Université de Nice-Sophia Antipolis — Deug MIAS-MI 1
Algorithmique & Programmation — 2003–2004

TP N° 11

Programmation modulaire

1 Préambule

Le but de cette séance est de vous familiariser avec la programmation modulaire. Nous avons vu deux manières de représenter un graphe (TD 08) : matrice d'adjacence et listes d'adjacence. Nous désirons implémenter divers algorithmes simples sur les graphes, mais nous voulons pouvoir choisir le mode de représentation du graphe. Dans ce TP, il ne sera question que de graphes orientés.

Vous créerez un module `graphmatrix.pas` et un module `graphlist.pas`, contenant les types et les procédures nécessaires. Toutes les fonctionnalités seront testées à l'aide d'un programme `graph.pas` dont l'en-tête sera :

```
program graph;
import graphm in 'graphmatrix.pas';
{import graphl in 'graphlist.pas';}
```

Ce programme appellera simplement les diverses procédures et fonctions implémentées. Pour le compiler, on utilisera l'option `automake` :

```
gpc --automake -o graph.exe graph.pas
```

Suivant que le programme importe le module `graphmatrix.pas` ou le module `graphlist.pas`, l'implémentation utilisée sera complètement différente, mais le programme devra fournir exactement les mêmes résultats.

Nous venons de voir que la manière d'importer un module depuis un programme est bien simple. La construction est simplement

```
import {nom du groupe d'exports} in {nom du fichier} ;
```

L'organisation d'un module est la suivante :

```
module {nom du module} interface ;
export {nom du groupe} = ( {liste des noms exportés} ) ;
{déclaration des noms exportés : constantes, types, variables,
  procédures et fonctions ; pour ces dernières, seul l'en-tête
  apparaît dans la partie interface ; pour les types, la définition
  n'apparaît pas}
end.

module {nom du module} implementation ;
import {importations variées, séparées par des points-virgules} ;
{déclaration des types exportés, cette fois-ci complète ; déclaration
  du corps des procédures et fonctions exportées, la liste de
  paramètres n'ayant pas besoin d'être répétée}
{déclarations non exportées}
end.
```

On peut combiner la partie interface et la partie implémentation. Dans ce cas, le mot-clé **`interface`** manque en tête, et la séparation entre les deux parties du module est simplement marquée par une ligne ne comportant que **`end;`**.

Vous trouverez sur le site les deux fichiers `graphmatrix.pas` et `graphlist.pas`, qui vous serviront de « squelette » pour les deux modules que vous allez devoir construire.

2 Représentation matricielle

Le but est la création (et le test) du module `graphmatrix.pas`.

a. Programmation du module

L'en-tête du module `graphmatrix` est le suivant :

```
module graphmatrix;
export graphm=(T_Graph,T_vertex,T_edge,createGraph,printGraph,
               isEdge,addEdge,deleteEdge,addVertex,reverseGraph);
```

Les déclarations de base seront les suivantes :

```
const
   sizemax  = 1000;
type
   T_vertex = 1..sizemax;
   T_edge   = record
                 first : T_vertex;
                 last  : T_vertex
              end;
   T_Graph  = record
                 graph : array[T_vertex,T_vertex]of integer;
                 size  : T_vertex
              end;
procedure createGraph (var g : T_Graph);

procedure printGraph(g : T_Graph);

function isEdge(e : T_edge;g: T_Graph):boolean;

procedure addEdge(e : T_edge;var g: T_Graph);

procedure deleteEdge(e : T_edge;var g: T_Graph);

procedure addVertex(var g : T_Graph);

procedure reverseGraph(var g : T_Graph);

end;
```

Décrivons à présent chacune des procédures et fonctions :

