classe.

```
1
2  #include <stdio.h>
3  #include "emplacement.h"
4
5  int main( void )
6  {
7          /* declaration d'un objet de type Emplacement */
8          Emplacement point;
9
10         /* appel des methodes publiques */
11         point.init( 10, 25 );
12         point.affiche();
13         point.deplace( 5, 5 );
14         point.affiche();
15
16         return 0;
17 }
```

Listing 4.4 – Exemple d'utilisation d'un objet

Dans le programme 4.4, fichier *emp_main.cpp*, on déclare un objet du type de classe *Emplacement*, nommé *point*. Ensuite on appelle différentes méthodes de cette classes, seulement les méthodes publiques car ce sont les seules accessibles en dehors de l'implémentation de la classe.

L'instruction $point.init(0, 0)$; permet d'appeler la méthode *init* de l'objet *point*. Ce dernier étant de type *Emplacement*, regardons dans l'implémentation de la classe le code de la méthode *init*. On voit que la fonction initialise x et y avec les paramètres *abscisse* et *ordonnee*. Les deux variables x et y sont des données membres (des attributs) de la classe. Après l'appel à *init* les variables x et y de l'objet *point* seront modifiées.

4.4.5 Utilisation des objets

Lorsque deux variables sont d'une même classe, elles sont compatibles entres elles et on peut donner les valeurs des données membres de l'une à l'autre par copie. Par exemple :

```
int main( void )
{
   Emplacement point, pointbis;

   point.init( 4, 3 );
   pointbis = point;
   pointbis.affiche();

   return 0;
}
```

Lorsque le programme exécute l'instruction $pointbis = point$; c'est le constructeur de copie qui va se charger de faire la copie de l'objet *point* dans l'objet *pointbis*. Notre classe ne possède pas de constructeur de copie spécifique, nous avons seulement le un constructeur simple et un destructeur. Dans ce cas c'est le compilateur qui créera un constructeur de copie par défaut, lequel recopie la valeur de chacun des attributs d'une classe à l'autre. Souvent ce constructeur par défaut n'est pas satisfaisant et il nous faut en créer un explicitement. Par exemple dans le cas où des attributs sont des pointeurs, le constructeur par défaut se contentera de copier l'adresse du pointeur alors qu'il est plus judicieux d'allouer le pointeur dans le nouvel objet et de recopier la valeur seulement.

4.4.6 Tableaux d'objets

Une classe est un type et peut tout à fait être utilisée dans un tableau, par exemple si l'on veut créer plusieurs emplacements on peut écrire :

```
Emplacement courbe[5];
courbe[0].init( 4, 3 );
courbe[1].affiche();
...
```

4.4.7 Allocation dynamique d'objets

On peut définir un pointeur sur un objet, comme pour les types standards. En C++ on utilise d'autres méthodes pour allouer et libérer la mémoire que *malloc* et *free*, on utilise les méthodes *new* et *delete* comme ceci :

```
Emplacement *point = NULL, *autre_point = NULL;

/* sans les parentheses, constructeur par defaut */
point = new Emplacement;
/* autre constructeur */
autre_point = new Emplacement( 1, 2 );

/* liberation de la memoire */
delete point;
delete autre_point;
```

Pour accéder aux membres d'un pointeur sur un objet, il faut utiliser la flèche (comme pour un pointeur de structure) :

```
Emplacement *point = new Emplacement;

point->init( 3, 4 );
point->affiche();

delete point;
```

Précédemment j'ai parlé du destructeur, il est appelé lorsque l'objet n'est plus utilisé ou détruit, ici le destructeur sera appelé sur l'instruction *deletepoint*;. Si l'objet est déclaré dans une fonction, il sera détruit après l'exécution de la fonction.

4.5 Héritage

L'héritage est une relation entre une classe principale (couramment appelée classe mère) et une (ou plusieurs) classe dérivée de celle-ci, cependant

plus spécifique (classe fille). L'enfant hérite des propriétés du parent (attributs et méthodes) mais, le plus souvent, ceux-ci s'ajoutent aux propres attributs et méthodes de l'enfant.

Lorsqu'un enfant a une méthode dont la signature (rappel : prototype de la méthode) est la même que celle d'une méthode du parent, la méthode de l'enfant prévaut sur celle du parent. Ce phénomène est connu sous le nom de polymorphisme.

On indique qu'une classe hérite d'une autre en ajoutant à la suite du nom de la classe deux points, la portée de l'héritage (on détaillera plus tard la portée) et le nom de la classe dont on hérite. On précise alors les nouveaux attributs et nouvelles méthodes que l'on redéfinit.

