T.E.R. et Algorithmique : Charte C

Le langage C est un langage extrêmement permissif, bien au delà du raisonnable pour qui veut (faire) programmer proprement. Pour cette raison, le C que vous allez être autorisé à utiliser dans le cadre du T.E.R. (et du cours d'algo) est une version restreinte qui vous obligera à respecter les règles de propreté imposées par Pascal. Ce document vous donne la liste des obligations et des interdictions.

D'une manière générale, vous écrirez vos programmes de façon à ce qu'ils soient compréhensibles par quelqu'un qui ne connait pas C (mais qui a compris les pointeurs, la récursivité...). Vous remarquerez qu'un programme Pascal est à peu près compréhensible par quelqu'un qui ne connait pas Pascal. Par contre, un programme C écrit par quelqu'un qui écrit "à la C" (pas dans le style cette charte) est illisible par un néophyte.

1 Les types

1.1 Entiers, réels, caractères

Le C contient les types entier *int*, réel *float* et caractère *char* qui seront ceux utilisés pour le projet.

(Il existe également d'autres types qui sont à peu près propres mais devraient être inutiles (long, double, ...), ou sales (signed char)).

Certaines versions de C font un amalgame entre *int* et *char* qu'il est interdit d'utiliser. Une conversion de caractère vers l'entier se fera de façon explicite avec un cast, par exemple, pour convertir un char entre '0' et '9' en l'entier correspondant, on écrira `i = (int) c - (int) '0' ;`

1.2 Booléens

Il n'existe pas de booléen en *C* et ce sont les entiers qui jouent ce rôle, 0 valant faux et le reste vrai. On introduira le type booléen par les déclarations suivantes :

```
typedef enum {
    false,
    true
} bool;
```

puis l'on fera une séparation claire des entiers et des booléens comme si le compilateur considérait qu'il s'agit de types différents. Il est par exemple absolument interdit d'abréger `if i<>0` en `if i` .

On fera attention au fait que true peut se présenter sous des formes variés (1,2,10,-1,...) de sorte que certaines expressions comme `bool b1, b2 ; ... if b1==b2 ...` ne marchent pas (car si b1=1 et b2=2, ils sont tous les deux true, donc le test égalité doit donner vrai, alors que le C le donnera comme étant faux).

1.3 Pointeurs et structures

Pour un pointeur nul (nil en Pascal), on utilisera NULL, et en aucun cas 0 (car 0 est un entier, pas un pointeur), et d'une manière générale, on fera une distinction claire entre pointeurs et entiers.

On n'utilisera pas le type void* (en dehors du pointeur rendu par malloc, qui est de ce type) qui enlève son type complet à un pointeur.

L'usage de structures dynamiques avec pointeurs et structures est bien entendu autorisé.

1.4 Tableau

En C, un tableau est en fait un pointeur vers la première case du tableau, et les notations tableaux peuvent se réécrire avec des notations pointeurs. On cachera cette confusion des types, notamment en utilisant systématiquement les notations tableaux (Utilisez `t[i]` et `&t[i]`, et pas les notations équivalentes `*(t+i)` et `(t+i)`). Vous serez néanmoins obligé de savoir que les tableaux sont des pointeurs pour comprendre certains comportements du compilateur.

Les tableaux de taille statique sont autorisés. L'utilisation de tableaux de taille dynamique permettant de décider à l'exécution la taille du tableau est acceptée s'ils sont de dimension 1 et sauf indication contraire.

2 Expressions et instructions

Une expression est quelque chose qui rend une valeur. Une instruction est quelque chose qui fait une action.

