```
/**
 ** Hanoi1 - programme de demonstration de la recursivite pour l'initiation
 **          au C 5 de GS Infos 21 : le celebre probleme des tours de hanoi.
 **
 **          Version recursive.
 **
 **          Philippe Manet        12 Avril 1992
 **
 **/

#include <stdio.h>

#pragma optimize -1


void hanoi ( short n, char a, char b, char c )

{

   /*
   * Deplacement de N disques de A a C avec B comme intermediaire.
    */

  if ( n > 0 ) {

      hanoi ( n - 1, a, c, b );
       printf ( "Deplacement d'un disque de \"%c\" vers \"%c\"\n", a, c );
      hanoi ( n - 1, b, a, c );

  }

}   /* hanoi () */


void main ( void )

{

   char buf[4];
  short n;

   /*
   * Scanf est bugge : on lit donc une chaine de caracteres que l'on decode
   * ensuite.
    */

   printf ( "Nombre de disques ? ");
  gets ( buf );
  n = atoi ( buf );

   hanoi ( n, 'A', 'B', 'C' );

}   /* main () */
```

```
/**
 ** Hanoi2 - programme de demonstration de la recursivite pour l'initiation
 **          au C 5 de GS Infos 21 : le celebre probleme des tours de hanoi.
 **
 **          Version non recursive.
 **
 **          Philippe Manet        12 Avril 1992
 **
 **/

#include <stdio.h>

#pragma optimize -1

/*
 * Pile utilisee pendant les deplacements des disques.
 */

struct {
   short count;
   struct {
      short n;
      char a;
      char b;
      char c;
    } entry[100];
} stack;


void push ( short n, char a, char b, char c )

{
   short i;

   i = stack.count++;
   stack.entry[i].n = n;
   stack.entry[i].a = a;
   stack.entry[i].b = b;
   stack.entry[i].c = c;

}  /* push () */


void pop ( short *n, char *a, char *b, char *c )

{
   short i;

   i = --stack.count;
   *n = stack.entry[i].n;
   *a = stack.entry[i].a;
   *b = stack.entry[i].b;
   *c = stack.entry[i].c;
```

```
}   /* pop () */


void swap ( char *x, char *y )

{
   char t;

   t = *x;
   *x = *y;
   *y = t;

}   /* swap () */


void hanoi ( short n, char a, char b, char c )

{

   /*
   * Deplacement de N disques de A a C avec B comme intermediaire.
    */

   stack.count = 0;

  do {

      if ( stack.count != 0 ) {

         pop ( &n, &a, &b, &c );
           printf ( "Deplacement d'un disque de \"%c\" vers \"%c\"\n", a, c );
         n--;
         swap ( &a, &b );

     }

      while ( n > 0 ) {

         push ( n, a, b, c );
         n--;
         swap ( &c, &b );

     }

   } while ( stack.count > 0 );

}   /* hanoi () */


void main ( void )

{
```

```
    char buf[4];
    short n;

    /*
    * Scanf est bugge : on lit donc une chaine de caracteres que l'on decode
    * ensuite.
    */

    printf ( "Nombre de disques ? ");
    gets ( buf );
    n = atoi ( buf );

    hanoi ( n, 'A', 'B', 'C' );

}   /* main () */
```

```
/**
 ** Hanoi3 - programme de demonstration de la recursivite pour l'initiation
 **          au C 5 de GS Infos 21 : le celebre probleme des tours de hanoi.
 **
 **          Version graphique.
 **
 **          Philippe Manet        12 Avril 1992
 **
 **/

#include <stdio.h>
#include <orca.h>

#include <Types.h>
#include <QuickDraw.h>

#pragma optimize -1

#define MAX_DISKS   14
#define BASE        180             /* base des tours */
#define ITOWER      100
#define TOWER1      60              /* position X des tours */
#define TOWER2      ( TOWER1 + ITOWER )
#define TOWER3      ( TOWER2 + ITOWER )
#define HDISK       10              /* hauteur d'un disque */
#define LDISK       10              /* moitie longueur du plus petit disque */
#define IDISK       2               /* moitie diff longueur entre 2 disques */

Rect  pos_disk[3][MAX_DISKS];
short num_disk[3][MAX_DISKS];
short height[3];
short num_disks;


void draw_disk ( short tower, short disk, short flag )

{

   short x;

   if ( flag ) {

      SetSolidPenPat ( disk );
        PaintRect ( &pos_disk[tower][disk] );

  } else {

       EraseRect ( &pos_disk[tower][disk] );

      /*
       * Redessine la portion de tour effacee.
       */
```

```
        if ( pos_disk[tower][disk].v1 >= BASE - ( num_disks + 1 ) * HDISK ) {

          SetSolidPenPat ( 15 );
          SetPenSize ( 4, 1 );
           x = tower == 0 ? TOWER1 - 2 : tower == 1 ? TOWER2 - 2 : TOWER3 - 2;
            MoveTo ( x, pos_disk[tower][disk].v1 );
             LineTo ( x, pos_disk[tower][disk].v2 - 1 );
          SetPenSize ( 1, 1 );

      }

    }

}   /* draw_disk () */


void pause ( void )

{

   short w;

   for ( w = 0; w < 22222; w++ );

}   /* pause () */


void move_disk ( char from, char to )

{

   short tower, disk, dest_x, dest_y, move_i;

   /*
    * Deplacement graphique d'un disque.
    */

   tower = from - 'A';
     disk  =  num_disk[tower][height[tower]--];

   /*
    * Deplacement vers le haut sur la tour de depart.
    */

  do {

       draw_disk ( tower, disk, false );
        pos_disk[tower][disk].v1 -= HDISK;
        pos_disk[tower][disk].v2 -= HDISK;
       draw_disk ( tower, disk, true );
      pause ();

   } while ( pos_disk[tower][disk].v1 >= BASE - ( num_disks + 3 ) * HDISK );
```

```
    /*
     * Deplacement horizontal.
     */

     dest_x = ( to - from ) * ITOWER + pos_disk[tower][disk].h1;
    move_i = LDISK * ( to > from ? 1 : -1 );

   do {

        draw_disk ( tower, disk, false );
         pos_disk[tower][disk].h1 += move_i;
         pos_disk[tower][disk].h2 += move_i;
        draw_disk ( tower, disk, true );
       pause ();

     } while ( pos_disk[tower][disk].h1 != dest_x );

     /*
      * Deplacement vers le bas sur la tour d'arrivee.
      */

    tower = to - 'A';
      num_disk[tower][++height[tower]] = disk;
     pos_disk[tower][disk] = pos_disk[from - 'A'][disk];
    dest_y = BASE - ( height[tower] + 1 ) * HDISK;

   do {

        draw_disk ( tower, disk, false );
         pos_disk[tower][disk].v1 += HDISK;
         pos_disk[tower][disk].v2 += HDISK;
        draw_disk ( tower, disk, true );
       pause ();

     } while ( pos_disk[tower][disk].v1 < dest_y );

}   /* move_disk () */


void hanoi ( short n, char a, char b, char c )

{

    /*
    * Deplacement de N disques de A a C avec B comme intermediaire.
     */

   if ( n > 0 ) {

       hanoi ( n - 1, a, c, b );
       move_disk ( a, c );
       hanoi ( n - 1, b, a, c );
```

```
    }

}   /* hanoi () */


void main ( void )

{

    char buf[4];
    short d;

    /*
     * Scanf est bugge : on lit donc une chaine de caracteres que l'on decode
     * ensuite.
     */

    printf ( "Nombre de disques ? ");
    gets ( buf );
    num_disks = atoi ( buf );
    if ( num_disks > MAX_DISKS )
        exit ();

    /*
     * Initialisation du mode graphique et dessin des tours et de la base.
     */

    startgraph ( 320 );

    SetPenMode ( modeCopy );
    SetForeColor ( 15 );

    SetPenSize ( 1, 10 );
    MoveTo ( 0, BASE );
    LineTo ( 320, BASE );

    SetPenSize ( 4, 1 );
    MoveTo ( TOWER1 - 2, BASE );
    LineTo ( TOWER1 - 2, BASE - ( num_disks + 1 ) * HDISK );
    MoveTo ( TOWER2 - 2, BASE );
    LineTo ( TOWER2 - 2, BASE - ( num_disks + 1 ) * HDISK );
    MoveTo ( TOWER3 - 2, BASE );
    LineTo ( TOWER3 - 2, BASE - ( num_disks + 1 ) * HDISK );

    /*
     * Calcul des rectangles des disques et dessin initial.
     */

    for ( d = 0; d < num_disks; d++ ) {

        pos_disk[0][d].h1 = TOWER1 - ( LDISK + IDISK * ( num_disks - 1 - d ) );
        pos_disk[0][d].h2 = TOWER1 + LDISK + IDISK * ( num_disks - 1 - d );
```

```
        pos_disk[0][d].v1 = BASE - HDISK * ( d + 1 );
        pos_disk[0][d].v2 = BASE - HDISK * d;
        num_disk[0][d] = d;
          num_disk[1][d] = num_disk[2][d] = -1;

    }

     height[0] = num_disks - 1;
     height[1] = height[2] = -1;

    SetPenSize ( 1, 1 );
    SetSolidBackPat ( 0 );

    for ( d = 0; d < num_disks; d++ )
        draw_disk ( 0, d, true );

    hanoi ( num_disks, 'A', 'B', 'C' );

    pause ();

    endgraph ();

}   /* main () */
```

Programmation n°2 : Structure de données "queue"

Dans le précédent numéro de GS Infos, nous avons abordé dans la dernière partie de l'article sur la programmation de la calculatrice, la structure de données "pile". Dans cet article, nous allons traiter d'une autre structure de données, qui est en bien des points similaire. En fait, elle est même le pendant de la "pile" : il s'agit de la "queue". Dans la littérature, vous trouverez plus souvent employé le terme de "file" qui est une abréviation de "file d'attente", sans doute pour encore mieux montrer l'analogie entre les 2 structures de données. Personnellement, je préfère le terme de "queue" que je vais donc employer tout au long de cet article.

Concepts de Queue

La structure de données "queue" est la réalisation informatique du concept de queue tel qu'on le connaît dans la vie courante, par exemple lorsque vous faites la queue devant une salle de cinéma.

D'un point de vue informatique, on considère la queue comme étant une liste, telle que ce concept a été introduit dans le précédent article, et qui sera détaillé dans le prochain numéro de GS Infos.

La propriété fondamentale des queues est que tous les éléments sont insérés à une extrémité et retirés de l'autre. Le côté où l'on insére les éléments est appelé la "queue" (c'est ce qui a donné son nom à la structure de données), tandis que celui duquel on retire les éléments est appelé la "tête". Les termes anglais correspondants sont "head" ou "front" pour la tête et "tail" ou "rear" pour la queue.

L'expression employée pour désigner le mouvement des éléments dans la queue est "premier arrivé, premier sorti", ce qui correspond bien à la notion de file d'attente. Le terme anglais correspondant est "first in, first out" que l'on abrège par "FIFO".

Si vous relisez l'article sur les piles du précédent numéro, vous constaterez que ces 2 structures sont quasiment identiques; en fait, la seule différence est que dans le cas d'une pile, les insertions et les suppressions se font du même côté, tandis que, pour une queue, les suppressions se font du côté opposé aux insertions.

Par conséquent les opérations sont identiques entre les 2 structures, seuls leurs effets changent. Donc, comme pour les piles, il n'y a que 5 opérations possibles pour les queues :

- Initialisation d'une queue;
- Test si la queue est vide;
- Ajout d'un élément à la fin de la queue; la littérature, voulant conserver la similitude avec les piles, utilise le terme d'"enfiler", que je trouve personnellement assez malheureux; n'ayant pas de meilleur terme à vous proposer, je me contenterai du mot anglais "enqueue";
- Suppression d'un élément en tête de la queue; le terme employé est "défiler", ce qui est tout aussi mauvais que "enfiler"; le mot anglais "dequeue" est beaucoup plus précis.
- Accès au premier élement de la queue sans le retirer.

Comme pour les piles, on peut aussi envisager une sixième opération pour déterminer si une queue est pleine. Je vous rappelle que cette primitive est en fait liée à une contrainte d'implémentation et ne fait pas partie de la définition du type abstrait "queue". En l'occurence, le remplissage d'une queue (et aussi d'une pile) est en géneral assez problématique, puisqu'il peut rendre impossible un traitement.

La queue, comme la pile d'ailleurs, est une structure de données temporaire, c'est à dire que tous les éléments qui y sont insérés sont destinés à être supprimés. En fait, l'état stable d'une queue se produit lorsqu'elle est vide; dans le cas contraire, cela signifie qu'il y a encore des éléments à traiter. En général, les opérations d'insertion et de suppression sont asynchrones (c'est à dire indépendantes); le programme doit donc faire en sorte d'être capable d'absorber les éléments (c'est à dire les supprimer) suffisamment vite par rapport au rythme de leur insertion, afin de ne pas se laisser 'déborder'. Dans certains cas (par exemple dans des programmes récursifs, voir l'article d'initiation au C de ce numéro), le programme doit effectuer un traitement des éléments résiduels, une fois que tous les éléments à traiter ont été mis dans la queue et qu'une partie d'entre eux a été traitée lors du processus qui les a insérés.

Utilisations d'une queue

Comme la pile, la queue est une des structures de base que l'on utilise très souvent.

L'emploi le plus typique de la queue est lorsqu'un programme ne peut traiter les requêtes qui lui sont soumises aussi vite qu'elles arrivent : on a alors un mécanisme qui reçoit ces requêtes, les met dans la queue, tandis que le processus principal les prend dans l'ordre dans lequel elles sont arrivées afin de les traiter tour à tour, ce qui montre bien l'asynchronisme des opérations indiqué plus haut. Vous en avez un exemple dans votre GS, avec la queue des événements gérée par la boîte à outils. Ces événements, correspondant par exemple au mouvement de la souris, l'appui sur le bouton ou encore l'utilisation du clavier, arrivent en général plus rapidement que l'application ne peut les traiter. La boîte à outils les met donc en attente et les délivre dans l'ordre de leur survenance lorsque l'application les demande. On est bien ici dans le cas décrit ci-dessus, car on considére que l'état stable de la queue des événements est celui où elle est vide, indiquant que le programme est en attente d'une action de l'utilisateur.

Une autre utilisation des queues est effectuée dans les systèmes d'exploitation multitâches. Les tâches en attente du CPU sont mises dans une queue; lorsque la tâche précédente a terminé son travail, ou effectue une action n'ayant pas besoin du CPU (comme par exemple une entrée/sortie avec un contrôleur intelligent comme la RAMfast sur le GS), ou encore lorsque le système a décidé qu'elle devait laisser la place au suivant (auquel cas elle est remise à la fin de la queue), la première tâche est alors retirée de la queue et mise en exécution. Dans la pratique, on assigne une priorité à chacune des tâches; on peut alors envisager d'avoir une queue pour chacune des priorités possibles, le système ne mettant en exécution une tâche d'une priorité donnée que lorsque les queues correspondant aux priorités supérieures sont vides; on peut aussi envisager de n'avoir qu'une seule queue et d'insérer les tâches au milieu en fonction de leur priorité. Cette fonction du système est désigné en anglais par le terme de "scheduling". Dans ce cas particulier, la queue du CPU n'est jamais vide, ce qui est assez logique, car un

ordinateur fait toujours quelque chose; les systèmes d'exploitation prévoient en général une tâche spéciale qui a la priorité la plus faible et qui est assez souvent une boucle infinie (en basic, cela serait 10 GOTO 10), cette tâche n'étant exécutée que lorsque le CPU n'a rien a faire d'autre.

C'est cette utilisation des queues que j'ai choisi de vous montrer en exemple; vous trouverez sur ce GS Infos un programme appelé "Sched" (dont le source "Sched.cc" a été écrit – comme d'habitude – avec ORCA/C), effectuant une simulation d'un système d'exploitation multitâches, multi-utilisateurs (jusqu'à 4 dans la version compilée, mais c'est paramétrable), dans lequel chaque tâche a la même priorité. L'interface utilisateur est loin d'être sophistiquée, mais, à mon avis, le source est bien plus intéressant que le programme final, pour comprendre le fonctionnement d'une queue et d'un système d'exploitation multitâches (c'est dans ce but là que je l'ai écrit). Ce programme fait appel à une librairie "Queue.lib" (dont le source est "Queue.cc") qui a été conçue de façon générique, et que vous pourrez donc utiliser dans vos propres programmes. Notez que la suite de l'article présentant l'implémentation de cette structure de données contient des extraits du source de cette librairie.

Les queues, comme les piles, sont aussi employées dans les traitements différés, et notamment lorsqu'on souhaite préserver l'ordre des éléments traités (avec une pile, les éléments sont traités dans l'ordre inverse de leur rencontre); c'est par exemple le cas pour le parcours d'un graphe (nous aurons sans doute l'occasion d'en reparler). Là encore, tous les éléments de la queue sont traités, et donc supprimés à un moment donné; le traitement se termine lorsque la queue devient vide.

Je pense que lorsque nous aborderons des thèmes plus complexes dans les prochains numéros, nous reverrons souvent soit les piles, soit les queues, employées comme structures annexes dans l'implémentation des algorithmes. Ces futures utilisations des queues correspondront néanmoins le plus fréquemment aux cas cités ci-dessus, qui sont vraiment très typiques.

Représentation d'une queue

Comme je vous l'ai expliqué dans le précédent GS Infos, une structure de données peut être implémentée de différentes manières. Il est cependant clair que l'on va avoir besoin de maintenir un pointeur (dans le sens le plus général du terme) sur la tête de la queue et un autre sur sa fin, le premier permettant d'accéder à l'élément à supprimer, tandis que le second indiquera la position du nouvel élément à ajouter.

On pourrait utiliser un tableau, comme nous l'avons fait pour représenter les piles, et les 2 pointeurs décrits précédemment correspondraient à 2 indices de ce tableau, indiquant les positions d'ajout et de suppression.

Le seul problème est que la queue va avoir tendance à se 'déplacer' dans le tableau au fur et à mesure des ajouts et des suppressions, rendant le début du tableau inutilisé, tandis qu'on finira par atteindre sa fin, interdisant l'ajout de nouveaux éléments, et provoquant donc artificiellement une condition de queue pleine.

On peut sophistiquer ce principe, en rendant le tableau "circulaire", c'est à dire que lorsque l'indice de l'élément à ajouter atteint la limite supérieure du tableau, on lui fait prendre la valeur 1, puis on le fait progresser normalement jusqu'à ce qu'il rejoigne l'indice de l'élément à

supprimer, rendant ainsi la queue pleine; la taille de la queue est alors limitée par celle du tableau. Dans une telle implémentation, le tableau forme donc bien un anneau 'virtuel', et il n'a donc plus ni de début, ni de fin.

