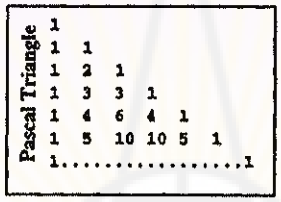
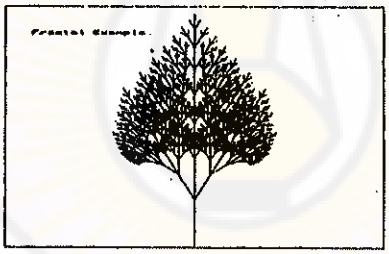
منشورات جامعة دمشق
كلية الهندسة المعلوماتية

البرمجة (2)





الدكتور زهير دحروج الدكتورة غيداء ربداوي

جامعة دمشق
١٤٣٤ - ١٤٣٥ هـ
٢٠١٣ - ٢٠١٤ م


وقل رب زدني علما
جامعة دمشق
Damascus University

المحتويات

– مقدمة		1
– الفصل الأول	بنية المعطيات: مجموعة «set»	5
– الفصل الثاني	بنية المعطيات: تسجيلة «record»	17
– الفصل الثالث	الملفات النصية «text»	33
– الفصل الرابع	الملفات الثنائية «file»	46
– الفصل الخامس	نمط المعطيات: مؤشِّر «pointers»	69
– الفصل السادس	تعليمة القفز «goto»	97
– الفصل السابع	توزيع البرنامج على وحدات Modular Programming	102
– الفصل الثامن	توطئة في الرسم البياني بواسطة المستقيمات	126
– الفصل التاسع	البرمجة غرضية التوجه في Turbo Pascal	158
– الفصل العاشر	برمجة الواجهات التخاطبية باستخدام أغراض المكتبة Turbo Vision	214
– أمثلة عامة		241
– مسائل للبرمجة		263
– ملحق (١)	المفردات المُعرَّفة في لغة باسكال	282
– ملحق (٢)	نصائح في البرمجة	285
– المراجع		287

Damascus University


وقل رب زدني علما
جامعة دمشق
Damascus University

مقدمة

يأتي كتاب **البرمجة (2)** متمماً للمفاهيم الأساسية في البرمجة المعروضة في الكتاب الأول **البرمجة (1)**. قدَّم الكتاب الأول أساسيات التفكير البرمجي، لحل مسألة باستخدام الحاسوب، بالاعتماد على فكرة التبسيط المتتالي للمسألة والتعبير عنه بلغة خوارزمية Pseudo-Code. تعكس بُنى التعبير عن الخوارزمية خطوات الحل الذي نريد، وهي شبيهة وقريبة من التعليمات الأساسية المتاحة في لغات البرمجة المختلفة. وتعتمد آلية التفكير البرمجي، التي نهجناها، على البرمجة المهيكلة Structured Programming ، ويُراد بها التعبير عن مسار الحل ببنى التحكم المهيكلة، فلكل تعليمة تحكم (تعليمة شرطية أو تعليمة تكرارية أو التعليمات المكافئة لأي منهما) كتلة من التعليمات، و هذه الكتلة قد تحتوي بدورها على تعليمة تحكم أخرى..وهكذا..

عالج الكتاب الأول تعليمات التحكم المختلفة إضافةً إلى الأنماط البسيطة للمعطيات (عددية، منطقية، محرفية). وقدَّم من الأنماط المركبة بنية الجدول أو المصفوفة array.
كما عالج الكتاب الأول مفهوم البرنامج الجزئي كاملاً.وبهذا يكون المتعلم قد استوعب الأساسيات، للتعبير عن التعليمات اللازمة لحل مسألة وصياغتها في برنامج، أوتنظيم هذا البرنامج في مجموعة برامج جزئية ينجز كل منها جزءاً محدداً من العمل، مع إمكان إطلاق عمومية الاستخدام لهذه البرامج الجزئية بتعريف وسطاء الدخل والخرج لكل منها.

ويأتي الكتاب الحالي ليقدِّم المزيد في مجال بنى المعطيات المُركَّبة Composed Data Structures: (المجموعة set، التسجيلة record، الملف file، المؤشرات pointers) . قد

تكون بنية المعطيات «المجموعة set » غير متاحة في الكثير من لغات البرمجة، ولكنها تعتبر مثالاً جيداً على إمكان الحل لمسألة ببنى معطيات مختلفة عن الأشكال التقليدية. أما تَعرُّف بنية «التسجيلة record » والتعامل مع الملفات فهما أساسيان لا يستغنى عنهما في العديد من التطبيقات.

أما الذين يريدون دراسة بنى المعطيات المتقدمة وخوارزمياتها، فلابد لهم من فَهْم استخدام المؤشِّرات «pointers» فهماً جيداً، علماً بأن شرح المؤشرات في هذا الكتاب مُدّعم بالأمثلة والأشكال التوضيحية لتذليل صعوبات فهمها.

وكما في الكتاب الأول، تعتمد الأمثلة البرمجية على لغة باسكال القياسية (لغة pascal كما عرّفها مصممها الأول Niklaus Wirth)، وهي متوافقة -لحد بعيد- مع جميع المترجمات المتاحة للغة pascal. و باستعراضنا للأفكار البرمجية العامة التي أسلفنا ذكرها نكون قد عالجنا لغة Pascal-Wirth كلها، ونحن نعتقد ضرورة معرفة الأفكار البرمجية التي تحويها كاملةً كأساس متين في البرمجة. وهنا نعود للتأكيد مرةً أخرى على أن الهدف من الكتاب ليس هو عرض لغة باسكال بقدر ماهو تعليم أساسيات البرمجة من بنى التحكم وبنى المعطيات الرئيسة. لكن عرض الأفكار البرمجية دون أمثلة ملموسة قابلة للتجريب والتوثق يبقى عملاً قاصراً. وقد سبق أن شرحنا في مقدمة الكتاب الأول سبب اختيارنا باسكال وعاءً لأمثلتنا البرمجية، لما تشمله من غنى في بنى التحكم وبنى المعطيات، إضافة إلى انتظامها القواعدي وخاصّة التأكد من دلالة العمليات (اختبار توافق الأنماط مع العمليات).

لن يكون الانتقال إلى أية لغة إجرائيّة أخرى (Procedural Language) أكثر من مسألة تأقلم، وتعرّف كيفية التعبير في اللغة الجديدة عن المفاهيم التي اكتسبها المتعلّم هنا. وبرغم

أن هدفنا الرئيس لم يكن تعليم لغة باسكال، ألا أننا توخينا الدقة في تناولنا الأمثلة بها، وصار بالإمكان أيضاً اعتماد كتابَيْ البرمجة (١) و (٢) كمرجعين أساسيين في تناول لغة باسكال، مما يسمح تحديد ماهو قياسي فيها ومايعتبر توسّعاً في مترجم اللغة ومحيط تطويرها.

وبسبب انتشار استخدام مترجمات Turbo Pascal (النسخة 5 حتى 7)، التي تستثمر المحيط البرمجي الخاص بالحواسيب الشخصية المتوافقة مع IBM-PC ، عمدنا إلى تخصيص فصلين يتضمنان بعض الإضافات المفيدة (إلى لغة باسكال النظامية أو Pascal-Wirth) : فصل يشرح استخدام المجتزآت أو الوحدات البرمجية Units كوسيلة للترجمة المنفصلة، أي توزيع نص البرنامج على عدة ملفات، وكأداة لاغنى عنها لبناء مكتبة من البرمجيات القابلة لعدة استخدامات. ويعطي الفصل الآخر بعض الأساسيات في استخدام الإظهار البياني، وهذا مما يمكّن من استغلال إمكانيات الشاشات المتاحة في الإظهار البياني في عدد من التطبيقات المفيدة.

وأخيراً، وبسبب ظهور البرمجة الغرضية التوجهObject Oriented Programming واعتمادها في مترجمات Turbo Pascal (النسخة 5 حتى 7)، فقد رأينا أنه من المفيد، للحصول على مرجع كامل في لغة باسكال وتوسّعاتها، أن نعرّج على هذا النمط من البرمجة، فنعرض في فصلين المفاهيم الأساسية فيه، ففي الفصل التاسع نعرض مفهوم الغرض والطرائق والوراثة، أما في الفصل العاشر فنتطرق إلى مكتبة Turbo Vision لنستخدم "أغراضها" في بناء تطبيقات ذات واجهات تخاطبية مريحة، وستمكننا أغراض هذه المكتبة من تنفيذ ذلك بسرعة كبيرة بسبب مفهوم "إعادة الاستخدام"، مما يوضّح الفائدة الكبيرة لهذا النوع من البرمجة.

يتميَّز الكتاب بتركيزه الاهتمام في الأمثلة المنتقاة بعناية بالغة، والتي تُلِحّ على إيصال الفكرة البرمجية وإظهار أهمية وظروف استخدامها. لقد عمدنا إلى وضع كل مثال في برنامج متكامل ذي فائدة تطبيقية. وجرى اختبار جميع الأمثلة البرمجية على الحاسوب قبل ضمها إلى نص الكتاب. ونحن ننصح المتعلمين أن يُنعموا النظر في هذه الأمثلة، وأن يجربوها ويعدلوها بغية التوثُّق من هضم الأفكار جيداً. إن دراسة البرمجة دون الخوض في البرامج وتشريحها وتجريبها بهدف الفهم العميق، تبقى من باب العلم بالشيء لا تَعَلُّمه. وكما قالت العرب سابقاً، إذا أردت أن تصبح شاعراً يجب أن تحفظ عشرة آلاف بيت من الشعر على الأقل، نقول نحن هنا، إن أردت أن تصبح مبرمجاً بحق، فيجب أن تجرب كتابة مئة برنامج نموذجي بيديك على الأقل.

إن المسائل الموجودة في آخر الكتاب موضوعة كنماذج ينبغي كتابة برامجها واختبارها على الحاسوب. ومالم يدرِّب المتعلم نفسه على ذلك، لن يصل إلى تعلم أساسيات البرمجة.

نرجو أن تعمّ الفائدة بنشر هذا الكتاب، وسنكون شاكرين لأية ملاحظات وتصحيحات للوصول بالمادة المعروضة إلى مستوىً أفضل.

د. زهير دحروج د. غيداء ربداوي

دمشق 1999

الفصل الأول

نمط المعطيات: مجموعة set

مقدمة:

كلنا يعرف المجموعة بالمعنى الرياضي العام، فالعناصر فيها ليست مكرّرة ولا يوجد بالضرورة علاقة ترتيب معرّفة على هذه العناصر.

يمكن أن تكون بنية المعطيات "مجموعة" مفيدةً في بعض التطبيقات إذا كانت هذه البنية مزوّدة بالعمليات المألوفة على المجموعات من انتماء وتقاطع واجتماع وفرق.

تسمح لغة البرمجة Pascal بتعريف بنية معطيات: مجموعة من عناصر تنتمي إلى نمط متقطع ، وذلك بتعريفها للنمط على أنه «مجمـــوعة» set of.

سنرى في هذا الفصل كيفية التعامل مع المجموعات في لغة Pascal، وسندرك أهميتها لبعض العمليات التي قد تكون صعبة التحقيق في حال عدم وجود نمط معطيات «مجموعة».

التصريح عن مجموعة set :

يأخذ التصريح عن متحول من نمط مجموعة الشكل التالي:

‹اسم المتحول›: set of ‹نمط متقطع›;

النمط المتـــقطع‹discrete type› هو أحـــد الأنماط (integer, char, boolean, Enumerated)

أمثلة:

```
S1 : set of 10..60;
smalls : set of 'a'..'z';
charset : set of char;
```

ملاحظة هامة:

مع أن لغة Pascal القياسية لاتضع حداً أعظم على طول المجال الذي تنتمي إليه عناصر المجموعة، فإن أكثر المترجمات Compilers لاتستطيع ذلك وتكتفي كحد أعظم بـ 256 عنصراً.

فالتصريح : s:set of integer; أو s: set of 1..500 لا تقبله أكثر المترجمات. ويرجع السبب على الأغلب إلى تحقيق فعالية أكثر في طريقة الخزن وتنفيذ العمليات.

العمليات على المجموعات:

1- الإسناد و التعبير عن الثوابت:

ليكن لدينا المتحول VS من نمط مجموعة.

```
VS : set of <discrete-type>;
```

يمكن أن يأخذ الإسناد الشكل:

```
VS := [ e1, e2, e3, f1..f2];
```

e1, e2, e3, f1,f2 تعابير من نمط عناصر المجموعة أي من النمط <discret-type> .

الإسناد بهذا الشكل يعني جعل المجموعة VS تحوي العناصر e1,e2,e3 إضافةً إلى العناصر الواقعة في المجال بين f1 و f2.

وتكتب المجموعة الخالية بالشكل: [].

أمثلة:

```
chs : set of char;
chs := ['a','b','c','u'..'z'];
```

إذن chs تحوي الأحرف a,b,c إضافةإلى الأحرف من u إلى z.

```
is :set of 1..50;
i, j : integer;
```

is := [5]; المجموعة تحوي عنصراً وحيداً هو الرقم 5.

is := [i, j, i-j]; المجموعة تحوي ثلاثة عناصر قيمة i وقيمة j وقيمة j-i.

العمليات الجبرية على المجموعات:

لتكن A و B مجموعتين عناصرهما من نمط واحد. وليكن e متحولاً له النمط نفسه أيضاً

يُعبَّر عن عمليات المجموعات كما يلي:

اسم العملية	العلاقة الرياضية	العملية في Pascal
التقاطع	$A \cap B$	A * B
الاجتماع	$A \cup B$	A + B
الفرق	A / B	A - B
احتواء أو مساواة	$A \supseteq B$	A >= B

احتواء تام	$B \supset A$	$B > A$
مساواة	$A = B$	$A = B$
عدم المساواة	$A \neq B$	$A <> B$
الانتماء	$e \in A$	e in A

أمثلة برمجية:

مثال1: اختبار الانتماء لمجموعة

يمكن أن نستخدم الشكل الثابت للمجموعة كاستخدامنا للشكل الثابت لأي نمط، أي دون تصريح، فكما يمكن استخدام الثوابت الرقمية (صحيحة أو كسرية) مثل: 150 و 3.1416 يمكن أيضاً استخدام ثابت من نمط مجموعة مثل: ['a','e','i','u'].

وكمثال تطبيقي على استخدام المجموعات بشكلها الثابت، يمكننا أن نضع خيارات المستخدم (الأوامر) في برنامج تخاطبي، بطريقة إدخال أحد الأحرف لاختيار العملية التي يريد تنفيذها (أول حرف لاسم العملية مثلاً). و نريد أيضاً أن يكون البرنامج متسامحاً مع المستخدم إن أخطأ في إدخال الحرف المناسب. يمكن تحقيق ذلك بالجزء البرمجي التالي:

```
repeat
  writeln('  (A)dd ');
  writeln(' (D)elete ');
  writeln(' (C)hange ');
  writeln(' (Q)uit ');
  writeln(' Input operation, A,D,C, Q ');
  readln( choice );
until ( choice in  ['A','a','D','d','C','c','Q','q']  );
```

تقوم مجموعة التعليمات السابقة باظهار الخيارات للمستخدم وطلب إدخال أحد الأحرف ['A','a','D','d','C','c','Q','q'] مكرّرةً ذلك حتى يُدخَل الخيار الصحيح. أي إنها تقوم بتكرار إظهار لائحة الخيار حتى إدخال حرف ينتمي إلى مجموعة الأحرف A,B,C,D أو الأحرف الصغيرة الموافقة:

choice in ['A','a','D','d','C','c','Q','q'] .

إن كتابة الشيء ذاته، بغير علاقة الانتماء إلى المجموعة، يجعل الكتابة على شكل شرط منطقي، مع عمليات or ، طويل جداً وليس بالأناقة نفسها :

(choice='A' or choice='a' or choice='D' or choice='d' or choice='C' or choice='c' or choice='Q' or choice='q')

مثال2: تقاطع مجموعتين

ليكن لدينا نظام دراسي يختار فيه الطلاب المواد التي يريدون متابعتها فهناك مجموعة من الطلاب تتبع مادة البرمجة prog ومجموعة تتبع مادة اللغة الانكليزية eng. ولتكن لدينا قوائم بأرقام الطلاب في كل من المادتين (مثلاً : الأرقام مخزونة في جدولين arrays).

المطلوب معرفة الطلاب (أي أرقام الطلاب) الذين يتبعون المادتين معاً.

لو فكرنا بالحل، دون اللجوء إلى المجموعات كبنية معطيات، لرأينا أن الحل طويل. ويقتضي المرور على كل عناصر الجدول الأول، والبحث عن وجود العنصر في الجدول الثاني، أي مقارنة كل عنصر في الأول بكل عناصر الجدول الثاني.

أما إذا أسندنا أرقام الطلاب إلى مجموعتين فالجواب عن المطلوب يكون بعملية بسيطة هي عملية التقاطع بينهما.

الخوارزمية:

سنفترض أن أرقام الطلاب محصورة ضمن المجال N..1 حيث N لاتتجاوز 255.

ليكن Aprog، Aeng الجدولين اللذين يحويان أرقام الطلاب للمادتين eng, prog على الترتيب و Nprog و Neng هي عدد الطلاب في كل منهما.

سنستخدم في الحل المجموعتين Sprog، Seng اللتين تحويان أرقام الطلاب للمادتين eng, prog على الترتيب. ستكون المراحل كالتالي:

1- {إسناد أرقام الطلاب من الجداول **array** إلى مجموعات **set**}

2- حساب التقاطع: S2 = Seng * Sprog

3- { كتابة مجموعة التقاطع S2}

الخطوة 1 سهلة، والخطوة 2 مباشرة، لذا سنكتفي بتفصيل الخطوة 3 لأن المجموعة كنمط مُركّب لايمكن كتابتها مباشرةً ولا يمكن تحديد عناصرها بسهولة، كما هو الحال بالوصول المباشر إلى عناصر المصفوفة بالأدلة.

تقوم كتابة مجموعة من عناصر من مجال N..1 على اختبار كل عنصر من المجال، إنْ كان منتمياً إلى المجموعة، وكتابته في هذه الحالة ثم نزعه من المجموعة. نكرر العمليات السابقة حتى يتحقق أحد الشرطين: فراغ المجموعة أو الانتهاء من عناصر المجال.

ونكتب ذلك بالشكل الخوارزمي التالي:

3- (كتابة مجموعة التقاطع S2)
i := 1
مادام (S2 ليست خالية) و (i <=N) كرر
| إذا i تنتمي لـ S2 نفذ
| | اكتب i
| انزع i من S2: S2 = S2 - [i]
| i = i + 1

```
Program SetArray( input, output );
(* intersection of tow sets. elements are first stored in arrays *)
Var
  Aprog, Aeng : array[1..100] of integer;
  Sprog, Seng, S2 : set of 1..150;
  nprog, neng, i  : integer;

Begin
  (* the total number of students in the arrays: nprog, neng *)
  writeln(' Total number in Programming Course < 100 :');
  readln(nprog);
  writeln(' Input ', nprog,' student numbers :');
  for i:=1 to nprog do
      readln( Aprog[i] );
  writeln(' Total number in English Course < 100 :');
  readln(neng);
  writeln(' Input ', neng,' student numbers :');
```

```
for i:=1 to neng do
      readln( Aeng[i] );
 (* Assign student numbers from the arrays to sets *)
Sprog := [];
for i:=1 to nprog do
   Sprog := Sprog + [ Aprog[i] ];
Seng := [];
for i:=1 to neng do
   Seng := Seng + [ Aeng[i] ];
(* intersection *)
S2 := Sprog * Seng;
(* writing S2 *)
writeln(' Students in both courses are :');
i := 1;
while ( (i<=150)  and (S2<>[]) )  do
 begin
   if  ( i in S2 ) then
    begin
     . writeln(i);
       S2 := S2 - [i];
    end;
   i := i + 1;
 end;
End.
```

تنفيذ البرنامج : (الفاصل هو معطيات المستخدم)

```
Total number in Programming Course < 100 :
10
Input 10 student numbers :
1
4
5
10
15
17
20
```

```
30
41
150
 Total number in English Course < 100 :
 15
 Input 15 student numbers :
5
9
14
15
25
30
41
50
60
90
100
105
106
120
150
 Students in both courses are :
5
15
30
41
150
```

مثال3: الحصول على الأعداد الأولية

يمكن استخراج الأعداد الأولية، ضمن مجال معين، دون استخدام عمليات ضرب أو قسمة وذلك باتباع خوارزمية خاصة تسمى خوارزمية المنخل أو sieve. وهي مبنية على فكرة بسيطة (خوارزمية إيراستوتين Erasthotene): نضع الأعداد من 2 إلى

نهاية المجال المطلوب في مجموعة (سنسميها مجموعة المنخل sieve)، ثم نبدأ بانتزاع الأعداد الأولية منها ووضعها في مجموعة primes، وكلما أخذنا عدداً ننزع كل مضاعفاته من المنخل (المضاعفات تحسب جمعاً). أما كيفية تحديد العدد الأولي فتجري بأخذ أول (أصغر) عنصر بين العناصر المتبقية في مجموعة المنخل.

مثال:

الأعداد الأولية في المجال 2..10

المنخل sieve	العنصر الأول next	مضاعفاته j	الأعداد الأولية primes
2,3,4,5,6,7,8,9,10	2	2,4,6,8,10	2
3,5,7,9	3	3,9	2, 3
5,7	5	5	2,3,5
7	7	7	2,3,5,7

لاشيء

يقوم البرنامج بتجسيد الفكرة الخوارزمية الواردة آنفاً، وكتابة النتيجة أي محتويات المجموعة primes على شكل مصفوفة للأعداد من 1 حتى 255، نكتب في هذه المصفوفة الأعداد الأولية فقط والباقي يترك فارغاً ——.

```
Program SievePrimes( input, output );
(* generate primes using sets, without multplication or division *)
Const N=255;
Var
  sieve, primes : set of 2..N;
  next, j       : integer;
Begin
 (* initialize *)
  sieve := [2..N];
  primes := [];
  next := 2;
  repeat    (* find next prime *)
    while (not (next in sieve)) do
       next := succ( next );
    primes := primes + [next];
    j := next;
    while (j<=N) do (* eliminate of multiple of "next" *)
      begin
        sieve := sieve - [j];
        j := j + next ;
      end;
  until (sieve = []);
 (* writing primes *)
 writeln(' Primes between 0..',N);
 for j:=1 to N do
  begin
  if ( j in primes ) then
      write(j:4)
  else
     write(' ---');
  if ( j mod 15 = 0 ) then
    writeln;
 end;
End.
```

نتيجة تنفيذ البرنامج: مصفوفة الأعداد الأولية 1..255:

```
Primes between 0..255
---   2   3 ---   5 ---   7 --- --- ---  11 ---  13 --- ---
---  17 ---  19 --- --- ---  23 --- --- --- --- ---  29 ---
 31 --- --- --- --- ---  37 --- --- ---  41 ---  43 --- ---
---  47 --- --- --- --- ---  53 --- --- --- --- ---  59 ---
 61 --- --- --- --- ---  67 --- --- ---  71 ---  73 --- ---
--- --- ---  79 --- --- ---  83 --- --- --- --- ---  89 ---
--- --- --- --- --- ---  97 --- --- --- 101 --- 103 --- ---
--- 107 --- 109 --- --- --- 113 --- --- --- --- --- --- ---
--- --- --- --- --- --- 127 --- --- --- 131 --- --- --- ---
--- 137 --- 139 --- --- --- --- --- --- --- --- --- 149 ---
151 --- --- --- --- --- 157 --- --- --- --- --- 163 --- ---
--- 167 --- --- --- --- --- 173 --- --- --- --- --- 179 ---
181 --- --- --- --- --- --- --- --- --- 191 --- 193 --- ---
--- 197 --- 199 --- --- --- --- --- --- --- --- --- --- ---
211 --- --- --- --- --- --- --- --- --- --- --- 223 --- ---
--- 227 --- 229 --- --- --- 233 --- --- --- --- --- 239 ---
241 --- --- --- --- --- --- --- --- --- 251 --- --- --- ---
```

الفصل الثاني
بنية المعطيات: تسجيلة Record

مقدمة:

درسنا إمكان تجميع مجموعة من التعليمات ككتلة متكاملة في بنية البرنامج الجزئي. كذلك تسمح معظم لغات البرمجة بتجميع المعطيات ككتلة يمكن التعامل معها ككيان واحد وذلك عن طريق بنية التسجيلة أو **record**. ولكي ندرك أهمية احتواء لغة البرمجة على بنية تسمح بتجميع المعطيات، نورد المثال التالي:

ليكن لدينا مجموعة من المعلومات الضرورية لتوصيف مسألة تنظيم إداري، مثلاً وصف مجموعة العاملين بأرقامهم الذاتية وأسمائهم ورواتبهم. نحتاج إذن، لكل عامل، إلى رقم ذاتي EmpNb و اسم EmpName و راتب EmpSalary. ونحتاج بالطبع إلى ثلاثة جداول لخزن المعلومات الخاصة بمجموعة العاملين.

لنفترض أننا نريد إصدار جدول مرتب بمعلومات العاملين، وليكن الترتيب تبعاً للاسم مثلاً. عندها نحتاج في خوارزمية الفرز إلى وضع التعليمات اللازمة لإجراء تبديل المواقع في الجداول الثلاثة معاً، بمعنى أننا خلال أي معالجة يجب أن ننتبه إلى أن المعلومات في الجداول الثلاثة مترابطة وينبغي المحافظة على هذا الترابط.

أما إذا عرّفنا مجموعة المعلومات الخاصة بالعامل كبنية معطيات مُركّبة (من اسم و رقم و راتب) هي بنية التسجيلة، فيصير بإمكاننا الحفاظ على ترابط المعلومات الخاصة بكل

عامل، إضافةً إلى إمكان التعامل مع جميع هذه المعلومات ككتلة من حيث الإسناد مثلاً.

قبل البدء بالتعريف الكامل لبنية التسجيلة، سنكتب حل مثال ترتيب قائمة معلومات العاملين ببرنامجين: الأول بلا بنية تسجيلة والثاني باستخدام بنية التسجيلة. لنلاحظ الفارق بين الأسلوبين خصوصاً إذا تخيلنا أن مجموعة المعلومات قد تكون أكثر من ثلاث معلومات كما في مثالنا.

يكفي الآن لفهم المثال بلغة pascal أن نتخيل بنية التسجيلة كاستمارة لها أسماء حقول ينبغي ملؤها، وأن المتحول الفعلي من نمط «تسجيلة» يحتوي معلومات مَلء الاستمارة.

في المثال بلا تسجيلة نحتاج إلى ثلاثة جداول (عدد أنواع المعلومات اللازمة) أما في حالة الحل بوجود بنية التسجيلة، نقوم بتعريف بنية تسجيلة العامل ولنسمها Employee بثلاثة حقول : رقم ذاتي EmpNb و اسم EmpName و راتب EmpSalary .

يجري التعريف بالكلمة **record** ، ثم نستخدم كمتحول لخزن مجموع العاملين جدولاً وحيداً عناصره من نمط التسجيلة السابقة:

```
TabEmp : array[1..MaxN] of Employee;
```

ومن ثَم TabEmp[i] هي تسجيلة (التسجيلة رقم i في الجدول السابق) . يُشار إلى حقل ضمن التسجيلة بوضع نقطة ثم اسم الحقل: مثلاً TabEmp[i].EmpName هو حقل الاسم للعنصر i في جدول التسجيلات.

إن خوارزمية الفرز المستخدمة هي خوارزمية الفرز بالتعويم المستخدمة مراراً في كتاب

البرمجة (1)، وهي تعتمد فكرة تعويم العنصر الأكبر (أو الأصغر) بالمرور على عناصر الجدول تباعاً، وتبديل مواقع العنصرين المتجاورين إذا كان ترتيبهما مخالفاً للترتيب المنشود.

برنامج ترتيب جدول معلومات العاملين دون استخدام بنية التسجيلة:

```
Program TableEmp( input, output );
(* simulting record structure by multiple tables *)

Const  MaxN=100;
Type   String15 =String[15]; (* array[1..15] of char; in standard *)

Var
    EmpNb     : array[1..MaxN] of integer;
    EmpName   : array[1..MaxN] of String15;
    EmpSalary : array[1..MaxN] of real;
    tNb   : integer;
    tName : String15;
    tSalary:real;
    i, j, N : integer;

Begin
 (* data reading *)
 writeln(' Input N <= ',MaxN);
 readln( N );
 writeln(' Input ',N,' data to sort');
 for i:=1 to N do
    readln( EmpNb[i], EmpSalary[i], EmpName[i] );
 (* data sorting *)
 for i:=1 to N-1 do  (* buble element *)
  for j:=1 to N-i do
```

```
  if ( EmpName[j] > EmpName[j+1] ) then   (* change positions *)
    begin
     tName      := EmpName[j];
     EmpName[j]  := EmpName[j+1];
     EmpName[j+1] := tName;
     tNb        := EmpNb[j];
     EmpNb[j]   := EmpNb[j+1];
     EmpNb[j+1] := tNb;
     tSalary    := EmpSalary[j];
     EmpSalary[j]  := EmpSalary[j+1];
     EmpSalary[j+1] := tSalary;
    end;
  (* data writing *)
  writeln(' Sorted Elements ');
  for i:=1 to N do
      writeln( EmpNb[i]:4, EmpSalary[i]:8:2, EmpName[i]);
End.
```

برنامج ترتيب جدول معلومات العاملين (العمل لمطلوب كما في البرنامج السابق)، ولكن مع استخدام بنية التسجيلة:

```
Program RecordExample( input, output );
(* using record in bublesort algorithms *)

Const  MaxN=100;
Type   String15 =String[15]; (* array[1..15] of char; in standard *)
Type   Employee = record
            EmpNb    : integer;
            EmpName  : String15;
            EmpSalary : real;
           end;

Var
    TabEmp     : array[1..MaxN] of Employee;
    tEmp   : Employee;
    i, j, N : integer;
```

```
Begin
 (* data reading *)
 writeln(' Input N <= ',MaxN);
 readln( N );
 writeln(' Input ',N,' data to sort');
 for i:=1 to N do
    readln( TabEmp[i].EmpNb, TabEmp[i].EmpSalary,
                                TabEmp[i].EmpName );
 (* data sorting *)
 for i:=1 to N-1 do  (* buble element *).
  for j:=1 to N-i do
   if ( TabEmp[j].EmpName > TabEmp[j+1].EmpName ) then
    (* change positions *)
     begin
      tEmp      := TabEmp[j];
      TabEmp[j]  := TabEmp[j+1];
      TabEmp[j+1] := tEmp;
     end;
  (* data writing *)
 writeln('  Sorted Elements ');
 for i:=1 to N do
  writeln( TabEmp[i].EmpNb:5, TabEmp[i].EmpSalary:8:2,
                                TabEmp[i].EmpName);
End.
```

إن تنفيذ كلا البرنامجين يعطي الشيء ذاته، لدينا هنا مثال على التنفيذ (الغامق هو الدخل):

```
Input N <= 100
8
Input 8 data to sort
1   6500.0  Mazen
2   4005.5  Ahmad
3   5000.0  Kassem
4   3010.0  Houssien
5   4200.0  Yazan
6   2900.0  Bilal
```

7 5400.0 Ryad
8 8000.0 Ihssan

Sorted Elements
2 4005.50 Ahmad
6 2900.00 Bilal
4 3010.00 Houssien
8 8000.00 Ihssan
3 5000.00 Kassem
1 6500.00 Mazen
7 5400.00 Ryad
5 4200.00 Yazan

لاحظ إصدار جدول المعلومات مرتباً وفق تسلسل الأسماء الأبجدي.

نمط المعطيات: تسجيلة Record

تسمح لنا بنية المعطيات «تسجيلة» **record** بتجميع عدد من عناصر المعطيات الواصفة لغرض ما في وحدة منطقية واحدة، وهذا ما يسمح بالتعامل معها ككتلة. نسمي عناصر هذه الكتلة بالحقول، وهذه الحقول ليست بالضرورة من النمط نفسه.

يأخذ التصريح عن متحول <rec_var> من نمط تسجيلة **record** الشكل التالي:

```
Var <rec_var> : record
  <field1> : <type1>;
  <field2> : <type2>;
    ....
    ....
  <fieldn> : <typen>;
end;
```

وعلى هذا، يجري التصريح بالكلمة المحفوظة للغة: **record** ولاننسى إنهاء التصريح بكلمة

end; . وبين الكلمتين record و end نضع أسماء الحقول <field1>,...<fieldn> وأنماطها <type1>,...,<typen> بنفس القواعد التي نستخدمها في التصريح عن المتحولات. إن تسمية الحقول هي اختيارية، كما هو اختيار أسماء التعريف بوجهٍ عام، أما أنماط الحقول فيمكن أن تكون أي نمط مُسبَق التعريف في اللغة، أو أي نمط مُعرَّف من قِبَل المُبرمج (مثلاً يمكن أن يكون الحقل بدوره تسجيلة أيضاً).

لاحظ عدم استخدام begin رغم استخدام end في إنهاء التعريف (يذكرنا هذا بحالة قواعدية مشابهة: تعليمة Case هي الأخرى تُنهي باستخدام end ولاتحتاج لكلمة begin)

يُفضَّل عادةً تعريف نمط تسجيلة بواسطة Type ثم استخدام اسم النمط المُعرَّف عند التصريح عن المتحولات.

يُشار إلى حقل ضمن المتحول من نمط تسجيلة بوضع اسم المتحول ثم نقطة وبعدها اسم الحقل. مثال: لإجراء عملية على الحقل <field2> نكتب: <rec_var>.<field2> .

أمثلة:

مثال1:

يتألف التاريخ date من ثلاث معلومات: يوم وشهر وسنة day, month, year . لذا يمكن تعريف نمط معطيات تسجيلة تاريخ بالشكل:

```
Type  date = record
    day, month, year : integer;
```

```
end;
```

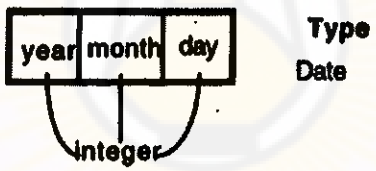
وكنا ذكرنا أن قواعد تعريف الحقول هي قواعد التصريح عن المتحولات نفسها، وعلى هذا فالكتابة التالية مكافئة تماما" للسابق:

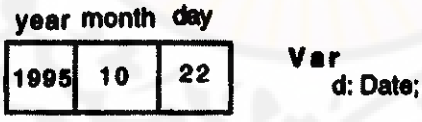
بتغيير ترتيب الحقول:

```
Type  date = record
   year,  day, month : integer;
 end;
```

أو الكتابة بإعلان نمط كل حقل على حدة:

```
Type  date = record
   year: integer;
   day: integer;
   month : integer;
end;
```





فإذا أردنا استخدام متحول d من نمط التسجيلة date نكتب: d : date;

لوضع التاريخ: 22/10/1995 في المتحول d:

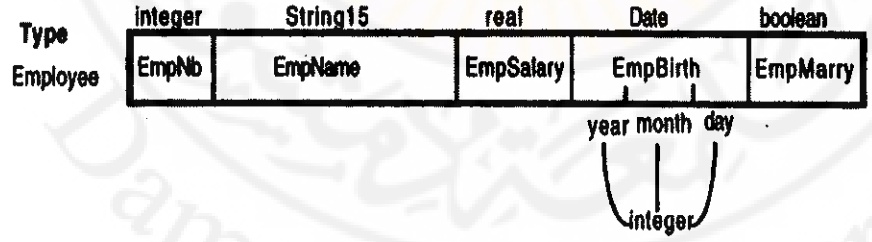
نسند قيمةً إلى كل حقل (بالإشارة إلى الحقل بواسطة النقطة مسبوقة باسم المتحول):

```
d.day := 22;   d.month := 10;  d.year:= 1995;
```

مثال2:

سنعمم المعلومات الخاصة بالعاملين التي ذكرناها في البرنامج التمهيدي وكانت: الاسم، الرقم، الراتب أو EmpName, EmpNb, EmoSalary لنضيف إليها حقلين آخرين هما تاريخ الميلاد والحالة العائلية متزوج أم لا أو EmpBirh, EmpMarry.

```
Type    Employee = record
            EmpNb    : integer;
            EmpName  : String15;
            EmpSalary : real;
            EmpBirth : date;
            EmpMarry : boolean;
          end;
```



لاحظ أننا جمعنا هنا أنماطاً مختلفة في توصيفنا لحقول التسجيلة، من بينها حقل من نمط

تسجيلة من نوع آخر هو نمط تسجيلة التاريخ date الذي عرفناه في المثال1.

ليكن لدينا الآن المتحول E من نمط Employee ونريد ملء (إسناد) قيمة تاريخ ميلاد في هذا المتحول:

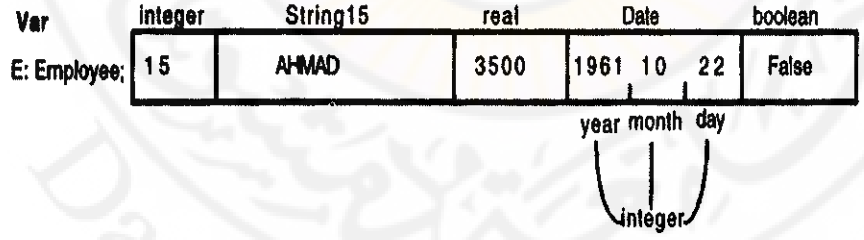
```
Var  E: Employee;
```

للوصول إلى حقل تاريخ الميلاد نكتب :

E.EmpBirth وهي بدورها من نمط تسجيلة، ولكي نصل إلى حقولها لابد من نقطة أخرى.

لدا نكتب :

```
E.EmpBirth.day  := 30 ;
E.EmpBirth.month  := 12 ;
E.EmpBirth.year := 1961 ;
```



اختصار تكرار اسم الحقل بالكلمة with

نلاحظ ضرورة تكرار اسم المتحول من نمط التسجيلة قبل ذكر الحقل. لكن يمكن اختصار هذا التكرار باستخدام الكلمة المحفوظة للغة **with** متبوعةً باسم المتحول من نمط تسجيلة.

```
with   <rec_var>  do
begin
  <field1> := ...;
  <field2> := ...;
    ......
  <fieldn> := ...;
end;
```

with هنا تعني: استخدم اسم المتحول التسجيلة <rec_var> كسابقة لأسماء الحقول إن ذكرت.

يمكن أن يستغنى عن **begin** و **end** إذا أردنا قصر مجموعة التعليمات المتاثرة بـ **with** على تعليمة واحدة.

أمثلة:

في البرنامج التمهيدي كتبنا إحدى التعليمات لقراءة حقول تسجيلة (عنصر جدول التسجيلات TabEmp[i]) بالشكل:

```
readln( TabEmp[i].EmpNb, TabEmp[i].EmpSalary,
        TabEmp[i].EmpName );
```

يمكن كتابة كلمة **with** والاستغناء عن تكرار اسم الحقل:

```
with   TabEmp[i] do
       readln( EmpNb, EmpSalary, EmpName );
```

لنأخذ مجموعة عمليات إسناد قيم (الحقول) إلى المتحول E من نمط تسجيلة Employee

```
(* Var  E: Employee; *)

with  E do
begin
EmpName := 'Ahmad
EmpNb := 15;
EmpBirth.year := 1961;
end;
```

يجب ملاحظة أننا نستطيع الاكتفاء بالإسناد لحقول معينة دون غيرها.

مثال برمجي:

لا يقتصر استخدام بنية المعطيات «تسجيلة» على الاستخدامات التنظيمية "الإدارية".

في الكثير من التطبيقات العلمية يوجد بُنى مُركَّبة، نحتاج فيها التسجيلة، لتمثيل شيء موصَّف بعددٍ من العناصر "الحقول" التي تُعرِّفه. فالعدد العقدي مؤلف من مركبتين (من

عددين حقيقين)، والأعداد الكسرية من صورة ومخرج أو بسط ومقام (من عددين صحيحين) ..... الخ.

في المثال التالي، نعرّف نمط الكسور Rational ، بتسجيلة لها حقلان الصورة والمخرج.

يقوم البرنامج بقراءة عددين كسريين. يجري إدخال الكسر بالصيغة: 7 / 3 ، ثم يحسب البرنامج جداء الكسرين ويكتب النتيجة. نُجري عملية إختزال على الكسر الناتج من الجداء، لهذا استخدمنا تابع القاسم المشترك الأعظم GCD ، الذي عرفنا طريقة حسابه في كتاب البرمجة (1). لإجراء اختصار الكسر، نكرّر قسمة الصورة والمخرج على قاسمها المشترك الأعظم حتى يأخذ هذا القاسم المشترك القيمة 1 .

```
Program Fractions;
(* This program illustrates the use of records with procedures *)
(* The task is to compute the product of tow rational numbers  *)

Type
  Rational = record
          Numerator : integer;
          Denominator: integer;
         end;

Var
  X, Y, Z : Rational;

Function GCD( A, B : integer):integer;
begin
  while ( (A-B)<>0 ) do
```

```
   if A>B then
      A := A-B
   else
      B := B-A;
  GCD := A;
end; (* GCD *)

Procedure ReadRational( var r : Rational );
var slash,space: char;
begin
  writeln(' Enter a fraction in the form N / M .');
  readln(r.Numerator, space,slash,space, r.Denominator);
end;(* ReadRational *)


Procedure Product( R1, R2 : Rational; var P: Rational);
var rgcd : integer;
begin
  P.Numerator   := R1.Numerator   * R2.Numerator;
  P.Denominator := R1.Denominator * R2.Denominator;
  with P do
   repeat
    rgcd := GCD( Numerator, Denominator);
     Numerator   := Numerator   div rgcd;
    Denominator := Denominator div rgcd;
   until rgcd=1;
end (* Product *);

Procedure PrintResult( R1, R2, P : Rational);
begin
  writeln; writeln;
  writeln( R1.Numerator:13,R2.Numerator:6,P.Numerator:6);
  writeln('--- * --- = ---':25);
  writeln( R1.Denominator:13,R2.Denominator:6,P.Denominator:6);
end; (* PrintResult *)
```

```
Begin
  ReadRational( X );
  ReadRational( Y );
  Product( X, Y , Z );
  PrintResult( X, Y, Z);
End.
```

أمثلة من تنفيذ البرنامج:

```
Enter a fraction in the form N / M
3 / 7

Enter a fraction in the form N / M
14 / 15

   3     14     2
 --- * --- = ---
   7     15     5


Enter a fraction in the form N / M
35 / 99

Enter a fraction in the form N / M
33 / 77

   35     33     5
 --- * --- = ---
   99     77     33
```

Enter a fraction in the form N / M
1 / 1

Enter a fraction in the form N / M
145 / 515

```
  1    145     29
--- * --- = ---
  1    515    103
```

الفصل الثالث
الملفات النصية
Text File

مقدمة:

يمكن اعتبار الملف النصي، في معظم لغات البرمجة، تعميماً للدخل والخرج النظاميين (standard input and output) أي لوحة المفاتيح و الشاشة. فمع الملفات النصية، تُقرأ البيانات من (أو تُكتب في) وحدة الخزن المغناطيسي بالآلية نفسها (تُستخدَم التعليمات ذاتها) التي تُقرأ أو تُكتب من وفي الدخل والخرج النظاميين.

فإذا طلبنا من تعليمات الكتابة أن ترسل المعلومات المطلوب كتابتها إلى ملف نصي بدلاً من الشاشة، وهذا ماسنرى كيفية تحقيقه فيما يلي، نحصل على ملف يحوي نصاً مطابقاً لما كان سيظهر في حالة الكتابة على الشاشة.

يمكننا إذن تعريف الملف النصي تعريفاً مبدئياً بأنه خزن لتتالي أحرف (مقروءة) في ملف.

قبل البدء بالتعريف العام للعمليات على الملف النصي، سنتعرف طريقة التعامل مع الملف النصي مباشرةً من الأمثلة. وآلية العمل هذه متشابهة في معظم لغات البرمجة.

مثال:

لدينا البرنامج البسيط التالي: يقوم البرنامج بكتابة جدول من عمودين وعشرة أسطر، العمود الأول يحوي قيمة n والعمود الثاني يحوي قيمة مجموع السلسلة 1/n ....1,1/2,.

```
Program  Nb( input , output );
(* Compute the sum of the harmonic series 1/n *)
Var
   n: integer;
   S: real;
Begin

   writeln(' n      sigma 1/n ');
   writeln('------   ----------');
   S := 0;
   for n:=1 to 10 do
    begin
      S :=  S + 1./n ;
      writeln( n:4, S:12:3 );
    end;

End.
```

إن تنفيذ البرنامج يعطينا الكتابة التالية على الشاشة:

```
 n     sigma 1/n
------  ----------
  1      1.000
  2      1.500
  3      1.833
  4      2.083
  5      2.283
  6      2.450
  7      2.593
  8      2.718
  9      2.829
 10      2.929
```

لتوجيه الكتابة إلى ملف نصي، وليكن اسمه myfile.nb بدلاً من ظهور الكتابة على الشاشة، نقوم بتعديل البرنامج كما هو مبين فيما بعد. وعند التنفيذ، نرى أن ملفاً جديداً قد وُجِدَ باسم myfile.nb ، ونستطيع قراءة هذا الملف (النصي) بواسطة أي محرر نصوص text editor، ونجد أنه يحوي النص الذي كان قد ظهر على الشاشة بواسطة البرنامج السابق.

نشير إلى التعديلات التي أدخلناها على البرنامج السابق بخط مختلف، وهي تقوم على التوالي بما يلي:

- التصريح عن متحول من نمط ملف نصي ;f : text .
- ربط المتحول ملف (f) بالملف الفعلي الذي سيخلق ;('myfile.nb',f)assign .
- تأهيل الملف للكتابة ;(f)rewrite .
- تغيير كل تعليمة كتابة (.....)writeln بتوجيه الكتابة إلى ملف بواسطة (f,.....)writeln أي إضافة اسم المتحول من نوع ملف نصي f متبوعاً بفاصلة، في الموقع الذي يلي قوس الفتح الخاص بتعليمة الكتابة.
- إغلاق الملف بعد إنهاء الكتابة فيه ;(f)close .

```
Program  NbWriteText( input , output );
(* Compute the sum of the harmonic series 1/n *)
(* Write to a text file *)
Var
  n: integer;
```

```
  S: real;
   f : text;
Begin
  assign(f,'myfile.nb');
  rewrite( f );

  writeln(f, ' n      sigma 1/n ');
  writeln(f, '------  ----------');
  S := 0;
  for n:=1 to 10 do
   begin
     S := S + 1./n ;
     writeln(f , n:4, S:12:3 );
   end;

 close( f );
End.
```

القواعد العامة للتعامل مع الملفات النصية:

1-التصريح عن الملفات النصية:

يُصرَّح عن متحول من نمط ملف نصي بالنمط text :

مثال:

```
f, ft : text;
```

2-العمليات على الملف النصي:

تأهيل الملف للكتابة:

```
rewrite( f );
```

تقوم هذه الإجرائية بفتح الملف f لعمليات الكتابة. ويوضع مؤشر الملف في أول الملف.

