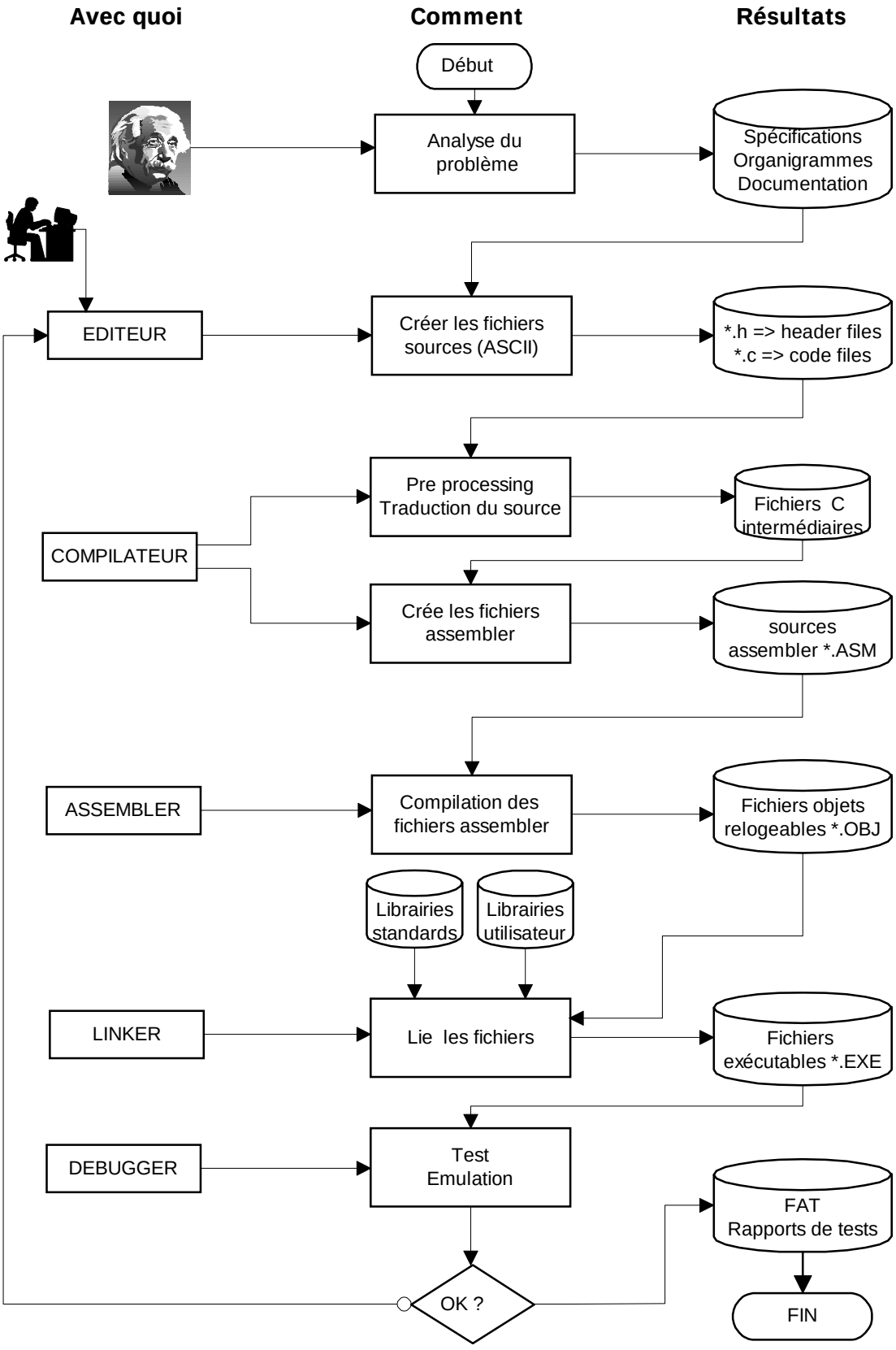
La pratique du langage C

TABLE DES MATIERES

1 La création d'un programme



1.1 L'analyse une étape indispensable pour un projet important.

C'est la phase la plus importante lors de la réalisation d'un programme informatique. Un projet important qui ne commence pas par une analyse sérieuse est destiné à de grandes difficultés ou à l'échec. Les coûts de développements de logiciels sont très élevés, le temps investi dans cette analyse permettra de diminuer le temps de développement global.

1.2 L'éditeur

Crée les fichiers sources *.c et les fichiers header *.h

1.3 Le précompilateur

Il transforme le code source C en code source C pur en exécutant les directives du précompilateur.

1.4 Le compilateur

Il transforme le code C pur en code objet relogeable *.obj

1.5 Le linker (Editeur de lien)

Il relie les fichiers objets et les librairies utilisateurs et systèmes pour créer le fichier exécutable *.exe

1.6 Le debugger

Permet de tester le programme.

1.7 Les commentaires

Il existe deux méthodes pour écrire des commentaires en C. Le commentaire délimité par /* indiquant le début et */ indiquant la fin du commentaire qui peut être sur plusieurs lignes.

```
/* J'affirme que cette ligne est un commentaire pouvant se répartir sur
plusieurs lignes.*/
```

Il est aussi possible d'avoir un commentaire se terminant par la fin de ligne. (attention, ne fonctionne par sur tous les compilateurs).

```
    // Ce commentaire se termine à la fin de la ligne.
    // Il ne fonctionne pas sur tous les compilateurs (issu du C++)
```

1.7.1 La pertinence des commentaires

Documenter un programme est une nécessité. Cette documentation doit permettre de comprendre pour pouvoir exécuter des modifications et des corrections de bugs éventuels.

1.7.2 En-tête du programme.

Le programme doit commencer par un en-tête, il doit comprendre au minimum le nom de l'entreprise, l'auteur du programme, le nom du fichier, la description du projet, la date de création, les versions avec une description des modifications ainsi que la date.

```
/*****************************************************************
Nom de fichier :      Monfichier
Auteur :              XXX
Description :         Programme de gestion de cartes à puces
Version :             1.0  10 jan 00 JMC Version de base.
Version :             1.1  20 jan 00 JMC
                     Correction du bug 123 (voir rapport x)
*****************************************************************/
```

1.8 Premier programme

```
/*****************************************************************
Fichier:         PrPrg.cpp
Auteur:          JMC
Description:     Premier exemple de programme
Version:         1.0
Modifications:   pas de modification
*****************************************************************/

#include <stdio.h>    // accès à la librairie standard input/output

/*****************************************************************
                      programme principal
*****************************************************************/
void main (void)                // fonction main obligatoire
{                               // début de bloc
   unsigned char MonChar;       // Variable locale de type caractère

   printf("Voulez vous afficher Hello ? [O/N] \n");

   MonChar = getchar();         // lecture d'un caractère du clavier

   if(MonChar == 'O')           // test si le caractère lu est O
      printf("Hello");          // Si O => Affichage à écran
}                               // fin de bloc
/*****************************************************************
                           end
*****************************************************************/
```

1.9 Règles de présentation

L'écriture en C est très compacte. Il faut tout faire pour rendre le programme le plus lisible possible. Pour faciliter la compréhension des logiciels, beaucoup d'entreprises éditent des règles de codage interne. Voici quelques règles de présentation des programmes.

- Avoir une présentation cohérente tout au long du projet
- Aligner dans la verticale les accolades de début et fin de bloc
- Indenter les lignes à l'intérieur d'un bloc
- Ne pas mélanger sur une même ligne un test et une instruction.
- Ecrire une seule instruction par ligne.

2 Les variables

Les variables permettent de stocker les données en RAM. Par défaut les variables sont signées. Pour gagner un bit supplémentaire on peut déclarer des variables non signées en ajoutant le préfixe **unsigned.**
Tous les types de variables permettent d'effectuer des calculs ou des opérations logiques.

Attention : Les tailles des variables peuvent changer en fonction de l'implémentation
Il faut vérifier la taille dans la documentation du compilateur.

2.1 Les types de variables

2.1.1 Le type char

Elle peut contenir au minimum 1 caractère du système.

```
char peut être utilisé pour des opérations arithmétiques.
Si les caractères sont en ASCII => char est un byte (octet).

char          ma_data;          // ou signed char)
unsigned char ma_data;          //
```

2.1.2 Le type int

```
Elle correspond à la taille d'un mot machine, peut être de 8, 16. 32, 64
bits etc.

int           ma_data;          // ou signed int
unsigned int  ma_data;          //
```

2.1.3 Le type short

```
Il est plus petit ou égal à un integer     short <= int.
```

```
short          ma_data;         // ou short int ou signed short int
unsigned short ma_data;         // ou unsigned short int
```

2.1.4 Le type long.

```
il est plus grand ou égal à integer   long >= int.
```

```
long          ma_data;          // ou long int ou signed short int
unsigned long ma_data;          // ou unsigned long int
```

2.2 Nombres de bits :

Le nombre de bits des différents types de variables varie avec l'implémentation. Exemple :

Type de variable	Borland C	GNU GCC
char	8	8
unsigned char	8	8
short	16	16
unsigned short	16	16
int	16	32
unsigned int	16	32
long	32	32
unsigned long	32	32

2.3 Le type float (nombre réel)

C'est la notation scientifique exprimée en mantisse et exposant. C'est un nombre réel

Il existe trois variantes avec 3 précisions possibles.

```
float       ma_variable;              // la plus faible précision
double      ma_variable;              // la précision moyenne
long double ma_variable;              // la plus grande précision
```

	nbr bits	signe	mantisse	exposant	max	min
float	32	1	23	8	$3,4\ 10^{38}$	$3,4\ 10^{-38}$
double	64	1	52	11	$1,7\ 10^{308}$	$2,3\ 10^{-308}$
long double	80	1			$1,1\ 10^{4932}$	$3,4\ 10^{-4932}$

2.4 Déclaration de variables avec initialisation ;

Les variables peuvent être initialisée lors de la déclaration :

Exemple :

```
unsigned char  ma_data = 'A';       // initialisation en ASCII
unsigned char  ma_data = 0xff;      // initialisation en hexadécimal
unsigned short ma_data = 65535;     // initialisation en décimal
int            ma_data = -32767;    // initialisation en décimal
```

3 Les constantes :

Les déclarations des constantes sont identiques à celles des variables mais sont précédées du préfix **const** et on leur attribue une valeur. Exemples :

```
const unsigned char  ma_data = 'A';    // initialisation en ASCII
const unsigned char  ma_data = 0xff;   // init. en hexadécimal
const unsigned short ma_data = 65535;  // initialisation en décimal
const int            ma_data = -32767; // initialisation en décimal
```

La différence entre une variable initialisée et une constante est petite.
Sur un PC les deux sont en DRAM, mais le compilateur empêche l'écriture d'une constantes.
Pour un logiciel embarqué, les variables seront en RAM, les constantes seront en mémoire flash, en EPROM ou en ROM.

4 Les chaînes de caractères (strings)

Les chaînes de caractères (ASCII ou autres) sont entourées de deux caractères ″.

```
″ceci est une chaîne de caractères″
```

4.1 Les caractères non imprimables :

Il peut être nécessaire d'incorporer des caractères non imprimables dans une chaîne de caractères. Ces caractères non imprimables font partie des caractères de contrôle. ils permettent principalement des manipulations du curseur.

\n	new line	nouvelle ligne
\t	horizontal tabulation	tabulation horizontale
\v	vertical tabulation	tabulation verticale
\b	back space	back space
\r	carriage return	retour du chariot
\f	form feed	saut de page
\a	audible alert	alarme sonore
\000	caractère octal 0	nul ASCII exprimé en octal
\x00	caractère hexadécimal	nul ASCII exprimé en hexa

4.2 La représentation des caractères réservés à la syntaxe des strings.

Un certain nombre de caractères sont utilisés pour la syntaxe des chaînes de caractères, pour cette raison ils ne peuvent pas être utilisés telle quel dans des chaînes de caractères, ils sont remplacés par une séquence de caractères.

\'	'
\\	\
\″	″
\?	?

4.3 Utilisation des caractères de contrôle

```
″c'est la ligne 1 \n c'est la ligne 2 \n c'est la ligne 3″
```

Le string s'affiche sur 3 lignes.

4.4 Le string de 1 caractère et le caractère.

Il ne faut pas confondre :

```
'A'     // est un caractère ASCII A
"A"     // représente deux caractères A ASCII et la fin de string NULL ASCII
```

5 Les entrées/sorties formatées

Elles permettent les entrées/sortie du clavier et écran, en effectuant une conversion de type.

5.1 La fonction de librairie <stdio> printf()

Cette fonction fait partie de la libraire stdio (standard input output).Elle permet l'affichage à l'écran ou sur l'imprimante. Les données numériques converties puis affichées dans un format spécifié.
Exemple :

```
printf("Hello boy")          // affiche Hello boy
```

5.1.1 Le type de l'affichage d'une valeur numérique

Le type de l'affichage d'un caractère numérique est spécifié par le tableau suivant :

%d	affichage d'un entier décimal signé
%o	affichage en octal
%x	affichage en hexadécimal
%u	affichage en non signé
%i	affichage d'un entier signé
%c	affichage d'un seul caractère
%s	affichage d'un string
%e	affichage en floating point avec exposant
%f	affichage en floating point sans exposant
%g	affichage comme f ou e suivant la valeur

5.1.2 Action sur le format de l'affichage (disposition)

L'affichage dépend du type de la variable et du nombre de caractères désirés.
Tous les caractères y compris le . sont comptés dans le nombre de caractères total.

```
printf("%3d",n);   // Le 3 indique que l'on désire afficher
                   // au minimum 3 caractères.
```

Exemples : Valeur entière décimale avec 3 caractères au minimum.

```
printf("%3d",n);   //  valeur n          affichage min 3 caractères
                   //   n = 20              ^20
                   //   n = 3               ^^3
                   //   n = 2358            2358
                   //   n = -5200           -5200
```

Exemples : Flotting sans exposant avec longueur par défaut.

```
printf("%f",n);    // flotting pt => par défaut 6 chiffres après le .
                   //        valeur n              affichage
                   //        1.2345              1.234500
                   //        12.3456789          12.345678
```

Exemples : Flotting sans exposant avec au minimum 10 caractères.

```
printf("%10f",n);  // minimum 10 caractères avec le point
                   // et 6 chiffres après le point
                   // valeur de n           affichage
                   //   1.2345           ^^1.234500
                   //   12.345679        ^12.345679
                   //   1.2345E5         123450.000000
```

Exemples : Flotting avec exposant et longueur par défaut.

```
printf("%e",n);            // affichage avec exposant,
                           // 6 chiffres après le point
                           //  valeur de n         affichage
```

```
                    //   1.2345          1.234500e+000
                    //   123.45          1.234500e+002
                    //   123.456789e8    1.234567e+010
                    //   -123.456789e8   -1.234568e+010
```

5.1.3 Action sur le format de l'affichage (précision)

Exemples : Flotting sans exposant, la longueur minimum et nombre de caractère après le point sont définis.

```
printf("%10.3f",n);        // float 10 caractères point compris
                           // 3 chiffres après le point
                           // valeur de n       affichage
                           //   1.2345         ^^^^^1.234
                           //   1.2345e3       ^^1234.500
                           //   1.2345e7       12345000.000
```

Exemples : Flotting avec exposant, la longueur minimum et nombre de caractère après le point sont définis.

```
printf("%12.4e",n);        // float exponentiel 12 caractères
                           // 4 chiffres après le point
                           // valeur de n          affichage
                           //   1.2345           ^1.2345e+000
                           //   123.456789e8     ^1.2345e+010
```

5.2 La fonction de librairie <stdio>scanf()

Cette fonction fait partie de la libraire stdio (standard input output)
Elle permet la saisie de données depuis le clavier. Exemple :

```
scanf("%d",&i)         // La variable i est la destination
                       // %d indique une variable entière signée
```

On passe à cette fonction le type de donnée et l'adresse de la variable de destination.