Le seul véritable inconvénient de cette solution est qu'elle limite arbitrairement la taille de la queue, ce qui peut poser un problème lorsque le rythme de mise en queue des éléments est nettement plus rapide que leur traitement (par principe, tous les éléments insérés dans une queue seront retirés à un moment ou à un autre), ce qui peut conduire au remplissage de la queue, et à la perte d'informations vitales.

Contrairement aux piles, on ne peut pas envisager la solution d'extension du tableau, telle que nous l'avons implémentée dans le précédent GS Infos, car si l'indice d'insertion a déjà fait le tour (c'est à dire qu'il a rejoint l'indice de suppression et que celui-ci n'est pas 1), ce qui est le cas le plus probable, il faudrait réorganiser complètement la queue dans le tableau, pour pouvoir bénéficier de l'espace supplémentaire. Si vous n'êtes pas sûrs d'avoir bien compris cette phrase, faites un dessin, cela deviendra tout de suite plus clair.

Pour nous affranchir de ces contraintes, nous allons étudier dans la suite de cet article une implémentation utilisant une liste linéaire chaînée simple. Nous verrons en détail dans le prochain article ce qu'est une telle structure de données. Pour l'instant, contentons-nous de savoir qu'il s'agit d'une structure chaînée, c'est à dire que chaque occurence de la liste est representée par un "nœud", et qu'un nœud pointe sur l'élément inséré dans la queue ainsi que sur le nœud suivant.

Une implémentation de la structure de données queue

Une structure C permettant de mettre en œuvre la structure chaînée décrite plus haut est alors :

```
typedef struct node *node;
typedef void *data_ptr;

struct node {
   data_ptr data;
   node     next;
};

typedef struct {
   node head;          /* tete de la queue */
   node tail;          /* queue de la queue */
} queue;
```

Le type node définit un nœud tel qu'il a été décrit ci-dessus, c'est à dire un pointeur sur l'élément à mettre dans la queue (ce pointeur étant générique comme dans l'implémentation de la pile du précédent numéro), ainsi qu'un pointeur vers le nœud suivant.

La structure de données queue est alors représentée par un pointeur sur le nœud de tête et un autre sur le nœud de queue, le pointeur sur le nœud suivant de la structure node assurant le chaînage des éléments de la queue.

L'initialisation de la structure queue se fait alors de la façon suivante :

```
boolean init_queue ( queue *Q )
{
   Q->head = Q->tail = NULL;
   return ( TRUE );
}
```

Cette fonction est on ne peut plus simple; la queue étant encore vide, il suffit d'initialiser les pointeurs sur la tête et la queue à 0 pour indiquer cet état.

Vous noterez que cette fonction (ainsi que toutes les autres d'ailleurs) accepte en paramètre un pointeur sur une queue. Le principe de la librairie est que le programme l'utilisant n'a pas à connaître le détail de cette structure; il doit la considérer comme opaque (à la limite elle pourrait avoir un type quelconque pour lui). Le passage du paramètre est nécessaire car le programme d'application peut vouloir gérer plusieurs queues avec la librairie; celle-ci ne doit donc pas fournir ses services en gérant une variable interne, et par conséquent en imposant une restriction sur le nombre de queues utilisables.

La fonction indiquant si une queue est vide ou non est aussi très simple. Elle se contente de tester si la tête pointe sur quelque chose ou non et de retourner le résultat obtenu.

```
boolean empty_queue ( queue *Q )
{
   return ( Q->head == NULL ? TRUE : FALSE );
}
```

L'insertion d'un élément à la fin de la queue est assez simple; elle fait appel à des techniques assez classiques de manipulation de pointeurs, les commentaires décrivant les opérations effectuées sur ces pointeurs, je ne les détaillerai pas plus :

```
boolean enqueue ( queue *Q, data_ptr data )
{
   node n;

   n = ( node ) malloc ( sizeof ( struct node ) );

   if ( n == NULL )
       return ( FALSE );

   n->data = data;
   n->next = NULL;

   if ( Q->head == NULL )
       Q->head = Q->tail = n;    /* Queue vide : noeud est le premier et le
       dernier */
   else {
       Q->tail->next = n;  /* Chainage du nouveau noeud a la suite du dernier
       actuel */
```

```
        Q->tail = n;              /* Le nouveau noeud devient le dernier */
    }

    return ( TRUE );
}
```

La suppression du premier élément d'une queue est encore plus simple que l'insertion; je vous laisse étudier les détails dans la fonction suivante :

```
boolean dequeue ( queue *Q, data_ptr *data )
{
    node n;

    if ( ( n = Q->head ) == NULL )
        return ( FALSE );

    *data = n->data;

    if ( Q->tail == Q->head )
        Q->head = Q->tail = NULL;   /* Queue ne contient qu'un element ->
        devient vide */
    else
        Q->head = n->next;   /* La tete de la queue devient le suivant du
        noeud retire */

    free ( n );

    return ( TRUE );
}
```

La dernière opération définie par le type abstrait queue permet d'obtenir l'élément en tête sans le retirer. L'utilisation de cette opération est assez rare, c'est pourquoi je ne l'ai pas implémentée dans la librairie citée plus haut. Voici tout de même une fonction effectuant ce travail :

```
node front ( queue *Q )
{
    return ( Q->head == NULL ? NULL : Q->head->data );
}
```

Sur la disquette GS Infos

Ces fonctions (sauf la dernière) ont été intégrées dans une librairie "Queue.lib" (dont le source est "Queue.cc"). Une démonstration de leur utilisation est réalisée dans le programme "Sched" (source "Sched.cc"). Une description succinte de ce programme a été donnée plus haut. Les commentaires au début du source décrivent en détail les principes mis en œuvre et le fonctionnement général du programme. Je ne peux que vous y renvoyer.

Dans le prochain numéro de GS Infos, nous traiterons la structure de données "liste", dont les structures pile et queue sont des cas particuliers.

```
/**
 ** Queue.cc - Routines de manipulation d'une queue.
 **            Cette librairie est decrite dans l'article "Programmation 2"
 **            de GS Infos numero 21 qu'elle accompagne.
 **
 **            v1.0        5 Avril 1992
 **
 **/

#pragma noroot
#pragma optimize -1

#include <Types.h>

#pragma lint -1

#include <stdlib.h>

#include "Queue.h"

/*
 * init_queue () - Initialisation de la queue passee en parametre.
 *                 Ceci consiste a initialiser les pointeurs de tete et
 *                 de queue a NULL, puisque la queue est encore vide.
 *                 On retourne TRUE si l'operation s'est bien deroulee et
 *                 FALSE dans le cas contraire. En fait, dans cette
 *                 implementation, ce sera toujours TRUE.
 */

boolean init_queue ( queue *Q )

{

   Q->head = Q->tail = NULL;
   return ( TRUE );

}  /* init_queue () */


/*
 * empty_queue () - Renvoie TRUE si la queue est vide et FALSE autrement.
 */

boolean empty_queue ( queue *Q )

{

   return ( Q->head == NULL ? TRUE : FALSE );

}  /* empty_queue () */


/*
 * enqueue () - Ajoute un element a la queue, donc necessairement a la fin.
 *              Retourne TRUE si cela a ete possible et faux dans le cas
 *              contraire, c'est a dire si on n'a pas pu allouer de memoire.
 */

boolean enqueue ( queue *Q, data_ptr data )
```

```
{

    node n;

    n = ( node ) malloc ( sizeof ( struct node ) );

    if ( n == NULL )
        return ( FALSE );

    n->data = data;
    n->next = NULL;

    if ( Q->head == NULL )

        /*
         * La queue ne contient aucun element. Le nouveau est donc a la fois
         * le premier et le dernier.
         */

        Q->head = Q->tail = n;

    else {

        /*
         * Chainage du nouveau noeud a la suite du dernier actuel.
         * Le nouveau noeud devient le dernier.
         */

        Q->tail->next = n;
        Q->tail = n;

    }

    return ( TRUE );

}  /* enqueue () */


/*
 * dequeue () - Suppression de l'element de tete qui est retourne a l'appelant.
 *              Retourne TRUE si cela a ete possible et faux dans le cas
 *              contraire, c'est a dire que la queue est vide.
 */

boolean dequeue ( queue *Q, data_ptr *data )

{

    node n;

    if ( ( n = Q->head ) == NULL )
        return ( FALSE );

    *data = n->data;

    if ( Q->tail == Q->head )

        /*
         * La queue ne contient que le seul element qui est retire.
         * Elle devient donc vide.
```

```
         */

        Q->head = Q->tail = NULL;

    else

        /*
         * La tete de la queue devient le suivant du noeud retire.
         */

        Q->head = n->next;

    free ( n );

    return ( TRUE );

}   /* dequeue () */
```

```
/**
 ** Queue.h - Definitions de la librairie de manipulation de la structure
 **           de donnees queue a inclure dans toute application utilisant
 **           cette librairie.
 **
 **           v1.0     5 Avril 1992
 **
 **/

/*
 * Definition des types noeud et queue.
 */

typedef struct node *node;
typedef void *data_ptr;

struct node {
   data_ptr data;
   node     next;
};

typedef struct {
   node head;          /* tete de la queue */
   node tail;          /* queue de la queue */
} queue;

/*
 * Prototypes des fonctions de la librairie.
 */

boolean init_queue ( queue * );
boolean empty_queue ( queue * );
boolean enqueue ( queue *, data_ptr );
boolean dequeue ( queue *, data_ptr * );
```

```
/**
 ** Sched.cc - Simulation de l'ordonnancement et de la soumission de travaux
 **            dans un systeme d'exploitation multitaches.
 **
 **            Le principe de la simulation est le suivant :
 **
 **            - une seule tache (job) peut etre executee a la fois par le CPU.
 **            - toutes les taches ont la meme priorite.
 **            - une nouvelle tache est demarree a un instant aleatoire et a
 **              une duree d'execution aleatoire. Ces taches sont lancees
 **              par des utilisateurs fictifs connectes a des terminaux
 **              (nombre aleatoire jusqu'a 4).
 **            - le simulateur gere sa propre horloge.
 **            - une tache est limitee a 20 unites de temps, au dela desquelles
 **              elle doit ceder sa place a la tache suivante (on dit qu'elle
 **              est preemptee) si elle n'a pas termine. Lorsque son tour
 **              revient, elle reprend la ou elle en etait reste.
 **            - un evenement est alors soit la soumission d'une nouvelle
 **              tache, soit la fin d'une tache, soit sa preemption.
 **            - l'objectif de la simulation est de mesurer le temps moyen
 **              qu'une tache attend dans la queue d'execution.
 **
 **            Ce programme constitue une demonstration de la structure de
 **            donnees "queue" decrite dans l'article "Programmation"
 **            du numero 21 de GS Infos.
 **
 **            v1.0    5 Avril 1992
 **
 **/

#pragma optimize -1

#include <Types.h>

#pragma lint -1

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#include "Queue.h"

#define END_JOB         0       /* numero evenement fin tache */
#define TIME_OUT        1       /* numero evenement expiration temps CPU */
#define MAX_TERMS       4       /* nb maximum de terminaux */
#define MAX_EVENTS      2 + MAX_TERMS       /* nb maximum evenements */

#define CPU_LIMIT      20       /* temps CPU maximal avant preemption */
#define MEAN_RUN       15       /* temps d'execution moyen d'une tache */
#define MEAN_SUBMIT    10       /* temps moyen de soumission */

#define BIG_TIME    30000       /* heure tres loin dans le futur */

#define MAX_JOBS      100       /* nombre total de taches simulees */

short system_clock;             /* horloge du simulateur */
short event_table[MAX_EVENTS]; /* table des evenements */
```

```
typedef struct {
   short num_term;          /* numero du terminal */
   short start_time;        /* heure de soumission de la tache */
   short queue_time;        /* temps d'attente dans la queue */
   short run_time;          /* duree d'execution de la tache */
} job;

job current_job;            /* la tache en cours d'execution */

short num_terms;            /* nombre de terminaux simules */

long total_queue_time;      /* duree totale de toutes les taches dans la queue */
short num_jobs;             /* nombre de taches executees */

boolean stop_simul;         /* TRUE si interruption simulation */

queue cpu_queue;            /* queue des taches en attente du CPU */

unsigned char *keyboard = (unsigned char *) 0xE0C000;
unsigned char *kbd_strobe = (unsigned char *) 0xE0C010;


/*
 * enqueue_job () - ajout d'une tache dans la queue du CPU.
 */

boolean enqueue_job ( short num_term, short start_time, short queue_time,
                      short run_time )

{

   job *j;

   j = (job *) malloc ( sizeof ( job ) );

   if ( j == NULL )
       return ( FALSE );

   j->num_term = num_term;
   j->start_time = start_time;
   j->queue_time = queue_time;
   j->run_time = run_time;

   if ( ! enqueue ( &cpu_queue, (data_ptr) j ) ) {
       free ( j );
       return ( FALSE );
   }

   return ( TRUE );

}  /* enqueue_job () */


/*
 * dequeue_job () - retrait d'une tache de la queue du CPU.
 */
```

```
boolean dequeue_job ( short *num_term, short *start_time, short *queue_time,
                      short *run_time )

{

   job *j;

   if ( ! dequeue ( &cpu_queue, (data_ptr *) &j ) )
       return ( FALSE );

   *num_term = j->num_term;
   *start_time = j->start_time;
   *queue_time = j->queue_time;
   *run_time = j->run_time;

   free ( j );

   return ( TRUE );

}  /* dequeue_job () */


/*
 * distribute () - calcul d'une distribution logarithmique autour d'une valeur
 *                 moyenne de facon a donner au generateur de nombres aleatoires
 *                 un ensemble de valeurs equi-propable.
 *                 La formule utilisee est : moyenne * ln(aleatoire), ce qui
 *                 fait qu'environ les 2/3 des nombres sont inferieurs a la
 *                 moyenne.
 *                 Notez qu'on calcule le log d'un nombre entre 0 et 1, et qu'il
 *                 est donc < 0, c'est pourquoi on le multiplie par -moyenne.
 */

short distribute ( short mean_time )

{

   return ( (short) ( -mean_time * log ( (double) rand () / (double) RAND_MAX ) ) );

}  /* distribute () */


/*
 * check_key () - controle qu'une touche n'a pas ete enfoncee, de facon a
 *                suspendre l'affichage, auquel cas on attend la frappe d'une
 *                autre touche pour continuer. Si de plus cette touche est ESC,
 *                on interrompt le programme.
 */

void check_key ( void )

{

   if ( *keyboard & 0x80 ) {        /* une touche a ete enfoncee */

       if ( ( *keyboard & 0x7F ) == 0x1B ) {  /* Escape */
           stop_simul = TRUE;
           return;
```

```
        }

        *kbd_strobe = 0;

        while ( ! ( *keyboard & 0x80 ) );    /* attente touche reprise */

        if ( ( *keyboard & 0x7F ) == 0x1B ) {   /* Escape */
            stop_simul = TRUE;
            return;
        }

        *kbd_strobe = 0;

    }

}   /* check_key () */


/*
 * next_event () - recherche de l'evenement ayant l'heure la plus petite, et
 *                 qui est donc celui a traiter.
 *                 On ajuste ensuite l'horloge du systeme et des autres
 *                 evenements pour simuler le temps qui passe.
 */

short next_event ( void )

{

    short event = 0, i, event_time;

    /*
     * Recherche du prochain evenement.
     */

    for ( i = 1; i < MAX_EVENTS; i++ )
        if ( event_table[i] < event_table[event] )
            event = i;

    /*
     * Ajustement de l'horloge des autres evenements.
     */

    event_time = event_table[event];

    for ( i = 0; i < MAX_EVENTS; i++ )
        event_table[i] -= event_time;

    system_clock += event_time;

    return ( event );

}   /* next_event () */


/*
 * start_job () - demarrage de la premiere tache de la queue si il n'y en a pas
 *                deja une d'active; dans le cas contraire, on ne fait rien.
```

```
 */

boolean start_job ( void )

{

   short num_term, start_time, queue_time, run_time;

   /*
    * Detection d'une tache courante.
    */

   if ( current_job.run_time != 0 )
       return ( TRUE );

   /*
    * Si la queue est vide, on n'a aucune tache a soumettre.
    */

   if ( empty_queue ( &cpu_queue ) )
       return ( TRUE );

   /*
    * Recuperation de la prochaine tache a executer et de ses parametres
    * temporels qui servent a ajuster le temps deja passe dans la queue, et
    * le temps d'execution de la tache courante.
    */

   if ( ! dequeue_job ( &num_term, &start_time, &queue_time, &run_time ) )
       return ( FALSE );

   current_job.num_term = num_term;
   current_job.run_time = run_time;
   current_job.queue_time = queue_time + system_clock - start_time;
   event_table[TIME_OUT] = CPU_LIMIT;
   event_table[END_JOB] = run_time;

   printf ( "%5d : demarrage tache du terminal %d, duree %d, a attendu %d\n",
            system_clock, num_term, run_time, current_job.queue_time );
   check_key ();

   return ( TRUE );

}  /* start_job () */


/*
 * submit_job () - soumission d'une tache en l'ajoutant a la fin de la queue
 *                 des taches en attente du CPU.
 *                 On essaye ensuite de demarrer la premiere tache de la queue
 *                 au cas ou ce serait celle que l'on vient d'ajouter.
 */

boolean submit_job ( short num_term )

{

   short run_time;
```

```
    if ( ! enqueue_job ( num_term - 1, system_clock, 0,
                                        run_time = distribute ( MEAN_RUN ) ) )
        return ( FALSE );

    printf ( "%5d : soumission tache du terminal %d, duree %d\n",
             system_clock, num_term - 1, run_time );
    check_key ();

    /*
     * On relance un nouveau job de ce terminal.
     */

    event_table[num_term] = distribute ( MEAN_SUBMIT );

    return ( start_job () );

}   /* submit_job () */


/*
 * finish_job () - fin d'execution d'une tache. On met a jour le temps total
 *                 passe dans la queue puis on lance la tache suivante.
 */

boolean finish_job ( void )

{

    event_table[END_JOB] = BIG_TIME;
    event_table[TIME_OUT] = BIG_TIME;

    num_jobs++;
    total_queue_time += current_job.queue_time;

    printf ( "%5d : fin tache du terminal %d, a attendu %d\n",
             system_clock, current_job.num_term, current_job.queue_time );
    check_key ();

    current_job.run_time = 0;
    return ( start_job () );

}   /* finish_job () */


/*
 * requeue_job () - preemption d'une tache qui a consomme tout le temps qui
 *                  lui etait imparti. Cette tache est remise a la fin de
 *                  la queue en tenant compte du temps passe dans le CPU
 *                  (pour qu'elle reprenne la ou elle en etait reste).
 */

boolean requeue_job ( void )

{

    short more_time;
```

```
    event_table[END_JOB] = BIG_TIME;
    event_table[TIME_OUT] = BIG_TIME;

    if ( ! enqueue_job ( current_job.num_term, system_clock,
                         current_job.queue_time,
                         more_time = current_job.run_time - CPU_LIMIT ) )
        return ( FALSE );

    printf ( "%5d : preemption tache du terminal %d, reste %d\n",
             system_clock, current_job.num_term, more_time );
    check_key ();

    current_job.run_time = 0;
    return ( start_job () );

}   /* requeue_job () */


/*
 * main () - programme principal.
 */

void main ( void )

{

    short num_event;

    /*
     * Initialisations de la queue du CPU, du generateur de nombres aleatoires,
     * de l'horloge, des statistiques, et du nombre de terminaux.
     */

    init_queue ( &cpu_queue );

    srand ( time ( NULL ) );

    total_queue_time = num_jobs = 0;
    system_clock = 0;
    current_job.run_time = 0;    /* indique aucune tache en cours */

    num_terms = ( rand () % ( MAX_TERMS - 1 ) ) + 2;
    printf ( "Simulation pour %d terminaux\n\n", num_terms );

    /*
     * Initialisation de la table d'evenements : fin du job et timeout cpu
     * tres loin dans le futur, et heure de soumission d'un premier job pour
     * chacun des terminaux simules.
     */

    event_table[END_JOB] = BIG_TIME;
    event_table[TIME_OUT] = BIG_TIME;

    for ( num_event = 2; num_event < num_terms + 2; num_event++ )
        event_table[num_event] = distribute ( MEAN_SUBMIT );

    for ( num_event = num_terms + 2; num_event < MAX_EVENTS; num_event++ )
        event_table[num_event] = BIG_TIME;    /* evenements inutilises */
```

```
    /*
     * Boucle tant que le nombre de jobs a simuler n'est pas atteint et
     * dispatching en fonction du premier evenement dans le temps.
     */

    stop_simul = FALSE;

    while ( ! stop_simul && num_jobs < MAX_JOBS )

        switch ( num_event = next_event () ) {

            case END_JOB  : finish_job ();
                            break;

            case TIME_OUT : requeue_job ();
                            break;

            default :       submit_job ( num_event );
                            break;

        }

    printf ( "\n\nNombre de jobs executes = %d\n"
             "Temps total dans la queue = %ld\n"
             "Temps moyen d'un job dans la queue = %f\n",
             num_jobs, total_queue_time,
             (double) total_queue_time / (double) num_jobs );

    printf ( "\nAppuyez sur une touche pour terminer : " );
    *kbd_strobe = 0;
    while ( ! ( *keyboard & 0x80 ) );
    *kbd_strobe = 0;

}   /* main () */
```