Reprenons la classe *Emplacement* et imaginons une nouvelles classe, *PointCouleur* qui hérite de *Emplacement*. La nouvelle classe possède un attribut *couleur* et est définie comme dans les listings 4.5 et 4.6. La classe hérite de *Emplacement*

```
1
2  #ifndef POINTCOULEUR_H_
3  #define POINTCOULEUR_H_
4
5  #include "emplacement.h"
6
7  class PointCouleur : public Emplacement
8  {
9  public:
10         PointCouleur( void );
11         PointCouleur( const char* c );
12         ~PointCouleur( void );
13
14         void affiche_pc( void );
15
16 private:
17         char couleur[64];
18 };
19
20
21 #endif /* POINTCOULEUR_H_ */
```

Listing 4.5 – pointcouleur.h

```
1
2  #include <stdio.h>
```

```
3  #include <string.h>
4  #include "pointcouleur.h"
5
6  PointCouleur::PointCouleur( void )
7  {
8          /* memset permet d'initialiser une zone memoire avec une
9           * valeur donne. ici l'appel permet d'initialiser la zone
10          * memoire ou est stocker 'couleur' avec des zeros.
11          * sizeof permet de connaitre sur combien d'octets est
12          * codee une variable. Dans le cas de couleur qui est
13          * un tableau de char, on aura la taille du tableau car
14          * un char est code sur un octet. */
15         memset( couleur, 0, sizeof( couleur ) );
16 }
17
18 PointCouleur::PointCouleur( const char* c )
19 {
20         /* on garde un emplacement pour le caractere de fin de
21          * chaine \0. */
22         strncpy( couleur, c, sizeof( couleur ) - 1 );
23 }
24
25 PointCouleur::~PointCouleur( void )
26 {
27 }
28
29 void PointCouleur::affiche_pc( void )
30 {
31         printf( "la couleur est %s\n", couleur );
32 }
```

Listing 4.6 – pointcouleur.cpp

Grâce à l'héritage, la classe *PointCouleur* possède ses propres attributs et méthodes mais aussi ceux de *Emplacement*. Elle peut donc accéder aux attributs et méthodes publiques de la classe mère en plus des siens comme le montre le programme d'exemple 4.7

```
1
2  #include <stdio.h>
3  #include "pointcouleur.h"
4
5  int main( void )
6  {
7          PointCouleur pc( "jaune" );
8
9          /* methode de PointCouleur */
```

```
10          pc.affiche_pc();
11          /* PointCouleur n'a pas de methode init c'est donc
12           * la methode de la classe mere Emplacement qui sera
13           * appelee. */
14          pc.init( 10, 5.5 );
15          /* idem cette methode est celle de Emplacement */
16          pc.affiche();
17
18          return 0;
19 }
```

Listing 4.7 – Exemple classe PointCouleur

Toutefois dans le programme 4.7 l'objet *pc* ne peut pas modifier les attributs hérités de *Emplacement*, nous allons voir pourquoi dans la partie suivante.

4.5.1 Accessibilité des membres d'une classe

Portée protégé

En plus de la portée publique (accessible par tous) et de la portée privée (uniquement accessible par la classe elle même), il existe une portée intermédiaire : la portée protégé, *protected*. Un attribut, ou une méthode, protégé est accessible uniquement par la classe et celles qui en dérivent, soit par la classe mère et ses filles.

Dans l'exemple précédent, 4.6, la méthode *affiche_pc* affiche la couleur seulement, on pourrait aussi afficher les coordonnées *abscisse* et *ordonnee* de la classe Emplacement. Si vous changer le code pour afficher ses deux variables vous aurez une erreur lors de la compilation car tout ce qui est déclaré privé n'est pas hérité. Dans la classe *Emplacement*, *abscisse* et *ordonnee* sont déclarées dans la zone privée, aucune classe fille ne pourra y accéder. Si vous remplacez *private* par *protected* dans la classe *Emplacement*, alors la classe *PointCouleur* pourra y accéder et la méthode *affiche_pc* pourrait être :

```
void PointCouleur::affiche_pc( void )
{
   printf( "{%f, %f}\n", abscisse, ordonnee );
   printf( "la couleur est %s\n", couleur );
}
```

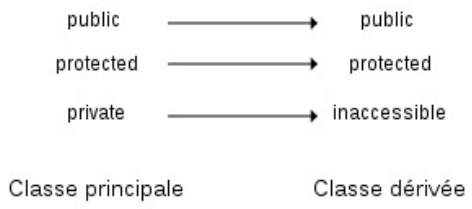


FIGURE 4.1 – Héritage public

Grâce à *protected* on peut garder une portée limitée, accès interdit à l'extérieur de la classe et autorisé en interne et par héritage. Encore que ceci dépend de la portée de l'héritage.

Héritage private, protected, public

En choisissant le portée de l'héritage (*private*, *protected*, *public*) on détermine l'accessibilité dans la classe dérivée des données membres et méthodes de la classe mère.

Héritage public c'est le plus simple, il est illustré par la figure 4.1, seules les membres privés de la classe mère changent de portée en devenant inaccessibles pour la fille. C'est l'héritage le plus fréquemment utilisé, il est utilisé dans le listing 4.5.