5.2.1 Acquisition en une fois de plusieurs variables.

```
scanf("%d %d",&i,&j)   // On saisit plusieurs variables en une fois.
```

L'utilisateur répond en séparant les variables a acquérir par des caractères espace ou par des caractères carriage return.
La fin de l'acquisition étant toujours signalée par un carriage return.

5.2.2 Imposition du format maximum

```
scanf("%3d",&i)      // On impose un format maximum pour la variable
```

Dans ce cas l'acquisition s'interrompt lorsque le nombre de maximum de caractères est atteint.

5.2.3 Caractère invalide dans un format :

La réception d'un caractère invalide termine l'acquisition de la donnée.
Par exemple si on entre une variable alphabétique dans un entier.

5.2.4 La valeur de retour de scanf()

```
int compte ;
compte = scanf("%3d",&i)     // compte = nombre de champs mis à jour
```

Le retour de la fonction scanf indique le nombre de champs correctement mis à jour.

```
compte = scanf(”%3d %3d %3d ”,&i,&j,&k); //compte = nbr de champs
                                          // saisis correctement
```

Dans ce cas valeur de compte sera de 3 seulement si les 3 champs ont été correctement remplis. Il est donc possible de contrôler si tous les champs à acquérir ont été correctement remplis.

6 Quelques fonctions de librairie

6.1 La fonction getchar() <stdio>

Cette fonction permet d'acquérir un seul caractère ASCII au clavier. Elle ne requière aucun paramètre. Le caractère de retour est le code de la touche qui doit être mis dans une variable de type char. Exemple :

```
char   c;                  //

c = getchar();             // keyboard input
```

6.2 La fonction de librairie putchar() <stdio>

Cette fonction permet d'afficher un seul caractère ASCII sur l'écran (imprimante). Le paramètre d'entrée est le nom de la variable contenant le caractère à afficher. Exemple :

```
char c ;

c = ‘A’;                // A ASCII dans c
putchar(c);             // A ASCII à l’écran
```

6.3 La fonction de librairie strlen() <string.h>

Cette fonction string length retourne la longueur d'une chaîne de caractère.
On peut l'appeler de 2 manières, en lui passant le string en paramètre, ou une solution plus élégante en passant l'adresse du string en paramètre.

Exemple avec passage du string sur le stack (peu recommandable).

```
int lng;                  // déclaration, stock la longueur du string

lng = strlen(”hello”);    // ici tout le string est mis sur le stack
```

Autre solution :

```
int lng;                    // stock la longueur du string
char *MonString = ”hello”;  // pointeur sur le string hello

lng = strlen(MonString);    // ici seule l’adresse du string est mise
                            // sur le stack.
```

7 Les opérateurs les plus usuels.

7.1 L'affectation

=	Affectation	MaData = 12 ;

7.2 Les opérateurs arythmétiques.

+	L’addition	MaData = MaData+12
-	La soustraction	MaData = 20-5 ;
*	La multiplication	MaData = MaData*2
/	La division	MaData = 50/4
%	Le modulo, reste de la division entière	MaData = 50%5

7.3 Les opérateurs de comparaison (relationnels)

==	identique	MaData == TempData ;
!=	différent	MaData != TempData ;
>	Plus grand que	MaData > TempData ;
>=	Plus grand ou égal que	MaData >= TempData ;
<	Plus petit que	MaData < TempData ;
<=	Plus petit ou égal que	MaData <= TempData ;

8 L'instruction if

Le mot then n'existe pas dans le langage C. Il est remplacé par des parenthèses qui entourent le ou les conditions.

```
if (MaData == ProvData)              // la condition
{                                    // le début de bloc
   MaData = MaData + 1;              // les instructions
   MaData2 = MaData + 2;             // les instructions
}                                    // la fin du bloc if
```

Remarque les accolades de début et fin peuvent être omises si le if conditionne une seule instruction.

8.1 L'instruction if else

La condition est entourée par des parenthèses.

```
if (MaData >= ProvData)              // la condition
{                                    // le début de bloc if
   MaData = MaData + 1;              // les instructions
   MaData2 = MaData + 2;             //
}                                    // la fin du bloc if
else
{                                    // le début de bloc else
   MaData = MaData + 3;              // les instructions
   MaData2 = MaData + 4;             //
}                                    // la fin du bloc else
```

Remarques

Les accolades de début et fin peuvent être omises si on a une seule instruction par bloc.
Le else se rapporte au if précédent à condition qu'il soit dans le même bloc
L'alignement du if et du else est indispensable pour faciliter la compréhension.
L'indentation à l'intérieur d'un bloc est indispensable pour clarifier le programme.
Une indentation de 3 caractères permet d'éviter de se trouver trop vite à droite de la page.

9 Les opérateurs logiques

&&	and logique pour des conditions	(i == TempData) && (j == TempData2);
¦¦	ou logique pour des conditions	(i == TempData) ¦¦ (j == TempData2);
!	not inversion logique, complément à 1	MaData = (!TempData);

9.1.1 Deux conditions doivent être réalisées

```
if ((MaData == ProvData) && (MaData2 == ProvData2)) // test avec and

{                                       // le début de bloc
   MaData = MaData + 1;                 // les instructions
   MaData2 = MaData + 2;
}                                       // la fin du bloc if
```

9.1.2 La conditions 1 ou 2 doit être réalisée

```
if ((MaData == ProvData) ¦¦ (MaData2 == ProvData2)) // test avec or
{                                       // le début de bloc
   MaData = MaData + 1;                 // les instructions
   MaData2 = MaData + 2;
}                                       // la fin du bloc if
```

9.1.3 Test avec la condition vraie => différente de 0.

```
if (MaData)                             // si MaData est != 0
{                                       // le début de bloc
   MaData = MaData + 1;                 // les instructions
   MaData2 = MaData + 2;
}                                       // la fin du bloc if
```

9.1.4 Test avec condition non vraie => égale à 0.

```
if (!MaData)                            // Si MaData == 0
{                                       // le début de bloc
   MaData = MaData + 1;                 // les instructions
   MaData2 = MaData + 2;
}                                       // la fin du bloc if
```

9.1.5 Remarque sur les parenthèses.

A la place de

```
if ((MaData == ProvData) && (MaData2 == ProvData2))
```

On peut écrire

```
if (MaData == ProvData && MaData2 == ProvData2)
```

Les parenthèses clarifient le programme, elles sont recommandées.

10 L'incrémentation et la décrémentation

Pour ajouter ou soustraire 1 à une variable on peut écrire :

```
i = i + 1 ;                    // ajoute 1 à la variable i
i = i – 1 ;                    // soustrait 1 à la variable i
```

On peut simplifier ces écritures en les remplaçant par :

```
i++ ;                          // ajoute 1 à la variable i
i-- ;                          // soustrait 1 à la variable i
```

10.1 La pré incrémentation et la pré décrémentation

L'incrémentation (décrémentation) est faite **avant** l'affectation

```
j = ++i ;                      // i est incrémenté puis affecté à j
j = --i ;                      // i est décrémenté puis affecté à j
```

10.2 La post incrémentation et la post décrémentation

L'incrémentation (décrémentation) est faite **après** l'affectation

```
j = i++ ;                      // i est affecté à j puis incrémenté
j = i-- ;                      // i est affecté à j puis décrémenté.
```

10.3 Exemple avec pré-incrémentation

On va affecter 1 à la variable n, et i = 6

```
i = 5 ;                        // initialisation de i
n = ++i -5 ;                   // n = 1 car i est incrémenté avant !
```

10.4 Exemple avec post-incrémentation

On va affecter 0 à la variable n, et i = 6

```
i = 5 ;                        // initialisation de i
n = i++ -5 ;                   // n = 0 car i est incrémenté après !
```

11 Opération sur une variable et affectation

On modifie la variable i.

```
i = i + 5 ;                    // ajoute 5 à la variable i
i = i – 5 ;                    // soustrait 5 à la variable i
```

On peut écrire la même opération sous une forme compacte.

```
i += 5 ;                       // ajoute 5 à la variable i
i -= 5 ;                       // soustrait 5 à la variable i
```

11.1 Les opérateurs d'opération/affectation

+=	L'addition	MaData += 12
-=	La soustraction	MaData -= 20-5 ;
*=	La multiplication	MaData *= 2
/=	La division	MaData /= 2
%=	Le modulo, reste de la division entière	MaData %= 4
&=	Opération logique AND	MaData &= 0xf0
¦=	Opération logique OR	MaData ¦= 0x01
^=	Ou exclusif	MaData ^= 0X01

12 Complément sur l'instruction if

12.1 Le if else simplifié

Lorsque une instruction if else peut être écrite en une ligne on peut simplifier son écriture de la manière suivante.

```
if (i>5)
   i--;
else
   i++ ;
```

Ces instructions peuvent être remplacées par:

```
(i>5)? i-- : i++;
```

Autre exemple.

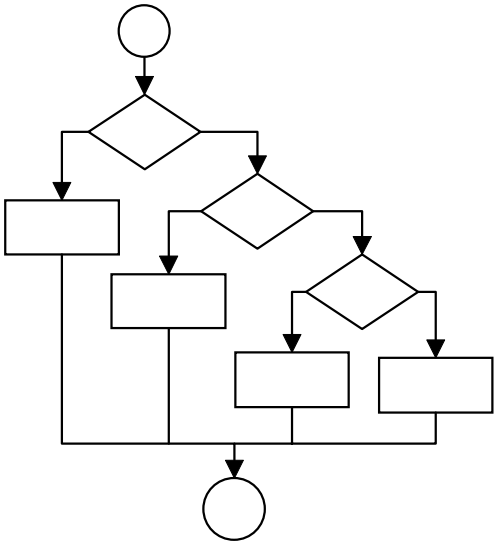
```
if (a>b)
   x=a;
else
   x=b ;
```

Ces instructions peuvent être remplacées par:

```
x = (a>b)? a:b;
```

12.2 Les if imbriqués

Il est possible d'imbriquer des if dans des if, en voici un exemple.



```
if (MaData >= ProvData)
{                                       // le début de bloc if
   MaData = MaData + 1;
   MaData2 = MaData + 2;
}                                       // la fin du bloc if
else
{                                       // le début de bloc else-1
   if (MaData == ProvData2)
   {                                    // le début de bloc if
      MaData = MaData + 2;
      MaData2 = MaData + 3;
   }                                    // la fin du bloc if
   else
   {                                    // le début de bloc else-2
      if (MaData <= ProvData3)
      {                                 // le début de bloc if
         MaData = MaData + 4;
         MaData2 = MaData + 5;
      }                                 // la fin du bloc if
      else
      {                                 // le début de bloc else-3
         MaData = MaData + 6;
         MaData2 = MaData + 7;
      }                                 // la fin du bloc else-3
   }                                    // la fin du bloc else-2
}                                       // la fin du bloc else-1
```

Remarques :

Pour conserver la maîtrise du logiciel la présentation est importante.
Cette présentation permet d'éviter un décalage à droite trop rapide.
Les parenthèses de début et fin de else ne sont pas obligatoire mais permettent une meilleure maîtrise du logiciel.

13 Les contrôles de flux

13.1 La boucle while (tant que)

La boucle while est exécutée tant que la condition entre parenthèse est réalisée.
Le test est fait en début de boucle, le traitement peut ne jamais être exécuté si la condition est déjà réalisée.

```
while(MaData <= ProvData)        // condition entre () est testée
                                 // en début de bloc
{                                // le début de bloc du while
   MaData = MaData + 1;
   MaData2 = MaData + 2;
}                                // la fin du bloc du while
```

13.1.1 La boucle while infinie.

Il peut être nécessaire de disposer d'une boucle infinie. Cette boucle infinie peut être réalisée de la manière suivante :

```
while(1)                         //La condition est toujours réalisée
{                                //le début de bloc while
   MaData = MaData + 1;
   MaData2 = MaData + 2;
}                                //la fin du bloc while
```

13.2 La boucle do while (faire tant que)

La boucle do while est exécutée tant que la condition entre parenthèse est réalisée.
Le test est fait en fin de boucle, le traitement est toujours exécuté au moins une fois quelque soit la condition d'entrée.

```
do
{                                // le début de bloc du while
   MaData = MaData + 1;
   MaData2 = MaData + 2;
} while(MaData <= ProvData);     // condition entre () est testée
                                 // en fin de bloc, traitement est
                                 // effectué au moins une fois.
```

13.3 La boucle for

La boucle for est exécutée selon trois conditions spécifiées en début de boucle. Elle est surtout utilisée pour des boucles avec compteurs de boucles.

```
for(cond-1 ;cond-2 ;cond-3)      // les conditions de la boucles
{                                // le début de bloc for
   MaData = MaData + 1;
   MaData2 = MaData + 2;
}                                // la fin du bloc for
```

13.3.1 Les conditions de la boucle for

```
for(cond-1 ;cond-2 ;cond-3)     // les 3 conditions de la boucles
                                // sont entre les ()
```

- cond-1 C'est l'initialisation du critère de boucle. Cette initialisation est effectuée une seule fois au début de la 1ère boucle.
  Remarque : Il est possible d'indiquer plusieurs critères séparés par des virgules et terminés par un ;

```
cond-2     C'est la condition de test, si elle vraie la boucle est
exécutée.
.Le test est effectué au début de la boucle.
```

Remarque : Il est possible d'indiquer une combinaison logique de plusieurs critères.

```
cond-3     C'est la modification du compteur de boucle. En pratique on
utilise la post
incrémentation, c'est à dire une incrémentation en fin de boucle.
```

Remarque : Il est possible d'indiquer plusieurs critères séparés par des virgules et terminés par un ;

13.3.2 Exemple d'une boucle for

```
for(i=0;i<3;i++)                        // conditions de la boucles
{                                       // le début de bloc for
   printf ("la valeur de i = %d/n",i);
}                                       // la fin du bloc for
```

L'exécution de cette boucle donnera le résultat suivant :

```
la valeur de i = 0
la valeur de i = 1
la valeur de i = 2
```

Il est possible d'omettre des conditions du for. Si la condition n'existe pas elle considérée comme vraie.

