ABRÉGÉ DU LANGAGE C

Cet abrégé contient surtout ce qui est nécessaire à la création d'applications simples pour petits systèmes embarqués (µP/µC 8/16 bits sans noyau temps réel). Les possibilités du C peu employées pour l'usage visé ne sont pas ou peu développées ici.

1) GÉNÉRALITÉS

Le langage C est à la fois un langage de haut niveau (il dispose des structures de base pour la programmation structurée –*voir § 10*-) et un langage de bas niveau car il permet l'accès aux données que manipulent les µP/µC (bit, octet, adresse).
Le langage C convient pour des applications très diverses depuis le petit système embarqué avec un µC 8 bits jusqu'au gros programme pour station de travail.

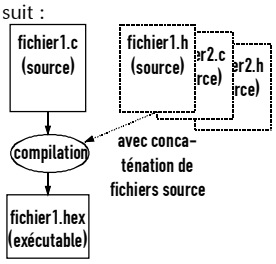
1.1) CHAÎNE DE COMPILATION

Les différents logiciels intervenant dans la création du fichier final, utilisé pour l'exécution finale ou la mise au point, constituent une **chaîne de développement ou chaîne de compilation**.

La compilation permet de traduire le texte source en une suite d'instructions pour le µP/µC.

La constitution d'une chaîne de compilation dépend du µC cible et du compilateur.

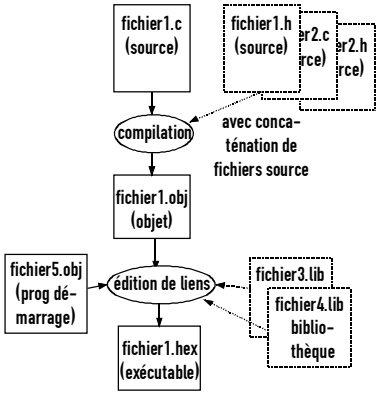
Avec certains µCs de faibles ressources et certains compilateurs, un seul fichier de départ est utilisé, ou, ce qui revient presque au même, plusieurs fichiers qui sont réunis avant la phase de compilation proprement dite, avec des directives placés dans les différents fichiers.
La chaîne de compilation peut se représenter comme suit :



Le compilateur CC5X pour PIC fonctionne de cette façon.
Les extensions varient d'un compilateur à l'autre.

Les fichiers source sont des fichiers texte ; le fichier exécutable est souvent un fichier binaire (impossible à ouvrir avec un éditeur de texte). Le fichier exécutable est utilisé pour la programmation de la mémoire programme cible (externe dans le cas d'un µP ou interne à un µC) ou pour la mise au point.

La plupart des compilateurs permettent d'utiliser des sous-programmes fournis déjà compilés et réunis en bibliothèques. Ils utilisent de plus un petit programme de démarrage qui exécute certaines tâches décrites plus loin. La chaîne de compilation peut alors se représenter comme suit :

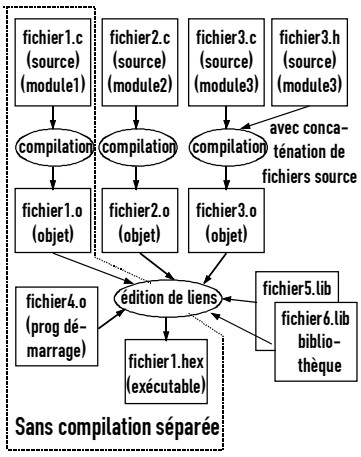


Le fichier objet est un fichier binaire (impossible à ouvrir avec un éditeur de texte).

Les fichiers bibliothèque sont des recueils de sous programmes déjà compilés (*Voir § Constitution d'un programme / Fonction / Fonction en bibliothèque et § Fonctions / Fonctions de la bibliothèque standard)*

Beaucoup de compilateurs permettent une **conception modulaire** : le programme source est découpé sur plusieurs fichiers qui sont traduits séparément puis réunis pour former un fichier exécutable. On utilise parfois de ce cas le terme de **compilation séparée**. Cette possibilité est très utilisée pour les applications de tailles suffisamment importantes.

Flux ***simplifié*** pour l'élaboration d'un programme exécutable :



La chaîne de compilation fait intervenir plusieurs logiciels :

- éditeur de texte pour l'écriture du programme source
- compilateur (compiler)
- éditeur de liens ou lieur (linker)

Le compilateur peut se décomposer en plusieurs logiciels, voir §19, Détail de la compilation.

L'édition de lien (linkage) réunit tous les fichiers objet (obtenus par compilation de programmes C utilisateur, extraits de bibliothèques) pour former le fichier exécutable utilisable par la suite (programmation ou déboguage).
Avec la plupart des compilateurs, l'édition de liens a toujours lieu, même pour un programme sur un seul module, car elle fait appel au programme de démarrage indispensable (parfois de façon transparente à l'utilisateur). Elle extrait aussi des bibliothèques les sous-programmes (fonctions) nécessaires. Une bibliothèque système peut être utilisée, parfois de façon transparente à l'utilisateur. Le programme source du programme de démarrage est le seul nécessairement écrit en langage d'assemblage.
Voir §19, Détail de la compilation.

Avec un **environnement de développement intégré (EDI)**, tous les logiciels présentés ci-dessus sont disponibles depuis la même interface graphique. Un EDI comprend en plus un gestionnaire de projet qui permet de déclarer tous les fichiers source pour un même fichier exécutable.

Des options sont disponibles pour chacune des phases.
Dans le cas d'un développement d'une application pour ordinateur avec un environnement de développement intégré (EDI) la compilation et l'édition de liens peuvent rester transparentes à l'utilisateur, avec l'utilisation des options par défaut.
Dans le cas d'une application pour système embarqué à µP, il faut nécessairement fixer des options ou entrer des commandes pour une adaptation aux particularités du système (adresses mémoire disponible, ...)

Le C dispose de directives qui permettent de sélectionner ou d'exclure certaines parties du fichier source pour la compilation. Ceci permet de compiler un même fichier pour différentes cibles ou différents environnements (certaines parties du programme ne sont utilisées qu'avec une certaine cible) ou pour inclure ou non des instructions pour la mise au point (voir ci-dessous). Les modifications à apporter au programme source pour compiler une version ou une autre sont très limitées.
Voir §23, compilation conditionnelle.

1.2) DÉBOGUAGE (MISE AU POINT)

Le déboguage consiste à exécuter le programme partie par partie pour vérifier son bon fonctionnement (pas à pas, entre points d'arrêt, ...).

Pour faciliter le déboguage, on peut utiliser les possibilités de compilation conditionnelle mentionnées plus haut. Il est possible par exemple d'intégrer dans le programme des instructions pour générer des impulsions pour vérifier la durée d'exécution de certaines parties du programme puis compiler ensuite la version définitive sans ces instructions.
Voir §23, Compilation conditionnelle.

La mise au point est différente selon que le µP/µC cible est le même ou non que celui utilisé pour la compilation.

Dans le cas d'application pour ordinateur, le déboguage s'effectue souvent avec le même EDI que pour la compilation.
Dans le cas d'application pour système embarqué développée sur ordinateur (compilation croisée), le déboguage peut ne pas s'effectuer depuis l'EDI de compilation. Plusieurs types de déboguage sont possibles.

DÉBOGAGE POUR SYSTÈME EMBARQUÉ

On commence en général à vérifier que le logiciel est correct d'un point de vue fonctionnel avec un logiciel de simulation sur ordinateur.

Le déboguage est ensuite réalisé avec le système cible et un ordinateur sur lequel est exécuté le logiciel de déboguage. La liaison entre les deux peut s'effectuer de différentes façons :

- Liaison directe pour **déboguage in situ**. Le µC dispose alors de connexions spéciales pour une liaison série (RS232 ou USB) et des ressources nécessaires pour une exécution par partie du programme (point d'arrêt). Cette possibilité commence à se développer sur des µCs récents. La mémoire programme est d'abord programmée par la liaison série. Puis les commandes de déboguage sont envoyées par cette même liaison. Le déboguage in situ n'est possible qu'avec des µC intégrant de la mémoire flash facile à programmer et reprogrammer depuis l'ordinateur de déboguage.
- Utilisation d'un **émulateur**. Celui-ci est connecté à l'ordinateur par une liaison série (cas le plus fréquent) ou parallèle et au système par une sonde qui se substitue au µP/µC et qui se connecte sur un support de CI. Un émulateur possède de la mémoire RAM dans laquelle est téléchargé le programme à déboguer. Cette mémoire se substitue à la mémoire ROM du système cible.
- Utilisation d'une **carte d'évaluation**. La carte comprend les ressources pour développer des applications (ports d'E/S, connecteurs pour des extensions, etc.) et des ressources supplémentaires pour la liaison série vers l'ordinateur. Certaines ressources du système normalement réservées à l'utilisateur sont utilisées pour le déboguage (une partie de la mémoire RAM, une interruption, etc.). Certaines cartes ne détournent que peu de ressources disponibles à l'utilisateur grâce à des techniques évoluées. Après déboguage, pour une réalisation finale autonome, il faut concevoir une carte avec le même schéma, sauf en ce qui concerne les ressources spécifiques pour le déboguage (liaison série, ROM pour le moniteur, ...).
  Industriellement, une carte d'évaluation n'est souvent utilisée, comme son nom l'indique, que pour évaluer la faisabilité d'un projet avec un µC, avant d'acheter un émulateur.

1.3) EXÉCUTION FINALE

Sur un ordinateur, le programme exécutable créé après la compilation est lancé par le système d'exploitation.

Pour un système embarqué, il faut programmer la mémoire du système (qui peut être intégrée au µC) à partir du fichier créé par la compilation. La programmation dépend du type de mémoire ; elle peut s'effectuer
- Lors de la fabrication du µC (ROM)
- Par l'utilisateur avec un programmateur de mémoire (PROM, EPROM, EEPROM)
- In situ avec une liaison série depuis un ordinateur (EEPROM, FLASH)

2) CONSTITUTION D'UN PROGRAMME

2.1) PROGRAMME

Programme : traitements sur des données (constantes et variables) réalisés à l'aide d'instructions.
Une instruction est écrite avec les **opérateurs** intégrés du C (+, -, =, ...). Plusieurs instructions peuvent être groupées pour former des **macros** (macros instructions) ou bien des sous programmes ou **fonctions**. *voir § 2.4.*

Lors de l'exécution du programme, les instructions en mémoire sont exécutées :
- **séquentiellement** (dans l'ordre de leurs emplacements en mémoire programme, ce qui correspond à l'ordre des instructions dans le programme source)
- **avec des rupture de séquence** qui dépendent de conditions

Pour permettre ces deux types d'exécution, les instructions font partie de **structures** (qui peuvent être imbriquées ou accolées) :
- **linéaires** (instructions exécutées séquentiellement)
- **itératives** (répétition de l'exécution d'un traitement sous certaines conditions ; boucle)
- **alternatives** (deux possibilités s'excluant mutuellement).

Les deux dernières structures sont réalisées avec des **instructions de contrôle**. Ce sont ces instructions qui permettent une **programmation structurée.**

Plusieurs instructions peuvent être groupées au sein d'un **bloc** délimité par **{ ... }**
Les blocs sont utilisés :
- avec les instructions de contrôle (l'emploi de blocs n'est pas nécessaire dans tous les cas)
- pour la définition des fonctions
- pour limiter la visibilité de certaines variables (un bloc est alors imbriqué dans un autre)

Voir plus loin pour chacun des points mentionnés.

2.2) DONNÉE

Donnée : constante ou variable.

Chaque donnée correspond à 1 ou plusieurs mots machines en mémoire ou registre du système cible (8, 16, 32, ... bits).

Avant toute utilisation, une donnée doit être créée (réservation de mémoire ou registre par le compilateur). Cette création peut s'effectuer par :

- une définition (cas le plus fréquent)
- une réservation dynamique de mémoire (*voir §24, Données « dynamiques »*).

Une définition crée une donnée en lui associant un **identificateur** (nom). *(Sur la syntaxe d'une définition, voir §7.2)*. Cette donnée pourra ensuite être manipulée par son identificateur dans la suite du programme.

Les différentes opérations réalisées par une définition de donnée sont :

- association d'un nom (identificateur) à une zone mémoire réservée pour ranger la donnée. L'espace mémoire peut correspondre à un ou plusieurs mots contigus avec une adresse de base (inconnue au programmeur).
- définition du domaine de valeur de la donnée
- définition des opérations possibles sur la donnée
- définition du domaine de visibilité ou portée de la donnée

On utilise parfois le terme déclaration à la place de définition, alors qu'une déclaration a un sens légèrement différent. Voir § Données : variables & constantes / Définition des variables

2.3) INSTRUCTION

Instruction : sert à préciser le sens de déroulement d'un programme. Notion très étendue qui inclut :

- la définition d'une variable, d'une constante ou d'une fonction
- l'appel d'une fonction réalisant une action
- l'affectation d'une valeur à une variable d'après une expression (cette dernière pouvant contenir un appel de fonction)
- ...

Les instructions peuvent être exécutées :

- les unes après les autres
- selon certaines conditions grâce à des instructions de contrôle.

On peut ainsi réaliser des structures de:

- choix
- boucles

Instructions simples

Les instructions sont écrites l'une à la suite de l'autre séparées par **;**

Bloc d'instructions

Dans la définition des fonctions et dans les structures plusieurs instructions successives correspondent à un bloc qui commence par **{** et se termine par **}**

Instruction vide

Peut être indispensable dans certains cas *(voir §10.2, Structures / Instructions de contrôle / if)*. L'instruction ne contient qu'un ;
L'instruction vide peut être utile pour réaliser des boucles d'attente ou de temporisation.

EXPRESSION

Une expression est une séquence

- d'opérandes (variable, constante, valeur retournée par une fonction)
- d'opérateurs
- de séparateurs (ex : () pour les priorités)

qui spécifie un calcul.

En C, la notion d'expression est très large. Toute expression possède une valeur.

*Ex d'expressions : 2*x, 2*x+y, x=2*x+y, a<b.*

Les expressions sont évaluées selon certaines règles de conversion, de groupement, d'associativité et de priorité, qui dépendent des opérateurs utilisés, de la présence de parenthèses, et des types d'opérandes.

Voir §8, Expressions et affectations pour plus de détail.

QUELQUES INSTRUCTIONS SIMPLES

définition ou déclaration	int a ; ... *(voir §7, Données)*
appel de fonction	printf(''bonjour'') ; *(voir §12, Fonctions)*
affectation	y=expression ; contenant éventuellement des appels de fonctions

Certaines expressions sans l'opérateur d'affectation = correspondent à des instructions avec affectation. *Voir §9.5) opérateurs d'affectation.*

EXEMPLES D'INSTRUCTIONS SIMPLES D'AFFECTATION

- y=2*x+3 ; la valeur évaluée d'après l'expression 2*x+3 est affectée à y. y est une « lvalue » (valeur à gauche du signe =). y et x sont des variables préalablement déclarées. x possède une valeur.
- y=sin(x) ; la valeur renvoyée par la fonction sin() est affectée à y. x est le paramètre ou l'argument de la fonction. La fonction sin(u) doit avoir été définie ailleurs.
- y=carre(sin(x)) ; instruction d'affectation avec appels imbriqués de fonctions. La valeur renvoyée par sin() est transmises à carre() qui elle-même renvoie une valeur dans l'expression qui l'a appelée.

Une instruction complexe peut comprendre plusieurs opérateurs et fonctions.
Une instruction, au sens du C, est une notion très étendue (*voir plus loin*).

2.4) FONCTION

Fonction : sous-programme qui permet une conception modulaire.

Une fonction est utilisée pour augmenter la lisibilité du programme source et/ou diminuer la taille du code produit.

Un programme est constitué de :

- **un programme principal + des sous programmes (fonctions)**
- **un ou plusieurs gestionnaires d'interruptions (interrupt handler ou interrupt service routine, ISR).**

En C le "programme principal" est une fonction qui se nomme obligatoirement **main**. « main » est le **point d'entrée** (entry point) du programme. L'exécution du fichier binaire exécutable commence par cette fonction (en réalité pas tout à fait –*voir plus loin § Détail de la compilation / Éditeur de liens*). Les gestionnaires d'interruption correspondent eux aussi à des fonctions.
Les fonctions précédentes peuvent elles aussi contenir des appels de fonctions.

Un programme C est un souvent un ensemble de fonctions.

ex :
Extrait de programme constitué d'une suite d'appels de fonctions :
...
AcquisitionAN1() ;
Calcul() ;
Affichage()
Tempo() ;
...

Une fonction :

- **réalise une action**, éventuellement d'après les informations qui lui sont transmises (ex : affichage message). Dans d'autres langages, on emploie le terme **procédure** dans ce cas.
- **renvoie une valeur** dans l'expression qui l'a appelée, d'après les informations transmises (ex : résultat d'un calcul)

Les informations transmises à une fonction s'appellent des **paramètres**.

Les fonctions peuvent être :

- déjà compilées dans des bibliothèques fournies avec le compilateur
- créées par l'utilisateur

Avec chaque compilateur, sont livrées des bibliothèques. Une bibliothèque regroupe plusieurs fonctions déjà compilées (gain d'occupation mémoire, meilleure organisation, possibilité de ne pas fournir le programme source...). Tous les compilateurs fournissent une bibliothèque « standard » (qui peut correspondre à plusieurs fichiers bibliothèques). Ceci permet d'étendre les possibilités du C avec ses opérateurs intégrés.

2.5) MACRO

Une macro-instruction ou macro est un nom (identificateur) qui correspond à un ensemble d'instructions.

Une macro est utilisée pour augmenter la lisibilité du programme.

Une macro doit d'abord être définie. Elle peut ensuite être appelée.

Des macros prédéfinies sont souvent fournies avec les compilateurs. Les définitions sont souvent placées dans des fichiers en-tête .h.

UTILISATION

Pour utiliser une macro, il suffit d'écrire son nom dans le programme source.

ex : InitLiaisonSerie ;

Il existe des macros avec paramètres. Leur utilisation est identique à celle d'une fonction sans valeur de retour (procédure).

Voir §13, Macros.

CODE PRODUIT

Lors de la 1ère phase de la compilation (préprocesseur), chaque occurrence du nom de la macro placée dans le programme source est remplacée par les instructions correspondantes. Dans les autres phases de la compilation, les instructions sont traduites en code exécutable.

3) C ET NORMALISATION

Le langage C a été normalisé par l'ANSI (American National Standard Institute). Il existe quelques différences entre le C ANSI et le C originel développé par ses créateurs.
Seul le C ANSI est mentionné dans ce document, sauf pour quelques cas particuliers (pour pouvoir lire d'anciens programmes).

Tous les compilateurs pour systèmes embarqués proposent des extensions au C ANSI, notamment pour :

- accéder directement aux registres avec adresses du µP/µC comme de simples variables
- les mécanismes de passage de paramètres aux fonctions.

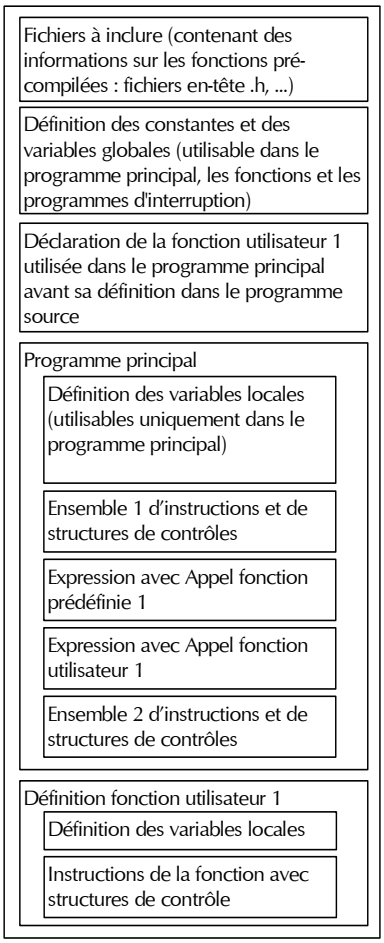
La plupart des compilateurs pour systèmes embarqués ont des options pour se conformer uniquement au C ANSI. Ceci n'offre aucun intérêt. Un programme développé pour un µP ne sera jamais portable à 100% sur un autre µP (registres différents, etc.)
Ce présent document mentionne quelques extensions.

4) STRUCTURE SIMPLIFIÉE D'UN PROGRAMME SOURCE

Un exemple de structure d'un fichier source est donné ci-dessous. Ce type de fichier a fréquemment l'extension .c.

On peut trouver aussi d'autres parties, non mentionnées ci-dessous :

- déclarations de données externes
- directives de compilation (*voir plus loin, inclusion de fichiers*)



Le programme principal commence par
void main (void) {
et se termine par
}

Pour un système embarqué, le programme principal comporte une boucle sans fin, dans laquelle est placée la plupart des instructions.

Chaque bloc correspondant à une définition de fonction, ... est entre **{ }**

Exemple de programme source très simplifié :
Toutes les lignes de ce programme sont détaillées dans la suite de cet abrégé.

```
#include <pic.h> /* fichier en-tête de définition des registres du µC*/

/* définition d'une variable globale */
unsigned char VarGlob1 ;

/*déclaration de fonctions définies plus loin */
unsigned int ConversionAN(void) ;
void AfficheLCD(unsigned int);

void main(void) /* programme principal */
{
 unsigned int R_CAN, /* variables locales */
              Temperature ;
InitPorts() ; // Fonction pour l'initialisation des
              // ports
InitLCD() ; // Fonction pour l'initialisation d'un
              // afficheur à cristaux liquides

for( ; ;) /* boucle sans fin */
        {
        R_CAN = ConversionAN() ;
        Tempature = (R_CAN * A) – B ;
        AfficheLCD(Temperature);
        ...
        }
}
/* définition de Fonct1 */
unsigned int ConversionAN (void)
{
...
}
...
/* définition du gestionnaire d'interruption */
interrupt void GestInter(void)
{
...
}
```

4.1) INCLUSION DE FICHIERS

Utilisé essentiellement pour les fichiers en-tête .h contenant les déclarations des fonctions et les macros.

Syntaxe
#include <NomFichier> (NomFichier entre crochets) fichier dans le répertoire standard du C (pour un fichier en-tête d'une fonction prédéfinie)
#include " NomFichier " fichier dans le répertoire courant
#include " c:\Chemin d'Accès\NomFichier " le fichier est dans le dossier spécifié

#include est une directive du préprocesseur. Voir §19.1, Détail de la compilation / Préprocesseur.

4.2) 2 FICHIERS SOURCE POUR UN MÊME MODULE

De nombreux programmeurs préfèrent placer toutes les déclarations de variables et les déclarations des fonctions utilisateur sur un fichier en-tête .h inclus dans le fichier « code » .c lors de la compilation grâce à la directive #include. Le fichier code est plus court et dans certains cas plus lisible.
Voir §25.4, Programme sur plusieurs fichiers.

5) GÉNÉRALITÉS SUR LA SYNTAXE

5.1) MISE EN PAGE

Elle est libre. Seule la lisibilité du programme guide la mise en page. Les règles souvent adoptées sont :

- 1 seule instruction par ligne
- indentation pour faire apparaître les structures de contrôle (*voir §10, Structures / Instructions de contrôle de flux et § 25.3, Indentations et position des {}*)

5.2) ÉLÉMENTS DU LANGAGE

Ils sont séparés par des **caractères inertes** : caractère espace, tabulation, commentaire (placé entre /* et */), caractère Entrée (ou retour chariot CR).
Les commentaires peuvent parfois être placés derrière // à condition de tenir sur une ligne (convention C++ acceptée par certains compilateurs C).