Héritage protected illustré par la figure 4.2, les membres publiques de la classe mère deviennent protégés pour la classe fille, les membres privés restent inaccessibles.

Héritage private illustré par la figure 4.3, les membres publiques, et protégés deviennent privés, les membres privés restent inaccessibles.

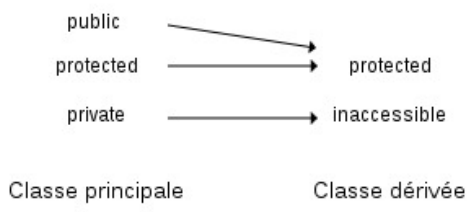


FIGURE 4.2 – Héritage protected

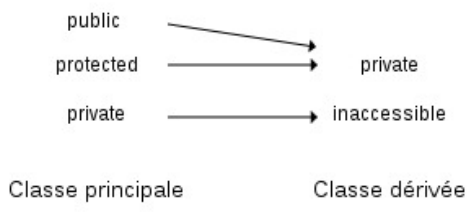


FIGURE 4.3 – Héritage private

4.5.2 Polymorphisme

Lorsqu'une méthode d'une classe dérivée a le même nom que celle de la classe mère, elle masque le méthode de la classe mère. On peut appeler la méthode de la classe mère en indiquant avant le nom de la méthode le nom de la classe mère suivi de deux fois deux points. Par exemple, reprenons la classe *PointCouleur*, renommons et changeons la méthode *affiche_pc* en *affiche* de cette manière :

```
void PointCouleur::affiche( void )
{
   printf( "la couleur est %s\n", couleur );
}
```

Dans le programme principale 4.7, changeons le main de cette manière :

```
int main( void )
{
   PointCouleur pc( "jaune" );
   pc.init( 10, 5.5 );

   pc.affiche();

   return 0;
}
```

Cette fois ce n'est plus la méthode *affiche* de *Emplacement* qui sera appelée mais la méthode *affiche* de *PointCouleur* car nous avons utilisé le polymorphisme. Ainsi nous pouvons faire en sorte que celle de *Emplacement* soit aussi appelée au modifiant légèrement le code de cette manière :

```
void PointCouleur::affiche( void )
{
   Emplacement::affiche();
   printf( "la couleur est %s\n", couleur );
}
```

Avec cette notion de *polymorphisme* un objet est capable de se spécifier en redéfinissant des méthodes de son parent de sorte à les rendre plus spécifiques.

4.6 Méthodes virtuelles

On dit d'une méthode qu'elle est virtuelle lorsque son but est d'être redéfinie dans les classes dérivée de la classe principale. Pour indiquer qu'une méthode est virtuelle on utilise le mot clé *virtual*.

4.6.1 Utilisation

Pour comprendre le concept des méthodes virtuelles prenons l'exemple d'une classe principale *Mammifere* et de trois classes dérivées, *Chien*, *Chat* et *Cochon*. Chaque classe contient une méthode *crier* qui affiche le cri de l'animal, un *Mammifere* n'a pas de cri en particulier cette méthode est destinée à être redéfinie dans les classes dérivées c'est pourquoi nous allons la déclarer en tant que virtuelle. Voici comment sont déclarées les classes :

```
class Mammifere
{
public:
   Mammifere( void );
   virtual void crier( void );
};

class Chien : public Mammifere
{
public:
   Chien( void );
   void crier( void );
};

class Chat : public Mammifere
{
public:
   Chat( void );
   void crier( void );
};

class Cochon : public Mammifere
{
public:
   Cochon( void );
   void crier( void );
```

```
};
```

Le mot clé *virtual* n'est pas répété dans les classes dérivées, il est seulement utilisé dans la classe principale. Voici maintenant l'implémentation de la méthode *crier* de chaque classe :

```
void Mammifere::crier( void ) { cout << "Oups !" << endl; }

void Chien::crier( void )     { cout << "Ouaf" << endl; }

void Chat::crier( void )      { cout << "Miaou" << endl; }

void Cochon::crier( void )    { cout << "Gruic" << endl; }
```

On peut déclarer un objet de chaque type et appeler la méthode *crier* de chacun, nous aurons l'affichage espéré, c'est-à-dire le cri de l'animal. Nous pouvons aller plus loin, c'est d'ailleurs le cas d'utilisation classique. Nous ne savons dans notre programme de quel animal nous aurons besoin, on peut créer un objet de chaque type et utiliser le bon le moment vu mais cette approche est complexe, il y a plus simple, en utilisant les pointeurs et la classe *Mammifere* comme dans le programme ci-dessous :

```
typedef enum _animal_
{
   e_chien = 1,
   e_chat,
   e_cochon,
} e_animal;

int main( void )
{
   int type = 0;
   Mammifere *animal = NULL;

   cout << "Quel cri souhaitez-vous ?" << endl;
   cout << "1 - chien" << endl;
   cout << "2 - chat" << endl;
   cout << "3 - cochon" << endl;
   cin >> type;

   switch( type )
```

```
    {
       case e_chien:
          animal = new Chien(); break;

       case e_chat:
          animal = new Chat(); break;

       case e_cochon:
          animal = new Cochon(); break;

       default:
          animal = new Mammifere(); break;
    }

    /* animal est un pointeur on utilise une fleche pour acceder
     * aux elements */
    animal->crier();

    return 0;
}
```

