1 Introduction à Microsoft Visual Studio

Exécuter Microsoft Visual Studio 2010 :

```
Menu Démarrer --> Tous les programmes -->
    Microsoft Visual Studio 2010 --> Microsoft Visual Studio 2010
```

A la première exécution, une fenêtre s'ouvre, dans laquelle il faut choisir :

```
Paramètre de développement Visual C++
```

puis cliquer sur "Démarrer".
Pour créer une solution :

```
Fichier --> Nouveau --> Projet
```

Dans la fenêtre :
- A droite : sélectionner Win32 Console Application
- En bas, première entrée de texte : le nom du projet.
- En bas, deuxième entrée de texte : le nom de la solution.

Cliquer sur OK. Dans la nouvelle fenêtre :
- A gauche : Paramètres de l'application
- A droite : sélectionner Projet vide. **NE PAS OUBLIER !**.

Cliquer sur "Terminer".
Dans l'explorateur à gauche, on va rajouter un fichier :

```
Click droit sur Fichiers Source --> Ajouter --> Nouvel Elément
```

Dans la nouvelle fenêtre :
- A gauche : Code
- A droite : C++ file (.cpp)
- En bas : le nom (exemple1.c)

Cliquer sur "Ajouter". Dans l'éditeur qui apparaît, on écrit le code.
Pour la compilation :

```
Générer --> Générer la solution   (ou F7)
Générer --> Regénérer la solution (ou Ctrl-Alt-F7)
```

Pour l'exécution :

```
Déboguer --> Démarrer le débogage   (ou Ctrl-F5)
Déboguer --> Exécuter sans débogage (ou F5)
```

Pour nettoyer la solution :

```
Générer --> Nettoyer la solution
```

2 Premier exemple : "Hello world"

Dans cette section, on va écrire le premier programme qui est traditionnellement utilisé par tous les cours et livres traitant de programmation informatique : l'affichage du message "Hello world". Dans notre cas, on va afficher ce message dans une console.
Un programme, exécutable par un ordinateur, est inexploitable par un être humain. On commence par écrire, dans un fichier texte, une série d'instructions

d'un langage informatique (ici le C), plus compréhensibles. Ensuite on utilise un programme, appelé "compilateur", qui transformera ce fichier texte en un fichier exécutable par un ordinateur.
On utilisera souvent le mot "programme" pour désigner à la fois le fichier exécutable et le fichier texte contenant les instructions en C. En général, il n'y a pas de confusion possible.

2.1 Le programme

A l'aide de l'éditeur de Visual Studio, on écrit les instructions suivantes dans le fichier *hello_world.c* :

```
#include <stdio.h>

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

2.2 Compilation et exécution

Comme il a été mentionné plus haut, on doit maintenant compiler le programme ci-dessus pour créer un fichier exécutable, qui va exécuter la suite d'instructions. Appuyer sur F7 pour compiler le programme. Puis Ctrl-F5 pour l'exécuter.
Normalement, dans le terminal, on doit voir le message suivant :