Les éléments du langage sont :

- les **identificateurs** (nom de variables, fonctions, ...) : lettres, _ et chiffres (sauf en

début car un chiffre en début correspond à une constante)
- les **mots réservés** par le système (instructions de contrôle, types de variables, qqes opérateurs,...)
- les **constantes** (commencent nécessairement par un chiffre)
- les **chaînes littérales** (*voir plus loin*)
- les **opérateurs**
- les **séparateurs** : {} pour un bloc d'instructions, [] pour un tableau, () pour une fonction, ...

Le C fait la distinction entres majuscules et minuscules

Mot réservés

Le tableau ci-dessous présente les mots réservés du C ANSI.

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

Chaque compilateur dispose de quelques mots réservés non normalisés pour permettre des extensions au C ANSI. Voir doc du compilateur utilisé.

Voir §25, Conseils d'écriture d'un programme source.

6) NOTATIONS UTILISÉES DANS L'ABRÉGÉ

Les mots placés entre < > doivent être remplacés par un des mots clés du C, un identificateur, etc.
Une exception est signalée dans laquelle les < > sont obligatoires.

Ce qui est placé entre [] signifie facultatif, sauf pour les tableaux et sauf mention contraire.

7) DONNÉES : VARIABLES & CONSTANTES

7.1) TYPE DE DONNÉES, PORTÉE, DURÉE D'EXISTENCE

TYPE DE DONNÉE

En langage C, toutes les données ont un type, mentionné dans la définition de la donnée.

Le **type** d'une donnée détermine :
- domaine de valeurs
- espace mémoire réservé dans le système cible
- opérations possibles et signification des bits en mémoire réservée (ex : MSB = bit de signe ou non)

Le type est défini par :
- la définition pour une variable ou une constante
- la règle d'écriture de la valeur pour une constante non définie (*voir §7.4), Constantes*)

Les types de données se décomposent en :
- **types simples ou primitifs** (voir ci-dessous)
- **types complexes ou construits à l'aide de types primitifs** (*voir §14, Types de données complexes*).

Ex de types
simples : entier signé sur 16 bits, entier non signé sur 8 bits
construits : tableau de 4 entiers non signés sur 8 bits

ATTRIBUTS DES DONNÉES

Chaque **donnée** possède plusieurs **attributs**. Les plus usuels sont :
- un type (déjà mentionné)
- une classe de stockage –ou classe d'allocation- (type d'emplacement en mémoire)
- une durée de vie (liée à la classe d'allocation) lors de l'exécution du programme (pendant tout le programme, durant l'exécution d'un bloc)
- une portée (visibilité ou domaine de validité) : partie du programme source où la donnée est accessible

Ces différents attributs sont fixés par :
- la syntaxe de la définition
- l'emplacement de la définition

PORTÉE

L'endroit où est définie une variable fixe sa portée, c'est-à-dire l'endroit où elle peut être utilisée dans le fichier source :

Variable globale (portée fichier) : définie en dehors de toute fonction ou du programme principal. La portée concerne tout le fichier source depuis l'endroit de sa définition jusqu'à la fin.
Une variable globale est utilisable dans le programme principal et dans les programmes d'interruptions.
Une même variable globale peut être utilisée dans plusieurs fichiers source (utilisable dans le cas de compilations séparées de fichiers source puis réunion des modules objet lors de l'édition de liens).

Variable locale (portée bloc) : utilisable uniquement par le bloc (généralement le corps d'une fonction -y compris main-) qui contient la définition et dans les blocs imbriqués.

DURÉE D'EXISTENCE

Durant toute l'exécution du programme (classe statique)
variables concernées :
- toutes les variables globales
- les variables locales dont on a forcé la classe d'allocation statique en ajoutant static devant le type de donnée lors de la définition *(static a aussi une autre signification dans la définition d'une variable globale)*

Les variables statiques sont allouées dans une zone mémoire réservée à ce type de variables.

Durant l'exécution d'un bloc (classe automatique)
Emplacements mémoire alloués lorsque le bloc ou la fonction commence à s'exécuter puis désalloués en fin.
Variables concernées : uniquement les variables locales
Les variables automatiques sont allouées sur la pile (sauf option contraire disponible uniquement pour certains µC de faibles ressources et sauf avec µC sans pointeur de pile manipulable par programme –PIC par ex-).

Durée de vie dynamique
Pour des variables créées « à la demande ». Par exemple, avec un logiciel interactif sur ordinateur, une nouvelle variable peut être créée suite à une saisie clavier et une validation.
La réservation de mémoire s'effectue à l'aide de fonctions spécifiques prédéfinies. En fin d'utilisation des variables, l'espace mémoire alloué est libéré lorsqu'il n'y en a plus besoin, toujours à l'aide de fonctions spécifiques.

Cette possibilité est utilisée pour les variables de type construit (*voir §24.1 Données « dynamiques » / listes liées*) dont on ne connaît pas le nombre d'éléments lors de l'écriture du programme.
Les variables dynamiques sont allouées sur le tas (heap).

Les données doivent être créées avant d'être utilisées.

7.2) TYPES SIMPLES DE DONNÉES

Type	Nb d'octets en mémoire	valeur / *résolution*
void		sans valeur
char (unsigned char)	1	valeur numérique (pouvant représenter le code d'un caractère) 0 à 255 / *1*
signed char	1	-128 à + 127 / *1*
int [1] (short int, short, signed short, signed int ...)	2	-32.768 à 32.767 *1*
unsigned int (unsigned short, ...)	2	0 à 65 535 *1*
long int (signed long int)	4	-2.147.483.648 à 2.147.483.647 *1*
unsigned long int	4	0 à 4.294.967.295 *1*
float	4	-3,4E38 à +3,4E38 *3,4E-38*
double	8	±1,7E308 *1,7E-308*
long double	10	±1,1E4932 *3.4E-4932*

*1 : la taille et les valeurs limites de **int**, dépendent souvent du système cible ; par exemple la taille peut être de 4 octets. **short int** correspond à 2 octets en C ANSI. Voir doc compilateur.*

Pour les types de données simples, l'**identificateur** correspond à la **valeur** de la donnée.

Ex : y=a+2. a : valeur (rangée dans une case mémoire)

Le C ANSI ne dispose pas de type booléen. Certains compilateurs proposent ce type de donnée (extension au C ANSI).

7.3) DÉFINITION & DÉCLARATION DES VARIABLES

Une **définition** permet de réserver de la place mémoire pour une variable.
Une **déclaration** n'est utilisée que dans le cas de variables globales utilisables sur plusieurs fichiers (compilation séparée). Une déclaration indique qu'une variable a été définie sur un autre fichier.
Voir §25.4 Conseils d'écriture d'un programme source / Programme sur plusieurs fichiers.

Remarque : les termes définition et déclaration sont souvent confondus.

Pour la forme complète des définitions, voir plus loin le §17, Structure d'une définition de donnée.

TYPE DE VARIABLE / PORTÉE / DURÉE D'EXISTENCE

La portée d'une variable est la partie du programme source où elle peut être utilisée.
La durée d'existence est la durée pendant laquelle un emplacement mémoire est réservé durant l'exécution du programme. Elle est liée à la classe d'allocation.

Le type de variable (globale/locale, etc.), la portée, la durée d'existence dépendent de la définition de la donnée et de son emplacement.

Type variable / Portée	Définition	Durée d'existence/ *Classe d'allocation*
Globale / tout fichier source [1]	en dehors de toute fonction	Exécution du prog./ *Statique*
Globale cachée [2] / fichier source	en dehors de toute fonction avec **static**	Exécution du prog./ *Statique*
Locale « rémanente »[3] /fonction	en début de fonction avec **static**[4]	Exécution du prog./ *Statique*
Locale / fonction ou bloc	en début de fonction ou bloc[4]	Exec fonction/ *Automatique*

1 : Définition normale dans un des fichiers, déclaration dans autre(s) fichier(s)
2 : la variable ne peut être utilisée dans un autre fichier source.
3 : la portée de la variable reste uniquement la fonction, mais sa classe d'allocation est statique ; sa valeur est donc conservée d'un appel à l'autre de la fonction (rémanence).
4 : Au tout début de bloc (variables locales) ; jamais après 1ère instruction autre que déclaration

Comme toute variable doit être définie avant d'être utilisée, les variables globales sont définies en début de fichier source.

LES DIFFÉRENTES DÉFINITIONS ET LEURS SYNTAXES

Définition simple
<type_1> <Var1>, <Var2>, ... ; // ex : int x, y ;
<type_2> <Var5>, <Var6>, ... ; // char a, b ;
voir aussi l'exemple ci-dessous.

Définition avec initialisation
<type> <var> = <valeur> ;

Exemple

```
int   x,    | une seule variable par
      y,    | ligne → place pour
      z=10 ;| commentaire et meilleure lisibilité
```

Définition avec emplacement en registre sans adresse
On peut demander au compilateur de placer, si possible, une variable locale dans un registre interne du µP/µC en faisant précéder sa définition ou déclaration de **register**. Cette possibilité est uniquement disponible pour les µP/µC avec des registres sans adresse (ex : A ou B avec 68HC11)
Syntaxe : register <type> <var> ;

Cette possibilité est surtout utile pour des systèmes embarqués avec des contraintes de temps d'exécution. Pour un ensemble d'instructions faisant plusieurs appels à une variable, il est plus rapide de la placer dans un registre interne plutôt que d'y accéder en RAM.

Définition avec emplacement en registre avec adresse (port d'E/S, registre de contrôle, ..)
Il n'y a pas de normalisation. Voir doc. du compilateur utilisé.
Seules sont concernées les variables globales.
Certains compilateurs utilisent une extension du C ANSI placée en fin de définition uniquement pour désigner l'adresse. Dans ce cas, il faut rajouter le modificateur de type **volatile** devant le type lors de la définition. *(Ceci empêche le compilateur de supprimer, lors de la phase d'optimisation, un accès à une variable qu'il jugerait inutile)*
Ex : avec compilateur Cosmic pour 68HC11
volatile <type> <nom> @<adresse> ;
volatile unsigned char DDRC @0x1007 ;
D'autres compilateurs utilisent une extension au C ANSI placée en début de définition pour indiquer qu'il s'agit d'un registre à fonction spéciale (Special Function Register).
Ex : avec compilateur IAR pour 80C196
sfrb IO_PORT0 = 0x0E ;

Définition *d'une variable modifiable de façon externe au programme*
Dans le cas où une variable peut être modifiée par une quelconque opération extérieure ou programme, comme un programme d'interruption ou un port E/S, il faut rajouter le modificateur de type **volatile** devant le type lors de la définition, pour la raison mentionnée ci-dessus.
syntaxe : volatile <type> <nom> ....

Définition *pour accès en lecture seule*
Certains registres ne sont accessibles qu'en lecture seule (ex : registre d'état, port d'entrée). S'ils sont définis avec le mot clé **const**, une erreur est générée lors de la compilation à chaque fois qu'une instruction spécifie une écriture.
syntaxe : const <type> <nom> adresse

***Définition* avec emplacement mémoire spécifique**
Voir §18.2, Organisation de la mémoire du système cible / Organisation de la mémoire dans les systèmes embarqués.
Avec certains µPs/µCs et compilateurs, il est possible ou nécessaire de définir la banque ou page (la terminologie dépend du constructeur) mémoire où est placée la donnée. Dans certains cas, ceci permet une optimisation de la taille du code produit et de la rapidité d'exécution en forçant un mode d'adressage.
La définition s'effectue avec une extension du C ANSI. Voir doc du compilateur utilisé.
Ex : avec le compilateur Cosmic pour 68HC11pour forcer le mode d'adressage direct
int Var1 @dir ;

DÉCLARATION

Une déclaration reprend la définition en étant précédée de **extern**.
Aucune réservation de mémoire n'est effectuée par une déclaration. La déclaration permet de donner des informations utiles au compilateur.

ex : extern unsigned char VarGlob1 ;

Aucune initialisation n'est ici possible ; l'initialisation ne peut s'effectuer que lors de la définition.

7.4) CONSTANTES

Les constantes sont des **valeurs numériques** ou des **chaînes de caractères**. Une constante peut être manipulée de différentes façons dans le programme source :

- directement par une **valeur**. Le type de la constante est défini par sa valeur et la règle d'écriture présentée ci-dessous.
  *Exemple : x=y+**25** ;.*
- par un **identificateur déclaré**. Dans ce cas, un emplacement mémoire est réservé (ROM avec système embarqué)
  Ex : x = y + Const1 ;
  Const1 a été au préalable définie. *Voir plus loin, Définition d'une constante.*
- par un **nom symbolique (symbole)**. Il faut d'abord définir l'équivalence entre un nom et une valeur. Lors de la compilation, chaque occurrence du nom choisi sera remplacée par la valeur de la constante.
  ex : x=y + CONST1 (il est d'usage d'écrire les symboles pour les constantes en majuscules). En début de fichier, l'équivalence est effectuée avec :
  #define CONST1 25
- par une **expression** ou n'interviennent que des valeurs, des symboles et des opérateurs. Ex : x= y + (VAL_MAX-VAL_MIN)/2. L'emploi d'une expression permet de rendre plus lisible les programmes. Lors de la 1ère phase compilation, la valeur de l'expression est calculée et cette valeur remplace l'expression lors des phases suivante de la compilation.

RÈGLE D'ÉCRITURE DES VALEURS NUMÉRIQUES

Type	Exemple
char, int	en décimal : 1 123 -5600 en hexadécimal : **0x**F4 **0X**5e en octal : **0**35
unsigned int	10000**U** 987**u**
long int	35000**L** -34**l**
unsigned long	10000**UL** 90000**ul**
float	4. -0.38 1.2e3 2.3E45

***Remarque* : la notation en binaire n'existe pas en C ANSI.**
Certains compilateurs ont une notation binaire, avec une extension du C ANSI.

RÈGLE D'ÉCRITURE DES CARACTÈRES ET CHAÎNES DE CARACTÈRES

Caractères

Un caractère est placé entre guillemets simples

Ex :
'a' '\x1B' (code hexadécimal)
\ introduit un code d'échappement[1]
(voir § Types de données complexes / Chaînes de caractères)

Chaînes de caractères

Une chaîne de caractère est placée entre guillemets doubles ". Les chaînes sont directement utilisées pour des fonctions d'affichage, ...

ex : Afficher("Bonjour ") ;

Voir §14.2 Types de données complexes / Chaînes de caractères.

DÉFINITION D'UNE CONSTANTE

Une définition est généralement utilisée pour une chaîne de caractère, un tableau de valeurs, ...

Syntaxe

const <type> <identificateur>=valeur ;

ex : const float Pi=3.1416 ;
const char Msg1[]= « Bonjour » ;
L'identificateur Msg1 est aussi l'adresse symbolique de la chaîne. Voir §14.2 sur les chaînes de caractères.

Emplacement de la définition / Portée

Idem Variables

CONSTANTE SYMBOLIQUE

#define <nom> <valeur>
/* pas de ; en fin */
ou
#define <nom> <expression constante>
une expression constante est uniquement constituée de constantes, d'opérateurs et d'autres symboles précédemment déclarés.

Le nom symbolique (symbole) peut être utilisé à la place de la valeur partout où une valeur est autorisée.

Le remplacement de nom par la valeur est effectué par le préprocesseur qui intervient juste avant la compilation proprement dite. Voir §19.1, Détail de la compilation / Préprocesseur.
Un symbole n'est pas un identificateur. Toutes les opérations permises avec un identificateur ne le sont pas forcément avec un symbole.

8) EXPRESSIONS ET AFFECTATIONS

8.1) NOTION D'EXPRESSION

En C, la notion d'expression est très large. On peut même considérer une variable isolée comme étant une expression (cas limite) de même qu'une affectation formée avec une variable, l'opérateur d'affectation (=) et une expression.
Quelques exemples d'expressions : &var, 2+y, x=2+y, a<b.

Les notions d'instruction et d'expression sont étroitement liées. Par exemple, x=2+y peut être considéré à la fois comme une expression et comme une instruction (avec le ; qui suit).
En C, une instruction est une expression suivie d'un ;
En C, toute expression possède :
- **une valeur**
- **un type** qui dépend du type de ses opérandes et de ses opérateurs

Par exemple l'expression a<b a une valeur et un type fixé par l'opérateur <.
Voir ci-dessous et §9, Opérateurs

Une expression peut correspondre à plusieurs calculs successifs en utilisant l'opérateur séquentiel. *Voir plus loin §9.6, Opérateurs/ Opérateurs divers / Opérateur séquentiel.*

8.2) TYPE ET VALEUR D'UNE EXPRESSION

Les différents types pour une expression sont :
- un des types de base (Entier, réel (float, ...))
- un type adresse (*voir §16, Pointeurs)*
- le type booléen (vrai/faux)

EXPRESSION DE TYPE BOOLÉEN

Les expressions de type booléen sont principalement utilisées dans les instructions de contrôle (*Voir §10, Structures / Instructions de contrôle).*
Une expression de type booléen est formée avec des opérateurs relationnels (>, <, ...) et logiques (&& (et), || (ou), ! (non)). Cette expression ne peut valoir que « vrai » ou « faux ».
Ex : (2*a) > (x+y)

Comme en réalité il n'existe pas de type simple booléen en C, les deux valeurs vrai et faux d'une expression de type booléen sont codées par des entiers (int) :
- **Vrai ↔ 1**
- **Faux ↔ 0**

EXPRESSION DE TYPE DE BASE / EXPRESSION DE TYPE BOOLÉEN

Le C autorise des constructions assez peu « logiques » mais qui permettent une grande compacité dans l'écriture d'un programme.

Comme une expression de type booléen correspond à un entier 0 ou 1, elle peut être incluse dans une expression numérique.
Ex : (x>y)*2 // vaut 0 ou 2

Une expression de type entier peut être assimilée à une expression de type booléen :
- **valeur différente de 0 ↔ vrai**
- **valeur égale à 0 ↔ faux.**

Voir §10 Structures / Instructions de contrôle

8.3) « LVALUE »

En C, une « lvalue » (Left) désigne ce qui peut intervenir à gauche d'un opérateur d'affectation (=).

8.4) ÉVALUATION D'UNE EXPRESSION

Lorsque plusieurs opérateurs interviennent dans une expression, son évaluation s'effectue selon :
- la priorité des opérateurs
- l'associativité des opérateurs (en cas de priorités identiques)

Les parenthèses permettent d'outrepasser les règles de priorités en forçant le calcul préalable de l'expression qu'elles contiennent.

La valeur d'une expression est du même type que ceux de ses opérandes, sauf dans les cas mentionnés ci-dessous.

8.5) CONVERSION DE TYPE IMPLICITE

Lorsqu'une expression, y compris une affectation, comporte des types différents, il y conversion automatique des types.
int → long → float → double → long double

En C ANSI, les types char sont systématiquement convertis en int si une opération arithmétique porte sur eux. Cette règle ne s'applique pas avec les compilateurs pour systèmes embarqués (économie d'espace mémoire, rapidité d'exécution).

Dans une affectation, il y a conversion systématique de la valeur de l'expression dans le type de la « lvalue ». Ceci peut se traduire par une perte de précision ou même un résultat sans signification.

8.6) CONVERSION DE TYPE EXPLICITE

Le programmeur peut forcer le type d'une expression avec l'opérateur **cast**. L'opérateur cast se compose de 2 parenthèses entre lesquelles est indiqué le type dans lequel il faut convertir l'expression qui suit.
Syntaxe : (<type>) <expression>
Ex : (int) (a/b)

8.7) AFFECTATION

Une affectation (=) peut être utilisée :
- lors de l'**initialisation** de tout type de données (constantes et variables)
- lors de l'**utilisation** de données

Dans ce dernier cas, une affectation ne peut être réalisée qu'entre une « lvalue » et une expression du même type. Les types autorisés sont :
- type simple. Ex : y = 3x + 2 ;
- type adresse (pointeur). Ex : ptVar1 = &Var1 ;
- type structure (voir plus loin, § Structures)

Il est impossible de réaliser une affectation avec les autres types complexes (tableaux, unions).

Une expression peut comprendre plusieurs affectations.

ex : PORTD = PORTC = Var1 ;
permet d'affecter la valeur de Var1 à PORTC puis à PORTD, en raison de l'associativité de droite à gauche de l'opérateur =. (*Voir ci-dessous, § opérateurs*)

9) OPÉRATEURS

Certains symboles peuvent avoir plusieurs significations. Le sens retenu dépend du contexte. Par exemple :
- * a deux significations en tant qu'opérateur (multiplication et contenu de l'adresse qui suit) et il permet aussi de déclarer un pointeur
- & correspond au ET logique bit à bit et à l'opérateur d'adressage

Les opérateurs ont des priorités qui vont de 1 à 15. 15 est la priorité la plus grande.

9.1) OPÉRATEURS ARITHMÉTIQUES

Les opérateurs arithmétiques ne portent que sur des nombres correspondant aux types de base et pour certains d'entre eux (+, -) sur les pointeurs. *Voir §16.2) Pointeurs et tableaux....*

opération	symbole	Syntaxe / remarque	Priorité /associativité
addition	+		12/g→d
soustraction	-		12/g→d
multiplication	*		13/g→d
division	/		13/g→d
reste entier division sur entiers	%	11%3 vaut 2	13/g→d
opposé	-		14/ g→d

9.2) OPÉRATEURS LOGIQUES

Ils sont de 2 types :
- les opérateurs qui produisent un résultat **int** (ou char selon compilateur et cible) 0 ou 1 et qui travaillent sur des conditions/expressions. Ces opérateurs sont proches des opérateurs relationnels. Ils sont présentés dans la même rubrique.
- les opérateurs de manipulation de bits qui effectuent des opérations au niveau des bits sur des entiers (on retrouve des instructions de ce type dans les microprocesseurs).

9.3) OPÉRATEURS RELATIONNELS (TESTS) ET LOGIQUE

voir § Expressions

Les opérateurs relationnels ont comme opérandes des nombres de types simples ou des pointeurs (sans mélanger les deux).
La valeur d'une expression utilisant un opérateur de ce type est « vrai » (**int** (ou char, selon compilateur et cible) 1) ou « faux » (**int** 0).

Les opérateurs logiques (non, et, ou) ont pour opérandes des expressions qui valent 0 (faux) ou 1 (vrai). Ils permettent de connecter plusieurs relations définies avec les opérateurs relationnels.
Ces opérateurs logiques peuvent aussi s'appliquer à des données ou des expressions de type int. Pour le C, tout entier ≠ 0 correspond à vrai.

opération/test	symbole	syntaxe /remarque	priorité / associativité
supérieur	**>**		10/ g→d
inférieur	**<**		10/ g→d
sup. ou égal	**>=**		10/ g→d
inf. ou égal	**<=**		10/ g→d
égal	**==**		9/ g→d
différent	**!=**		9/ g→d
non	**!**	***!cond/expr***	14/ g←d
prend la valeur 0 (faux) si cond/expr vaut #0 (vrai), la valeur 1 dans le cas contraire			
et	**&&**	***cond/expr1***	5/ g→d
&& cond/expr2 prend la valeur 1(vrai) si les 2 cond/expr valent #0, la valeur 0 dans le cas contraire			
ou	**||**	***cond/expr1***	4/ g→d
|| cond/expr2 prend la valeur 1(vrai) si au moins une des 2 cond/expr vaut #0 (vrai), la valeur 0 dans le cas contraire			

Avec les règles mentionnées plus haut et les opérateurs du tableau, on peut construire des expressions du type :
(A<B) && (C>D)
(A<B) == !(C>D) /* l'expression prend la valeur 1 lorsque les deux expressions A<B et !(C>D) sont vraies (valeur 1 pour chacune), c'est-à-dire lorsque A<B et C<=D */

9.4) OPÉRATEURS DE MANIPULATION DE BITS

Les opérateurs de manipulation de bits ne peuvent porter que sur des entiers de la taille d'un ou plusieurs mots machine (à vérifier).

opération	symbole	syntaxe /remarque	priorité / associativité
et	**&**		8/ g→d
ou inclusif	**|**		6/ g→d
ou exclusif	**^**		7/ g→d
complément	**~**		14/ g←d
décalage à droite	**>>**	var >> nb_dec[1]	11/ g→d
les bits ajoutés à gauche sont : • 0 pour une variable « unsigned » • le bit de signe pour une variable « signed »			
décalage à gauche	**<<**	var << nb_dec[1]	11/ g→d
les bits ajoutés à droite sont des 0			

1 : Le nombre de décalage peut être une constante ou une variable ou une expression.

