Chapitre 6

Les variables (types, déclaration, portée)

Nous allons à présent parler de ce qu'on appelle, en C, les variables.
Qu'est-ce qu'une variable ? Pour faire court et simple, c'est un emplacement mémoire, de taille limitée, qui est à la disposition du programme, pour que le programmeur puisse y mémoriser des données dont la valeur n'est pas fixe.
Au cours de ce chapitre, nous verrons comment créer des variables, quels sont les types de variables que l'on peut utiliser, quelles sont les valeurs maximales et minimales qu'elles peuvent contenir, ...

Avant toute chose, je tiens à attirer votre attention sur le fait que ce que l'on appelle "variable", en C, est différent de ce qui est appelé de la même façon en TI-BASIC. Aux yeux du BASIC, une variable est un fichier, qui apparaît dans le menu VAR-LINK, qui peut être utilisé par plusieurs programmes, et même en dehors de tout programme.
En C, une variable est interne au programme, n'apparaît pas en dehors du programme, et ne peut pas être utilisée ailleurs que dans le programme. C'est de ce genre de variable dont nous allons parler dans ce chapitre, et c'est ce que nous nommerons "variables" dans la suite de ce tutorial.

Il existe plusieurs types de variables, en C ; au cours de ce chapitre, nous nous limiterons aux variables arithmétiques, qui sont celles dont nous avons l'usage pour des programmes simples. Plus tard, nous parlerons des tableaux, puis des pointeurs, tant redouté par les débutants, et enfin des structures, qui permettent de regrouper plusieurs données dans une seule variable, mais je pense qu'il vaut mieux ne pas apporter trop de nouveautés à la fois, et commencer par le plus simple. Les chaînes de caractères ne constituent pas un type en elles-même ; à cause de leur importance, nous leur consacrerons un chapitre entier, une fois que nous aurons étudié les tableaux et les pointeurs.

I:\ Les différents types de variables, leurs modificateurs, et leurs limites

En C, il existe deux familles de variables concernées par ce chapitre : les entiers, et les réels (souvent appelés "nombres en virgule flottante", communément abrégé en "flottants"). Ces deux familles sont découpées en plusieurs tailles, permettant de stocker des nombres plus ou moins grands.

A: Les entiers

1: Ce que dit la norme

Tout d'abord, précisons que l'écriture suivante :

```
sizeof(type)
```

renvoie la taille, en octets, que prend une variable du type précisé ; cela pourra nous servir dans la suite de ce chapitre. Je vous encourage d'ailleurs, lorsque vous avez besoin d'utiliser le nombre d'octets que fait un type de variables, à toujours employer sizeof plutôt que la taille que vous pensez que fait une variable. Ainsi, votre code sera plus facilement portable sur d'autres machines. (d'autant plus que la taille, en octet de chaque type n'est pas fixé par la norme !).
Sachez aussi que sizeof n'est pas réservé aux variables entières seulement.

La taille en octets des types de variables entières n'est pas fixé par la norme ANSI, ni par la norme GNU.
La seule chose qui est précisée, c'est que `sizeof(char)` doit être inférieure ou égale à `sizeof(short)`, qui doit être inférieure à `sizeof(int)`, qui doit être inférieure ou égale à `sizeof(long)`, qui doit elle-même être inférieure ou égale à `sizeof(long long)`, sachant que int correspond généralement au mot-machine.

2: Ce qu'il en est pour nos calculatrices

Maintenant que nous avons vu ce que dit la norme, voyons ce qu'il en est pour nos calculatrices, et ce que signifie les différents types énoncés ci-dessus.

Écriture complète	Écriture concise, généralement utilisée	Taille, avec les options par défaut	Intervalle de valeurs
`signed char`	`char`	8 bits, 1 octet	de –128 à 127
`unsigned char`	`unsigned char`	8 bits, 1 octet	de 0 à 255
`signed short int`	`short`	16 bits, 2 octets	de -32 768 à 32 767
`unsigned short int`	`unsigned short`	16 bits, 2 octets	de 0 à 65 535
`signed int`	`int`	16 bits, 2 octets	de -32 768 à 32 767
`unsigned int`	`unsigned int`	16 bits, 2 octets	de 0 à 65 535
`signed long int`	`long`	32 bits, 4 octets	de -2 147 483 648 à 2 147 483 647
`unsigned long int`	`unsigned long`	32 bits, 4 octets	de 0 à 4 294 967 296
`long long int`	`long long`	64 bits, 8 octets	de -9223372036854775808 à 9223372036854775807
`unsigned long long int`	`unsigned long long`	64 bits, 8 octets	de 0 à 18446744073709551615