```
Hello world!
```

2.3 Explications du code

On reprend le programme en C (on utilisera aussi la dénomination "code", à la place de "programme en C"), en numérotant les lignes.

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

2.4 La fonction "main"

Tout programme en C doit comporter une et une seule mention de la fonction *main*. Comme en mathématiques, une fonction est un objet qui a un nom, prend des paramètres, fait des opérations et retourne une valeur. Ici, à la ligne 3, on voit que la fonction *main* ne prend pas de paramètres et renvoie un entier **int**. Les opérations à effectuer sont écrites après l'accolade ouvrante (qui doit

impérativement suivre une définition de fonction) et se terminent avant l'accolade fermante, c'est-à-dire les lignes 5 et 6.
Cette fonction doit retourner une valeur (un entier). Pour cela, on utilise l'expression **return** à la fin de la fonction, en lui assignant la valeur de retour. Ici, à la ligne 6, on dit que la valeur de retour de la fonction *main* sera toujours 0. Pour plus d'informations, voir la section 6.4.

2.5 L'instruction "printf"

Pour afficher un message dans le terminal, on utilise la fonction standard **printf**. On lui donne comme argument la chaîne de caractères que l'on veut afficher. Ici, la chaîne de caractères est donnée comme un ensemble de lettres, entourées de guillemets.
La fonction **printf** n'est pas comprise par défaut par le compilateur et on a besoin de lui spécifier sa déclaration. Pour cela, on inclut un fichier, *stdio.h*. C'est le but de la ligne 1 du programme.

3 Généralités sur le langage C

Cette section expose quelques règles générales intervenant dans l'écriture d'un programme en C.

3.1 Les identificateurs

Les identificateurs servent à identifier les objets manipulés dans le programme. Ce sont les variables, les fonctions, etc... Ils sont formés d'une suite de caractères, choisis parmi les lettres de l'alphabet majuscules ou minuscules (pas de lettres accentuées), de chiffres et du caractère de soulignement. Un identificateur ne peut pas commencer par un chiffre. De plus, la casse est prise en compte.

3.2 Les mots clés

Les mots clés sont des identificateurs qui sont réservés par le langage. Ils ne peuvent pas être utilisés comme noms de fonctions ou de variables. En voici une liste des plus couramment utilisés :

auto	default	float	register	struct	volatile
break	do	for	return	switch	while
case	double	goto	short	typedef	
char	else	if	signed	union	
const	enum	int	sizeof	unsigned	
continue	extern	long	static	void	

3.3 Les séparateurs

Pour que le compilateur puisse comprendre la suite d'instructions d'un programme en C, il faut qu'il puisse distinguer les identificateurs et mots clés successifs. Pour cela, on peut utiliser indifféremment un espace, une tabulation, une fin de ligne (retour chariot), en nombre quelconque, avec une possibilité d'utiliser plusieurs d'entre eux successivement (une espace, suivi d'un retour chariot,

est un séparateur).
Il se peut que la syntaxe du langage C impose un séparateur (virgule, point-virgule, parenthèse, accolade, crochet, etc...). dans ce cas, il n'est pas nécessaire d'utiliser un des séparateurs ci-dessus (espace, etc...).
L'utilisation de plusieurs séparateurs consécutivement est très utile pour écrire un code clair, qui sera facilement compréhensible par d'autres personnes que le programmeur. Plusieurs styles existent, et en général, chaque programmeur a son propre style. Celui proposé par GNU
(http ://www.gnu.org/prep/standards/standards.html)
est un exemple de styles très lisible. L'éditeur de Visual Studio en propose un par défaut.
En particulier, voici quelques règles couramment utilisées :

- Mettre un espace après une virgule.
- Mettre un espace autour du signe "=".
- Aller à la ligne après un point-virgule.
- Indenter le code (utiliser la touche Tab).
- Surtout, faire un code consistant et ne pas modifier la forme au cours du programme.

3.4 Les commentaires

Le langage C permet de mettre des commentaires dans le code, pour expliquer certaines choses que le programmeur juge importantes (code à améliorer, erreurs connues à corriger plus tard, detail d'un algorithme, etc...). Les commentaires sont ignorés par le compilateur et n'ont aucune incidence sur le programme crée.
En C, un commentaire commence par /* et se termine par */ Tout texte compris entre ces deux "balises" est ignoré. En particulier, un commentaire peut s'étendre sur plusieurs lignes.
Il ne faut pas abuser de commentaires, ceux-ci pouvant rendre illisible le code, alors que leur but est le contraire.

4 Les variables

4.1 Généralités

Le but d'un ordinateur étant de faire des calculs, on va devoir utiliser des variables pour stocker des valeurs, puis manipuler ces variables. Avant de manipuler une variable en C (ANSI), il faut effectuer deux étapes :

- il faut **déclarer** la variable, ce qui revient à lui assigner un type,
- il faut **définir** la variable, ce qui revient à lui donner une valeur.

La syntaxe de la **déclaration** d'une variable est la suivante :

```
type nom_variable;
```

où *type* est le type de la variable (par exemple entier ou réel), *nom_variable* est un identificateur, qui est bien sur le nom de la variable. On termine l'instruction par un point virgule. On ne peut pas déclarer deux variables de même nom dans le même bloc d'instruction (à une exception près, voir section 5.1).
On verra plus en détail les types dans la secion 4.2.
Par exemple, pour déclarer une variable de type **int**, ayant pour nom *a* :

```c
int a;
```

Pour déclarer plusieurs variables de même type, on peut utiliser la syntaxe suivante :

```
type var1, var2, var3;
```

c'est-à-dire mettre à la suite les noms des variables, en les séparant par des virgules. Par exemple :

```c
int a, b, c;
```

La syntaxe d'une **définition** d'une variable est la suivante :

```
nom_variable = valeur;
```

où *nom_variable* est le nom d'une variable précédemment déclarée (si ce n'est pas le cas, le compilateur ne compilera pas et génèrera une erreur), et *valeur* est soit une valeur numérique correspondant au type de *nom_variable*, ou bien une autre variable précédemment déclarée et définie de même type, ou bien la valeur retournée par une fonction. On termine l'instruction par un point virgule.
Par exemple, pour déclarer deux variable a et b, définir a à 12 et b à a :

```c
int a;
int b;

a = 12;
b = a;
```

On peut aussi définir une variable lors de sa déclaration. En général on initialise ces variables avec des valeurs numériques :

```c
int a, b = 12, c;
double d = 1.0;
```

4.2 Les types

Une variable doit forcément avoir un (unique) type. Le langage C possède des types prédéfinis. Ils entrent dans deux catégories : les types entiers et les types flottants (équivalent des réels en informatique), que nous allons voir dans cette section. Le langage C dispose d'un autre type, les pointeurs, qui sont reliés aux chaînes de caractères. Nous les verrons plus tard.
Le langage serait relativement pauvre si on n'avait que ces types à notre disposition. Il permet en fait de construire des types nouveaux grâce à des types précédemment déclarés. Ce sont les structures et les énumérations. Il existe un autre type de structures, que sont les unions, dont on ne parlera pas.

4.2.1 Les types entiers

Il existe cinq types d'entiers :
- **char**
- **short**

– **int**
– **long** (ou **long int**)
– **long long** (ou **long long int**)

Le dernier type ne fait pas partie de la norme C ANSI.
Contrairement aux nombres entiers en mathématiques, les valeurs que peuvent prendre des variables de type entier sont limitées. Par défaut, les variables ayant un des types ci-dessus sont *signées* (elles peuvent prendre des valeurs positives comme des valeurs négatives, chacune dans un certain intervalle), sauf **char**, dont le caractère signé ou non dépend de la configuration du compilateur. Pour Visual Studio, **char** est non signé. Pour que le compilateur considère qu'elles ne peuvent pas prendre de valeurs négatives, on place le mot réservé **unsigned** devant le type. Par exemple :