9.5) OPÉRATEURS D'AFFECTATION

2 types d'opérateur d'affectation :
- affectation simple =
- affectation élargie. Réalise d'abord une opération puis une affectation simple. Ceci permet une écriture très condensée.

opération	symbole	syntaxe /remarque	priorité / associativité
Affectation simple	=	var=expression	2/ g←d
opération puis affectation	**+=** **-=** ***=** **/=** **%=** **&=** **|=** **^=** **>>=** **<<=**	var+=expr équivaut à var=var +expr	2/ g←d
incrémentaion/décrémentaton et affectation	**++** (1) **- -** (1)	pré/post incrémetation pré/post décrémetation	14/ g←d

(1) quand l'opérateur est placé avant l'expression, il y a pré incrémentation/décrémentation ; quand il est placé après, il y a post inc/déc. Lors de l'évaluation, il faut distinguer la valeur de la variable (ou expression) sur laquelle porte l'opérateur de la valeur de l'expression constituée de l'opérateur et de la variable (ou expression).

Exemple

expression (initialement, i=5)	valeur de l'expression après évaluation	valeur de la variable i après évaluation
++i	6	6
i++	5	6

Avec l'expression i++ ou ++i incluse dans une expression plus large, on obtient :

expression	*valeurs lors de l'évaluation*
n=++i-5	*++i : 6, n : 1*
n=i++-5	*i++ : 5, n : 0*

Lorsque l'opérateur porte sur une « lvalue » d'une affectation simple, le terme à droite est d'abord évalué puis la valeur est affectée à la « lvalue » après ou avant inc/déc selon position de l'opérateur.

ex : *pointeur++=variable
l'incrémentation est effectuée en dernier (ici après avoir mis la variable dans l'adresse pointée)

9.6) OPÉRATEURS DIVERS

Certains des symboles présentés ci-dessous sont employés pour plusieurs opérateurs.

Symbole opérateur	Rôle / Description	Priorité /associativité
()	Ordre d'évaluation d'une expression.	15/ g→d
[]	Accès à une cellule d'un tableau	15/ g→d
.	Opérateur de champ. Permet d'accéder à un champ d'une structure (*voir § Structure / Utilisation*)	15/ g→d
->	Permet d'accéder aux différents champs d'une structure à partir de son adresse de début (*voir § Structures / Utilisation*)	15/ g→d
sizeof()[1]	Renvoie la taille de l'expression ou du type en argument	14/ g←d
&	Opérateur d'adressage : fournit l'adresse de l'opérande qui doit être une « lvalue » *(voir passage de paramètre d'une fonction)*	14/ g←d
*	Opérateur d'indirection : donne le contenu d'un adresse *(voir passage de paramètre d'une fonction)*	14/ g←d
(type)	Opérateur « cast » : effectue la conversion forcée au type mentionné de la variable ou de l'expression entre () qui suit. Ex : c=(int) (a/b)	14/ g←d
? : *utilisation pour écriture condensée. Pour utilisateur expérimenté.*	Opérateur conditionnel. 3 opérandes = expressions. 1er opérande : exp.1 qui joue le rôle d'une cond. Le résultat de l'opération est : • valeur de exp.2 si valeur de exp.1≠0 (cond. vraie) • valeur de exp.3 si valeur de exp.1=0 (cond. fausse) Ex : max = a>b ? a : b max= valeur de a ou b selon la valeur de la condition a>b Une autre écriture possible est : if (a>b) max=a ; else max=b ;	3/ g←d
,	Opérateur séquentiel. Permet de grouper plusieurs expressions dans une seule expression. Ex : for(i=0, j=1, k=5 ;... ;...) est équivalent à : i=0 ; j=1 ; k=5 ; for(;... ;...) *utilisation pour écriture condensée. Pour utilisateur expérimenté. Voir ci-dessous*	1/ g→d

1 : L'opérateur sizeof est utilisé lors de l'allocation dynamique de mémoire. Voir § 24.1 Gestion dynamique de la mémoire.

OPÉRATEUR SÉQUENTIEL

L'opérateur séquentiel (,) permet de rassembler syntaxiquement 2 expressions en une seule.
La valeur et le type de l'expression complète sont ceux de la 2ème expression.

Ex : l'expression
x=3, y==0
réalise une affectation puis prend la valeur vrai (int 1) ou faux (int 0) selon la valeur de y

Les opérateurs séquentiels sont principalement utilisés avec les instructions de contrôle de flux.
Voir ci-dessous.

9.7) OPÉRATEURS ET EFFETS DE BORD

Un effet de bord consiste à modifier la valeur d'un opérande pendant le traitement d'une expression. Les opérateurs d'affectation simple et élargie ont des effets de bord. Avec une affectation simple, c'est uniquement la « lvalue » qui est modifiée.
Dans quelques cas rares, l'évaluation d'une expression peut dépendre de l'ordre d'évaluation qui peut varier d'un compilateur à l'autre. Il faut éviter d'utiliser des expressions de ce type.

10) STRUCTURES / INSTRUCTIONS DE CONTRÔLE DE FLUX

Les structures de contrôle de flux sont bâties avec les « instructions » de contrôle. Elles utilisent des expressions de type booléen. *Voir §8.2, Expressions et affectations / Type et valeur d'une expression.*
Ces expressions constituent des *conditions* pour exécuter un bloc d'instructions ou un autre.
Les « instructions » de contrôle ne sont pas des instructions comme les autres. Elles ne se terminent pas par « ; ».

10.1) ECRITURE DES EXPRESSIONS AVEC LES INSTRUCTIONS DE CONTRÔLE

Comme mentionné dans le § Expression /Type et valeur d'une expression, une expression de type booléen peut être remplacée par une expression de type entier.

ex : if(n!=0) ... est équivalent à if(n)

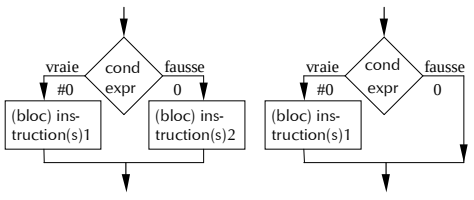
Valeur de n	Valeur de l'expression n !=0
0	0
≠ 0	1

Pour le C, 1 et une valeur quelconque ≠ 0 correspondent tous deux à « vrai ».
if(n==0) ... est équivalent à f(!n) ... (! renvoie un entier 0 lorsque n#0 et 1 lorsque n=0)

Valeur de n	Valeur de !n	Valeur de l'expression n ==0
0	1	1
≠ 0	0	0

Ces écritures compactes sont très utilisées ↑

10.2) STRUCTURE ALTERNATIVE



if (cond/expr)
{bloc inst 1 ;}
else {bloc inst 2 ;}

if (cond/expr)
{bloc inst 1 ;}

Si à la place d'un bloc, il n'y a qu'une seule instruction, les accolades sont facultatives.

L'opérateur conditionnel **? :** permet de réaliser une structure alternative complète où {bloc instructions 1} et {bloc instructions 2} affectent 2 valeurs différentes à une même variable. *Voir §9.2, Opérateurs / Opérateurs divers.*

STRUCTURES IMBRIQUÉES

L'absence de délimitation de la structure peut être risque d'erreur pour les structures imbriquées, car un **if** peut comporter ou non un **else**.
Un **else** se rapporte toujours au dernier **if** rencontré auquel un **else** n'a pas été attribué.
Dans certains cas, on peut utiliser une instruction vide pour traduire une imbrication.

```
Ex :
if (cond1)
  {inst_1 ;}
  if (cond2)
    {inst_2 ;}
    else ; /* signifie que else n'est pas utilisé */
  else {inst_3 ;} /* se rapporte au 1er if */

l'écriture suivante est à préférer
if (cond1)
  {inst_1 ; // pas de }
  if (cond2)
    {inst_2 ;}
   } // fin if (cond1)
   else {inst_3 ;} /* se rapporte au 1er if */
```

10.3) CHOIX MULTIPLE (SÉLECTION OU AIGUILLAGE)

switch(<expression_selecteur>)
{**case <const1>** : {instruction(s)1 ;} **break** ;
case <const2> : {instruction(s)2 ;} **break** ;
...
[**default** : {instruction(s)n ;]}

const1 est un entier ou une expression renvoyant un entier dont expression_selecteur peut prendre la valeur.
expression_selecteur est d'un type correspondant à un entier : char, signed char, int, etc.
« break » fait sortir de la structure de choix multiple.
La ligne « default » est facultative.

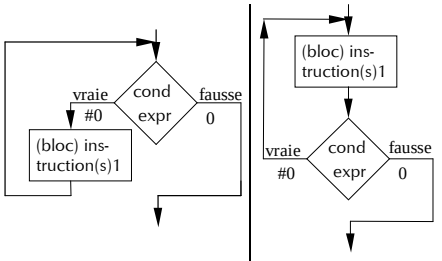
Il est possible d'omettre le « break » dans des cas particuliers :

- exécution d'un même bloc d'instructions pour plusieurs « case »
  Ex : ....
  {case 0 : {instruction(s)1 ;} break ;
  case 1 :
  case 2 :
  case 3 : {instruction(s)2 ;} break ;
  ....
- exécution du bloc d'instructions du « case » et de celui (ceux) de la (des) ligne(s) suivante(s).
  Ex : ...
  { case 0 : {instruction(s)1 ;} break ;
  case 1 : {instruction(s)2 ;}
  case 2 : {instruction(s)3 ;}
  case 3 : {instruction(s)4 ;} break ;
  ....

Remarque : Dans certaines conditions, la traduction de la structure fait appel à une portion

de programme déjà compilée et rangée dans la bibliothèque système. Il existe plusieurs portions de programme qui sont appelées selon le nombre de « case », leurs valeurs, etc.
Avec certains compilateurs et un nombre réduit de « case », il arrive que la traduction donne un code exécutable plus long que des structures if + quelques opérations.
Dans la plupart des cas, les constx sont traduites par des int (2 octets).

10.4) ITÉRATION (BOUCLE)

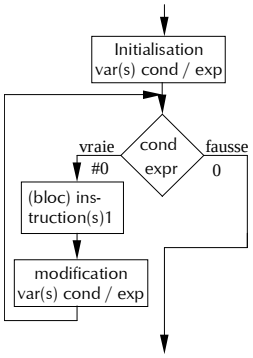


while (<cond/expr>) {instructions ;}	**do** {instructions ;} **while** (<cond/expr>) ;

Attention : il faut placer un ; après la condition / expression qui suit le while dans la structure do ... while.

10.5) ITÉRATION avec initialisation des variables de contrôle et modification après exécution du corps de la boucle

Cette itération utilise l'instruction **for (... ;... ; ...)**. L'utilisation la plus courante est décrite ci-dessous.



for (<initialisation var(s) de cond/expr> ; <cond/exp> ; <modification var(s) de cond/exp>)
{(bloc) instruction(s) 1}

L'initialisation et la modification se font avec des affectations (simple ou élargie, *voir § opérateurs*)
Ex :
for (n=1 ; n<100 ; n++)
{(bloc) instruction(s) 1}

10.6) UTILISATIONS PARTICULIÈRES DE for (... ; ... ; ...)

La syntaxe générale de l'instruction est :
for ([expression_1] ; [expression_2] ; [expression_3])
{....}
expression_1 est évaluée 1 seule fois avant d'entrer dans la boucle ; expression_2 est évaluée avant chaque parcours de boucle ; expression_3 est évaluée en fin de chaque parcours.
Chacune des 3 expressions est facultative. Lorsque l'expression 2 est absente, elle est considérée comme vraie (ceci permet de réaliser des boucles sans fin, voir plus loin).
Plusieurs actions peuvent être groupées avec l'opérateur séquentiel (voir plus loin).

Quand l'opérateur séquentiel est utilisé dans expression_2, la condition à évaluer est placée en dernier.
ex :
for(exp1 ; exp2a, cond2b ; exp3)
{...}
*pour obtenir l'équivalent d'une telle structure, il faut utiliser un **if** et un **goto***

10.7) INTERRUPTION du déroulement D'UNE BOUCLE

L'instruction **break** permet de sortir d'une boucle, en passant à l'instruction qui suit la boucle. **break** suit un choix.

Ex :
while(<cond/expr>)
{...
if (<cond/expr2>) **break** ;
...}

*Remarque : Lorsque cela est possible, il est préférable de ne pas employer **break**.*

emploi de break	*forme préférable*
while (1) *{c=getchar() ;* *if (c=='q') break ;* *...}*	*while ((c=getchar()) !='q')* *{...}* *//voir plus loin, § écriture condensée*

10.8) SAUT D'INSTRUCTIONS DANS UNE BOUCLE, POUR UN PASSAGE

L'instruction **continue** permet de sauter les instructions de la boucle qui suivent son emploi. **continue** suit un choix.

Ex :

while(<cond/expr>)
{...
if (<cond/expr2>)
continue ;
<inst n> ;
...<inst n+p> ;}
	si cond/expr2 vraie ou ≠ 0, inst n à inst n+p sont sautées pour un passage, avant nouvelle évaluation du « if »

10.9) SAUT À UNE ÉTIQUETTE

A proscrire. Cette possibilité existe cependant avec l'instruction « **goto label ;** ».
label : est placé devant l'instruction à atteindre. Ne pas oublier **:** derrière label placé devant l'instruction à atteindre. Il n'y a pas **:** derrière goto label. Par contre il faut ; comme en fin de toute instruction.

10.10) BOUCLE SANS FIN

Tous les programmes pour systèmes embarqués ont leur corps principal constitué d'une boucle sans fin, de même que beaucoup de programmes pour ordinateurs.
2 possibilités pour une boucle sans fin.

while(1) {...} /* n'importe quelle valeur ≠0 convient */
for (; ;) {...}

Remarque : On peut sortir d'une boucle sans fin avec l'instruction break ou goto (à proscrire).

10.11) ÉCRITURES CONDENSÉES DANS CONDITION/EXPRESSION

AFFECTATION PUIS TEST

Il est possible d'écrire dans la même condition/expression une affectation réalisée avant l'évaluation de « condition/expression ».
Voir expressions et affectations.
Ex :

écriture condensée	*écriture normale*
if ((c=getchar()) !='q') {...}	c=getchar() ; if (c !='q') {...}

L'opérateur () étant très prioritaire, l'affectation est réalisée en 1er; l'expression (c=getchar()) prend la valeur de c. C'est cette valeur qui est comparée à 'q' (code de la lettre q).

Remarque : avec l'écriture condensée, il n'y a pas de ; après l'affectation

UTILISATION DE L'OPÉRATEUR SÉQUENTIEL

L'opérateur permet de réaliser des constructions ayant des équivalences simples ou au contraire des constructions n'ayant pas d'équivalences simples.

La construction suivante
for (i=3, j=4, ; i<10 ; i++, j++)
{ ...}
est équivalente à
j=4 ;
for (i=3 ; i<10 ; i++)
{..
j++}

La construction suivante n'a pas d'équivalence simple.
for(exp1 ; exp2a, cond2b ; exp3)
{...}
Seule cond2b est prise en compte pour la sortie de la boucle. exp2a est évaluée avant cond2b.
Si exp2a est une affectation, il y aura autant d'affectations que de parcours de boucle + une affectation avant la sortie. C'est cette dernière qui sera prise en compte en sortie de la structure.
(voir la figure représentative du for)

12) FONCTIONS

12.1) GÉNÉRALITÉS

Quelques généralités sur les fonctions sont présentées au § 2.4.

UTILISATION D'UNE FONCTION

L'utilisation d'une fonction correspond à un **appel** de cette fonction. Elle s'effectue avec l'identificateur de la fonction.

Lorsqu'une fonction retourne une valeur, celle-ci remplace l'identificateur de la fonction.

*Ex : y=3*sin(x) ;*
x est l'information transmise à la fonction pour effectuer le calcul. La valeur calculée avec x (le sinus de x) remplace sin dans l'expression.

DÉFINITION / DÉCLARATION D'UNE FONCTION

Lorsqu'une fonction appelée n'est pas déjà compilée dans une bibliothèque, elle doit être définie dans un des fichiers source. La **définition** décrit ce que réalise la fonction. Elle est compilée en même temps que le fichier source qui la contient.

Lorsque la définition n'est pas sur le même fichier source que l'appel ou lorsqu'elle est placée après l'appel sur le même fichier source, la fonction doit être déclarée. La **déclaration** donne des informations utiles pour des vérifications réalisées lors de la compilation (nombre et taille des paramètres, etc.)

Les déclarations sont souvent situées dans un fichier inclus en début du fichier source lors de la première phase de la compilation. *Voir § 25.4, Programme sur plusieurs fichiers et §19, Détail de la compilation.*

PORTÉE

Comme pour une variable, l'endroit où est déclarée ou définie une fonction fixe sa portée. (*pour plus de détail, voir §12.1 Fonction / Définition)*

déclaration globale, en dehors de toute fonction ou du programme principal. La portée concerne tout le fichier source depuis l'endroit de déclaration jusqu'à la fin. La fonction peut être appelée dans le programme principal et les blocs de niveaux inférieurs ainsi que dans les programmes d'interruption
déclaration locale, dans le bloc d'une fonction. Seule cette fonction peut appeler la fonction déclarée.

Une fonction peut être définie dans un fichier et utilisée dans un autre à condition qu'elle soit déclarée dans chaque fichier l'utilisant.
Voir §12.1 Fonction / Définition

CODE PRODUIT PAR UNE FONCTION UTILISATEUR

La définition est traduite lors de la compilation par le code exécutable de la fonction ; la déclaration ne donne que des renseignements au compilateur, elle n'est pas traduite par du code exécutable ; l'appel de fonction est traduit par un appel de sous programme.
Le code exécutable de la fonction n'existe qu'une seule fois dans le programme objet final, même si la fonction est utilisée plusieurs fois.

FONCTION EN BIBLIOTHÈQUE

Pour une fonction en bibliothèque :

- le texte correspondant à la définition de la fonction est déjà compilé et le code correspondant est placé dans des bibliothèques (fichiers qui comprennent chacun plusieurs fonctions). Ce code est inclus lors de l'édition de liens.
- la déclaration de la fonction existe déjà dans un fichier dit « en-tête » d'extension h (header). Pour déclarer une fonction en bibliothèque, il suffit de mentionner dans le programme source que le fichier .h adéquat doit être inclus lors de la 1ère phase de compilation (pré-processeur).

Une fonction peut nécessiter des **paramètres**. Ceux-ci sont définis lors de la définition de la fonction puis transmis lors de l'appel de la fonction.

Le nombre de paramètres d'une fonction sont en général fixés lors de la définition.
Il est aussi possible de définir une fonction avec un nombre variable de paramètres.
Voir plus loin § Fonctions avec nombre variable de paramètres.

« FONCTIONS » INTÉGRÉES OU INTRINSÈQUES

Certains compilateurs fournissent des « fonctions » intégrées (built-in functions) ou intrinsèques qui permettent de réaliser des tâches normalement écrites avec des opérations d'affectation, etc.

Par exemple, le compilateur CCS pour PIC fournit de nombreuses « fonctions » dont OUTPUT_B().
Usage : OUTPUT_B(Valeur) ;
Cette fonction permet de fixer la valeur du port B. Avec un autre compilateur, on aurait écrit PORTB= Valeur ;

Ces « fonctions » ne sont pas de véritables fonctions. Elles sont traduites lors de la compilation par une ou plusieurs instructions, sans les mécanismes d'appel de fonction décrit plus loin.

Voir § 21) Particularités pour systèmes emarqués.

12.2) DÉFINITION

décrit ce que fait la fonction ainsi que les paramètres nécessaires avec leurs types.

La **définition** est constituée de 2 parties:

- l'**en-tête** qui comprend:
  - le type du résultat retourné
  - le nom de la fonction
  - le (les) paramètres(s) avec son (leurs) type(s)
- le **bloc ou corps de la fonction** qui indique ce que fait la fonction.

L'en-tête correspond à la partie **interface** de la fonction. C'est elle qui est utilisée pour la déclaration et pour l'appel.
Le bloc ou corps est aussi appelé **implémentation** de la fonction.

L'indication des paramètres avec leurs types correspond à une définition de variables. Des emplacements mémoires sont utilisés pour le passage de ces arguments. *Pour le détail, voir §22 Passage de paramètre.*

Les paramètres sont dits formels ou muets (dummy) (les identificateurs des paramètres importent peu). *Voir ci-dessous "transmission de paramètres"*

PORTÉE

Une fonction définie dans un fichier peut être utilisée dans n'importe quel fichier source d'un même projet (les fichiers objets correspondant sont réunis lors de l'édition de liens).

La portée proprement dite dans un fichier source dépend de l'endroit où se trouve la déclaration (si la définition est placée après l'appel ou dans un autre fichier source).
Voir plus loin, § 12.2 Déclaration / Emplacement.

Une limitation de portée au seul fichier source contenant la définition est possible (ceci permet d'éviter des conflits de noms). *Voir ci-dessous la syntaxe.*

SYNTAXE C ANSI

Syntaxe normale
en-tête
<type_retourné> <nom_fonct> (<type_1> <arg_1>, <type_2> <arg_2>, ...)
bloc ou corps de la fonction
{[définition des variables locales]
[déclaration de variables externes]
[déclaration de fonctions à portée locale]
liste des instructions}
entre [] : facultatif
Il n'y a pas de ; en fin de l'en-tête ni après } du corps

Si une fonction ne renvoie aucune valeur, ou ne nécessite pas de paramètre, le type est **void**.

Pour un exemple de définition, voir ci-dessous, dans « retour d'une fonction ».

Syntaxe avec limitation de portée
La portée d'une fonction peut être limitée au seul fichier source qui contient sa définition avec **static** placé en début d'en-tête.

ANCIENNE SYNTAXE

<type_retourné> <nom_fonct> (<arg_1>, <arg_2>, ...)
<type_1> <arg_1> ;
<type_2> <arg_2> ;
...
{.... /* bloc de la fonction */ }

Ex : int Somme(x, y)
int x, y ;
{...}

RETOUR D'UNE FONCTION

Le retour peut s'effectuer avec ou sans valeur retournée.
Le retour peut s'effectuer en un seul point ou en plusieurs points de la fonction. Dans ce dernier cas, selon les valeurs de paramètres transmis ou de variables globales, certaines instructions ne sont pas exécutées.

RETOUR SANS VALEUR DE RETOUR

Lorsque le retour s'effectue en fin de la fonction, il n'y a pas d'instruction spécifique.

Ex : void IncVar1(void)
{Var1 = Var1 +1 ;} / Var1 est une variable globale */*

Remarque : Pour simplifier l'écriture de certaines fonctions, il est possible de placer une ou plusieurs instructions ***return****.*

Ex : les 2 implémentations de la fonction suivante sont équivalentes.

```
interrupt void Inters(void)
{....                          |  {....
    if(TMR2IF==1)              |      if(TMR2IF==1)
        {...                   |          {...}
        return ;}              |      else {
    if(T0IF==1)                |          if (T0IF==1)
        {...                   |              {...}
        return ;}              |          }
....}                          |  ...}
```

Ceci est surtout utile avec plusieurs structures if imbriquées.

RETOUR AVEC VALEUR DE RETOUR

Si la fonction retourne une valeur dans une expression qui l'a appelée, sa définition se termine par l'instruction **return** suivi de l'expression dont la valeur est à retourner, dans le cas ou le retour s'effectue en fin de fonction.

