Notes de cours "Introduction au langage C"

Philippe WAILLE (UFR IMA, université Joseph Fourier)

Septembre 2012

Table des matières

0

1 Types de base

Nom	Norme C	Nom abrégé	Remarque
Entiers relatifs			
int			
short int		short	
long int		long	
long long int	c99	long long	
signed char	c99		char pour calcul entier
Entiers naturelss			
unsigned int			
unsigned short int		unsigned short	
unsigned long int		unsigned long	
unsigned long long int	c99	unsigned long long	
unsigned char	c99		char pour calcul entier
Nombres à virgule flottante			
float			
double			precision ++
long double			precision ++ ++ (rare)
Types entiers pour représenter les caractères			
char			ASCII, iso_latin1, utf8
whar_t	c99		utf32
Booléens			
_Bool ou bool	c99		
typedef int bool	$\neq$ c99		stockage dans int
Chaînes de caractères			
char []			[] et '\0' en fin
wchar_t []			
Autres types			
void			Absence de type
T *			Adresse/pointeur de type T

TABLE 1 – Principaux types C

L'attribut c99 indique les types récement introduits dans la dernière norme de C.

Typedef permet de définir de nouveaux types à partir de types C de base, notamment pour déclarer des structures de données complexes.

```
float f (int x) {
```

```
  return (float) x * 1.25;
}
typedef float func_int_to_float (int);   /* nommer ce type de fonction */
fun_int_to_float *pt_func = fsomme;      /* un pointeur dessus */
```

2 Taille des types et écriture des constantes numériques

Préfixes de base de numération entière :
- aucun (décimal) : $64 = 64_{10}$
- 0 (octal) : $0102 = 66_{10}$ $(1 * 8 * 8 + 2)$
- 0x (hexadécimal) : $0x102 = 258_{10}$ $1 * 16 * 16 + 2)$

Tailles classiques sur une machine 64 bits		
Type ou attribut	Taille en bits (t)	octets (sizeof)
char	8	1
wchar_t	32	4
short	16	2
int	32	4
long	32	4
long long	64	8
float	32	4
double	64	8
long double	128	16
T *	64	8

3 Représentation des caractères

Stockage des caractères : par son numéro de code dans un entier $\geq$ taille_codage. Les caractères sont habituellement codés dans le type char en codage ASCII ou iso _xxxx. Le type wchar_t est utlisé avec le codage UTF32. Le type char sert aussi en cas de représentation en codage utf8.

Char et wchar_t sont assimilables à des types entiers. Le type char est souvent équivalent à signed char, parfois à unsigned char.

Un caractère entre quotes (') est la constante entière (de type int) correspondant à son code.

caractère ASCII	hexa	octal	notation C	commentaire
b	\x62	\076	'b'	peut aussi s'écrire 0x62, 076 ou 98
'	\x27	\047	'\''	
"	\x22	\042	'\"'	
\	\x5c	\0134	'\\'	
"line feed"	\x0a	\012	'\n'	(passage à la ligne)
"carriage return"	\x0d	\015	'\r'	(retour en début de ligne)
tabulation horiz.	\x09	\011	'\t'	
"backspace"	\x08	\010	'\b'	(retour en arrière d'un caractère),
"form feed"	\x0c	\014	'\f'	(saut de page)
"nul"	\x0	\00	'\0'	sans effet (fin de chaîne)

4 Déclaration de variable

Le type d'une variable permet de déterminer le nombre d'octets occupés par son contenu et la manière de les interpréter. Une **déclaration ordinaire** de variable **définit le type**, **réserve de la mémoire** pour stocker la variable, et **spécifie** éventuellement son **contenu initial** au démarrage du programme.

```
long long1, long2;       /* valeur initiale implicite : 0 */
long l = 0x1234567L;     /* avec valeur initiale au lancement de l'exécution */
unsigned short s;
double pi = 3.14;
char c;
char b1 = 'b', b2 = 0x62, b3 = 076, b4 = 98;   /* valeurs initiales : 'b' */
```

5 Constantes symboliques et énumération

Avant le compilateur proprement dit, le fichier à compiler est passé au préprocesseur qui gère les directives #**xxx**. La directive #**define** permet de déclarer des constantes symboliques : elle déclenche une substitution de texte avant la phase de compilation.

```
#define CTE1 3.0
#define CTE2 (4.1+9.3)           /* après passage préprocesseur */

x = (y + CTE1) * (CTE2 +1);       x = (y + 3.0) * ((4.1+9.3) +1);
z = CTE2;                         z = (4.1+9.3);
```

Le mot-clé **enum** permet de typer un sous-ensemble fini de valeurs entières nommées. La suite de noms de valeurs est associée à la suite croissante des entiers naturels (l'utilisation de = permet de forcer une association précise).

```
typedef enum bool {FALSE=0, TRUE=1};  // redondant en C99

// Ce code
enum couleur {NOIR, ROUGE, VERT, JAUNE,
              BLANC = 7, BLEU, ORANGE=13};
enum couleur ma_couleur;

// équivaut à
#define NOIR  0
#define ROUGE 1
#define VERT  2
#define JAUNE 4
#define BLANC 7
#define BLEU  8
#define ORANGE 13
int ma_couleur;

// les 2 méthodes permettent d'écrire
ma_couleur = VERT;
```

6 Gestion des booléens

Le type _Bool (C99)

Le type **_Bool** est ajouté par la norme C99. Complètant la famille des types entiers, il se limite au sous-ensemble de valeurs 0,1. Lors d'une affectation d'un autre type d'entier à un _Bool, la règle de conversion (interprétation des entiers comme booléens) est implicitement appliquée :

- Entier 0 → booléen 0 (interprétée comme faux)
- Entier ≠ 0 → booléen 1 (interprétée comme vrai)

Opérateur booléens

Les opérateurs booléens **!** (négation) , **&&** (ET), || (OU)
et de comparaison : $\mathbf{x} < \mathbf{y}$, $\mathbf{x} <= \mathbf{y}$, $\mathbf{x} > \mathbf{y}$, $\mathbf{x} >= \mathbf{y}$,
$\mathbf{x} == \mathbf{y}$ (égalité), $\mathbf{x} \mathbf{!}= \mathbf{y}$ (non égalité)

- acceptent tout type d'opérande entier (aussi bien int que _Bool)
- retournent une valeur entière booléenne 0 ou 1.

Utilisation de int en l'absence de _Bool (avant norme C99)

Le fonctionnement des opérateurs booléens est identique, mais en l'absence du type _Bool, on utilise le type int pour déclarer les variables à valeurs booléennes.

Définition de bool, true et false

Il souvent pratique de définir ainsi les constantes symboliques true et false :

```
#define true 1    // définitions contenues dans stdbool.h de C99
#define false 0
```

Il est commode de nommer bool les booléens comme dans d'autres langages :

```
#include <stdbool.h>        // C99 : contient typedef _Bool bool;
// ou bien
#include "mybool.h"         // avant C99  : contient typedef int bool;
bool b;
int e = 5;
// Piège du forceur (bool)  !
// Effet de b = (bool) e : affecte 1 ou 5 à b selon typedef utilisé
// Ecrire à la place b = (e != 0)
```

7 Expression conditionnelle

Syntaxe : *expression_ condition* **?** *expression_ alors* **:** *expression_ sinon*

L'expression *expression_ condition* est évaluée. Selon sa valeur l'expression conditionnelle retourne la valeur de *expression_ alors* ou celle de *expression_ sinon*.

```
troismax = 3 *((x < y) ? y : x);

/* code équivalent */
if (x < y)
```

```
  max = y;
else
  max = x;
troismax = 3 * max;
```

8 Les opérateurs de calcul usuels

Opération	ADA	C	Commentaires
addition	+	+	
soustraction	-	-	
multiplication	*	*	
div : quotient	/	/	entière ou flottante selon type opérandes
div : reste	rem,mod	%	
puissance	**		→ appel de fonction log/exp
valeur absolue	abs		→ appel de fonction
non logique	not	!	1 booléen par entier, ne pas confondre avec le not bit à bit (~)
ou logique	or else	||	Attention : **double** barre
et logique	and then	&&	Attention : **double** &
ouex logique	xor		→ ((!a && b) || (a && !b))
inférieur strict	<	<	retourne entier 1 pour vrai
inférieur	< =	<=	
supérieur strict	>	>	
supérieur	> =	>=	
égalité	=	==	Attention : **double** égal
inégalité	/=	!=	

```
float x,y;
x = 5.0 / 2.0;                                  /* x = 2.5 */
y = (float) ((int) 5.0 / (int) 2.0);            /* y = 2.0 */
```

9 Instructions élémentaires : affectation, si, tant que, répéter

variable = **expression** ; **/* Pas de deux points avant = */**
while (condition) instruction ; **/* Pas de ; après la condition */**
do instruction **while** (condition) ;
if (condition) instruction_alors **else** instruction_sinon
if (condition) instruction_alors **/* Pas de then ni de endif */**

N'importe quelle expression est utilisable comme condition et il n'y a ni **then** ni **elsif** ni **endif** et pas de deux points avant le égal dans la syntaxe C.