```
unsigned int a;
```

Dans ce cas, la variable est de type *non signé*. Le mot clé **signed** existe pour renforcer le fait qu'un variable est *signée*, mais c'est plus une information pour quelqu'un qui lit un code qu'autre chose. En général, on n'utilise pas ce mot clé, sauf dans des cas très particuliers.
Voici un tableau qui résume les intervalles de définition pour chaque type, signé ou non, pour un ordinateur de type x86 32 bits avec un compilateur standard (*gcc*, *MSVC*) :

nom	bits	signé	non signé
char (ns)	1	-128,...,127	0,...,255
short (s)	2	-32768,...,32767	0,...,65535
int (s)	4	-2147483,...,2147483647	0,...,4294965
long (s)	4	-2147483,...,2147483647	0,...,4294965
long long (s)	8	-2^{63},...,$2^{63}-1$	0,...,$2^{64}-1$

Dans ce tableau, après le nom du type, (s) signifie que par défaut, le type est signé et (ns) qu'il est non signé.
Les valeurs numériques peuvent être exprimées en base 10 (notation standard), en base 8 (on fait précéder la valeur par le chiffre 0 : 014 vaut 12 en base 10) et en base 16 (on fait précéder la valeur par le chiffre 0 puis la lettre x : 0x2f vaut 47 en base 10).

4.2.2 Les types flottants

Les types flottants sont utilisés pour simuler des réels. La dénomination "flottant" vient de la représentation des nombres réels, sous la forme *mantisse/exposant*, où la position de la virgule ne bouge pas, contrairement à la notation décimale. Par exemple, 0.01 se note 1.0E-2, tandis que 15.1 se note 1.51E1.
Comme pour les entiers, ils ne peuvent prendre des valeurs que dans un certains intervalle. Il existe trois types de nombres flottants, chacun pouvant prendre des valeurs dans un intervalle plus ou moins grand. Ce sont les types :

– **float**
– **double**
– **long double**

L'intervalle de définition de ces types, de même que leur précision, sont très grands. Nous ne les donnerons pas ici. Il faut juste remarquer que :

- Les calculs sur les nombres flottants sont plus lents que les calculs sur les nombres entiers.
- Le type **float** n'est pas adapté à du calcul de précision, car il introduit beaucoup d'erreurs d'arrondi. Le type **double** est en général très suffisant.

Les notations des valeurs numériques de nombres flottants sont les suivantes :

- En notation décimale, on utilise le point. On met un signe + (celui-ci étant optionel) ou - devant le nombre. On peut omettre le 0 si le nombre est strictement inférieur à 0 (comme -.38)
- En notation exponentielle, on utilise la lettre E, précédée de la mantisse et suivie de l'exposant : 12.3E-4, par exemple.

4.2.3 Les structures

Le langage C permet de définir de nouveaux types, à partir de types précédemment déclarés. On utilise pour cela le mot clé **struct**. La sytaxe est la suivante :

```
struct nouveau_type
{
  type1 champ1;
  type2 champ2;
  ...
};
```

Il consiste donc en le mot clé **struct**, suivi du nom du type que l'on choisit, on ouvre une accolade. Puis on énumère les *champs* (ou bien les *membres*) de la structure, sous la forme *type / nom_du_champ / point virgule*. On ferme l'accolade et on n'oublie pas le point virgule final, qui est très souvent oublié. On peut mettre autant de champ que l'on veut. Par exemple :

```
struct point3d
{
  int x;
  int y;
  int z;
};
```

Le nom du type est alors **struct point3d**, et non juste **point3d**. On verra comment rémédier à ce problème dans la section 4.2.5.
Une fois une variable ayant pour type une structure déclarée, il faut pouvoir initialiser ses champs. La syntaxe est la suivante :

```
nom_var.champ = valeur;
```

Par exemple, avec notre structure **struct point3d** :

```
struct point3d point;

point.x = 12;
point.y = 26;
point.z = 3;
```

La même notation est utilisée pour récupérer la valeur d'un champ. Par exemple :

```
struct point3d point;
int           x;

point.x = 12;
point.y = 26;
point.z = 3;

x = point.x;
```

Comme il a été dit, on peut utiliser dans une structure n'importe quel type qui a été précédemment déclaré. On peut ainsi définir le type *segment* utilisant notre structure *point3d* :

```
struct segment
{
  struct point3d p1;
  struct point3d p2;
};
```

Donc, pour accéder à la coordonnée x de p1, on utilise naturellement la syntaxe :

