Cours « Complément de langage C pour l'Electronique »

1. Rappels de langage C

Pointeurs typés

Présentation

A une variable correspond un emplacement mémoire caractérisé par une adresse et une longueur (par exemple 4 octets consécutifs pour un `long int`). C'est, bien sur, le compilateur qui assure la gestion de la mémoire et affecte à chaque variable un emplacement déterminé. On peut accéder à la valeur de cette adresse grâce à l'opérateur unaire &, dit opérateur d'adressage.

Un pointeur est une variable d'un type spécial qui pourra contenir l'adresse d'une autre variable. On dit qu'il pointe vers cette variable. Celui-ci devra aussi connaître le type de la variable vers laquelle il pointe puisque la taille d'une variable (en octets) dépend de son type. La déclaration d'un pointeur devra donc indiquer le type d'objet vers lequel il pointe (on dit d'un pointeur qu'il est typé). La syntaxe est la suivante:

```
type *identificateur;
```

Par exemple, pour déclarer un pointeur vers un entier, on écrira:

```
int* péntier;
```

ou encore:

```
int *péntier;
```

On peut réaliser des opérations sur des pointeurs de même type. On peut en particulier affecter à un pointeur un autre pointeur du même type ou l'adresse d'une variable de même type que celle du pointeur.

On accède à la valeur stockée à l'adresse contenue dans un pointeur grâce à l'opérateur unaire, dit de référencement ou d'indirection: *

Dans l'exemple suivant :

```
int a;
int* p_a;
p_a=&a;
```

*p a et a font référence au même emplacement mémoire (et ont donc la même valeur).

Un pointeur peut par ailleurs pointer vers un autre pointeur.

On peut aussi incrémenter un pointeur. Cela revient à augmenter sa valeur de la taille de l'objet pointé et donc à pointer sur l'objet suivant du même type (s'il en existe un!).

La déclaration d'un pointeur alloue un espace mémoire pour le pointeur mais pas pour l'objet pointé. Le pointeur pointe sur n'importe quoi au moment de sa déclaration. Il est conseillé d'initialiser tout pointeur avant son utilisation effective avec la valeur NULL (constante prédéfinie qui vaut 0) ce qui, par convention, indique que le pointeur ne pointe sur rien.

Pointeurs et tableaux

La déclaration d'un tableau réserve de la place en mémoire pour les éléments du tableau et fournit une constante de type pointeur qui contient l'adresse du premier élément du tableau. Cette constante est identifiée par l'identificateur donné au tableau (sans crochets). C'est cette constante de type pointeur qui va permettre de manipuler facilement un tableau en particulier pour le passer en paramètre d'une fonction puisqu'on ne passera à la fonction que l'adresse du tableau et non tous ses éléments.

L'exemple suivant :

```
int tab[10];
```

déclare un tableau de 10 éléments. tab contient l'adresse du premier élément et donc l'expression :

```
tab == &tab[0]
```

est VRAIE. On a donc de même :

```
*tab == tab[0]
tab[1] == *(tab+1)
tab[k] == *(tab+k)
```

Les deux écritures sont autorisées.

La différence entre un tableau et un pointeur, est qu'un tableau est une constante non modifiable alors qu'un pointeur est une variable.

Passage de paramètres à une fonction

On a vu que les paramètres passés à une fonction le sont par valeur et ne peuvent pas être modifiés par l'exécution de la fonction. Ceci est très contraignant si l'on souhaite qu'une fonction renvoie plusieurs résultats.

Par exemple, si l'on souhaite écrire une fonction permutant deux entiers, le code suivant qui parait correct ne fonctionnera pas :

```
void permutation(int a, int b)
{
int c;
c=a;
a=b;
b=c;
}
```

L'appel de la fonction :

```
permutation(x,y);
```

n'aura ainsi aucun effet sur x et sur y.

La solution consiste à passer, pour les paramètres que l'on souhaite voir modifier par la fonction, non plus les valeurs de ces paramètres mais les valeurs des adresses de ces paramètres.
Les paramètres de la fonction devront donc être des pointeurs. On accèdera aux valeurs proprement dites à l'intérieur de la fonction grâce à l'opérateur d'indirection *.

Si l'on reprend l'exemple précédent, cela donne:

```
void permutation(int* p_a, int* p_b)
{
int c;
c=*p_a;
*p_a=b;
*p_b=c;
```

```
}
```

Remarque: on aurait pu garder les identificateurs initiaux a et b !

