NGUYỄN TÔ THÀNH

LẬP TRÌNH NÂNG CAO

trên ngôn ngữ Pascal

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN TÔ THÀNH

LẬP TRÌNH NÂNG CAO TRÊN NGÔN NGỮ PASCAL

(In lần thứ 4)

Sách dùng cho:

- Sinh viên Đại học
- Học sinh khá, giỏi
- Giáo viên dạy tin học

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

LỜI GIỚI THIỆU

Pascal là ngôn ngữ lập trình được đưa vào giảng dạy trong hầu hết các trường đại học ở nước ta với mục tiêu cung cấp những kiến thức và kỹ năng cơ sở về lập trình cho sinh viên, từ đó cho phép sinh viên tìm hiểu và làm chủ một cách dễ dàng các ngôn ngữ lập trình bậc cao khác như C, C++, ... Cuốn **LẬP TRÌNH NÂNG CAO TRÊN NGÔN NGỮ PASCAL** của tác giả **Nguyễn Tô Thành** không chỉ bổ sung những kiến thức nâng cao về lập trình Pascal chưa được đề cập đến trong các giáo trình *Tin học đại cương*, mà còn cung cấp cho người đọc những hiểu biết rất bổ ích về kỹ thuật *lập trình có cấu trúc* (*structured programming*). Tác giả **Nguyễn Tô Thành** đã giảng dạy nhiều năm về *cơ sở Toán học cho Tin học*, *cấu trúc dữ liệu và giải thuật* và đặc biệt về *kỹ thuật lập trình* cho sinh viên Khoa Công nghệ Thông tin, trường Đại học Bách khoa Hà nội, đồng thời liên tục tham gia bồi dưỡng kiến thức cho các đội tuyển Olympic Tin học của Việt nam. Cuốn sách có thể coi là một công trình khoa học nghiêm túc và cập nhật, đúc kết tâm huyết, trình độ và kinh nghiệm quý báu của tác giả về kỹ thuật lập trình. Cuốn sách không chỉ phục vụ trực tiếp cho sinh viên chuyên ngành *Công nghệ Thông tin*, *Toán-Tin* của các trường đại học mà còn rất bổ ích đối với bất cứ ai *cần* và *yêu thích* lập trình cho máy tính. Xin trân trọng giới thiệu cuốn sách với bạn đọc gần xa.

PGS. TS. Nguyễn Thúc Hải

Trưởng khoa Công nghệ Thông tin
Trường Đại học bách khoa Hà nội

LỜI NÓI ĐẦU

Cuốn **LẬP TRÌNH NÂNG CAO TRÊN NGÔN NGỮ PASCAL** đề cập một số vấn đề thuộc về kỹ thuật lập trình trên nền ngôn ngữ Pascal, vì thế bên cạnh những bổ sung kiến thức về ngôn ngữ mà những người mới chỉ qua những giáo trình "Tin học đại cương" còn thiếu, cuốn sách còn đề cập đến những vấn đề cơ bản của kỹ thuật lập trình trên một ngôn ngữ lập trình có cấu trúc nói chung.

Để thuận tiện cho việc thử chương trình, chúng tôi dùng cài đặt của Turbo Pascal phiên bản 7, là phiên bản thông dụng hiện nay. Vì thế bạn đọc cần nắm vững tính lôgic của vấn đề, không nên phụ thuộc vào những hạn chế của cài đặt, để có thể dễ dàng chuyển sang những cài đặt khác.

Đối tượng của cuốn sách là những người đã qua giáo trình "Tin học đại cương", trong đó đã làm quen với lập trình Pascal cơ sở. Những cấu trúc câu lệnh cũng như các kiểu dữ liệu cơ bản nhất của Pascal như số nguyên, số thực, ký tự, xâu ký tự, lôgic, bản ghi, mảng, tập hợp, ... xem như bạn đọc đã quen thuộc. Một số thuật toán cơ bản của tin học như tích lũy, tìm kiếm, sắp xếp, ... chúng tôi cũng xem như bạn đọc đã biết.

Một số vấn đề trong cuốn sách có liên quan đến những kiến thức trong những giáo trình cơ bản khác của Tin học như "Kiến trúc máy tính", "Hệ điều hành", "Cấu trúc dữ liệu và giải thuật", ... Vì vậy, cuốn sách này có thể hỗ trợ thêm cho bạn đọc trong việc tìm hiểu các giáo trình trên và ngược lại, việc liên hệ tốt các kiến thức liên quan sẽ giúp bạn khai thác cuốn sách này một cách hiệu quả.

Một số khái niệm như chương trình con, file - do vai trò đặc biệt quan trọng của chúng trong kỹ thuật lập trình nên mặc dù trong các giáo trình lập trình cơ sở bạn đọc đã được làm quen (nhưng còn sơ sài và chưa thấu đáo) - cũng sẽ được giới thiệu một cách đầy đủ và hệ thống trong những chương đầu của cuốn sách.

Chúng tôi coi trọng tính lôgic của việc lập trình, vì thế trong các thí dụ minh họa, bên cạnh văn bản chương trình nguồn để các bạn có thể thử được ngay trên máy, chúng tôi cố gắng đưa vào việc phân tích và thiết kế chương trình để các bạn có thể dễ dàng thay đổi và phát triển. Một số thí dụ mang tính chất mô hình để có thể áp dụng cho một lớp các bài toán, một số khác mang tính gợi ý để các bạn luyện tập.

Cuốn sách bao gồm những chương sau đây:

CHƯƠNG TRÌNH CON: Chương này đề cập đến những khía cạnh khác nhau của kỹ thuật viết chương trình con, đặc biệt trong việc tổ chức các biến (chung, riêng), các tham số (truyền theo trị, biến), các khả năng đệ quy. Ngoài ra trong chương còn nhấn mạnh vai trò của chương trình con trong kỹ thuật lập trình Top-Down, một vấn đề mà những người mới chỉ qua các giáo trình cơ sở về lập trình chưa có dịp luyện tập.

GIAO TIẾP FILE: Bên cạnh việc khái quát các khái niệm và vai trò của file, chương này còn trình bày khá chi tiết và hệ thống các thao tác trên các loại file của Turbo Pascal, đặc biệt trên file văn bản là nơi thường xuyên giao tiếp với chương trình. Một số thuật toán trên file còn xa lạ với nhiều người mới lập trình như tìm kiếm nhanh trên file định kiểu theo chỉ số, sắp xếp file tuần tự nhờ kỹ thuật trộn file, cũng được giới thiệu cùng với những chương trình minh họa. Những bạn đọc nào còn lúng túng trong việc dùng file, cần xem kỹ chương này.

QUẢN LÝ BỘ NHỚ: Nội dung chương này hầu như là mới đối với những người mới lập trình. Người đọc sẽ làm quen với khái niệm địa chỉ và biết phân biệt nó với dữ liệu, do đó sẽ hiểu rõ hơn bản chất việc tổ chức và lưu trữ các dữ liệu bên trong bộ nhớ của máy. Những kiến thức trong chương này có liên quan nhiều đến các giáo trình "Kiến trúc máy tính" và "Hệ điều hành", vì thế chúng tôi chỉ nhấn mạnh nguyên tắc và cách thức truy cập bộ nhớ nhờ các công cụ của Turbo Pascal. Ngoài ra, người đọc cũng được làm quen với những thao tác trên dữ liệu mà không quan tâm đến kiểu dữ liệu, thông qua việc dùng các tham biến không định kiểu. Một số địa chỉ có ích như vùng Video Ram (quản lý màn hình văn bản) được giới thiệu để bạn đọc thử nghiệm và triển khai các ứng dụng.

KIỂU CON TRỎ: Qua chương này bạn đọc sẽ thấy rõ vai trò đặc biệt của các con trỏ trong kỹ thuật lập trình trong việc khai thác bộ nhớ. Danh sách liên kết (cùng với các biến thể của nó) được giới thiệu khá kỹ qua những thao tác cơ bản nhất. Đây là một chương khó đối với những người mới lập trình, vì thế chúng tôi cố gắng đưa vào những mô hình có tính khái quát để bạn đọc dễ áp dụng trong từng trường hợp cụ thể. Những kiến thức trong chương này có liên quan nhiều đến giáo trình "Cấu trúc dữ liệu và giải thuật".

THƯ VIỆN LẬP TRÌNH: Chương này giới thiệu vai trò của thư viện nói chung trong kỹ thuật lập trình, đặc biệt trong các phát triển ứng dụng, sau đó giới thiệu Unit, một công cụ của Turbo Pascal nhằm giúp người dùng xây dựng và phát triển thư viện. Bên cạnh việc trình bày cú pháp viết Unit, chúng tôi còn nhấn mạnh cách khai thác Unit cũng

như những phương pháp xây dựng và phát triển chúng. Cuối chương có điểm qua một cách khái quát các thư viện chuẩn của Turbo Pascal cung cấp cho người dùng.

MỘT SỐ THƯ VIỆN CHUẨN: Chương này trình bày một cách hệ thống và chi tiết 3 unit tiêu biểu nhất của Turbo Pascal là CRT, DOS và GRAPH (unit SYSTEM xem như đã quen trong các giáo trình Pascal cơ sở) với những mục đích:

- Cung cấp kho tài nguyên dưới dạng các hàm, thủ tục, kiểu dữ liệu đã được Turbo Pascal xây dựng nhằm giúp người dùng dễ dàng triển khai các ứng dụng của mình. Bạn đọc sẽ thấy một cách hệ thống hơn các mối liên quan của các hàm, thủ tục, kiểu dữ liệu trong những lĩnh vực ứng dụng khác nhau mà không còn đơn thuần là những cú pháp gọi chúng.
- Giúp người đọc biết cách khai thác một thư viện sao cho có hiệu quả. Đây là một vấn đề quan trọng trong việc tự học của của các bạn.
- Giúp người đọc học tập được cách xây dựng một thư viện, một trong những dạng sản phẩm phần mềm đang được quan tâm và phát triển hiện nay.

Các thí dụ trong chương này khá phong phú, đặc biệt trong phần trình bày các unit DOS và GRAPH. Các bạn có thể tìm thấy trong các thí dụ này những gợi ý cho những ứng dụng trong các lĩnh vực thiết kế trò chơi, xây dựng phần mềm dạy học, ...

Chương cuối cùng là **MỘT SỐ THÍ DỤ**. Đây là chương nhằm cụ thể hóa những vấn đề mà cuốn sách đã đề cập về kỹ thuật lập trình qua một số bài toán. Chúng tôi cố gắng chọn những bài mà có thể khái quát hóa việc thiết kế chúng. Trong số những thí dụ, chúng tôi có chọn một số đề thi Olympic quốc gia, quốc tế để các bạn giáo viên cũng như các học sinh các lớp chuyên Tin tham khảo. Trong lời giải, chúng tôi cũng cố gắng thể hiện tinh thần ***Cấu trúc dữ liệu + Giải thuật = Chương trình*** mà N.Wirth đã đưa ra. Các bài toán đều được trình bày lời giải lôgic trước, sau đó mới đến bước xây dựng chương trình, bao gồm các khâu tổ chức dữ liệu và thiết kế các chương trình con. Các chương trình đều được xây dựng theo nguyên tắc Top-Down: các khối lớn được xây dựng trước, sau đó mới đến các khối con của chúng. Các chương trình đều đã được chạy thử và chú thích đầy đủ để các bạn dễ dàng tìm hiểu. Cuối mỗi thí dụ, chúng tôi có dành một số vấn đề cho bạn đọc suy nghĩ và phát triển.

Một phụ lục cuối sách giới thiệu tóm tắt môi trường làm việc của Turbo Pascal phiên bản 7 để giúp bạn đọc thuận tiện khi biên soạn và chạy chương trình.

Do điều kiện thời gian, một số vấn đề thuộc về lập trình hướng đối tượng cũng như các thư viện tương ứng của Turbo Pascal (Turbo Vision) chưa có dịp bàn đến trong cuốn sách này. Các thí dụ minh họa và ứng dụng còn có thể bổ sung phong phú hơn nữa, ... Hy vọng rằng, theo thời gian, cuốn sách sẽ được hoàn thiện dần.

Cuốn sách này dành cho các sinh viên ngành Tin học hoặc những ngành khác muốn đi sâu hơn về kỹ thuật lập trình. Nó cũng là tài liệu tham khảo cho các lớp kỹ sư bằng hai về

Tin học khi học những giáo trình có liên quan đến kỹ thuật lập trình. Ngoài ra nó cũng là tài liệu giúp thêm các giáo viên dạy Tin học trong các trường phổ thông cũng như học sinh trong các lớp chuyên Tin và những học sinh phổ thông khá, giỏi khác quan tâm đến lĩnh vực này.

Chắc chắn cuốn sách còn nhiều thiếu sót, mong các bạn góp ý trực tiếp cho tác giả để có thể hoàn chỉnh nội dung cho các lần xuất bản sau. Tác giả xin chân thành cảm ơn.

Hà nội tháng 9 năm 2001

Nguyễn Tô Thành

Bộ môn Khoa học Máy tính
Khoa Công nghệ Thông tin
Đại học Bách khoa Hà nội
Tel 8692463 - 8696121

MỤC LỤC

1

CHƯƠNG TRÌNH CON

1. Chương trình con và vai trò của chương trình con

Chương trình là một dãy lệnh được xây dựng cho máy tính nhằm hoàn thành một công việc nào đấy. Công việc này có thể bao gồm nhiều công việc nhỏ và đến lượt chúng lại được tổ chức như những chương trình. Như vậy, lời gọi các chương trình này sẽ được xuất hiện trong các lệnh của chương trình đang xét. Chúng được gọi là những *chương trình con*. Khái niệm chương trình con mang tính tương đối: một chương trình có thể là chương trình con của một chương trình khác. Người ta thường dùng thuật ngữ *chương trình chính* để chỉ chương trình ở mức ngoài cùng (được gọi bởi người sử dụng).

Chương trình con là một công cụ không thể thiếu của các ngôn ngữ lập trình bậc cao. Có thể nói, việc lập trình có cấu trúc hiện nay chủ yếu dựa vào kỹ thuật thiết kế các chương trình con. Vai trò của chương trình con trong kỹ thuật lập trình có thể tóm tắt trong mấy điểm chính sau:

- Cho phép phân rã những bài toán phức tạp thành nhiều bài toán đơn giản hơn, từ đó cho phép kiểm soát toàn bộ chương trình, dễ dàng sửa chữa và phát triển.
- Nâng cao tính độc lập trong việc thiết kế, che giấu nội tại. Cho phép xây dựng những chương trình lớn, có nhiều người tham gia.
- Là phương thức thực hiện lập trình Top-Down (từ tổng thể đến chi tiết), thích hợp với hầu hết các thiết kế trong lập trình cấu trúc.
- Cho phép xây dựng thư viện lập trình, kế thừa các kết quả trước đấy, giảm chi phí và công sức trong việc viết chương trình.

Mỗi một ngôn ngữ lập trình có cách xây dựng riêng các chương trình con, tuy nhiên về mặt ý nghĩa cũng như cách dùng đại thể là giống nhau. Chương này giới thiệu cách dùng các chương trình con trong Turbo Pascal, nhưng nhiều điều có thể áp dụng cho những ngôn ngữ khác.

2. Khai báo chương trình con. Hàm và thủ tục

Giống như mọi đối tượng tham gia chương trình (hằng, biến, kiểu), mỗi chương trình con đều phải đặt tên theo nguyên tắc chung của hệ thống. Mỗi khi lời gọi với tên của chương trình con nào xuất hiện trong phần câu lệnh, thì chương trình con ấy được thực hiện.

Các chương trình con được gọi trong một chương trình cần được hệ thống nhận biết. Bất cứ một tên chương trình con nào xuất hiện trong lời gọi mà hệ thống không hiểu, đều nhận được lỗi *unknown identifier* khi biên dịch. Những chương trình con được hệ thống nhận biết sẵn là những chương trình con của hệ thống và những chương trình con đã được biên dịch trong các thư viện đã liên kết với chương trình. Tất cả các chương trình con còn lại trong chương trình đều phải được khai báo trước khi dùng.

Turbo Pascal có hai cách tổ chức chương trình con là thủ tục và hàm.

2.1. Thủ tục

Thủ tục là một chương trình con nhằm thực hiện một công việc nào đấy. Cũng như một chương trình độc lập, thủ tục gồm 3 phần: phần đầu (tiêu đề), phần khai báo và phần câu lệnh.

Phần đầu thủ tục được khai báo với từ khóa PROCEDURE, tiếp theo là tên thủ tục. Nếu thủ tục có chứa các tham số (parameters) thì sau phần tên là dãy lệnh khai báo các tham số được viết giới hạn trong cặp ngoặc tròn:

```
procedure Init;    {thủ tục không tham số}
procedure DrawFig(i, j: integer; w, h: word);    {thủ tục có tham số}
```

Dãy lệnh khai báo tham số gồm nhiều lệnh, ngăn cách nhau bởi dấu chấm phẩy. Mỗi lệnh khai báo gồm tên (hình thức) của tham số (trước tên có thể có từ khóa VAR), dấu hai chấm và kiểu của tham số. Nếu có nhiều tham số cùng kiểu thì có thể gộp tên của chúng lại thành một danh sách, cách nhau dấu phẩy. Thí dụ thủ tục `DrawFig` ở trên có 2 tham số `i, j` kiểu `integer` và 2 tham số `w, h` kiểu `word`. Kiểu của tham số có thể là bất cứ kiểu gì đã được định nghĩa, nhưng phải được khai báo qua tên (định danh). Thí dụ, khai báo dưới đây là không được phép:

```
procedure Enter(var a: array[1..10] of integer);
```

mà phải thay bằng cách đặt tên cho kiểu `array[1..10] of integer` như dưới đây:

```
Type
  arr = array[1..10] of integer;
  procedure Enter(var a: arr);
```

Để gọi một thủ tục không tham số, ta chỉ cần viết tên thủ tục trong câu lệnh. Thí dụ lệnh:

```
Init;
```

sẽ gọi thủ tục `Init` thực hiện chức năng của mình. Nếu thủ tục có chứa tham số thì ta phải truyền các giá trị tham số cho lời gọi thủ tục. Các giá trị này phân cách nhau bởi dấu phẩy, theo đúng thứ tự và đảm bảo tính tương thích kiểu như đã khai báo. Thí dụ lệnh:

```
DrawFig(-1, 2, 100, 60);
```

sẽ gọi thủ tục `DrawFig` thực hiện với các giá trị `i` = -1, `j` = 2, `w` = 100 và `h` = 60.

Phần khai báo của thủ tục giống như phần khai báo trong một chương trình, nó chứa các đối tượng riêng mà thủ tục đó dùng, bao gồm các hằng, biến, kiểu và các chương trình con (nếu có) của thủ tục.

Phần câu lệnh của thủ tục mô tả những công việc mà thủ tục đó phải thực hiện, cũng được bắt đầu bằng từ khóa BEGIN và kết thúc bằng từ khóa END giống như một chương trình, tuy nhiên sau END là một dấu chấm phẩy thay vì dấu chấm (dấu chấm chỉ được dùng khi kết thúc chương trình).

Thí dụ dưới đây xây dựng thủ tục `Center(s: st80)`, hiển thị nội dung xâu ký tự `s` thuộc kiểu `st80` (dài tối đa 80 ký tự) vào vị trí giữa dòng màn hình. Thủ tục `Center` dùng thủ tục chuẩn `Writeln` đưa ra màn hình nội dung xâu `s` sau một số dấu trắng (space) thích hợp.

```
Program Thidu;
Type
  st80 = string[80];

  procedure Center(s: st80);
  begin
    Writeln(' ':(80-Length(s)) div 2, s);
  end;

Begin
  Center('Hello');
  Center('This is DEMO! Press Enter to quit');
  Readln;
End.
```

2.2. Hàm

Hàm là một chương trình con nhằm tính toán và trả lại một giá trị nào đấy. Cũng như thủ tục, hàm gồm 3 phần: phần đầu (tiêu đề), phần khai báo và phần câu lệnh.

Phần đầu của hàm gồm từ khóa FUNCTION, tiếp theo là tên hàm và dãy lệnh khai báo tham số (nếu có). Điểm khác biệt với thủ tục là phần này cần khai báo thêm kiểu giá trị trả lại của hàm, sau dấu hai chấm. Turbo Pascal chỉ cho phép kiểu giá trị của hàm là một trong những kiểu chuẩn của nó (hoặc kiểu con của những kiểu này) đã được định danh như `byte`, `integer`, `LongInt`, `char`, `string`, `boolean`, ..., không cho phép các kiểu phức hợp như `record`, `array`.

Phần khai báo của hàm giống như đã trình bày trong thủ tục.

Phần câu lệnh của hàm thực hiện chức năng của nó, cũng được viết trong phạm vi hai từ khóa là BEGIN và END (kết thúc bởi dấu chấm phẩy). Tuy nhiên cần đảm bảo rằng, khi ra khỏi hàm, giá trị của nó phải được xác định bằng một lệnh gán nào đó cho tên hàm (không có phần tham số). Giá trị gán cho tên hàm phải đảm bảo tương thích với kiểu hàm đã khai báo.

Khác với thủ tục, lời gọi hàm là lời gọi giá trị của nó để sử dụng, vì thế nó cần được xuất hiện ở những chỗ thích hợp như giá trị của toán hạng trong biểu thức, giá trị ở vế phải trong một phép gán, giá trị truyền vào tham số của thủ tục, hàm ...

Thí dụ dưới đây xây dựng hàm `UCLN(a, b: integer)` trả về ước chung lớn nhất của 2 số nguyên (dương) `a`, `b`. Giá trị ước chung lớn nhất được tính theo thuật toán Euclide:

```
Program Thidu;

  function UCLN(a, b: integer): integer;
  begin
    while a <> b do
    if a > b then a := a-b else b := b-a;
    UCLN := a;        {gán giá trị cho tên hàm}
  end;

Var a, b: integer;
Begin
  Write('Cho cac so nguyen duong a, b: ');
  Readln(a, b);
  Writeln('UCLN cua cac so da cho la: ', UCLN(a, b));
  Readln;
End.
```

Việc tổ chức các chương trình con dưới hai dạng hàm và thủ tục là nét riêng biệt của Pascal, nhằm thích hợp với thói quen của người sử dụng. Thông thường khi cần thực hiện một công việc nào đó người ta dùng thủ tục, còn khi cần tính một giá trị nào đó, người ta dùng hàm. Tuy nhiên điều này là không bắt buộc. Giữa hàm và thủ tục có thể chuyển đổi cho nhau: bất cứ công việc nào được thực hiện bởi thủ tục cũng đều thực hiện được bằng hàm và bất cứ giá trị nào được tính bằng hàm cũng đều tính được nhờ thủ tục. Hiện nay người ta có xu hướng dùng tất cả các chương trình con dưới dạng hàm (chẳng hạn trong ngôn ngữ C) ngay cả khi không quan tâm đến giá trị trả lại. Trong những trường hợp như vậy, người ta thường gán cho hàm một giá trị lôgic nhằm ghi nhận trạng thái thành công hay thất bại của công việc.

Thí dụ, giả sử đã định nghĩa kiểu dữ liệu `matrix` để lưu trữ một ma trận. Để tính ma trận nghịch đảo của một ma trận, người ta có thể xây dựng thủ tục:

```
procedure Invert(var A: matrix; var Ok: boolean);
```

trong đó biến `A` nhận giá trị đầu vào của ma trận. Kết quả ma trận nghịch đảo được gửi lại vào `A`. Biến lôgic `Ok` dùng để ghi nhận trạng thái của kết quả: nếu ma trận ban đầu có

nghịch đảo thì `Ok` nhận giá trị `True`, trái lại nó nhận giá trị `False`. Lời gọi dùng thủ tục trên sẽ là:

```
Invert(A, Ok);
if not Ok then Writeln('Ma tran suy bien')
else ...    {A là ma trận nghịch đảo cần tìm}
```

Thủ tục trên có thể thay thế bằng hàm như sau:

```
function Invert(var A: matrix): boolean;
```

trong đó biến `A` được dùng giống như cũ, giá trị trả về của hàm là `True` nếu `A` có nghịch đảo và `False` nếu trái lại.

Khi đó lời gọi hàm tương đương sẽ là:

```
if not Invert(A) then Writeln('Ma tran suy bien')
else ...    {A là ma trận nghịch đảo cần tìm}
```

2.3. Thoát khỏi chương trình con

Thông thường, khi thực hiện hết các lệnh (trong phần câu lệnh) thì chương trình con tự kết thúc. Tuy nhiên có thể thoát khỏi chương trình con ở bất cứ chỗ nào trong phần câu lệnh bằng cách gọi thủ tục `Exit` (không tham số). Thủ tục này kết thúc chương trình con hiện hành và nhảy đến lệnh sát ngay sau lời gọi chương trình con này. Có thể coi thủ tục `Exit` như là một lệnh `goto` đến một vị trí được xác định không cần nhãn. Nhờ thủ tục `Exit` mà ta có thể bố trí nhiều điểm thoát khỏi chương trình con tùy thuộc vào những điều kiện khác nhau (chẳng hạn xây dựng hàm trả lại giá trị được tính bằng nhiều công thức). Hiện nay trong lập trình cấu trúc, người ta có xu hướng không dùng câu lệnh `goto` vì nó phá vỡ tính cấu trúc của chương trình. Giải pháp dùng `Exit` trong các chương trình con cho phép dễ dàng viết chương trình mà không cần dùng bất cứ một lệnh `goto` nào.

Ngoài thủ tục `Exit` dùng để thoát khỏi chương trình chứa nó, Turbo Pascal còn xây dựng thủ tục `Halt` cho phép kết thúc chương trình bất cứ lúc nào. Điểm khác biệt giữa `Halt` và `Exit` là cấp độ thoát: `Exit` chỉ thoát khỏi chương trình con chứa nó (nghĩa là vẫn nằm trong lòng của chương trình gọi chương trình con này), còn `Halt` sẽ kết thúc chương trình chính (chương trình ở mức ngoài cùng) dù lời gọi nó ở bất cứ đâu. Như thế `Exit` chỉ thực sự tương đương `Halt` nếu như dùng nó ở chương trình chính.

Ghi chú. Phiên bản 7 của Turbo Pascal thiết kế thêm các thủ tục `Break`, `Continue` cho phép ngắt (hoặc lặp lại) một vòng lặp (`while`, `repeat`, `for`) giữa chừng. Thủ tục `Break` thoát khỏi vòng lặp chứa nó, giống như một lệnh `goto` đến một nhãn có địa chỉ ngay sau lệnh lặp tương ứng. Thủ tục `Continue` làm cho vòng lặp được tiếp tục thực hiện với lần lặp tiếp theo, giống như một lệnh `goto` đến một nhãn có địa chỉ ngay sát trên lệnh lặp tương ứng. Việc dùng các thủ tục này cho phép bố trí nhiều điểm thoát khỏi (hoặc lặp lại) vòng lặp. Cùng với việc dùng `Exit`, `Halt`, các thủ tục `Break`, `Continue` cho phép dễ dàng viết các chương trình Pascal không dùng lệnh `goto`.

3. Địa phương và toàn cục

Những đối tượng (hằng, biến, kiểu, hàm, thủ tục) khai báo trong chương trình chính được dùng cho toàn bộ khối chương trình và các khối con khác của nó. Người ta gọi các đối tượng này là *toàn cục (chung)*. Cũng với những đối tượng đó, nếu khai báo trong một chương trình con, thì chỉ được dùng trong chương trình con đó và trong các khối con của nó. Người ta gọi những đối tượng này là *địa phương (riêng)*. Như vậy khái niệm địa phương hay toàn cục có tính chất tương đối: một đối tượng là chung cho các khối này lại là riêng của một số khối khác. Việc cấp phát bộ nhớ cho các khối khác nhau là độc lập, vì thế hai đối tượng cùng tên có thể nằm ở hai khối khác nhau. Điều này đảm bảo tính độc lập và toàn vẹn dữ liệu khi xây dựng chương trình. Thí dụ dưới đây minh họa sự khác nhau giữa biến địa phương và toàn cục:

```
Program Thidu;
Var i: integer;

  procedure Test;
  var i: integer;
  begin
    i := 5;
  end;

Begin
  i := 1;
  Test;
  Writeln('i = ', i);
  Readln;
End.
```

Khi chạy chương trình, ta nhận được thông báo `i = 1`, nghĩa là thủ tục `Test` không gây ảnh hưởng gì đến giá trị của `i`, vì biến `i` trong `Test` khác với biến `i` của chương trình. Thí dụ này cũng chứng tỏ nếu biến riêng trùng tên với biến bên ngoài thì chương trình con luôn ưu tiên biến riêng của nó (và như thế không thể truy cập biến bên ngoài được).

Nếu bạn bỏ câu lệnh khai báo biến `i` trong chương trình chính thì sẽ nhận được lỗi *unknown identifier* khi biên dịch câu lệnh `Writeln('i = ', i);` Điều đó có nghĩa việc khai báo biến `i` trong `Test` chỉ có tác dụng riêng trong nó. Nếu bạn bỏ câu lệnh khai báo biến `i` trong `Test` thì chương trình chạy bình thường nhưng với kết quả thông báo `i = 5`, nghĩa là biến `i` dùng trong `Test` cũng chính là biến `i` bên ngoài.

Việc tổ chức các đối tượng này là địa phương trong khi các đối tượng kia là toàn cục, hoàn toàn phụ thuộc vào chức năng cần thiết kế cho chúng. Nói chung, những đối tượng nào tham gia các khâu vào, ra chung của chương trình, hoặc có nhiệm vụ chuyển giao các giá trị giữa các chương trình con, thì cần phải được tổ chức toàn cục. Trái lại những đối tượng nào chỉ phát sinh khi giải quyết các công việc riêng lẻ thì cần phải tổ chức địa phương. Việc tùy tiện khai báo các biến, bất kể tại vị trí nào trong chương trình (chỉ cốt

để không mắc lỗi *unknown identifier*) sẽ dẫn đến các lỗi lôgic (chương trình chạy thông nhưng cho kết quả sai) rất khó phát hiện và sửa chữa. Thí dụ dưới đây cho thấy lỗi của việc khai báo biến riêng trong khi cần phải dùng biến chung:

```
Program Thidu;
Var a, b: integer;

  procedure Nhap;
  var a, b: integer;        {cần phải bỏ khai báo này}
  begin
    Write('Nhap vao 2 so nguyen a, b: ');
    Readln(a, b);
    Writeln('Cac gia tri da nhap: a = ', a, ', b = ', b);
  end;

Begin
  Nhap;
  Writeln('Tong cua 2 so da nhap la: ', a+b);
End.
```

Chương trình đơn giản này thông báo tổng của 2 số nguyên được nhập từ bàn phím qua việc dùng thủ tục `Nhap`. Tuy nhiên kết quả không như mong muốn. Lỗi ở đây là thủ tục `Nhap` khai báo các biến chứa các giá trị được nhập như những biến riêng, trong khi chương trình chính cần dùng chúng như những biến toàn cục. Kết quả sẽ như mong đợi nếu bỏ đi câu lệnh khai báo các biến `a, b` trong thủ tục `Nhap`.