```
struct segment seg;

seg.p1.x = 2;
```

4.2.4 Les énumérations

Les énumérations permettent de définir des variables de type entier, ayant des valeurs qui se suivent automatiquement. La syntaxe est la suivante :

```
enum nouveau_type
{
  variable1,
  variable2,
  ...
  variablen
};
```

Elle consiste donc en le mot clé **enum**, suivi du nom du type que l'on choisit, on ouvre une accolade. Puis on énumère les variables, sous la forme *identifiant / virgule*, sauf pour la dernière variable. On ferme l'accolade et on n'oublie pas le point virgule final, qui est très souvent oublié. Très souvent, les identifiants sont en majuscule. La valeur de ces variables est donnée automatiquement : la première variable vaut 0, la seconde 1, etc... Comme pour les structure, le nom du nouveau type est **enum nouveau_type**
Il existe d'autres syntaxes que l'on ne verra pas ici.

4.2.5 Déclaration de types

Comme on l'a vu, pour les structures et les énumérations (de même que pour les unions, que nous n'avons pas abordé), le nom des type est un peu long. Le

langage C permet néanmoins de rajouter un nom de type à l'aide du mot clé **typedef**. Son utilisation est la suivante :

```
typedef nom_type nouveau_nom_type;
```

Par exemple :

```
typedef double mon_double;

typedef struct segment mon_segment;
typedef struct point3d point3d;

struct point3d
{
  int x;
  int y;
  int z;
};

struct segment
{
  point3d p1;
  point3d p2;
};
```

On doit noter deux choses :

1. L'"ancien" nom est toujours valide. On n'a fait que rajouter un nom de type.
2. L'utilisation de **typedef** est une déclaration de type. Elle peut se faire avant la définition du type lui-même, s'il n'existe pas, comme dans l'exemple ci-dessus avec le type **struct** *segment*, où la définition (là où on a explicitement nommé les champs de la structure) est donnée après la déclaration.

4.2.6 Chaîne de caractère statique

Une chaîne de caractère est une succession de caractères. Les chaîne de caractère dites statique sont de la forme : "caractères". Les caractères sont des lettres ou des chiffres entourées de guillements. Elles sont utilisables telles quelles (comme valeur passée à une fonction par exemple) ou sont stockées dans une variables de type **char ***. Par exemple :

```
char *str;

str = "une chaine";
```

Il existe des caractères spéciaux, utilisés en général pour un affichage :

- \n : pour aller à la ligne.
- \t : pour insérer une tabulation.
- \r : pour revenir en début de ligne.
- %* : pour certaines valeurs de '*', la fonction **printf** permet d'afficher certaines valeurs.

4.2.7 Précisions sur le type char

Le type **char** sert aussi à représenter les caractères, en tant que caractères ASCII :

http ://www.tableascii.com/

Elle sert à faire le lien entre la touche de votre clavier et le nombre à utiliser pour représenter cette touche. Par exemple, la lettre 'a' a pour code ASCII 97 (en décimal). Le caractère 'a' est représenté ainsi en C : une apostrophe, le caractère, puis l'apostrophe :

```
char a;

a = 'a';
printf ("valeur : %c %d\n", a, a);
```

Ceci va afficher dans le terminal :

```
valeur : a 97
```

Certains caractères de la table ASCII ne peuvent pas se représenter avec une lettre ou un autre symbole, comme le retour à la ligne ou bien les caractères accentués. On utilise pour cela des caractères d'échappement, qui sont constitués d'une apostrophe, puis d'un anti-slah, puis d'une lettre ou un nombre, puis d'une apostrophe. Ainsi, le retour à la ligne est défini ainsi :

```
char retour_ligne;

retour_ligne = '\n';
```

D'où la présence de ce caractère quand on utilise la commande **printf**. On ne dresse pas de liste des caractères d'échappement. Le seul vraiment utilisé de façon courante est le retour à la ligne.

4.2.8 Les tableaux

Les structures peuvent "représenter" un ensemble de variables pouvant être de type différent. Cela peut etre assez restrictif lorsque l'on veut représenter un nombre fini, mais grand d'objets de même type. Les tableaux pallient à cette restriction. La syntaxe de la déclaration d'un tableau est la suivante :

```
type nom_var[nbr_elts];
```

Le type **type** doit être connu et *nbr_elts* doit être connu au moment de la compilation (en général une valeur numérique). Par exemple, pour déclarer un tableau d'entiers de 100 éléments :

```
int tab[100];
```

Les éléments sont indexés à partir de 0. Pour initialiser la valeur du quatrième élément à 12, on utilise la syntaxe suivante :

```
int tab[100];

tab[3] = 12;
```

Noter qu'on utilise 3 et non 4, vu que les indices commencent à 0.
On peut bien sur récupérer cette valeur en la stockant dans une variable du même type que les éléments du tableau :

