

Cours de C Industriel

3ème année POLYTECH NANCY / ESSTIN

Thibaut MOREL-DAMY

Semestre 01

Contenu du cours : 12h de CM, 12h de TD et 12h de TP

Sommaire

Partie 1 : les généralités sur l'informatique	4
1.1/ Un ordinateur c'est quoi ?	4
1.2/ Un langage de programmation c'est quoi ?.....	4
1.3/ Le niveau d'un langage c'est quoi ?	5
1.4/ Langage compilé et langage interprété :	5
1.5/ Les différentes unités d'un langage :	6
1.6/ L'exemple qui nous intéresse : Le langage C :	7
1.7/ Un compilateur c'est quoi ?	8
1.7.1/ Compiler un programme.....	8
1.7.2/ L'architecture d'un microprocesseur.....	9
Partie 2 : les outils que nous utiliserons.....	12
2.1/ Ecrire son code.....	12
2.2/ Compiler son code.	12
2.3/ Compiler son code ... plus simplement !.....	12
Partie 3 : Le langage C	13
3.1/ Le langage C c'est quoi ?	13
3.2/ Les fichiers .c et .h.....	13
3.2.1/ Définition et implémentation.	13
3.2.2/ La signature d'une fonction.	14
3.2.3/ La portée d'une variable.	15
3.2.4/ Les includes , les defines et les macros.....	15
3.2.5/ Les fonctions en C.	19
3.2.6/ La fonction main.	19
3.3/ Les variables en C.....	20
3.3.1/ Déclarer et initialiser une variable.....	21
3.3.2/ Manipuler des nombres.....	23
3.3.3/ Manipuler des chaînes de caractères.	24
3.3.4/ Les tableaux.	27
3.3.5/ Les listes	29
3.3.6/ Le type void.....	29
3.3.6/ Regrouper des données	29
3.4/ Les pointeurs.....	32
3.4.1/ L'espace mémoire d'un programme.....	33

3.4.2/ La copie des variables.....	33
3.4.3/ Les pointeurs.....	35
3.4.4/ La gestion de la mémoire.....	41
3.4.5/ Les opérations sur la mémoire.	42
3.4.6/ Les fonctions qui renvoient des pointeurs.....	44
3.5/ Les opérateurs conditionnels.....	45
3.5.1/ Le if, else et else if.....	45
3.5.2/ Le switch	47
3.5.3/ Les opérateurs sur les booléens	48
3.6/ Les boucles.....	48
3.6.1/ La boucle for	48
3.6.2/ La boucle while	49
3.6.3/ La boucle do while	49
3.6.4/ Les mots clé continue et break	49

Partie 1 : les généralités sur l'informatique.

1.1/ Un ordinateur c'est quoi ?

Un ordinateur est une machine conçue pour traiter automatiquement des informations (cf l'étymologie du mot informatique). Sauf qu'un ordinateur est agnostique, c'est-à-dire que c'est une machine qui manipule des données tout en ne sachant pas ce que c'est. Le domaine de recherche de l'intelligence artificielle tend à gommer cette limitation, mais il est toujours nécessaire de dire à un ordinateur quelles données sont à traiter et comment il faut le faire. C'est le principe de la programmation : écrire un programme revient à dire à la machine comment les données que nous allons lui fournir doivent être traitées.

Pour réaliser ces opérations, un ordinateur est comme un être humain : il doit avoir une unité qui effectue tous les calculs nécessaires à la réalisation des opérations que nous lui demandons. Cette unité s'appelle le processeur. C'est en quelque sorte la partie du cerveau qui réalise les différentes opérations (les calculs en eux même) et qui met à disposition les résultats.

Le deuxième composant essentiel est la mémoire (appelée aussi RAM ou mémoire vive). Cette mémoire est un ajout technologique permettant au processeur de stocker les résultats de calculs intermédiaires qu'il effectue pour pouvoir les utiliser tout au long des processus qu'il réalise. Une instruction pouvant nécessiter plusieurs milliers de calculs, cette mémoire tampon est essentielle pour la bonne exécution des calculs. On peut noter qu'il serait possible d'utiliser la mémoire morte, à savoir le disque dur de l'ordinateur (ou une mémoire flash comme une clé USB) mais les temps d'accès à ses ressources sont très importants, et la rapidité d'exécution du programme s'en trouverait dégradée. Cependant cette utilisation de la mémoire morte est parfois mise en place dans un cas très particulier : si les programmes exécutés sur un ordinateur utilisent toute la mémoire vive, alors une partie de cette mémoire va être déchargée sur le disque dur. On appelle cela le swap. A noter que l'exécution des programmes est fortement ralentie par ce processus, et il provoque une détérioration du disque dur due à son utilisation intensive.

La structure de la mémoire est assez simple quand on ne rentre pas dans des considérations techniques. L'unité atomique (la plus petite unité disponible) est le bit. On peut voir un bit comme un interrupteur ou une lampe : s'il contient la valeur 1 alors cette lampe est allumée, s'il contient la valeur 0, alors la lampe est éteinte.

Pour plus de commodité mathématique, les bits sont regroupés par paquet de 8 pour former des octets. Un octet peut posséder une valeur allant de 0 (soit en représentation binaire 00000000) à 255 (soit en représentation binaire 11111111).

C'est le nombre d'octets qui permet de savoir la quantité de mémoire disponible. Un ordinateur possédant 16Go de RAM met donc à disposition 16 milliards d'octets pour les différents programmes que le processeur devra exécuter.

1.2/ Un langage de programmation c'est quoi ?

Comme nous l'avons vu au chapitre précédent un ordinateur est un ensemble de briques technologiques destiné à réaliser des opérations. Le rôle du développeur (et donc le votre) est de décrire précisément à l'ordinateur les instructions à réaliser pour que ce dernier puisse effectuer les opérations et fournir à l'utilisateur les résultats. Pour ce faire, les informaticiens ont créé des

langages de programmation utilisant des syntaxes standardisées. Cette notion de standard est très importante puisqu'elle permet de garantir à un développeur que la machine cible comprendra ce qu'il ou elle lui demande.

Remarque : le paragraphe précédent est un peu une profession de foi : dans la pratique, il arrive souvent que des différences importantes existent entre les matériels disponibles. Ces contraintes de portabilité peuvent être tellement importantes qu'elles peuvent à elles seules empêcher l'utilisation d'un langage. C'est pourquoi il est essentiel de connaître les limites des langages que vous pratiquez et ce afin de voir venir ce genre de problèmes.

Il existe une multitude de langages de programmation. Nous traiterons ici du langage C, développé dans les laboratoires BELL pour écrire le noyau UNIX (le noyau est un ensemble de programme utilisé pour effectuer toutes les opérations basiques nécessaires à un système d'exploitation) qui est un langage bas niveau (cf partie suivante).

Si vous ne deviez retenir qu'une définition, un langage de programmation est une langue possédant une syntaxe et un vocabulaire spécifique au travers desquels un développeur est capable d'interagir avec un ordinateur pour lui dire quoi faire.

1.3/ Le niveau d'un langage c'est quoi ?

Le niveau d'un langage est une notion permettant de situer un langage de programmation par rapport à sa position entre le développeur et la machine cible. En définitive, ce n'est rien de plus qu'un indicateur permettant de dire si un langage est plutôt proche de la langue de l'ordinateur (des 0 et des 1) ou bien de celle du développeur (dans notre grande famille de l'informatique, c'est l'anglais).

Le langage de programmation qui est juste au dessus des 0 et 1 utilisés par le processeur est appelé langage « assembleur » et est spécifique à la famille du processeur exécutant les programmes. Cette notion de famille de processeur sera développée plus tard.

1.4/ Langage compilé et langage interprété :

Il existe deux grandes familles de langages de programmation : les langages compilés et les langages interprétés. Pour classer un langage dans l'une ou l'autre de ces familles, il faut regarder comment le processeur accède aux instructions contenues dans le code écrit par le développeur.

Langage compilé : Les instructions sont directement traductibles pour l'ordinateur. Le résultat est un programme directement exécutable par le processeur.

Langage interprété : Les instructions dictées par le développeur passent dans un premier programme appelé interpréteur : c'est ce programme qui va traduire les instructions en quelque chose de compréhensible par le processeur. Il y a donc un ou plusieurs intermédiaires.

Ces deux familles de langages ont chacune des avantages et des inconvénients.

Les programmes compilés sont souvent plus rapide à l'exécution. Pour comprendre cette différence, il est possible de faire le rapprochement entre les différents intermédiaires entre le programme et le processeur et ceux présents dans une chaîne commerciale : si le temps d'exécution correspond au prix d'un produit, on peut comprendre que plus la vente d'un produit nécessite l'intervention

d'intermédiaires, plus le prix de ce produit va être élevé car tous ces intermédiaires vont prendre une marge. A contrario, si un client final va acheter son produit directement au producteur, alors il a toutes les chances de le payer à un tarif moindre. C'est exactement la même avec le temps d'exécution d'un programme : plus il y a d'intermédiaires en charge de la traduction des instructions entre le programme et le processeur, plus le temps nécessaire pour effectuer un calcul est élevé. Pour un programme nécessitant une grande manipulation de données, et donc des temps d'exécution importants, on préférera des langages compilés car ils sont souvent plus optimisés.

Remarque : Devant la progression incessante des puissances de calculs de nos ordinateurs, ces différences tendent à disparaître. Elles ne seront donc réellement impactantes que pour des applications très consommatrices de ressources.

A l'inverse, un des énormes avantages des programmes interprétés réside dans le fait qu'ils sont beaucoup plus portables que leurs cousins compilés. Un programme compilé est dépendant de la « langue » que le processeur comprend (on parle d'architecture). Tant qu'un interpréteur (le programme interprète) est disponible pour une architecture, alors il ne sera pas nécessaire de retravailler le code et de re-fabriquer le programme pour le cas d'un langage interprété. Par contre un langage compilé nécessitera que l'on compile le programme pour la nouvelle architecture cible.

Par exemple, le HTML est un langage interprété. Pour comprendre ce que le code dit, on utilise des navigateurs web (edge, mozilla, safari, ...) qui eux sont des programmes compilés : on n'utilise pas le même programme firefox sur un processeur Ivy bridge et un processeur ARM (deux architectures de processeur). Par contre, le code HTML lui ne change pas.

1.5/ Les différentes unités d'un langage :

La plupart des langages de programmation sont structurés. Cela signifie qu'il est possible pour le développeur de découper les instructions qu'il donne à la machine en sous-unités spécifiques permettant une meilleure présentation du code. Les principales unités de développement à connaître pour le C sont les suivantes :

Les variables : Elles sont le cœur du programme puisqu'elles contiennent les données que le programme doit manipuler.

Les constantes : Ce sont des variables particulières : leurs valeurs sont fixées. Elles sont utiles pour par exemple mettre à disposition des informations communes à l'ensemble du programme.

Les fonctions : Ce sont des conteneurs permettant de regrouper plusieurs instructions qui ont un but commun. Une fonction prend des variables en entrées (appelées arguments) et retourne une variable ou non en sortie. On peut les voir comme des boîtes noires effectuant des opérations. Quand on utilise une fonction, on ne se préoccupe pas de comment elle fait les choses, mais plutôt de ce qu'elle demande en entrée et ce qu'elle renvoie en sortie.

Les bibliothèques : Ce sont des ensembles de fonctions, constantes, variables, ... déjà développées qui permettent d'effectuer des tâches spécifiques et d'être réutilisées dans un programme. Par exemple si l'on souhaite faire une fonction effectuant des additions, il serait dommage d'avoir à la « coder » à chaque programme qui doit effectuer des additions. On met donc cette fonction dans une bibliothèque pour pouvoir la réutiliser dans d'autres programmes.

Ces structures ne sont pas exhaustives et dépendent largement du langage utilisé et de l'implémentation. Dans le cadre du C, ce sont les principales qui nous intéressent.

1.6/ L'exemple qui nous intéresse : Le langage C :

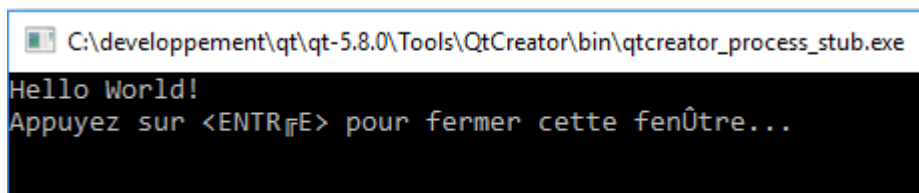
Nous attaquons le vif du sujet : le langage C. Pour le situer par rapport aux notions que nous avons vu précédemment, c'est un langage compilé plutôt bas niveau. Cela signifie principalement que le programme construit à partir du code du développeur est fabriqué pour une architecture spécifique (le C est un langage compilé) et que ce même code est plutôt proche du langage de la machine (le C est relativement bas niveau).

