Le langage C

Algo – Chapitre 1

I. Syntaxe du langage C

```
/// MACROS

#include <lib.h>      // inclusion de fichier

#ifndef __NOM_FICHIER__ // protection contre les inclusions multiples
#define __NOM_FICHIER__
// contenu du fichier
#endif

/// DEFINTION DE TYPES

typedef typeExistant nouveauType;                  // redéfini un type
enum [NomType] = {id1 [=val1], id2 [=val2 ], ...}; // déf. un type énuméré ayant pour nom "enum [NomType]"
struct [NomType] { type1 attr1; ... };             // déf. une structure ayant pour nom "struct [NomType]"
union [NomType] { type1 attr1; ... };              // déf. une union ayant pour nom "struct [NomType]"
typedef enum {id1 [=val1], ...} NomType;           // donner le nom "NomType" à un type ci-dessus
// Rq : struct réserve l'espace de tous les attributs (tous champs utilisables en même temps), unions réserve
l'espace du plus gros (1 champ utilisable à la fois)

/// CONSTANTES ET VARIABLES

#define CONSTANTE valeur       // défini une constante par macro
const type CONSTANTE [= val];  // défini une variable "constante"
type nomVariable [= val];      // défini une variable

typeDonnees tbl1Dim[taille];               // défini une variable tableau nommée tbl1Dim
typeDonnees tblNDim[taille1]...[taillen];  // défini une variable tableau nommée tblNDim
typeDonnees tbl1Dim[] = {el1, el2, ...}    // défini une variable tableau nommée tbl1Dim

/// FONCTIONS

// param1 : passage par valeur (E) / *param2 : passage par adresse (E/S)
// type3 (*fct3) (fct3Type1, ...) fonction passée en paramètre
typeRetour nomFonction(type1 param1, type2 *param2, type3 (*fct3) (fct3Type1, ...), ...) {
       *param2 = ...; instr;
}
nomFonction(var1, &var2, fct, ...); // appel

/// STRUCTURES CONDTIONNELLES

 // si... sinon...
 if (cond)
        instr;
 [else
        instr;]

 // cas où....
 // on exécute toutes les instructions entre
 le premier case valide et le premier break
 rencontré.
 switch(selecteur) {
        case cas1:
               instr;
               [break;]
        ...
        default;
               instr;
 }

// pour...
for(init; cond_arret; instr_fin_itér.)
       instr;
// ex : for (i = 0; i < n; i++)

// tant que...
while(cond)
       instr;

// répeter... tant que...
do
       instr;
while (cond);

/// I/O

// Patterns : %d : int, %f : réel, %s : str, %c car
printf("str"[, var1, var2, ...]); // afficher str, en remplaçant les patterns %x par les variables vars
scanf("patterns", &var1, &var2, ...) // met les vals de types "patterns" dans les variables pointées

// fichiers (tous binaires)
FILE *f;
f = fopen(const char *path, const char *mode); // modes : r, r+, w, w+, a, a+
fread(...); fwrite(...); fseek(...); ... fclose(f);
```