تُمسَح محتويات الملف إن كان الملف موجود سابقاً أو يُحْدَث ملف جديد إن لم يكن الملف موجوداً.

تأهيل الملف للقراءة:

reset(f);

تقوم هذه الإجرائية بفتح الملف f ، الموجود سابقاً، لعمليات القراءة. ويوضع مؤشر الملف في أول الملف.

3-وصل الملف المنطقي بالملف الفعلي:

نسمي المتحول من نمط ملف نصي ملفاً منطقياً، لأنه يُعامل كأي بنية معطيات أخرى مزودة بعمليات مُعرّفة. لكن الفارق الأساسي هو أن استخدامنا للملف يهدف إلى التعامل مع الخزن الدائم (المغناطيسي) والتعامل مع ملفات فعلية.

لذلك ، كي تأخذ العمليات على المتحول ملف تحقيقاً على ملف فعلي، لابد من ربط الملف الفعلي (اسم الملف على وحدة الخزن الدائم) بالمتحول من نمط ملف.

ويجب أن تجري عملية الربط قبل تنفيذ أي عملية على الملف.

لا يُعرِّف Pascal النظامي طريقة الربط ويتركها للمترجمات.

في Turbo_Pascal يجري الربط بالشكل:

assign(f, <filename>);

حيث f هو المتحول «ملف نصي» أي f : text;

<filename> هو سلسلة حرفية (ثابت أو متحول) تحوي اسم الملف الفعلي على وحدة

الخزن الدائمة، وبالطبع تخضع التسمية لقواعد تسمية الملفات على نظام الاستثمار المستخدم.

مثال:

assign(f , 'myfile.out');

تقوم بالوصل بملف myfile.out في المجلد الحالي.

assign(f , 'C:\mydir\myfile.out');

تقوم بالوصل بملف myfile.out في المجلد mydir على القرص الصلب :C.

* ملاحظة: يقوم بعض المترجمات بالربط الفعلي عند التأهيل: ;(reset(f,<filename> و ;rewrite(f, <filename>).

في Pascal-Wirth نقوم بذكر الملفات كأسماء متحولات ضمن ترويسة البرنامج. مثال : program P(f, input, output); حيث f:text; وعند تنفيذ البرنامج وليكن اسم ملف البرنامج التنفيذي P.EXE نستدعيه بالشكل: P.EXE <filename> فيحدث الوصل بين الملف المنطقي f والملف الفعلي <filename> على وحدة الخزن.

4-تعليمات الكتابة:

تُطبَّق تعليمات الكتابة نفسها أي write و writeln على الملف النصي وذلك بكتابة اسم المتحول من النمط text بعد قوس الكتابة.

أمثلة:

كتابة عبارة حرفية:

writeln(f , ' n sigma 1/n ');

أوكتابة تعابير حسابية:

writeln(f , n:4, S:12:3);

أو كتابة مع البقاء على نفس السطر:

write(f , n);

أو كتابة سطر والانتقال إلى سطرجديد :

writeln(f);

5-تعليمات القراءة:

كل ماذكرناه فيما يتعلق بالكتابة يمكن ذكره للقراءة باستخدام read و readln.

6-التوابع:

لدينا تابعان منطقيان معرفان على الملف النصي:

.var f:text;

eoln(f)

للاستعلام عن الوصول إلى نهاية السطر في ملف القراءة:

eoln(f) التابع المنطقي الدال على نهاية السطر(end of line)

يعطي القيمة المنطقية "صح" إذا وصلنا إلى نهاية السطر في حالة القراءة (يكافىء ذلك

في حالة الدخل النظامي الضغط على مفتاح نهاية سطر).

مثال:

حساب عدد الأحرف في سطر:

يمكن تحقيق ذلك بالخوارزمية التالية: مادام السطر غير منته كرّر: اقرأ حرفاً وزد العداد واحداً. وعند الوصول إلى نهاية السطر اقرأ من السطر التالي.

```
nc := 0;
while ( not eoln( f ) ) do
    begin
        read(f, c);
        nc := nc + 1;
    end;
readln(f);
```

eof(f)

للاستعلام عن الوصول إلى نهاية ملف القراءة:

eof(f) هوالتابع المنطقي الدال على نهاية الملف(end of file)

يعطي القيمة المنطقية "صح" إذا وصلنا إلى نهاية الملف في حالة القراءة .

مثال:

حساب عدد الأسطر في ملف:

يمكن تحقيق ذلك بالخوارزمية التالية: مادام الملف غير منته اقرأ سطراً (انتقل إلى السطر التالي)، وزد العداد واحداً.

```
nl := 0;
```

```
while ( not eof( f ) ) do
    begin
        readln(f);
        nl := nl + 1;
    end;
```

7 – إغلاق الملف:

```
close( f );
```

تقوم هذه الإجرائية بإغلاق الملف، وهي ليست من الوظائف المُعرَّفة في لغة Pascal النظامية ولكنها موجودة في أكثر المترجمات، وربما تكون ضرورية في بعضها مثل Turbo-Pascal الذي يفرض إغلاق الملف في حالة الانتهاء من الكتابة فيه، لكي يحفظ آخر ما كتب في الملف على وحدة الخزن الدائم.

ملاحظات هامة:

الملف يفتح إما للقراءة أوللكتابة:

عند التعامل مع الملفات التسلسلية (الملف النصي هو حالة خاصة منها)، لابد من الانتباه إلى أن الملف يُفتح للقراءة فقط بالإجرائية reset أويفتح للكتابة فقط بالإجرائية rewrite، أي إن العمليات على الملف إما قراءة فقط أو كتابة فقط. فإذا أردنا قراءة ملف كنا قد فتحناه للكتابة فلابد من إعادة فتحه للقراءة قبل الشروع بعمليات القراءة.

الدخل والخرج النظامين والملفات النصية في نظم الاستثمار:

تسمح معظم نظم الاستثمار (DOS, UNIX مثلاً) بالتعامل مع الملفات النصية كتعميم للدخل والخرج النظاميين وذلك دون الحاجة إلى أي تغيير في نص البرنامج وذلك ما يسمى عادة بتغيير توجيه الدخل أو الخرج (Input/Output redirection).

ليكن لدينا برنامج تنفيذي (امتداد اسم ملفه في DOS هو : .com, .exe)، وليكن اسم ملف البرنامج التنفيذي هو prog.exe .

لتوجيه كتابة نتائج البرنامج (كل إظهارات write و writeln) إلى ملف نصي (ولنسمه PROG.OUT) بدلاً من الإظهار على الشاشة، يكفي أن نكتب على سطر الأوامر:

PROG > PROG.OUT

أي إننا غيرنا توجيه خرج البرنامج إلى الملف PROG.OUT عوضاً عن الإظهار على الشاشة، بمجرد وضعنا إشارة > متبوعةً باسم ملف الخرج الذي نريد.

والآن لنقم بالدخل من ملف عوضاً عن الدخل النظامي أو لوحة المفاتيح:

ليكن لدينا برنامج تنفيذي اسمه كملف تنفيذي progi.exe :

لتوجيه أخذ (قراءة) معطيات البرنامج (كل معطيات الدخل المطلوبة بـ read و readln) من ملف نصي (ولنسمه PROGI.IN) بدلاً من أخذ المعطيات من لوحة المفاتيح، يكفي أن نكتب على سطر الأوامر:

PROGI < PROGI.IN

أي إننا غيرنا توجيه دخل البرنامج ليكون من الملف PROG.IN عوضاً عن الإدخال من لوحة المفاتيح، بمجرد وضعنا إشارة < متبوعةً باسم ملف الدخل الذي نريد.

كما يمكن أن نتعامل بالدخل والخرج مع ملفات في وقت واحد:

PROG < PROG.IN > PROG.OUT

يمكننا إذن الاستغناء عن برمجة الملفات النصية بلغة Pascal في حالة القراءة كاملةً من ملف، أوالكتابة كاملةً إلى ملف . ولكن في الكثير من التطبيقات قد نحتاج إلى أن تظهر بعض الكتابات على الشاشة، مثل العبارات الدالة على مايجب إدخاله كمعطيات، وفي الوقت نفسه نريد كتابة بعض النتائج على الملف النصي. في هذه الحالة لابد من استخدام الملفات النصية text داخل البرنامج.

الملف هو بالضرورة معامل دخل وخرج « var » في أي برنامج جزئي :

عندما نعرِّف برنامجاً جزئياً أحد معاملاته من نمط text ، يجب أن يكون هذا المعامل وسيط دخل و خرج أي أنه يُسبق بكلمة **var** .

مثال:

برنامج يقرأ الأحرف الخمس الأولى من السطر الثالث في ملف نصي.

```
Program SubProgFile;
Var f: text;
    first5= array[1..5] of char;
    i, k : integer;

Procedure SkipLine( var f: text );
begin
  if not eof(f) then
    readln(f);
end; (* SkipLine *)

Begin
```

```
 (*  assign(f,'mytext.txt');  *)
reset(f);
SkipLine( f );
SkipLine( f );
i :=0;
while ( not eof(f)  and i<=5) do
   begin
      i := i + 1;
      read( f , first5[ i ] );
   end;
for  k:=1  to i do
      write( first5[ i ] );
End.
```

مثال على قراءة ملف:

نريد قراءة الملف النصي الذي سبق وكتبنا فيه الأعداد في المثال الأول.

لاحظ في البرنامج أن تعليمتي readln(f); readln(f) مهمتهما تجاوز قراءة السطر الأول والثاني في الملف حيث كنا قد كتبنا ترويسة الجدول. وبعدها، للقيام بقراءة الأعداد المخزونة في الملف، نكرِّر قراءة عددين من الملف readln(f,n,S); وإظهارهما على الشاشة . يجري التكرار مادمنا لم نصل إلى نهاية الملف:

. while (not eof(f)) do

يمكن أن نلاحظ أيضاً أننا اكتفينا بمعرفتنا نسق الملف، -وهو تتالي عددين في كل سطر (الأول صحيح والثاني كسري)- لكي نقرأ كامل الملف. ولم يكن ثمة حاجة لمعرفة أن عدد

الأسطر هو عشرة، وجرى ذلك باعتمادنا على تكرار القراءة حتى آخر الملف .while (not eof(f)) do

```
Program  NbReadText( input , output );
(* read from a text file *)
Var
   n: integer;
   S: real;
   f : text;
Begin
   assign(f,'myfile.nb');
   reset(f);

   readln(f);
   readln(f);
   while ( not eof(f) ) do
     begin
        readln(f,n,S);
        writeln(n:4,S:12:3);
    end;

  close(f);
End
```

إن تنفيذ البرنامج يعطينا الكتابة التالية على الشاشة:

```
   1       1.000
   2       1.500
   3       1.833
   4       2.083
   5       2.283
   6       2.450
   7       2.593
   8       2.718
   9       2.829
  10       2.929
```

الفصل الرابع
الملف الثنائي
نمط المعطيات file

مقدمة:

عندما درسنا الملفات النصية text تعرّفنا بعضَ التقنيات المتاحة في لغات البرمجة، والتي تسمح بالتعامل مع الخزن الدائم أوالملفات. ولكن الملفات النصية تقتصر على خزن المعلومات بشكل نصي (مجموعة أحرف مقروءة) وعلى هذا يمكن قراءة الملف باستخدام أي محرر للنصوص editor. هذه ليست بالضرورة الطريقة المثلى في الخزن للعديد من التطبيقات كما سنرى، إذ إن بعض التطبيقات تريد خزن المعلومات على وحدة الخزن الدائم واسترجاعها فيما بعد بفعالية دون الاهتمام بكون طريقة الخزن نصية. نسمي هذه الملفات غير النصية بالملفات الثنائية binary files . سنتعرف في هذا الفصل طريقة تعامل لغة Pascal مع هذا النوع من الملفات.

نمط المعطيات ملف file :

يُعرَّف نمط المعطيات ملف file سلسلة من عناصر لها نفس النوع (النمط).

يُصرَّح عن متحول <vf> من نمط ملف عناصره نمطها <type> بالشكل:

```
<vf> : file of <type>;
```

يمكن أن تكون <type> ، أيُّ نمط عناصر الملف، من أيِّ نمط (عدا نمط ملف: ملف من ملفات هو شيء غير وارد).

أمثلة:

- ملف من أعداد حقيقية:

```
fr : file of real;
```

- ملف من تسجيلات:

```
frec : file of record
          name : array[1..20] of char;
            nb :  integer;
       end;
```

في لغة Pascal النظامية، يُشار إلى العنصر الحالي أو مكان توضعنا في الملف (ضمن سلسلة عناصرالملف) باسم المتحول من نمط ملف متبوعاً بالرمز ^.

مثال: fr^ هو العنصر الحالي من الملف fr المعرَّف آنفاً، ومن ثَمّ هو عنصر من نمط حقيقي يُعامل كما يُعامل أي متحول من نمط حقيقي. مثلاً: ليكن r : real; عندها يمكن أن نكتب:

r := fr^; وتعني إسناد قيمة المتحول r مكان قيمة العنصر الحالي في الملف fr^.

أو

fr^ := r; وتعني إسناد قيمة العنصر الحالي في الملف fr^ إلى المتحول r.

هناك إذن مؤشر ضمني يعطينا العنصر الحالي في الملف ونعبر عنه بالرمز ^.

ملاحظة هامة:

الفارق بين بنيتي المعطيات array و file.

إن بنية المعطيات «ملف file»، بالتعريف السابق كمجموعة من عناصر لها النمط نفسه، يشبه للوهلة الأولى بنية المعطيات «مصفوفة array » فهي أيضاً تجميع لعناصر من النمط نفسه.

لكن الخلافات الجوهرية بين الملف والمصفوفة (كما سنتبينه عند تفصيل عمليات الملف) هي:

1- الوصول إلى عنصر معين:

في المصفوفات نصل إلى أي عنصر بواسطة دليل العنصر (ترتيبه) مثلاً A[5] تعطينا العنصر الخامس ضمن المصفوفة A. على حين الملفات في Pascal النظامية هي تسلسلية فقط، وهذا يعني أن الوصول إلى أي عنصر في الملف يقتضي بالضرورة المرور (بالتسلسل) على جميع العناصر السابقة له. ومن ثَم للمصفوفات أفضلية بطريقة الوصول المباشر إلى العناصر.

2- الحجم :

عند التصريح عن المصفوفة نحدِّد مباشرةً الحجم الأعظم، في حين لايُحدَّد حجم الملف ، ويمكن إضافة أي عدد من العناصر. والحجم الأعظم الذي يمكن أن يبلغه الملف هو كبير جداً مقارنة بالحجوم التي يمكن تعريفها للمصفوفة، لأن الملف هو بنية موجهة للخزن الدائم أو الذاكرة الثانوية، أما المصفوفة فتخزن في الذاكرة الرئيسية، وهذه بالطبع صغيرة الحجم مقارنةً بالذاكرة الثانوية. إذن للملفات أفضلية في عدد العناصر الممكن تخزينها.

العمليات على الملف file:

سنرى فيما يلي بعض التشابه بين العمليات المعرّفة هنا، وتلك التي عرفناها على الملف النصي.

تأهيل الملف للقراءة:

reset(f);

تقوم هذه الإجرائية بفتح الملف f (الموجود سابقاً) لعمليات القراءة، ويوضع مؤشر الملف في أول الملف.

تأهيل الملف للكتابة:

rewrite(f);

تقوم هذه الإجرائية بفتح الملف f لعمليات الكتابة، ويوضع مؤشر الملف في أول الملف. تمسح محتويات الملف إن كان الملف موجود سابقاً أو يُنشأ ملف جديد إن لم يكن الملف موجوداً.

وصل الملف المنطقي بالملف الفعلي:

تخضع هذه العملية للقواعد ذاتها التي ذكرناها بخصوص الملف النصي، ولا ضير في إعادة التذكير بها هنا. نسمي المتحول من نمط ملف ملفاً منطقياً، لأنه يُعامَل كأي بنية معطيات أخرى مزودة بعمليات معرّفة. لكن الفارق الأساسي هو أن استخدامنا للملف يهدف إلى التعامل مع الخزن الدائم (المغناطيسي) والتعامل مع ملفات فعلية.

لذلك، ولكي تأخذ العمليات على المتحول ملف تحقيقاً على ملف فعلي، لابد من ربط الملف الفعلي (اسم الملف على وحدة الخزن الدائم) مع المتحول من نمط ملف.

ويجب أن تجري عملية الربط قبل تنفيذ أي عملية على الملف، حتى قبل reset أو rewrite.

لا يُعرِّف Pascal النظامي طريقة الربط ويتركها للمترجمات.

في Turbo_Pascal يجري الربط بالشكل:

assign(fb, <filename>);

حيث fb هو المتحول من نمط ملف file مثلاً ; fb : file of real

<filename> هو سلسلة حرفية (ثابت أو متحول) تحوي اسم الملف الفعلي على وحدة الخزن الدائمة، وبالطبع تخضع التسمية لقواعد تسمية الملفات على نظام الاستثمار المستخدم.

مثال:

assign(fb, 'MYFILE.BIN');

تقوم بالوصل بملف MYFILE.BIN في المجلد الحالي.

assign(fb , 'C:\MYDIR\MYFILE.BIN');

تقوم بالوصل بملف myfile.bin في المجلد MYDIR على القرص الصلب :C.

بعد تنفيذ عملية الوصل أو الربط بين المتحول من نمط ملف و الملف الفعلي، تجري العمليات بالإشارة إلى المتحول فقط (f b هنا)، وتُنجَز هذه العمليات على الملف الفعلي المربوط بذلك المتحول.

تعليمات الكتابة:

ليكن لدينا المتحول ملف fb المعرّف كما يلي:

fb : file of <type>;

والمتحول V من نمط عناصر الملف، أي:

V : <type>;

كتابة محتوى المتحول V على الملف fb يمكن أن تجري بأحد الأسلوبين التاليين:

write(fb, V);

أو بالإشارة إلى العنصر الحالي:

fb^ := V;
put(fb);

وفي كلا الأسلوبين تجري الكتابة على الملف ويتقدم مؤشر الملف إلى بداية عنصر جديد .

تعليمات القراءة:

قراءة محتوى العنصر الحالي في الملف fb للمتحول V والانتقال إلى العنصر التالي يمكن أن تجري بأحد الأسلوبين التاليين:

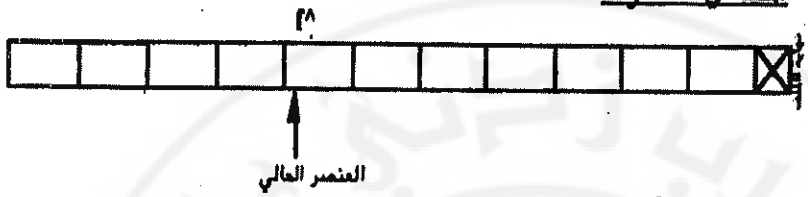
read(fb, V);

أو بالإشارة إلى العنصر الحالي:

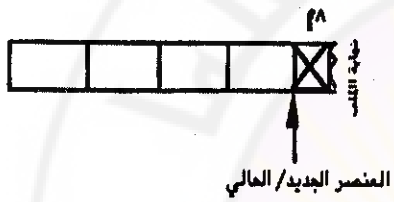
V := fb^;
get(fb);

f : file of <type>;

الملف في حالة قراءة



الملف في حالة كتابة



ملاحظة هامة:

برغم تعريف مفهوم مؤشر الملف، والعنصر الحالي للملف بواسطة (^) في Pascal النظامية، ووجود الإجرائيات Put و Get ، فإن مترجمات turbo-pascal لا تحققهما، وتقتصر العمليات فيهما على الإجرائيتين read و write، وهما كافيتان لمعظم التطبيقات.

التابع المنطقي نهاية ملف:

eof(f)

التابع المنطقي الدال على نهاية الملف(end of file)

يعطي القيمة المنطقية "صح" إذا وصلنا إلى نهاية الملف في حالة القراءة .

إغلاق الملف:

close(f);

تقوم هذه العملية بإغلاق الملف، وهي ليست من الوظائف المعرَّفة في لغة Pascal النظامية ولكنها موجودة في أكثر المترجمات، وربما تكون ضرورية في بعضها مثل Turbo-Pascal الذي يفرض إغلاق الملف في حالة الانتهاء من الكتابة فيه، لكي يحفظ آخر ما كُتبَ في الملف على وحدة الخزن الدائم.

الملف يفتح إما للقراءة أوللكتابة:

عند التعامل مع الملفات التسلسلية (الملفات في Pascal النظامية هي ملفات تسلسلية)، لابد من الانتباه إلى أن الملف يفتح للقراءة فقط بالإجرائية reset أويفتح للكتابة فقط بالإجرائيةrewrite. أي إن العمليات على الملف إما قراءة فقط أو كتابة فقط. فإذا أردنا قراءة ملف كنا قد فتحناه للكتابة لابد من إعادة فتحه للقراءة قبل الشروع بعمليات القراءة.

الملف هو بالضرورة معامل دخل وخرج « var » في أي برنامج جزئي :

عندما نعرف برنامجاً جزئياً أحد معاملاته من نمط file، يجب أن يكون هذا المعامل وسيط دخل و خرج أي أنه يُسبق بكلمة var .

مثال:

الفوارق بين الملف النصي والملف الثنائي.

يفضّل البعض عدم الفصل بين الملف النصي text والملف بتعريفه العام **file of**، إذ يُمكن مبدئياً أن نقول أن الملف النصي text هو نمط مُسبق التعريف: **Type text = file of char;**

إن سهولة شرح الملفات النصية كتعميم للدخل والخرج النظامين، تجعلنا نؤيد فكرة شرحهما منفصلين.

وبالرغم من وجود الكثيرمن العمليات المشتركة على الملفات بنوعيها النصي والثنائي (reset, rewrite, eof, close)، فإن هناك فارقاً أساسياً في طريقة خزن المعلومات، وسنرى هذا من خلال المثال التالي:

لدينا برنامج بسيط يقوم بحساب الأعداد المساوية مجموعَ السلسلة: 1/k...1/3,1/2,1.

حيث k تتغير في المجال 1..200.

نريد خزن هذه الأعداد على ملف (الخزن الدائم). سنقوم بتخزين هذه المعلومات بشكلين:

1- البرنامج يقوم بالخزن ضمن ملف ثنائي نُسمِه بـ nb.bin.

2- البرنامج يقوم بالخزن ضمن ملف نصي نُسمِه بـ nb.txt.

إن البرنامجين متشابهان جداً، ويمكن حصر الفوارق بما يلي:

- في حالة الملف النصي نصرح عن المتحول الملف بـ text.
- في حالة الملف الثنائي نصرح عن المتحول الملف بـ .... **file of**.

- في حالة الملف النصي يمكن أن نستخدم writeln لكتابة كل عدد على سطر.

- في حالة كتابة الملف الثنائي **يجب أن نستخدم write فقط** فالملف هنا عبارة عن تتال لأعداد حقيقية، مخزونة بنفس الطريقة الممثلة بها بالحاسوب، وليس هناك أي دلالة لفكرة نهاية سطر في حالة الملفات الثنائية.

الفارق بين الملفين الناتجين من التنفيذ (الملف الثنائي nb.bin والملف النص nb.txt):

أولاً:

الملف النصي: يمكن قراءته بواسطة أي منقح نصوص text editor وهو من الشكل:

1.0000000000E00
1.5000000000E00
.....
.....

الملف الثنائي: لايمكن قراءته بواسطة أي محرر نصوص (يمكن أن تظهر على الشاشة أشكال غريبة عشوائية).

ثانياً:

الملف النصي: يأخذ حجم تخزين أكبر من حجم التخزين اللازم لتخزين الملف الثنائي. وعلى سبيل المثال، هنا: الملف النصي بحجم 4000 byte بينما الملف الثنائي 800 byte.

وهذا يمكن تفسيره ببساطة: كتابة عدد بالشكل النصي بالتنسيق المذكور آنفاً يأخذ نحو 20 حرفاً (18 حرفاً كتابة أعداد بالتنسيق الأسي، إضافةً إلى نهاية السطر التي تحسب

حرفاً أو حرفين في بعض النظم)، في حين أن تمثيل العدد الحقيقي بشكل ثنائي (يختلف تبعاً لمترجم اللغة) يمكن أن يقتصر على 4 bytes فقط.

إن الفوارق بين الملف الثنائي والملف النصي تستند إلى فكرة تمثيل البيانات. ففي حين يعتمد الملف النصي على التعبير عن البيانات بنص (تتالي أحرف)، نرى أن **الملف الثنائي هو صورة "ثنائية" (تتالي خانات 0 و 1) لتمثيل البيانات في الحاسوب.** فإذا تذكرنا مثلاً تمثيل الأعداد الصحيحة في الحاسوب، وليكن التمثيل على 16 bits، فعندها يُخزن المتحول الصحيح n (var n:integer) في الذاكرة على 16 bits، وما تعليمة كتابة هذا المتحول على ملف ثنائي fi (var fi: file of integer) بالشكل write(fi,n) إلا مجرد نسخ لهذه الخانات الثنائية (16 bits) الحاوية للمتحول كما هي ونقلها إلى الملف.

برنامج الكتابة على ملف ثنائي:

```
Program BinaryFile( input, output );
Var   fb : file of real;
    r  : real;
    k  : integer;
Begin
  assign( fb,   'fnb.bin');
  rewrite( fb );
  r := 0 ;
  for k:=1  to  200 do
   begin
     r := r + 1/k ;
     write( fb, r );
   end;
```

```
  close( fb );
End.
```

برنامج الكتابة على ملف نصي:

```
Program TextFile( input, output );
Var  ft : text;
     r : real;
     k : integer;
Begin
   assign( ft,   'fnb.txt');
   rewrite( ft );
   r := 0 ;
   for k:=1 to 200 do
    begin
      r := r + 1/k ;
      writeln( ft, r );
    end;
  close( ft );
End.
```

القراءة من الملفات السابقة:

سنقوم بقراءة الملفات السابقة ونقلها إلى جدول أعداد array وليكن A.

الخوارزمية بسيطة وهي كالتالي:

1- ربط الملف الفعلي بمتحول الملف

2- تأهيل الملف للقراءة

3- إعطاء عداد العناصر(الموجودة في الملف) k القيمة البدائية 0.

4- { قراءة عناصر الملف للجدول A }

5- {كتابة محتوى الجدول A على الشاشة}

```
4- { قراءة عناصر الملف للجدول A }
مادام ( لم ينته الملف) كرر
        k := k + 1 |
        | اقرأ عنصراً من الملف للمتحول Ak
N := k
```

```
5- {كتابة محتوى الجدول A على الشاشة}
من أجل k تأخذ 1 حتى N كرر
        | اكتب Ak
```

تتشابه إذن برامج قراءة الأعداد من الملفات السابقة، مع فارق أساسي: القراءة من ملف نصي تجري بـ readln ، في حين تجري القراءة من ملف ثنائي بالضرورة باستخدام read (مفهوم نهاية السطر غير وارد في الملفات الثنائية). وفيما يتعلق بآلية التنفيذ العملي، هناك سرعة أكبر لدى قراءة الملفات الثنائية ويعود السبب إلى كون القراءة من ملف ثنائي هي مجرد نسخ من الذاكرة الثانوية إلى المتحول (A[k] في المثال) ، في حين يعقب القراءة من ملف نصي تحويل من تمثيل العدد كسلسلة أحرف إلى التمثيل الثنائي كنمط عددي يكوِّن محتوى متحول القراءة (A[k] في المثال).

برنامج القراءة من الملف الثنائي:

```
Program BinaryFileRead( input, output );
Var  fr : file of real;
     k, N: integer;
     A  : array[1..200] of real;
Begin
   assign( fr, 'fnb.bin');
   reset( fr );
   k := 0;
   while ( not eof( fr ) ) do
    begin
      k := k+1;
      read( fr, A[ k ] );
    end;
   N := k;
   for k := 1 to N do
       writeln( A[k] );
   close( fr );
End.
```

برنامج القراءة من الملف النصي:

```
Program TextFileRead( input, output );
Var  ft : text;
     k, N: integer;
     A  : array[1..200] of real;
Begin
   assign( ft, 'fnb.txt');
   reset( ft );
   k := 0;
```

```
  while ( not eof( ft ) ) do
   begin
     k := k+1;
     readln( ft, A[ k ] );
   end;
  N := k;
  for k := 1 to N do
    writeln( A[k] );
  close( ft );
End.
```

مثال متقدم:

نورد فيما يلي مثالاً على استخدام الملفات الثنائية. وليست عناصر هذه الملفات بالضرورة من نمط بسيط. سنتعامل هنا مع ملفات من تسجيلات (file of record). إن آلية العمل للكتابة على (أو القراءة من) الملف تبقى ذاتها، سواء كانت عناصر الملف بسيطة (أعداد كما في الأمثلة السابقة) أم مُركَّبة (تسجيلات) كما في المثال الحالي.

يتعامل المثال هنا مع بنية تسجيلات الموظفين كما أوردناها في الفصل الخاص بالتسجيلة.

لدينا معطـــــــــــيات الموظفين (رقم، راتب، اسم) مكتوبة كنص مقروء في ملف نصي (employe.txt).

1-نقوم بقراءة محتويات الملف النصي السابق المربوط بالملف المنطقي (ftxt) ، و كتابتها في ملف ثنائي (employ1.bin) مربوط بالملف المنطقي (fbin1).

2-نجري بعدها عملية قراءة من هذا الملف الذي كتبناه قبل قايل (fbin1) لجدول ونظهر على الشاشة البيانات المقروءة، ليتأكد لنا أن ما خزن في الجدول TabEmp قد كان صحيحاً.

3- نقوم بترتيب الجدول (جدول الموظفين) وفق ترتيب الأسماء الأبجدي .

4-نخزن عناصر الجدول (تسجيلات الموظفين المرتبة) في ملف ثنائي آخر (employ2.bin) مربوط بالملف المنطقي (fbin2).

5-نقوم بفتح الملف المرتب (fbin2) وقراءة محتوياته (المرتبة) وإظهارها على الشاشة.

```
Program FileRecordExample( input, output );
(* using file of record *)
Const MaxN=100;
Type String20 = String[20] ; (*array[1..20] of char; in standard *)
Type  Employee = record
                  EmpNb    : integer;
                  EmpName  : String20;
                  EmpSalary : real;
                 end;

Var   ftxt : text;
      fbin1, fbin2 : file of Employee;
      TabEmp       : array[1..MaxN] of Employee;
      tEmp    : Employee;
      i, j, N : integer;
```

```
Begin
(* reading from a text file (ftxt) and  writing to a binary file (fbin1) *)
 assign(ftxt,'employe.txt');
 reset(ftxt);
 assign(fbin1,'employ1.bin');
 rewrite(fbin1);
 while not eof(ftxt) do  (** 1 **)
   begin
    readln(ftxt, tEmp.EmpNb, tEmp.EmpSalary, tEmp.EmpName );
    write(fbin1,tEmp);
   end;
 close(ftxt);  close(fbin1);

 (* Reading a binary file into  anarray and displaying its contents *)
 reset( fbin1 );
 writeln(' Displaying the data written in the binary file :');
 i:=0;
 while not eof(fbin1)  do  (** 2 **)
   begin
    read(fbin1,tEmp);
    with tEmp  do
      writeln(EmpNb:4, EmpSalary:10:2, EmpName );
    i := i + 1;
    TabEmp[i] := tEmp;
   end;
 N := i;

 (* data sorting *)          (** 3 **)
 for i:=1 to N-1 do  (* buble element *)
  for j:=1 to N-i do
   if ( TabEmp[j].EmpName > TabEmp[j+1].EmpName ) then
    (* change positions *)
     begin
      tEmp      := TabEmp[j];
      TabEmp[j]  := TabEmp[j+1];
      TabEmp[j+1] := tEmp;
```

```
    end;

 (* writing the sorted array to a binary file (fbin2)  *)
  assign(fbin2,'employ2.bin');
  rewrite(fbin2);          (** 4 **)
  for i:=1 to N do
     write( fbin2, TabEmp[i] );
  close(fbin2);

 (* Reading the binary file (fbin2) that contains sorted data  and displaying
its contents *)
  reset(fbin2);          (** 5 **)
  writeln;  writeln(' Sorted Elements ');  writeln;
  while ( not eof(fbin2) ) do
    begin
      read(fbin2,tEmp);
      with tEmp do
        writeln(EmpName, EmpNb:4, EmpSalary:10:2);
    end;
End.
```

مثال على تنفيذ البرنامج:

المعطيات الموجودة أصلاً في الملف النصي (employe.txt).

```
10      3000    Zynab Badoui
50      4000    Ahmad Mattar
90      8000    Bader Khatib
60      7000    Yamen Anbar
30      5000    Kassem Moustafa
70      4500    Hanan Lala
```

نتيجة تنفيذ البرنامج:

```
Displaying the data written in  the binary file :
 10    3000.00    Zynab Badoui
 50    4000.00    Ahmad Mattar
 90    8000.00    Bader Khatib
 60    7000.00    Yamen Anbar
 30    5000.00    Kassem Moustafa
 70    4500.00    Hanan Lala

 Sorted Elements

Ahmad Mattar          50    4000.00
Bader Khatib          90    8000.00
Hanan Lala            70    4500.00
Kassem Moustafa       30    5000.00
Yamen Anbar           60    7000.00
Zynab Badoui          10    3000.00
```

ملاحظة:

جرى إظهار التسجيلات المرتبة، التي كنا قد خزناها في الملف الثنائي المرتب، بذكر الاسم أولاً. لكن هذا لا يعني أنها خزنت في الملف بطريقة مختلفة (ببنية مختلفة) عن الملف الثنائي السابق (الملف دون ترتيب). كل ما في الأمر، أننا أردنا إظهار النتائج مرتبةً بوضع الاسم

أولاً، لكي نوضح للقارىء أن إظهار عناصر التسجيلة يمكن أن يجري بالترتيب الذي نريد.

العمليات الإضافية على الملفات (عمليات غير قياسية not standard)

كل ماذُكر عن الملفات، حتى الآن، يتعلق بالملفات التسلسلية (Sequential Files)،وفيها يجري الوصول إلى عنصر تلو الآخر. والعمليات إما قراءة أو كتابة، ولايجوز التعامل مع الملف بالقراءة والكتابة في الوقت ذاته.

هكذا هي الملفات كما هي معرّفة في Pascal القياسية. ولكن ذلك ليس عاماً في لغات البرمجة. إذ يتيح العديد منها إمكان الوصول المباشر لعنصر معين (دون المرور بالتسلسل على ماقبله)، ويمكن قراءة العناصر وكتابتها في الوقت ذاته.

سنعرض هنا التوسع الموجود في Turbo-Pascal لإتاحة العمليات الإضافية على الملفات (file of حصراً) والمذكورة آنفاً (وصول مباشر و قراءة/كتابة).

يوجد متحول FileMode مُضمّن مع Turbo-Pascal يحدّد طبيعة التعامل مع الملف : قراءة فقط، كتابة فقط، قراءة وكتابة معاً، ويأخذ FileMode القيم 0 ، 1 ، 2 وفقاً للحالات الثلاث على الترتيب. والقيمة الإفتراضية له هي 2 أي سماحية القراءة والكتابة معاً.

الوصول المباشر: Seek(f: file of ..., i : integer);

تضع هذه الإجرائية الموقع الحالي للملف f في الموقع i ، أي على العنصر i من الملف. يبدأ ترقيم العناصر بالقيمة 0 .

التوابع المتاحة:

Function FilePos(var f: file of ...) : integer;

يعطي رقم العنصر الحالي في الملف f .

Function FileSize(var f: file of ...) : integer;

يعطي عدد العناصر الموجودة في الملف.

أما عمليات القراءة والكتابة فتجري كما عهدناها في الموقع الحالي للملف باستخدام read و write، ويجب ألا ننسى أن كلتا العمليتين تتقدم بمؤشر الملف خطوة للأمام.

مثال:

نخدم البرنامج التالي العمليات المذكورة آنفاً. يقوم هذا البرنامج بكتابة أعداد صحيحة (الأعداد: 0 ، 10 ، 20 ،.... 90) في ملف ويغلقه. يُفتَح الملف من جديد، ثم يقوم البرنامج بتكرار القراءة إبتداءاً من آخر الملف، مع اظهار رقم العنصر المقروء وقيمته، وتغيير القيمة (كتابة عدد آخر مكان السابق: رقم العنصر x 100). وأخيراً، نُظهر محتويات الملف (بعد التغيير) بطريقة تسلسلية كما فعلنا في أمثلة سابقة .

```
program AccessFileRW;
var  f:file of integer;
   findx,i,pf,vi:integer;
begin
assign(f,'fnum.bin');
rewrite(f);
(* fill the file with elements 0,10,20.....90 *)
for i:=0 to 9 do
```

```
 begin
   vi := i*10;
   write(f,vi);
 end;

close(f);

reset(f);
writeln('Position  Value');
writeln('--------  -----');
for findx:=FileSize(f)-1 downto 0 do
 (* acces file startating from the end *)
 begin
   Seek(f,findx);
   pf:=FilePos(f);   write(pf:8); (*write positon *)
   read(f,i);          writeln(i:6); (* write value *)
  (* change the file element which has been read *)
  (* new value= position*100 *)
   vi := i*100;
   Seek(f,pf);
   write(f,vi);
 end;
writeln;
close(f);

reset(f);
writeln('File Size =',FileSize(f));
writeln;
writeln('The values of file elements are :');
while not eof(f) do
 begin
   read(f,vi);
    write(vi,', ');
 end;
writeln;
end.=
```

نتيجه تنفيذ البرنامج:

```
Position  Value
--------  -----
       9     90
       8     80
       7     70
       6     60
       5     50
       4     40
       3     30
       2     20
       1     10
       0      0

File Size =10

The values of file elements are :
0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000,
```

الفصل الخامس
نمط المعطيات:
مُؤشِّر Pointer

مقدمة:

المتحول "الساكن" static variable أو المحجوز بشكل ساكن، هو ما نصرِّح عنه في البرنامج ويُحدَّد باسم تعريف identifier . يُوصَف بالساكن لأن الذاكرة المُكرَّسة له تبقى محجوزةً له طوال مدة تنفيذ البرنامج. يجب الانتباه إلى أن الذاكرة المُكرَّسة للمتحولات المُعرَّفة محلياً Local variable ضمن برنامج جزئي، تبقى محجوزةً خلال استدعاء البرنامج الجزئي أو تنفيذه .

يوجد في العديد من لغات البرمجة (Pascal, C, Ada) إمكان لحجز متحولات بشكل ديناميكي، أي إن أماكن هذه المتحولات وعددها لاتُحدَّد سلفاً (بشكل ساكن) بل تتغير تبعاً لآلية تنفيذ البرنامج. نسمي هذه المتحولات بالمتحولات الديناميكية dynamic variable.

إن المبدأ الأساسي في التعامل مع المتحولات الديناميكية هو تكريس ذاكرة (ديناميكية) البرنامج، يقوم بالحجز فيها لاحتواء متحولات واستخدامها وتحرير الحجز (إعادة الذاكرة المحجوزة لاستخدامات أخرى) فيما بعد. أما طريقة الحجز فتجري باستخدام مؤشرات pointers (عناوين) إلى الذاكرة "الديناميكية" المسماة Heap أو كومة وذلك بسبب فكرة

الأخذ والإعادة منها وإليها. لذا فإن استخدامنا للمتحولات الديناميكية لا يجري مباشرةً بل عن طريق مؤشر.

إن فكرة الاستخدام الديناميكي للذاكرة والمؤشرات تنطوي على فائدتين هامتين:

الاستغلال الأفضل للذاكرة:

نستخدم الذاكرة الديناميكية بواسطة المؤشرات في كثير من التطبيقات التي لانعرف فيها سلفاً عدد المتحولات التي نحتاج إليها أثناء التنفيذ، ويمكن استخدام الذاكرة المكرسة لهذه المتحولات (بعد انعدام حاجتنا إليها) لصالح متحولات ديناميكية أخرى.

استغلال فكرة المؤشر لإنشاء بنى معطيات متقدمة:

نستخدم المؤشرات عندما نحتاج في بعض التطبيقات إلى نمذجة معطيات ذات ارتباطات معقدة. مثلاً نحتاج إلى تمثيل بنية شجرية أو سلسلة من العناصر...الخ.

نمط المعطيات: مؤشرفي لغة Pascal

التصريح عن نمط «مؤشر»:

الفكرة الحدسية عن مؤشر هي أنه عنوان أو دليل إلى متحول أو خانة ذاكرة. في لغة Pascal يُعرَّف المؤشر (بالضرورة) إلى نمط معيَّن، وعلى هذا تُكوِّن المؤشرات إلى نفس النمط نمطاً، ومن ثَمَ يمكن الإسناد فقط بين المتحولات المشيرة إلى النمط نفسه.

يجري التصريح بالشكل:

```
Var  PT : ^ T;
```

أي إن المتحول PT هو مؤشر (^) إلى متحولات من النمط T.

للوصول إلى المتحول الذي يشير إليه PT نستخدم الكتابة PT^.

يمكن إذن الكتابة التالية:

```
T ə PT^   <=   Var   PT  :  ^ T;
```

ليكن لدينا VT متحولاً من النمط T ، أي Var VT : T;

عندها يمكن الإسناد بين VT والمتحول المشار إليه بـ PT:

PT^ := VT; وبالعكس VT := PT^;.

مثال:

```
var  pr : ^real;   x :real;
x := pr^;    pr^ := 3.1416;
```

وكما هي القاعدة للمتحولات من نمط واحد، يمكن الإسناد بين المؤشرات لنفس النمط:

```
Var P1, P2 :  ^ T;
```

يمكن الإسناد P1 := P2;.

<u>أما الإسناد بين مؤشرين لنمطين مختلفين فلايجوز:</u>

```
Var   PT1 : ^T1;   PT2 : ^T2;
```

PT1 := PT2; تعتبر خطأ.

تذكر ما يلي:

P^ تُقرأ المتحول المؤشَّر عليه بواسطة P .

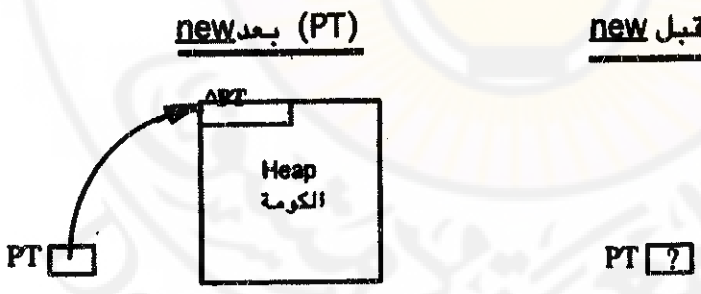
العمليات على المؤشرات:

1-حجز متحول بواسطة مؤشر:

تتلخص العملية الأساسية للحجز بالإجرائية new.

Var PT : ^T;

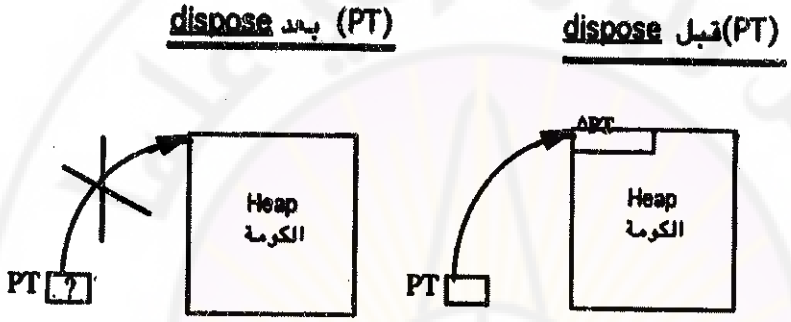
new(PT); تقوم بحجز متحول من النمط T جاعلةً PT يشير إلى ذلك المتحول.



وبهذا صار لدينا متحول من النمط T يُشار إليه بالمؤشر PT ونكتبه بالشكل: PT^.

2-تحرير الذاكرة المحجوزة بمؤشر:

أما العملية المعاكسة أي تحرير الذاكرة المحجوزة بالمؤشر PT فتجري بالإجرائية dispose(PT)



3-القيمة الخاصة nil للمتحول من النمط «مؤشر» :

إن القيمة الموجودة ضمن المتحول من نمط «مؤشر» ليست معروفة لنا. إنها تحوي عنواناً، لانستطيع معرفة قيمته، وما بنا حاجة إليه، لكنه يوصلنا إلى المتحول الذي يشير إليه بالكتابة ^< اسم المؤشر> .

هناك قيمة خاصة للمتحول من النمط «مؤشر» مُسبقة التعريف بالاسم nil يمكن استخدامها في التطبيقات على المؤشرات. تفيد هذه القيمة الخاصة للدلالة على أن المتحول المؤشر ذا القيمة nil لايشير إلى أي متحول.

وعلى هذا، إذا كان:

PT := nil;

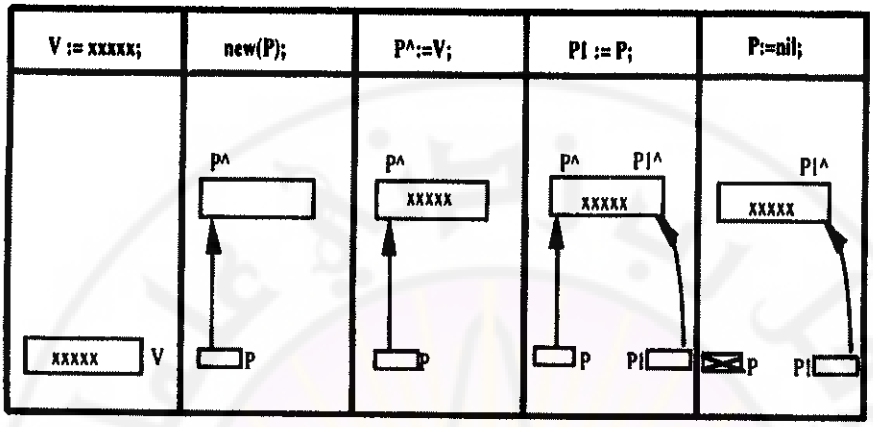
فلايجوز لنا الكتابة: PT^ إذ إننا نطلب المتحول المشار إليه بـ PT ، و PT يحوي nil أي لايشير. سنرى أهمية استخدام القيمة الخاصة nil في التطبيقات التي ننشىء بها بنى معطيات متقدمة بواسطة مؤشرات، إذ يمكن استخدامها للدلالة على نهاية سلسلة أوللدلالة عدم وجود عناصر في بنية مثلاً. يمكن اعتبارnil القيمة "الصفرية" للمؤشرات.