Thí dụ dưới đây minh họa tình huống ngược lại: một biến dùng chung không đúng nhiều khi gây ra những giá trị không thể kiểm soát được:

```
Program Thidu;
Var i, k: integer;
    s, t: string;

  function Found(c: char; s: string): boolean;
  {cần khai báo biến i ở đây}
  begin
    i := 1;
    while (i <= Length(s)) and (c <> s[i]) do Inc(i);
    Found := i <= Length(s);
  end;

Begin
  Write('Vao xau nguon: '); Readln(t);
  Write('Vao xau dich : '); Readln(s);
  k := 0;
  for i := 1 to Length(t) do
    if Found(t[i], s) then Inc(k);
  Write('So ky tu cua xau nguon tim thay trong xau dich la: ', k);
  Readln;
```

```
End.
```

Chương trình nhập từ bàn phím hai xâu ký tự. Xâu đầu gọi là xâu nguồn, xâu sau gọi là xâu đích. Chương trình tiến hành đếm số ký tự của xâu nguồn có mặt trong xâu đích và đưa số đếm được ra màn hình. Việc kiểm tra một ký tự của xâu nguồn có mặt trong xâu đích được thực hiện bởi hàm lôgic `Found(c, s)`, trong đó `s` là xâu đích và `c` là ký tự được kiểm tra. Chương trình đủ đơn giản để bạn có thể khẳng định được tính đúng đắn của các câu lệnh. Tuy nhiên khi chạy chương trình, bạn sẽ thấy rằng kết quả là không ổn định: cùng giá trị các xâu nguồn và đích, chỉ thay đổi trật tự các ký tự trong xâu, chương trình cho những số đếm được khác nhau mà lẽ ra số này không thay đổi. Lỗi của chương trình không nằm trong phần các câu lệnh tính toán mà nằm trong việc không khai báo biến `i` là biến riêng của hàm `Found`. Chú ý rằng, hàm `Found` được gọi trong thân vòng lặp `for i` của chương trình chính, vì thế việc dùng biến `i` mà không khai báo nó là biến riêng của hàm `Found`, sẽ có nguy cơ làm hỏng giá trị của biến điều khiển `i` của vòng lặp này (như thí dụ đã làm). Để chương trình chạy đúng, chỉ cần bạn bổ sung thêm lệnh khai báo biến `i` là biến riêng của hàm `Found`, khi ấy, việc thay đổi giá trị của biến `i` trong hàm này sẽ không ảnh hưởng gì đến giá trị của biến `i` bên ngoài. Nói chung, các biến phát sinh nội bộ như các biến điều khiển vòng lặp, biến lưu trữ các giá trị trung gian, ... cần phải khai báo như những biến địa phương. Những lỗi như trong thí dụ vừa nêu là rất khó phát hiện, đặc biệt trong các chương trình phức tạp. Những người lập trình kinh nghiệm thường rất cẩn thận trong khâu thiết kế các biến chung, riêng trước khi viết chương trình.

Các ngôn ngữ lập trình có tính cấu trúc cao (như Pascal) thường đòi hỏi rất chặt chẽ về mặt khai báo. Điều này có vẻ nhiêu khê cho nhiều người, nhất là những người mới học lập trình. Thật ra, đây lại là điều có lợi: trước hết, nó hạn chế thói quen tùy tiện khi lập trình (mà khi có thói quen này, bạn chỉ có thể viết được những chương trình đơn giản), sau nữa nó nâng cao được tính ổn định của chương trình, hạn chế được những hiểm họa tiềm ẩn do có những lỗi lôgic chưa được phát hiện, tiết kiệm được công sức khi phải sửa lỗi hoặc phát triển chương trình, đặc biệt trong việc xây dựng các chương trình lớn, cần nhiều người tham gia.

4. Cách dùng tham số

4.1. Vai trò của tham số

Các tham số khai báo ở đầu chương trình con dùng để gửi các giá trị vào để chương trình xử lý, cũng như là nơi lấy các kết quả ra mà chương trình đã xử lý xong. Có thể nói, các tham số là công cụ để chương trình con giao tiếp với môi trường bên ngoài.

Các tên tham số khai báo được dùng trong chương trình con như tên các biến địa phương đã được khai báo, vì thế không được khai báo các biến riêng của chương trình con trùng với tên các tham số của nó. Điểm khác biệt giữa tham số và biến riêng của chương trình con là khi bắt đầu tham gia các câu lệnh, các tham số chỉ có tên chứ chưa có giá trị như các biến. Chỉ khi xuất hiện lời gọi chương trình con, các tham số mới được truyền giá

trị từ bên ngoài và chương trình con sẽ được hoạt động với bộ giá trị ấy. Cũng vì vậy chúng được gọi là các *tham số hình thức*, và khi gọi chương trình con, các tham số này có thể nhận bất cứ một tên biến nào khác với tên hình thức đã khai báo, miễn là giá trị của những tên này là tương thích kiểu với kiểu của tham số tương ứng. Các tên bên ngoài được truyền vào chương trình con được gọi là các *tham số thực* vì chúng thực sự tham gia chương trình con với tư cách là một giá trị.

Khi truyền tham số cho chương trình con, cần phải đảm bảo rằng việc truyền là tương ứng 1-1, đúng thứ tự và tương thích kiểu. Thí dụ với thủ tục được khai báo:

```
procedure DrawFig(i, j: integer; w, h: word);
```

các lời gọi sau đây là có lỗi khi biên dịch:

```
DrawFig(5, -1, 7);          {không tương ứng 1-1}
DrawFig(-1, 3, 4, 1.5);     {không tương thích kiểu}
```

Hiện nay, hầu hết các ngôn ngữ bậc cao đều tổ chức hai cách truyền tham số khi gọi một chương trình con, đó là *truyền theo trị* và *truyền theo biến*. Việc truyền tham số theo các cách khác nhau nhiều khi gây ra những kết quả khác nhau, điều này dẫn đến những lỗi lôgic rất khó phát hiện. Phần dưới đây thảo luận sự khác nhau giữa hai cách truyền tham số và cách dùng chúng cho đúng để đạt được mục đích đã đề ra của chương trình.

4.2. Truyền theo trị

Việc truyền tham số theo trị được thực hiện qua bản sao. Giá trị bên ngoài (của hằng, biến, hàm hoặc biểu thức) được chép vào một vùng nhớ được cấp phát tương ứng với kích thước tham số. Chương trình con sẽ làm việc với dữ liệu chứa trong bản sao này theo những lệnh đã xây dựng cho tham số tương ứng. Vì vậy, việc truyền theo trị có những đặc điểm sau:

- Nếu trong chương trình con có những lệnh làm thay đổi giá trị của tham số hình thức thì những thay đổi này không có ảnh hưởng gì đến giá trị của biến được truyền ở đầu vào vì những thay đổi này chỉ được thực hiện trên bản sao tương ứng.
- Tốn một ít bộ nhớ và thời gian cho việc sao chép (tùy thuộc kích thước của tham số tương ứng).
- Cho phép giá trị ở đầu vào có thể là những giá trị của hằng, biến, hàm hoặc biểu thức. Nếu đầu vào truyền hàm hoặc biểu thức thì những giá trị này được tính trước khi truyền cho tham số.

4.3. Truyền theo biến

Việc truyền tham số theo biến được thực hiện vào chính địa chỉ của biến được truyền, nghĩa là mọi lệnh của chương trình con đối với tham số hình thức cũng chính là các lệnh đối với biến này. Vì vậy việc truyền theo biến có những đặc điểm sau:

- Nếu trong chương trình con có những lệnh làm thay đổi giá trị của tham số hình thức thì những thay đổi này cũng chính là những thay đổi trên biến được truyền.

- Không tốn thêm bộ nhớ và thời gian do không phải sao chép.
- Chỉ cho phép đầu vào là giá trị của biến.

4.4. Khai báo truyền theo trị và truyền theo biến

Một chương trình con có thể có những tham số này thì truyền theo trị, còn những tham số khác thì truyền theo biến. Việc phân biệt một tham số là truyền theo trị hay truyền theo biến được nằm trong lời khai báo tham số đó ở đầu chương trình con.

Lệnh khai báo gồm tên tham số, tiếp theo là kiểu của tham số (ngăn cách bởi dấu hai chấm) sẽ xác định tham số đó được truyền theo trị, còn nếu thêm vào lệnh khai báo này từ khóa VAR, trước tên tham số, thì sẽ xác định tham số đó được truyền theo biến.

Thí dụ, thủ tục

```
Init(a: integer; var b, c: integer);
```

có 3 tham số nguyên, trong đó tham số đầu truyền theo trị, còn 2 tham số sau truyền theo biến.

Người ta thường gọi ngắn gọn các tham số truyền theo trị là các *tham trị*, còn các tham số truyền theo biến là các *tham biến*.

4.5. Hiệu ứng lề

Việc truyền theo trị và truyền theo biến nhiều khi dẫn đến những kết quả lôgic khác nhau. Sự khác nhau này được gọi là hiệu ứng lề. Thí dụ dưới đây cho thấy hiệu ứng lề sẽ cho những kết quả không như suy luận thông thường:

```
Program Thidu;

  function F(var x: integer): integer;
  begin
    x := x+1;
    F := x;
  end;

Var a: integer;
Begin
  a := 5;
  Writeln(F(a)+F(a));
  Readln;
End.
```

Chương trình đơn giản trên đưa ra màn hình giá trị `F(a)+F(a)` với `a` = 5. Bằng suy luận thông thường, kết quả đúng phải là 12 vì tại `a` = 5, `F(a)` cho giá trị 6. Tuy nhiên, khi chạy chương trình bạn sẽ nhận được thông báo 13. Bây giờ sửa biểu thức `F(a)+F(a)` bằng biểu thức `2*F(a)` và chạy lại chương trình, bạn sẽ nhận được kết quả 12. Sự khác nhau này thật khó giải thích cho một người chỉ biết làm toán mà không quen lập trình.

Thật sự, chương trình vẫn thực hiện đúng những lệnh mà con người viết ra, chỉ có điều ở đây xuất hiện hiệu ứng lề, do hàm `F` được tổ chức truyền theo biến đối với đối số `x` của

nó. Lệnh `x := x+1` trong hàm `F` sẽ làm biến `a` tăng lên 1 đơn vị mỗi khi gọi `F(a)`. Khi thực hiện biểu thức `F(a)+F(a)`, giá trị `F(a)` được gọi hai lần. Tại lần gọi thứ nhất `a` = 5, do đó `F(a)` = 6, tại lần gọi thứ hai `a` = 6 do đó `F(a)` = 7 và ta nhận được kết quả 13. Trong khi đó, biểu thức `2*F(a)` chỉ gọi giá trị `F(a)` một lần, vì thế ta nhận được kết quả 12. Nếu sửa lại việc truyền cho đối số `x` của hàm `F` là theo trị thì không còn sự khác nhau như vậy nữa.

Việc tổ chức truyền theo trị hay truyền theo biến cho một tham số là không thể tùy tiện vì nó có thể dẫn đến những kết quả không mong muốn. Hai thí dụ dưới đây minh họa cả hai tình huống có thể xảy ra. Thí dụ thứ nhất cho thấy một kết quả sai khi truyền theo trị, trong khi thí dụ thứ hai cho thấy một kết quả sai khi truyền theo biến.

Thí dụ 1.

```
Program TD1;

  procedure SwapInt(x, y: integer);
  var t: Integer;
  begin
    t := x; x := y; y := t;
  end;

Var x, y: integer;
Begin
  x := 1; y := 2;
  SwapInt(x, y);
  Writeln('x = ', x, ', y = ', y);
  Readln;
End.
```

Trong thí dụ này, thủ tục `SwapInt(x, y)` dùng để đổi giá trị giữa hai biến nguyên `x` và `y`. Tuy nhiên, khi chạy chương trình, điều này không xảy ra. Giá trị của hai biến `x` và `y`, trước và sau khi gọi thủ tục `SwapInt(x, y)` vẫn không đổi. Lỗi xảy ra do thủ tục `SwapInt(x, y)` tổ chức truyền theo trị nên các giá trị của các biến `x` và `y` không bị ảnh hưởng bởi các lệnh đổi giá trị trong thủ tục này. Nếu sửa lại việc khai báo các tham số trong thủ tục `SwapInt` là truyền theo biến (thêm từ khóa VAR trước các tên tham số `x, y`) thì chương trình sẽ cho kết quả như mong muốn.

Thí dụ 2.

```
Program TD2;

  function UCLN(var a, b: integer): integer;
  begin
    while a <> b do
    if a > b then a := a-b else b := b-a;
    UCLN := a;
  end;
```

```
Var tu, mau, d: integer;
Begin
  Write('Cho tu so: '); Readln(tu);
  Write('Cho mau so: '); Readln(mau);
  d := UCLN(tu, mau);
  Writeln('Dang toi gian cua phan so da cho la ',
          tu div d, '/', mau div d);
  Readln;
End.
```

Chương trình trên đưa ra màn hình dạng tối giản của một phân số khi nhập từ bàn phím các giá trị tử số và mẫu số của nó. Chương trình xây dựng hàm `UCLN(a, b)` trả về ước chung lớn nhất của hai số nguyên (dương) `a`, `b` và dùng hàm này để tính `d` là ước chung lớn nhất của tử và mẫu đã nhập. Dạng tối giản của phân số nhận được bằng cách cùng chia tử và mẫu của phân số cho `d`. Tính đúng đắn của các công thức không có gì phải bàn. Tuy nhiên khi chạy chương trình, bạn sẽ luôn nhận được kết quả là 1/1 cho mọi phân số. Lỗi lôgic này xảy ra do hàm `UCLN` được tổ chức truyền theo biến, nên sau lời gọi `d := UCLN(tu, mau)`, ta được đồng thời các giá trị `d`, `tu`, `mau` bằng nhau. Nếu sửa lại việc khai báo các tham số trong hàm `UCLN` là truyền theo trị (bỏ từ khóa VAR trước các tên `a`, `b`) thì chương trình sẽ cho kết quả đúng.

4.6. Truyền theo trị hay truyền theo biến?

Từ những thảo luận trên, ta có thể rút ra một số kết luận dưới đây về việc truyền tham số cho chương trình con:

1. Nếu trong thân chương trình con không có những lệnh làm thay đổi giá trị tham số hình thức thì việc truyền theo biến hay truyền theo trị không gây ra một sự khác biệt nào về kết quả mà chương trình con trả lại.

2. Trong trường hợp, chương trình con có những lệnh làm thay đổi giá trị tham số hình thức thì phải tùy vào nhiệm vụ của chương trình con đối với tham số mà quyết định cách truyền:

- Nếu chương trình con có nhiệm vụ phải thay đổi giá trị tham số thực thì phải tổ chức truyền theo biến cho tham số này. Thí dụ các tham số nhận giá trị ở đầu ra trong các thủ tục nhập, hoán chuyển, cập nhật, ... Thủ tục `Nhap(x)` có nhiệm vụ nhập giá trị cho biến `x` sẽ chẳng có tác dụng gì nếu nó được tổ chức truyền theo trị.
- Nếu tham số thực được dùng để gửi giá trị ở đầu vào cho chương trình con hoạt động mà không được làm hỏng giá trị này sau khi gọi chương trình con, thì phải tổ chức truyền theo trị cho tham số này. Có rất nhiều tình huống cần đảm bảo sự toàn vẹn của các dữ liệu vào sau khi đã được các chương trình con dùng. Thí dụ 2 trong **4.5** là một minh họa.

3. Trong các trường hợp việc truyền theo biến hay theo trị không ảnh hưởng đến kết quả lôgic của chương trình thì người ta thường chọn cách truyền theo trị cho các tham số

thuộc kiểu chuẩn như `integer`, `char`, `boolean`, `real`, ... vì nó cho phép truyền hằng, hàm, biểu thức (những tham số loại này kích thước không đáng kể), còn các tham số thuộc kiểu phức hợp như mảng, bản ghi, ... thì truyền theo biến để tiết kiệm bộ nhớ, tăng tốc độ khi truyền (những tham số loại này không có hằng, hàm, biểu thức). Riêng tham số kiểu `File` hoặc tham số không định kiểu thì luôn phải tổ chức truyền theo biến.

5. Cấp phát bộ nhớ cho chương trình con.

Việc cấp phát bộ nhớ cho các biến trong chương trình con (biến địa phương) được thực hiện trong vùng stack (ngăn xếp), trong khi việc cấp phát cho các biến trong chương trình chính (biến toàn cục) được thực hiện trong vùng data (dữ liệu). Trong quá trình chạy chương trình, mỗi khi gọi chương trình con, các biến của nó sẽ được cấp phát cho đến khi chương trình con kết thúc, các biến này lại tự động giải phóng. Như thế vùng cấp phát cho chương trình con được gọi ở mức trong cùng sẽ được giải phóng đầu tiên (vào sau ra trước). Nhờ vậy các biến của chương trình chính là độc lập với các biến của chương trình con và các biến của chương trình con này là độc lập với các biến của chương trình con khác, mặc dù chúng trùng tên.

Nếu khai báo quá nhiều biến địa phương cho một chương trình con thì khi gọi chương trình con này có thể bị lỗi tràn stack (*stack overflow*). Lỗi này cũng gặp khi hệ thống phải cung cấp cho một chuỗi quá dài lời gọi các chương trình con lồng nhau (chẳng hạn lời gọi chương trình con đệ quy). Hiện nay, phiên bản 7 của Turbo Pascal ngầm định dành sẵn 16 Kb cho vùng stack. Đây là kích thước phù hợp với nhiều ứng dụng thông thường. Bạn có thể nới rộng kích thước này (không quá 64Kb) bằng cách dùng định hướng biên dịch `{$M StackSize, HeapMin, HeapMax}` trong chương trình để tăng giá trị `StackSize` hoặc vào chức năng Options/Memory Size của môi trường Turbo Pascal đặt lại giá trị cho kích thước stack.

6. Chương trình con đệ quy

Thông thường, một chương trình con được gọi khi nó đã được xây dựng trước đấy. Tuy nhiên, một số ngôn ngữ lập trình (trong đó có Pascal) cho phép một chương trình con, trong khi đang xây dựng, lại được gọi chính nó. Những chương trình con như vậy được gọi là *chương trình con đệ quy*. Một chương trình con được xây dựng bằng đệ quy cũng giống như một khái niệm được xây dựng bằng quy nạp trong toán học.

Để minh họa, ta xét khái niệm giai thừa trong toán học. Định nghĩa n giai thừa (n!), với n nguyên dương có thể đưa ra bằng hai cách:

Cách 1:

$n! = 1.2.3. \ldots .(n-1).n$

Cách 2:

$1! = 1$

$n! = (n-1)!.n$

Với cách đầu, n! được định nghĩa qua phép nhân, đó là cách định nghĩa thông thường: một khái niệm chưa biết (giai thừa) được xây dựng từ một khái niệm đã biết (phép nhân). Với cách sau, n! được định nghĩa quy nạp (hay đệ quy): một khái niệm chưa biết (giai thừa) được xây dựng qua chính nó. Dĩ nhiên để đảm bảo tính xác định, quá trình đệ quy phải có điểm dừng. Định nghĩa 1! = 1 trong công thức trên là điểm như vậy. Nhờ nó mà định nghĩa n! = (n-1)!.n xác định được n! với mọi giá trị n nguyên, dương.

Trong Pascal, ta có thể xây dựng hàm tính n! bằng cả hai cách đã trình bày:

Cách 1:

```
function GiaiThua(n: integer): LongInt;
var i: integer;
    p: LongInt;
begin
  p := 1;
  for i := 1 to n do p := p*i;
  GiaiThua := p;
end;
```

Cách 2:

```
function GiaiThua(n: integer): LongInt;
begin
  if n < 2 then GiaiThua := 1
  else GiaiThua := GiaiThua(n-1)*n;
end;
```

Hàm `GiaiThua` xây dựng theo cách đầu là hàm bình thường. Hàm `GiaiThua` xây dựng theo cách sau là hàm đệ quy (trong hàm có lời gọi chính nó). Cả hai hàm đều cho phép tính n! với n nguyên ≥ 0 (quy ước 0! =1).

Mặc dù về lôgic, hai hàm trên cho kết quả như nhau, nhưng về mặt hệ thống, chúng hoàn toàn được thực hiện khác nhau. Khi tính n!, hàm thứ nhất được gọi 1 lần không phụ thuộc vào giá trị n, trong khi hàm thứ hai được gọi một số lần tỷ lệ với n.

Một điểm lợi hiển nhiên khi dùng đệ quy là tiết kiệm công sức cho người lập trình. Hầu như hàm `GiaiThua` trong cách sau là chép lại công thức quy nạp. Một bài toán phát biểu bằng quy nạp toán học dễ dàng được cài đặt bằng đệ quy. Tuy nhiên việc dùng đệ quy thường phải trả giá cho việc tốn kém bộ nhớ và số phép toán thực hiện. Để minh họa cho điều này, ta xét thí dụ dưới đây.

Trong toán học, dãy Fibonaci $F_1, F_2, ..., F_n, ...$ được định nghĩa quy nạp như sau:

$F_1 = F_2 = 1$

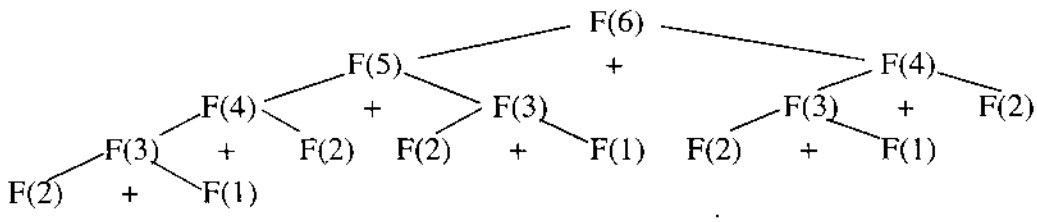
$F_n = F_{n-1}+F_{n-2}$ (n = 3, 4, ...)

Bây giờ cần xây dựng hàm F(n) trả về số hạng thứ n của dãy Fibonaci. Bằng cách chép hầu như nguyên văn công thức trên, ta có hàm đệ quy:

```
function F(n: integer): LongInt;
begin
  if n < 3 then F := 1
```

```
  else F := F(n-1)+F(n-2);
end;
```

Giả sử cần tính F(6). Khi đó lời gọi F(6) sẽ kéo theo các lời gọi dưới đây:



Trong sơ đồ trên, số phép cộng phải thực hiện là 7, trong đó F(4) phải tính lại 2 lần, F(3) phải tính lại 3 lần. Các con số này được tăng lũy tiến theo n. Vùng nhớ stack được cấp phát tỷ lệ với độ sâu n.

Tuy nhiên, ta có thể xây dựng hàm F(n) không đệ quy như sau:

```
function F(n: integer): LongInt;
var x, y, z: LongInt;
    i: integer;
begin
  x := 1; y := 1;
  for i := 3 to n do
  begin
    z := x+y;
    x := y;
    y := z;
  end;
  F := y;
end;
```

Về mặt hình thức, hàm này viết phức tạp hơn hàm trước, tuy nhiên về mặt tính toán và sử dụng bộ nhớ, nó hiệu quả hơn nhiều. Các giá trị được tính đúng 1 lần, số phép cộng phải thực hiện bằng n-2 (tuyến tính theo n), bộ nhớ cấp phát không tăng theo n. Dưới đây là một vài giá trị để bạn so sánh:

n	31	32	33	34	35
số phép + (không đệ quy)	29	30	31	32	33
số phép + (đệ quy)	1346268	2178308	3524577	5702886	9227464

Trong kỹ thuật lập trình hiện nay, có cả một lĩnh vực chuyên nghiên cứu việc khử đệ quy nhằm nâng cao tính hiệu quả của chương trình. Nói chung, bạn không nên lạm dụng đệ quy trong những tình huống như thí dụ vừa nêu.

Mặc dù vậy, đệ quy là có ích khi phải giải những bài toán có tính quy nạp, nó tiết kiệm công sức lập trình rất nhiều. Nhiều tình huống nếu không dùng đệ quy thì việc cài đặt trở nên rất phức tạp. Một thí dụ điển hình là lời giải bài toán Tháp Hà nội.

Bài toán Tháp Hà nội.

Có một chồng gồm n cái đĩa, đĩa nhỏ được xếp trên đĩa lớn. Cần phải chuyển chồng đĩa này đến một vị trí khác theo nguyên tắc: mỗi lần chỉ chuyển 1 đĩa và chỉ được xếp đĩa nhỏ trên đĩa lớn. Trong quá trình chuyển, cho phép dùng một vị trí thứ ba làm trung gian.

Lời giải bài toán này được thực hiện bằng quy nạp khá đơn giản. Gọi 1, 2, 3 là số hiệu tương ứng với các vị trí xuất phát, tập kết và trung gian. Lời giải gồm các bước:

- Chuyển n-1 đĩa từ vị trí 1 đến vị trí 3 (vị trí 2 làm trung gian). Bước này được thực hiện nhờ giả thiết quy nạp.
- Chuyển 1 đĩa (đĩa lớn nhất) từ vị trí 1 đến vị trí 2.
- Chuyển n-1 đĩa từ vị trí 3 đến vị trí 2 (vị trí 1 làm trung gian). Bước này được thực hiện nhờ giả thiết quy nạp.

Lời giải này được cài đặt bằng một thủ tục đệ quy mà việc viết nó hầu như chép lại các bước đã nêu. Thủ tục `MoveDisk(n, a, b, c: integer)` trình bày dưới đây thực hiện việc chuyển `n` đĩa từ vị trí `a` đến vị trí `b` nhờ vị trí `c` làm trung gian:

```
procedure MoveDisk(n, a, b, c: integer);
begin
  if n > 0 then
  begin
    MoveDisk(n-1, a, c, b);
    Writeln(a, '-->', b);      {thông báo chuyển 1 đĩa}
    MoveDisk(n-1, c, b, a);
  end;
end;
```

Trong chương trình chính, sau khi nhập `n` (số đĩa), chỉ cần lời gọi `MoveDisk(n, 1, 2, 3)` là việc chuyển đĩa được thực hiện. Mặc dù lời gọi này là đơn giản, nhưng thực sự hệ thống phải thực hiện một vòng lặp với 2^n-1 bước chuyển đĩa. Bạn thử hình dung, nếu không dùng đệ quy thì việc cài đặt lời giải sẽ phức tạp như thế nào.

Một điểm cần chú ý khi thiết kế đệ quy là phải đảm bảo điều kiện dừng. Nếu quên điều này thì lời gọi đệ quy không kết thúc và sẽ gặp phải lỗi tràn stack. Lỗi này cũng sẽ gặp nếu độ sâu đệ quy là quá lớn so với kích thước stack mà hệ thống có thể cung cấp.

Để khắc phục việc tràn stack khi dùng đệ quy, bạn cần thực hiện một số kỹ thuật sau:

- Đảm bảo sự có mặt điểm dừng trong thân chương trình con đệ quy.
- Giảm tối đa các biến riêng cũng như các tham trị của chương trình con đệ quy, đặc biệt đối với những biến kích thước lớn (như mảng, bản ghi).
- Tăng kích thước stack mà hệ thống dành cho chương trình. Thông thường hiện nay Turbo Pascal ngầm định kích thước stack là16384 byte, bạn có thể tăng con số này lên (tối đa là 65520) bằng cách dùng định hướng biên dịch `{$M StackSize, HeapMin, HeapMax}` ở đầu chương trình.

Khả năng đệ quy của Pascal còn cho phép xây dựng các chương trình con gọi lẫn nhau: A gọi B trong khi B gọi A. Muốn vậy cần dùng từ khóa FORWARD để khai báo trước một chương trình con mà chưa cần biết nội dung của nó.

```
procedure Flip(n: integer); forward;

procedure Flop(n: integer);
begin
  Write('  Flop ', n);
  if n > 0 then Flip(n-1);
end;

procedure Flip;
begin
  Write('  Flip ', n);
  if n > 0 then Flop(n-1);
end;
```

Trong thí dụ trên, thủ tục `Flip` được khai báo trước (chưa có nội dung) với từ khóa FORWARD để thủ tục `Flop` viết sau đấy gọi nó mà không gặp lỗi. Nội dung của `Flip` được viết sau `Flop`, trong đó lại xuất hiện lời gọi `Flop`. Khi gặp lời gọi một thủ tục, chúng sẽ gọi lẫn nhau cho đến khi kết thúc. Chẳng hạn lời gọi `Flop(5)` sẽ cho kết quả trên màn hình:

```
 Flop 5  Flip 4  Flop 3  Flip 2  Flop 1  Flip 0
```

Cũng giống như đệ quy thông thường, trong các chương trình con gọi lẫn nhau phải đảm bảo có điểm dừng để việc gọi đệ quy kết thúc.

7. Thiết kế giao diện menu

Một số chương trình ứng dụng thực hiện nhiều chức năng một cách độc lập. Để người dùng có thể dùng bất kỳ chức năng nào một cách thuận tiện, người ta thiết kế nhiều chương trình con, trong đó mỗi chương trình con thực hiện một chức năng và chương trình chính chỉ có nhiệm vụ xây dựng một hệ giao diện menu để gọi các chương trình con này. Các dữ liệu dùng chung như dữ liệu giả thiết, dữ liệu kết quả, dữ liệu chuyển giao giữa các chương trình con, cần được lưu trữ trong các biến toàn cục. Các dữ liệu riêng phục vụ cho chương trình con nào, cần được lưu trữ trong các biến địa phương của chương trình con ấy. Việc xây dựng một hệ giao diện menu là việc kết hợp trình bày trên màn hình với việc dùng bàn phím và chuột để người dùng lựa chọn các đề mục một cách thuận tiện.