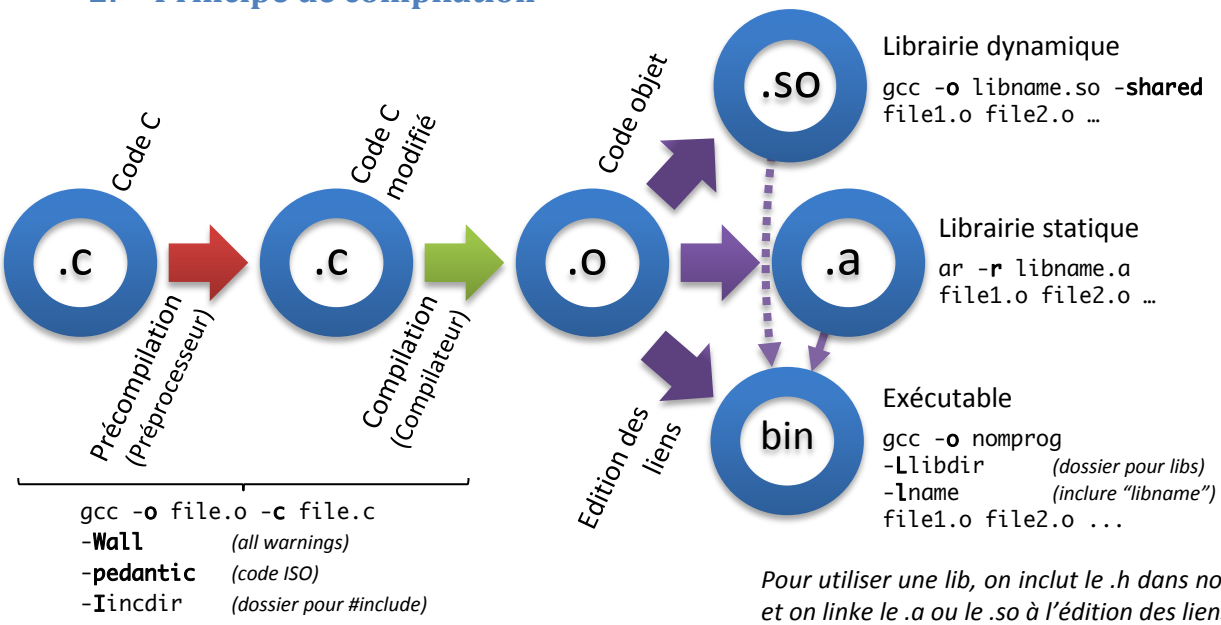
```c
/// MAIN

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
	// argc : nbr de paramètres passés
	// argv : tableaux des paramètres (ex : argv[2] pour le 3e paramètre)
	return EXIT_SUCCESS;
}
```

II. Compilation d'un projet en C

1. Organisation des fichiers

- src/ Sources .c (types et fcts privés et corps des fcts publiques) et modules .o
- include/ En-têtes .h (types et signatures de fcts publics)
- bin/ Exécutable
- lib/ Librairies .a et .so

2. Principe de compilation



Pour utiliser une lib, on inclut le .h dans notre code et on linke le .a ou le .so à l'édition des liens.

3. Makefile

Principe	Exemple	
VARIABLE = valeur **cible** : dépendance actions **$@** : cible **$<** : 1e dépendance **$^** : ttes dépendances	SRCDIR=src LIBDIR=lib BINDIR=bin INCDIR=include CC = gcc AR = ar CFLAGS=-Wall -pedantic -I$(INCDIR) LDFLAGS=-L$(LIBDIR) EXEC=tri all : $(BINDIR)/$(EXEC)	$(BINDIR)/tri : $(LIBDIR)/libech.a $(SRCDIR)/triMin.o $(SRCDIR)/main.o $(CC) $(LDFLAGS) -o $@ $(SRCDIR)/triMin.o $(SRCDIR)/main.o -lech $(LIBDIR)/lib%.a : $(SRCDIR)/%.o $(AR) -q $@ $^ $(SRCDIR)/%.o : $(SRCDIR)/%.c $(CC) -o $@ -c $< $(CFLAGS) clean : rm $(BINDIR)/* rm $(SRCDIR)/*.o rm $(LIBDIR)/*.a

III. Les types

Signification	Type de donnée C	Taille (octets)	Plage de valeurs acceptée
Vide (fct sans sortie)	`void`		
Caractère [non signé]	`[unsigned] char`	1	0 à 255 (-128 à 127)
Entier court [non signé]	`[unsigned] short int`	2	0 à 65 535 (-32 768 à 32 767)
Entier [non signé]	`[unsigned] int`	2 ou 4	0 à 4 294 967 295 (-2 147 483 648 à 2 147 483 647)
Entier long [non signé]	`[unsigned] long int`	4	
Flottant (réel)	`float`	4	3.4×10^{-38} à 3.4×10^{38}
Flottant double	`double`	8	1.7×10^{-308} à 1.7×10^{308}
Flottant double long	`long double`	10	3.4×10^{-4932} à 3.4×10^{4932}

Mots clés possibles devant un type

- `const` : variable non-modifiable.
- `static` (var globale ou fct) : limite la portée au fichier courant.
- `static` (var locale) : conserve son état entre 2 exécutions de la fonction.
- `extern` : var ou fct définie dans un autre fichier.