L'exécution du corps de **do** et **while** est répétée si la condition est vraie (condition de continuation). Dans la boucle **do**, le corps est exécuté au moins une fois.

x = y est une **comparaison en ADA** et une **affectation en C ⇒ Piège !**

```c
/* pgcd en C  */
while (x != y)
  if (x == y)
    printf ("termine\n");
  else if (x > y)
    x = x - y;
  else
    y = y - x;
```

```
-- pgcd en ADA
while x /= y loop
  if  (x = y)
    Put_Line("termine"); New_Line;
  elsif (x > y) then
    x := x - y;
  else
    y := y - x;
  end if;
end loop;
```

```c
/* lire et ignorer les A */
do
  c = getc (fichier);
while (c == 'A');
```

A quel if correspond le else ? $\Rightarrow$ par convention le if le plus proche : if (x $>$ y).

10 Blocs d'instructions et d'expressions

Une instruction composée est obtenue en regroupant un ensemble d'instructions dans un bloc délimité par des accolades (au lieu de **begin ... end** en ADA ou PASCAL).

Le début du bloc peut contenir des définitions de variables locales au bloc d'instructions (de préférence uniquement dans la déclaration des fonctions). Le bloc peut être vide.

Utilisations typiques :
- corps d'une boucle **do** ou **while**
- branche alors ou sinon d'un **if**
- corps d'une définition de fonction

Une expression composée est une liste d'expressions séparées par des virgules. La valeur retournée par cette expression est celle de la dernière expression (la plus à droite). Utilisation : boucles for à plusieurs variables de boucles.

```c
/* instruction composée */
if (a_permuter) {
  int tampon;
  tampon = x;
  x = y;
  y = tampon;
} else {
  x = 0;
}
```

```c
/* expression composée */
y = (z=3),(x=4), 3+3;

/* et code équivalent */
z=3;
x=4;
y=6;
```

11 L'affectation C est une expression !

L'affectation a un statut très spécial dans le langage C.

L'affectation **var** = **expression_droite** est considérée comme une expression dont la valeur est celle de *expression_ droite*, et qui a pour effet de bord de copier cette valeur dans *var*.

```
/* Copier jusqu'a fin du fichier (EOF) */
/* Exemple classique d'utilisation     */

while ((c = getc (entree)) != EOF)
  putc (c, sortie);
```

```
/* Code do while equivalent */
do {
   c = getc (entree);
   if (c != EOF) putc (c,sortie);
} while (c != EOF);
```

```
/* Eviter ce genre de code */
x = (y = (z = 5) *2 ) + 3;
```

```
z = 5;
y = z * 2;
x = y + 3;
```

```
if ((x=y) != 3) x++;
```

```
x = y;
if (x != 3) x++;
```

12 Pièges : instruction vide et affectation au lieu de comparaison

Une expression suivie d'un point-virgule en constitue une instruction. Un bloc d'instructions vide constitue une instruction (vide). Un point-virgule seul en constitue une aussi ⇒ **Piège !**.

Autre piège classique : écrire une **affectation** (simple égal) **au lieu d'une comparaison** (double égal) dans une condition. Cette erreur classique ne génère pas d'erreur de syntaxe à la compilation.

En l'absence de type booléen, la syntaxe du langage C permet d'utiliser n'importe quelle expression entière comme condition [1] or l'affectation en C est une expression [2], donc son utilisation comme condition est légale.

```
while (x < y); {  /* ; -> corps vide ! */
  x++;
  y--;
}

do
  y++;
while (x = y); /* =  -> affectation ! */
               /* au lieu de comparaison */
```

```
while (x < y){  /* equivalent */
}
x++;
y--;

do {
   y++; x = y;
} while (x != 0);
```

13 Formes abrégées d'affectation

Formes abrégées d'affectation :
- +=, -= *= /= %= (arithmétique)
- «, » (décalages à gauche et à droite)
- &=, |=, ^= (opérations bit à bit)

```
/* exemples */     /* equivalent */
x -= 3;            x = x + 3;
x <<= 2;           x = x << 2;
x |= 4;            x = x | 4;
```

Opérateurs d'incrémentation ("plusplus") et de décrémentation ("moinsmoins")"
Placés à droite : retournent l'ancienne valeur.
Placés à gauche : retournent la nouvelle valeur.

1. interprétée comme faux si et seulement si sa valeur est zéro : voir section 6
2. Retournant la valeur de son membre droit

```
x++;              x = x + 1;   /* séquence de code équivalente */
x--;              x = x - 1;
y = x--;          y = x; x = x - 1;
y = --x;          x = x - 1; y = x;
```

14 Traduction de parcourant et instruction for

for (initialisation ; condition ; mise_à_jour) corps

Exemple : parcourir en ascendant l'intervalle [3 ... N-1]

```
j = 3;          /* initialisation */
while (j < N) { /* condition */
  j = j + i;    /* Corps */
  s = s + 1;
  j++;          /* Mise à jour */
}
```

```
/*   init   cond   maj */
for (j = 3; j < N; j++) {
  j = j + i;   /* Corps */
  s = s + 1;
}
```

```
/*   <- init    ->  <-cond-> <-   maj      -> */
for (i=0, p = tete; i < max; i++, p = p -> next) {
  somme += p -> valeur;
}
```

```
i = 0;          /* init */
p = tete;
while (i < max) { /* cond */
  somme +=  p -> valeur;
  i++;          /* maj */
  p = p -> next;
}
```

15 Boucles infinies, break et continue

```
while (++x < N) {
  if (stop()) break;
  s = chercher(x);
  if (s==NULL) continue;
  f1(s); f2(s);
}
```

```
while ((++x < N) && !stop()) {
  s = chercher(x);
  if (s != NULL) {
    f1(s); f2(s);
  }
}
```

Une boucle (**for, do, while**) dont la condition est une constante non nulle ne termine pas.
Dans le corps d'une boucle l'exécution de l'instruction
- **break** fait sortir de la boucle.
- **continue** supprime l'exécution du reste du corps et passe au tour de boucle suivant

16 Définition d'une fonction ou d'une procédure

Définition d'une fonction = définition du prototype + définition du corps

Prototype (ou signature) = déclaration du type de la fonction.
- Type de résultat retourné. Si **void** : déclaration d'une procédure.
- Nom de la fonction ou procédure
- Type et nom (le nom peut être omis si le prototype n'est pas accompagné du corps) de chacun des paramètres formels ou **void** (rien dans l'ancienne norme K&R) si la fonction n'a pas de paramètre

Définition du corps :

- Alloue statiquement et initialise la mémoire pour stocker les instructions composant le corps de la fonction.
- Enumère les variables locales de la procédure → allocation dynamique de mémoire pendant l'exécution (gérée par le compilateur).

Les paramètres sont passés par valeur (comme les paramètres **in** en ADA). L'instruction **return exp** termine l'exécution de la fonction et retourne la valeur **exp** à l'appelante.

```
char                              /* définition complète : */
lire_chiffre (void)               /* prototype  +          */
{                                 /* corps de la fonction  */
  char lu;
  do {
     lu = getchar ();
  } while ((lu != EOF) && (lu >= '0') && (lu <= '9'));
  return lu;
}

void
ecrire_y  (unsigned long y);      /* prototype procédure seul */

double
mon_sinus (double x);             /* prototype fonction seul */
```

```
   void                          void                          long
   au_revoir (void)              ecrire_a (int a)              fois_deux (long x)
   {                             {                             {
     printf ("au revoir\n");       printf ("a = %d\n",a);        return x+x;
   }                             }                             }
```

```
   unsigned long                                unsigned long
   facto (unsigned int n)                       facto (unsigned int n)
   {                                            {
     unsigned long result;                        /* variante sans variable locale */
     if (n != 0)                                  if (n !=  0) return  n * facto (n-1);
        result = n * facto (n-1);                 return 1;
     else                                       }
        result = 1;
     return result;
   }
```

17 Directive préprocesseur #include

La directive #include indique au processeur d'inclure le contenu d'un fichier de prototypes (fichier de type header : xxx.h) référencé à partir du répertoire courant ou d'un répertoire standard du système (/usr/include).

```
#include <stdio.h>       /*  répertoire public : /usr/include/stdio.h   */
                         /*  fonctions de bilbiothèques standard (libc) */

#include "monfichier.h" /*  répertoire courant : ./monfichier.h        */
                         /*  fonctions de l'utilisateur                 */
```

18 Modules à compilation séparée, interfaces

Un module définit des variables et des procédures ou fonctions. Elle peuvent être privées (inaccessibles ou inconnues des autres modules) ou partagées : définies (exportées) par un module (avec allocation de mémoire de stockage) et accédées ou appelées (importées) dans un ou plusieurs autre modules. En C, la notion de module correspond à celle de fichier.