قبل البدء بشرح بعض الأمثلة على استخدام المؤشر، نشير إلى أن البساطة النظرية حول المؤشر (العمليات هي: new، dispose، nil فقط) لاتعني السهولة في استخدامه إذا لم نتوخ الحذر والانتباه في التعامل معه، فالإسناد الخاطىء للمؤشر يجعلنا نجد صعوبةً في ملاحقة الخطأ واكتشافه.

ننصح بالاستعانة بالأشكال لملاحقة العمليات على المؤشرات.

في الشكل المعطى مثال على تمثيل العمليات.

العملية وشكل يمثل أثرها:



لدينا المتحولات التالية: var P, P1: ^T; V: T;

العمليات من اليسار إلى اليمين:

1- V := xxxxx;

إسناد قيمة للمتحول V وهو من النمط T.

2- nèw(P)

خلق أو حجز متحول من النمط T نشير إليه بالمتحول "المؤشر" P.

3- P^ := V;

المتحول V و P^ هما من نفس النمط T، نستطيع إسناد قيمة V إلى المتحول المشار إليه بواسطة P أي المتحول P^.

P1 := P; -4

المؤشر P1 هو من نفس نمط P كلاهما مؤشران لمتحولات من النمط T.

نتيجة الإسناد يصبح للمؤشرين P و P1 القيمة نفسها، ولهذا فهما يؤشران إلى نفس الخانة أي إن: P1^ هو نفسه P^.

P := nil; -5

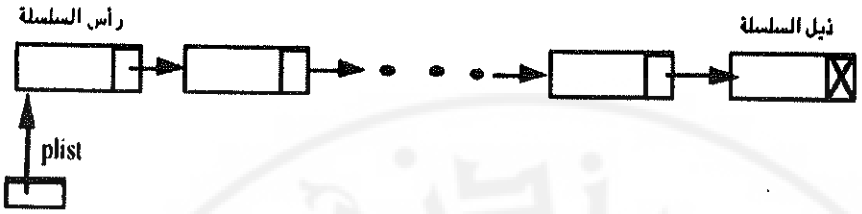
إسناد القيمة الخاصة **nil** إلى المؤشر P تجعله لا يشير إلى أي مكان، غالباً مايُصطلح في الأشكال الممثلة للمؤشرات، على التعبير عن القيمة **nil** بوضع الرسم X في خانة المؤشر.

المؤشرات في إنشاء بنى معطيات متقدمة:

ذكرنا أن المؤشرات هي الأداة لاستخدام الذاكرة الديناميكية، ولكن استعمالاتها كثيرة في نمذجة بنى معطيات متقدمة حين نريد التعبير عن الترابط بين عناصر.

لإنشاء مثل هذه البنى نقوم بتجميع معطيات العنصر مع مؤشر (أومؤشرات) في نفس البنية، وكما نعلم يجري تجميع معطيات متنوعة ككتلة بواسطة التسجيلة **record**، لذلك نجد في معظم البنى التي تستخدم المؤشر في التمثيل أن اللبنة الأساسية هي تسجيلة تحوي مؤشراً (غالباً ما يشير إلى نمط التسجيلة نفسه).

لنأخذ مثالاً على بنية بسيطة نسبياً (ستكون محور أمثلتنا على العمليات عن المؤشرات) سلسلة مترابطة من العناصر كل عنصر فيها يدل على العنصر التالي.



والآن كيف نعرف مثل هذه البنية اعتماداً على المؤشرات؟

لنحدد حقول عنصر السلسلة أولاً، ولتكن مؤلفة من حقل تحديد (نسميه مفتاح key) وليكن عدداً صحيحاً أو سلسلة محارف، إضافةً إلى حقول أخرى لاتهمنا الآن في سياق البنية. ولكن حقلاً أساسياً من حقول عنصر السلسلة سنضعه ليُعبِّر عن ارتباطه بالعنصر الذي يليه وهو عبارة من مؤشر. ونعرف أنه لايكفي أن نقول مؤشر بل مؤشر إلى ماذا (إلى أي نمط) ؟ والجواب أنه مؤشر إلى نفس نمط العنصر الذي ينتمي إليه (أي الذي هو حقل منه).

لكسر هذه العَوْدية في التعريف، أي الكتابة المباشرة: نمط العنصر هو تسجيلة أحد حقولها هو مؤشر إليها التي تُكتَب بالشكل:

```
TYPE ELEM = record
              key  : integer;
              next : ^ELEM ;
            end  ;
```

نقوم بتعريف نمط المؤشر أولاً على أن نحدد تعريف النمط المشار إليه فيما بعد:

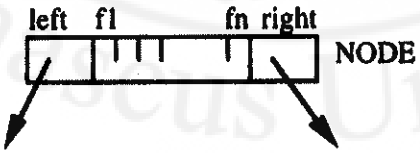
```
Type PELEM = ^ELEM;
Type ELEM  = record
       key  : integer;
       next : PELEM ;
     end;
```

بعد تعريفنا هذا يمكن أن نشير إلى السلسلة بمتحول "مؤشر" يدل على أول عنصر(رأس السلسلة) نسميه plist مثلاً، وهذا كافٍ لتحديد السلسلة إذا كنا سنبنيها كما أسلفنا بأن كل عنصر فيها يشير إلى العنصر التالي، فمؤشر الرأس يوصلنا إلى أول عنصر، والمؤشر الذي يتضمنه هذا العنصر (قيمة الحقل next) يوصلنا إلى العنصر التالي، وهكذا دواليك، حتى نصل إلى آخر عنصر في السلسلة، الذي سنجعل في مؤشره القيمة المميزة nil دلالة على أنه آخر العناصر (ذيل السلسلة). سنتناول هذه البنية بالتفصيل في المثال البرمجي لهذا الفصل.

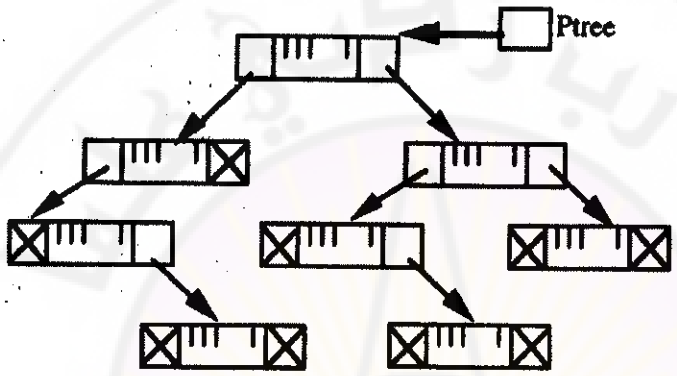
١ه العديد من البنى المتقدمة المفيدة اعتماداً على المؤشر:

بنية شجرية ثنائية الفروع أي كل عقدة NODE لها فرعان على الأكثر (وهذه بنية مستخدمة في الكثير من التطبيقات المعلوماتية)، لبناء هذه البنية نضيف إلى حقول العنصر مؤشرين أحدهما يشير إلى الفرع اليساري left والآخر يشير إلى الفرع اليميني right.

```
TYPE
PNODE =  ^ NODE;
NODE = record
            f1,...,fn : ....;
          left, right : PNODE;
        end;
Var    Ptree : PNODE;
```



أما الشجرة عند بنائها فتأخذ تمثيلاً مشابهاً لما هو في الشكل التالي حيث نضع القيمة الخاصة **nil** عندما لايكون للعقدة فرع.



مثال برمجي:

التعامل مع سلسلة عناصر مرتبة

سنقوم بخزن مجموعة من العناصر على نحوٍ مُرتَّب ضمن سلسلة مترابطة، سنفترض أن هناك حقلاً قابلاً للترتيب (مفتاح Key) كما ذكرنا عن بنية السلسلة آنفاً.

```
Type PELEM = ^ELEM;
Type ELEM  = record
       key  : integer;
       next : PELEM ;
     end;
```

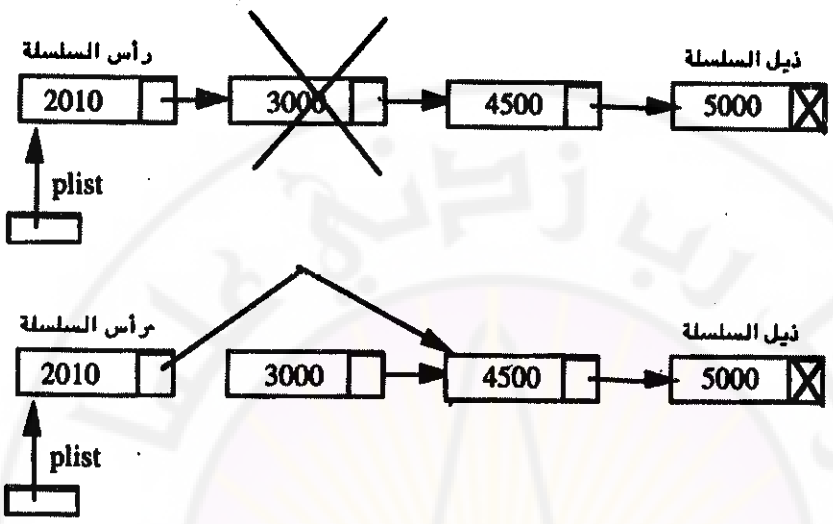
سنقوم بالعمليات الأساسية على هذه السلسلة بشكل تخاطبي.

تتضمن العمليات مايلي:

- حشر عنصر ضمن السلسلة (Insert)
- حذف عنصر من السلسلة (Delete)
- البحث عن عنصر (Find)
- إظهار محتويات سلسلة العناصر (Display)

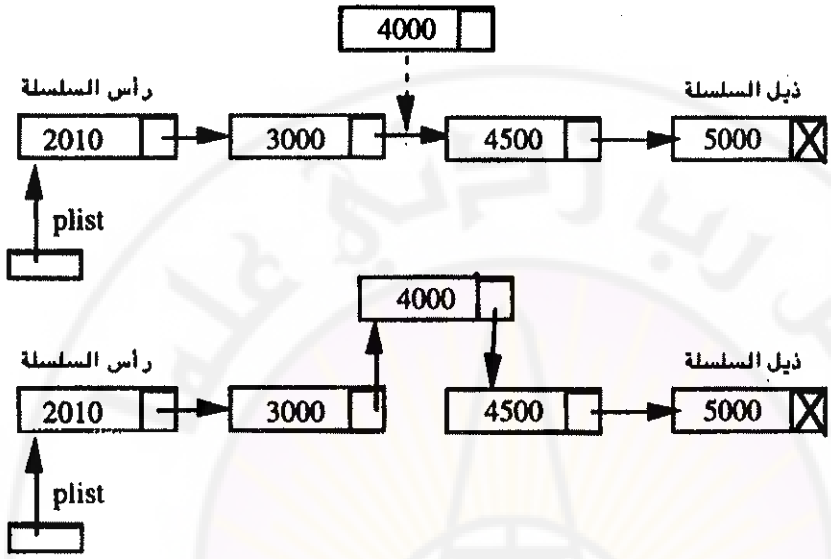
سنوضح مباشرةً أهمية استخدام المؤشرات لمثل هذا النوع من العمليات. فالبنى التقليدية تجعلنا نخزن العناصر مرتبةً ضمن جدول (array)، وأي إضافة لعنصر جديد في موقع ترتيبه المناسب، تؤدي إلى إزاحة العديد من العناصر وكذلك الحال في حالة حذف عنصر. أما بنية السلسلة المعتمدة على المؤشرات فتعالج هذه العمليات بيسر.

يوضح الشكل التالي حذف العنصر 3000 من السلسلة:



نقوم فقط بتبديل مؤشر العنصر السابق له ليؤشر إلى العنصر اللاحق له في الترتيب وهكذا يصبح العنصر 3000 غير مربوط بالسلسلة، فيصير محذوفاً منها.

يوضح الشكل التالي إضافة (حشر) العنصر 4000 في السلسلة المرتبة:



نقوم فقط بجعل مؤشرالعنصر السابق له بالترتيب (ذي القيمة «المفتاح» 3000) يؤشر إليه، وجعل مؤشر هذا العنصر الجديد يؤشر إلى العنصر التالي بالترتيب (4500).

الخوارزمية العامة للبرنامج

المعطيات والمعطيات اللازمة:

```
Type PELEM = ^ELEM;
Type ELEM  = record
       key  : integer;
       next : PELEM ;
     end;
```

e : ELEM; عنصر من نمط عناصر السلسلة

plist : PELEM; مؤشر إلى رأس السلسلة

pe : PELEM; مؤشر إلى عنصر نمطه من عناصر السلسلة.

r: char; حرف يحدد نوع العملية التي يختارها المستخدم:

'I','i' تعني Insert أو حشر

'D','d' تعني Delete أو حذف

'L','l' تعني List أو إظهار

'F','f' تعني Find أو بحث

'Q','q' تعني Quit أو خروج

الخوارزمية :

كُدْ

1-{خذ خياراً (r) من المستخدم}

2- في حالة الخيار r

| حشر : | اقرأ e.key

| | 2-1- {احشر العنصر e ضمن السلسلة المشار إلى رأسها بـplist}

| حذف : | اقرأ e.key

| | 2-2- (احذف العنصر e من السلسلة المشار إلى رأسها بـplist)
| اظهار : | 3-2 - (أظهر عناصر السلسلة المؤشر المشار إلى رأسها بـplist)
| بحث : | اقرأ e.key
| | 2-4- (ابحث عن العنصر e من السلسلة المشار إلى رأسها بـplist)
| | (صح موجود، خطأ غير موجود، و pe مؤشر إلى مكان وجوده)
| | اكتب نتيجة البحث

حتى (الخيار خروج)

1-(خذ خياراً (r) من المستخدم)

نحققه بالإجرائية menu التي تظهر للمستخدم الخيارات وتقرأ منه أحد الخيارات (حرف يدل على العملية)

Procedure menu(var c: char);

تأخذ هذه الإجرائية حرف c (الخيار) كمعامل خرج.

2-1- (احشر العنصر e ضمن السلسلة المشار إلى رأسها بالمؤشر plist)

نحققه بطلب الإجرائية Insert .

Procedure Insert(var ph : PELEM; e : ELEM);

المعاملات الشكلية للإجرائية هي: العنصر e الذي نريد إدخاله في السلسلة المرتبة المشار إليها بالمعامل ph المشير إلى أول عنصر، هذا المعامل هو معامل دخل وخرج لأن قيمة هذا المؤشر قد تختلف (مثلاً إذا حشرنا عنصراً قبل العنصر الأول في السلسلة يصبح المؤشر يدل على العنصر الجديد). وسنفصل هذه الإجرائية لاحقاً.

2-2- {احذف العنصر e من السلسلة المشار إلى رأسها بـplist}

نحققه بطلب الإجرائية Delete .

Procedure Delete(**var** ph : PELEM; e: ELEM) ;

المعاملات الشكلية للإجرائية هي: العنصر e الذي نريد حذفه (يمكن الاكتفاء بقيمة المفتاح) من السلسلة المرتبة المؤشر إليها بالمعامل ph المشير إلى أول عنصر، هذا المعامل هو معامل دخل وخرج لأن قيمة هذ المؤشر قد تتغير (مثلاً إذا حذفنا العنصر الأول في السلسلة يصبح المؤشر يدل إلى العنصر التالي).

2-3 - {أظهر عناصر السلسلة المشار إلى رأسها بـplist}

نحققه بطلب الإجرائية Display .

Procedure DispLay(ph : PELEM);

المعاملات الشكلية للإجرائية هي فقط المعامل ph المشير إلى أول عنصر في السلسلة وبالطبع هو معامل دخل فقط هنا، لأننا نعاين قيمة العناصر لكتابتها على الشاشة ولانغير بالسلسلة.

<u>الخوارزمية هنا بسيطة:</u>

نستخدم متحولاً مساعداً p من نمط مؤشر ليكون دليلنا على عناصر السلسلة، نعطيه كقيمة بدائية قيمة مؤشر رأس السلسلة ph ، ثم نعطيه تباعاً قيمة المؤشر (إلى التالي أي next) في العنصر الحالي (العنصر المشار إليه بالدليل p) حتى نصل إلى آخر عنصرأي عندما تصبح قيمة الدليل p مساوية لـ nil.

p := ph;
مادام (p لايساوي nil) كرر
| اكتب p^.key
| p := p^.next

2-4- {ابحث عن العنصر e من السلسلة المشار إلى رأسها بـplist }
نحققه بطلب التابع Find .

Function Find(ph : PELEM ; e :ELEM; **Var** pe: PELEM) : boolean;

المعاملات الشكلية للتابع هي: العنصر e الذي نريد البحث عنه (يمكن الاكتفاء بقيمة المفتاح) في السلسلة المرتبة المشار إليها بالمعامل ph المؤشر إلى أول عنصر. هذان المعاملان (e و ph) هما معاملا دخل. التابع يعيد قيمة منطقية boolean : True إن وجد العنصر وFalse إن لم يكن موجوداً .نضيف أيضاً معامل خرج هو مؤشر pe إلى مكان العنصر إن وجد في السلسلة، ونعطيه القيمة **nil** إن لم يكن موجوداً.

سنفصل إجرائية حشر (إدخال) عنصر في السلسلة المرتبة:

المُعاملات:

ph مؤشرإلى رأس السلسلة، ونفترض أن السلسلة فارغة، إذا كانت فيه القيمة **nil**.

e العنصر المطلوب إدخاله في السلسلة وهو عبارة عن تسجيلة تحوي مفتاحاً يهمنا في الترتيب.

سنميز الحالات التالية:

1-السلسلة فارغة أي: ph = nil

2-مفتاح العنصر e أصغر من مفتاح أول عنصر أي: e.key < ph^.key

في هاتين الحالتين تتغير قيمة العنصر الأول، ومن ثَمَ تتغير قيمة المؤشر ph.

3 -الحالة العامة عدا الحالتين السابقتين، أي إن مكان حشر العنصر سيكون في مكان ما في السلسلة، مختلفٍ عن الموقع الأول.

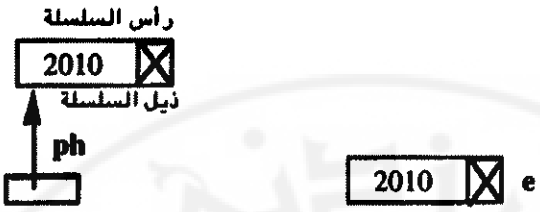
1-السلسلة فارغة أي: ph = nil



نقوم بمايلي:

```
if ( ph = nil ) then   (* list is empty *)
  begin
     new( ph );
     e.next := nil;
     ph^ := e ;
  end
```

فيصبح الوضع كالتالي:



2-مفتاح العنصر e أصغر من مفتاح أول عنصر أي: e.key < ph^.key

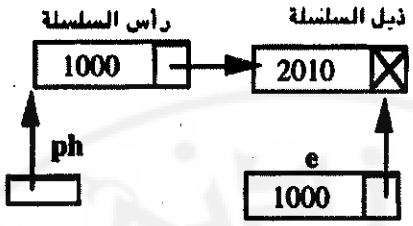
لنفترض أننا في السلسلة الحاوية للعنصر 2010 ونريد إضافة العنصر ذي المفتاح 1000:



العمليات اللازمة:

```
if ( e.key < ph^.key ) then  (* (e) is less than the first element *)
  begin
    e.next := ph ;
    new(ph);
    ph^ := e;
  end
```

يصبح الوضع كالتالي:



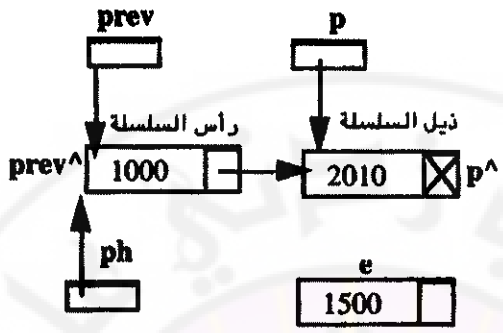
3 –الحالة العامة عدا الحالتين السابقتين، أي إن مكان حشر العنصر سيكون في مكان ما في السلسلة، مختلفٍ عن الموقع الأول.

في البدء، لابد من تحديد المكان الذي سيُحشر فيه العنصر، ولما كانت طريقة بنائنا للسلسلة تقوم على ترتيب العناصر، كان المكان الصحيح للعنصر المُدخَل هو قبل أول عنصر يكبره. وأثناء بحثنا عن أول عنصر يكبر العنصر الجديد، نستخدم دليلاً هو المؤشر p للدلالة على الموقع الحالي أثناء البحث، ونحتفظ بمؤشر آخر prev هو المؤشر إلى العنصر السابق. سنرى ضرورة هذا المؤشر لعملية الحشر. ليكن العنصر المطلوب إدخاله هو 1500.

يُحدَّد المكان الذي سيوضع فيه العنصر بالعمليات:

```
p:= ph;    prev := p;
while (( p<>nil )  and (p^.key < e.key))  do
  begin
    prev :=p;  p:=p^.next;
  end;
```

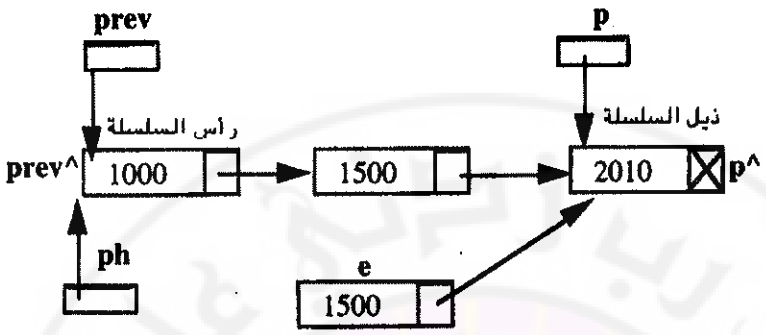
فنصل إلى وضع شبيه بالتالي:



أما إدخال العنصر e إلى السلسلة فيجري بتوجيه المؤشرات بالشكل المناسب: نجعل المؤشر الموجود ضمن prev^ يؤشر إلى العنصر الجديد، أما مؤشر العنصر الجديد فنجعله يؤشر إلى العنصر p^، كما في التعليمات التالية (نشترط أيضاً لإدخال العنصر ألا يكون موجوداً سابقاً):

```
if ( ( p<>nil ) and  ( p^.key <> e.key )) or (p=nil) then
  (*the first condition makes sure that there is no redundancy *)
 begin
   e.next := p;
   new( prev^.next );
   prev^.next^ := e ;
 end;
```

فيصبح الوضع كالتالي:



امتحن حالة كون العنصر المضاف أكبر من آخر عنصر، تجد أن التعليمات السابقة تصح أيضاً.

ملاحظة هامة جداً:

لاحظ أننا نهيء القيم في العنصر e ، ومنها المؤشر الموجود ضمنه، قبل الإسناد إلى العنصر المخلوق حديثاً في السلسلة بواسطة new. انتبه إلى الخطأ الكبير الذي يمكن أن يحصل إذا نسينا عملية new هنا!!

النص الكامل لبرنامج العمليات على سلسلة مرتبة:

```
Program ListProcessing( input, output );
(*                                                          *)
(*  Example of pointer use : manipulation of ordered list with one key *)
(*                                                          *)
Type
  PELEM = ^ELEM;
  ELEM  = record
            key  : integer;
            next : PELEM ;
          end;
Var
  plist,pe : PELEM ;
  e        : ELEM ;
  r        : char ;


Procedure  menu( var c: char );
(*
 *  get user selected operation
 *)
Begin
   repeat
     writeln(' (I)nsert  (F)ind  (D)elete  (L)ist  (Q)uit :');
     readln( c );
   until ( c in  ['i','I','f','F','d','D','l','L','q','Q']  );
End;(* menu *)



Procedure  Insert( var ph : PELEM;  e : ELEM );
(*
 *  insert element (e) in the list pointed by (ph)
 *)
```

```
Var  p, prev : PELEM;

Begin
  if ( ph = nil ) then   (* the list is empty *)
     begin
        new( ph );
        e.next := nil;
        ph^ := e ;
     end
  else  (* list not empty *)
   if ( e.key < ph^.key ) then  (* (e) is less than first element *)
     begin
       e.next := ph ;
       new(ph);
       ph^ := e;
     end
  else   (* (e) is anywhere in the list *)
   begin
     p:= ph;    prev := p;
     while (( p<>nil )  and (p^.key < e.key)) do
       begin
         prev :=p;  p:=p^.next;
       end;

   if ( ( p<>nil ) and ( p^.key <> e.key )) or (p=nil) then
       (*the first condition makes sure that there is no redundancy *)
      begin
        e.next := p;
        new( prev^.next );
        prev^.next^ := e ;
      end;

  end;
End;(* Insert *)
```

```
Function  Find( ph : PELEM ;  e :ELEM; Var pe: PELEM ) : boolean;
(*
 *  Find  the element (e) in the list pointed to by (ph),
 *        (pe) is the pointer to the element in the list when find returns
TRUE.
 *)
Var p :PELEM;
Begin
  p :=ph;
  while  ( (p<>nil)  and (p^.key < e.key) )  do
      p := p^.next;
  if ( p <> nil)  and ( p^.key = e.key )  then
    begin
      pe := p ;  find := true;
    end
 else
     find := false ;
End;(* Find *)


Procedure Delete( var ph : PELEM;  e: ELEM ) ;
(*
 *  Delete the element (e) from the list pointed to by (ph) .
 *)
Var p,prev : PELEM;

Begin

 if ( ph<>nil )  then
   begin
      p :=ph;    prev := ph;
      while ( ( p<>nil)  and (p^.key < e.key) ) do
       begin
          prev := p;   p:=p^.next;
       end;
```

```
    if ( p=ph ) then
     begin
      ph:= p^.next ;  dispose(p);
     end
    else
     if ( ( p <> nil)  and ( p^.key = e.key ) ) then
      begin
       prev^.next := p^.next ;
       dispose( p );

      end;

  end;(* ph<>nil *)

End;(* delete *)


Procedure DispLay( ph : PELEM );
(*
 *  Display the elements of the list pointed to by (ph)
 *)
Var p :PELEM;

Begin
  p := ph ;
  while ( p<>nil ) do
   begin
     writeln( p^.key );
     p := p^.next ;
   end;
End;(* dispLay *)
```

```
Begin  (** Main Program **)
  plist := nil ;
  Repeat
     menu( r );
     case  r  of
       'I','i' :  begin
                   writeln(' input :');
                   readln( e.key );
                   insert(plist, e );
                  end;
      'L','l' :  Display( plist );
      'D','d' : begin
                   writeln(' input :');
                   readln( e.key );
                   Delete(plist, e );
                  end;
      'F','f' : begin
                   writeln(' input :');
                   readln( e.key );
                   writeln( Find(plist, e, pe ));
                 end;
    end;(* case *)
  Until ( r in ['Q','q'] );
End.
```

الفصل السادس
تعليمة القفز
Goto

مقدمة:

يحتوي العديد من لغات البرمجة على تعليمة القفز Goto والتي تنقل تنفيذ البرنامج (بالقفز) إلى نقطة معينة. ذكرنا منذ البداية، عند شرح الأساسيات، ضرورة الابتعاد عن هذه التعليمة للتعبير عن التحكم، والاعتماد على البنية المهيكلة للخوارزمية Structured Programming وذلك تسهيلاً لقراءة البرنامج وتنقيحه.

تجدر الإشارة إلى أن أي تعبير عن التحكم بـ Goto يمكن الاستعاضة عنه بأحد أشكال بنى التحكم التي عرفناها (..if, while..., if ...else...).

وإذا رأى المبرمج سهولة التعبير في بعض الحالات الاستثنائية جداً بواسطة Goto فينبغي أن يكون ذلك بالقفز نحو الأمام ، لا بالرجوع، وذلك لكي نحافظ على مبدأ القراءة التسلسلية للبرنامج.

تعليمة Goto في لغة Pascal :

كي نعنون النقطة التي نريد القفز إليها لابد من التصريح عن "عنوان تعليمة" بواسطة كلمة التصريح label .

الشكل القواعدي:

label <N1>,<N2>,.....<Nn>;

التصريح بواسطة **label** يقع في قسم التصريحات في السطر التالي مباشرة لترويسة البرنامج. <N1>,<N2>,.....<Nn> هي أرقام صحيحة موجبة تستخدم عناوين للقفز.

مثال:

label 100;

label 200,1000,2000; أو

أما تعليمة القفز **Goto** فتأخذ الشكل:

Goto <N>;

حيث <N> هي عنوان تعليمة، معرَّف بواسطة **label** ومستخدم في جسم البرنامج عنواناً لتعليمة.

تجري عنونة نقطة في البرنامج بوضع (:<N>) رقم مصرح عنه بواسطة **label** قبل النقطة المعنية.

مثال: 100: writeln('....goto point');

يجب ملاحظة أن التعليمات المعنونة بأرقام تُنفَّذ تسلسلياً أيضاً أي دون أن يكون قد حدث القفز إليها بواسطة Goto . فهي في هذه الحالة مختلفة عن تصريح البرامج الجزئية التي لاتنفذ إلا بالاستدعاء.

مثال برمجي:

ليكن لدينا البرنامج التالي :حساب الجذور التربيعية لمجموعة أعداد. نقرأ عدد الأعداد، ثم

نكرّر قراءة عدد وحساب جذره التربيعي. نريد الخروج من البرنامج بمجرد قراءة عدد سالب. للخروج سنستخدم تعليمة **Goto**.

```
Program TryGoto( input, output );
(* Show Goto use *)
Label 100;
Var
  i : integer;
  N : integer;
  x : real ;
Begin
  i:=1;
  readln( N );
  for i:=1 to N do
    begin
      readln( x );
      if ( x < 0 ) then
                  goto 100;
      writeln( x:10:6, sqrt(x):10:6 );
    end;
100:
End.
```

ملاحظة أخيرة:

في العديد من لغات البرمجة المهيكلة (C وAda مثلاً) يمكن الاستغناء بسهولة أكبر من Pascal عن تعليمة Goto ، إذ إن هناك بعض تعليمات تقوم مقامها للحالات الاضطرارية: الخروج من حلقة (break في C و exit في Ada)، و الخروج من البرنامج الجزئي (return). ونكرر هنا، أننا في لغة Pascal قادرون على الاستغناء عنها باستخدام متحولات منطقية في شروط التعليمات الشرطية والتكرارية، وعدم استخدامها إلا في حالات خاصة جداً، مثل الخروج من حلقات متراكبة أو من برنامج جزئي لدى حدوث خطأ ما.

سنرى فيما يلي مثالاً على تلك الحالات التي نستخدم فيها تعليمة القفز، وهنا قد تكون مبررة إلى حدٍ ما. في مجموعة التعليمات التالية، يوجد ثلاث حلقات تكرارية، ويقتضي منطق البرنامج أن نتوقف أو أن نترك كل كتلة التعليمات التكرارية في حال حدوث شرط ما (مثلناه بالمتحول المنطقي event) في التعليمة التي أشرنا إليها بالتعليق (****).

```
while ( ..)   do
begin
    ..............
  while (.. ) do
    begin
       ..............
      while  (.. ) do
        begin
           ..................
          if event  then      (****)
               goto 1000; (* exit from all loops *)
           ..................

        end;
      end;
end;
1000 :
```

إذا أردنا أن نكتب مجموعة التعليمات السابقة مستغنين عن تعليمة القفز، فإننا على ذلك لقادرون. ويكون ذلك بأن نضيف إلى شرط التكرار في جميع الحلقات السابقة، شرط عدم حصول الشرط event.

```
while (..) and not event do
begin
    .............
  while ( ..) and not  event do
    begin
       .............
      while  (.. ) and not  event do
        begin
            ..............
          (****)
        end;
      end;
end;
```

في هذا المثال الذي أوردناه، قد يكون التعبير البرمجي، بوجود تعليمة القفز، له وضوح القراءة نفسه كما في حال الاستغناء عنها، ولكنه أكثر فعـــــــالية إذ لا يُلزم باختبار الشرط (event) إلا مرة واحدة، وذلك في مكان حدوثه.

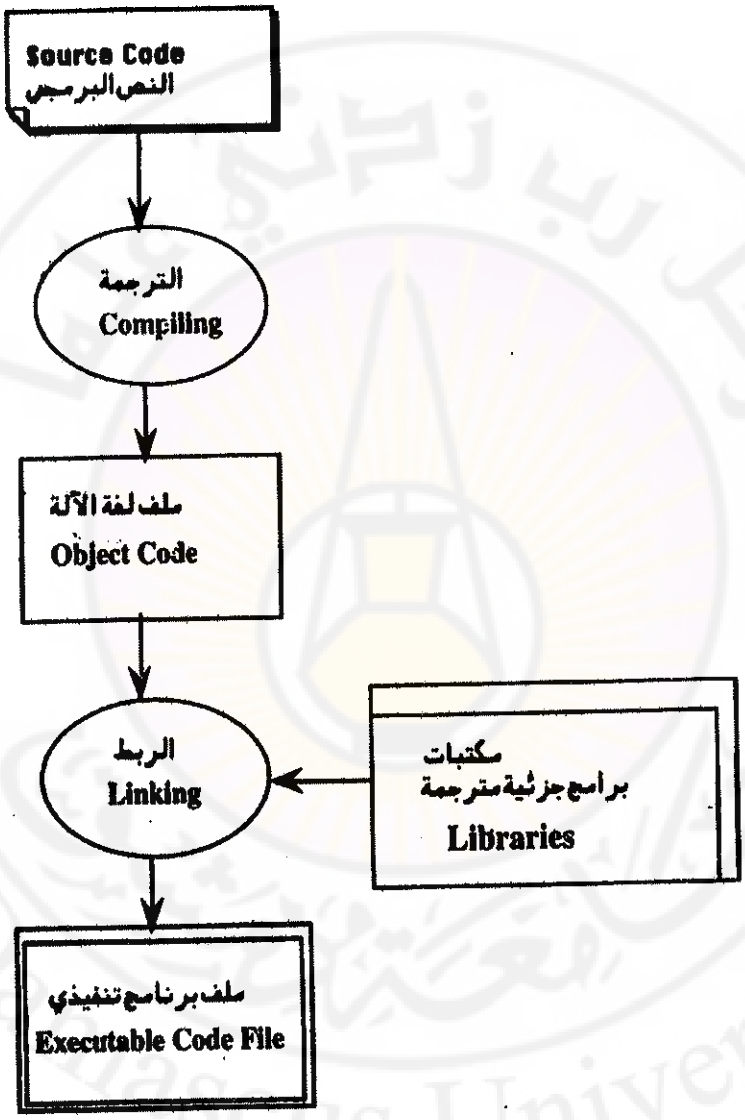
الفصل السابع
توزيع البرنامج على وحدات
Modular Programming

مقدمة: توزيع البرنامج على عدة ملفات

إن توزيع البرنامج على عدة ملفات ضرورة عملية لا غنى عنها من أجل التطبيقات البرمجيةالكبيرة. و يعتبر وجود برامج من مرتبة آلاف السطور (بل قُل عشرات ومئات الآلاف في العديد منها) أمراً عادياً في كثير من التطبيقات. ولايمكننا في هذه الحال أن نتخيل وجود كامل البرنامج ضمن ملف واحد، وأن نقوم بإعادة ترجمته كله عند أي تعديل أو تنقيح أو إضافة.

لقد عرفنا كيف نقسِّم البرنامج إلى برامج جزئية، وهذه المعرفة كانت خطوة أساسية في التغلب على التعقيد، عن طريق تقسيم المسألة الكبيرة إلى أجزاء صغيرة تكون السيطرة عليها أسهل. وقد لاحظنا أن البرنامج الجزئي هو كيان مستقل له تكامله المنطقي ويقوم بعمل محدد، ويُعرَّف بوضوح طريقة تواصله مع العالم الخارجي (مع الأجزاء الأخرى) عن طريق الوسطاء أو المعاملات أو عن طريق المتحولات المشتركة أو الكلية.

يستقرُّ العديد من البرامج الجزئية التي نكتبها، حين حلّنا لمسألة معينة بعد توثقنا من صحتها ولا نحتاج إلى تعديلها إلا لماماً، ومن ثَم لاحاجة لنا إلى الرجوع إلى ترجمتها في كل مرة نبني فيها البرنامج الرئيسي. ومن هنا كانت الفكرة الأساسية، منذ بدايات لغات البرمجة العالية المستوى، بتوزيع البرنامج على عدة ملفات، كل ملف منها يحوي عدداً من البرامج الجزئية، وتجري ترجمة كل ملف على حدة وبشكل مستقل. وبعدها تُربط الأجزاء المُترجَمة بهدف بناء البرنامج التنفيذي. وبذلك لن تكون ثمة حاجة إلى إعادة الترجمة إلا للملفات التي يجري تعديلٌ على نصها البرمجي، وهذا ما نسميه بالترجمة المستقلة وقد عرفت منذ لغة Fortran.



مراحل الانتقال من النص البرمجي إلى الملف التنفيذي

يوضح الشكل السابق طريقة الانتقال من ملف البرنامج النصي (المكتوب بلغة برمجة عالية المستوى) إلى ملف برنامج تنفيذي. ونميز فيه المراحل التالية:

- الترجمة Compiling : وفيها يقوم المترجم Compiler بنقل النص البرمجي الأساسي (المصدر) Source Code إلى لغة الآلة (الهدف) Object Code. تنقلنا عملية الترجمة إلى لغة الآلة، ولكن الناتج لايزال غير جاهز للتنفيذ، فهناك معلومات لم تكتمل بعد.

- الربط Linking: وفيها تقوم أداة الربط Linker باستكمال المعلومات الناقصة في ملف لغة الآلة السابق (الناتج من الترجمة). هذه المعلومات هي في الغالب نصوص برامج جزئية يوجد استدعاء أو طلب لها في نص البرنامج المصدر (المُتَرجَم) ولم تكتب نصوصها البرمجية في ملف المصدر ذاته. مثال ذلك البرامج الجزئية مسبقة التعريف predefined (التوابع الحسابية sin، cos مثلاً).

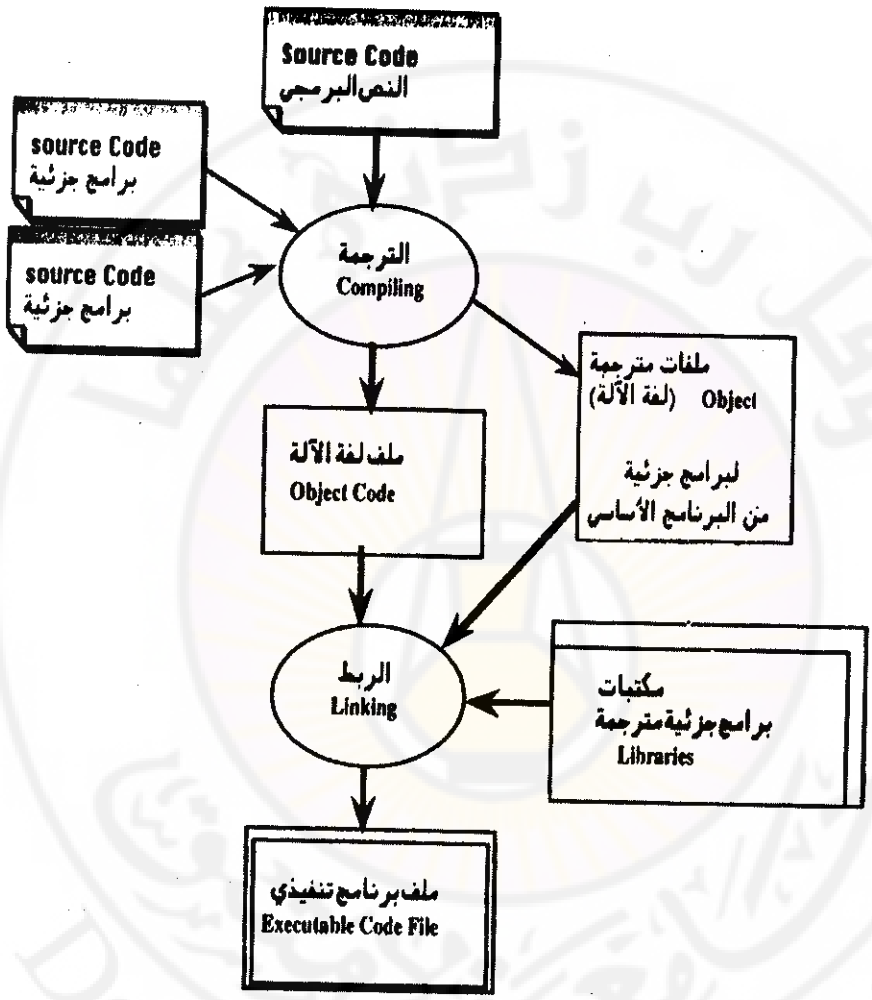
تقوم أداة الربط بأخذ هذه البرامج الجزئية الناقصة (بصيغة لغة الآلة)، من المكتبات (Libraries) المرافقة لمحيط تطوير اللغة، وربطها بالملف السابق للحصول على الملف التنفيذي النهائي.

الترجمة المستقلة والترجمة المنفصلة:

إن توزيع البرنامج، على الوجه السابق، على عدة ملفات، وترجمة كل ملف على حدة، ثم ربط الأجزاء المترجمة لتكوين الملف التنفيذي، هو ما سميناه الترجمة المستقلة

(Independent Compilation)، والتسمية «مستقلة» تحمل مدلولها، بمعنى أن كل جزء برمجي (برنامج جزئي أو البرنامج الأساسي) يُترجَم مستقلاً عن الأجزاء الأخرى، دون أي اختبار لتوافق الأجزاء بعضها مع بعض. ومثال ذلك عدم التحقق من توافق استدعاء أو طلب البرنامج الجزئي، مع تعريفه الأساسي (عدم التوثُّق من توافق عدد المعاملات وأنماطها في استدعاء البرنامج الجزئي مع تعريفه أو ترويسته).

يَصنُّع المخطط السابق (مخطط الانتقال من النص البرمجي إلى البرنامج التنفيذي) على ملف برنامج أساسي يستدعي أو يطلب برامج جزئية (مُتَرجَمة سابقاً) في مكتبة نظامية، ويمكن تطبيقه بسهولة في حال الترجمة المستقلة، لتقوم عملية الربط بإضافة البرامج الجزئية التي نكتبها نحن، وسبق لنا ترجمتها بشكل مستقل.



الترجمة المستقلة لأجزاء البرنامج

يقتصر دور المترجم هنا على التحقق القواعدي من شكل استدعاء البرنامج الجزئي (اسم تعريف متبوع بمعاملات محصورة بين قوسين). أما دور أداة الربط فلا يتعدى استكمال النقص في الملفات المُترجَمة، فكل استدعاء لبرنامج جزئي لم يرد تعريفه في الملف المُترجَم يجري البحث عنه في الملفات المُترجَمة الأخرى، المطلوب ربطها لتكوين البرنامج الكلي التنفيذي النهائي. وإذا لم يعثر عليها هناك، يجري البحث عنها في المكتبات المُلحَقة. وتُضاف ترجمات البرامج الجزئية الناقصة إلى الملف التنفيذي النهائي.

إن آلية الترجمة المستقلة بالشكل السابق قد تُبيّت أخطاءً لا يمكن اكتشافها إلا حين التنفيذ الفعلي للبرنامج، وذلك حين يجري التنفيذ على معاملات فعلية غير متوافقة مع معاملات التعريف (المعاملات الشكلية).

هذا ما يجري في لغات مثل C و Fortran. فهذه اللغات هي أصلاً ليست قوية التنميط (Strongly Typed Language) بعكس الحال في لغات مثل: Pascal، Ada، Modula-2، التي تتحقق، في جميع العمليات، ومنها استدعاء البرامج الجزئية، من توافق العملية مع المعاملات المؤثرة فيها. إذ إنها بهذه الميزة تضمن للمبرمج -منذ الترجمة- التوثُّق من صحة مدلول عملياته، لا من شكلها القواعدي فقط.

إن عملية توزيع البرنامج على عدة ملفات، وترجمة كلٍ على حدة مع اختبار التوافق، هي أصعب في اللغات القوية التنميط. في هذه اللغات نقسم البرنامج إلى مجتزآت أو وحدات

برمجية (Module أو Unit أو Package)، يمكن ترجمة كل وحدة على وجهٍ منفصل. هذه اللغات نسميها باللغات الوحدوية أو Modular Programming Languages. ونسمّي ترجمة الأجزاء الموزّعة على ملفات ترجمةً منفصلة Separate Compilation ، الترجمة هنا منفصلة وليست مستقلة، فعند ترجمة أي مجتزأ يجري التحقق من توافقه مع الأجزاء الأخرى التي يطلبها (والمُترجَمة قبله). سنأتي فيما يلي على شرح ماهية هذه المجتزآت أو الوحدات وآلية الترجمة المنفصلة في هذه اللغات الوحدوية.

الخواص العامة للغات الوحدوية:

تتكوّن الوحدة (unit or module) في اللغات الوحدوية من جزأين:

- الواجهة Interface
- التحقيق Realisation or Implementation

هذان الجزأن منفصلان وقد تسمح بعض اللغات، التي تضع الوحدوية في أولوياتها (مثل Ada و Modula-2) بالترجمة المستقلة لكل جزء، و يمكن أن تحوي الوحدة الجزأين في ملف واحد.

1-واجهة الوحدة:

هي تجميع للتصريحات عن الأغراض التي تتيحها الوحدة لمستخدميها (البرامج أو الوحدات الأخرى)، من هذه الأغراض: الثوابت Constants الأنماط Types المتحولات العامة (أو المشتركة) global variables والبرامج الجزئية Subprograms.