Voici l'exemple « hello world » du langage C :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

Ceci est presque le programme le plus simple qu'il soit possible de faire. Voici la sortie que l'on obtient :



```
C:\developpement\qt\qt-5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe
Hello World!
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Si nous nous intéressons lignes par lignes à ce que ce programme fait, voici ce que nous pouvons en dire :

#include <stdio.h> : C'est ce que l'on appelle une directive de préprocesseur. Ce fichier `stdio.h` (standard input output quand on dépile le nom) contient différentes fonctions du système d'exploitation dont nous aurons besoin (dans notre cas c'est la fonction `printf` appelée plus bas). Cette opération « include » permet de dire au compilateur que nous aurons besoin des fonctions présentes dans ce fichier `.h` et qu'il est donc normale qu'il ne trouve pas la définition de la méthode `printf` dans notre code.

int main(int argc, char *argv[]) {...} : Ceci est une fonction, plus spécifiquement nommée « main » qui renvoie un entier (un integer, c'est-à-dire un nombre entier abrégé en « int »). On remarque aussi la présence de deux paramètres passés à cette fonction (`int argc, char *argv[]`). Nous définirons plus tard à quoi cela correspond. Ce qu'il faut bien retenir c'est que cette fonction `main` est la plus importante de notre programme (et la seule pour l'instant). En effet, c'est le point d'entrée de tout programme écrit en C. Pour résumer, la première instruction qui sera exécutée est la première instruction contenue dans cette fonction.

Remarque : Une instruction est une opération utilisateur comprise entre le début d'une ligne et le caractère « ; ».

La première instruction de notre programme est donc bien `printf(...)`, et non pas la fonction `main`.

`printf("Hello World!\n");` : ceci est la première instruction de notre programme. Elle demande au processeur d'appeler la fonction `printf` (qui est fournie dans la bibliothèque standard du C). Nous donnons à cette fonction un argument qui est `"Hello World!\n"`, et c'est ce qui sera affiché en sortie de la console.

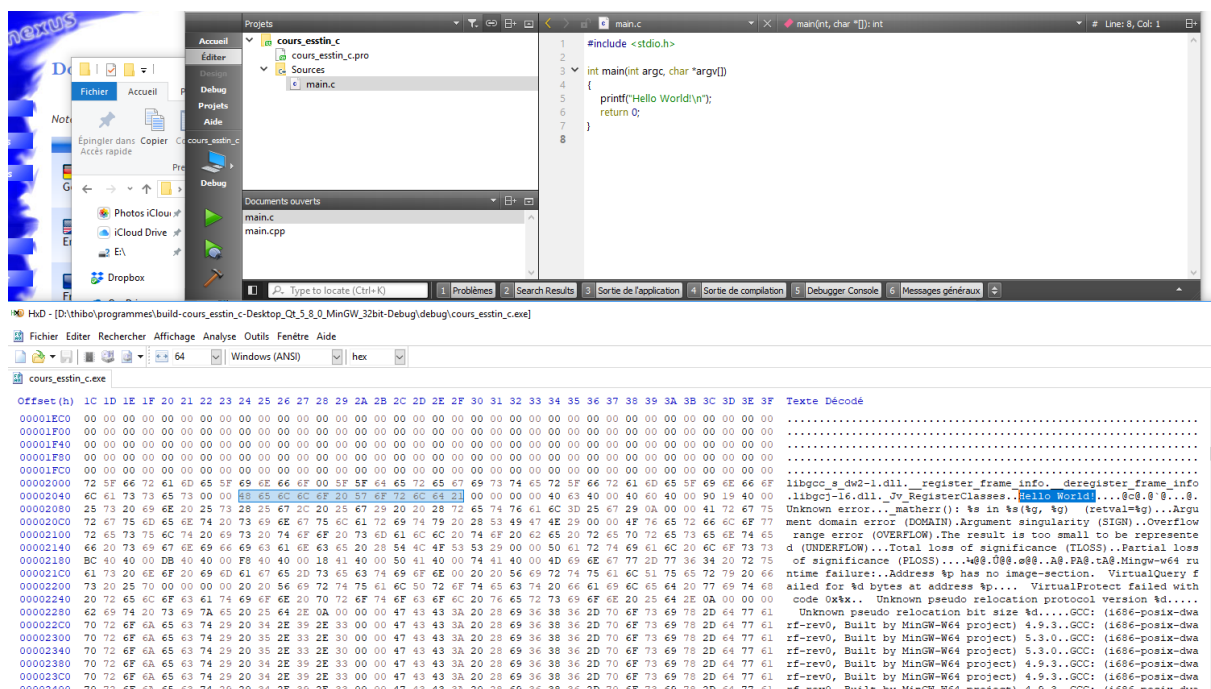
`return 0;` : Comme nous l'avons vu un peu plus en amont, notre fonction `main` doit renvoyer un nombre entier (symboliser par le « `int` » avant le nom de la fonction). Cet « `return` » est la dernière instruction de la fonction, c'est donc lui qui est en charge de retourner cet entier. Dans notre cas nous renvoyons 0 car tout s'est bien passé. Il aurait été possible de renvoyer certaines valeurs normalisées qui auraient indiquées à notre système d'exploitation que quelque chose a mal tourné durant l'exécution de notre programme. Pour le moment, la notion essentielle est l'utilisation du mot clé `return` pour transférer à l'appelant de notre fonction le résultat de l'exécution.

1.7/ Un compilateur c'est quoi ?

1.7.1/ Compiler un programme.

Un compilateur est un programme (encore un) qui est en charge de transformer un code utilisateur en code machine. C'est en quelque sorte un traducteur qui permet au développeur de ne pas avoir à écrire son programme directement en code machine (ce qui est réalisable mais fastidieux).

Si nous reprenons notre exemple précédent :



En haut nous avons ce que le développeur a entré, et en dessous ce qui est mis dans la mémoire pour que le processeur puisse l'exécuter. La différence d'informations est très importante. En effet, le développeur a entré 99 octets, et le compilateur a généré un exécutable (le programme à destination du processeur) pesant 49 152 octets. Rien qu'en temps de frappe, il est évident que cela serait beaucoup laborieux pour le développeur de coder pour la machine, d'autant que le programme

obtenu est difficilement compréhensible pour un être humain. C'est la raison pour laquelle les compilateurs ont été développés.

Remarque : En anglais, le mot compilateur devient compiler. Il est fréquent de trouver ce terme même dans des textes en français. En règle générale, compiler renvoie beaucoup plus de résultats de recherche que compilateur. Ce mot est donc à préférer donc si vous avez besoin d'avoir des informations sur ces programmes.

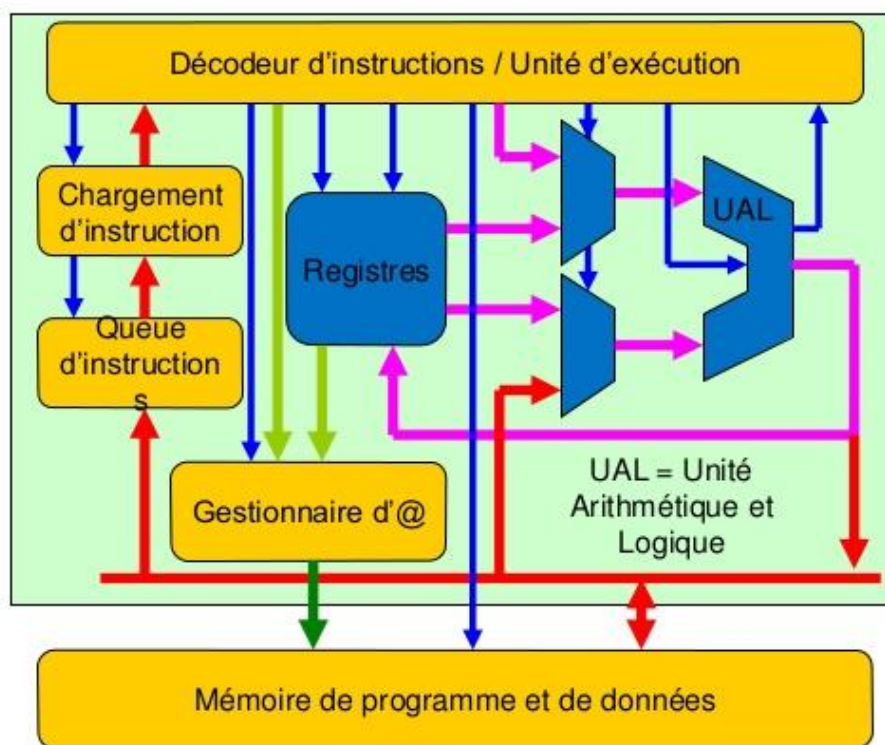
Il n'est pas possible d'évoquer un compilateur sans évoquer la notion d'architecture. Cependant, il est à noter que ce qui va suivre n'est qu'à titre purement informatif car cela ne concerne pas directement le langage C. Mais il est tout de même important de savoir comment cela fonctionne pour une compilation.

1.7.2/ L'architecture d'un microprocesseur.

Si il y a une invention qui est à l'origine de l'informatique moderne, c'est bien celle du transistor. Pour rappel, le transistor n'est rien qu'un commutateur qui permet de laisser passer ou non du courant. Jusque la pas de révolution extrême puisque nous savions le faire (un simple bouton suffisait). Les principales différences avec les interrupteurs mécaniques est que certaines entreprises ont trouvé un moyen de fabriquer des transistors en très grande quantité dans des espaces extrêmement réduits. Forts de cette découverte, ils se sont interrogés sur la possibilité d'assembler tous ces transistors en chaîne pour effectuer des opérations : le microprocesseur était né. Sa particularité : embarquer des primitives d'opérations extrêmement simples qu'il est possible d'appeler avec un programme.

Pour ce chapitre, nous allons simplifier à l'extrême un microprocesseur pour comprendre comment un enchainement de transistor est capable de faire des opérations mathématiques.

Structure interne d'un CPU



1 : structure d'un microprocesseur ([source](#) page 11).

On remarque qu'il y a beaucoup d'unités différentes (et encore, c'est un schéma très simplifié). Celle qui nous intéresse particulièrement est celle notée « Registres ». Il faut voir cela comme un ensemble de conteneurs dans lequel l'on va aller déposer des informations. Par exemple imaginons que nous souhaitons effectuer une addition. Dans la zone mémoire des registres (en définitive les registres sont un ensemble d'espaces mémoire) nous aurons deux registres permettant de stocker les deux nombres à additionner et une troisième zone qui contiendra le résultat.

Par exemple pour un programme suivant en C écrit par le développeur :

```
int additioner(int nombre01, int nombre02){
    return nombre01 + nombre02;
}
```

Le compilateur va repérer qu'il devra faire effectuer au microprocesseur une opération qu'il connaît, à savoir l'addition de deux nombres. Cependant, en lisant ces trois lignes, le processeur ne sera pas en mesure d'effectuer l'opération, il va donc falloir traduire cela pour lui. Le compilateur va donc lui préparer le travail en convertissant le programme en instructions processeur :

- Si les deux registres utilisés pour l'addition sont les registres A et B, je charge nombre01 dans le registre A et le nombre02 dans le registre B. J'ai donc mes deux nombres stockés en registre sous la forme de 0 et de 1.
- Le stockage effectué, je demande à l'unité arithmétique et logique (UAL) de déclencher le calcul.

- Une chaîne de transistor est câblée entre ses deux registres. Le résultat électrique de cette chaîne est une suite de 0 et de 1 qui est stockée dans le registre C (celui définit comme le registre de destination de l'opération d'addition).
- Il ne reste plus qu'au processeur qu'à aller lire le résultat dans le registre C. J'ai effectué une opération d'addition.

L'exemple précédent est relative simple. Mais il démontre bien comment à partir d'instructions processeur très basiques il est possible de faire pratiquement tout ce que le développeur demande au programme.

Les instructions de base telles que déplacer des données dans les registre, additionner les deux registres ensemble, déplacer le résultat dans le registre de résultat (le registre C dans notre exemple) sont standardisées : c'est l'architecture du processeur. Toutes les architectures ne possèdent pas les mêmes instructions basiques, c'est pourquoi il faut le compilateur qui correspond à l'architecture pour traduire un programme exécutable par le processeur. Les compilateurs sont un peu des traducteurs embarquant une langue spécifique. Si un processeur parle « allemand » et que vous utilisez un « traducteur français » alors il y a peu de chance que le processeur comprenne ce que vous lui demandez.

Vous connaissez surement deux types d'architectures processeurs : l'X86 et l'ARM.

L'X86 a été créée par Intel et AMD, les deux fabricants principaux. C'est celle des processeurs qui équipe la majorité des PC.

L'ARM est un standard édité par l'entreprise du même nom. Elle est la base de la majorité des microprocesseurs de vos smartphones.

Même si vous compilez un programme avec un compilateur X86, il ne sera pas exécutable sur votre téléphone puisque les instructions utilisées par le processeur de ce dernier ne sont pas celles que le compilateur a utilisées pour traduire.

Un point important à noter et qu'un compilateur possède aussi une adhérence avec le système d'exploitation pour lequel il est destiné. Cela provient des bibliothèques qui interviennent dans les fonctions de plus haut niveau. Si l'on prend l'exemple de la lecture d'un fichier, Windows, Mac OS et Linux ne vont pas faire les choses de la même façon et les bibliothèques utilisées ne seront pas réalisées de la même façon. Une lecture de fichier compilée pour un Linux tournant sur une architecture physique (un processeur) X86 ne sera pas réalisable sur un Windows en X86. Il y a donc une double dépendance du compilateur (architecture physique et système d'exploitation).

Remarque : Il est possible de compiler un programme pour une machine cible différente de la machine hôte (la machine qui contient le compilateur et qui fabrique le programme). Par exemple il est possible de compiler sous Linux (donc avec un système d'exploitation Linux et un processeur X86) pour faire un programme pour un téléphone portable (par exemple pour un Iphone tournant sous OS X). Cette opération est appelée une compilation croisée (ou cross compilation pour nos amis Anglais). Cependant, il est essentiel de bien maîtriser le langage et les étapes de compilation pour effectuer ce genre d'opération.

Remarque : Un compilateur DOIT toujours savoir ce qu'il manipule ! Cette notion est essentielle car elle est à l'origine de beaucoup d'erreur lorsque l'on compile un programme. Le typage des variables est un élément essentiel du C (si ce n'est le plus important). Nous reviendrons sur cet axiome dans la partie de ce cours consacrée à la mémoire.

Partie 2 : les outils que nous utiliserons

Pour fabriquer un programme C, il faut plusieurs logiciels relativement simples pour la plupart. Il est possible de trouver des suites incluant toutes ces fonctionnalités : ces logiciels sont appelés des environnements de développement intégrés (ou EDI pour les intimes). Un des plus connus est Eclipse pour les développeurs java, sachant que ce logiciel possède une suite pour développer en C et C++. Cependant, nous allons effectuer les opérations « à la main » afin de bien comprendre comment cela fonctionne.

2.1/ Ecrire son code.

Pour la base de la base il nous faut un éditeur de texte. Notepad de Windows suffirait, mais il est agréable d'avoir quelques fonctions supplémentaires telles que la coloration syntaxique ou encore des fonctions de recherche avancée. Il vous est possible d'utiliser n'importe quel éditeur de texte si vous avez des préférences. Notepad++ est très complet et permet de faire vraiment beaucoup de choses.

2.2/ Compiler son code.

Comme nous l'avons vu précédemment, il est nécessaire de compiler un programme C pour pouvoir l'exécuter. Il nous faut donc un compilateur. Il en existe plusieurs en fonction de ce que nous souhaitons faire (quel système d'exploitation cible et quel architecture de processeur).

Le plus connu concernant le C et C++ est GCC qui est un compilateur développé à la base pour Linux. Une version Windows existe et est appelée MinGW (minimal GCC for Windows).

Cependant ce compilateur est relativement lourd car très complet. Nous utiliserons donc plutôt un compilateur minimal, étant donné que nous n'aurons pas nécessairement besoin de toutes les fonctions très avancées proposées par GCC. Ce compilateur C s'appelle TCC (tiny C compiler). Il est de plus possible de l'installer en mode portable (c'est dire sur une clé USB par exemple). Ce programme est disponible [ici](#).

2.3/ Compiler son code ... plus simplement !

En théorie, il est possible de travailler avec seulement les deux programmes que je vous ai indiqués précédemment. Cependant cela peut se révéler très compliqué sur les gros projets, l'appel au compilateur s'effectuant en ligne de commande.

Pour palier à cette contrainte, un programme appelé MAKE peut être utilisé. Il permet à partir d'un script (appelé Makefile) de gérer toutes les étapes de la compilation.

Exemple d'un Makefile :