Et l'appel devra se faire en passant en paramètres les adresses des variables à modifier grâce à l'opérateur d'adressage & :

```
permutation(&x,&y);
```

Allocation dynamique

La déclaration de variables dans la fonction main ou globalement réserve de l'espace en mémoire pour ces variables pour toute la durée de vie du programme. Elle impose par ailleurs de connaître avant le début de l'exécution l'espace nécessaire aux stockages de ces variables et en particulier la dimension des tableaux. Or dans de nombreuses applications le nombre d'éléments d'un tableau peut varier d'une exécution du programme à l'autre.

La bibliothéque stdlib fournit des fonctions qui permettent de réserver et de libérer de manière dynamique (à l'exécution) la mémoire. La fonction qui permet de réserver de la mémoire est malloc(). Sa syntaxe est :

void* malloc(unsigned int taille)

La fonction malloc réserve une zone de taille octets en mémoire et renvoie l'adresse du début de la zone sous forme d'un pointeur non-typé (ou NULL si l'opération n'est pas possible). En pratique, on a besoin du type d'un pointeur pour pouvoir l'utiliser. On souhaite d'autre part ne pas avoir à préciser la taille de la mémoire en octets surtout s'il s'agit de structures. L'usage consiste donc pour réserver de la mémoire pour une variable d'un type donné à:

- déclarer un pointeur du type voulu,
- utiliser la fonction sizeof(type) qui renvoie la taille en octets du type passé en paramètre,
- forcer malloc à renvoyer un pointeur du type désiré.

Par exemple, pour réserver de la mémoire pour un entier, on écrira:

```
int* entier;
entier=(int*) malloc(sizeof(int));
```

Ceci est surtout utilisé pour des tableaux, par exemple pour un tableau de N « complexe » (cf. le paragraphe sur les structures) on écrirai :

```
struct complexe * tabcomp;
tabcomp=(struct complexe*) malloc(N*sizeof(struct complexe));
```

La fonction free() permet de libérer la mémoire précédemment réservée. Sa syntaxe est:

```
void free(void* p)
```

2. Algorithmes

Notion d'algorithme

Un algorithme est une suite de traitements élémentaires effectués sur des données en vue d'obtenir un résultat en un nombre finis d'opérations.

Traitement élémentaire: traitement pouvant être effectué par un ordinateur. Notion relative. Exemple: le calcul de la racine carrée d'un nombre peut être considéré comme élémentaire en C en utilisant la bibliothèque mathématique. Il ne l'est pas pour un microcontrôleur programmé en assembleur.

Exemple: l'algorithme d'Euclide calculant le PGCD de deux entiers:

Soit la division euclidienne de a par b, a = bq+r. L'algorithme d'Euclide est basé sur le principe que les diviseurs communs de a et b sont les mêmes que ceux de b et r. En remplaçant a par b et b par r et en divisant à nouveau, on obtient deux entiers ayant les mêmes diviseurs que les entiers a et b d'origine.

Finalement, on obtient deux entiers divisibles entre eux (r = 0) dont le plus petit est le PGCD de a et de b.

Représentation des algorithmes

Un programme est la réalisation d'un algorithme. Pour s'affranchir d'un langage de programmation particulier, différentes techniques de représentation sont utilisées.

Pseudo langage

Permet de représenter formellement un algorithme indépendamment de l'implémentation (langage). Basé sur les instructions disponibles dans la plupart des langages.

Structures élémentaires :
- Entrées/Sorties: LIRE, ECRIRE
- affectation: X ← Y
- instruction conditionnelle: SI condition ALORS instructions SINON instructions FIN SI
-répétition :
  TANT QUE condition FAIRE instructions FIN TANT QUE
  POUR i=0 A n FAIRE instructions FIN POUR
  FAIRE instructions TANT QUE condition

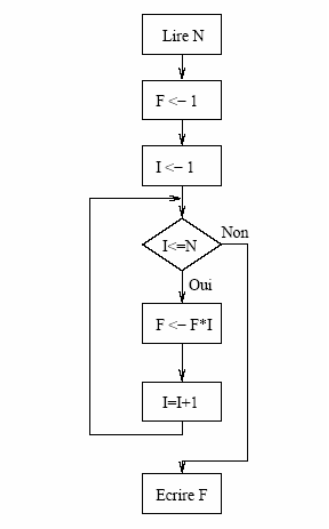
Exemple: calculer la factorielle de N (version itérative) :

```
LIRE N
F ← 1
POUR I=1 A N FAIRE
F ← F * I
FIN POUR
ECRIRE F
```

Il n'existe pas de formalisme universel. Chaque auteur à sa syntaxe particulière.

Organigramme

Un organigramme permet de représenter graphiquement un algorithme. Exemple pour le calcul de la factorielle:



En pratique, cette représentation devient vite illisible pour des problèmes complexes.