Version C

```
/* fichier user_io.h */
void
Write_Int (int value);


/* fichier user_io.c */
#include <stdio.h>


void
Write_Int (int value)
{
  printf ("Voici :");
  printf ("%u",value);
  printf ("\n");
}


/* fichier calcul.h */

extern int coeff;
void Calcul (int val);


/* fichier calcul.c */
#include "user_io.h"
int coeff;
void
Calcul (int val)
{
  Write_Int(val * coeff);
}


/* fichier main.c */
#include "calcul.h"
void
main (void)
{
  coeff=4;
  Calcul (5);
}
```

Version ADA

```
/* fichier user_io.ads */
package User_io is
  procedure Write_Int (Value:in Integer);
end User_Io;

/* fichier user_io.adb */
with Ada.Text_Io;
with Ada.Integer_Text_Io;

package body User_Io is
  procedure Write_Int (Value:in Integer) is
  begin
    Ada.Text_Io.Put("Voici :");
    Ada.Integer_Text_Io.Put(Value);
    Ada.Text_Io.New_Line;
  end Write_Int;
end User_Io;

/* fichier calcul.ads */
package Calcul is
   coeff : Integer;
   procedure Calcul (Val: in Integer);
end Calcul;

/* fichier calcul.adb */
with User_io;

package body Calcul is
  procedure Calcul (Val: in Integer) is
  begin
     User_io.Write_Int(Val * coeff);
  end Calcul;
end Calcul;

/* fichier principal.adb */
with Calcul;

procedure Principal is
begin
     Calcul.coeff := 4;
     Calcul.Calcul (5);
end Principal;
```

Les variables à l'extérieur des corps de fonctions sont exportables (par défaut[3]) ou privées (déclarées avec l'attribut **static**).

L'interface (fichier **m.h**) d'un module **m**, incluse dans tous les modules qui utilisent les variables et fontions exportées par le module **m** (fichir **m.c**), permet de spécifier les types des variables et les prototypes des fonctions exportées par le module **m**. Elle regroupe des déclarations de contantes et de types, mais **aucune** déclaration avec **réservation de mémoire** (extern : voir section 30).

Chacun des modules peut être compilé séparémént des autres, mais si un fichier .h est modifié, tous les modules qui l'incluent doivent être recompilés.

19 Etapes de compilation, bibliothèques, makefile

19.1 Etapes de compilation

La commande de compilation **cc** ou **gcc** enchaîne automatiquement les différentes étapes de traduction pour générer un fichier binaire exécutable à partir des fichiers passés en paramètre. Par convention, la nature du contenu d'un fichier et la prochaine étape de traduction à appliquer sont déduits du suffixe du nom du fichier (tableau 19.1).

Suffixe	Type de contenu	Traitement à appliquer	Résultat	gcc
f.h	source C	à insérer dans f.i ou f.s (lors de ccp de f.c ou f.S)		
f.S	source asm avec #xxx	préprocesseur (cpp)	f.s	-E
f.c	source C avec #xxx		f.i	
f.i	source C sans #xxx	compilateur	f.s	-s
f.s	source asm sans #xxx	assembleur (as)	f.o	-c
f*yyy*.o	objet binaire relogeable	bibliothèques statiques (ar)	lib*nom*.a	
		bibliothèques partagées (ld)	lib*nom*.so	
		édition de liens (ld)	binaire exécutable	

TABLE 2 – Suffixes des fichiers et étapes de compilation (posix)

La colonne gcc indique l'option de gcc à utiliser pour stopper la compilation après une étape donnée (les fichiers résultat des étapes intermédiaires ne sont pas conservés). A titre d'exemple **gcc -c user_io.c** permet de stopper la compilation après la génération du fichier binaire relogeable user_io.o.

Le préprocesseur traite principalement les inclusions et substitutions de texte associées aux directive **#include** et **#define**. Le compilateur interne compile séparément chaque fichier source en fichier binaire relogeable .o ou en fichier source en langage d'assemblage .s.

19.2 Edition de liens et bibliothèques

L'édition de liens fusionne en un unique fichier binaire exécutable l'ensemble des fichiers objets compilés séparément : ceux de l'utilisateur, divers fichiers de code standard[4] (crt*zzz*.o) et la bibliothèque C livrés avec le compilateur, et d'autres bibliothèques éventuelles. Elle permet d'associer à chaque variable ou fonction importée dans un module son adresse (définie dans la table des

3. L'inverse aurait été plus prudent, comme dans le langage d'assemblage (directive .global pour exporter)
4. exécuté lors du démarrage ou de la terminaison, dont initialisations diverses de variables utilisées par certaines fonctions de bibliothèques (exemple : malloc)

user_io.o	$\xleftarrow{\textit{préprocesseur+compilation/assemblage}}$	user_io.h+user_io.c
calcul.o	$\xleftarrow{\textit{préprocesseur+compilation/assemblage}}$	user_io.h+calcul.c
main.o	$\xleftarrow{\textit{préprocesseur+compilation/assemblage}}$	calcul.h+main.c
calcul	$\xleftarrow[\textit{édition de liens (option } -lc)]{\textit{édition de liens (options } -static\ -lc)}$	calcul.o+main.o+user_io.o+crt*zzz*.o+lib*c*.a calcul.o+main.o+user_io.o+crt*zzz*.o+lib*c*.so

TABLE 3 – Principales étapes de compilation du programme calcul (section 18)

symboles du fichier objet relogeable du module exportateur).

Une bibliothèque contient un ensemble de fonctions ou procédures précompilées, offrant divers services utiles (exemple : entrées/sorties print et scanf), qui sont liées aux fichier(s) de type objet binaire relogeable de l'utilisateur. Les suffixes **.a** et **.so** [5] des bibliothèques signifient **archive** (pour l'édition de liens statique) et **shared object** (pour l'édition de liens dynamique).

L'édition de liens **statique** (option -static) fabrique un fichier **exécutable complet**, au prix d'une **duplication du code** des fonctions de bibliothèques dans tous les fichiers binaire exécutables qui les utilisent.

L'édition de liens de liens **dynamique** crée un fichier exécutable incomplet. Elle consiste à **retarder** la liaison avec les fonctions de bibliothèques jusqu'au **début** de chaque **exécution** pour éviter la duplication du code des bibliothèques tant sur le disque que en mémoire centrale (dans un système multiprogrammé).

L'option **-l***nom.suffixe* spécifie le nom d'une bibliothèque (fichier **lib***nom***.a** ou **lib***nom***.so**) à utiliser lors de l'édition de liens. L'option **-L***répertoire* spécifie un répertoire où trouver cette bibliothèque [6].

19.3 Make et Makefile

La commande unix **make** permet d'appliquer un ensemble de règles de (re)compilation décrites dans un fichier **Makefile**.

La commande **make** *cible* crée le fichier *cible* ou le met à jour s'il est devenu obsolète. Pour celà, make va récusivement mettre à jour chacun des fichiers utilisés pour la génération de *cible* (user_io.o, calcul.o et main.o pour make calcul), puis regénérer *cible* si la date de dernière modification d'un de ces derniers n'est pas antérieure à celle de *cible*.

Si la règle associée à une cible à mettre à jour est absente ou incomplète, make essaiera d'utiliser un ensemble variables et de règles génériques par défaut. Voici un exemple de règle générique implicite : f.o dépend de f.c et on le crée avec la commande *${CC} ${CFLAGS} -c f.c*.