La distinction entre les 2 doit être claire et sans ambiguïté, or C est d'un dangereux laxisme sur cette disctinction. Il vous est rigousement interdit de profiter de ce laxisme. En particulier :

2.1 L'affectation

Lorsque l'on effectue une affectation, on range une valeur (qui est le résultat de l'expression droite) dans une case mémoire (qui est l'adresse de ce qui se trouve à gauche). Il s'agit donc là d'une action, c'est à dire d'une instruction. L'instruction et l'expression devant être distinguées, on n'utilisera jamais l'affectation comme une expression, et en particulier on n'écrira PAS `x=y=2 ;` (En C, on dit que `y=2 ;` avec le point virgule est une instruction, ce qui est ok, et que `y=2` sans le point virgule est une expression, ce qui est un abus puisqu'il y a un effet de bord, à savoir le placement de 2 dans la mémoire `y`)

2.2 $i++$

Là encore, quand on écrit $i++$, il y a une action qui est effectuée, à savoir l'incrémentation de i, il ne peut donc s'agir que d'une instruction, et il est interdit de l'utiliser autrement (donc `j=i++` est interdit). (Comme pour l'affectation, C considère qu'il s'agit d'une expression ou d'une instruction suivant qu'il y a ou non un point-virgule. Comme pour l'affectation, dire que ca peut être une expression est un abus).

2.3 Procédures et fonctions

Là encore, on ignorera la possibilité de C de faire des abus et on considérera que :

Une fonction est quelquechose qui calcule un résultat à partir de données. Elle ne fait rien d'autre, ce qui veut dire en particulier qu'elle n'a pas d'effet de bord, et qu'elle ne modifie pas la valeur de ses arguments. Une appel d'une fonction est faite pour obtenir son résultat, et donc $f(...)$ est une expression.

Une procédure est quelquechose qui agit sur ses données, elle ne rend pas de résultat au sens de la fonction (mais elle peut avoir des arguments OUT). Son appel est fait pour agir, et donc $p(...)$ est une instruction.

En C, la limite est floue. Vous ne vous approcherez pas de la frontière. Vous n'écrirez que des fonctionnalités qui sont clairement soit des fonctions (pas d'effet de bord, un résultat rendu, utilisation comme expression), soit des procédures (pas de résultat rendu i.e. void, effet de bord sur les arguments, utilisation comme instruction).

Il faut malheureusement noter que vous serez obligés d'utiliser des fonctionalités de C qui font des mélanges, par exemple malloc, qui fait une action (trouver une zone mémoire libre et l'asservir) et qui rend un résultat, à savoir l'adresse de cette mémoire que vous allez stocker dans votre pointeur. malloc n'est donc ni procédure ni fonction, mais un bâtard. Notez que Pascal est par contre propre sur ce point puisque la syntaxe Pascal est `new(p) ;` de sorte que new y est clairement une procédure.

3

Les constructeurs d'instructions classiques, {} (pour begin end), if, while, do while, for, switch sont autorisés sous réserve des règles qui suivent :

3.1 goto, break, continue

Le goto est un archaïsme. Il est interdit.

Le break est un goto caché et est donc en principe interdit, mais il est autorisé dans l'utilisation normale du switch, et est toléré dans tous les cas où son utilisation évite des lourdeurs et améliore la lisibilité. L'on doit pouvoir connaître la condition de sortie de boucle au niveau du mot-clef "while".

Par conséquent, si on sort par un break, on mettra un commentaire au niveau du mot-clef "while" pour signaler la sortie `break`.

Le continue est également un goto caché et est donc interdit sauf tolérance pour les cas où il améliore la lisibilité. Les tolérances sont accordées plus difficilement que pour le break.

3.2 For

Apres le mot-clef if, il y a une parenthèse qui contient 3 choses : tout d'abord une initialisation d'une variable compteur (donc une instruction), puis un test sur cette variable pour savoir si l'on continue (donc une expression de type booléen), et enfin une incrémentation ou décrémentation de cette variable (donc une instruction). Une autre utilisation ne correspondrait pas au principe du for et doit être remplacée par un while. Notez que C considère que la syntaxe est for(exp;exp;exp) (C peut le faire car il confond dans bien des situations l'expression et l'instruction), ce que l'on ne peut que regretter (et qui explique l'absence de point virgule après l'incrémentation, les 2 autres points-virgules étant d'ailleurs des séparateurs et pas des signes de fin d'expression)

4

Une possibilité C qui n'est pas citée dans ce document est à priori interdite (sauf ce que j'aurais oublié de citer, et que le bon sens dit être propre). Pour qu'ils ne soient pas interdits, je cite : les constantes, les commentaires, l'usage du typedef, les fonctions du préprocesseur (#define, #include, ...). On peut également faire du "sucre syntaxique" pour améliorer la lisibilité :

```
#define AND &&
#define OR ||
#define NOT !
```

Pour mémoire, je considère les pointeurs vers des fonctions comme propres, mais je ne vois pas à quoi ils pourraient vous servir pour le TER ou le cours d'algo.