```
int tab[100];
int x;

tab[3] = 12;
x = tab[3];
```

4.3 Opérateurs et expressions

Le langage C dispose des opérateurs classiques : les opérateurs arithmétiques, les opérateurs relationnels, les opérateurs logiques et l'opérateur d'affectation. Il dispose aussi d'une large palette d'opérateurs d'incrémentation.

4.3.1 Opérateurs arithmétiques

Les opérateurs arithmétiques sont ceux couramment utilisés en mathématiques. On peut donc faire une addition (+), une soustraction (-), une multiplication (*) ou une division (/) sur des variables ou des valeurs de type entier ou flottant.
L'opérateur modulo (%), qui n'agit que sur des entiers, donne le reste de la division euclidienne. Il est à noter que la division de deux entiers donne un entier (le résultat de la division entière).
Les priorités des opérateurs sont celles que l'on trouve en mathématiques.

Remarque sur la division et les nombres flottants. Le code suivant :

```
double d = 1 / 2;
```

déclare et initialise un nombre flottant à 0.0 et non pas 0.5. La raison est que le compilateur commence par faire la division entre 1 et 2, qui sont des entiers. Donc il effectue la division Euclidienne, dont le résultat est 0. Puis il stocke cette valeur dans la variable *d*. Pour corriger ce problème, il faut que l'un des opérandes au moins soit un flottant, soit en utilisant la notation des nombres flottants, soit en utilisant le transtypage (transformation d'un type en un autre). Par exemple :

```
double d1 = 1.0 / 2;
double d2 = 1 / 2.0;
double d3 = 1.0 / 2.0;
double d4 = (double)1 / 2;
```

4.3.2 Opérateurs relationnels

Le langage C permet de comparer des valeurs. On trouve l'égalité (==), la différence (!=), inférieur ou égal (<=), inférieur strictement (<), supérieur ou égal (>=) et supérieur strictement (>). Il faut bien faire attention à l'égalité de deux valeurs et ne pas oublier un =.
Le résultat fourni par un opérateur de relation est normalement un booléen (un type qui ne peut prendre que deux valeurs, soit vrai, soit faux). Dans le langage C, le résultat est un entier, avec la convention suivante :

- Un résultat faux correspond à la valeur entière 0.
- Un résultat vrai correspond à une valeur entière non nulle (1, -1, 15, etc...).

Un exemple de comparaison : $a + 5 > b/3$.
Ces opérateurs sont en général utilisés dans les instructions de contrôles (*cf* Section 5) pour des tests.

Remarque importante sur les nombres flottants : faire une comparaison entre deux nombres flottants avec l'opérateur == peut mener à un mauvais résultat. En effet, des calculs successifs entraînent des erreurs d'arrondi et le résultat théorique voulu n'est plus du tout le même que le résultat numérique obtenu. Ce qu'il faut faire est voir si la valeur absolue de la difference entre ces deux nombres flottants est suffisamment petite ou non.

4.3.3 Opérateurs logiques

Ces opérateurs agissent sur des valeurs pouvant être vraie ou fausses (donc des valeurs retournées par les opérateurs relationnels que l'on vient de voir). On trouve le *et logique* (&&), le *ou logique* (||), la *négation* (!).
La convention reste la même :

- Un résultat faux correspond à la valeur entière 0.
- Un résultat vrai correspond à une valeur entière non nulle (1, -1, 15, etc...).

En particulier un opérande d'un opérateur logique peut être un entier quelconque.

4.3.4 Opérateur d'affectation

L'affectation est faite grace au symbole =. Le seul point à mentionner est que l'on ne peut faire intervenir des expressions que dans le membre de droite, comme pour $c = b + 3$. L'instruction $c + 3 = b$ n'a pas de sens et provoquera une erreur lors de la compilation.

4.3.5 Opérateurs d'incrémentation

On rencontre très souvent dans les codes les expressions $i = i + 1$, ou bien $n = n - 2$, qui incrémente ou décrémente une variable.
Le langage C a ainsi introduit des opérateurs unaires qui permettent d'écrire ces expressions plus simplement. Les opérateurs sont les suivants :

- ++ : incrémente de 1.
- −− : décrémente de 1.

L'utilisation la plus simple est la suivante :

```
int i;

i = 0;
i++;
```

A la fin de la dernière ligne, i vaudra $0 + 1$, c'est-à-dire 1. On peut bien sûr utiliser cette opérateur à l'intérieur d'une instruction. Par exemple :

```
int i, n;

i = 5;
n = i++ - 5;
```

Ceci a pour effet de donner la valeur $5 - 5 = 0$ à n est ensuite d'incrémenter i. La position de l'opérateur d'incrémentation est importante, car utiliser ++i, qui est une expression valide, dans une expression peut changer la valeur finale. Par exemple, en reprenant l'exemple ci-dessus :