Programmation n°3 : les listes

Dans cet article, nous allons poursuivre l'étude des structures de données avec les "listes", qui sont une généralisation des "piles" et des "queues" dont nous avons traité dans les 2 précédents numéros de GS Infos.

Concepts de Liste

Une des structures de données très fréquemment utilisée est désignée sous le nom de "liste". Il s'agit de la même idée que celle de la liste que l'on peut établir dans la vie quotidienne, comme la liste des numéros de téléphone de ses amis, ou des choses à faire dans la semaine ...

De quoi s'agit-il ? Une "liste" est un ensemble ordonné d'éléments. Ordonné ne veut pas forcément dire trié, mais simplement qu'un ordre peut être déterminé par la manière dont les éléments sont insérés dans la liste. Par exemple, un nouvel élément pourrait être toujours ajouté en tête de la liste, ou en queue, ou à un autre endroit arbitraire.

A quoi sert une liste ? Comme son nom l'indique, une liste permet de représenter un ensemble d'objets similaires et dont le nombre peut ne pas être déterminé à l'avance, afin d'en dresser la liste. Ces objets peuvent avoir une relation entre eux, ou pas, selon les besoins. C'est un peu la structure fourre-tout, que l'on utilise lorsqu'une structure plus spécifique n'apporte pas grand chose de plus.

En tant que type de données abstrait, une liste possède un certain nombre de propriétés :

- Elle peut avoir Ø ou plus éléments.
- Un nouvel élément peut être ajouté à une liste à n'importe quel moment et n'importe où.
- N'importe quel élément d'une liste peut être supprimé à tout moment.
- Un élément quelconque d'une liste peut être accédé indépendamment des autres éléments.
- Une liste peut être traversée de façon à visiter successivement chacun de ses éléments.

Une des difficultés de l'implémentation du type liste est la variété des

opérations possibles qu'il peut subir, et qui dépendent en fait de l'utilisation particulière faite par le programme; par exemple et par définition, une liste peut posséder plusieurs opérations d'insertion, de suppression et d'accès d'un élément. C'est pourquoi nous ne verrons qu'un sous-ensemble des opérations possibles; celui-ci vous permettra cependant d'adapter les fonctions présentées à vos propres besoins.

Etant donné que les éléments sont ordonnés dans la liste, on peut distinguer pour chaque élément son "précédent" (le premier élément n'en aura pas, bien entendu), et son "suivant" (qui n'existera pas pour le dernier élément de la liste).

Il existe plusieurs catégories de listes. Ces catégories indiquent comment, à partir d'un élément donné, on peut éventuellement accéder à son précédent et à son suivant. Pratiquement, on distingue 4 catégories de listes qui sont en fait la combinaison de 2 grandes classes :

• La première classe indique si on peut distinguer ou non un premier et un dernier éléments. On dit qu'une liste est "linéaire" dans le cas où ces éléments existent, c'est à dire que le premier élément n'a pas de précédent et qu'un moyen externe à la liste permet d'y accéder; le dernier élément, lui, n'a pas de suivant. Au contraire, une liste est "circulaire" lorsqu'elle ne comprend pas de premier ni de dernier; d'un point de vue pratique, cela veut dire que chaque élément de la liste a un précédent et un suivant, formant en quelque sorte un anneau (par rapport à la liste linéaire, cela pourrait correspondre au fait que le suivant du dernier de cette liste est le premier, et par conséquent le précédent du premier est le dernier).

• La seconde classe indique si, à partir d'un élément quelconque, on peut accéder à son suivant et à son précédent ou à un seul des 2 éléments voisins. Lorsqu'un élément ne connaît qu'un seul voisin, on dit que la liste est "simple"; en général, il s'agit du suivant, mais ce n'est pas obligatoire. Pour accéder au précédent d'un élément d'une telle liste, on doit la parcourir depuis le début (dans le cas d'une liste linéaire - pour une liste circulaire, on part de l'élément actuel), jusqu'à l'élément voulu, tout en mémorisant le précédent de chaque élément visité. Lorsqu'on peut accéder aux 2 voisins d'un élément, on dit que liste est "double"; dans ce cas, on peut accéder à n'importe quel élément à partir de n'importe quel autre.

Ces 2 classes se combinent pour former les 4 catégories de liste évoquées plus haut :

• Les listes "linéaires simples" sont les plus proches des listes réelles

et de l'idée que l'on peut se faire d'une liste. Elles sont aussi les plus fréquemment employées (en tout cas par moi ;-) lorsqu'on utilise des structures chaînées (la manipulation de pointeur est relativement simple), et sont décrites en détail dans la suite de cet article, concernant notamment leur implémentation.

• Les listes "linéaires doubles" sont intéressantes lorsqu'on a besoin de parcourir une liste de façon non séquentielle (c'est à dire que l'on doit faire des retours fréquents vers des éléments déjà visités); en revanche, elles sont plus compliquées à implémenter dans le cas de listes chaînées (il faut manipuler 2 pointeurs).

• Les listes "circulaires" présentent l'avantage de permettre d'ajouter de nouveaux éléments en queue très rapidement; il suffit en effet de maintenir un pointeur sur le dernier élément, et grâce au lien vers le premier élément, on peut accéder très facilement à l'ensemble de la liste. Ces listes peuvent indifféremment être "simples" (n'avoir qu'un seul lien vers le suivant ou le précédent) ou "doubles". L'exemple le plus typique de l'emploi de ce type de liste est celui du langage "Lisp" (d'ailleurs le nom du langage est l'abréviation de "List Processor" ou traitement de listes).

Nous aborderons donc dans cet article la façon de créer une liste linéaire simple, d'insérer un élément en tête, en queue, et à un endroit quelconque de la liste, de supprimer le premier élément, le dernier ou un autre quelconque ainsi que la liste entière, d'accéder à un élément quelconque et de traverser complètement la liste.

Remarquez que pour l'instant, nous n'avons pas encore parlé de représentation d'une liste. C'est donc bien un type abstrait, puisque nous avons pu clairement définir les opérations et les propriétés liées à la liste, sans pour autant rentrer dans l'implémentation (même si j'ai utilisé le mot 'pointeur', cela ne fait pas forcément référence à une structure chaînée).

Représentation d'une liste

Une méthode couramment employée pour représenter une liste est l'utilisation d'un tableau. Cette solution est acceptable dans le cas où la liste subit peu de destructions d'éléments, car dans ce cas il faut remonter les éléments suivant celui détruit afin de boucher le trou ainsi créé, ce qui peut être une opération coûteuse. Les insertions se feront en général

uniquement en fin de liste, afin d'éviter le même inconvénient, ce qui peut nécessiter la mise en œuvre d'un algorithme de tri si l'on a besoin que les éléments aient un ordre plus prévisible.

Un autre problème de l'utilisation d'un tableau est que celui-ci a en principe une taille fixe prédéterminée (on peut contourner partiellement ce problème en allouant une zone mémoire pendant l'exécution), ce qui limite arbitrairement la taille de la liste, et est donc contraire aux propriétés énoncées plus haut.

Notez néanmoins qu'il n'y a pas de rapport direct entre la notion de tableau et celle de liste : la première est une implémentation possible de la seconde, qui est un type abstrait. On désigne cette implémentation par le terme de liste contiguë, ce qui indique que les éléments de la listes sont contigüs en mémoire.

Listes chaînées

Pour pallier à ces inconvénients, on emploie maintenant le mécanisme de pointeurs offert par des langages comme C ou Pascal. Par conséquent, on ne consomme de la mémoire que lorsqu'il est nécessaire d'ajouter un nouvel élément, cette mémoire étant libérée lorsque l'élément est détruit. La mémoire ainsi allouée doit pouvoir être combinée logiquement de façon à former une seule entité que constitue la liste. La seule limite de taille d'une telle liste devient la mémoire disponible.

Une structure de données ainsi définie est appelée "liste chaînée". Chaque élément est dénommé un "nœud" qui peut comprendre un certain nombre de champs, dont un défini par la structure et qui permet de pointer sur le nœud suivant (éventuellement le précédent) de la liste.
Si l'on veut pouvoir mettre n'importe quel type d'éléments dans la liste, un nœud ne doit compter que 2 rubriques : en plus du pointeur sur le nœud suivant, le nœud contient un pointeur générique vers l'information qu'il représente, cette information étant totalement indépendante de la structure de liste.

Les différents nœuds d'une liste chaînée ne sont donc pas nécessairement contigüs en mémoire (contrairement à la représentation d'une liste sous forme d'un tableau). Par conséquent, la recherche d'un élément donné dans la liste doit se faire en parcourant la liste depuis le début et en suivant les chaînages, jusqu'à trouver l'élément recherché.

Pour résumer, une liste chaînée et un nœud sont définis de la manière suivante :

- Une liste chaînée est un pointeur sur un nœud.
- Un nœud d'une liste comprend 2 rubriques :
  Un pointeur vers le nœud suivant.
  Un pointeur vers les données représentées par le nœud.

Une telle définition d'une liste chaînée est récursive, puisqu'une liste chaînée est un pointeur sur un nœud, qui à son tour pointe sur une liste chaînée. La récursivité s'arrête avec une liste vide qui est représentée par un pointeur NULL en C et NIL en Pascal. Cependant, s'agissant d'une récursivité finale (la récursivité a lieu après tous les traitements sur un nœud), elle peut être remplacée facilement par une itération (comme nous l'avons vu dans le précédent numéro), ce qui est nettement moins coûteux.

Les manipulations de pointeurs étant plus complexes que ce que nous avons vu pour les piles et les queues, je les ai représentées sous forme de figures. Vous trouverez ces figures dans les fichiers "Prog.3.Lst.F1.4" correspondant aux figures 1 à 4, et "Prog.3.Lst.F5.7" pour les figures 5 à 7. Ces fichiers sont au format "Apple Preferred" et peuvent être visualisées avec tout programme de dessin tel que "Platinum Paint". Dans ces figures, les flèches en gras représentent les chaînages créés par l'opération matérialisée par la figure, tandis que les flèches en pointillés correspondent aux chaînages supprimés par cette même opération.

La figure 1 montre la représentation graphique d'une liste. On y voit bien la tête de la liste représentée par L et la fin de la liste matérialisée par le symbole NULL. Pour simplifier la figure, la donnée a été placée directement au niveau du nœud, alors que d'après la définition précédente, elle aurait dû être pointée.

Une implémentation d'une liste chaînée

Pour changer un peu, et ne pas être accusé de sectarisme, les exemples ci-dessous seront tantôt en C, tantôt en Pascal.

Une liste chaînée telle qu'elle vient d'être définie peut se déclarer de la manière suivante, selon le langage :

- En C :

```
typedef void *data_ptr;
typedef struct node *list;
struct node {
        data_ptr data;
        list     next;
};
```

• En Pascal :

```
type data_ptr = ^char;
     list = ^node;
     node = record
              data : data_ptr;
              next : list;
            end;
```

Vous voyez que les définitions sont identiques en C et en Pascal (aux différences syntaxiques de chaque langage près).

Les routines qui vont être amenées à manipuler cette structure n'auront bien entendu aucune idée de ce que représente le pointeur "data". De plus, la zone pointée devra être gérée à côté de la liste, en général par le programme qui utilise ces routines, et cela de façon spécifique au type d'informations à mettre dans la liste.

Du point de vue de la structure de données, c'est l'idéal, puisque l'on a bien séparé la représentation du type des données spécifiques à prendre en compte. Du point de vue de la programmation, et du GS (qui n'est tout de même pas une machine très puissante), l'utilisation de plusieurs couches de fonctions n'est sans doute pas aussi souhaitable, d'autant plus que la liste est un type employé très fréquemment et que les opérations ne sont pas très compliquées. C'est pourquoi je vous recommande vivement d'intégrer les mécanismes décrits ici dans une structure comprenant à la fois vos données et le pointeur vers le nœud suivant et d'utiliser directement les algorithmes présentés aux bons endroits dans vos programmes. La forme présentée ici reste malgré tout utile pour la clarté des explications.

Création d'une liste

Une nouvelle liste étant une liste vide, et cette dernière étant représentée par un pointeur nul, la création de cette liste est très simple : il suffit d'initialiser le pointeur sur la tête de la liste à NULL en C, et à NIL en Pascal. Le programme d'application va donc appeler une routine d'initialisation

de liste qui va se contenter de retourner ce pointeur nul. Par exemple, en Pascal (en supposant que l'on dispose d'une UNIT gérant les listes) :

```
USES listes;
VAR ma_liste : list;
BEGIN
        ma_liste := create_list;
END;
```

La fonction create_list étant simplement :

```
FUNCTION create_list : list;
BEGIN
        create_list := NIL;
END;
```

Bien entendu, le programme aurait pu lui-même initialiser sa liste à NIL (et il devra le faire si il intègre directement les manipulations sur la liste); cependant, l'initialisation de la liste aurait pu contenir d'autres instructions nécessaires à la librairie et que le programme n'a pas à connaître; d'autre part, pour le programme, la liste est un type "opaque" (c'est à dire qu'il ne connaît pas la manière dont le type est implémenté), qu'il ne sait donc pas initialiser.

Insertion d'un élément

Maintenant que nous disposons d'une liste (bien qu'encore vide), nous pouvons lui ajouter des données qui seront donc représentées chacune par un nœud. Ces données peuvent être insérées dans la liste, soit en tête des nœuds déjà existants, soit en queue, soit enfin au milieu selon une règle déterminée.

Le cas d'insertion le plus simple est en tête de la liste; en effet, il suffit alors de créer le nouveau nœud (c'est à dire d'allouer la mémoire nécessaire à son stockage) puis d'assigner ce nœud au pointeur représentant la liste, sans oublier d'avoir fait pointer le suivant de ce nouveau nœud vers l'ancienne tête de liste (c'est à dire le premier nœud de cette liste).
Cela a l'air bien compliqué, alors que l'exemple Pascal ci-dessous montre qu'en fait c'est très simple (cette procédure est illustrée par la figure 2) :

```
PROCEDURE prepend_node ( var L : list; data : data_ptr );
```

```
VAR node : list;                    (* déclaration d'un noeud *)
BEGIN
      NEW ( node );                 (* création du noeud *)
       node^.data := data;          (* enregistrement de la donnée *)
       node^.next := L;             (* le suivant pointe sur l'ancienne liste *)
      L := node;                    (* noeud devient la nouvelle tête de liste *)
END;
```

Vous noterez que le traitement d'une liste précédemment vide n'est pas isolé, car dans ce cas l'affectation de l'ancienne tête de liste au suivant du nouveau nœud résulte en l'initialisation de ce dernier à NIL, constituant ainsi une liste de un élément (ce cas se produit uniquement lors de la première insertion). L'appel à cette procédure se fait alors simplement par :

```
prepend_node ( ma_liste, data_ptr(@ma_donnee) );
```

Dans la pratique, ce devrait être une fonction booléenne afin qu'elle puisse indiquer le succès ou l'échec de l'insertion (qui ne peut se produire que lorsqu'il n'y a plus assez de mémoire).