```

CC=tcc
CFLAGS=-W -Wall -pedantic
LIBS=
LIB=
OUTPUTDIR=%cd%\build
EXEC=hello

all: $(EXEC)

hello: hello/main.o
$(CC) hello/main.c -o hello/hello.exe
cd hello/ && del *.o && copy "hello.exe" "../build/hello.exe" && del hello.exe && cd ../

```

Ce fichier contient toutes les commandes nécessaires à la compilation de notre programme. Le code utilisé est disponible dans votre répertoire sous le dossier hello.

Pour plus d'informations sur les Makefile, je vous invite à lire cette page :

<https://ensiwiki.ensimag.fr/index.php?title=Makefile>

Partie 3 : Le langage C

3.1/ Le langage C c'est quoi ?

Le langage C n'est pas un langage orienté objet, c'est-à-dire que la notion de classe n'existe pas contrairement à d'autres langages (comme le java par exemple). Il existe des solutions pour faire du pseudo-objet en C mais je vous déconseille fortement de vous lancer dans cette entreprise, car cela peut être très compliqué à maintenir.

3.2/ Les fichiers .c et .h

Il existe deux extensions de fichier utilisées pour le langage C :

- Les .h qui contiennent toutes les choses que l'on souhaite mettre en commun (h signifie header et donc entête).
- Les .c (comme le nom du langage) qui contiennent le code exécutable.

La sous partie suivante explique pourquoi ces deux types de fichiers sont utilisés et comment les alimenter.

3.2.1/ Définition et implémentation.

Comme nous l'avons vu dans la partie 1, le langage C doit être compilé pour pouvoir être exécuté sur un ordinateur. Cependant la multitude des plateformes physique et logicielles (des processeurs et des systèmes d'exploitation) pose le problème de la compatibilité des programmes. En effet, comment, tout en considérant que l'on ne s'évitera pas une compilation par plateforme, faire en sorte que l'on puisse coder une bonne fois pour toutes un programme ? C'est là que la notion de définition et d'implémentation prend tout son sens.

La définition permet de fixer des règles de base pour utiliser un code. C'est à cela que servent les .h mentionnés précédemment : à dire comment une fonction doit être appelée, quels sont les

paramètres qu'elle prend, est-ce qu'elle renvoie un type de variable, si oui lequel, ... Ces fichiers peuvent aussi contenir des constantes utilisées par le programme. Plus généralement, ces fichiers vont contenir des choses communes qui permettent au compilateur de définir ce qu'il est en droit d'attendre des différentes instructions. Les fichiers .h ne sont jamais compilés en C.

L'implémentation est l'étape durant laquelle le développeur écrit spécialement ce que le programme fait. Ce code « opérationnel » est écrit dans les .c pour être par la suite compilé. Le compilateur sait faire le lien entre la définition des fonctions dans les .h et leurs implémentations dans les .c. D'ailleurs, une fonction non implémentée provoquera une erreur de compilation (si elle est appelée dans un .c mais qu'elle n'a jamais été écrite).

Le principal attrait de ce découpage est que lorsque l'on veut compiler son programme sur une autre plateforme, il n'est souvent pas nécessaire de changer les définitions (donc les .h). Par contre les .c sont différents puisqu'eux sont souvent dépendants de la plateforme (que cela soit au niveau du processeur ou du système d'exploitation). Globalement, ce découpage est très utilisé dans une multitude de langages pour permettre une meilleure portabilité. Un exemple très parlant est celui des bibliothèques que vous pourriez télécharger pour utiliser des fonctionnalités déjà codées : souvent une bibliothèque C est disponible sous la forme de dossiers compressés pour les différents systèmes d'exploitation (et parfois même pour les différentes architectures de processeurs). Il vous suffit de télécharger la bibliothèque qui correspond à votre plateforme cible pour compiler, et ce même si vous avez auparavant fait un programme pour une autre plateforme. Le .h sera le même, et donc vos appels de fonction seront les mêmes, et le portage en sera simplifié car il n'y aura pas besoin de retoucher votre code appelant.

3.2.2/ La signature d'une fonction.

J'ai évoqué dans la partie précédente le fait que le compilateur est capable de faire le lien tout seul entre la définition et l'implémentation d'une fonction. Pour effectuer cette opération, il se base sur la notion de signature d'une fonction. Il faut voir cette signature comme un identifiant unique.

Une fonction en C ressemble à cela :

```
type_variable nom_de_la_fonction(type_variable argument_1, type_variable argument_2)
```

On peut isoler deux choses importantes :

- **type_variable** : une fonction renvoie toujours quelque chose (même si ce n'est pas le cas, il y a un type pour le dire). Si la fonction a des arguments (les deux autres **type_variable**) alors ces arguments sont typés.
- **nom_de_la_fonction** : une fonction a toujours un nom.

La signature d'une fonction est composée comme suit :

- On prend en compte le type de retour (le premier type_variable).
- On prend en compte le nom de la fonction.
- On prend en compte le nombre d'arguments.
- On prend en compte le type des arguments.
- On prend en compte l'ordre des arguments (par rapport à leur type).

Par exemple la signature des fonctions qui suivent est la même :

```
int additionner(int nombre_01, int nombre_02)
int additionner(int nombre_un, int nombre_deux)
```

Par contre, si l'on considère la signature de cette fonction :

```
int additionner(int nombre_01, double nombre_02)
```

Toutes les signatures suivantes seront différentes :

```
double additionner(int nombre_01, int nombre_02) //le type de retour est différent
double additionner(double nombre_01, int nombre_02) //le type du premier paramètre est différent
int additionneR(int nombre_01, double nombre_02); //le nom de la fonction est différent
int additionner(int nombre_01, double nombre_02, int nombre_03); //le nombre d'arguments est différent
```

Cette notion de signature est essentielle en C car elle permet au développeur et au compilateur de s'y retrouver. Soyez très attentif quand vous écrivez vos programmes.

Les signatures de fonctions sont écrites dans les .h et les implémentations dans les .c.

3.2.3/ La portée d'une variable.

La portée d'une variable est une notion plutôt simple : elle permet de définir quelles parties de votre programme pourront utiliser les variables que vous déclarez.

Il existe trois types de variable ayant des utilisations bien spécifiques :

La variable locale : Cette variable est déclarée dans une fonction. Elle n'est utilisable qu'à l'intérieur de cette fonction. Quand le programme sort de la fonction, elle n'existe plus.

La variable globale : Il y a deux configurations pour ce type de variable. Soit elle est déclarée dans un .c et dans ce cas toutes les fonctions contenues dans le .c y auront accès, soit elle est déclarée dans un .h, ce qui autorisera tous les fichiers .h et .c incluant ce .h (je reviendrai sur cette notion d'include) à travailler avec.

La variable statique : C'est la plus dangereuse des variables. Sa portée est celle du programme, ce qui signifie que tout le monde peut l'utiliser. A manier avec délicatesse car ce genre de variable peut provoquer des comportements hasardeux du fait que trop de modifications à différents endroits d'un programme peut amener à ne plus connaître l'état de cette variable à un instant t.

3.2.4/ Les includes , les defines et les macros.

Ces trois notions sont des composantes importantes du langage C. Elles interviennent surtout sur la phase de compilation et servent à simplifier le travail du développeur et le programme de façon générale.

3.2.4.1/ Les includes

Les includes sont la base de l'utilisation de .h dans un programme C. C'est ce mot clé qui permet de dire au processeur que nous allons utiliser un .h.

Si nous reprenons l'exemple que nous avons utilisé précédemment :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

On remarque que la première ligne fait appel à une directive include. En fait, dans notre cas, nous demandons au compilateur d'aller lire le fichier stdio.h. Quand il passera sur la ligne printf, alors il connaîtra la fonction puisqu'il avait récupéré sa signature dans le fichier stdio.h. Sans cela, le compilateur nous aurait dit qu'il ne connaît pas la fonction et aurait provoqué une erreur de compilation. C'est principalement comme cela que les fichiers .h sont utilisés, au travers de cette directive include.

Remarque : le symbole # est essentiel. Il permet d'indiquer au compilateur qu'il faut aller chercher le fichier dont le nom suit avant d'effectuer les opérations de compilation.

En fait la compilation n'est pas une seule et même opération, mais un ensemble de tâches réalisées séquentiellement. Dans un premier temps, le compilateur va aller lire les fichiers .c. S'il trouve des includes, alors il va aller lire les fichiers .h qui sont indiqués. Une fois qu'il possède toute cette connaissance, il va essayer de fabriquer le programme. Cependant, quand il compile un .c, il ne connaît que les signatures des fonctions extérieures (rappelez-vous, les instructions sont contenues dans les .c et non dans les .h). Une fois tous les fichiers .c compilés, il y a une opération appelée assembly link (assemblage de liens). C'est durant cette opération que les fichiers .o (les fichiers .c compilés) sont parcourus pour essayer de recoller les signatures de fonctions appelées dans le code avec l'implémentation contenues dans les .c.

3.2.4.2/ Les defines

Il est parfois nécessaire de définir des constantes pour un programme C. Le premier réflexe que nous pourrions avoir serait de définir une variable globale static et constante pour réaliser cette opération. Or le C nous propose un autre mécanisme beaucoup mieux adapté à ce genre de problème : l'utilisation de la directive define. Regardons le code qui suit :

Nous avons créé un nouveau fichier qui s'appelle constantes.h. Ce fichier a la forme suivante :

```
#ifndef CONSTANTES_H
#define CONSTANTES_H

#define CONSTANTE_ENTIERE 3

#endif // CONSTANTES_H
```

Dans ce fichier, une variable CONSTANTE_ENTIERE est définie (à l'aide du mot clé define). Elle s'est vue attribuée la valeur 3. Après la définition de mon include, il m'est possible d'utiliser cette constante dans mon programme C. Nous avons notre nouveau fichier C :


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

int main(int argc, char *argv[])
{
    int a = 5;
    int b = 0;
    b = a + CONANTE_ENTIERE;
    return 0;
}
```

Ne programme ne fais rien à part réaliser une addition. Et encore, il ne traite pas le résultat. Le point réellement important est l'utilisation de cette variable CONANTE_ENTIERE. Après avoir réalisé l'include du fichier .h qui la contient (deuxième ligne), il m'est possible de l'utiliser telle quelle dans mon programme.