```
int i, n;

i = 5;
n = ++i - 5;
```

On commence par incrémenter i puis on retranche 5. la valeur finale de n est dont $6 - 1 = 5$, et non 0. Donc $++i$ a pour valeur celle de i **après** incrémentation, tandis que $i++$ a pour valeur celle de i **avant** incrémentation.
La plupart du temps, on utilise ces opérateurs dans les boucles (voir la section 5), ou dans les indices de tableaux, ou les opérations arithmétiques sur les pointeurs.
Il existe aussi des opérateurs d'incrémentation dits *élargies*. Ce sont des opérateurs binaires, qui premettent d'incrémenter ou de décrémenter d'une valeur de plus de 1. On peut aussi utiliser la division et la multiplication, les opérateurs de manibulation de bits, etc... En voici les exemples les plus courants :

- $i+=2$ est équivalent à $i = i + 2$
- $i-=2$ est équivalent à $i = i - 2$
- $i*=2$ est équivalent à $i = i * 2$
- $i/=2$ est équivalent à $i = i/2$

5 Les instructions de contrôles

Jusqu'à présent, on ne peut que créer des variables et leur assigner des valeurs, c'est-à-dire pas grand chose. Comme la plupart des langages informatiques, le langage C permet en plus de tester des valeurs, et de faire des boucles. La syntaxe des tests ou boucle est particulière en C. Contrairement à d'autres langages, une balise de début de test ou de boucle est utilisée, mais il n'y a pas de balise de fin. Par exemple, en Scilab, on utilise la syntaxe suivante pour afficher les valeurs de 1 à 10 :

```
for i = 1 : 10
  disp(i);
end
```

L'instruction **end** dit que les instructions qui doivent être exécutées dans la boucles sont comprises entre le **for** et ce **end**.
En C, la syntaxe, que nous allons voir plus loin, est :

```
int i;

for (i = 1; i <= 10; i++)
  printf ("%d\n", i);
```

Autrement dit, aucune balise n'indique la fin de l'instruction **for**. Comment faire alors quand 2 instructions doivent s'exécuter dans chaque itération de la boucle ?
Pour cela, on introduit la notion de bloc d'instructions, qui est une partie du programme qui contient des instructions, ces instructions commençant après une balise ouvrante et se terminant avant une balise fermante, exactement ce qui manquait pour la boucle **for**.

5.1 Blocs d'instructions

Un bloc d'instructions est une suite d'instructions placées entre { et }. Les instructions à l'intérieur d'un bloc sont quelconques (mais néanmoins valides...). On peut créer un ou plusieurs blocs à l'intérieur d'un bloc. Voici un exemple de deux blocs :

```
{ }
{
  int i;

  i = 3;
}
```

Bien sur, le premier bloc, vide, est ici inutile et le deuxième peut être remplacé juste par ses deux instructions.
Une instruction simple (déclaration de variable, affectation de variable, etc...) doit être terminée par un point virgule. Ainsi,

```
{
  int i;

  i = 3
}
```

est un bloc non valide car la dernière instruction ne se termine pas par un point virgule.
On peut ajouter autant de blocs d'instructions dans un bloc déjà existant. Par exemple :

```
{
  int i;

  i = 3;
```

```
  {
    int j;

    j = i + 3;
  }
}
```

La portée d'une variable (c'est-à-dire le lieu du programme où la variable peut être utilisée) commence au moment où elle est déclarée, et se termine lorsque le bloc est fermé. En particulier elle peut être utilisée dans tout sous-bloc mais pas dans un sur-bloc. De plus on peut déclarer deux variables ayant le même nom dans deux blocs différents. Par exemple :

```
{
  int i;

  i = 3;
  printf ("%d\n", i);

  {
    int i;

    i = 4;
    printf ("%d\n", i);
  }
  printf ("%d\n", i);
}
```

va afficher respectivement 3, 4 et 3. En effet, la deuxième variable i remplace la première dans le sous bloc. Elle est détruite à la fin du sous bloc. Juste après la fin du sous bloc, c'est la première variable qui est à nouveau valide.

5.2 Instructions conditionnelles

Les instructions conditionnelles permettent d'exécuter des portions de code lorsqu'une condition est réalisée. Il existe trois types d'instruction conditionnelle, mais seules les deux principales seront présentées.

5.2.1 L'instruction if

La syntaxe est la suivante :

```
if (test)
  instructions
```

test est une expression dont la valeur est booléenne (0 signifie *faux* et une valeur non nulle signifie *vrai*). *instructions* est une instruction ou un bloc d'instruction si plusieurs instructions sont à exécuter si *test* est vrai.
Si on a besoin de l'alternative, la syntaxe est la suivante :

```
if (test)
  instructions
```

```
else
  instructions
```

Là encore, si plusieurs instructions sont à exécuter, dans la partie **if** ou la partie **else**, un bloc d'instructions est nécessaire. Par exemple :

```
{
  int i;

  /* code qui modifie i */

  if (i == 3)
    printf ("i vaut 3\n");
  else
  {
    printf ("i ne vaut pas 3\n");
    printf ("i vaut %d\n", i);
  }

  if (i != 4)
  {
    printf ("i ne vaut pas 4\n");
    printf ("i vaut %d\n", i);
  }
  else
    printf ("i vaut 4\n");
}
```

5.2.2 L'instruction switch

TODO : A faire

5.3 Instructions de boucle

Les boucles servent à exécuter des instructions un certains nombre de fois. Souvent ce nombre n'est connu qu'au moment de l'exécution. Les boucles permettent de gérer ceci. Nous allons voir 2 types de boucle (il y en a trois), les plus utilisées : la boucle **for** et la boucle **while**.
Les boucles peuvent servir à calculer des quantités mathématiques telles que $\sum_{i=i_0}^{n} u_i$, ou $\prod_{i=i_0}^{n} u_i$, ou bien encore les termes d'une suite. Elles permettent de remplir un tableau, de lire le contenu d'un fichier, etc...

5.3.1 L'instruction for

La syntaxe de la boucle **for** est la suivante :