Il est également possible de définir des règles de compilation génériques explicites. La dernière règle à droite indique par exemple que tout fichier nom.o dépend des fichiers suivants : nom.c

5. sous unix/posix, .dll pour dynamically linked library sous windows
6. l'éditeur de liens dynamique utilise la variable d'environnement **LD_LIBRARY_PATH** pour rechercher les bibliothèques partagées.

nom.h et global.h. Avec les règles implicites et la définition des variables **CC** et **CCFLAGS**, il est souvent possible d'omettre la commande de génération d'une cible.

```
# Exemple de fichier Makefile
OBJECTS = user_io.o calcul.o main.o

calcul:      ${OBJECTS}
             gcc -o calcul ${OBJECTS}

# | cible:   | dépendances
# v          v
calcul.o:    calcul.c user_io.h
             gcc -Wall -c calcul.c
#            ^
#            |
# Commande pour generer la cible

# recompiler user_io.o si user_io.c
#                  ou user_io.h
#                  est modifié
user_io.o:   user_io.c user_io.h
             gcc -Wall -c user_io.c

# N'utiliser que des TABULATION(s)
# N'utilisez AUCUN caractère EPSACE !
# entre les : et la liste de dépendance
#   < ici   >    ainsi que
main.o:      main.c calcul.h
             gcc -Wall -c main.c
#   < ici   >
#            en debut de ligne
```

```
# Un Makefile plus générique
CC = gcc
CFLAGS = -Wall

OBJECTS = user_io.o calcul.o main.o

calcul:      ${OBJECTS}
             gcc -o calcul ${OBJECTS}

user_io.o:   user_io.c user_io.h

calcul.o:    calcul.c user_io.h

main.o:      main.c calcul.h
```

AS	Nom de l'assembleur
CC	Nom du compilateur C
ASFLAGS	Options à passer à **AS**
CFLAGS	Options à passer à **CC**
$@	Nom du fichier cible
$^	Liste de dépendances
$<	Premier fichier de $^

```
# Une règle générique explicite
%.o:         %.c %.h global.h
             gcc -c -Wall $<
# ou (si CC et CFLAGS définis)
%.o:         %.c %.h global.h
```

20 Adresses : opérateurs & et *

La déclaration de la variable x spécifie que x est un contenant, à savoir sizeof(x) cases mémoire consécutives, numérotées, de un octet (= unité adressable) chacune.

Le numéro d'octet en mémoire est l'adresse. Une adresse est un entier naturel.

L'opérateur **&**, appliqué à un contenant mémoire, en retourne l'adresse.
&x donne l'adresse de stockage (du premier octet) de x.

Opérateur ***** : adresse_octet ⇒ contenant mémoire (*adr signifie Mem[adr])

Les deux opérateurs se neutralisent : ***&x** *est égal à* **x**, **&*a** *est égal à* **a**.

```
#include <stdio.h>

long a = 0x12345678;
char b = 'x';
long c = 0xaabbccdd;    /* adresse multiple de sizeof(long) */
char d = 'a';
```

```
char *pt = &b;          /* pointeur intialisé à l'adresse de b */

void
main () {
   printf ("adresse_de_a :0x%lx contenu de a : 0x%lx\n",(unsigned long) &a,a);
   printf ("adresse_de_b :0x%lx contenu de b : 0x%1c\n",(unsigned long) &b,b);
   printf ("adresse_de_c :0x%lx contenu de c : 0x%1c\n",(unsigned long) &c,c);
   printf ("adresse_de_d :0x%lx contenu de d : 0x%lx\n",(unsigned long) &d,d);
   printf ("adresse_de_pt :0x%lx contenu de pt : 0x%lx\n",(unsigned long) &pt,pt);
}
```

nom du contenant	octets	adresse du contenant	contenu
a	4	0x080496c0	0x12345678
b	1	0x080496c4	0x78 (code ASCII 'x')
c	4	0x080496c8	0xaabbccdd
d	1	0x080496cc	0x61 (code ASCII 'a')
pt	4	0x080496d0	0x080496c4

L'affectation **b** = **d** peut aussi s'écrire ***&b** = ***&d** et signifie :

contenant variable b	$\xleftarrow{\textit{prend pour nouveau contenu 8 bits}}$	contenu de la variable d
Mem[adresse_de_b]	$\xleftarrow{\textit{prend pour nouveau contenu 8 bits}}$	Mem[adresse_de_d]
octet Mem[0x80496c4]	$\xleftarrow{\textit{prend pour nouveau contenu}}$	octet Mem[0x80496cc]

L'affectation **a** = **c** peut aussi s'écrire ***&a** = ***&c** et signifie :

contenant de la variable a	$\xleftarrow{\textit{prend pour nouveau contenu 32 bits}}$	contenu de la variable c
Mem[adr_de_a]	$\xleftarrow{\textit{prend pour nouveau contenu 32 bits}}$	contenu de Mem[adr_de_c]
octet Mem[0x80496c0]	$\xleftarrow{\textit{prend pour nouveau contenu}}$	octet Mem[0x80496c8]
octet Mem[0x80496c1]	$\xleftarrow{\textit{prend pour nouveau contenu}}$	octet Mem[0x80496c9]
octet Mem[0x80496c2]	$\xleftarrow{\textit{prend pour nouveau contenu}}$	octet Mem[0x80496ca]
octet Mem[0x80496c3]	$\xleftarrow{\textit{prend pour nouveau contenu}}$	octet Mem[0x80496cb]

21 Type "adresse de", constantes adresses, pointeurs

Pourquoi utiliser un type spécial et pas un entier naturel ordinaire pour les adresses ?
- pour indiquer combien d'octets transférer lors de l'accès via un pointeur ?
- pour spécifier comment interpréter le contenu de ces octets (flottant, entier) ?
- piur vérifier la cohérence de type : distinction entre adresse et contenu et entre natures de contenu (ne pas affecter une adresse d'entier à un pointeur de float).

Il y a autant de types "adresses de" que de types d'objets dont on peut prendre l'adresse, comment les noter ?
- **access** / **access all** *type_pointé* en ADA.
- *type_pointé* * en C : l'application de l'opérateur * à une expression de ce type rend un objet du type *type_pointé*[7]

7. Les concepteurs du C auraient pu choisir la notation **& type_pointé**, qui a été retenue en C++ pour les

21.1 Version ADA

```
  type ACCESS_INT is access all Integer;

  Entier1 : aliased Integer   := 123;
  Entier2 : Integer;

  Mon_pointeur : ACCESS_INT;

begin
  Mon_pointeur := Entier1'Access;   -- Entier2'Access serait illegal;
  Entier2 := Mon_pointeur.all + 2;  -- Entier2 := Entier1 + 2.
  Mon_pointeur.all = 13;            -- Entier1 := 13;
end  mon_programme;
```

Protection spécifique ADA : seules les variables déclarées **aliased** sont accessibles via un pointeur. Un pointeur contenant la constante **null** ne repère rien.

21.2 Version C

Un pointeur ne repérant rien contient la constante **NULL**[8]. Le type **void *** est utilisé pour stocker des adresses d'objets de n'importe quel type. Il doit être converti en **T *** pour accéder à un objet de type **T**.

```
typedef char * pointeur_de_char;

long entier1 = 123;
long entier2;
char b = 'x';
long *mon_pointeur;            /* initialisé à NULL */

mon_pointeur = & entier1;
entier2 = *mon_pointeur+2;     /* entier2 = entier1+2 (entier2=*&entier1+2) */
*mon_pointeur = 13;            /* entier1 = 13 */

mon_pointeur = &entier2;
(*mon_pointeur)++;             /* entier2 ++ */
...

char c;
pointeur_de_char pcar = &b;    /* repère b dès chargement du programme */
c = *pcar;                     /* c = b  (c = *&b)*/

void *pointe_tout;             /* n'importe quelle sorte d'adresse */
pointe_tout = &entier2;
entier1 = * (long *) pointe_tout;  /* entier1 = entier2 */
/* mais cette affectation est illégale : entier1 = *pointe_tout    */
pointe_tout = &b;
* (char *) pointe_tout = 'y';          /* b = 'y' */
```

références aux objets. La notation **.all** suggère que ADA traite un scalaire comme une structure à un seul champ (dépourvu de nom).

8. la constante **NULL** est généralement définie comme **(void *) 0**.

Ne confondez pas taille d'un pointeur (taille d'une adresse) et taille d'objet repéré :
- sizeof(**char**) $\neq$ sizeof(**char ***)
- sizeof(**long**) $\neq$ sizeof(**char**)
- sizeof(**char ***) = sizeof(**long ***) = sizeof(**void ***)

Rappel : la déclaration d'un pointeur réserve de la place pour stocker l'adresse contenue dans le pointeur, mais pas pour le contenu pointé. N'oubliez pas d'initialiser le pointeur avec l'adresse d'une variable déclarée statiquement ou avec une adresse de bloc retournée par malloc ou calloc.