En fait durant la phase de pré-compilation (rappelez-vous, quand le compilateur exécute les directives de préprocesseur, va lire les .h, ...) mon compilateur va tomber sur cette valeur CONANTE_ENTIERE. Or, remarquant que cette dernière est définie dans constantes.h, il va aller chercher la valeur et l'utiliser pour remplacer la ligne. Bien que ne l'utilisateur ne le sache pas, notre fichier .c qui est envoyé à la compilation ressemble en fait à cela :

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

int main(int argc, char *argv[])
{
    int a = 5;
    int b = 0;
    b = a + 3;
    return 0;
}
```

C'est à toute la force des directives define : elles permettent de remplacer à la volée des valeurs dans le code utilisateur tout en ne définissant ces valeurs qu'une fois pour toute. On les utilise beaucoup pour définir des constantes dépendantes des plateformes cible (processeur et système d'exploitation) car en utilisant le bon .h, tout le programme est compilé de façon adaptée aux différentes plateformes.

Remarque : Bien que je vous ai dit que le langage C est un langage à typage fort, je n'ai pas déclaré de type pour ma variable CONANTE_ENTIERE. En fait un define est toujours de la forme :

```
#define nom_de_la_variable valeur_de_la_variable
```

C'est le compilateur qui attribue automatiquement un type à cette variable lors qu'il lit la valeur.

Remarque : Vous aurez sûrement remarqué dans le fichier constantes.h la présence d'un autre type de define sur la première ligne : #ifndef CONANTE_H.

En fait ceci est un define comme nous avons vu précédemment mais conditionnel. En gros cette ligne dit au processeur que si le fichier `CONSTANTES_H` n'est pas défini (`ifndef` signifie if not define) alors il doit aller à la deuxième ligne qui elle est une directive `define` du fichier `CONSTANTES_H`. Si le fichier est déjà défini (par exemple le compilateur l'a déjà lu car un fichier `.c` qu'il a lu précédemment l'utilisait) alors le compilateur le connaît déjà et il va directement à la dernière ligne (`endif` correspondant au `ifndef`). Cette mécanique est très importante pour éviter de tomber dans des problèmes de redondance cyclique.

3.2.4.3/ Les macros

Les macros sont un des éléments les plus dangereux en C. En effet, le principe de macro est basé sur les defines vues précédemment. Cela signifie concrètement que bien que ce soit le développeur qui les écrit, la majorité des opérations sont faites durant la compilation, phase où l'être humain n'a que très peu la main.

Exemple d'une macro affichant bonjour :

```
#ifndef CONSTANTES_H
#define CONSTANTES_H

#define CONSTANCE_ENTIERE 3

#define MACRO_AFFICHER printf("macro afficher !\n");

#endif // CONSTANTES_H
```

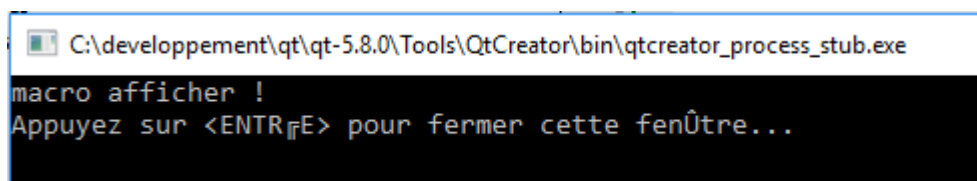
Et le `.c` qui appelle cette macro :

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

int main(int argc, char *argv[])
{
    MACRO_AFFICHER;
    return 0;
}
```

On peut remarquer que la macro est déclarée à l'aide d'un `define`. Cela signifie que son nom appelé dans le `.c` sera remplacé par la valeur de la macro contenue dans le `.h`.

Ce programme donnera le résultat suivant :



```
C:\developpement\qt\qt-5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe
macro afficher !
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Points essentiels pour l'utilisation des macros :

- Par convention leur nom est en majuscule.
- Elles sont écrites dans des .h.
- Il faut toujours pouvoir justifier de leur utilisation par rapport à une fonction.
- Certaines macros sont fournies par des bibliothèques externes et sont très utiles, notamment pour récupérer la ligne d'un fichier ou sont nom (par exemple pour créer des erreurs et faciliter la maintenance).
- Il est relativement difficile de trouver une erreur à l'intérieur d'une macro, ce qui ne facilite pas la maintenance du programme.

3.2.5/ Les fonctions en C.

Comme cela a été abordé dans le 3.2.2, les fonctions sont une composante essentielle du C. Elles permettent une bien meilleure visibilité lors des développements et surtout une factorisation. Cette notion de factorisation exprime le fait que si une suite de plusieurs instructions est utilisée dans plusieurs endroits d'un code, il est nécessaire de mettre ce bloc d'instructions dans une fonction et d'appeler cette dernière ou cela est nécessaire dans le code plutôt que de dupliquer le bloc d'instruction. Cela à plusieurs avantages :

- Si il y a une erreur dans le bloc d'instructions, une seule correction suffira (plutôt que de retrouver tous les blocs similaires dans le code et d'effectuer la même modification).
- Un programme C est un programme compilé, ce qui signifie qu'une fois que les instructions utilisateurs sont traduites en instructions processeur, elles sont chargées en mémoire. Dupliquer les mêmes instructions utilisateurs (et donc les mêmes instructions processeur) va donc conduire à une augmentation de la mémoire utilisée. C'est clairement une mauvaise chose pour les très gros programmes.
- Le découpage en fonction permet une meilleure lisibilité du code, on pense à ses camarades qui devront reprendre ce que l'on développe.
- Si je code mes fonctions dans des .c dédiés et que je mets les signatures de ces fonctions dans des .h séparé, je pourrais faire des bibliothèques utilisables par d'autres personnes.

Il est impossible de surcharger des fonctions, c'est-à-dire d'avoir le même nom pour plusieurs fonctions qui renvoient des types différents, ou qui n'ont pas le même nombre d'arguments, ou qui ont le même nombre d'arguments mais que ces derniers sont de types différents. Tout cela provient de la signature de la fonction (voir la partie dédiée).

Globalement les fonctions en C sont une des parties les plus importantes. Votre rôle de développeur consistera à trouver le découpage le plus optimisé possible pour respecter l'équilibre entre les différents points que nous avons évoqué précédemment.

3.2.6/ La fonction main.

Il y a une fonction particulière en langage C : c'est la fonction main. Comme évoqué plus haut, c'est le point d'entrée de notre programme. C'est la première fonction qui sera appelée lorsque l'exécution commencera.

Cette fonction retourne un nombre entier (un type int). En règle générale, on lui demande de retourner 0 car c'est ce qui va indiquer au système d'exploitation que l'exécution du programme n'a pas rencontré de problème. Il est sinon possible de retourner des entiers définis par les différents systèmes d'exploitation pour indiquer qu'une erreur particulière s'est produite. Ces valeurs sont

dépendantes du système d'exploitation, même si une certaine standardisation existe entre les différentes implémentations.

On remarque la présence de deux arguments dans la signature de la fonction. Ils constituent une mécanique permettant à l'utilisateur qui appelle le programme en ligne de commande d'interagir avec le programme. Voici un exemple de ce qu'il est possible de faire avec le système :

Considérant le code C suivant :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("message transmis au programme par l'utilisateur : %s\n",argv[1]);
    return 0;
}
```

Ce programme, une fois compilé peut être exécuté et donner le résultat suivant :

```
Microsoft Windows [version 10.0.17134.345]
(c) 2018 Microsoft Corporation. Tous droits réservés.

D:\thibo\programmes\build-cours_esstin_c-Desktop_Qt_5_8_0_MinGW_32bit-Debug\debug>cours_esstin_c.exe bonjour
message transmis au programme par l'utilisateur : bonjour

D:\thibo\programmes\build-cours_esstin_c-Desktop_Qt_5_8_0_MinGW_32bit-Debug\debug>cours_esstin_c.exe "bienvenu dans ce cours sur le C"
message transmis au programme par l'utilisateur : bienvenu dans ce cours sur le C

D:\thibo\programmes\build-cours_esstin_c-Desktop_Qt_5_8_0_MinGW_32bit-Debug\debug>cours_esstin_c.exe "nous avons donc reussit à transmettre des informations a notre programme"
message transmis au programme par l'utilisateur : nous avons donc reussit à transmettre des informations a notre programme
```

Le premier argument (argc de type int) sert à transmettre le nombre d'arguments qui va suivre (on ne connaît pas à l'avance ce que l'utilisateur va taper en lançant le programme). Le tableau d'argument (argv[] de type tableau de tableau de caractères) va être généré en découpant via les espaces les arguments entrés. C'est pour cela que si nous voulons afficher un message comportant des espaces, il est nécessaire de mettre notre argument entre guillemets (symbole ").

Si l'on souhaite que notre programme soit exécuté, il faudra nécessairement qu'il possède une fonction main, et seule le code qui sera appelé dans cette fonction (ou dans les sous fonctions appelées par la fonction main) sera exécuté.

Remarque : Il est possible de déclarer la fonction main comme retournant un type vide (void). Cependant, il y a clairement une perte d'information puisqu'il ne sera pas possible de caractériser les erreurs auprès du système d'exploitation.

3.3/ Les variables en C

Comme tous les langages de programmation, le langage C est fait pour automatiser les traitements de l'information. Pour ce faire, il est nécessaire de manipuler des informations. C'est dans ce cadre que le concept de variable prend tout son sens.

Le langage C embarque un certain nombre de types à travers des mots clé prédéfinis (on parle de types basiques). Il est cependant possible de créer ses propres types pour faciliter les manipulations de données que nous souhaitons réaliser.

3.3.1/ Déclarer et initialiser une variable.

Avant toute chose, il est nécessaire de connaître les manipulations qui permettent de faire connaître à notre programme les variables que nous souhaitons utiliser.

Cet apport de connaissance s'effectue auprès de deux opérations distinctes appelées déclaration et initialisation.

La déclaration :

Cette primitive permet de dire au programme que je souhaite utiliser une variable en précisant le nom de cette variable et le type qu'elle possède. Elle s'effectue de la façon suivante :

```
type_variable nom_variable ;
```

Cette déclaration permet à notre programme de savoir que nous allons utiliser une variable nommée `nom_variable` dans laquelle nous allons stocker des données de type `type_variable`.

Remarque : Le langage C est un langage typé ! Si vous oubliez le type de la variable, le compilateur n'en voudra pas et vous ne pourrez pas générer l'exécutable. Un nom de variable se déclare sans espaces à l'intérieur. Les caractères alpha numériques sont acceptés ainsi que `_`.

L'initialisation :

C'est une étape cruciale de la création d'une variable. Par contre, contrairement à la déclaration, si vous ne le faite pas, le compilateur n'y verra que du feu.

Une initialisation de variable est en fait une affectation de valeur juste après la déclaration de cette variable. On utilise l'opérateur d'affectation (le mot opérateur n'est pas anodin, nous en verrons d'autres par la suite) symbolisé par « = ». Cette opération s'effectue de la manière suivante :

```
nom_variable = valeur ;
```

C'est très simple dans la pratique mais ne pas le faire peut provoquer de graves défaillances logicielles. Explications :

Comme je l'ai indiqué précédemment, un programme est un ensemble d'instructions processeur chargées en mémoire. Considérant cela, il faut savoir que lorsque l'on demande au système d'exploitation d'exécuter notre programme, ce système va lui réserver de la mémoire, au regard des fonctions à charger et des variables utilisées. Pour ce faire, il calcul à peu près quelle quantité sera utilisée pour réaliser les opérations demandées. Or cette mémoire peut être une mémoire recyclée : si un autre programme a été exécuté avant et qu'il est terminé, alors le système d'exploitation peut tout à fait décider de réallouer cet espace mémoire à notre nouveau programme (cette opération est à l'entière discrétion du système d'exploitation). Oui mais voilà, la mémoire n'est pas nettoyée entre deux utilisations. Il est donc tout à fait possible que lorsque nous déclarons nos variables de programme, alors ces dernières possèdent déjà une valeur (inscrites par l'ancien programme). Cela signifie que nos variables sont dans des états de valeurs incohérents. Il est donc nécessaire d'effectuer la phase d'initialisation pour être certain de ce qu'elles contiennent (et donc que notre programme réagira comme nous l'avons demandé).