13.3.3 La boucle for infinie

```
for(;;)                                 // Sans condition la boucles
{                                       // est sans fin
   MaData = MaData + 1;
   MaData2 = MaData + 2;
}                                       // la fin du bloc for
```

Remarque : Chaque condition est optionnelle.

13.3.4 La boucle for a plusieurs conditions.

```
for(i=0,j=4;i<3 && j>0;i++,j--)            // Conditions multiples
{                                          // Début de bloc
printf (”la valeur de i = %d j=%d/n”,i,j);
}                                          // fin du bloc for
```

L’exécution de cette boucle donnera le résultat suivant :

```
la valeur de i = 0 j = 4
la valeur de i = 1 j = 3
la valeur de i = 2 j = 2
```

13.4 L’instruction break (rupture)

L’instruction break permet de sortir d’une boucle for, while, do while par un saut à la fin de la boucle. (nous verrons que cette instruction est aussi utilisée dans un switch)
Exemple :

```
while(1)                          // La boucle est infinie
{                                 //
   MaData = MaData + 1;
   if (MaData >= 10)
     break;                       // sortie du while par un saut
}                                 // à la fin de la boucle.
```

L’instruction break fonctionne de la même manière dans les boucles while, do while et for.

13.5 L’instruction continue

L’instruction continue permet de sauter à l’itération suivante est de sauter un traitement.
La fin du traitement de la boucle est sauté et l’on passe directement à la boucle suivante.
Exemple :

```
for(i=0 ;i<3;i++)                      //
{                                      //
   if (i == 1)                         //Si cette condition est vraie
     continue ;                        //On saute à la boucle suivante
   printf (”la valeur de i = %d/n”,i);
}                                      // la fin du bloc for
```

L’exécution de cette boucle donnera le résultat suivant :

```
la valeur de i = 0
la valeur de i = 2
```

L’instruction continue fonctionne de la même manière dans les boucles while, do while et for.

13.6 L'instruction GOTO(Aller à)

L'instruction goto a régné sur la programmation informatique jusqu'à l'arrivée du langage Pascal. Son usage irraisonné a donné naissance à des problèmes de logiciel totalement insurmontables. Actuellement le goto est bannit de la programmation sérieuse. Il reste cependant quelques cas ou le goto est encore nécessaire.
Il s'agit par exemple du traitement d'exceptions nécessitant un redémarrage du programme, ou d'un programmation devant être optimisée en vitesse d'exécution. Attention danger, a utiliser avec extrême prudence et modération.

```
reset_all:                          // label pour le saut du programme

if (b==0)                           // risque de division par 0
   goto reset_all                   // erreur, nouvelle initialisation
else
   {
      a = a/b ;
   }
```

13.7 L'instruction switch (sélection)

L'instruction switch permet un aiguillage vers différents blocs d'instructions en fonction d'une condition. La condition du switch est entre les (), le programme saute au cas correspondant.

```
switch(i)                  // i est la condition du switch
{
   case 0 :                // Est exécuté si i == 0
      j++ ;
      break ;              // Saut à la fin du switch
   case 1 :                // Est exécuté si i == 1
      j-- ;
      break ;              // Saut à la fin du switch
   case 2 :                // Est exécuté si i == 2
      j = j + 2;
      break ;              // Saut à la fin du switch
   default :               // Est exécuté en cas d'erreur,
      i = 0;               // la valeur de i n'est pas prévue.
      break ;              // saut à la fin du switch
}
```

Remarque : l'instruction break permet le saut à la fin du switch. En cas d'absence du break, le traitement du case suivant est effectué.

```
switch(i)
{
   case 0 :                // Exécuté si i == 0
      printf ("C'est le cas 0");
      break ;              // saut à la fin du switch
   case 2 :                // Exécuté si i == 2
      printf ("C'est le cas 2");
   case 'A' :              // Exécuté si i == au caractère ascii A
      printf ("C'est le cas A");
      break ;              // saut à la fin du switch
   default :               // Erreur, valeur de i non prévue.
      printf ("C'est le cas d'erreur");
      break ;              // saut à la fin du switch
}
```

Remarques :

- Le case (i == 1) n'est pas prévu dans le switch.
- Le case (i == 2) n'a pas d'instruction break, le traitement du case 'A' sera effectué à la suite de case 2.

Ce programme donnera les résultats suivants :

avec i == 0 => on exécute le case 0, on affiche
C'est le cas 0

avec i == 1 ou (i > 2 et i != 'A') =>On exécute la case default car le cas n'est pas prévu.
C'est le cas d'erreur

avec i == 2 => on exécute le case 2 et 3 car l'instruction break du cas 2 manque.
C'est le cas 2
C'est le cas A

avec i == 'A' => on exécute le case 'A'
C'est le cas A

14 Le type enum (énuméré)

Il peut être nécessaire d'affecter des valeurs à une liste de symboles. Le type enum permet de réaliser cette liste. Exemple:

```
enum couleur          // nom facultatif de l'énumération
     {
     ROUGE,           // rouge vaut 0
     BLEU,            // bleu vaut 1
     VERT = 4,        // vert vaut 4
     JAUNE            // jaune vaut 5
     };
```

Si aucune valeur n'est précisée le premier élément vaut 0. Chaque élément suivant vaut 1 de plus que le précédent excepté si sa valeur est définie.

14.1 Le switch et l'énumération

Un programme sérieux ne doit contenir aucun chiffre. C'est aussi vrai pour le switch.
Pour assurer la sécurité et fiabilité du switch, **il est indispensable chaque fois que c'est faisable de baser chaque cas (case) sur une énumération**. Un switch sera ainsi fiable et modifiable. On a alors :

```
enum Status     // Enumération des cas du switch
{
   ETAT_0,      // ETAT_0 vaut 0
   ETAT_1,      // ETAT_1 vaut 1
   ETAT_2       // ETAT_2 vaut 2
};

switch(i)                  // i est la condition du switch
{
   case ETAT_0 :           // Est exécuté si i == ETAT_0 == 0
      j++ ;
      break ;              // Saut à la fin du switch
   case ETAT_1 :           // Est exécuté si i == ETAT_1 == 1
      j-- ;
      break ;              // Saut à la fin du switch
   case ETAT_2 :           // Est exécuté si i == ETAT_2 == 2
      j = j + 2;
      break ;              // Saut à la fin du switch
   default :               // Est exécuté en cas d'erreur,
      i = 0;               // la valeur de i n'est pas prévue.
      break ;              // saut à la fin du switch
}
```

Pour obtenir une fiabilité maximum **il est indispensable de baser le switch sur une énumération**. On obtient ainsi une **fiabilité maximum** et on crée ainsi un **programme modifiable**.

15 Les problèmes dus à la fonction scanf( )

La fonction scanf() pose des problèmes insurmontables. Elle ne peut pas être utilisée dans un programme professionnel pour les raisons suivantes.

- Aucune maîtrise n'est possible lors de la conversion du code ASCII du clavier en une valeur numérique.
- Lors d'un appel à la fonction scanf, le PC est en attente d'une pression sur le clavier, pendant cette attente aucune tâche ne peut être exécutée.
- Lors de plusieurs lectures an moyen d'une boucle le buffer tampon n'est pas encore vidé lors de la boucle suivante, alors le même caractère peut être lu plusieurs fois.

15.1 La maîtrise de la conversion de type.

Pour maîtriser la conversion de type il faut faire l'acquisition des caractères frappés au clavier en deux temps.

- Acquérir les données dans un buffer temporaire de type char[ ], ce qui permet l'acquisition sans conversion, les caractères sont stockés en code ASCII.
- Convertir le code ASCII en valeur numérique en utilisant la procédure sscanf().

15.1.1 La fonction gets()

Elle permet l'acquisition d'un string sans conversion.

15.1.2 La fonction sscanf()

Cette fonction convertit un string en une valeur numérique. Elle est appelée avec plusieurs paramètres.

15.1.2.1 En entrée

- Le string à convertir
- Le (les) types de la (des) conversions.
- La ou les variables de destination.

15.1.2.2 En sortie

- Le compteur de conversion, qui indique le nombre de conversions réussies.

Exemple:

```
void main (void)
unsigned char     TempString[80];         // string temporaire
unsigned int      CntInput = 0;           // nombre de champs acquis
unsigned int      JourMois;               //
unsigned int      Mois;                   //
unsigned int      Année;                  //

clrscr();
while(1)
{
   printf("entrer une date selon le format jj mm aa\n");
   gets(TempString);
   CntInput = sscanf(TempString,"%d %d %d", &JourMois, &Mois, &Année);
   if ((CntInput == 3) && (JourMois >= 1) && (JourMois <= 31)
                       && (Mois >= 1) && (Mois <= 12) && (Année <= 99))
      break;
   printf("le format n'est pas juste, recommencer !\n");
};
```

Cette méthode permet d'effectuer la conversion ASCII => numérique après coup avec sécurité.

15.2 La lecture au vol du clavier.

Pour supprimer l'attente il faut tester si une touche du clavier est pressée avant la lecture. Cette détection est faite par la fonction Kbhit(). Si une touche est pressée alors nous effectuons la lecture du clavier. Chaque caractère est lu individuellement, l'un après l'autre, au moyen de la fonction getch().

15.2.1 La fonction kbhit()

Cette fonction permet de savoir si une touche est pressée. Les retourne un status true si une touche est pressée.

15.2.2 La fonction getch()

Elle lit un caractère ASCII du clavier. Elle n'affiche pas d'écho à l'écran. Le programmeur doit lui-même programmer l'envoi de l'écho si nécessaire.

15.2.3 La fonction putch()

Elle affiche un caractère ASCII à l'écran. Le programmeur effectue lui-même l'écho du caractère tapé.
Exemple:

```
while(1)
{
   if (kbhit())            // test si une touche est pressée
   {
      c=getch();           // lecture de la touche
      putch(c);            // écho à l'écran
   }
   else
   {
                           // place pour une tâche urgente qui est
                           // effectuée chaque fois qu'aucune touche
                           // du clavier est pressée.
   }
}
```

Cette méthode permet d'effectuer des tâches simultanées, par exemple l'affichage du temps réel dans une zone de l'écran et en même temps saisir une commande au clavier.

15.2.4 Remarques.

Certaines touches du clavier envoient deux caractères comme par exemple les flèches, les touches de fonctions, etc. Dans ce cas le premier caractère reçu est le #0 et un deuxième appel à getch doit être effectué.

16 Les fonctions

Le langage C ne connaît pas la notion de procédure. Seule la notion de fonction existe.
Cette fonction peut avoir zéro ou plusieurs paramètres d'entrée, elle peut avoir zéro ou un paramètre de retour.
La fonction main est la première fonction obligatoire rencontrée par le programme et appelée depuis le boot loader.

16.1 Fonction sans paramètre

Une fonction sans paramètre d'entrée et sans paramètre de retour se déclare de la manière suivante :

```
void MaFonction(void) ;          // Fonction sans paramètre
{                                // void = vide
   MaData++ ;
}
```

Remarque : Le mot anglais void veut dire vide.
Le premier mot void indique qu'il n'y a par de paramètre de retour.
MaFonction est le nom de la fonction.
Le 2ème mot (void) indique qu'il n'y a pas de paramètre d'entrée.

16.1.1 Invocation d'une fonction sans paramètre

```
MaFonction(void) ;    // Invocation d'une fonction sans paramètre
MaFonction() ;        // le mot void (vide) peut être omis
```

16.2 Fonction avec paramètres d'entrée

Une fonction avec des paramètres d'entrée et sans paramètre de retour se déclare de la manière suivante :

```
void MaFonction(int MonParam1,float MonParam2)
{                           //fonction avec 2 paramètres d'entrée
   MonParam1++ ;
   MonParam2 = MonParam2 + 100 ;
   printf(" %3d %3.3f",MonParam1, MonParam2) ;
}
```

Cette fonction à deux paramètres d'entrée, le paramètre MonParam1 est un integer, le paramètre MonParam2 est un flotting point.
Cette fonction incrémente le paramètre d'entrée param1, et ajoute100 au paramètre d'entrée MonParam2, puis affiche le résultat.

16.2.1 Invocation d'une fonction avec paramètres d'entrée

```
int MonInt=3;
float MonFloat=6;

MaFonction(MonInt, MonFloat);   // appel de fonction avec paramètres
```

On passe à cette fonction des **copies** des variables MonInt et MonFloat.
Ces copies sont **mises sur le stack avec l'adresse de retour** de la fonction.
La fonction appelée ne peut pas modifier les variables MonInt et MonFloat puisqu'elle a reçu des copies de ces variables.

16.3 Fonction avec un paramètre de retour.

Une fonction avec un paramètre de retour se déclare de la manière suivante :

```
int MaFonction(void)       // cette fonction retourne un integer
{
int MonIntLocal ;

   MonIntLocal = 10;
   return(MonIntLocal) ;   // La fonction retourne MonIntLocal
}
```

Cette fonction a un paramètre de retour qui est un integer. Le mot return permet de passer la variable de retour à l'appelant.

16.3.1 Invocation avec récupération du paramètre de retour.

```
int MonInt;

MonInt = MaFonction(void); // Récupération du paramètre de retour
```

On affecte à MonInt le paramètre de retour de la fonction MaFonction.
Après l'exécution de cette ligne de code MonInt aura la valeur retournée par la fonction MaFonction, c'est à dire 10 dans cet exemple.