Ex de définition

```
int Carre(int u)  | en-tête. u : paramètre formel
{int v ;          | corps de la fonction
v=u*u ;           | on aurait pu aussi écrire
return v ;}       | return(u*u) ;
```

Comme précédemment, il peut y avoir plusieurs instructions return.

EMPLACEMENT

Une fonction est définie en dehors de toute autre fonction, n'importe où dans le fichier source.
Il est possible de placer la définition d'une fonction :

- avant son appel dans le même fichier source
- après son appel dans le même fichier source
- dans un autre fichier source que celui contenant l'appel

Le 1er cas correspond à des programmes très simples. Dans ce cas, il est inutile de déclarer la fonction.
Dans les 2 autres cas, il faut absolument déclarer la fonction. *Voir ci-dessous, § Déclaration*

Le 3ème cas correspond à des programmes importants : les définitions de fonctions sont regroupées sur un ou plusieurs fichiers source compilés séparément. *Voir §25.4, Conseils d'écriture d'un programme source / Programme sur plusieurs fichiers*

12.3) DÉCLARATION

La déclaration d'une fonction donne des informations indispensables au compilateur.

Elle est inutile si la fonction est définie avant son appel dans le même fichier source. Il est cependant préférable de toujours déclarer une fonction ; le programme reste valide même après modification et réorganisation.
Le C ANSI autorise à ne pas déclarer une fonction qui renvoie un entier (int). Il est cependant préférable de toujours déclarer une fonction ; ceci permet d'éviter des erreurs.

Pour les fonctions prédéfinies, les déclarations sont dans des fichiers .h qu'il faut inclure avec la directive #include.
Voir §2.4, Constitution d'un programme / Fonctions / Portée
La déclaration reprend l'en-tête de la fonction, en omettant généralement les identificateurs des paramètres.

SYNTAXE C ANSI

déclaration normale :
<type_retourné> <nom_fonct> (<type_1> <arg_1>, <type_2> <arg_2>, ...) ;
déclaration complète (**protoype**)
idem ci-dessus plus identificateurs arguments, comme dans l'en-tête de la définition. Les identificateurs des arguments sont susceptibles d'apporter quelques éclaircissements au lecteur d'un programme ; ils sont ignorés par le compilateur.
déclaration pour fonction définie dans autre fichier
idem déclaration normale ou avec **extern** placé en début de déclaration (pour faciliter la localisation de la définition pour le programmeur)

Une déclaration est une instruction, elle se termine par ;

Ex de déclaration
int Carre(int) ;
int Trinome(int, int, int) ;

EMPLACEMENT

Voir §2.4, Constitution d'un programme / Fonctions / Portée
Déclaration globale : en dehors de toute fonction, avant l'appel (en général en début du fichier source).
Déclaration locale : au tout début du bloc de la fonction appelante (y compris main) ; jamais après 1ère instruction autre que déclaration.

RÔLE

La déclaration d'une fonction permet au compilateur d'effectuer :

- une vérification sur les paramètres entre la déclaration et l'appel (type et nombre). En cas d'erreur un message est généré lors de la compilation.
- éventuellement une conversion automatique de type (*Voir §8.5, Expressions et affectations / Conversion de type implicite*).

12.4) APPEL

- constitue une instruction simple (fonction effectuant une action –procédure-)
- ou correspond à un opérande dans une expression. La fonction renvoie alors une valeur

Lors de l'appel, il faut spécifier le nom de la fonction et ses ***paramètres*** qui sont appelés ***paramètres effectifs*** ou ***arguments***.

Remarque : il est aussi possible de remplacer le nom de la fonction par un pointeur vers cette fonction (en C, le nom de la fonction correspond à son adresse). voir §16 sur les pointeurs.

Lors du déroulement du programme, du point de vue du programme source, les paramètres formels reçoivent une copie des paramètres effectifs, selon leurs positions (leurs noms n'importent pas).

Pour les exemples, voir plus loin, dans le §12.5, Transmission des paramètres

12.5) NOMBRE DE PARAMÈTRES

Le nombre de paramètres peut être :

- **fixe**. Ce nombre est fixé lors de la définition de la fonction. Cas le plus fréquent.
- **variable**. L'en-tête de la fonction a une syntaxe particulière ; la définition de la fonction utilise des fonctions prédéfinies spécifiques.
  Voir plus loin. Exemple de fonction avec un nombre d'argument variable : printf()

12.6) TRANSMISSION ET TYPES DE PARAMÈTRES

Il y a 2 façons de transmettre les paramètres lors de l'appel de la fonction :

- **transmission par copie de valeur** (ou plus simplement par valeur). Du point de vue du programme source, le paramètre formel de la définition de la fonction reçoit une copie provisoire du paramètre effectif employé lors de l'appel de la fonction. La fonction ne peut pas modifier la valeur de ce paramètre effectif. Cas de transmission le plus fréquent. *Pour des détails sur le passage de paramètres, voir le §22).*
- **transmission par adresse**[1]. C'est l'adresse de la variable (ou plus exactement une copie de l'adresse) qui est passée à la fonction. La fonction appelée ne travaille pas sur une copie de la donnée transmise, mais sur la donnée elle-même. La fonction appelée peut donc modifier un paramètre effectif utilisé dans la suite du programme.

  1: *On utilise parfois le terme* ***passage par référence****, mais il est préférable de l'éviter car en C++ cette désignation correspond à un type de passage de paramètre par adresse avec une syntaxe particulière.*

Pour tous les types de données, sauf un (tableau), on peut choisir une transmission par copie ou par adresse. Un tableau est forcément passé par adresse. *Voir §14.1, Types de données complexes / Tableaux.*
Les arguments sont souvent transmis par valeur pour les données de types simples et par adresse pour les données de types complexes (structures, ...). Il est beaucoup plus rapide, lors de l'exécution du programme, de transmettre l'adresse d'une variable de type complexe que de transmettre cette variable en entier.

TRANSMISSION DE PARAMÈTRE(S) PAR VALEUR

Ex d'emploi d'une fonction avec transmission par valeur :

... int somme(int, int) ; ... z=somme(a,b) ; ...	utilisation ***déclaration*** ***appel*** dans une expression. a et b : paramètres effectifs
int somme(int x, int y) {int s ; s=x+y ; return(s) ;}	***définition*** (placée ici après l'utilisation). x et y : paramètres formels

TRANSMISSION DE PARAMÈTRE(S) PAR ADRESSE

Le bloc appelant doit transmettre l'adresse (ou les adresses) de la (des) variable(s) à la fonction. La fonction qui reçoit ce(s) paramètre(s) doit avoir comme paramètre(s) formel(s) un type particulier qui permet de recevoir une adresse : il s'agit d'un **pointeur**. La fonction appelée fait un appel indirect à la (aux) variable(s), par l'intermédiaire de leur(s) adresse(s).

Quelques fois, une fonction nécessite comme paramètre une adresse de pointeur. Pour plus de détail, voir la partie sur les pointeurs.

Opérateur d'adressage &

L'opérateur & (qui est aussi utilisé pour effectuer un ET bit à bit entre nombres entiers) permet d'obtenir, quand il est placé devant un nom de variable, l'adresse de cette variable.

Pointeur

Type de donnée qui correspond à une adresse

Déclaration

<type_donnée>* <nom_pointeur>
signifie que 'nom_pointeur' est un pointeur vers une donnée de type 'type_donnée'. * signifie ici : est un pointeur vers le type qui précède (* est aussi utilisé pour la multiplication des nombres).

Accès indirect aux variables

Pour accéder, via un pointeur, à une autre donnée, on utilise l'opérateur *, mais cette fois avec la signification (la troisième) d'opérateur d'indirection. Dans une instruction d'un programme, * signifie « contenu de l'adresse» qui suit. L'adresse est désignée ici par un pointeur.

Ex d'emploi d'une fonction avec transmission par adresse, avec utilisation des 3 significations de * :

... void multx2(int*) ; .... multx2(&Var1) ; ...	utilisation ***déclaration*** ***appel.*** L'adresse de Var1 est transmise à la fonction
void multx2(int* ptX[1]) {*ptX=*ptX*2 ;}	***définition*** (placée ici après l'utilisation). ptX : pointeur vers entier *ptX : contenu de l'adresse ptX ...*2 : multiplié par 2

1 : l'identificateur commence par pt pour indiquer qu'il s'agit d'un pointeur.

la fonction multx2 modifie la valeur de la variable Var1.

Le mécanisme de transmission des paramètres est détaillé § Passage de paramètres et variables locales.

12.7) VALEUR RETOURNÉE

En général une fonction renvoie un scalaire (entier, réel) ou un caractère.
La norme ANSI autorise une fonction à fournir :

- un scalaire ou un caractère (cas le plus employé)
- une structure (*voir plus loin* ; dans ce cas, la seule expression possible contenant l'appel de la fonction est une affectation à une autre structure)
- une adresse qui doit être affectée à un pointeur (*voir §16, Pointeurs*). (appel de fonction du type NomPointeur = fct(...) ;). Cette adresse peut correspondre à l'identificateur d'un tableau, ...

12.8) RÉ-ENTRANCE ET ESPACES RAM ET ROM UTILISÉS PAR UNE FONCTION

Une fonction est **réentrante** si :

- elle s'appelle elle-même (fonction **récursive**)
- elle peut être appelée par le programme principal et un programme d'interruption.

Dans ce dernier cas, le programme d'interruption peut s'exécuter lorsque la fonction est en cours d'exécution.

Le C ANSI prévoit que toutes les fonctions sont ré-entrantes.

Les fonctions ré-entrantes nécessitent des espaces ROM (code) et RAM (lors de l'exécution) sensiblement plus importants et une durée d'exécution plus importante que des fonctions non ré-entrantes.

Selon les µP/µC et les compilateurs :

- toutes les fonctions sont systématiquement ré-entrantes (programmes pour PC par ex.)
- les fonctions peuvent être ré-entrantes ou non selon des options de compilation
- les fonctions ne sont jamais ré-entrantes (µC de très faibles ressources, type PIC entrée de gamme. Le C ANSI n'est pas respecté)

Les espaces ROM et RAM utilisés par une fonction sont expliqués dans le § Passage de paramètres / Variables locales

12.9) FONCTIONS DE LA BIBLIOTHÈQUE STANDARD

La bibliothèque standard est souvent décomposée en plusieurs fichiers. Les noms fréquents sont : stdio.lib (standard input output), stdlib.lib, math.lib ....
Voir la documentation du compilateur.
Voir le §20 Bibliothèques et gestion de bibliothèques.

Lorsqu'on utilise une fonction pré-compilée d'une bibliothèque, il faut la déclarer avant.

Ceci s'effectue en incluant le le fichier .h (header = en-tête) contenant toutes les déclarations des fonctions dans la bibliothèque. La directive à utiliser est #include.
Ex : #include math.h
Voir § Structure simplifiée d'un programme source / Inclusion de fichiers.

Parmi les fonctions de la bibliothèque standard, on trouve les catégories décrites ci-dessous.

Pour l'usage, voir la documentation du compilateur.

FONCTIONS D'ENTRÉE / SORTIE

Elles sont utilisées pour la saisie, l'affichage, ... Avec un système embarqué, ces fonctions permettent de gérer une liaison série, ...
Voir plus loin fonctions de la bibliothèque standard d'entrée / sortie

FONCTIONS DE MANIPULATION DE CHAÎNES DE CARACTÈRES

Ces fonctions commencent par str (string). Les chaînes sont repérées par leurs adresses symboliques. *Voir § Types de données complexes / Chaînes de caractères*
Quelques exemples de fonctions :

- strcpy : copie d'une chaîne dans une autre.
- strcat : concaténation de chaînes
- strlen : détermination de la longueur d'une chaîne (length)
- strchr : recherche d'un caractère dans une chaîne

FONCTIONS DE MANIPULATION DE DONNÉES

Ces fonctions permettent de manipuler des données qui peuvent être des chaînes ou des nombres. Les données sont repérées par leurs adresses symboliques. La taille mémoire utilisée par les données doit être spécifiée.
Quelques exemples de fonctions :

- memcpy : copie d'une donnée d'une zone mémoire à une autre
- memcmp : comparaison de 2 données

FONCTIONS DE TRANSFORMATION DE CHAÎNES DE CARACTÈRES EN NOMBRES

Les chaînes sont manipulées par leurs adresses symboliques. Quelques exemples de fonctions :

Conversions de chaînes vers des nombres :

- atof (ASCII to float) : chaîne vers réel
- atoi (ASCII to integger) : chaîne vers entier

Il n'existe pas de fonctions de conversions de nombres vers des chaînes dans la bibliothèque standard.

FONCTIONS DE CALCUL

Les fonctions portent sur des réels (float, double, ... -*voir doc du compilateur utilisé*-).
fonctions trigonométriques (sin, cos, tan, asin, acos, atan, sinh, cosh, tanh), puissance (pow), racine carrée (sqrt), logarithme (log, log10),

exponentielle (exp), nombre aléatoire (rand), ...

FONCTIONS DE GESTION DYNAMIQUE DE LA MÉMOIRE

Les 2 fonctions de bases sont :

- **malloc()** pour allouer de la mémoire et retourner un pointeur sur le bloc alloué
- **free()** pour libérer un bloc mémoire

Ces deux fonctions sont décrites en détail plus loin.

Pour des applications sur ordinateurs, on trouve aussi des bibliothèques de fonctions graphiques (tracé de courbes, d'axes, ...), des fonctions de gestions de fichiers, ...

12.10) FONCTIONS SYSTÈME

Pour la traduction du programme source, le compilateur a parfois besoin de faire appel à des fonctions système contenues dans une bibliothèque système (Machine Library). Celle-ci est parfois appelée bibliothèque d'exécution (run-time library).
Ces fonctions déjà compilées sont utilisées lors de la phase d'édition de liens.

12.11) FONCTIONS AVEC UN NOMBRE VARIABLE DE PARAMÈTRES

Ces fonctions sont peu utilisées, surtout dans les systèmes embarqués.
Une fonction de ce type est la fonction précompilée printf(). *Voir § Fonctions de la bibliothèque standard.*

Une fonction avec un nombre variable de paramètres doit au moins avoir un paramètre fixe. Le nombre d'autres paramètres n'est pas déterminé lors de la définition et peut être quelconque.

DÉFINITION

L'en-tête doit se terminer par ... après la liste du ou des paramètres fixes.

Le corps de la fonction doit utiliser des macros et un type de données spécifiques pour récupérer les paramètres en nombre variable.
Ces macros et type de donnée sont standard et livrées avec la quasi totalité des compilateurs (fichier stdarg.h)

Le tableau ci-dessous indique les différentes instructions à réaliser.

instruction à réaliser	exemple
définir un pointeur de type va_list[1] qui contiendra l'adresse du premier paramètre facultatif	va_list PtParam ;
affecter l'adresse du premier paramètre facultatif à PtParam avec la macro va_start().	va_start(PtParam, <NomDernierParObligatoire>) ;
récupérer chaque paramètre facultatif avec la macro va_arg(). A chaque appel de va_arg, le pointeur est incrémenté d'un nombre correspondant à la taille du paramètre.	va_arg(PtParam, <TypeParam>) ;
affecter la valeur NULL au pointeur de paramètre. Règle indispensable dans le C ANSI.	va_end(PtParam) ;

La gestion de la fin de liste des paramètres variables est laissée au soin de l'utilisateur. Les diverses possibilités sont :

- utilisation d'une « sentinelle », valeur particulière qui indique le dernier paramètre
- transmission du nombre de paramètres variable dans un des paramètres fixes. Une variable décrémentée après chaque extraction d'un paramètre variable permet de connaître le dernier paramètre
- transmission d'un code pour chaque paramètre variable dans une chaîne de caractère faisant partie d'un des paramètres fixes (*Voir § Fonctions de la bibliothèque standard / printf())*

Exemple de définition

Le seul paramètre fixe est ici le nombre de paramètres variables

```
void FonctParVar(int NbParVar, ...)
{ va_list PtPar ; /* déf d'un pointeur */
  int i, ParVar ;
  va_start (PtPar, NbParVar) ; /*adresse du 1er
        param facultatif → PtPar */
  for(i=1 ; i<=NbParVar ; i++)
        { ParVar = va_arg(PtPar, <TypeParam>) ;
        /* récupération et traitement des
        paramètres facultatifs */
        ....
        }
  va_end(PtPar) ;
}
```

APPEL

L'appel s'effectue comme n'importe quelle fonction.

exemple :
FonctParVar(3, Var1, Var2, Var3) ;

12.12) FONCTIONS & FONCTIONS « IN-LINE »

Une fonction habituelle correspond à un sous programme. L'appel d'une telle fonction dans le programme source correspond donc à appel de sous programme lors de l'exécution, avec les mécanismes habituels (adresse de retour rangée dans la pile, ...). Le mécanisme pour le passage des paramètres est décrit dans le § 22.
Beaucoup de compilateurs offre la possibilité d'utiliser des fonctions « in-line ». Il faut rajouter un mot clé lors de la définition (voir doc compilateur). Lors de la compilation, chaque appel dans le programme source est traduit par les instructions de la fonction, sans appel de sous programme. Ce type de fonction permet d'augmenter la lisibilité du programme source. Une fonction « in-line » est très proche d'une macro.

Certains compilateurs pour µC de faibles ressources choisissent automatiquement le type de fonction : normal ou in-line. Si une fonction n'est utilisée qu'une seule fois dans un programme source, elle sera considérée comme une fonction « in-line ». Ceci permet de diminuer la taille du code produit, d'augmenter la rapidité d'exécution et d'économiser un niveau de pile. Cette possibilité n'existe que s'il n'y a pas de compilation séparée.

Exemple : le compilateur CCS pour PIC place un GOTO vers le code de la fonction lors de l'unique appel de cette fonction. En fin des instructions de la fonction, un autre GOTO renvoie à la suite du programme.

13) MACROS

Voir le § 2.5 pour la présentation générale des macros.

Une macro (ou macro-commande ou macro – instruction) permet de remplacer :

- une expression
- une ou plusieurs instructions

par un nom.

Une macro doit d'abord être définie, avant de pouvoir être utilisée.

13.1) MACRO SANS PARAMÈTRE

DÉFINITION

Toutes les macros sont généralement définies en début du fichier source, après les inclusions des fichiers d'en-tête.

#define <nom> <texte de remplacement>

texte de remplacement : correspond à une ou plusieurs instructions ou une expression (mise entre parenthèses de préférence)

Texte de remplacement sur plusieurs lignes :
il faut insérer \ (antislash) en fin de chaque ligne, sauf sur la dernière.

```
ex :  #define InitLiaisonSerie \
      BAUD=vitesse_transmission ; \
      SCCR1 = valeur1 ; \
      SCCR2 = valeur2 ;
```

UTILISATION

Il suffit d'écrire le nom de la macro. Lors de la 1ère phase de la compilation, le nom sera remplacé par la ou les instructions.

Ex :
...
InitLiaisonSerie
...
Attention : si le dernier « ; » est placé dans le texte de remplacement, il ne faut pas placer un autre « ; » après la macro.

[1] *va_list est un pointeur générique void* (Voir plus loin, § Pointeur / Pointeur sur donnée / Type de pointeurs).*

13.2) MACRO AVEC PARAMÈTRE

DÉFINITION

Même emplacement que pour macro sans paramètre.
#define <nom>(<param1>, <param2>, ...) <texte de remplacement>
Aucun espace entre nom et (
param1, ... sont les paramètres formels qui seront remplacés par les paramètres réels lors de l'appel de la macro.

ex : #define Add(x,y) (x)+(y)

Remarque : pour éviter les erreurs, il est conseillé de placer entre parenthèses les paramètres formels dans le texte de remplacement.

UTILISATION

Utilisation identique à celle d'une fonction.

ex : putchar(c) ;
Dans le C pour PC, putchar est un macro prédéfinie dans le fichier stdio.h

14) TYPES DE DONNÉES COMPLEXES

Les types de données agrégés ou structurés comprennent les **tableaux**, les **chaînes de caractères**, les **structures**.

14.1) TABLEAUX

Un **tableau** (array) est un ensemble d'éléments (ou cellules) de même type désignés par un identificateur unique ; chaque élément est repéré par un indice (tableau à une dimension) ou plusieurs indices (tableaux à plusieurs dimensions). Un tableau à plusieurs dimensions est un tableau composé de sous tableaux.

Le type d'une cellule peut être :
- un type simple (char, int, ...)
- un pointeur
- un type complexe (structure → tableau de structures, ...)

DÉFINITION DE TABLEAU

syntaxe : <type_var> <nom_tableau> [<dimension1>] [<dimension2] [...] avec type_var : type de variable contenu dans les cellules.

```
ex : int TA[7] /* le tableau TA contient 7 éléments qui sont des entiers */
int TB[8][11]  /* le tableau TB est constitué de 8X11 éléments qui sont des entiers */
```

INITIALISATION DE TABLEAU

L'initialisation totale ou partielle s'effectue lors de la définition. Les valeurs fournies doivent être du type constante ou expression constante.
2 possibilités pour l'initialisation lors de la déclaration :

- le nombre d'éléments est indiqué.

```
Ex : int TA[3]={0,2,6} ;
char Table[2][3] = { {1,2,3},
                     {4,5,6} } ;
```

- le nombre d'élément n'est pas indiqué ; il est déterminé automatiquement lors de la compilation

```
Ex : int TA[]={0,2,6} ;
```

L'initialisation d'un tableau après sa définition se fait élément par élément.

ACCÈS À UN ÉLÉMENT D'UN TABLEAU

L'accès à un élément d'un tableau s'effectue par son indice. En C, le 1er indice est 0
Avec un tableau défini TA[7], TA correspond aux éléments TA[0] jusqu'à TA[6].
Pour un tableaux à plusieurs indices T[N][P], les éléments sont ordonnés comme suit :
T [0] [0], T [0] [1], T [0] [2], ..., T [0] [P-1],
T [1] [0], T [1] [1], T [1] [2], ..., T [1] [P-1],
...
T [N-1] [0], T [N-1] [1], T [N-1] [2], ..., T [1] [P-1]
Ils sont rangés dans cet ordre en mémoire.

UTILISATION D'UN TABLEAU

Chaque élément d'un tableau s'utilise comme n'importe quelle variable.

```
Ex : TA[2]=8 ;
Var1=2*TA[3] ;
```

L'identificateur d'un tableau correspond à son adresse de début.
L'identificateur peut donc être utilisé comme une adresse, par exemple pour passer l'adresse d'un tableau à une fonction qui reçoit ce type d'argument.

```
Ex : avec tableau TA précédent,
EffaceTab(TA); /* la fonction reçoit l'adresse du tableau et peut donc modifier ses éléments */
```

Voir le § ci-dessous « Tableau en paramètre d'une fonction ».

Les éléments d'un tableau peuvent aussi s'utiliser via un pointeur. *Voir §16.2 Pointeurs sur donnée/ Pointeurs et tableaux / Pointeur avec indice.*

Il n'est pas possible d'utiliser tous les éléments d'un tableau (pour une lecture ou une affectation) avec un opérateur du C, autrement que lors de l'initialisation.

INDICE D'UN TABLEAU

Un indice peut prendre la forme de n'importe quelle expression arithmétique de type entier (ou caractère, compte tenu des règles de conversion systématique).

```
Ex : TA[N+3*Var1]
```

Lors de plusieurs accès successifs à des éléments d'un tableau, dans le cas d'une boucle par exemple, il est possible de placer un des opérateurs ++ ou – dans l'indice.

```
Ex : DateHeure[Indice++]=RCREG;
```

TABLEAU DE TAILLE VARIABLE

Voir § Données dynamiques.

TABLEAU EN PARAMÈTRE D'UNE FONCTION

Un tableau ne peut être transmis que par adresse. (*voir §12.5 « Transmission et types de paramètres »).*.