Ce petit exemple illustre ce problème de recyclage de la mémoire :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int variable_type_entiere;

    printf("valeur de ma variable de type entier : %d\n", variable_type_entiere); //on affiche la
    valeur de notre variable

    if(variable_type_entiere == 0){ //si ma variable vaut 0
        printf("je fais le traitement prevu dans le cas ou ma variable etait a zero\n");
    } else { //sinon
        printf("ma variable ne vaut pas zero, peut etre que je ne l'ai pas initialisee !\n");
    }

    return 0;
}
```

Le résultat de la lecture de cette variable non initialisée :

```
valeur de ma variable de type entier : 101
ma variable ne vaut pas zero, peut etre que je ne l'ai pas initialisee !
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

La valeur 101 notre variable provient d'une ancienne zone mémoire qui n'a pas été nettoyée. Si maintenant je fais les choses correctement :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int variable_type_entiere;
    variable_type_entiere = 0;

    printf("valeur de ma variable de type entier : %d\n", variable_type_entiere); //on affiche la valeur
    de notre variable

    if(variable_type_entiere == 0){ //si ma variable vaut 0
        printf("je fais le traitement prevu dans le cas ou ma variable etait a zero\n");
    } else { //sinon
        printf("ma variable ne vaut pas zero, peut etre que je ne l'ai pas initialisee !\n");
    }

    return 0;
}
```

Alors j'obtiens le résultat recherché :

```
valeur de ma variable de type entier : 0
je fais le traitement prevu dans le cas ou ma variable etait a zero
Appuyez sur <ENTRÉE> pour fermer cette fenÔtre...
```

Pensez donc toujours à initialiser vos variables, c'est essentiel et c'est considéré comme une excellente pratique !

3.3.2/ Manipuler des nombres.

Nous avons vu dans le chapitre précédent comment déclarer une variable. Il est maintenant temps de voir les différents types de variable numériques qu'il existe en C.

Mot clé du type	Description	Taille	Caractère d'affichage	Plage de valeur
short int	Petit entier	2 octets	%d, %u en unsigned et %x en hexadécimale	-32768 à 32767 ou 0 à 65535 en unsigned
int	Entier	4 octets	%d, %u en unsigned et %x en hexadécimale	-2147483648 à 2147483647 ou 0 à 4294967295 en unsigned
float	Petit décimal	4 octets	%f ou %e en notation scientifique	$3.4 \cdot 10^{-38}$ à $3.4 \cdot 10^{38}$
double	Décimal	8 octets	%f ou %e en notation scientifique	$1.7 \cdot 10^{-308}$ à $1.7 \cdot 10^{308}$
long double	Grand décimal	10 octets	%f ou %e en notation scientifique	$3.4 \cdot 10^{-4932}$ à $3.4 \cdot 10^{4932}$

Unsigned signifie que le bit utilisé pour effectuer le signe (- ou +) est utilisé cette fois ci pour code de l'information numérique. Cela explique qu'en mode unsigned (le suffixe unsigned est utilisé devant le type, par exemple unsigned int) il est possible de stocker de plus grandes valeurs, mais seulement supérieure ou égale à 0.

Il est possible d'utiliser les opérateurs mathématiques classiques sur ces variables (+, -, *, /).

Un autre type d'opérateurs utilisé sur ces variables sont les opérateurs de comparaison. Ils renvoient un type spécifique de données : les booléens qui peuvent prendre une valeur vraie (true) ou fausse (false). Ces opérateurs sont les suivants :

$A > B$: Si A est supérieur à B, alors vrai. Faux sinon.

$A < B$: si A est inférieur à B, alors vrai. Faux sinon.

$A \geq B$: si A est supérieur ou égal à B, alors vrai. Faux sinon.

$A \leq B$: si A est inférieur ou égal à B, alors vrai. Faux sinon.

$A == B$: si A est égale en valeur numérique à B, alors vrai (rappelez-vous, l'opérateur = simple est l'opérateur d'affectation, et non de comparaison). Faux sinon.

$A != B$: si A n'est pas égal à B, alors vrai. Faux sinon.

Ces opérateurs sont régulièrement utilisés avec les conditions que nous verrons plus tard.

3.3.3/ Manipuler des chaînes de caractères.

Nous avons vu dans le chapitre précédent comment manipuler des chiffres, mais il est parfois nécessaire de manipuler des caractères et non des nombres. Par exemple, dans les chapitres précédents, nous avons utilisé la fonction printf qui est en charge d'écrire des choses dans la console. Cette fonction travaille avec des caractères.

Le type char : c'est le type utilisé pour stocker un caractère. Il est codé sur 1 octet (soit huit bits). J'utilise le %c pour l'afficher (nous reviendrons sur cette notion d'opérateur %). Un char se déclare comme suit :

```
char caractere;
caractere = 'a';
```

Je déclare une variable nommée caractere avec le type char. Je mets la valeur a dans ma variable. J'encadre ma valeur a par des caractères quote (') pour dire au compilateur que c'est un caractère que je stocke dans cette zone mémoire et pas une autre variable nommée a.

Travailler avec un caractère est bien, mais il est parfois nécessaire de manipuler des ensembles de caractères, comme une phrase par exemple.

Attention, l'exemple qui va suivre va faire appel à des notions que nous n'avons pas encore abordées, comme les tableaux et les pointeurs. Ce qui compte c'est la façon dont les chaînes de caractères sont utilisées et affichées, leur déclaration viendra plus tard.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char chaine_de_caracteres_01[] = "ceci est ma premiere chaine de caracteres";
    char* chaine_de_caracteres_02 = "ceci est ma deuxieme chaine de caracteres";

    printf("valeur de la chaine 01 : %s\n",chaine_de_caracteres_01);
    printf("valeur de la chaine 02 : %s\n",chaine_de_caracteres_02);
    return 0;
}
```

Ce programme donne le résultat suivant :


```
valeur de la chaine 01 : ceci est ma premiere chaine de caracteres
valeur de la chaine 02 : ceci est ma deuxieme chaine de caracteres
Appuyez sur <ENTRÉE> pour fermer cette fen tre...
```

La chose à retenir ici c'est l'utilisation des doubles quotes (") pour encapsuler les chaînes de caractères. Ce symbole indique au compilateur que la variable que nous déclarons est un assemblage de caractères qui doivent être stockés les uns à la suite des autres en mémoire.

Remarque : On remarque l'utilisation de l'opérateur %s dans la fonction printf. Cet opérateur permet de dire que le %s (s pour string, ou ficelle, ou chaîne de caractères en informatique) doit être remplacé par la valeur contenue dans la zone mémoire chaine_de_caractere_01 ou 02 (en fonction de ce que je passe en paramètres dans la fonction printf).

Le détail de la fonction printf :

La fonction printf que nous utilisons depuis le début de ce cours fait partie d'une famille de fonction assez étrange puisqu'il est possible de lui passer un nombre infini de paramètres. Par exemple les deux appels suivants sont valides :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char chaine_de_caracteres_01[] = "ceci est ma premiere chaine de caracteres";
    char* chaine_de_caracteres_02 = "ceci est ma deuxieme chaine de caracteres";

    printf("premiere facon d'appeler la fonction printf :\n");
    printf("valeur de la chaine 01 : %s\n",chaine_de_caracteres_01);
    printf("valeur de la chaine 02 : %s\n",chaine_de_caracteres_02);

    printf("deuxieme facon d'appeler la fonction printf :\n");
    printf("valeur de la chaine 01 : %s\nvaleur de la chaine 02 : %s\n",chaine_de_caracteres_01,
chaine_de_caracteres_02);
    return 0;
}
```

Avec pour résultat la sortie console suivante :

```
premiere facon d'appeler la fonction printf :
valeur de la chaine 01 : ceci est ma premiere chaine de caracteres
valeur de la chaine 02 : ceci est ma deuxieme chaine de caracteres
deuxieme facon d'appeler la fonction printf :
valeur de la chaine 01 : ceci est ma premiere chaine de caracteres
valeur de la chaine 02 : ceci est ma deuxieme chaine de caracteres
Appuyez sur <ENTRÉE> pour fermer cette fen tre...
```

En fait, le programme va utiliser le premier paramètre comme règle de formatage des paramètres qui suivent. Par exemple, dans notre cas, le premier paramètre de la fonction printf contient deux fois %s, ce qui signifie que le deuxième paramètre va être interprété comme une chaîne de

caractères et le troisième paramètre aussi. Lors de l'affichage, les deux %s sont remplacé par les valeurs contenues dans chaine_de_caracteres_01 et chaine_de_caracteres_02.

Remarque, il aurait tout à fait été possible d'entrer le paramètre format dans une variable dédiée. Par exemple le code suivant est strictement équivalent à ce que nous avons fait précédemment :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char chaine_de_caracteres_01[] = "ceci est ma premiere chaine de caracteres";
    char* chaine_de_caracteres_02 = "ceci est ma deuxieme chaine de caracteres";
    char* format_chaine_de_caracteres_01 = "valeur de la chaine 01 :";
    char* format_chaine_de_caracteres_02 = "valeur de la chaine 02 :";

    printf("premiere facon d'appeler la fonction printf :\n");
    printf("%s %s\n", format_chaine_de_caracteres_01, chaine_de_caracteres_01);
    printf("%s %s\n", format_chaine_de_caracteres_02, chaine_de_caracteres_02);
    return 0;
}
```

Avec pour résultat dans notre console :

```
premiere facon d'appeler la fonction printf :
valeur de la chaine 01 : ceci est ma premiere chaine de caracteres
valeur de la chaine 02 : ceci est ma deuxieme chaine de caracteres
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Il est aussi possible de mixer les types de paramètres :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char chaine_de_caracteres_01[] = "ceci est ma premiere chaine de caracteres";
    char* chaine_de_caracteres_02 = "ceci est ma deuxieme chaine de caracteres";
    char* format_chaine_de_caracteres = "valeur de la chaine ";

    printf("premiere facon d'appeler la fonction printf :\n");
    printf(
        "%s %d : %s\n%s %d : %s\n",
        format_chaine_de_caracteres,
        1,
        chaine_de_caracteres_01, format_chaine_de_caracteres,
        2,
        chaine_de_caracteres_02
    );
    return 0;
}
```

```
premiere facon d'appeler la fonction printf :
valeur de la chaine 1 : ceci est ma premiere chaine de caracteres
valeur de la chaine 2 : ceci est ma deuxieme chaine de caracteres
Appuyez sur <ENTRÉE> pour fermer cette fenÊtre...
```

Cela fait un peu peur de prime abord mais dès que vous maitriserez ces subtilités vous n'aurez aucuns problèmes à utiliser ces raccourcis.

Les caractères spéciaux :

Vous avez du remarquer l'utilisation du `\n` lors des affichages des chaines de caractères depuis le début de ce cours.

Ce caractère `\n`, bien qu'il soit composé de deux caractères d'un point vue écriture, est considéré comme un SEUL caractère par le programme. Il fait partie d'une famille de caractères spéciaux conçus pour effectuer certaines opérations lors de l'affichage des strings :

`\r` : carriage return. Retour en début de ligne.

`\n` : line feed. On passe à la ligne suivante.

`\t` : tabulation.

`\0` : le caractère null (cela aura son importance dans la suite).

Vous pouvez utiliser ces caractères dans toutes les chaines de caractères pour les formater. Il en existe d'autres mais ils ne seront pas utilisés dans ce cours.

3.3.4/ Les tableaux.

Nous avons vu des variables unitaires précédemment (un entier, un nombre à virgule, un caractère). Or il est parfois nécessaire de manipuler des ensembles de données de même type. Cela s'appel un tableau. Ce tableau est un ensemble mémoire continu dans lequel il est possible d'aller effectuer des opérations sur un élément particulier. Un tableau est typé et se déclare comme suit :

```
type_du_tableau nom_du_tableau[taille_du_tableau]
```

Attention à bien mettre les [], ce sont eux qui indiquent au compilateur que c'est un tableau de variable qui va être utilisé.

Exemple de déclaration d'un tableau de 10 entiers :

```
int mon_premier_tableau[10];
```

Pour accéder à un élément particulier du tableau, j'utilise son indice.

L'indice : dans un tableau, la position d'un élément par rapport au début du tableau est appelée indice de l'élément. Attention cet indice commence à 0 ! Ce qui signifie que le premier élément de notre tableau aura l'indice 0.