Comme on peut le remarquer, avec les options par défaut, ou, plutôt, sans options particulières, les int sont codés comme des short. Cela dit, il est possible de passer au compilateur une option lui indiquant de considérer les int comme des long...
Je vous conseille donc de toujours utiliser des short à la place des int (autrement dit, lorsque vous voulez une valeur sur 16 bits), afin d'être plus précis, et, lorsque vous voulez une valeur sur 32 bits, utilisez des long. C'est ce qui se fait toujours ou presque parmi les développeur pour TI ; autant que vous suiviez cette habitude.

Une autre habitude qu'il peut être bon de prendre est d'utiliser un type correspondant aux valeurs que vous pouvez vouloir mémoriser dedans. Par exemple, n'utilisez par des long long pour stocker un nombre compris entre 0 et 10 ; ce serait une perte de temps ridicule, surtout sachant que les long long ne sont pas utilisables directement par le microprocesseur (il travaille sur 32 bits au maximum), et doivent être "traduit" afin d'être utilisables, ce qui prend du temps.
Les nombres sur un ou deux octets sont ceux avec lesquels le processeur travaille le plus rapidement ; ensuite viennent les nombres sur 4 octets, et enfin ceux sur plus, qui reçoivent un traitemant particulier.

Aussi, si vous savez que vous ne travaillerez qu'avec des valeurs positives ou nulles, autant utiliser une variable de type unsigned. Cela permettra au compilateur de vous prévenir si, par mégarde, vous utiliser une valeur négative, et votre code source sera plus compréhensible : la personne qui le lira saura que la variable qu'elle a sous les yeux ne passe jamais en dessous de 0, et qu'il n'est pas nécessaire de réfléchir à ce qu'il se passerait si cela arrivait.

Enfin, le tableau ci-dessus vous présente une écriture que j'ai surnommé "complète", et une écriture plus concise. Comme noté en tête de colonne, ce sont les écritures concises qui sont généralement utilisées : un programmeur C ne se fatiguera pas à écrire quelque chose d'inutile, et profitera au maximum des fonctionnalités et de la souplesse du langage !

B: Les flottants

1: Ce que dit la norme

Pour ce qui est des nombres flottants, la norme définit ceci : `sizeof(float)` doit être inférieure ou égale à `sizeof(double)`, qui doit elle même être inférieure ou égale à `sizeof(long double)`.

2: Ce qu'il en est pour nos calculatrices

Sur nos machines, les trois types float, double, et long double sont tous équivalents.
On utilisera généralement le type `float`, car c'est celui qui correspond le mieux aux noms de fonctions du TIOS.

Un float, comme nous pourrons les utiliser sur TI, est compris entre 1e-999 et 9.999999999999999e999, avec une précision de 16 chiffres significatifs.
Pour les curieux, le format d'encodage de flottants sur TI n'est pas celui généralement utilisé sur PC, qui demande trop de calculs pour l'encodage et le décodage, mais SMAP II BCD.

II:\ Déclaration, et Initialisation, de variables

A: Déclaration de variables

Déclarer une variable est une opération extrêmement simple, qui nécessite deux choses :

- Le type de de variable que l'on souhaite créer,
- et le nom que l'on souhaite lui donner.

Le type de variable que l'on souhaite créer dépend, bien évidemment, de nos besoins, et nous le choisirons parmi ceux proposés plus haut.
Le nom de la variable, quand à lui, doit répondre à quelques règles, que nous étudierons plus loin. Pour l'instant, il nous suffit de savoir qu'il peut contenir des lettres, et le caractère underscore (généralement, Alt-Gr + 8, sur les claviers azerty).

On commence par écrire le type de la variable souhaitée, et on le fait suivre par le nom que l'on veut lui donner, en terminant le tout, bien entendu, par un point virgule, pour marquer la fin de l'expression.
Par exemple, pour déclarer une variable contenant un entier de type short, nommée 'a', nous utiliserons la syntaxe suivante :

```
short a;
```

Pour déclarer une variable de type nombre flottant, nommée 'pi', nous utiliserons la même syntaxe, comme suit :

```
float pi;
```

Il en va de même pour les autres types de variables ; pour cette raison, nous ne donnerons pas d'exemples supplémentaires.