Dans un premier temps on créel un objet de classe *Mammifere*, classe générale de base, puis lorsque l'on connaît le type final, on instancie un objet du type souhaité.

4.6.2 Méthode virtuelle pure

Dans l'exemple précédent on remarque qu'un objet de la classe *Mammifere* n'est pas utile en soit car il n'y a pas de cri associé. La méthode *crier* de *Mammifere* n'a aucun intérêt d'être implémentée dans cette classe, ainsi on peut créer une méthode virtuelle pure, qui ne possède pas d'implémentation dans la classe de base mais uniquement dans ses dérivées. Pour cela on déclare la méthode nulle comme ceci :

```
class Mammifere
{
public:
   Mammifere( void );
   virtual void crier( void ) = 0; /* methode virtuelle pure */
};
```

On dit d'une classe possédant au moins une méthode virtuelle pure qu'elle est abstraite. Instancier un objet d'une classe abstraite n'a aucun sens et est interdit. Si on reprend l'exemple précédent on peut corriger le programme principal ainsi :

```
typedef enum _animal_
{
   e_chien = 1,
   e_chat,
   e_cochon,
} e_animal;

int main( void )
{
   int type = 0;
   Mammifere *animal = NULL;

   cout << "Quel cri souhaitez-vous ?" << endl;
   cout << "1 - chien" << endl;
   cout << "2 - chat" << endl;
   cout << "3 - cochon" << endl;
   cin >> type;

   switch( type )
   {
      case e_chien:
         animal = new Chien(); break;

      case e_chat:
         animal = new Chat(); break;

      case e_cochon:
         animal = new Cochon(); break;

      default:
         break;
   }

   if( NULL != animal ) { animal->crier(); }
   else { cout << "erreur !" << endl; }
```

```
    return 0;
}
```

Le plus souvent on utilise des classes abstraites pour créer des interfaces où l'implémentation sera faite dans les classes dérivées, l'interface servant de guide à l'implémentation.

Chapitre 5

Gestion de version

Dans ce chapitre nous allons voir un moyen de *stocker* du code de façon pérenne, avec un historique, afin de pouvoir revenir en arrière en cas de problèmes, ou encore travailler en parallèle. Mais surtout nous allons voir comment *versionner* du code, tout ceci grâce à un gestionnaire de version nommé SVN (Subversion).

5.1 Intérêt

Pourquoi utiliser un gestionnaire de version et quel est son but ?

Imaginez que vous soyez en train de développer seul un projet conséquent. Au fur et à mesure de vos développements vous compilez et testez votre application, naturellement vous trouvez des problèmes et faites des corrections. Bien entendu tout le code source de votre programme est sur votre machine, vous faites une mauvaise manipulation ou votre machine plante et vous perdez tout votre programme. Avez vous fait une copie de sauvegarde de votre projet quelque part ?

Vous travaillez toujours seul sur votre projet, après avoir corrigé plusieurs problèmes vous obtenez une version stable mais incomplète. Vous continuez vos développement et avez de nouveau des problèmes, en fin de compte la version d'avant était bien mieux et tout ce que vous avez ajouté n'est pas bon. Comment revenir à la version stable ? Avez vous pensé à l'archiver quelque part ?

Bref, en utilisant un gestionnaire de version vous pouvez éviter les problèmes précédents. Son intérêt est de stocker, centraliser, un ensemble de fichiers en conservant l'historique des modifications. Ainsi on peut à tout instant revenir à la version précédente d'un fichier, voire d'un ensemble de fichiers. Généralement le gestionnaire de version est installé sur un serveur, ce qui permet également de travailler en équipe sur un même projet et de centraliser le tout sur le serveur.

Il existe plusieurs logiciels pour gérer les versions, CVS, l'ancêtre de SVN, TFS le système de Microsoft intégré à Visual Studio, ou encore Git plus récent et proposé par Linux Torvalds. J'ai choisi SVN car il est encore très largement utilisé dans les entreprises et projets Open Source.

5.2 Subversion

Subversion, ou SVN, fonctionne sous forme de client serveur, les fichiers sont stockés sur un serveur, local ou distant, où tourne SVN. On y accède grâce à un client qui propose plusieurs commandes pour effectuer des actions propres à SVN permettant de stocker votre code, le récupérer, ou encore le versionner.