22 Gestion de paramètres résultat

22.1 version ADA

```
procedure maxi (a : in Integer ; max : in out Integer) is
begin
if (a > max) Then max := a; endif;
end maxi;

-- calcul de max (a,b,c)
maximum := a;
maxi (b,maximum);
maxi (c,maximum);
```

22.2 version C

En C, tous les paramètres sont passés par valeur (équivalent du type in). Si l'appelante passe une expression, l'expression est évaluée et la procédure reçoit le résultat de l'évaluation. Si l'appelante passe une variable, la valeur de l'expression est une copie de la valeur de la variable : la procédure peut modifier cette copie, mais pas la variable de l'appelante.

Il est cependant possible de passer la valeur d'un pointeur, ce qui revient à effectuer un passage de paramètre par adresse. A partir de l'adresse d'une variable, il est possible de lire et de modifier son contenu, ce qui constitue une manière de réaliser l'équivalent des paramètres de type résultat et valeur-résultat (**out** et **in out** de ADA). Notons qu'avec n paramètres de type pointeur, une fonction C peut retourner n+1 résultats.

```
long tab[10];
void
maxi (int a, int *max)
{
   if (a > *max) *max = a;
}

/* appel */
maximum = a:
maxi (b, &maximum);
maxi (c, &maximum);
```

23 Allocation dynamique de mémoire

23.1 Version ADA : new

```
type TAB is array(0..7) of Integer;

type POINTEUR is access Integer;
type POINTEUR_TAB is access TAB;

X: Integer;
Ptr : POINTEUR;
Ptrtab POINTEUR_TAB;
...
Ptr := new Integer;   -- new appelle l'allocateur dynamique de mémoire
Ptrtab := new TAB;
Ptr.all = 3;
Ptrtab(3) = 6;
```

23.2 Version C : malloc/calloc

En langage C, l'allocateur dynamique de mémoire correspond aux fonctions de bibliothèque **malloc** et **calloc** qui réservent de la mémoire et retournent une adresse de type **void ***.

```
#include <malloc.h>
int x, *ptr, *ptrtab;
...
ptr = (int *) malloc(sizeof(int));
ptrab = (int *) calloc (sizeof(int),8); /* allouer tableau de 8 ints */
if ((ptr == NULL) || (ptrtab == NULL)) {
  /* traiter l'erreur */
  }
*ptr = 3;
ptrtab[3] = 6;
free (ptr); free (ptrtab);      /* liberer la mémoire */
```

24 Tableaux à une dimension

24.1 Exemple en ADA

```
Tableau_3_a_7 : array (Integer range 3 ..7) of Integer;
begin
  Tableau_3_a_7 (5) := 120;
  Tableau_3_a_7 (2) := 2;  -- erreur détectée : hors intervalle
  Tableau_3_a_7 (8) := 0;  -- erreur détectée : hors intervalle
end
```

24.2 Tableaux en C

Il n'existe pas de vrai type tableau en C : il est juste possible de réserver unnbloc de mémoire pour stocker pour N éléments contigus (éventuellement avec initialisation), avec les particularités suivantes :

- Pas de choix de l'intervalle des indices : **toujours [0 ... N-1]**.
- Possibilité d'initialisation partielle (valeurs manquantes implicitement initialisées à 0). Possibilité de déduire la taille du tableau du nombre de valeurs initiales.
- Le nom du tableau est la constante adresse de son premier élément : **T équivalent à &(T[0])**.
- **Pas de vérification de validité** de l'indice à l'exécution : ni l'intervalle des indices ni le nombre d'éléments ne sont stockés en mémoire et vérifiés pendant l'exécution.
- Pour passer un tableau à une procédure, il faut généralement passer deux paramètres : le tableau (autrement dit l'adresse de son premier élément) et sa taille. La taille peut éventuellement être omise lorsque la convention suivante est utilisée : un $(n+1)^{ième}$ élément, constante 0 (pour tableau d'entiers) ou NULL (pour un tableau de pointeurs), est utilisé comme marqueur de fin de tableau.

```
#define MAX_TAB 10
#define NB_PUIS2 5
int indice;
unsigned long somme;

int victime = 3;                              /* tabentier[-1] ? */
int tabentier [MAX_TAB];
int premiers [] = {1,2,3,5,7,11};             /* dimension déduite : 6 */
unsigned long puis2[NB_PUIS2] = {1,2,4};      /* initialisation partielle */
char voyelles[6] = {'a','e','i','o','u','y'}; /* voyelles[0] : 'a' */
char  lettre_o = 'o';                         /* voyelles[6] ?     */

void
main (void)
{
  tabentier [3] = 3;
  puis2[3] = 8;
  puis2[4] = 16;
  indice  = -1;
                                /* Utilisation d'indice hors bornes */
  tabentier [indice] = 0;       /* effet probable : victime = 0     */
  voyelles [6] = 'n';           /*                  lettre_o = 'n'  */

  somme = 0;
  for (indice = 0; indice < NB_PUIS2; indice ++)
    somme += puis2[indice];
}
```

Règle de calcul d'adresse : l'élément **tabentier[i]** est stocké à l'adresse de début du tableau (**tabentier**, synonyme de **&(tabentier[0])**), auquel est ajouté l'indice (**i**) implicitement multiplié par la taille d'un élément (**sizeof(int)**).

Noter la présence de l'opérateur **&** appliqué à un élément de tableau (**p = &tableau[0]**), et son absence devant le nom de tableau qui est déjà un pointeur sur son premier élément (**p = tableau**).

25 Chaînes de caractères

En C, les chaînes de caractères sont représentées sous la forme de tableaux de type char. La taille d'un tableau C n'étant pas stockée en mémoire, la chaîne est délimitée par un marqueur de fin de chaîne : un zéro qui est le code ASCII du caractère non affichable appelé NUL [9]. Cette marque de fin de chaîne peut être notée **0** ou **'\ 0'**.

En revanche, le langage C offre une facilité syntaxique d'écriture pour les constantes chaînes de caractères.

```
#include <stdio.h>

/* Plusieurs manières de déclarer des tableaux contenant la chaîne "ok" */
const char ok1[3] = {'o','k','\0'};
const char ok2 [] = {'o','k',0};
const char ok3 [] = "ok";

const char halala [] = "ha_la_la";

unsigned long Nombre_de_a (const char *chaine) {  /* ou (char chaine[]) */
   unsigned long total ;
   const char *p;
   total = 0;
   for (p=chaine; *p != 0; p++)
      if (*p == 'a') total ++;
   return total;
}

void
main () {
   printf ("nombre de a : %lu\n",Nombre_de_a(ok1));
   printf ("nombre de a : %lu\n",Nombre_de_a(halala));
   printf ("nombre de a : %lu\n",Nombre_de_a("constante_chaine"));
}

/* Le dernier printf est equivalent a ce genre de code */
{
   const char nom_fabrique_par_compilateur [] = "constante_chaine";
   printf ("nombre de a : %lu\n",Nombre_de_a(nom_fabrique_par_compilateur));
}
```

26 Arithmétique sur les pointeurs

Soit un pointeur **p** contenant une adresse d'élément de tableau : **p = &t[i]**. Après ajout d'un entier **j** au pointeur **p**, **p** contient l'adresse d'élément de tableau **&t[i+j]**. C'est pourquoi l'entier **j** ajouté à un pointeur est **implicitement multiplié** par la **taille du type** d'objet pointé (on ajoute **j * sizeof(type_pointé)**).

9. ne pas confondre le caractère de nom NUL, représenté par le code ASCII 0 codé sur un octet, avec la constante adresse NULL, habituellement définie comme l'adresse 0 codée sur 4 octets (8 octets sur un processeur 64 bits) .

L'opérateur **[]** n'est qu'une notation : **& t[i] s'écrit aussi t+i** et **t[i] s'écrit aussi *(t+i)**.