Notez qu'il est parfaitement possible de définir plusieurs variables, du même type, sur une seule ligne logique. Pour cela, vous précisez le type de variables, et ensuite, la liste des noms de variables, séparés par des virgules, comme indiqué ci-dessous :

```
char a, b, c,
           d, e;
```

GCC, et donc, TIGCC, permet de déclarer des variables un peu n'importe où dans son code source, du moment qu'on les déclare avant de les utiliser. Cela dit, je vous conseille de regrouper vos déclarations de variables en début de bloc, c'est-à-dire après les accolades ouvrantes, afin de plus facilement les retrouver. Si vous avez dix variables à déclarer, autant toutes les déclarer au même endroit, plutôt qu'une parci et une parlà ! De la sorte, si vous devez modifier quelque chose, vous vous y retrouver plus facilement. De même, vous n'aurez pas à vous demander au moment d'utiliser une variable que vous savez avoir déclaré si sa déclaration est avant ce que vous voulez écrire (ce qui correspond à ce que le compilateur attend), ou après (ce qui générerait une erreur à la compilation).

B: Initialisation de variables

On appelle "initialisation" d'une variable le fait de lui donner une valeur pour la première fois. Pour donner une valeur à une variable (on dit "affecter" une valeur à une variable), on utilise l'opérateur d'affectation, '=' (Le symbole égal).
Cet opérateur affecte à la variable placée à sa gauche le résultat de l'expression placée à sa droite. Au cours de ce chapitre, nous n'utiliserons que des nombres comme expression, mais nous verrons plus tard que le terme d'expression regroupe bien plus que cela.

Notez que l'opérateur d'affectation, en C, fonctionne dans la sens inverse de celui que vous avez pu utiliser si vous avez programmé en TI-BASIC ; celui du TI-BASIC affecte la valeur à sa gauche dans la variable à sa droite.

Il est possible d'initialiser une variable en deux endroits : au moment de sa déclaration, et ailleurs dans le source.
En fait, seule l'affectation à la déclaration est un cas particulier, et constitue véritablement une initialisation. Donner une valeur à une variable à un autre moment qu'à sa déclaration se fait toujours de la même façon, que ce soit ou non la première fois.

1: A la déclaration

Comme nous l'avons laissé entendre, il va nous falloir utiliser l'opérateur d'affectation, en ayant à sa gauche notre variable, et à sa droite la valeur que nous souhaitons donner à celle-ci.
Cela dit, nous voulons que cette affectation se fasse à la déclaration de la variable, et, pour déclarer une variable, nous avons vu qu'il fallait écrire son type à gauche de son nom.

En combinant les deux, nous obtenons pour, par exemple, déclarer une variable de type long, nommée 'mavar', en l'initialisation à 234567, cette syntaxe :

```
long mavar = 234567;
```

De la même façon, pour déclarer une valeur de type flottant nommée pi, et approximativement égale à la valeur de PI, nous utiliserons ceci :

```
float pi = 3.141592;
```

Comme vous pouvez le remarquer, le séparateur entre la partie entière, et la partie décimale n'est pas, en C, la virgule, mais le point (que nous appelons alors "point décimal", afin de bien le faire correspondre à sa fonction).

Ici encore, il est possible de déclarer, et d'initialiser, plusieurs variables sur une même ligne logique, toujours en les séparant par des virgules :

```
short a=20, b=40, c, d=56;
```

Les variables a, b, et d seront initialisées ; la variable c ne le sera pas. Ceci est parfaitement possible, et ne pose absolument aucun problème au compilateur.

2: Ailleurs

Pour affecter une valeur à une variable ailleurs dans le code source, que ce soit ou non la première fois que nous le faisons, nous utiliserons la même syntaxe, mais sans préciser le type de la variable (ce qui reviendrait à la redéclarer, et une variable ne peut pas être déclarée deux fois !).
Une chose importante : en C, pour pouvoir utiliser une variable, il faut qu'elle ait été déclarée. Affecter une valeur à une variable revient à l'utiliser ; il faut donc, avant de l'initialiser, bien l'avoir déclarée ! Si vous essayez d'utiliser une variable non déclarée, le compilateur générera une erreur.
Une autre remarque est que le type d'une variable est fixé à sa déclaration, et ne peut pas changer. Par exemple, si vous avez déclaré une variable comme étant d'un des types entiers, vous ne pouvez pas lui affecter une valeur contenant un point décimal ! Cela aussi provoquerait une erreur de la part du compilateur. De la même façon, si vous avez déclaré une variable comme étant unsigned, vous ne pouvez pas lui affecter une valeur négative.