16.4 Fonction avec paramètres d'entrée et de retour.

Une fonction avec paramètre d'entrée et de retour se déclare de la manière suivante :

```
int MaFonction(int MonParam1,int MonParam2)
{          // reçoit un int et un float et retourne un integer
   int MonIntLocal ;

   MonIntLocal = MonParam1 + MonParam2;
   return(MonIntLocal) ;
}
```

Cette fonction a un paramètre de retour qui est un integer.

16.4.1 Invocation avec récupération du paramètre de retour.

```
int MonInt = 2;
int MonInt2 = 4;

MonInt = MaFonction(MonInt,MonInt2);// Paramètres d'entrée,de retour
```

On affecte à MonInt le paramètre de retour de la fonction MaFonction à laquelle on a passer sur le stack une copie des variables MonInt et MonInt2.
Après l'exécution de cette ligne de code MonInt aura la valeur retournée par la fonction MaFonction, c'est à dire 6 dans cet exemple.

16.5 Sortie prématurée d'une fonction sans paramètre de retour.

De la même manière qu'il est possible de sortir d'une boucle avant la fin par l'instruction break, Il est possible de sortir d'une fonction par l'instruction return.

```
void MaFonction(int MonParam)   // fonction sans paramètre de retour
{
void MonIntLocal ;

   MonParam++;
   if (MonParam > 10)           // condition du retour
      return(void);             // Retour prématuré de la fonction
   MonIntLocal = MonIntLocal + MonParam;
}
```

L'exécution est interrompue si MonParam > 10, la fin de la fonction n'est pas exécutée.

16.6 Sortie prématurée d'une fonction avec paramètre de retour.

De la même manière qu'il est possible de sortir d'une boucle avant la fin par l'instruction break, Il est possible de sortir d'une fonction par l'instruction return.

```
int MaFonction(int MonParam)    // fonction sans paramètre de retour
{
   int MonIntLocal=2;

   MonParam++;
   if (MonParam > 10)                // condition du retour avancé
      return(MonParam);              // Retour prématuré
   return(MonIntLocal + MonParam);   // retour normal
}
```

L'exécution est interrompue si MonParam > 10, la suite de la fonction n'est pas exécutée.
Si MonParam > 10, elle retourne MonParam
Si MonParam <= 10 elle retourne le résultat de l'opération (MonIntLocal + MonParam).

Remarque : Si une fonction est prévue avec un paramètre de retour, tous les chemins de retour doivent retourner un paramètre cohérent avec la déclaration de la fonction.

16.7 Documentation des fonctions

Une fonction doit pouvoir être utilisée sans avoir à analyser le code. L'entête de la fonction doit fournir tous les renseignements permettant son utilisation, elle comprend au minimum :

- Le nom de la fonction
- La description de sa fonctionnalité (ce qu'elle fait)
- La description des paramètres en entrée
- La description du paramètre de retour.

Si la fonction n'utilise pas de paramètre ce renseignement doit être mentionné. Exemple :

```
/*****************************************************************
FONCTION:       DayOfWeek
BUT:            Cette fonction détermine le jour de la semaine
EN ENTREE:      Year_IN    L'année
                Month_IN   Le mois
                Day_IN     Le jour du mois
EN SORTIE:      Jour de la semaine (1 = lundi, 7 = dimanche)
*****************************************************************/
```

17 Variables globales et locales.

17.1 Les variables globales

Les variables globales déclarées dans l'en-tête du programme principal (en dehors de la fonction main).

```
#include   <stdio>

int MaVarGlobale ;              // cette variable est globale

void main (void)
{

}
```

Les variables globales sont accessibles depuis toutes les fonctions du programme.

17.1.1 Durée de vie des variables globales

Les variables globales ont une durée de vie égale au temps d'exécution du programme. Le linker attribue à ces variables une position mémoire RAM valide pour toute la durée d'exécution du programme.

17.1.2 Documentation des variables globales

Toutes les variables globales doivent être documentées.

17.2 Les variables locales

Les variables locales sont déclarées au début d'une fonction après l'accolade ouvrante du début de bloc.

```
#include   <stdio>


void main (void)
{
   int MaVarLocale;             // Variable locale à la fonction main.

}
```

Les variables locales sont accessibles uniquement depuis la fonction ou elles ont été déclarées.

17.2.1 Durée de vie des variables locales

Les variables locales ont une durée de vie égale au temps d'exécution de la fonction dans laquelle elles sont déclarées. Lors du début de l'exécution d'une fonction une position RAM sur le stack est attribuée à chaque variable locale.

17.3 Les variables locales statiques

Dans certain cas on peut vouloir conserver l'état d'une variable locale entre deux exécutions d'une même fonction. On déclare alors une variable locale statique. Cette variable est similaire à une variable globale à l'exception qu'elle n'est accessible que depuis la fonction où elle a été déclarée.

```
#include   <stdio>


void main (void)
{
   static int MaVarLocale;           // variable locale et statique

}
```

17.3.1 Durée de vie des variables locales statiques

Les variables locales statiques ont une durée de vie égale au temps d'exécution du programme. Le linker attribue à ces variables une position mémoire RAM valide pour toute la durée d'exécution du programme. Elles sont accessibles uniquement depuis la fonction ou elles ont été déclarées.

17.4 Choix du genre de variables (globales, statiques, locales).

Lors de la réalisation de logiciels importants, garder une maîtrise parfaite du logiciel est indispensable mais pas toujours facile à réaliser. Quelques règles élémentaires permettent de contribuer à conserver la maîtrise du logiciel. L'oubli de ces règles simples mais indispensables peuvent mener à la catastrophe.

17.4.1 L'usage des variables globales doit être minimum.

Elles sont **dangereuses car elles peuvent être modifiées par toutes les fonctions du programme.** Leur nombre doit être limité au maximum.

17.4.2 Les variables locales doivent être utilisées au maximum.

Chaque fois que c'est possible, on utilisera des variables locales ou locales statiques, on aura ainsi la certitude que notre variable ne pourra pas être modifiée par erreur par une autre fonction.

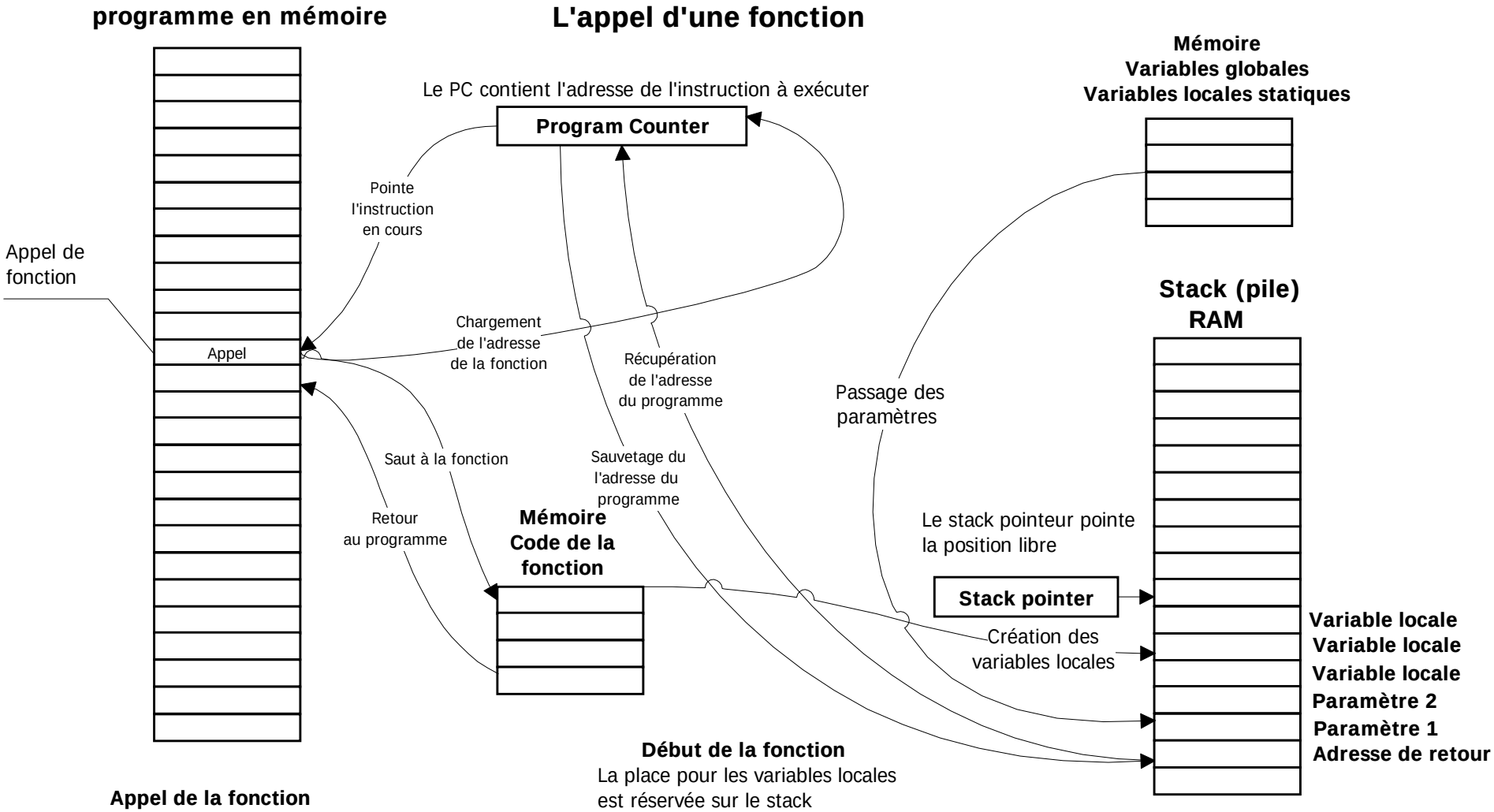
17.4.3 Le passage de paramètres doit être maximum.

Le passage de paramètre permet d'éviter l'utilisation des variables globales. Le passage de paramètre transmet une copie de la variable et non pas la variable elle-même. Pour ces deux raisons la fiabilité d'un programme est augmentée par l'usage maximum du passage de paramètres.

17.4.4 Utilisation des variables globales

Lors de l'utilisation de variables globales, il faut pour en garder la maîtrise avoir **un seul écrivain** mais on peut avoir **plusieurs lecteurs**. La variable est modifiée par une seule fonction mais peut être lue par tout le programme. (Similitude avec la programmation orientée objet.)

L'appel d'une fonction



Appel de la fonction

1 Le programme rencontre un appel de fonction
2 L'adresse du programme dans le PC est sauvée dans le stack
3 Les paramétres à transmettre sont copiés dans le stack
3 L'adresse de la fonction est chargée dans le PC

Début de la fonction

La place pour les variables locales est réservée sur le stack

Fin de la fonction

Le stack dépile les variables locales et les paramétres qui sont perdus
L'adresse de retour est remise dans le PC

18 Les prototypes des fonctions

Le prototype d'une fonction est une déclaration faite en début de fichier. Elle fournit les caractéristiques d'une fonction, le nombre et le type de ses paramètres d'entrées et du paramètre de sortie.

18.1 Les prototypes des fonctions utilisateurs.

Le compilateur effectue la création du code selon l'ordre logique du début à la fin du fichier. Si un appel de fonction est rencontré avant que cette fonction ait été compilée, le compilateur, génère une erreur car les caractéristiques de la fonction sont encore inconnues.

```
void main(void)
{
   int MaData ;
   int MaData2 = 5 ;

   MaData = Mafonction(MaData2); // MaFonction n'est pas encore connue
                                 // erreur de compilation !
}

int MaFonction(int MonParam)
{
   return(MonParam);
}
```

Dans cet exemple le compilateur va générer une erreur car au moment de compiler la fonction main, les caractéristiques de la fonction MaFonction sont encore inconnues. Pour résoudre ce problème on déclare en début de fichier le prototype de la fonction à appeler. Les caractéristiques des fonctions sont données par les prototypes des fonctions.

```
//================== prototype des fonctions ======================

int MaFonction(int);              // prototype de la fonction MaFonction

//====================== début des fonctions ======================

void main(void)
{
   int MaData ;
   int MaData2 = 5 ;

   MaData = Mafonction(MaData2) ; // MaFonction est connue (prototype)
                                  // => pas d'erreur
}

int MaFonction(int MonParam)
{
   return(MonParam);
}
```

Maintenant le compilateur a dès le début de la compilation une information complète sur les caractéristiques des fonctions à compiler. Il pourra effectuer l'appel dans la fonction main, il sait que la fonction MaFonction est caractérisée par un int en paramètre de retour et un int en paramètre d'entrée. Les prototypes des fonctions, permettent d'écrire celles-ci dans un ordre quelconque.