DÉFINITION DE LA FONCTION

Le paramètre qui reçoit l'adresse du tableau doit être de type pointeur. Comme le nom d'un tableau correspond à son adresse, donc à un pointeur, on peut aussi mentionner un tableau (avec pour les éléments le même type que celui des éléments du tableau transmis en paramètre lors de l'appel).
Pour l'accès aux éléments du tableau dans le corps de la fonction, on peut utiliser le formalisme tableau ou le formalisme pointeur.

Pour un **tableau à 1 dimension**, on peut utiliser les possibilités suivantes pour l'en-tête et l'accès à un des éléments.

Forme de l'en-tête	accès à un élément dans le corps de la fonction
void Fct1(char Tab[10])[1]	Tab[i] ou *(Tab + i)
void Fct1(char Tab[])	Tab[i] ou *(Tab + i)
void Fct1(char* PtTab)	*(PtTab + i)[2]

1: Le nombre d'éléments du tableau n'est utile que pour la lisibilité du programme. Pour le compilateur, seule compte l'adresse du début de tableau qui est transmise lors de l'appel de la fonction.
2 : Voir §16 sur les pointeurs et plus particulièrement 16.2, Addition et soustraction d'un entier
Tab est un pointeur local à la fonction. C'est une « Lvalue » qui peut être modifiée avec une opération d'affectation.

Pour un **tableau à 2 dimensions et plus**, il est indispensable de donner la taille des sous tableaux pour que le compilateur puisse déterminer l'adresse de chacun des éléments pour y accéder. (*Voir plus haut, § Accès à un élément d'un tableau*). La seule forme d'en-tête est donc celle avec un tableau dont les tailles des sous tableaux sont données.

Forme de l'en-tête	accès à un élément dans le corps de la fonction
void Fct2(char Tab[10][5])	Tab[i][j]
void Fct2(char Tab[][5])	Tab[i][j]

La première dimension du tableau n'est pas indispensable pour déterminer l'adresse de chacun des éléments.

APPEL DE LA FONCTION

Le nom d'un tableau correspond à son adresse de début. On peut donc donner le nom du tableau ou l'adresse du premier élément.
Les 3 appels suivants sont équivalents :

```
Fct1(Tab1) ;
Fct1(&Tab1[0]) ;
Fct1(PtTab1); /* PtTab1 est un pointeur initialisé avec l'adresse du début du tableau Tab1 */
```

14.2) CHAÎNES DE CARACTÈRES

En C, il n'existe pas de type chaîne (string) qu'il est possible d'utiliser dans une définition.

Une **chaîne de caractères** (string) n'est rien d'autre qu'un tableau dont les éléments sont des caractères. Une chaîne de caractères se termine par le code terminateur nul (NULL = \0).

Une chaîne peut être utilisée :

- **directement** (en toutes lettres ou chiffres) lors d'initialisation de tableau ou d'un pointeur (*voir § pointeurs*)
- **directement** (en toutes lettres ou chiffres) en paramètre de certaines fonctions
- **indirectement par son adresse** qui est celle de la 1ère case du tableau où elle est rangée et qui correspond au nom de ce tableau (voir § précédent sur les tableaux)

CONSTITUTION D'UNE CHAÎNE

La chaîne est une suite de caractères placée entre guillemets (double quote " ").
Chaque caractère est remplacé par son code ASCII dans le tableau qui correspond à la chaîne.

Ex : "Bonjour"

Attention : le code de certains caractères dépend du système d'exploitation. DOS utilise les codes ASCII OEM alors que Windows utilise les codes ASCII ANSI.

Il est possible de placer dans une chaîne certains caractères non imprimables tels que saut de ligne ou tabulation ainsi que des codes qui ne correspondent pas à un caractère disponible au clavier.
Ceci s'effectue avec les codes d'échappement.

CODE D'ÉCHAPPEMENT

Utilisé pour une constante caractère ou dans une chaîne de caractère. Introduit par \

code	valeur hexa	caractère	effet
\a	07	BEL (BELL)	cloche ou bip
\b	08	BS (Back-Space)	retour arrière
\f	0C	FF (Form Feed)	saut de page
\n	0A	LF (Line Feed)	saut de ligne
\r	0D	CR (Carriage Return)	retour chariot = Entrée
\t	09	HT (Horizontal Tab)	tabulation horizontale
\v	0B	VT (Vertical Tab)	tabulation verticale
\\	5C	\	
\'	2C	'	
\''	22	''	
\?	3F	?	
N	valeur octale du code d'un caractère		
\x*N* ou \X*N*	valeur hexadécimale du code d'un caractère. Ex : \x1B pour ESC		

Ex de chaîne avec un code d'échappement :

"Temperature en \xDF C"
Cette chaîne permet d'écrire Temperature en ° C sur un LCD qui ne comprend pas le code ASCII de ° mais qui affiche ce caractère pour avec le code DE.

UTILISATION DIRECTE D'UNE CHAÎNE.

Une chaîne peut être directement utilisée lors de l'initialisation d'un tableau ou comme paramètre d'une fonction.

Initialisation de tableau

Ex : char Msg[]="bonjour" ; /* le nb d'éléments du tableau est automatiquement déterminé, les codes des caractères sont placés dans le tableau, \0 est placé en fin de tableau */
La chaîne peut ensuite être manipulée avec son adresse qui est Msg, dans certaines fonctions.

Utilisation comme paramètre d'une fonction

Un tableau (sans nom) avec les codes des caractères est automatiquement construit et l'**adresse** de ce tableau est passée à la fonction.

Ex : printf("bonjour") ; /* l'adresse du tableau qui contient bonjour est transmis à la fonction */

UTILISATION D'UNE CHAÎNE PAR SON ADRESSE

Identique à l'utilisation d'un tableau.

Ex : avec un tableau de caractères Msg1 déjà défini
scanf("%s",&Msg1) ;

« VALEUR » D'UNE CHAÎNE DE CARACTÈRES

La « valeur » d'une chaîne de caractère concerne les opérations d'affectation réalisable directement avec une chaîne.

Une chaîne de caractères est une constante. Chaque fois que le compilateur rencontre une constante chaîne dans une affectation, il construit un tableau (sans nom) qui contient les caractères et remplace la chaîne par l'adresse de début du tableau. Une chaîne ne peut être affectée qu'à une variable qui peut contenir une adresse : un pointeur (*voir § Pointeurs*)

Ex : char PtMsg ; /* définition d'un pointeur */*
*PtMsg = "Bonjour"; /*PtMsg est un pointeur sur le tableau qui contient les codes de Bonjour*
Voir aussi ci-dessus Utilisation d'une chaîne par son adresse.

MANIPULATION DES CHAÎNES DES CARACTÈRES

Les chaînes de caractères peuvent se manipuler avec les fonctions de la bibliothèque standard présentées précédemment.

14.3) STRUCTURES

Une structure permet de désigner sous un seul nom un ensemble de données pouvant être de types différents. L'accès à chaque élément de la structure (nommé champ) se fait par son nom au sein de la structure.

Utilisation d'une structure en 3 étapes :

- **Déclaration d'un type modèle de structure** (ou plus simplement d'un type structure). Certains modèles sont déjà déclarés et livrés avec le compilateur C.
- **Définition d'une donnée de type modèle de structure** (on notera en abrégé « structure » la variable dont le type est un modèle de structure).
- **Utilisation des éléments de la structure**

COMPOSITION D'UNE STRUCTURE

Une structure est composée d'éléments ou membres ou encore **champs**. Chaque champ contient :

- un type simple de données
- ou un pointeur sur un type de données
- ou un type complexe de données (structure imbriquées, ...)

Les structures les plus simples sont constituées avec des champs contenant chacun un type simple de donnée.

DÉCLARATION D'UN TYPE (MODÈLE DE) STRUCTURE

La déclaration d'un type modèle de structure (ou encore type de structure) s'effectue comme suit, avec le mot clé **struct** :

```
struct <NomModèle>
        {<type> <champ1> ;
        <type> <champ2> ;
        <type> <champ3> ; ...} ;
```

Attention au **;** en fin de définition après **}**

Ex1 :
```
struct tTampon
{unsigned char Var1 ;
unsigned char Var2 ;
} ;
```

Ex 2 de modèle de structure (prédéfinie, livrée avec le compilateur Borland C) : le modèle time qui peut contenir l'heure (heures, minutes, secondes, centièmes de secondes).
```
struct time {
  unsigned char  ti_min;   /* minutes  */
  unsigned char  ti_hour; /* heures   */
  unsigned char  ti_hund; /* centièmes de
    secondes */
  unsigned char  ti_sec;   /* secondes */
};
```

La déclaration d'un nouveau type ne réserve pas de place en mémoire. Ceci s'effectue lors de la définition d 'une variable basée sur ce type.

DÉFINITION D'UNE DONNÉE DE TYPE (MODÈLE DE) STRUCTURE

La définition d'une variable de type modèle de structure s'effectue comme suit :
struct <NomModèle> <NomStructure1> [,<NomStructure2>, ...] ;

La définition de la donnée peut s'effectuer en même temps que la déclaration du modèle. <NomStructure> est placé avant le ; final

Ex : struct tTampon[2]
{...
....} Tampon1 ;

Dans ce dernier cas, il est possible d'omettre le nom du modèle, à condition, bien sûr qu'on n'ait pas à définir par la suite d'autres variables de ce type.
Ex : struct {...} Tampon1;

INITIALISATION D'UNE STRUCTURE

L'initialisation totale ou partielle s'effectue lors de la définition.
Les valeurs fournies doivent être du type constante ou expression constante.
Ex : struct tTampon[3] *Tampon1 = {valeur_champ1, ..., valeur_champn}*

L'initialisation d'une structure après sa définition se fait champ par champ.

UTILISATION D'UNE STRUCTURE

On peut utiliser une structure de 2 façons :
- de manière globale (affectation d'une structure à une autre de même modèle)
- en travaillant sur chacun de ses champs

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant.

L'identificateur d'une structure ne correspond pas à son adresse de début, comme c'est le cas pour un tableau. Pour travailler avec l'adresse d'une structure, il faut utiliser un pointeur.

La désignation d'un champ d'une structure varie selon qu'on emploie l'identificateur de cette structure ou un pointeur vers cette structure.
(voir § sur les pointeurs, plus loin).

Désignation d'un champ d'une structure repérée par son identificateur

La désignation d'un champ se note en faisant suivre le nom de la donnée structure de l'opérateur « point » (.), suivi du nom du champ, tel qu'il a été défini dans le modèle (le nom du modèle lui-même n'intervenant d'ailleurs pas).

```
Ex d'utilisation du modèle de structure précédent :
#include   <stdio.h>
#include   <dos.h>

int main(void)
{
  struct  time t;  /* définition de la variable  structure t basée sur le modèle time (prédéfini) */

  gettime(&t);    /* gettime() : fonction prédéfinie qui remplit les champs de time à partir de l'horloge temps réel du PC */
  printf("The current time is:%2d: %02d: %02d.%02d\n",    t.ti_hour,    t.ti_min, t.ti_sec,  t.ti_hund);
  return 0;
}
```

[2],[3] t pour indiquer qu'il s'agit d'un type

Désignation d'un champ d'une structure repérée par un pointeur

Voir aussi le § Pointeurs / Pointeur sur donnée / Pointeurs et structures.

Il faut d'abord déclarer un pointeur soit explicitement par une définition, soit implicitement lors de la définition du paramètre d'une fonction : <ModèleStructure>* <NomPointeur>

Ex : tTampon ptTampon*[4]

Chaque champ s'utilise avec NomPointeur suivi de -> et du nom du champ défini dans le modèle de structure.
<*NomPointeur*> -> <*champ*>// pas de *
Remarque : il est aussi possible de désigner le champ par :
(<NomPointeur>). <champ> /* () indispensables car l'opérateur . a la priorité la plus grande */*

On utilise un pointeur sur une structure dans le cas d'un passage d'adresse d'une structure en paramètre d'une fonction (*Voir ci-dessous)* ou dans le cas de liste chaînée (*Voir § Liste liée*).

PASSAGE DE L'ADRESSE D'UNE STRUCTURE EN PARAMÈTRE D'UNE FONCTION

Il est beaucoup plus rapide, lors de l'exécution du programme, de transmettre l'adresse d'une structure que de transmettre cette structure en entier. La transmission de l'adresse s'effectue avec un pointeur *(voir § sur les pointeurs, plus loin).*
Syntaxe dans définition fonction :
<Type> <fonc> (struct <NomModèle>* <NomPointeur)
{ ...

/* Pour accès à un champ : <NomPointeur> -> <NomChamp> *Voir ci-dessus* */
}

Syntaxe pour l'appel
<fonc> (&<NomStruct>) ;

14.4) CHAMPS DE BITS (BITS FIELDS)

Un champ de bit est un type de structure particulière qui permet de ranger dans un mot machine (taille variable selon le processeur du système cible) des informations de taille diverses non standard.
Ex : une donnée de 4 bits + 1 bit d'état + une donnée de 3 bits ...

Un champ de bits s'utilise en 3 étapes, comme une structure. Chaque champ a une longueur exprimée en nombre de bits.

DÉCLARATION D'UN TYPE

Forme identique à celle d'une structure avec :
- 2 types autorisés seulement pour les champs : int et unsigned int (même si chaque champ ne fait qu'un bit) (C ANSI) pour un mot machine de la taille d'un « int » [certains compilateurs acceptent char et unsigned char pour un mot machine de la taille d'un « char » (µP/µC 8 bits)].
- la longueur de chaque champ précisé en nombre de bits après « : » placé en fin

[4] *pt pour indiquer qu'il s'agit d'un pointeur*

Syntaxe
struct <nom_modèle>
{<type> <champ1> : 1 ;
<type> <champ2> : 2 ;
<type> <champ3> : 1 ; ...};

Les champs sont souvent définis du poids faible (1ère ligne) vers le poids fort (dernière ligne). La norme ANSI ne précise rien. *Voir doc compilateur utilisé.*

Les valeurs que peut prendre un champ de bit dépendent uniquement de la longueur du champ.
Norme C ANSI (à vérifier) : les valeurs peuvent être uniquement ≥ 0.
ex :

longueur champ	valeur
1 bit	0 ou 1
3 bits	0 à 7

DÉFINITION ET UTILISATION

Idem structure

INTÉRÊT DES CHAMPS DE BITS POUR SYSTÈME EMBARQUÉ

Pour un test ou une programmation d'une fonction d'un µP/µC (fin de temporisation, validation CAN...), il est parfois nécessaire d'accéder qu'à un seul bit d'un registre de contrôle ou d'état.
Par exemple la mise à un d'un registre de contrôle peut s'effectuer par une instruction du type RegCtrl.BitCtrl = 1;
La traduction sera courte avec des µP possédant des instructions de manipulation de bit.
Pour ceux n'en possédant pas, l'emploi d'un champ de bit est parfois déconseillé par l'éditeur du compilateur. Il est préférable de procéder alors par masquage sur une donnée d'un octet. *Voir doc compilateur.*

14.5) UNION

Une union est une structure spéciale. Elle permet de partager un emplacement mémoire par plusieurs champs pouvant correspondre à des types de données différentes. Lors du déroulement du programme, l'emplacement mémoire n'est occupé que par un des champs ; les champs ne sont jamais accessibles en même temps.
L'emplacement mémoire réservé pour une union correspond à la plus grande taille de chacun de ses champs.

Une union s'utilise en 3 étapes, comme une structure, le mot clé **union** remplace **struct**.

EMPLOI

Une union peut être utilisée pour accéder à un port d'E/S dans sa totalité ou bit à bit. Il suffit alors d'utiliser une union entre un octet

et un champ de bits. Ceci n'est possible qu'avec certains compilateurs.

```
Ex :
// Déclaration
union tPort
{
unsigned char Byte;
struct
  {unsigned char b0:1;
  unsigned char b1:1;
  unsigned char b2:1;
  unsigned char b3:1;
  unsigned char b4:1;
  unsigned char b5:1;
  unsigned char b6:1;
  unsigned char b7:1;
  }Bit;
};

//Définition
volatile union tPort PortC @ 0x1003;
...
// Utilisation
PortC.Bit.b0 = 1;
...
PortC.Byte = 0x7E ;
```

14.6) DONNÉES DE TYPES COMPLEXES IMBRIQUÉS

Il est possible de créer :

- des tableaux de structures
- des structures avec des tableaux dans les champs
- des structures de structures
- etc

Les règles d'emploi découlent de celles utilisées pour les différents types.
Pour l'utilisation de types complexes imbriqués, l'associativité de gauche à droite des opérateurs permet d'éviter l'emploi de parenthèses tout en gardant une syntaxe « naturelle ».

DÉCLARATION

Il faut d'abord déclarer la variable ou le modèle de plus bas niveau, c'est-à-dire celui qui ne fait appel à aucune variable ou modèle non déjà déclaré.

EXEMPLE 1

Le modèle de structure tEns[5] a été déclaré ailleurs. Il contient les champs champ1, champ2, ..., champn

Définition d'un tableau de structures avec initialisation des différents champs.
struct tEns Table[2]={{c1cel0, c2cel0, ...cncel0}, {c1cel1, c2cel1, ..., cncel1}} ;
*avec : cncel0 = **c**hamp **n cel**lule **0** du tableau*

Désignation d'un élément : *Table[1].champ1*
Cet élément correspond à c1cel1

EXEMPLE 2

Cet exemple concerne un tableau de structures dont un des champs est un tableau.

[5] *t indique qu'il s'agit d'un type*

Déclaration d'un type

```
typedef const struct
{
unsigned char LigneActive;
unsigned char CodesTouchesLigne[4];
} TLigne;
```

Définition

```
TLigne TableLecture[4]={
{0b11101111,'1','2','3','F'},
{0b11011111,'4','5','6','E'},
{0b10111111,'7','8','9','D'},
{0b01111111,'A','0','B','C'}};
```

Accès à un élément
Il s'effectue avec un pointeur (*voir ci-dessous*)

Remarque : les données de types complexes imbriqués font souvent appel aux pointeurs pour les accès ou pour leurs constructions. Voir plus loin le § Pointeurs / Pointeur sur donnée / Pointeur et structures.

15) TYPES DE DONNÉES PERSONNALISÉS

15.1) ÉNUMÉRATION (ENUM)

Permet de déclarer les différentes « valeurs » que prendre une variable. Les « valeurs » que peuvent prendre la variable sont désignées par des noms et traduites en internes par des entiers (int) :

- de façon transparente à l'utilisateur.
- avec les valeurs données par l'utilisateur (initialisation)

DÉFINITION D'UNE VARIABLE DE TYPE ÉNUMÉRATION

La définition d'une variable de type énumération s'effectue comme pour une structure. Il existe plusieurs méthodes :
1) déclaration d'un type énumération puis ensuite utilisation de ce type dans une définition d'une ou plusieurs variables
2) définition des variables de type énumération sans préciser de nom pour le type énumération
3) déclaration d'un type énumération et définition des variables.

Ci-dessous, on illustre les méthodes 2 et 3.

Avec affectation automatique identificateur / valeur numérique.

syntaxe	exemple
enum [<nom_type>] {<valeur1>, <valeur2>, ...} <nom_var>;	enum [tCommande] {MARCHE, ARRET, VEILLE} Commande;

La 1ère valeur numérique est 0; les autres suivent. Dans l'exemple, MARCHE vaut 0, ARRET vaut 1, VEILLE vaut 2.
Avec <nom_type> (dans l'exemple tCommande) : méthode 3. Sans : méthode 2.

Avec affectation par l'utilisateur

Même syntaxe avec une affectation sur les lignes souhaitées.
exemple:

```
enum tCommande
        {MARCHE=1,
        ARRET=5,
        VEILLE} Commande;
```

VEILLE vaut 6.

UTILISATION

En général dans test
exemple : if (Commande==MARCHE) ...

```
switch(Commande)
        {case MARCHE : ...
```

15.2) TYPES DE DONNÉES SYNONYMES

Pour une meilleure lisibilité des programmes ou une économie d'écriture, il est possible de déclarer des types synonymes.
Cette déclaration personnalisée peut s'appliquer à n'importe quel type de donnée, mais on l'utilise surtout pour les structures.

REDÉCLARATION DE TYPE SIMPLE

typedef <type_prédéfini> <nom_remplmnt1>, [<nom_remplmnt2>, ...] ;

```
ex : typedef int entier ;
typedef volatile unsigned char registre ;
/* registre est synonyme de volatile unsigned car */
```

utilisation comme n'importe quel type de donnée.

REDÉCLARATION DE TYPE MODÈLE DE STRUCTURE

plusieurs possibilités

Possibilité 1

typedef struct <NomModèle>
{.....} <NomSynonyme> ;

NomSynonyme est équivalent à struct NomModèle
utilisation pour définition d'une donnée :
NomSynonyme NomStruct1[, NomStruct2, ...]
Remarque : on arrive au même résultat en utilisant les lignes suivantes :
struct <NomModèle>
{...} ;
typedef struct <NomModèle> <NomSynonyme>

Possibilité 2

typedef struct
{....} <NomSynonyme> ;

NomSynonyme est équivalent à struct ModèleSansNom
L'utilisation pour une définition est le même que précédemment :
NomSynonyme NomStruct1[, NomStruct2, ...]
Remarque : cette dernière façon de procéder ne peut être traduite sans typedef. Dans tous les autres cas, il faut définir un NomModèle.

REDÉCLARATION D'UN TABLEAU AVEC SA TAILLE

le nom synonyme est placé après typedef, avant le nombre d'éléments du tableau.

ex :
typedef char Ligne [80] ;
Ligne est équivalent à char[80]

16) POINTEURS

Un pointeur est une donnée dont la valeur est une adresse de début d'une zone mémoire où est mémorisée une autre donnée ou fonction[6] (ou encore un autre pointeur). Un pointeur pointe sur une donnée (ou fonction) ou désigne l'adresse de cette donnée. Le pointeur est lui-même rangé à une adresse en mémoire.
Un pointeur **référence** une donnée.

16.1) UTILISATION DES POINTEURS

POINTEUR SUR UNE DONNÉE

Les pointeurs sont prévus pour réaliser un accès indirect à la donnée pointée.
Ils sont utilisés dans les cas suivant :

- dans une fonction pour recevoir en argument l'adresse d'une donnée (*voir § Fonctions / Transmission de paramètres / Transmission par adresse*)
- pour accéder à un élément d'une structure de données complexe (ex : tableau de structures) et pour construire des données complexes
- pour un accès à des données dynamiques (*voir § Données dynamiques*). Ces données n'ont pas d'identificateur et ne sont accessible qu'avec un pointeur.
- pour un accès indirect à un registre avec une adresse définie, par exemple un registre de configuration d'un port d'E/S (les compilateurs pour systèmes embarqués offrent des extensions au C ANSI qui permettent un autre moyen d'accès)

UTILISATION DANS UNE FONCTION POUR LA RÉCEPTION D'ARGUMENT

Les pointeurs sont utilisées dans les fonctions qui ont comme argument l'identificateur d'un tableaux ou d'une chaîne de caractère (nom de tableau ou de chaîne = adresse de début).

Les pointeurs sont fréquemment utilisés avec les fonctions qui manipulent des structures car il est plus rapide de passer l'adresse que de passer tous les constituants de la structure.

TYPES DE DONNÉES UTILISABLES AVEC DES POINTEURS

Les pointeurs peuvent servir pour manipuler tout type de donnée, sauf les champs de bit (pas d'adresse unique pour chaque champ de bit) et les données "register" (pas d'adresse).

Les pointeurs permettent de construire des structures de données complexes. *Voir § Types de données complexes / listes chaînées ainsi que ci-dessous les § Tableau de pointeurs.*

Pour pouvoir changer de donnée pointée (ex: passage d'un élément au suivant dans un tableau), il est possible d'effectuer quelques opérations élémentaires sur les pointeurs, c'est-à-dire sur les adresses des données qu'ils contiennent.
De même pour effectuer des test sur des zones de données pointées (début de tableau, fin de tableau, ...), il est possible d'effectuer des comparaison sur des pointeurs.
Voir ci-dessous § Arithmétique des pointeurs.