5.2.1 Fonctionnement

SVN est capable de gérer plusieurs projets, et on parle alors de *repository* ou dépôt. Un repository est un espace nommé qui va contenir les sources d'un (ou plusieurs) projet, par exemple si vous vouliez utiliser SVN pour archiver les sources d'un devoir pendant que vous travailliez dessus, vous auriez sûrement nommé votre repository *devoir03* pour le devoir 3. Sinon vous pouviez également créer un repository *devoir* et créer différents projets pour chaque devoir comme vous feriez localement avec votre système de fichiers.

Le repository est donc l'emplacement où sont stockés les fichiers du (ou des) projet. Par convention à la racine de chaque projet du repository on crée trois répertoires nommés *trunk*, *branches* et *tags*. Si on reprend l'exemple précédent avec le repository *devoir*, on aurait l'arborescence suivante :

```
/devoir
   |
   |---/devoir01
```

```
|       |
|       |---/trunk
|       |---/branches
|       |---/tags
|---/devoir02
|       |
|       |---/trunk
|       |---/branches
|       |---/tags
|---/devoir03
|       |
|       |---/trunk
|       |---/branches
|       |---/tags
```

trunk (tronc) ici se trouvent les fichiers sources qui évoluent au cours du projet, c'est en quelque sorte les développements en cours, ou officiels.

branches ici se trouvent des branches du tronc, des développements en parallèle de celui-ci. On crée des branches principalement lorsque les développements sont trop éloignés des développements officiels, par exemple pour une fonctionnalité spécifique à un client. Certains développeurs utilisent des branches également pour travailler dans leur coin sans être gênés par l'évolution du tronc puis, une fois leur développement stable, réintègre leur branche dans le tronc. A terme une branche n'est pas vouée à persister, il vaut mieux soit la tuer, soit l'intégrer dans le tronc.

tags ici se trouvent des versions livrées (release) à un moment où le tronc est stable et doit être livré, lorsque l'on *tag*, SVN fait une copie de la version actuelle du tronc vers le répertoire tags dans un sous-répertoire portant un nom que l'on spécifie.

Nous venons de voir comment est généralement architecturé un repository SVN, voyons désormais les commandes de base pour interagir avec un repository SVN.

5.2.2 Commandes de base

Je vais expliquer ici les actions de base lorsque l'on utilise SVN, à savoir *checkout*, *commit*, *update* et *revert* qui peut être très utile.

checkout est sans doute la première commande que vous allez utiliser, elle permet tout simplement de récupérer les sources d'un repository. Imaginons que je veuille travailler sur le projet *devoir03* du serveur *www.exemplesvn.com/svn/*. L'url à utiliser sera *www.exemplesvn.com/svn/devoir/devoir03/trunk*. Je veux les sources du tronc pour travailler sur le code en cours, mais en changeant simplement l'adresse on peut récupérer ce que l'on veut. Après l'exécution de la commande, j'aurai sur ma machine une copie du contenu du répertoire *trunk* du repository *devoir03*.

commit permet de mettre à jour le repository avec les modifications faites sur votre machine. L'action *commit* est toujours accompagnée d'un message qui décrit en quoi concernent les modifications de manière à avoir un historique détaillé des modifications. La mise à jour n'écrase pas le code du serveur, SVN va garder l'historique des modifications en attribuant à chaque fichier un numéro appelé révision. A tout instant on peut revenir en arrière grâce à cette révision.

update permet de mettre à jour le code local sur la machine avec les dernières modifications du serveur. Si quelqu'un travaille sur le même repository en même temps que vous, avec cette commande vous allez synchroniser votre code local avec le serveur et récupérer toutes les dernières versions de fichiers.

revert permet de revenir à la version d'origine d'un fichier modifié localement. Toutes les modifications seront perdus et on retrouve le fichier d'origine.

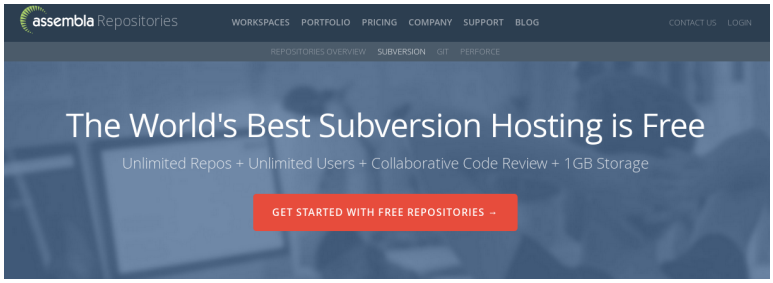
Pour mieux comprendre le fonctionnement il faut passer à la pratique.

5.3 La base en pratique

Nous allons voir en pratique la base pour travailler avec SVN, mais pour cela nous devons avoir un serveur SVN pour stocker notre code, et plus tard le récupérer, et un client SVN.