L'insertion en fin de liste est un petit peu plus compliquée puisque la seule information dont on dispose est le pointeur sur la tête de liste; il nous faut donc parcourir la liste jusqu'au dernier nœud, puis le faire pointer vers le nouveau nœud (le dernier nœud contient NIL avant l'ajout). Ce nouveau nœud doit lui pointer sur NIL, car il devient effectivement le dernier de la liste. Nous devons ici traiter séparément le cas d'une liste précédemment vide, car on ne pourra pas la parcourir; la liste devient simplement le nœud (encore une fois, ce cas ne se produit que lors du premier ajout). Ecrit en Pascal, cela devrait être un peu plus compréhensible (la figure 3 résume le déroulement des opérations) :

```
PROCEDURE append_node ( var L : list; data : data_ptr );
VAR node, last : list;
BEGIN
      NEW ( node );                 (* création du nouveau noeud *)
       node^.data := data;          (* enregistrement de la donnée *)
       node^.next := NIL;           (* ce noeud sera le dernier de la liste *)
     IF L = NIL THEN
            L := node               (* la liste devient le nouveau noeud *)
    ELSE BEGIN
            last := L;              (* recherche du dernier noeud actuel *)
           WHILE last^.next <> NIL DO last := last^.next;
           last^.next := node;              (* chainage du nouveau noeud *)
```

```
    END;
END;
```

L'appel à cette procédure se fait alors simplement par :

```
apppend_node ( ma_liste, data_ptr(@ma_donnee) );
```

Dans la pratique, ce devrait être une fonction booléenne afin qu'elle puisse indiquer le succès ou l'échec de l'insertion (qui ne peut se produire que lorsqu'il n'y a plus assez de mémoire).

L'insertion en milieu de liste est en fait une généralisation de l'insertion en queue, puisqu'au lieu de chercher le dernier nœud, on va en chercher un quelconque, mais selon une règle bien définie. L'objectif de ce type d'insertion est de minimiser le parcours de la liste lorsque l'on a besoin d'accéder à l'un de ses nœuds; par conséquent, l'insertion d'un élément au milieu obéit très souvent à une règle de tri (par exemple dans l'ordre croissant ou alphabétique selon le type de données mis en liste), ainsi, on pourra arrêter la recherche dès que l'on aura trouvé un élément "après" celui recherché. Bien entendu, ce type d'insertion ne peut pas s'employer simultanément avec les 2 précédents, car ils fausseraient l'ordre établi.

Pour insérer un nouveau nœud au milieu, il nous faut rechercher celui qui se situera juste "après" après cette insertion (c'est à dire celui qui lui est immédiatement supérieur), sachant qu'il ne peut y en avoir aucun si le nouvel élément est le plus grand de tous (auquel cas, il se retrouvera en queue de liste); nous devons aussi mémoriser le nœud précédent celui trouvé, car notre nouveau nœud s'insérera juste entre les 2, c'est à dire que le pointeur vers le nœud suivant de ce nœud "précédent" sera changé pour pointer vers le nouveau nœud, dont le pointeur sur le suivant pointera vers le nœud "supérieur" (ce nœud était anciennement pointé par le suivant du nœud "précédent").
La seule vraie difficulté est la détermination du point d'insertion, puisque nous ne savons pas ce que représente la donnée; il nous faudra donc appeler une fonction du programme utilisateur passée en paramètre.
Voyons sur un exemple (en C pour changer) ce que tout cela donne (et sur la figure 4 pour une représentation graphique du processus) :

```
void insert_node ( list *L, data_ptr data, short (*compare)() )
{
        list node, current, previous;

         node = (list) malloc ( sizeof ( struct node ) );       /* création noeud */
```

```
    node->data = data;                          /* enregistrement de la donnée */
      /* recherche du noeud "supérieur" et mémorisation du "précédent" */
    for ( current = *L, previous = NULL;
          current != NULL && (*compare) ( current->data, data ) < 0;
           previous = current, current = current->next );
    if ( previous == NULL )
        *L = node;                              /* Insertion en tete de liste */
    else
        previous->next = node;                  /* Insertion entre previous et current */
    node->next = current;                       /* nouveau noeud pointe sur "supérieur" */
}
```

C'est un petit peu plus compliqué que pour les insertions précédentes.
Le paramètre (*compare)() définit un pointeur sur une fonction externe qui est appelée dans la boucle for. C'est cette fonction qui va déterminer la position à laquelle sera inséré le nouveau nœud. On lui passe 2 paramètres : le premier indique la donnée contenue dans le nœud actuellement traversé et le second la donnée à insérer; elle nous retourne un entier indiquant l'ordre relatif de ces données :

- < Ø si le nœud actuel de la liste se trouve logiquement avant celui à insérer.
- = Ø si les 2 données sont identiques : on peut décider soit d'insérer le nouveau nœud avant ou après celui identique (ici c'est avant) ou encore de rejeter les doublons.
- > Ø si le nœud actuel de la liste se trouve logiquement après celui à insérer.

La boucle précédente recherche donc le premier nœud dont la donnée est immédiatement supérieure (ou éventuellement identique) à celle du nœud à insérer. Bien évidemment, on s'arrête aussi lorsque l'on arrive en fin de liste sans avoir satisfait la comparaison.
Le cas d'une liste vide n'est pas traité spécifiquement puisque la boucle s'arrêtera dès le premier passage ("current" sera NULL).
La mémorisation du nœud précédent se fait simplement en recopiant le pointeur sur le nœud actuellement traité avant de passer au suivant; le pointeur sur ce précédent est d'abord initialisé à NULL pour détecter le cas où le premier élément satisfait la comparaison (ou que la liste est vide).
L'insertion est ensuite un simple jeu de pointeurs : le nœud mémorisé en tant que précédent (ou la tête de liste si il est nul, indiquant une insertion en première position) pointe sur le nouveau nœud qui pointe à son tour sur le nœud qui doit être son suivant (si le nouveau nœud doit être inséré à la fin de la liste ou que celle-ci est vide, le pointeur

suivant sera bien nul, et satisfera donc les règles).

L'appel à cette procédure se fait alors simplement par :

```
insert_node ( ma_liste, data_ptr(@ma_donnee), ma_compare_routine );
```

Dans la pratique, ce devrait être une fonction booléenne afin qu'elle puisse indiquer le succès ou l'échec de l'insertion (qui ne peut se produire que lorsqu'il n'y a plus assez de mémoire).
Il faudra aussi avoir écrit une fonction spécifique ma_compare_routine qui pourrait ressembler à :

```
short ma_compare_routine ( data_ptr data1, data2 )
{
        if ( data1 < data2 )
            return ( -1 );
        else if ( data1 == data2 )
            return ( 0 );
       else
            return ( 1 );
}
```

Bien sûr, il ne s'agit que d'un exemple, puisque data1 et data2 sont en principe des pointeurs vers les véritables données.

Suppression d'un nœud

Comme pour l'insertion, on peut envisager plusieurs méthodes de suppression d'un nœud : soit le premier, soit le dernier, soit un autre quelconque.

Encore une fois, la suppression du premier nœud est la plus simple puisqu'on peut l'accéder directement. Il suffit de faire pointer la tête de la liste vers le suivant de la liste (c'est à dire le suivant du premier nœud) puis de libérer la mémoire occupée par ce premier nœud. Le deuxième élément devient ainsi la nouvelle tête de liste; si la liste n'en comprenait qu'un, le résultat de cette opération donne une liste vide. Si la liste est vide au départ, une erreur doit être signalée, car il n'y a bien évidemment rien à supprimer. Tout ceci donne en Pascal, et sur la figure 5 :

```
FUNCTION delete_first ( var L : list ) : data_ptr;
VAR node : list;
BEGIN
      IF L = NIL THEN delete_first := NIL  (* erreur - liste vide *)
   ELSE BEGIN
           node := L^.next;                    (* préservation nouvelle tête de liste *)
           delete_first := L^.data;          (* on retourne la donnée *)
         DISPOSE ( L );
           L := node;                          (* liste démarre au second élément *)
   END;
END;
```

Le pointeur vers le deuxième nœud doit être préservé dans une variable temporaire avant de libérer la mémoire occupée par le premier, puisqu'après cette libération, on ne peut plus légalement accéder aux champs de la structure. Cette fonction retourne la donnée (en principe un pointeur sur celle-ci) afin que le programme puisse libérer la mémoire qu'elle occupe.

La destruction du dernier nœud d'une liste est à peine plus compliquée : il suffit de rechercher l'avant-dernier nœud puis de faire pointer son suivant sur NIL avant de libérer la mémoire occupée par le dernier. Si la liste ne comprend qu'un seul nœud, il n'y aura pas d'avant-dernier, et la liste retournée sera alors vide. C'est bien entendu une erreur de supprimer un nœud lorsque la liste est vide. Enfin, on retourne la donnée pour libération éventuelle de la mémoire qu'elle occupe par le programme. Traduit en Pascal et graphiquement sur la figure 6, tout ceci donne :

```
FUNCTION delete_last ( var L : list ) : data_ptr;
VAR node, previous : list;
BEGIN
      IF L = NIL THEN delete_last := NIL                (* erreur - liste vide *)
    ELSE BEGIN
           previous := NIL;
          node := L;
          WHILE node^.next <> NIL DO BEGIN
                 previous := node;                       (* avant dernier noeud *)
                 node := node^.next;                     (* dernier noeud *)
          END;
           IF previous = NIL THEN
                 L := NIL                                (* liste d'un seul noeud devient vide *)
          ELSE
                  previous^.next := NIL;        (* suppression dernier noeud *)
    END;
END;
```

La suppression d'un nœud quelconque d'une liste est la généralisation des 2 fonctions précédentes. Le programme doit spécifier la donnée correspondant au nœud à supprimer ainsi qu'une fonction booléenne permettant de comparer successivement la donnée de chacun des nœuds avec celle recherchée; ici, nous avons juste besoin de connaître l'identité des données sans relation d'ordre. De notre côté, nous devons mémoriser le nœud précédent de celui recherché, et lorsqu'il est trouvé, par un simple jeu de pointeurs, il nous faut le faire pointer sur le suivant du nœud à supprimer. Nous retournerons NULL si le nœud n'est pas trouvé (ceci traite aussi du cas d'une liste vide) et la donnée lorsqu'il a été détruit afin que le programme libère la mémoire occupée par cette dernière. Voyons ce que tout cela donne en C et sur la figure 7 :

```
data_ptr delete_node ( list *L, data_ptr data, boolean (*compare)() )
{
        list node, previous;

          /* recherche du noeud à supprimer et mémorisation du précédent */
        for ( node = *L, previous = NULL;
              node != NULL && ! (*compare) ( node->data, data );
              previous = node, node = node->next );
        if ( node == NULL )                    /* Noeud inexistant */
           return ( NULL );
        if ( previous == NULL )
           *L = node->next;                    /* Suppression du premier noeud */
```

```
        else
                previous->next = node->next;      /* Suppression autre noeud */
        data = node->data;
        free ( node );

        return ( data );
}
```

La fonction compare utilisée est ultra-simple. Elle peut s'écrire par exemple :

```
boolean mon_compare ( data_ptr data1, data2 )
{
      return ( data1 == data2 );
}
```

Il vous faudra bien sûr substituer data1 et data2 par les véritables données.

Destruction complète d'une liste

Après ce que nous venons de voir, vous devriez imaginer assez facilement les actions à effectuer : il nous faut parcourir la liste, puis pour chaque élément, appeler une fonction externe libérant la mémoire occupée par la donnée. Nous devrons aussi mémoriser dans une variable temporaire le nœud suivant celui à détruire, car nous n'avons pas le droit d'accéder à ses champs, une fois que la mémoire qu'il occupait a été libérée. Si la liste est vide, on ne fait rien, mais on ne signale pas non plus d'erreur. Après cette opération, la liste est complètement réinitialisée.

Ce qui donne en C :

```
void destroy_list ( list *L, void (*delete_data) ( data_ptr ) )
{
        list node;

        if ( *L != NULL )
             /* boucle sur chacun des noeuds de la liste
                  node pointe sur le noeud suivant celui à détruire */
             for ( node = (*L)->next; *L != NULL; ) {
                   /* libération mémoire occupée par la donnée */
                  (*delete_data) ( (*L)->data );
                  free ( *L );              /* libération mémoire occupée par le noeud */
                  if ( ( *L = node ) != NULL )                    /* si pas dernier noeud */
```

```
                node = node->next;                    /* on va le supprimer */
        }
}
```

Parcours de liste

En dehors des insertions et des suppressions de nœuds, nous avons besoin d'être capables d'accéder à l'un d'entre eux et de parcourir complètement l'ensemble de la liste pour effectuer un traitement sur chacun de ses nœuds (par exemple pour les visualiser). En fait, nous avons déjà quasiment vu ces opérations dans les routines précédentes. Nous pouvons donc écrire les 2 fonctions suivantes en C :

```
boolean find_node ( list L, data_ptr data, boolean (*compare)() )
{
      list node;

      for ( node = L; node != NULL && (*compare)(node->data,data); node = node->next
)
     return ( node != NULL );
}
```

La fonction "compare" est identique à celle utilisée par "delete_node". Elle renvoie TRUE si le nœud existe et FALSE dans le cas contraire. Nous ne faisons que répercuter ce résultat.

```
void traverse_list ( list L, void (*process_data)() )
{
      list node;

      for ( node = L; node != NULL; node = node->next )
            (*process_data)(node->data);
}
```

Pour chacun des nœuds de la liste, on appelle une fonction externe "process_data" qui effectue le traitement approprié sur la donnée. Telle que cette fonction est écrite, on ne peut pas s'arrêter en cours de route (si par exemple la fonction externe considère qu'elle a terminé son travail après un certain nombre de nœuds).

Pour pallier à ce problème, on peut traverser la liste en gardant le contrôle, au lieu de le passer à la fonction de parcours. Pour cela, on

utilise un mécanisme que l'on désigne sous le nom de "contexte". Lors du premier appel, le programme appelant l'initialise à NULL pour indiquer à la routine de se placer en début de liste. Ensuite, cette variable est gérée automatiquement par la routine de parcours, et on ne doit surtout pas la changer. Une fonction de parcours basée sur ce principe pourrait être écrite ainsi en Pascal :

```
FUNCTION get_node ( L : list; var context : list ) : data_ptr;
BEGIN
      IF context = NIL THEN
           context := L                              (* initialisation au debut de la liste *)
     ELSE
            context := context^.next;      (* element suivant de la liste *)
      IF context = NIL THEN
           get_node := NIL                          (* arrive en fin de liste *)
     ELSE
            get_node := context^.data;    (* retourne donnée *)
END;
```

Le fait de passer le contexte en paramètre (plutôt que d'utiliser une variable interne à la librairie) permet de parcourir simultanément plusieurs listes. Avec cette approche, on peut arrêter le parcours lorsque cela est nécessaire.

Autres opérations

Nous avons fait le tour des principales opérations utilisées avec les listes. On pourrait aussi ajouter une fonction vérifiant si une liste est vide, ce qui s'écrirait en Pascal :

```
FUNCTION empty_list ( L : list ) : boolean;
BEGIN
        empty_list := L = NIL;
END;
```

Les autres fonctions que l'on pourrait développer sont :

• Obtention directe du premier et du dernier élément (cas spécifiques de la fonction find_node précédente).
• Concaténation de 2 listes.
• Copie d'une liste, avec ou sans copie des données (c'est à dire que les 2 listes pourraient pointer sur les mêmes données; attention toutefois aux

problèmes que cela pose en cas de destruction d'un nœud).
• Comparaison de 2 listes, soit avec les éléments à la même position dans chacune des listes, soit en vérifiant que les mêmes éléments existent dans les 2 listes, même si ils ne sont pas dans le même ordre.
• Et tout ce qui peut vous passer par la tête ...

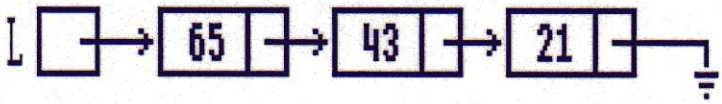
Le plus gros inconvénient des listes linéaires simples est la nécessité d'effectuer un parcours de cette liste quasi-complet à chaque fois que l'on souhaite accéder à un élément, ce qui peut poser un très gros problème si la liste est très longue. Dans un prochain article, nous verrons des méthodes permettant de pallier à ces problèmes.

Sur la disquette

Sur la disquette GS Infos, vous trouverez une librairie complète utilisable telle quelle et réalisant l'ensemble des opérations décrites tout au long de cet article, ainsi que quelques unes de celles décrites dans le paragraphe précédent (les autres sont laissées en exercice au lecteur). Cette librairie vous est proposée sous 2 versions : l'une écrite en ORCA/C, l'autre en ORCA/Pascal, le tout dans le sous-catalogue "/Prog.3". Chacune est mise en œuvre par un programme de démonstration. Le sous-catalogue "/Prog.3/Sources" contient les sources des librairies et des programmes de démonstration dans chacun des langages, tandis que le sous-catalogue "/Prog.3/Librairies" contient les librairies avec lesquelles vous pourrez linker vos propres programmes, ainsi que l'interface de l'UNIT pour ORCA/Pascal.

Vous avez sans doute remarqué qu'à chaque fois qu'une fonction était passée en paramètre, je vous ai montré du code C au lieu du Pascal comme pour les autres (d'ailleurs, j'espère ne pas avoir mélangé les 2 dans le texte; heureusement les librairies sont bien dans le bon langage !). La raison en est qu'en général Pascal ne dispose pas de cette possibilité (c'est le cas de TML Pascal II). Cependant, ORCA/Pascal permet de passer une procédure ou une fonction en paramètre; ainsi la librairie Pascal contient exactement la même chose que son homologue C, mais elle n'est pas utilisable telle quelle avec TML Pascal II. Je pense que ce n'est pas trop grave, car elles sont avant tout faites pour être démontées et copiées/collées dans vos propres programmes (essentiellement pour des raisons de performance), si bien que vous n'aurez plus besoin de cette facilité.

Figure 1 : représentation d'une liste chaînée



Tête de la liste

Fin de la liste = NULL

Figure 2 : insertion d'un nœud en tête d'une liste.