Dưới đây trình bày một mô hình thiết kế menu cho chương trình chính một cách đơn giản, trong đó dùng các thủ tục xử lý màn hình văn bản chuẩn và các hàm nhận phím do unit CRT của Turbo Pascal cung cấp. Mục đích của mô hình đơn giản này chỉ nhằm để

giúp việc thử các chức năng của chương trình một cách dễ dàng. Các bạn có thể thay đổi và phát triển mô hình để tăng tính mỹ quan và thuận lợi khi dùng.

```
Uses CRT;
{khai báo các biến toàn thể ở đây}

  function Menu: char;
  begin
    ClrScr;
    Writeln('        Main menu');
    Writeln('        =========');
    Writeln('        1. Item 1');
    Writeln('        2. Item 2');
    Writeln('        3. Help');
    Writeln('        4. Quit');
    Writeln;
    Write('        Enter a choice: ');
    repeat
      Menu := ReadKey;
    until Menu in ['1'..'4'];
  end;

  procedure Item1;
  {khai báo các biến địa phương ở đây}
  begin
    ClrScr;
    Writeln('This is Item 1');
    Write('Press Enter to return menu... ');
    Readln;
  end;

  procedure Item2;
  {khai báo các biến địa phương ở đây}
  begin
    ClrScr;
    Writeln('This is Item 2');
    Write('Press Enter to return menu... ');
    Readln;
  end;

  procedure Help;
  begin
    ClrScr;
    Writeln('This is Help');
    Write('Press Enter to return menu... ');
    Readln;
```

```
  end;

Begin
  repeat
    case Menu of
      '1': Item1;
      '2': Item2;
      '3': Help;
      '4': Halt;
    end;
  until false;
End.
```

Hàm Menu trình bày danh mục các chức năng của chương trình và chờ người dùng chọn bằng cách ấn phím tương ứng. Hàm trả về giá trị ký tự của phím được chọn. Tùy giá trị này mà chương trình rẽ nhánh đến chức năng tương ứng. Nội dung các chương trình con chức năng được cài đặt dần. Xong chương trình con nào, có thể thử chương trình con ấy mà không phụ thuộc vào các chương trình con khác.

8. Phương pháp Top-Down

Phương pháp lập trình Top-Down là phương pháp lập trình từ tổng thể đến chi tiết, cho phép phân rã một công việc phức tạp thành nhiều công việc đơn giản hơn.

Phương pháp Top-Down là phương pháp cơ bản hiện nay trong lập trình cấu trúc. Nhờ phương pháp này, việc lập trình mang tính công nghệ cao, cho phép xây dựng các chương trình lớn, nhiều người tham gia, thuận tiện cho việc bảo trì và mở rộng.

Tư tưởng chủ yếu của phương pháp này là từ yêu cầu vào/ra của bài toán, ta phân quá trình giải quyết thành những khối lớn, mỗi khối thực hiện một vấn đề nào đấy như một bài toán riêng biệt, được thực hiện bởi một chương trình con. Đến lượt mỗi khối, lại được phân rã thành những khối nhỏ hơn, nhằm giải quyết từng phần của khối đang xét, và cũng được thực hiện bởi chương trình con tương ứng, ... Các chương trình con thường được liên lạc với nhau qua các biến chung hoặc qua các tham số. Ngoài ra, trong khi xây dựng các chương trình con, bạn cũng cần nhận diện các chương trình con phục vụ cho nhiều chương trình (như các chương trình con thực hiện các thao tác vào/ra, tìm kiếm, sắp xếp, giao diện, ...). Những chương trình con này cần được thiết kế cẩn thận như một bài toán độc lập và được tổ chức các tham số thích hợp cho nhiều tình huống. Những chương trình con như thế được gọi là các *chương trình con thư viện* (hay *thường trình*), chúng cần được lưu lại thành những thư viện lập trình để dùng lâu dài. Nhờ các thư viện mà việc lập trình mang tính kế thừa, tiết kiệm được nhiều công sức lập trình cũng như nâng cao được độ tin cậy của các chương trình. Có thể nói, việc luyện tập lập trình cấu trúc là việc luyện tập phân tích và thiết kế các chương trình con.

Để lấy thí dụ, ta xét một bài toán đơn giản dưới đây:

Trên mặt phẳng, xét một dãy các đoạn thẳng, mỗi đoạn được xác định bởi hai điểm đầu mút của chúng. Mỗi điểm trên mặt phẳng được xác định bởi cặp số thực biểu diễn hoành độ, tung độ của nó. Vấn đề là cần xác định các lớp đồng phương của dãy đoạn đã cho. Lập trình giải bài toán đã nêu, dữ liệu nhập từ bàn phím, trong đó lần lượt từng đoạn thẳng được nhập các tọa độ của các điểm đầu mút. Kết quả đưa ra màn hình số lớp đồng phương và với mỗi lớp cần chỉ ra số hiệu các đoạn thuộc lớp đó. Các đoạn thẳng được đánh số từ 1, theo thứ tự nhập.

Trước hết về mặt tổ chức dữ liệu, kiểu bản ghi tỏ ra thích hợp trong việc xây dựng kiểu điểm và kiểu đoạn như sau:

```
Type
  KieuDiem  =  record
                 x, y: real;        {tọa độ của điểm}
               end;
  KieuDoan  =  record
                 A, B: KieuDiem;    {hai đầu mút của đoạn}
               end;
```

Khi đó, dãy các đoạn thẳng được lưu trữ vào một mảng `Doan` có các thành phần thuộc `KieuDoan`, mỗi thành phần `Doan[j]` lưu trữ đoạn số hiệu `j`, `j` = 1, 2, ... `n`, trong đó `n` là số đoạn thuộc dãy. Gọi số lớp đồng phương của các đoạn thẳng là `m` và đánh số các lớp này từ 1 đến `m`. Dùng mảng `SoHieu[j]`, `j` = 1, 2, ..., `n`, lấy giá trị nguyên trong khoảng từ 1 đến `m`, để ghi nhận số hiệu của lớp đồng phương chứa đoạn `j`. Giá trị ban đầu của các số hiệu này là 0 với ý nghĩa các đoạn tương ứng chưa được xét.

Về công việc của chương trình có thể chia thành 3 khối:

1. *Khối nhập dữ liệu*: Nhiệm vụ của khối này là nhập từ bàn phím số đoạn thuộc dãy vào biến `n` và các thông tin tọa độ của các đoạn thẳng vào các biến `Doan[j]`, `j` = 1, 2, ..., `n`. Để thực hiện khối này, ta xây dựng thủ tục `Nhap` (không tham số).

2. *Khối xác định các lớp đồng phương*: Nhiệm vụ của khối này là xác định các lớp đồng phương của các đoạn đã cho bằng cách gửi số lớp vào biến `m` và gửi số hiệu của lớp chứa đoạn `i` vào các biến `SoHieu[i]`, `i` = 1, 2, ..., `n`. Khối này được thực hiện bởi thủ tục `XacDinh` (không tham số).

3. *Khối đưa ra kết quả*: Nhiệm vụ của khối này là đưa ra màn hình kết quả đã tìm thấy, bao gồm số lớp đồng phương (giá trị `m`) và số hiệu của mỗi đoạn trong từng lớp. Thủ tục `XemKQ` (không tham số) sẽ thực hiện khối này.

Để xác định các giá trị `SoHieu[i]`, `i` = 1, 2, ..., `n`, ta cần gán cùng một giá trị cho các đoạn đồng phương. Vì vậy hàm `DongPhuong(i, j: integer): boolean`, trả về `True` nếu đoạn `i` đồng phương với đoạn `j`, được xây dựng trong chương trình để phục vụ cho thủ tục `XacDinh`. Cũng như vậy thủ tục `Xem(k: integer)` hiển thị lên màn hình chỉ số các đoạn thẳng thuộc lớp đồng phương `k`, được xây dựng để phục vụ cho thủ tục `XemKQ`.

Dưới đây là toàn văn chương trình:

```
Program Thidu;
Type
  KieuDiem = record
               x, y: real;   {tọa độ của điểm}
             end;
  KieuDoan = record
               A, B: KieuDiem;      {hai đầu mút của đoạn}
             end;
Var
  n, m: integer;
  Doan: array[1..50] of KieuDoan;
  SoHieu: array[1..50] of integer;

  procedure Nhap;
  {nhập thông tin các đoạn thẳng}
  var i: integer;
  begin
    Write('So doan thang: '); Readln(n);
    for i := 1 to n do
    begin
      Writeln('Vao toa do doan ', i);
      with Doan[i] do
      begin
         Write('  toa do dau mut thu nhat: ');
         Readln(A.x, A.y);
         Write('  toa do dau mut thu hai: ');
         Readln(B.x, B.y);
      end;
    end;
  end;

  function DongPhuong(i, j: integer): boolean;
  {kiểm tra các đoạn thẳng số hiệu i và j có đồng phương không}
  begin
    DongPhuong :=
    Abs((Doan[i].B.x-Doan[i].A.x)*(Doan[j].B.y-Doan[j].A.y)-
    (Doan[i].B.y-Doan[i].A.y)*(Doan[j].B.x-Doan[j].A.x)) < 1e-9;
  end;

  procedure XacDinh;
  {xác định các lớp đồng phương của các đoạn thẳng}
  var i, j: integer;
  begin
    for i := 1 to n do SoHieu[i] := 0;
    m := 0;
    for i := 1 to n do
```

```
    if SoHieu[i] = 0 then
    begin
      Inc(m);
      SoHieu[i] := m;
      for j := i+1 to n do
      if SoHieu[j] = 0 then
        if DongPhuong(i, j) then SoHieu[j] := m;
    end;
  end;

  procedure Xem(k: integer);
  {hiển thị lên màn hình các đoạn thẳng thuộc lớp đồng phương k}
  var i: integer;
  begin
    for i := 1 to n do
      if SoHieu[i] = k then Write(i:5);
    Writeln;
  end;

  procedure XemKQ;
  {hiển thị các lớp đồng phương lên màn hình}
  var k: integer;
  begin
    Writeln('So lop: ', m);
    Writeln('Cac lop:');
    for k := 1 to m do
    begin
      Write('Lop ', k, ': '); Xem(k);
    end;
  end;

Begin
  Nhap; XacDinh; XemKQ;
  Readln;
End.
```

2

GIAO TIẾP FILE

File là nơi cung cấp dữ liệu cho chương trình, cũng là nơi gửi kết quả đã được chương trình xử lý. Việc hiểu file một cách thấu đáo sẽ giúp cho người lập trình đọc dữ liệu không sai sót cũng như xuất kết quả đúng khuôn dạng yêu cầu. Ngoài ra, kỹ thuật dùng file còn giúp người lập trình khai thác một cách hiệu quả bộ nhớ ngoài, hỗ trợ tích cực cho bộ nhớ trong vốn đã hạn hẹp.

1. File và phân loại file

File là tập hợp của những dữ liệu được cất giữ bên ngoài máy tính, có liên quan đến nhau hoặc cùng phục vụ cho một chương trình trong máy tính và được tổ chức sao cho có thể được lưu trữ, tìm kiếm, xử lý và trao đổi như một thể thống nhất. Hiện nay, chủ yếu file được cất giữ trên các đĩa từ, vì thế khi nói đến file, người ta thường hiểu ngầm là những file trên đĩa.

Thông thường, file chứa những dữ liệu cung cấp cho chương trình, khi cần, chúng được nạp vào máy để chương trình xử lí. Kết quả đã xử lí của chương trình cũng thường được tổ chức dưới dạng file và được gửi ở đầu ra. Một cách tổng quát, trong các thao tác vào/ra hoặc truyền dữ liệu giữa các máy tính, có thể xem file là dòng dữ liệu vận chuyển từ thiết bị này sang thiết bị khác, bản thân mỗi thiết bị, do đó, cũng được coi là một file (chẳng hạn bàn phím, màn hình, máy in, ...).

File cũng có thể là những dữ liệu được cất tạm thời trong quá trình xử lý. Chúng hỗ trợ cho bộ nhớ trong khi bộ nhớ này không đủ. Nhiều phần mềm hiện nay đòi hỏi một dung lượng còn trống trên đĩa để "nháp". Những file như vậy sẽ được xóa đi khi chương trình đã dùng xong.

Người ta cũng quan niệm các lệnh máy là những dữ liệu. Vì thế những file lưu trữ các lệnh máy là những file đặc biệt. Chúng không cung cấp dữ liệu cho các chương trình khác mà dùng để thực hiện các lệnh đã ghi trong chúng. Người ta gọi những file này là những file *mã máy* hoặc những file *thực hiện*. Trong DOS, chúng được ghi nhận bởi phần mở rộng là .COM hoặc .EXE. Tuy nhiên trong khuôn khổ chương này, những file như vậy không bàn đến và thuật ngữ file ở đây được hiểu là những file dữ liệu thông thường.

Về mặt tổ chức dữ liệu, có thể phân những file này thành hai loại: file văn bản và các file dữ liệu có cấu trúc.

a. File văn bản (còn gọi là *Text File* hay *ASCII File*): là file mà trong đó dữ liệu được ghi dưới dạng các ký tự theo mã ASCII. File văn bản thường để lưu trữ các dữ liệu có dạng văn bản như các bản báo cáo, các bức thư, giấy mời, các tài liệu ghi chép (bài giảng, bài học,...). Nó cũng là dạng lưu trữ các chương trình nguồn của các ngôn ngữ lập trình như Pascal, C, Fortran, Basic, Foxbase, ... Vì mã ASCII là một chuẩn giao tiếp nên file văn bản thường được dùng như một công cụ chung để trao đổi dữ liệu giữa các chương trình. Các file văn bản được trực tiếp soạn thảo, in ấn một cách dễ dàng. Hầu hết các ứng dụng hiện nay đều có khả năng ghi và đọc các file văn bản.

b. File dữ liệu có cấu trúc: Đây là những file chứa những dữ liệu được tổ chức theo một quy cách nhất định. Để cảm nhận nội dung của file, cần phải hiểu quy cách này. Những file dữ liệu có cấu trúc thường là sản phẩm mang nét riêng biệt của những chương trình tạo ra chúng. Chẳng hạn các file cơ sở dữ liệu, các bảng tính, các file ảnh, các file âm thanh, ... Một số file văn bản đặc biệt như các file DOC của Microsoft Word cũng có thể xem như các file dữ liệu có cấu trúc. Các file dữ liệu có cấu trúc rất đa dạng, chúng phản ánh các kiểu dữ liệu khác nhau được tổ chức bên trong máy tính.

Người ta cũng phân loại các file theo cách thức truy cập. Đó là file *truy cập tuần tự* và file *truy cập trực tiếp*. Trong một file tuần tự, các mục dữ liệu phải được truy cập theo thứ tự chúng được lưu trữ, nghĩa là, để truy cập đến một thành phần nào đó, ta phải bắt đầu từ đầu file và đi qua tất cả các thành phần đứng trước. Ngược lại, các file truy cập trực tiếp cho phép định vị đến bất cứ nơi nào của file để thao tác bằng cách đặc tả vị trí của nó, thông thường là số hiệu thành phần. Tuy nhiên trong trường hợp cần thiết vẫn có thể xử lý tuần tự một file truy cập trực tiếp bằng cách đi lần lượt qua các thành phần của nó theo thứ tự của số hiệu.

Ngôn ngữ Pascal chuẩn chỉ cho phép các file tuần tự, nhưng nhiều phiên bản, trong đó có Turbo Pascal, cho phép cả hai.

2. Vai trò của file và những khía cạnh cần lưu ý

File là công cụ tổ chức lưu trữ dữ liệu để phục vụ cho chương trình. Việc tổ chức tốt hệ thống các file là một kỹ thuật quan trọng trong việc xây dựng các phần mềm hiện nay, đặc biệt với xu hướng các phần mềm ngày càng được phát triển tinh vi và chất lượng bộ nhớ ngoài ngày càng được nâng cao. Các phần mềm nổi tiếng hiện nay đều có một hệ thống file rất phong phú. Hiểu rõ cách tổ chức dữ liệu trên file, người lập trình mới có thể đọc vào dữ liệu không bị sai sót cũng như đưa ra kết quả theo đúng yêu cầu.

Ngoài vai trò là các dữ liệu đầu - cuối của chương trình, file còn là nơi lưu trữ các dữ liệu trung gian trong quá trình tính toán. Do bộ nhớ trong bị hạn chế nên nhiều lúc nó không thể đảm đương nổi việc lưu các kết quả tạm thời. Việc dùng file một cách thích hợp trong những tình huống như vậy là một giải pháp hiệu quả để có thể giải được những

bài toán kích thước lớn. Ngoài ra, có một số chương trình không thể chạy trong một khoảng thời gian liên tục, việc tổ chức các file cất giữ các kết quả trung gian trong từng giai đoạn là một một biện pháp cần thiết để chương trình có thể chạy nối tiếp.

Một điểm cần lưu ý là tốc độ truy cập file chậm hơn rất nhiều so với tốc độ truy cập bộ nhớ trong, vì thế người ta cố gắng tránh việc phải đọc đi đọc lại file nhiều lần. Dùng bộ nhớ trong một cách thích hợp để hỗ trợ cho file là một biện pháp kỹ thuật nhằm tăng tốc độ chương trình.

Chương này bàn về các khả năng giao tiếp dữ liệu qua các file của một chương trình Turbo Pascal, tuy nhiên nó cũng là cách tiếp cận chung của các ngôn ngữ lập trình khác. Các thủ tục và hàm chuẩn được giới thiệu trong phần này đều được xây dựng trong unit System.

3. File trong Turbo Pascal

File là một kiểu dữ liệu lưu trữ ở bộ nhớ ngoài nên nó có những đặc điểm khác với các kiểu dữ liệu tổ chức bên trong bộ nhớ:

- Kiểu file không thể là thành phần của bất cứ kiểu nào.
- Không được dùng phép gán đối với các biến file.
- Chương trình không thể truy cập trực tiếp dữ liệu trên file mà phải qua 2 thao tác cơ bản là đọc (read) và ghi (write).
- Tại một thời điểm, thao tác đọc/ghi chỉ làm việc với một thành phần của file. Việc định vị đến các thành phần file được thực hiện nhờ một cơ chế lôgic, gọi là con trỏ file.
- Tham số kiểu file trong các chương trình con phải là tham biến.

Trong việc truy cập, Turbo Pascal tổ chức 3 kiểu file: file định kiểu, file văn bản và file không định kiểu.

3.1. File định kiểu

a. Khai báo và các thao tác cơ bản

File định kiểu là file gồm nhiều thành phần, các thành phần được xác định cùng một kiểu như nhau. Kiểu thành phần của file có thể là một kiểu bất kỳ (ngoại trừ kiểu file). Cú pháp khai báo một file định kiểu là:

```
TYPE
  FileType = FILE OF DataType;
```

trong đó `FileType` là file định kiểu được đặt bởi người dùng, `DataType` là một tên kiểu đã được định nghĩa (có thể là kiểu chuẩn của hệ thống hoặc một kiểu phức hợp được định nghĩa bởi người dùng), còn các từ viết bởi các chữ cái lớn là từ khóa.

Các thành phần của file được hệ thống đánh số từ 0 (gọi là số hiệu thành phần). Chúng được định vị bởi con trỏ file và được truy cập bởi các lệnh đọc hoặc ghi. Lúc mới mở file,

con trỏ file được định vị tại thành phần đầu tiên (có số hiệu 0) và cứ sau mỗi thao tác đọc hoặc ghi, con trỏ file tự động dịch tới thành phần kế tiếp. Số thành phần của file không được biết khi biên dịch và về mặt lôgic, có thể lớn tùy ý. Có thể xem file định kiểu như một mảng được lưu trữ ở bộ nhớ ngoài, trong đó số hiệu thành phần đóng vai trò giá trị chỉ số và con trỏ file đóng vai trò biến chỉ số.

↓ con trỏ file

0	1	2	3	4	5	6	7	8	9

Hình vẽ trên mô tả một file gồm 10 thành phần, đang được định vị tại thành phần thứ 3 (có số hiệu 2).

Dưới đây là một thí dụ khai báo các file định kiểu:

```
Type
  Rec = Record
          Id   : string[5];
          Cost: integer;
          Yes  : boolean;
        End;
  RecFile = file of Rec;
  IntFile = file of integer;
```

Trong thí dụ trên, `RecFile` là file định kiểu có thành phần thuộc kiểu `Rec` (kiểu bản ghi), mỗi thành phần của nó chiếm 9 byte, `IntFile` là file định kiểu có thành phần là `Integer`, mỗi thành phần của nó chiếm 2 byte. Dữ liệu được lưu trữ trên file giống như lưu trữ trong bộ nhớ. Chẳng hạn với `RecFile`, mỗi thành phần gồm 9 byte theo thứ tự sau: byte đầu là độ dài của xâu `Id`, 5 byte tiếp theo là các ký tự của xâu này, 2 byte tiếp (byte thứ 7 và byte thứ 8) là số nguyên `Cost` (trong đó byte thứ 7 là byte thấp và byte thứ 8 là byte cao), byte cuối cùng (byte thứ 9) là giá trị lôgic `Yes`. Trong đa số ứng dụng, thành phần của file định kiểu được tổ chức dưới dạng bản ghi. Cũng vì vậy, mỗi thành phần của file định kiểu còn được gọi là mỗi bản ghi của nó.

Biến file định kiểu được khai báo giống như các biến kiểu khác. Chúng có thể khai báo qua các kiểu file đã được định danh:

```
Var
  f: RecFile;
  g: IntFile;
```

hoặc trực tiếp qua cú pháp khai báo kiểu file:

```
Var
  f: file of Rec;
  g: file of integer;
```

Biến file quản lý vùng đệm mà hệ thống cung cấp khi làm việc với file, nó không chứa nội dung dữ liệu như các biến khác, vì thế không thể dùng phép gán cho các biến file cũng như truyền theo trị cho các tham số kiểu file. Để làm việc với một file, ta tiến hành theo các bước:

1. Đăng ký biến file với một tên file trên đĩa bằng lời gọi thủ tục:

```
Assign(f, name)
```

trong đó `f` là biến file, `name` là một `string`, biểu diễn tên file trên đĩa (có thể bao gồm tên ổ đĩa và đường dẫn). Sau lời gọi thủ tục này, biến file `f` sẽ đại diện cho file có tên `name` cho đến khi đăng ký lại. Tham biến `f` trong các hàm, thủ tục dưới đây đều được hiểu là biến file đã đăng ký với một tên file nào đó.

2. Mở file để chuẩn bị truy cập. Có hai trường hợp:

- Mở một file đã có bằng lời gọi thủ tục `Reset(f)`, trong đó `f` là biến file tương ứng với một file đã có. Sau lời gọi thủ tục này, ta có thể truy cập file bằng các thao tác đọc hoặc ghi. Chú ý rằng nếu file chưa có hoặc vì một lý do nào đấy không mở được, thì hệ thống sẽ báo lỗi và chương trình bị dừng.
- Mở một file để tạo mới bằng lời gọi thủ tục `Rewrite(f)`, trong đó `f` là biến file đã gắn với một tên nào đó. Sau lời gọi thủ tục này, thường là các thao tác ghi để tạo nội dung mới. Chú ý rằng, thủ tục `Rewrite` không kiểm tra file đã có hay chưa, nghĩa là nếu file đã có thì nó sẽ bị ghi đè, vì thế phải thận trọng khi gọi thủ tục này để tránh mất dữ liệu. Tốt hơn cả, trước khi gọi thủ tục này, nên tổ chức một thủ tục kiểm tra sự tồn tại của file để quyết định có ghi đè hay không.

3. Thao tác đọc được thực hiện bởi thủ tục `Read(f, x)`, thao tác ghi được thực hiện bởi thủ tục `Write(f, x)`, trong đó `x` là một biến có cùng kiểu với kiểu thành phần file `f`. Thủ tục `Read(f, x)` đọc nội dung thành phần của file `f` mà con trỏ đang định vị vào biến `x`, ngược lại, thủ tục `Write(f, x)` ghi nội dung của biến `x` lên thành phần của file `f` mà con trỏ đang định vị. Sau mỗi thao tác đọc hoặc ghi, con trỏ file tự động dịch đến thành phần kế tiếp. Thủ tục `Read` và `Write` được mở rộng tự nhiên cho một danh sách biến với ý nghĩa việc đọc hoặc ghi được lặp lại lần lượt theo danh sách này. Chẳng hạn lời gọi `Read(f, x, y)` tương đương với hai lời gọi `Read(f, x); Read(f, y)`.

4. Sau khi thao tác xong với file cần đóng nó lại bằng thủ tục `Close(f)`. Đây là việc làm cần thiết để đảm bảo an toàn dữ liệu. Đặc biệt khi ghi xong, nếu không đóng file thì những dữ liệu sau lần ghi cuối cùng có thể không được cất lên file. Một số thủ tục chỉ được phép thực hiện khi đã đóng file.

File định kiểu của Turbo Pascal là loại file truy cập trực tiếp, nghĩa là có thể truy cập đến bất cứ thành phần nào của file, không cần theo thứ tự. Turbo Pascal hỗ trợ cho người dùng thủ tục

```
Procedure Seek(var f; n: LongInt);
```

để định vị con trỏ file `f` đến thành phần có số hiệu `n` (chú ý, Pascal đánh số các thành phần bắt đầu từ 0). Ở đây, `f` là biến file tương ứng với file đang mở, `n` là một biểu thức kiểu `LongInt`. Thủ tục này được dùng trước lời gọi các thủ tục `Read` hoặc `Write` để đọc hoặc ghi vào nơi mong muốn.

Chú ý rằng, có thể định vị ra ngoài phạm vi file để ghi, khi đó file sẽ được kéo dài đến thành phần mới tạo. Tuy nhiên, nếu đọc tại vị trí ngoài phạm vi file, sẽ gặp lỗi *Disk read*

error. Thí dụ, hiện file `f` đang có 5 thành phần (đánh số từ 0 đến 4), các lệnh sau là không có lỗi:

```
Seek(f, 9);
Write(f, Rec);
```

và sẽ kéo dài `f` đến 10 thành phần, trong đó thành phần cuối cùng (số hiệu 9) có nội dung vừa ghi. Các thành phần từ số hiệu 5 đến số hiệu 8 là chưa có nội dung. Nhưng cũng với giả thiết `f` như trên, các lệnh:

```
Seek(f, 9);
Read(f, Rec);
```

sẽ gặp lỗi *Disk read error* tại lời gọi `Read`.

Turbo Pascal cũng xây dựng một số các hàm dưới đây nhằm trả lại một số thông tin về trạng thái của một file đang được truy cập. Tham biến `f` trong các hàm này được hiểu là biến file tương ứng với file đang mở.

Để biết kích thước của file, bạn dùng hàm:

```
Function FileSize(var f): LongInt
```

trả lại số thành phần của file `f`. Chú ý rằng hàm trả lại số thành phần của file chứ không phải là số byte của file, muốn lấy kích thước của file tính theo byte, bạn cần lấy giá trị của hàm nhân với kích thước của một thành phần.

Vì bất cứ file nào cũng có thể xem như file có thành phần là 1 byte nên dùng hàm trên, bạn có thể xây dựng hàm dưới đây trả về kích thước tính theo byte của một file bất kỳ, có tên trên đĩa là `name`:

```
Function SizeOfFile(name: string): LongInt;
Var f: file of byte;
Begin
  Assign(f, name); Reset(f);
  SizeOfFile := FileSize(f);
  Close(f);
End;
```

Lời gọi `Seek(f, FileSize(f))` sẽ đưa con trỏ vào cuối file `f` (sau thành phần cuối cùng của file). Đây là thao tác định vị sau khi mở file để chuẩn bị ghi bổ sung.

Ngược với bổ sung, bạn có thể xóa một số thành phần ở cuối file bằng cách dùng thủ tục:

```
Procedure Truncate(var f);
```

trong đó `f` là biến file đang mở. Thủ tục này sẽ cắt bỏ các thành phần tính từ vị trí con trỏ hiện tại đến hết file. Vị trí hiện tại của con trỏ sẽ trở thành vị trí cuối file (end of file).

Để biết hiện nay đang ở thành phần nào của file, bạn dùng hàm:

```
Function FilePos(var f): LongInt;
```

trả về số hiệu thành phần của file `f` đang được định vị.

Trong tiến trình xử lý file, nhiều khi cần kiểm tra đã hết file chưa để tránh truy cập vào nơi không chứa dữ liệu. Turbo Pascal xây dựng hàm:

```
Function Eof(var f): boolean;
```

trả về giá trị lôgic `True` nếu đang ở cuối file `f` và trả về `False` trong các trường hợp còn lại (`Eof`: End of file). Dùng hàm này, ta có thể xây dựng mô hình duyệt tuần tự các thành phần của file `f` như sau:

```
Reset(f);
while not Eof(f) do
begin
   {đọc và thao tác trên một thành phần}
   ........................
end;
```

b. Xử lý file định kiểu

File định kiểu là những file lưu hành trong nội bộ, phụ thuộc vào việc tổ chức kiểu thành phần. Chúng thường là công cụ để trao đổi dữ liệu giữa các chương trình trong cùng một hệ thống, nơi mà việc tổ chức dữ liệu được thống nhất khuôn dạng trong việc đọc và ghi. Nhờ khả năng truy cập trực tiếp, file định kiểu giống như một mảng được lưu trữ ở bộ nhớ ngoài, vì thế các thao tác trên mảng có thể chuyển một cách tự nhiên cho file định kiểu. Dưới đây là một số vấn đề thường gặp khi xử lý file định kiểu.

Tạo file định kiểu

Để tạo một file định kiểu, cần phải tổ chức kiểu thành phần cho nó. Có thể tạo file dưới nhiều hình thức: Nhập dữ liệu từ bàn phím, chuyển dữ liệu từ file khác hoặc dữ liệu được kết xuất trong chương trình. Dưới đây là một thí dụ tạo một file định kiểu mà dữ liệu được kết xuất ngẫu nhiên, nhằm nhanh chóng tạo một file có kích thước lớn để phục vụ việc thử một số chương trình minh họa.