POINTEUR SUR UNE FONCTION

Une fonction peut être appelée par son identificateur (qui correspond à son adresse) ou par l'intermédiaire d'un pointeur. L'emploi de pointeurs de fonctions est assez rare, sauf dans quelques cas spécifiques.
Les pointeurs sur fonctions permettent de :

- fixer les adresses des fonctions gestionnaires d'interruption (table des vecteurs d'interruption) Possibilité utilisée uniquement avec certains compilateurs. *Voir ci-dessous § Tableau de pointeurs sur fonctions.*
- appeler une fonction ou une autre selon le calcul d'un indice (ou index). *Voir ci-dessous § Tableau de pointeurs sur fonctions.*
- appeler une fonction ou une autre selon une affectation préalable de pointeur (utilisation rare)
- recevoir, dans une fonction, l'adresse d'une autre fonction transmise en argument. Une fonction ou une autre peut être passée en argument (nom d'une fonction = adresse de cette fonction) (utilisation rare)

16.2) POINTEUR SUR DONNÉE

Avant toute utilisation, un pointeur doit être défini, comme toute donnée. Cette définition est implicite lors de la définition d'une fonction qui a en argument un pointeur.
Il faut ensuite lui affecter l'adresse de la donnée pointée. Ceci s'effectue par une affectation simple ou automatiquement dans le cas d'un paramètre d'une fonction.
On peut ensuite manipuler indirectement la donnée.

DÉFINITION

<type_donnée>* <nom_pointeur> ;
nom_pointeur est une variable de type type_donnée* = pointeur sur type_donnée ou nom_pointeur est un pointeur vers une variable de type type_donnée ou encore nom_pointeur a pour valeur l'adresse d'une variable de type type_donnée

ex : unsigned char ptCaractere*[7]

Les pointeurs sont "typés". Ceci est utile pour les opérations arithmétiques qui portent sur eux.

* n'a pas de place imposée : il peut être collé après type_donnée, avant nom_pointeur ou placé entre les deux. Il est plus logique de coller * après type_donnée.

La pratique veut que l'identificateur de nom_pointeur commence par p ou P ou Pt, ...

Attention :
<type_donnée>* <NomPtr1>, <NomPtr2>, ... ;
ne déclare pas 2 pointeurs, mais un seul NomPtr1. Les autres variables sont de type type_donnée. Pour déclarer plusieurs pointeurs de même type, il faut utiliser plusieurs lignes ou placer * avant chaque identificateur.

Remarque : dans certains cas, il est nécessaire de connaître précisément l'organisation mémoire du système cible pour déclarer un pointeur. *Voir § Organisation mémoire du système cible / Organisation de la mémoire dans les systèmes embarqués.*

AFFECTATION EXPLICITE

L'affectation peut s'effectuer lors de la définition ou après. Plusieurs types d'affectation :

- Directe. *Ex : <nom_pointeur> = <adresse> ; (non reconnu par certains compilateurs).* [1]
- Par l'adresse d'une donnée.
  Ex : <nom_pointeur>=&<nom_donnée>
  ou <nom_pointeur>=<nom_tableau>
  ou <nom_pointeur>=&<nom_tableau>[0]
  etc.
- Égalité avec un autre pointeur.
  Ex : <nom_pointeur1>=<nom_pointeur2>

[1] *A éviter en général; le programmeur ne connaît en général pas l'adresse d'une donnée ; de plus celle-ci peut changer après modification du programme. On peut utiliser cette possibilité pour définir l'adresse d'un port d'E/S qui sera accessible via un pointeur (voir plus loin, § utilisation pour un accès indirect à la donnée pointée).*

AFFECTATION EN PASSAGE DE PARAMÈTRE D'UNE FONCTION

On reprend l'exemple du § Fonctions / Transmission de paramètre par adresse.

... void multx2(int*) ; .... multx2(&a) ; ...	*déclaration* ***appel.*** L'adresse de a est transmis à la fonction
void multx2(int* ptX) {*ptX=*ptX*2 ;}	*définition* ptX : pointeur vers entier **lors de l'exécution, ptX reçoit l'adresse de a** ...

UTILISATION POUR UN ACCÈS INDIRECT À LA DONNÉE POINTÉE

Il faut utiliser l'opérateur d'indirection * qui a la même symbole que celui utilisé pour la définition d'un pointeur.
Pour l'utilisation dans une fonction, voir l'exemple § ci-dessus.
Pour l'affectation d'une valeur dans un registre de contrôle par exemple, il faut d'abord déclarer un pointeur qui contient l'adresse de ce registre, puis utiliser l'opérateur d'indirection.

ex :
unsigned char* ptPortE_S1 = 0x330 ;
...

[6] *il s'agit de l'adresse de début de la zone de mémorisation de la donnée ou de la fonction*

[7] *pt permet de distinguer un pointeur. On peut aussi utiliser p, P, Pt, etc.*

*ptPortE_S1=0x43 ;

TYPES DE POINTEURS

Un pointeur correspond à la fois à une adresse de la donnée pointée et au type de cette donnée. Ce "typage" des pointeurs est nécessaire pour l'arithmétique des pointeurs (*voir ci-dessous*).
Les tailles de l'adresse d'un pointeur et de l'adresse qu'il contient dépendent de la cible. Pour une même cible de type programme DOS sur ordinateur, elles peuvent varier selon le modèle mémoire retenu (voir § Organisation de la mémoire du système cible).
Avec un système embarqué à architecture Von Neumann, la taille des adresses est fixe (ex 16 bits pour un petit µC). Pour une architecture Harvard, la taille du pointeur dépend la situation de la donnée pointée (ROM ou RAM).

POINTEUR NUL (NULL)

En C, une donnée ne peut jamais être rangée à l'adresse 0.
Le pointeur "nul" contient l'adresse 0; il signifie qu'il ne pointe sur aucune donnée valide. Ceci est utilisé dans un élément d'une liste pour signifier qu'il n'y a pas d'élément suivant, par une fonction retournant normalement un pointeur valide pour signaler une erreur, ...

A la place de l'adresse 0, on utilise la constante NULL définie dans un fichier en-tête.
L'affectation d'un pointeur nul est :
NomPtr = NULL;

*POINTEUR GÉNÉRIQUE void**

Certaines fonctions manipulent des pointeurs sur des données (argument ou valeur retournée) qui peuvent varier d'un appel à l'autre.

Il existe en C un pointeur sur une donnée de type quelconque **void*** qui est utilisable dans de tels cas.

En général un pointeur générique est implicitement déclaré dans la définition d'une fonction.
Ex: en-tête de la fonction malloc()
void* malloc(unsigned int <NbOctets>)

Un pointeur générique doit ensuite être affecté à un autre pointeur soit explicitement, soit lors du passage d'argument.

Voir allocation dynamique de mémoire.

POINTEURS ET TABLEAUX OU CHAÎNES DE CHARACTÈRES

Il est possible d'accéder aux éléments d'un tableau ou d'une chaîne de caractères avec un pointeur.
Il est donc possible de réaliser quelques opérations sur un pointeur pour accéder à chacun des éléments.

En général, on commence par initialiser le pointeur à l'adresse de début du tableau ou de la chaîne, après avoir défini un pointeur sur un type correspondant aux éléments du tableau ou à un type caractère dans le cas d'une chaîne.

ex :
unsigned char* PtTab ;
...
PtTab = Tab ;

A l'aide des opérations décrites ci-dessous, on peut accéder à un autre élément.
Toutes les opérations avec les pointeurs tiennent automatiquement compte du type et de l'occupation mémoire des données pointées.

L'intérêt de l'utilisation d'un pointeur est de pouvoir réaliser le même traitement sur différents tableaux.

ADDITION ET SOUSTRACTION D'UN ENTIER

La quantité ajoutée (nb d'adresses) dépend du type de donnée pointée
Si PtTab pointe sur l'élément Tab[i] d'un tableau, alors PtTab+n pointe sur Tab[i+n]
ex : ElmntTab = *(PtTab +n) ;

INCRÉMENTATION / DÉCRÉMENTATION

Idem que pour addition / soustraction d'un entier.
ex : ElmntTab = *(PtTab ++) ;

POINTEUR AVEC INDICE

Non reconnu par tous les compilateurs.
Une case quelconque d'un tableau peut être pointée par le pointeur muni de l'indice de la case.
Si PtTab pointe sur le début d'un tableau Tab, PtTab[n] pointe sur Tab[n]. PtTab[n] est équivalent à PtTab +n
Un seul indice est possible avec un pointeur.

SOUSTRACTION DE 2 POINTEURS

Fournit le nombre d'élément du tableau compris entre les 2 pointeurs.

COMPARAISON DE 2 POINTEURS

Possible avec <, >, <=, >=, ==, !=

Les comparaisons n'ont de sens qu'entre 2 pointeurs pointant sur 2 éléments d'une même donnée (tableau, chaîne de caractère).
Un pointeur peut aussi être comparé au pointeur nul (*voir plus loin*).

TABLEAU DE POINTEURS

Définition : <type>* <nom> [<taille>] ;
Exemple : int* TableauPointeurs [10]

voir le § structure d'une définition

Un tableau de pointeurs peut être utilisé pour mémoriser les adresses de messages à afficher, etc.

POINTEUR ET STRUCTURES

Un pointeur est souvent utilisé pour accéder aux champs d'une structure.

Il faut d'abord déclarer un modèle de structure, puis une variable de ce type et enfin un pointeur sur ce type.

Un exemple simple est donnée au § Types de données complexes / Structures / Utilisation d'une structure.

Exemple sur des données complexes imbriquées

Déclaration
typedef const struct
{unsigned char LigneActive;
unsigned char CodesTouchesLigne[4];
} TLigne;

Définition
TLigne TableLecture[4]={
{0b11101111,'1','2','3','F'},
{0b11011111,'4','5','6','E'},
{0b10111111,'7','8','9','D'},
{0b01111111,'A','0','B','C'}};
...
TLigne* PtLigneTableLect=&TableLecture[0];
...

Accès à un élément

CodeToucheApp=(PtLigneTableLect -> CodesTouchesLigne[NumCol]);

remarque : on aurait aussi pu écrire : (*PtLigneTableLect). CodesTouchesLigne[NumCol])

16.3) POINTEUR SUR FONCTION

Un pointeur sur fonction correspond à l'adresse de début de la fonction.
Avant toute utilisation, un pointeur doit être défini, comme toute donnée. La définition est implicite dans une fonction qui a comme paramètre un pointeur sur fonction.
L'adresse de la fonction doit ensuite être affectée au pointeur.
L'appel de la fonction peut se faire avec son nom ou avec le nom d'un pointeur sur cette fonction.

SYNTAXE DÉFINITION

<type> (* <nomPtrFonction>) ([<type1>, <type2>,...]) ;
*Exemple : int (*ptfFct) () ;*
déclare un pointeur ptfFct sur une fonction qui retourne un entier et dont le nb et le type d'argument n'est pas précisé.

AFFECTATION EXPLICITE

<nomPtrFonction> = <NomFonction> ;
le nom de la fonction correspond à son adresse.
ex : ptfFct = Fct1 ;
Aucune parenthèse utilisée lors de l'affectation

AFFECTATION EN PASSAGE DE PARAMÈTRE À UNE FONCTION

La fonction doit être définie pour recevoir l'adresse d'une fonction dans un pointeur.

ex : float Integ(float (*ptfFct)(float), ...)
{ ...}

Lors de l'appel de la fonction, l'adresse de l'a fonction en argument est copiée dans le pointeur.

ex : Resultat = Integ(sin(), ...) ;
/* ptfFct reçoit l'adresse de sin() */

UTILISATION POUR UN APPEL DE FONCTION

Le pointeur de fonction doit d'abord avoir été initialisé avec l'adresse d'une fonction.
L'appel de la fonction s'effectue par :
(*<nomPtrFonction>)([<param1>,<param2>,...]]
ex : Valeur = (*ptfFct)(x) ;

TABLEAU DE POINTEURS SUR FONCTIONS

Un tel tableau contient les adresses de début des fonctions.
Ce type de tableau peut être utilisé pour :

- initialiser des vecteurs d'interruption (*avec certains compilateurs pour systèmes embarqués uniquement*). La définition seule du tableau suffit
- appeler une fonction ou une autre selon le calcul d'un indice (ou index)

DÉFINITION

La définition combine une définition de tableau de pointeurs imbriqués dans une définition de pointeur sur fonction.

syntaxe
<type> (* <nom_tableau>[<nb_case>] ([<param1>,<param2>,...][8]) ;

Exemple :
void (*TabPtrFonc[]) ()={fct1, fct2..., fctn}
/* définit un tableau de pointeurs sur n fonctions, avec initialisation des valeurs. fct1 correspond ici à l'adresse de début de cette fonction. */

INITIALISATION DE VECTEURS D'INTERRUPTION

Dans la plupart des µPs/µCs, les vecteurs d'interruptions se suivent sur une zone de mémoire d'un seul bloc.
Pour initialiser les vecteurs d'interruption, il faut :

1) déclarer dans un fichier seulement un tableau de pointeurs sur fonction et l'initialiser avec les noms des gestionnaires d'interruptions (fonctions)
2) compiler ce fichier
3) lors de l'édition de liens, fixer l'adresse de départ du fichier objet résultant de la compilation au début de la zone des vecteurs d'interruptions. Si un vecteur d'interruption n'est pas utilisé, placer NULL. (possible uniquement avec certains compilateurs)

ex : pour vecteur d'interruption du 68HC11 avec compilateur Cosmic

```
void (* const _vectab[])() = {
    InterLiaisonSerieSync , /* SCI */
    InterLiaisonSerieAsync, /* SPI  */
    NULL, /* Pulse acc input  */
    NULL, /* Pulse acc overf  */
    ...
    NULL, /* cop clock fail   */
    _stext /* RESET */
    };
```

le nom du tableau n'a aucune importance

APPEL D'UNE FONCTION SELON LE CALCUL D'UN INDICE

Cette méthode d'appel peut remplacer une structure switch(...) .
Après avoir calculé l'indice du tableau permettant d'accéder à la fonction voulue, l'appel s'effectue avec la **syntaxe** suivante :
(*<nomPtrFonction>[<indice>])([<param1>, <param2>,...])

ex : Valeur = 3+ (*ptfFonct[Indice])() ;

16.4) POINTEUR DE POINTEUR

Les pointeurs de pointeurs sont utilisés pour manipuler le contenu d'un pointeur (adresse de la donnée pointée).

Avec un pointeur de pointeur, on peut gérer des tableaux dynamiques de pointeurs. Non développé pour l'instant dans cet abrégé.

UTILISATION AVEC DES POINTEURS SUR TYPES DE DONNÉES DU C

Certaines fonctions manipulent un pointeur qui contient l'adresse d'une donnée. Pour pouvoir modifier le contenu (adresse de la donnée) de ce pointeur, la fonction doit recevoir en argument l'adresse de ce pointeur, c'est-à-dire un pointeur de pointeur.
Un pointeur de pointeur est déclaré par :
<Type>** <NomPtrPtr> ;

*Ex : la fonction **strtod()** convertit une chaîne de caractère en un nombre et renvoie dans un pointeur, dont l'adresse est passée en argument, l'adresse du 1er caractère non traduit (normalement la fin de la chaîne).*
Le prototype de la fonction est :
double strtod(char PtrChaine, char** PtrPtrFinChaine)*
PtrChaine reçoit l'adresse de début de la chaîne, PtrPtrFinChaine reçoit l'adresse du pointeur qui contiendra l'adresse du dernier caractère traduit.
L'appel de la fonction s'effectue par
Resultat = strtod(NomChaine, &PtrFinChaine) ;
PtrFinChaine doit avoir été déclaré avant.

UTILISATION AVEC UN CHAMP D'UN ÉLÉMENT D'UNE LISTE LIÉE

Une liste liée est souvent décrite par un descripteur qui contient un pointeur sur le premier élément de la liste, un sur le dernier élément et un sur l'élément en cours de traitement.
Chaque élément contient un pointeur sur l'élément suivant.
A compléter

17) STRUCTURE D'UNE DÉFINITION DE DONNÉE

Ce § récapitule les définitions rencontrées précédemment.
Une définition comprend 4 parties :

- un « descripteur » optionnel
- un type de base
- un déclarateur
- un initialiseur optionnel

Une définition se termine toujours par un point virgule.

Ex : char ptCouleurs[]={« Rouge », « Vert », « Bleu »} ;*
char : type de base
** ptCouleurs[] : déclarateur*
{...} : initialiseur
ptCouleurs[] est un tableau de pointeurs sur des chaînes de caractères

Un descripteur est un mot clé initial tel que extern spécifiant un attribut de ce qui est déclaré.
Un déclarateur se compose d'un nom et de quelques opérateurs de déclaration en option.
Les opérateurs les plus courants sont :

*	pointeur	préfixe
[]	tableau	postfixe
()	fonction	postfixe

Les opérateurs de déclaration de type postfixe possèdent une influence supérieure à celle des préfixes. Ceci explique que dans l'exemple précédent ptCouleurs est un tableau de pointeurs sur chaînes.

18) ORGANISATION DE LA MÉMOIRE DU SYSTÈME CIBLE

18.1) SEGMENTS

Les données, variables et constantes, et les instructions occupent dans le système cible des espaces mémoire bien déterminés. On utilise le terme de **segment** de mémoire.
L'organisation de la mémoire dépend du système cible. Dans un ordinateur, tous les segments sont en mémoire vive ; dans un système embarqué, des segments sont en RAM et d'autres en ROM.
Le nombre et l'appellation des segments varient selon les compilateurs et les cibles. On retrouve toujours :

- 1) un segment pour le code exécutable, les constantes déclarées, les valeurs initiales des variables globales données lors des définitions
- 2) un segment pour les données statiques (variables globales, taille connue à la compilation)
- 3) un segment pour les variables locales et les paramètres des fonctions : la **pile** (stack) (*voir § Passage de paramètres / variables locales*). *Ce segment n'est pas la pile avec les µC sans pointeur de pile manipulable (ex PIC).*
- 4) un segment pour les variables dynamiques : le **tas** (heap) (*sauf pour les µC de*

[8] *Les 2ème crochets [] signifient facultatifs, alors que les 1er sont la marque d'un tableau.*

faibles ressources qui n'ont pas la possibilité de gérer les variables dynamiques avec le programme en C)

Pour les systèmes embarqués, le segment 1 est en ROM (ou EPROM, ...), les autres segments sont en RAM.
Voir plus loin les particularités pour systèmes embarqués.

Le compilateur détermine automatiquement ces segments d'après le ou les fichiers source. Dans certains cas, il est cependant nécessaire de connaître l'organisation en segments de la mémoire, car il faut fixer des options ou donner des commandes pour la compilation.
Par exemple, avec une application en mode DOS sur un ordinateur, des options permettent de choisir entre plusieurs modèles de mémoire, chaque modèle étant caractérisé par les tailles maximales des segments.
Pour un système embarqué, avec la plupart des µPs/µCs et compilateurs, il faut fixer les adresses limite de chaque segment. Ceci s'effectue avec un fichier de commande utilisé par l'éditeur de liens. *Voir § Détail de la compilation / Éditeur de liens.*

18.2) ORGANISATION DE LA MÉMOIRE DANS LES SYSTÈMES EMBARQUÉS

La connaissance de l'organisation de la mémoire est utile dans les cas suivants :
- µPs/µCs avec possibilité d'accès au code et données par banque ou page : les variables et les pointeurs peuvent être placés dans une page particulière (meilleure efficacité dans l'adressage), etc.
- définition des pointeurs avec des µCs à architecture Harvard (bus séparés pour le code + les constantes en ROM et les données en RAM. ex : PIC)

Des mots-clé qui sont des extensions du C ANSI doivent être utilisés lors de la définition de certains pointeurs et de certaines données.
Cette partie ne peut être détaillée ici. Voir doc du compilateur utilisé.

19) DÉTAIL DE LA COMPILATION

Ce qu'on appelle couramment la compilation correspond à plusieurs phases. Chacune de ces phases met en œuvre un logiciel spécifique. L'enchaînement des phases est automatique et transparente à l'utilisateur dans le cas d'environnement de développement intégré.

Les programmes mis en œuvre sont :
- le préprocesseur
- le compilateur qui peut être décomposé en plusieurs parties
- l'optimiseur de code
- l'assembleur
- l'éditeur de liens

Des commandes ou des options s'appliquent en général aux différentes phases de la compilation.

19.1) PRÉPROCESSEUR

Le préprocesseur produit un fichier texte. Il traite les directives introduites par #. Il permet de réaliser :
- des substitutions de texte (*voir précédemment équivalence symbolique et macro*)
- des inclusions de fichiers texte (pour les en-tête, définitions de fonctions, ...)
- des suppressions / sélections de parties du fichier source selon des conditions (compilation conditionnelle, *voir plus loin*)
- la suppression des commentaires

19.2) COMPILATEUR

Le compilateur lit ce que génère le préprocesseur et crée les lignes en langage d'assemblage correspondantes. Le compilateur lit le texte source une seule fois du début du fichier à la fin. Cette lecture conditionne les contrôles qu'il peut faire, et explique pourquoi toute variable ou fonction doit être déclarée avant d'être utilisée.

La 1ère phase consiste en une analyse lexicale et syntaxique. S'il y des erreurs décelées durant cette phase, un message d'erreur est généré et la traduction en langage d'assemblage n'est pas exécutée.

Certaines lignes d'un fichier sources utilisant uniquement des instructions standard du C peuvent être traduites avec des appels de fonctions prédéfinies (sous programmes), ou des portions de programme terminées par un saut (au programme appelant), dont le code exécutable est placé dans une bibliothèque d'exécution (runtime library ou machine library). Ces bibliothèques doivent être utilisées par l'éditeur de liens.

La plupart des compilateurs rajoute le caractère souligné _ devant les identificateurs lors de la traduction. Ceci est important lorsque des parties de programmes sont écrites en langage d'assemblage, avec utilisation d'identificateurs déclarés en langage C.

La compilation possède souvent des possibilités d'optimisation. Une compilation avec optimisation est plus longue mais génère un code plus compact.

19.3) OPTIMISEUR DE CODE

L'optimiseur mentionné ici intervient après la compilation (qui peut posséder aussi une optimisation). Il remplace des séquences types d'instructions en langage d'assemblage par d'autres plus courtes ou plus rapides à l'exécution.
L'optimisation est surtout importante pour les systèmes embarqués de faibles ressources.

19.4) ASSEMBLEUR

L'assembleur prend le code généré par le compilateur, éventuellement modifié par l'optimiseur, et génère un fichier en format relogeable (les adresses exactes des différents segments seront déterminées plus tard) ou absolu.
Le format relogeable est utilisé avec une compilation séparée. Ce fichier possède des références insatisfaites qui seront résolues par l'éditeur de liens.

19.5) ÉDITEUR DE LIENS OU LIEUR

L'éditeur de liens prend le ou les fichiers en format relogeable et les associe pour créer un module exécutable. Il se sert de bibliothèques pour résoudre les références indéfinies (noms des fonctions qui n'ont pas été définies dans les fichiers source), en particulier la bibliothèque standard. Il utilise aussi un module spécial, qui contient le code de démarrage du programme.
C'est au niveau de l'éditeur de liens que sont fixées les adresses absolues de chaque segment

FICHIER DE COMMANDE

La plupart des éditeurs de liens utilisent un fichier de commande[1] qui contient ou peut contenir :
- la référence du µP/µC cible (uniquement pour les éditeurs de liens « universels »)
- les adresses des segments
- la référence du programme de démarrage
- la référence de la ou des bibliothèques systèmes
- diverses initialisations

1 : les fichiers de commande varient beaucoup d'un compilateur à l'autre. *Voir la doc du compilateur utilisé.*

PROGRAMME DE DÉMARRAGE

Ce programme permet de réaliser un certain nombre d'initialisations puis d'appeler **main**. Ce programme est écrit en langage d'assemblage. L'éditeur du compilateur fournit un ou plusieurs programmes de démarrage (fichier source et exécutable). Son nom est du style CstartUp. L'utilisateur peut éventuellement modifier le fichier source puis l'assembler.
Parmi les initialisations concernant les systèmes embarqués, on trouve :
- l'initialisation du pointeur de pile quand cela est nécessaire
- la mise à 0 de toutes les variables globales non initialisées
- l'initialisation des variables globales initialisées.