Par exemple si je veux aller stocker la valeur 3 dans le cinquième élément de mon tableau, j'écrirais ceci :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int mon_premier_tableau[10]; //declaration du tableau de 10 entiers
    mon_premier_tableau[4] = 3; //affectation de la valeur 3 au cinquième élément

    //affichage de la valeur du cinquième élément
    printf("la valeur contenue dans le cinquième élément de mon premier tableau est :
%d",mon_premier_tableau[4]);
    return 0;
}
```

```
la valeur contenue dans le cinquieme element de mon premier tableau est : 3
Appuyez sur <ENTRÉE> pour fermer cette fenÊtre...
```

Remarque : Pour l'instant notre tableau n'est pas initialisé, ce qui signifie que tout ce que vous irez lire dedans n'est pas forcément cohérent avec le reste de votre programme. Nous verrons plus tard comment mettre une valeur par défaut mais gardez à l'esprit que nous ne sommes pas forcément dans une situation très sécurisée.

La taille d'un tableau est donc essentielle pour savoir quand on est entrain de travailler dans l'espace mémoire qui est réservé au tableau, et quand on ne l'est plus. C'est une des limitations principales des tableaux en C : si l'on perd l'information sur la taille du tableau, il est très difficile de savoir combien il y a d'éléments dedans (pour les parcourir par exemple). Il existe des techniques pour retrouver cette information mais il est préférable de toujours savoir la taille des tableaux que l'on manipule.

Les tableaux de char :

C'est un cas particulier des tableaux. Considérons le code suivant même si nous n'avons pas encore vu l'opérateur sizeof :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int mon_premier_tableau[10]; //declaration du tableau de 10 entiers

    char* mon_premier_tableau_char = "ccccccccc"; //déclaration du tableau de 10 caractères

    printf("taille du tableau d'entiers en octet : %d\ntaille du tableau de char en octet :
%d\n",sizeof(mon_premier_tableau),sizeof("ccccccccc"));
    return 0;
}
```

```
taille du tableau d'entiers en octet : 40
taille du tableau de char en octet : 11
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

On remarque que la taille de notre tableau de 10 int est de 40 octets. Cela est cohérent puisque un entier est codé sur 4 octets, le tableau prend donc au total 4 octets x 10 emplacements = 40 octets. Par contre le tableau de char prend lui 11 octets alors qu'un char n'est codé que sur 1 octet. Cette différence de 1 octet s'explique par le fait que le compilateur, lorsque nous allons lui déclarer notre tableau entre double quotes, va automatiquement rajouter le caractère '\0' (char null) a la fin de la chaîne. S'il fait cela, c'est tout simplement pour savoir quand il faut s'arrêter par exemple lorsqu'il va lire les données pour les afficher. Il sait que lorsqu'il est dans le cas d'une string (et donc d'un tableau de char) et qu'il rencontre le caractère \0, c'est que le tableau est terminé.

3.3.5/ Les listes

Le langage C ne possède pas d'implémentation standard des listes comme on peut les trouver dans d'autres langages. Il sera nécessaire de développer ce type vous-même si l'utilité s'en fait ressentir.

3.3.6/ Le type void

Le langage C étant très fortement typé, cette particularité a été poussée jusqu'à créer un type vide. C'est le type void, qui est un type primitif.

Exemple d'une fonction d'affichage qui ne renvoie rien :

```
void afficher(int entier_a_afficher){
    printf("la valeur de l'entier est : %d\n",entier_a_afficher);
}
```

Vous noterez l'absence du mot clé return. Il nous aurait cependant été possible d'écrire ceci, le résultat aurait été le même :

```
void afficher(int entier_a_afficher){
    printf("la valeur de l'entier est : %d\n",entier_a_afficher);
    return;
}
```

Ce type est obligatoire si une fonction ne retourne rien. Par contre le return ne l'est pas dans le cas d'une fonction de type de retour void.

3.3.6/ Regrouper des données

Il est parfois nécessaire de regrouper certaines données entre elles pour n'avoir à manipuler un type de donnée contenant toutes les informations nécessaires.

3.3.6.1/ Les structures

Cette entité particulière est appelée une structure en C. Elle permet de regrouper dans un espace contigu en mémoire des données de différents types tout en s'assurant que l'intégrité de cet ensemble soit préservée. Une structure a cette forme :

```

struct nom_structure {
    type_variable01 variable1;
    type_variable02 variable2;
    type_variable03 variable3;
};

```

Il y a plusieurs choses essentielles dans cet exemple :

- Le mot clé struct qui indique au compilateur que ce qui va suivre doit être traité comme une seule et même entité.
- Le nom de la structure. Il pourra être utilisé comme un type classique par la suite.
- Une liste de variables dont le type est spécifié.

Remarque : Il ne faut pas oublier le ; à la fin de la déclaration. Il est tout à fait possible de mettre plus que trois variables (dans les faits il n'y a pas vraiment de limite).

L'accès aux éléments contenus dans un structure s'effectue avec l'opérateur « . ».

Exemple :

```

#include <stdio.h>

struct structure_eleve {
    short int age;
    char nom[30];
    int num_etudiant;
};

void afficherEleve(struct structure_eleve etudiant ){
    printf("nom : %s\tage : %d\t num etudiant : %d\n",etudiant.nom,
etudiant.age,etudiant.num_etudiant);
}

int main(int argc, char *argv[])
{

    //déclaration de deux structures structure_eleve
    struct structure_eleve eleve_01;

    //initialisation des champs de ma structure
    eleve_01.age = 22;
    strcpy(eleve_01.nom,"jean");
    eleve_01.num_etudiant = 11111;

    afficherEleve(eleve_01);

    return 0;
}

```

Ce programme affiche le résultat suivant :

```
nom : jean      age : 22      num etudiant :11111
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Dans le code, il y a l'utilisation de la fonction strcpy qui permet de recopier des chaînes de caractères. La déclaration d'une structure ne nécessite pas forcément d'utiliser le mot struct avant le type de la variable, mais le mettre permet d'indiquer quel type d'entité on est entrain de manipuler.

Dans les faits, il est courant de déclarer les structures dans des .h pour que d'autres .c puissent les utiliser au travers des includes.

Remarque : dans notre exemple, notre structure contient un short int, un int et un tableau de 30 caractères. Notre structure a donc une taille de $2 + 4 + 30 = 36$ octets.

3.3.6.2/ Les unions

Les unions sont très similaires aux structures sinon qu'en fait les variables qui sont contenues à l'intérieur ne sont pas toutes accessibles à un instant t.

Nous avons vu précédemment que la taille d'une structure est la somme de toutes les variables qui la compose. Dans le cas d'une union, sa taille va être la taille maximale atteinte par une des variables qui la composent. Dans les faits cela signifie que les zones mémoires des différentes variables se recouvrent. Si nous reprenons notre exemple précédent :

Dans le cadre de la structure, la mémoire ressemblera à cela du point de vue octet :

short int	short int	char	char	char	char	char	char	char	char	char	Char
char	char	char	char	char	char	char	char	char	char	char	char
char	char	char	char	char	char	char	char	int	int	int	int

Dans le cas d'une union, cela va être un peu différent :

short int / int / char	short int / int / char	int / char	int / char	char	char	char	char	char	char	char	Char
char	char	char	char	char	char	char	char	char	char	char	char
char	char	char	char	char	char						

Notre union, bien qu'elle possède les mêmes variables internes que notre structure, n'a une taille que de 30 octets.

On voit bien que certains octets sont utilisés pour encoder toutes les variables contenues dans notre union. Cela signifie que lorsque nous irons lire les données, si nous allons chercher une variable autre que celle inscrite, alors cette variable sera dans un état incohérent.

Par exemple :

```

#include <stdio.h>

union union_eleve {
    short int age;
    char nom[30];
    int num_etudiant;
};

void afficherEleve(union union_eleve etudiant ){
    printf("nom : %s\tage : %d\t num etudiant :%d\n",etudiant.nom,
etudiant.age,etudiant.num_etudiant);
}

int main(int argc, char *argv[])
{

    //déclaration de deux structures structure_eleve
    union union_eleve eleve_01;

    //initialisation des champs de ma structure
    eleve_01.age = 22;
    strcpy(eleve_01.nom,"jean");
    eleve_01.num_etudiant = 11111;

    afficherEleve(eleve_01);

    return 0;
}

```

Ce programme donne maintenant le résultat suivant :

```

nom : g+      age : 11111      num etudiant :11111
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...

```

On voit bien que les données age et nom ne renvoie absolument pas ce que l'on souhaite. En fait, la dernière chose à avoir été entrée dans la mémoire est le numéro d'étudiant, c'est donc lui qui est dans la zone mémoire et qui s'affiche dans les variables.

Il faut faire très attention quand on manipule des unions.

3.4/ Les pointeurs

Comme nous l'avons vu précédemment, en C tout est une histoire de mémoire. Or, dans les exemples précédents, nous nous contentions de manipuler des valeurs contenues dans des zones de cette mémoire sans jamais avoir la possibilité de manipuler directement la zone mémoire en question. Pour pallier ce manque, les pointeurs ont été inventés.

Un pointeur est une variable contenant l'ADRESSE d'une autre variable. Ça ne parait pas très utile comme ça mais c'est en fait à la base de la majorité des mécanismes utilisés dans le langage C. S'ils n'existaient pas, nous n'aurions que très peu de programmes développés en C.

3.4.1/ L'espace mémoire d'un programme.

Petit retour en arrière. Nous avons vu précédemment que lorsque l'on demande l'exécution d'un programme, le système d'exploitation va lui allouer de la RAM pour qu'il puisse travailler et être chargé. Oui mais contrairement à ce que l'on pourrait croire, le système d'exploitation ne va pas mettre tous les programmes qui s'exécutent en même temps dans le même espace mémoire, car il serait très difficile de savoir quel octet de cette mémoire appartient à qui. Un système d'exploitation va donc définir un ensemble de segment mémoire par programme en cours d'exécution. Un programme va donc posséder un bout de mémoire qui lui est réservé et pouvoir faire à peu près ce qu'il souhaite dedans.

Il existe cependant deux types de zones mémoires disponibles pour un même programme : la pile et le tas.

3.4.1.1/ La pile

La pile est la mémoire centrale du programme. C'est en elle que les instructions processeur sont chargées pour être exécutées (rappelez-vous les histoires de registres et d'opérations primitives). Cette zone mémoire fonctionne en Last In First Out (LIFO) : premier arrivé = premier sorti (un peu comme une pile de quelque chose, on ne peut prendre que celui est au dessus, en ensuite le suivant). Cette pile (ou stack en anglais) est utilisée pour gérer les appels des fonctions par exemple. Si j'appelle une sous fonction dans une fonction principale (par exemple printf dans main) alors je vais placer mes instructions correspondantes à printf au dessus de la pile et je vais laisser mon processeur dépiler. Une fois qu'il aura exécuté toutes les instructions de printf, il va trouver en dessous les instructions correspondantes à la fonction main et les exécuter à la suite. La fin de la pile se traduit par la fin de la fonction main. On parle dans ce cas la de pile d'exécution. Les accès à cette zone mémoire sont extrêmement rapides puisque l'on se contente d'empiler des choses puis de les dépiler. Il n'y a pas besoin d'avoir une gestion manuelle de cette mémoire puisque le chemin d'exécution est tout tracé. Cette zone mémoire est cependant limitée en taille. Il ne faut pas trop la remplir sous peine d'avoir un problème d'espace.

3.4.1.2/ Le tas

C'est presque l'opposé de la pile. C'est un espace mémoire dans lequel le développeur aura la possibilité de gérer pratiquement tout ou presque. On appelle cela l'allocation dynamique de mémoire. Ce terme barbare signifie juste que cette zone mémoire va être réservée pour que le développeur y effectue les opérations qu'il souhaite.

Elle est allouée de façon dynamique par le système d'exploitation en fonction des demandes rédigées par le développeur. La contre partie est que ce dernier a complètement la main pour faire ce qu'il veut, les bonnes comme les mauvaises choses !

Le tas n'est pas limité en taille ... si ce n'est que par la limite physique de la RAM. Il est tout à fait possible de remplir physiquement cette RAM si le code du développeur est incomplet.

3.4.2/ La recopie des variables.

On parle de recopie des variables pour dire que lorsque l'on travaille avec une fonction, les variables qui composent les paramètres ou encore le retour de cette fonction ne sont pas celles que l'on a déclaré mais en fait des variables de même type qui contiennent une recopie de la valeur. Ce

mécanisme souterrain est appelé la recopie de variables, et le langage C est un langage travaillant par recopie.

Dans l'exemple suivant, je vais afficher les adresses des variables qui sont passées en paramètres et l'adresse de la variable de retour avant l'appel de ma fonction et après :

```
#include <stdio.h>

int additionnerEntiers(int nombre_01, int nombre_02){
    int result = nombre_01 + nombre_02;
    printf("dans ma fonction d'addition, les adresses de mes deux variables sont :\n\tnombre_01 :
%p\n\tnombre_02 : %p\n\tl'adresse de mon resultat est : %p\n",
        &nombre_01,
        &nombre_02,
        &result
    );
    return result;
}

int main(int argc, char *argv[])
{
    int nombre_01 = 4;
    int nombre_02 = 5;
    printf("avant l'entree dans ma fonction d'addition, l'adresse des deux variable est :\n\tnombre_01
: %p\n\tnombre_02 : %p\n",
        &nombre_01,
        &nombre_02);

    int resultat = additionnerEntiers(nombre_01,nombre_02);
    printf("l'adresse de mon resultat apres la fonction est : %p\n",&resultat);
    return 0;
}
```

```
avant l'entree dans ma fonction d'addition, les adresses de mes deux variables sont :
    nombre_01 : 0060FEAC
    nombre_02 : 0060FEA8
dans ma fonction d'addition, les adresses de mes deux variables sont :
    nombre_01 : 0060FE90
    nombre_02 : 0060FE94
    l'adresse de mon resultat est : 0060FE7C
l'adresse de mon resultat apres la fonction est : 0060FEA4
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Comme on le remarque, après le passage dans la fonction d'addition, toutes les adresses de variables sont différentes, ce qui signifie que l'on ne manipule pas les mêmes choses mais bien des copies de variables.

Dans l'absolu il n'y a pas vraiment de problème à cela si l'on considère que ce qui compte n'est pas quelle zone de la mémoire l'on utilise mais ce qu'elle contient (la valeur). La recopie en C garantit que cette valeur est transmise entre les différentes entités de variables (comme lorsque les paramètres sont recopiés dans notre fonction d'addition, ils contiennent toujours les valeurs 4 et 5 malgré le fait que cela ne soit plus les mêmes zones mémoires qui sont utilisées). Oui mais parfois il

est nécessaire de travailler sur une zone mémoire particulière. De plus, nous manipulons ici des entier (donc des paquets de 4 octets). Les temps de recopie sont négligeables. Par contre imaginons que nous travaillons avec des structures de plusieurs millions d'octets (plusieurs Mo de RAM) : ces temps de recopie pourraient devenir significatifs.

3.4.3/ Les pointeurs.

Pour manipuler directement de la mémoire, on utilise donc le concept de pointeur. Le pointeur est une variable qui contient l'adresse d'une autre variable. La principale différence c'est que lorsque l'on sait où se trouve quelque chose, on peut se permettre d'aller directement l'utiliser, plutôt que de travailler sur une copie de cette chose sans le savoir.

Remarque : étymologiquement, le pointeur est une variable qui pointe sur une autre variable.

3.4.3.1/ Le pointeur est une variable comme une autre.

Tout est dans le titre : le pointeur est une variable C comme une autre. Elle possède un nom, un type et une valeur (en l'occurrence l'adresse d'une autre variable de même type). Sa déclaration s'effectue comme suit :