نقول إنّ هذه الأغراض (أو اسماء التعريف identifiers التي تحددها) مُصدَّرة من قِبَل الوحدة.

يقتصر التصريح عن البرامج الجزئية على ترويستها header فقط: اسم البرنامج الجزئي (ونمطه إن كان تابعاً Function) إضافةً لأسماء الوسطاء وأنماطها.

تسمح بعض اللغات الوحدوية إمكانية التصريح عن النمط Type دون تحديد مركباته (نمط خاص Private) مع تأجيل تحديد التعريف الفعلي إلى مابعد الواجهة. مما يسمح بتعريف النمط مجرداً دون تعريف مُركباته خارج الوحدة وبالتالي قصر التعامل مع متحولات النمط عبر البرامج الجزئية المُصدَّرة من قبل الوحدة.

2 -التحقيق:

يتضمن جزء التحقيق لوحدة ما، التعريفَ التفصيلي للأغراض المذكورة في الواجهة، ويُحدِّد على الخصوص نصوص البرامج الجزئية المذكورة في الواجهة.

3 -الاتصال بين الوحدات:

يمكن أن تتضمن بداية الواجهة وبداية التحقيق، قائمةَ الأغراض المستورَدة (فهي إذن مصدَّرة من قبل وحدات أخرى) للاستخدام في الوحدة (في واجهتها أوفي تحقيقها). يأخذ التصريح عن الاتصال مع (الاستخدام من) الوحدات الأخرى أشكالاً مختلفة تبعاً للّغة.

في Modula-2

(1) **FROM** ‹اسم الوحدة› **IMPORT** ‹قائمة أسماء تعريف الأشياء›;

أو

(2) **IMPORT** ‹قائمة أسماء الوحدات›;

إنّ التصريح عن استخدام وحدة (استيراد اسم الوحدة) يعني قابلية استخدام جميع الأشياء المُصرَّح عنها في واجهتها: الحالة (2)، على حين نرى في (1) تحديداً للأشياء اللازمة فقط من وحدات معينة.

يجب أن نعلم أنّ الأشياء المُعرَّفة في واجهة الوحدة مُصرَّحٌ بها تلقائياً في جزء تحقيقها.

في مترجمات لغة Pascal التي تقبل الوحدوية مثل:

IBM-PC Turbo Pascal
Mac Think's Pascal

تُستورَد واجهة الوحدة كاملة بعبارة **Uses**

Uses ‹اسم الوحدة›;.

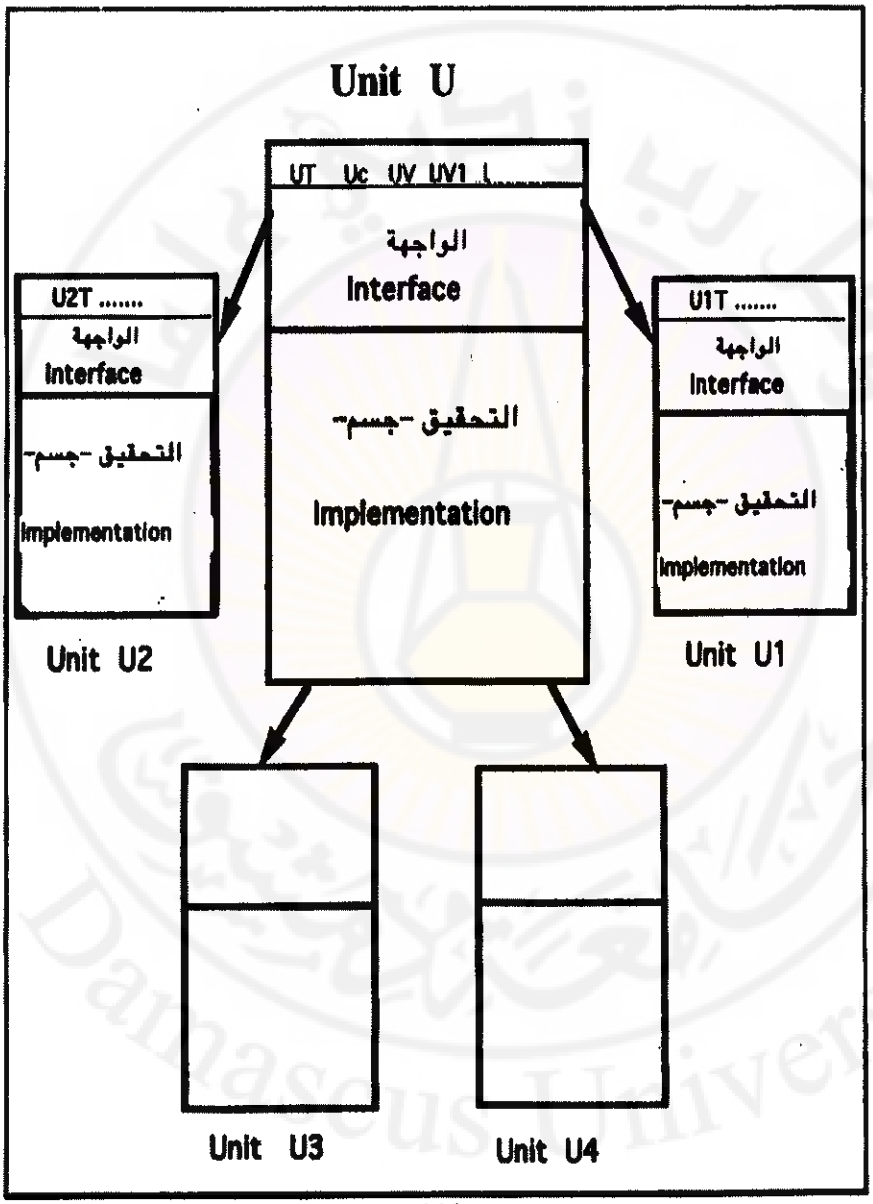
4 - أمثلة على اللغات الوحدوية:

سنعطي فيما يلي أمثلة على شكل البرنامج في اللغات الوحدوية. والغاية من ذلك بناء تصوّر جيد عن التعبير الوحدوي في البرامج.

سنأخذ مثالاً لوحدة U تستخدم أغراضاً من الوحدتين U1,U2 وتصدّر الأغراض التالية :

- ثابت UC : Constant
- نمط UT : Type
- المتحولات UV, UV1 : Variables
- الإجرائية UP : Procedure
- التابع UF : Function

أما التحقيق فيعتمد على وحدتين إضافيتين U3,U4 ، ويستخدم (داخلياً) البرنامجين الجزئيين IPU, IPF : الضروريين فقط للتحقيق الداخلي، وليس لهما استخدام خارج الوحدة. الأغراض المستوردة من الوحدتين U2،U1 تبدأ أسماؤها بالسابقة U2،U1 على الترتيب.


Unit U
UT Uc UV UV1 I
الواجهة
Interface
التحقيق -جسم-
Implementation
U2T .....
الواجهة
Interface
التحقيق -جسم-
Implementation
Unit U2
U1T .....
الواجهة
Interface
التحقيق -جسم-
Implementation
Unit U1
Unit U3
Unit U4

5- الوحدوية في لغة Pascal :

نُذكّر هنا أن لغة Pascal النظامية (Wirth Pascal) لاتتضمن البناء الوحدوي، ولايوجد أي إمكان للترجمة المنفصلة لأجزاء البرنامج، و بعبارة أخرى ينبغي وجود كامل البرنامج في ملف واحد.

المثال المعطى فيما يلي يتبع الوحدوية المتبعة في مترجمات pascal الحديثة والمنتشرة على الحواسيب الشخصية : Turbo Pascal (4 and later) on IBM-PC
Think's Pascal on Macintosh

تعتمد هذه المترجمات (إلى حدٍ بعيد) في تبنيها الوحدوية، على النموذج المعرُف بالأصل من قبَل UCSD Pascal المطوّر في جامعة سان دييغو كاليفورنيا Unisversity of Calfornia, San Diego ، والذي انتشر في البدايات الأولى للحواسيب الشخصية (أوائل الثمانينيات).

يعتمد Turbo Pascal اعتباراً من الاصدار 4 هذه الطريقة، وسنرى فيما يلي كيفية الكتابة الوحدوية في هذا المترجم للمثال العام المذكور آنفاً.

لأخذ الوحدوية بعين الاعتبار تضمن اللغة الكلمات المحفوظة الإضافية التالية:

Unit للتصريح عن الوحدة، يتبع هذه الكلمة اسم تعريف الوحدة.

Uses للتصريح عن اسم الوحدة التي سنستخدم منها أغراضاً.

Interface التصريح عن واجهة الوحدة.

Implementation التصريح عن جزء التحقيق، وينتهي هذا الجزء بـ **End.** كنهاية البرنامج.

```
Unit U;
Interface    { واجهة الوحدة U }
Uses    U1,U2 ; { U1, U2 سنستخدم أغراضاً من الوحدتين }
Const   Uc = ..... ;
Type    UT= record
            a : U1T;
            b : integer;
            .. : .....
        end;
Var     UV  : U2T;
      UV1 : UT;
procedure   UP(   p1 : UT ;  var  p2 : U1T );
function    UF(   v : UT  )  :  real;

Implementation    { تحقيق الوحدة U }
Uses    U3,U4 ; { U3, U4 سنستخدم في التحقيق أغراضاً من الوحدتين }
const     .....;
Type      .....;
Var       .....;
        .....;
procedure   IPU;
{ IPU : ذات استخدام داخلي فقط أي غير مصدرة }
begin
      U4.g(...).....;
      .....;
end; {IPU}
function    IFU  :  real ;
{ IFU : ذات استخدام داخلي فقط أي غير مصدرة }
```

```
 begin
      U3.f(...).....;
      .....;
end; {IFU}
procedure   UP {(   p1 : UT ;  var  p2 : UIT )};
{ تحقيق UP  المصرح عنه في الواجهة}
begin
      .....;
      .....;
end; {UP}
function     UF {(   v : UT  ) : real};
{ تحقيق UF  المصرح عنه في الواجهة}
  begin
      .....;
      .....;
  end; {UF}
End.
```

كتابة برنامج باستخدام وحدات:

سنقوم الآن بكتابة أحد البرامج ، التي مرت معنا، بطريقة وحدوية. سنأخذ مثالاً استخدمنا فيه عدداً من الإجرائيات: برنامج ضرب مصفوفة بشعاع. كنا قد كتبنا هذا البرنامج بطريقة أنيقة، جعلنا فيه برنامجاً جزئياً" لكل خطوة من الحل، فقد وضعنا إجرائيات لقراءة المصفوفة و كتابة المصفوفة والشعاع إضافةً إلى إجرائية الضرب.

سنرى البرنامج بشكله السابق، وكيف سنعيد توزيعه إلى ملفين ملف يحوي الوحدة التي تحوي الأنماط الأساسية والبرامج الجزئية، وملف البرنامج الأساسي الذي يستخدم الوحدة لإجراء التنفيذ.

البرنامج بالشكل التقليدي في ملف واحد:

```
Program AXBS( input, output );
(* make Matrix multiplication : A*X = B using Subprograms *)
Const  MaxN= 8;
       MaxM= 8;
Type
   MatrixNM = array[ 1..MaxN, 1..MaxM ] of real;
   Vector   = array[ 1..MaxN ] of real;
Var
   A : MatrixNM;
   X : Vector;
   B : Vector;
   N, M : integer;


Procedure ReadMat( var N,M : integer;  var A: MatrixNM );
var  l, c : integer;
begin
  writeln(' Input N, M <=',MaxN,',',MaxM);
  readln( N , M );
  writeln(' Input Matrix ',N,'x',M);
 for l:=1  to N   do  {read line of A}
   begin
     writeln('Matrix[',l,']  ',M,' elements');
    for c:=1 to M   do
         read(A[l,c]);
    readln;
   end;
end; (* ReadMat *)

Procedure ReadVect( M: integer; Var X: Vector );
var  c : integer;
begin
  writeln(' Input X ');
```

```
 for c:=1 to M   do
   begin
     writeln('Vector[',c,']');
    readln( X[c] );
   end;
end;(* ReadVect *)

Procedure MultMatVect( N,M :integer; A :MatrixNM; X :Vector;
                       Var B :Vector );
 { Multiply a Matrix by a vetor : computing A*X = B }
var  l, c : integer;
begin
 for l:=1  to N   do
   begin
     B[l] := 0;
     for c:=1 to M   do
       B[l] := B[l] +  A[l,c]*X[c];
   end;
end; (* MultMatVect *)

Procedure WriteMat( N,M : integer;  A: MatrixNM );
var  l, c : integer;
begin
 for l:=1  to N   do
   begin
     for c:=1 to M   do
         write(A[l,c]:8:2);
     writeln;
   end;
end;(* WriteMat *)

Procedure WriteVect( M: integer; Var X: Vector );
var  c : integer;
begin
 for c:=1 to M   do
    writeln( X[c]:8:2 );
end;(* WriteVect *)
```

```
Begin
 { read  A and X }
 writeln(' Matrix by Vector Multiplication A*X=B');
 writeln('-------------------------------------');
 ReadMat( N,M, A );
 ReadVect(  M, X );
 MultMatVect( N, M, A, X, B);
 { writing All togther }
 writeln('   A');
 WriteMat( N, M, A );
 writeln('   X');
 WriteVect( M, X );
 writeln('   B');
 WriteVect( N, B );
End.
```

لكتابة البرنامج السابق باستخدام الوحدات، يكفي أن نعرّف وحدة ولنسمها مثلاً MATVECT واجهتها تحوي الأنماط اللازمة، كما كانت معرفة في البرنامج ، مع إضافة ترويسات البرامج الجزئية إلى هذه الواجهة. أما التحقيق فهو نصوص البرامج الجزئية ويبقى على حاله. بعدها يمكن أن نضع البرنامج الرئيسي (المتحولات و التعليمات) في ملف منفصل، شرط أن نذكر أننا نستخدم الوحدة MATVECT.

ملف الوحدة MATVECT : matvect.pas

انتبه : يجب مطابقة اسم الملف مع اسم تعريف الوحدة.

```
Unit MATVECT;

Interface
Const   MaxN= 8;
        MaxM= 8;
Type
   MatrixNM = array[ 1..MaxN, 1..MaxM ] of real;
   Vector   = array[ 1..MaxN ] of real;

Procedure ReadMat( var N,M : integer;  var A: MatrixNM );
Procedure ReadVect( M: integer;. Var X: Vector );
Procedure MultMatVect(  N,M :integer; A :MatrixNM; X :Vector;
                        Var B :Vector );
Procedure WriteMat(  N,M : integer;   A: MatrixNM );
Procedure WriteVect( M: integer; Var X: Vector );

Implementation

Procedure ReadMat( var N,M : integer;  var A: MatrixNM );
var  l, c : integer;
begin
  writeln(' Input N, M <=',MaxN,',',MaxM);
  readln( N , M );
  writeln(' Input Matrix ',N,'x',M);
  for l:=1  to N   do  {read iine of A}
    begin
      writeln('Matrix[',l,']  ',M,' elements');
     for c:=1  to M   do
          read(A[l,c]);
     readln;
    end;
end; (* ReadMat *)
```

```
Procedure ReadVect( M: integer; Var X: Vector );
var  c : integer;
begin
  writeln(' Input X ');
  for c:=1  to M   do
    begin
      writeln('Vector[',c,']');
     readln( X[c] );
    end;
end;(* ReadVect *)

Procedure  MultMatVect(  N,M :integer; A :MatrixNM; X :Vector;
                                  Var B :Vector );
  { Multiply a Matrix by a vetor : computing  A*X = B }
var  l, c : integer;
begin
  for l:=1  to N   do
    begin
      B[l] := 0;
      for c:=1  to M   do
        B[l] := B[l] +  A[l,c]*X[c];
    end;
end; (* MultMatVect  *)

Procedure  WriteMat(  N,M : integer;   A: MatrixNM );
var  l, c : integer;
begin
  for l:=1  to N   do
    begin
      for c:=1  to M   do
          write(A[l,c]:8:2);
      writeln;
    end;
end;(* WriteMat *)
```

```
Procedure WriteVect( M: integer; Var X: Vector );
var  c : integer;
begin
 for c:=1 to M   do
    writeln( X[c]:8:2 );
end;(* WriteVect *)

End. (* Unit MATVECT *)
```

ملف البرنامج AXBS : paxb.pas

(يستخدم الوحدة MATVECT السابقة)

```
Program AXB( input, output );
(*  make Matrix multiplication :  A*X = B using Unit *)
uses  MATVECT;
Var
   A : MatrixNM;
   X : Vector;
   B : Vector;
   N, M : integer;
Begin
 { read  A and X }
 writeln(' Matrix by Vector Multiplication A*X=B');
 writeln('-----------------------------------');
 ReadMat(  N,M, A );
 ReadVect(   M, X );
 MultMatVect( N, M, A, X, B);
 { writing All togther }
 writeln('   A');
 WriteMat( N, M, A );
 writeln('   X');
 WriteVect( M, X );
 writeln('   B');
 WriteVect( N, B );
End.
```

ملاحظات عامة عن الوحدات في Turbo Pascal:

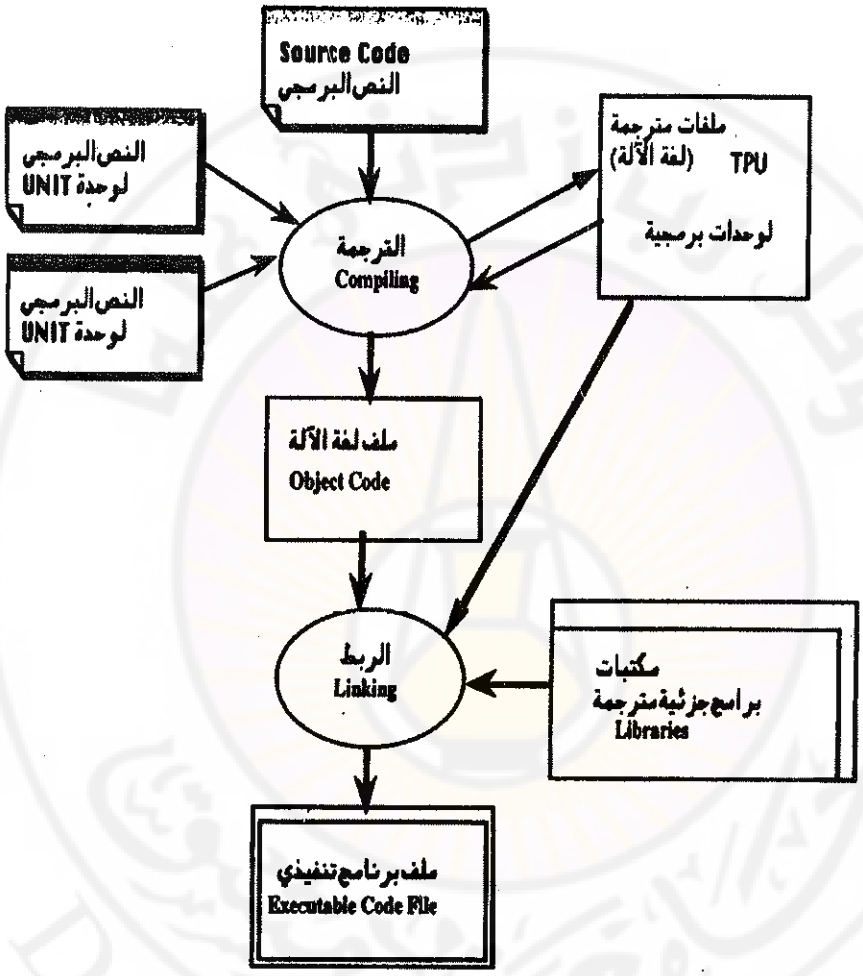
في Turbo Pascal (4 and later) يصطلح على تسمية الملف المُترجَم عن ملف نصي يحوي وحدة، باللاحقة TPU اختصاراً للكلمات Turbo Pascal Unit . ففي المثال السابق تجري ترجمة الملف matvect.pas إلى ملف matvect.tpu. بالطبع، عندما نستخدم أو نطلب أغراضاً من وحدة بعبارة **uses** يجب أن يكون هناك ملف tpu بنفس الاسم موجوداً إما في المجلد الحالي، أو في المجلد الذي نعرفه كمصدر أو مكان للبحث عن الوحدات.

إن لكل وحدة كيانها المستقل، ومن ثَم أسماء التعريف التي تقدمها مستقلة عن غيرها، وعندما نستخدم أكثر من وحدة في البرنامج يمكن أن نستخدم اسم الوحدة متبوعاً باسم تعريف الغرض المُستخدَم منها، عندما يوجد أي فرصة للالتباس، أو وجود الاسم نفسه في أكثر من وحدة. مثلاً يمكننا دوماً كتابة MATVECT.WriteMat في البرنامج السابق حين استدعائنا للإجرائية WriteVect.

وأخيراً تجدر الملاحظة أن تقنية الترجمة المنفصلة بواسطة الوحدات تتيح إمكان إغناء المبرمج لمحيطه البرمجي بمكتبة برامج جزئية تهمُّه، بحيث يستطيع استخدامها وقت يشاء دونما عناء. على سبيل المثال يزود المترجم Turbo Pascal بمجموعة كبيرة من البرامج الجزئية (إجرائيات وتوابع) لا نعتبرها مسبقة التعريف أو مُضمنة في اللغة (predefined) مثل التوابع والإجرائيات النظامية abs، sin ..... إذ إن البرامج الجزئية المتضمنة في وحدات، كل منها متخصصة في مجال معين، أن يستخدمها المبرمج بوضع كلمة **Uses**

متبوعة باسم الوحدة اللازمة. على سبيل المثال تتضمن الوحدة graph.tpu مجموعة التوابع والإجرائيات اللازمة للرسم البياني. وسنأتي في الفصل القادم إلى تعرُّف جزءٍ من محتوياتها. وبالطريقة ذاتها يمكن أن نبني الوحدات الخاصة بنا، سنقوم بذلك أيضاً حين نريد في الفصل القادم كتابة إجرائيات بيانية خاصة بأسلوبنا في تناول الرسم البياني.

ولنتذكر معاً أليات الترجمة المنفصلة باستخدام الوحدات مقارنةً بما بدأنا به الفصل حول «الترجمة المستقلة». يُنظر هنا إلى الملفات المُترجَمة سلفاً Object Files (التي تحوي ما نستدعيه أثناء الترجمة) منذ مرحلة الترجمة Compilation ولا ندع الأمر إلى مرحلة الربط Linking



الترجمة المنفصلة لأجزاء البرنامج
باستخدام الوحدات Units

الفصل الثامن
توطئة في الرسم البياني بواسطة المستقيمات

سنوسع في هذا الفصل قدرتنا على الإظهار، التي كانت مقتصرة على الشكل النصي (أحرف على الشاشة) إلى الرسم البياني. و هذا الفصل، وقد بدأنا نسير إلى ماهو أبعد من البنى الأساسية في البرمجة (بعد معالجة أساسيات البرمجة ومعرفتها باستعراض كامل لغة باسكال القياسية)، هو خطوة جيدة تتبع تعرفنا الوحدات البرمجية. سنتعرف هنا وحدات برمجية معطاة مع محيط التطوير، وسنبني عليها وحدات خاصة بأسلوبنا، ونمسك إمساكاً سليماً أساسيات الرسم البياني على نحوٍ بسيط وجميل وممتع ومفيد.

سنقدم هنا نموذجاً تعليمياً لآلية الرسم البياني المستند إلى المستقيم (القطعة المستقيمة) كركيزة أساسية بيانية، وهذا مما يتيح لنا برمجة العديد من الرسوم الهندسية الثنائية البعد (Graphics 2D : tow Dimensions).

سنقدِّم الشرح بأمثلة عبر Turbo Pascal، ولكن الأفكار والبرمجيات يمكن تحقيقها في أي بيئة برمجية أخرى دون عناء.

ونهدف بمنهجيتنا هنا، إلى تعليم "آلية تعلم" استخدام وظائف متاحة في مكتبات برمجية

وليست من أصل لغة البرمجة، وفي موضوعنا الراهن استخدام ما يخص البيانيات. وليس الهدف تلقين واستعراض كل ماهو متاح في مكتبة بيانية معينة.

جرى استخدام هذه الأمثلة في عدد من المؤسسات التعليمية، ولمسنا نجاح هذه المنهجية في الوصول السريع إلى ترسيخ المبادىء، خصوصاً مايتعلق بنظم الإحداثيات المستقلة عن الطرفي البياني، وقد انعكس ذلك على إمكان المتعلم تحقيق تطبيقات مفيدة وجميلة الاظهاردون صعوبات.

1-الواجهة البرمجية والبيانيات

لاتحتوي معظم لغات البرمجة على توابع وإجرائيات نظامية (predefined) للرسم البياني، ويعود السبب في ذلك إلى أمرين أساسيين:

- تنوع التطبيقات البيانية، وهذا مما يؤدي إلى اختلاف كبير في تحديد التوابع والإجرائيات الأساسية اللازمة وطرق تعريفها.
- تنوع الطرفيات البيانية، واختلاف إمكاناتها من حيث الدقة وطريقة تعريف الإحداثيات و عدد الألوان المتاحة....الخ، وهذا مما يضيف صعوبات إضافية في عملية التعريف النظامية للإجرائيات البيانية الأساسية اللازمة لاستيعاب إمكانات التجهيزات المختلفة.

لقد جرت محاولات لتعريف نواة أساسية من الإجرائيات اللازمة للتطبيقات البيانية الثنائية البعد، وأدت هذه الجهود إلى تعريف الأساسيات تعريفاً مستقلاً عن اللغة، ضمن مايسمى بالتقييس العالمي (International Standard Organization: ISO) لنواة نظام بياني أو

Graphical Kernel System : GKS (اعتمد التقييس GKS في منتصف الثمانينيات) وتلا هذا التقييس تعريف طريقة ربط هذا النظام ربطاً قياسياً بلغات برمجة مختلفة مثل: Fortran, Pascal, C, Ada.

وعلى أهمية هذا التقييس، من حيث إمكان برمجة التطبيقات البيانية برمجةً مستقلةً عن خصوصية الطرفيات البيانية، وعلى وجه قياسي يسمح بناقلية البرامج من وسط إلى آخر، فإن انتشاره لم يحدث على نطاق واسع جداً بسبب عموميته التي كانت تجعله نظاماً مُستهلِكاً في الحجم والحساب. لكن طريقة البرمجة البيانية التي يعرِّفها التقييس جديرة بأن تُعرَف، إذ تكوِّن إطاراً أنيقاً للبرمجة، وخصوصاً في تعريف التحويلات الهندسية والاستقلال عن إحداثيات الطرفي البياني. وفي شرحنا للبرمجة البيانية هنا نستوحي بعض الأفكار المعتمدة في ذلك التقييس.

تحاول المترجمات compilers الحالية استغلال وجود الشاشات البيانية على الحواسيب الشخصية، وتزويد المبرمجين بمجموعة من التوابع والإجرائيات اللازمة للرسم البياني، في صيغة مكتبة library أو وحدة برمجية unit.

ليس الهدف هنا دراسة جميع التوابع والإجرائيات البيانية المتاحة عبر مترجم Turbo Pascal فعددها كبير جداً، ويتجاوز ما هو متاح من إجرائيات وتوابع بلغة Wirth-Pascal القياسية.

نريد فقط أن نعطي المُبرمِج الأساسيات في الرسم البياني، وتعريفه بالخطوات الرئيسية

ليتمكن فيما بعد من أن يتعرف بنفسه مايريد، من طريق الأمثلة الموجودة مع المُساعِد Help المعطى مع المترجم.

2 -الأساسيات البيانية:

كي نتمكن من برمجة رسم شكل بياني مبسط يكفي أن يتاح لنا:

إجرائية رسم مستقيم (قطعة مستقيمة)، وتحوي معاملات هذه الإجرائية على إحداثيات المستقيم. إذن لابد من معرفة نظام الإحداثيات المتاح: أي مبدأ الإحداثيات ومجال الإحداثيات (أو الدقة) على السينات X والعينات Y.

بالطبع يمكن أن نضيف أيضاً إلى الأساسيات إجرائية كتابة نص في نقطة، وإجرائية تعريف الألوان. وبهذه الإجرائيات يمكننا استعمال البيانيات في عدد من التطبيقات.

ويجب -في معظم النظم البيانية وقبل استدعاء أي برنامج جزئي بياني- تأهيل الجهاز initialisation بإجرائية وهي بمثابة الدخول في طور الإظهار البياني، وهناك الإجرائية المعاكسة للخروج من هذا الطور.

3 -الإظهار البياني في Turbo Pascal 7:

سنعطي الآن كيفية الإظهار البياني باستخدام مترجمات Turbo Pascal/DOS (اعتباراً من الاصدار 5).

يعتمد الإظهار البياني على التوابع والإجرائيات المتضمنة في الوحدة unit المسماة graph لذا لابد لبرنامج الإظهار البياني من العبارة ;Uses graph.

3-1-نظام الإحداثيات:

مبدأ الإحداثيات في الزاوية العليا اليسارية من الشاشة، أي إن محور السينات موجه من اليسار إلى اليمين ومحور العينات من الأعلى إلى الأسفل.

3-2-مجال الإحداثيات:

يمكن للعديد من الشاشات البيانية المتاحة على الحواسيب الشخصية IBM-PC إعطاء مجالات إحداثيات مختلفة وهذا مانسميه الدقة resolution. والدقة الأكثر استخداماً في البرمجة هي الدقة المتوافقة مع البطاقة البيانية EGA والتي تتيح 480 نقطة على المحور Y و 640 على المحور X، نسمي نقطة على الشاشة «عنصر صورة» أو "عنصورة" PICture ELement أو PIXEL.

3-3-تعلم البيانيات بتطبيق الأمثلة:

سنحاول عرض عمل الإجرائيات الأساسية البيانية بطريقة تسمح بتعلم آلية عمل الإجرائيات الأخرى من قبل المتعلم بمفرده.

يمكن أن نطلب المساعدة أو الشرح Help على graph ثم نستفسر عن آلية عمل line . ونرىضمن الشرح أن هناك مثالاً كاملاً، نقوم بنسخه إلى ملف ثم نقوم بتنفيذه وتعُرف آلية العمل عبر بالتطبيق العملي.

والمثال المذكور هو التالي (بعد أن أضفنا ترويسة البرنامج):

```
program TryLine;
uses Crt, Graph;
var Gd, Gm: Integer;
begin
 Gd := Detect;
 InitGraph(Gd, Gm, ' ');
 if GraphResult <> grOk then
  Halt(1);
 Randomize;
 repeat
  Line(Random(200), Random(200), Random(200), Random(200));
 until KeyPressed;
 Readln;
 CloseGraph;
end.
```

إن تنفيذ هذا البرنامج يؤدي إلى تكرار إظهار مستقيمات عشوائية في الزاوية العليا اليسارية (المجال [0,200]x[0,200]) حتى يقوم المستخدم بضغط أحد المفاتيح.

ترى في هذا المثال ،على صغره، العديد من أسماء التعريف identifiers التي لم يسبق للعارف بلغة pascal القياسية تَعَرُّفها (ولم تكن الحاجة موجودة فهي غير قياسية).

من الهام جداً ملاحظة استخدام الوحدة البيانية: uses Graph; وسنشرح كل الأسماء المبهمة:

Detect ثابت معرَّف في الوحدة Graph

InitGraph إجرائية مُعرَّفة في الوحدة Graph مهمتها التأهيل للإظهار البياني

GraphResult متحول عام معرَّف في الوحدة Graph

grOK ثابت معرَّف في الوحدة Graph

Halt إجرائية مسبقة التعريف في المترجم Turbo Pascal مهمتها إيقاف البرنامج

إن مهمة أسطر البرنامج حتى الآن (من السطر 4-1 في جسم البرنامج) هي التأهيل البياني، مع مناقشة الحالات المختلفة أثناء التأهيل. لذا يمكننا للاستخدامات البسيطة أخذ التعليمات كما هي دون تبديل.

Randomize إجرائية مسبقة التعريف في المترجم Turbo Pascal مهمتها تأهيل توليد أعداد عشوائية.

Random تابع مسبق التعريف مع المترجم Turbo Pascal يقوم بإعطاء أو توليد عدد عشوائي صحيح موجب ضمن مجال (في حالتنا هنا 200).

Line وهي الإجرائية الأساسية لنا في الوحدة Graph، وتقوم برسم قطعة مستقيمة.

KeyPressed تابع معرَّف ضمن الوحدة Crt يعطي القيمة true عندما يضغط المستخدم أحد المفاتيح.

CloseGraph إجرائية معرفة في الوحدة Graph مهمتها إنهاء الإظهار البياني.

بعد أخذ هذا المثال يمكنك التبديل فيه، بأخذ مجال الأعداد العشوائية بصيغة مختلفة مثلاً.

أتاح لنا المثال السابق تعرُّف ثلاث أساسيات بيانية لاغنى عنها:

InitGraph
Line
CloseGraph

هل لاحظت أنك تستطيع استيعاب البنية العامة للبرنامج، فالجديد هو أسماء تعريف خاصة، لكن لا شيء جديد فيما يخص لغة البرمجة.

ملاحظة عملية هامة:

ينبغي وجود الملف EGAVGA.BGI في مجلد تنفيذ البرنامج لتتمكن من الإظهار البياني.

سنأخذ مثالاً آخر هو استخدام إظهار نص:

بطلب الشرح Help حول إجرائية OutText نحصل على المثال (العينة) التالي:

```
program TryOutText;

{Sample code for the OutText procedure.}

uses Graph;

var Gd, Gm: Integer;
begin
 Gd := Detect;
 InitGraph(Gd, Gm, '');
 if GraphResult <> grOk then
  Halt(1);
 OutText('Easy to use');
 ReadIn;
 CloseGraph;
end.
```

يقوم البرنامج بإظهار العبارة Easy to use في أعلى الشاشة (الموقع الحالي هو (0,0)). يمكن أن نختبر البرنامج لإظهارها في موقع مختلف بأن نغير الموقع الحالي بالإجرائية MoveTo، يمكن أن نسبق عملية الإظهار بالكتابة: MoveTo(100, 200) فتظهر العبارة بالموقع 100,200 من الشاشة.

3- 4-بعض الأساسيات الإضافية:

البرنامج التالي مشابه للمثال البياني الأول، مع بعض الإضافات التي تسمح لنا بالإظهار على كامل مجال الإحداثيات مع تنويع الألوان.

العمليات الإضافية هي التالية:

GetMaxX، GetMaxY: توابع معرفة في الوحدة Graph تسمح لنا بالحصول على مجال الإحداثيات على X ، Y على الترتيب.

SetColor : إجرائية معرفة في الوحدة Graph تسمح لنا بتحديد لون الإظهار.

SetViewPort: إجرائية معرَّفة في الوحدة Graph تسمح لنا بتحديد مستطيل الإظهار البياني على الشاشة. وبوضع المعامل الأخير ClipOn يُقطع كل إظهار تتجاوز إحداثياته حدود هذا المستطيل، على حدود المستطيل التي نسميها غالباً بوابة الرؤية أو Viewport.

```
Program GExample1;
Uses Graph, Crt;

Var
Gd, Gm: Integer;
Xmax, Ymax : integer;
```

```
Begin
(*********************************)
(**          Graphic Mode Initialisation   **)
 Gd := Detect;
 InitGraph(Gd, Gm, ' ');
 if GraphResult <> grOk then
   Halt(1);
(*********************************)
(**                  Drawing                  **)
 Xmax :=GetmaxX; Ymax := GetMaxY;
  SetViewPort(0, 0, Xmax, Ymax,ClipOn);
Randomize;
repeat
  SetColor( Random(16) );
  Line(Random(Xmax), Random(Ymax), Random(Xmax),
Random(Ymax));
until KeyPressed;

(*********************************)
(**       End of Graphic Mode          **)
  CloseGraph;
End.
```

مثال 1 : رسم تابع قطبي (الفراشة)

لرسم أي تابع يكفي أن نحسب قيم التابع لعدد كاف من النقاط من أجل عدد من المتحولات، تمثل المتحولات الإحداثيات X للشكل، وقيم التابع الإحداثيات Y له. ونصل بين هذه النقاط (x,y) -على التتالي- للحصول على المنحني.

رسم منحني تابع:

معطيات المسألة:

التابع: y = G(x)

مجال الرسم : x0, xn

عدد النقاط المطلوبة لرسم المنحني هو N

الخوارزمية العامة

dx := (xn-x0)/N (dx هو خطوة الرسم على المحور X)

x1 := x0

y1 := G(x0)

من أجل i=1 حتى N كرر

| x2 := x0 + i * dx

| y2 := G(x2)

| رسم القطعة المستقيمة (بين النقطتين) (x2,y2)------ (x1,y1)

| إسناد النقطة الثانية إلى النقطة الأولى أي: x1 := x2, y1 :=y2

ماذا يلزمنا للرسم البياني هنا ؟

يلزمنا فقط تحديد مجال إحداثيات الشكل، أي المستطيل الحاوي نقاط التابع، لنعرف معاملات التحويل، أي نقل الإحداثيات من مجال التطبيق (مجال منحني التابع هنا) إلى مجال إحداثيات الجهاز، وسنسمي هذا المستطيل بالنافذة في شرحنا لتحويل الإحداثيات.

لرسم تابع قطبي: R = F(U) حيث R نصف القطر و U هي الزاوية، ننتقل إلى الشكل الديكارتي المذكور آنفاً، باستخدام العلاقة:

$x = F(U) * \cos(U)$ و $y = F(U) * \sin(U)$. أما آلية الرسم البياني فمشابهة.

التابع القطبي F(U) المُعرَّف في البرنامج التالي يرسم شكلاً جميلاً جداً يشبه الفراشة.

$$F(u) = \exp(\cos(2*u)) - 2*\cos(4*u) + 0.3*\cos(16*u) + 1.5*(\sin(-u/16))^3$$

سندع التعليمات المسؤولة عن التأهيل البياني ضمن إجرائية نسميها GUinitgraph، ثم نحسب إحداثيات نقاط التابع القطبي، ثم نضرب بنسبة تكبير، ونقوم بانزياح مناسب للحصول على إحداثيات ضمن مجال الشاشة (لاحظ الصعوبة في استخدام الإحداثيات وخصوصاً أن المحور Y يتجه نحو الأسفل!!).

```
Program Papi;
Uses Graph;
Var
 u , u0, du, R  : real;
 x1, y1, x2, y2  : real;
 ix1, ix2, iy1, iy2 :integer;
 i : integer;

Procedure GUinitgraph;
var
 grDriver, grMode, ErrCode, Detect: Integer;
begin
 grDriver := Detect;
 InitGraph(grDriver, grMode,' ');
 ErrCode := GraphResult;
 if ErrCode <> grOk then
    begin
      WriteLn('Graphics error:',
      GraphErrorMsg(ErrCode));
      Halt(1);
    end;
end; (* GUinitGraph *)
```

```
Function F( u : real ) :real;
var  sinu16 : real;
begin
  sinu16 := sin(-u/16);
  F :=  EXP(COS(2*u)) - 2*COS(4*u) + 0.3*COS(16*u)
       + 1.5*sinu16*sinu16*sinu16;
end;(* F(u) *)

Begin (*  program body *)
  GUinitgraph;
  du := 0.034;
  u0 := 0;
  R := F(u0);
  x1 := R * COS(u0);
  y1 := R * SIN(u0);
  for i := 2 to  2500  do
  begin
    u := u0 + i * du;
    R := F(u) ;
    x2 := R * COS(u);
    y2 := R * SIN(u);
    ix1 := round(x1*40);  ix2 := round(x2*40);
    iy1 := round(y1*40);  iy2 := round(y2*40);
    (* Resolution: x:0..639, y:0..479 *)
    line(320+ix1,240-iy1, 320 + ix2,240-iy2);
    x1 := x2 ;
    y1 := y2 ;
  end;
  readln;
CloseGraph
End.
```



Damascus University

4-تقنيات الإظهار البياني بطريقة مستقلة عن الجهاز البياني:

نلاحظ مما سبق سهولة تعرُّف إمكانات الإجرائيات البيانية المزودة مع المترجم (إجرائيات الوحدة البيانية). إلا أنه يجب التغلب على مشاكل هامة حتى نتمكن من استخدام البيانيات لتطبيقاتنا الخاصة، أهمها استخدام جملة الإحداثيات المناسبة لتطبيقاتنا، وعدم الاضطرار إلى "الالتفاف" كل مرة لنوفق بين الإحداثيات التي نريد وإحداثيات الجهاز البياني.

سنعرض فيما يلي أسلوباً مبسطاً يُجرِّد العمليات البيانية الأساسية، ويسهل التعبير عن إحداثيات الأشكال التي نريد.

للانتقال من نظام إحداثيات (جملة ديكارتية متعامدة مباشرة) إلى آخر، لابد من عملية تقييس (تصغير أو تكبير) وعملية انسحاب. إذن للانتقال من إحداثيات التطبيق إلى إحداثيات الجهاز، لابد من تلك العمليات. سنختصر هذه العمليات بتعريف إجرائية تحويل، تحدِّد معاملات هذا الانتقال بأسلوب مبسط. وذلك بتحديد مستطيل يغلف إحداثيات التطبيق نسميه **Window** أو**النافذة** وتحديد مستطيل يعرِّف مكان إظهار الشكل على الشاشة نسميه **Viewport** أو**بوابة الرؤية**.

بعد ذلك يمكن أن نعبِّر عن إحداثيات الشكل الفعلية (إحداثيات التطبيق) تاركين جانباً خصوصية إحداثيات الشاشة.

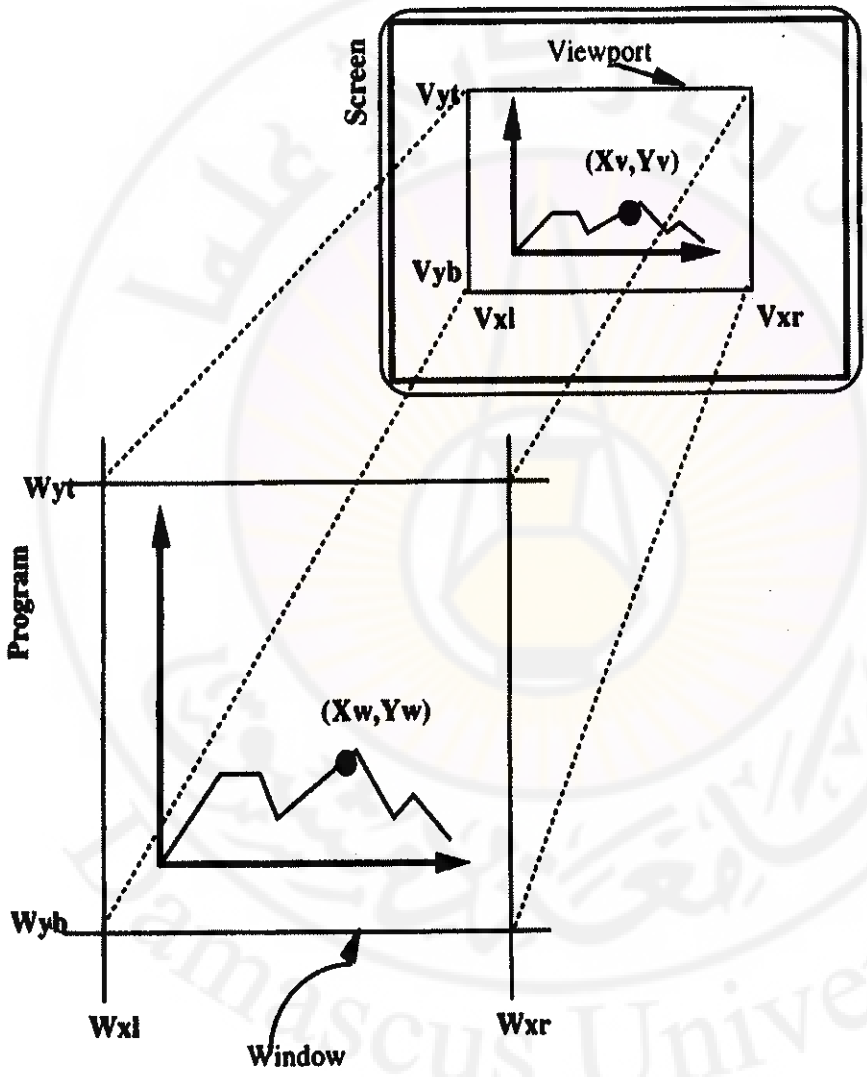
أما آلية تحقيق الإنتقال من نظام إحداثيات إلى آخر، المبني هنا على تحويل النافذة (التحويل من مستطيل إلى آخر)، فيجري بعلاقة تناسب بسيطة (انظر الشكل):

$$(Xv-Vxl)/(Vxr-Vxl) = (Xw-Wxl)/(Wxr-Wxl)$$
$$(Yv-Vyb)/(Vyt-Vyb) = (Yw-Wyb)/(Wyt-Wyb)$$

تقول العلاقة: إن نسبة مسافة نقطة عن بداية المستطيل (في محور السينات)، على طول المستطيل في النافذة، تساوي نسبة مسافة مسقط النقطة عن بداية المستطيل(في محور السينات)، على طول المستطيل في بوابة الرؤية. والكلام نفسه عن النسبة بين مسافة العينات على الارتفاع.

هذه الصيغة هي التي تعطي علاقة حساب إحداثيات (بوابة الرؤية Viewport) أي إحداثيات الجهاز (الشاشة): Xv,Yv بدلالة إحداثيات (النافذة Window) أي إحداثيات التطبيق : Xw,Yw . وهذا الحساب يُغلَّف ويُضمَّن في الوحدة البرمجية المبسطة التي ننشئها ونسميها GU.

القاعدة العامة للتحويل من نافذة Window في البرنامج إلى بوابة رؤية Viewport على الشاشة



سنستخدم إحداثياتنا الخاصة عن طريق بناء وحدة برمجية unit خاصة بذلك، نسميها GU نضمّن فيها الأساسيات اللازمة للرسم البياني التي تستخدم إحداثياتنا الخاصة بنا.

```
Unit GU;

Interface

 Procedure GUInitgraph;

 Procedure GUCloseGraph;

 Procedure GUWindowView(wxb,wyb, wxt, wyt: real;
                        vxt,vyt, vxb, vyb: integer);
(* (wxb,wyb) bottom left,  (wxt, wyt) top      right  of Window  *)
(* (vxt,vyt)   top    left,  (vxb, vyb) bottom right  of Viewport *)


Procedure GUline( x1, y1, x2, y2 : real );

Procedure GUText( x,y:real; S: String);

Procedure GUSetColor( c : integer );
```

تقتصر واجهة هذه الوحدة على الإجرائيات التالية:

GUInitGraph إجرائية التأهيل للإظهار البياني.