Quelques règles d'utilisation pour terminer :

On ne déclare pas les types n'importe où mais en debut de programme (où dans un fichier séparé), exceptionnellement en début de fonctionalité.

On ne déclare pas les variables n'importe où mais en début de fonctionalité (Ou, éventuellement, au début du corps d'une boucle pour empêcher la connaissance de la variable en dehors de la boucle).

Annexe : Equivalents Pascal-C

A gauche, les versions Pascal, à droite, l'équivalent C.

Les types bool et entier

En C, le type booléen n'existe pas. Son rôle est joué par les entiers. Pour écrire des programmes propres et lisibles, on l'introduit de la manière suivante :

```
typedef enum {
    false,
    true
} bool;
```

On fera attention à ce que toute valeur fausse est égale à FALSE, mais que toute valeur vraie n'est pas nécessairement TRUE (valeurs 2,-1,...)

```
var i : integer ;                    int i ;
    b : boolean ;                    bool b ;
```

Les tableaux

```
type tableau = array[1..3] of integer ;     typdef int tableau[3];
var t : tableau ;                           tableau  t;
var u : array[1..4] of integer ;            int u[4];
```

Note : En C, les tableaux commencent toujours avec l'indice 0, donc u[4] signifie que l'on a les cases u[0], u[1], u[2] et u[3].

Les types structurés récursifs

```
type arbre = ^recarbre ;
     recarbre = record
                valeur : integer ;
                fg,fd : arbre
                end ;

var a1,a2 : arbre ;
```

```
typedef struct tmp_arbre {
           int     valeur ;
           struct tmp_arbre  *fg,*fd;
                 } r_arbre;
typedef r_arbre *arbre;

arbre  a1,a2;
```

Les procédures

```
procedure changer(i:integer; var j:integer);
begin ...
    changer(x,y); ...
end ;
```

```
void  changer(int i, int *j);
{ ...
   changer(x,&y) ; ...
}
```

Les fonctions

```
function taille (a:arbre) : integer ;
begin ... taille := 0 ; ... end ;
```

```
int  taille (arbre a);
{ ... return(0) ; ...}
```

On notera qu'en C, la fonction ne fait plus rien après le return (Il y a un break_function implicite). Il n'y a pas de tel phénomène en Pascal.

Le for-do

```
for i:= 1 to 3 do T[i]:=i;
```

```
for (i= 1; i <= 3;i++)  T[i]=i;
```

Le While

```
while (t[k]>t[k+1]) do k:=k-1;
```

```
while (t[k]>t[k+1]) { k=k-1;}
```

Le repeat-until

```
repeat
     i:= i+1;
until ((i>= n) or (R[i]<>0));
```

```
 do{
      i= i+1;
 } while ((i<n) && (R[i]=0));
```

Remarquez que la condition a été remplacée par sa négation.

Les pointeurs

```
var a,b : arbre ;
a:=nil;
new(a);
a^.valeur:=5;
a^.suiv:=nil;
new(b);
a^.suiv:=b;
dispose(a);
```

```
arbre a,b ;
a = NULL ;
a = (arbre) malloc(sizeof (r_arbre));
a->valeur = 5 ;
a->suiv = NULL ;
b = (arbre) malloc(sizeof (r_arbre));;
a->suiv = b ;
free(a);
```

(précision: si p est une variable de type structure on accéde a un champs de la structure par p.champs. Si p est un pointeur vers la structure, on peut y accéder par $p- > champs$ qui est un équivalent de $(*p).champs$.)

Les écritures

```
write('oui');
writeln('On a T[',i,'] = ',T[i]) ;
```

```
printf("oui");
printf("On a T[%d] = %d\n",i,T[i]);
```