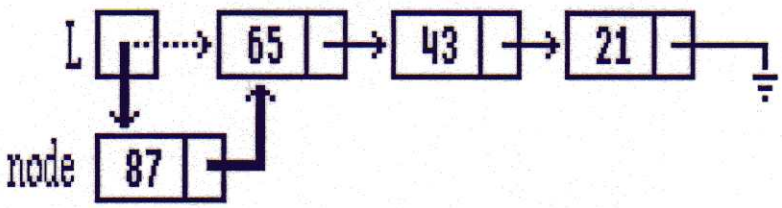


Figure 3 : insertion d'un nœud en fin de liste

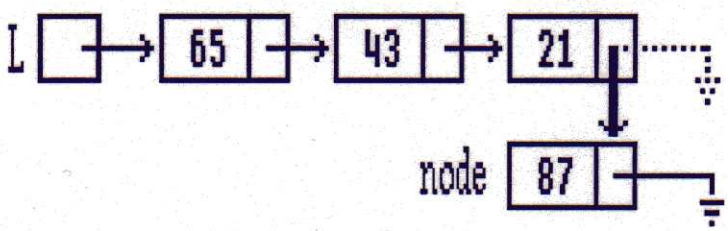


Figure 4 : insertion d'un nœud au milieu d'une liste

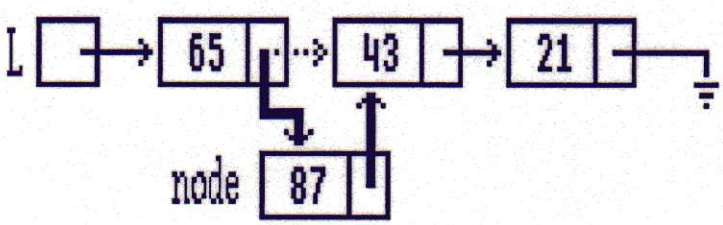


Figure 5 : Suppression du premier nœud d'une liste

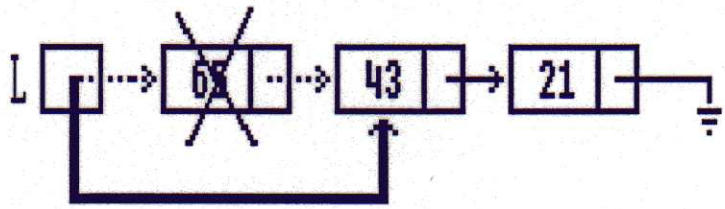


Figure 6 : Suppression du dernier nœud d'une liste

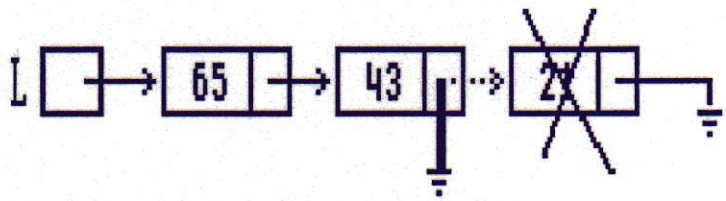
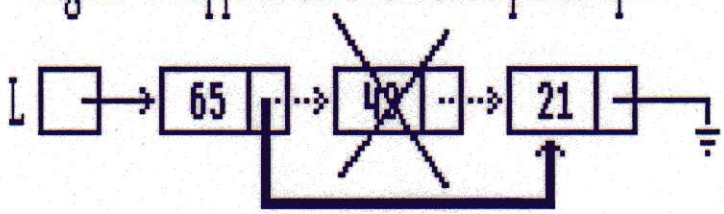


Figure 7 : Suppression d'un nœud quelconque



Programmation n°4 : manipulations de bits

Avec cet article, nous allons faire une petite pause dans l'étude des structures de données. Je vous propose en effet de voir en détail les opérateurs de manipulation de bits, qui nous seront fort utiles pour la suite de cette série, notamment dans le prochain article.

Cette discussion sera basée sur les opérateurs disponibles dans le langage C (comme d'habitude). Toutefois, tout ce que je vais raconter est applicable en Pascal, les 2 compilateurs disponibles sur le GS offrant, à titre d'extension, les mêmes possibilités que le C (pour ORCA/Pascal, les opérateurs sont strictement identiques à ceux de ORCA/C, tandis que TML Pascal définit des fonctions pour chacun de ces opérateurs). Même si votre langage de prédilection est l'assembleur, je pense que vous trouverez des informations pertinentes sur le sujet; d'ailleurs, une partie du code accompagnant cet article est en assembleur (eh oui! tout peut arriver !).

Rappels sur les opérateurs de manipulation de bits

C dispose de 6 opérateurs spécifiques travaillant au niveau du bit, c'est à dire que chaque bit du mot (de 8, 16 ou 32 bits) qui le contient est considéré indépendamment des autres bits de ce mot, et que ces opérateurs permettent d'agir individuellement sur 1 ou plusieurs bits. Ces opérateurs correspondent d'ailleurs à des instructions que l'on trouve sur la plupart des processeurs.
En voici la liste :

• "&" effectue un "et" binaire : chaque bit du résultat vaut 1 si et seulement si les 2 bits correspondant (ie à la même position) des opérandes sont à 1; autrement le bit du résultat vaudra Ø. Par constraste, l'opérateur "&&" effectue un "et" logique, c'est à dire que le résultat vaut 1 si les 2 opérandes sont non nuls,sinon Ø. Cet opérateur correspond à l'instruction "AND" du 65C816.

• "|" effectue un "ou" binaire : chaque bit du résultat vaut Ø si et seulement si les 2 bits correspondant des opérandes valent Ø; si l'un des bits des opérandes vaut 1, le bit résultat vaudra 1. L'opérateur "||" effectue un "ou" logique, c'est à dire que le résultat vaut 1 si l'un des 2 opérateurs est non nul, autrement il vaut Ø. Cet opérateur correspond à l'instruction "ORA" du 65C816.

• "^" effectue un "ou exclusif" binaire : chaque bit du résultat vaut 1 si et seulement si les 2 bits correspondant des opérandes sont opposés (ie l'un vaut Ø et l'autre 1); si les 2 bits considérés sont identiques, le résultat vaudra Ø. Il n'y a pas d'opérateur logique équivalent. Cet opérateur correspond à l'instruction "EOR" du 65C816.

• "~" effectue un "complément à 1" : chaque bit du résultat vaut 1 si le bit correspondant de l'opérande est Ø, et vice-versa. Par contraste, l'opérateur "!" effectue un "non" logique, c'est à dire que le résultat vaudra Ø si l'opérande est non nul, et 1 si il est nul. L'opérateur unaire "-", non spécifique aux manipulations de bits, réalise le "complément à 2"; essentiellement, il effectue l'opération suivante : résultat = ~ opérande + 1. Cet opérateur n'a pas d'instruction machine équivalente, mais peut être réalisé facilement en effectuant un "EOR $FFFF".

• "<<" effectue un "décalage à gauche" : l'opérande de gauche est décalé à gauche du nombre de bits spécifié par l'opérande de droite; autrement dit, les bits les plus significatifs de l'opérande sont perdus, et des Ø remplacent les bits les moins significatifs décalés vers des positions supérieures. Un décalage à gauche correspond à une multiplication par 2^bits (sauf en cas de débordement de capacité), où "^" indique l'opération d'élévation à la puissance et bits est le nombre de bits décalés.

Cet opérateur correspond à l'instruction "ASL" du 65C816, excepté qu'il faut la répéter autant de fois qu'il y a de bits à décaler.

• ">>" effectue un "décalage à droite" : l'opérande de gauche est décalé à droite du nombre de bits spécifié par l'opérande de droite; autrement dit, les bits les moins significatifs de l'opérande de gauche sont perdus, et des Ø remplacent les bits les plus significatifs décalés vers des positions inférieures, si l'opérande de gauche est non signé; si il est signé, le signe est propagé dans les positions décalées. Un décalage à droite est équivalent à une division par 2^bits, où "^" indique l'opération d'élévation à la puissance et bits est le nombre de bits décalés, sauf dans le cas où le nombre est signé et que le résultat deviendrait Ø; dans ce cas, il sera égal à -1. Cet opérateur correspond à l'instruction "LSR" du 65C816 lorsque l'opérande à décaler est non signé, et n'a pas d'équivalent direct s'il est signé, mais ce cas est assez rare.

Les instructions "ROL" et "ROR" du 65C816 n'ont pas d'équivalent en C, ni en ORCA/Pascal; TML Pascal dispose des fonctions "BRotL" et "BRotR" mais elles ne sont pas strictement identiques aux instructions machines censées correspondre. Ce n'est pas bien génant, car on n'en pas spécialement besoin; de plus, elles font appel à la "carry" qui n'est pas accessible dans un langage évolué. Enfin, elles peuvent être réalisées très facilement à l'aide des opérateurs précédents.

Utilisation de ces opérateurs

Maintenant que nous avons les bases, nous allons pouvoir utiliser ces opérateurs pour réaliser des opérations un peu plus sophistiquées.

Dans la panoplie des opérateurs listée ci-dessus, vous avez sans doute remarqué qu'il manque la possibilité de positionner un ou plusieurs bits à 1 ou à Ø dans un opérande; ces opérations sont pourtant des opérations de base, d'autant plus qu'elles sont disponibles en assembleur avec les instructions "TSB" et "TRB". C'est donc la première chose que nous allons réaliser, d'autant que c'est très facile !

Positionnement d'un bit à 1

Commençons par le positionnement d'un bit quelconque à 1 dans un entier (sur 8, 16 ou 32 bits, cela n'a pas d'importance).

Si vous regardez la définition du "ou" donnée plus haut, vous vous apercevrez qu'un bit du résultat est à 1 dès lors que le bit correspondant de l'un des opérandes est à 1. Donc pour mettre un bit à 1 dans un entier, il suffit de faire un "ou" entre ce bit et un 1 : si le bit en question est déjà à 1, il ne changera pas; en revanche, s'il vaut Ø, le résultat du "ou" avec 1 le mettra bien à 1. De même, si je fais un "ou" entre un bit de l'entier considéré et Ø, le bit en question ne changera pas; en quelque sorte, un "ou" avec Ø est un "NOP".

Maintenant, le problème est de mettre un 1 en face du bit à positionner.
En fait ce que l'on veut créer c'est un "masque" dont tous les bits sont à Ø (pour ne pas changer les bits correspondant de l'entier), sauf celui que l'on veut positionner, et qui sera donc à 1. La question est donc de transformer quelque chose comme "bit x" dans le masque correspondant.

Prenons maintenant l'opérateur "<<". Pour créer notre masque, il nous suffit de décaler à gauche un 1 du nombre de bits désiré, de façon à ce qu'il se retrouve à la position voulue dans le masque. En effet, la constante 1 correspond à un masque ayant le bit Ø positionné; il nous faut donc 'pousser' ce bit de façon à ce qu'il se retrouve à la position recherchée.

Avec ces 2 opérateurs, on peut donc réaliser l'opération de positionnement d'un bit à 1 très simplement :

```
nombre = nombre | ( 1 << bit);
```

"Nombre" est l'entier considéré, et "bit" le numéro du bit devant être mis à 1. Cette expression peut être 'simplifiée' (cela dépend du point de vue) ainsi :

```
nombre |= 1 << bit;
```

L'intérêt de cette deuxième forme est que le nombre ne sera évalué qu'une seule fois.

Pour ne pas avoir à se rappeler cette formule, le plus simple est d'écrire une fonction qui accepte 2 arguments : le "nombre" et le "bit" à positionner.
Mais en C, on peut faire mieux; étant donné la simplicité du calcul, il est préférable d'utiliser une macro, par exemple :

```
#define set_bit(n,b)    ( ( n ) |= ( 1   << ( b ) ) )
```

Vous avez sans aucun doute remarqué la profusion de parenthèses : lorsqu'on définit une macro, il est souhaitable d'entourer chacun des arguments de la macro par des parenthèses. En effet, ceux-ci peuvent être des expressions et la macro procède par substitution; on veut donc éviter que l'expression résultant de l'expansion ne soit pas évaluée telle qu'elle a été écrite dans le programme, notamment à cause des règles de précédence. On prend donc un maximum de précautions, et on entoure chaque argument de parenthèses; ce n'est pas bien génant car cela reste au niveau de la définition de la macro.

Positionnement d'un bit à Ø

Il nous faut maintenant réaliser le pendant de l'opération précédente, à savoir positionner un bit quelconque d'un entier à Ø.

Cette opération est un petit peu plus compliquée que la précédente. En effet, le "ou" est inadapté à cette situation, puisqu'on ne peut pas faire un "ou" avec quelque chose dans le but d'avoir un Ø en résultat : un "ou" avec 1 va mettre un 1, et un "ou" avec Ø est un "NOP".

En étudiant les opérateurs dont on dispose, le seul qui puisse mettre un bit à Ø de façon certaine est le "et", puisqu'un "et" avec Ø donne Ø tandis qu'un "et" avec un 1 est un "NOP".

Le problème est donc désormais de créer un masque avec le bit à positionner à Ø et tous les autres à 1, de façon à ne pas changer les autres bits du nombre. A priori, l'opérateur de décalage à gauche ne convient pas puisqu' il ne permet pas de faire glisser un Ø dans un ensemble de 1. Revenons sur notre masque : si on inverse tous ses bits, on retrouve le masque utilisé pour positionner un bit à 1. A l'inverse, je sais créer un masque avec un bit à 1; si j'inverse tout (ce qui est facile grâce à l'opérateur "~"),
j'aurai bien un masque avec le bit à positionner à Ø et tous les autres à 1.

Par conséquent, la mise à Ø d'un bit quelconque est :

```
nombre &= ~ ( 1 << bit )
```

On peut donc définir une macro qui permet donc de s'affranchir de l'écriture de cette formule :

```
#define clear_bit(n,b)      ( ( n ) &= ( ~ ( ( b ) << 1 ) ) )
```

Cette macro ressemble comme 2 gouttes d'eau à la précédente. Il serait peut-être donc préférable d'avoir une macro générique qui applique une opération entre un nombre et un bit, et que cette macro soit utilisée par les autres. Ceci est facilité par le fait qu'un argument d'une macro peut être n'importe quoi, et pas nécessairement une variable, puisque l'expansion d'une macro consiste à substituer ses arguments dans la définition de la macro.

Je peux donc créer une macro que j'appellerai "bit_op", définie ainsi :

```
#define bit_op(n,b,op)      ( ( n ) op ( 1 << ( b ) ) )
```

"n" est le nombre considéré, "b" est le numéro du bit à manipuler, et "op" est un ou plusieurs opérateurs C tels que l'expression soit valide. Avec cette définition, je peux réécrire les macros précédentes de la façon suivante :

```
#define set_bit(n,b)          bit_op ( n, b, |= )
#define clear_bit(n,b)     bit_op ( n, b, &= ~ )
```

Le fait d'imbriquer les macros ainsi améliore légèrement la lisibilité et permet éventuellement de remplacer "bit_op" par une autre version (comme nous le verrons plus loin) sans nécessiter la redéfinition de toutes les macros. En revanche, cela n'a aucun impact sur les performances, puisque les macros seront substituées par le compilateur, et le programme exécutera l'opération finale, comme si elle avait été codée directement.

Autres opérations sur 1 bit

On peut compléter les 2 macros précédentes par :

```
#define toggle_bit(n,b)     bit_op ( n, b, ^= )
#define test_bit(n,b)         bit_op ( n, b, & )
```

La première utilise la propriété de l'opération "ou exclusif" ou "xor" pour offrir une fonction d'inversion de bit.

La seconde permet de tester si un bit est à 1 ou pas : il suffit d'effectuer un "et" entre le nombre considéré et un masque à 1 pour obtenir le résultat.
Notez que contrairement aux autres macros, celle-ci ne modifie pas son argument; elle se contente de le tester.

Généralisation à des masques de plusieurs bits

Toutes ces macros présentent un inconvénient : elles n'agissent que sur un seul bit à la fois. Il serait donc souhaitable de disposer d'un autre jeu de macros travaillant directement sur un masque de bits. Avec ce que nous avons vu, rien de plus facile :

```
#define mask_op(n,m,op)          ( ( n ) op ( m ) )

#define set_mask(n,m)             mask_op ( n, m, |= )
#define clear_mask(n,m)        mask_op ( n, m, &= ~ )
#define toggle_mask(n,m)      mask_op ( n, m, ^= )
#define test_mask(n,m)          mask_op ( n, m, & )
```

Ces macros sont strictement identiques à leurs homologues agissant sur un seul bit, excepté qu'il n'est plus nécessaire de calculer le masque, celui-ci étant fourni en paramètre.

Toutes ces macros, ainsi que plusieurs autres sont définies dans le fichier "Bit.h" que vous trouverez sur votre disquette "GS Infos". Les macros non décrites ici sont suffisamment triviales pour ne pas nécessiter d'explications supplémentaires; de toute manière, elles sont abondamment commentées dans le fichier "Bit.h".

Multiplications et divisions

Lors de la description des opérateurs "<<" et ">>", j'ai indiqué qu'ils étaient équivalents à une multiplication et à une division par une puissance de 2. Par rapport à la multiplication et à la division, ils présentent cependant un avantage indéniable : ils sont beaucoup (mais alors vraiment beaucoup !) plus rapides, d'autant plus que ce sont des instructions du 65C816, alors que ce n'est pas le cas de la division ni même de la multiplication.

Je ne peux que vous recommander d'utiliser des décalages à chaque fois que cela est possible, et ce malgré le fait qu'on ne peut les utiliser qu'à la place de multiplications/divisions entières par des puissances de 2. Cela semble être une très forte restriction, mais en fait il y a énormément de programmes qui peuvent en bénéficier, sans doute parce que l'informatique est essentiellement binaire.

Pour vous convaincre de la différence de rapidité entre les 2, j'ai écrit un petit programme de mesure de performances. Ce programme effectue 50000 multiplications et 50000 décalages à gauche; pour que cela soit significatif, le décalage est fait sur 14 bits, et pour rester cohérent, on multiplie par 16384. Enfin, histoire d'être dans le cas le plus défavorable, les calculs sont effectués sur des entiers longs. Pour que les tests soient complets, la même comparaison est effectuée entre 50000 divisions et 50000 décalages à droite. Enfin, j'ai effectué le même test avec le calcul du reste en utilisant l'équivalence suivante :

mod(x,p2) = x & ~(p2 - 1)

c'est à dire que le reste de la division par une puissance de 2 correspond à la remise à Ø des bits de poids faible.

Voici les résultats de l'exécution de ce programme :

Multiplication ... duree = 20 secondes
Decalage ... duree = 4 secondes
Division ... duree = 22 secondes
Decalage ... duree = 5 secondes
Reste ... duree = 22 secondes
And ... duree = 1 secondes

On constate immédiatemment la différence de performances entre le décalage et la multiplication/division : environ 4 à 5 fois. J'ai bien entendu effectué plusieurs exécutions, et à chaque fois le résultat est similaire.
La différence est encore bien plus grande pour le reste, puisque cela est quasiment instantanné si on utilise un masquage des bits de poids faible.

Une deuxième partie du test est plus proche de ce que l'on a besoin normalement, c'est à dire de multiplier ou diviser un petit nombre par une petite puissance de 2 (en général 2, 4 ou 8).