Kiểu thành phần của file được tổ chức là kiểu bản ghi, nhằm lưu thông tin của một tài khoản gửi ngân hàng:

```
BankRec = record
            HoTen: string[20];      → họ tên chủ tài khoản
            SoTaiKhoan: integer;    → số tài khoản đăng ký
            SoTien: real;           → số tiền gửi
          end;
```

Chương trình hỏi từ bàn phím số thành phần cần tạo sau đó sẽ tạo một cách ngẫu nhiên nội dung của các thành phần này và ghi lên file (tên file được hỏi từ bàn phím). Giá trị `HoTen` được tổ hợp ngẫu nhiên từ các giá trị `Ho`, `Dem`, `Ten` định sẵn trong chương trình (bạn có thể thêm bớt theo ý muốn). Giá trị `SoTaiKhoan` được tạo ngẫu nhiên trong khoảng từ 1 đến 30000 sao cho hai thành phần khác nhau luôn đảm bảo có `SoTaiKhoan` khác nhau (vì thế cần giả thiết số thành phần của file không quá 30000). Giá trị `SoTien` được tạo ngẫu nhiên trong khoảng từ 1 triệu đến 100 triệu (không lấy số lẻ).

```
Program GenFile;
Const
  TkMax = 30000;
```

```
  TienMin = 1000000;
  TienMax = 100000000;
  SoHo = 18;
  SoDem = 16;
  SoTen = 32;
  Ho: array[1..SoHo] of string[6] =
     ('Mac', 'Duong', 'Hoang', 'Cao', 'Do', 'Dinh', 'Pham', 'Bui',
      'Nguyen', 'Tran', 'Le', 'Vu', 'Phan', 'Doan', 'Ho', 'Quach',
      'Vuong', 'Dang');
  Dem: array[1..SoDem] of string[5] =
      ('khac', 'manh', 'huy', 'thanh', 'the', 'minh', 'duy', 'van',
       'thi', 'thu', 'hong', 'bich', 'to', 'anh', 'duc', 'bao');
  Ten: array[1..SoTen] of string[6] =
      ('Dung', 'Thanh', 'Mai', 'Chung', 'Ngoc', 'Hung', 'Cuong',
       'Binh', 'Thuong', 'Hoai', 'Tung', 'Nguyet', 'Hang', 'Thuy',
       'Huong', 'An', 'Lan', 'Long', 'Kim', 'Linh', 'Truong', 'Tho',
       'Tuan', 'Ba', 'Ha', 'Son', 'Hai', 'Nga', 'Giang', 'Truc',
       'Thong', 'Nhat');
Type
  BankRec = record
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: real;
            end;
  IntArr = array[1..TkMax] of integer;
Var
  p: IntArr;
  n, m: integer;
  f: file of BankRec;
  name: string;

  procedure Init;
  {khởi tạo các giá trị ban đầu}
  var i: integer;
  begin
    Randomize;
    Writeln('========= Sinh ngau nhien file of BankRec =========');
    Write('So thanh phan (khong qua 30000): '); Readln(n);
    Write('Ghi len file: '); Readln(name);
    m := TkMax;
    for i := 1 to m do p[i] := i;
  end;

  function RandStr: string;
  {sinh ngẫu nhiên HoTen}
  var j: integer;
      ht: string;
```

```
begin
  j := 1+Random(SoHo);
  ht := Ho[j];
  j := 1+Random(SoDem);
  ht := ht+' '+Dem[j];
  j := 1+Random(SoTen);
  ht := ht+' '+Ten[j];
  RandStr := ht;
end;

function RandInt: Integer;
{sinh ngẫu nhiên SoTaiKhoan}
var j: integer;
begin
  j := 1+Random(m);
  RandInt := p[j];
  p[j] := p[m];
  Dec(m);
end;

function RandReal: Real;
{sinh ngẫu nhiên SoTien}
begin
  RandReal := TienMin+Int(Random*(TienMax-TienMin));
end;

procedure WriteFile;
{ghi dữ liệu lên file}
var i: integer;
    r: BankRec;
begin
  Assign(f, name); Rewrite(f);
  for i := 1 to n do
  begin
    with r do
    begin
      HoTen := RandStr;
      SoTaiKhoan := RandInt;
      SoTien := RandReal;
    end;
    Write(f, r);
  end;
  Close(f);
end;

Begin
  Init; WriteFile;
```

```
End.
```

Dưới đây là nội dung file BANK.DAT gồm 10 thành phần được tạo ngẫu nhiên bởi chương trình trên:

Họ tên	Số tài khoản	Số tiền gửi
Tran thi Thuy	15941	35061703
Dinh van Truong	2741	50802942
Le huy Cuong	6916	52955174
Nguyen duc Kim	19048	54934497
Doan thanh Lan	27425	49782573
Phan anh Nguyet	24246	49295776
Pham minh Thong	11871	77347024
Ho duy Hai	29856	6000716
Dang thu Mai	21688	20171410
Vu khac Binh	10789	68402739

Duyệt file định kiểu

Trong một số bài toán thống kê hay tổng hợp, cần xử lý tuần tự các thành phần của file, từ thành phần đầu cho đến hết. Mô hình dưới đây là thích hợp cho các xử lý kiểu này:

```
Reset(f);
while not Eof(f) do
begin
  {đọc và thao tác trên một thành phần}
  ........................
end;
```

Thí dụ dưới đây là chương trình hiển thị lên màn hình nội dung từng thành phần của `file of BankRec`. Mỗi thành phần được hiển thị trên một dòng gồm các mục: số thứ tự (bắt đầu từ 1), họ tên, số tài khoản và số tiền gửi. Cứ sau 20 dòng, chương trình ngừng lại chờ người dùng ấn phím Enter.

```
Program ShowFile;
Type
  BankRec = record
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: real;
            end;

Var
  f: file of BankRec;
  name: string;
  c: LongInt;
  k: byte;
  r: BankRec;

Begin
```

```
  Writeln('========= Xem noi dung file of BankRec =========');
  Write('Ten file: '); Readln(name);
  Assign(f, name); Reset(f);
  c := FileSize(f);
  Writeln('So thanh phan = ', c);
  Writeln('Kich thuoc file = ', c*SizeOf(BankRec));
  Writeln('An phim Enter de xem ... ');
  Readln;
  k := 0; c := 0;
  while not Eof(f) do
  begin
    Inc(k); Inc(c);
    Read(f, r);
    with r do
      Writeln(c:9, '. ', HoTen, ' ':21-Length(HoTen),
              SoTaiKhoan:6, SoTien:11:0);
    if k = 20 then
    begin
      Readln;
      k := 0;
    end;
  end;
  Close(f);
  Writeln('Het file! An phim Enter de ket thuc ... ');
  Readln;
End.
```

Sắp xếp file định kiểu

Về lôgic, có thể áp dụng các thuật toán sắp xếp mảng cho một file định kiểu, trong đó các thao tác truy cập thành phần mảng được thay bằng các thao tác truy cập thành phần file. Tuy nhiên tốc độ truy cập file chậm hơn rất nhiều so với truy cập mảng, điều này làm cho thời gian chờ đợi nhiều khi không thể chấp nhận được. Mặt khác, hầu như không thể chép toàn bộ file vào một mảng để có thể áp dụng các kỹ thuật sắp xếp trong (chẳng hạn quick-sort) vì kích thước một thành phần thường quá lớn. Trong những tình huống như vậy, người ta phải sử dụng các thuật toán sắp xếp ngoài hoặc dùng một phần bộ nhớ trong để hỗ trợ cho việc sắp xếp.

Một trong các thuật toán sắp xếp ngoài là kỹ thuật trộn file, sẽ được trình bày trong mục file tuần tự (file text) dưới đây. Phần này trình bày kỹ thuật dùng mảng chỉ số để thực hiện việc sắp xếp trên mảng này thay cho việc sắp xếp trên file (gọi là sắp xếp theo chỉ số).

Mảng chỉ số là một mảng gồm hai trường: Một trường lưu số hiệu thành phần của file và một trường lưu giá trị của khóa sắp xếp (trường khóa). Mặc dù một thành phần của file có thể rất lớn, nhưng một thành phần của mảng chỉ số là khá nhỏ để có thể lưu chúng ở bộ nhớ trong. Việc sắp xếp mảng này có thể tiến hành bằng những thuật toán quen thuộc,

chẳng hạn quick-sort, sau đó nhờ mảng này, ta có thể truy cập các thành phần của file theo thứ tự đã thiết lập. Để không phải sắp xếp lại mảng này mỗi khi cần truy cập file theo thứ tự, người ta lưu chúng vào một file, gọi là file chỉ số (index file). Vì giá trị trường khóa được suy từ số hiệu thành phần nên để tiết kiệm không gian lưu trữ, chỉ cần giữ lại trường số hiệu này. Như thế có thể coi file chỉ số là `file of integer`. Mỗi một file dữ liệu, có thể kèm nhiều file chỉ số khác nhau để có thể truy cập theo nhiều thứ tự khác nhau. Kích thước của các file chỉ số là khá nhỏ so với file dữ liệu, vì thế có thể nạp chúng vào bộ nhớ trong dưới dạng mảng. Việc truy cập các mảng này thay cho truy cập file làm tăng đáng kể tốc độ của chương trình.

Chương trình dưới đây sắp xếp `file of BankRec` theo chiều tăng dần của số tài khoản nhờ việc thiết lập một mảng chỉ số. Việc sắp xếp được tiến hành trên mảng chỉ số bằng thuật toán quick-sort. Cuối chương trình, các chỉ số được lưu lại dưới dạng một `file of integer` (file chỉ số) với tên trùng với tên file dữ liệu nhưng có phần mở rộng là .IDX để nhấn mạnh đây là index file. Giới hạn mảng chỉ số không quá 10000 phần tử.

```
Program SortByIndex;
Const
  max = 10000;
Type
  BankRec = record
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: real;
            end;
  IndRec = record                {kiểu chỉ số}
             SoHieu: integer;    {số hiệu thành phần}
             Khoa: integer;      {giá trị khoá}
           end;
  IndArr = array[0..max] of IndRec;
Var
  BankFile: file of BankRec;     {file dữ liệu}
  IndFile: file of integer;      {file chỉ số}
  name: string;                  {tên file}
  n: LongInt;                    {số thành phần của file}
  Idx: IndArr;                   {mảng chỉ số}

  procedure Init;
  var Rec: BankRec;
      i: integer;
  begin
    Writeln('=== Sap xep so tai khoan tang dan theo chi so ===');
    Write('Ten file: '); Readln(name);
    Assign(BankFile, name); Reset(BankFile);
    n := FileSize(BankFile);
```

```
  Writeln('So thanh phan = ', n);
  Writeln('Kich thuoc file = ', n*SizeOf(BankRec));
  If n > max then
  begin
    Write('Vuot kich thuoc mang! An Enter de ket thuc ... ');
    Readln;
    Halt;
  end;
  Writeln('Bat dau sap xep ... ');
  {tạo mảng chỉ số}
  i := 0;
  while not Eof(BankFile) do
  begin
    with Idx[i] do
    begin
      SoHieu := i;
      Read(BankFile, Rec);
      Khoa := Rec.SoTaiKhoan;
    end;
    Inc(i);
  end;
end;

procedure QuickSort;
{sáp xếp mảng chỉ số}
var imid: integer;
    t: IndRec;
    kmid: integer;

  procedure Sort(l, r: integer);
  var i, j: integer;
  begin
    imid := (LongInt(l)+r) div 2;
    kmid := Idx[imid].Khoa;
    i := l; j := r;
    repeat
      while Idx[i].Khoa < kmid do Inc(i);
      while Idx[j].Khoa > kmid do Dec(j);
      if i <= j then
      begin        {hoán chuyển Idx[i], Idx[j]}
        t := Idx[i]; Idx[i] := Idx[j]; Idx[j] := t;
        Inc(i); Dec(j);
      end;
    until i > j;
    if l < j then Sort(l, j);
    if i < r then Sort(i, r);
  end;
```

```
  begin  {QuickSort}
    Sort(0, n-1);
  end;

  procedure WriteInd;
  {ghi chỉ số ra file với tên là name+'.idx'}
  var j: integer;
  begin
    j := Pos('.', name);
    if j > 0 then name := Copy(name, 1, j-1);
    name := name+'.idx';
    Assign(IndFile, name); Rewrite(IndFile);
    for j := 0 to n-1 do Write(IndFile, Idx[j].SoHieu);
    Close(IndFile);
    Writeln('Da ghi chi so ra file ', name);
  end;

Begin
  Init; QuickSort; WriteInd;
  Write('An Enter de ket thuc ... ');
  Readln;
End.
```

Việc sắp xếp theo chỉ số không gây sự thay đổi trình tự vật lý của file dữ liệu nhưng nhờ file chỉ số người dùng vẫn truy cập được các thành phần của file dữ liệu theo thứ tự đã thiết lập (vì thế thứ tự này được gọi là thứ tự lôgic). Chẳng hạn, khi chạy chương trình trên đối với file BANK.DAT, ta nhận được file chỉ số BANK.IDX với nội dung (10 số nguyên theo thứ tự) 1, 2, 9, 6, 0, 3, 8, 5, 4, 7 ghi nhận trình tự các số hiệu thành phần của file BANK.DAT cần truy cập để được thứ tự tăng dần của số tài khoản. Theo chỉ dẫn này, thứ tự lôgic của các thành phần của file BANK.DAT như sau:

Ho tên	Số tài khoản	Số tiền gửi	Số hiệu thành phần
Dinh van Truong	2741	50802942	1
Le huy Cuong	6916	52955174	2
Vu khac Binh	10789	68402739	9
Pham minh Thong	11871	77347024	6
Tran thi Thuy	15941	35061703	0
Nguyen duc Kim	19048	54934497	3
Dang thu Mai	21688	20171410	8
Phan anh Nguyet	24246	49295776	5
Doan thanh Lan	27425	49782573	4
Ho duy Hai	29856	6000716	7

Một ứng dụng quan trọng của mảng chỉ số là xây dựng việc tìm kiếm nhanh theo giá trị của khóa sắp xếp. Thao tác tìm kiếm là một thao tác thường xuyên trong việc liên kết và

cập nhật dữ liệu, vì thế việc tăng tốc độ cho thao tác này là một yêu cầu không thể thiếu trong các chương trình quản lý, đặc biệt trên các bộ dữ liệu lớn.

Tìm kiếm nhanh theo chỉ số

Mảng chỉ số cho phép áp dụng thuật toán tìm kiếm nhị phân để xác định số hiệu thành phần của file dữ liệu có giá trị trường khóa trùng với một giá trị cho trước. Giả thiết rằng file dữ liệu (một file định kiểu) đã được sắp xếp theo chỉ số theo một trường khóa nhờ một file chỉ số tương ứng. Các bước tìm kiếm theo chỉ số như sau:

- Đầu tiên nạp file chỉ số vào mảng chỉ số (ngoài số hiệu thành phần, nhớ thêm trường ghi nhận giá trị khóa tương ứng).
- Sau đó tìm kiếm nhị phân trên mảng chỉ số để định vị đến thành phần của mảng có giá trị khóa trùng với giá trị đã cho.
- Cuối cùng, từ giá trị của trường ghi nhận số hiệu thành phần, định vị đến thành phần tương ứng của file dữ liệu.

Việc tìm kiếm trên mảng chỉ số rõ ràng là nhanh hơn rất nhiều so với việc trực tiếp phải đọc các thành phần tương ứng của file.

Chương trình dưới đây minh họa một ứng dụng của tìm kiếm theo chỉ số. Chương trình thực hiện việc ghi lại số tiền gửi của một tài khoản trong `file of BankRec`. Số tài khoản được hỏi lặp lại từ bàn phím. Việc xác định thành phần có số tài khoản cho trước được thực hiện trên mảng chỉ số nhờ thuật toán tìm kiếm nhị phân. Chương trình kết thúc khi vào một số tài khoản âm hoặc bằng 0. Giả thiết file dữ liệu đã được sắp xếp theo chiều tăng dần của số tài khoản nhờ một file chỉ số có cùng tên với phần mở rộng là .IDX. Giới hạn có không quá 10000 tài khoản.

```
Program UpdateFile;
Const
  max = 10000;
Type
  BankRec = record
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: real;
            end;
  IndRec = record
             SoHieu: integer;
             Khoa: integer;
           end;
  IndArr = array[0..max] of IndRec;
var
  BankFile: file of BankRec;
  IndFile: file of integer;
  name: string;
  n: LongInt;
```

```
Idx: IndArr;

procedure CreatInd;
{tạo mảng chỉ số}
var i, j: integer;
    Rec: BankRec;
begin
  Write('Ten file can cap nhat: '); Readln(name);
  Assign(BankFile, name); Reset(BankFile);
  n := FileSize(BankFile);
  Writeln('So thanh phan = ', n);
  Writeln('Kich thuoc file = ', n*SizeOf(BankRec));
  If n > max then
  begin
    Write('Vuot kich thuoc mang! An Enter de ket thuc ... ');
    Readln;
    Halt;
  end;
  {mở file chỉ số tương ứng}
  i := Pos('.', name);
  if i > 0 then name := Copy(name, 1, i-1);
  name := name+'.idx';
  Assign(IndFile, name); Reset(IndFile);
  for i := 0 to n-1 do
  begin
    Read(IndFile, j);
    Seek(BankFile, j);
    Read(BankFile, Rec);
    with Idx[i] do        {ghi nhận vào mảng chỉ số}
    begin
      SoHieu := j;
      Khoa := Rec.SoTaiKhoan;
    end;
  end;
  Close(IndFile);
end;

function Location(Acc: integer): integer;
{tìm số tài khoản Acc trong mảng chỉ số theo nhị phân, nếu thấy trả về số hiệu thành phần,
trái lại trả về -1}
var mid, first, last: integer;
begin
  first := 0;
  last := n-1;
  while first <= last do
  begin
    mid := (LongInt(first)+last) div 2;
```

```
      if Acc < Idx[mid].Khoa then
        last := mid-1
      else
        if Acc > Idx[mid].Khoa then
          first := mid+1
        else
        begin
          Location := Idx[mid].SoHieu;
          exit;
        end;
    end;
    Location := -1;
  end;

  procedure Update;
  {cập nhật số tiền gửi}
  var Acc, RecNum: integer;
      Rec: BankRec;
  begin
    repeat
      Write('So tai khoan? '); Readln(Acc);
      if Acc > 0 then
      begin        {xác định số hiệu thành phần có số tài khoản Acc}
        RecNum := Location(Acc);
        if RecNum = -1 then
          Writeln('Khong co so tai khoan nay!')
        else
        begin    {ghi nhận số tiền gửi}
          Seek(BankFile, RecNum);    {định vị đến thành phần cần sửa}
          Read(BankFile, Rec);       {đọc nội dung thành phần}
          with Rec do
          begin
            Writeln('Ho ten: ', HoTen);
            Writeln('So tai khoan: ', SoTaiKhoan);
            Writeln('So tien: ', SoTien:10:0);
            Write('Ghi lai so tien: ');
            Readln(SoTien);
          end;
          Seek(BankFile, RecNum);
          Write(BankFile, Rec);      {ghi lại thành phần đã sửa}
        end;
      end;
    until Acc <= 0;
    Close(BankFile);
  end;

Begin
```

```
  CreatInd; Update;
  Write('An Enter de ket thuc ... ');
  Readln;
End.
```

3.2. File văn bản

a. Khai báo và các thao tác cơ bản

File văn bản trong Turbo Pascal được hiểu theo nghĩa chuẩn, các dữ liệu được lưu trữ dưới dạng các ký tự trong bảng mã ASCII. Các ký tự được liên kết thành từng dòng, dấu hết dòng được ghi nhận bởi cặp ký tự điều khiển có mã 13 (ký tự trở về đầu dòng - CR) và ký tự có mã 10 (ký tự xuống dòng - LF). Không có quy định về chiều dài dòng cho file văn bản. Thí dụ file văn bản gồm hai dòng như dưới đây:

`ABC` → gồm 3 ký tự A, B, C viết liền nhau, cuối dòng có dấu xuống dòng

`D` → gồm 1 ký tự D, cuối dòng có dấu xuống dòng

sẽ gồm 8 byte theo thứ tự có giá trị như sau: 65 (mã của A), 66 (mã của B), 67 (mã của C), 13 (mã của CR), 10 (mã của LF), 68 (mã của D), 13 (mã của CR), 10 (mã của LF).

Turbo Pascal dùng tên chuẩn `Text` đặt cho kiểu file văn bản. Vì thế lời khai báo biến file văn bản `f` sẽ là:

```
Var
  f: Text;
```

Sự khác nhau cơ bản giữa file văn bản và file định kiểu là việc truy cập file văn bản không được quy định theo từng thành phần có cùng một kiểu xác định như trong file định kiểu. Các thành phần trong các thao tác đọc hoặc ghi trên file văn bản có thể thuộc kiểu ký tự, xâu ký tự, số, hoặc lôgic. Khi ghi những giá trị này lên file văn bản, hệ thống sẽ tự động chuyển sang dạng ký tự tương ứng. Chẳng hạn việc ghi giá trị nguyên 1785 lên file văn bản sẽ không phải là 2 byte như việc lưu trữ kiểu `integer` mà là 4 byte, mã tương ứng của các chữ số 1, 7, 8, 5. Cũng như vậy, theo chiều ngược lại, khi đọc chuỗi ký tự 1785 từ file văn bản vào một biến nguyên, dữ liệu sẽ được chuyển sang 2 byte theo cách thức của `integer`.

Cũng vì vậy, file văn bản chỉ có thể truy cập tuần tự chứ không thể truy cập trực tiếp như file định kiểu. Các thủ tục `Seek`, `Truncate` cũng như các hàm `FileSize`, `FilePos` là không được dùng cho các file văn bản. Ngoài ra cũng không thể vừa đọc vừa ghi trên file văn bản như file định kiểu.

Tuy nhiên vì file văn bản không phụ thuộc vào định nghĩa kiểu thành phần nên nó là công cụ giao tiếp chung giữa các chương trình. Bất cứ một chương trình nào cũng có thể đọc dữ liệu từ file văn bản và kết xuất dữ liệu dưới dạng văn bản. Các thiết bị vào, ra chuẩn như bàn phím, màn hình, máy in được xem như những dạng riêng của file văn bản, vì vậy các thao tác vào, ra trên file văn bản giống hệt như trên các thiết bị này. Việc soạn thảo hoặc hiệu chỉnh một file văn bản không phải tốn công lập trình như file định kiểu (mỗi một kiểu thành phần cần có chương trình tương ứng riêng) mà có thể dùng bất cứ

một hệ soạn thảo văn bản nào (là những chương trình tiện ích vốn rất phong phú hiện nay). Chẳng hạn, có thể dùng ngay hệ soạn thảo của chính Turbo Pascal để xây dựng hoặc hiệu chỉnh dữ liệu.

Các bước thao tác trên file văn bản về đại thể cũng giống như trên file định kiểu. Đầu tiên cần đăng ký biến file văn bản (`f`) với tên file trên đĩa (`name`) bằng thủ tục `Assign(f, name)`. Sau đó mở file bằng thủ tục `Reset(f)` nếu nó đã có, để chuẩn bị đọc hoặc mở file bằng thủ tục `Rewrite(f)` nếu định tạo mới, để chuẩn bị ghi. Chú ý rằng sau `Reset(f)` chỉ có thể đọc và sau `Rewrite(f)` chỉ có thể ghi. Mỗi lần đọc hoặc ghi xong con trỏ lại được định vị đến vị trí kế tiếp. Việc định vị chỉ có thể tiến hành tuần tự nhờ các thao tác đọc hoặc ghi. Việc đọc dữ liệu từ file văn bản giống như việc đọc dữ liệu từ bàn phím, nhờ các thủ tục `Read(f, var1, var2, ...)`, `Readln(f, var1, var2, ...)` trong đó `var1, var2, ...` là những biến thuộc trong các kiểu `char, string, real, integer` (chú ý không có kiểu `boolean`). Các biến ký tự được đọc theo từng đơn vị ký tự, các biến xâu ký tự được đọc theo đúng độ dài của nó hoặc gặp dấu xuống dòng, còn các biến số được đọc cho đến khi gặp dấu cách hoặc dấu xuống dòng. Sự khác nhau giữa `Read` và `Readln` là `Readln` đọc hết một dòng để định vị đến dòng kế tiếp (các dữ liệu thừa nếu có sẽ bị bỏ qua) còn `Read` thì không. Việc ghi dữ kiệu ra file văn bản cũng giống như việc đưa chúng ra màn hình nhờ các thủ tục `Write(f, exp1, exp2, ...)`, `Writeln(f, exp1, exp2, ...)` trong đó `exp1, exp2, ...` là các biểu thức thuộc các kiểu `char, string, real, integer, boolean`. Khuôn dạng đưa ra file văn bản của các biểu thức này cũng được quy định giống như khi đưa ra màn hình. Thủ tục `Writeln(f, ...)` sẽ ghi thêm cặp ký tự xuống dòng sau khi đã ghi giá trị của các biểu thức. Sau khi thao tác xong cần đóng file bằng thủ tục `Close(f)` để đảm bảo an toàn dữ liệu.

Turbo Pascal thiết kế thêm cho file văn bản thủ tục `Append(f)` để mở văn bản chuẩn bị ghi nối tiếp. File được mở bằng `Append` cần phải đã có. Sau khi mở bằng `Append` chỉ được phép ghi. Thủ tục `Append` giúp cho việc ghi nối tiếp kết quả ra file văn bản trong tình huống chương trình phải chạy qua nhiều giai đoạn.

Cũng giống như file định kiểu, hàm `Eof(f)` được dùng cho file văn bản để kiểm tra đã định vị đến cuối file chưa. Ngoài ra, Turbo Pascal còn xây dựng thêm hàm `SeekEof(f)` giống như hàm `Eof(f)`, trả về `True` nếu đang ở cuối file và `False` nếu trái lại, ngoại trừ nó nhẩy qua tất cả các dấu cách, tab và dấu xuống dòng trước khi trả về trạng thái cuối của file. Vị trí dấu xuống dòng cũng là một thông tin quan trọng trong việc xử lý file văn bản, vì thế Turbo Pascal cung cấp các hàm `Eoln(f)`, `SeekEoln(f)` trả về `True` nếu đang ở cuối dòng và `False` nếu trái lại (`Eoln`: End of line). Hàm `SeekEoln(f)` giống như hàm `Eoln(f)` ngoại trừ nó nhẩy qua tất cả các dấu cách, tab trước khi trả về trạng thái cuối dòng. Các hàm `SeekEof(f)`, `SeekEoln(f)` là có ích khi dùng để đọc các giá trị số trong file văn bản.

Mỗi biến file văn bản có một vùng đệm mặc định 128 byte để dùng cho các thủ tục đọc/ghi, vùng này là đủ lớn cho hầu hết các ứng dụng. Tuy nhiên trong một số trường hợp

các thao tác đọc/ghi được lặp lại nhiều lần, vùng đệm lớn hơn sẽ làm tốc độ truy cập nhanh hơn vì ít phải chuyển động đầu từ. Turbo Pascal xây dựng thủ tục

```
procedure SetTextBuf(var f: text; var Buf [; size: word]);
```

để xác định vùng đệm mới cho file văn bản. Ở đây, `f` là biến file văn bản, `Buf` là một biến không định kiểu xác định vùng đệm mới, tham số tùy chọn `size` xác định kích thước vùng đệm này, nếu không được dùng thì toàn bộ vùng nhớ của `Buf` sẽ được sử dụng. Vùng đệm mới có hiệu lực cho đến khi gặp lời gọi `Assign` mới.

Chú ý rằng `SetTextBuf` cần được gọi trước khi mở file để tránh việc mất dữ liệu do thay đổi vùng đệm, ngoài ra `Buf` cần là một biến toàn thể để không bị thu hồi khi đang thao tác vào/ra đối với file.

Turbo Pascal còn xây dựng thủ tục `Flush(f)` cho biến file văn bản `f` để đảm bảo đẩy tất cả các dữ liệu được ghi ở vùng đệm ra ngoài đĩa. Thủ tục này thường được dùng để đảm bảo tất cả những gì đã ghi lên file thực sự được ghi ra đĩa mà không phải đóng file.

b. Xử lý file văn bản

Việc xử lý dữ liệu trên các file văn bản khá đa dạng vì chúng có thể chuyển đổi thành nhiều kiểu dữ liệu khác nhau. Dưới đây là các tình huống thường gặp.

Xử lý theo từng dòng

Việc xử lý tuần tự theo từng dòng của văn bản là dạng thường gặp trong việc truy cập dữ liệu. Theo cách này, mỗi dòng của văn bản được đọc vào một biến `s` kiểu `string` nhờ thủ tục `Readln(f, s)`. Sau đó dòng `s` được xử lý (kết xuất, cập nhật, lưu trữ các thông tin trong `s`). Việc đọc và xử lý từng dòng được tiến hành kế tiếp cho đến khi hết file. Mô hình xử lý theo từng dòng có thể tóm tắt như sau:

```
Reset(f);
while not Eof(f) do
begin
  Readln(f, s);
  .............      {xử lý dòng s}
end;
```

Một ứng dụng của mô hình này là việc xây dựng những chương trình biến đổi văn bản theo những quy tắc nào đấy. Văn bản được biến đổi từng dòng theo quy tắc đã đề ra. Kết quả biến đổi được ghi lên một file văn bản khác. Để thực hiện việc biến đổi một dòng, một hàm `Change(s: string): string` được thiết kế, trong đó giá trị trả về của hàm là dòng nhận được từ dòng `s` sau khi biến đổi. Chương trình dưới đây đảo ngược các ký tự trong dòng của file văn bản TEST.TXT và ghi kết quả biến đổi ra file văn bản TSET.TXT theo mô hình đã nêu:

```
Program ChangeText;

  function Change(s: string): string;
```

```
  var i, n: integer;
      c: char;
  begin
    n := Length(s);
    for i := 1 to n div 2 do
    begin
      c := s[i];
      s[i] := s[n-i+1];
      s[n-i+1] := c;
    end;
    Change := s;
  end;

Var Source, Target: Text;
    s: string;

Begin
  Assign(Source, 'TEST.TXT'); Reset(Source);
  Assign(Target, 'TSET.TXT'); Rewrite(Target);
  while not Eof(Source) do
  begin
    Readln(Source, s);
    Writeln(Target, Change(s));
  end;
  Close(Source); Close(Target);
End.
```

Bạn có thể thay đổi nội dung hàm `Change` trong chương trình trên để được nhiều chương trình biến đổi văn bản khác nhau, chẳng hạn những chương trình chuyển đổi các font tiếng Việt trong các file văn bản.