- La procédure `createGraph` permet à l'utilisateur de créer un graphe du nombre de sommets souhaité (entre 1 et `sizemax`), puis d'indiquer la présence ou l'absence de chacun des arcs possibles.
- La procédure `printGraph` affiche la matrice d'adjacence du graphe indiqué en paramètre.
- La fonction booléenne `isEdge` permet de tester la présence de l'arc indiqué en paramètre dans le graphe.
- La procédure `addEdge` permet d'ajouter un arc.
- La procédure `deleteEdge` permet la suppression d'un arc.
- La procédure `addVertex` autorise l'ajout d'un sommet.
- La procédure `reverseGraph` change le graphe g passé en paramètre en son graphe inverse (i.e. le graphe orienté obtenu à partir de g en inversant l'orientation de chaque arc).

```
const
   sizemax = 1000;
```

```
type
   T_vertex = 1..sizemax;
   T_edge   = record
                 first : T_vertex;
                 last  : T_vertex
              end;
   T_Graph  = record
                 graph : array[T_vertex,T_vertex]of integer;
                 size  : T_vertex
              end;
procedure createGraph (var g : T_Graph);

procedure printGraph(g : T_Graph);

function isEdge(e : T_edge;g: T_Graph):boolean;

procedure addEdge(e : T_edge;var g: T_Graph);

procedure deleteEdge(e : T_edge;var g: T_Graph);

procedure addVertex(var g: T_Graph);

procedure reverseGraph(var g: T_Graph);

end;

procedure createGraph;
var i,j : integer;
begin
   writeln('taille␣du␣graphe');
   readln(g.size);
   for i:=1 to g.size do
      for j:=1 to g.size do
      begin
         repeat
            writeln('Arete␣entre␣',i,'␣et␣',j,'␣("1"␣ou␣"0")');
            readln(g.graph[i,j])
         until (g.graph[i,j]=0) or (g.graph[i,j]=1)
      end
end; { createGraph }

procedure printGraph;
var i,j : integer;
begin
   for i:=1 to g.size do
   begin
      for j:=1 to G.size do
         write(g.graph[i,j]:4);
      writeln
   end
end; { printGraph }


function isEdge;
begin
   isEdge:= g.graph[e.first,e.last]=1
end; { searchEdge }
```

```
procedure deleteEdge;
begin
   g.graph[e.first,e.last]:=0
end; { deleteEdge }

procedure addEdge;
begin
   g.graph[e.first,e.last]:=1
end; { addEdge }

procedure addVertex;
var i : integer;
begin
   g.size:=g.size+1
end; { addVertex }

procedure reverseGraph;
var i,j : T_vertex;
   tmp  : integer;
begin
   for i:=1 to g.size do
      for j:=i+1 to g.size do
      begin
         tmp:=g.graph[i,j];
         g.graph[i,j]:=g.graph[j,i];
         g.graph[j,i]:=tmp
      end
end;

end.
```

.....

b. Test

Testez chacune des possibilités offertes par le module `graphmatrix` à l'aide du programme `graph`.

```
program graph;

import graphm in 'graphmatrix.pas';
{import graphl; in 'graphlist.pas';}

var
   g : T_Graph;
   e : T_edge;
begin
   createGraph(g);
   printGraph(g);
   writeln;
   addVertex(g);
   printGraph(g);
   Writeln('Nombre de sommets: ',g.size);
   write('Ajouter une arete: origine '); readln(e.first);
   write('arrivee ');read(e.last);
   addEdge(e,g);
   printGraph(g);
   write('Suppression d''une arete: origine ');
   readln(e.first);
```

```
   write('arrivee ');
   readln(e.last);
   if isEdge(e,g) then
      deleteEdge(e,g)
   else
      writeln('L''arête (',e.first,',',e.last,') n''est pas une arête du graphe.');
   printGraph(g);
   writeln('Graphe inverse');
   reverseGraph(g);
   printGraph(g);
   writeln
end.
```

.....