Analyse et complexité d'un algorithme

Il existe souvent plusieurs algorithmes permettant de résoudre un même problème. Exemple: les algorithmes de tri. Le choix du meilleur algorithme implique une analyse de ses performances. En général, le critère le plus important est celui du temps nécessaire à son exécution. Celui ci dépend le plus souvent de la quantité de données à traiter. Par exemple, le temps nécessaire pour trier un ensemble d'objets dépend du nombre d'objets.

L'étude de la complexité d'un algorithme consiste essentiellement à évaluer la dépendance entre le temps d'exécution et le volume de données. Les résultats s'expriment de manière qualitative: On cherche par exemple à savoir si la complexité croit de manière linéaire ou polynomiale avec le volume n de données.

Par exemple, la recherche d'un élément particulier dans un ensemble de n éléments non triés consiste à examiner chaque élément jusqu'à trouver celui que l'on cherche. Dans le meilleur cas, cela prendra une seule comparaison, dans le pire des cas, il faudra effectuer n comparaisons. En moyenne on aura n=2 comparaisons à effectuer.

On s'intéresse en général à la complexité d'un algorithme:
- dans le meilleur cas,
- en moyenne,
- dans le pire des cas.

On exprime la complexité d'un algorithme en ne gardant que le terme variant le plus avec n et en omettant les termes constants. Par exemple, un algorithme nécessitant $100n^3+1000n^2+5000n+10000$ instructions élémentaires sera dit de complexité $O(n^3)$. Un algorithme ne dépendant pas du volume de données sera dit de complexité $O(1)$. En général un algorithme de complexité $O(n \log n)$ sera plus efficace qu'un algorithme de complexité $O(n^2)$ mais ceci peut n'être vrai que si n est assez grand. En effet la complexité mesure le comportement asymptotique d'un algorithme quand n tend vers l'infini.

Le tableau suivant donne quelques exemples de complexité que l'on retrouve couramment. Soit E un ensemble de n données:

Algorithme	Complexité
Accés au 1er élément de E	$O(1)$
Recherche dichotomique (E trié)	$O(\log n)$
Parcours de E	$O(n)$
Tri rapide	$O(n \log n)$
Parcours de E pour chaque élément d'un ensemble F de même dimension	$O(n^2)$
Génération de tous les sous-ensembles de E	$O(2^n)$
Génération de toutes les permutations de E	$O(n!)$

Le tableau suivant donne les ordres de grandeurs des différentes complexités en fonction de la taille de l'ensemble de données :

Complexité	n=1	n=4	n=16	n=64	n=256	n=4096
$O(1)$	1	1	1	1	1	1
$O(\log n)$	0	2	4	6	8	12
$O(n)$	1	4	16	64	256	4096
$O(n \log n)$	0	8	64	384	2048	49152
$O(n^2)$	1	16	256	4096	65536	16777216
$O(2^n)$	2	16	65536	18446744073709551616		
$O(n!)$	1	24	20922789888000			

En résumé, la complexité d'un algorithme est la courbe de croissance des ressources qu'il requiert (essentiellement le temps) par rapport au volume des données qu'il traite.

Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même.

Exemple: calcul de la factorielle d'un nombre.

Définition itérative: n! = F(n) = n*(n _ 1)*(n _ 2)*…*2 * 1

Ce qui donne en C:

```
long int fact(int N)
{
  long int F=1;
  int i;
  for (i=1; i<=N; i++)
    F=F*i;
  return F;
}
```

Définition récursive: F(n) = n*F(n _ 1); F(0) = 1. Soit en C:

```
long int factr(int N)
{
  if (N==0)
    return 1;
  else
    return N*factr(N-1);
}
```

Si, en général, la version récursive d'un algorithme est plus élégante à écrire que sa version itérative [2], elle est cependant plus difficile à mettre au point et moins efficace en temps calcul.

Pour qu'un algorithme récursif fonctionne:
- il doit exister un cas terminal que l'on est sûr de rencontrer.
- un mécanisme de pile est par ailleurs nécessaire. Il est naturel en C dans le passage de paramètres à une fonction.

Algorithmes de tri

Il s'agit d'un problème classique (et utile) de l'algorithmique. On considère un ensemble d'éléments possédant une relation d'ordre total (Exemple: entiers, réels, caractères).

On cherche à ordonner cet ensemble dans l'ordre croissant (ou décroissant).

Tri par sélection

Principe : soit un ensemble de n éléments indicés de 0 _a n-1. On suppose que les m premiers éléments (0 à m-1) sont triés. On cherche la position k du plus petit élément parmi les éléments m _a n-1. On le permute avec l'élément m. L'ensemble se trouve donc trié de l'indice 0 _a l'indice m.

On parcourt ainsi l'ensemble de l'élément m=0 _a l'élément n-2.

Illustration

En gras, les éléments déjà triés, en italique, les éléments à permuter.