5.3.1 Serveur

Il est possible d'installer un serveur sur sa machine, ou sur une machine tiers, cependant pour gagner du temps je vous propose d'utiliser des serveurs



Assembla is the best Subversion host based on reliability, collaborative features, and innovation. Sign up to see why over one million users have trusted Assembla for their coding needs.

FIGURE 5.1 – Assembla - page d'accueil

gratuits en ligne comme le propose *assembla*. Rendez-vous à l'adresse `https://www.assembla.com/subversion/` et suivez les étapes ci-dessous :

- Cliquez sur le bouton "GET STARTED WITH FREE REPOSITORIES" (fig. 5.1).
- Renseignez les trois champs et cliquez sur le bouton "CREATE ACCOUNT" (fig. 5.2)
- Une fenêtre apparaît, indiquez votre prénom et nom puis cliquez sur le bouton "Next" (fig. 5.3)
- Choisissez un *repository* de type Subversion (choix par défaut) puis cliquez sur "Next" (fig. 5.4)
- Donnez un nom à votre *repository*, par exemple *democtu*, puis cliquez sur "Next".
- Si tout est ok vous devriez avoir le message *Done ! You can start working on your repository now.*. Cliquez sur "Got To My Repository" (fig. 5.5).

Vous arrivez désormais sur la page de gestion de votre repository (fig. 5.6) où se situent plusieurs informations utiles, tout d'abord on remarque l'architecture *trunk branches tags* créée automatiquement et l'url de votre repository, à côté du champ *Checkout URL*, dans mon cas *https ://subversion.assembla.com/svn/democtu/*. Côté serveur il n'y a plus rien à faire on peut désormais passer au client.