Voici les résultats de ce test :

Multiplication ... duree = 6 secondes
Decalage ... duree = 2 secondes
Division ... duree = 6 secondes
Decalage ... duree = 2 secondes

Là encore, la différence est spectaculaire ! En fait, elle le serait sans doute encore plus si j'avais utilisé des entiers courts plutôt que des longs. Je vous laisse faire la comparaison vous-mêmes.

Pour vous éviter de vous creuser la tête pour chercher l'équivalence entre une multiplication (ou une division) et le décalage correspondant, j'ai mis au point un jeu de macros (qui est aussi dans "Bit.h") :

```
#define mul(n,m)    ( n << find_first_set ( m ) )
#define div(n,m)    ( n >> find_first_set ( m ) )
#define mod(n,m)    ( n & ~ ( m - 1 ) )
```

"n" est le nombre à multiplier, et "m" est le multiplicateur qui doit donc être une puissance de 2.

Cette macro fait appel à une fonction "find_first_set" qui recherche le premier bit à 1 de son argument, et dans ce cas précis, trouve donc l'exposant utilisé par cette puissance de 2. En fait "find_first_set" est une autre macro définie ainsi :

```
#define find_first_set(n)          find_first ( n, 1 )
```

La fonction réelle est "find_first"; son deuxième argument donne la valeur du bit à rechercher, et vaut 1 ou Ø.

Cette macro est accompagnée des macros suivantes :

```
#define find_first_clear(n)        find_first ( n, 0 )
#define find_first_set_l(n)        find_first_long ( n, 1 )
#define find_first_clear_l(n)      find_first_long ( n, 0 )
```

Les macros en "_l" et la fonction "find_first_long" sont identiques aux précédentes, exceptées que "n" doit être un entier long et non un entier court.

Les performances obtenues avec ces macros sont :

mul(n,m) macro ... duree = 8 secondes
div(n,m) macro ... duree = 9 secondes
mod(n,m) macro ... duree = 1 secondes

On constate que les macros sont 2 fois moins performantes que l'utilisation directe du décalage, ce qui est tout à fait normal, étant donné que la fonction "find_first" effectue aussi un décalage du même nombre de bits pour trouver le premier à 1.

Les fonctions "find_first" et "find_first_l" ont été écrites en assembleur, afin d'obtenir la performance maximale (elles sont dans le fichier source "Find.Bit.asm"). Elles sont incluses dans la librairie "Bit.lib" qui se trouve sur ce GS Infos, et peuvent être utilisées dans d'autres contextes que celui décrit ici. En fait, la recherche du premier bit à 1 ou Ø est une fonction utile dans un certain nombre de situations. Par exemple, si vous avez un tableau de moins de 32 octets, et que vous savez qu'il n'est pas nécessairement plein, vous pouvez lui adjoindre un entier court ou long selon la taille de votre tableau, et

positionner le bit correspondant à chaque élément utilisé. Un appel à "find_first_set_l" vous donne le premier élément utilisé et "find_first_clear_l" le premier disponible.

Pour mesurer l'efficacité réelle du compilateur C, et de ma version assembleur, j'ai aussi écrit une version en C des fonctions "find_first" et "find_first_long". Je les ai appelées "find_first_cc" et "find_first_long_cc"; le source est dans le fichier "Bit.cc" et l'objet dans la librairie "Bit.lib".

J'ai seulement effectué le premier test de multiplication avec cette fonction, et voici le résultat :

find_first_cc ... duree = 10 secondes

Bien que l'optimisation effectuée par ORCA/C soit loin d'être optimale, par rapport à la version que j'ai écrite en assembleur, la différence de performance n'est pas trop significative (25% plus mauvais). On peut donc continuer à utiliser le C, sans trop se poser de questions, sauf si l'on fait vraiment des choses compliquées, et qu'alors l'assembleur permette d'améliorer de façon significative les performances.

Il me faut enfin vous indiquer que ces tests ont été effectués sur un GS rom 1, avec une ZIP 8 MHz et 16 Ko de cache. Les résultats peuvent être différents sur un GS de base; je parle bien entendu des rapports et pas des durées qui sont forcément plus longues.

Chaînes de bits

Tout ce que nous avons vu jusqu'à présent est bien joli, mais malheureusement ne s'applique qu'à un type scalaire prédéfini, c'est à dire à un entier sur 8, 16 ou 32 bits. Il serait bien agréable de pouvoir effectuer les mêmes manipulations sur des entités plus grandes, par exemple 64 ou 128 bits.

C'est bien entendu tout à fait possible. Pour cela, on va définir un nouveau type (ce n'est pas une structure de données à proprement parler) : la "chaîne de bits" ou "tableau de bits" ou en anglais, "bitstring" ou "variable length bit field" (on emploie aussi parfois, mais de façon impropre, le terme de "bitmap").

L'idée générale est de travailler non plus au niveau de l'octet, mais au niveau de chaque bit. On va donc considérer une chaîne de bits de longueur finie. Bien entendu, pour la machine, l'entité de base reste l'octet ou le mot, et il va nous falloir effectuer une correspondance entre notre unité de base et celle de la machine. De plus, il n'est pas question non plus de gaspiller de la place; on va donc compacter notre chaîne de bits à l'intérieur d'octets ou de mots. Comme le GS est une machine 16 bits, il paraît tout à fait naturel d'exploiter des mots courts comme unité de compactage de notre chaîne de bits.

Par conséquent, notre chaîne de bits va être matérialisée par un tableau de mots de 16 bits, dont la dimension sera égale au nombre de bits maximum de la chaîne divisé par 16, arrondi au nombre supérieur. Ainsi si je veux travailler sur une chaîne de 80 bits, je déclarerai un tableau de 5 mots de 16 bits.

Comme on est en train de créer un nouveau type, et histoire de se simplifier la vie, nous allons construire une librairie permettant de gérer les chaînes de bits.

On va donc d'abord définir un type "bitstring" permettant de masquer les détails de l'implémentation à l'utilisateur de la librairie :

```
typedef unsigned short *BitString;
```

A partir de là, on peut créer une fonction "create_bitstring" qui réalise la création d'une chaîne de bits de la taille demandée en allouant le tableau de mots de 16 bits sous-jacent :

```
BitString create_bitstring ( unsigned short size )
{
    short num_words;
     BitString bit_str;

    num_words = ( size >> 4 ) + ( ( size & 0x0f ) ? 1 : 0 );
    if  ( ( bit_str = calloc ( num_words + 1, sizeof ( short ) ) ) != NULL )
        *bit_str = size;
     return ( bit_str );

}    /* create_bitstring () */
```

Dans cette implémentation, on alloue un mot de plus que la taille nécessaire, de façon à y stocker la taille de la chaîne de bits. On pourrait se servir de cette information pour valider le numéro de bit dans les fonctions en acceptant un en paramètre, mais je ne l'ai pas fait dans cette implémentation (la raison en est que ces fonctions sont implémentées par des macros, et que ce test les aurait compliquées inutilement). On utilise la fonction de la librairie standard "calloc" plutôt que "malloc", car elle présente l'avantage d'initialiser la mémoire allouée à Ø, ce qui nous est fort utile ici; on a en effet envie que tous les bits soient à Ø pour commencer.

Comme il faut toujours laisser les choses dans l'état dans lequel on les a trouvées, il nous faut aussi une fonction de destruction d'une chaîne de bits, qui est réduite à sa plus simple expression :

```
void delete_bitstring ( BitString bit_str )
{
    free ( bit_str );
}    /* delete_bitstring () */
```

Nous pouvons maintenant utiliser notre chaîne de bits pour réaliser les opérations classiques : positionnement d'un bit à 1 ou à Ø, test d'un bit à 1 et inversion d'un bit. Ca ne vous rappelle rien ? Ce sont justement les macros que nous avons définies au début de cet article. L'idéal serait de créer une nouvelle version de "bit_op" fonctionnant avec une bit string, et les autres macros fonctionneraient sans aucun changement.

C'est bien évidemment ce que nous allons faire; voici cette macro :

```
#define bit_op(var,bit,op)   ( var[((bit) >> 4) + 1] op ( 1 << ((bit) & 0x0f) ) )
```

Whaou ! Voilà qui nécessite quelques explications :

Notre chaîne de bits étant en fait constitué d'un tableau de mots de 16 bits, la première chose à faire est de déterminer le mot qui sera affecté; c'est ce que réalise "var[((bit) >> 4) + 1]" : la division du numéro de bit par16 (le nombre de bits dans un mot) donne le numéro de mot, tandis que l'ajout de 1 tient compte du mot supplémentaire au début de la chaîne de bits, et contenant la taille de cette chaîne.

La deuxième partie de la formule calcule le numéro du bit dans le mot déterminé précédemment; ceci est effectué par ((bit) & 0x0f) : puisque chaque mot comprend 16 bits, le numéro du bit est donc compris entre Ø et 15; il nous suffit donc de prendre le reste de la division par 16 pour obtenir ce numéro de bit.

Voyons comme cela marche sur un exemple : prenons par exemple le bit 69 d'une chaîne de 80 bits : la division par 16 donne le 4, mais comme le premier indice est Ø, ce la nous donne le cinquième mot du tableau; on lui ajoute ensuite 1 pour prendre en compte le mot Ø qui contient la taille de la chaîne de bits. Le reste de la division par 16 donne 5, ce qui au final donne pour ce bit 69, le cinquième bit du sixième mot.

Les macros de C trouvent ici toute leur justification : on peut ainsi pousser les possibilités du langage à ses limites, tout en en masquant la complexité dans une macro; le code principal reste lui parfaitement lisible.

Cette macro, ainsi que les prototypes des fonctions sont définies dans le fichier "BitString.h". Afin de ne pas avoir à redéfinir les macros effectuant les opérations de base (set_bit,clear_bit,toggle_bit et test_bit), "Bit.h" est inclus après la définition de "bit_op". Et pour qu'il ne redéfinisse pas sa propre version de "bit_op", "BitString.h" définit le symbole _BIT_STRING avant l'inclusion. En fait, toutes les parties de "Bit.h" non pertinentes pour les chaînes de bits sont exclues grâce à la compilation conditionnelle.

Pour être tout à fait complet, "BitString.cc" définit une nouvelle fonction "find_first" utilisée par les macros "find_first_set" et "find_first_clear" à la place de celle implémentée dans "Find.Bit.asm". Voici cette fonction :

```
short find_first ( BitString bit_str, unsigned short flag )
{
    unsigned short bit;

    for ( bit = 0; bit < *bit_str; bit++ )
        if ( ( test_bit ( bit_str, bit) != 0 ) == flag )
            return ( bit );
    return ( 0 );
}   /* find_first () */
```

Cette fonction recherche le premier bit de la chaîne ayant une valeur égale à celle du flag. Pour ce faire, on isole chacun des bits tour à tour en partant du bit Ø à l'aide de la macro "test_bit", et on force le résultat à être Ø ou 1 grâce au test "!= 0". Il ne nous reste plus qu'à comparer cette valeur avec celle du bit recherché, et spécifiée par le flag.

La librairie contient enfin la fonction "get_bitstring_size()" qui retourne la taille stockée dans le premier mot de 16 bits.

Sur la disquette
==========

Vous trouverez sur votre disquette GS Infos les fichiers suivants :

- "Bit.h" définition des macros permettant de manipuler des bits
- "Bit.cc" implémentation C de "find_first" et "find_first_long"
- "Find.Bit.asm" implémentation assembleur des fonctions ci-dessus
- "Bit.lib" librairie contenant les versions C et assembleur
- "Test.Speed.cc" démonstration des différences de performance entre :

- "Test.Speed" un décalage et une multiplication/division
- "BitString.h" définition macros/prototypes librairie chaîne de bits

- "BitString.cc" implémentation C librairie de gestion de chaînes de bits
- "BitString.lib" libraririe avec laquelle linker
- "Test.BS.cc" source C du programme de démonstration de cette librairie
- "Test.BitString" le programme de démonstration des chaînes de bits

Je vous propose de réflechir au problème suivant : comment permuter 2 nombres "a" et "b" (c'est à dire qu'après l'opération "a" doit contenir l'ancienne valeur de "b" et vice-versa), sans utiliser de variable intermédiaire.

Allez, je vous aide un peu ... vous devez utiliser un des opérateurs de manipulation de bits que nous avons vu dans cet article; vous vous en seriez doutés, non ? Pour la solution complète, il vous faudra patienter jusqu'au prochain GS Infos ...

Nouvelles du front

Trois nouveaux produits ont été annoncés récemment par Byte Works pour le programmeur sur GS (qui a dit qu'il n'y avait jamais rien de nouveau sur GS ?!). Ces 3 produits sont disponibles chez Resource Central, et sans doute aussi chez Byte Works, mais comme je n'ai reçu aucun courrier de leur part pour me les proposer ... En voici un bref résumé :

• ORCA/Debugger : Ce permet permet d'aider à la mise au point des programmes C et Pascal. Il remplace (plus ou moins) le debugger inclus dans Prizm. La différence principale est qu'il fonctionne en mode texte au lieu du mode bureau, ce qui permet de l'utiliser avec tous les types de programmes réalisables sur GS. Il fonctionne sur un principe similaire à GSBug avec lequel il est compatible, c'est à dire qu'il s'agit d'une init et qu'il est donc résident en permanence. Sa différence principale avec GSBug est que ORCA/Debugger permet de faire la mise au point au niveau du code source, tandis que GSBug sert à la mise au point en langage machine. Mais on peut avoir les 2 en même temps ! $39.95 chez Resource Central au lieu de $50, prix catalogue. Je ne peux pas vous en dire plus, car je ne l'ai pas encore reçu ...

• Toolbox Programming in Pascal : il s'agit d'un cours de programmation de la boîte à outils du GS, utilisant le langage Pascal (ah bon ?). Il s'inscrit dans la lignée des cours d'initiation au Pascal et au C que Byte Works a déjà produit, et dont j'ai lu le plus grand bien. Comme ses prédécesseurs, ce cours est très axé sur la pratique, et les 400 pages du livre sont accompagnées de 4 disquettes d'exemples (soit plus de 3 méga-octets !).
Son seul inconvénient est qu'il est en anglais 8-} $75 chez Resource Central.

• Programmers System 6 Reference : Byte Works a écrit le tome IV du manuel de référence de la toolbox du GS. Ce livre contient donc la description de toutes les nouveautés apportées par le système 6.0 aussi bien dans la boîte à outils que par GS/OS. Il s'adresse donc aux programmeurs. On peut penser que le délai de livraison du système 6.0 par Byte Works a été occasionné par la volonté de réaliser ce manuel afin de l'inclure dans le paquet, mais comme je n'ai toujours rien reçu, je ne peux rien affirmer. En tout cas, si vous ne l'avez pas commandé, je ne peux que vous conseiller d'acheter ce livre. $45 chez Resource Central.

Programmation N°5

le type "Ensemble"

Dans cet article, nous allons mettre en œuvre les opérations sur les bits que nous avons vues dans le précédent numéro de GS Infos pour développer une nouvelle structure de données correspondant au type abstrait "ensemble".

A priori, vous devez penser que cet article ne s'adresse qu'aux programmeurs en C, puisque le langage Pascal dispose d'un type "set" prédéfini. Il est vrai que l'implémentation que je vous propose pour accompagner cet article, est écrite en C.

Toutefois, je pense que cet article peut aussi être intéressant à lire pour les programmeurs Pascal, car il montre comment ça marche, et donc peut aider à mieux comprendre quand et comment utiliser ce type, et ce d'autant plus que son existence en tant que type prédéfini masque son niveau d'abstraction bien plus élevé que celui des autres types, et par conséquent le coût qu'implique son utilisation non faite en connaissance de cause.

Mais avant d'aborder le sujet du jour, je voudrais d'abord apporter une légère correction au précédent article : j'ai en effet écrit que les opérateurs de manipulation de bits en ORCA/Pascal étaient identiques à ceux d'ORCA/C; c'est vrai à l'exception de l'opérateur xor (ou exclusif) qui est "^" en C et "!" en ORCA/Pascal, puisque "^" est déjà utilisé par Pascal comme opérateur d'indirection.

A propos du où exclusif, il est temps maintenant de donner la solution à l'exercice que je vous avais proposé la dernière fois. Je vous en rappelle l'intitulé : comment permuter 2 nombres "a" et "b" (c'est à dire qu'après l'opération "a" doit contenir l'ancienne valeur de "b" et vice-versa), sans utiliser de variable intermédiaire.

Vous devez maintenant vous douter que la solution va faire appel à l'opération ou exclusif (quel bel enchaînement ;-). Allez, je vais cesser de vous faire languir;
voici tout de suite une macro répondant au problème :

```
#define swap(a,b)     a ^= b, b ^= a, a ^= b
```

Hum, je pense que cela nécessite quelques explications complémentaires.
L'opérateur ou exclusif a une propriété intéressante que je n'ai pas exposé dans le précédent article; j'avais juste dit qu'il avait une certaine propriété à propos de la macro toggle_bit(), mais j'ai involontairement omis de dire laquelle.

Cette propriété est telle que si j'effectue 2 fois l'opération xor avec le même opérande, je retrouve ma valeur de départ (autrement dit, le où exclusif d'un nombre avec lui-même est égal à Ø), ainsi a ^ b ^ b = a.

Donc, pour vérifier que la macro précédente est bien la solution au problème, il nous suffit de voir en détail son action :

- Après "a ^= b", "a" vaut "a ^ b";
- Dans "b ^= a", remplaçons "a" par sa valeur : on obtient donc "b = b ^ a ^ b", soit, d'après la propriété précédente, "b = a";
- Effectuons la même substitution dans "a ^= b" : nous avons alors "a = a ^ b ^ a", ce qui, après simplification, donne "a = b";

CQFD. A noter que l'on peut obtenir le même résultat avec la soustraction à condition que les nombres

soient non signés; cependant le ou exclusif est en général plus rapide que la soustraction. Je vous laisse faire la démonstration vous-mêmes; le calcul est exactement le même à l'opérateur près.

Notez enfin, que contrairement à ce que je vous avais dit dans le précédent article, je n'ai pas pris de précautions particulières pour entourer les paramètres de la macro par des parenthèses. La raison en est que ces paramètres doivent être obligatoirement des variables (ou plus précisément des valeurs-g) puisqu'on va leur affecter une nouvelle valeur. Il n'est donc pas utile d'ajouter des parenthèses dans ce cas. J'ai aussi utilisé l'opérateur séquentiel "," pour séparer les 3 expressions; j'aurai pu tout aussi bien séparer les expressions par un ";".