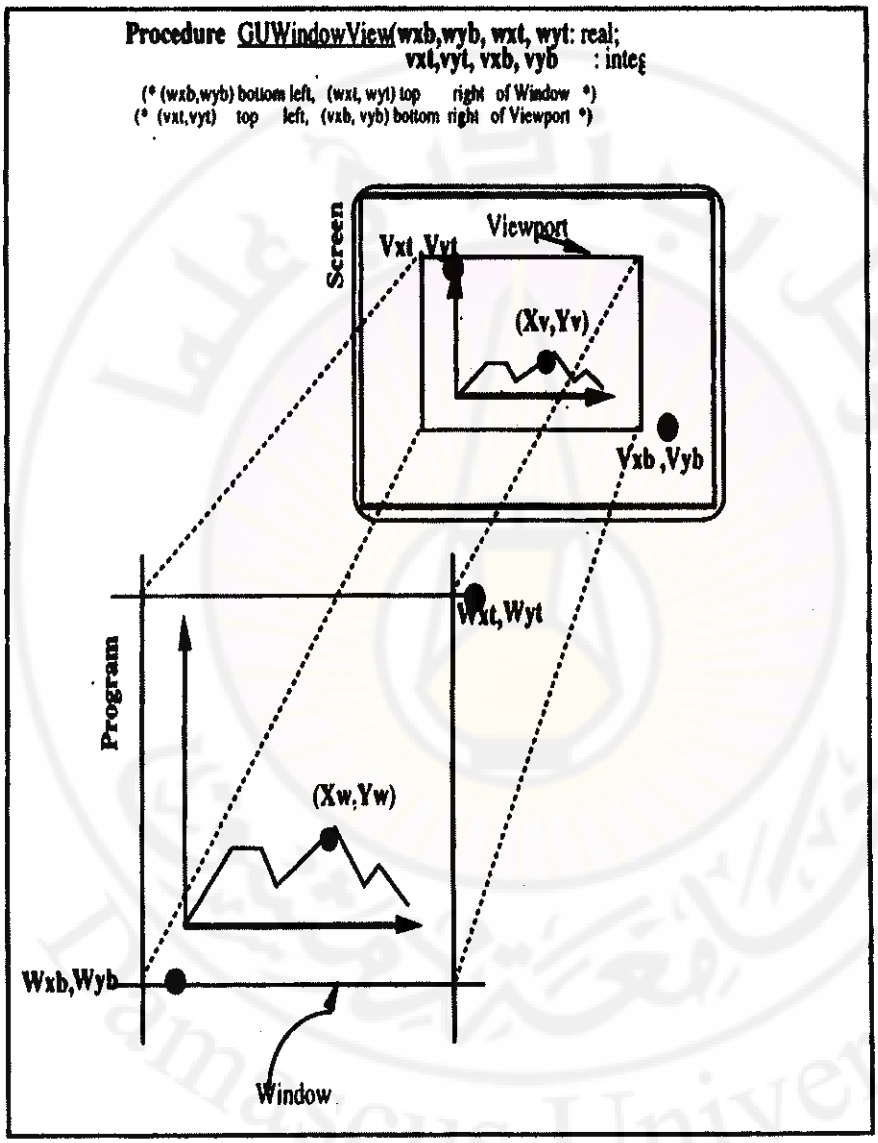
GUCloseGraph إجرائية إنهاء الإظهار البياني.

GUWindowView إجرائية تحديد معاملات التحويل من إحداثيات التطبيق إلى إحداثيات الشاشة، معاملات الإجرائية:

حدود المستطيل الحاوي لإحداثيات التطبيق، ويحدد بنقطتين: (wxb,wyb) الزاوية السفلى من اليسار و (wxt, wyt) الزاوية العليا من اليمين، وحدود المستطيل في الشاشة حيث نريد الإظهار ويحدد بنقطتين:(vxt,vyt) الزاوية العليا من اليسار و (vxb,vyb) الزاوية السفلى من اليمين. (أنظر الشكل).

GULine إجرائية إظهار قطعة مستقيمة، ومعاملاتها هي إحداثيات نقطتي القطعة: ويُعبَّر عن الإحداثيات هنا، بإحداثيات التطبيق أي أنها محتواة في المستطيل:
wxb,wyb, wxt, wyt .

GUText إظهار العبارة النصية S في إحداثي التطبيق x,y.


Procedure GUWindowView(wxb,wyb, wxt, wyt: real;
vxt,vyt, vxb, vyb : integ
(* (wxb,wyb) bottom left, (wxt, wyt) top right of Window *)
(* (vxt,vyt) top left, (vxb, vyb) bottom right of Viewport *)
Screen
Vxt ,Vyt
Viewport
(Xv,Yv)
Vxb ,Vyb
Wxt,Wyt
Program
(Xw,Yw)
Wxb,Wyb
Window

مثال 2 : رسم تابع قطبي (الفراشة) باستخدام إحداثيات التطبيق

لرسم التابع القطبي F(U) المعرف آنفاً:

$$F := \exp(\cos(2*u)) - 2*\cos(4*u) + 0.3*\cos(16*u) + 1.5*(\sin(-u/16))^3$$

يكفي أن نعرف حدود هذا التابع لنعرف قيم التحويل، أو بتعبير أدق، مستطيل إحداثياتنا، وهي المعاملات الأولى لإجرائية التحويل GUWindowView . ولمّا كانت قيم sin و cos محصورة في المجال [1,+1-] يمكن أن نستنتج أن التابع لا يتجاوز القيمة:

1.5 + 0.3 + 2 + 2.7 أي 6.5. وعلى ذلك نحدد التحويل بالإجرائية:

GUWindowView بنافـــــــــــــــذة [-6.5,+6.5]x[-6.5,+6.5].

```
Program papi_GU;

Uses GU;

Var
u , u0, du, R : real;
x1,y1,x2,y2 :real;
i : integer;

Function F(U:real):real;
{** The Polar Function **}
var  sinu16 : real ;
begin
  sinu16 := sin(-u/16);
  F := exp(cos(2*u))  - 2*cos(4*u) + 0.3*cos(16*u)
     + 1.5*sinu16*sinu16*sinu16;
end;(* F *)
```

```
Begin
(***********************************)
(**       Graphic Mode Initialisation      **)
  GUInitGraph;
  GUWindowView( -6.5, -6.5, 6.5, 6.5 , 50,50, 450, 450);

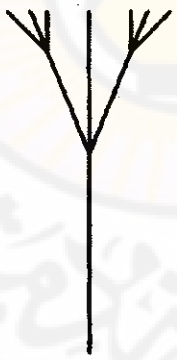
(***********************************)
(**                Drawing                 **)
  du := 0.03;
  u0 := 0.0;
  R := F(u0);
  x1 := R*cos(u0);
  y1 := R*sin(u0);
  For  i:=2 to 2500 do
    begin
       u :=
      u0 + i*du;
      R := F( u );
      x2 := R*cos(u);
y2 := R*sin(u);
     GUline(x1,y1,x2,y2);
      x1 := x2; y1 := y2;
    end;
readln;

(***********************************)
(**       End of Graphic Mode             **)
  GUCloseGraph;
End.
```

مثال 3 : الرسم التقطيعي fractal (الشجرة)

يمكن أن نرسم الكثير من الأشكال في الطبيعة بتكرار نفس النموذج بمقاييس مختلفة، مثلاً يمكن تخيل الشجرة بأنها مجموعة أغصان بشكل معين، تتكرر بشكل مصغر (كفروع) أكثر من مرةلتعطي الشكل الكلي.

يرسم البرنامج التالي شجرة باعتماد هذا التكرار بأسلوب برمجة عودية recursive، وصولاً إلى شكل متكامل للشجرة، اعتماداً على تكرار مجموعة أغصان. لاحظ تفرع الشجرة من أجل تكرارين فقط. أما عندما يزداد عدد التكرار ويتناقص طول الغصن، فنحصل على مايشبه شكل الأوراق.



```
Program tree_GU;

Uses GU;

Procedure tree( ng : integer;  l, ti :real );
 var a,px,py : real;

 procedure branch( ox, oy : real; nr : integer; l,t :real );
  var  Px1,Px2,Py1,Py2 : real;
   begin
     if ( (nr<>0) and ( l >0) ) then
     begin
        Px1 := ox;   Py1 := oy;
        Px2 := ox + l*cos(t);
        Py2 := oy + l*sin(t);
        nr := nr - 1;
         GULine( Px1,Py1, Px2,Py2);
        branch( px2,py2, nr, l-l*0.45, t+ti);
        branch( px2,py2,  nr, l-l*0.15, t   );
        branch( px2,py2, nr, l-l*0.45, t-ti);
     end;
  end;(* branch *)
begin
   a := pi/2;
   px := 0.5;  py := 0;
   branch(px, py, ng, l, a )
end;(* tree *)

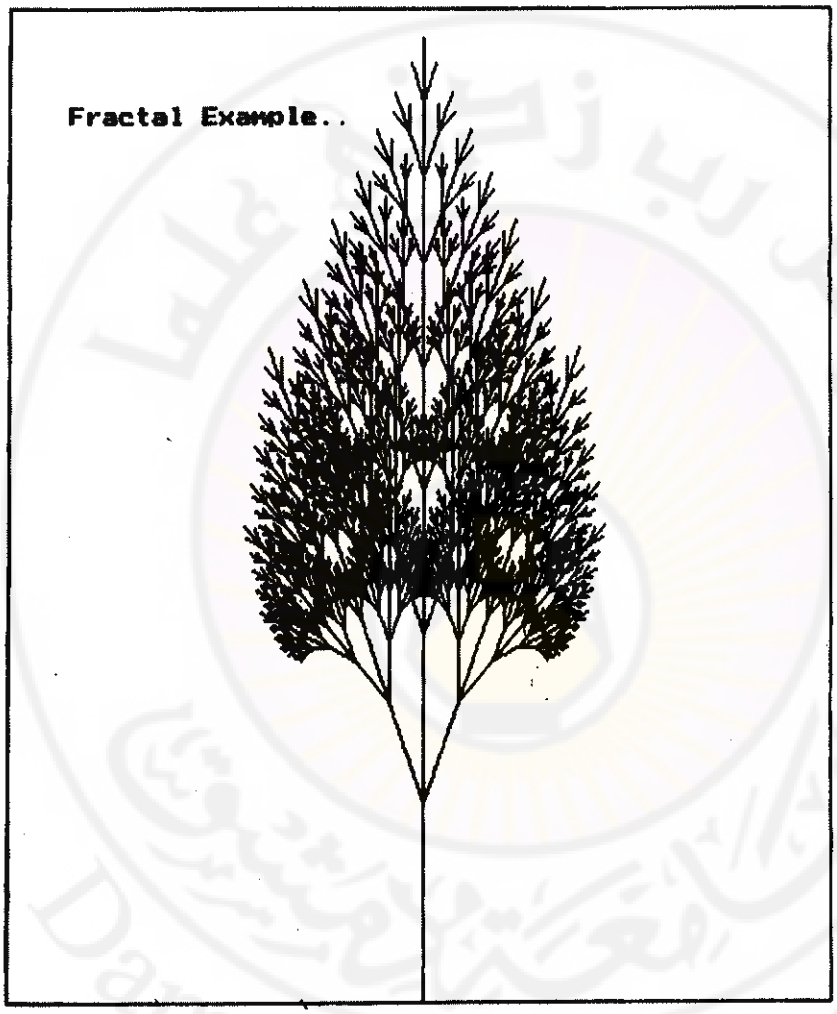

Begin
(*********************************)
(**     Graphic Mode Initialisation        **)
  GUInitGraph;
  GUWindowView( 0, 0, 1, 1 , 50,50, 450, 450);
```

```
(*********************************)
(**              Drawing          **)

    tree(8, 0.2, pi/8 );
    GUText(0.05 ,0.9, ' Fractal Example..');
    readln;
(*********************************)
(**          End of Graphic Mode          **)
  GUCloseGraph;

End.
```

Damascus University


Fractal Example..

مثال 4 : الرسم على نوافذ متعددة (عدة إسقاطات) في الشاشة

يمكننا أن نستخدم أكثر من تحويل إسقاط (عددة بوابات رؤية Viewport) على الشاشة.

المثال التالي يرسم منحني التابع القطبي: ;F(W,U)=(1+cos(W*U))

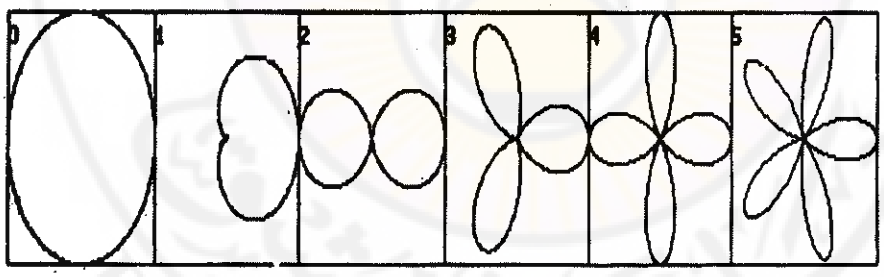
U هي الزاوية.

F هو المطال.

W وسيط نعطيه في مسألتنا القيم 0 حتى 5.

يرسم لنا التابع القطبي السابق وردةُ عددُ بتلاتها هو W.

نريد أن نرسم التابع لكل قيم W السابقة معاً على الشاشة.



إحداثيات المنحني (هي قيمة التابع ;(1+cos(W*U)) مضروبةً بـ Sin أو Cos) محصورة في المجال [2+,2-]x[2+,2-].

هذا المجال يعرِّف نافذة التحويل.

خوارزمية الرسم على عدة بوابات رؤية كل منها خاص بقيمة محددة W:

من أجل W تساوي 0 حتى 5 كرر

```
{ ضع التحويل ذا بوابة الرؤية(w*100,100),(100+ w*100, 200) }
{مستطيلات بوابة الرؤية: موقع Y بين 100 و 200 ،
أما X فتنزاح بمقدار 100 لكل تكرار }
{ F(W,U)=(1+cos(W*U)) ارسم منحني التابع القطبي}
{ ارسم قيمة W في الزاوية العليا للمستطيل}
```

آلية البرمجة مشابهة للسابق، الجديد هو تغيير التحويل عدة مرات ورسم التابع مع w مختلفة في كل تحويل، والجديد أيضاً استخدام Str لتحويل عدد إلى سلسلة حرفية لنتمكن من إظهار "نص" الأرقام بواسطة GUText

```
Program MultiWin;
Uses GU;
Var
u , u0, du, R,pi : real;
x1,y1,x2,y2 :real;
i,w,N : integer;
stw : string;

Function F(W,U:real):real;
{** The Polar Function **}
begin
  F:=(1+cos(W*U));
end;(* F *)
```

```
Begin
 GUInitGraph;
 pi :=3.1416;  du:= ((2*pi)/5.0)/100.0;
 for w:=0 to 5 do
  begin
   GUsetColor(w+1);
   GUWindowView(-2, -2, 2, 2 , w*100,100,100+ w*100, 200);
  str(w, stw);  (* convert number w to string  stw *)
  GUText(-2,1.75,stw);  (* draw string *)
  u0 := 0.0;
  R := F(w, u0);
  x1 := R*cos(u0);
  y1 := R*sin(u0);
  For  i:=2 to 500 do
   begin
     u := u0 + i*du;
      R := F( W,u );
      x2 := R*cos(u);
      y2 := R*sin(u);
      GUline(x1,y1,x2,y2);
     x1 :=x2; y1 := y2;
   end;
end; (* for w *)
readln;
  GUCloseGraph;
End.
```

5-النص البرمجي الكامل للو حدة البيانية المبسطة للرسم GU:

```
Unit GU;

Interface

 Procedure GUInitgraph;

 Procedure GUCloseGraph;

 Procedure GUWindowView(wxb,wyb, wxt, wyt: real;
                        vxt,vyt, vxb, vyb: integer);
(* (wxb,wyb) bottom left,  (wxt, wyt) top     right  of Window   *)
(* (vxt,vyt)   top    left,  (vxb, vyb) bottom right  of Viewport *)

 Procedure GUline( x1, y1, x2, y2 : real );

 Procedure GUText( x,y:real; S: String);

 Procedure GUSetColor( c : integer );


Implementation

uses graph ;

Var
UWXB,UWXT,UWYB,UWYT : real;
UVXB,UVYB : integer;
UMAXY :integer;
alpha, beta : real;
```

```
Procedure GUinitgraph; .
var
 grDriver, grMode,ErrCode : Integer;
begin
grDriver := Detect;
InitGraph(grDriver, grMode,' ');
   ErrCode := GraphResult;
   if ErrCode <> grOk then
     Begin
     WriteLn('Graphicserro:', GraphErrorMsg(ErrCode) );
     Halt(1);
    End;
end; (* GUinitGraph *)


procedure GUCloseGraph;
begin
  closegraph;
end; (* GUInitGraph *)


procedure  GUWindowView(wxb,wyb, wxt, wyt: real;
              vxt,vyt, vxb, vyb: integer);
begin
    alpha :=  (vxb - vxt)/(wxt-wxb);
    beta  :=  (vyb - vyt)/(wyt-wyb);
    UMAXY :=GetMaxY;
    UVXB := vxt;
    UVYB := UMAXY-vyb;
    UWXB := wxb;   UWXT := wxt;
    UWYB := wyb;   UWYT := wyt;
    rectangle(vxt,vyt, vxb, vyb);
end;(* WindowView *)
```

```
Procedure GULine( x1, y1, x2, y2 : real );
var ix1,ix2,iy1,iy2 :integer;
begin
   ix1 := round(+UVXB + (x1-UWXB)*alpha ) ;
   ix2 :=  round(+UVXB + (x2-UWXB)*alpha ) ;
   iy1 := round( UVYB+ (y1-UWYB)*beta );
   iy2 := round( UVYB+ (y2-UWYB)*beta );
   line( ix1,UMAXY-iy1, ix2,UMAXY-iy2);
end;(* Uline *)


Procedure GUText( x,y:real; S: String);
var ix,iy :integer;
begin
     ix := round(UVXB+ (x-UWXB)*alpha );
     iy := round( UVYB+ (y-UWYB)*beta );
     MoveTo(ix,UMAXY-iy);
     OutText(S);
end; (* GUText *)


Procedure GUSetColor( c: integer );
begin
  SetColor( c );
end; (* GUSetColor *)
End.
```

الفصل التاسع

البرمجة الغرضية التوجّه

في لغة باسكال

١-مفهوم الغرض:

تعتبر البرمجة الغرضية التوجّه أحد التوسّعات التي أضيفت إلى لغة باسكال القياسية، ذلك أن أهم مشكلة تواجهنا في البرمجة المهيكلة التي استخدمناها حتى الآن، هي مشكلة متطلبات الزبون المتغيرة، سواء أكان هذا التغيّر يحدث خلال مرحلة تطوير البرنامج أم خلال مدة استثماره. وهذا يعني أن الزبون قد يرغب في إضافة وظائف جديدة، أو يعدل آلية عمل وظائف كانت موجودة سابقاً. وهنا يجب على المسؤولين عن تطوير النظام البرمجي أن يسعوا لتنفيذ هذه الوظائف دون الإخلال بما أنجز سابقاً. وقد تبيّن للعاملين في حقل البرمجة أن التغيّر يطرأ غالباً على الوظائف على حين تبقى المعطيات مستقرة، أو تتطور باتجاه إضافة بعض الحقول دون تغيير جذري فيها. وقد رأينا في الفصل الثاني (التسجيلات) أننا نستطيع أن نعرّف بنية معطيات، تسمى تسجيلة، تتضمن عدة حقول تصف في مجموعها كائناً برمجياً. والتسجيلة هنا هي التي تمثّل المعطيات المستقرة، أما العمليات التي تخضع لها فهي عمليات قابلة للتغيّر تَبَعاً لطبيعة المعالجة المرغوب في تنفيذها.

ولإيضاح ماسبق نأخذ المثال التالي:

```
Type    Employee= record
                  name: string;
                  birthdate: record
                             day, month, year: integer;
                             end;
                  address: string;
                  salary: real;
                  end;
```

جرى اختيار الحقول المذكورة آنفاً في توصيف الكائن البرمجي "موظف" باعتبار أن هذه الحقول هي التي توصّف كل موظف في النظام البرمجي، وهي التي ستخضع للمعالجة. ولكن هذه المعطيات التي تصف الموظف لاتكفي لوصف آلية المعالجة التي سيخضع لها ضمن النظام البرمجي. فللتعامل مع الموظف لا بدّ من كتابة مجموعة عمليات (توابع أو إجرائيات) تسمح بإعطاء قيم للحقول، أو قراءتها أو تعديلها، أو إجراء حسابات عليها. فبنية المعطيات هنا بنية "منفعلة" لا تحتوي ضمناً على مايدل على العمليات. ولو نظرنا إلى الأشياء من حولنا، لوجدنا أن لها ردود أفعال ذاتية تجاه مايطلب منها. فمثلاً للتعامل مع طابعة نرسل لها أمراً لطباعة ملف فتطبعه بنفسها. فلماذا لانحاول أن نفعل الشيء نفسه مع بنى المعطيات في باسكال ونجعلها فاعلة لا منفعلة؟

إن مفهوم بنى المعطيات الفاعلة يقودنا إلى مفهوم الغرض **Object** و البرمجة الغرضية التوجّه. وسنشرح ذلك عن طريق مثال يوضّح المفاهيم السابقة ويبيّن الفرق بين الآلية التي كنّا نتعامل بها مع بنى المعطيات في البرمجة المهيكلة، وما سنضيفه في البرمجة الغرضية التوجّه.

لنأخذ بنية المعطيات الممثّلة لطالب، وهي تسجيلة تتضمن الحقول التالية: حقلين للاسم

واللقب وحقل لعلامة التخرج. للتعامل مع التسجيلة لا بدّ من بناء إجرائيات لإعطاء قيم ابتدائية للحقول، وإدخال قيم الحقول لطالب معين، وإظهار معطيات طالب على الشاشة، وإذا كانت علامة التخرج لطالب، أكبر من 70٪ يُبنى ملف يتضمن رسالة تهنئة للطالب.

في البرمجة المهيكلة يبنى البرنامج حول التسجيلة المسماة studentrec وتكتب الإجرائيات التي تسمح بالتعامل معها:

```
Program  Department;
Type
        string15=String[15];
        Studentrec= Record
                        fname,
                        lname: string15;
                        average: real;
                        End;
Var  student: Studentrec;

Procedure Init(var student:Studentrec);
 Begin
  With student do
   begin
     fname:=' ';
      lname:=' ';
     average:=0.0;
   end
 end;

Procedure FillRecord(var student:Studentrec);
 begin
  With student do
    begin
        Write('Enter the student''s first name: ');
        Readln(fname);
        Write('Enter the student''s last name: ');
```

```
        Readln(lname);
        Write('Enter the student''s current average: ');
        Readln(average);
    end
end;

Procedure SendHonorsLetter( student:Studentrec);
var
  letterfile:text;
  counter:integer;
begin
  assign(letterfile,'prn');
  rewrite(letterfile);
  for counter:=1 to 10 do
     Writeln(letterfile);
   Writeln(letterfile,'Dear Mr./Mrs.',student.lname,':');
  Writeln(letterfile);
  Writeln(letterfile,'Your average is over 70 %. ' );
  Writeln(letterfile,'congratulations. ');
  Writeln(letterfile);
  Writeln(letterfile,'Sincerly,');
  Writeln(letterfile);
  Writeln(letterfile,'Dean of Students');
  close(letterfile);
end;

Procedure DisplayInfo( student:Studentrec);
begin
  With student do
    begin
       Writeln('The student''s  name is: ',fname,' ',lname,'.');
      Writeln('The student''s  average is : ',average:5:2,'.' );
     if average>70 then
       SendHonorsLetter(student);
    end
end;
```

```
begin
  Init(student);
  FillRecord(student);
  DisplayInfo(student);
end.
```

نلاحظ في هذا المثال أن التسجيلة الممثلة للطالب تعامل كعنصر منفعل نمرره في كل الإجرائيات. فلملء حقول التسجيلة تمرر كوسيط لإجرائية FillRecord ، ولإظهار المعلومات تمرر لإجرائية DisplayInfo وكذلك الحال في إجرائية SendHonorsLetter.

في البرمجة الغرضية التوجه، نريد كما أسلفنا الذكر، أن نجعل الطالب عنصراً فعالاً. ولذا نستخدم مفهوم الغرض **object**.

نعرف النمط التالي:

```
Type
studentrec= Object
            fname,
            lname: string15;
            average: real;
            Procedure Init;
            Procedure FillRecord;
            Procedure SendHonorsLetter;
            Procedure DisplayInfo;
            end;
```

نلاحظ هنا أننا طورنا مفهوم التسجيلة باتجاه مفهوم الغرض. فالغرض يضم الحقول الموصفة لبنية المعطيات، مضافاً إليها جميع الإجرائيات والتوابع اللازمة للتعامل مع معطيات الغرض نفسه. ويطلق على هذه العملية اسم التغليف أو الكبسلة **Encapsulation** وكأننا غلفنا المعطيات والعمليات المؤثرة فيها بغلاف أو كبسولة.

وتستخدم الكلمة المحجوزة **OBJECT** كما نرى لتعريف نمط الغرض "طالب". وتجدر الإشارة هنا إلى أنه يجب إيراد حقول المعطيات قبل العمليات.

نستطيع الآن التعامل مع الأغراض كما كنا نفعل مع الأنماط الأخرى، فنعرف مثلاً متحولاً من طبيعة الغرض "طالب" فنكتب:

```
var student:studentrec;
```

٢ -الطرائق Methods:

بقي علينا الآن أن نكتب المحتوى التفصيلي للعمليات المعرفة داخل بنية الغرض. والحقيقة أننا نطلق على العمليات اسم الطرائق **methods** لأنها تعرف الطريقة التي سيكون عليها رد فعل الغرض تجاه طلب خارجي، أي التي سيجيب بها عندما يطلب منه القيام بعملية.

وفيما يلي إعادة للبرنامج الذي يتعامل مع بنية الطالب بعد إدخال مفهوم الغرض:

```
Program  Department;
Type
   string15= String[15];
  Studentrec= OBJECT
                fname,
               lname: string15;
                average: real;
```

```
            Procedure Init;
            Procedure FillRecord;
            Procedure SendHonorsLetter;
            Procedure DisplayInfo;
            end;

Procedure Studentrec.Init;
 begin
      fname:=' ';
      lname:=' ';
     average:=0.0;
 end;

Procedure Studentrec.FillRecord;
 begin
   Write('Enter the student"s first name: ');
   Readln(fname);
   Write('Enter the student"s last name: ');
   Readln(lname);
   Write('Enter the student"s current average: ');
   Readln(average);
 end;

Procedure Studentrec.SendHonorsLetter;
var
  letterfile:text;
  counter:integer;
 begin
   assign(letterfile,'prn');
  rewrite(letterfile);
  for counter:=1 to 10 do
    Writeln(letterfile);
   Writeln(letterfile,'Dear Mr./Mrs.',student.lname,':');
   Writeln(letterfile);
   Writeln(letterfile,'Your average is over 70 %. ');
   Writeln(letterfile,'congratulations. ');
```

```
    Writeln(letterfile);
    Writeln(letterfile,'Sincerly,');
    Writeln(letterfile);
    Writeln(letterfile,'Dean of Students');
    close(letterfile);
  end;

Procedure Studentrec.DisplayInfo;
  begin
    Writeln('The student''s  name is: ',fname,' ',lname,'.');
    Writeln('The student''s  average is : ',average:5:2,'.' );
    if average>70 then
         SendHonorsLetter;
  end;

var student:Studentrec;

begin
  student.Init;
  student.FillRecord;
  student.DisplayInfo;
end.
```

نلاحظ في هذا المثال أن بنية Studentrec تحتوي تعريفاً بالإجرائيات (الطرائق) على حين يصرَّح عن الإجرائيات (الطرائق) كاملة خارج البنية. والدلالة على كون الإجرائية(الطريقة) المصرح عنها تابعة لغرض ما، يوضع اسم الغرض قبل اسم الإجرائية (الطريقة) وتليه نقطة« . » ، وذلك لإزالة الالتباس في حال التصريح عن عدة أغراض يحتوي كل منها على نفس اسم الإجرائية (الطريقة).

كما يلاحظ أن الإجرائية (الطريقة) لا تأخذ كوسيط متحولاً من نمط الغرض، لأننا نعتبر أن الغرض نفسه يتضمن (يكبسل) المعطيات والعمليات، وبذلك فإن العمليات ترى

المعطيات مباشرة دون حاجة إلى تمريرها كوسيط.

نطلق على الحقول التي تحتوي المعطيات اسم الواصفات.

وفي الحقيقة، من المفيد أن يوضع الغرض في وحدة منفصلة unit، لأنه يحتوي على بنية مستقلة، وعلى العمليات التي تتعامل معها مباشرة، وهذا مما يسهل إمكان استخدامه لاحقاً. وعلى هذا يمكننا أن نكتب المثال السابق كما يلي:

```
Unit my_obj;
Interface
Type
   string15= String[15];
   Studentrec= OBJECT
               fname,
               lname: string15;
               average: real;
               Procedure Init;
               Procedure FillRecord;
               Procedure SendHonorsLetter;
               Procedure  DisplayInfo;
               end;

Implementation
Procedure Studentrec.Init;
 begin
  fname:=' ';
     lname:=' ';
     average:=0.0;
 end;

Procedure Studentrec.FillRecord;
 begin
   Write('Enter the student''s first name: ');
   Readln(fname);
```

```
    Write('Enter the student''s last name: ');
    Readln(lname);
    Write('Enter the student''s current average: ');
  Readln(average);
  end;

Procedure Studentrec.SendHonorsLetter;
var
  letterfile:text;
  counter:integer;
begin
  assign(letterfile,'prn');
  rewrite(letterfile);
  for counter:=1 to 10 do
    Writeln(letterfile);
  Writeln(letterfile,'Dear Mr./Mrs.',student.lname,':');
  Writeln(letterfile);
  Writeln(letterfile,'Your average is over 70 %. ' );
  Writeln(letterfile,'congratulations. ');
  Writeln(letterfile);
  Writeln(letterfile,'Sincerly,');
  Writeln(letterfile);
  Writeln(letterfile,'Dean of Students');
  close(letterfile);
end;

Procedure Studentrec.DisplayInfo;
begin
Writeln('The student''s  name is: ',fname,' ',lname,'.');
  Writeln('The student''s  average is : ',average:5:2,'.' );
  if average>70  then
      SendHonorsLetter;
end;
end;
```

٣-الوراثة Inheritance:

قد نحتاج إلى تطوير بنية المعطيات الممثلة للطالب ، كأن نحتاج إلى إضافة حقل يبين : هل الطالب المتخرج طالب دراسات عليا يقوم بالتحضير للدكتوراه في القسم؟ يجب إذن أن يحتوي "الطالب" إضافة إلى الحقول السابقة الذكر، على حقل منطقي يشير إلى أنه يحضر للدكتوراه.

في البرمجة المهيكلة يجب علينا أن نبني تسجيلة جديدة تتضمن التسجيلة السابقة studentrec ونضيف إليها الحقول الجديدة.فنعرف تسجيلة تسمى "طالب دراسات عليا" gradstudentrec كمايلي:

```
Type    gradstudentrec= record
                        studentinfo:studentrec;
                        PhDcandidate:boolean;
                        end;
```

في هذه الحالة نضطر إلى تعديل الإجرائيات Init, FillRecord, SendHonorsLetter, DisplayInfo للتسجيلة الجديدة لأنها تحتوي حقلاً إضافياً، أو نطلق أسماء جديدة على هذه الإجرائيات منعاً لحدوث تضارب مع أسماء الإجرائيات المعرفة للتسجيلة studentre، فنطلق عليها مثلاً اسم gradInit, gradFillRecord, gradSendHonorsLetter, gradDisplayInfo وبالطبع فإنّ هذين الحلين يؤديان إلى تعديلات كبيرة في البرنامج في أماكن عديدة.

في البرمجة الغرضية التوجه تختلف آلية العمل، ويستخدم مفهوم **الوراثة inheritance**. يعتمد مفهوم الوراثة على مايشبه الوراثة لدى الأحياء، فكما يرث الابن صفاته الوراثية من أبيه، كلون الشعر ولون البشرة ولون العيون والطول.....إلخ، فإننا نقوم في البرمجة الغرضية التوجه ببناء غرض نشتقه من غرض أب، فيرث الغرض الابن جميع حقول الغرض الأب، ونستطيع إضافة ما نشاء من صفات خاصة جديدة . وفي مثالنا عن الطالب المتخرج نعرف غرضاً يسمى طالب متخرج Gradstudentrec كغرض مشتق من الغرض الأب Studentrec كما يلي:

```
Type    Gradstudentrec= object(Studentrec)
                        PhDcandidate:boolean;
                        Procedure Init;
                        Procedure Fillrecord;
                        Procedure SendHonorsLetter;
                        end;
```

ونلاحظ هنا أننا وضعنا اسم الغرض الأب، الذي اشتققنا منه الغرض، بعد الكلمة المحجوزة object. إن أي متحول من النمط Gradstudentrec يحتوي على جميع الخصائص المذكورة في تعريفه وتعريف أبيه Studentrec، أي الحقول fname, lname, average, PhDcandidate. أما الإجرائيات التي تتعامل مع هذه الحقول فقد عرفناها وهي Init, Fillrecord, SendHonorsLetter . إن هذه الإجرائيات تحمل نفس أسماء الإجرائيات التابعة للغرض الأب ولكنها كما سنرى مختلفة المعنوى، فنحن نعيد كتابتها في الغرض الابن. ولو عرفنا متحولاً من النمط Gradstudentrec ثم طلبنا منه أن يعطي نفسه قيمته الابتدائية فإنه سيقوم باستدعاء الإجرائية Init الخاصة به أي المعرفة ضمنه، لا الإجرائية Init المعرفة ضمن الغرض الأب.

أما الإجرائية DisplayInfo فلا حاجة بنا إلى إعادة تعريفها، ولذا فإن الغرض Gradstudentrec يرثها من أبيه، ومن ثم إذا طلب من متحول من النمط Gradstudentrec أن يظهر المعلومات الخاصة به فإنه سيستدعي الإجرائية DisplayInfo المعرفة ضمن الغرض الأب لعدم وجود أخرى ناسخة لها بنفس الاسم في الغرض الابن.

وفيما يلي المثال كاملاً:

مثال١:

```
Program DepartmentOBJ;
Uses my_obj;
Type Gradstudentrec= object(Studentrec)
                     PhDcandidate:boolean;
                     Procedure Init;
                     Procedure Fillrecord;
                     Procedure SendHonorsLetter;
                     end;


Procedure Gradstudentrec.Init;
 begin
  Studentrec.init;
  PhDcandidate:=false;
 end;


Procedure Gradstudentrec.FillRecord;
var candidate:char;
```

```
 begin
   Studentrec.FillRecord;
   Write('Is the student a PhD candidate (Y/N)? ');
   Readln(candidate);
   if candidate='Y' then
     PhDcandidate:=true
   else
     PhDcandidate:=false
end;

Procedure Gradstudentrec.SendHonorsLetter;
var
  letterfile:text; '
  counter:integer;
begin
  assign(letterfile,'prn');
  rewrite(letterfile);
  for counter:=1 to 10 do
    Writeln(letterfile);
   Writeln(letterfile,'Dear Mr./Mrs.',student.lname,':');
  Writeln(letterfile);
  Writeln(letterfile,'Your average is over 70 %.
          This means that' );
  Writeln(letterfile,'You will receive your PhD
          "with distinction" ' );
  Writeln(letterfile,'from this institution ' );
  Writeln(letterfile,'congratulations. ');
  Writeln(letterfile);
  Writeln(letterfile,'Sincerly,');
  Writeln(letterfile);
  Writeln(letterfile,'Dean of Students');
  close(letterfile);
end;
```

```
var gradstudent: Gradstudentrec;

begin
  gradstudent.Init;
  gradstudent.FillRecord;
  gradstudent.DisplayInfo;
end.
```

ملاحظة:

إن تنفيذ البرنامج المذكور آنفاً سيعطي النتائج المتوقعة جميعها، عدا تلك الناتجة من تنفيذ الاستدعاء Gradstudent.DisplayInfo . سنناقش في فقرة "الطرائق الافتراضية وتعددية الأشكال" سبب المشكلة وحلولها. ينصح الطالب هنا بعدم الانتقال إلى الفقرة المشار إليها، وبمتابعة القراءة بالتسلسل.

٤- النموذج Instance:

نلاحظ هنا أن استدعاء الإجراء Init حدث وفق الصيغة gradstudent.Init وذلك لأن الغرض المعرّف هنا وهو Gradstudentrec هو نمط معطيات. ويجري عادة التعامل مع المتحولات في البرنامج ، لا مع نمط المعطيات مباشرة، فنقوم بتعريف متحول من النمط ، كما فعلنا عندما صرّحنا عن المتحول بالشكل : var gradstudent: Gradstudentrec; يسمى gradstudent باصطلاح البرمجة الغرضية التوجه نموذجاً Instance.

مثال٢:

سنقوم أولا بتعريف الغرض "موظف" Employee، وواصفاته هي: الاسم والعنوان والعمليات المؤثرة في هذه الواصفات، وهي: عملية إعطاء القيم الابتدائية Init وعملية الحصول على الاسم GetName وعملية الحصول على العنوان GetAddress. ولنضع هذا الغرض في وحدة unit.

```
UNIT uemploy;
Interface
Type
        Employee =Object
                name:string;
                address:string;
                Procedure Init(newname,newaddress:string);
                Function GetName:string;
                Function GetAddress:string;
                end;

Implementation

Procedure Employee.Init;
        begin
                name:=newname;
                address:=newaddress;
        end;

Function Employee.GetName;
        begin
                GetName:=name;
        end;
```

```
Function Employee.GetAddress;
      begin
            GetAddress:=address;
      end;
end;
```

نريد الآن أن نبني صنفاً محدداً من الموظفين، وهو صنف الموظفين الذين يعملون بدوام جزئي. وسنطلق عليهم اسم PartTime. سنشتق الموظف ذا الدوام الجزئيPartTime من الغرض الأب موظف Employee فنكتب:

```
type PartTime=Object(Employee)
                  hoursworked:real;
                  hourlyrate:real;
                  Procedure Init(newname,newaddress:string;
                                 hourslogged:real);
                  Function CalcPay:real;
                  end;
```

إن أي متحول من النمط PartTime يحتوي على جميع الخصائص المذكورة في تعريفه وتعريف أبيه Employee، أي الحقول name, address, hoursworked, hourlyrate, والعملية الجديدة Init و العملية CalcPay، حيث يرث الحقول (الواصفات name, address من أبيه Employee، كما يرث العمليات (الطرائق GetName, GetAddress أيضاً. ويصبح لدينا الوحدة:

```
UNIT uemploy2;
Interface
Type
 Employee =Object
        name:string;
        address:string;
        Procedure Init(newname,newaddress:string);
        Function GetName:string;
        Function GetAddress:string;
        end;
```

```
PartTime=Object(Employee)
        hoursworked:real;
        hourlyrate:real;
        Procedure  Init(newname,newaddress:string; hourslogged:real);
        Function CalcPay:real;
        end;
```

Implementation

```
Procedure Employee.Init;
        begin
                name:=newname;
                address:=newaddress;
        end;

Function Employee.GetName;
        begin
                GetName:=name;
        end;

Function Employee.GetAddress;
        begin
                GetAddress:=address;
        end;

Procedure PartTime.Init;
        begin
                Employee.Init(newname,newaddress);
                hoursworked:=hourslogged;
                hourlyrate:=4.50;
        end;

Function PartTime.CalcPay;
        begin
                calcPay:=hoursworked*hourlyrate;
        end;
end;
```

تتجلى الفائدة الحقيقية من الوراثة في سماحها بتطوير بنى معطياتنا دون الحاجة إلى خلخلة البرامج، فنحن نعيد كتابة العمليات التي يطرأ عليها التغيير فقط مع حفاظنا على التسميات.

كما أن للوراثة فائدة كبيرة تظهر حين نتعامل مع كائنات برمجية عدة، تشترك في انتمائها إلى أصل واحد يعطيها خصائص مشتركة، ولكن لكل منها خصائص مميزة إضافية تتطلب معالجة خاصة. في هذه الحال، نصنع شجرة وراثة يكون الجذر فيها عنصراً يتميز بالصفات المشتركة، ويتفرع عنه أغراض يمثل كل منها كائناً محدداً.

لجعل الموضوع أكثر وضوحاً، سنعرض مثالاً يبين معالجة الفكرة بالطريقة المهيكلة التقليدية، ثم يعالجها بواسطة الأغراض، ويظهر الكسب الحقيقي بوضوح وجلاء.

مثال٣:

نريد بناء نظام برمجي لتخزين محتويات مكتبة وفهرستها. تقسم محتويات المكتبة إلى عدة أنواع، فمنها الوثائق، ومنها الكتب، ومنها المقالات. وتعرّف الوثيقة بعنوانها واسم مؤلفها والسنة التي صدرت فيها والرمز المعطى لها ضمن قيود المكتبة. أما الكتاب، فيعرّف بنفس هذه الواصفات مضافاً إليها الناشر وسعر البيع. ويعرّف المقال بنفس واصفات الوثيقة مضافاً إليها اسم المجلة التي نشر فيها ورقم المجلد ورقم الصفحة.

تقتضي آلية التفكير في البرمجة التقليدية (المهيكلة) أن تعرّف ثلاثة أنواع من التسجيلات: تسجيلة للوثيقة، وتسجيلة للكتاب وتسجيلة للمقال. والتعامل مع هذه التسجيلات بعمليات إعطاء القيمة الابتدائية، وإسناد المعلومات الخاصة بوثيقة أو كتاب أو مقال، وإعطاء الرمز الموافق له ضمن قيود المكتبة، وفهرسته وإظهار توصيفه، لابد من كتابة الإجرائيات المناسبة لكل بنية من هذه البنى، فهي برغم اشتراكها في جزء من الواصفات، إلا أنها تختلف فيما بينها في بعض الواصفات الأخرى. ونرى في المثال أننا قمنا بكتابة الإجرائيات InitDocument, DescribeDocument, CatalogDocument وذلك للتعامل مع الوثيقة، كما قمنا بكتابة الإجرائيات InitBook, DescribeBook, CatalogBook للتعامل مع الكتاب، وقمنا بكتابة الإجرائيات InitArticle, DescribeArticle, CatalogArticle للتعامل مع المقال، أما إجرائية إعطاء الرمز، فهي واحدة لجميع هذه البنى لأنها تؤثر في حقل واحد فقط ومشترك بينها وهو الحقل code.

```
Program PubDemo;
Type
        MedStr = String[80];
        Document = Record
                        Author, Title, Code :MedStr;
                        Year:Integer;
                        end;(*Document*)

        Book =          Record
                        Author,Title,
                        Code,Publisher:MedStr;
                        Year:Integer;
                        Price:Real;
                        end; (*Book*)
```

```
    Article =   Record
                Author, Title, Code : MedStr
                Year : Integer;
                Journal : MedStr;
                Volume, Page : Integer;
                end; (* Article*)

Procedure InitDocument(Var Pub:Document;Au,Ti : MedStr;
                       Yr:Integer);
    Begin
        with Pub do
          begin
            Author:=Au;
            Title:=Ti;
            Year:=Yr;
            Code:=' ';
          end;
    End; (* InitDocument *)

Procedure DescribeDocument ( Pub:Document);
    Begin
      with Pub do

        writeln(code,':   ',Author,' , ',Title, ' [', Year,' ] ');

    End; (* DescribeDocument*)

Procedure GetCode   ( Var TheCode: MedStr);
    Begin
        write('Code? ');
        Readln(TheCode);
    End; (*GetCode*)

Procedure  CatalogDocument(Var Pub:Document);
    Begin
        DescribeDocument(Pub);
        GetCode(Pub.Code);
    End; (* CatalogDocument*)
```

```
Procedure InitArticle(Var Art:Article; Au, Ti, Jour:MedStr;
                      Yr,Vol,Pg:Integer);
    Begin
        with Art do
          begin
            Author := Au;
            Title:=Ti;
            Year:=Yr;
            Code:=' ';
            Journal:=Jour;
            Volume:=Vol;
            Page :=Pg;
          end;
    End; (*InitArticle*)


Procedure DescribeArticle(Art:Article);
  Begin
      with Art do
        begin
          writeln(code,':   ',Author,' , ',Title, ' [', Year,' ] ');
          write(Journal, ',  ');
              writeln ('volume',Volume, '; p.', Page);
        end;
  End; (* DescribeArticle*)


Procedure CatalogArticle (Var Art : Article);
    Begin
        DescribeArticle( Art);
        GetCode (Art.Code);
    End; (* CatalogArticle*)
```

```
Procedure InitBook(Var TheBook:Book; Au, Ti, Pub:MedStr;
                              Yr:Integer;  Pr:Real);
        Begin
              with TheBook do
                  begin
                   Author:=Au;
                   Title:= Ti;
                   Code:= ' ';
                   Year := Yr;
                   Publisher := Pub;
                   Price := Pr;
                  end;
        End; (* InitBook*)

Procedure DescribeBook(TheBook:Book);
    Begin
              with TheBook do
                begin
                   writeln ( Code,': ',Author,',  ',Title,'[',  Year,']');
                  writeln   ('$', Price : 1 : 2, ',  ', Publisher);
                end;
    End; (* DescribeBook*)


Procedure CatalogBook  ( Var TheBook : Book);
        Begin
           DescribeBook ( TheBook);
           GetCode ( TheBook.Code);
        End; (*     CatalogBook *)


Var     MyBk:Book;
        MyArt:Article;
        MyPub:Document;
```

```
Begin
InitDocument (MyPub, 'Many Signers','Declaration of Independence',
              1776);
InitArticle (MyArt, 'Mosteller, F. ','The Mystery of the Missing Corpus',
              'Psychometrika', 1958, 23, 279);
InitBook ( MyBk, 'Prograrmning, O.O.', 'I, Object', 'OOP Press', 1989,
            22.95);
writeln ( 'BOOK:');
CatalogBook( MyBk);
DescribeBook(MyBk);
Readln;
End.
```

باعتماد مبادئ البرمجة الغرضية التوجه، يمكننا في مرحلة أولى أن نعرّف الغرض "وثيقة" (Document)، ذلك أننا نقوم ببناء غرض لكل عنصر في المسألة، يهمنا أن نتعامل معه.