La syntaxe étant similaire à celle déjà étudiée, nous ne donnerons qu'un seul exemple. Supposons que nous ayons une variable de type unsigned short, nommée varshort, qui ait été déclarée correctement.
Pour lui affecter la valeur 10, nous utiliserons la syntaxe suivante :

```
varshort = 10;
```

Il en va de même pour les autres types simples, ceux que nous avons vu au cours de ce chapitre.

III:\ Quelques remarques concernant les bases, et le nommage des variables

A: Une histoire de bases

Tous les jours, nous sommes amenés à utiliser des nombres, et nous le faisons généralement en base 10, c'est à dire en utilisant des chiffres allant de 0 à 9.

En C, il est possible d'utiliser la base 2 (binaire), la base 8 (octale), la base 10 (décimale), et la base 16 (hexadécimale).
Les nombres en binaire sont préfixés de 0b (Le chiffre zéro, suivi de la lettre 'b'), les nombres en base 16 de 0x ou 0X, et les nombres en base 8 d'un 0 tout seul. La base 10 est la base par défaut, ce qui explique pourquoi les nombres exprimés en décimal ne sont pas préfixés. Les nombres que nous avons utilisés pour haut dans ce chapitre sont donc, comme la logique le voudrait, en base 10, qui est la base que nous avons l'habitude d'utiliser.

Le binaire est souvent utilisé car il permet de représenter de façon claire des données conformément à la façon dont elles sont codées dans la machine ; par exemple, le courant passe, ou ne passe pas, la charge magnétique est positive, ou négative...
A chaque fois, on a deux états possibles. Le binaire code ses nombres à l'aide de 0 ou de 1, c'est-à-dire deux états possibles, qui correspondent aux deux états possibles.
Chaque chiffre d'un nombre est appelé "digit". Pour le langage binaire, il s'agit de "binary digit", communément abrégé en "bit". Un bit correspond à la plus petite unité d'information possible.

L'héxadécimal, est utilisé pour faciliter l'écriture et la recopie de données binaires. En effet, un digit hexadécimal peut coder 16 valeurs, ce qui correspond à 4 bits. Ainsi, un mot d'un octet sera codé sur deux digits, en hexadécimal, au lieu de huit en binaire, diminuant fortement le risque d'erreurs à la recopie.
Les nombres en base 16 sont codés en utilisant les dix chiffres habituels, de 0 à 9, et les six premières lettres de l'alphabet, de A à F.

La base octale n'est que rarement utilisée. Elle code ses nombres en utilisant les chiffres allant de 0 à 7.

Nous ne détaillerons pas ici comment convertir un nombre d'une base dans une autre, afin de ne pas rendre ce chapitre trop long, d'autant plus que ce n'est pas le sujet. Cela dit, si vous n'avez jamais étudié cela en cours, ou que vous pensez avoir besoin de révisions, vous pouvez consulter cette page.

B: Nommage des variables

La norme C est assez souple pour ce qui concerne le nommage des variables, mais aussi assez stricte sur certains points :

Un nom de variable

- Peut contenir des lettres de A à Z, et de a à z, ainsi que les dix chiffres de 0 à 9, et le caractère underscore (tiret souligné : '_').
- Doit commencer par une lettre.

Vous devez préter attention au fait que le C est sensible à la case, ce qui signifie que les majuscules et les minuscules sont reconnues comme des caractères différents. Par exemple, mavar, MAVAR, et mAvAr sont trois variables différentes !

Je vous conseille d'adopter une habitude concernant le nommage des variables, et de vous y tenir ; vous vous y retrouverez plus facilement de la sorte.
Voici plusieurs exemples d'habitudes de nommages ; libre à vous d'en retenir une, ou d'en choisir une autre.

- Séparer les 'mots' constituant le nom de la variable par des underscore ; par exemple : ceci_est_une_variable.
- Mettre la première lettre de chaque 'mot' en majuscule : CeciEstUneVariable.
- Même chose, sauf pour le premier mot (il me semble que c'est ce qui est généralement fait en langage JAVA) : ceciEstUneVariable.
- Préfixer le nom de la variable par quelques lettres indiquant son type ; par exemple, l_ pour un long, f_ pour un float, ... (On se rapproche de la notation hongroise beaucoup utilisée par les programmeurs sous interface windows utilisant les MFC).