```
for (instruction1 ; valeur ; instruction2)
  instruction3
```

instruction1, *instruction2* et *instruction3* sont des instructions et *valeur* est une valeur (numérique, d'une variable ou bien retournée par une fonction). En

général, c'est le résultat d'un test, par exemple $i \leq n$.
La boucle ci-dessus a le comportement suivant :

1. *instructions1* est exécutée.
2. Tant que la valeur *valeur* est différente de 0, *instruction3* est exécutée, puis *instruction2* est exécutée.
3. dès que *valeur* vaut 0, les itérations s'arrêtent.

L'instruction *intruction3* doit être remplacée par un bloc d'instructions si plusieurs instructions doivent être exécutée dans la boucle (comme pour *if*). Dans ce cas, à chaque itération de la boucle, toutes les instructions de ce bloc sont exécutées.
instruction1, *instruction2*, *instruction3* et *valeur* sont tous optionnels. Dans le cas où *instruction3* n'est pas présent, il faut mettre un point-virgule après la parenthèse fermante, ou bien un bloc d'instruction vide. Voici un exemple pour calculer la factorielle de 5 :

```
int i;
int res;

res = 1;
for (i = 1; i <= 5; i++)
  res = res * i;
```

On peut mettre à la place de *instruction1* et *instruction2*, plusieurs instructions, séparées par des virgules. On peut ainsi re-écrire le code du calcul de factorielle 5 de manière plus concise :

```
int i;
int res;

for (i = 0, res = 1; i < 5; ++i, res *= i) { }
```

ou bien encore plus concis :

```
int i;
int res;

for (i = 0, res = 1; i < 5; res *= ++i) { }
```

5.3.2 L'instruction while

La syntaxe de l'instruction **while** est la suivante :

```
while (valeur)
  instruction
```

valeur est, comme pour **for** une valeur numérique, d'une variable ou bien retournée par une fonction et c'est en général le résultat d'un test.
La boucle **while** ci-dessus a le comportement suivant :

1. tant que la valeur *valeur* est différente de 0, *instruction* est exécutée.
2. dès que *valeur* vaut 0, les itérations s'arrêtent.

L'instruction *intruction* doit être remplacée par un bloc d'instructions si plusieurs instructions doivent être exécutée dans la boucle (comme pour *if*). Dans ce cas, à chaque itération de la boucle, toutes les instructions de ce bloc sont exécutées.
Contrairement à la boucle **for**, la syntaxe n'inclut pas une instruction qui est exécutée avant la première itération, ni à la fin de chaque itération. Il faut les rajouter soi-même. Néanmoins, les deux type de boucles sont équivalentes. On peut ainsi re-écrire le calcul de factorielle 5 ainsi :

```
int i;
int res;

res = 1;
i = 1;
while (i <= 5)
{
  res *= i;
  i++;
}
```

ou bien de manière plus concise :

```
int i;
int res;

res = 1;
i = 0;
while (++i <= 5)
{
  res *= i;
}
```

ou bien

```
int i;
int res;

res = 1;
i = 0;
while (i < 5)
{
  res *= ++i;
}
```

6 Découpage de programme : les fonctions

Les fonctions permettent d'exécuter des parties de programmes. Comme les fonctions en mathématiques, elles peuvent prendre des paramètres. En C, une fonction peut ne renvoyer aucune valeur.
Comme pour les variables, pour être utilisée, une fonction doit être préalablement déclarée. Puis elle doit être définie pour que son code soit exécuté.

6.1 Déclaration d'une fonction

La syntaxe de la déclaration d'une fonction est la suivante :

```
type_retour nom_fct (param\'etres);
```

Ceci est le *prototype* de la fonction *nom_fct*. *type_retour* est un type connu. Une fonction qui renvoie une valeur devra s'assurer que ce qu'elle renvoie est une valeur de type *type_retour*. *nom_fct* est le nom de la fonction et est un identificateur. *paramètres* sont les paramètres de la fonction. On ne peut pas déclarer deux fonctions avec le même nom dans un programme.
La syntaxe des paramètres est la suivante :

```
type1 var1, type2 var2,...
```

c'est-à-dire des couples type / identificateur, séparés par une virgule.
Lorsque la fonction ne renvoie rien, *type_retour* doit avoir la valeur **void**, qui est un mot clé du C. De plus, si une fonction ne prend pas de paramètres, soit on ne met rien entre les parenthèses, soit on met **void**.
Voici quelques exemples de déclaration de fonctions :

```
voif f1(void);

void f2(int i);

double f3(void);

int f4(float f, int j);

char f5 (int i, long double d, short j);
```

6.2 Définition d'une fonction

La définition d'une fonction consiste à écrire les instructions que cette fonction va exécuter lorsqu'elle sera utilisée. On dit aussi *implenter* ou *implémenter* une fonction. La syntaxe de la définition est la suivante :