18.2 Les prototypes des fonctions standards

Lorsqu'un programme utilise des fonctions de librairie du système, le même genre de problème est rencontré. L'appel d'une fonction du système est résolu conjointement par le compilateur et par le linker.
Le compilateur prépare le passage des paramètres, le linker s'occupe des adresses des fonctions appelées (linker = éditeur de liens).
Lors d'un appel d'une fonction de librairie, il est donc nécessaire de fournir au compilateur une information complète sur les paramètres des fonctions des librairies du système.

```
//================== prototype des fonctions ======================

int MaFonction(int);          // prototype de la fonction MaFonction

//====================== début des fonctions ======================

void main(void)
{
   printf("Hello/n");     // Appel d'une fonction de librairie, erreur
                          // le prototype est inconnu
}
```

Dans le cas suivant la structure d'appel de la fonction de librairie est inconnue et le compilateur va générer une erreur "missing function prototype".

```
//================ prototypes des fonctions standards ==============

#include <stdio.h>         // prototype des fonctions de librairie

//================== prototype des fonctions ======================

int MaFonction(int);       // prototype de la fonction MaFonction

//====================== début des fonctions ======================

void main(void)
{
   printf("Hello\n");     // Appel d'une fonction de librairie
                          // le prototype est dans stdio.h => OK
}
```

De la même manière que l'on a déclaré nos propres prototypes de fonctions, on va inclure à notre fichier les prototypes des fonctions de librairie se trouvant dans les fichiers header *.h
La déclaration #include <stdio.h> permet d'inclure dans notre fichier les prototypes des fonctions se trouvant dans la librairie stdio.lib

18.3 Les prototypes des fonctions en programmation multi-fichiers

Des programmes de grandes tailles sont découpés en plusieurs fichiers. Si l'on désire qu'une fonction puisse être appelée depuis un autre fichier il est nécessaire de mettre à disposition les prototypes de nos propres fonctions. On déclare alors notre propre fichier header.h dans lequel on va mettre les déclarations des prototypes de nos fonctions. Toutes les déclarations des prototypes seront alors faites dans notre propre fichier header.

```
/*****************************************************************
Fichier         MonFichier.h
Description :   Fichier header de mon fichier.c
Auteur :        JMC
Version :       1.0
Modifications : sans modification
*****************************************************************/
//================ prototypes de mes fonctions ====================

int MaFonction(int);       // prototype de la fonction MaFonction

//=================================================================
```

Ce fichier header MonFichier.h sera inclus chaque fois que l'on désire appeler une fonction du fichier MonFichiers.c

```
/*****************************************************************
Fichier         MonFichier.c
Description :   Main program du système xxx
Auteur :        JMC
Version :       1.0
Modifications : nothing
*****************************************************************/
//======================= fichiers inclus =========================

#include <stdio.h>         // Prototype des fonctions de librairie
#include "MonFichier.h"    // Les prototypes de mes fonctions

//====================== début du programme =======================

void main(void)
{
   int MaData ;
   int MaData2 = 5 ;

   printf(Hello\n);
   MaData = Mafonction(MaData2) ; // MaFonction est connue
                                  // le prototype est dans MonFichier.h
}

int MaFonction(int MonParam)
{
   return(MonParam);
}
```

19 Les directives du préprocesseur

Comme nous l'avons déjà vu le préprocesseur permet la transformation du langage C écrit par le programmeur en un langage C "pur".
Nous avons déjà vu un certain nombres de ces directives du preprocesseur, elles commence toutes par le signe #

19.1 La directive préprocessor #include

Elle permet d'inclure un fichier dans un autre.

19.1.1 L'inclusion des header d'une librairie standard est faite comme suit:

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

19.1.2 L'inclusion des header des fichiers utilisateur est faite comme suit :

```
#include "MonHeader.h"
```

19.2 La directive préprocessor #define

Cette directives à deux fonctionnalités distinctes, la définition de symboles et la définition de macro-instructions. Elle permet le remplacement d'un symbole par la valeur qu'on a attribuée à ce symbole, cette attribution pouvant être une suite quelconque de caractères.

19.2.1 #define en tant que définition de symboles alphanumérique.

La directive #define permet d'attribuer un symbole à un string quelconque.
Ce string peut contenir une valeur numérique, une chaîne de caractères, une instruction.
Il remplace d'une suite de caractères complexe par un symbole simple.

```
#define HELLO   "Bonjour monsieur Dupont, comment allez-vous\?"

printf (HELLO);     // HELLO est remplacé par la valeur de son #define
```

Dans cet exemple le pré-compilateur va remplacer le symbole HELLO par la phrase attribuée au symbole HELLO c'est à dire ("Bonjour monsieur ……..")
Si ce texte doit être afficher à plusieurs endroits dans le programme l'écriture sera simplifiée.

19.2.2 Les labels des défines

L'usage professionnel veut que l'on écrive **le label du define entièrement en majuscule**.
Cette habitude permet de repérer immédiatement les defines dans un code source.

```
#define BONJOUR    "Bonjour monsieur Dupont"

printf (BONJOUR);     // Lors de la pré-compilation BONJOUR est remplacé
                      // par le texte "Bonjour monsieur Dupont"
```

19.2.3 #define en temps que définition de symboles numériques.

La directive #define permet d'attribuer à un symbole une valeur numérique. L'usage veut que le programmeur écrive ces symboles entièrement en majuscules. Cette directive est très importante car elle permet d'augmenter la lisibilité et elle facilite les modifications éventuelles.
Rappelons qu'un programme professionnel ne doit comprendre aucun chiffre, il doit comprendre que des symboles auquel on a attribué une valeur numérique.

Ce qu'il ne faut jamais faire !

```
for (i=0 ;i < 10;i++)
{
   …
}
if (a < 10)
   a++ ;
```

Si la valeur 10 doit être changée il faut faire deux modifications, la valeur 10 n'est pas documentée. C'est faux !

Ce qu'il faut toujours faire !

```
#define MA_LIMITE    10   // MA_LIMITE est un symbole = 10

for (i=0 ;i < MA_LIMITE;i++)
{
   …
}
if (a < MA_LIMITE)
   a++ ;
```

Le changement de la limite nécessite une seule modification, elle est auto-documentée. C'est juste !

19.2.4 #define en temps que définition de macro-instructions

Le #define remplaçant un symbole par un string il est possible d'attribuer à ce symbole des instructions.

```
#define CARRE(a) a*a            // Macro calculant le carré


void main()
{
   int MaData=2;
   int result;

   result = CARRE(MaData);      // le préprocesseur va remplacer cette
                                //  instruction par MaData*MaData
}
```

Il est ainsi possible de créer ses propres macro-instructions que l'on mettra si possible en librairie utilisateur.

19.2.5 #define en temps que définition de macro de macro

Le #define remplaçant un symbole par un string il est possible d'attribuer à ce symbole instruction contenant une instruction.

```
#define DIF(a,b) a-b            // Macro différence
#define CARRE(a) a*a            // Macro calculant le carré


void main()
{
   int MaData = 2;
   int MaData2 = 3;
   int result;

   result = DIF(CARRE(MaData),CARRE(MaData2));
                               // le préprocesseur va remplacer
                               // la macro instruction par :
                               //  MaData*MaData – MqaData2*MaData2
}
```

Il est ainsi possible d'utiliser des macro-instructions utilisant d'autres macro-instructions.

19.3 La directive préprocessor de compilation conditionnelle.

Les directives de compilation conditionnelle permettent de créer différents fichiers exécutables avec le même fichier source. Ceci est très utile pour générer différentes versions d'un logiciel avec le même fichier source. Cette pratique permet aussi d'activer des fonctions de test.

```
#define DEBUG

#ifdef DEBUG               // Compilé si DEBUG est défini
   printf("C'est la version debug")
#else                      // compilé si DEBUG n'est pas défini
   printf("C'est la version standard")
#endif                     // fin de la compilation conditionnelle.
```

Selon que DEBUG est défini ou pas on va créer deux logiciel différents, la partie non activée est considérée comme du commentaire.

```
#define FRENCH

#ifdef GERMAIN             // Compilé si GERMAIN est défini
   printf("C'est la version allemande")
#elseif FRENCH             // compilé si FRENCH est défini
   printf("C'est la version française")
#else                      // compilé si FRENCH et GERMAIN ne
                           //sont pas défini
   printf("C'est la version anglaise")
#endif                     // fin de la compilation conditionnelle.
```

Ici on peut compiler trois variantes selon le choix de la langue. Les deux parties non activées sont traitées comme du commentaire.

20 Les tableaux

Le tableau est un ensemble d'éléments du même type rangé dans une même unité. On accède à un élément du tableau par le ou les indices (index) de l'élément désiré. Les tableaux peuvent avoir une ou plusieurs dimensions. Tous les éléments d'un tableau sont obligatoirement du même type.
Il y a autant d'indice que de dimension au tableau.
Les indices sont toujours des entiers positifs.

Exemple tableau à 1 dimension de 10 éléments contenant 10 nombres réels.

1.0	3.1	7.7	1.3	1.4	1.5	4.3	1.7	1.8	1.9

20.1 Déclaration d'un tableau à une dimension

```
#define    DIM_TABLEAU      10                  // dimension du tableau

int MonTableau[DIM_TABLEAU];                    // déclaration du tableau
```

Cette déclaration crée un tableau appelé MonTableau dont le nombre d'éléments est égal à DIM_TABLEAU.

20.1.1 Accès à un élément du tableau

On accède à un élément d'un tableau grâce à l'indice du tableau.
L'indice est toujours un entier positif compris dans les dimensions du tableau.
Si on crée un tableau de 10 éléments, l'index du premier élément est 0, l'index du dernier élément est 9.

```
0 <= i < DIM_TABLEAU
```

L'accès à un élément du tableau se fait de la manière suivante :

```
MaDonnée = MonTableau[i];                       // lecture de l'élément du tableau
```

Exemple : On copie le 5ème élément dans le 6ème élément du tableau.

```
#define    DIM_TABLEAU      10
float MonTableau[DIM_TABLEAU];

{
   int i;
   float MaDonnée;

   i = 5 ;                                      // initialisation de l'indice
   MaDonnée = MonTableau[i] ;                   // lecture de l'élément du tableau
   MonTableau[i+1] = MaDonnée ;                 //copie dans l'élément suivant
}
```

L'élément d'indice 5 est copié dans MaDonnée.

1.0	1.1	1.2	1.3	1.4	**1.5**	1.6	1.7	1.8	1.9

MaDonnée est copiée dans la position 6 du tableau.

1.0	1.1	1.2	1.3	1.4	1.5	**1.5**	1.7	1.8	1.9

20.2 Déclaration d'un tableau à plusieurs dimensions.

Un tableau peut avoir n dimensions avec n∈ entier positifs.

Exemple un tableau à 2 dimensions : de 5 * 6 éléments contenant des réels.

1.0	1.1	1.2	1.3	1.4
2.0	2.1	2.2	2.3	2.4
3.0	3.1	3.2	3.3	3.4
4.0	4.1	4.2	4.3	4.4
5.0	5.1	5.2	5.3	5.4
6.0	6.1	6.2	6.3	6.4

```
#define    DIM_1      5                    // 1er dimension
#define    DIM_2      6                    // 2ème dimension

int MonTableau[DIM_1][DIM_2];              // déclaration du tableau
```

Cette déclaration crée un tableau appelé MonTableau comprenant 30 éléments organisés en matrice de 5 * 6 éléments.

20.2.1 Accès à un élément du tableau

On accède à un élément d'un tableau grâce à deux indices (un par dimension).
L'indice est toujours un entier positif compris dans les dimensions du tableau.

```
0 <= i < DIM_1        0 <= j < DIM_2
```

L'accès à un élément du tableau se fait de la manière suivante :

```
MaDonnée = MonTableau[i][j];                // lecture de l'élément du tableau
```

Exemple : l'accès au 5ème élément du tableau.

```
#define    DIM_1      4                    // 1er dimension du tableau
#define    DIM_2      5                    // 2ème dimension du tableau
float MonTableau[DIM_1][DIM_2];

{
   int i,j;
   float MaDonnée;

   i =2 ;                                  // initialisation de l'indice}
   j =3 ;                                  // initialisation de l'indice

   MaDonnée = MonTableau[i][j];            // lecture de l'élément du tableau
   MonTableau[i+1][j] = MaDonnée;          // copie dans l'élément suivant
}
```

L'élément d'indice 2,3 est copié dans MaDonnée.

```
MaDonnée = MonTableau[i][j];        // lecture de l'élément du tableau
```

1.0	1.1	1.2	1.3	1.4
2.0	2.1	2.2	2.3	2.4
3.0	3.1	3.2	3.3	3.4
4.0	4.1	**4.2**	3.3	4.4
5.0	5.1	5.2	5.3	5.4
6.0	6.1	6.2	6.3	6.4

MaDonnée vaut 4.2

```
MonTableau[i+1][j] = MaDonnée;      // copie dans l'élément suivant
```

MaDonnée est copiée dans l'élément en position 4,3 du tableau.

1.0	1.1	1.2	1.3	1.4
2.0	2.1	2.2	2.3	2.4
3.0	3.1	3.2	3.3	3.4
4.0	4.1	4.2	**4.2**	4.4
5.0	5.1	5.2	5.3	5.4
6.0	6.1	6.2	6.3	6.4

L'élément **MonTableau[i+1][j] vaut 4.2**

Remarque : Il est possible de déclarer des tableaux de tableaux.

20.3 Tableaux de constantes avec initialisation

Un tableau de constante est initialisé au moment de sa déclaration. Cette déclaration est faite de la manière suivante :

```
const int MonTab[2][3] = {{1,3,6},{10,15,18}};//tableau de constantes
```

La lecture de ce tableau donne les résultats suivant :

```
MonTab[0][0] = 1      MonTab[0][1] = 3      MonTab[0][2] = 6
MonTab[1][0] = 10     MonTab[1][1] = 15     MonTab[1][2] = 18
```

21 Les structures

Les structures permettent de ranger dans une même unité des données de types différents. On accède à un élément de la structure par le nom de la structure suivit d'un point puis du nom de l'élément de la structure.

21.1 Déclaration d'un genre de structure

```
struct stock
{
   unsigned long référence ;
   int quantité ;
   float prix ;
};
```

Cette déclaration définit une structure qui n'est pas encore utilisée. A ce stade aucune réservation de mémoire est encore effectuée.

21.2 Réservation de la mémoire pour une structure

Pour utiliser cette structure on déclare.

```
struct stock PièceDétachées;
```

On a réservé une zone mémoire pour une structure de type stock appelée PieceDétachées.

21.3 Utilisation d'une structure

On accède à l'élément de cette structure de la manière suivante :

```
float coût;

coût = PièceDétachées.prix;
```

On affecte à coût le prix se trouvant dans la structure PièceDétachées.

21.4 Structure de structure

Il est possible de déclarer une structure contenant une autre structure. exemple :

```
struct Composant
{
   unsigned int réf2;
   int min;
   int max;
};
struct stock
{
   unsigned long référence ;
   int quantité ;
   float prix ;
   struct Composant divers;
};
```

La structure stock contient une structure de type Composant appelée divers,
La réservation de la place en mémoire est la suivante :

```
struct stock MonStock;
```

On a déclaré une structure de type stock appelée MonStock.
On l'accède à la variable min par :

```
MaDonnée = MonStock.divers.min ;
```

On affecte à MaDonnée un élément de structure de structure.

21.5 Tableau de structures

Il est possible de créer des tableaux de structures. Exemple :

```
#define    DIM_STOCK       100

struct stock PièceDétachées[DIM_STOCK];
```

On a créé un tableau de 100 structures de type PiècesDétachées.
Accès à cette structure est le suivant :

```
int idx = 3 ;
float coût ;

coût = PièceDétachées[idx].prix;
```

On affecte à coût le prix se trouvant dans le tableau de structure.