Concepts d'Ensemble

Dans le numéro 3 de cette série, je vous avais présenté la structure de données liste comme étant un ensemble ordonné d'éléments. Le terme ensemble dans ce contexte n'était pas forcément bien choisi car il sous-entend un certain nombre de caractéristiques que nous allons voir dans cet article.

Un "ensemble" (que l'on appelle "set" en anglais), en tant que tel, est un type de données abstrait permettant de représenter une collection de données quelconques. Contrairement à la liste, il n'y a pas de relation d'ordre des objets dans un ensemble. De plus, chaque objet d'un ensemble est unique; on ne peut donc pas avoir, par exemple, plusieurs occurences d'un même nombre dans un ensemble d'entiers.

Vous avez sans doute reconnu ici la définition d'un ensemble au sens mathématique du terme. Il s'agit effectivement de représenter sous forme informatique, les concepts sous-jacents aux ensembles.

Du fait qu'il s'agit d'un concept bien défini, le nombre d'opérations que l'on peut effectuer avec les ensembles sont limitées, contrairement aux listes qui offrent une très grande variété d'opérations en fonction de la manière dont on veut les utiliser.

Un objet d'un ensemble est appelé un "membre" ou parfois un "élément"; la propriété fondamentale d'un ensemble est la relation d'appartenance qui s'établit entre les éléments et les ensembles : un objet appartient (c'est alors un membre) ou n'appartient pas à l'ensemble.

A partir de cette propriété, on peut définir les opérations de base que l'on peut appliquer sur un ensemble et ses membres :

- Création d'un ensemble,
- Destruction d'un ensemble,
- Ajout d'un membre à l'ensemble,
- Suppression d'un membre de l'ensemble,
- Test si un objet est membre de l'ensemble ou pas.

On peut ensuite compléter ces opérations par des opérations plus sophistiquées telles que :

- Union de 2 ensembles,
- Intersection de 2 ensembles,
- Différence symétrique entre 2 ensembles,
- Complément d'un ensemble,
- Test si 2 ensembles sont identiques, disjoints ou ont une intersection non vide,
- Test si un ensemble est un sous-ensemble d'un autre ensemble,
- Test si l'ensemble est vide ou pas,

• Comptage du nombre de membres d'un ensemble.

Dans la suite de cet article, nous allons réaliser l'implémentation de ces opérations, ainsi que de quelques autres qui sont secondaires, mais néanmoins fort utiles lorsqu'on doit manipuler des ensembles.

Utilisation des ensembles

Comme les autres structures de données que nous avons étudié jusqu'à présent, les ensembles sont utilisés très souvent, notamment par les programmeurs Pascal, qui en disposent de base dans le langage, même si c'est parfois à mauvais escient.

L'utilisation typique d'un ensemble correspond au cas où l'on cherche à savoir si un objet est membre ou pas d'un ensemble, particulièrement lorsque celui-ci est disparate.

Cependant, pour pouvoir vérifier l'appartenance d'un objet à un ensemble, il faut d'abord ajouter les membres à l'ensemble en question, ce qui peut représenter une opération coûteuse. De la même manière, le test d'appartenance peut coûter plus ou moins cher selon l'implémentation de l'ensemble : heureusement en Pascal, comme dans l'implémentation que je vous propose ci-après, le temps nécessité pour le test d'appartenance est constant, c'est à dire qu'il ne dépend pas du nombre de membres de l'ensemble.

Donc, lorsque l'ensemble est continu, il est préférable d'utiliser une comparaison de l'élément avec les bornes plutôt que de tester l'appartenance de ce membre à l'ensemble, car cette dernière opération est bien plus coûteuse.

Pour vous donner un exemple concret de cette différence : supposez que vous voulez savoir si une variable "c", que vous venez par exemple de lire au clavier, est une lettre ou pas; en effectuant un test d'appartenance, vous pouvez par exemple écrire, en Pascal :

```
IF c IN ['A'..'Z','a'..'z'] THEN ...
```

C'est vrai que cette solution a l'avantage d'être élégante. Malheureusement, cette simplicité apparente masque le fait qu'il s'agit d'un type abstrait et que par conséquent le 65C816 n'a aucune idée de ce que peut être un ensemble. Le compilateur doit donc générer pas mal de code d'une part pour construire cet ensemble, et d'autre part appeler une routine de la librairie Pascal pour vérifier que le caractère appartient ou non à l'ensemble. Si ce test est effectué dans une boucle, comme c'est souvent le cas, cela peut coûter assez cher.

Dans ce cas, il est souhaitable, à mon avis, de tester si le caractère est dans l'intervalle voulu, ce qui peut s'écrire, toujours en Pascal :

```
IF ( ( c >= 'A' ) AND ( c <= 'Z' ) ) OR ( ( c > = 'a' ) AND ( c <= 'z' ) ) THEN ...
```

J'admets que cette écriture est un peu plus lourde que la précédente; toutefois, dans ce cas précis, elle est nettement moins coûteuse en terme de taille de code, et par conséquent de temps d'exécution. On peut cependant supprimer la moitié des tests en utilisant les propriétés du code ASCII :

```
cc := c & $5F;                    (* conversion du caractère en majuscule *)
IF ( cc >= 'A' ) AND ( cc <= 'Z' ) THEN ...
```

La conversion précédente est incorrecte si le caractère n'est pas une lettre, mais comme elle ne peut pas non plus créer une lettre dans ce cas, elle est acceptable.

En revanche, lorsqu'on doit gérér des objets disparates et que l'on peut distinguer ni continuité, ni ordre, l'ensemble s'avère être la structure de données idéale pour représenter ces objets. Un exemple pourrait être la gestion de l'ensemble des états dans un automate; en simplifiant, un automate est un programme permettant de reconnaître si un mot fait partie d'un langage ou non; cette technique est notamment utilisée dans les compilateurs. J'essaierai de revenir sur ce sujet dans un prochain article, si cela vous intéresse.

Représentation d'un ensemble

On peut représenter un ensemble de plusieurs manières selon le type des objets que l'on veut utiliser. Si celui-ci est quelconque, le plus simple est d'utiliser une liste, telle que nous l'avons déjà vu, sans se préoccuper toutefois de l'ordre des membres, et en faisant attention à ne pas créer de doublons, c'est à dire que l'ajout d'un élément existant déjà devra être refusé.

Cependant, très souvent, les éléments que l'on veut gérer sont de type scalaire : ce sont des entiers ou des caractères. D'ailleurs, Pascal impose que les éléments d'un ensemble soient scalaires. On pourra notamment utiliser des types énumérés ou des caractères qui sont convertis en entiers par le compilateur C. Cela signifie que l'on peut utiliser une méthode plus économique pour représenter un ensemble.

Puisque je vous ai dit au tout début de cet article que nous allions utiliser les techniques que nous avons étudiées dans le précédent GS Infos, la méthode que nous allons employer consiste à associer un bit à chacun des éléments de l'ensemble, qui sont eux-mêmes des nombres entiers (les caractères entrant aussi dans cette catégorie). Le test d'appartenance d'un élément de l'ensemble consistera alors à vérifier si le bit correspondant est à 1 (c'est un membre) ou à Ø.

Je vous renvoie à cet article pour toutes les opérations de base sur les bits, et si cela n'est pas clair pour vous, je vous conseille de réétudier la partie concernant les chaînes de bits, car nous allons utiliser les mêmes techniques.

Voyons donc tout de suite une structure permettant de représenter un ensemble :

```
typedef unsigned short      SET_UNIT_TYPE;
#define DEF_UNITS           4

typedef struct _set {
    unsigned char nunits;          /* Nombre d'unites (words) du tableau de bits */
    unsigned char complement;      /* vrai si l'ensemble a ete complemente */
    unsigned short nbits;          /* Nombre de bits dans le set */
    SET_UNIT_TYPE *bits;           /* Pointeur sur tableau de bits */
    SET_UNIT_TYPE defbits[DEF_UNITS];    /* Tableau de bits par defaut */
} *set;
```

Pour que les manipulations sur les bits soient les plus efficaces possibles, on a tout intérêt à les compacter dans le mot le plus grand possible pouvant être manipulé par une instruction du microprocesseur, soit dans le cas du GS, un mot de 16 bits, d'où la définition du type SET_UNIT_TYPE.

L'ensemble est alors représenté par un tableau initial de bits, dont j'ai fixé la taille à 4 mots, soit 64 bits, ce qui permet de représenter 64 éléments. Dans cette implémentation, un ensemble est dynamique et sa taille sera ajustée en fonction des besoins, la limite étant de 4096 éléments, ce qui est amplement suffisant (avec ORCA/Pascal, cette limite est de 2048 éléments). La librairie que je vous propose ne fait aucun contrôle à ce sujet, ce sera donc à vous de faire attention en l'utilisant. L'augmentation du tableau se fera par multiples de 4 mots, soit 64 éléments, jusqu'à concurrence du numéro de bit nécessaire pour représenter l'élément que l'on souhaite ajouter. Ceci signifie que le tableau n'est pas nécessairement plein

avant d'être augmenté. Par exemple, si je souhaite ajouter le nombre 200 à mon ensemble, il me faudra agrandir mon tableau pour qu'il contienne 16 mots, soit 256 éléments (même si tous les autres bits sont à Ø); le multiple de 4 mots précédent ne me permettait de stocker que 192 éléments.

Le champ "bits" de la structure pointe sur le tableau de bits, qui est soit le tableau initial préalloué, soit un tableau alloué dynamiquement lorsqu'il y a plus de 64 éléments. Ceci constitue une bonne illustration de l'équivalence entre tableaux et pointeurs dont j'ai parlé dans l'initiation au C de ce numéro.

Les champs "nunits" et "nbits" donnent la taille de l'ensemble, respectivement en nombre de mots et en nombre de bits. On gère les 2 valeurs afin d'éviter d'avoir à les calculer à chaque fois que l'on en a besoin, ce qui permet de gagner du temps, alors que cela ne consomme qu'un ou deux octets supplémentaires selon celui que l'on considère superflu. Je reviendrai un peu plus loin sur le champ "complement".

Vous remarquerez que les différents champs sont bien alignés à un multiple de leur longueur, et que malgré qu'ils aient l'air d'être dans l'ordre opposé à mes recommandations, ils respectent bien la règle que j'ai exposé dans mon précédent article d'initiation au C (voir GS Infos 23 si vous ne vous en rappelez plus).

Création et destruction d'un ensemble

Maintenant que nous sommes équipés d'une structure représentant l'ensemble, on peut écrire une fonction l'initialisant :

```
set create_set ( void )
{
   set s;

    s = (set) malloc ( sizeof ( struct _set ) );
   if ( s == NULL )
       return ( NULL );
    s->nunits = DEF_UNITS;
    s->nbits = DEF_BITS;
    s->complement = 0;
    s->bits = s->defbits;
    memset ( s->defbits, 0, UNITS_TO_BYTES ( DEF_UNITS ) );
   return ( s );
}
```

Cette fonction est assez triviale : elle se contente d'allouer la structure (ce qui permet de gérer plusieurs ensembles simultanément) et de remplir les différents champs qui la composent avec les valeurs initiales. Vous trouverez les constantes et les macros non décrites ici dans le fichier "Set.h" accompagnant cet article; elles y sont abondamment commentées.

Après la création, il nous faut aussi une fonction détruisant un ensemble :

```
void delete_set ( set s )
{
    if ( s->bits != s->defbits )
        free ( s->bits );
   free ( s );
}   /* delete_set () */
```

Cette fonction est encore plus simple que la précédente puisqu'elle n'a qu'à libérer la mémoire occupée par la structure représentant l'ensemble, ainsi que le tableau de bits si celui-ci n'est plus le tableau initial préalloué.

Opérations de base sur les membres

Nous pouvons maintenant écrire les fonctions réalisant l'ajout, la suppression et le test d'appartenance d'un membre dans un ensemble :

```
#define  bit_op(s,b,op)          ( (s)->bits[NUM_UNIT(b)] op ( 1 << NUM_BIT(b) ) )

#define  add_member(s,b)         ( ( (b) >= (s)->nbits ) ? extend_set ( s, b ) \
                                                         : bit_op ( s, b, |= ) )
#define  remove_member(s,b)  ( ( (b) >= (s)->nbits ) ? 0 : bit_op ( s, b, &= ~ ) )
#define  is_member(s,b)      ( ( (b) >= (s)->nbits ) ? 0 : bit_op ( s, b, & ) )
#define  test_member(s,b)    ( ( is_member ( s, b ) == 0 ) == (s)->complement )
#define  add_range(s,b1,b2)        range_op ( s, b1, b2, ~0 )
#define  remove_range(s,b1,b2)     range_op ( s, b1, b2,  0 )
```

En fait, ces opérations sont implémentées sous forme de macros; attention donc aux problèmes d'effets de bord que j'ai déjà exposé. Vous avez d'ailleurs sans doute reconnu les mêmes macros que celles que nous avons développées dans le précédent article, avec tout de même quelques petites différences :

- Comme un ensemble peut être agrandi dynamiquement, la macro "add_member" doit se demander si l'élément à ajouter rentre dans l'ensemble existant ou si ce dernier doit être agrandi, auquel cas il appelle la fonction "extend_set", que voici :

```
#define  EXTEND(bit)      ( ( NUM_UNIT(bit) + DEF_UNITS ) & ~ ( DEF_UNITS - 1 ) )

static short extend ( set s, unsigned short size )
{
    SET_UNIT_TYPE *bits;
   if ( s == NULL || size <= s->nunits )
       return ( 1 );
    bits = (SET_UNIT_TYPE *) malloc ( UNITS_TO_BYTES ( size ) );
   if ( bits == NULL )
       return ( 0 );
    memcpy ( bits, s->bits, UNITS_TO_BYTES ( s->nunits ) );
    memset ( bits + s->nunits, 0, UNITS_TO_BYTES ( size - s->nunits ) );
    if ( s->bits != s->defbits )
       free ( s->bits );
   s->bits = bits;
   s->nunits = size;
    s->nbits = UNITS_TO_BITS ( size );
   return ( 1 );
}   /* extend () */
```

```
short extend_set ( set s, unsigned short bit )
{
    if ( ! extend ( s, EXTEND ( bit ) ) )
        return ( 0 );
    return ( bit_op ( s, bit, |= ) );
}   /* extend_set () */
```

Ca a l'air un peu compliqué, mais en fait cela ne l'est pas : la fonction "extend_set" utilise une fonction interne "extend" en lui donnant la nouvelle taille du tableau de bits calculée par la macro "EXTEND" selon les principes exposés plus haut (elle calcule le nouveau nombre de mots à l'aide d'une macros annexe non listée ici, en l'arrondissant au multiple de 4 supérieur), puis elle positionne le bit correspondant à l'élément ajouté.

La fonction "extend" est celle qui effectue le travail : elle alloue un nouveau tableau de bits, puis elle recopie l'ancien dans le nouveau et initialise les bits supplémentaires à Ø, et, enfin, libère la mémoire occupée par l'ancien tableau si il avait déjà été agrandi.

• Je vous propose 2 macros de test d'appartenance : la première "is_member" vérifie simplement si le bit correspondant à l'élément recherché est bien à 1. La seconde "test_member" appelle la première macro pour déterminer l'appartenance puis inverse ou non le résultat en fonction de l'attribut complément. Ce champ indique si l'ensemble est normal ou s'il a été complémenté, c'est à dire que le sens des bits est inversé : on est alors en logique négative; dans ce cas un bit à 1 indique que l'élément correspondant n'appartient pas à l'ensemble, tandis qu'un bit est à Ø dans le cas contraire. L'attribut complément est positionné par la macro :

```
#define complement_set(s)          ( (s)->complement ^= 1 )
```

En fait, il s'agit d'une bascule : chaque appel à cette macro inverse l'attribut complément, et établit donc la le sens donné aux bits à 1 : en logique positive, un bit à 1 correspond à un membre de l'ensemble, tandis qu'en logique négative, un bit à 1 signifie que l'élément correspondant n'appartient pas à l'ensemble.

Toutefois, cette manière de complémenter un ensemble pose quelques problèmes avec les opérations plus avancées sur les ensembles; c'est pourquoi, je les ai implémentées sans tenir compte de cet attribut. A la place, j'ai réalisé une autre manière de complémenter un ensemble en inversant physiquement tous les bits, en voici l'implémentation :

```
void invert_set ( set s )
{
    SET_UNIT_TYPE *b, *e;
    for ( b = s->bits, e = b + s->nunits; b < e; b++ )
        *b = ~ *b;
}   /* invert_set () */
```

Cette fonction a pour effet de supprimer tous les membres de l'ensemble et d'y ajouter tous les éléments qui n'étaient pas membres auparavant. Elle nécessite que l'ensemble ait été agrandi à son maximum avant d'effectuer cette opération, par exemple en ajoutant le plus grand élément puis en le supprimant. En effet, si on ne faisait pas cet agrandissement, les éléments ajoutés après l'inversion auraient un sens contraire à ceux ajoutés avant. De plus, comme le tableau de bits est toujours plus grand que le nombre de membres de l'ensemble, on aura, après une inversion, des membres en trop; il faudra donc ne pas en tenir compte. En revanche, cette méthode a l'avantage de simplifier considérablement l'implémentation des opérations globales sur les ensembles, telles que l'union ou l'intersection. Notez qu'en Pascal, ce problème n'existe pas puisque Pascal n'offre pas la possibilité de complémenter un ensemble.

L'utilisateur de la librairie pourra choisir entre les 2 méthodes selon ses besoins, c'est à dire en fonction

des opérations qu'il compte effectuer.