IV. Les constantes implicites

Entiers	`123` 123_{10} (Décimal)		`0123` 123_8 (Octal)		`0x12F` $12F_{16}$ (Hexa)
Réels	`2.` 2,0	`.3` 3,0	`2e4` 2×10^4		`2.3e4` $2{,}3\times10^4$
Caractères	`'a'` a	`'\001'` Code ASCII 001	`'\n'` Retour ligne	`'\t'` Tab	`'\\'` \	`'\''` '
Chaînes de caractères	`"chaine"` Fin de chaîne marquée par `'\0'` (ajouté automatiquement par le compilateur)					

V. Les opérateurs

Comparaison	==	!=	<	<=	>	>=			
Logique	&& et	|| ou	! non						
Calcul	+	-	*	/	% mod	& et bàb	| ou bàb	^ xor bàb	
Affectation	+=	-=	*=	/=	%=	&=	|=	^=	=
Incrémentation	++	--							

bàb : bit-à-bit

VI. Exemples de fonctions de la bibliothèque standard

1. Assert.h : préconditions

`assert(expr. bool)` termine le programme si faux et si `NDEBUG` n'est pas défini. Exemples :

```
/* #define NDEBUG */ assert(a > 0)
```

2. Stdarg.h : fonctions d'arité variable

```
void arite_variable(int nbArg, ...) {
    va_list argCourant;
    va_start(argCourant, nbArg);          // initialise l'aritée variable
    for (i = 0; i <= nbArg; i++) {
        valeur = va_arg(argCourant, int); // récupère l'argument suivant de type int
    }
    va_end(argCourant);                   // termine l'arité variable
}
```

VII. Allocation statique et dynamique

1. Segments mémoire et allocation

Les entités utilisées sont placées dans la mémoire vive dans divers segments :

- **statique ou *text* :** programmes et sous-programmes
- ***bss* :** variables globales
- ***data* :** constantes
- **tas ou *heap* :** espaces alloués dynamiquement
- **pile ou *stack* :** espaces alloués statiquement

- **Allocation statique :** Allocation prévue à la compilation (variables locales, paramètres, …)
- **Allocation dynamique :** Allocation non prévue à la compilation, écrite par le programmeur.

2. Les pointeurs

Un *pointeur* **p** est une variable de type *« pointeur sur T »* noté **T*** référençant une adresse mémoire permettant de stocker une information de type **T**. Un pointeur à **NULL** ne pointe sur rien.

***p** permet d'accéder à l'espace mémoire pointé par p. **&var** permet d'obtenir l'adresse de la variable var. *(On peut aussi utiliser ++, --, + et - aves les pointeurs)*

Exemple :

```
int* p;
int i;

p = NULL;
p = &i;
p* = 3;
```

3. Allocation dynamique

- **Allocation :** `Type* p = (Type*) malloc(n*sizeof(Type));`
- **Libération :** `free(p);`

VIII. Les tableaux

`type t[taille];` Un tableau **`t`** est un **pointeur constant vers la première case**. **`t[x]`** est une **variable** ayant la valeur de la x+1^{ième} case de `t` (`t` indicé de 0 à taille-1).

Il est forcément passé par adresse vers une fonction, et peut donc toujours être modifié par la fonction. Dans la signature de la fonction, on peut écrire `type t[]` sans indiquer la taille.

IX. Les chaînes de caractères

Les chaînes de caractères sont généralement des tableaux de caractères (il faut n+1 cases pour stocker n caractères). 2 façons de déclarer une chaîne que l'on veut passer d'une fonction à l'autre :

Déclarée dans l'appelante : allocation statique ou dynamique	Déclarée dans l'appelée : allocation dynamique

`void saisie(char buffer[]) {`
`  fgets(buffer, MAX, stdin);`
`}`
 	
`int main() {`
`  char  buffer[MAX];`
`  saisie(buffer);`
`}`
	`char* saisie() {`
	`  char* buffer = (char*) malloc(MAX*sizeof(char));`
	`  fgets(buffer, MAX, stdin);`
	`  return buffer;`
	`}`
	
	`int main() {`
	`  char* buffer = saisie();`
	`  free(buffer);`
	`}`