18	10	3	25	9	*2*
2	*10*	*3*	25	9	18
2	**3**	*10*	25	*9*	18
2	**3**	**9**	*25*	*10*	18
2	**3**	**9**	**10**	*25*	*18*
2	**3**	**9**	**10**	**18**	**25**

Algorithme

Soit à trier un tableau de N éléments (entiers), t[0] à t[N-1]

POUR m=0 à N-2 FAIRE
k ← p (indice du plus petit élément entre t[m] et t[N-1])
SI k différent de m ALORS permuter t[k] et t[m] FIN SI
FIN POUR

Programmation en C

Le programme peut se décomposer en trois fonctions:
- une fonction de calcul de l'indice du plus petit élément entre t[m] et t[N-1],
- une fonction de permutation,
- la fonction de tri proprement dite.

```
/* ------------------------ indice du plus grand élément entre m et n-1 */
int indice_min(int t[], int m, int n)
{
  int i;
  int imin;
  imin=m;
  for (i=m+1; i<n; i++)
    if (t[i]<t[imin])
      imin=i;
  return imin;
}

/* ---------------------------------------- permutation de 2 entiers */
void permute(int *a, int *b)
{
  int c;
```

```
  c=*a;
  *a=*b;
  *b=c;
}

/* ------------------------------------------------- tri par selection */
void tri_selection(int t[], int n)
{
  int m;
  int p;
  for (m=0; m<N-1; m++)
  {
    p=indice_min(t,m,N);
    if (p!=m)
      permutte(&t[p],&t[m]);
  }
}
```

Analyse : cet algorithme nécessite n^2 comparaisons et n permutations. Pour un nombre d'éléments donné, il effectue le même nombre d'opérations que les éléments soient pratiquement déjà triés ou totalement en désordre. Sa complexité est en $O(n^2)$.

Tri à bulle

Principe : le principe consiste à parcourir les éléments de l'ensemble de i=0 _a n-1 en permutant les éléments consécutifs non ordonnés.

L'élément le plus grand se trouve alors en bonne position. On recommence la procédure pour l'ensemble de i=0 à n-2 sauf si aucune permutation n'a été nécessaire à l'étape précédente. Les éléments les plus grands se déplacent ainsi comme des bulles vers la droite du tableau.

Illustration :

18	*10*	3	25	9	2
10	*18*	*3*	25	9	2
10	3	*18*	*25*	9	2
10	3	18	*25*	*9*	2
10	3	18	9	*25*	*2*
10	*3*	18	9	2	**25**
3	*10*	*18*	9	2	**25**
3	10	*18*	*9*	2	**25**
3	10	9	*18*	*2*	**25**
3	*10*	9	2	**18**	**25**
3	*10*	*9*	2	**18**	**25**
3	9	*10*	*2*	**18**	**25**
3	*9*	2	**10**	**18**	**25**
3	*9*	*2*	**10**	**18**	**25**
3	*2*	**9**	**10**	**18**	**25**
2	**3**	**9**	**10**	**18**	**25**

En italique, les deux éléments à comparer, en gras les éléments en bonne place.

Algorithme

k ← N - 1
FAIRE
POUR i=0 à J-1 FAIRE
SI t[i] > t[i+1] ALORS
permuter t[i] et t[i+1]

permutation=VRAI
FIN SI
FIN POUR
TANT QUE permutation=VRAI

Analyse : dans le pire des cas, le nombre de comparaisons et le nombre de permutations à effectuer sont de n^2. Dans le meilleur des cas (ensemble déjà trié), le nombre de comparaisons est de n _ 1 et l'algorithme est donc de complexité linéaire.

Tri par insertion

Principe : on prend 2 éléments et on les met dans l'ordre. On prend un troisième élément qu'on insère dans les 2 éléments déjà triés, etc.

Un élément m va être inséré dans l'ensemble déjà trié des éléments 0 _a m-1. Ce qui donnera m+1 éléments triés (0 à m). L'insertion consiste à chercher l'élément de valeur immédiatement supérieure ou égale à celle de l'élément à insérer. Soit k l'indice de cet élément, on décale les éléments k à m-1 vers k+1 à m et l'on place l'élément à insérer en position k.

Illustration

En gras, l'élément à insérer dans la partie triée du tableau.

18	**10**	3	25	9	2
10	18	**3**	25	9	2
3	10	18	**25**	9	2
3	10	18	25	**9**	2
3	9	10	18	25	**2**
2	3	9	10	18	25

Analyse

Dans le pire des cas, le nombre de comparaisons et de n^2. Dans le meilleur des cas, il est de N. L'algorithme est de complexité $O(N^2)$, mais il est plus efficace que les deux précédents si le tableau est en partie trié.