Cela dit, je vous conseille aussi de choisir des noms qui ne soient pas trop longs (car fatiguant à taper, et vous risquer d'en oublier la moitié), ni trop court (afin que vous sachiez à quoi sert la variable).
Une habitude généralement prise est d'utiliser des noms de variables en une lettre (en particulier 'i', puis 'j', 'k', ...) en tant que variables de boucles ; nous étudierons dans quelques chapitres ce que cela signifie ; si j'y pense, je repréciserai ceci à ce moment là.

IV:\ Portée des variables, et espace de vie

Comme nous l'avons déjà dit, nous devons impérativement déclarer les variables avant de pouvoir les utiliser. Cela dit, il y a quelques éléments supplémentaires qu'il peut être bon de connaître.

Plus haut, nous avons, très brièvement, parlé de la notion de bloc, qui correspond à tout ce qui est compris entre une accolade ouvrante et l'accolade correspondante. Par exemple, une fonction, telle _main, que nous avons déjà utilisé, est un bloc.
Pour bien illustrer ce fait, voici comment nous l'avons écrite il y a quelques chapitres :

```
void _main(void)
{ // Début d'un bloc
    ST_helpMsg("Hello World !");
    ngetchx();
} // Fin du bloc
```

Cela dit, un bloc ne correspond pas nécessairement à une fonction. Il est d'ailleurs possible d'imbriquer les des blocs, et nous le ferons extrêmement souvent dans l'avenir.

Il existe deux genres différents de variables, dont la portée, c'est-à-dire la portion de programme où elles sont visibles, est différente : les variables locales, et les variables globales.

A: Variables locales

Une variable locale est une variable qui est déclarée dans une fonction, ou, plus généralement dans un bloc.
Plusieurs règles définissent les zones où les variables locales existent. Nous allons voir chacune de ces règles, en leur associant à chaque fois un exemple, afin de bien les illustrer.

Une variable locale existe à partir du moment où vous la déclarez, que ce soit en début de bloc si vous avez suivi le conseil que j'ai donné plus haut, ou plus loin dans le corps du bloc si vous avez préféré déclarer votre variable seulement au moment de son utilisation, jusqu'à la fin du bloc dans lequel elle est déclarée :

```
{ // Début du bloc
    // Instructions diverses, ou aucune instruction.
    // la variable a n'a pas été déclarée, et ne peut donc pas être utilisée.
    short a;
    // La variable a existe, à présent.
} // Fin du bloc => La variable a cesse d'exister.
```

Il en va de même quelque soit la profondeur du bloc ; qu'il s'agisse ou non d'un bloc enfant n'a absolument aucune influence sur ce point, comme nous pouvons le remarquer ci-dessous :

```
{
    // La variable a n'existe pas ; il est impossible de l'utiliser
    {
        // La variable a n'existe toujours pas.

        short a;
        // A présent, la variable a existe ; il devient possible de l'utiliser.

        // Fin du bloc dans lequel la variable a a été déclarée.
        // La variable a cesse donc d'exister.
    }

    // La variable a n'existe pas :
    // Puisqu'elle a été crée dans un bloc enfant, elle a été détruite
    // à la fin du bloc enfant qui l'avait créé.
}
```

Une variable locale est visible depuis les blocs enfants de celui dans lequel elle a été déclarée, à partir du moment où ces blocs enfants sont localisés après sa déclaration :

```
{
    short a;
    // La variable a, déclarée, peut être utilisée dans ce bloc
    {
        // On est dans un bloc enfant de celui dans lequel
        // la variable a a été créée
        // On peut donc utiliser la variable a ici.


        // La variable a a été créée dans un bloc père de celui dont on est
        // à la fin. La variable a n'est donc pas détruite.
    }
    // La variable a existe toujours, et peut donc être utilisée.

    // On arrive à la fin du bloc dans lequel la variable a a été déclarée
    // La variable a est donc détruite.
}
```