On peut aussi comparer deux pointeurs s'ils repèrent des éléments d'un même tableau. La différence entre deux pointeurs ou deux constantes de type adresse est implicitement divisée par la taille du type d'objet pointé pour retourner la différence entre les indices des éléments repérés.

```
#define MAX_TABLEAU 6

int indice;
long somme, produit;
long *p:

long tableau [MAX_TABLEAU] = {0,10,20,30,40,50};
long *ptr1 = & tableau[1]; /* Ces 2 pointeurs repèrent tous les deux */
long *ptr2 = tableau + 1;  /* l'élément tableau[1]                   */

void
main (void)
{
  *ptr1 = 2;          /* tableau [1] = 2;              */
  ptr1 = ptr1 + 4;    /* repère maintenant tableau [5] */
  *ptr1 = 0;          /* tableau [5] = 0               */
  somme = 0;

  for (p = tableau; p < tableau + MAX_TABLEAU; p++)
    somme += *p;
                      /* syntaxe equivalente           */
  for (p = &tableau[0]; p < &tableau[MAX_TABLEAU]; p++)
    somme += *p;

  ptr2 = &(tableau[0]);
  printf ("%d_%d\n", ptr1 - ptr2,
                     (char *) ptr1 - (char *)ptr2);  /* affiche 5_20 */
}
```

27 Application simultanée de * et ++ ou --

Les expressions ***p++**, ***p--**, ***++p** et ***--p** (attention à ne pas en abuser au détriment de la lisibilité des programmes) désignent le contenu en mémoire à l'adresse correspondant à la valeur

- **initiale** de p si l'**opérateur ++ ou -- suit (à droite de)** la variable p
- **modifiée** de p si l'**opérateur ++ ou -- précède (à gauche de)** la variable p

```
variable = *pointeur--;  /* signifie */  variable = *pointeur; pointeur --
variable = *++pointeur;                  pointeur ++; variable = *pointeur;

/* deux copies de chaînes de caractères */
do {c=*source; *dest=c; source++; dest++;} while (c != '\0');
while ((*dest++ = *source++) != '\0') {}
}
```

28 Structures (enregistrements)

La définition de variables enregistrement se fait généralement en deux étapes :

1. Déclaration du type d'enregistrement, précisant les noms et types des membres
2. Définition de variables de ce nouveau type, avec allocation de mémoire.

28.1 Exemple ADA

```
-- declaration d'un nouveau type structure
type POINT_XY is
  record
    X : Integer;
    Y : Integer;
  end record;

-- déclaration du type pointeur de ce genre de structure
type ACCESS_POINT_XY is access POINT_XY;

-- declaration de variables de ce type
Origine     : aliased POINT_XY := (3,7);
Destination : POINT_XY;

Pointeur_point : ACCESS_POINT_XY;
Pointeur2 :      ACCESS_POINT_XY;

-- utilisation
begin
  Origine.X := 4;
  Destination := (7,8);
  Origine := Destination;
  Pointeur_Point := Origine'Access;
  Pointeur_Point.Y := 6;             -- acces via un pointeur
  Pointeur_Point.X := 3;             -- acces via un pointeur
  Pointeur2 := new POINT_XY;         -- allocation dynamique
  Pointeur2.Y = 4;
end ma_procedure
```

28.2 Exemple C

```
/* Déclaration du type struct _point_xy */
struct _point_xy {
  int x;
  int y;
  };

/* Nommage du type struct _point_xy */
typedef struct _point_xy point_xy;

/* Declaration du type pointeur de ce genre de structure */
/* on peut ecrire aussi   typedef point_xy *pt_point_xy */
typedef struct _point_xy *pt_point_xy;
```

```
/* Définition des variables avec allocation statique de mémoire */
struct _point_xy origine = {3,7};
point_xy destination;

/* Deux pointeurs de structure _point_xy */
point_xy *pointeur_point;
pt_point_xy pointeur2;

/* Déclarations combinées */

/* Déclarer le type struct _point_rt et le nommer point_rt */
typedef struct _point_rt {
  float : r;
  float : t;
  } point_rt;

/* Déclarer une variable sans donner de nom au type de struct */
struct {
  char *nom;
  char *prenom;
  } banal = {"Pierre","Martin"};

/* Attention au contraintes d'alignement : sizeof(struct _mal_aligne) = 8 */
struct _mal_aligne {
  long l;     /* sizeof(long) = 4 */
  char c;     /* sizeof(char) = 1 */
  } m1,m2;    /* 3 octets entre m1.c et m2.l pour respecter alignement */

struct _point_xy creer_point_xy (int x, int y)
{
struct _point_xy resultat;
resultat.x = x;
resultat.y = y;
return resultat;
}

void
main (void)
{
  origine.x = 4;                          /* Affecte x de origine      */
  destination = (struct _point_xy) {7,8};  /* Affecte x et y à 7 et 8   */
  origine = destination;                  /* Copie x et y de dest      */
  pointeur_point = &origine;
  (*pointeur_point).y = 6;                 /* acces via le pointeur p   */
  pointeur_point -> x = 3;                 /* p -> x abrégé pour (*p).x */
  pointeur2 = malloc(sizeof(struct _point_xy));  /* alloc dynamique      */
  if (pointeur2 == NULL) { /* traiter l'échec */ }
  pointeur2 -> y = 4;
}
```

29 Structures pour listes chaînées

Les listes chaînées sont construites avec des structures dont (au moins un) un membre est un pointeur de structure de même type.

```
Version C                                Version ADA

/* fichier liste.h */                    type Point;
typedef struct _point                    type Point is
{                                          record
   int x;                                     X : Integer;
   int y;                                     Y : Integer;
   struct _point *suivant;                    Suivant : Chainage;
   } point;                                end record;

typedef point *chainage;                 type Chainage is access Point;

/* fichier liste.c */
point doublets[10];                      Doublets: array (0..9) of Point;
chainage tete;                           Tete : Chainage;
chainage queue;                          Queue : Chainage;
```

Variante : on peut déclarer une liste préinitialisée lors du chargement du programme.

```
/*************************************************************************/
/* Une liste allouée et initialisée statiquement dans un tableau         */
/*         ------    -------   ------    ------                           */
/* tete -->|0|0|-|-->|2|4|-|-->|1|3|-|-->|4|0| |                          */
/*         -------   -------   ------- ^ ------                           */
/*            1         0        2     |   3                             */
/* queue ------------------------------                                  */
/*************************************************************************/

#include <stdio.h>
#include "liste.h"

/*************************************************************************/
/*                                 ---------------------------------     */
/*                                 | 2 , 4,                  -----|-----  */
/*                                 |          elements[0]         |<-- |  */
/*                                 ---------------------------------  | |  */
/*     tete --------------------->| 0 , 0                   -----|--- |  */
/*                                 |          elements[1]         |    |  */
/*                                 ---------------------------------    |  */
/*                                 | 1 , 3                   -----|--- |  */
/*                                 |          elements[2]         |<-|--  */
/*                                 ---------------------------------  |    */
/*    queue --------------------->| 4 , 0,               //////  |  |    */
/*                                 |          elements[3]         |<--    */
/*                                 ---------------------------------      */
/*************************************************************************/
```

```
struct _point_xy elements[4] = {
  {2,4,elements+2},
  {0,0,elements},
  {1,3,elements+3},
  {4,0,NULL}
};

chainage tete  = elements+1;
chainage queue = elements+3;

void
main (void)
{
   struct _point_xy *pt;
   for (pt = tete; pt != NULL; pt = pt -> suivant)
     printf ("Element à %8lx : %d, %d, suivant : %8lx\n",
              (unsigned long) pt, pt -> x, pt ->y,
              (unsigned long) pt -> suivant);
}
```

30 Extern : déclaration limitée à la définition du type

Par défaut, les déclarations de variables et de fonctions [10] remplissent un double rôle :
- Définir le type de la variable ou le prototype de la fonction
- Allouer statiquement et initialiser de la mémoire pour stocker le contenu (d'une variable) ou le corps (autrement dit les instructions) d'une fonction.

L'attribut **extern** spécifie que la déclaration n'est qu'une spécification de type sans allocation d'espace de stockage. On l'utilise principalement :
- Dans les fichiers d'en-tête (**.h**) pour définir le type des variables partagées entre modules compilés séparément.
- Dans les définitions d'objets qui se référencent mutuellement.

L'attribut **extern** est optionel pour la définition du prototype des fonctions (dans les fichiers d'en-tête) : un prototype de fonction sans corps est implicitement interprété comme une simple spécification du prototype de la fonction.

```
/* fichier main.h */
struct profession {
  const char *nom;
  struct outil *instrument;
  };
struct outil {
  const char *nom;
  struct profession *metier;
  };
```

```
/* fichier calcul.h */
extern void calcul (void);
extern unsigned long total;
```

10. en incluant les procédures considérées comme des fonctions C retournant void

```
/* fichier main.c */
#include <stdio.h>
#include "main.h"
#include "calcul.h"

extern struct profession routier;
struct outil camion = {"camion",&routier};

struct profession routier =
          {"chauffeur",&camion};

void
main ()
{
   total=0;
   calcul ();
}
```

```
/* fichier calcul.c */
#include "main.h"
#include "calcul.h"

unsigned long total;
void
calcul (void)
{
    total++;
}
```

31 Attribut static

La visibilité de la variable ou de la fonction et la classe de stockage dépendent de l'emplacement de la déclaration et de la présence ou non de l'attribut **static**.