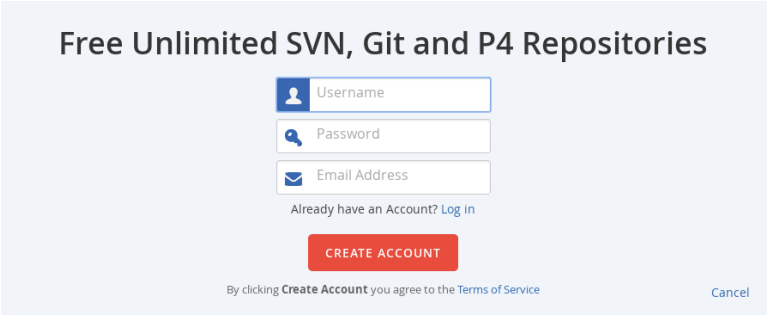


FIGURE 5.2 – Assembla - création compte

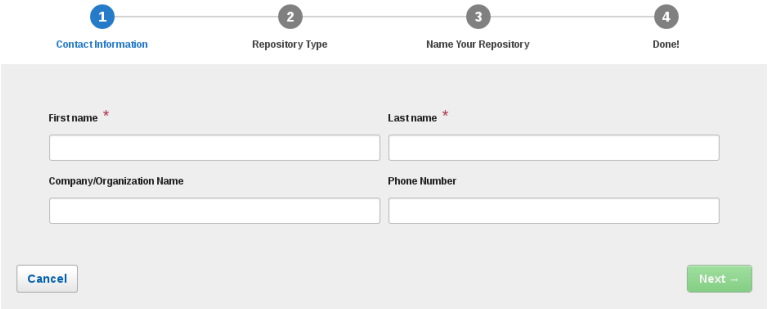


FIGURE 5.3 – Assembla - informations

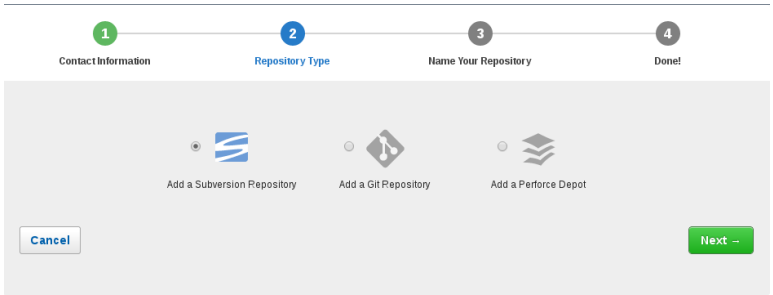


FIGURE 5.4 – Assembla - type de dépôt

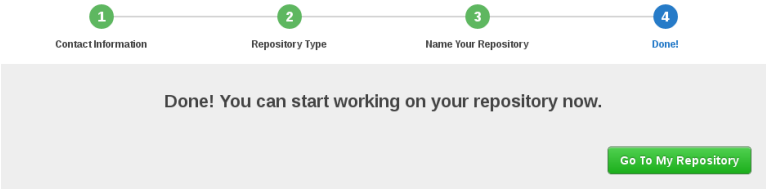


FIGURE 5.5 – Assembla - done

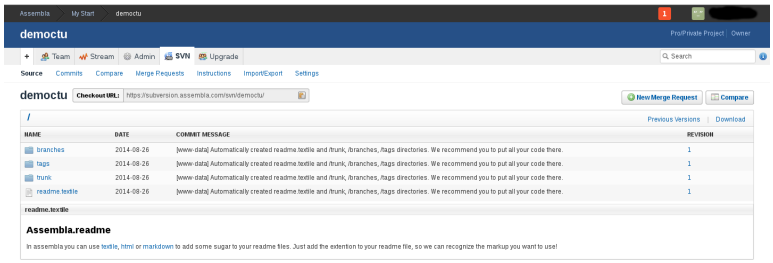


FIGURE 5.6 – Assembla - accueil dépôt

5.3.2 Client

Première étape, installer un client, et naturellement il en existe plusieurs, seule l'interface en ligne de commande et commune entre les différentes plateformes.

Pour Windows il existe plusieurs clients en ligne de commande pour Windows, je vous conseille d'installer *SlikSVN* disponible ici `https://sliksvn.com/download/`. Après l'installation SVN sera automatiquement ajouté au PATH de votre environnement. Ouvrez une console et tapez la commande *svn*. Si l'installation a fonctionné vous aurez le message *Entrer 'svn help' pour l'aide*.

Pour Linux vous pouvez installer SVN simplement via le gestionnaire de paquets de votre distribution (sudo yum install svn).

Checkout

Le projet à été créé côté serveur précédemment, avant d'écrire du code nous allons redescendre le repository du projet sur notre machine de façon à avoir un répertoire de travail en local, synchronisé avec le serveur SVN.

Pour cela il nous faut l'adresse du repository, dans mon exemple précédent, l'adresse est *https ://subversion.assembla.com/svn/democtu/*. Cette adresse est la racine du repository, comme nous allons travailler sur le tronc, autant se synchroniser directement dessus et utiliser l'adresse *https ://subversion.assembla.com/svn/democtu/**trunk***.

Dans un terminal, tapez la commande :

```
svn checkout https://subversion.assembla.com/svn/democtu/
    trunk democtu
```

Comme c'est la première fois, le client vous demandera de vous authentifier, utilisez les informations entrées lors de la création de votre compte sur *assembla*. A la fin de l'opération vous avez un répertoire *democtu*, dans mon cas, synchronisé avec le *trunk* de votre repository et ne contenant aucun fichier, mis à part un répertoire caché nommé *.svn*. Voilà notre machine est prête pour travailler sur notre projet.

Commit

Créons un fichier dans notre répertoire *democtu*, *helloworld.c* contenant un peu de code tout de même, comme par exemple :

```
#include <stdio.h>

int main( void )
{
   printf( "Hello World !\n" );
   return 0;
}
```

Ce fichier récemment créé n'existe pas encore sur le serveur SVN, il est non versionné, nous allons justement le *commiter* soit l'envoyer sur le serveur.

Dans un terminal, à la racine de votre dossier de travail, tapez la commande :

```
svn status
```

Cette commande permet d'avoir un status des fichiers de votre copie locale. Vous aurez en réponse :

```
?       helloworld.c
```

Le ? signifie que le fichier est inconnu sur repository, en effet nous venons de le créer, avant de pouvoir faire un commit il faut l'ajouter comme nouvel élément en tapant :

```
svn add helloworld.c
```

Si vous afficher à nouveau le status des fichiers vous verrez que cette fois le fichier est précédé d'un A qui signifie que le fichier est nouvellement ajouté (Added) et il est désormais possible de le *commiter*. Pour la commande *commit* on peut soit préciser un fichier et ce sera le seul à être envoyé sur le serveur, soit ne pas préciser de fichier et SVN enverra tous les fichiers pouvant être *commités*, soit les fichiers nouvellement ajoutés et ceux existant déjà sur le serveur mais modifiés localement.

```
svn commit -m "premiere version"
```

En réponse au *commit*, vous devriez avoir :

```
Adding         helloworld.c
Transmitting file data .
Committed revision 2.
```

SVN a automatiquement modifier la révision du fichier, ici 2. On peut voir l'historique des *commit* en tapant :

```
svn log
```

On ne voit pas notre message, ni même la révision 2, car nous ne sommes plus synchronisé avec le serveur, il faut se mettre à jour.

Update

Pour se mettre à jour, rien de plus simple... ou presque. Tapez la commande :

```
svn update
```

SVN affichera alors les fichiers mis à jours, et la révision à laquelle vous vous trouvez, normalement celle de votre dernier commit. Cette fois en tapant :

```
svn log
```

Vous verrez, pour chaque commit, la révision qui a été donnée, le message, l'utilisateur qui a fait le commit, et la date et l'heure.