Pour l'initialisation des variables globales, un bout de programme recopie leurs valeurs placées en ROM les unes à la suites des autres vers leurs emplacements en RAM qui se suivent aussi.

FICHIER EXÉCUTABLE

Dans le cas d'un système embarqué, le module exécutable contient des adresses absolues. Dans le cas d'un micro-ordinateur, le module exécutable est chargé en mémoire de travail par un chargeur sous contrôle du système d'exploitation. C'est lors de cette phase que sont fixées les adresses absolues des segments.

L'éditeur de liens utilise :

- le fichier de commande précédemment décrit
- les fichiers objet produits par l'assembleur + optimiseur après le compilateur
- la bibliothèque système fournie avec le compilateur (parfois non documentée)
- les bibliothèques documentées fournies avec le compilateur
- les bibliothèques créées par l'utilisateur
- le programme de démarrage fourni avec le compilateur (souvent personnalisable)

L'éditeur de liens n'extrait des bibliothèques que le code correspondant aux fonctions utilisées.

Tous les fichiers ou bibliothèques utilisés par l'éditeur de liens doivent être indiqués à celui-ci.
Les bibliothèques, hors bibliothèque système, sont connues avec les fichiers .h (de même nom que les bibliothèques) inclus dans les fichiers source.
Les différents fichiers objets et la bibliothèque système peuvent être :

- nommés dans un fichier de commande écrit par l'utilisateur
- en partie déduits des fichiers source du projet et en partie sélectionnés avec des boîtes de dialogue (cas d'un environnement de développement intégré avec gestion de projet)

La bibliothèque système devant systématiquement être utilisée, certains compilateurs ne nécessitent pas de la mentionner.

20) BIBLIOTHÈQUES

20.1) BIBLIOTHÈQUES ET GESTION DES BIBLIOTHÈQUES

Les bibliothèques sont des recueils de fonctions.

La quasi totalité des compilateurs est livrée avec des fichiers .lib qui sont des recueils de fonctions déjà compilées. A chaque fichier .lib correspond un fichier .h qui contient la déclaration des fonctions.
L'utilisateur peut créer ses propres bibliothèques. Un gestionnaire de bibliothèques (librarian), fourni avec le compilateur, permet de créer et modifier les bibliothèques (en ajoutant ou enlevant des fonctions, etc.)

Quelques compilateurs pour µCs de faible ressource ont un fonctionnement différent.
Avec le compilateur CCS pour µC PIC, les bibliothèques sont intégrées dans le ou les fichiers exécutables (.exe ou .dll). L'utilisateur ne peut créer ses propres bibliothèques compilées.
Avec le compilateur CC5X pour µC PIC, seules quelques unes des fonctions habituellement fournies sont disponibles. Elles sont dans des fichiers .h. Ceux-ci comprennent la définition des fonctions. Des directives particulières permettent de ne pas compiler les fonctions non utilisées. L'utilisateur peut créer ses propres bibliothèques non compilées avec la même organisation.

20.2 FONCTIONS DE LA BIBLIOTHÈQUE STANDARD POUR LES ENTRÉES SORTIES

Sur un PC, les échanges se font la plupart du temps avec le flux stdin (entrée) et stdout (sortie) définis dans le fichier stdio.h
L'entrée standard est par défaut le clavier et la sortie standard est l'écran.

Sur un µC, les fonctions pour les entrées/sorties font appel aux fonctions de base putchar() et getchar(). Il est ainsi possible de définir la liaison série asynchrone pour l'entrée et la sortie standard. Une commande « d'impression (print) » correspond à l'émission d'une suite de caractères.
Avec les compilateurs pour µCs de faibles ressources, toutes les fonctions ci-dessous ne sont pas fournies. Lorsque prinft() est fournie, les codes de format supportés sont réduits.

SORTIE : putch, putchar (STDIO.H)

Sur un PC, la macro putch affiche un caractère sur l'écran.
Sur un PC, la macro putchar permet la sortie d'un caractère sur le flux stdout. Elle renvoie le caractère sorti.
Utilisation : putch(c), putchar(c)
Valeur renvoyée : si succès, c ; si echec, EOF (fin de fichier).

SORTIE : printf (STDIO.H)

La fonction printf permet une sortie **formatée** sur flux stdout (pour un PC, défini dans le fichier stdio.h). Pour chaque donnée à sortir, le programmeur spécifie :

- un texte d'accompagnement
- le format d'affichage (entier en notation décimale, réel sous la forme mantisse/exposant, ...) des variables (ou des expressions avec variable(s)) qui suivent
- les noms des variables (ou des expressions avec variable(s)) dont on veut afficher les valeurs

Les 2 derniers points sont facultatifs
printf retourne le nb de caractères sortis.

Pour un système embarqué, seules quelques unes des possibilités mentionnées ci-dessous sont disponibles. printf() nécessite en effet des ressources relativement importante (taille du code en mémoire, temps d'exécution). Il est même possible de « personnaliser » printf() avec quelques options, avant de compiler la fonction et de la placer dans la bibliothèque standard.

La forme générale de printf est :
printf("contrôle" [, exp1, exp2, ...]) ;
[] : facultatif

La partie contrôle est écrite entre ". Elle contient le texte à afficher et les indications de format pour les arguments (expressions) qui suivent.
Les arguments correspondent à des expressions ou des variables et n'existent que si une ou plusieurs indications de format ont été fournies dans la partie contrôle.

*Ex : **printf("a vaut %d, b vaut %d", a, b) ;** lors de l'exécution du programme, on obtient le message suivant : a vaut [valeur de a en décimal], b vaut [valeur de b en décimal]*

Partie contrôle de printf

Contient du texte et les formats d'affichage définis par des codes de format commençant par %. Le premier code concerne la 1ère variable indiquée après la partie contrôle ; le 2ème code, la 2ème variable, ...
On peut inclure dans le texte un code d'échappement pour une mise en forme de l'affichage (saut de ligne, tabulation, ...)

CODES DE FORMAT

Structure d'un code de format :
%[drapeaux][largeur][.précision][h|l|L] code conversion
[] : facultatif

Le **code conversion** est le seul obligatoire. Il précise à la fois le type de l'expression et la façon de représenter sa valeur

code	Type de l'expression / représentation
d	Nombre entier signé / décimal
o	Nombre entier non signé / octal
x (X)	Nombre entier non signé / hexadécimal chiffres : 0, ..., a b, c ... à f (A, B, ..., F)
u	Nombre entier non signé / décimal
c	caractère ASCII
s	chaîne de caractère
e E	Nombre réel / virgule flottante avec 1 chiffre avant la virgule, 6 chiffres après la virgule et 3 chiffres pour l'exposant
f	Nombre réel / virgule fixe avec 6 chiffres après la virgule
g	comme %e ou %f selon la valeur à afficher
p	Pointeur / hexadécimal

drapeau	signification
-	justification à gauche
+	signe toujours présent
^	impression d'un espace au lieu de +
#	forme alternée ; n'affecte que les codes o, x, X, e, E, f, g, G comme suit : 0 : fait précéder de 0 toute valeur non nulle x ou X : fait précéder de 0x ou 0X la valeur affichée e, E ou f : le point décimal apparaît toujours g ou G : même effet que e ou E, mais les zéros de droite ne seront pas supprimés.

La **largeur** désigne le nombre total de chiffres affichés (avant et après la virgule).

Largeur	signification
n	au minimum, n caractères seront affichés, éventuellement complétés par des blancs à gauche

0n	au minimum, n caractères seront affichés, éventuellement complétés par des zéros à gauche
*	la largeur effective est fournie dans la liste d'expressions : exp1, exp2, ...

n : constante entière en notation décimale

La **précision** peut avoir plusieurs significations, comme indiqué dans le tableau suivant :

précision	signification
.n	dépend du code de conversion : d, i, o, u, x ou X : au moins n chiffres seront imprimés. Si le nombre comporte moins de n chiffres, l'affichage est complété par des zéros à gauche e, E ou f : n chiffres après le point décimal, avec arrondi du dernier g, G ou s : au maximum n chiffres significatifs ou caractères seront affichés
.0	dépend du code de conversion : d, i, o, u, x ou X : choix de la valeur par défaut de la précision e, E ou f : pas d'affichage du point décimal
*	la valeur effective de n est fournie dans la liste des expressions.

Voir Particularités pour systèmes embarqués

ENTRÉE : getchar (STDIO.H)

La macro getchar renvoie le caractère entré depuis stdin. En cas d'erreur, la fonction renvoie EOF.

ENTRÉE : scanf (STDIO.H)

La fonction scanf permet une entrée formatée depuis le flux stdin. Plusieurs données séparées par un séparateur peuvent être entrées avec un seul appel de scanf.
scanf renvoie le nombre de données convenablement lues.
Ne sont traitées ici que les possibilités les plus couramment employées.
Pour chaque donnée à entrer, le programmeur spécifie :

- le format de saisie (entier en notation décimale, ...) de chaque donnée à entrer
- l'adresse où est mémorisée chacune des données

La forme générale de scanf est :
scanf("liste_format ", liste_d'adresses)

La liste_format contient un code format par donnée à entrer dont l'adresse suit.
La liste_d'adresses contient une adresse pour chacun des codes format.
Ex : scanf("%d %d", &a, &b)

Règles d'écriture de la liste_format
les codes format peuvent collés ou être séparé par un ou plusieurs espaces.

Séparateur entre données lors de l'entrée
Il faut placer entre 2 données un séparateur qui peut être : un espace, une tabulation horizontale, une fin de ligne, un retour chariot (Entrée), une tabulation verticale, un changement de page.

CODES DE FORMAT
Structure d'un code de format :

%[*][largeur][F|N][h|l] code conversion
[] : facultatif

Le **code conversion** est le seul obligatoire. Il précise à la fois le type de l'expression et la façon de représenter sa valeur

Les codes conversions sont les mêmes que pour printf, à l'exception de c.
c correspond à 1 caractère lorsque la largeur de saisie est de 1 caractère (valeur par défaut) ou à une suite de n caractères si la largeur de saisie le spécifie.

La **largeur** indique le nombre maximal de caractères à prendre en compte.

21) PARTICULARITÉS POUR SYSTÈMES EMBARQUÉS

Particularités	Répercussions
Programme en ROM	Valeurs initiales des variables globales en ROM recopiées en RAM après la mise sous tension
Taille réduite du programme (qqs ko à qqs diz de ko)	Compacité du code produit à la compilation
Exécution rapide (« temps réel »)	• Variables placées en RAM interne ou registres internes au µP/µC car accès plus rapide • Possibilité de mélanger des parties ou modules écrits en langage d'assemblage avec le corps du programme écrit en C.
Utilisation de nombreuses sources d'interruption	Manipulation possible des fonctions d'interruptions depuis le source en C, avec établissement de la table des vecteurs d'interruption.
Accès direct aux registres de contrôle et d'état	Accès directs possibles sans passer par des fonctions prédéfinies ou des pointeurs
Taille RAM réduite	Possibilité de ne pas utiliser la pile pour le passage de paramètres (empilage lors de chaque appel de fonction imbriquée)

Pour couvrir ces particularités, les compilateurs & éditeurs de liens utilisent :

- un programme de démarrage (startup) lié lors de la phase d'édition de liens pour la recopie des valeurs des variables statiques (entre autres)
- des possibilités du C peu exploitées pour les programmes pour PC
- des extensions du C ANSI

21.1) ACCÈS DIRECT AUX REGISTRES

Pour un accès direct à un registre, il faut définir l'adresse absolue d'un registre. L'utilisation s'effectue de façon classique.

DÉFINITION DE L'ADRESSE D'UN REGISTRE

Ceci s'effectue avec des extensions du C ANSI. Il n'y a donc pas de normalisation

Ex : définition d'un registre 8 bits

compilateur	syntaxe	remarque
Cosmic	char DDRC @ 0x1007 ;	@ précède l'adresse absolue
IAR	sfrb IO_PORT0=0x0E;	sfrb : mot-clé, extension C ANSI

UTILISATION

Comme n'importe quelle variable

Ex : DDRC=0xA7 ;

RESTRICTIONS SUR L'UTILISATION DES REGISTRES

Modification indépendamment du programme

Le mot clé **volatile** (C ANSI) permet d'indiquer au compilateur qu'un registre peut être modifié indépendamment du programme (registre d'entrée, registre d'état).

Exemple avec compilateur COSMIC pour 68HC11 :
volatile char PORTE @ 0x100A ;

Accès registres en lecture seule

Le mot clé **const** (C ANSI) placé devant la définition d'un registre permet d'indiquer qu'il n'est accessible qu'en lecture (ceci revient à dire qu'il est modifié en dehors du programme). Si le programme spécifie une écriture, une erreur est générée lors de la compilation.

Exemple avec compilateur IAR pour 80C196 :
const sfrb AD_RESULT_LO = 0x00 ;

« FONCTIONS » INTÉGRÉES OU INTRINSÈQUES

Certains compilateurs fournissent des « fonctions » intégrées (built-in functions) ou intrinsèques qui permettent de réaliser des accès direct aux divers registres (port, commande, état).
Par exemple, le compilateur CCS pour PIC fournit de nombreuses « fonctions » dont OUTPUT_B().
Usage : OUTPUT_B(Valeur) ;
Cette fonction permet de fixer la valeur du port B. Avec un autre compilateur, on aurait écrit PORTB= Valeur ;
Ces « fonctions » ne sont pas de véritables fonctions. Elles sont traduites lors de la compilation par une ou plusieurs instructions, sans les mécanismes d'appel de fonction décrit précédemment.

21.2) GESTIONNAIRE D'INTERRUPTION

Un gestionnaire d'interruption (interrupt handler ou Interrupt Service Routine) est une

fonction particulière. Elle ne peut être appelée par le programme lui-même. Elle entre en service uniquement après une interruption matérielle.

Une fonction gestionnaire d'interruption est introduite par un mot clé. Il n'y a pas de norme.

- La fonction doit renvoyer à une table des vecteurs d'interruptions lorsque les différentes sources d'interruptions appellent chacune un gestionnaire différent. Il existe plusieurs possibilités selon les compilateurs :
- une directive particulière permet de déclarer une fonction d'interruption et de l'associer à un n° d'interruption défini dans la documentation du µP/µC ; le compilateur construit automatiquement la table des vecteurs d'interruptions. L'écriture de la fonction d'interruption est identique à une fonction normale (ex compilateur INTEL C96)
- l'utilisateur indique dans la définition de la fonction d'interruption l'adresse du vecteur d'interruption (ou un déplacement par rapport à une base définie ailleurs) ; la table des vecteurs d'interruptions est automatiquement construite (compilateur IAR pour 80C196)
- l'utilisateur doit lui même placer les adresses (correspondant aux noms des fonctions) dans la table des vecteurs d'interruptions (compilateur COSMIC)

Exemples de définition de fonctions d'interruption

Compilateur INTEL C96
```
#pragma interrupt (timer1_isr=0)
void timer1_isr (void)
{...}
```
#pragma ... précède une définition classique de fonction.
0 est le n° de l'interruption donnée dans la documentation technique du 80C196KB. La table des vecteurs d'interruption est automatiquement construite.

Compilateur IAR pour 80C196
```
interrupt [0x00] void TM_OVFL(void)
{...}
```
0x00 est le déplacement qui permet de calculer, au moment de l'édition de lien, l'adresse de la fonction d'interruption. La table des vecteurs d'interruption est automatiquement construite

Compilateur Cosmic pour 68HC11
```
@ interrupt void timer_ovfl (void)
{...}
```
L'utilisateur doit placer timer_ovfl (qui correspond à l'adresse de début de la fonction) dans la table des vecteurs d'interruptions (tableau de pointeurs sur fonctions)

Compilateur Hitech pour PIC
```
interrupt void TraitInter(void)
{...}
```
Le PIC ne possède qu'un seul gestionnaire d'interruption. Dans ce gestionnaire, il faut que le programmeur place les tests des différents drapeaux des sources d'interruptions possibles.

Pour les µC PIC, le compilateur CCS possède des directives qui permettent d'écrire uniquement la partie du gestionnaire qui concerne chacune des interruptions. Les tests des drapeaux et les acquittements sont automatiquement générés.

21.3) CONTRÔLE DE BAS NIVEAU DU µP/µC

Il faut pouvoir depuis le programme source en C autoriser / inhiber les interruptions et avoir accès à d'autres instructions de bas niveau qui dépendent des µP/µC (mise en veille, ...).
Ceci peut s'effectuer de 2 façons, selon les compilateurs :

- en insérant une ou plusieurs lignes d'assembleur avec une fonction C spécifique (voir plus loin)
- en utilisant des fonctions C spécifiques qui sont traduites chacune par une instruction assembleur ou une courte séquence d'instructions.

« FONCTIONS » C SPÉCIFIQUES

Les compilateurs pour µC disposent très souvent de « fonctions » spécifiques ou intrinsèques (built-in).
Par exemple, le compilateur IAR dispose de quelques fonctions « intrinsèques », dont :
enable_interrupt() ; // permet de valider les interruptions.

Certaines de ces « fonctions » sont traduites par un seul code machine (une ligne en langage d'assemblage). Il ne s'agit pas de fonction au sens habituel du terme, avec le mécanisme d'appel, etc.

21.4) INSERTION DE LIGNE EN LANGAGE D'ASSEMBLAGE OU DE CODE OPÉRATOIRE

Cette possibilité doit être utilisée avec précaution, car il ne faut pas modifier les registres utilisés par le reste du programme (registres bien souvent inconnus du programmeur).
Cette possibilité doit surtout être utilisée pour des instructions du type validation/inhibition des interruptions, ...

Il n'y a pas de normalisation.
Ex :

Compilateur	Syntaxe
Cosmic pour HC11	_asm (« cli «) ; /*cli : mnémonique */
IAR pour 80C196	_opc (0xFB) ; /* FB code opératoire */
Hitech pour PIC	#asm nop #endasm

Il est préférable de lier des modules obtenus par assemblage aux modules obtenus par compilation lors de l'édition de liens, plutôt que d'insérer des lignes d'assembleur ou des codes opératoires.

21.5) MÉLANGES DE MODULES OBTENUS PAR ASSEMBLAGE ET COMPILATION (SOURCE LANGAGE D'ASSEMBLAGE / SOURCE C)

La définition du sous programme (module) s'effectue en langage d'assemblage ; dans le programme source en C, l'appel de ce sous programme correspond à un appel de fonction.
Il est possible de transmettre des paramètres.

Pour appeler une fonction (module) obtenue par assemblage dans un programme source en C, il faut :

- que la fonction appelée en langage C ait un nom (identificateur) public (vrai par défaut) et que le nom du sous programme en langage d'assemblage soit déclaré externe
- que le nom en langage d'assemblage soit celui de l'identificateur en langage C précédé du caractère _ (MaFonction → _MaFonction). Ceci est vrai pour tous les compilateurs.
- respecter le passage de paramètres (convention d'appel) et le passage de la valeur de retour, si nécessaire, entre modules.

Voir ci-dessous, passage de paramètres

Les différents segments (code, variables, ...) doivent être identiques à ceux utilisés par défaut lors de la compilation.

22) PASSAGE DE PARAMÈTRES / VARIABLES LOCALES

Le passage de paramètres est transparent à l'utilisateur lorsque tous les modules sont écrits en C.
Le type de passage de paramètre est cependant important lorsqu'on dispose d'une RAM et d'une ROM de faibles capacités (µC des systèmes embarqués) ou lorsqu'on veut lier des modules avec des sources en langage C et en langage d'assemblage. Une fonction (sous programme) écrite en langage d'assemblage doit respecter le passage des paramètres imposé par le programme appelant écrit en C.

22.1) LES DIFFÉRENTES ÉTAPES DE L'EXÉCUTION D'UNE FONCTION

L'appel d'une fonction correspond à un appel de sous programme.

PRÉPARATION À L'APPEL

Lors de l'exécution, le programme appelant place en mémoire ou dans des registres les paramètres de la fonction. Le programme appelant appelle ensuite le sous programme.

APPEL

La pile est utilisée automatiquement pour sauvegarder l'adresse de retour au programme appelant (compteur ordinal).

EXÉCUTION DE LA FONCTION

La fonction utilise éventuellement des emplacements mémoire pour les variables locales. Si ces emplacements sont dans la pile, la fonction modifie le pointeur de pile en conséquence, pour libérer de l'espace dans cette pile (*voir plus loin)*
La fonction accède aux paramètres.
En fin, elle place la valeur de retour dans un emplacement mémoire ou en registre. L'emplacement peut être le même que l'un des paramètres.
Avant le retour au programme appelant, le sous programme doit remettre l'environnement du µP/µC dans son état initial (pointeur de pile si celui-ci a été modifié).

RETOUR

Lors du retour l'adresse de retour est récupérée et le pointeur de pile est replacé dans le même état que lors de l'appel.

Le programme appelant récupère éventuellement la valeur de retour et libère les emplacements mémoire utilisés pour le passage de paramètres.

22.2) DIFFÉRENTS TYPES DE PASSAGE DE PARAMÈTRES

Le passage de paramètres n'est pas normalisé en C.
Les paramètres et la valeur de retour peuvent être passés de 2 façons différentes :

- par une zone réservée dans la pile
- par des registres internes du µP/µC (ou des emplacements mémoire) prévus à cet usage par le compilateur

Les variables locales et les variables temporaires nécessaires à l'exécution de la fonction sont aussi sauvegardées de la même façon que les paramètres. Les § suivants ne mentionnent donc que les passages de paramètres

Dans les applications pour ordinateurs, qui disposent de grandes quantités de RAM, les paramètres et les variables locales sont dans la pile.
Les compilateurs pour µC à destinations de systèmes embarqués ont des options pour minimiser au maximum la taille de RAM et donc de la pile.

Selon les compilateurs et les options, on dispose des possibilités suivantes pour le passage des paramètres:

- tous les paramètres dans une zone de la pile
- 1er paramètre par registre et autres par la pile (méthode très utilisée avec les compilateurs pour systèmes embarqués)
- tous les paramètres par registres ou zone mémoire

Ces différentes possibilités peuvent êtres choisies

- globalement par une option de compilation
- localement pour chaque fonction en faisant précéder son en-tête par un mot clé.

22.3) PASSAGE DE PARAMÈTRE(S) UNIQUEMENT PAR LA PILE

La préparation à l'appel se fait par empilage des paramètres. L'ordre d'empilage (paramètre le plus à gauche ou à droite dans la liste empilé en premier) dépend du compilateur.
La réservation de place pour les variables s'effectue par manipulation du pointeur de pile (décrémentation par exemple).
La libération de cet espace s'effectue de la même façon.