١ـ الغرض "وثيقة" Document:

```
Document = OBJECT
              Author, Title, Code:  Medstr;
              Year : Integer;
              Procedure Init (Au,Ti:MedStr;Yr:Integer);
              Procedure Describe;
              Procedure Getcode;
              Procedure  Catalog;
              End; (* Document*)
```

ونذكّر هنا بأننا أضفنا الطرائق التي تعرّف تصرّف الوثيقة إلى حقول المعطيات لتكوين بنية المعطيات الممثلة للغرض. ويجب علينا إذن تفصيل هذه الطرائق كما يلي.

```
Procedure Document.Init ( Au,Ti : MedStr;Yr:Integer);
        Begin
                Author:=Au;
                Title:=Ti;
                Year:=Yr;
                Code:=' ';
        End; (* Document.Init *)

Procedure Document.Describe;
        Begin
            writeln(code,':   ',Author,' , ',Title, ' [', Year,' ] ');
        End; (* Document.Describe*)

Procedure Document.GetCode;
        Begin
            write('Code? ');
            Readln(Code);
        End; (*Document.GetCode*)

Procedure  Document.Catalog;
        Begin
            Describe;
            GetCode;
        End; (*Document.Catalog*)
```

نذكر هنا بأننا في التصريح عن كل عملية من العمليات الواردة أنفاً لانمرر الغرض كوسيط بل على العكس نقوم بالتعامل مع حقوله مباشرة، ذلك أن هذه الحقول مكبسلة مع الطرائق وتُرى من قِبَلِها مباشرة.

يبين البرنامج التالي كيفية التعامل مع الغرض Document :

```
(*program to illustrate how to define methods on objects*)
(*$V-*)
Program DocDemo;
Type
 MedStr = String [80];
 Document = OBJECT
              Author, Title, Code:  MedStr;
              Year: Integer;
              Procedure Init (Au,Ti:Medstr;Yr:Integer);
              Procedure Describe;
              Procedure GetCode;
              Procedure  Catalog;
              End; (* Document*)


(* ************************methods************************ *)
 Procedure Document.Init ( Au,Ti : MedStr;Yr:Integer);
        Begin
                  Author:=Au;
                  Title:=Ti;
                  Year:=Yr;
                  Code:=' ';
        End; (* Document.Init *)

Procedure Document.Describe;
        Begin
             writeln(code,':   ',Author,' , ',Title, ' [',  Year,' ] ');
        End; (* Document.Describe*)

Procedure Document.GetCode;
        Begin
             write('Code? ');
             Readln(Code);
        End; (*Document.GetCode*)
```

```
Procedure Document.Catalog;
        Begin
              Describe;
              GetCode;
        End; (*Document.Catalog*)

(**************General-Purpose Routines******************)
Procedure GetString(message:String;Var value:String);
      Begin
         write ( Message,' ');
        Readln ( Value);
      End; (* GetString*)

Procedure GetInteger(message:STRING;Var value:INTEGER);
      Begin
         write( Message,' ');
        Readln ( Value);
         End; (* GetInteger *)

Var MyDoc : Document;
      writer, Name : MedStr;
      Year : INTEGER;

Begin
   Repeat
         GetString ( 'Author? (ENTER to stop)', writer);
         If writer <> ' ' Then
              begin
                   GetString('Title',Name);
                   GetInteger('Year',Year);
                   MyDoc.Init(writer, Name, Year);
                   MyDoc.Catalog;
                   MyDoc.Describe;
              end;
  Until writer=' ';
End.
```

٥-التحميل الزائد Overriding:

لنتابع مثالنا ولنعرف بقية الأغراض.

٢- الغرض "كتاب" Book:

تبحث البرمجة الغرضية التوجه عن العوامل المشتركة بين الأغراض، لكي يمكّنها ذلك من بناء شجرة الاشتقاق التي ستقوم الوراثة عليها. إن الغرض الأب يحتوي المواصفات المشتركة، على حين يضيف كل من الأبناء صفاته الخاصة، وكأنّ الأب يمثل صفات الصنف العام، على حين يمثل الابن نوعاً خاصاً من هذا الصنف.

يمكننا بالعودة إلى مثالنا، وعلى هذا الأساس، اعتبار الوثيقة ممثلة للصنف الأب، للكتاب، ذلك أن الكتاب وثيقة من نوع خاص، وله نفس واصفات الوثيقة مضافاً إليها اسم الناشر والسعر. ونعرّف الغرض "كتاب" Book كما يلي:

```
Book = OBJECT (Document)
            Publisher : Medstr;
            Price : REAL;
            Procedure Init(Au, Ti, Pub:MedStr; Yr  INTEGER;
                                Pr :REAL);
            Procedure Describe;
            Procedure Catalog;
        End; (* Book *)
```

يرث Book من Document جميع واصفاته أي أنه يضاف إلى حقول المعطيات Publisher و Price المذكورة صراحة في تعريف Book جميع حقول المعطيات المذكورة في Document وهي:

Author, Title, Code,Year ، كما يرث جميع الطرائق المعرفة على الغرض الأب Document.

ولكن ما السبب في تصريحنا في بنية الغرض Book عن الطرائق Init, Describe, catalog برغم أنه كان مصرحاً عنها في Document ويمكن وراثتها كما ورثنا حقول الواصفات؟

للإجابة عن هذه السؤال نأخذ الحالة التي لانعيد فيها تعريف العمليات وننظر ماذا يحدث. لنفترض أننا عرّفنا الغرض كتاب Book كما يلي:

```
Book = Object (Document)
            Publisher : Medstr;
            Price : REAL;
        End; (* Book *)
```

لنأخذ الآن نموذجاً (instance) من Book: `var MyBk: Book;`

عند حصول الاستدعاء التالي: MyBk.Init. يبحث المترجم compiler عن الطريقة Init ضمن حقول النموذج MyBk فلا يجدها، عندئذ يتجه نحو الأب وهو الغرض Document ليأخذ تعريف الطريقة Init منه لأنه يرثها منه، وعندها تستدعى Init المعرّفة في Document فتقوم هذه بدورها بإعطاء القيم الابتدائية للحقول Author, Title, Code,Year فقط ، أما بقية الحقول فتبقى غير معرّفة.

يتضح الآن أننا محتاجون إلى طريقة تعرّف ضمن الغرض Book وتقوم بإعطاء القيم الابتدائية لجميع حقول المعطيات المعرّفة ضمن Book. وهنا تبرز قوة البرمجة الغرضية التوجه، إذ تتيح لنا إعادة تعريف العملية Init ، أي كتابة طريقة في الغرض الابن تحمل نفس اسم الطريقة في الغرض الأب دون حدوث تضارب لدى المترجم.

تسمّى هذه الآلية **التحميل الزائد**، بمعنى أننا حمّلنا في الاسم نفسه إجرائيتين مختلفتين.

وعند الاستدعاء إذا كان النموذج instance الذي يستدعي Init هو من النمط Document، فإن الإجرائية المنفذة هي تلك المعرّفة في Document، أما إذا كان النموذج الذي يستدعي Init هو من النمط Book، فإن الإجرائية المنفذة هي تلك المعرّفة في Book.

يمكننا الآن تعريف الطرائق المذكورة في التعريف الصحيح للغرض Book حيث أ... تعريف الطرائق Init, Describe, catalog .

```
Procedure Book.Init(Au,Ti,Pub:MedStr;Yr:Integer; Pr:Real);
    Begin
        Document.Init ( Au,Ti,Yr);
        Publisher := Pub;
        Price := Pr;
    End; (* Book.Init*)

Procedure Book.Describe;
    Begin
        Document.Describe;
        writeln ( '$', Price:1:  2, ',' , Publisher);
    End; (* Book.Describe*)
```

```
Procedure Book.Catalog;
        Begin
              Describe;
              GetCode;
        End;(* Book.Catalog*)
```

يلاحظ هنا أن الإجرائيةGetCode المستدعاة في Book.Catalog هي تلك المعرّفة في Document، فلم يُعَدْ تعريفها لأنّها لاتؤثر في الحقول الخاصة بـ Book، بل تؤثر في الحقل code المعرّف في الأب والمورّث لجميع الأبناء.

٣ـ الغرض "مقال" Article:

كما فعلنا في الغرض Book، فإننا سنعرّف الغرض Article كابن لـ Document باعتبار المقال وثيقة من نوع خاص، وله نفس واصفات الوثيقة مضافاً إليها اسم المجلة ورقم المجلد ورقم الصفحة. ونعرّف الغرض "مقال" Article كما يلي:

```
Article = OBJECT (Document)
              Journal : MedStr;
              Volume, Page : Integer;
              Procedure Init ( Au, Ti, Jour : MedStr;
                                Yr, Vol, Pg :  Integer);
              Procedure Describe;
              Procedure Catalog;
        End; (* Article *)


Procedure  Article.Init (Au,Ti,Jour:  MedStr;
                         Yr,Vol,Pg:Integer);
        Begin
              Document.Init ( Au,Ti,Yr);
              Journal := Jour;
```

```
        Volume := Vol;
        Page:=Pg;
    End; (* Article.Init*)


Procedure Article.Describe;
    Begin
        Document.Describe;
        writeln ( Journal,  ',' );
        writeln ( 'volume ', volume,  '; p.' , page);
    End; (* Article.Describe*)


Procedure  Article.Catalog;
    Begin
        Describe;
        GetCode;
    End;(* Article.Catalog*)
```

والآن نكتب برنامجاً كاملاً يضم التعريفات التي قمنا بها للأغراض الثلاثة,Document

.Book, Article

```
(* Program to illustrate how to specify methods to
  override methods inherited from ancestor object types.  *)
(* $V-*)

 Program OverrideDemo;
 Type MedStr = String[ 80];
        Document = OBJECT
                        Author, Title, Code:  Medstr;
                        Year : Integer;
```

```
                Procedure  Init (Au,Ti:Medstr;Yr:Integer);
                Procedure Describe;
                Procedure Getcode;
                Procedure  Catalog;
              End; (* Document*)

Book =   Object (Document)
                Publisher : Medstr;
                Price : Real;
                Procedure  Init (Au, Ti, Pub:MedStr;
                                 Yr:  Integer;  Pr:Real);
                Procedure Describe;
                Procedure Catalog;
              End; (* Book *)


Article =  OBJECT (Document)
                Journal : MedStr;
                Volume, Page : Integer;
                Procedure Init ( Au, Ti, Jour : MedStr;
                                 Yr, Vol, Pg : Integer);
                Procedure Describe;
                Procedure Catalog;
              End; (* Article *)

(* **********************methods********************** *)
Procedure Document.Init ( Au,Ti : MedStr;Yr:Integer);
        Begin
                Author:=Au;
                Title:=Ti;
                Year:=Yr;
                Code:=' ';
        End; (* Document.Init *)


Procedure Document.Describe;
        Begin
```

```
        writeln(code,':   ',Author,' , ',Title, ' [', Year,' ] ');
      End; (* Document.Describe*)


Procedure Document.Getcode;
      Begin
       write('Code? ');
       Readln(Code);
      End; (*Document.GetCode*)


Procedure  Document.Catalog;
      Begin
       Describe;
       GetCode;
      End; (*Document.Catalog*)

Procedure  Book.Init (Au, Ti, Pub:  MedStr;
                          Yr:Integer;  Pr:Real);
      Begin
       Document.Init ( Au,Ti,Yr);
       Publisher := Pub;
       Price := Pr;
      End; (* Book.Init*)

Procedure Book.Describe;
      Begin
       Document.Describe;
       writeln ( '$', Price:1:   2,  ',' , Publisher);
      End; (* Book.Describe*)

Procedure Book.Catalog;
      Begin
       Describe;
       GetCode;
      End;(* Book.Catalog*)
```

```
Procedure Article.Init(Au, Ti,Jour:  MedStr;Yr,Vol,Pg:Integer);
        Begin
         Document.Init ( Au,Ti,Yr);
         Journal := Jour;
         Volume := Vol;
         Page:=Pg;
         End; (* Article.Init*)

Procedure Article.Describe;
        Begin
         Document.Describe;
         writeln ( Journal,  ',' );
         writeln ( 'volume ', volume,  ';  p.' , page);
        End; (* Article.Describe*)

Procedure  Article.Catalog;
        Begin
         Describe;
         GetCode;
        End;(* Article.Catalog*)

(* ***************General-Purpose Routines**************** *)
Procedure  GetString(message:String;Var value:String);
       Begin
          write ( Message,' ');
         Readln ( Value);
       End; (* GetString*)

 Procedure GetInteger(message:String;Var value:Integer);
       Begin
          write( Message,' ');
         Readln ( Value);
       End; (* GetInteger *)
```

```
Procedure GetReal( message:String;Var Value:Real);
      Begin
         write( Message,' ');
        Readln ( Value);
      End; (* GetReal *)

Var     MyDoc : Document;
        MyBk : Book;
        MyArt : Article;
        writer, Name,Pub, Jour : MedStr;
        Choice, Year, Vol, Pg : Integer;
        Price : Real;

Begin

        GetInteger ( '0) Quit; 1) Doc; 2) Book; 3) Article', Choice);
        while Choice <> 0 do
         Begin
                    GetString(' Author?', writer);
                    Getstring('Title', Name);
                    GetInteger('Year',Year);

                    Case Choice of

                    1 :Begin
                        MyDoc.Init ( writer, Name, Year);
                        MyDoc.Catalog;
                        MyDoc.Describe;
                      End;
                    2 :Begin
                        GetString('Publisher', Pub);
                        GetReal('Price',Price);
                        MyBk.Init ( writer, Name, Pub, Year, Price);
                        MyBk.Catalog;
                        MyBk.Describe;
                      End;
```

```
                3 :Begin
                    GetString('Journal', Jour);
                    GetInteger('Volume',Vol);
                    GetInteger('Page',Pg);
                    MyArt.Init (writer, Name, Jour, Year, Vol, Pg);
                    MyArt.Catalog;
                    MyArt.Describe;
                  End;
                End; (*case*)
                GetInteger ( '0) Quit; 1) Doc; 2) Book; 3) Article',
                              Choice);
        End; (*while *)
End.
```

٦- الطرائق الافتراضية وتعددية الأشكال

:Virtual methods and Polymorphism

مثال ١:

لنعد الآن إلى مثال الطالب المتخرج والمسجّل في الدكتوراه في القسم.

رأينا في المثال الأول الوارد في فقرة الوراثة البرنامج التالي :

```
Program DepartmentOBJ;
Uses my_obj;
Type   Gradstudentrec= Object(Studentrec)
                PhDcandidate:boolean;
                Procedure Init;
                Procedure FillRecord;
                Procedure SendHonorsLetter;
                End;
```

```
Procedure Gradstudentrec.Init;
 Begin
  Studentrec.init;
  PhDcandidate:=false;
 End;

Procedure Gradstudentrec.FillRecord;
Var  candidate:char;
 Begin
   Studentrec.FillRecord;
   write('Is the student a PhD candidate (Y/N)? ');
    Readln(candidate);
   if candidate='Y' then
     PhDcandidate:=true
   else
     PhDcandidate:=false
End;

Procedure Gradstudentrec.SendHonorsLetter;

Var
  letterfile:text;
  counter:integer;
 Begin
   assign(letterfile,'prn');
  rewrite(letterfile);
  for counter:=1 to 10 do
    writeln(letterfile);
   writeln(letterfile,'Dear Mr./Mrs.',student.lname,':');
   writeln(letterfile);
   writeln(letterfile,'Your average is over 70 %. This means that' );
   writeln(letterfile,'You will receive your PhD "with distinction" ' );
   writeln(letterfile,'from this institution ' );
   writeln(letterfile,'congratulations. ');
   writeln(letterfile);
```

```
    writeln(letterfile,'Sincerly,');
    writeln(letterfile);
    writeln(letterfile,'Dean of Students');
    close(letterfile);
  End;

Var gradstudent: Gradstudentrec;

Begin
  gradstudent.Init;
  gradstudent.FillRecord;
  gradstudent.DisplayInfo;
End.
```

وذكرنا في نهاية تلك الفقرة أن تنفيذ البرنامج يؤدي إلى نتيجة غير متوقعة، فعند تنفيذ الاستدعاء في السطر الأخير وهو الاستدعاء التالي:gradstudent.DisplayInfo نتوقع أن نجد رسالة يوافق محتواها المحتوى الذي كتبناه لهذا الغرض في الإجرائية Gradstudentrec.SendHonorsLetter، ولكن الحقيقة أن الرسالة التي نجدها هي الرسالة التي كُتب محتواها عند تعريف الإجرائية Studentrec.SendHonorsLetter، فماالسبب في ذلك برغم قيامنا بإعادة تعريف الإجرائية المذكورة عند بناء الغرض الابن GradStudentrec ؟

يكمن السر في أننا لم نُعدْ تعريف الإجرائية DisplayInfo التي تستدعي SendHonorsLetter، فهي إذن موروثة من الغرض الأب Studentrec، وعند الترجمة يؤدي الاستدعاء Gradstudent.DisplayInfo إلى استدعاء الإجرائية DisplayInfo المعرفة في الأب والتي تعتبر حقلاً من حقوله وجزءاً منه. تحوي هذه الإجرائية بدورها استدعاءً للإجرائية SendHonorsLetter التي هي بدورها جزءٌ من الغرض الأب نفسه. هذا الاستدعاء يظهر الرسالة غير المتوقعة.

ولما كان الربط بين استدعاء إجرائية ما ونوع الغرض الذي يجب أن تؤخذ منه نسخة الإجرائية يجري في مرحلة الترجمة Compile Time، فإن هذا الأسلوب من الربط المبكّر يسمى الربط الستاتيكي أو الساكن **Static Binding** .

إن الحل البديهي لهذه المشكلة هو تعريف الطريقة DisplayInfo في الغرض GradStudentrec كما يلي:

```
Program DepartmentOBJ2;

Uses my_obj;


Type        Gradstudentrec= Object(Studentrec)
                            PhDcandidate:Boolean;
                            Procedure Init;
                            Procedure FillRecord;
                            Procedure SendHonorsLetter;
                            Procedure DisplayInfo;
                            End;

Procedure Gradstudentrec.Init;
 Begin
  Studentrec.init;
  PhDcandidate:=false;
 End;

Procedure Gradstudentrec.FillRecord;
Var candidate:char;
  Begin
   Studentrec.FillRecord;
   write('Is the student a PhD candidate (Y/N)? ');
   Readln(candidate);
```

```
   if candidate='Y' then
      PhDcandidate:=true
   else
      PhDcandidate:=false
End;

Procedure Gradstudentrec.SendHonorsLetter;
Var
  letterfile:text;
  counter:integer;
 Begin
   assign(letterfile,'prn');
   rewrite(letterfile);
   for counter:=1 to 10 do
     writeln(letterfile);
writeln(letterfile,'Dear Mr./Mrs.',student.lname,':');    writeln(letterfile);
    writeln(letterfile,'Your average is over 70 %. This means that' );
    writeln(letterfile,'You will receive your PhD "with distinction" ' );
    writeln(letterfile,'from this institution ' );
    writeln(letterfile,'congratulations. ');
    writeln(letterfile);
    writeln(letterfile,'Sincerly,');
    writeln(letterfile);
    writeln(letterfile,'Dean of Students');
    close(letterfile);
 End;

Procedure Gradstudentrec.DisplayInfo;
Begin
     writeln('The student"s  name is: ',fname,' ',lname,'.');
     writeln("The student"s  average is : ',average:5:2,'.' );
     if average>70  then
          SendHonorsLetter;
 End;

var  gradstudent: Gradstudentrec;
```

```
Begin
  gradstudent.Init;
  gradstudent.FillRecord;
  gradstudent.DisplayInfo;
End.
```

يستخدم الحل المتبع هنا مفهوم الربط الستاتيكي نفسه، إذ يعرّف الطريقة DisplayInfo كجزء من الغرض، لذا يؤدي الاستدعاء ضمنها لـ SendHonorsLetter إلى تنفيذ النسخة المصرّح عنها في Gradstudentrec.

يلاحظ هنا أننا في تعريفنا لـ DisplayInfo ضمن Gradstudentrec كتبنا الطريقة تماماً كما هي في Studentrec دون أي تعديل. فلو اتبعنا هذه الآلية في العمل كلما وجدنا حالة مشابهة، لوجدنا أنفسنا نكرر كتابة الكثير من الإجرائيات بسبب الربط الستاتيكي، وهذا يخالف ماادعيناه في البداية من أننا نختصر في البرمجة الغرضية التوجه كتابة البرامج.

هنالك حل أفضل لهذه المشكلة يعتمد على **الطرائق الافتراضية virtual methods** وهي التي لايجري ربطها في زمن الترجمة. إن التصريح عن طريقة بأنها افتراضية يعني أنّنا نُعلِم المترجم بأننا في زمن الترجمة لا نعرف أية نسخة من هذه الطريقة يجب أن نستدعي، وبأننا نؤجل القرار في هذا حتى وقت التنفيذ، ويسمى هذا الأسلوب من الربط المتأخر **الربط الديناميكي Dynamic Binding**.

تسمح هذه الالية لغرض ما بأن يكون تعدد الاشكال polymorphic فهو يستجيب لرسالة ما (طلب تنفيذ خدمة) بطرائق مختلفة. فمثلاً عندما يتلقى الغرض رسالة

DisplayInfo فعليه أولاً أن يحدد هويته كفرض، ثم يحدد النسخة التي سينفذها للطريقة SendHonorsLetter للرد على الرسالة.

عند التصريح عن طريقة افتراضية تضاف الكلمة المحجوزة **virtual** في نهاية ترويسة الطريقة داخل الفرض.

يبين البرنامج التالي كيف حولنا الطريقة SendHonorsLetter إلى طريقة افتراضية.

```
Unit my_obj2;
(* changes the unit my_obj to make SendHonorsLetter a virtual method
   instead of a static method. principal feature :
    virtual methods, whereby a program can choose at run-time which
    method should be executed, and a Constructor routine , which
    is absolutely required in any object Type
    that uses virtual  methods. An object  variable's Constructor MUST be
called to set   up the VMT before any calls are made to the
   object's other methods.*)


Interface
Type
        string15= String[15];
          Studentrec= OBJECT
                      fname,
                      lname: string15;
                      average: real;
                      Constuctor Init;
                      Procedure FillRecord;
                      Procedure SendHonorsLetter;virtual;
                      Procedure  DisplayInfo;
                      End;
```

```
Implementation

Constuctor Studentrec.Init;
 Begin
      fname:=' ';          lname:=' ';
     average:=0.0;
 End;

Procedure Studentrec.FillRecord;
 Begin
    write('Enter the student"s first name: ');
    Readln(fname);
    write('Enter the student"s last name: ');
    Readln(lname);
    write('Enter the student"s current average: ');
    Readln(average);
 End;

Procedure Studentrec.SendHonorsLetter;
Var
   letterfile:text;
   counter:integer;
 Begin
   assign(letterfile,'prn');
   rewrite(letterfile);
   for counter:=1 to 10 do
     writeln(letterfile);
   writeln(letterfile,'Dear Mr./Mrs.',student.lname,':'); writeln(letterfile);
   writeln(letterfile,'Your average is over 70 %. ' );
   writeln(letterfile,'congratulations. ');
   writeln(letterfile);
   writeln(letterfile,'Sincerly,');
   writeln(letterfile);
   writeln(letterfile,'Dean of Students');
   close(letterfile);
 End;
```

```
Procedure Studentrec.DisplayInfo;
 Begin
     writeln('The student''s  name is: ',fname,' ',lname,'.');
     writeln('The student''s  average is : ',average:5:2,'.' );
     if average>70  then
        SendHonorsLetter;
 End;
End;
```

٧ -البواني Constructors:

نلاحظ وجود فرقين هامّين بين الوحدتين my_obj2 وmy_obj وهما:

١- وجـود كلمـة virtual في نهـاية السطر الذي نصـرّح فـيـه عن اسم الطريقـة SendHonorsLetter المتضمنة في الغرض.

٢- استبدال كلمة Procedure التي كانت مستخدمة قبلاً للتصريح عن الإجرائية Init بالكلمة **Constructor**، وهذا أمر حيوي لابد منه في حالة الطرائق الافتراضية. ولننعم النظر في الأمر لنعرف لِمَ يعتبر الأمر حيوياً وعلى هذه الدرجة من الخطورة.

عندما تجري ترجمة البرنامج، فإن المترجم يعطي لجميع الاستدعاءات الستاتيكية عناوين الذاكرة التي توجد فيها الطرائق. أما في حالة الطرائق الافتراضية، فإن المترجم لايعرف سلفاً أية طريقة سيستدعي عند وجود طلب، ولهذا، لايُعطى أي عنوان في الذاكرة لطريقة افتراضية بسبب عدم وجود عنوان محدد. إن هذا يعني أنه عند الترجمة لاتعرف الاستدعاءات أين يجب عليها أن تجد الإجرائيات المستدعاة. وعند التنفيذ يجب أن تكون هناك وسيلة لإيجاد الطرائق التي يحتاج البرنامج لتنفيذها. يقوم الباني Constructor

بهذه المهمة، فعند تنفيذ البرنامج يجب أن يُستدعى الباني Constructor قبل استدعاء أية طريقة افتراضية،يهيئ الباني جدول الطريقة الافتراضية Virtual Method Table VMT التي تُعلِم البرنامج بأماكن وجود الطرائق الافتراضية. هذا وإن عدم وجود بانٍ يؤدي إلى توقف البرنامج عن العمل.

إن التصريح عن الباني في الغرض لايقتضي أن يقوم الباني بعمليات تهيئة للحقول، فيمكن أن يكون الباني خالياً من أية تعليمة، ويقتصر دوره عندها على تهيئة جدول الطريقة الافتراضية VMT كما في الباني التالي :

```
Constructor AnotherInitMethod;
begin

end;
```

ومن الضروري هنا أن نشير إلى أنه يجب تكرار تعريف الباني في كل غرض يتعامل مع الطرائق الافتراضية، وإن عدم التقيد بهذه الملاحظة قد يؤدي إلى كارثة في البرنامج.

لنقم الآن بإعادة كتابة البرنامج DepartmentOBJ2 بتعريف الطرائق الوهمية. يلاحظ هنا أن البرنامج يستخدم الوحدة my_obj2 بدلاً من my_obj. وهذا يعني أنه يفترض أن تترجم هذه الوحدة قبل ترجمة البرنامج وتكون موجودة في نفس الدليل الذي يجري العمل ضمنه.

```
Program DepartmentOBJ3;
( * changes the program DepartmentOBJ2 by using virtual methods to
    eliminate the redundant code in the program. Principal features :
      virtual methods, whereby a program can choose at run-time which
  method should be executed, and a Constructor routine , which is
absolutely required in any object Type that uses virtual methods. An
object variable's Constructor MUST be called to set up the VMT
    before any calls are made to the object's other methods.* )
```

```
Uses my_obj2;
Type    Gradstudentrec= object(Studentrec)
                        PhDcandidate:boolean;
                        Constructor Init;
                        Procedure Fillrecord;
                        Procedure SendHonorsLetter;virtual;
                        End;

Constructor Gradstudentrec.Init;
 Begin
  Studentrec.init;
  PhDcandidate:=false;
 End;

Procedure Gradstudentrec.FillRecord;
var  candidate:char;
 Begin
   studentrec.FillRecord;
   write('Is the student a PhD candidate (Y/N)? ');
   Readln(candidate);
   if candidate='Y' then
     PhDcandidate:=true
   else
     PhDcandidate:=false
End;

Procedure Gradstudentrec.SendHonorsLetter;
var
  letterfile:text;
  counter:integer;
 Begin
   assign(letterfile,'prn');
   rewrite(letterfile);
   for counter:=1 to 10 do
     writeln(letterfile);
    writeln(letterfile,'Dear Mr./Mrs.',student.lname,':');
```

```
    writeln(letterfile);
    writeln(letterfile,'Your average is over 70 %. This means that' );
    writeln(letterfile,'You will receive your PhD "with distinction" ' );
    writeln(letterfile,'from this institution ' );
    writeln(letterfile,'congratulations. ');
    writeln(letterfile);
    writeln(letterfile,'Sincerly,');
    writeln(letterfile);
    writeln(letterfile,'Dean of Students');
    close(letterfile);
  End;

Procedure Gradstudentrec.DisplayInfo;
Begin
    writeln('The student"s  name is: ',fname,' ',lname,'.');
    writeln('The student"s  average is : ',average:5:2,'.' );
    if average>70  then
        SendHonorsLetter;
 End;

Var  gradstudent: Gradstudentrec;

Begin
  gradstudent.Init;
  gradstudent.fillrecord;
  gradstudent.displayinfo;
End.
```

نلاحظ هنا أنه جرى التصريح عن بانٍ Init في الفرض Gradstudentrec ، كما عُرِّفت الطريقة SendHonorsLetter كافتراضية.

الآن وقد أصبحت SendHonorsLetter افتراضية في كلٍ من Gradstudentrec و Studentrec فإن المترجم لايربط استدعاءها في DisplayInfo بعنوانها في الذاكرة عند

الترجمة. أما عند التنفيذ، فعندما تُستدعى SendHonorsLetter يقوم البرنامج بمعرفة هوية الغرض المستدعي، ثم يستدعي الطريقة SendHonorsLetter المناسبة.

في البرنامج المبيَّن آنفاً يستدعي نموذج من الغرض Gradstudentrec الطريقة SendHonorsLetter ومن ثَم تؤخذ نسخة SendHonorsLetter المعرّفة لدى الغرض Gradstudentrec.

مثال ٢:

سنعيد كتابة البرنامج OverrideDemo الذي يتناول فهرسة الوثائق ، وقد كتبناه في نهاية فقرة الوراثة ، غير أننا سنعيد كتابته باستخدام الطرائق الافتراضية.

```
Program VirtualDemo;
(* $V-*)

Type MedStr = String [ 80];
     Document = Object
                  Author, Title, Code:  Medstr;
                  Year : Integer;
                  Constructor Init (Au,Ti:Medstr;Yr:Integer);
                  Procedure Describe;Virtual;
                  Procedure GetCode;
                  Procedure  Catalog;Virtual;
                  End; (* Document*)

       Book =    Object (Document)
                  Publisher : Medstr;
                  Price : Real;
                  Constructor Init (Au, Ti, Pub : MedStr;
                                    Yr :Integer; Pr  : Real);
                  Procedure Describe;Virtual;
                 End; (* Book *)


       Article =  Object (Document)
                  Journal : MedStr;
                 Volume, Page : Integer;
                  Constructor Init ( Au, Ti, Jour : MedStr;
                                     Yr, Vol, Pg :   Integer);
                  Procedure Describe;Virtual;
                  End; (* Article *)
```

```
(* **********************methods************************** *)
 Constructor Document.Init ( Au,Ti : MedStr;Yr:Integer);
        Begin
                    Author:=Au;
                    Title:=Ti;
                    Year:=Yr;
                    Code:=' ';
        End; (* Document.Init *)

Procedure Document.Describe;
        Begin
         writeln(code,':   ',Author,' , ',Title, ' [',  Year,' ] ');
        End; (* Document.Describe*)

Procedure Document.GetCode;
        Begin
         write('Code? ');
         Readln(Code);
        End; (*Document.GetCode*)

Procedure  Document.Catalog;
        Begin
         Describe;
         GetCode;
        End; (*Document.Catalog*)


 Constructor  Book.Init (Au, Ti, Pub : MedStr;Yr:Integer;
                     Pr:Real );
        Begin
         Document.Init ( Au,Ti,Yr);
         Publisher := Pub;
         Price := Pr;
        End; (* Book.Init*)
```

```
Procedure Book.Describe;
        Begin
         Document.Describe;
         writeln ( '$', Price:1:   2,  ',' , Publisher);
        End; (* Book.Describe*)

(* LOOK! NO  Book.Catalog  *)

Constructor Article.Init (Au, Ti, Jour:  MedStr;   Yr,Vol,Pg:Integer);
        Begin
         Document.Init ( Au,Ti,Yr);
         Journal := Jour;
         Volume := Vol;
         Page:=Pg;
        End; (* Article.Init*)

Procedure Article.Describe;
        Begin
         Document.Describe;
         writeln ( Journal,  ',' );
         writeln ( 'volume ', volume,  ';  p.' , page);
        End; (* Article.Describe*)

(* LOOK! NO  Article.Catalog  *)

(* *************General-Purpose Routines*************** *)
Procedure  GetString(message:STRING;Var value:STRING);
       Begin
          write ( Message,' ');      Readln ( Value);
       End; (* GetString*)

 Procedure GetInteger(message:STRING;Var value:Integer);
       Begin
          write( Message,' ');
         Readln ( Value);
       End; (* GetInteger *)
```

```
Procedure GetReal( message:STRING;Var Value:REAL);
      Begin
         write( Message,' ');
        Readln ( Value);
      End; (* GetReal *)

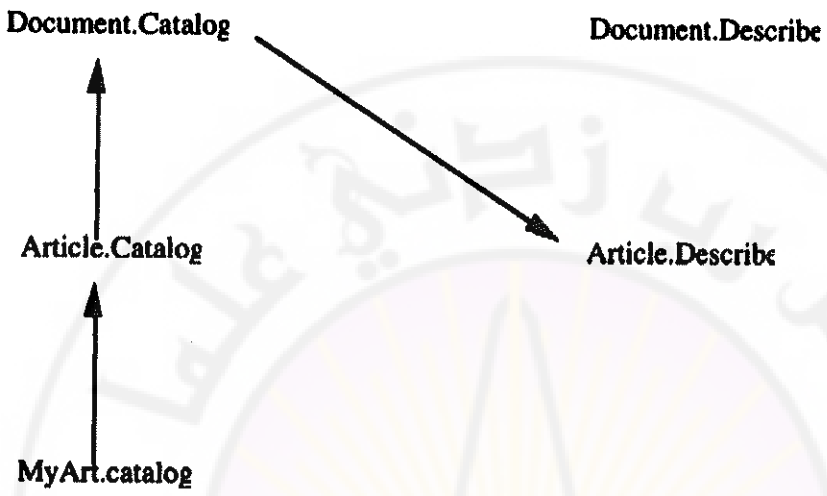
Var    MyDoc : Document;
       MyBk : Book;
       MyArt : Article;
       writer, Name,Pub, Jour : MedStr;
       Choice, Year, Vol, Pg : Integer;
       Price : Real;

Begin
      GetInteger ( '0) Quit; 1) Doc; 2) Book; 3) Article', Choice);
      while Choice <> 0 do
       Begin
                  GetString(' Author?', writer);
                  Getstring('Title', Name);
                  GetInteger('Year',Year);
                  Case Choice of
                   1 :Begin
                      MyDoc.Init ( writer, Name, Year);
                      MyDoc.Catalog;
                      MyDoc.Describe;
                     End;
                  2 :Begin
                      GetString('Publisher', Pub);
                      GetReal('Price',Price);
                      MyBk.Init ( writer, Name, Pub, Year, Price);
                      MyBk.Catalog;
                      MyBk.Describe;
                    End;
                  3 :Begin
                      GetString('Journal', Jour);
                      GetInteger('Volume',Vol);
```

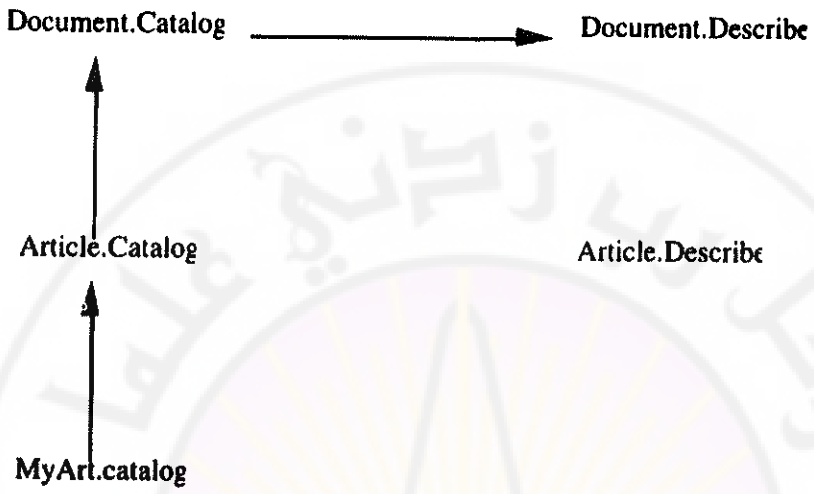
```
                GetInteger('Page',Pg);
                MyArt.Init ( writer, Name, Jour, Year, Vol, Pg);
                MyArt.Catalog;
                MyArt.Describe;
              End ;
            End; (*case*)
            GetInteger ( '0) Quit; 1) Doc; 2) Book; 3) Article',
                         Choice);
        End; (*while *)
End.
```

نلاحظ هنا أننا عرّفنا كلاً من Catalog وDescribe على أنه افتراضي. لندقق النظر في الطريقة Catalog المستدعاة من قبل النموذج MyArt. يبحث البرنامج عن هذه الطريقة ضمن الغرض Article أولاً. فإذا لم يجده، فإنه يبحث عنه في الغرض الأب، وهو في حالتنا الغرض Document فإن لم يجده بحث في الغرض الأب له وهكذا...

وعندما يجد البرنامج الطريقة Catalog يبدأ بتنفيذها فيجد ضمنها استدعاءً للطريقة Describe وهي أيضاً طريقة افتراضية. وهنا يقوم البرنامج بالبحث عن هذه الطريقة ابتداء من الغرض الذي بدأنا الاستدعاء منه في البداية، وهو هنا الغرض Article، وهذا يعني أننا حددنا في Catalog استدعاء لطريقة يأتي ربطها متأخراً. ويجري الاستدعاء وفق المخطط التالي:



إذا لم نستخدم الربط الديناميكي أو التحديد المتأخر للطرائق، أي أزلنا كلمة Virtual المضافة، واستبدلنا كلمة Constructor بكلمة Procedure، فحين نطلب الطريقة Catalog مع النموذج MyArt يجري البحث ضمن الغرض Document ، وعندها تنفذ الطريقة Catalog المعرّفة في Document والتي تستدعي بدورها الطريقة Describe التي يبدأ البحث عنها في الغرض Document والأغراض "الأجداد" الموجودة في شجرة وراثته. ويجري الاستدعاء وفق المخطط التالي:



٨-الهوادم Destructors:

كما عرّفنا في بداية العمل مع الطرائق الافتراضية بواني Constructors تهيئ بيئة العمل وتتبنى الجداول المطلوبة، ينصح بعد الانتهاء من العمل باستخدام هوادم تقوم بتنظيف البيئة. يعرّف الهادم بعد الكلمة المحجوزة Destructor ، ويجري استدعاؤه بعد الانتهاء من العمل مع الغرض. من الأفضل أن نعرّف في كل غرض، يتضمن طرائق افتراضية، هادماً ولو لم نضع بين عبارتي begin و end أية تعليمة، وذلك ليقوم بهدم الجداول التي أنشأها لباني.

يجب استدعاء الهادم قبل الانتهاء من البرنامج، شريطة عدم استدعاء أي منهج وهمي بعده، لأن ذلك يؤدي إلى أخطاء في التنفيذ ولو لم يخبرنا المترجم عنها.

الفصل العاشر

بناء الواجهات التخاطبية

بواسطة أغراض مكتبة Turbo Vision

١- ماهي Turbo Vision؟

إنها مكتبة تتضمن أنماطاً لأغراض يمكن استدعاؤها والتعامل معها بطريقة البرمجة الغرضية التوجه في باسكال.

تسمح هذه المكتبة بإضافة قوائم الاختيار ومستطيلات الحوار، للتطبيقات التي يبنيها المبرمج، إضافة إلى إمكان التعامل مع الفأرة، ومجموعة كبيرة من الخصائص التي يمكن إضافتها إلى التطبيقات، وذلك دون الحاجة لكتابة أسطر ترميز لهذه الخصائص، وإنما يكفي استدعاؤها من مكتبة Turbo Vision.

إن تطوير تطبيق ما باستخدام هذه المكتبة، سيسمح ببناء تطبيقات لها نفس واجهةTurbo Pascal. وهذا يعني أن التعامل المألوف في البرمجة التقليدية مع الدخل والخرج التقليديين لم يعد صالحاً للتطبيق هنا. فمثلاً لم يعد بمقدورنا استخدام تعليمة writeln في البرامج، فقد جرى استبدال هذه العملية وسواها من عمليات الإظهار بطرائق جديدة للتعامل مع المستثمر وهي النوافذ وقوائم الاختيار ومستطيلات الحوار.

إن تعلم مكتبة Turbo Vision أمر يتطلب جهداً كبيراً وإصراراً ورجوعاً دائماً إلى ملفات المساعدة الموجودة في Turbo Pascal. وسنقوم في هذه الفقرة بإيضاح الخصائص

الرئيسية للمكتبة، وهذا مما يسمح بالإقلاع في التعامل معها، ويتعلم المبرمج تدريجياً بحسَب الأمثلة المتاحة تفاصيل أكثر وأكثر.

يتطلب استخدام المكتبة أن يكون المبرمج ماهراً في التعامل مع الأغراض والمؤشرات.

يعتمد المبدأ الأساسي في البرامج المكتوبة باستخدام Turbo Vision على مبدأ معالجة الأحداث events. والحدث هو عملية ضغط لأحد مفاتيح لوحة المفاتيح، أو الضغط على زر الفأرة، أو أية عملية يمكن أن تحدث في أي وقت ودون توقع سابق من قبل البرنامج. فالبرنامج المصنوع وفق طريقة Turbo Visionيتميز بكونه لايُقيد المستثمر بمجموعة محددة من الخيارات، بل تجري المعالجة تَبَعاً لنوع الحدث الذي يقع، والذي تقوم بمعالجته طريقة خاصة method تسمى HandleEvent كما سنرى لاحقاً.

٢ـ هرمية الأغراض في Turbo Vision:

إن جميع الأغراض في Turbo Vision تنحدر من أصل مشترك وهو الغرض Object.

والأغراض في Turbo Vision على نوعين: أغراض تصنف بأنها عناصر الإظهار أو الرؤية views وأخرى ليس لها علاقة بالإظهار. فعنصر الإظهار يعبّر عن أي شيء يمكن إظهاره على الشاشة، وجميع هذه العناصر تنحدر من الغرض TView، وهي تتضمن قوائم الخيارات، و مستطيلات الحوار، والنوافذ، وقضبان الإزاحة، وعناصر نصية أخرى.

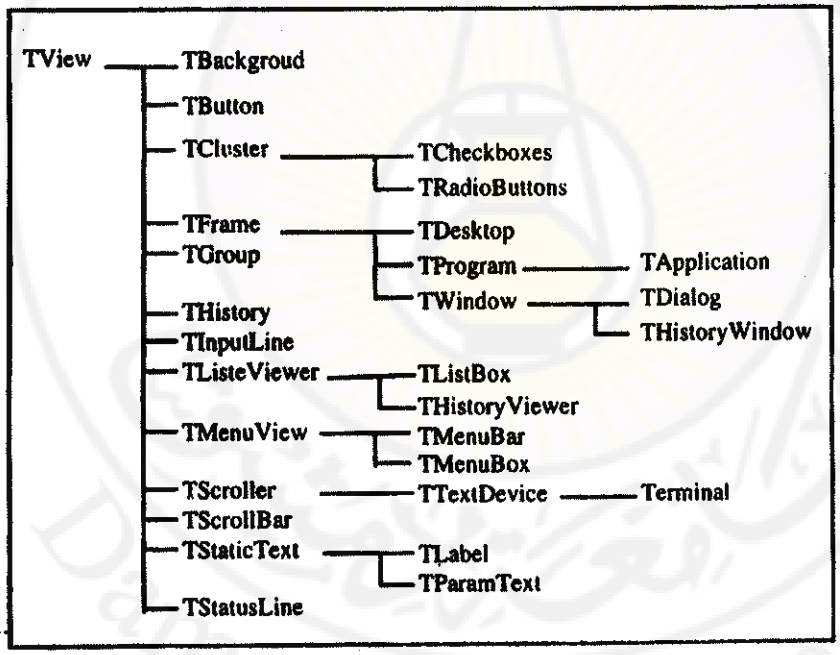
إن Turbo Visionتعمل في النمط المحرفي بدلاً من النمط البياني، لذا فهي تتعامل مع عناصر مستطيلة الشكل تُرسم على الشاشة.

ولما كانت عناصر إظهار Turbo Vision أغراضاً، فإنها مزودة بطرائقها الخاصة لمعالجة

المهام المسندة إليها.

من الضروري أن نشير هنا إلى أنّ عناصر الرؤية هي الوحيدة في Turbo Vision التي يسمح لها بالإظهارات على الشاشة، وجميع عناصر الرؤية يجب أن تكون قادرة على تهيئة نفسها، ورسم نفسها على الشاشة، وتغيير حجمها، أو موضعها، وإغلاق نفسها عندما تنتهي مهمتها.

يمكن رؤية هرميةِ عناصر الرؤية في المخطط التالي:



أما بقية أغراض Turbo Vision (غير عناصر الرؤية وهي: TCollection, TStream, TResourceFile, TStringList, TStrListMaker) فهي تقوم بمهام عديدة في الخلفية لتدعم عناصر الرؤية.