21.6 Structure contenant des tableaux.

Il est aussi possible de déclarer des tableaux dans des structures. Exemple :

```
#define    DIM_STOCK       100

struct stock
{
   unsigned long référence ;
   int quantité ;
   float prix ;
   int MonTableau[DIM1];
};
```

Ici la structure contient un tableau d'integer
On accède à un élément de ce tableau par :

```
int i=2;
int Ref2;

Ref2 = PièceDétachées.MonTableau[i];
```

On affecte à ref2 la variable se trouvant dans le tableau qui est dans la structure.

21.7 Remarques :

Toutes les combinaisons de tableaux et de structures sont possibles.
Un projet sera clarifié, si l'on réunit les variables globales dans des structures.

22 Les pointeurs

Un pointeur est une variable qui contient l'adresse d'une autre variable. Cette possibilité permet l'adressage indirect. Nous avons vu qu'il n'est pas raisonnable, lors d'un appel de fonction de passer un tableau ou une structure en paramètre. Les raisons sont :

- Le temps d'exécution pour mettre une copie de tout un tableau sur le stack.
- Le taux d'occupation du stack.
- L'impossibilité pour une fonction de retourner un tableau ou une structure.

Il est par contre possible au retour d'une fonction de retourner une adresse d'une structure ou d'un tableau. On aura ainsi un programme rapide utilisant une taille de stack raisonnable.

Remarque : Le pointeur lui-même contient toujours une adresse de la mémoire, il a donc toujours la même taille. La valeur pointée à une taille variable.

22.1 Déclaration d'un pointeur

```
int  *MonPntInt;      // Déclaration d'un pointeur sur un int
```

On a déclaré un pointeur de nom MonPntInt qui pourra contenir l'adresse d'un int.

```
long *MonPntlong;     // Déclaration d'un pointeur sur un long
```

On a déclaré un pointeur de nom MonPntLong qui pourra contenir l'adresse d'un long.

22.2 Initialisation d'un pointeur.

```
long MonLong ;              // Déclaration d'un long
long *MonPntlong;           // Déclaration d'un pointeur sur un long

MonPntLong = &MonLong ;     // Initialisation du pointeur avec
                            // l'adresse
```

On a déclaré un pointeur appelé MonPntLong et on l'a initialisé avec l'adresse de MonLong.

22.3 Accès à une variable pointée.

```
long MonLong = 2;           // Déclaration d'un long
long Tempo;                 // Déclaration d'un long
long *MonPntlong;           // Déclaration d'un pointeur sur un long
MonPntLong = &MonLong;      // Initialisation du pointeur avec une
                            // adresse
Tempo = *MonPntLong;        // affectation avec la valeur pointée.
```

On a affecté à Tempo, la valeur pointée par MonPntLong, c'est à dire MonLong.
En effet MonPntLong contient l'adresse de MonLong. Tempo est maintenant égal à 2.
Monlong a été copié dans Tempo par adressage indirect.

22.4 Remarque :

Le symbole ∗ à deux significations bien distinctes :

```
int  ∗MonPntInt;          // ici ∗ indique une déclaration d'un pointeur.
MonLong2 = ∗MonPntLong;   // ici ∗ indique l'indirection.
```

22.5 Appel de fonction avec passage de pointeurs en paramètres :

Une fonction peut avoir un ou plusieurs pointeurs en paramètres. Ceci est valable pour les paramètres d'entrée et pour le paramètre de sortie.
Exemple :

A l'appel de MaFonction on lui passe comme paramètre l'adresse de MaData ;

```
MaFonction(&MaData);                // Appel de la fonction avec
                                    // le passage d'une adresse

(void) MaFonction(int *MonPntInt) //paramètre d'entrée est un
                                    // pointeur sur un int
{
   *MonPntInt = 0 ;                 // affectation de la variable
                                    // pointée (dans ce cas MaData)
}
```

La fonction MaFonction a comme paramètre d'entrée un pointeur sur un int
La fonction affecte à 0 la valeur pointée par MonPntInt.
Cette méthode permet de passer comme paramètre un pointeur sur une structure ou un pointeur sur un tableau.

22.6 Pointeurs sur un tableau.

```
#define    DIM_TABLEAU     10
#define    DIM_TAB    (DIM_TABLEAU-1)

int Tab[DIM_TABLEAU], *PntTab;        // déclarations
```

On a déclaré un tableau de int et un pointeur sur un int.

```
PntTab = &Tab;                        // Initialisation du pointeur
```

On a affecté au pointeur l'adresse du premier élément du tableau Tab
On peut accéder au premier élément du tableau par

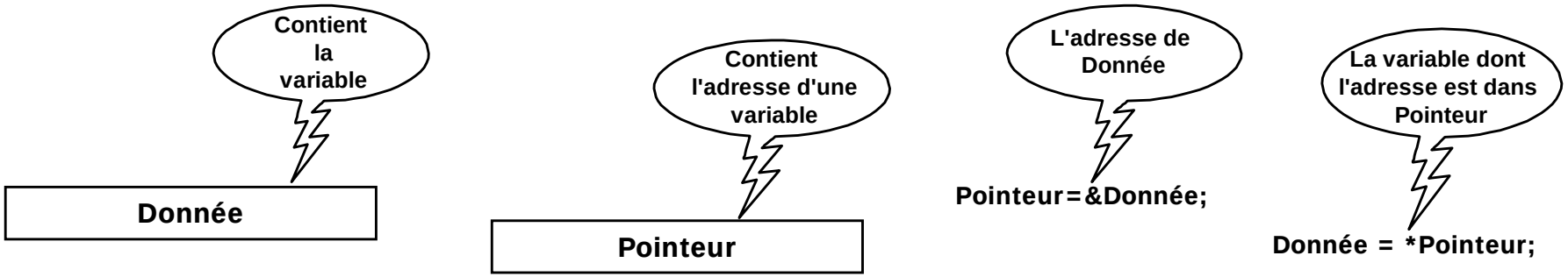
```
MaData = Tab[0];                      // Accès direct au tableau
MaData = *PntTab;                     // Accès indirect au tableau
```

On peut accéder au dernier élément du tableau par

```
MaData = Tab[DIM_TAB];                // Accès direct au tableau
MaData = *(PntTab + DIM_TAB);         // Accès par pointeur
```

On ajoute au pointeur du tableau le décalage désiré et on effectue l'accès indirect.

Adressage direct et indirect (par pointeur)



Passage de paramètres par valeur

MonResultat = MaFonction(Donnée);

Voici une copie de la donnée

Voici la réponse

int MaFonction(int Param)

return(resultat);

Passage de paramètres par adresse

Pointeur=&Donnée;
PntResultat = MaFonction(Pointeur);

Resultat = *PntResultat;

La donnée se trouve à cette adresse

La réponse se trouve à cette adresse

int* MaFonction(int* Param)

return(&resultat);

22.6.1 Remarque

Il existe de nombreuses subtilités possibles pour accéder aux éléments d'un tableau au moyen d'un pointeur, (incrémentation de pointeurs etc). Le choix à été ici volontairement limité à l'essentiel.

22.7 Pointeur sur une structure.

On peut déclarer un pointeur sur une structure.

On déclare une structure stock.

```
struct stock
{
   unsigned long référence ;
   int quantité ;
   float prix ;
   struct Composant divers;
};
```

La structure est définie mais pas encore utilisée.

22.7.1 Réservation en mémoire.

On réserve la place en mémoire pour une structure et un pointeur sur une structure.

```
struct stock MonStock, *PntMonStock;
```

Ici on a déclaré une structure MonStock de type Stock et on a créer un pointeur sur la structure appelé PntMonStock.

On pourra donc initialiser le pointeur sur la structure de la manière suivante :

```
PntMonStock = &MonStock;   // initialisation du pointeur sur la
                           // structure
```

Le pointeur contient maintenant l'adresse du début de la structure MonStock.

22.7.2 Accès à un élément de la structure par le pointeur.

Le pointeur contient l'adresse du début de la structure.
Pour accéder à un élément de la structure on utilise le symbole –>
Exemple :

```
MaData = PntMonStock->quantité ;     //Lecture d'un élément de la
                                     //structure pointée.
```

On a copier dans MaData l'élément quantité de la structure stock, pointée par PntMonStock.

Autre méthode donnant le même résultat

```
MaData = (*PntMonStock).quantité ;   //Lecture d'un élément de la
                                     //structure pointée.
```

22.8 Arithmétique sur les pointeurs

Il est souvent possible de se passer d'arithmétique sur les pointeurs. On utilise alors le pointeur comme référence auquel on ajoute un offset (décalage).
Cette méthode est plus simple que d'effectuer de l'arithmétique sur les pointeurs.

Exemple: accès par modification du pointeur

```
struct stock MonStock[4], *PntMonStock; // tableau de structure + ptr

PntMonStock = &MonStock[0];             // Adresse de la structure

                                        // Accès à la structure suivante

MaData = ++PntMonStock->quantité ;      //Lecture de la 2ème structure

/* Nous avons incrémenté le pointeur pour accéder à la 2ème structure.
   Il va être difficile de conserver la maîtrise de la donnée pointée. */
```

Exemple : Accès par offset sur un pointeur de référence. (index)

```
struct stock MonStock[4], *PntMonStock; // tableau de structure + ptr

PntMonStock = &MonStock[0];             // Adresse de la structure

                                        // Accès à la 2ème structure

MaData = (PntMonStock+1)->quantité ;   // Lecture de la 2ème structure

// Le pointeur pointe toujours sur la même structure, on opère par offset
// (décalage) par rapport à au pointeur. Le contenu du pointeur est
// inchangé, la gestion du pointeur devient simple.
```

Bien qu'il soit possible de faire de l'arithmétique sur un pointeur, cette solution est à éviter autant que possible.

22.8.1 L'incrémentation des pointeurs

Lors de l'incrémentation d'un pointeur, l'adjonction an contenu du pointeur dépend du type de variable pointée. Exemple :

```
int * PntInt;        // pointeur sur un int
long * PntLong;      // pointeur sur un long

PntInt++;            // la valeur du pointeur est augmentée de 1
                     // le prochain int est à l'adresse suivante

PntLong++;           // Si un long est composé de deux mots machine, la
                     // valeur du pointeur est augmentée de 2
                     // le prochain long est deux adresses plus loin
```

23 Strings et pointeurs sur des strings

En C il la variable string est une déclaration d'un tableau de caractère auquel on peut attribuer un pointeur. La création d'une chaîne de caractères sans pointeur est la suivante :

```
char MonString [ ] = ”hello” ;
```

De cette manière on n'a pas de pointeur contenant l'adresse du string.
On peut aussi créer un tableau de string de la manière suivante.

```
const char * jour [7] = {″lundi″,″mardi″,″mercredi″,″jeudi″,″vendredi″,″samedi″,″dimanche″} ;
```

Cette déclaration est un exemple de la grande puissance du langage C

En une seule ligne on a créé grâce à l'adjonction du caractère *

```
7 strings (chaînes de caractères)
un tableau de pointeurs sur chaque string (chaînes de caractères).
```

Ces pointeurs contiennent les adresses du chacun des 7 strings. On pourra ainsi passer à une fonction l'adresse du string comme paramètre et non pas le string entier. L'emploi de cette méthode à 2 avantages majeurs :

```
La place sur le stack sera économisée.
L'appel de la fonction sera rapide.
```

On accède au string désiré en passant en paramètre un index sur un tableau qui lui-même contient l'adresse du string.
Il est aussi à remarquer que grâce à cette méthode la taille mémoire est optimisée pour chaque string.

On peut donc accéder à un string du tableau par :

```
int i = 4 ;                              // 4 pour jeudi
printf(”le jour est %s”, jour[i-1]) ;   // On affiche jeudi
```

Le résultat est Le jour est jeudi

Rappel : en C le premier élément d'un tableau est toujours l'élément d'indice 0.
Le dernier élément à donc l'indice 6.

24 Le casting(changement de type)

Il est à tout moment possible de transférer des données d'un type dans un autre. Cette opération n'est cependant pas sans risque car les données peuvent être tronquées.
Cette méthode comporte des dangers et doit être utilisée avec circonspection.

Exemple: transfert d'un long dans un short

```
long  i = 165364 ;                    //
short j;

// affectation d'un long dans un short;

j = (short)i;        // les bits les plus significatifs sont perdus.
```

25 La manipulation de bits

Le langage C permet des manipulations de bit. On peut ainsi réaliser des programmes de bas niveau, c'est à dire proche de la machine et du code assembler.
Pour manipuler des bits il est courant d'utiliser des masques, en général défini sous la forme #define. Ces masques sont utilisés avec les fonctions logiques and, or, exclusif or.

25.1 Définition de masques

```
#define BIT_0    0x0001     //binaire 0000 0000 0000 0001 => msk bit 0
#define BIT_1    0x0002     //binaire 0000 0000 0000 0010 => msk bit 1
#define BIT_2    0x0004     //binaire 0000 0000 0000 0100 => msk bit 2
#define BIT_3    0x0008     //binaire 0000 0000 0000 1000
#define BIT_4    0x0010     //binaire 0000 0000 0001 0000
#define BIT_5    0x0020     //binaire 0000 0000 0010 0000
#define BIT_6    0x0040     //binaire 0000 0000 0100 0000
#define BIT_7    0x0080     //binaire 0000 0000 1000 0000 => msk bit 7

#define BIT_1_2  BIT_1 | BIT_2  // 0000 0000 0000 0110
                                // masque des bits 1 et 2
```

25.2 Les compléments de masques

On peut inverser un masque avec l'opérateur **~** ,tous les bits du masque sont inversés.

```
#define BIT_0        0x0001      // en binaire 0000 0000 0000 0001
#define NOT_BIT_0    ~BIT_0      // en binaire 1111 1111 1111 1110
```

25.3 Mise à un d'un bit (fonction or)

Pour mettre un bit à un d'une variable on utilise la fonction **ou** avec le masque concerné.
Exemple : Mise à 1 du bit 2

```
int i ;                    // Déclaration de i

i = 0 ;                    // initialisation à 0
i = i | BIT_2 ;            // mise à 1 du bit 2
```

25.4 Mise à un d'un bit en écriture compactée

On peut réaliser la même fonction avec une écriture plus compacte.

```
int i ;               // Déclaration de variable

i = 0 ;               // initialisation de i

i |= BIT_2 ;          // mise à 1 du bit 2 en forme compacte
                      // i = i | BIT_2 est équivalent à i |= BIT_2
```

25.5 Mise à 0 d'un bit (fonction &).

Pour mettre à 0 un bit on utilise une fonction & avec le complément du masque.

```
int MaData ;           // Déclaration de MaData

MaData = 0xffff;       // initialisation avec ffff hexa

MaData &= ~BIT_2;      // mise à 0 du bit 2 en utilisant le
                       // complément du masque avec l'opérateur ~
```

25.6 Inversion d'un bit (fonction exclusif or).

La fonction ou exclusif à la table de vérité suivante :

in A	in B	out
0	0	0
1	0	1
0	1	1
1	1	0

On constate que :

- Si l'entrée IN-B = 0 => la sortie OUT = IN-A.
- Si l'entrée IN-B = 1 => la sortie OUT = NOT IN-A

Nous avons donc une inversion de bit en fonction de notre masque d'un bit IN-B.