La partie de la pile qui contient les variables locales et éventuellement les variables temporaires s'appelle l'**environnement** de la fonction.
La partie de la pile utilisée lors de l'exécution de la fonction contient les paramètres, l'adresse de retour, les variables ; on appelle cette zone « **stack frame »** (cadre dans la pile).
Au niveau de l'exécution de la fonction, ce bloc est géré par le compilateur comme un tableau. Chaque paramètre ou variable locale est accessible par la fonction avec l'**adresse de base de l'environnement** et un indice correspondant au déplacement.
Ex : le paramètre1 est à l'adresse de base +8
Le paramètre2 est à l'adresse de base + 10

L'**adresse de base** peut être fixée par :

- le **pointeur de pile**
- ou un pointeur spécifique d'environnement (**frame pointer**) placé dans un registre non dédié (type registre d'index)
- ou par un **pointeur de base** de l'environnement (base pointer), placé dans un registre dédié (sur certains µPs / µCs)

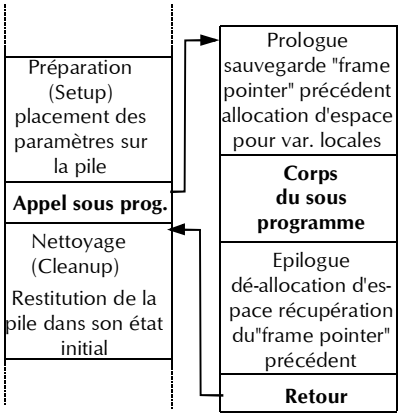
Le choix du type de pointeur utilisé dépend des ressources du µP / µC et ses différents modes d'adressage (par exemple avec un 68HC11, il est impossible d'utiliser le pointeur de pile comme base pour un adressage indexé et il est donc nécessaire d'utiliser un autre pointeur pour accéder aux variables et paramètres). Lorsqu'un pointeur spécifique est utilisé, le pointeur de l'environnement précédent est mémorisé à chaque appel de fonction.

La valeur de retour est placée à la place du 1er paramètre si sa taille le permet, sinon c'est un pointeur vers l'adresse de retour qui est mis à la place du 1er paramètre.

Il existe de grandes différences entre les compilateurs sur le « stack frame ». Voir la doc du compilateur utilisé.

EXEMPLE

Passage de paramètres avec utilisation d'un « frame pointer » différent du pointeur de pile :

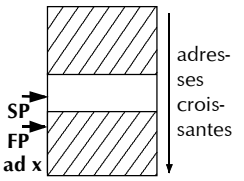


Les figures suivantes représentent l'allure de la pile lors des différentes étapes de l'exécution d'une fonction.

Pour simplifier, on n'a pas représenté avec exactitude la taille de chaque paramètre, variable, ...

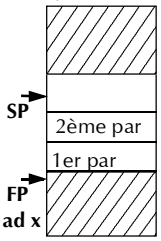
SP : Pointeur de pile (stack pointer)
FP : Pointeur de l'environnement (frame pointer)

1 État initial de la pile



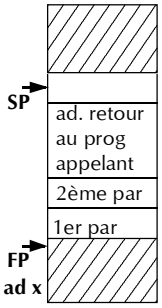
SP pointe sur un emplacement libre. FP pointe sur la base de l'environnement du programme appelant

2 Après empilage des paramètres

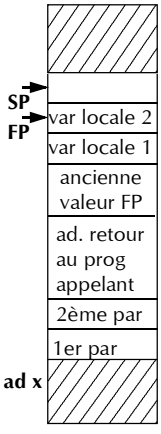


l'ordre d'empilage dépend du compilateur

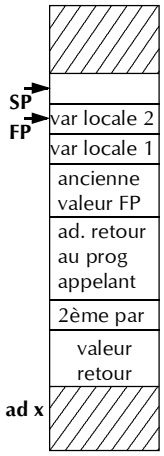
3 Après appel du sous programme



4 Après exécution du prologue de la fonction

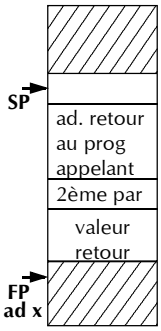


5 Après exécution du corps de la fonction

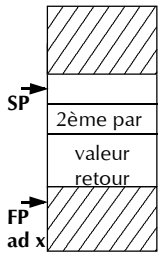


la valeur de retour remplace le 1er paramètre

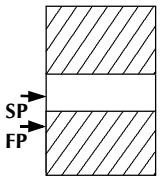
6 Après l'épilogue



7 Après retour au programme appelant



8 Après nettoyage



AVANTAGE

Toutes les fonctions sont ré-entrantes.

INCONVÉNIENT

Par rapport à l'autre mode de passage, taille du programme plus importante et traitement plus lent à cause du prologue, de l'épilogue et du nettoyage *(voir § précédent)*.
Les appels de fonctions imbriqués nécessitent une taille importante de pile.

22.4) PASSAGE DE PARAMÈTRE(S) UNIQUEMENT PAR DES REGISTRES PRÉVUS OU ZONE MÉMOIRE

Les paramètres et variables locales sont dans une zone de mémoire qui peut être partagée avec d'autres fonctions. L'éditeur de lien n'autorise le partage des zones mémoires que pour les fonctions qui le permettent (fonctions non imbriquées).

La préparation à l'appel se fait par mémorisation des paramètres dans une zone mémoire.
Seule l'adresse de retour au programme principal est sauvegardé dans la pile.
Les variables locales sont allouées de façon contiguë aux paramètres.
Lors de la phase de traduction en langage d'assemblage de la compilation, le compilateur génère un symbole (pointeur) pour accéder à la zone mémoire.

Ex : le compilateur Cosmic pour HC11 génère un symbole du même nom que la fonction suivi de .L (MaFonction → _MaFonction.L)

Le corps du sous programme utilise le pointeur pour accéder aux paramètres et aux variables locales.

Ex : l'adresse du 1er paramètre est _MaFonction.L, l'adresse du 2ème paramètre est _MaFonction.L + 2, ...

Inconvénient : les modules ne sont pas réentrants

Avantage : taille de la pile réduite dans certains cas (nombreuses fonctions imbriquées) et taille du programme en ROM diminuée pour les µC peu puissants qui ont besoin de plusieurs instructions pour maintenir le « stack frame ».
Le compilateur peut avoir plusieurs possibilités pour le choix du segment qui recevra les paramètres et les variables locales.

Ex : le compilateur Cosmic pour 68HC11 permet d'utiliser ce mode de passage de paramètre avec une option globale ou une option locale.

22.5) PASSAGE DE PARAMÈTRE(S) PAR REGISTRE ET PAR PILE

Le premier paramètre (s'il ne prend pas trop de place) est passé par registres internes déterminés du µP/µC et les autres par la pile.
Les variables locales, si elles ne prennent pas trop de place, sont placées dans des registres.
Pour une fonction récursive, la pile est systématiquement utilisée.
Lors de la compilation d'un gestionnaire d'interruption, les registres utilisés pour les paramètres et les variables locales sont sauvegardés si le gestionnaire appelle des fonctions.
Les deux mécanismes précédents permettent la ré-entrance des fonctions.

Avantages : rapidité d'exécution si un seul paramètre (assez fréquent dans des applications pour systèmes embarqués), possibilité d'utiliser des fonctions réentrantes.

Ex : le compilateur IAR pour 80c196 réserve 8 registres d'un octet pour le passage du 1er paramètre (pour permettre le passage jusqu'au type « double »). Il réserve de 8 à 32 octets pour les variables locales.

23) COMPILATION CONDITIONNELLE

La compilation conditionnelle permet de sélectionner ou d'exclure certaines parties du programme source. La compilation conditionnelle est utilisée principalement dans 2 cas :

- Même programme source pour différentes cibles ou différents environnements
- Ajout / suppression d'instructions pour le déboguage.

La compilation conditionnelle s'effectue avec le préprocesseur.

La compilation conditionnelle utilise :

- La valeur d'une expression
- La définition ou non d'un symbole

La 1ère méthode nécessite aussi la définition d'une constante symbolique (symbole).

23.1) SYMBOLE

Les symboles peuvent être définis par l'utilisateur dans le programme source avec la directive #define (*déjà mentionnée dans les § Données / Constantes symboliques et Macro*).

Un symbole peut avoir une valeur fixée explicitement ou aucune valeur explicite.

Valeur explicite :
#define <NomSymbole> <Valeur>

Aucune valeur explicite
#define <NomSymbole>
Dans ce cas, le préprocesseur, sait que le symbole a été défini.
Quand un symbole est uniquement défini sans valeur, il vaut 1 en interne pour le compilateur.

Un symbole peut aussi être défini dans un fichier en-tête livré avec le compilateur ou au niveau de l'environnement de développement (le « compilateur »), de façon automatique ou selon des options (case à cocher par exemple). Ceci permet la compilation différente d'une même programme source selon le compilateur.

23.2) COMPILATION LIÉE À LA DÉFINITION D'UN SYMBOLE

EXEMPLE

```
#define MISE_AU_POINT
/* définition par le programmeur. Il suffit de mettre cette ligne en remarque pour changer les blocs compilés */
#ifdef MISE_AU_POINT
...
... /* instructions à compiler si MISE_AU_POINT est défini */
#else
...
... /* instructions à compiler si MISE_AU_POINT n'est pas défini */
#endif
```

DIRECTIVES UTILISABLES

#ifdef #ifndef #else #endif
#if defined (idem #ifdef) #if !defined (idem #ifndef)

Les directives peuvent être imbriquées.

22.3) COMPILATION LIÉE À LA VALEUR D'UNE EXPRESSION

EXEMPLE

```
#if CAS == 1
...
... /* instructions compilées lorsque CAS =1 */
#endif
#if CAS == 2
...
... /* instructions compilées lorsque CAS = 2 */
#endif
```

Il suffit de définir CAS à 1 ou à 2 pour compiler une partie ou une autre.

DIRECTIVES UTILISABLES

#if #else #endif #elif

#elif permet d'imbriquer un if dans un else
#if CAS == 1
...
#elif CAS == 2
..
#endif

#if peut être suivi d'une condition/expression décrite au § Structures / Instructions de contrôle / Condition et expression

24) DONNÉES « DYNAMIQUES »

Les données « dynamiques » sont les données à durée de vie dynamique ou encore des données de classe de mémorisation dynamique (*Voir § Constitution d'un programme / Donnée / Durée d'existence*). Ces données sont regroupées dans des structures évoluées (Voir *ci-dessous § Liste liée*).
Les données dynamiques font appel à la gestion dynamique de la mémoire.

24.1) GESTION DYNAMIQUE DE LA MÉMOIRE

La gestion dynamique est utilisée quand le programmeur emploie des structures de données dont la taille est inconnue lors de la compilation. C'est par exemple le cas avec l'utilisation de tableaux de taille variable (*voir § Tableaux de taille variable*) et de listes chaînées (*voir § Liste liée*). Le nombre d'éléments du tableau ou de la liste peut dépendre d'une saisie clavier, d'une information reçue par la liaison série, ...
A chaque fois qu'il faut allouer un nouvel élément, on appelle une fonction d'allocation mémoire. Lorsqu'un élément n'est plus utile, une fonction permet de libérer la mémoire.
La mémoire est allouée sur le tas (heap).

DOMAINE D'EMPLOI DE DONNÉES DYNAMIQUES

La gestion dynamique est fréquemment employée pour une cible de type micro-ordinateur.
Pour un système embarqué, la gestion dynamique est peu utilisée (surtout avec des systèmes de faibles de ressources). Le besoin ne se fait pas sentir d'une telle gestion (peu d'interactivité) ; de plus les fonctions utilisées nécessitent beaucoup de mémoire. Par exemple la fonction malloc() présentée ci-dessous nécessite plus de 200 octets en mémoire programme pour un 68HC11 (µC 8 bits).
Certains compilateurs pour petits µC (exemple PIC de Microchip) ne comprennent pas de fonctions pour la gestion dynamique.

La gestion dynamique utilise des fonctions de la bibliothèque standard.

ALLOCATION MÉMOIRE

Plusieurs fonctions permettent d'allouer dynamiquement de la mémoire. Seule **malloc()** est présentée ici.

La fonction malloc() permet d'allouer un bloc mémoire d'une taille égale à la valeur transmise en argument. Elle renvoie un pointeur générique sur ce bloc mémoire ; ce pointeur doit être affecté à une variable de type pointeur déjà déclarée. Il peut explicitement être mis au type (cast) du pointeur affecté (conseillé) ou la conversion peut être implicitement réalisée par le compilateur.

Syntaxe utilisation
<PtrDonnee> = (<TypeDonnee>*) malloc(<NbOctets>); /* ici conversion explicite au type <TypeDonnee>* du pointeur retourné */

Pour la taille mémoire à réserver, on utilise fréquemment l'opérateur sizeof.
PtrNouvelElement = (Element) malloc (sizeof(Element))*

DÉ-ALLOCATION MÉMOIRE

Une fois qu'un élément n'est plus utile, il faut libérer la mémoire.
Dans le cas d'un programme pour ordinateur, ceci peut s'effectuer à la suite d'une commande interactive tapée au claver ou la souris ou à la sortie du programme.
En fin de programme, la libération de la mémoire utilisée pour les données statiques et automatiques est automatique mais pas pour les données dynamiques. Cette libération est à la charge du programmeur.

La fonction **free()** permet de libérer l'espace mémoire alloué et de le rendre disponible pour de nouvelles données dynamiques.

syntaxe de l'utilisation
free (<NomPtr>);
/* <NomPtr> est le nom du pointeur qui désigne l'élément à supprimer de la mémoire. Il doit être du même type que celui utilisé lors de l'allocation */

Pour libérer complètement la mémoire utilisée par une liste liée, il faut libérer successivement la mémoire pour tous les éléments.

24.2) TABLEAUX DE TAILLE VARIABLE

TABLEAU SIMPLE

La création et l'utilisation d'un tableau simple résulte de ce qui a été vu au § précédent Allocation mémoire et au § Pointeurs / Pointeur sur donnée / Arithmétique des pointeurs / Pointeur avec indice.

La création d'un tableau s'effectue avec :
ptTab = (<type>*) malloc (Nb*sizeof(<type>)) ;

L'utilisation s'effectue avec ptTab[Indice]. 0 ≤ Indice < Nb.

TABLEAU À 2 DIMENSIONS

Plusieurs méthodes sont possibles pour la création.

MÉTHODE 1

Même méthode que pour un tableau simple. Le tableau à 2 dimensions est considéré comme un tableau à 1 dimension. L'utilisateur doit écrire le code qui permet de passer de 2 indices IndiceLi et IndiceCol à un seul.

Création par :
ptTab=(<type>*) malloc(NbLi * NbCol *sizeof (<type>));
/* ptTab doit avoir été déclaré comme un pointeur sur <type> */

Accès avec :
ptTab[IndiceLi*NbCol+IndiceCol]
Seul un indice est autorisé avec un pointeur.

MÉTHODE 2

On crée un tableau de pointeurs pour les lignes. Ce tableau est lui même repéré par un pointeur qui est donc un pointeur de pointeurs ou plutôt d'un tableau de pointeurs.

Création :
ptTab_ptLigne = (<type>**) malloc (NbLi*sizeof (<type>*));
/* ptTab_ptLigne doit avoir été déclaré comme un pointeur de pointeur sur <type>
Pour chaque ligne, il faut allouer de la mémoire, par exemple en utilisant une boucle for */
for (indice = 0 ; indice < NbCol, indice++)
{ ptTab_ptLigne[indice] = (<type>*) malloc (NbCol*sizeof(<type>));
}

Accès :
Il faut utiliser un pointeur intermédiaire
ptLigneCour=ptTab_ptLigne[IndiceLi];
/* ptLigneCour doit avoir été déclaré comme un pointeur sur <type> */
Chaque colonne de la ligne courante peut être accédé avec ptLigneCour[IndiceCol]

TABLEAU DE STRUCTURES

On peut procéder comme précédemment pour un tableau à 2 dimensions, avec la méthode 2.
Il faut d'abord allouer de la mémoire pour un tableau de pointeurs sur les structures puis allouer de la mémoire pour chacune des structures, comme précédemment.
L'accès à un élément de la structure s'effectue avec ptTab_ptStruct[indice]->NomChamp.

24.3) LISTE LIÉE

Une **liste liée** ou **liste chaînée** ou plus simplement **liste** est composée d'éléments reliés entre eux par des **pointeurs** (*Voir plus avant § Pointeurs*). Chaque élément est une structure qui est elle-même composée de champs qui contiennent des données et d'un ou deux champs qui contiennent un pointeur sur l'élément suivant et un pointeur sur l'élément précédent.

DÉFINITION D'UNE ÉLÉMENT D'UNE LISTE LIÉE

struct <NomModèle>
{ // Champs pour données de l'application
struct <NomModèle>* <PtElementSuivant> ;
} ;
PtElementSuivant est un pointeur sur l'élément (la structure) suivant de la liste.

CRÉATION ET UTILISATION D'UNE ÉLÉMENT D'UNE LISTE LIÉE

La création d'une variable de ce type s'effectue de façon dynamique. *Voir § Gestion dynamique de la mémoire.*
Les éléments d'une liste chaînée s'utilisent avec des pointeurs. L'accès à un champ se fait avec l'opérateur -> comme mentionné dans le § Utilisation d'une structure
.
Ex :
Pour ajouter un élément à une liste chaînée (*voir § Liste liée*), il faut :

- 1) déclarer un pointeur PtNouvelElement sur un type modèle de structure déjà défini ModeleStruct
- 2) créer le nouvel élément (une variable de type ModeleStruct) avec une fonction d'allocation dynamique et affecter le pointeur renvoyé au pointeur PtNouvelElement en effectuant un changement explicite de type
- 3) remplir les champs du nouvel élément avec les données de l'application. L'accès à un champ s'effectue avec PtNouvelElement -> Champ
- 4) affecter, dans l'élément précédent de la liste, le pointeur qui désigne l'élément suivant (contenu dans un champ de la structure) au pointeur PtNouvelElement désignant le nouvel élément créé

ModeleStruct* PtNouvelElement ; // 1)
PtNouvelElement = (struct ModeleStruct*) malloc (sizeof(struct ModeleStruct) ; // 2)
...
Le 4) peut s'effectuer de différentes façons selon que liste soit repérée ou non par un descripteur de liste.

DESCRIPTEUR DE LISTE

Un élément d'une liste chaînée peut être accessible depuis un descripteur de liste (structure) formé de pointeurs sur le premier élément, le dernier élément et l'élément en cours d'accès.
Pour modifier les pointeurs du descripteur de liste, il faut manipuler des pointeurs de pointeurs. Voir § Pointeur de pointeur.

A compléter

25) CONSEILS D'ÉCRITURE D'UN PROGRAMME SOURCE

Il n'y a pas de norme concernant l'écriture d'un programme. Les conseils donnés ici sont suivis par beaucoup de développeurs.

25.1) DOCUMENTATION D'UN PROGRAMME

Un programme doit absolument être documenté. Ceci est fait par une présentation, des commentaires et un choix judicieux des identificateurs.

PRÉSENTATION D'UN PROGRAMME ET DES FONCTIONS

Chaque fichier source doit être présenté pour être exploitable par la suite.

```
/*************************************************/
/* Projet: ....
/* Auteur: ...
/* Nom du fichier: ...
/* Date de 1ère création: ....
/* Révisions: ...
/* Chaîne de compilation: ...
/*************************************************/

/*************************************************/
/* Détail du module
/* ....
/*************************************************/
```

Chaque fonction doit être présentée et décrite.

```
/*************************************************/
/* Nom fonction : main
/* Niveau de pile : 0
/* Appelée par : -
/* Appelle : Init(), TestMode(), ...
/*************************************************/

/*************************************************/
/* Nom fonction : GestionCdeEtMsgs
/* Appelée par : main
/* Niveau de pile : 1
/* Appelle : Transcodage (niveau de pile 3), ...
/* Paramètres d'entrée : ...
/* Valeur retournée : ...
/* Description : ...
/*************************************************/
```

CHOIX DES IDENTIFICATEURS

Ils doivent correspondre à la donnée ou la fonction représentée.
AireRectangle = Cote1 * Cote2
est meilleur que z = x * y.
Pour que l'identificateur ne soit pas trop long, on peut utiliser des abréviations compréhensibles. Ex VALID_INTER_TIMER1.

COMMENTAIRES

Les commentaires doivent décrire ce que fait le programme, comment il le fait, ce que représentent les paramètres et les variables, les restrictions d'usage, le erreurs connues. Ils doivent être courts et informatifs.
Les commentaires doivent être séparés en 2 parties :

- un commentaire pour chaque ensemble d'instructions, fonction, etc.
- + des commentaires pour chacune des lignes qui le nécessitent.

Exemple :

...

```
/* Tests des demandes d'interruptions dans l'ordre des priorités. Priorité la plus élevée timer2 pour l'interruption périodique */

if (TMR2IF == 1)
/* Traitement interruption périodique du Timer2 */
{
    PORTC = 0;  /* extinction */

    if (TypeCde == FREQ_VAR)
      {
      if (Matrice == DROITE)
        {

        /* Une impulsion MEM */
        RD7 = 1;
        RD7 = 0;

        /* Fermeture d'un des interrupteurs colonnes. Solution sans table = pas de sous programme (spécificité du PIC), plus de rapidité */
        switch (Col)
          {
          case 0:    PORTC=0b00000001;
                     Col +=1; break;
          ...
          }
      ...
```

25.2) ÉCRITURE DES IDENTIFICATEURS

Il n'y a pas de convention universellement adoptée pour l'écriture des identificateurs. Les règles suivantes sont très répandues :

- les identificateurs pour les variables et les fonctions sont formés de la juxtaposition de plusieurs mots écrits en minuscules avec le début de chaque mot en majuscule. Ex : PrixTotal
- les identificateurs pour les constantes sont en majuscules avec _ entre chaque mot. Ex : PRIX_ARTICLE
- les noms des types personnels commencent par t ou T. Ex : TEntier
- les noms des pointeurs commencent par p ou pt ou Pt ou Ptr
- les noms des pointeurs de fonctions commencent par ptf

25.3) INDENTATIONS ET POSITION DES { }

Les accolades peuvent être placées pour mettre en évidence les structures.

Plusieurs styles sont possibles.

Le suivant est le plus clair pour un programmeur peu expérimenté, mais il consomme beaucoup d'espace.

```
void main(void)
{
    ....
    for( ; ;)
    {
        ...
        if(...)
            {
                ....
            }
            else
            {
                ...
```

```
            }
        ...
    }
}
```

Le style ci-dessous est plus compact

```
void main(void){
    ....
    for( ; ;)  {
        ...
        if(...) {
                ....
        }
        else {
                ...
        }
        ...
    }
}
```

25.4) PROGRAMME SUR PLUSIEURS FICHIERS

L'approche modulaire permet de placer la totalité du programme sur plusieurs fichiers.

Chaque module regroupe en général des fonctions qui un rapport entre elles. Un module correspond à 2 fichiers :

- un fichier .h qui contient les définitions et/ou déclarations des fonctions et des variables globales
- un fichier .c qui contient les définitions des fonctions

Le fichier .h est destiné à être inclus, avec la directive #include, non seulement dans le fichier .c correspondant mais aussi dans les autres fichiers .c du projet.

Il faut utiliser les directives de compilation conditionnelle pour éviter que des emplacements mémoires ne soient réservées plusieurs fois, à partir de fichiers source différents, pour les mêmes variables globales

Chaque fichier source .c doit contenir la définition d'un symbole utilisé pour la compilation conditionnelle du fichier .h inclus.

EXEMPLE

Fichier Source1.c

```
#define SOURCE1

...
#include Source1.h
#include Source2.h
...
```

Fichier Source1.h

```
...
/************************************************/
/* Définitions globales des modèles de structu-
/* res, unions, etc.
/************************************************/
...

#ifdef SOURCE1
/************************************************/
/* Définition des données globales utilisées
/*Prototypes des fonctions définies dans le
/* module "HOME"
/************************************************/
    unsigned char VarGlob1,
                  VarGlob2 ;
    ....

#else
/************************************************/
/* Déclarations externes pour les autres mo
/* dules
/************************************************/
      extern unsigned char VarGlob1,
                           VarGlob2 ;
    ...

#endif
```

Lorsque Source1.h sera inclus dans un autre fichier que Source1.c, la partie après ifdef ne sera pas compilée.