La mémoire de stockage des variables dites statiques est allouée (dans data/bss) statiquement pour la durée de l'exécution du programme. Celle des variables dites dynamiques est allouée dans la pile le temps d'exécuter un appel de fonction [11] (allocation dans le prologue et libération dans l'épilogue de la fonction). Dynamiques par défaut, les variables locales des fonctions perdent leur valeur entre deux appels.

Emplacement de déclaration		Méthode de stockage/ valeur entre 2 appels		Attribut static		Visibilité limitée à
Hors d'une fonction	→	statique/conservée		absent	→	tous fichers
				présent	→	fichier local
Intérieur d'une fonction		statique/conservée	←	présent		fonction locale
		dynamique/perdue	←	absent		

Deux choix des concepteurs du langage C sont criticables :
- L'attribut **static** agit sur deux aspects différents (classe de stockage et visibilité) selon l'emplacement de la déclaration.
- Une variable déclarée à l'extérieur d'une fonction est partagée/exportée par défaut.

```
/* fichier f1.c */
static int x=3;

/* x et calcul de f1 */     /* distincts */
static void calcul(void)
{ x++; }
```

```
/* fichier f2.c */
static int x=5;

/* de x et calcul de f2 */
static void calcul(void)
{ x=0; }
```

```
unsigned long compteur ()
{
```

11. ou un bloc d'instructions : on peut déclarer des variables locales à une fonction ou à un bloc d'instructions

```
static nombre_d_appels = 0;  /* initialisée au chargement du programme */
nombre_d_appels ++;
printf ("J'ai été appelée %u fois\n",nombre_d_appels);
}
```

32 Attribut const

L'attribut de stockage **const** spécifie que le contenu de la variable déclarée ne doit pas être modifiée (le compilateur essaiera alors de la stocker dans une section protégée contre les accès en écriture [12]). On peut généralement obtenir le même résultat avec #define, sauf :

1. si on souhaite manipuler l'adresse de la constante ainsi définie (ou fabriquer un tableau de constantes).
2. pour interdire la modification du contenu repéré par une adresse ou un pointeur.

A gauche de *, const indique que contenu repéré par l'adresse qui suit ne doit pas être modifié. A droite de *, const indique que l'adresse contenue dans le pointeur ne doit pas être modifiée.

```
                                    /* equivalent #define, commentaires */
const double RAC2 = 1.414;          /* #define RAC2 ((double) 1.414) */
const double CONS [2] = {3.14, 2.718};
double VAR [2];
double *ptd = VAR;                 /*  ptd++ legal, *ptd = ... legal */
                                   /*  ptd = CONS interdit           */

const double *ptcd =  CONS;        /*  ptcd++ legal, *ptcd = ... interdit */

int * const INTMSK = 0x3ff4008;    /* #define INTMSK ((int *) 0x3ff4008) */
                                   /* INTMSK++ interdit, *INTMSK legal */

const double * const cptcd = & RAC2;  /* cptcd ++ interdit et
                                         *cptcd = ... interdit */

/* Cette procédure ne doit pas modifier les chaînes de caractères */
/* dont elle recoit l'adresse (*ch1 = ... interdit)               */
int compare_chaine (const char *ch1, const char *ch2);
```

33 Constructeur selon : switch/case

La principale spécificité du contructeur **switch** de C est de nécessiter une instruction **break** à la fin de chaque cas. **Piège :** en l'absence de **break**, l'exécution continue avec le code du cas suivant.

```
-- version ADA
case Meteo is
   when SOLEIL => Put ("Mettre creme solaire);
   when PLUIE  => Put ("Ouvrir parapluie);
   when NEIGE  => Put ("Sortir la pelle);
   when VENT   => Put ("Rentrer le linge);
```

12. par exemple text ou rodata dans le monde POSIX

```
   when others => Put ("Ou avez-vous pris la meteo ?);
end case;

/* Version C */
switch (meteo) {
  case SOLEIL:  printf ("Mettre creme solaire); break;
  case PLUIE :  printf ("Ouvrir parapluie");break;
  case NEIGE :  printf ("Sortir la pelle");break;
  case VENT  :  printf ("Rentrer le linge");break;
  default    :  printf ("Ou avez-vous pris la meteo ?.);
             }

/* En oubliant le break à la fin de chaque branche */
for (x=0; x < 10; x++)
  {
  switch (x) {
    case 0 :      y = 0;       /* execute ensuite y = 1 du cas 1 */
    case 1 :      y = 1; break
    case 2 :      y = 2;       /* execute ensuite y = 4 du cas 4 */
    case 4 :      y = 3; break;
   default:       y = 6;
           }
  printf ("y = %d ",y); /* x==0 ou x==1 --> y=1 */
  }                     /* x==2 ou x==4 --> y=3 */
```

34 Compléments sur les tableaux et les pointeurs

34.1 Tableaux à n dimensions

Un tableau à n dimensions peut être considéré comme un tableau a une dimension dont chaque élément est lui-même un tableau à n-1 dimensions. Les éléments d'un tableau étant stockés côte à côte en mémoire, on peut parcourir un tableau à n dimensions avec un pointeur et une boucle.

```
typedef int tab4 [4]; /* le type tableau de 4 entiers */
tab4 u [3] = { {0,1,2,3}, {100,101,102,103}, {200,201,202,203} };

/* declaration du même type de tableau sans utiliser de typedef */
int t [3][4] = { {0,1,2,3}, {100,101,102,103}, {200,201,202,203} };

/* u[i]   est un tableau de 4 entiers                                    */
/* (u[i])[j] est le j_ième élément de ce tableau                         */
/* on peut l'écrire tout simplement u[i][j]                              */
/* Ordre de rangement en mémoire :                                       */
/* u[0][0], u[0][1], u[0][2], u[0][3], u[1][0], u[1][1], ..., u[2][3] */
/* t[0][0], t[0][1], t[0][2], t[0][3], t[1][0], t[1][1], ..., t[2][3] */

void
main (void)
{
   int i,j,*ptr;
```

```
   for (i = 0; i < 3; i++)
     for (j = 0; j < 4; j++)
        printf ("t[%d][%d] = %d,u[%d][%d] = %d\n", i,j,t[i][j],i,j,u[i][j]);
   for (ptr = &t[0][0]; ptr <= &t[2][3]; ptr++)
     printf ("%d ",*ptr);
   printf ("\n");

   /* affiche  a l'execution */
   t[0][0] = 0,    u[0][0] = 0
   t[0][1] = 1,    u[0][1] = 1
   t[0][2] = 2,    u[0][2] = 2
   t[0][3] = 3,    u[0][3] = 3
   t[1][0] = 100,  u[1][0] = 100
   t[1][1] = 101,  u[1][1] = 101
   t[1][2] = 102,  u[1][2] = 102
   t[1][3] = 103,  u[1][3] = 103
   t[2][0] = 200,  u[2][0] = 200
   t[2][1] = 201,  u[2][1] = 201
   t[2][2] = 202,  u[2][2] = 202
   t[2][3] = 203,  u[2][3] = 203
   0 1 2 3 100 101 102 103 200 201 202 203
}
```

34.2 Pointeurs de pointeurs et de fonctions

Toute entité C stockée en mémoire peut être repérée par un pointeur : on peut donc définir des pointeurs de fonctions et des pointeurs de pointeurs. Un pointeur de pointeur permet de passer à une fonction l'adresse d'un pointeur à modifier (en ADA, ceci devrait logiquement correspondre à un paramètre **out** ou **in out** de type **access**).