Exemple pratique d'inversion d'un bit d'une variable.

```
int MaData ;          // Déclaration de MaData


MaData = 0xffff;      // initialisation de ma data avec ffff

MaData ^= ~BIT_2;     // inversion du bit 2 par ou exclusif
                      // MaData = 1111 1111 1111 1011 binaire

                      // on inverse une deuxième fois le bit 2
MaData ^= ~BIT_2;     // inversion du bit 2 par ou exclusif
                      // MaData = 1111 1111 1111 1111 binaire (0xffff)
```

25.7 Décalage à gauche (multiplication par une puissance de 2).

Il est possible de décaler d'effectuer un décalage à gauche (shift left) par l'opérateur <<

```
int i;                // Déclaration de i

i = BIT_0;            // i = 0000 0000 0000 0001 binaire

i = i << 2;           // rotation à gauche de 2 bits
                      // i = 0000 0000 0000 0100 binaire
```

25.8 Décalage à droite (division par une puissance de 2).

Il est possible de décaler d'effectuer un décalage à droite (shift right) par l'opérateur >>

```
int i;                // Déclaration de i

i = BIT_3;           // i = 0000 0000 0000 1000 binaire

i = i >> 2;           // rotation à droite de 2 bits
                      // i = 0000 0000 0000 0010 binaire
```

25.9 Structure de bits, optimisation de la place mémoire.

Le champ de bits permet d'optimiser la taille des variables en rassemblant des bits ayant une fonction quelconque dans le même mot machine. Il s'agit en fait d'un cas particulier de la structure.

```
struct status_gen                // status général
{
   unsigned OK_WD        : 1 ;   // watch dog status d'un bit
   unsigned ErrorCnt     : 3 ;   // errors counter de 3 bits
   unsigned RadioOk      : 1 ;   // Radio reception flag d'un bit
};
struct status_gen status ;       // variable status (utilise 5 bits).
```

La réservation de mémoire status est un mot machine qui comprend trois éléments distincts.
Un flag pour le watch dog, un compteur de 3 bits, un flag de réception radio.
Tous ces bits sont rassemblés dans le même mot machine.

25.10 Manipulation du champ de bits.

Ce champ de bit s'accède comme les éléments d'une structure.

```
if (status.OK_WD)                // test si OK_WD = 1

status.OK_WD = !status.OK_WD;    // inversion du bit OK_WD

status.ErrorCnt++;               // incrémentation du compteur 3 bits
```

25.11 Résumé des opérateurs de bits.

Ces opérateurs peuvent s'appliquer aux variables de type char, short, int, long.

&	**and, (et niveau du bit)**
|	**or (ou niveau du bit)**
^	**exclusif or (ou exclusif niveau bit)**
<<	**décalage à gauche**
>>	**décalage à droite**
~	**complément à 1**

Rappel des opérations logiques :

0 & 0 = 0	**0 | 0 = 0**	**0 ^ 0 = 0**
1 & 0 = 0	**1 | 0 = 1**	**1 ^ 0 = 1**
0 & 1 = 0	**0 | 1 = 1**	**0 ^ 1 = 1**
1 & 1 = 1	**1 | 1 = 1**	**1 ^ 1 = 0**

25.11.1 Remarques

- L'ordre de rangement des bits peut changer en fonction du compilateur utilisé. Un programme utilisant les structures de bits n'est pas toujours portable.
- Il est possible d'ajouter le type int dans la déclaration du champ de bits, ceci n'a pas grand sens puisque int signifie un mot machine. Nous pouvons déclarer dans une structure de bits :

```
unsigned int MonCnt2bits : 2          // Ces deux déclarations
unsigned MonCnt2bits : 2              // sont identiques

int      MonCnt2bits : 2              // Idem pour un champ signé
signed   MonCnt2bits : 2              // deux déclarations identiques
```

26 L'union

La déclaration d'une union permet de stocker deux variables de types différents à la même adresse physique. Cette possibilité peut avoir des aspects pratiques importants.

26.1 Accès aux même données sous deux formes différentes.

Exemple : Une structure de données est transférée par une communication série. La réception ou transmission des données se fait sous forme d'octets. L'utilisation des données est faite sous forme de structures. La même zone mémoire sera accédée par deux process sous deux formes différentes.
Le process de communication accède les données sous forme de tableau de caractères.
Le process de gestion des données accède les données sous forme d'une structure.
Une déclaration d'union permet de stocker à la même adresse un tableau et une structure.

26.2 Economie de la place mémoire.

L'union peut aussi servir à économiser de la place mémoire si les données ne sont pas utilisées simultanément.

```
struct divers                    // définition d'une structure
{
   char Data1;
   long Data2
   int Data3;
};

union MonUnion                   // déclaration de l'union
{
   struct Divers MaStruct;       // La structure et le tableau
   int MonTab[8];                // sont aux même adresses
};
```

26.3 Accès à un élément de l'union.

On accède à un élément d'une union de la même manière que l'on accède à un élément d'une structure. Exemple :

```
char tempo,tempo2;               // variables temporaires

Tempo = MaStruct.Data1;          // Accès à Data1 en temps qu'élément
                                 // d'une structure
Tempo2 = MonTab[0];              // Accès à la même donnée par
                                 // un élément du tableau
                                 // tempo et tempo2 ont été affectés
                                 // avec la même donnée.
```

26.4 Accès aux bits d'un mot.

Si on déclare une union entre un integer et une structure de bits, on a :

- La possibilité de manipuler les bits
- La possibilité de faire des opérations sur le mot.

Exemple :

```
struct status              // définition d'une structure de bits
{
   unsigned Alarm : 1;     // alarm bit
   unsigned WD_AL : 1;     // watch dog alarm
   unsigned AlCnt : 5;     // compteur d'alarme
};                         //

union MonUnion             // déclaration de l'union
{
   struct  status StatBits;// Accès aux bits du status général
   int     StatGen;        // Accès au mot status général
};

// utilisation de l'union
#define MAX_CNT_AL    4    // nombre maximum d'alarmes

StatGen = 0                // initialisation des alarmes

if (StatBits.AlCnt > 4)    // si le compter 5 bits > 4
{
   ……                      // traitement de l'alarme
};
```

27 L'allocation dynamique de la mémoire.

Cette méthode permet allouer une zone mémoire temporairement à différentes données. La même zone mémoire va être utilisée par différentes tâches au cours du temps. Lorsque ces données ne seront plus utilisées alors la place sera libérée et pourra être utilisée par d'autres données. L'adresse exacte ou elles seront stockées les données sera connue seulement à l'exécutions. Pour cette raison les données stockées en allocation dynamique doivent être accédée en mode indirect par des pointeurs.

27.1 Les dangers de l'allocation dynamique de la mémoire.

L'allocation dynamique est la cause de nombreuses nuits blanches dans les entreprises. Il est difficile de prouver qu'il n'y aura jamais de débordement de la zone mémoire réservée à l'allocation dynamique. Il est fondamental de ne pas oublier de libérer la place précédemment réservée sous peine de catastrophe inéluctable.

27.2 L'opérateur sizeof (taille des données)

Il est indispensable avant d'allouer la mémoire dynamique de connaître la taille des données à stocker. L'opérateur sizeof permet de connaître cette taille afin d'allouer la mémoire nécessaire. Exemple :

```
tempo int;                  // variable temporaire

struct str                  // définition d'une structure
{
   unsigned int data1;      //
   unsigned long data2;     //
   unsigned long data3;     //
};                          //

#define SIZE    10          // nombre maximum d'alarmes

struct str[SIZE]            // tableau de structure à mettre
                            // en allocation dynamique.

tempo = sizeof(str)         // tempo est affecté avec la taille
                            // du tableau de structure
```

27.3 L'allocation de mémoire dynamique malloc (stdlib.h)

La fonction malloc permet la réservation de la mémoire dynamique. Le paramètre est la taille nécessaire. Exemple :

```
struct str                      // définition d'une structure
   {
   unsigned int data1;          //
   unsigned long data2;         //
   };                           //

struct str MaStr;               // structure à mettre
                                // en allocation dynamique.
struct str * MaStrPtr;          // pointeur sur la structure

MaStrPtr = malloc(sizeof(str))  // la mémoire dynamique est allouée
                                // le pointeur est initialisé.
```

27.4 Utilisation de la mémoire dynamique.

Notre pointeur à été initialisé par la fonction malloc avec l'adresse de notre structure en allocation dynamique. L'accès aux données est effectué par adressage indirect par le pointeur. Voir le chapitre sur les pointeurs.

27.5 La libération de la mémoire dynamique free (stdlib.h)

Dès que les données en mémoire dynamique ne sont plus utiles il est indispensable de libérer la place sous peine de gros problèmes.

```
struct str                        // définition d'une structure
   {
   unsigned int data1;            //
   unsigned long data2;           //
   };                             //

struct str MaStr;                 // structure à mettre
                                  // en allocation dynamique.
struct str * MaStrPtr;            // pointeur sur la structure

MaStrPtr = malloc (sizeof(str)) // la mémoire dynamique est allouée
                                  // le pointeur est initialisé.
     //     utilisation de la mémoire dynamique

     //     fin de l'utilisation de la mémoire dynamique
free(MaStrPtr)                    // libération de la mémoire
```

28 Les fichiers

Un fichier est un ensemble de données stockées sur un support autre que la mémoire centrale. Il est en général situé sur un disque, une disquette, une bande, carte mémoire PCMCIA etc. L'information contenue dans un fichier est en général conservée de manière permanente (pas de perte d'information en l'absence d'alimentation).

28.1 Les types généraux de fichiers

Il existe des fichiers à accès séquentiel et des fichiers à accès direct.
Un fichier à accès séquentiels contient des informations de tailles différentes. La position de chaque élément du fichier est inconnue. Pour accéder à un élément du fichier il faut lire tous les éléments précédents.
Un fichier à accès direct contient des informations de tailles identiques. La position de chaque élément du fichier est connue. On peut accéder directement à un élément du fichier, par saut à l'élément recherché.

28.2 L'accès direct et séquentiel en langage C

En C on ne fait pas de distinction entre les fichiers à accès direct et les fichiers à accès séquentiel. En C tous les fichiers permettent l'accès séquentiel et direct.

28.3 Déclaration d'un pointeur de fichier

On déclare en fait un pointeur sur un enregistrement d'un fichier.

```
FILE *MonFichier;     // déclaration d'un pointeur sur un
                      // enregistrement d'un fichier
```

Remarque : Les déclarations des fichiers doivent précéder les variables.

28.4 Ouverture d'un fichier, fonction fopen( )

Le pointeur précédemment déclaré est associé à un fichier sur disque.

```
/*****************************************************************
FONCTION   : fopen
BUT        : Ouvre un fichier selon un mode désiré
EN ENTREE  : pointeur sur le fichier
           : Mode de l'ouverture.
EN SORTIE  : != 0x00  => pointeur sur le fichier
             = 0x00  => erreur
*****************************************************************/
int fopen(FILE *FilePtr, const char * mode)     // Prototype
```

28.4.1 Les mode d'ouverture d'un fichier

"r"	read	Ouvre un fichier texte en lecture seule. (le fichier doit exister)
"w"	write	Crée ou efface le fichier texte pour l'écriture
"a"	append	Ouvre ou crée le fichier, pointe la fin de fichier pour ajouter à la fin.(pas d'effacement)
"r+"	read+	Ouverture de fichiers texte pour mise à jour (lecture/écriture)
"w+	write+	Crée ou efface un fichiers texte pour mise à jour (Lecture/écriture)
"a+"	append+	Ouvre ou crée un fichier texte et pointe la fin pour la mise à jour (lecture/écriture)

28.5 Fermeture d'un fichier (fclose)

```
/*****************************************************************
FONCTION   : fclose
BUT        : Ferme le fichier
EN ENTREE  : pointeur sur le fichier
EN SORTIE  : 0x00 => OK
             EOF  => erreur
*****************************************************************/
int fclose(FILE *FilePtr)            // function prototype
```

Exemple : Ouverture d'un fichier en lecture puis en lecture/écriture puis fermeture du fichier.

```
int ErrCode;                      // Code d'erreur éventuelle

FILE *MonFichier;                 // déclaration d'un pointeur sur un
                                  // enregistrement d'un fichier

Erreur = fopen(MonFichier, "r")  // Ouverture du fichier en lecture
Erreur = fclose(MonFichier)      // Fermeture du fichier

Erreur = fopen(MonFichier, "w+")// Ouverture en lecture/écriture
Erreur = fclose(MonFichier)      // Fermeture du fichier
```

28.6 Implémentation avec distinction des fichier texte ou binaire

Pour certaines implémentation les fichiers texte sont différenciés des fichiers binaires. Dans ce cas une lettre est ajoutée, t pour les fichiers textes et b pour les fichiers binaires. On a alors :

Texte	Binaire
"rt"	"rb"
"wt"	"wb"
"at"	"ab"
"wt+"	"wb+"
"rt+	"rb+
"at+"	"ab+"

28.7 Les fichiers ASCII (texte)

Ce type de fichier ne contient une suite de caractères ASCII. Chaque enregistrement est terminé par les caractères ASCII CR LF. On distingue les fichiers ASCII uniquement par leur contenu.

28.8 Entrée sortie d'un caractère

Les fonctions d'entrée/sortie permettent la conversion des caractères pendant l'écriture ou la lecture du fichier.