Xử lý dữ liệu số

Các dữ liệu số có thể đọc từ file văn bản như đọc từ bàn phím. Các giá trị số cần được ghi cách nhau bởi ít nhất một dấu trắng hoặc dấu xuống dòng và phải theo đúng thứ tự như khi đọc. Để tránh đọc sót hoặc đọc vào nơi không chứa số liệu, người ta thường ghi thêm số các giá trị của nhóm số cần đọc vào trước những giá trị này. Chẳng hạn dãy gồm n số $a_1, a_2, ..., a_n$ thường được ghi trên file văn bản theo quy cách:

n

$a_1\ a_2\ ...\ a_n$

và lệnh đọc dãy vào mảng `a[1]`, `a[2]`, ..., `a[n]` tương ứng sẽ là:

```
Readln(f, n);
for i := 1 to n do Read(f, a[i]);
```

Một ma trận m dòng, n cột (a_{ij}) thường được ghi trên file văn bản theo quy cách:

$m\ n$

$a_{11}\ a_{12}\ ...\ a_{1n}$

a_{21} a_{22} ... a_{2n}

.....

a_{m1} a_{m2} ... a_{mn}

và lệnh đọc ma trận vào mảng `a[i, j]` tương ứng sẽ là:

```
Readln(f, m, n);
for i := 1 to m do
  for j := 1 to n do Read(f, a[i, j]);
```

Trong những trường hợp cần đọc các giá trị số cho đến hết dòng hoặc hết file (mà không biết trước số các giá trị cần đọc), người ta thường dùng các hàm `SeekEoln(f)` hoặc `SeekEof(f)` để tránh đọc sót hoặc đọc vào nơi không chứa số liệu. Thí dụ

```
n := 0;
while not SeekEof(f) do
begin
  Inc(n);
  Read(f, a[n]);
end;
```

Các kết quả kiểu số cũng có thể đưa ra file văn bản. Quy cách ghi chúng ra file văn bản giống hệt như đưa ra màn hình. Người ta thường tổ chức ghi các số liệu kết quả ra file văn bản theo những khuôn dạng giống như số liệu đầu vào để tiện trao đổi giữa các chương trình. Thí dụ dưới đây ghi ma trận (a_{ij}) m dòng, n cột ra file văn bản theo khuôn dạng đã nêu (giả thiết các phần tử của ma trận là nguyên và có không quá 3 chữ số):

```
Writeln(f, m, ' ', n);
for i := 1 to m do
begin
  for j := 1 to n do Write(f, a[i, j]:5);
  Writeln(f);
end;
```

Việc lưu trữ các số liệu trên các file văn bản rất thuận tiện cho các thao tác biên tập và hiệu chỉnh vì có thể tiến hành nhờ một hệ soạn thảo văn bản hoặc in ra giấy bất cứ lúc nào mà không phải lập các chương trình riêng.

Xử lý dữ liệu bản ghi

Các dữ liệu thuộc kiểu bản ghi mang tính tổng hợp vì gồm nhiều trường có các kiểu dữ liệu khác nhau. Thông thường kiểu bản ghi thích hợp đối với việc lưu trữ trên file định kiểu, trong đó mỗi thành phần của file là một bản ghi. Tuy nhiên chúng cũng có thể lưu trữ trên file văn bản, điều này có lợi cho việc thêm, bớt, hiệu chỉnh dữ liệu hoặc in ra giấy nhờ một hệ soạn thảo có sẵn. Việc lưu trữ trên file văn bản còn là một biện pháp trao đổi dữ liệu giữa các phần mềm vốn đã có rất nhiều cách tổ chức dữ liệu khác nhau.

Để lưu trữ trên file văn bản, mỗi bản ghi được tổ chức trên một dòng, trong đó mỗi trường chiếm một nhóm cột. Dấu hiệu ngăn cách giữa các trường được quy định bởi việc đọc dữ liệu trên file văn bản: Nếu là trường xâu ký tự thì nó được xác định bởi độ dài của xâu, còn nếu là trường số thì nó được phân biệt bởi những dấu cách. Cũng có thể quy định

một ký tự ngăn cách giữa các trường, điều này làm đơn giản khi biên tập nhưng khi đọc phải xây dựng các thủ tục tách từng trường và chuyển đổi kiểu dữ liệu.

Chẳng hạn có thể ghi mỗi bản ghi kiểu `BankRec` (xem chương trình xử lý tài khoản trong mục trước) lên một dòng của file văn bản theo quy cách: 21 vị trí đầu dành cho trường `HoTen`, tiếp theo là ít nhất một dấu cách, sau đó là trường `SoTaiKhoan`, tiếp theo là ít nhất một dấu cách, và cuối cùng là trường `SoTien`. Số các dấu cách giữa các trường có thể điều chỉnh để dữ liệu thẳng cột, giúp cho việc biên tập và hiệu chỉnh dễ dàng. Với quy cách như vậy, lệnh đọc một dòng văn bản vào biến bản ghi `Rec` kiểu `BankRec` sẽ là

```
with Rec do Readln(f, HoTen, SoTaiKhoan, SoTien);
```

và ngược lại, lệnh ghi nội dung biến bản ghi `Rec` lên một dòng văn bản theo quy cách đã nêu sẽ là:

```
with Rec do Writeln(f, HoTen, ' ':21-Length(HoTen),
                    SoTaiKhoan:6, SoTien:11:0);
```

Dưới đây là hai chương trình minh họa. Chương trình `RecToText` sẽ chuyển nội dung một `file of BankRec` sang file văn bản và chương trình `TextToRec` sẽ chuyển theo chiều ngược lại.

```
Program RecToText;
{chuyển file of BankRec sang file văn bản}
Type
  BankRec = record
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: real;
            end;
Var
  BankFile: file of BankRec;
  TextFile: text;
  BankName, TextName: string;
  Rec: BankRec;

Begin
  Write('Doc File of BankRec: '); Readln(BankName);
  Assign(BankFile, BankName); Reset(BankFile);
  Write('Ghi File Text: '); Readln(TextName);
  Assign(TextFile, TextName); Rewrite(TextFile);
  while not Eof(BankFile) do
  begin
    Read(BankFile, Rec);
    with Rec do
      Writeln(TextFile, HoTen, ' ': 21-Length(HoTen),
              SoTaiKhoan:6, SoTien:11:0);
  end;
  Close(BankFile); Close(TextFile);
```

```
End.

Program TextToRec;
{chuyển file văn bản sang file of BankRec}
Type
  BankRec = record
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: real;
            end;
Var
  BankFile: file of BankRec;
  TextFile: text;
  BankName, TextName: string;
  Rec: BankRec;

Begin
  Write('Doc File Text: '); Readln(TextName);
  Assign(TextFile, TextName); Reset(TextFile);
  Write('Ghi File of BankRec: '); Readln(BankName);
  Assign(BankFile, BankName); Rewrite(BankFile);
  while not Eof(TextFile) do
  begin
    with Rec do
      Readln(TextFile, Hoten, SoTaiKhoan, SoTien);
    Write(BankFile, Rec);
  end;
  Close(BankFile); Close(TextFile);
End.
```

Bạn cũng có thể viết một chương trình tương tự như `GenFile` để sinh ngẫu nhiên một file văn bản chứa thông tin các tài khoản.

c. Kỹ thuật trộn file

Trong một số tình huống file cần sắp xếp là rất lớn, không thể đủ nạp vào bộ nhớ trong ngay cả với mảng chỉ số. Những trường hợp như vậy, người ta phải dùng bộ nhớ ngoài để sắp xếp (gọi là sắp xếp ngoài).

Phần này thảo luận thuật toán sắp xếp nhờ trộn file (merge-sort), là một kỹ thuật thường dùng hiện nay để thực hiện việc sắp xếp ngoài đối với những file chỉ được phép truy cập tuần tự như file văn bản (dĩ nhiên nó cũng được dùng cho cả file định kiểu). Mỗi thành phần của file văn bản ở đây, được hiểu là một dòng (lưu trữ một bản ghi) theo quy cách đã thảo luận ở mục trên.

Giả thiết giá trị khóa là giá trị của một trường nào đấy của thành phần file và cần phải sắp xếp lại file này theo thứ tự tăng dần của khóa.

Ta sẽ gọi *một đoạn* của file là một dãy dài nhất các thành phần kế tiếp của nó có giá trị khóa tăng dần. Một file bất kỳ bao giờ cũng hình thành một cách tự nhiên những đoạn như vậy. Nếu file chỉ có một đoạn thì có nghĩa là nó đã được sắp.

Kỹ thuật trộn file đòi hỏi phải có hai file phụ để "nháp". Xong công việc, các file phụ này có thể xóa đi. Ký hiệu file cần sắp xếp là `f` và hai file phụ là `f1` và `f2`. Việc sắp xếp nhờ trộn file được hoàn thành sau nhiều bước sắp xếp. Mỗi bước sắp xếp gồm hai giai đoạn như sau:

Giai đoạn 1. Chép luân phiên các đoạn của `f` sang các file `f1` và `f2` (đoạn thứ nhất sang `f1`, đoạn thứ hai sang `f2`, đoạn thứ ba sang `f1`, ...). Chú ý rằng các đoạn được chép sang các file `f1` hoặc `f2` có thể được kết hợp với nhau để hình thành một đoạn dài hơn. Giai đoạn 1 được gọi là *giai đoạn chia*.

Giai đoạn 2. Tiến hành trộn hai file `f1` và `f2` và ghi kết quả trộn vào `f`. Việc trộn được thực hiện theo từng đoạn: Đoạn thứ nhất của `f1` trộn với đoạn thứ nhất của `f2` để được một đoạn của `f`, đoạn thứ hai của `f1` trộn với đoạn thứ hai của `f2` để được một đoạn tiếp theo của `f`, ... Nếu file nào hết trước thì các đoạn còn lại của file kia sẽ được chép nốt sang `f`. Việc trộn một đoạn của `f1` với một đoạn của `f2` được tiến hành như sau: So sánh hai thành phần được đọc của các file `f1` và `f2` mà ta ký hiệu tương ứng là `Rec1` và `Rec2` (chúng được bắt đầu từ các thành phần thứ nhất). Nếu khóa của `Rec1` nhỏ hơn khóa của `Rec2` thì `Rec1` được chép sang `f` và `Rec1` được chuyển đến thành phần kế tiếp của `f1`, trái lại `Rec2` được chép sang `f` và `Rec2` được chuyển đến thành phần kế tiếp của `f2`, ... Cứ như vậy cho đến khi một đoạn của một file (`f1` hoặc `f2`) được chép hết, khi đó những thành phần còn lại của đoạn của file kia sẽ được chép nốt để được một đoạn của `f`. Giai đoạn 2 được gọi là *giai đoạn trộn*.

Rõ ràng sau mỗi bước sắp xếp, số đoạn của file `f` sẽ ít đi. Quá trình sẽ kết thúc khi `f` chỉ còn một đoạn, nghĩa là nó đã được sắp.

Giả thiết thủ tục `SplitFile(f, f1, f2)` thực hiện giai đoạn chia file `f` sang các file `f1`, `f2` và thủ tục `MergeFile(f1, f2, f, NumSub)` thực hiện giai đoạn trộn các file `f1`, `f2` vào file `f` và trả về `NumSub` là số đoạn của `f` sau khi trộn. Khi đó quá trình sắp xếp `f` nhờ trộn file được thực hiện như sau:

```
Repeat
  SplitFile(f, f1, f2);
  MergeFile(f1, f2, f, NumSub);
until NumSub = 1;
```

Chương trình dưới đây thực hiện việc sắp xếp nhờ trộn file đối với file văn bản lưu trữ các tài khoản có cấu trúc dòng như đã mô tả trong phần trước. Các thành phần của file được sắp xếp tăng dần theo số tài khoản. Các chú thích được ghi trong chương trình.

```
Program Sort;
{sắp xếp file tuần tự nhờ trộn file}
Type
  BankRec = record
```

```
            HoTen: string[20];
            SoTaiKhoan: integer;
            SoTien: real;
          end;
var
  f: text;
  name: string;

  procedure ReadRec(var FromFile: text; var Rec: BankRec);
  {đọc một thành phần của FromFile vào Rec}
  begin
    with Rec do Readln(FromFile, HoTen, SoTaiKhoan, SoTien);
  end;

  procedure WriteRec(var ToFile: text; var Rec: BankRec);
  {ghi Rec lên một thành phần của ToFile}
  begin
    with Rec do Writeln(ToFile, HoTen, ' ':21- Length(HoTen),
                        SoTaiKhoan:6, SoTien:11:0);
  end;

  procedure CopyRec(var FromFile, ToFile: text; var Rec: BankRec;
                    var Remain, EndSub: boolean);
  {chép một thành phần của FromFile sang ToFile, thành phần được chép là Rec. Sau khi chép xong thành
  phần kế tiếp được trả về Rec, trạng thái còn file được trả về Remain, trạng thái hết đoạn được trả về
  EndSub}
  var temp: integer;
  begin
    WriteRec(ToFile, Rec);
    temp := Rec.SoTaiKhoan;
    EndSub := Eof(FromFile);
    Remain := not EndSub;
    if Remain then
    begin
      ReadRec(FromFile, Rec);
      EndSub := Rec.SoTaiKhoan < temp;
    end;
  end;

  procedure CopySubFile(var FromFile, ToFile: text; var Rec: BankRec;
                        var Remain: boolean);
  {chép một đoạn của FromFile sang ToFile, bắt đầu từ Rec. Sau khi chép xong thành phần đầu của đoạn
  kế tiếp được trả về Rec, trạng thái còn file được trả về Remain}
  var EndSub: boolean;
  begin
    EndSub := false;
    while not EndSub do
      CopyRec(FromFile, ToFile, Rec, Remain, EndSub);
```

```
end;

procedure SplitFile(var FromFile, ToFile1, ToFile2: text);
{chia FromFile sang ToFile1 và ToFile2}
var FileNum: byte;
    Remain: boolean;
    Rec: BankRec;
begin
  Rewrite(ToFile1); Rewrite(ToFile2); Reset(FromFile);
  ReadRec(FromFile, Rec);
  FileNum := 1;
  Remain := true;
  while Remain do
  begin {chép luân phiên các đoạn của FromFile sang ToFile1, ToFile2 cho đến hết FromFile}
    if FileNum = 1 then
      {chép một đoạn của FromFile sang ToFile1}
      CopySubFile(FromFile, ToFile1, Rec, Remain)
    else
      {chép một đoạn của FromFile sang ToFile2}
      CopySubFile(FromFile, ToFile2, Rec, Remain);
    FileNum := 3-FileNum;
  end;
  Close(ToFile1); Close(ToFile2); Close(FromFile);
end;

procedure MergeSubFile(var FromFile1, FromFile2, ToFile: text;
                       var Rec1, Rec2: BankRec;
                       var Remain1, Remain2: boolean);
{trộn một đoạn của FromFile1 với một đoạn của FromFile2 để được một đoạn của ToFile, thành phần
đầu của các đoạn tương ứng là Rec1, Rec2. Sau khi trộn xong thành phần đầu của các đoạn kế tiếp tương
ứng được trả về Rec1, Rec2, trạng thái còn FromFile1 được trả về Remain1, trạng thái còn FromFile2
được trả về Remain2}
var EndSub1, EndSub2: boolean;
begin
  EndSub1 := false; EndSub2 := false;
  while not (EndSub1 or EndSub2) do
  {chép các thành phần của FromFile1 và FromFile2 sang ToFile cho đến khi hết một đoạn}
    if Rec1.SoTaiKhoan < Rec2.SoTaiKhoan then
      {chép một thành phần của FromFile1 sang ToFile}
      CopyRec(FromFile1, ToFile, Rec1, Remain1, EndSub1)
    else
      {chép một thành phần của FromFile2 sang ToFile}
      CopyRec(FromFile2, ToFile, Rec2, Remain2, EndSub2);
  if EndSub1 then
    {chép nốt đoạn của FromFile2 sang ToFile}
    CopySubFile(FromFile2, ToFile, Rec2, Remain2)
  else
    {chép nốt đoạn của FromFile1 sang ToFile}
```

```
    CopySubFile(FromFile1, ToFile, Rec1, Remain1);
end;


procedure MergeFile(var FromFile1, FromFile2, ToFile: text;
                    var NumSub: LongInt);
{trộn FromFile1 với FromFile2 vào ToFile. Sau khi trộn xong, số đoạn của ToFile được trả về NumSub}
var Remain1, Remain2: boolean;
    Rec1, Rec2: BankRec;
begin
  Rewrite(ToFile); Reset(FromFile1); Reset(FromFile2);
  ReadRec(FromFile1, Rec1);
  ReadRec(FromFile2, Rec2);
  Remain1 := true; Remain2 := true;
  NumSub := 0;
  while Remain1 and Remain2 do
  begin {trộn các đoạn của FromFile1 và FromFile2 vào ToFile cho đến khi hết một file}
    MergeSubFile(FromFile1, FromFile2, ToFile,
                 Rec1, Rec2, Remain1, Remain2);
    Inc(NumSub);
  end;
  while Remain1 do
  begin {chép nốt các đoạn của FromFile1 sang ToFile nếu còn}
    CopySubFile(FromFile1, ToFile, Rec1, Remain1);
    Inc(NumSub);
  end;
  while Remain2 do
  begin {chép nốt các đoạn của FromFile2 sang ToFile nếu còn}
    CopySubFile(FromFile2, ToFile, Rec2, Remain2);
    Inc(NumSub);
  end;
  Close(ToFile); Close(FromFile1); Close(FromFile2);
end;

procedure MergeSort(var f: text);
{sắp xếp file f nhờ trộn file}
var NumSub: LongInt;
    f1, f2: text;
begin
  {đăng ký các file nháp}
  Assign(f1, 'Temp1');
  Assign(f2, 'Temp2');
  {bắt đầu trộn}
  repeat
    SplitFile(f, f1, f2);
    MergeFile(f1, f2, f, NumSub);
  until NumSub = 1;
```

```
    {xóa các file nháp}
    Erase(f1); Erase(f2);
  end;

Begin
  Write('Ten file can sap xep: '); Readln(name);
  Assign(f, name);
  Writeln('Bat dau sap xep ...');
  MergeSort(f);
  Write('Da sap xep xong! An phim Enter de ket thuc ... ');
  Readln;
End.
```

Chương trình trên có thể dùng để sắp xếp file định kiểu có thành phần thuộc kiểu **BankRec**, trong đó chỉ cần thay các thủ tục **ReadRec**, **WriteRec** bằng các thủ tục chuẩn **Read**, **Write**.

Để minh họa thuật toán này, bạn có thể cho chương trình trên chạy thử với một file và ghi nhận các kết quả trung gian sau mỗi lần chia, trộn. Chẳng hạn file được thử **f** (là file văn bản được chuyển từ file BANK.DAT) ban đầu có nội dung theo thứ tự (để gọn, chỉ viết các giá trị khóa của các thành phần của file, trong đó các đoạn được thể hiện bằng những nhóm gạch dưới):

```
f:     15941  2741 6916 19048 27425  24246  11871 29856  21688  10789
```

Sau lần chia thứ nhất các file **f1**, **f2** là:

```
f1:    15941 24246  21688
f2:    2741 6916 19048 27425  11871 29856  10789
```

Sau lần trộn thứ nhất file **f** là:

```
f:     2741 6916 15941 19048 24246 27425  11871 21688 29856  10789
```

Sau lần chia thứ hai các file **f1**, **f2** là:

```
f1:    2741 6916 15941 19048 24246 27425  10789
f2:    11871 21688 29856
```

Sau lần trộn thứ hai file **f** là:

```
f:     2741 6916 11871 15941 19048 21688 24246 27425 29856  10789
```

Sau lần chia thứ ba các file **f1**, **f2** là:

```
f1:    2741 6916 11871 15941 19048 21688 24246 27425 29856
f2:    10789
```

Sau lần trộn thứ ba file **f** là:

```
f:     2741 6916 10789 11871 15941 19048 21688 24246 27425 29856
```

Đến đây file **f** đã được sắp.

Trường hợp tồi nhất của merge-sort xảy ra khi các khóa có thứ tự ngược lại. Có thể thấy rằng trong trường hợp này để sắp xếp một file gồm n thành phần, cần đòi hỏi $\log_2 n$ thao tác chia và trộn và mỗi khóa trong n thành phần phải được thử trong mỗi thao tác. Vì thế trong trường hợp xấu nhất, và có thể chỉ ra rằng trong trường hợp trung bình cũng vậy, độ

phức tạp của thuật toán này là $O(n.\log_2 n)$. Tuy nhiên cũng cần thấy rằng merge-sort chậm hơn nhiều so với quick-sort vì được thực hiện ở bộ nhớ ngoài.

Lẽ tự nhiên, kỹ thuật merge-sort cũng có thể áp dụng để sắp xếp một cấu trúc tuần tự của bộ nhớ trong (chẳng hạn danh sách liên kết). Mặc dù có độ phức tạp cũng như quick-sort nhưng merge-sort kém hiệu quả về không gian hơn hẳn so với các phương pháp sắp xếp nội trú khác vì phải dùng tới ba danh sách (một danh sách được sắp xếp và hai danh sách nháp).

Trong những tình huống cần tăng tốc độ sắp xếp cho các file dữ liệu lớn mà tình trạng ban đầu của những file này không tốt (các đoạn được sắp tự nhiên quá ngắn), người ta thường dùng bộ nhớ trong hỗ trợ bằng cách chép từng đoạn (càng dài càng tốt) của file vào mảng và sắp xếp các đoạn này nhờ các phương pháp sắp xếp nội trú nhằm kéo dài các đoạn được sắp tự nhiên của file trước khi dùng merge-sort.

d. Một thí dụ ứng dụng

Vì chỉ được phép truy cập tuần tự, các file văn bản khó tránh được việc phải đọc đi đọc lại nhiều lần trong những ứng dụng nếu chúng không được sắp xếp. Điều này làm cho rất tốn thời gian. Việc các file tuần tự được sắp xếp làm giảm nhẹ rất nhiều cho các thao tác tìm kiếm. Dưới đây xét một ứng dụng của các file tuần tự đã sắp thứ tự. Đó là việc cập nhật một file chủ (master file) từ một file giao dịch (transaction file).

Giả thiết file chủ là một file văn bản, trong đó mỗi dòng là một bản ghi lưu trữ số tiền gửi của một tài khoản (theo cấu trúc đã mô tả ở những phần trước). Cứ mỗi lần có một tài khoản đến gửi hoặc rút tiền, một bản ghi lại được nối vào file giao dịch (cũng là file văn bản), dưới dạng một dòng gồm hai trường: số tài khoản và số tiền gửi (hoặc rút) tương ứng (quy ước nếu số tiền rút thì ghi số âm). Cần chú ý những đặc điểm sau của dữ liệu:

- Trong file chủ, hai dòng khác nhau phải có số tài khoản khác nhau nhưng trong file giao dịch có thể xuất hiện nhiều dòng có cùng số tài khoản.
- Một tài khoản trong file chủ có thể không có mặt trong file giao dịch nhưng các tài khoản xuất hiện trong file giao dịch nhất thiết phải có mặt trong file chủ.

Bài toán đặt ra là phải tính lại số tiền còn gửi của mỗi tài khoản trong file chủ dựa trên các số liệu mà file giao dịch cung cấp. Kết quả được chép sang một file chủ mới có cấu trúc giống như cũ.

Dĩ nhiên ở đây cần giả thiết rằng các file đủ lớn để không thể chép toàn bộ chúng vào bộ nhớ trong rồi dùng các kỹ thuật tìm kiếm nội trú để giải quyết.

Lời giải sẽ đạt hiệu quả nhất khi hai file đều được sắp xếp (chẳng hạn tăng dần) theo số tài khoản. Trong trường hợp như thế, ta có một thuật toán duyệt các file đúng một lần để cập nhật các dòng của file chủ từ các dòng của file giao dịch. Các bước của thuật toán như sau:

Bước 1. Đọc dòng đầu tiên của file chủ vào bản ghi `MasterRec` và đọc dòng đầu tiên của file giao dịch vào bản ghi `TransRec`.

Bước 2. Đặt trạng thái dừng là sai.

Chừng nào trạng thái dừng còn chưa đúng còn làm:

- Nếu các số tài khoản của `MasterRec` và `TransRec` là bằng nhau thì thực hiện:
  - Cập nhật số tiền của `MasterRec` từ số tiền của `TransRec`.
  - Nếu đã hết file giao dịch thì đặt trạng thái dừng là đúng, trái lại đọc tiếp dòng của file giao dịch vào `TransRec`.
- Nếu trái lại thì thực hiện:
  - Chép `MasterRec` sang file chủ mới.
  - Đọc tiếp dòng của file chủ vào `MasterRec`.

Bước 3. Chép `MasterRec` sang file chủ mới.

Bước 4. Sao nốt các dòng còn lại của file chủ sang file chủ mới.

Chương trình dưới đây cập nhật file chủ có tên là BANK.TXT từ file giao dịch có tên là TRANS.TXT và ghi kết quả đã cập nhật lên file chủ mới có tên là NEWBANK.TXT. Để tiện phát biểu, chương trình xây dựng các thủ tục đọc/ghi một dòng của các file sang các bản ghi tương ứng.

```
Program UpdateFile;
Type
  BankRec = record    {bản ghi file chủ}
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: real;
            end;
  AcctRec = record    {bản ghi file giao dịch}
              SoTaiKhoan: integer;
              SoTien: real;
            end;
Var
  Master, Trans, NewMaster: text;
  MasterRec: BankRec;
  TransRec: AcctRec;
  Stop: boolean;

  procedure ReadMaster(var Rec: BankRec);
  {đọc một dòng của file chủ vào bản ghi Rec}
  begin
    with Rec do
      Readln(Master, HoTen, SoTaiKhoan, SoTien);
  end;

  procedure ReadTrans(var Rec: AcctRec);
  {đọc một dòng của file giao dịch vào bản ghi Rec}
  begin
```

```
    with ARec do
      Readln(Trans, SoTaiKhoan, SoTien);
  end;

  procedure WriteNewMaster(var BRec: BankRec);
  {ghi bản ghi BRec lên một dòng của file chủ mới}
  begin
    with BRec do
      Writeln(NewMaster, Hoten, ' ': 21-Length(HoTen),
              SoTaiKhoan:6, SoTien:11:0);
  end;

Begin
  {đăng ký và mở các file}
  Assign(Master, 'BANK.TXT');
  Assign(Trans, 'TRANS.TXT');
  Assign(NewMaster,''NEWBANK.TXT');
  Reset(Master); Reset(Trans);
  Rewrite(NewMaster);
  ReadMaster(MasterRec); {đọc dòng đầu tiên của file chủ vào bản ghi MasterRec}
  ReadTrans(TransRec); {đọc dòng đầu tiên của file giao dịch vào bản ghi TransRec}
  {cập nhật các dòng của file chủ từ các dòng của file giao dịch}
  Stop := false;
  while not Stop do
    if MasterRec.SoTaiKhoan = TransRec.SoTaiKhoan then  {khóa trùng nhau}
    begin  {cập nhật bản ghi MasterRec từ bản ghi TransRec}
      MasterRec.SoTien := MasterRec.SoTien+TransRec.SoTien;
      if Eof(Trans) then Stop := true  {hết file giao dịch}
      else ReadTrans(TransRec);  {đọc dòng tiếp của file giao dịch vào TransRec}
    end else  {khóa khác nhau}
    begin
      WriteNewMaster(MasterRec);  {ghi bản ghi MasterRec sang file chủ mới}
      ReadMaster(MasterRec); {đọc dòng tiếp của file chủ vào bản ghi MasterRec}
    end;
  WriteNewMaster(MasterRec); {ghi bản ghi MasterRec sang file chủ mới}
  while not Eof(Master) do  {sao nốt các dòng của file chủ sang file chủ mới}
  begin
    ReadMaster(MasterRec);
    WriteNewMaster(MasterRec);
  end;
  {đóng các file}
  Close(Master); Close(Trans); Close(NewMaster);
  Write('Hoan tat viec cap nhat. An Enter de ket thuc ... ');
  Readln;
End.
```

Trong chương trình trên, có thể thay các file văn bản bằng các file định kiểu tương ứng, trong đó các lệnh đọc, ghi một dòng của file văn bản được thay bằng các lệnh đọc, ghi một thành phần của file định kiểu.

Dưới đây là nội dung các file được thử:

BANK.TXT

```
Dinh van Truong     2741   50802942
Le huy Cuong        6916   52955174
Vu khac Binh       10789   68402739
Pham minh Thong    11871   77347024
Tran thi Thuy      15941   35061703
Nguyen duc Kim     19048   54934497
Dang thu Mai       21688   20171410
Phan anh Nguyet    24246   49295776
Doan thanh Lan     27425   49782573
Ho duy Hai         29856    6000716
```

NEWBANK.TXT

```
Dinh van Truong     2741   50802942
Le huy Cuong        6916   87955174
Vu khac Binh       10789  103402739
Pham minh Thong    11871   77347024
Tran thi Thuy      15941   35061703
Nguyen duc Kim     19048   74934497
Dang thu Mai       21688   20171410
Phan anh Nguyet    24246   61295776
Doan thanh Lan     27425   30782573
Ho duy Hai         29856   56000716
```

TRANS.TXT

```
  6916   20000000
  6916   15000000
 10789   35000000
 19048   25000000
 19048  -10000000
 19048    5000000
 24246   12000000
 27425  -19000000
 29856   20000000
 29856   30000000
```

e. Các file thiết bị

Các thiết bị vào ra chuẩn của DOS như bàn phím, màn hình, máy in được xem như những file văn bản đặc biệt với các tên được quy định sẵn như CON (bàn phím hoặc màn hình), PRN (máy in). Vì thể có thể đọc, ghi lên các thiết bị này như những file văn bản với các tên quy định tương ứng. Điều này giúp cho người lập trình không phải viết những đoạn chương trình riêng để xử lý việc vào, ra các thiết bị khác nhau.