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

int x,y;
/* déclaration de 2 pointeurs d'entier avec contenu initiaux  */
int *px=&x, *py =&y;
/* pointeur qui sera affecté dans la procédure mon_alloc */
int *tableau_dynamique;
/* un pointeur de pointeur d'entier */
int **pp;

void
mon_alloc (unsigned long taille, void **adresse)
{
   void * res;
   res = malloc((size_t) taille);
   if (res == NULL) { /* gérer l'erreur */}
   *adresse = res;
}

void
```

```
main(void)
{
   pp = &px;
   **pp = 3;        /* *(*&px) --> *px --> *&x  --> x = 3 */
   *pp = &y;        /* px pointe maintenant sur y */

   mon_alloc (1000,&tableau_dynamique);
   tableau_dynamique[10]=4;
}
```

34.3 Exemple de tableau de pointeurs

```
#include <stdio.h>

char msg_bonjour [] = "bonjour !";
int autrechose;
char msg_merci [] = "merci !";
int encoreunautre;
char msg_au_revoir [] = "au revoir !";

/* un tableau de 4 pointeurs repérant des chaînes de caractères */
/* les chaînes ne sont pas contigües en mémoire */
char * messages [] = {msg_bonjour,msg_merci,msg_au_revoir,NULL};

void
afficher (char *msg[])   /* ou (char **msg) */
{
   char **m, *chaine;
   for (m=msg; *m != NULL; m++)
     {
     chaine = *m; printf ("%s\n",chaine);
     }
}

void data   (char *m) { printf ("data%s\n"  ,m); }
void collec (char *m) { printf ("collec%s\n",m); }
void initia (char *m) { printf ("initialisa%s\n",m); }

/* type pointeur de procedure à un paramètre pointeur de char */
typedef void (*procedure_char) (char *);

procedure_char commandes [3] = {data,collec,initia};

/* argc : nombre d'éléments de argv                    */
/* argv : les mots de la ligne de commande             */
/* envp : les définitions de variables d'environnement */
/* Exemple : "SHELL=/bin/tcsh"                         */
/* Pas de envc : envp termine par NULL                 */

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

```
{
  int i;
  for (i = 0; i< argc; i++)
      printf ("argv[%d] = %s\n",i,argv[i]);
  afficher (envp);
  afficher (messages);
  for (i = 0; i< 3; i++)
      (*(commandes[i])) ("tion");
  return 0;
}

/* execution */
mandelbrot> ./mon_programme truc bidule
argv[0] = ./mon_programme
argv[1] = truc
argv[2] = bidule
SHELL=/bin/tcsh
USER=waillep
...
LUSTRE_INSTALL=/usr/local/lustre
bonjour !
merci !
au revoir !
datation
collection
initiation
```

35 Entrées/sorties formattées : printf, scanf

printf ("texte affiché tel que %*format* texte",expression) ;

scanf ("%*format*",&variable) ;

Documentation : **man 3 printf** ou **man 3 scanf**. Printf retourne le nombre de caractères affichés. Scanf retourne le nombre d'éléments lus. A noter : pour lire une chaîne (format %s) avec scanf, on passe le tableau non précédé de & (le nom du tableau est déjà un pointeur).

Quelques formats usuels	
Format	Nature
%d	Entier relatif en décimal
%3d	Idem, affiche 3 chiffres
%u	Entier naturel en décimal
%lu	Idem pour un entier long
%x	Entier en hexdécimal
%c	Caractère (ASCCI dans un char)
%f	Nombre à virgule flottante
%s	Chaîne de caractères (char*)

```
int var;
char lu[100];
if (scanf ("%d",&var) != 1)
  {/* gestion d'erreur */ }
else
  {
  printf ("var = %d\n", var);
  /* utiliser_valeur_de (var) */
  }
if (scanf ("%s",lu) == 1)
  printf ("Lu : -->%s<-\n",lu);
```

36 Unions

Lorsque les membres d'une structure ne sont jamais utilisés simultanément, on peut "superposer" les membres, c'est-à-dire tous les stocker à la même adresse : en C, remplacer struct par union.

36.1 Record variant en ADA

```
type COORDONNEES is (XY,RT);

type POINT (nature : COORDONNEES := NONE) is
  record
    case nature is
       when XY => X : Integer; Y : Integer;
       when RT => R : FLOAT; T : FLOAT;
    end case;
  end record;

Origine : POINT;
Destination : POINT;

begin
  Origine := (XY,3,4);
  Destination := (RT,3.0,0.7);
end programme
```

36.2 Structure et union en C

```
enum coordonnees {XY=1, RT=3} ;

struct rectangulaire {int x; int y;};
struct polaire {float r; float t;};

union xy_rt {struct rectangulaire xy; struct polaire rt;};

struct point {
   enum coordonnees nature;
   union xy_rt valeur;
};

struct point origine;
struct point destination;

void
calcul (void)
{
   origine.nature = XY;
   origine.valeur.xy = (struct rectangulaire) {3,4};
   destination.nature = RT;
   destination.valeur.rt = (struct polaire) {3.0, 0.7};
}
```

```
void
afficher (struct point *s)
{
   if (s->nature == XY) { printf ("x = %d  y=%d\n",
                        s->valeur.xy.x, s->.valeur.xy.y );}
   if (s->nature == RT) { printf ("r = %f  t=%f\n",
                        s->valeur.rt.r, s->valeur.rt.t );}
}
```

Remarque : on pourrait réaliser beaucoup moins proprement ces affectations avec des conversions de type pointeur, en exploitant le fait que tous les membres de l'union sont stockés à la même adresse.

```
/* Comment réaliser ceci avec les conversions de type de pointeurs ?   */
/* destination.valeur.rt =     (struct polaire) {2.0,1.57}; */
/* origine.valeur.xy     =     (struct rectangulaire) {-1,-2}; */

* (struct polaire *) &destination.valeur = (struct polaire) {2.0,1.57};
* (struct rectangulaire *) &origine.valeur = (struct rectangulaire) {-1,-2};
```

37 Opérateurs entiers bit à bit

On considère ici chaque entier comme une collection de bits ou de booléens correspondant à sa représentation en base deux.

La représentation du décalage à gauche de b bits de x (**dec_b_bits_gauche = x « b**) est telle que b bits 0 sont ajoutés à droite (poids faibles) et b bits supprimés à gauche (poids forts) de la représentation en binaire de x.

Le décalage de b bits à droite (**dec_b_bits_droite = x » n**) de x supprime à l'inverse b bits à droite et ajoute b bits à gauche.
On utilise un décalage **logique** pour un entier x naturel (**unsigned**) : il ajoute b fois un bit à 0 en poids fort. On utlise un décalage **arithmétique** pour un entier x **relatif** : il ajoute b fois le bit de signe (bit de poids fort) de x en poids fort.

En C on note respectivement ~ , **&** ,| et ^ les opérateurs Non (un seul opérande, calcule le complément à un), Et, Ou et Ou Exclusif. Tous opèrent colonne par colonne : chaque bit d'un entier est interprété comme la représentation d'un booléen.

```
/*********************************************************************/
/*   Non             Et            Ou               Ou exclusif      */
/*   0011 1100       0011 1100     0011 1100        0011 1100        */
/* ~               & 1100 1001   | 1100 1001      ^ 1100 1001        */
/*   ---------       ---------     ---------        ---------        */
/*   1100 0011       0000 1000     1111 1101        1111 0101        */
/*********************************************************************/

unsigned char x = 0x3c;
unsigned char y = 0xc9;
```

```
unsigned char and, or, xor, not;

not = ~x;          /* not = 0xc3 */
and = x & y ;      /* and = 0x08 */
or  = x | y ;      /* or  = 0xfd */
xor = x ^ y;       /* xor = 0xf5 */
```

Attention : ne confondez pas les opérateurs bit à bit ~ , & et | avec les opérateurs booléeens !, && et || utilisés pour exprimer les conditions : ces derniers interprètent chaque valeur entière comme un seul booléen.

38 Paramètres de main : argc, argv, envp

Les paramètres de la procédure main sont deux tableaux de pointeurs de chaînes de caractères nommés argv et envp. Le contenu de argv correspond aux arguments de la ligne de commande utilisée pour lancer l'exécution. Envp contient des définitions de variables d'environnement sous la forme "nom_de_variable=définition".

La taille de argv est définie par le paramètre **argc** et le dernier élément de envp est suivi d'un pointeur **NULL**.

```
/* fichier listarg.c */
#include <stdio.h>

int main (int argc, char *argv[], char*envp[])
{
   int i;
   printf ("Ligne de commande : ");
   for (i = 0; i <argc; i++)
     printf ("%s ",argv[i]);
   printf ("\n\nEnvironnement : \n");
   i = 0;
   while (envp[i] != NULL)
     printf ("%s\n");
   printf ("\n");
   return 0;
}
```

```
mandelbrot> ./listarg -g 123 456
Ligne de commande : ./listarg -g 123 456

Environnement :
USER=waillep                    /* quelques définitions parmi */
...
HOST=mandelbrot                 /* toutes les variables d'environnement */
...
HOME=/u/w/waillep               /* affichées */

mandelbrot>
```

39 Mots réservés du lannage C

Vous ne pouvez utiliser les mot réservés du langage comme noms de variable ou de fonction. En voici la liste :

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