3 Représentation par listes d'adjacences

Dans cette partie, vous réutiliserez différents algorithmes de manipulation de listes vus au cours des précédentes séances de TD et de TP.

c. Programmation du module

Voici l'en-tête et les déclarations du module `graphlist` :

```
module graphlist;
export graphl=(T_Graph,T_vertex,T_edge,createGrah,printGraph,
               isEdge,addEdge,deleteEdge,addVertex,reverseGraph);

const
   sizemax  = 1000;
type
   T_vertex = 1..sizemax;
   T_edge   = record
                 first : T_vertex;
                 last  : T_vertex
              end;
   qp       = ^qnode;
   qnode    = record
                 vertex : T_vertex;
                 next   : qp
              end;
   T_Graph  = record
                 graph : array[T_vertex]of  qp;
                 size  : T_vertex
              end;
```

Implémentez dans ce module les mêmes fonctionnalités que précédemment. Les déclarations de procédures et fonctions seront identiques à celles du module `graphmatrix` (pourquoi ?). N'oubliez pas cependant l'initialisation des pointeurs.

```
procedure initGraph( var g : T_Graph);
var i : integer;
begin
   for i:=1 to sizeMax do
      g.graph[i]:=nil
end; { initGraph }

procedure createGraph;
var
```

```
   i,j  : integer;
   node : qp;
begin
   writeln('taille du graphe');
   readln(g.size);
   initGraph(g);
   for i:=1 to g.size do
   begin
      writeln('Entrez les successeurs de ',i, '(0 pour passer au sommet suivant)');
      repeat
         readln(j)
      until (j>=0) and (j<= g.size);
      while j<>0 do begin
         new(node);
         node^.vertex:=j;
         node^.next:=g.graph[i];
         g.graph[i]:=node;
         repeat
            readln(j)
         until (j>=0) and (j<= g.size)
      end
   end
end; { createGraph }

procedure printGraph;
var
   i    : integer;
   node : qp ;
begin
   for i:=1 to g.size do
   begin
      write(' Successeurs de ',i,' :');
      node:=g.graph[i];
      while node<>nil do begin
         write(node^.vertex:4);
         node:=node^.next
      end;
      writeln
   end
end; { printGraph }


function isEdge;
var
   found : boolean;
   node  : qp;
begin
   node:=g.graph[e.first];
   found:=false;
   while (node<>nil) and not found do begin
      found:= (node^.vertex)=(e.last);
      node:=node^.next
   end;
   isEdge:=found
end; { isEdge }

procedure deleteEdge;
var
```

```
   node,fathernode : qp;
   found           : boolean;
begin
   node:=g.graph[e.first];
   fathernode:=g.graph[e.first];
   found:=false;
   if (node<>nil) and ((node^.vertex)=(e.last)) then
   begin
      found:=true;
      g.graph[e.first]:=node^.next
   end
   else begin
      node:=node^.next;
      while (node<>nil) and not found do begin
         found:= (node^.vertex)=(e.last);
         if found then
            fathernode^.next:=node^.next
         else begin
            fathernode:=node;
            node:=node^.next
         end
      end
   end
end; { deleteEdge }

procedure addEdge;
var node : qp;
begin
   new(node);
   node^.vertex:=e.last;
   node^.next:=g.graph[e.first];
   g.graph[e.first]:=node
end; { addEdge }

procedure addVertex;
begin
   g.size:=g.size+1
end; { addVertex }

procedure reverseGraph;
var
   node  : qp;
   gtemp : T_Graph;
   i     : integer;
   e     : T_Edge;
begin
   initGraph(gtemp);
   for i:=1 to g.size do
   begin
      node:=g.graph[i];
      e.last:=i;
      while node<>nil do begin
         e.first:=node^.vertex;
         addEdge(e,gtemp);
         node:=node^.next
      end;
   end;
   for i:=1 to  g.size do
```

```
      g.graph[i]:=gtemp.graph[i]
end; { reverseGraph }

end.
```

d. Testez le module `graphlist` à l'aide du programme `graph`.