٢ـ بنية البرنامج في Turbo Vision:

١- ترويسة البرنامج

programHeading;

٢- فقرة التصريح عن الوحدات المستخدمة سواء من مكتبة Turbo Vision أو الوحدات الخاصة بنا

UsesTVunits , OurUnits;

٣- الثوابت المعرفة

Const declarations;

٤- نمط الغرض الخاص بتطبيقنا وهو مشتق من الغرض TApplication

TYPE Declare my applicatioh object as descendant of TApplication

٥- متحولات البرنامج وتتضمن:

Var

مؤشرات إلى الأغراض الديناميكية

pointers to dynamic object

والنموذج المعرف من نمط الغرض الخاص بالتطبيق

my application object instance

٦-الطرائق المعرفة على الأغراض الجديدة Methods for my application objects

أو الطرائق التي أعيدت كتابتها من أغراض موجودة سابقاً في المكتبة

استدعاء الباني	Call myApplication.Init
استدعاء الطريقة Run	Call myApplication.Run
استدعاء الهادم	Call myApplication.Done

ملاحظات:

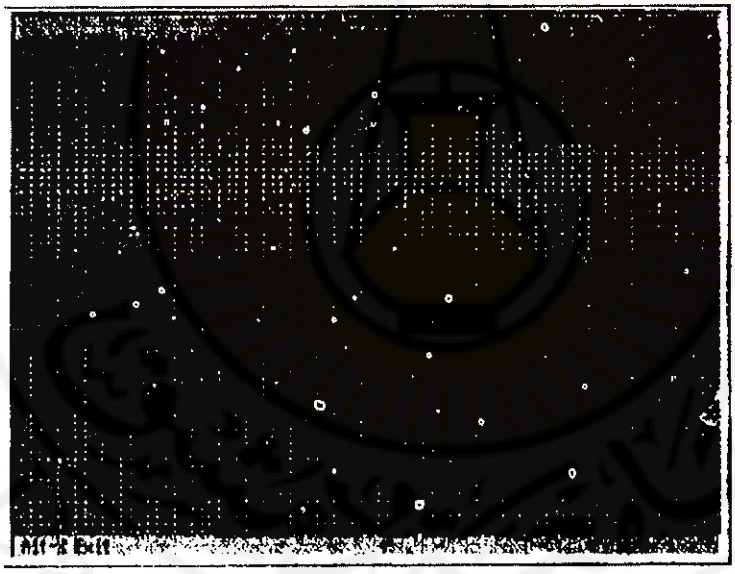
-myApplication هو اسم اصطلحنا عليه لتسمية تطبيقنا. يمكن المبرمجَ اختيارُ الاسم الذي يعبر عن تطبيقه وهو نموذج من غرض مشتق من الغرض TApplication.

- إن وحدات Turbo Vision هنا تمثل قائمة الوحدات التي نريد استخدامها من مكتبة Turbo Vision. ويُنصح عادة باستخدام الوحدات التالية لمعظم البرامج: app, dialogs, divers, menus, objects, views.

ونكتب الآن البرنامج الذي نريد بناءه:

```
programTVdemo1 ;
usesapp;
typeTMyApp= object(TApplication)
            end;
varMyApp:TMyApp;
begin
MyApp.Init;
MyApp.run;
MyApp.done;
end.
```

يعطي تنفيذ هذا البرنامج البسيط تطبيقاً له الواجهة التالية:



تظهر هنا نافذة شبيهة بتلك التي تظهر في واجهة Turbo Pascal قبل فتح أي ملف، ولكنها تخلو من أية قائمة حوار في السطر العلوي الذي يمثل قائمة الخياراتMenu Bar ، على حين يحتوي السطر السفلي والمسمى خط الحالةStatus Line على Alt-X Exit وذلك للتمكن من إغلاق هذه النافذة. وهذه هي الصفات العامة المشتركة لأي تطبيق: نافذة، وقائمة خيارات خالية، وخط حالة يتضمن فقط Alt-X Exit. لصنع تطبيق محدد علينا إذن إضافة قائمة خيارات تمثل الوظائف المختلفة التي يقوم بها التطبيق، ومحتويات خط الحالة الذي يتضمن اسم العملية وبجانبها الزر (أو الأزرار) التي يجب الضغط عليها لتحقيق هذه العملية.

٣ ـ بناء قائمة خيارات منسدلة وخط حالة:

سنضيف الآن إلى المثال السابق وحدات جديدة، ونزود التطبيق بقائمة خيار وخط حالة. لو نظرنا إلى الغرض TApplication لوجدنا أنه يرث من الغرض Tprogram الطريقتين التاليتين InitStatusLine, InitMenubar وهما طريقتان افتراضيتان يمكننا إعادة كتابة محتواهما (وفق مبدأ التحميل الزائد overriding)، ذلك أن الأولى تملأ قائمة الخيار والثانية تملأ خط الحالة. وفيما يلي ملخص الغرض TApplication.

TAPPLICATION OBJECT SUMMARY APP.TPU
Inheritance:
TApplication > TProgram > TGroup > TView > TObject

نعني هنا أن الغرض TApplication ابن الغرض TProgram الذي هو بدوره ابن الغرض

TGroup ابن الغرض TViewابن الغرض TObject.

```
Primary Methods
ConstructorInit ;
DestructorDone; Virtual;
ProcedureHandleEvent (Var Event: TEvent); Virtual;

Primary Inherited Methods from TProgram

ProcedureInitMenuBar ; Virtual ;
ProcedureInitStatusLine ; Virtual ;
ProcedureRun ; Virtual ;
FunctionValidView ( P : PView) : PView ;
FunctionValid ( Command : WORD) : BOOLEAN ;
FunctionCommandEnabled ( Command : WORD) : BOOLEAN ;
ProcedureDisableCommands ( Commands : TCommandSet) ;
ProcedureEnableCommands ( Commands : TCommandSet) ;
ProcedureGetExtent ( VAR Extent : TRect) ;
```

لايهمنا هنا فهم جميع هذه الطرائق وإنما معرفة أن InitMenuBar و InitStatusLine موروثة من TProgram وهي virtual فيمكن إعادة تعريفها في برنامجنا.

لنكتب الآن برنامجاً يقوم بملء قائمة الخيار وخط الحالة للتطبيق TmyApp:

```
ProgramTVdemo2;
Usesapp,objects,drivers,views,menus;
ConstcmNewWindow=101;
TypeTMyApp= object(TApplication)
        procedure InitMenuBar; virtual;
        procedure InitStatusLine; virtual;
       end;
(************methods  for TMyApp Object type*******)
ProcedureTMyApp.InitMenuBar;
```

```
VarR : TRect;
Begin
GetExtent(R);
 R.B.Y := 1;
MenuBar := new (PMenuBar, Init (R, NewMenu(
NewSubMenu ('~F~ile', hcNoContext, NewMenu (
NewItem ('~N~ew', 'F4', kbF4, cmNewWindow, hcNoContext,
NewLine (
NewItem ('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
Nil)))),
NewSubMenu ('~W~indow', hcNoContext, NewMenu (
NewItem ('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
NewItem ('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
Nil))),
Nil)))));
 end;

procedureTMyApp.InitStatusLine;
var  R : TRect;
GetExtent(R);
R.A.Y := R.B.Y - 1;
StatusLine := NEW (PStatusLine, Init (R,
NewStatusDef (0, $FFFF,
NewStatusKey ('', kbF10, cmMenu,
NewStatusKey ('~Alt-X~ Exit', kbAltx, cmQuit,
NewStatusKey ('~F4~ New', kbF4, cmNewWindow,
NewStatusKey ('~Alt-F3~ Close', kbAltF3, cmClose,
Nil)))),
Nil)
));
end;

(************Global variable declaration*******)
varmyApp:TMyApp;
```

```
(************main  body of Program *******)

begin
myApp.Init;
myApp.run;
myApp.done;
end.
```

إن أفضل طريقة لتعلّم Turbo Vision هي تفحص برنامج مكتوب يستخدم هذه المكتبة. وكما ذكرنا فإن البرنامج المذكور آنفاً يستخدم المكتبة لبناء برنامج مزود بقائمة خيار وخط حالة. وكما في Turbo Pascal فإن قائمة الخيارات تحتوي على كلمات يمثل كل منها خياراً يعتبر تجميعاً لعمليات ترد ضمن هذا الخيار. وإن الضغط على أحد هذه الخيارات الموجودة في قائمة الخيارات يؤدي إلى انسدال مستطيل يحوي خيارات فرعية. يسمى هذا النوع من القوائم بالقوائم المنسدلة.

لإضافة القوائم المنسدلة وخط الحالة، يجب أن نستخدم الوحدات Objects, Drivers, Views, Menus, APP، ثم نعرف ثابتاً يُستخدم كوسيط بين خيار فتح نافذة في قائمة، وبين الطريقة الحالية في الغرض "نافذة"، والتي تسمح بفتح النافذة.

إن نمط الغرض الوحيد في البرنامج هو TMyAPP وهو ابن الغرض TApplication. وسنعيد تعريف الطريقتين الافتراضيتين اللتين تقومان بإنشاء قائمة الخيار وخط الحالة. وقبل الخوض في التفاصيل نلفت الانتباه إلى أن جسم البرنامج يقتصر على ثلاثة أسطر: myAPP.Init, myAPP.Run, myAPP.Done.

هذه الطرائق التي يجب أن ينفذها النموذج myApp المأخوذ من TMyAPP موروثة من الغرض TApplication. وهنا تتضح أهمية الغرض الأب في أنه يأخذ على عاتقه الكثير من العمل، وليس على المبرمج أن يهتم بهذه الطرائق والعمليات الموروثة.

لنعد الآن إلى TMyAPP.InitMenuBar التي تهيئ قائمة الخيار المنسدلة.

إن قائمة الخيار هي مستطيل مؤلف من سطر واحد وعرضه ثمانون عموداً. لذا، ولتمثيل قائمة خيار، سنعرف متحولاً من النمط TRect وهو غرض إظهار موجود في Turbo Vision. يحدّد هذا الغرض بإحداثيات زاويته اليسرى العليا وزاويته السفلى اليمنى.

عندما نستدعي GetExtent الموروثة من TView، تجري تهيئة الغرض "مستطيل" ليملأ كامل الشاشة. يجب إذن أن نقوم بتقليص حجمه لجعل طوله يعدل سطراً واحداً في أعلى الشاشة، أي جعل إحداثيات b.y المماثلة لارتفاع الزاوية اليمنى الدنيا أقل بـ 1 من الزاوية اليسرى العليا وهي إحداثيات a.y ومن ثمّ نحصل على مستطيل مكون من سطر واحد، في أعلى الشاشة.

سنقوم بجعل هذا المستطيل قائمة خيارات باستدعاء طريقة من MenuBar وهنا يبدأ تعقيد Turbo Vision.

يسمح لنا Turbo pascal باستخدام الإجرائية New كتابع. ويمكننا أن نقوم بإعطاء قيمة ابتدائية للمتحول الديناميكي المنشأ في نفس التعليمة التي تسمح بإنشائه وهي تعليمة New على الشكل التالي: New (p, init (p)) ، وهنا، سنستخدم New كتابع لإنشاء غرض ديناميكي هو قائمة خيار باستخدام New(PMenuBar حيث PMenuBa هو نمط المؤشر على غرض من نمط قائمة خيار معرف في Turbo Vision ، و اسم Init r,NewMenu(التي تضع قائمة الخيار في الغرض "مستطيل" المأخوذ TRe. إن الجزء NewMenu من الاستدعاء هو الذي يبني القوائم المنسدلة المتصلة بالقائ رئيسية، ولبناء كل منها نستدعي الطريقة SubMenu . تأخذ هذه الطريقة أربعة وسطاء: اسم

الخيار، وثابت للدلالة على المساعدةhelp الموجودة على الشاشة التي تشرح هذا الخيار (إن وجدت)، ومؤشر نحو العنصر الأول في الخيار، و مؤشر نحو الخيار الفرعي التالي في القائمة المنسدلة. إن NewMenu و NewSubMenu و NewItem كلها توابع تعيد مؤشراً نحو أغراض من النمط خيار.

في كل استدعاء لـ NewSubMenu و NewItem يرد اسم الخيار الفرعي على شكل سلسلة محارف بين حاصرتين، ولجعل أحد الأحرف في سلسلة المحارف أكثر ظهوراً على الشاشة (أي بلون مختلف) نضعه بين مَدَّتين ، فمثلاً 'F~ile' تجعل الحرف F يبدو بلون مختلف على الشاشة في كلمة File. كذلك نستخدم ثوابت معرفة سلفاً في Turbo Vision للأزرار فمثلاً kbF4 تدل على أن الزر F4 يكافئ استدعاء الخيار File. أما الثوابت مثل cmNewWindow، فهي عادةً غير معرفة في Turbo Vision ويجب تعريفها في البرنامج. إنّ HcNoContext هنا يدل على عدم وجود معلومات مساعدة على الشاشة للخيارات المتوفرة. لدينا في نهاية الطريقة التي تعرّف قائمة الخيار سلسلة من Nil والأقواس بسبب الاستدعاءات المتداخلة.

لامجال للاستغناء عن هذه الطريقة في صنع قوائم الخيار. ومع الممارسة تصبح العملية أسهل بكثير مما تبدو عليه.

إنّ بناء خط الحالة يجري بطريقة مماثلة لبناء قائمة الخيار، فأولاً نعرّف متحولاً كمستطيل لخط الحالة، ثم نعرّف مؤشراً من نمط خط الحالة PStatusLine ونستدعي التابع New لإنشاء خط حالة ديناميكي. يمكن في هذا المثال تجاهل الوسيطين 0 و $FFF اللذين يعرّفان المساعدة المحلية Contextual Help.

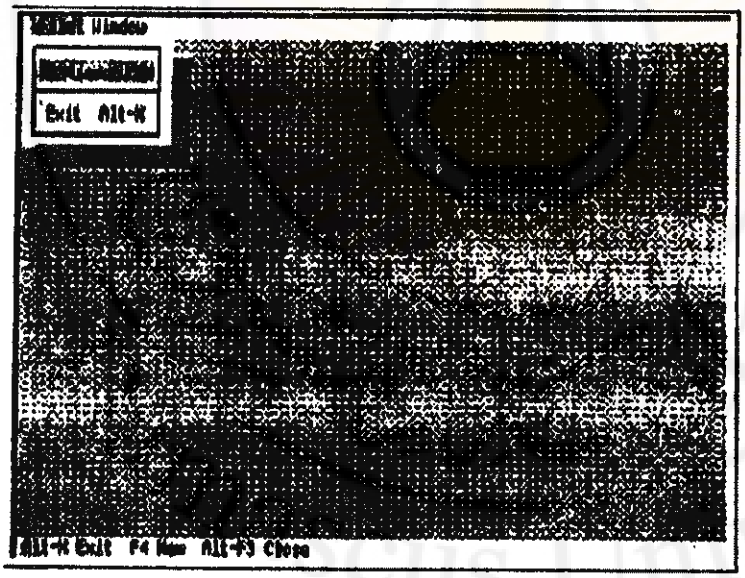
إن كل استدعاء لـ NewStatusKey يُظهر سلسلة محارف في خط الحالة، ويربط به زراً

من لوحة المفاتيح أو أكثر من زر في آن واحد.

في مثالنا هذا cmQuit هي أمر معرّف في Turbo Vision. وكما في الطريقة السابقة تنتهي الطريقة بسلسلة أقواس و Nil.

بعد إنهاء هاتين الطريقتين نعرّف myApp وهو متحول من النمط TMyApp، ونستدعي الطرائق الخاصة به في البرنامج الرئيسي. تقوم الطرائق Init و Run و Doneبتنفيذ النسخ الخاصة بالغرض الذي عرّفناه.

يبين الشكل التالي نتيجة تنفيذ البرنامج السابق:



٤- بناء نافذة على الشاشة:

سوف نهتم الآن بثلاث مسائل هامة:

١- كيف نربط أوامرَ محددة بقوائم خيار منسدلة.

٢- كيف نبني ونفتح نوافذَ على سطح الشاشة.

٣- كيف نظهر نصاً داخل هذه النوافذ .

يبين البرنامج التالي TVDemo3 هذه العمليات:

```
Program TVDemo3;
uses app,objects,drivers,views,menus;
const WinCount:Integer=0;
    cmNewWindow=101;

type Tmyapp=object(Tapplication)
        procedure HandleEvent(var Event:TEvent); virtual;
        procedure InitMenuBar; virtual;
        procedure InitStatusLine; virtual;
        procedure NewWindow;
      end;


PDemoWindow=^TDemoWindow;
TDemoWindow=object(TWindow)
              constructor Init(Bounds:TRect;WinTitle:string;
                               WindowNo:WORD);
              end;

PInterior=^TInterior;
```

```
TInterior=object(TView)
            constructor Init(var Bounds:TRect);
           procedure Draw;virtual;
            end;

(********************methods for TInterior*****************)
 constructor TInterior.Init(var Bounds:TRect);
  begin
    TView.Init(Bounds);
    GrowMode:=gfGrowHiX+gfGrowHiY;
    Options:=Options or ofFramed;
  end;


 procedure TInterior.Draw;
  const
     Greeting:string='Hello,new programmer!';
  begin
    TView.Draw;
    WriteStr(4,2,Greeting,$01);
  end;


(*****************methods for TDemoWindow************)
constructor TDemoWindow.Init(Bounds:TRect;WinTitle:string;
                                  WindowNo:word);

 var
    s:string;
    Interior:PInterior;
begin
   str(WindowNo,S);
   TWindow.Init(Bounds,WinTitle+' '+S,wnNoNumber);
  GetClipRect(Bounds);
   Bounds.Grow(-1,-1);
  Interior:=New(PInterior,Init(Bounds));
```

```
    Insert(Interior);
  end;

 (*******************methods for TMyApp*****************)
 procedure Tmyapp.HandleEvent(var Event:TEvent);
 begin
   TApplication.HandleEvent(Event);
   If Event.what=evCommand then
     begin
     case Event.Command of
        cmNewWindow:NewWindow;
        else
        Exit;
        end;
    ClearEvent(Event);
    end
end;


procedure Tmyapp.InitMenuBar;
var R : TRect;
begin
 GetExtent(R);
 R.B.Y := 1;
 MenuBar := new (PMenuBar, Init (R, NewMenu(
    NewSubMenu ('~F~ile', hcNoContext, NewMenu (
      NewItem ('~N~ew', 'F4', kbF4, cmNewWindow, hcNoContext,
     NewLine (
      NewItem ('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
     Nil)))),
   NewSubMenu ('~W~indow', hcNoContext, NewMenu (
     NewItem ('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
     NewItem ('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
    Nil))),
    Nil)))));
end;
```

```
procedure TMyApp.InitStatusLine;
var R : TRect;
begin
 GetExtent(R);
 R.A.Y := R.B.Y - 1;
 StatusLine := NEW (PStatusLine, Init (R,
   NewStatusDef (0, $FFFF,
     NewStatusKey ('', kbF10, cmMenu,
     NewStatusKey ('~Alt-X~ Exit', kbAltx, cmQuit,
      NewStatusKey ('~F4~ New', kbF4, cmNewWindow,
      NewStatusKey ('~Alt-F3~ Close', kbAltF3, cmClose,
      Nil)))),
    Nil)
    ));
end;


Procedure TMyApp.NewWindow;
 var
  window:PDemoWindow;
  r:TRect;

  begin
    INC(WinCount);
     r.Assign(0,0,32,7);
    RANDOMIZE;
    r.Move(RANDOM(47),RANDOM(16));
     Window:=New(PDemoWindow,Init(r,'Greeting
Window',WinCount));
     DeskTop^.Insert(Window);
  end;

(**************Global Variable declarations***************)
var myApp:TMyApp;
```

```
(*****************Main body of the program*****************)
begin
myApp.Init;
myApp.run;
myApp.done;
end.
```

سنهتم بتفصيل العمليات الأساسيةفي البرنامج.

إن أول مـايلاحظ في البـرنامج هو إضـافـة طريقـتين جـديدتين إلى الغـرض TMyApp الطريقة HandleEvent التي تقوم بقراءة الدخل ، والطريقة NewWindow التي تبني وتهيئ النافذة. كما سنعرّف غرضاً ابناً من TWindow وهو الغرض المعرّف سلفاً في Turbo Vision. وسنعرّف مؤشراً على النافذة التي سنستعملها من الغرض الجديد.

إن النافذة هي إطار قد يحتوي شيئاً، ولكنّه لايكون دوماً مرفقاً بمحتوى داخلي. فإذا أردنا وضع محتوى ضمن هذا الإطار، فعلينا أن نعرّفه بأنفسنا. ولهذا السبب عرّفنا غرضاً اسمه TInterior يُعبّر عن المحتوى الداخلي للإطار، ويستطيع أن يرسم نفسه داخل الغرض "نافذة"، كما عرّفنا مؤشراً على هذا النمط.

إن الباني للغرض TInterior يستدعي الطريقة Init من سلفه TView، وإن استخدام بعض الثوابت المعرفة في Turbo Vision يسمح بجعل هذا المحتوى يكبر ليملأ الشاشة أو يتقلص، أو يتحرك مع النافذة التي وضع داخلها.

تستدعي الطريقة Draw المعّرفة في TInterior الطريقة TView.Draw لمسح محتوى النافذة (أي إظهارها فارغةً)، ثم تستدعي الطريقة WriteStr لإظهار رسالة تحيّة ضمن النافذة.

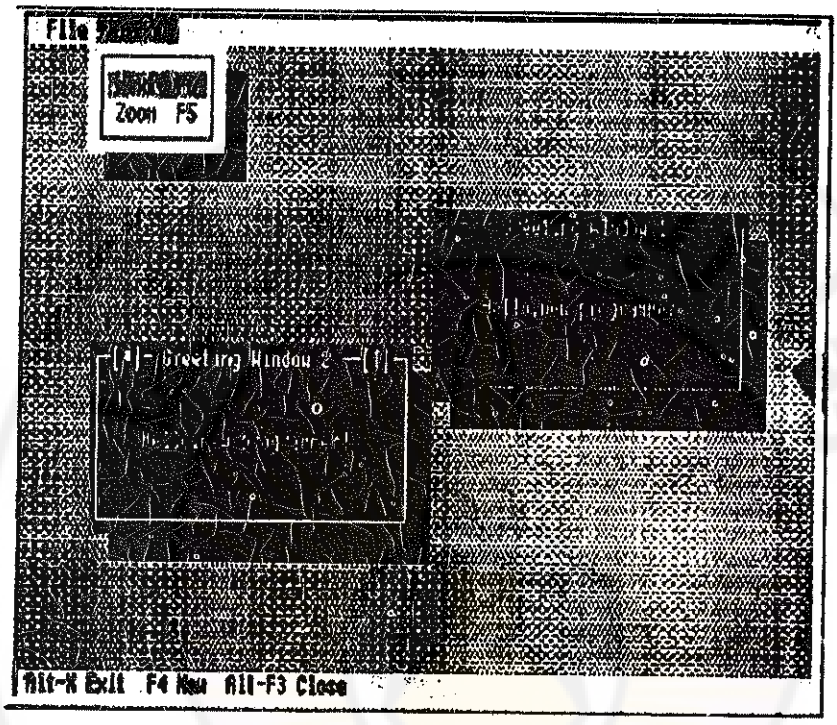
تقوم الطريقة Init الخاصة بـ TDemoWindow بتعريف مؤشر وتهيئته لربط النافذة (أي

الإطار) مع محتوى النافذة، ثم يُقحم المحتوى داخل الإطار.

تقوم الطريقة TMyApp.HandleEvent بواسطة تعليمة Case بتحديد الطريقة التي يجب أن تنفذ بحسب الثابت الذي يتلقاه البرنامج والذي يعبّر عن الأمر، وفي مثالنا هذا جرى تعريف ثابت واحد يقوم بتفعيل الطريقة NewWindow.

تهيئ الطريقة TMyApp.ViewWindow مؤشراً نحو الغرض "نافذة"، ثم تستدعي الطريقة Assign التابعة للغرض TRect لتهيئة مستطيل ذي حجم محدد هو حجم النافذة، ثم تقوم باستخدام مولّد أعداد عشوائية لاختيار موقع عشوائي على الشاشة للزاوية اليسرى العليا للمستطيل، ثم تستدعي New وتظهر النافذة على الشاشة مع استدعاء للطريقة Insert التابعة للغرض DeskTop المعرفة في Turbo Vision.

يبين الشكل التالي نتيجة تنفيذ البرنامج السابق:



٥ـ إضافة شريط إزاحة إلى النوافذ:

سنضيف الآن إلى النافذة التي بنيناها على الشاشة شريطي إزاحة، لنسمح باستعراض محتوى النافذة الداخلي من اليمين واليسار أو الأعلى والأسفل باستخدام الفأرة. كما سنعرض في النافذة محتوى ملف ما وليكن البرنامج الذي نستخدمه لبناء هذا التطبيق.

يقوم البرنامج TVDemo4 بهذه العمليات.

```
Program TVDemo4;
Uses app,objects,drivers,views,menus;
Const SourceFile='TVDemo4.pas';
MaxLines=200;
WinCount:Integer=0;
cmNewWindow=101;
(*****************Global variable declarations***************)
Var LineCount:INTEGER;
Lines:array[0..MaxLines-1] of PString;

(*****************Object type declarations***************)

Type   Tmyapp=Object(Tapplication)
                procedure HandleEvent(var Event:TEvent); virtual;
                procedure InitMenuBar; virtual;
                procedure InitStatusLine; virtual;
                procedure NewWindow;
               End;


PDemoWindow=^TDemoWindow;

TDemoWindow=object(TWindow)
              constructor Init(Bounds:TRect;WinTitle:string;
                               WindowNo:WORD);
              procedure MakeInterior(Bounds:TRect);
              end;

PInterior=^TInterior;

TInterior=object(TScroller);
        constructor Init(var Bounds:TRect;
                         AHScrollBar, AVScrollBar: PScrollBar);
        procedure Draw;virtual;
        end;
```

```
(****miscellaneous global procedures not tied to objects types******)
procedure ReadFile;
var FileToRead:Text;
LineString:string;
begin
 LineCount:=0;
 Assign(FileToRead,sourceFile);
 {$I-}
 reset(FileToRead);
 {$I+}
 If IOResult<>0 then
       begin
         writeln(sourceFile,'not found');
        halt(1);
       end;
 while not eof(FileToRead) and (LineCount<MaxLines) do
       begin
        readln(FileToRead,LineString);
        Lines[LineCount]:=NewStr(LineString);
       Inc(LineCount);
       end;
  close(FileToRead);
end;{ of readfile procedure}

procedure DoneFile;
var counter:INTEGER;
begin
   for counter:=0 to LineCount-1 do
      if Lines[counter]<>NIL then
        DisposeStr(Lines[counter]);
end;

(***************methods for TInterior object type*************)
constructor TInterior.Init(var Bounds:TRect;AHScrollBar,
                           AVScrollBar:PScrollBar);
```

```
begin
TScroller.Init(Bounds,AHScroliBar,AVScrollBar);
GrowMode:=gfGrowHiX+gfGrowHiY;
Options:=Options or ofFramed;
Setlimit(100,LineCount);
end;

procedure TInt( rior.Draw;
var Color:Byte;
i,y:integer;
b:TDrawBuffer;
begin
Color:=GetColor(1);
for y:=0 to Size.y-1 do
   begin
      MoveChar(b,' ', Color,Size.x);
      i:=Delta.y+y;
      if (i<LineCount) and (Lines[i]<>NIL) then
         MoveStr(b,Copy(Lines[i]^,Delt  1,Size.x),Color);
      WriteLine(0,y,Size.x,1,b);
   end
end;

(***********methods for TDemoWindow object type***********)
constructor TDemoWindow.Init(Bounds:TRect;WinTitle:string;
                             WindowNo:WORD);

var
NumString:string[3];
begin
str(WindowNo,NumString);
TWindow.Init(Bounds,WinTitle+' '+NumString,wnNoNumber);
MakeInterior(Bounds);
end;

procedure TDemoWindow.MakeInterior(Bounds:TRect);
var  HScrollBar,VScroli  PScrollBar;
```

```
        Interior:PInterior;
        r:TRect;
begin
VScrollBar:=StandardScrollBar(sbVertical+sbHandleKeyboard);
HScrollBar:=StandardScrollBar(sbHorizontal+sbHandleKeyboard);
GetExtent(Bounds);
Bounds.Grow(-1,-1);
Interior:=New(PInterior,Init(Bounds,HScrollBar,VScrollBar));
Insert(Interior);
end;

(**********methods for TMyApp object type*************)
procedure Tmyapp.HandleEvent(VAR Event:TEvent);
begin
TApplication.HandleEvent(Event);
If Event.what=evCommand then
   begin
      case Event.Command of
       cmNewWindow:NewWindow;
        else
       Exit;
    end;
ClearEvent(Event);
end
end;

procedure Tmyapp.InitMenuBar;
var R : TRect;
begin
GetExtent(R);
R.B.Y := 1;
MenuBar := NEW (PMenuBar, Init (R, NewMenu(
NewSubMenu ('~F~ile', hcNoContext, NewMenu (
NewItem ('~N~ew', 'F4', kbF4, cmNewWindow, hcNoContext,
NewLine (NewItem ('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
Nil)))),
```

```
NewSubMenu ('~W~indow', hcNoContext, NewMenu (
NewItem ('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
NewItem ('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
Nil))),
Nil)))));
end;

procedure TMyApp.InitStatusLine;
var R : TRect;
begin
GetExtent(R);
R.A.Y := R.B.Y - 1;
StatusLine := NEW (PStatusLine, Init (R,
NewStatusDef (0, $FFFF,
NewStatusKey ('', kbF10, cmMenu,
NewStatusKey ('~Alt-X~ Exit', kbAltx, cmQuit,
NewStatusKey ('~F4~ New', kbF4, cmNewWindow,
NewStatusKey ('~Alt-F3~ Close', kbAltF3, cmClose,
Nil)))),
Nil)
));
end;

procedure TMyApp.NewWindow;
var
window:PDemoWindow;
r:TRect;
begin
inc(WinCount);
r.Assign(0,0,32,7);
randomize;
r.Move(random(47),random(16));
Window:=New(PDemoWindow,Init(r,'Greeting Window',WinCount));
DeskTop^.Insert(Window);
end;
```

```
(**************More Global Variable declarations***************)
var myApp:TMyApp;

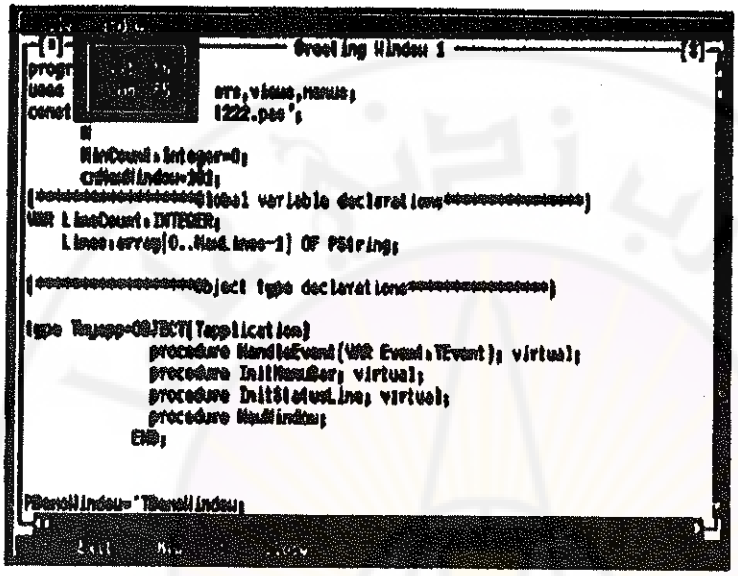
(******************Main body of the program*****************)
begin
ReadFile;
myApp.Init;
myApp.run;
myApp.done;
DoneFile;
end.
```

لقد قمنا في البرنامج بتعديل رئيسي وهو تغيير أبي الغرض TInterior. فقد كان في TVDemo3 ابناً لـ TView، أما الآن فهو ابنٌ لـ TScroller ، وهذا مما يسمح لنا بالتجول ضمنه باستخدام أشرطة الإزاحة.

إن الإجرائيات ReadFile و Done لا ترتبط بأي غرض. فالأولى تفتح ملفاً نصياً وتشحنه ضمن محتوى النافذة، والثانية تُفرِّغ محتوى النافذة عندما تغلق. إن عملية شحن ملف داخل غرض تجري بإنشاء جدول من المؤشرات نحو سلاسل محارف النص، حيث تقوم كل سلسلة بتخزين سطر من الملف.

أما أشرطة الإزاحة نفسها فهي أغراض ديناميكية تُبنى باستدعاء الطريقة TDemoWindow.MakeInterior التي تستخدم الأغراض الممثلة لأشرطة الإزاحة المعرّفة في Turbo Vision.

يبين الشكل التالي نتيجة تنفيذ البرنامج السابق:



خلاصة:

رأينا في الأمثلة الواردة ضمن هذه الفقرة، أن مكتبة Turbo Vision غنية بالأغراض وذات قدرة كبيرة على مساعدتنا في بناء التطبيقات، فهي تسمح لنا بإضافة قوائم خيارات منسدلة، ومستطيلات حوار، ومعالَجة إدخالات الفأرة، وخواص أخرى كثيرة لم تُذكر في أمثلتنا.

هذه المكتبة معقدة جداً، ولمزيد من المعلومات يجب الرجوع إلى دليل Turbo Vision.

أمثلة عامة

مثال1- إصدار نتائج الدوري

يجري الدوري بين عدد من الفرق بمباريات بين جميع الفرق المشاركة. فكل فريق يجري مباراتين مع كل من الفرق الأخرى، واحدة على أرضه وأخرى على أرض الخصم.

تُحسب نقاط المباريات (العلامات) وفق القاعدة التالية:

الربح يعطي ثلاث نقاط

التعادل يعطي نقطة واحدة

الخسارة صفر

المطلوب كتابة برنامج ينظم إصدار نتائج الدوري.

إن المطلوب غير مُوَّصفٍ توصيفاً جيداً ومحدَّداً. إذ لابد من تحديد مدخلات المسألة ومخرجاتها بدقة.

التوصيف الدقيق للمسألة:

- المعطيات:

أسماء الفرق

نتائج المباريات

- النتائج:

جدول بترتيب الفرق وفقاً لعدد النقاط

جدول بترتيب الفرق وفقاً لعدد الأهداف

والآن إذا تأملنا المعطيات نرى أن تحديد معلومات نتائج المباريات أمر هام. بالطبع هناك طرفا المباراة: الفريق الأول والفريق الثاني من جهة، وعدد أهداف الفريق الأول وعدد أهداف الفريق الثاني في المباراة من جهة أخرى. ولكن إذا ثبتّنا الدخل بهذه الطريقة مباشرةً، سنلاقي عدداً من الإرباكات في إدخال المعطيات إلى برنامجنا للمعالجة، خصوصاً بسبب وجود مباراتين لا مباراة واحدة بين الفريقين وضرورة إعطاء أسماء الفريقين في كل مباراة.

سنعالج موضوع إدخال المعطيات كما يلي:

أولاً: سنحدّد طريقة الدخل بأنها من ملفات نصية، وليست تخاطبية من وحدة الدخل/الخرج النظامية، والسبب هو سهولة كتابة نتائج المباريات، لعدد كبير من الفريق في ملف نصي ننشئه في أي محرر نصوص، ويمكن تدقيقه وتعديله بيسر، ويهوّن ذلك اختبار البرنامج على معطيات متعددة.

ثانياً: توخياً لسهولة التعامل مع المعطيات في برنامجنا، نعطي الفرق المشاركة أرقاماً.

سيكون لدينا ملف نصي أول يحوي أرقام الفرق وأسماءها.

أما دخل نتائج المباريات، فنستخدم فيه أرقام الفرق التي اعتمدناها بدلاً من استخدام أسمائها.

ونضع النتيجة على سطر بالشكل التالي (ونقرأ من اليسار إلى اليمين) :

‹عدد أهداف الفريق ب› ‹عدد أهداف الفريق أ› ‹رقم الفريق ب› ‹رقم الفريق أ›

أي إن نتيجة المباراة بين الفريق ‹رقم الفريق أ› و الفريق ‹رقم الفريق ب› التي جرت على أرض الفريق ‹رقم الفريق أ› كانت بعدد أهداف ‹عدد أهداف الفريق أ› أما عدد أهداف

الفريق الخصم في هذه المباراة فهو <عدد أهداف الفريق ب>

مثال على الدخل بهذا الأسلوب :

أرقام الفرق وأسماءها في الملف TEAMNAME.TXT

* TEAMNAME.TXT : team-number team-name

1 Karama
2 Wathba
3 Ethad
4 Majed

ملف دخل نتائج المباريات ونسقه بالشكل:

TEAMGOAL.TXT

* TEAMGOAL.TXT : team-number1 team-number2 goals1 goals2

1 2 4 1
1 3 3 2
1 4 2 2
2 1 4 5
2 3 1 1
2 4 3 2
3 1 0 0
3 2 5 2
3 4 3 2
4 1 0 1
4 2 5 5
4 3 1 2

بنى المعطيات المستخدمة في البرنامج :

بهدف خزن نتائج المباريات، نعرف مصفوفة ببعدين أدلتها أرقام الفرق، وعناصرها تسجيلة تحوي عدد أهداف المباراة للفريقين في المباراة (انتظر لتعرف أي مباراة منهما!).

نعرف نمط هذه البنية بالشكل:

```
Type TGoals = array[1..MaxTeam, 1..MaxTeam] of
                    record
                      homegl, hostgl :integer;
                    end;
```

وهكذا يمكن للمتحول (المصفوفة) من ذاك النمط VGoals استيعاب نتائج كافة مباريات الدوري، كما يلي:

العنصر (التسجيلة) VGoals[i,j] يحوي نتيجة المباراة بين الفريق i والفريق j على ملعب i . عدد أهداف i في هذه المباراة هو VGoals[i,j].homegl وعدد أهداف الفريق j فيها هو VGoals[i,j].hostgl.

وهكذا عبرنا في التعليق في البرنامج:

(* VGoals[i,j].homegl=gi & VGoals[i,j].hostgl=gj <=> input: i j gij *)

أي في حالة دخل نتيجة المباراة هو: i j gi gj (المباراة بين i و j على ملعب i حقق فيها i أهدافاً عددها gi و حقق j أهدافاً عددها gj) نخزن في مصفوفة المباريات (مصفوفة أهداف المباريات):

VGoals[i,j].homegl=gi & VGoals[i,j].hostgl=gj

أما البنية الهامة في المعالجة فهي البنية التي تخزن كل المعلومات عن الفريق، ونعرّفها في البرنامج بنمط التسجيلة TInfoTeam وتحوي المعلومات (الحقول) التالية: رقم الفريق TmNb، اسمه TmName، عدد نقاطه الكلي TScore، عدد أهدافه الكلي TGoals، عدد مباريات الفوز على أرضه TWinMatchHome ، عدد مباريات الفوز خارج أرضه

.TWinMatchOut

```
Type TInfoTeam = record
        TmNb   : integer;
        TmName : array[1..20] of char; (* STRING in TP7 *)
        TScore: integer;
        TGoals : integer;
        TWinMatchHome : integer;
        TWinMatchOut  : integer;
        end;
```

والعمل في البرنامج يجري على النحو التالي:

تجري المعالجة على متحولات كلية:

```
VTabTeam : TTabTeam;    {جدول معلومات الفرق}
VGoals   : TGoals;      { مصفوفة الأهداف}
```

1- قراءة المعطيات (ملفات المعطيات) وخزنها في متحولات البنى السابقة، باستخدام الإجرائية:

```
Procedure GetData(Var N :integer );
```

المعامل الوحيد للإجرائية هو N عدد الفرق المشاركة ونستخلصه من عدد الأسطر في ملف أسماء الفرق.

2- استخلاص المعلومات بناءاً على المعطيات، أي ملء حقول تسجيلات معلومات الفرق من النمط TInfoTeam، باستخدام الإجرائية:

```
Procedure GetInformation;
```

3- إظهار جدول معلومات الفرق، باستخدام الإجرائية:

Procedure DisplayTeams;

4- فرز جدول معلومات الفرق مرتبةً وفق عدد النقاط، باستخدام الإجرائية:

Procedure SortTeamsScore;

5- فرز جدول معلومات الفرق مرتبةً وفق عدد الأهداف، باستخدام الإجرائية:

Procedure SortTeamsGoal;

تتعامل الإجرائيات السابقة جميعها مع المتحول العمومي VTabTeam : الجدول الحاوي لمعلومات الفرق.

```
Program Champion( input, output );
(*** Compute Scores of the Teams in a Championship ***)
(*
 * Data are in files with format:
 *   TEAMNAME.TXT :  team-number  team-name
 *   TEAMGOAL.TXT :  team-number1  team-number2  goals1  goals2
 *              - match played in team-number1's ground -
 * Score Rule:
 *        3 points   win
 *        1 point    equality
 *        0 point    lost
 * OUPUT :
 *  - total matches gained and lost on the team's ground.
 *  - order of teams by score
 *  - order of teams by goal numbers
 * Listing is a table with columns :
 * team-number  team-name  score  goals match-win-in  match-win-out
 *)
```

```
Const  MaxTeam=25;

Type

 TGoals = array[1..MaxTeam, 1..MaxTeam] of
                          record
                            homegl, hostgl :integer;
                          end;
 (* when VGoals is a variable of type TGoals then     *)
 (* VGoals[i,j].homegl=gi & VGoals[i,j].hostgl=gj <=> *)
 (*          input:  i  j  gi gj                      *)

 RInfoTeam =  record
           TmNb   : integer;
           TmName : String[20];
           (* array[1..20] of char; in Wirth-Pascal *)
           TScore: integer;
           TGoals : integer;
           TWinMatchHome : integer;
           TWinMatchOut  : integer;
          end;

 TTabTeam = array[1..MaxTeam] of RInfoTeam;

Var
   VTabTeam : TTabTeam;
   VGoals   : TGoals;
   N :integer;


Procedure  GetData(Var N :integer );
Var
   fteam, fgoals   : text;
   i :integer;
   fnb,snb :integer;
```

```
Begin
   assign(fteam,'teamname.txt');
   reset(fteam);
   i:=0;
   while (not eof( fteam ) ) do
    begin
     i:= i + 1;
     with  VTabTeam[i] do
        readln(fteam, TmNb,TmName);
    end;
   N:= i;
   close(fteam);
   assign(fgoals,'teamgoal.txt');    reset(fgoals);
   while (not eof( fgoals ) ) do
     readln(fgoals, fnb,snb,
           VGoals[fnb,snb].homegl,VGoals[fnb,snb].hostgl);
  writeln(' Data OK');
end (* GetData *);

Procedure GetInformation;
var i,j : integer;
begin
   for i := 1  to  N  do (* N globale Variable *)
     with  VTabTeam[i] do
        begin
          TScore := 0;       TGoals := 0;
          TWinMatchHome:=0;
          TwinMatchOut :=0;
        end;

  for i:=1 to N  do
    for  j:=1 to  N do
     if ( i<>j ) then
      begin
       VTabTeam[i].TGoals:=VTabTeam[i].TGoals + VGoals[i,j].homegl
                                  + VGoals[j,i].hostgl;
```

```
        if VGoals[i,j].homegl > VGoals[i,j].hostgl then
            with VTabTeam[i] do
            begin
              TScore := Tscore + 3;
              TWinMatchHome := TWinMatchHome + 1;
            end

          else

           if VGoals[i,j].homegl < VGoals[i,j].hostgl then
            with VTabTeam[j] do
            begin
              TScore := Tscore + 3;
              TWinMatchOut := TWinMatchOut + 1;
            end

           else
            begin
              VTabTeam[i].Tscore := VTabTeam[i].Tscore + 1;
              VTabTeam[j].Tscore := VTabTeam[j].Tscore + 1;
            end;
       end;
end(* GetInformation *) ;

Procedure DisplayTeams;
Var i : integer;
begin
 writeln('TmNb  TmName     TScore  TGoals TWinMatchHome
TwinMatchOut');
 for i:=1 to N do
   with  VTabTeam[i] do
    writeln(TmNb:4, TmName,'          ', TScore:2, TGoals:6,
        TWinMatchHome:8,TwinMatchOut:14);
end; (* DisplayTeams *)
```

```
Procedure SortTeamsScore;
var i,j : integer;
  VR : RInfoTeam;
begin
   for i := 1 to N-1 do (* N globale Variable *)
    for j:=1 to N-i do
      if VTabTeam[j].TScore < VTabTeam[j+1].TScore then
       begin
        VR := VTabTeam[j];
        VTabTeam[j] := VTabTeam[j+1];
        VTabTeam[j+1] := VR;
       end;
end; (* SortTeamsScore *)

Procedure SortTeamsGoal;
var i,j : integer;
  VR : RInfoTeam;
begin
   for i := 1 to N-1 do (* N globale Variable *)
    for j:=1 to N-i do
      if VTabTeam[j].TGoals < VTabTeam[j+1].TGoals then
       begin
        VR := VTabTeam[j];
        VTabTeam[j] := VTabTeam[j+1];
        VTabTeam[j+1] := VR;
       end;
end; (* SortTeamsGoal *)

Begin
  GetData( N );
  GetInformation;
  writeln; writeln(' Teams Results '); writeln;
  DisplayTeams;
  SortTeamsScore;
  writeln; writeln(' Teams Results : Ordered by Scores'); writeln;
  DisplayTeams;
```

```
  SortTeamsGoal;
  writeln;  writeln(' Teams Results : Ordered by Goals'); writeln;
  DisplayTeams;
End.
```

ملف دخل أسماء الفرق وأرقامها، ونسقه بالشكل:

* TEAMNAME.TXT : team-number team-name

```
1 Karama
2 Wathba
3 Ethad
4 Majed
```

ملف دخل نتائج المباريات ونسقه بالشكل:

* TEAMGOAL.TXT : team-number1 team-number2 goals1 goals2

```
1 2 4 1
1 3 3 2
1 4 2 2
2 1 4 5
2 3 1 1
2 4 3 2
3 1 0 0
3 2 5 2
3 4 3 2
4 1 0 1
4 2 5 5
4 3 1 2
```

نتيجة تنفيذ البرنامج:

```
Data OK

 Teams Results

TmNb   TmName    TScore   TGoals TWinMatchHome TwinMatchOut
   1   Karama       14       15            2            2
   2   Wathba        5       16            1            0
   3   Ethad        11       13            2            1
   4   Majed         2       12            0            0

 Teams Results : Ordered by Scores

TmNb   TmName    TScore   TGoals TWinMatchHome TwinMatchOut
   1   Karama       14       15            2            2
   3   Ethad        11       13            2            1
   2   Wathba        5       16            1            0
   4   Majed         2       12            0            0

 Teams Results : Ordered by Goals

TmNb   TmName    TScore   TGoals TWinMatchHome TwinMatchOut
   2   Wathba        5       16            1            0
   1   Karama       14       15            2            2
   3   Ethad        11       13            2            1
   4   Majed         2       12            0            0
```

مثال2- الرسم العام لمنحني تابع y=f(x)

نريد كتابة برنامج عمومي يسمح برسم التوابع الديكارتية y=f(x) بقطع النظر عن علاقة تعريف التابع.