Chẳng hạn đoạn chương trình sau cho phép người dùng lựa chọn việc kết xuất ra màn hình, máy in hay file trên đĩa. Biến `ch` thuộc kiểu `byte` ghi nhận sự lựa chọn theo quy định: 1 - màn hình, 2 - máy in, các giá trị còn lại - file trên đĩa.

```
case ch of
  1: name := 'CON';
  2: name := 'PRN';
  else Write('Ten file: '); Readln(name);
end;
Assign(f, name); Rewrite(f);
..................
```

. . . . . .

3.3. File không định kiểu

a. Khai báo và các thao tác cơ bản

File không định kiểu là một thiết kế đặc biệt của Turbo Pascal, dành cho các truy cập trên file mà không cần quan tâm đến các kiểu dữ liệu (chẳng hạn trong các thao tác sao chép hoặc truyền dữ liệu) nhằm tăng tốc độ cho những thao tác này. Turbo Pascal dùng từ khóa `FILE` để khai báo file không định kiểu:

```
VAR
  f: FILE;
```

Chú ý rằng, trong lệnh khai báo, không có định nghĩa kiểu thành phần. Thực chất file không định kiểu giống như một file định kiểu, trong đó mỗi thành phần của nó là một đoạn các byte, do người sử dụng tự quy định độ dài mỗi khi dùng. Việc dùng file không định kiểu cũng giống như dùng file định kiểu, ngoại trừ có khác đôi chút về các lệnh mở file và các lệnh đọc, ghi.

Mở file không định kiểu

Sau khi đăng ký biến file `f` với một tên file bằng thủ tục `Assign`, file không định kiểu được mở bằng thủ tục `Reset` (đối với file đã có) hoặc bằng thủ tục `Rewrite` (đối với tạo file mới) như một file định kiểu, nhưng các thủ tục này có thêm một tham số thứ hai kiểu `word`, sau tham biến file, để xác định độ dài một thành phần (tính theo số byte):

```
Reset(var f: file [; n: word]);
Rewrite(var f: file [; n: word]);
```

Tham số thứ hai là không bắt buộc, nếu không dùng nó thì độ dài thành phần được mặc định là 128.

Tuy độ dài thành phần là do người sử dụng quy định, nhưng do mỗi thành phần là một đơn vị dịch chuyển của con trỏ file nên trong trường hợp độ dài file không chia hết cho độ dài thành phần, một số byte thừa ở cuối file sẽ bị bỏ qua. Vì thế để đảm bảo việc truy cập dữ liệu được trung thực, độ dài thành phần phải là một ước của độ dài file. Để điều này không phụ thuộc vào độ dài file (là giá trị không biết trước), người ta thường mở file với độ dài thành phần là 1.

Đọc file không định kiểu

Khác với file định kiểu, file không định kiểu được đọc theo từng khối nhờ thủ tục:

```
BlockRead(var f: file; var Buf; NumRec: word [; var NumRead: word]);
```

trong đó `f` là biến file không định kiểu, `Buf` là một biến kiểu bất kỳ, đóng vai trò vùng đệm chứa dữ liệu được đọc từ file, `NumRec` là số thành phần chỉ định cho việc đọc, còn tham biến `NumRead` nếu được dùng, sẽ ghi nhận số thành phần thực sự được đọc.

Kích thước của `Buf` phải đủ chứa dữ liệu được đọc vào. Nó được tính bằng `NumRec` nhân với độ dài của mỗi thành phần. Trong trường hợp không có gì trục trặc, `NumRead`

luôn bằng NumRec, ngoại trừ lần đọc cuối, NumRead có thể nhỏ hơn (số các thành phần còn lại). Cũng vì vậy, dấu hiệu NumRead bằng 0 thường dùng để kiểm tra việc đọc đã hoàn thành:

```
BlockRead(f, Buf, NumRec, NumRead);
while NumRead > 0 do
begin
  {xử lý khối dữ liệu được đọc}
  ....................
  BlockRead(f, Buf, NumRec, NumRead);
end;
```

Ghi file không định kiểu

Giống như đọc, việc ghi ra file không định kiểu được tiến hành theo từng khối nhờ thủ tục:

```
BlockWrite(var f: file; var Buf; NumRec: word [; var NumWrite: word]);
```

trong đó f là biến file không định kiểu, Buf là một biến kiểu bất kỳ, đóng vai trò vùng đệm chứa dữ liệu sẽ được ghi lên file, NumRec là số thành phần chỉ định cho việc ghi, còn tham biến NumWrite nếu được dùng, sẽ ghi nhận số thành phần thực sự được ghi. Nếu NumWrite nhỏ hơn NumRec thì việc ghi có lỗi.

Chương trình dưới đây sao chép một file bất kỳ sang file khác. Các file được khai báo dưới dạng không định kiểu. Độ dài mở file được chọn là 1. Vùng đệm được chọn là một mảng có kích thước 16 Kb (16384 byte). Việc sao chép được thực hiện trong một vòng lặp gồm các thao tác đọc từng khối của file nguồn rồi chép khối được đọc sang file đích. Cuối chương trình có tổng kết số byte được sao chép để kiểm tra.

```
Program CopyFile;
Var
  SourceFile, TargetFile: file;
  SourceName, TargetName: string;
  Buf: array[1..16384] of byte;
  NumRead: word;
  Count: LongInt;
Begin
  Write('Ten file nguon: '); Readln(SourceName);
  Write('Ten file dich: '); Readln(TargetName);
  Assign(SourceFile, SourceName);
  Reset(SourceFile, 1);
  Assign(TargetFile, TargetName);
  Rewrite(TargetFile, 1);
  Count := 0;
  BlocRead(SourceFile, Buf, 16384, NumRead);
  while NumRead > 0 do
  begin
    BlockWrite(TargetFile, Buf, NumRead);   {không kiểm tra ghi}
```

```
    Inc(Count, NumRead);
    BlocRead(SourceFile, Buf, 16384, NumRead);
  end;
  Close(TargetFile); Close(SourceFile);
  Writeln('So byte da sao chep: ', Count);
  Write('An Enter de ket thuc ... ');
  Readln;
End.
```

Bạn có thể thử chương trình trên với độ dài mở file lớn hơn 1 và chú ý xem điều gì có thể xảy ra.

Mặc dù việc mở file với độ dài bằng 1 dường như làm cho file được truy cập không khác gì `file of byte`, nhưng nhờ các thao tác đọc, ghi từng khối mà thời gian truy cập nhanh hơn nhiều. Cũng có thể thấy rằng việc truy cập càng nhanh nếu kích thước vùng đệm càng lớn.

File không định kiểu cũng được truy cập trực tiếp giống như file định kiểu nhờ thủ tục `Seek`. Các hàm `FilSize`, `FilePos` được giữ nguyên ý nghĩa đối với file không định kiểu.

3.4. Xử lý lỗi vào/ra

Thông thường khi gặp lỗi vào/ra với file, chương trình sẽ bị dừng. Để người dùng chủ động xử lý lỗi loại này, Turbo Pascal xây dựng hàm `IOResult`, trả về một số nguyên, cho biết trạng thái của thao tác vào/ra. Nếu giá trị trả về của hàm khác 0 thì có nghĩa việc vào/ra gặp lỗi. Trước khi dùng `IOResult` cần phải tắt chế độ kiểm tra vào/ra bằng cách dùng định hướng biên dịch `{$I-}` (ngầm định `{$I+}`). Thông thường `IOResult` dùng để kiểm tra lỗi mở file bằng `Reset` khi tên file không tìm thấy. Thí dụ dưới đây xác định sự có mặt của file trước khi truy cập:

```
  {$I-}
  Reset(f);
  {$I+}
  If IOResult <> 0 then
    Writeln('File not found')
  else     {truy cập file}
    ...............
```

Thí dụ dưới đây mở một file định kiểu để nạp thêm các bản ghi, nếu file đã có thì ghi bổ sung, nếu file chưa có thì tạo file mới:

```
{$I-}
  Reset(f);
  {$I+}
  If IOResult <> 0 then   {tạo file mới}
    Rewrite(f)
  else                    {định vị đến cuối file}
    Seek(f, FileSize(f);
```

Thí dụ dưới đây hỏi lại trước khi quyết định ghi đè lên một file đã có:

```
{$I-}
Reset(f);
{$I+}
If IOResult = 0 then
begin
  Write('File da co! Co ghi de khong (y/n)? ');
  Readln(ans);
  if ans = 'n' then exit;
end;
Rewrite(f);
```

Giá trị trả về của `IOResult` trong trường hợp có lỗi khá đa dạng, nó tùy thuộc vào lời gọi các thủ tục vào/ra trước đó. Dưới đây là một số giá trị trả về (khác 0) thường gặp của `IOResult`:

2	File not found	Không tìm thấy file
3	Path not found	Không tìm thấy đường dẫn
4	Too many open files	Mở quá nhiều file
5	File access denied	File không thể thâm nhập

3.5. Một số thao tác khác

Ngoài các thao tác trên các file dữ liệu, Turbo Pascal còn xây dựng một số thao tác trên các file theo nghĩa hệ thống. Một số các hàm, thủ tục được xây dựng trong unit Dos sẽ được giới thiệu trong chương tương ứng. Dưới đây là một số thủ tục được xây dựng trong unit System, cho phép xóa hoặc đổi tên một file bất kỳ (giống như các lệnh nội trú DEL và REN của DOS).

```
procedure Erase(var f);
```

sẽ xóa file `f` trên đĩa. Ở đây, `f` là biến file bất kỳ, đã được đăng ký với một tên file trên đĩa và phải ở trong trạng thái không được mở.

```
procedure Rename(var f; NewName: string);
```

sẽ đặt tên file mới `NewName` cho file `f`. Cũng giống như trên, `f` là biến file bất kỳ, đã được đăng ký với một tên file trên đĩa và phải ở trong trạng thái không được mở.

Chẳng hạn sau khi cập nhật lại file tài khoản BANK.TXT và ghi sang file tài khoản mới NEWBANK.TXT (xem chương trình `UpdateFile` ở phần trên), nếu không cần giữ nội dung file cũ, bạn có thể đổi NEWBANK.TXT thành BANK.TXT như sau:

```
Assign(f, 'BANK.TXT');
Erase(f);
Assign(f, 'NEWBANK.TXT');
Rename(f, 'BANK.TXT');
```

Thí dụ dưới đây xóa một số các file trong một danh sách file. `NumFile` là số các file trong danh sách, `Name[i]` là tên file thứ `i` của danh sách, `Delete[i]` là biến lôgic ghi nhận trạng thái cần xóa của file thứ `i`:

```
for i := 1 to NumFile do
if Delete[i] then
begin
  Assign(f, Name[i]);
  Erase(f);
end;
```

Nếu xóa một file không có, hoặc đổi tên file bằng một tên đã có, thì hệ thống sẽ báo lỗi và dừng chương trình. Có thể bẫy những lỗi này bằng cách dùng định hướng biên dịch `{$I-}` và gọi hàm `IOResult`. Thí dụ dưới đây xây dựng hàm `RenResult`, đổi file có tên là `OldName` sang tên mới là `NewName` và trả về một số nguyên nhằm thông báo kết quả. Hàm trả về 0 nếu không có lỗi, trả về 1 nếu không thấy file có tên `OlName`, trả về 2 nếu file có tên `NewName` đã có, các lỗi còn lại, hàm trả về 3:

```
function RenResult(OldName, NewName: string): integer;
var f: file;
begin
  Assign(f, Oldname);
  {$I-}
  Rename(f, Newname);
  {$I+}
  case IOResult of
    0: RenResult := 0;
    2: RenResult := 1;
    5: RenResult := 2;
    else
       RenResult := 3;
  end;
end;
```

3

QUẢN LÝ BỘ NHỚ

Một trong những khía cạnh của lập trình là phải chú ý đến việc dùng bộ nhớ. Khéo thu xếp bộ nhớ dùng trong chương trình, bạn có thể khai thác tối đa vùng nhớ mà hệ thống cho phép cũng như tăng tốc độ truy cập dữ liệu của chương trình. Chương này nhằm thảo luận một số cách thức mà bộ nhớ được hệ thống cung cấp cho chương trình cũng như một vài kỹ thuật truy cập chúng. Các biến, hàm, thủ tục chuẩn giới thiệu trong chương này đều được định nghĩa trong unit System.

1. Bộ nhớ trong

1.1. Mạch flip-flop

Mạch flip-flop là thiết bị điện tử có thể chứa một trong hai mức điện thế cao và thấp tương ứng với hai giá trị lôgic 1 và 0. Nội dung của mạch flip-flop gọi là bit. Tập hợp 8 mạch flip-flop tạo thành một ô nhớ bên trong máy tính, gọi là một byte. Như vậy một byte có thể biểu diễn 256 giá trị khác nhau. Dung lượng bộ nhớ của máy tính được đo bởi số lượng các ô nhớ của nó.

1.2. Segment và offset

Để quản lý các ô nhớ cần phải đánh các địa chỉ cho chúng. Vì giá trị địa chỉ không vượt quá 16 bit nên người ta chia các ô nhớ thành từng đoạn, mỗi đoạn gồm $2^{16} = 65536$ ô (64 Kb). Các đoạn được phân biệt với nhau bởi địa chỉ đoạn, gọi là *segment*. Các ô trong một đoạn được phân biệt với nhau bởi địa chỉ tương đối so với ô đầu của đoạn (được đánh bắt đầu từ 0), gọi là *offset*. Như vậy địa chỉ của một ô nhớ bao gồm hai thành phần: địa chỉ đoạn (segment) và địa chỉ tương đối (offset).

Khi cung cấp một vùng nhớ cho một đối tượng (biến, hàm, thủ tục), bao giờ hệ thống cũng cung cấp một vùng gồm các ô nhớ liên tục, bắt đầu từ một ô nào đấy. Vì thế, người ta coi địa chỉ của một vùng nhớ là địa chỉ của ô đầu tiên thuộc vùng và kích thước một vùng nhớ được cấp phát không bao giờ vượt quá kích thước một đoạn (64 Kb). Trong các

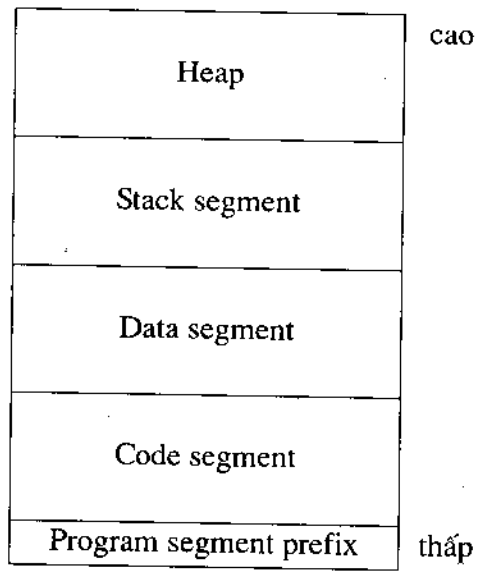
tài liệu, giá trị địa chỉ thường được viết ở hệ hexa. Trong Turbo Pascal, mỗi khi cần chỉ rõ từng thành phần, giá trị địa chỉ được viết dưới dạng segment:offset. Thí dụ $0001:$0010 mô tả một địa chỉ có segment là 0001h và offset là 0010h.

Để lấy các giá trị segment và offset, Turbo Pascal cung cấp hai hàm chuẩn `Seg(x) : word` và `Ofs(x) : word`. Hàm `Seg(x)` trả lại giá trị segment và hàm `Ofs(x)` trả lại offset của vùng nhớ cấp phát cho đối tượng `x` (`x` có thể là biến, hàm hoặc thủ tục). Thí dụ dưới đây đưa ra màn hình các giá trị địa chỉ mà hệ thống đã cấp phát:

```
Program Thidu;
Var x: integer;

  procedure Test;
  begin
    Writeln('Day la thu tuc Test');
  end;

Begin
  Writeln('Dia chi cap cho thu tuc Test la ',
          Seg(Test), ':', Ofs(Test));
  Writeln('Dia chi cap cho bien x la ',
          Seg(x), ':', Ofs(x));
  Readln;
End.
```

1.3. Cấp phát bộ nhớ

Dưới đây là bản đồ cấp phát vùng nhớ cơ sở (640Kb) cho một chương trình EXE khi được nạp (không kể phần dành cho DOS). Thứ tự trình bày theo chiều từ địa chỉ thấp đến địa chỉ cao:



Program segment prefix: Gồm 256 byte để đánh lại các địa chỉ của các biến, hàm, thủ tục trong chương trình. Địa chỉ đoạn của vùng này được lưu trong biến `PrefixSeg`.

Code segment: Bao gồm đoạn chứa mã của chương trình chính và các đoạn chứa mã của các đơn vị chương trình (unit) liên kết với chương trình chính (mỗi đoạn có thể 64 Kb vì thế tổng lại có thể rất lớn). Địa chỉ đoạn của vùng này được lưu trong thanh ghi CS và được trả về bởi hàm `CSeg: word`.

Data segment: là đoạn lưu các biến chung và các khai báo hằng, kiểu của chương trình chính. Địa chỉ đoạn của vùng này được lưu trong thanh ghi DS và được trả về bởi hàm `DSeg: word`. Việc cấp phát được thực hiện trước khi thi hành các lệnh của chương trình và không thay đổi trong suốt quá trình chạy chương trình. Nếu tổng số các biến của chương trình chính vượt quá 64 kb thì sẽ gặp lỗi *Too many variables* khi biên dịch. Để khắc phục lỗi này, người ta thường dùng một số cách:

- Chỉ giữ lại những biến chung thật cần thiết, các biến khác đưa vào các chương trình con tương ứng.
- Xin cấp phát động thay vì cấp phát tĩnh.

Stack segment: Vùng nhớ này lưu các biến riêng cũng như các mã lệnh của các chương trình con và được tháo gỡ theo quy tắc LIFO (last in first out - vào sau ra trước). Địa chỉ đoạn của nó được lưu trong thanh ghi SS và được trả về bởi hàm `SSeg: word`. Đỉnh của stack được phát triển theo một chồng hướng xuống và địa chỉ tương đối của nó được cho bởi hàm `SPtr: word`. Việc cấp phát được thực hiện mỗi khi gặp lời gọi chương trình con và được thu hồi khi chương trình con kết thúc. Lỗi *Stack overflow error* (tràn stack) sẽ được phát hiện trong quá trình chạy chương trình khi không đủ cung cấp stack cho lời gọi chương trình con. Lỗi này thường gặp khi dùng các chương trình con đệ quy, trong đó xuất hiện lời gọi lồng nhau quá dài. Vì vậy, để nâng cao hiệu quả khi dùng stack, nhất là khi có chương trình con đệ quy, bạn cần lưu ý một số điểm:

- Trong chương trình con đệ quy cố gắng không dùng các biến riêng cũng như các tham trị có kích thước lớn và phải đảm bảo có điểm dừng.
- Tăng kích thước stack đến giới hạn cho phép. Mặc dù đoạn stack có thể lên tới 64 Kb nhưng hệ thống ngầm định chỉ cung cấp 16Kb stack cho chương trình. Bạn có thể can thiệp giá trị này bằng cách dùng định hướng chương trình dịch `{$M StackSize, HeapMin, HeapMax}` để thay đổi lại giá trị `StackSize` cho phù hợp.

Heap: Đây là vùng nhớ còn lại, gọi là vùng tự do. Đỉnh của nó được phát triển theo một chồng hướng lên và có địa chỉ lưu tại biến con trỏ `HeapPtr`. Vùng này được dùng khi cấp phát động, sẽ được giới thiệu trong chương kế tiếp.

Thí dụ dưới đây minh họa việc cấp phát bộ nhớ cho một chương trình:

```
Program Demo;
Var a, k: integer;
    b: array[1..5] of integer;
```

```
  procedure WriteHexWord(w: word);
  const
    HexChars: array [0..$F] of Char =
      '0123456789ABCDEF';
  begin
    Write('$', HexChars[Hi(w) shr 4],
              HexChars[Hi(w) and $F],
              HexChars[Lo(w) shr 4],
              HexChars[Lo(w) and $F]);
  end;

  procedure Test;
  var a: ShortInt;
      b: LongInt;
  begin
    Writeln('Hien dang o trong thu tuc Test');
    Write('  Dia chi cua bien rieng a la ');
    WriteHexWord(Seg(a));
    Write(':'); WriteHexWord(Ofs(a));
    Writeln;
    Write('  Dia chi cua bien rieng b la ');
    WriteHexWord(Seg(b));
    Write(':'); WriteHexWord(Ofs(b));
    Writeln;
    Write('  Dia chi cua dinh stack la ');
    WriteHexWord(Sseg);
    Write(':'); WriteHexWord(SPtr);
    Writeln;
    Writeln('An phim Enter de ve chuong trinh chinh ...');
    Readln;
  end;

Begin
  Writeln('Cac chi tiet ky thuat cua chuong trinh:');
  Write('  Program segment prefix: ');
  WriteHexWord(PrefixSeg);
  Writeln;
  Write('  Code segment: ');
  WriteHexWord(CSeg);
  Writeln;
  Write('  Data segment: ');
  WriteHexWord(DSeg);
  Writeln;
  Write('  Stack segment: ');
  WriteHexWord(SSeg);
  Writeln;
  Write('  Dia chi cua bien chung a la ');
```

```
  WriteHexWord(Seg(a));
  Write(':'); WriteHexWord(Ofs(a));
  Writeln;
  for k := 1 to 5 do
  begin
    Write('  Dia chi cua bien chung b[', k, '] la ');
    WriteHexWord(Seg(b[k]));
    Write(':'); WriteHexWord(Ofs(b[k]));
    Writeln;
  end;
  Writeln('Truoc khi vao thu tuc Test');
  Write('  Dia chi cua dinh stack la ');
  WriteHexWord(Sseg);
  Write(':'); WriteHexWord(SPtr);
  Writeln;
  Writeln('An phim Enter de vao thu tuc Test ...');
  Readln;
  Test;
  Writeln('Sau khi ra khoi thu tuc Test');
  Write('  Dia chi cua dinh stack la ');
  WriteHexWord(Sseg);
  Write(':'); WriteHexWord(SPtr);
  Writeln;
  Writeln('An phim Enter de ket thuc ...');
  Readln;
End.
```

Chú ý khai thác các khía cạnh sau đây trong thí dụ:

- Thủ tục `WriteHexWord(w: word)` dùng để viết một giá trị `word` trong hệ hexa.
- Các biến chung được cấp phát trong vùng data.
- Các biến riêng được cấp phát trong vùng stack.
- Đỉnh stack tạo thành một chồng hướng xuống và được tháo gỡ theo quy tắc LIFO.
- Các thành phần của một mảng được cấp phát trong một vùng liên tục.

2. Các mảng Mem

Turbo Pascal cung cấp các mảng chuẩn `Mem`, `MemW`, `MemL` để truy cập trực tiếp các ô nhớ. Mảng `Mem` truy cập 1 ô (kiểu `Byte`), mảng `MemW` truy cập 2 ô (kiểu `Word`) và mảng `MemL` truy cập 4 ô (kiểu `LongInt`). Các giá trị chỉ số trong các mảng này là các địa chỉ (gồm cả segment và offset) của vùng cần truy cập và được viết dưới dạng đầy đủ cả hai thành phần như đã nêu. Chẳng hạn có thể truy cập ô nhớ có địa chỉ 0001h (segment), 0010h (offset) bằng việc lấy ra (hoặc gán lại) `Mem[$0001:$0010]` như 1 byte.

Nhờ các mảng Mem, ta có thể truy cập bất cứ vùng nhớ nào. Tuy nhiên điều này cần thận trọng vì có thể làm hỏng các thông tin hệ thống. Tốt nhất chỉ truy cập vùng nhớ nào mà đã biết rõ ý nghĩa dữ liệu chứa trong nó.

```
Program Thidu;
Var s: string;
    k: byte;
    segment, offset: word;
Begin
  segment := Seg(s); offset := Ofs(s);
  s := 'ABCDEFGHI';
  for k := 1 to 9 do
  begin
    Mem[segment:offset] := k;
    Writeln(s);
  end;
  Readln;
End.
```

Thí dụ trên truy cập vào ô đầu của vùng nhớ lưu trữ xâu ký tự `s` qua thành phần `Mem[segment:offset]` của mảng `Mem`, vì thế có thể thay đổi lại độ dài của `s` qua việc gán lại giá trị cho thành phần này.

Thí dụ dưới đây truy cập vào vùng quản lý màn hình văn bản, vùng này được gọi là Video Ram, có địa chỉ đoạn là B800 và bắt đầu từ địa chỉ tương đối 0. Vùng này gồm 4000 byte, trong đó mỗi cụm 2 byte lưu trữ một ký tự trên màn hình gồm byte thấp lưu mã của ký tự và byte cao lưu thuộc tính mầu của ký tự đó (4 bit thấp cho mầu chữ và 4 bit cao cho mầu nền). Như thế vùng Video Ram gồm 2000 ký tự (kể cả thuộc tính mầu), phân thành 80 cột và 25 dòng. Hai byte đầu tiên (có địa chỉ offset 0 và 1) biểu diễn ký tự dòng 1 cột 1, hai byte kế tiếp biểu diễn ký tự dòng 1, cột 2, ..., hai byte cuối cùng (có địa chỉ offset 3998 và 3999) biểu diễn ký tự dòng 25, cột 80. Việc truy cập vào những byte này sẽ tác động trực tiếp đến màn hình của hệ thống.

```
Program EmptyScr;
var k: integer;
Begin
  for k := 0 to 3999 do Mem[$B800:k] := 0;
End.
```

Chương trình trên đặt tất cả 4000 byte trong vùng Video Ram bằng 0, vì thế sau khi thực hiện chương trình, ta nhận được một màn hình rỗng (không có gì thể hiện trên màn hình), thậm chí sau đó dùng các lệnh thông báo (miễn là những lệnh này chỉ gửi các ký tự lên màn hình, không đụng đến thuộc tính mầu), ta cũng không nhìn thấy gì trên màn hình vì mầu chữ và mầu nền tại mọi vị trí đã được đặt bằng 0.

Chương trình dưới đây đặt lại các ô chứa thuộc tính mầu của vùng Video Ram bằng 7 (nghĩa là mầu chữ trắng, mầu nền đen), các ô còn lại (chứa ký tự) không thay đổi. Vì thế các ký tự hiện có trong vùng Video Ram lại được nhìn thấy trên màn hình:

```
Program DefaultScr;
var k: integer;
Begin
  for k := 0 to 1999 do Mem[$B800:2*k+1] := 7;
End.
```

Bằng cách tính toán vị trí các ô nhớ tương ứng, thủ tục dưới đây gửi xâu ký tự `s` lên màn hình tại cột `x`, dòng `y`:

```
procedure WriteString(x, y: integer; s: string);
var i, k: integer;
begin
  k := (y-1)*80+x-1;
  for i := 1 to length(s) do Mem[$B800:2*(i-1)+2*k] := Ord(s[i]);
end;
```

Các mảng Mem còn được dùng để tách các byte hoặc các word của một vùng dữ liệu nhằm nghiên cứu các kiểu dữ liệu khác nhau. Chẳng hạn để truy cập các byte thấp và cao của biến `x` kiểu `integer` (hay `word`), ta có thể dùng `Mem[seg(x):ofs(x)]` (byte thấp) và `Mem[seg(x):ofs(x)+1]` (byte cao) thay cho các hàm chuẩn `Lo(x)` và `Hi(x)` (`Lo(x)` trả lại byte thấp, `Hi(x)` trả lại byte cao) mà hệ thống đã xây dựng. Cũng như vậy, ta có thể truy cập word thấp của biến `x` kiểu `LongInt` bằng `MemW[seg(x):ofs(x)]` và word cao của nó bằng `MemW[seg(x):ofs(x)+2]`.

3. Biến tuyệt đối

Turbo Pascal còn thiết kế một kiểu biến đặc biệt, cho phép truy cập vào một địa chỉ xác định trước gọi là *biến tuyệt đối*. Lời khai báo biến tuyệt đối như sau:

```
VAR
  VarName: DataType ABSOLUTE Address;
```

trong đó `VarName` là tên biến tuyệt đối, `DataType` là kiểu dữ liệu, `ABSOLUTE` là từ khóa và `Address` là địa chỉ có một trong hai dạng: Hoặc có dạng segment:offset hoặc là một tên biến đã được khai báo trước.