• Les macros "add_range" et "remove_range" sont à utiliser lorsqu'on veut ajouter ou supprimer une série d'éléments consécutifs. Ainsi, et par exemple, au lieu d'appeler 26 fois "add_member" pour ajouter chacune des lettres, vous pouvez appeler "add_range" en indiquant les bornes de l'intervalle, par exemple "add_range(s,'a','z')". Ces macros font appel à la fonction "range_op", dont voici l'implémentation :

```
short range_op ( set s, unsigned short first, unsigned short last,
                 SET_UNIT_TYPE val )
{
    unsigned short  uf, ul, bf, bl, m1, m2;
    SET_UNIT_TYPE    *b, *e;

    /*
      * On s'assure que les 2 elements sont bien dans l'ordre croissant avant de
      * faire quoi que ce soit, sinon on les inverse.
      * On agrandit ensuite le tableau de bits si necessaire.
      */
    if ( first > last ) {
        uf = first;
        first = last;
        last = uf;
    }
    if ( ! extend ( s, EXTEND ( last ) ) )
        return ( 0 );

    /*
      * On commence par remplir les elements du tableau de bits concernes dans
      * leur totalite, c'est a dire tous ceux strictement compris entre les
      * elements correspondant aux bits extremes de la serie.
      * Il est possible que cette boucle ne fasse rien si les 2 extremes
      * correspondent a des elements voisins.
      */
    uf = NUM_UNIT ( first );
    ul = NUM_UNIT ( last );
    for ( b = s->bits + uf + 1, e = s->bits + ul; b < e; b++ )
        *b = val;
    bf = NUM_BIT ( first );
    bl = NUM_BIT ( last );

    if ( uf == ul ) {

        /*
          * Cas ou tous les membres sont situes dans le meme element.
          * On cree un masque contenant les bits concernes soit a 1, soit a 0
          * selon que l'on doit ajouter ou supprimer les membres.
          * Principe :
          * - "~0" cree un masque avec tous les bits a 1;
          * - ">> BITS_IN_UNIT (1) - ( bl - bf + 1 )" met a 0 les bits de poids
          *   fort, le nb de bits etant la difference entre le nb de bits total
          *   et le nb de bits a positionner;
          * - "<< bf" replace les bits a 1 en position;
          * - si on doit effacer les membres, on inverse tous les bits (les bits
          *   non affectes deviennent 1 et ceux a positionner 0) et on effectue
```

```
            *    un "&" avec les bits exitants;
             * - si on doit ajouter les membres, on doit seulement effectuer un "|"
            *    avec les bits existants. CQFD.
            */
             m1 = ~0 >> ( UNITS_TO_BITS ( 1 ) - ( bl - bf + 1 ) ) << bf;
           if ( val == 0 )
              *e &= ~ m1;
         else
              *e |= m1;

      } else {

          /*
             * Cas ou les 2 membres extremes ne sont pas dans le meme element.
             * On doit donc intervenir sur chacun de ces elements separement.
              * Pour l'element correspondant au bit inferieur, on doit agir sur
             * les bits de celui specifie jusqu'au nombre de bits de l'element.
              * Pour l'element correspondant au bit superieur, on doit agir sur
             * les bits de 0 a celui specifie.
             * Le principe est alors :
             * - "~0" : creation d'un masque avec tous les bits a 1;
              * - "<< bf" met les bits inferieurs jusqu'a "bf" a 0;
              * - ">> ( UNITS_TO_BITS ( 1 ) - bl - 1 )" met les bits superieurs
            *    apres "bl" a 0;
              * - si on doit effacer les membres, on inverse tous les bits (les
             *    bits affectes deviennent donc a 1 et les autres a 0) et on
             *    effectue un "&" avec les bits existants pour chacun des 2 elements.
              * - si on doit ajouter les membres, on n'a plus qu'a faire un "|"
             *    avec les bits existants pour chacun des 2 elements;
             */
            b = s->bits + uf;
           m1 = ~0 << bf;
            m2 = ~0 >> ( UNITS_TO_BITS ( 1 ) - bl - 1 );
           if ( val == 0 ) {
              *b &= ~ m1;
              *e &= ~ m2;
           } else {
              *b |= m1;
              *e |= m2;
         }

      }
       return ( 1 );
   }   /* range_op () */
```

Plutôt que de redétailler son fonctionnement, j'ai préféré laisser les commentaires expliquant les différents cas, et qui décrivent tout ce qu'il y a savoir sur cette fonction. Si cela ne vous semble pas assez clair, essayez de l'exécuter à la main en lui donnant un intervalle donné, par exemple les lettres de A à Z ou l'ensemble du code ASCII.

Opérations sur les ensembles

Le type abstrait ensemble définit un certain nombre d'opérations globales : union, intersection, différence symétrique (l'ensemble résultat contient les membres qui sont dans chacun des 2 ensembles opérés mais pas dans l'autre). Ces différentes opérations sont similaires entre elles, il est donc souhaitable de n'avoir qu'une seule fonction les implémentant toutes, et d'offrir un jeu de macros pour accéder à chacune d'entre elles. Voici ce que cela donne :

```
#define UNION           1       /* x est dans set1 ou set2 */
#define INTERSECTION    2       /* x est dans set1 et set2 */
#define DIFFERENCE      3       /* (x dans set1) et (x pas dans set2) */
#define ASSIGN          4       /* set1 = set2 (affectation) */
#define set_union(dest,src)             set_op  ( UNION,        dest, src )
#define set_intersection(dest,src)      set_op  ( INTERSECTION, dest, src )
#define set_difference(dest,src)        set_op  ( DIFFERENCE,   dest, src )
#define assign_set(dest,src)            set_op  ( ASSIGN,       dest, src )

void set_op ( short op, set dest, set src )
{
    SET_UNIT_TYPE    *dest_bits, *src_bits;
   short           size, extra;
    size = src->nunits;
    if ( (short) dest->nunits < size )
       extend ( dest, size );
    extra = (short) dest->nunits - size;
    dest_bits = dest->bits;
    src_bits = src->bits;
   switch ( op ) {
      case UNION        : while ( --size >= 0 )
                            *dest_bits++ |= *src_bits++;
                      break;
       case INTERSECTION : while ( --size >= 0 )
                            *dest_bits++ &= *src_bits++;
                         while ( --extra >= 0 )
                           *dest_bits++ = 0;
                      break;
       case DIFFERENCE   : while ( --size >= 0 )
                            *dest_bits++ ^= *src_bits++;
                      break;
      case ASSIGN       : while ( --size >= 0 )
                            *dest_bits++ = *src_bits++;
                         while ( --extra >= 0 )
                           *dest_bits++ = 0;
                      break;
   }
}   /* set_op () */
```

Aux trois opérations citées précédemment, j'en ai ajouté une quatrième :
l'assignation d'un ensemble à un autre existant déjà. La fonction "set_op" réalise l'ensemble de ces opérations. Il est à noter que l'un des 2 ensembles (appelé "dest") sera remplacé par le résultat de l'opération. Par conséquent, cet ensemble doit avoir au moins la même taille que l'autre (appelé "src"), et il est agrandi si ce n'est pas le cas.

Cette fonction travaille en 2 temps : elle opère d'abord sur les éléments communs des ensembles, c'est à dire sur la taille du tableau de bits de "src", puis si cela est nécessaire sur le reste du tableau de bits de "dest", lorsqu'il est plus le plus grand des deux.

A priori, la réalisation de cette fonction n'a pas l'air très élégante : une meilleure écriture aurait consisté à effectuer la boucle while puis à faire le switch selon le type d'opération, plutôt que d'avoir une boucle identique dans chacun des cas. La méthode employée a l'avantage d'être beaucoup plus efficace en termes de performances, puisqu'on n'a pas besoin de réévaluer le switch à chaque tour de boucle.

Ceci étant dit, l'implémentation des différentes opérations est assez triviale; on travaille sur un mot à la fois, et on applique les opérateurs de manipulation de bits en fonction de l'opération demandée : "ou" pour l'union, "et" pour l'intersection, "ou exclusif" pour la différence symétrique et une simple copie pour l'affectation. Pour la partie supplémentaire de l'ensemble de destination, on part du principe que les éléments de l'ensemble source sont tous à Ø (puisqu'en fait ils n'existent pas); la deuxième boucle (qui n'est pas exécutée si "dest" n'est pas plus grand que "src") modifie donc les bits de "dest" en conséquence, lorsque c'est nécessaire. Je vous laisse vérifier par vous-mêmes que cela donne bien le résultat voulu.

L'assignation d'un ensemble à un autre dans la fonction précédente présuppose que l'ensemble de destination existe déjà. Ce n'est pas toujours le cas : on peut vouloir aussi faire une simple copie d'un ensemble tout en créant cette copie. Bien entendu, on peut utiliser les fonctions "create_set" et "assign_set", mais on peut faire beaucoup mieux; voici donc la fonction "copy_set" qui effectue une copie très efficace :

```
set copy_set ( set s )
{
   set s2;
    s2 = (set) malloc ( sizeof ( struct _set ) );
    if ( s2 == NULL )
        return ( NULL );
     s2->nunits = s->nunits;
     s2->nbits = s->nbits;
     s2->complement = s->complement;
     if ( s->bits == s->defbits ) {
          s2->bits = s2->defbits;
           memcpy ( s2->defbits, s->defbits, UNITS_TO_BYTES ( DEF_UNITS ) );
   } else {
          s2->bits = (SET_UNIT_TYPE *) malloc ( UNITS_TO_BYTES ( s->nunits ) );
        if ( s2->bits == NULL ) {
           free ( s2 );
            return ( NULL );
      }
          memcpy ( s2->bits, s->bits, UNITS_TO_BYTES ( s->nunits ) );
   }
    return ( s2 );
}   /* copy_set () */
```

Le principe de cette fonction est d'allouer directement un tableau de bits de la taille nécessaire, et de copierle tableau de bits original d'un seul coup, grâce à la fonction "memcpy" de la librairie C standard.

Les macros "clear_set" et "fill_set" permettent respectivement de créer un ensemble vide, et un ensemble plein. Elles utilisent toutes deux une autre fonction de la librairie C standard : "memset" pour positionner tous les bits du tableau à Ø ou à 1.

```
#define clear_set(s)     memset ( (s)->bits,   0, UNITS_TO_BYTES ( (s)->nunits ) )
#define fill_set(s)      memset ( (s)->bits, ~0, UNITS_TO_BYTES ( (s)->nunits ) )
```

La macro "clear_set" ci-dessus ne change rien à la taille de l'ensemble, c'est à dire que celui-ci peut

contenir un très grand tableau de bits qui seront tous à Ø après l'opération. On peut donc avoir aussi envie de tronquer l'ensemble lorsqu'on le rend vide, ce qui est fait par la fonction "truncate_set" :

```
void truncate_set ( set s )
{
    if ( s->bits != s->defbits ) {
        free ( s->bits );
        s->bits = s->defbits;
   }
    s->nunits = DEF_UNITS;
    s->nbits = DEF_BITS;
    memset ( s->defbits, 0, UNITS_TO_BYTES ( DEF_UNITS ) );
}   /* truncate_set () */
```

Cette fonction réalise l'équivalent des 2 appels à "delete_set" puis "create_set" sauf qu'elle évite la libération puis la réallocation de la mémoire utilisée par la structure décrivant l'ensemble; elle est donc un peu plus efficace. En revanche, cette fonction n'est vraiment utile que pour réinitialiser un ensemble ayant eu un grand élément; dans le cas contraire, il est préférable d'utiliser la macro "clear_set".

Tests sur les ensembles

Au delà du test d'appartenance d'un membre à un ensemble, il peut être intéressant de savoir si deux ensembles ont des éléments en commun ou non, voire même de vérifier si ils sont identiques. Comme pour les opérations globales sur les ensembles, ces tests sont similaires. Il semble donc naturel de procéder de la même manière, c'est à dire d'écrire une fonction commune, et un jeu de macros correspondant à chacun des tests voulus :

```
#define EQUIVALENT 1            /* set1 et set2 sont equivalents (egaux) */
#define DISJOINT   2            /* set1 et set2 sont disjoints */
#define INTERSECT  3            /* set1 et set2 ont des elements communs */

#define set_equivalent(s1,s2)       ( test_set ( s1, s2 ) == EQUIVALENT )
#define set_disjoint(s1,s2)         ( test_set ( s1, s2 ) == DISJOINT )
#define set_intersect(s1,s2)        ( test_set ( s1, s2 ) == INTERSECT )

short test_set ( set s1, set s2 )
{
   short            size, result;
    SET_UNIT_TYPE    *b1, *b2;
    result = EQUIVALENT;
    size = s1->nunits > s2->nunits ? s1->nunits : s2->nunits;
    extend ( s1, size );
    extend ( s2, size );
    b1 = s1->bits;
    b2 = s2->bits;
    for ( ; --size >= 0; b1++, b2++ )
        if ( *b1 != *b2 ) {
            /*
             * Les 2 ensembles ne sont pas equivalents. Si on constate
             * une intersection, on peut retourner immediatemment ce resultat.
```

```
            * En revanche, on doit continuer a parcourir les 2 sets pour
            * verifier si ils sont disjoints ou pas.
          */
           if ( *b1 & *b2 )
              return ( INTERSECT );
         else
              result = DISJOINT;
      }
    return ( result );        /* EQUIVALENT ou DISJOINT */
}   /* test_set () */
```

Les 3 macros précédentes définissent les tests qu'il est possible d'effectuer (équivalence ou disjonction ou intersection non nulle de 2 ensembles). La fonction "test_set" compare chaque des mots constituant les tableaux de bits des 2 ensembles, après les avoir éventuellement agrandis pour qu'ils aient tous les deux la même taille. Je vous laisse étudier le commentaire ci-dessus pour de plus amples détails sur le fonctionnement intime de cette routine.

Le dernier test que l'on peut effectuer entre deux ensembles consiste à déterminer si un ensemble est un sous-ensemble d'un autre, ce qui donne la fonction "subset" ci-dessous :

```
short subset ( set s, set ss )
{
    SET_UNIT_TYPE    *b, *sb;
   short             common, extra;
    if ( ss->nunits > s->nunits ) {
        common = s->nunits;
        extra = ss->nunits - common;
   } else {
        common = ss->nunits;
       extra = 0;
   }
    b = s->bits;
    sb = ss->bits;
    for ( ; --common >= 0; b++, sb++ )
        if ( ( *sb & *b ) != *sb )       /* des membres du subset */
           return ( 0 );                 /* ne sont pas dans le set */
    while ( --extra >= 0 )
        if ( *sb++ )
           return ( 0 );
    return ( 1 );
}   /* subset () */
```

Cette fonction renvoie 1 si l'ensemble "ss" est un sous-ensemble de "s" et Ø dans le cas contraire. A noter que l'ensemble vide est sous-ensemble de n'importe quel ensemble, y compris de l'ensemble vide lui-même; cette fonction renverra donc 1 dans ces différents cas. On vérifie d'abord que les membres appartenant à la partie commune du sous-ensemble recherché appartiennent aussi tous au sur-ensemble, puis on s'assure que lorsque le sous-ensemble a un tableau de bits plus grand que le sur-ensemble, il n'y a aucun bit à 1; le contraire indiquerait que le sous-ensemble a des membres qui ne sont pas dans le sur-ensemble (les bits manquants dans ce dernier sont considérés comme étant des bits à Ø).

Vous noterez que toutes les fonctions de test ne sont pas prévues pour fonctionner avec un ensemble complémenté. Si vous voulez aussi utiliser cette fonctionnalité, il vous faudra soit inverser physiquement les ensembles, soit modifier les fonctions en question pour tenir compte de l'attribut complément.

Affichage et parcours d'un ensemble

Le dernier jeu de fonctions permet de parcourir un ensemble afin d'obtenir tous les membres, par exemple pour afficher le contenu de l'ensemble. Ces fonctions ne sont compilées que si le symbole "ALL" est défini lors de la compilation (par exemple en utilisant la commande "compile Set.cc cc=(-dALL)"), car elles sont d'un usage moins fréquent :

```
short next_member ( set s, unsigned short *context )
{
    SET_UNIT_TYPE    *b, m;
    /*
     * On saute tous les premiers elements du tableau ne comptant pas de membre,
     * en tenant compte du fait que le set peut etre complemente.
     */
    if ( *context == 0 ) {
        m = s->complement ? ~0 : 0;
        for ( b = s->bits; *b == m && *context < s->nbits; ++b )
            *context += UNITS_TO_BITS ( 1 );
    }
    /*
     * Maintenant on cherche le premier prochain bit a 1, indiquant un membre
     * tout en tenant compte d'un set complemente.
     */
    while ( (*context)++ < s->nbits )
        if ( test_member ( s, *context - 1 ) )
            return ( *context - 1 );
    return ( -1 );              /* indique qu'il n'y a plus de membres */
}   /* next_member () */

void scan_set ( set s, void (*func)( unsigned short member ) )
{
    unsigned short  context;
    short           member;
    context = 0;
    member = next_member ( s, &context );
    while ( member >= 0 ) {
        (*func)( member );
        member = next_member ( s, &context );
    }
}   /* scan_set () */
```

La fonction "next_member" retourne chacun des membres un à un en utilisant un système de contexte que j'avais expliqué dans l'article sur les listes. La fonction "scan_set" utilise "scan_set" pour appeler une fonction passée en paramètre avec chacun des membres obtenus. Encore une fois, il n'y a pas grand chose à ajouter aux commentaires contenus dans la définition de ces fonctions.

La dernière fonction définie dans la librairie est la fonction "count_members"; elle calcule le nombre de membres d'un ensemble. Elle est utilisée par la macro.

Elles ne sont aussi intégrées dans la librairie que si le symbole "ALL" est défini (ce qui est le cas dans la version présente sur la disquette). La raison en est que l'implémentation choisie utilise un tableau de 256 octets, comme cela a été expliqué dans le précédent article. Je ne la reliste donc pas ici.

Voilà qui termine la description de la librairie de manipulation des ensembles. Il y aurait certes beaucoup d'autres choses à dire à son sujet, mais je pense que l'étude du source devrait répondre à toutes vos questions; les commentaires sont notamment très détaillés. Si toutefois, vous avez trouvé certaines explications un peu trop succintes et que vous désirez avoir plus d'informations, n'hésitez pas à m'en faire part; je reviendrai alors dans un prochain article sur les points qui vous ont causé des problèmes. De même, la lecture du source de la librairie et du programme de démonstration vous indiqueront en détail comment utiliser les différentes fonctions disponibles, au cas où les explications précédentes seraient insuffisantes.

Comme vous avez pu le constater en lisant cet article, j'ai utilisé le C à son maximum, notamment au niveau de certaines macros. Ceci semble déplaire fortement à l'optimiseur d'ORCA/C, et le programme ne fonctionne plus si il est compilé avec les optimisations; je n'ai pas d'ailleurs pas cherché où était l'origine des problèmes faute de temps. En revanche, il semble ne pas y avoir de problème lorsque le code n'est pas optimisé. N'hésitez pas pour autant à me signaler tout bug que vous pourriez rencontrer.

Sur la disquette GS Infos

- "Set.h" : le fichier interface de la librairie décrivant le type ensemble ainsi que les macros et les prototypes des fonctions permettant de le manipuler.
- "Set.cc" : l'implémentation des fonctions de manipulation des ensembles.
- "Set.lib" : librairie de manipulation des ensembles.
- "Test.Set.cc" : programme de test d'une partie des fonctions et des macros gérant les ensembles; il permet de voir comment les utiliser, et n'est certes pas complet.
- "Test.Set" : programme de test exécutable.