Si vous avez des modifications dans un fichier et que vous faites un update, si côté serveur, ce même fichier à évoluer entre temps (commit d'un collègue), SVN va s'efforcer d'intégrer les modifications dans votre copie locale sans écraser vos modifications. Mais dans certains cas il peut y avoir conflit, si les modifications sont au même emplacement dans le code et que SVN est incapable de le faire seul. Dans ce cas il vous votre avis, garder que vos modification, que celles du serveur, ou éditer le fichier et faire le travail d'intégration vous même.

Revert

C'est une commande très utile lorsque l'on commet une erreur en local et que l'on veut revenir à la version d'origine d'un fichier, ou d'un ensemble. Admettons, vous avez un fichier source de plusieurs centaines de lignes de codes, vous traquez un bug et en tentant de le corriger vous ajoutez des traces, modifiez un peu le code, faites le ménage, pour finalement vous rendre compte que le bug en question est ailleurs et vous le corrigez. Le fichier sur lequel vous avez passé des heures à le *détruire* n'est plus du tout semblable à l'original. Plutôt que de faire des *ctrl+z (ou undo)* en cascade pour revenir en arrière, SVN vous propose la commande *revert*.

Nous ne possédons qu'un petit fichier mais il suffira amplement pour servir d'exemple. Modifions le un peu, par exemple :

```
#include <stdio.h>

int main( void )
{
   int i = 0;

   while( i < 10 )
   {
      printf( "Hello World !\n" );
      i++
   }
   return 0;
}
```

Après modification, en tapant la commande pour vérifier le status SVN de votre copie locale vous verrez que *helloworld.c* apparaît avec un M en tête

pour dire que le fichier à été modifié par rapport à l'original, avant de faire un *revert* nous allons voir quelles sont les modifications. Tapez la commande :

```
svn diff helloworld.c
```

Vous verrez alors des parties de codes avec des signes + et -, + signalant des ajouts et - des retraits. Pour revenir à la version originale tapez :

```
svn revert helloworld.c
```

Si vous demandez à nouveau le status SVN de votre copie locale vous ne verrez plus de différences.

Nous avons désormais vu les principales commandes de SVN sur un exemple très simple. La partie suivante va un peu plus loin et présente la création de branche de tag.

5.3.3 Plus loin dans la pratique

Notre programme d'exemple permet d'afficher le message *Hello World !* et c'est ce qui était demandé par le client, très bien, dans ce cas taguons le pour garder cette version dans un coin de notre repository.

Tags

Les tags sont une copie nommée du trunk à une révision donnée. C'est comme si vous copiez tout ce qu'il y a à la racine du trunk, dans un sous répertoire de /tag portant le nom de votre version. Notre projet est stable, et doit être livré, créer un tag par livraison est une bonne pratique à adopter. Pour créer un tag de la révision courante du trunk il faut taper la commande suivante :

```
svn copy https://subversion.assembla.com/svn/democtu/trunk
    https://subversion.assembla.com/svn/democtu/tags/0.1
```

Cette action aura pour effet de recopier le trunk vers le répertoire tags/0.1. J'ai choisi de nommer le répertoire avec le numéro de version mais tout est possible. On peut également taguer le code local plutôt que le trunk mais ce n'est pas conseillé car on perd le synchronisme des tags avec le trunk si on n'avait en local des différences importantes avec le trunk. Toutefois c'est possible en spécifiant le répertoire local comme source plutôt que le trunk, comme ceci :

```
svn copy . https://subversion.assembla.com/svn/democtu/tags
    /0.1
```

Branches

Les branches se créent de la même manière que les tags, c'est une copie d'un répertoire vers un sous répertoire de /branches. On créé des branches pour développer une fonctionnalité bien spécifique qui sort du cadre des spécifications du projet. A terme les branches sont soit intégrées dans le tronc, soit détruites ou oubliées (le développement n'est plus maintenu).

Imaginons que nous devions répondre à une demande client bien spécifique pour notre programme, créons une branche comme ceci :

```
svn copy https://subversion.assembla.com/svn/democtu/trunk
    https://subversion.assembla.com/svn/democtu/branches/
    branch.essai
```

La branche est créée, pour travailler dessus il faut faire un checkout de cette dernière :

```
svn checkout https://subversion.assembla.com/svn/democtu/
    branches/branch.essai branch.essai
```

Et voilà ! on peut faire des modifications sur nos fichiers, les commiter, soit toutes les opérations standards que SVN permet.

5.3.4 Clients graphiques

En dehors de la ligne de commande qui permet de faire toutes les opérations SVN possibles mais reste un peu rustique, il existe des clients graphiques plus élaborés comme *TortoiseSVN* pour Windows et *RabbitVCS* pour Linux. Tous les deux s'intègre au menu contextuel de l'explorateur de fichiers ce qui permet de réaliser toutes les opérations de base de SVN avec le clic droit.