Lệnh khai báo biến tuyệt đối không dẫn đến việc cấp phát bộ nhớ vì nó chỉ "nhúng" biến tuyệt đối vào địa chỉ đã chỉ định. Nhờ biến tuyệt đối, ta có thể truy cập vào một vùng nhớ theo kiểu dữ liệu đã khai báo. Như vậy, việc dùng biến tuyệt đối cũng giống như việc dùng các mảng Mem, nhưng về kiểu dữ liệu nó linh động hơn vì có thể khai báo kiểu dữ liệu theo ý muốn.

```
var
  s: string;
  b: array[0..255] of byte absolute s;
```

Thí dụ trên khai báo hai biến `s` và `b`, trong đó `b` là biến tuyệt đối được "nhúng" vào địa chỉ của `s`. Vì kích thước `s` và `b` là như nhau nên thực chất chúng quản lý cùng một vùng nhớ mặc dù chúng truy cập theo các kiểu dữ liệu khác nhau. Biến tuyệt đối `b` là hình ảnh đổi kiểu dữ liệu của xâu `s` sang các byte: `b[0]` là độ dài của `s`, `b[i]` (`i` = 1, 2, ...) là mã của ký tự `s[i]`. Việc thao tác trên `b` cũng chính là việc thao tác trên `s`. Chẳng hạn, có thể đổi tất cả các chữ cái in trong `s` thành các chữ cái thường tương ứng (các ký tự khác giữ nguyên) bằng cách chỉ biến đổi trên `b` (giả thiết `i` là biến nguyên đã khai báo):

```
for i := 1 to b[0] do
  if b[i] in [65..90] then b[i] := b[i]+32;
```

Khai báo dưới đây, "nhúng" mảng `Scr[i, j]` kiểu `word` ($1 \le i \le 25$, $1 \le j \le 80$) vào vùng Video Ram. Khi đó `Scr[i, j]` chính là nội dung dòng `i`, cột `j` của màn hình, trong đó byte thấp là ký tự, byte cao là thuộc tính mầu:

```
var
  Scr: array[1..25, 1..80] of word absolute $B800:0;
```

Với khai báo này, ta có thể gửi ký tự `c` lên màn hình tại dòng `i`, cột `j` với mầu chữ `t` và mầu nền `b` bằng phép gán:

```
      Scr[i, j] := Ord(c)+256*(t+16*b);
```

Một địa chỉ khác có ích cho việc tính thời gian thực hiện một đoạn chương trình. Đó là 4 byte dưới dạng một `LongInt` bắt đầu từ địa chỉ đoạn 0 và địa chỉ tương đối 46C. Nội dung của 4 byte này ghi nhận thời điểm hiện tại của đồng hồ máy. Bạn có thể lấy giá trị này từ một biến `Tcurr` (kiểu `LongInt`) "nhúng" vào địa chỉ đã nêu. Bằng cách cất thời điểm ban đầu vào biến `Tbegin` (kiểu `LongInt`) bạn tính được khoảng thời gian thực hiện đoạn chương trình là hiệu số `Tcurr-Tbegin` (tính theo nhịp đồng hồ). Để đổi ra giây (gần đúng), bạn lấy hiệu số này chia cho 18.2. Dưới đây là mô hình tính khoảng thời gian thực hiện một đoạn chương trình:

```
Var Tbegin: LongInt;
    Tcurr: LongInt absolute 0:$46C;
Begin
  ....................
  Tbegin := Tcurr;    {ghi nhận thời điểm bắt đầu}
  {thực hiện đoạn chương trình}
  ....................
  Writeln('Thoi gian thuc hien: ',
          (Tcurr-Tbegin)/18.2:1:2, '(sec)'); {đổi ra giây, lấy 2 số lẻ}
  ....................
End.
```

Trong thí dụ trên có thể thay biến `Tcurr` bằng thành phần `MemL[0:$46C]` của mảng `MemL`

4. Một số hàm và thủ tục liên quan

Turbo Pascal xây dựng một số hàm và thủ tục chuẩn, liên quan đến việc xử lý bộ nhớ. Dùng các hàm và thủ tục này, bạn có thể phát triển thành những thư viện có ích.

4.1. Hàm `SizeOf(x)` : `word`, trả lại kích thước của `x` (tính theo byte), `x` có thể là tên biến hay tên kiểu. Hàm thường được dùng để xác định kích thước vùng nhớ trước khi truy cập hay cấp phát, giá trị của nó được tính dựa trên cấu trúc của kiểu đã khai báo. Thí dụ, nếu `x` là `integer` thì giá trị của hàm là 2, nếu `x` là `string` thì giá trị của hàm là 256, nếu `x` là mảng 2 chiều với các thành phần kiểu `real` thì giá trị của hàm được tính theo công thức 6×m×n, trong đó m và n tương ứng là số giá trị của các miền chỉ số.

4.2. Tham biến không định kiểu

Turbo Pascal cho phép xây dựng các chương trình con với những tham biến không định kiểu trong các thao tác truy cập vùng nhớ mà không quan tâm đến kiểu dữ liệu (chẳng hạn khởi tạo giá trị, sao chép, hoặc hoán chuyển dữ liệu). Điều này làm tăng tốc độ truy cập dữ liệu của chương trình. Các tham biến không kiểu trong chương trình con được khai báo sau từ khóa VAR và không có tên kiểu dữ liệu kèm theo.

Vì các tham biến không định kiểu chỉ xác định địa chỉ khi được truyền mà không có thông tin về kích thước vùng nhớ, nên các tham biến này cần được dùng kèm theo các tham số xác định những kích thước tương ứng.

Dưới đây là một số thủ tục chuẩn của Turbo Pascal làm việc trên các vùng nhớ, trong đó có các tham biến không định kiểu để xác định địa chỉ của các vùng này. Một tham trị kiểu `word` được dùng kèm để xác định kích thước.

4.3. Thủ tục `FillChar(var x; size: word; value: byte)`: Điền đầy một vùng nhớ bắt đầu từ ô đầu tiên cấp cho `x`, số ô được điền là `size`, giá trị điền cho mọi ô là `value`. Thí dụ, với `x` là một biến `string`, lời gọi `FillChar(x, SizeOf(x), 65)` sẽ tạo ra xâu `x` gồm 65 chữ cái A. Thông thường giá trị `SizeOf(x)` sẽ được truyền cho tham trị `size`. Khi đó thủ tục `FillChar` được dùng để khởi gán giá trị ban đầu cho mọi loại biến. Chẳng hạn lời gọi `FillChar(x, SizeOf(x), 0)` sẽ khởi gán tất cả các byte của `x` bằng 0. Trong trường hợp này, nếu `x` là xâu thì `x` sẽ được khởi gán rỗng, `x` là mảng số thì mọi thành phần của nó được khởi gán 0, nếu `x` là mảng lôgic thì mọi thành phần của nó sẽ được khởi gán `False`. Rõ ràng việc gán bằng `FillChar` như vậy sẽ hiệu quả hơn nhiều so với việc phải gán từng thành phần của `x`, nhất là khi `x` có kích thước lớn. Ngoài ra, để tiện cho người dùng, thủ tục `FillChar` còn cho phép tham trị `value` có thể thuộc kiểu `char` hoặc kiểu `boolean`. Lúc đó, thủ tục sẽ đổi giá trị này một cách tương thích sang giá trị byte tương ứng. Thí dụ lời gọi `FillChar(x, SizeOf(x), False)` sẽ được chuyển sang lời gọi `FillChar(x, SizeOf(x), 0)`, còn lời gọi `FillChar(x, SizeOf(x), 'A')` sẽ được chuyển sang lời gọi `FillChar(x, SizeOf(x), 65)`.

Chương trình làm rỗng màn hình ở trên (xem 2) có thể viết lại bằng việc dùng thủ tục `FillChar` thay cho việc gán từng byte:

```
Program EmptyScr;
var b: byte absolute $B800:0;
Begin
  FillChar(b, 4000, 0);
End.
```

4.4. Thủ tục `Move(var x, y; size: word)`: Sao chép dữ liệu thuộc vùng nhớ có địa chỉ của `x` sang vùng nhớ có địa chỉ của `y`, số lượng các ô được sao chép cho bởi tham trị `size`. Sau lời gọi thủ tục này, `size` ô đầu tiên của `y` sẽ có nội dung giống hệt `size` ô đầu

tiên của x. Cẩn thận trọng khi dùng thủ tục này vì việc ghi chồng dữ liệu không được kiểm tra. Một cách an toàn là dùng SizeOf để xác định size.

Có thể coi thủ tục Move là khái quát hóa của phép gán giá trị giữa hai biến, trong đó không quan tâm đến kiểu dữ liệu. Dùng Move, ta có thể gán những giá trị khác kiểu cho nhau (thực chất là tương thích kiểu). Chẳng hạn một mảng gồm 4 ký tự có thể xem là tương thích kiểu với một LongInt, chúng có thể gán giá trị cho nhau qua thủ tục Move như trong thí dụ sau:

```
Var
  a: array[1..4] of char;
  b: LongInt;
  i: integer;
Begin
  for i := 1 to 4 do a[i] := Chr(i+64);     {a = ABCD}
  Move(a, b, SizeOf(a));                    {sao chép a sang b}
  Writeln('b = ', b);                       {b = 1145258561}
  Readln;
End.
```

Trong nhiều thuật toán, cần thực hiện việc đổi giá trị giữa hai biến. Trước đây, thủ tục này được thực hiện bởi các phép gán, vì thế các thủ tục đổi giá trị chỉ được xây dựng riêng cho từng kiểu dữ liệu, không thể lấy thủ tục đổi giá trị kiểu này dùng cho việc đổi giá trị kiểu khác mặc dù bản chất của việc đổi là như nhau. Dùng thủ tục Move thay cho phép gán, ta có thể xây dựng thủ tục đổi giá trị cho mọi kiểu biến:

```
procedure SwapVal(var x, y; size: word);
var
  temp: array[1..10000] of byte;
begin
  Move(x, temp, size);      {thay cho temp := x}
  Move(y, x, size);         {thay cho x := y}
  Move(temp, y, size);      {thay cho y := temp}
end;
```

Lời gọi SwapVal(x, y, SizeOf(x)) sẽ đổi giá trị của hai biến x và y cho nhau, không phụ thuộc vào kiểu của biến. Thủ tục SwapVal được xây dựng với các tham biến x và y không định kiểu để có thể dùng cho bất cứ kiểu nào, tham trị size trong thủ tục là cần thiết, nó xác định số ô được đổi. Với những biến cụ thể, size sẽ nhận giá trị từ hàm SizeOf. Trong thủ tục, biến trung gian temp được dùng để lưu giá trị khi hoán chuyển, vì thế nó có thể là bất cứ kiểu gì miễn là có kích thước không dưới kích thước của x và y. Việc khai báo temp là một mảng 10000 byte có nghĩa thủ tục chỉ được áp dụng cho các biến có kích thước không quá 10000. Hạn chế này sẽ được khắc phục bằng cách cấp phát động cho temp mỗi khi gọi tới thủ tục (xem chương kế tiếp).

Thí dụ dưới đây xây dựng hai thủ tục: Cất màn hình vào một biến và khôi phục màn hình từ biến đã cất. Trong các thủ tục này, đều dùng đến thủ tục Move.

```
procedure GetScr(var x);
{cất màn hình vào biến x}
```

```
var b: byte of $B800:0;
begin
  Move(b, x, 4000);
end;

procedure PutScr(var x);
{khôi phục màn hình từ biến x}
var b: byte of $B800:0;
begin
  Move(x, b, 4000);
end;
```

Biến được truyền cho `x` trong các thủ tục trên là không định kiểu, nó có thể là bất cứ kiểu gì miễn là đủ 4000 byte, chẳng hạn `array[0..3999] of byte`, hoặc `array[1..25, 1..80] of word`.

Dùng các thủ tục trên, có thể xây dựng các thủ tục cất màn hình hiện có lên một file và ngược lại, thủ tục nạp màn hình đã được cất trên file vào máy. File ở đây gồm 4000 byte, được tổ chức giống như vùng Video Ram.

```
procedure SaveScr(name: string);
{cất màn hình lên file có tên name}
var f: file;
    x: array[1..4000] of byte;
begin
  GetScr(x);
  Assign(f, name); Rewrite(f, 4000);
  BlockWrite(f, x, 1);
  Close(f);
end;

procedure LoadScr(name: string);
{nạp màn hình từ file có tên name}
var f: file;
    x: array[1..4000] of byte;
begin
  Assign(f, name); Reset(f, 4000);
  BlockRead(f, x, 1);
  Close(f);
  PutScr(x);
end;
```

Các hàm chuẩn dưới đây được Turbo Pascal xây dựng để thao tác trên các byte của các kiểu dữ liệu chuẩn 2 byte như `integer` hoặc `word`:

4.5. Hàm `Lo(x): byte`, trả lại giá trị byte thấp của `x`, trong đó `x` là biểu thức kiểu `integer` hay `word`. Nếu `x` là biến thì `Lo(x)` trùng với `Mem[seg(x):ofs(x)]`.

4.6. Hàm `Hi(x)` : `byte`, trả lại giá trị byte cao của `x`, trong đó `x` là biểu thức kiểu `integer` hay `word`. Nếu `x` là biến thì `Hi(x)` trùng với `Mem[seg(x):ofs(x)+1]`.

4.7. Hàm `Swap(x)` : `word`, trả lại giá trị nhận được từ `x` bằng cách hoán chuyển các byte thấp và cao của `x`, trong đó `x` là biểu thức kiểu `integer` hay `word`.

```
Program Thidu;
Var
  x, y: word;
Begin
  Write('x = '); Readln(x);
  Writeln(x, ' = ', Lo(x), '+256*', Hi(x));
  y := Swap(x);
  Writeln('Swap(', x, ') = ', y, ' = ', Lo(y), '+256*', Hi(y));
  Readln;
End.
```

4

KIỂU CON TRỎ

Chương này giới thiệu kiểu con trỏ, cho phép thao tác trên các địa chỉ của bộ nhớ. Các hàm và thủ tục chuẩn được giới thiệu trong chương này đều thuộc unit System.

1. Định nghĩa và khai báo

Kiểu con trỏ là một kiểu dữ liệu đặc biệt, dùng để biểu diễn những giá trị địa chỉ. Như thế biến con trỏ là những biến dùng để lưu những giá trị địa chỉ của bộ nhớ. Nhờ các biến con trỏ, ta có thể thao tác trên các giá trị địa chỉ cũng như truy cập dữ liệu ở các vùng nhớ một cách linh hoạt. Ngoài ra, biến con trỏ còn là công cụ để xin cấp phát trên vùng Heap (cấp phát động) và xây dựng kiểu dữ liệu động, cho phép chương trình có thể khai tác tối đa và hiệu quả bộ nhớ.

Vì giá trị địa chỉ bao gồm địa chỉ offset (1 word) và địa chỉ segment (1 word) nên mỗi biến con trỏ cần cung cấp 2 word (4 byte) để lưu địa chỉ: word thấp lưu địa chỉ offset và word cao lưu địa chỉ segment.

Turbo Pascal xây dựng hai kiểu con trỏ: Con trỏ định kiểu và con trỏ không định kiểu.

1.1. Con trỏ định kiểu

Trong định nghĩa, con trỏ định kiểu cần xác định kiểu dữ liệu mà nó trỏ đến. Turbo Pascal dùng ký hiệu ^ trước tên kiểu dữ liệu để khai báo con trỏ định kiểu:

```
Type
  PtrType = ^DataType;
```

Trong khai báo trên, `PtrType` là tên kiểu con trỏ được định nghĩa, còn `DataType` là tên kiểu dữ liệu (có thể là các kiểu chuẩn hoặc kiểu có cấu trúc bất kỳ) xác định kiểu dữ liệu được trỏ. Kiểu con trỏ, cũng giống như mọi kiểu khác, có thể là thành phần của bất cứ một kiểu có cấu trúc nào (như mảng hoặc bản ghi).

Chú ý rằng vì kiểu con trỏ chỉ lưu địa chỉ nên nó có thể khai báo trước kiểu dữ liệu được trỏ. Chẳng hạn, có quyền khai báo:

```
Type
```

```
  PtrType = ^DataType;
  DataType = record
                .......
             end;
```

Biến kiểu con trỏ được khai báo theo quy cách cũng giống như các biến kiểu khác:

```
Var
  PtrVar = PtrType;      (qua tên kiểu con trỏ)
```

hoặc

```
Var
  PtrVar = ^DataType;  (qua tên kiểu dữ liệu được trỏ)
```

Thí dụ

```
Type
  Rec = record
          Id: string[5];
          Cost: integer;
        end;
Var
  p: ^integer;
  q: ^string;
  r: ^Rec;
```

Trong thí dụ trên, các biến `p`, `q`, `r` là những biến con trỏ trỏ tới những kiểu dữ liệu khác nhau, tuy nhiên chú ý rằng, do chỉ lưu địa chỉ nên chúng đều có kích thước 4 byte, không phụ thuộc vào kích thước của kiểu dữ liệu mà chúng trỏ tới.

Mặc dù về bản chất, giá trị của các biến con trỏ đều là địa chỉ, nhưng Turbo Pascal chỉ coi là tương thích trong các phép gán hoặc truyền giá trị cho tham số đối với những con trỏ định kiểu cùng trỏ tới một kiểu dữ liệu. Chẳng hạn, trong thí dụ trên, các biến con trỏ `p`, `q`, `r` là không tương thích.

1.2. Con trỏ không định kiểu

Con trỏ không định kiểu là con trỏ không quan tâm đến kiểu dữ liệu mà nó đang trỏ đến. Turbo Pascal dùng tên chuẩn `Pointer` để khai báo kiểu con trỏ này:

```
Var
  p: Pointer;
```

Con trỏ không định kiểu là tương thích với mọi kiểu con trỏ khác trong các phép gán hoặc truyền dữ liệu cho tham số. Con trỏ không định kiểu được dùng trong các trường hợp chỉ quan tâm đến địa chỉ mà không quan tâm đến kiểu dữ liệu.

2. Các thao tác trên con trỏ

2.1. Phép gán giá trị cho con trỏ

Một giá trị địa chỉ có thể gán cho con trỏ bằng những cách thức sau đây:

Cách 1. Dùng toán tử `@`: `p := @x`

trong đó `p` là một biến con trỏ bất kỳ, `x` là tên biến hoặc hàm hay thủ tục. Toán tử `@` trả lại địa chỉ (kiểu `Pointer`) của toán hạng mà nó tác động. Lệnh `p := @x` gán địa chỉ của `x` cho `p`, nghĩa là `p` giữ địa chỉ của `x`. Trong trường hợp này người ta thường nói `p` đang trỏ tới `x`.

Cách 2. Dùng hàm `Addr`: `p := Addr(x)`

trong đó `p` là một biến con trỏ bất kỳ, `x` là tên biến hoặc hàm hay thủ tục. Hàm `Addr(x)` trả lại địa chỉ (kiểu `Pointer`) của `x`. Như thế hai lệnh gán `p := @x` và `p := Addr(x)` là tương đương.

Cách 3. Dùng hàm `Ptr`: `p := Ptr(segment, offset)`

trong đó `p` là một biến con trỏ bất kỳ, `segment`, `offset` là các giá trị thuộc kiểu `word`. Hàm `Ptr(segment, offset)` trả lại địa chỉ (kiểu `Pointer`) có các thành phần đoạn và tương đối được xác định bởi `segment` và `offset`.

Một giá trị đặc biệt được xây dựng cho con trỏ là `Nil` (giá trị rỗng). Lệnh gán `p := Nil` nhằm xác định con trỏ `p` chưa trỏ đến cái gì cả. Đây là một thao tác cần thiết khởi gán cho `p` khi chưa dùng tới, nhằm đảm bảo an toàn dữ liệu.

2.2. Phép gán giữa hai con trỏ

Hai biến con trỏ có thể gán giá trị cho nhau trong các trường hợp tương thích. Khi đó chúng cùng trỏ tới một địa chỉ. Chú ý rằng hai con trỏ định kiểu, trỏ tới các kiểu dữ liệu khác nhau, là không tương thích, tuy nhiên chúng lại tương thích với một con trỏ không định kiểu (`pointer`).

Thí dụ với khai báo:

```
Var
  p: ^integer;
  q: ^string;
  r: pointer;
```

phép gán `p := q` là không tương thích, tuy nhiên các phép gán `r := q; p := r` lại là tương thích.

2.3. Phép so sánh hai con trỏ

Hai giá trị con trỏ có thể so sánh bằng nhau (=) hay khác nhau (<>) miễn là chúng tương thích. Giá trị trả lại của phép so sánh bao giờ cũng thuộc kiểu lôgic. Chú ý rằng không có phép so sánh lớn hơn (>) hay nhỏ hơn (<) đối với những giá trị con trỏ.

```
Var
  x: string;
  p: ^byte;
  q: ^integer;
  r: pointer;
Begin
  p := @x;
  q := @x;
  r := @x;
```

```
  Writeln(p = r); ← được phép, có giá trị True
  Writeln(p = q); ← không được phép vì không tương thích
  Readln;
End.
```

Trong thí dụ trên, cả ba con trỏ p, q, r đều trỏ tới x, không phụ thuộc kiểu của x cũng như kiểu mà p, q trỏ đến. Biểu thức p = r là được phép và luôn cho giá trị True trong khi đó, biểu thức p = q là không được phép vì lỗi không tương thích, mặc dù lúc này p và q cùng trỏ tới x.

2.4. Truy cập dữ liệu

Để truy cập dữ liệu mà con trỏ p đang trỏ, Turbo Pascal dùng ký hiệu p^. Ký hiệu này đóng vai trò như một biến, có nội dung là nội dung của vùng dữ liệu mà p đang trỏ. Dùng p^, bạn có thể thay cho các mảng Mem và các biến tuyệt đối trong việc truy cập bộ nhớ. Khi dùng p^ cần chú ý phân biệt hai trường hợp:

Trường hợp p là *con trỏ định kiểu*. Khi đó p^ là biến định kiểu có địa chỉ được p lưu giữ. Kiểu của p^ được xác định là kiểu mà p trỏ tới. Mọi thao tác trên biến định kiểu thông thường đều được áp dụng cho p^. Tuy nhiên p^ là một biến định kiểu đặc biệt, nó có thể thay đổi địa chỉ trong quá trình hoạt động bằng cách gán giá trị địa chỉ tương ứng cho p, vì thế p^ có thể truy cập vào bất cứ vùng nhớ nào theo kiểu dữ liệu của nó. Có thể xem p^ như một biến tuyệt đối có khả năng thay đổi địa chỉ nhờ phép gán.

Thí dụ:

```
Var
  s: string;
  p: ^byte;
```

Với khai báo trên, khi thực hiện lệnh gán p := @s, ta được p^ là byte đầu tiên trong vùng nhớ lưu trữ s, vì thế nó là độ dài của s. Lệnh gán p^ := 23 lúc này sẽ xác định lại độ dài của s là 23. Khi gán p := Ptr($B800, 5), ta được p^ là byte thứ 6 trong vùng Video Ram, vì thế nó là mầu của vị trí dòng 1, cột 3 trên màn hình. Lệnh gán p^ := 23 lúc này (23 được viết dưới hệ hexa là $17) sẽ đặt vị trí tương ứng có mầu chữ (4 bit thấp) là 7 (trắng) và mầu nền (4 bit cao) là 1 (xanh lam).

Khai báo

```
Type
  Table = array[1..25, 1..80] of word;
Var
  p: ^Table;
```

sẽ "nhúng" biến p^ (kiểu Table) vào vùng Video Ram nhờ lệnh gán p := Ptr($B800, 0). Lúc này phần tử p^[i, j] (kiểu word) sẽ là nội dung của dòng i, cột j trên màn hình, trong đó byte thấp là nội dung ký tự, byte cao là nội dung mầu.

Trường hợp p là *con trỏ không định kiểu*. Khi đó p^ là một biến không định kiểu có địa chỉ được lưu tại p, nó chỉ được dùng ở những vị trí tham biến không định kiểu cùng với

một tham số kích thước kèm theo. Trường hợp này, p^ được dùng khi không quan tâm đến kiểu dữ liệu được truy cập.

Thí dụ chương trình làm "rỗng" màn hình (xem chương trước) có thể viết lại nhờ một con trỏ không định kiểu (pointer) như sau:

```
Program EmptyScr;
Var p: pointer;
Begin
  p := Ptr($B800, 0);          {trỏ tới Video Ram}
  FillChar(p^, 4000, 0);       {đặt tất cả bằng 0}
End.
```

Chú ý rằng, với p là pointer, không được dùng p^ như là những biến định kiểu thông thường, chẳng hạn p^ không được tham gia trong các phép gán, nhập, xuất, ... Thí dụ, những lệnh sau đây là không được chấp nhận: Write(p^), Read(p^), p^ := ...

3. Cấp phát động

Việc dùng p^ bằng cách gán cho p một giá trị địa chỉ không phải là việc cung cấp một biến p^ mới, vì thế phải hết sức thận trọng khi truy cập p^. Muốn p^ thực sự được cấp phát như một biến, phải dùng kỹ thuật cấp phát động cho p. Cấp phát động được thực hiện bằng câu lệnh trong thân chương trình (chứ không phải bằng khai báo) và được thu hồi nếu cần (cũng bằng câu lệnh trong chương trình). Vùng được cấp phát động bao giờ cũng là vùng nhớ tự do (Heap). Địa chỉ của vùng cấp phát động được quản lý bởi các biến con trỏ. Việc cấp phát động cho phép người lập trình sử dụng bộ nhớ linh hoạt và tiết kiệm, đặc biệt, nhờ cấp phát động, người ta có thể xây dựng được những kiểu dữ liệu động, cho phép khai thác tối đa bộ nhớ.

Cấp phát động là một ứng dụng quan trọng của con trỏ. Việc dùng con trỏ thường được gắn liền với việc cấp phát động, đến nỗi nhiều người cho rằng cứ dùng con trỏ là phải cấp phát động, mặc dù điều này là không bắt buộc.

Những biến được xin trong vùng Heap nhờ cấp phát động, được gọi là những biến động (*dynamic variable*) để phân biệt với những biến thông thường được xin từ khai báo.

3.1. Các thủ tục cấp phát động

Turbo Pascal tổ chức hai cách cấp phát động: Một, dành cho các con trỏ định kiểu và hai, dành cho con trỏ bất kỳ (không quan tâm đến kiểu).

Cấp phát định kiểu. Giả sử p là một con trỏ định kiểu. Thủ tục New(p) sẽ cấp một vùng nhớ trên Heap cho con trỏ p với kích thước bằng kích thước của kiểu dữ liệu và gán địa chỉ của vùng này cho p. Khi đó biến động định kiểu p^ sẽ truy cập vùng được cấp phát chừng nào p còn giữ địa chỉ vùng này. Mọi thao tác trên biến thông thường đều được áp dụng cho p^.

Thí dụ khai báo:

```
Var
```

```
  p: ^integer;
```

sẽ xác định việc cấp phát một biến động kiểu nguyên `p^` sau lời gọi `New(p)`. Nó được dùng như một biến nguyên thông thường.

Một ứng dụng đơn giản của cấp phát kiểu này là dùng Heap để san sẻ bớt gánh nặng cho vùng Data khi cần cung cấp quá nhiều cho các biến chung. Thí dụ, khai báo dưới đây:

```
Type
  matrix = array[1..100, 1..100] of real;
Var
  A, B: matrix;
```

đòi hỏi việc cấp 120000 byte của vùng Data, chắc chắn sẽ gặp lỗi *Too many variables* khi biên dịch. Lỗi này có thể khắc phục một cách đơn giản mà không ảnh hưởng đến các việc tổ chức khác của chương trình bằng cách thay một khai báo tĩnh bằng một cấp phát động. Chẳng hạn khai báo trên có thể thay bằng khai báo dưới đây chỉ đòi hỏi cấp 60004 byte trong vùng Data:

```
Type
  matrix = array[1..100, 1..100] of real;
Var
  A: matrix;
  B: ^matrix;
```

với chú ý rằng trong thân chương trình phải gọi `New(B)` trước khi dùng và sau đó thay `B^` vào các vị trí xuất hiện `B`.

Vùng nhớ đã cấp phát cho con trỏ `p` bằng thủ tục `New(p)` sẽ được thu hồi nhờ thủ tục `Dispose(p)`. Vùng nhớ đã cấp phát sẽ được hệ thống bảo vệ cho đến khi được thu hồi.

```
Program ThiDu;
Type
  st18 = string[18];
Var
  p: ^st18;
Begin
  New(p);
  p^ := 'Now you see it...';
  Writeln('p^ khi cap phat = ', p^);
  Dispose(p);
  Writeln('p^ khi thu hoi  = ', p^);
  Readln;
End.
```

Chương trình trên đưa ra nội dung `p^` trước và sau khi thu hồi. Chú ý rằng mặc dù không có lệnh nào làm thay đổi nội dung của `p^` trước khi thu hồi nhưng sau khi thu hồi có thể `p^` vẫn bị thay đổi do không còn được bảo vệ.

Cấp phát không định kiểu. Có thể xin cấp phát một số byte trên vùng Heap cho một con trỏ mà không quan tâm đến kiểu dữ liệu bằng thủ tục `GetMem(var p: pointer; size: word)`. Con trỏ `p` được thích ứng với bất kỳ kiểu con trỏ nào. Tham trị `size` xác định kích thước (tính bằng byte) xin cấp phát. Nếu không xác định kiểu, biến động `p^` chỉ

được dùng như một biến không định kiểu. Để thu hồi vùng nhớ được cấp phát bằng `GetMem` cho con trỏ `p`, bạn dùng thủ tục `FreeMem(var p: pointer; size: word)` trong đó `size` là kích thước cần thu hồi.

Việc cấp phát động không định kiểu được dùng trong những tình huống cần thích ứng với các kiểu dữ liệu khác nhau, trong đó kích thước của dữ liệu không biết trước. Thí dụ dưới đây xây dựng thủ tục `SwapVal`, hoán chuyển giá trị của 2 biến `x`, `y` có kiểu bất kỳ, không phụ thuộc vào kích thước của các biến này (xem chương trước):

```
procedure SwapVal(var x, y; size: word);
var
  temp: pointer;
begin
  GetMem(temp, size);        {cấp phát động cho temp}
  Move(x, temp^, size);
  Move(y, x, size);
  Move(temp^, y, size);
  FreeMem(temp, size);       {thu hồi vùng nhớ đã cấp phát cho temp}
end;
```

Lời gọi `SwapVal(x, y, SizeOf(x))` được áp dụng cho mọi biến `x`, `y` để đổi giá trị của các biến này cho nhau. Việc dùng biến động `temp^` trong thủ tục (thay cho việc khai báo tĩnh trước đây) nhằm thích ứng với mọi kích thước của dữ liệu.