```
type_pointeur *nom_pointeur ;
```

Le caractère * est obligatoire, comme quand le cas des tableaux avec [], c'est lui qui indique au compilateur que cette variable contient l'adresse d'une autre variable. Le type du pointeur peut être n'importe quel type primitifs (int, char, double ...) ou bien déclaré (structures, unions). On peut même stocker l'adresse d'une fonction dedans (mais ça ne sera pas développé dans ce cours).

Si nous prenons un schéma de ce qu'est une variable :

variable		
type	nom	valeur
Ce que l'on peut mettre dedans.	Comment je l'appelle pour l'utiliser.	Ce que je veux tant que c'est en accord avec le type.

Et maintenant celui d'un pointeur :

pointeur		
type	*nom	valeur
Type de la variable dont je vais stocker l'adresse.	Comment je l'appelle pour l'utiliser.	L'adresse de la variable du même type que mon pointeur.

C'est sensiblement la même chose, sauf que mon pointeur contient une information de type adresse. Il est possible de faire beaucoup de chose avec les pointeurs.

3.4.3.2/ Les opérations sur les pointeurs.

Il existe différentes opérations sur les pointeurs. Elles sont toutes destinées à faire quelque chose de différent.

La déclaration d'un pointeur : comme vu précédemment, si l'on souhaite utiliser un pointeur (ou une variable) il faut le déclarer comme ceci : `type_pointeur *nom_pointeur`.

Comme les autres variables, il est nécessaire de l'initialiser. C'est un peu plus simple que dans le cas des autres variables puisque la seule valeur par défaut est le mot clé NULL.

Exemple de déclaration et d'initialisation de deux pointeurs (un double et un int) :

```
double *pointeur_sur_double;  
pointeur_sur_double = NULL;  
  
int *pointeur_sur_int;  
pointeur_sur_int = NULL;
```

Remarque : Même si ce n'est « qu'une bonne pratique » d'initialiser les variables, c'est OBLIGATOIRE dans le cas des pointeurs. Au vu du fait qu'ils contiennent l'adresse d'une zone de la mémoire, si vous les utilisez sans les avoir initialisé, alors vous allez potentiellement faire des opérations sur une zone mémoire qui contient une variable, un morceau de fonction, ou même qui n'est pas dans le périmètre mémoire de votre programme (dans ce dernier cas, vous allez vous faire jeter par le système d'exploitation : on appelle cela une segmentation fault ou SEGFALT).

L'opérateur d'affectation : Nous avons vu qu'il était utilisé pour initialiser les pointeurs. Il est aussi possible d'utiliser l'opérateur d'affectation (le =) pour mettre l'adresse d'une variable qui nous intéresse. C'est ce que fait le code suivant :

```
double variable_double;  
variable_double = 0.0;  
double *pointeur_sur_double;  
pointeur_sur_double = NULL;  
pointeur_sur_double = &variable_double;
```

Pour récupérer l'adresse d'une variable, j'utilise le &. Placé en tête du nom de ma variable, il me permet de récupérer son adresse et de la mettre dans la zone valeur de mon pointeur au travers de l'opérateur =. Mon pointeur_sur_double contiendra donc l'adresse de ma variable variable_double.

Il est aussi possible d'utiliser l'opérateur d'affectation (=) pour y mettre la valeur d'un pointeur de même type. On se retrouvera donc avec plusieurs pointeurs pointant sur une même variable.

Afficher l'adresse d'une variable : il suffit pour cela d'utiliser le caractère de formatage %p dans un printf par exemple. Soit on déclare un pointeur sur notre variable dont on souhaite afficher l'adresse, soit on utilise directement le symbole & devant le nom de la variable lors de l'utilisation du printf.

Exemple :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int nombre_01;
    nombre_01 = 0;

    int *pointeur_int_01 = NULL;
    pointeur_int_01 = &nombre_01;

    printf("valeur par le pointeur : %p\nvaleur directement par l'adresse de la variable : %p\n",pointeur_int_01,&nombre_01);

    return 0;
}
```

```
valeur par le pointeur : 0060FEA8
valeur directement par l'adresse de la variable : 0060FEA8
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

On peut voir que les deux adresses sont strictement identiques, ce qui signifie que l'on manipule bien la même zone mémoire (et donc la même variable).

L'opérateur de déréférencement : Il est possible au travers du pointeur de modifier la valeur contenue dans la variable qu'il pointe. On appelle cette opération le référencement du pointeur.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int nombre_01;
    nombre_01 = 0;

    int *pointeur_int_01 = NULL;
    pointeur_int_01 = &nombre_01;

    printf("valeur de la variable nombre_01 : %d\n",nombre_01);

    *pointeur_int_01 = 3;
    printf("valeur de la variable nombre_01 apres modification via le pointeur : %d\n",nombre_01);

    return 0;
}
```

```
valeur de la variable nombre_01 : 0
valeur de la variable nombre_01 apres modification via le pointeur : 3
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Avec l'utilisation du symbole * en suffixe du nom de mon pointeur, j'ai pu affecter une valeur à ma variable pointée sans même connaître son nom.

Si nous reprenons le premier exemple que nous avons utilisé pour mettre en évidence la copie de variable :

```
#include <stdio.h>

void additionnerEntiers(int *nombre_01, int *nombre_02, int *resultat){
    *resultat = *nombre_01 + *nombre_02;
    printf("dans ma fonction d'addition, les adresses de mes deux variables sont :\n\tnombre_01 :
%p\n\tnombre_02 : %p\n\tl'adresse de mon resultat est : %p\n",
        nombre_01,
        nombre_02,
        resultat
    );
}

int main(int argc, char *argv[])
{
    int nombre_01 = 4;
    int nombre_02 = 5;
    printf("avant l'entree dans ma fonction d'addition, les adresses de mes deux variables sont
:\n\tnombre_01 : %p\n\tnombre_02 : %p\n",
        &nombre_01,
        &nombre_02);

    int resultat;
    additionnerEntiers(&nombre_01,&nombre_02,&resultat);
    printf("l'adresse de mon resultat apres la fonction est : %p\n",&resultat);
    return 0;
}
```

```
avant l'entree dans ma fonction d'addition, les adresses de mes deux variables sont :
    nombre_01 : 0060FEAC
    nombre_02 : 0060FEA8
dans ma fonction d'addition, les adresses de mes deux variables sont :
    nombre_01 : 0060FEAC
    nombre_02 : 0060FEA8
    l'adresse de mon resultat est : 0060FEA4
l'adresse de mon resultat apres la fonction est : 0060FEA4
Appuyez sur <ENTRÉE> pour fermer cette fen tre...
```

On remarque que cette fois il n'y a pas eu de copie des différentes variables puisque j'ai travaillé avec leurs adresses.

Remarque : Les tableaux en C sont en fait des pointeurs. Un nom de variable représentant un tableau est en fait un pointeur sur le premier élément du tableau. Il est donc possible de manipuler des tableaux en utilisant un pointeur sur le premier élément. Par exemple :

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int tableau_entier[10];
    int *pointeur_entier;
    pointeur_entier = NULL;

    pointeur_entier = tableau_entier;
    printf("valeur de l'adresse de mon premier element de tableau : %p\n",
           valeur de ma variable
           tableau : %p\n",
           valeur de mon pointeur : %p\n",
           &(tableau_entier[0]),
           tableau_entier,
           pointeur_entier);

    return 0;
}

```

```

valeur de l'adresse de mon premier element de tableau : 0060FE84
valeur de ma variable tableau : 0060FE84
valeur de mon pointeur : 0060FE84
Appuyez sur <ENTRÉE> pour fermer cette fen tre...

```

La manipulation des tableaux peut donc s'effectuer sans probl me au niveau des pointeurs.

3.4.3.3/ L'op rateur sizeof.

C'est un op rateur du C tr s utilis  car il permet de conna tre la taille d'une variable. Il s' crit sizeof(*ma_variable*) ou bien sizeof(*mon_type*) et renvoie un nombre d'octet.

3.4.3.3/ Le cas du pointeur void *

Comme tous les types du C, il est possible de d clarer un pointeur de type void (type vide). Or, ce pointeur est un peu diff rent des autres : il permet de pointer sur une variable (donc de contenir l'adresse de cette variable) sans que le type de cette derni re ne soit connu. Son utilisation ne concerne que des cas   la marge car comme   chaque fois en C que l'on ne conna t pas ce que l'on manipule c'est extr mement dangereux.

Exemple :

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int variable_entier = 3;
    int *pointeur_entier;
    pointeur_entier = NULL;
    void *pointeur_void_sur_entier;
    pointeur_void_sur_entier = NULL;

    pointeur_entier = &variable_entier;
    pointeur_void_sur_entier = &variable_entier;
    printf("valeur pointeur entier : %p\nvaleur pointeur void : %p\n",
        pointeur_entier,
        pointeur_void_sur_entier);

    *pointeur_entier = 5;
    printf("valeur apres modification via pointeur entier : %d\n", variable_entier);

    int *pointeur_entier_02 = (int*) pointeur_void_sur_entier;
    *pointeur_entier_02 = 9;
    printf("valeur apres modification via pointeur void : %d\n", variable_entier);

    return 0;
}

```

```

valeur pointeur entier : 0060FEA0
valeur pointeur void : 0060FEA0
valeur apres modification via pointeur entier : 5
valeur apres modification via pointeur void : 9
Appuyez sur <ENTRÉE> pour fermer cette fen tre...

```

Ceci est    viter absolument bien qu'il soit important de savoir que cela existe.

3.4.3.4/ Le cas des structures.

Le d r f rencement des structures est un peu particulier. Nous avons vu que dans le cas des pointeurs, il faut utiliser l'op rateur de d r f rencement *. Lorsque l'on a un pointeur sur une structure, il est possible d'acc der aux variables internes via l'op rateur ->.


```

#include <stdio.h>

struct exemple_dereferencement {
    int nombre_01;
    char caractere_02;
};

int main(int argc, char *argv[])
{
    struct exemple_dereferencement structure_test;
    structure_test.nombre_01 = 0;
    structure_test.caractere_02 = 'a';
    struct exemple_dereferencement *pointeur_structure_test;
    pointeur_structure_test = NULL;
    pointeur_structure_test = &structure_test;
    printf("valeur nombre_01 : %d\t valeur caractere_02 : %c\n",(*pointeur_structure_test).nombre_01,(*pointeur_structure_test).caractere_02);

    //modification à l'aide de l'opérateur *
    (*pointeur_structure_test).nombre_01 = 1;
    (*pointeur_structure_test).caractere_02 = 'b';
    printf("valeur nombre_01 : %d\t valeur caractere_02 : %c\n",(*pointeur_structure_test).nombre_01,(*pointeur_structure_test).caractere_02);

    //modification à l'aide de l'opérateur ->
    pointeur_structure_test->nombre_01 = 2;
    pointeur_structure_test->caractere_02 = 'c';
    printf("valeur nombre_01 : %d\t valeur caractere_02 : %c\n",
        pointeur_structure_test->nombre_01,pointeur_structure_test->caractere_02);
    return 0;
}

```

A noter l'utilisation des () pour l'opérateur *, ce qui rend l'opérateur -> plus commode dans la plus part des cas de figure impliquant des pointeurs sur des structures.

3.4.4/ La gestion de la mémoire.

Le chapitre précédent nous a montré qu'il était possible de gérer la mémoire en utilisant des pointeurs pour manipuler des variables au travers de leurs adresses. Or les pointeurs ont aussi une autre utilité.

Toutes les opérations que nous avons effectuées précédemment n'ont été réalisées que dans la partie de la mémoire appelée la pile. Nous n'avons pas utilisé le tas.

3.4.4.1/ Les garbage collectors.

Le tas est une zone mémoire dont la gestion est à la discrétion de l'utilisateur. Il lui est donc possible de remplir cette zone sans que cela ne provoque un quelconque avertissement du système d'exploitation.

Certains langages (le java par exemple) possèdent des logiciels capable de détecter quand une zone de données déclarée par le développeur dans le tas n'est plus utilisée et donc de nettoyer cette zone

pour une future réutilisation. Ces programmes sont appelés garbage collector (des éboueurs). Cette notion n'existe pas en C, c'est donc le développeur qui va gérer ce nettoyage à la main.

3.4.5/ Les opérations sur la mémoire.

Les pointeurs sont la base de tout travail sur le tas. Tout ce qui va être gestion de cette zone mémoire passe par eux au travers de fonctions bien spécifiques qui sont intégrées dans la plus part des compilateurs.

3.4.5.1/ L'allocation dynamique de mémoire.

Allouer dynamiquement de la mémoire signifie que le développeur va demander au programme d'aller déposer un objet dans le tas pour pouvoir le réutiliser plus tard. Cette opération est très utile quand il a un besoin de créer une variable qui sera réutilisée dans différents points du programme. En effet, les variables ont une portée et si on ne veut pas passer par une variable statique ou globale, il est nécessaire de créer une variable dans une zone de la mémoire qui ne sera pas affectée par la fonction dans laquelle nous nous trouvons. Cette zone mémoire est le tas. Elle est commune à l'ensemble du programme et accessible par tous, tant que les appelants connaissent l'adresse mémoire et le type (donc la taille en octets) de la variable qu'ils souhaitent utiliser.

Une des fonctions permettant l'allocation dynamique de mémoire est la fonction malloc. Elle s'utilise comme suit :