28.8.1 Ecrit un caractère dans un fichier

```
/*****************************************************************
FONCTION   : fputc
BUT        : Ecrit un caractère dans le fichier
EN ENTREE  : pointeur sur le fichier
EN SORTIE  : Retourne le caractère écrit ou EOF si erreur
*****************************************************************/
int fputc(int c, FILE * FilePtr)                //function prototype
```

28.8.2 Lit un caractère dans un fichier

```
/*****************************************************************
FONCTION   : fgetc
BUT        : Lit le caractère pointé dans le fichier
EN ENTREE  : pointeur sur le fichier
EN SORTIE  : Retourne le caractère lu
*****************************************************************/
int fgetc(FILE * MonFichierPtr)       // Prototype de la fonction
```

28.9 Entrée sortie d'un string

28.9.1 Ecrit une chaîne de caractères (string) dans un fichier

```
/*****************************************************************
FONCTION   : fputs
BUT        : Ecrit une chaîne de caractères dans le fichier
EN ENTREE  : chaîne de caractères
           : pointeur sur le fichier
EN SORTIE  : en cas d'erreur le caractère EOF, sans int positif.
*****************************************************************/
int fputs("Hello", FILE * FilePtr)              //function prototype
```

28.9.2 Lit une chaîne de caractères d'un fichier

```
/*****************************************************************
FONCTION   : fgets
BUT        : Lit n caractères pointés dans le fichier
EN ENTREE  : Pointeur de destination
           : Nombre de caractère à lire
           : pointeur sur le fichier
EN SORTIE  : 0x00 en cas d'erreur due à la fin de fichier, ou le ptr
*****************************************************************/
 int fgets(char *ch, int n, FILE * FilePtr)     //function prototype
```

28.10 Entrée sortie formatée

Ces entrées sorties formatées fonctionnent de manière similaire aux fonctions scanf et printf.
Voir le chapitre des entrées/sorties formatées.

28.10.1 La fonction fscanf

Lit un stream (fichier) puis effectue une conversion éventuelle.

```
/*****************************************************************
FONCTION   : fscanf
BUT        : Lit les caractère pointé avec conversion
ENTREE_1   : pointeur sur le fichier
ENTREE_X   : Pour les autres arguments voir la fonction scanf
EN SORTIE  : Retourne le nombre de valeurs lues
             ou EOF si erreur
*****************************************************************/
int fscanf(FILE *FilePtr,……)      //
```

28.10.2 La fonction fprintf

Effectue une conversion éventuelle puis écrit des caractères dans un stream (fichier).

```
/*****************************************************************
FONCTION   : fprintf
BUT        : Ecrit dans un fichier
ENTREE_1   : pointeur sur le fichier
ENTREE_X   : Pour les autres arguments voir la fonction printf
EN SORTIE  : Retourne le nombre de caractère écrits
             En cas d'erreur une valeur négative
*****************************************************************/
int fprintf(FILE *FilePtr,……)        //
```

Exemple:

28.11 Les fichiers prédéfinis.

Il est possible de considérer un périphérique comme un fichier texte. On peut par exemple considérer l'imprimante ou l'écran comme des fichiers de sortie, le clavier comme un fichier d'entrée. Il sera alors possible de copier un fichier vers l'imprimante ou à l'écran.
Le nom de ces pseudo-fichiers texte sont les suivants :

```
stdin          // Entrée par défault (clavier)
stdout         // Sortie par défault (CRT)
strerr         // Sortie de l'affichage d'erreurs (écran)
stdaux         // Unité auxiliaire.
stdprt         // Printer
```

28.12 Ecriture et lecture de blocs binaire, accès direct

28.12.1 Ecriture binaire dans un fichier (fwrite)

```
/*****************************************************************
FONCTION   : fwrite
BUT        : Ecrit dans un fichier les données pointées
EN ENTREE  : pointeur sur les données à écrire
           : Taille du bloc (ou de la données à écrires).
           : Nombre de blocs (ou de données à écrire)
           : Fichier de destination
EN SORTIE  : Le nombre d'objets écrits
*****************************************************************/
int fwrite (&n, sizeof(n), nbr, FilePtr)   // Prototype
```

Exemple : Ouverture d'un fichier en écriture avec effacement puis écriture et fermeture.

```
const char[5] MonString = {Essai};    // constante à écrire
int Erreur;                           //Erreur éventuelle

FILE *MonFichier;                     // déclaration du pointeur sur
                                      // l'enregistrement du fichier

Erreur = fopen(MonFichier, "w")       // Ouverture en écriture
fwrite(&MonString,SizeOff(MonString), 1, MonFichier); // écriture
Erreur = fclose(MonFichier)           // Fermeture du fichier
```

28.12.2 Lecture d'un fichier (fread)

```
/*****************************************************************
FONCTION   : fread
BUT        : Lit le fichier, stock les données à l'adresse prévue
EN ENTREE  : pointeur sur les données à lire
           : Taille du bloc (ou des données à lire).
           : Nombre de bloc (ou de données à lire)
           : Fichier de destination
EN SORTIE  : Le nombre d'objets lus
*****************************************************************/
int fread (&n, sizeof(n), nbr, FilePtr)         // Prototype
```

Exemple : Ouverture d'un fichier en lecture puis lecture et fermeture.

```
char[5] MonString;                    // destination de la lecture
int Erreur;                           //Erreur éventuelle

FILE *MonFichier;                     // déclaration du pointeur sur
                                      // l'enregistrement du fichier

Erreur = fopen(MonFichier, "r")       // Ouverture en lecture
fread(&MonString,SizeOff(MonString), 1, MonFichier); // écriture
Erreur = fclose(MonFichier)           // Fermeture
```

28.13 Accès direct des fichiers

Tous les fichiers permettent l'accès direct. La fonction seek va nous permettre de sélectionner l'enregistrement désiré.

28.13.1 Sélection d'un enregistrement.

Sélection d'un enregistrement. La référence du déplacement peut être choisie par rapport au début du fichier, à la fin du fichier ou à la position courante du pointeur dans le fichier.

```
/*****************************************************************
FONCTION   : fseek
BUT        : Sélectionne un enregistrement du fichier
ENTREE 1   : Nom du pointeur de fichier (stream)
ENTREE 2   : déplacement (offset) selon ENTREE 3
ENTREE 3   : SEEK_SET (0) => ENTREE 2 = offset du début du fichier
           : SEEK_CUR (1) => ENTREE 2 = offset relatif à la
                                        position courante
           : SEEK:END (2) => ENTREE 2 = offset de la fin du fichier
EN SORTIE  : = 0      => OK
           : != 0     => erreur
*****************************************************************/
int fseek(FILE *stream, nbr, SEEK_SET)              //function
prototype
```

Exemple : Lecture de l'enregistrement 5 à partir du début du fichier.

```
char[5] MonString;                    // Destination
int Erreur;                           // Erreur éventuelle

FILE *MonFichier;                     // déclaration du pointeur sur
                                      // l'enregistrement du fichier
Erreur = fopen(MonFichier, "r")       // Ouverture en lecture
fseek(MonFichier, 4, SEEK_SET)        // sélection du record
fread(&MonString,SizeOff(MonString), 1, MonFichier); // écriture
Erreur = fclose(MonFichier)           // Fermeture du fichier
```

28.13.2 Test de fin de fichier

```
/*****************************************************************
FONCTION   : feof
BUT        : Test la fin du fichier
ENTREE 1   : Nom du pointeur de fichier (stream)
EN SORTIE  : = 0      => pas de fin
           : != 0     => fin de fichier
*****************************************************************/
int feof(FILE *stream);                          //function prototype
```

28.13.3 Position du pointeur dans un fichier

```
/*****************************************************************
FONCTION   : ftell
BUT        : Renvoie la position du pointeur dans un fichier
ENTREE 1   : Nom du pointeur de fichier
EN SORTIE  : Position du pointeur sur l'enregistrement
           : -1 => erreur
*****************************************************************/
long ftell (FILE *stream);                       //function prototype
```

29 Le typedef

La directive typedef permet de personnaliser des données basées sur un type existant.
Cette méthode permet de créer des synonymes des types existant.
Exemple :

```
typedef enum
{
   LUNDI,
   MARDI,
   MERCREDI,
   JEUDI,
   VENDREDI,
   SAMEDI,
   DIMANCHE
} Day;

Day jour;            // on a créé une variable de type day de nom jour
                     // pouvant prendre les valeurs Lundi, Mardi, etc.
                     // En réalité Lundi vaut 0, Dimanche vaut 6
```

Autre exemple:

```
typedef struct
{
   int Heure,
   int Minute,
   int Seconde,
} Time;

typedef struct
{
   int jour,
   int Mois,
   int Année,
} Date;

typedef struct
{
   Time,
   Date,
} temps;

temps RealTime;      // on a créé une variable de type temps
                     // de nom RealTime
                     // temps est un "nouveau type" qui est une
                     // structure de structure.
```

30 Les pointeurs de fonctions

L'appel d'une fonction par l'intermédiaire d'un pointeur est une possibilité intéressante. Elle permet par exemple de modifier une partie du programme puis de mettre à jour les appels des fonctions en modifiant les pointeurs vers ces nouvelles fonctions. Il est aussi possible d'effectuer des modifications de comportement du programme en modifiant les pointeurs vers les fonctions et donc de changer les fonctions appelées.

Exemple :

```
int result            // résultat
int (*adf)(void)      // déclaration d'un pointeur de fonction
                      // avec un int en sortie et rien en entrée.

int fct1 (void)       // Prototype de la fonction 1
int fct2 (void)       // Prototype de la fonction 2

// Ces deux fonctions (fct1,fct2) doivent avoir les mêmes paramètres
// mais peuvent effectuer des tâches différentes.

/* Il est possible d'appeler l'une ou l'autre fonction par un
   appel indirect par échange de l'adresse de la fonction stockée
   dans le pointeur de fonction */

// Affectation du pointeur de fonction

adf = fct1;           // le pointeur de fonction est affecté avec
                      // l'adresse de la fonction fct1

result = *adf(void)   // appel de la fonction fct1

adf = fct2;           // le pointeur de fonction est affecté avec
                      // l'adresse de la fonction fct2

result = *adf(void)   // appel de la fonction fct2

/* la même ligne peut appeler deux fontions différentes suivant
   le contenu du pointeur de fonction */
```

30.1 Exemple avec un tableau de pointeurs de fonctions

La modification d'un index permet de choisir la fonction appelée.

Exemple :

```
int result            // résultat
int i                 // index

int fct1 (void)       // Prototype de la fonction 1
int fct2 (void)       // Prototype de la fonction 2
int fct3 (void)       // Prototype de la fonction 3

int (*adf[3])(void) = {fct1, fct2,fct3};
                      // tableau de pointeurs de fonctions
                      // initialisé avec les adresses des fonctions

scanf(%d, &i)         // saisie de l'index (0 à 2)
result = (*adf[i])()  // appel de la fonction pointée

/* la fonction appelée dépend de l'index i
     si i = 0 => appel de la fonction fct1
     si i = 2 => appel de la fonction fct3 */
```

30.2 Passage d'un pointeur de fonction en paramètre à une fonction

Une fonction peut avoir comme paramètre d'entrée un pointeur de fonction. Elle pourra elle même appeler des fonctions différentes selon le pointeur de fonction reçu.

Exemple :

```
int result            // résultat
int i                 // index

int fct1 (void)       // Prototype de la fonction 1
int fct2 (void)       // Prototype de la fonction 2
int fct3 (void)       // Prototype de la fonction 3

int MaFct (int(*f)(void))  // Prototype de la fonction MaFct
                           // avec l pntr de fonction f en paramètre

/* la fonction pointée par f sera appelée depuis la fonction MaFct

résult = MaFct(fct2); // appel avec le pntr de fonction en paramètre.

/* la fonction appelée depuis MaFct dépend du pointeur de
   fonction reçu en paramètre. */
```

NUL 0 0x00	DLE 16 0x10	SP 32 0x20	0 48 0x30	@ 64 0x40	P 80 0x50	' 96 0x60	p 112 0x70
SOH 1 0x01	DC1 17 0x11	! 33 0x21	1 49 0x31	A 65 0x41	Q 81 0x51	a 97 0x61	q 113 0x71
STX 2 0x02	DC2 18 0x12	" 34 0x22	2 50 0x32	B 66 0x42	R 82 0x52	b 98 0x62	r 114 0x72
ETX 3 0x03	DC3 19 0x13	# 35 0x23	3 51 0x33	C 67 0x43	S 83 0x53	c 99 0x63	s 115 0x73
EOT 4 0x04	DC4 20 0x14	$ 36 0x24	4 52 0x34	D 68 0x44	T 84 0x54	d 100 0x64	t 116 0x74
ENQ 5 0x05	NAK 21 0x15	% 37 0x25	5 53 0x35	E 69 0x45	U 85 0x55	e 101 0x61	u 117 0x75
ACK 6 0x06	SYN 22 0x16	& 38 0x26	6 54 0x36	F 70 0x46	V 86 0x56	f 102 0x62	v 118 0x76
BEL 7 0x07	ETB 23 0x17	' 39 0x27	7 55 0x37	G 71 0x47	W 87 0x57	g 103 0x63	w 119 0x77
BS 8 0x08	CAN 24 0x18	(40 0x28	8 56 0x38	H 72 0x48	X 88 0x58	h 104 0x64	x 120 0x78
HT 9 0x09	EM 25 0x19	) 41 0x29	9 57 0x39	I 73 0x49	Y 89 0x59	i 105 0x65	y 121 0x79
LF 10 0x0A	SUB 26 0x1A	* 42 0x2A	: 58 0x3A	J 74 0x4A	Z 90 0x5A	j 106 0x6A	z 122 0x7A
VT 11 0x0B	ESC 27 0x1B	+ 43 0x2B	; 59 0x3B	K 75 0x4B	[91 0x5B	k 107 0x6B	{ 123 0x7B
FF 12 0x0C	FS 28 0x1C	, 44 0x2C	< 60 0x3C	L 76 0x4C	\ 92 0x5C	l 108 0x6C	¦ 124 0x7C
CR 13 0x0D	GS 29 0x1D	- 45 0x2D	= 61 0X3D	M 77 0x4D	] 93 0x5D	m 109 0x6D	} 125 0x7D
SO 14 0x0E	RS 30 0x1E	. 46 0x2E	> 62 0x3E	N 78 0x4E	^ 94 0x5E	n 110 0x6E	~ 126 0x7E
SI 15 0x0F	US 31 0x1F	/ 47 0x2F	? 63 0x3F	O 79 0x4F	_ 95 0x5F	o 111 0x6F	DEL 127 0x7F