سيكون لدينا في البرنامج تعريف التابع المطلوب رسمه معطى بالترويسة التالية:

```
function f( x:real):real;
```

والبرنامج مكتوب بحيث يقوم برسم منحني هذا التابع مهما يكن تعريفه، إذ يمكن إعادة استخدام البرنامج نفسه في حال تابع جديد: نغيّر نص التابع فقط، ويبقى الباقي على حاله دون أي تغيير، نترجم البرنامج ونجري التنفيذ فيرسم التابع الجديد.

المعطيات الأخرى اللازمة لإنجاز المطلوب يدخلها المستخدم وهي:

x1,x2 : بداية ونهاية مجال السينات الذي سيرسم التابع ضمنه.

dx : خطوة الرسم.

S: عبارة نصية تشكّل عنواناً يظهر على الرسم.

القيد الذي نضعه هو أن عدد النقاط، التي ستُحسب للتابع، يجب ألا يتجاوز حداً ثابتاً، وليكن 300. هذا القيد يأتي من فكرة محتملة لاستخدام جدول (array) لخزن قيم التابع قبل رسمه.

الخوارزمية أو خطوات العمل:

-{ قراءة معطيات المسألة: مجال الرسم، خطوة الرسم، عنوان الشكل}:

```
readln(x1,x2,dx,title);
```

- {تحديد عدد نقاط الرسم N وتثبيت الخطوة}:

إذا ((N ناتج قسمة المجال على الخطوة) أكبر من (الحد الأعظم NMax)) **نفّذ**

| اجعل N مساوية للقيمة NMax، واحسب الخطوة dx على هذا الأساس

- {حساب قيم التابع f(x) وخزنه في الشعاع TY}

- {حساب القيمة الصغرى والعظمى للتابع (للشعاع) TY}

نستخدم إجرائية MinMax تعطينا المطلوب: ymin, ymax.

أما حدود X فهي معطاة بالشكل:

xmin:=x1; xmax :=x1+N*dx

والآن نكون قد قمنا بالحساب الأساسي اللازم لتحديد الإطار الحاوي لما نريد رسمه، فقد حُدِّدت إحداثيات المستطيل المُغلِّف للشكل أي نافذة الرسم Window.

ونبدأ طور رسم البياني:

- {التأهيل البياني}

GUInitGraph;

- { تحديد الإسقاط : النافذة وبوابة الرؤية}

GUWindowView(xmin, ymin, xmax, ymax , 100,50, 550, 450);

- { رسم المستقيمات المتعاقبة للمنحني: Xi = x1 + i*dx و Yi =TY[i] }

تعطينا الخطوات السابقة رسم منحني التابع. ونكرر رسم أي تابع (يكفي تغيير نص صيغة التابع في نص البرنامج وإعادة التنفيذ).

وما تبقى من البرنامج هو إضافات توضيحية على الرسم:

- رسم العنوان
- رسم محور السينات إذا كان للتابع قيم سالبة وموجبة
- رسم قيم حدود الشكل للسينات والعينات

"الرسم" الكتابي الذي نعنيه هنا هو إظهار عبارة نصية في موقع على الرسم:

GUText(x,y, S) : x,y الموقع ، S النص المطلوب رسمه.

هنا يجب حساب الموقع "النسبي" لإظهار النص، وهذا مافعلناه بحساب أبعاد الشكل على المحورين.

RX := xmax-xmin; :المجال على X

RY := ymax-ymin; :المجال على Y

ومن ثَمَّ نعين موقع إظهار العنوان title مثلاً :

على السينات ابتداءً من ثلث المستطيل: xst:=xmin + RX/3;

وعلى العينات فوق النافذة بنحو العشر: yst := ymax + RY/10;

أما إظهار قيمة عددية على الشكل فهو متعذر ما لم نحوّل العدد إلى نص أي من real إلى String، وهذا ما تنجزه الإجرائية STR الموجودة مع Turbo-Pascal؛ مثال ذلك:

str(ymin:8:2,st); تقوم بتحويل القيمة ymin إلى السلسلة الحرفية st وبالنسق 8:2 أي على ثماني مسافات ، منهما خانتان بعد الفاصلة.

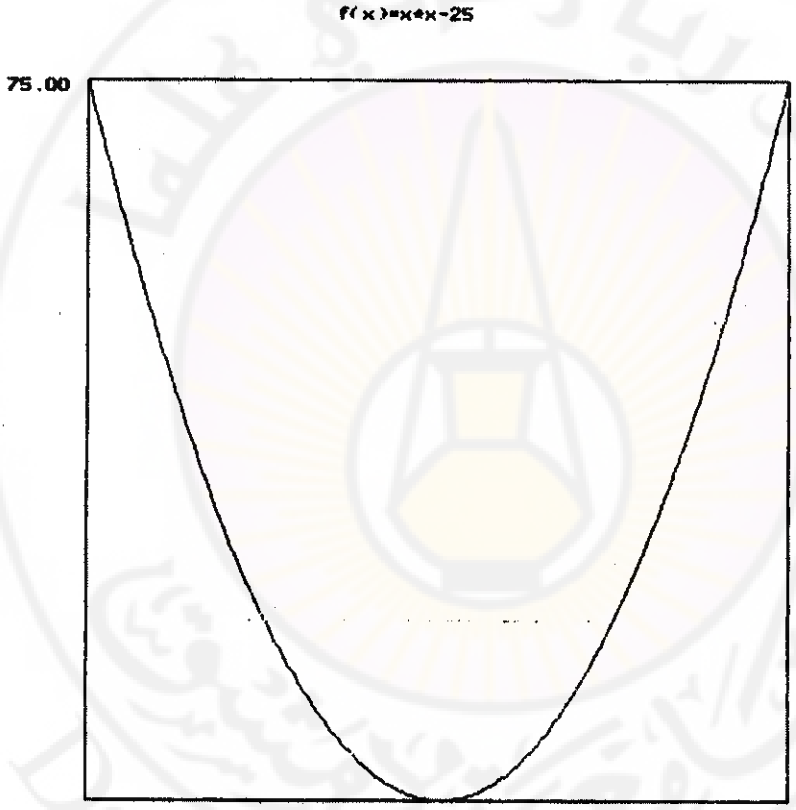
فيما يلي إظهار للتابع: f(x)= x*x -25

يليه النص الكامل للبرنامج. وفيه يكفي تغيير جسم التابع :

```
Function F(x:real):real;
begin
  F := x*x - 25;
end;(* F *)
```

إلى أي تابع نريد، وإعادة التنفيذ لنُظهر منحني التابع الجديد. وهذا ما فعلناه على توابع أخرى ترى أشكالها بعد نص البرنامج.

اقرأ نص البرنامج جيداً بعد فهمك لخطوات الحل وتأمل فيه ملياً، واتخذ منه مثالاً لكتابة برامجك بصيغة أنيقة وعمومية. لاحظ أن الجهد المبذول هو نفسه تقريباً في حال اقتصارنا على مسألة رسم منحنٍ لتابع وحيد لانعرف حدوده سلفاً. أما وقد أطلقنا العمومية في طريقة الحل بوضع متحولات البرنامج لأي تابع، فقد أعطينا لبرنامجنا شمولية رائعة ، وصار بين أيدينا أداة سهلة لتعرّف شكل التابع الذي نريد.


f(x)=x*x-25
75.00

```
Program GUCurve;
Uses GU;

Const NMax = 300;
Type  TTab = array[0..NMax] of real;
Var
 xmin,ymin,xmax, ymax :real;
 x1,x2,dx, xst,yst, RX,RY :real;
 i, N : integer;
 TY : TTab;
 st,title : String;

Function F(x:real):real;
begin
  F := x*x - 25;
end;(* F *)

Procedure MinMax(N: integer; VT : TTab; var vmin, vmax :real);
var i: integer;
begin
 vmin:=VT[0];  vmax:=VT[0];
 for i:=1 to N do
  begin
  if VT[i] < vmin then
      vmin := VT[i];
  if VT[i] > vmax then
      vmax := VT[i];
  end;
end (* MinMax *);

Begin
 (* Get: first x, last x, step, title *)
 writeln(' Input: x1  x2  dx  title ');
 readln(x1,x2,dx,title);
```

```
(* Compute Number of points to draw the curve *)
N:= round( (x2-x1)/dx );
if N >=NMax then
 begin
   dx := (x2-x1) / NMax;
   N  := NMax;
 end;
(* Compute the function f and store the values in the table TY *)
for i:=0 to N do
     TY[i] := f(x1+i*dx);

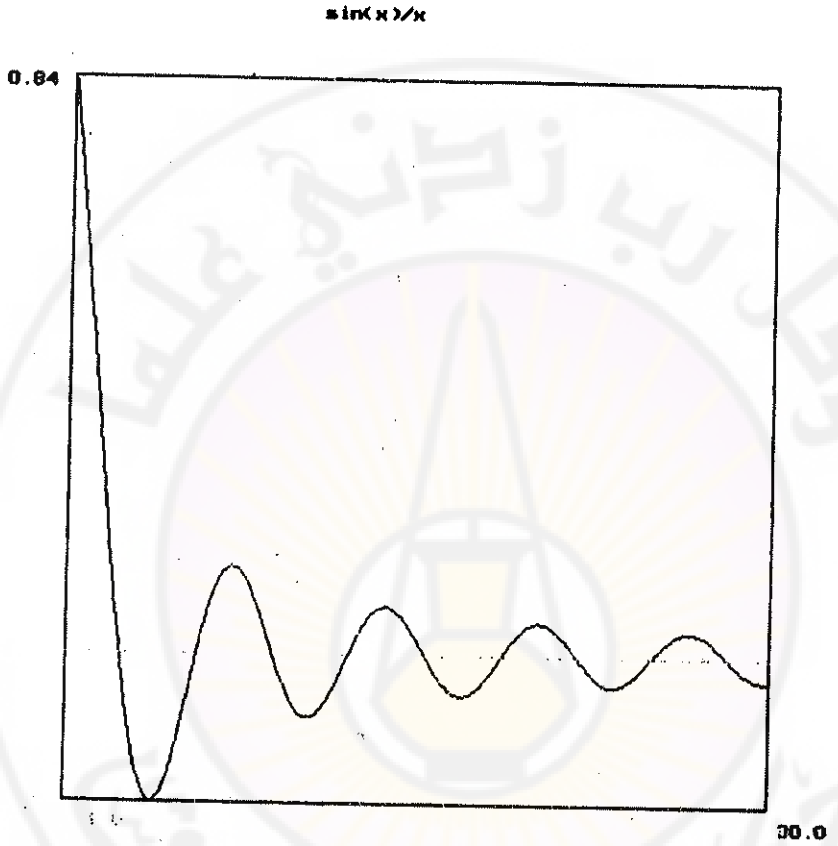
xmin:=x1; xmax :=x1+N*dx;
(* Get Min and Max of the function *)
MinMax(N, TY, ymin, ymax );

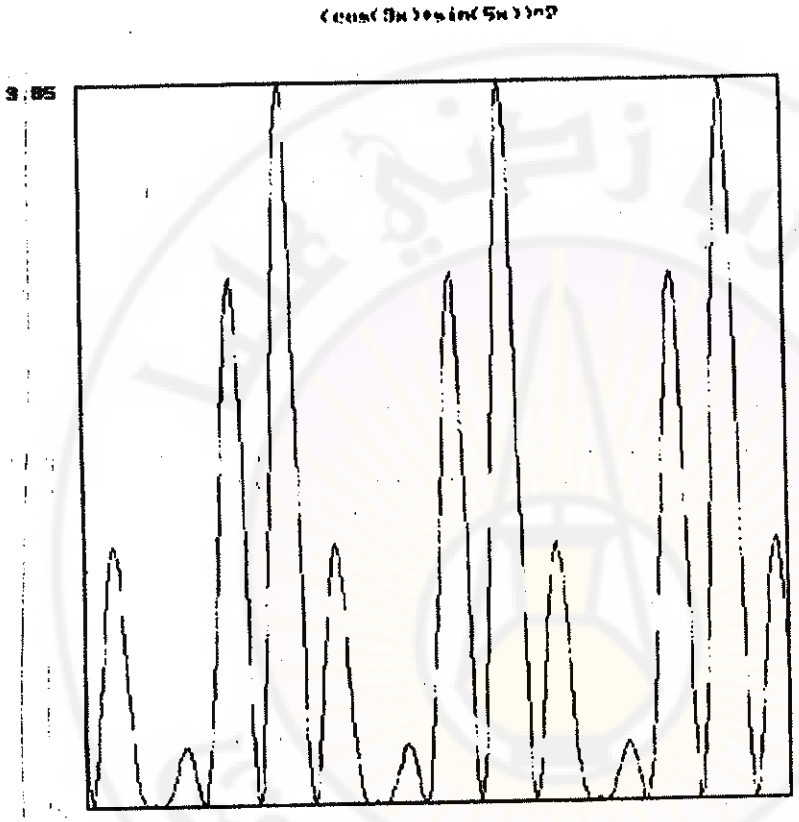

(**********************************)
(** Graphic Mode Initialisation **)
  GUInitGraph;
  GUWindowView( xmin, ymin, xmax, ymax , 100,50, 550. 450);
(**********************************)
(**       Drawing            **)
 For i:=0 to N-1 do
  begin
    GULine(xmin+i*dx,TY[i],xmin+(i+1)*dx,TY[i+1]);
  end;

 RX := xmax-xmin;    RY := ymax-ymin;
(* Draw the title string *)
 xst:=xmin + RX/3;   yst := ymax + RY/10;
 GUSetColor(5);
 GUText(xst,yst,title);

 (* draw X axis if the function has negative and postive values *)
 if ymin*ymax < 0 then
    GULine(xmin,0,xmax,0);
```

```
(* convert values of border (ymin,ymax,xmin,xmax)    *)
(*                    to strings to display  with GUText    *)
  str(ymin:8:2,st);
  xst := xmin-RX/6;  yst := ymin;
  GUtext(xst,yst,st);
  str(ymax:8:2,st);
  xst := xmin-RX/6;  yst := ymax;
  GUtext(xst,yst,st);
  str(xmin:5:1,st);
  xst := xmin;  yst := ymin-RY/50;
  GUtext(xst,yst,st);
  str(xmax:5:1,st);
  xst := xmax;  yst := ymin-RY/50;
  GUtext(xst,yst,st);
  readln;
(*********************************)
(**   End of Graphic Mode      **)
  GUCloseGraph;
End.
```


sin(x)/x
0.84
00.0


3.85

مسائل للبرمجة

مسائل في بنية المعطيات "مجموعة" Set

المسألة 1:

سوف نتعامل في هذه المسألة مع مجموعات قواسم الأعداد الصحيحة الموجبة، لتساعدنا على إيجاد العديد من العلاقات الكائنة بينها.

سنقصر الاهتمام في البداية على مجال الأعداد الموجبة من 1 إلى 255 . إذ يمكن ألا يتمكن الكثير من المترجمات التعامل مع مجموعات تتعدى هذا المجال، ولكن الحل سيبقى عاماً.

بدايةً سيقوم البرنامج بإيجاد مجموعة القواسم الخاصة بكل من أعداد المجال المدروس، وسنخزن لكل عدد مجموعة قواسمه ما عدا نفسه وفق البنية التالية:

```
Const MaxN=255;
Type range=1..maxN;
Type SetDivizer= set of range;
Type TabSetDiv = array[range] of SetDivizer;
```

نعرّف إذن نمط جدول مجموعات أعداد صحيحة TabSetDiv، ونخزن في متحول من هذا النمط مجموعات القواسم. فإذا كان لدينا المتحول VTabSet من النمط TabSetDiv، عندها تكون نتيجة الخزن لمجموعات القواسم فيه كما يلي:

```
VTabSet[1] =[1];
```

```
VTabSet[2] =[1];
VTabSet[3] =[1];
VTabSet[4] =[1,2];
.....
VTabSet[20] =[1,2,4,5,10];
.....
VTabSet[51] =[1,3,17];
.....
VTabSet[85] =[1,5,17];
.....
.....

VTabSet[255] =[1,3,5,15,17,51,85];
```

يُطلب من البرنامج حساب مجموعة القواسم الخاصة بكل من الأعداد في المجال 1 إلى 255 وخزنها في جدول وفق البنية المبينة آنفاً، و نقوم على التوازي بوضع مجموع قواسم هذه الأعداد في جدول. كما يُطلب كتابة برنامج جزئي procedure يسمح بإظهار محتوى مجموعة أعداد صحيحة.

والآن يُطلب من البرنامج، باستخدام المجموعات وجدول مجاميع القواسم، إيجاد مايلي.

أ- إظهار مجموعة قواسم أي عدد ضمن مجال المسألة

ب- إظهار الأعداد الكاملة: (العدد الكامل يساوي مجموع قواسمه)

ج-إظهار مجموعة القواسم المشتركة لعددين

د- إظهار مجموعة المضاعفات المشتركة لعددين والواقعة ضمن المجال.

ه- إظهار مجموعة الأعداد الأولية

مسائل في بنية المعطيات "تسجيلة" Record

المسألة 2:

المطلوب في هذه المسألة هو التعامل مع الأعداد العقدية. إن بنية المعطيات المستخدمة لتمثيل عدد عقدي هي تسجيلة، لها حقلان: حقل المُركِّبة الحقيقية وحقل المُركِّبة التخيلية للعدد العقدي.

البرامج الجزئية التي تحقق العمليات العقدية، والمطلوب برمجتها هي:

Cadd : جمع عددين عقديين.

Csub : طرح عددين عقديين.

Cmul : ضربُ عددين عقديين.

Cdiv : قسمة عددين عقديين.

CRead : قراءة عدد عقدي.

CWrite : كتابة عدد عقدي.

CSet : إسناد قيمة لعدد عقدي.

CAbs : تابع يحسب طويلة العدد العقدي.

يقوم البرنامج الأساسي باختبار البرامج الجزئية السابقة: يقوم بقراءة عددين عقديين ويجري العمليات الحسابية، ثم يظهر النتائج بالشكل:

```
( 25.500 + 50.250 i ) + ( 75.500 + 10.250 i ) = ( 101.000 + 60.500 i )
( 25.500 + 50.250 i ) - ( 75.500 + 10.250 i ) = ( ........ ........ i )
( 25.500 + 50.250 i ) X ( 75.500 + 10.250 i ) = ( ..... ......... i )
( 25.500 + 50.250 i ) / ( 75.500 + 10.250 i ) = ( ....... ........ i )
```

المسألة 3:

المطلوب في هذه المسألة التعامل مع التاريخ. بنية المعطيات المستخدمة لتمثيل التاريخ، هي تسجيلة بالشكل:

```
Type  date = record
   year, day, month : integer;
end;
```

المطلوب كتابة البرامج الجزئية التالية:

SetDate : يسند إلى متحول من نمط "تاريخ" ثلاث قيم صحيحة تمثل اليوم، والشهر والعام.

WriteDate :يُظهر التاريخ على الشاشة بالشكل المألوف، أي 31/10/1999.

GreaterDate : تابع منطقي يقارن بين تاريخين ويعطي القيمة "صح" إن كان الأول أكبر تماماً من الثاني.

BetweenDate : يعطي الفرق بين تاريخين.

يقوم البرنامج الأساسي باستدعاء البرامج الجزئية السابقة واختبارها.

المسألة 4:

المطلوب في هذه المسألة حساب علامات الطلاب وإصدار قوائمها.

يُعرَّف كل طالب برقم ذاتي (عدد صحيح موجب)، واسم (سلسلة حرفية لا تتجاوز 25 حرفاً)، إضافةً إلى علامات المواد: رياضيات، برمجة، لغة عربية، لغة أجنبية. تُحسب العلامة من 100 .

المطلوب قراءة المعلومات الخاصة بالطلاب وعلاماتهم، وحساب معدلاتهم وإصدار قوائم العلامات مرتبةً وفق المعدلات. كما يُطلب حساب وسطي العلامات والانحراف المعياري في كل مادة.

ملاحظة: عدد الطلاب لايتجاوز100 . وننصح، لسهولة اختبار البرنامج على عينات غير بسيطة، بكتابة المعطيات بطريقة كتابتها على الدخل النظامي، ولكن على ملف نصي باستخدام أي محرر نصوص، ثم بتنفيذ البرنامج على مستوى نظام التشغيل، مع تغيير إتجاه الدخل ليأخذ البرنامج معطياته من الملف المُنشأ بدلاً من لوحة المفاتيح. أي استدعاء البرنامج التنفيذي على النحو التالي:

‹اسم ملف الدخل› > ‹البرنامج التنفيذي› :C .

مسائل في الملفات النصية Text

المسألة 5:

المطلوب كتابة برنامج يستطيع قراءة أي ملف نصي، ويحسب المعلومات التالية الخاصة بهذا الملف:

- عدد الأسطر الكلي
- عدد الأحرف الكلي
- عدد الأحرف الأبجدية
- جدول تواتر الأحرف الأبجدية (الحرف الصغير والكبير متكافئان): أي عدد مرات ورود الحرف في نص الملف مقسوماً على عدد الأحرف الأبجدية فيه.

المسألة 6:

المطلوب كتابة إجرائية :

```
procedure GetLine( var f: text;  var  l: Line;  var n:integer);
```

حيث Line هو نمط جدول محرفي:

```
Type Line = array[1..80] of char;
```

تقوم الإجرائية بقراءة السطر الحالي من الملف النصي f للجدول المحرفي l وتضع عدد الأحرف الفعلية للسطر في المعامل n .

والآن نريد استخدام الإجرائية السابقة لقراءة ملف نصي سطراً تلو الآخر، وحساب عدد الكلمات في كل سطر. تُعرَّف الكلمة بأنها تتالي أحرف لايتضمن الفراغ ولا أدوات التنقيط

اكتب برنامجاً يقوم بحساب عدد الكلمات في كل من أسطر ملف نصي، واختبره على عدة ملفات نصية.

المسألة 7:

المطلوب قراءة ملف نصي وكتابته في ملف نصي آخر بعد تحويل كل حرف صغير فيه إلى حرف كبير.

مسائل في الملفات الثنائية File

المسألة 8:

اكتب برنامجاً يقوم بالعمليات التالية:

أ-الخزن في ملف ثنائي لمعطيات تعريف الكتب في مكتبة. يُوصف الكتاب برقم تسلسلي وعنوان واسم المؤلف وتاريخ الإصدار، إضافةً إلى حقل منطقي يدل على الإعارة. يقوم البرنامج بقراءة المعلومات من ملف معطيات نصي.

ب- تسجيل الإعارة أو الإعادة في الملف، يقوم المستخدم بإعطاء جدول بأرقام الكتب المعارة والمعادة، ليسجلها البرنامج على الملف.

ج- إظهار لائحة بالكتب المعارة.

المسألة 9:

لدينا ملفان ثنائيان من الأعداد الصحيحة مرتبان تصاعدياً، نريد دمجهما بحيث نحصل على ملف واحد مرتب يحوي عناصر الملفين. سنقوم بالدمج دون استخدام الجداول في البرنامج لأننا نفترض أن الملفان كبيرين جداً، وهذا مانسميه بالفرز الخارجي.

اكتب أولاً برنامجاً ينشيء الملفات الثنائية من ملفات نصية تحوي أعداداً صحيحة.

ثم اكتب برنامج الدمج الذي ينشيء الملف الثنائي الناتج، ثم يقوم بكتابته أيضاً في ملف نصي.

مسائل في المؤشرات

المسألة 10:

يُطلب إعادة كتابة البرنامج للمثال المعطى بخصوص التعامل مع سلسلة مرتبة.

يقوم البرنامج الجديد باستخدام بنية تحوي مؤشرين بدلاً من مؤشر واحد: مؤشر إلى العنصر السابق ومؤشر إلى العنصر اللاحق. وللإشارة إلى السلسلة (مزدوجة الترابط) نستخدم مؤشرين: الأول يشير إلى الرأس والثاني إلى الذيل.

كما يُطلب تعديل إجرائية إظهار السلسلة بحيث يُعطى خيار الإظهار على نحو تصاعدي أو تنازلي.

المسألة 11:

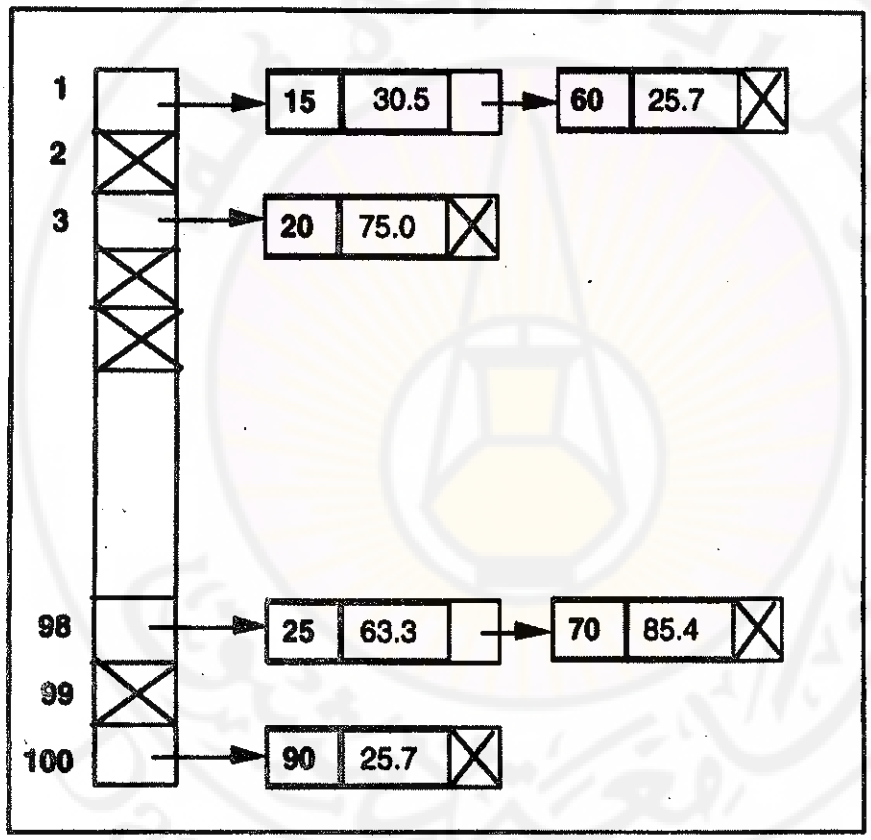
تتعامل مسألتنا مع ما يُسمَّى «المصفوفات المثقوبة»، وهي مصفوفات تغلب فيها القيمة صفر، فمحدودة هي القيم المختلفة عن الصفر.

وبهدف الاختصار في الذاكرة اللازمة لمصفوفات كبيرة من هذا النوع، سنستخدم المؤشرات المرسومة في الشكل المرفق، والمعرَّفة كما يلي:

```
Const MaxN= 100;
Type PEMAT =  ^EMAT;
Type EMAT= record
 begin
   col : integer;
   val : real;
   pnext : PEMAT;
end;

Type HoleMat = array[1..MaxN] of PEMAT;
```

وبهذا، تكون المصفوفة جدولاً من المؤشرات، يؤشّر كل منها إلى سلسلة العناصر غير الصفرية في سطر المصفوفة. ويكون عنصر المصفوفة ممثلاً بتسجيلة، عناصرها: رقم العمود و قيمة العنصر ومؤشر إلى عنصر آخر في السطر نفسه.



تمثيل مصفوفة مثلثية باستخدام المؤشرات

والآن، وقد عرفنا طريقة تعريف نمط المصفوفة، يطلب كتابة البرامج الجزئية التالية:

ReadHMat : يقرأ عناصر المصفوفة من الدخل ويخزنها في متحول من نمط HoleMat

WriteHMat : يكتب مصفوفة من النمط HoleMat (العناصر المختلفة عن الصفر فقط).

AddHMat : يجمع مصفوفتين من النمط HoleMat ، و يضع الناتج في مصفوفة من النمط ذاته.

SumHMat : يجمع جميع عناصر مصفوفة من النمط HoleMat .

MaxMat : يعطي العنصر الأكبر بين عناصر مصفوفة من النمط HoleMat .

اكتب الآن البرنامج الأساسي الذي يستدعي البرامج الجزئية السابقة ويختبرجيداً صحة عملها.

مسائل في إنشاء الوحدات البرمجية Unit

المسألة 12:

يُطلب تحقيق مسألة التعامل مع الأعداد العقدية باستخدام وحدة برمجية، أي نقل تعريف النمط العقدي ومجموعة البرامج الجزئية، التي عرفناها لتتعامل معه، إلى وحدة برمجية، ولنُسمِّها UComplex . وبذلك، يمكن أن تُستخدم UComplex من قبل البرامج التي تحتاج إلى العمليات العقدية. استخدم UComplex في البرنامج الذي سوف تكتبُه، للتوثُّق من صحة البرامج الجزئية.

اكتب برنامجاً آخر يستخدم أيضاً الوحدة البرمجية نفسها، ويحسب التعبير العقدي :
(C1 + C2) * (C1-C2)
ويتحقق أن التعبير السابق يساوي التعبير: C1*C1 - C2*C2

والآن، خذ الوحدة UComplex وغيرها إلى وحدة جديدة UAComplex ، على النحو التالي:

نُغيِّر تعريف النمط Complex ليستخدم **array** بدلاً من **record**:

```
Type   Complex= array(1..2) of real;
```

نُغيِّر تحقيق الوحدة وفقاً لذلك، محتفظين بترويسات البرامج الجزئية على حالها .

أعِد ترجمة Compilation وتنفيذ البرامج السابقة باستخدام الوحدة الجديدة UAComplex .

لاحظ أهمية استخدام الوحدات البرمجية في كتابة أنماط مجردة. إن تعريفنا لنمط وقَصْر التعامل مع هذا النمط على مجموعة من البرامج الجزئية المجتمعة في وحدة، يتيح لنا الحفاظ على برامجنا التي تستخدم هذه الوحدة دون تغيير، حتى حين تتغير ماهية تحقيق النمط الذي يستخدمه البرنامج.

المسألة 13:

يمكن تمثيل كثير الحدود بجدول array نخزن فيه أمثال كثير الحدود وبعدد صحيح يمثل درجته .ولنضع حداً أعظم لدرجة كثير الحدود القيمة 30 .

يمكن أن نعرّف نمط كثير الحدود Polynom كما يلي:

```
Type Polynom= record
begin
   cof : array[0..30] of real;
   degree : integer;
end;
```

وهكذا، لتمثيل كثير الحدود: $1 + 3X^{10}$ في متحول P، (var P: Polynom:) نقوم بعمليات الإسناد:

```
P.degree := 10;
for i:=0 to 10 do
   P.cof[i] := 0;
P.cof[0] := 1;
P.cof[10]:=3;
```

والآن، وقد استوعبنا تعريف نمط كثير الحدود، المطلوب هو كتابة وحدة برمجية unit تضمن تعريف النمط ومجموعة البرامج الجزئية التالية التي تتعامل معه:

ReadPol : يقرأ كثير الحدود من الدخل، يُعطى الدخل بإعطاء الدرجة والمثل لكل حد مختلف عن الصفر.

WritePol : يكتب كثير الحدود، ويظهر الحد بالشكل (xxxxx.xx)X^n للحدود غير الصفرية فقط . خذ بالحسبان كتابة كثير الحدود الطويل على أكثر من سطر.

ZeroPol : يعطي لكثير الحدود، القيمة صفر.

AddPol : يجمع كثيري الحدود، و يضع الناتج في كثير حدود آخر.

SubPol: يطرح كثيري الحدود، و يضع الناتج في كثير حدود آخر.

ProdPol: يضرب كثيري الحدود، و يضع الناتج في كثير حدود آخر..

اكتب برنامجاً يستخدم الوحدة البرمجية السابقة ويختبر جميع عملياتها.

مسائل في الرسم البياني

في المسائل التالية، يعتمد الإظهار البياني حصراً على وحدة الرسم البياني Unit GU المعطاة في الفصل الثامن، والمعرَّفة بالواجهة التالية:

```
Unit GU;

Interface
 Procedure GUInitgraph;
 Procedure GUCloseGraph;
 Procedure GUWindowView(wxb,wyb, wxt, wyt : real;
                        vxt,vyt,  vxb, vyb : integer);
Procedure GULine( x1, y1, x2, y2 : real );
Procedure GUText( x,y:real; S: String);
Procedure GUSetColor( c : integer );
```

المسألة 14:

المطلوب كتابة برنامج لرسم منحني التابع الوسيطي:

$$X(u) = u * \cos(u)$$
$$Y(u) = u * \sin(u)$$

غيّر خطوة تَحوّل u ولاحظ تَغيُّر شكل المنحني.

المسألة 15:

نريد تمثيل مجموعة من المعطيات بشكل بياني دَرَجيّ Histogram.

يأخذ ملف المعطيات الصيغة التالية:

title
name value
....

حيث title هو سلسلة حرفية تكوّن عنوان الشكل البياني، ويلي العنوان جدول قيم القياسات: name هو اسم تعريف القياس، value هي قيمة القياس.

مثال:

Rain in Country
HOMS 450
HAMAH 350
DAMAS 240
ALEP 160
TARTOUS 240

القيود على المعطيات هي التالية:

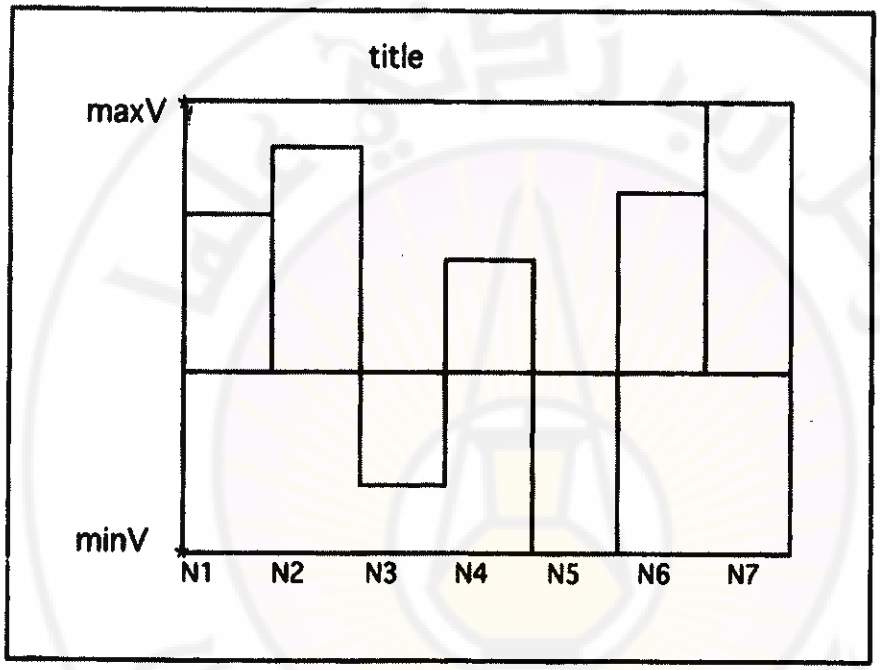
لايحتوي الملف على أكثر من ثماني قيم.

title العنوان لا يزيد عن 40 حرفاً.

name اسم تعريف القياس لايزيد عن 8 أحرف.

value القيمة، عدد كسري محصور في المجال [1000+,1000-] ونهتم في القسم الكسري بخانتين فقط.

يأخذ الإظهار البيّاني لملف المعطيات الشكل التالي:



حيث Ni هو اسم القياس، maxV أعلى قيمة، minV أدنى قيمة، والمسافات خارج المستطيل الداخلي ثابتة دوماً.

نريد كتابة برنامج يقبل كدخل له، أيَّ ملف نصي للمعطيات يحترم الصيغة المعرّفة آنفاً، يعطي مستخدم البرنامج اسم الملف، فيُظهِر البرنامج المعطيات بشكل بياني كما هو معرَّف بالشكل السابق.

المطلوب:

1- تحديد المتحولات الأساسية للمسألة وتعيين إحداثيات مكوّنات الشكل وفق هذه المتحولات.

2- كتابة خوارزمية العمل وصولاً إلى الشكل البياني (اعتماداً على مجموعة الأساسيات البيانية المعرّفة في وحدة الرسم البياني GU).

3- تحديد البرامج الجزئية اللازمة لتنفيذ الخوارزمية.

4- نقل الخوارزمية إلى برنامج بلغة Pascal يقوم بأداء المطلوب.

المسألة 16:

نريد كتابة برنامج عمومي يسمح برسم منحني التوابع القطبية، بقطع النظر عن تعريف التابع.

سيكون لدينا في البرنامج تعريف التابع القطبي المطلوب رسمه:

```
function R( teta:real):real;
```

والبرنامج مكتوب بحيث يقوم برسم منحني هذا التابع مهما يكن تعريفه، إذ يمكن إعادة استخدام نفس البرنامج من أجل تابع جديد: نغيّر نص التابع فقط، ويبقى الباقي على حاله دون أي تغيير، نترجم البرنامج ونجري التنفيذ فيرسم التابع الجديد.

المعطياتُ الأخرى اللازمة لإنجاز المطلوب يدخلها المستخدم وهي:

t1,t2 : بداية ونهاية المجال الزاوي الذي سيرسم التابع فيه.

dt : خطوة الرسم.

S: عبارة نصية تكون عنواناً يظهر على الرسم.

القيد الذي نضعه هو أن عدد النقاط، التي ستُحسب للتابع، يجب ألا يتجاوز حداً ثابتاً وليكن 500. هذا القيد يأتي من فكرة محتملة لاستخدام جدول (array) لخزن قيم التابع قبل رسمه.

أكتبْ خوارزمية الحساب اللازمة لتحديد التحويل (الإسقاط) المناسب. حدد النافذة window وبوابة الرؤية viewport، التي ستستخدمها للرسم، اكتب خوارزمية الرسم. (حاول الاعتماد على فكرة التبسيط المتتالي والبرامج الجزئية إن أمكن).

أنقل الخوارزمية إلى برنامج بلغة Pascal بالاعتماد على الأساسيات البيانية المتاحة في الوحدة البرمجية GU المعروفة لديك. واختبره على توابع قطبية مختلفة: ابدأ بتابع المثال الذي عرفناه، $R = (1 + \cos(1 * u))$

مسائل في البرمجة غرضية التوجه

المسألة 17:

في المثال 2 الصفحة 172 نريد أن نضيف غرضاً ابناً للغرض Employee، اسمه الغرض FullTime. بحيث يتميز عنصر من هذا النمط بحقلين إضافيين هما الراتبsalary، وعدد سنوات الخدمةyears.نفترض أن الوحدة uemploy متوفرة لدينا، ونريد تزويد الغرض بالطرائق التالية:

Init : تعطي قيمة لاسم الموظف وعنوانه وراتبه.

UpdateYears: تضيف عاماً إلى عدد سنوات الخدمة

CalcLeave: تحسب الإجازات بحيث يستحق الموظف الذي أمضى أقل من عشر سنوات في الخدمة 15 يوماً مضافاً إليها عدد سنوات الخدمة، بينما يستحق 25 يوماً إذا كان عدد سنوات خدمته عشر سنوات فما فوق.

Report: تُظهر على الشاشة معلومات الموظف(اسمه، عنوانه، إجازاته المستحقة وذلك باستدعائها لـ CalcLeave).

والمطلوب بناء الغرض FullTime ووضعه ضمن وحدة Unit اسمها uFullTime و كتابة برنامج يختبر هذه العمليات المعرفة ضمن الوحدة.

المسألة 18:

نريد بناء غرض يسمى "موظف قديم" LongServing ابن الغرض FullTime ، له نفس حقوله ولكنه يختلف عنه في أنه قد خدم في المؤسسة 15 عاماً على الأقل. يتميز الموظف القديم بأنه يحق له يوم إجازة إضافي كل عام (أسوة ببقية الموظفين)، إضافة إلى يوم آخر عن كل سنتي خدمة بعد تجاوز 15 سنة خدمة. والمطلوب تعريف الغرض LongServing وإعادة كتابة CalcLeave و Init (تجري التهيئة بوضع 15 كقيمة ابتدائية في عدد سنوات الخدمة).

والآن يطلب كتابة برنامج يتضمن نموذجاً من الغرض LongServing وتهيئته واستدعاء Report. برر ماذا يحصل !

أعد النظر في هرمية الوراثة وعرّف الطريقة CalcLeave على أنها افتراضية virtual في

الأماكن المناسبة وأعد استدعاء Report على موظف من نمط LongServing . علل النتيجة.

مسائل في استخدام أغراض مكتبة Turbo Vision

المسألة 19:

أضف خياراً فرعياً إلى قائمة الخيارات المنسدلة من الخيار File في التطبيق الذي بنيناه في الفصل العاشر بحيث يسمح للمستثمر باختيار ملف نصي من دليل ما، ثم جهّز البرنامج بحيث يفتح الملف المختار ويشحن محتواه ضمن النافذة.

المسألة 20:

أضف مستطيل حوار إلى التطبيق الذي بنيناه في الفصل العاشر بالعودة إلى الغرض TDialog الموجود في Turbo Vision واشتقاق غرض منه. ثم ضع في محتوى مستطيل الحوار سؤالاً :أيريد المستثمر إغلاق التطبيق أم لا؟

ملحق (1)

المفردات المُعرَّفة في Pascal-Wirth

سنورد هنا كل ماهو سابق التعريف أو معرَّف ضمنياً في لغة Pascal كما عرَّفها مصممها Niklaus Wirth . وسنسرد الكلمات الإضافية المحجوزة والخاصة بمترجمات Turbo Pascal .

سنصنف الأغراض السابقة التعريف حسب نوعها. هذه الجداول ضرورية لأي مبرمج بلغة Pascal كي يميّز الإضافات الموجودة في مترجم اللغة بين يديه وماهو معرَّف نظامياً وعلى نحوٍ قياسي ويمكن أن يجده في أي مترجم.

أسماء التعريف النظامية standard identifiers :

الثوابت Constant:

false, true, maxint

الأنماط Types:

integer, real, boolean, char, text

معاملات البرنامج Program parameters:

input, output

التوابع Functions:

abs, arctan, chr, cos, eof, eoln, exp, ln, odd, ord, pred
round, sin, sqr, sqrt, succ, trunc

الإجرائيات Procedures:

get,dispose, new, pack, page, put, read, readln, reset, rewrite, unpack,
write, writeln

الكلمات المحفوظة للّغة Reserved words:

and	**end**	**nil**	**set**
array	**file**	**not**	**then**
begin	**for**	**of**	**to**
case	**function**	**or**	**type**
const	**goto**	**packed**	**until**
div	**if**	**procedure**	**var**
do	**in**	**program**	**while**
downto	**label**	**record**	**with**
else	**mod**	**repeat**	

جرى التعامل مع جميع مفردات باسكال القياسية، باستثناء تلك التي لم تعد تُستخدم إلا فيما ندر جداً:

page للقفز صفحة في ملف نصي

packed , pack, unpack للتعامل مع جداول الحروف، وصار يُستغنى عنها باستخدام النمط String.

الكلمات الإضافية المحفوظة في Turbo-Pascal:

للتعامل مع الوحدات البرمجية:

unit, uses, interface, implementation

للتعامل مع الأغراض:

object, virtual, constructor, destructor

Damascus University

نصائح في البرمجة

Programming Proverbs

نذكر فيما يلي بعض القواعد العامة التي ينبغي أن تبقى حاضرة في أذهاننا أثناء عملنا البرمجي. سأسرد هنا بعض هذه القواعد مأخوذةً من مرجع: Programming Proverbs للمؤلف H.F. Ledgard المعروف في المجال التربوي فيما يخص البرمجة.

1 - عرّف المسألة المطروحة تعريفاً كاملاً.

2- فكّر أولاً، ثم برمج فيما بعد.

3- استخدم الطريقة التنازلية في التحليل (Top Down).

4- جزّىء البرنامج إلى وحدات منطقية (Logical Units).

5-استخدم البرامج الجزئية (Subprograms).

6- تجنب تعليمات القفز(Goto).

7 - انتبه للمتحولات المشتركة (Common and Global Variabls).

8- اعتن بقواعد اللغة (Language Syntax) منذ البداية.

9- اختر بعناية أسماء التعريف (Identifiers): أسماء معبّرة عن المضمون.

10- تجنب تغيير متحولات الحلقة التكرارية (For).

11 - حاول استخدام اللغة القياسية (Standards) وخفف استخدام خصوصيات المترجم Compiler .

12- لاتفترض أبداً أن الحاسوب يفترض شيئاً (مثلاً: القيمة البدائية لمتحول عددي لن تكون من تلقاء نفسها صفراً).

13- استخدم التعليقات ضمن برنامجك.

14 - اعتن بتنسيق البرنامج (جماله النصي).

15 - اكتب دوماً توثيقاً (Documentation) جيداً لبرنامجك ، سيفيدك ذلك مستقبلاً.

16 - حاول اختبار البرنامج يدوياً قبل اختباره حاسوبياً.

17- لاتهتم بجمال إظهار النتائج قبل تحقق صحتها.

18 - لاتهمل جمالية إظهار النتائج وسهولة استخدام البرنامج بعد اختباره.

19 - لاتتهيب استبعاد البرنامج برمته والبدء من جديد عندما تسوء الأمور.

20- أعد قراءة دليل اللغة (Language Guide) حيناً بعد حين.

المراجع

1- K. Jensen & N. Wirth
Pascal, User Manual and Report, second edition
1978, Springer-Verlag

2- W. Findlay & D. Watt
PASCAL, An Introduction to Methodical Programming, second edition
1981, computer science press

3- Patrick Cousot
Introduction a l'algorithmique et a la programmation en Pascal
1988, Mc Graw Hill.

4- Henry F. Ledgard
Programming Proverbs
1975, HAYDEN Book Company

5- Schneider et Bruel
Advanced Programming and problem Solving with Pascal
1981,John Wiley and Sons Inc.

6- Douglas W. Nance
Pascal, Understanding Programming and Problem Solving
1989, 2nd edition,West Publishing Company

7- Philippe Schweizer
Infographie II
1987, Presses Polytechniques Rommandes

8- Borland Pascal 1996

9- Scott D.Palmer
Mastering Turbo Pascal 6
1991,Tech Publications

10- Turbo Pascal 7 , 1997
الدليل الكامل
ياسر نهاد العقاد
دار شعاع

11- Stephen O'BRIEN
Turbo Pascal 6 .The Complete Reference
1991,Osborne Mc Graw-Hill

12-David Sargent
Coding in Turbo Pascal
1994, The McGraw-Hill Book Company

Damascus University