```
//initialisation d'un pointeur
int *pointeur_int = NULL;

//demande d'allocation dynamique d'une zone mémoire de la taille d'un int
pointeur_int = malloc(sizeof(int));
```

A la fin de ces deux instructions, le pointeur pointeur_int possède l'adresse d'une zone mémoire contenant une variable de type int, le tout dans la zone mémoire du tas. Cette variable n'est pas concernée par la portée de la fonction. Tant qu'elle n'est pas détruite par le développeur, elle continuera à exister.

Remarque : La fonction malloc renvoie un pointeur de type void*. Cela est dû au fait qu'elle va en fait se contenter de réserver une quantité donnée d'octets (fixée par ce que renvoie l'opérateur sizeof). Elle ne sait donc pas ce qu'elle manipule. Imaginez que cette fonction renvoie un pointeur typé, alors il aurait été nécessaire de coder cette fonction pour chaque type connu (si le type renvoyé change, alors la signature n'est plus la même et il faut une nouvelle implémentation de la fonction). De plus, pour chaque nouveau type que le développeur crée (des structures par exemple) il aurait fallu que ce dernier code cette fonction malloc. Toute cette énergie dépensée vaut bien que l'on prenne le risque de manipuler très temporairement un pointeur void*, mais incite à être très prudent sur ce que l'on fait.

3.4.5.2/ la libération de la mémoire.

Comme indiqué dans le paragraphe précédent, c'est le développeur qui a la main sur le tas. Notre exemple montre que nous avons alloué dynamiquement de la mémoire, mais nous n'avons pour l'instant pas supprimé cette variable à la fin de l'utilisation.

Il faut bien être conscient que si vous oubliez de supprimer une variable allouée dynamiquement, personne ne le fera pour vous. Il faut donc être très attentif quand l'on effectue de l'allocation dynamique de mémoire. De façon générale en informatique, et plus particulièrement en langage C, tout ce qui est ouvert, alloué, ... doit être fermé, libéré, ... que l'on parle d'allocation dynamique, de flux, ... Gardez toujours cette règle à l'esprit.

La fonction `free` est spécifiquement dessinée pour cela. Elle s'utilise comme suit :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    //initialisation d'un pointeur
    int *pointeur_int = NULL;

    //demande d'allocation dynamique d'une zone mémoire de la taille d'un int
    pointeur_int = malloc(sizeof(int));

    //manipulation de la variable int située dans le tas

    *pointeur_int = 3;
    printf("valeur de la variable dans le tas avant de la modifier : %d\n", *pointeur_int);

    *pointeur_int = 9;
    printf("valeur de la variable dans le tas apres l'avoir modifiee : %d\n", *pointeur_int);

    //je libère le tas car j'ai fini d'utiliser ma variable
    free(pointeur_int);
    //je n'oublie pas de mettre la valeur de mon pointeur à null car il ne pointe plus sur rien
    pointeur_int = NULL;

    return 0;
}
```

Ce programme donne ce résultat :

```
valeur de la variable dans le tas avant de la modifier : 3
valeur de la variable dans le tas apres l'avoir modifiee : 9
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Nous avons donc alloué dynamiquement un `int` sur le tas, utilisé ce dernier, puis libéré l'espace mémoire car nous n'en avons plus besoin.

Les fonctions `malloc` et `free` ne sont pas les seules permettant de gérer la mémoire dynamiquement, mais elles sont une bonne base et permettent de presque tout faire.

3.4.5.3/ Les fuites mémoire.

Que se passe-t-il si un développeur alloue une zone mémoire sur le tas (en utilisant la fonction `malloc`) mais qu'il perd l'adresse de cette zone mémoire ? Il provoque un phénomène que l'on qualifie de fuite mémoire. En somme, cette zone allouée dans le tas n'est plus utilisée mais le

système d'exploitation ne l'a pas référencée comme étant libre. Il ne pourra donc pas la remettre à disposition d'un programme qui ferait la demande d'un emplacement.

Cette mémoire sera récupérée lorsque la machine physique sera redémarrée, mais tant que cela n'a pas été le cas, elle est perdue pour le système d'exploitation. Il est possible de provoquer une saturation de la RAM si l'on ne fait pas attention à ce genre de chose.

La seule façon d'éviter ce cas de figure est de toujours libérer la mémoire que l'on a alloué dynamiquement. Plus simplement, pour chaque fonction `malloc()` présente dans votre code, il faut la fonction `free()` correspondante.

3.4.6/ Les fonctions qui renvoient des pointeurs.

Il peut arriver que certaines fonctions renvoient des pointeurs sur des variables déclarées dans les corps de ces fonctions. Cependant il faut faire très attention à cela car si la mémoire n'est pas gérée comme il le faut dans les fonctions, alors il peut y avoir des problèmes de cohérence de la mémoire.

Considérons le code suivant :

```
#include <stdio.h>

int* addition(int nombre_01, int nombre_02){
    int resultat;
    resultat = 0;
    resultat = nombre_01 + nombre_02;
    int *pointeur_resultat;
    pointeur_resultat = NULL;
    pointeur_resultat = &resultat;
    return pointeur_resultat;
}

int main(int argc, char *argv[])
{
    int a;
    int b;
    a = 3;
    b = 2;

    int *pointeurResultat = addition(a,b);

    return 0;
}
```

Normalement il n'y a pas de problème, cependant si nous nous intéressons à la fonction `addition`, nous remarquons que le pointeur de type entier que nous renvoyons pointe sur l'adresse d'une variable locale (interne à la fonction `addition`). Or, une fois que nous sommes sorti de la fonction, cette variable n'existe plus ! Si nous souhaitons afficher le résultat, nous risquons de déréférencer un pointeur qui ne pointe plus sur rien.

Il faudra donc plutôt utiliser une allocation dynamique. Il faut modifier le code précédent comme suit :

```
#include <stdio.h>

int* addition(int nombre_01, int nombre_02){
    int resultat;
    resultat = 0;
    int *pointeur_resultat;
    pointeur_resultat = NULL;
    pointeur_resultat = malloc(sizeof(int));
    *pointeur_resultat = nombre_01 + nombre_02;
    return pointeur_resultat;
}

int main(int argc, char *argv[])
{
    int a;
    int b;
    a = 3;
    b = 2;

    int *pointeurResultat = addition(a,b);
    free(pointeurResultat);
    return 0;
}
```

Cette fois nous avons enregistré notre variable dans un espace mémoire où elle ne risque pas de disparaître. Nous pouvons donc la réutiliser sans aucun problème. Il faut juste penser à libérer l'espace mémoire une fois que nous n'en avons plus besoin.

3.5/ Les opérateurs conditionnels

Il est parfois nécessaire de vérifier certaines conditions pour pouvoir atteindre certaines parties d'un programme plutôt que d'autres. En C, la base de ces conditions est un type qui n'est pas référencé : le booléen.

Un booléen n'est pas utilisable tel quel. Ce n'est pas un type primitif. Il faut plutôt le voir comme une opération de comparaison. Il est à la base de toutes les instructions conditionnelles.

3.5.1/ Le if, else et else if

Le if est la structure de comparaison la plus classique. Elle s'écrit comme cela :

```
if(condition){
    //on fait quelque chose
}
```

La principale difficulté est sur condition. C'est cette valeur qui in fine doit être placée à vrai ou faux. Pour ce faire nous avons à notre disposition les multiples opérateurs de comparaison référencés au chapitre sur les variables numériques. On peut par exemple écrire ce genre de choses :

```
int a;  
int b;  
a = 3;  
b = 2;  
  
if(a > b){  
    //condition vraie  
}  
  
if(a < b){  
    //condition fausse  
}  
  
if(a >= b){  
    //condition vraie  
}  
  
if(a <= b){  
    //condition fausse  
}  
  
if(a != b){  
    //condition vraie  
}  
  
if(a == b){  
    //condition fausse  
}
```

On peut de plus inverser les conditions :

```
int a;  
int b;  
a = 3;  
b = 2;  
  
if( !(a > b)){  
    //condition fausse  
}  
  
if( !(a != b)){  
    //condition fausse  
}  
  
if( !(a == b)){  
    //condition vraie  
}
```

L'utilisation de l'opérateur d'inversion ! permet de donner l'inverse d'une condition booléenne. Attention, les parenthèses ont leur importance, elles permettent de prioriser le calcul de la condition, puis d'inverser le résultat.

La condition else est un complément du if. Elle permet d'effectuer des instructions dans le cas où la condition du if n'est pas vraie.

```
if(acondition){  
    //condition vraie  
} else {  
    //dans le cas où la condition n'est pas vérifiée on fait les instructions de ce bloc  
}
```

Le else if est un complément du else. Cette instruction conditionnelle permet d'effectuer une nouvelle vérification avant d'exécuter les instructions contenues dans le bloc else. Si la condition demandée par cette vérification est fausse, alors le bloc else n'est pas exécuté. Le bloc elseif est en fait un if imbriqué dans le bloc else. Il est donc tout à fait possible de le compléter avec un bloc else final.

```
if(condition1){  
    //condition1 est vraie  
} else if(condition2) {  
    //dans le cas où la condition1 n'est pas vraie mais que la condition2 est vraie  
} else {  
    //aucunes des deux conditions (condition1 et condition2) n'est vérifiée.  
}
```

3.5.2/ Le switch

Le switch est un type de structure décisionnelle permettant d'effectuer une suite d'instructions en fonction d'une condition entière.

```

int condition;
switch(condition){

    case 0 :{
        //si condition vaut 0 on exécute ce bloc
        break;
    }

    case 1 :{
        //si condition vaut 1 on exécute ce bloc
        break;
    }

    case 255 :{
        //si condition vaut 255 on exécute ce bloc
        break;
    }

    default : {
        //si la valeur de condition n'est pas répertoriée, alors on exécute ce bloc
        break;
    }

}

```

Attention, les mots clé break sont obligatoires. S'ils ne sont pas présents, il y a des chances pour que les cas suivants (qui ne satisfont pas à la condition) soient tout de même exécutés.

3.5.3/ Les opérateurs sur les booléens

Il est possible d'assembler les conditions à l'aide des opérateurs && (et logique) et || (ou logique).

Dans le cas ou nous avons deux conditions : condition1 qui est vraie et condition2 qui est fausse :

- condition1 && condition2 => faux
- condition1 || condition2 => vrai
- condition1 && condition1 => vrai

3.6/ Les boucles

Les boucles sont des structures d'instructions utilisées pour faire des traitements récursifs sur des ensembles de données.

3.6.1/ La boucle for

Les boucles for sont spécialisées dans les itérations avec entier.

```

int iterateur;
for(iterateur = 0, iterateur < 10 ; iterateur++){
    //bloc d'instructions
}

```

Dans cet exemple, on fixe l'entier iterateur à 0 lors de l'entrée dans la boucle for. Puis on va itérer tant que la variable iterateur est inférieure à 10. A chaque tour de boucle, on ajoute 1 à la variable

iterateur (c'est le sens de l'instruction `iterateur++`). Dès que la condition n'est plus vraie (iterateur est supérieur ou égal à 10) on sort de la boucle `for`.

3.6.2/ La boucle `while`

La boucle `while` est relativement simple. Tant qu'une condition est vraie, elle va exécuter les instructions qu'elle contient.

```
while(condition){
    //on exécute le bloc d'instruction. Arrivé à la fin, on recommence au début de la boucle.
}
```

Attention à bien vérifier que la condition se termine bien à un moment, sinon le programme risque d'entrer dans une boucle infinie.

3.6.3/ La boucle `do while`

Elle est très similaire à la boucle `while`, sauf que contrairement à cette dernière, elle effectue au moins une fois les instructions qu'elle contient.

```
do {
    //on exécute le bloc d'instruction au moins une fois.
    //Arrivé à la fin, on vérifie la condition et on recommence au début de la boucle si elle est vraie.
}while(condition);
```

3.6.4/ Les mots clé `continue` et `break`

Ces mots clé n'ont une signification qu'à l'intérieur des structures d'itération (`for`, `while` et `do while`). Ils servent à réguler l'exécution des boucles.

```
while(condition){
    //premier block d'instructions
    if(condition2){
        //si la condition est vraie
        break; //alors on peut sortir de la boucle et le deuxième bloc d'instructions ne sera pas exécuté
    }
    //deuxième block d'instructions
}
```

```
while(condition){
    //premier block d'instructions
    if(condition2){
        //si la condition est vraie
        continue; //alors on retourne au début de la boucle et le deuxième bloc d'instructions ne sera pas exécuté
    }
    //deuxième block d'instructions
}
```