Cela dit, si vous définissez dans un bloc une variable de même nom qu'une variable définie dans le bloc père, la variable du bloc père sera masquée par la variable du bloc courant, ce qui signifie que la variable que vous utiliserez sera celle du bloc enfant, et non celle du bloc père. Ceci est un comportement susceptible de vous induire en erreur, par exemple si vous ne pensez pas au fait que vous avez déclaré une variable de même nom que dans le bloc père !
L'exemple ci-dessous illustre bien ceci :

```
{
    short a;
    // On a déclaré une variable a, de type short.
    // On considérera que cette variable est la variable a n°1
    // On peut, naturellement, utiliser la variable a
    // ce sera a n°1 qui sera utilisée.
    {
        float a;
        // On a déclaré, dans ce bloc, une autre variable a, de type float
        // On considérera que cette variable est la variable a n°2

        // Si, dans ce bloc, on utilise une variable nommée a,
        // le compilateur utilisera la variable a n°2,
        // car la déclaration de a dans ce bloc
        // masque la déclaration de a déclarée dans le bloc père


        // On arrive à la fin du bloc qui a créé la variable
        // a n°2. Celle-ci est donc détruite.
    }
    // Si, maintenant, on utilise la variable a, ce sera la n°1,
    // puisqu'on est dans le bloc où c'est celle-ci qui a été déclarée.
    // Précisons que son contenu n'aura nullement été affecté par la
    // déclaration et l'utilisation de la variable a n°2 dans le bloc fils.

    // Fin du bloc dans lequel la variable a n°1 a été déclarée.
    // Elle va donc être détruite.
}
```

Le seul cas présentant un danger pour le programmeur est le dernier ; les autres ne sont qu'une question de logique et d'habitude, à partir du moment où l'on sait qu'une variable doit être déclarée avant de pouvoir être utilisée.

B: Variables globales

Nous allons à présent pouvoir étudier le cas des variables globales. Une variable "globale" est une variable qui est définie en dehors de tout bloc.
Attention, il ne s'agit pas d'une variable qui n'est définie dans aucun bloc, bien au contraire ! En effet, une variable globale est utilisable dans tous les blocs du programme, ou, du moins, dans tous les blocs qui suivent sa définition.

Par exemple, si nous définissons une variable globale avant la fonction _main, cette variable sera valable dans le code de la fonction _main, comme on pourrait s'y attendre. Mais, si l'on définit d'autres fonctions que _main, comme nous apprendrons à le faire dans quelques chapitres, cette variable sera aussi accessible depuis ces autres fonctions, ce qui n'était pas possible avec des variables locales.

Une variable globale se définit exactement de la même façon qu'une variable locale, mis à part le fait que sa déclaration se fait en dehors de tout bloc. Son initialisation peut se faire à la déclaration, comme pour les variables locales.
Ensuite, on peut l'utiliser de la même façon que les variables locales : la seule différence est, j'insiste, que sa déclaration se fait en dehors de tout bloc.

Deux choses sont à noter, concernant les variables globales :

- Si le programme n'est ni archivé, ni compressé, les variables globales concervent leur valeur entre chaque exécution du programme.
- Utiliser des variables globales fait augmenter la taille du programme, car elles sont mémorisées dans celui-ci, contrairement aux variables locales, qui sont créées au moment de leur déclaration.

Je finirai par une petite remarque : utiliser des variables globales est souvent considéré comme assez sale. En effet, le fait que ces variables puissent être masquées par des variables locales (exactement de la même façon qu'une variable locale d'un bloc peut masquer une variable déclarée dans un bloc père) et soient accessibles depuis n'importe quel point du programme rend leur utilisation assez "risquée", si je puis me permettre d'utiliser un terme aussi fort.
Conjointement avec la seconde remarque que j'ai fait juste au dessus, cela a pour conséquence que les variables globales sont assez peu utilisées ; je vous conseille de suivre cette habitude, et de ne les utiliser que lorsqu'il n'est pas possible de faire autrement, c'est-à-dire, rarement !

Il reste encore des éléments concernant les variables dont je n'ai pas parlé, en particulier, l'utilité et l'utilisation des mots-clés auto, register, static, extern, volatile, et const, mais je ne pense pas qu'ils soient utiles au niveau où nous en sommes, et vous n'en n'aurez pas besoin avant quelques temps. Je n'ai pas envie de rendre ce chapitre, déjà assez long, et probablement quelque peu rébarbatif, encore plus long, surtout pour quelque chose qui n'est pas extrêmement utile ; nous les présenterons donc lorsque nous en en aurons l'usage.