```
type_retour nom_fct (param\'etres)
{
  instructions
}
```

c'est-à-dire la déclaration de la fonction sans le point-virgule final, et on fait suivre un bloc d'instruction, avec accolades obligatoires.
Si le type de retour est différent de **void**, alors la fonction doit retourner une valeur. Pour cela on utilise l'instruction **return**, suivie d'une valeur, suivie d'un point-virgule.

ATTENTION : contrairement aux structures de contrôle, même s'il n'y a qu'une seule instruction, le bloc d'instruction est obligatoire. De plus, la règle des blocs s'appliquent : si une variable est déclarée dans une fonction, et une autre variable de **même nom** est déclarée dans une autre fonction, ces deux

variables non aucun lien entre elles, elles n'existent que dans leur bloc respectif. Ceci s'applique aussi aux paramètres des fonctions.

On peut ainsi, calculer la factorielle d'un nombre dans une fonction, puis l'utiliser dans la fonction *main*. Voici un programme complet (voir la section 6.3 pour l'utilisation de la fonction **printf**) :

```
#include <stdio.h>

int factorielle(int n)
{
  int res;

  if (n < 0)
  {
    printf ("Erreur : valeur negative : %d\n", n);
    printf ("-1 est retourne\n");
    return -1;
  }

  for (res = 1; n; res *= n--) { }

  return res;
}

int main()
{
  printf("valeur de factorielle 5 : %d\n",
         factorielle(5));

  return 0;
}
```

6.3 La fonction *printf*

La fonction **printf** permet d'afficher des messages dans la console. Pour afficher un message basique, la syntaxe est la suivante :

```
printf(chaîne de caractères);
```

Par exemple

```
int main ()
{
  printf("bonjour\n");

  return 0;
}
```

Pour des messages évolués, affichant des valeurs de variables, la syntaxe est la suivante :

```
printf(chaîne de caractères, val1, val2,...);
```

et la chaîne de caractère foit être "formatée" : là où on veut mettre la valeur de "val1", on met %* dans la chaîne, où '*' est remplacé par un ou plusieurs caractères :

- pour un entier : %d
- pour un entier non signé : %u
- pour un char : %c
- pour un long : %ld
- pour un double : %f
- pour une chaîne de caractère : %s

Il en existe bien d'autres. Des exemples peuvent être trouvés dans la section 6.2 ou 6.4.

6.4 La fonction *main*

La fonction *main* doit apparaître lorsqu'un programme est créé. De plus elle ne doit apparaître qu'une seule fois, meme si le programme est constitué de plusieurs fichiers. La raison est que, lorsqu'un programme est exécuté, l'ordinateur cherche l'endroit où il doit commencer ce programme. Cet endroit est la fonction *main*.

Le prototype de la fonction *main* ne peut avoir que deux formes :

```
int main ();

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

c'est-à-dire, soit aucun paramètre, soit deux paramètres, le premier étant un entier, le deuxième étant un tableau de chaînes de caractères (ou bien un double pointeur sur un **char**). L'intérêt de la deuxième syntaxe est de pouvoir passer des paramètres au programme lorsque celui-ci est exécuté dans une console. Par exemple, la commande *cd* permet de changer de répertoire. Elle est suivie du nom du répertoire. C'est grâce à la fonction main que le programme *cd* peut récupérer le nom du répertoire en question.

Pour faire de même (c'est-à-dire récupérer les paramètres passés en ligne de commande), on utilise les paramètres de la fonction *main*. *argc* indique le nombre de paramètres passés en ligne de commande, et *argv* est un tableau de chaîne de caractères stockant ces paramètres.

Le nom du programme étant lui-même sur la ligne de commande, il est aussi référencé dans le tableau *argv* et *argc* est toujours supérieur ou égal à un. Par exemple :

```
int main (int argc, char *argv[])
{
  printf ("nombres de parametres  : %d\n", argc);
  printf ("nom du programme       : %s\n", argv[0]);

  return 0;
}
```

La fonction *main* renvoie aussi une valeur (son type de retour est **int**). Cette valeur peut-être utilisée pour savoir si le programme s'est bien terminé ou bien s'il y a eu une erreur. C'est à celui qui écrit le programme de s'assurer que la valeur retournée est la bonne. Par convention, si on retourne la valeur 0, le programme s'est bien déroulé. Sinon, il y a eu une erreur. Pour eviter tout mélange dans les valeurs, le standard permet l'utilisation de deux constantes : **EXIT_SUCCESS** et **EXIT_FAILURE**. Pour les utiliser, il faut inclure le fichier *stdlib.h*. Par exemple, avec notre fonction factorielle :

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

int factorielle (int n)
{
  int res;

  if (n < 0)
  {
    printf ("Erreur : valeur negative : %d\n", n);
    printf ("-1 est retourne\n");
    return -1;
  }

  for (res = 1; n; res *= n--) { }

  return res;
}

int main ()
{
  int fact;

  fact = factorielle (5);
  if (fact < 0)
  {
    printf ("erreur dans le calcul de la factorielle\n");
    return EXIT_FAILURE;
  }

  printf ("valeur de factorielle 5 : %d\n",
          factorielle (5));

  return EXIT_SUCCESS;
}
```

Table des matières