Một ứng dụng khác của cấp phát không định kiểu là xây dựng mảng động, nghĩa là mảng mà kích thước của nó được xác định tùy theo việc dùng. Để làm điều này, con trỏ `p` được khai báo trỏ tới kiểu mảng một phần tử, mỗi khi dùng, thủ tục `GetMem` sẽ được gọi đến để cấp phát cho `p` với kích thước yêu cầu và lúc này có thể dùng `p^` như một mảng với kích thước đã cấp phát. Chú ý rằng việc dùng mảng động sẽ gây ra lỗi *Rang check error* (nếu hệ thống đã bật chế độ kiểm tra này) vì đã cố tình dùng vượt chỉ số so với khai báo (nhưng sẽ không gây ra lỗi lôgic nào nếu chỉ dùng trong phạm vi được cấp phát) vì thế cần tắt chế độ kiểm tra này nếu hệ thống đã bật (giá trị ngầm định là tắt), bằng cách dùng định hướng biên dịch `{$R-}`. Thí dụ dưới đây minh họa việc dùng một mảng động, trong đó mảng `p^` được xây dựng lại trong mỗi lần lặp với kích thước hỏi từ bàn phím:

```
Type
  Arr = array[1..1] of integer;
Var
  p: ^Arr;
  n: word;
Begin
  Write('Kich thuoc mang (vao 0 de ket thuc): ');
  Readln(n);
  while n > 0 do
  begin
    {cấp phát mảng động kích thước n}
    GetMem(p, n);
    {truy cập p^[1], ..., p^[n]}
    ..................
```

```
    {thu hồi mảng đã cấp phát}
    FrreMem(p, n);
    {cấp phát lại}
    Write('Kich thuoc mang (vao 0 de ket thuc): ');
    Readln(n);
  end;
End.
```

Thu hồi hàng loạt. Có thể thu hồi hàng loạt bắt đầu từ một địa chỉ cấp phát nào đấy thay cho việc thu hồi lẻ tẻ từng vùng. Muốn vậy, bạn cần đánh dấu địa chỉ này bằng việc dùng thủ tục `Mark(var Top: pointer)`. Biến trỏ `Top` sẽ ghi nhận địa chỉ đỉnh Heap trước khi cấp phát. Sau đấy thủ tục `Release(var Top: pointer)` sẽ thu hồi tất cả các vùng được cấp phát bắt đầu từ địa chỉ được lưu trong `Top` cho đến đỉnh Heap hiện tại.

```
Mark(Top);        {đánh dấu vị trí bắt đầu cấp phát}
{các lệnh cấp phát}
.............
Release(Top);  {thu hồi các vùng đã cấp phát trong phạm vi từ Mark(Top) đến Release(Top)}
```

3.2. Quản lý Heap

Để quản lý vùng Heap, unit System có xây dựng một số biến và hàm chuẩn sau đây:

```
Var HeapEnd: Pointer;
```

Biến `HeapEnd` trỏ tới điểm cuối cùng của Heap được chương trình sử dụng.

```
Var HeapOrg: Pointer;
```

Biến `HeapOrg` trỏ tới điểm bắt đầu của Heap được dùng cấp phát động.

Các biến `HeapEnd`, `HeapOrg` được hệ thống khởi gán khi chương trình bắt đầu và không thay đổi giá trị trong suốt quá trình chạy chương trình.

```
Var HeapPtr: Pointer;
```

Biến `HeapPtr` trỏ tới đỉnh của Heap, tức là đỉnh của vùng nhớ cấp phát động (cũng là đáy của vùng nhớ còn tự do). Khởi đầu, giá trị của `HeapPtr` bằng giá trị của `HeapOrg` và được phát triển theo một chồng hướng lên.

```
Function MemAvail: LongInt;
```

Hàm `MemAvail` trả lại tổng số kích thước (tính theo byte) của các khối còn tự do trong Heap. Các khối còn tự do không nhất thiết liền kề nhau do hiện tượng phân mảnh.

```
Function MaxAvail: LongInt;
```

Hàm `MaxAvail` trả lại kích thước (tính theo byte) của khối còn tự do liền nhau lớn nhất trong Heap. Nếu kích thước này nhỏ hơn kích thước xin cấp phát thì sẽ gây ra lỗi tràn Heap khi chạy chương trình.

Dưới đây là chương trình minh họa trạng thái của Heap khi cấp phát động:

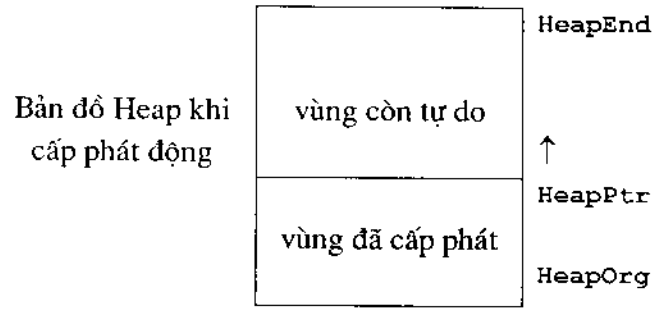
```
Program DemoHeap;
Type arr = array[1..65535] of byte;
Var
  p, q: ^arr;
  i: integer;
```

```
  Top: pointer;   {ghi nhận định Heap để thu hồi hàng loạt}

  procedure DisplayHeap;
  {hiển thị trạng thái của Heap}
  begin
    Writeln('  HeapOrg = ', Seg(HeapOrg^), ':', Ofs(HeapOrg^));
    Writeln('  HeapEnd = ', Seg(HeapEnd^), ':', Ofs(HeapEnd^));
    Writeln('  HeapPtr = ', Seg(HeapPtr^), ':', Ofs(HeapPtr^));
    Writeln('  MaxAvail = ', MaxAvail, ', MemAvail = ', MemAvail);
  end;

Begin
  Writeln('Trang thai Heap truoc khi cap phat:');
  DisplayHeap;
  Writeln('An Enter de bat dau cap phat');
  Readln;
  Mark(Top);      {đánh dấu vị trí bắt đầu cấp phát}
  i := 1; q := Nil;
  while MaxAvail >= SizeOf(arr) do
  begin
    New(p);
    Writeln('Trang thai Heap sau khi cap phat lan ', i, ':');
    DisplayHeap;
    if i = 2 then q := p;    {lưu địa chỉ của vùng được cấp lần 2}
    i := i+1;
  end;
  Writeln('Lan cap phat ', i, ' khong du Heap!');
  if q <> Nil then
  begin
    Writeln('An Enter de thu hoi vung nho duoc cap lan 2');
    Readln;
    Dispose(q);
    Writeln('Bay gio trang thai Heap la:');
    DisplayHeap;
  end;
  Writeln('An Enter de thu hoi vung nho duoc cap lan cuoi');
  Readln;
  Dispose(p);
  Writeln('Bay gio trang thai Heap la:');
  DisplayHeap;
  Writeln('An Enter de thu hoi hang loat');
  Readln;
  Release(Top);   {thu hồi tất cả vùng còn lại}
  Writeln('Bay gio trang thai Heap la:');
  DisplayHeap;
  Writeln('An Enter de ket thuc');
  Readln;
```

```
End.
```

Cần lưu ý mấy điểm khi theo dõi kết quả của chương trình:

- Các giá trị `HeapEnd`, `HeapOrg` không thay đổi.
- Giá trị `HeapPtr` phát triển theo một chồng hướng lên.
- Có thể cấp phát cho một con trỏ nhiều lần, nhưng con trỏ này chỉ quản lý vùng được cấp cuối cùng. Muốn quản lý vùng được cấp trước đấy, phải lưu địa chỉ vào một con trỏ khác.
- Việc thu hồi các vùng cấp không phải vùng sau cùng sẽ gây ra hiện tượng phân mảnh. Khi đó giá trị của các hàm `MaxAvail` và `MemAvail` không còn giống nhau.



4. Danh sách liên kết

Một trong những ứng dụng quan trọng của cấp phát động là tạo ra những cấu trúc dữ liệu động, cho phép khai thác linh hoạt và tối đa bộ nhớ. Các cài đặt dùng cấu trúc dữ liệu động không bị hạn chế (về lôgic) bởi kích thước của dữ liệu, và đặc biệt thuận lợi khi phải dùng thường xuyên các thao tác chèn hoặc xóa. Mục này thảo luận một ứng dụng điển hình của cấp phát động là xây dựng danh sách liên kết.

4.1. Định nghĩa và khai báo

Danh sách liên kết là một dãy các phần tử có cùng kiểu dữ liệu, trong đó cần chỉ rõ mối liên kết trước-sau của các phần tử trong danh sách. Một dạng cài đặt thường gặp của danh sách liên kết là dùng mảng, trong đó mối liên kết được ẩn dưới các giá trị kề nhau của chỉ số. Việc dùng mảng cho phép truy cập nhanh đến các phần tử của danh sách nhờ các giá trị chỉ số. Tuy nhiên các hạn chế của việc dùng mảng là:

- Khai thác bộ nhớ không linh hoạt do phải khai báo trước.
- Không thể làm việc với những danh sách lớn hơn 64 Kb vì mảng cần được cấp một vùng nhớ liên tục.
- Các thao tác chèn, xóa đòi hỏi phải dồn chỉ số cho các phần tử mảng.

Việc cấp phát động cho các phần tử của danh sách cho phép khai thác bộ nhớ tốt hơn. Một mặt, nó không phụ thuộc vào kích thước danh sách vì không phải khai báo trước. Mặt khác, do các phần tử của danh sách được cấp phát riêng rẽ nên danh sách không bị hạn

chế bởi giới hạn 64 Kb. Ngoài ra, như sẽ thấy, các thao tác chèn, xóa không đòi hỏi phải dồn phần tử.

Để xây dựng danh sách liên kết bằng cấp phát động, mỗi phần tử của nó (mà ta sẽ gọi là *nút - node*) cần phải xác định hai thành phần: Một là nội dung dữ liệu mà nút lưu trữ, hai là địa chỉ của nút kế tiếp. Một cách tự nhiên là dùng kiểu bản ghi để định nghĩa một nút như vậy:

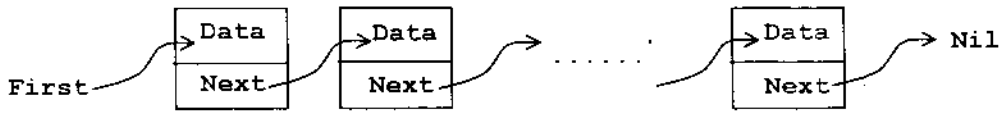
```
Type
  DataType = ...    {kiểu của dữ liệu}
  PtrType = ^Node;
  Node = record
           Data: DataType;    {lưu trữ dữ liệu}
           Next: PtrType;     {trỏ đến nút kế tiếp}
         end;
```

Khai báo trên định nghĩa kiểu `PtrType` trỏ tới `Node`, trong đó `Node` là kiểu bản ghi mô tả một nút gồm hai trường: Trường `Data` dùng để lưu dữ liệu với giả thiết tên kiểu là `DataType` (là một kiểu nào đấy đã được định nghĩa, có thể gồm nhiều thành phần) còn trường `Next` thuộc kiểu `PtrType`, dùng để lưu địa chỉ của nút kế tiếp. Chú ý rằng, trong khai báo trên, `PtrType` cần phải định nghĩa trước `Node`.

Để quản lý danh sách, cần có một biến toàn thể thuộc kiểu `PtrType` lưu địa chỉ của nút đầu tiên trong danh sách. Ta sẽ đặt tên cho biến này là `First`:

```
Var
  First: PtrType;
```

Từ giá trị của `First` ta có thể truy cập đến nút đầu tiên, sau đó từ trường `Next` của nút này, ta có thể truy cập đến nút thứ hai, ... Để báo hết danh sách, trường `Next` của nút cuối cùng cần trỏ vào `Nil`. Bằng cách như vậy, ta có thể truy cập được toàn bộ các nút của danh sách.



4.2. Các thao tác trên danh sách liên kết

Giả thiết danh sách liên kết đã được định nghĩa như khai báo ở mục trên, với `First` (biến toàn thể) là con trỏ trỏ tới nút đầu danh sách. Dưới đây là một số thao tác cơ bản trên danh sách:

Khởi tạo danh sách rỗng

Để khởi tạo danh sách rỗng (chưa có nút nào), ta gán `Nil` cho con trỏ `First`:

```
First := Nil;
```

Điều này cũng có nghĩa giá trị của biểu thức lôgic `First = Nil` được dùng để kiểm tra danh sách rỗng hay không.

Duyệt danh sách

Để lần lượt đi qua các nút của danh sách, cần phải khởi động một con trỏ `CurrPtr` trỏ đến nút đầu tiên, rồi đi tiếp qua danh sách theo các trường liên kết và xử lý dữ liệu lưu trữ trong mỗi nút:

```
CurrPtr := First;      {khởi động tại nút đầu tiên}
while CurrPtr <> Nil do
begin
  {xử lý CurrPtr^.Data}
  ..............
  CurrPtr := CurrPtr^.Next;      {đi tới nút kế tiếp}
end;
```

Thao tác trên bao gồm cả trường hợp danh sách là rỗng. Khi đó vòng lặp `while` không được thực hiện lần nào.

Chèn một nút vào danh sách

Giả sử cần chèn một phần tử có nội dung dữ liệu là `Item` (có kiểu `DataType`) vào danh sách. Vị trí cần chèn được xác định là sau nút đang được trỏ bởi con trỏ `PredPtr` (với quy ước nếu con trỏ này rỗng thì vị trí chèn là vị trí đầu tiên). Thao tác được tiến hành theo các bước:

- Xin cấp phát một nút mới cho con trỏ `TempPtr` để lưu `Item`.
- Nối nút này vào danh sách tại vị trí cần chèn:
  - Nếu `PredPtr` là `Nil` (chèn vào đầu danh sách) thì gán `TempPtr^.Next` bằng `First` rồi sau đó ghi nhận lại `First` bằng `TempPtr`,
  - Trái lại (chèn vào sau `PredPtr`) thì gán `TempPtr^.Next` bằng `PredPtr^.Next` rồi sau đó ghi nhận lại `PredPtr^.Next` bằng `TempPtr`.

Mô hình chèn một nút như sau:

```
New(TempPtr);                  {xin cấp phát nút mới}
TempPtr^.Data := Item;         {ghi nhận dữ liệu}
if PredPtr = Nil then          {chèn vào đầu danh sách}
begin
  TempPtr^.Next := First;
  First := TempPtr;
end
else                           {chèn vào sau PredPtr}
begin
  TempPtr^.Next := PredPtr^.Next;
  PredPtr^.Next := TempPtr;
end;
```

Để ý rằng, việc chèn một nút không ảnh hưởng đến mối liên kết của các nút đứng sau vị trí chèn, vì thế không có hiện tượng phải dồn phần tử như trong cài đặt mảng.

Xóa một nút khỏi danh sách

Đây là thao tác ngược với thao tác chèn. Giả sử cần loại bỏ nút đứng sau nút đang được trỏ bởi `PredPtr` (cũng giống như trên, nếu con trỏ này rỗng thì có nghĩa là nút đầu tiên bị xóa). Dĩ nhiên việc xóa chỉ được tiến hành khi danh sách là khác rỗng:

- Ghi nhận con trỏ `TempPtr` là nút cần xóa: Gán `TempPtr` bằng `PredPtr^.Next` (nếu `PredPtr` là rỗng thì gán `TempPtr` bằng `First`).
- Loại nút cần xóa ra khỏi danh sách: Đặt lại `PredPtr^.Next` bằng `TempPtr^.Next` (nếu `PredPtr` là rỗng thì đặt lại `First` bằng `TempPtr^.Next`).
- Thu hồi vùng nhớ được trỏ bởi `TempPtr`.

Mô hình xóa một nút như sau:

```
if First <> Nil then              {chỉ xóa khi danh sách khác rỗng}
begin
  if PredPtr = Nil then           {xóa nút đầu}
  begin
    TempPtr := First;
    First := TempPtr^.Next;
  end
  else                            {xóa nút sau PredPtr}
  begin
    TempPtr := PredPtr^.Next;
    PredPtr^.Next := TempPtr^.Next;
  end;
  Dispose(TempPtr);               {thu hồi vùng nhớ}
end;
```

Cũng giống như chèn, việc xóa một nút không ảnh hưởng đến mối liên kết của các nút đứng sau, vì thế không phải dồn phần tử như trong cài đặt mảng.

Xây dựng danh sách

Để xây dựng một danh sách, người ta tiến hành thêm những nút mới vào bắt đầu từ một danh sách rỗng. Việc chèn một nút vào một vị trí thích hợp nào đấy trong danh sách, đòi hỏi trước tiên phải xác định được nút đi trước vị trí này. Đây là cách tạo danh sách mà mối liên kết cần đảm bảo một trật tự cho trước trên các nút. Thông thường, khi không quan tâm đến trật tự các nút, người ta thường chọn vị trí chèn là trước nút đầu hoặc sau nút cuối danh sách để bớt được việc tìm kiếm các nút này.

a. Nối nút mới vào đầu danh sách: Việc nối nút mới vào đầu danh sách rất đơn giản, bắt đầu là đặt trường liên kết của nút mới trỏ vào nút đầu, sau đó ghi nhận lại nút đầu là nút mới. Dưới đây là mô hình tạo danh sách, bắt đầu từ danh sách rỗng, bằng cách lặp lại việc nối một nút vào đầu danh sách. Giá trị `Stop` dùng trong mô hình là một biểu thức lôgic, nhằm kiểm tra việc kết thúc của vòng lặp:

```
First := Nil;                     {khởi gán danh sách rỗng}
while not Stop do                 {vòng lặp nối vào đầu danh sách}
begin
  New(TempPtr);                   {xin cấp phát nút mới}
  TempPtr^.Data := Item;          {ghi nhận dữ liệu}
  TempPtr^.Next := First;         {nối vào trước First}
  First := TempPtr;               {ghi nhận lại First}
end;
```

Danh sách được tạo theo mô hình này có chiều truy cập ngược với chiều tạo: Phần tử tạo sau được truy cập trước. Người ta gọi nó là danh sách LIFO (*Last In First Out - vào sau ra trước*). Cũng vì thế, trong nhiều tài liệu, con trỏ trỏ tới nút đầu danh sách LIFO được đặt tên là `Last` thay cho `First` với ý nghĩa nhấn mạnh đây là phần tử cuối. Danh sách LIFO là phương tiện để xây dựng ngăn xếp (stack).

b. Nối nút mới vào cuối danh sách: Việc nối một nút mới vào cuối danh sách có phức tạp hơn chút ít. Trước hết cần phải tổ chức thêm con trỏ `Last` luôn trỏ tới cuối danh sách nhằm không phải tìm kiếm lại. Như thế, sau mỗi lần nối, phải cập nhật lại giá trị cho con trỏ này. Ngoài ra việc nối vào một danh sách rỗng phải được xử lý riêng vì khi đó không có nút đi trước. Mô hình dưới đây trình bày cách tạo một danh sách bằng cách lặp lại việc nối một nút vào cuối danh sách. Giá trị `Stop` dùng trong mô hình là một biểu thức lôgic, nhằm kiểm tra việc kết thúc của vòng lặp:

```
First := Nil;                    {khởi gán danh sách rỗng}
while not Stop do                {vòng lặp nối vào cuối danh sách}
begin
  New(TempPtr);                  {xin cấp phát nút mới}
  TempPtr^.Data := Item;         {ghi nhận dữ liệu}
  if First = Nil then            {ghi nhận First nếu danh sách rỗng}
    First := TempPtr
  else                           {nối vào cuối danh sách}
    Last^.Next := TempPtr;
  Last := TempPtr;               {ghi nhận lại Last}
  Last^.Next := Nil;             {trỏ hết danh sách}
end;
```

Chú ý rằng, trong vòng lặp, tình huống gán giá trị cho `First` được thực hiện đúng lần đầu tiên khi danh sách rỗng. Vì vậy, trong một số trường hợp, tình huống này được xử lý riêng, trước khi thực hiện lặp.

Danh sách được tạo theo mô hình trên được gọi là danh sách FIFO (*First In First Out - vào trước ra trước*) vì chiều truy cập cũng là chiều tạo. Danh sách FIFO là phương tiện để xây dựng hàng đợi (queue).

Tìm kiếm một nút trong danh sách

Giả sử cần tìm một nút trong danh sách có nội dung dữ liệu là `Item` (có kiểu `DataType`). Việc tìm kiếm được tiến hành tuần tự theo chiều duyệt danh sách bắt đầu từ `First` nhờ con trỏ `CurrPtr`. Mỗi khi đi đến một nút, vùng dữ liệu của `CurrPtr` lại được so sánh với `Item`. Nếu tìm thấy, con trỏ `CurrPtr` sẽ xác định nút cần tìm, trái lại con trỏ `CurrPtr` sẽ có giá trị `Nil`:

```
CurrPtr := First;
while (CurrPtr <> Nil) and (CurrPtr^.Data <> Item) do
  CurrPtr := CurrPtr^.Next;
```

Để tiện cho việc trình bày, trong mô hình trên, toán tử so sánh khác nhau <> được lạm dụng cho các giá trị kiểu `DataType`.

Nhiều khi việc tìm kiếm cần phục vụ cho việc xóa hay chèn một nút vào danh sách. Khi đó, ta cần xác định thêm nút đi trước vị trí cần xóa hoặc chèn. Để làm thí dụ, xét bài toán dưới đây:

Giả sử danh sách đã được xếp theo chiều tăng dần của trường dữ liệu (để tiện trình bày, ta giả thiết trường dữ liệu `Data` ở đây chính là trường khóa và các toán tử so sánh được áp dụng cho trường này). Cần chèn nút có nội dung dữ liệu `Item` vào đúng vị trí của nó trong danh sách. Như thế, hoặc nút này được chèn lên đầu hoặc nút này được chèn sau nút cuối cùng trong danh sách thỏa mãn `Data < Item`. Bài toán được giải quyết qua các bước sau:

Đầu tiên xác định nút cần đi trước nút cần chèn nhờ con trỏ `PredPtr` và nút sau đấy nhờ con trỏ `CurrPtr`:

```
PredPtr := Nil; CurrPtr := First;
while (CurrPtr <> Nil) and (CurrPtr^.Data < Item) do
begin
  PredPtr := CurrPtr;
  CurrPtr := CurrPtr^.Next;
end;
```

Chú ý rằng cài đặt trên bao gồm cả tình huống vị trí cần chèn là nút đầu hoặc nút cuối của danh sách: `CurrPtr` trỏ vào nút đầu là tương đương với `PredPtr` có giá trị rỗng và `PredPtr` trỏ vào nút cuối là tương đương với `CurrPtr` có giá trị rỗng.

Sau đó, tạo nút có nội dung `Item` nhờ con trỏ `TempPtr` rồi chèn nó vào vị trí sau `PredPtr` và trước `CurrPtr`:

```
New(TempPtr);                 {xin cấp phát nút mới}
TempPtr^.Data := Item;        {ghi nhận dữ liệu}
TempPtr^.Next := CurrPtr;     {chèn vào trước CurrPtr}
if CurrPtr = First then       {chèn vào đầu danh sách}
  First := TempPtr;
else                          {chèn vào sau PredPtr}
  PredPtr^.Next := TempPtr;
```

Chú ý. Trong các cài đặt tìm kiếm đã nêu, cần giả thiết hệ thống chạy dưới chế độ đánh giá biểu thức lôgic không đầy đủ (định hướng biên dịch `{$B-}`) để tránh lỗi truy cập vào một nút không có. Các phiên bản hiện nay của Turbo Pascal đều ngầm định chạy dưới chế độ này.

4.3. Thí dụ ứng dụng: Xây dựng bảng từ

Bảng từ của một văn bản là một danh sách chứa tất cả các từ khác nhau của văn bản đó. Các từ trong danh sách cần được xếp theo thứ tự abc. Một cách hiệu quả để xây dựng một bảng từ như thế là dùng danh sách liên kết. Bắt đầu từ một danh sách rỗng, ta lần lượt đọc các từ trong văn bản. Mỗi từ được đọc ra cần chèn vào vị trí thích hợp nếu nó chưa có trong danh sách. Để đọc được các từ trong văn bản, cần phải thỏa thuận trước một định nghĩa nào đấy về từ.

Dưới đây trình bày một chương trình xây dựng bảng từ. Văn bản chứa các từ này được lưu trữ trong một file văn bản. Để đơn giản, ta giả thiết rằng các từ trong văn bản gồm

toàn chữ cái to và có độ dài không quá 20, bất cứ một ký tự nào không phải là chữ cái to đều được xem như dấu phân cách từ.

Bảng từ được xây dựng như một danh sách liên kết, trong đó mục dữ liệu của mỗi nút là một từ. Ban đầu danh sách được khởi động rỗng. Quá trình đọc một từ được lặp lại cho đến hết file. Mỗi từ đọc được, sẽ được trao cho một thủ tục kiểm tra, hoặc thủ tục này xác định được từ đang xét đã có trong danh sách, hoặc nó tìm được các nút trước và sau vị trí cần chèn của từ để vẫn đảm bảo được thứ tự trong danh sách. Nếu từ đã có thì nó được bỏ qua, trái lại, nó được chèn vào danh sách tại vị trí đã tìm. Danh sách hoàn tất khi quá trình đọc kết thúc. Cuối cùng là việc duyệt danh sách để đưa ra lần lượt nội dung các từ đã được tìm.

```
Program TextConcordance;
{xây dựng bảng từ của văn bản}
Type
  DataType = string[20];        {kiểu dữ liệu của từ}
  PtrType = ^Node;
  Node = record
           Data: DataType;      {nội dung từ}
           Next: PtrType;       {trỏ tới nút kế tiếp}
         end;
Var
  First: PtrType;     {trỏ đầu danh sách}

  procedure CreatListFrom(Name: string);
  {tạo danh sách từ file văn bản có tên Name}
  var
    F: text;
    Item: DataType;
    Found: boolean;
    PredPtr, CurrPtr: PtrType;

    function GetWord: DataType;
    {trả lại từ được đọc của văn bản, nếu không có, trả lại rỗng}
    var W: DataType;
        More: boolean;
        Ch: char;
    begin
      W := ''; More := false;
      while not More and not Eof(f) do
      begin             {bỏ qua các ký tự khác chữ cái to}
        Read(F, Ch);
        More := Ch in ['A'..'Z'];
      end;
      while More do
      begin             {tích lũy từ}
```

```
      W := W+Ch;
      if not Eof(f) then
      begin
        Read(F, Ch);
        More := Ch in ['A'..'Z'];
      end else More := false;
    end;
    GetWord := W;
  end;

  procedure Search(Item: DataType; var Found: boolean;
                   var PredPtr, CurrPtr: PtrType);
  {tìm nút có nội dung Item trong danh sách, nếu thấy thì Found nhận giá trị True, trái lại Found
   nhận giá trị False và PredPtr, CurrPtr tương ứng nhận nút trước và sau vị trí cần chèn}
  begin
    PredPtr := Nil; CurrPtr := First;
    while (CurrPtr <> Nil) and (CurrPtr^.Data < Item) do
    begin
      PredPtr := CurrPtr;
      CurrPtr := CurrPtr^.Next;
    end;
    if CurrPtr = Nil then Found := false
    else Found := CurrPtr^.Data = Item;
  end;

  procedure InsertList(Item: DataType;
                       var PredPtr, CurrPtr: PtrType);
  {chèn nút có nội dung Item vào danh sách; PredPtr, CurrPtr tương ứng là các nút trước
   và sau vị trí cần chèn}
  var TempPtr: PtrType;
  begin
    New(TempPtr);
    TempPtr^.Data := Item;
    TempPtr^.Next := CurrPtr;
    if CurrPtr = First then
      First := TempPtr
    else
      PredPtr^.Next := TempPtr;
  end;

begin  {CreatListFrom}
  Assign(F, Name); Reset(F);
  First := Nil;
  while not Eof(F) do
  begin
    Item := GetWord;
    if Item <> '' then
```

```
      begin
        Search(Item, Found, PredPtr, CurrPtr);
        if not Found then
          InsertList(Item, PredPtr, CurrPtr);
      end;
    end;
    Close(F);
  end;

  procedure WriteListTo(Name: string);
  {ghi danh sách lên file văn bản có tên Name}
  var CurrPtr: PtrType;
      F: text;
      Count: byte;
  begin
    Assign(F, Name); Rewrite(F);
    CurrPtr := First; Count := 0;
    while CurrPtr <> Nil do
    begin
      Inc(Count);
      Write(F, CurrPtr^.Data, ' ':21-Length(CurrPtr^.Data));
      CurrPtr := CurrPtr^.Next;
      if Count = 4 then
      begin
        Writeln(F); Count := 0;
      end;
    end;
    Close(F);
  end;

Begin
  CreatListFrom('VANBAN');
  WriteListTo('BANGTU');
End.
```

Chương trình gồm hai thủ tục: Thủ tục `CreatListFrom(Name: string)` tạo danh sách từ file văn bản có tên `Name` với nút đầu được trỏ bởi `First` (biến toàn cục) và thủ tục `WriteListTo(Name: string)` ghi danh sách này lên file văn bản có tên `Name`, trong đó mỗi dòng ghi 4 từ. Các chương trình con còn lại được thiết kế để phục vụ thủ tục `CreatListFrom`:

- Hàm `GetWord: DataType`, trả lại một từ được đọc trong văn bản, nếu không có từ nào được đọc, hàm trả về một từ rỗng.
- Thủ tục `Search(Item: DataType; var Found: boolean; var PredPtr, CurrPtr: PtrType)` tìm nút có nội dung `Item` trong danh sách. Nếu tìm thấy, `Found` sẽ nhận giá trị `True`, trái lại, `Found` sẽ nhận giá trị `False`. Trường hợp này,

thủ tục xác định vị trí cần chèn nút có nội dung `Item` vào danh sách để đảm bảo thứ tự cho bảng từ: `PredPtr` sẽ trỏ vào nút trước và `CurrPtr` sẽ trỏ vào nút sau vị trí cần chèn.

- Thủ tục `InsertList(Item: DataType; var PredPtr, CurrPtr: PtrType)` chèn nút có nội dung `Item` vào danh sách, sau nút được trỏ bởi `PredPtr` và trước nút được trỏ bởi `CurrPtr`. Thủ tục bao gồm cả tình huống vị trí chèn là đầu hoặc cuối danh sách.

Chương trình được chạy thử cho file văn bản có tên là VANBAN và kết quả được ghi ra file văn bản có tên là BANGTU.

Nội dung file VANBAN được thử:

```
- the PROGRAM heading specifies the PROGRAM's name AND parameters;
  it is purely informative AND has no meaning TO the PROGRAM itself.
- the USES clause lists the UNITs used by the PROGRAM.
- the LABEL, CONSTant, VARiable, PROCEDURE, AND FUNCTION declaration
  parts can be listed IN any order, AND repeated any number OF times.
- the statement part specifies the statements TO be executed when the
  PROGRAM runs.
```

Kết quả trên file BANGTU:

```
AND        CONST      FUNCTION   IN
LABEL      OF         PROCEDURE  PROGRAM
TO         UNIT       USES       VAR
```

5. Một số biến thể của danh sách liên kết

Một hạn chế của danh sách kiên kết "tiêu chuẩn" được cài đặt trong mục trước là mọi nút chỉ được đi đến một cách tuần tự từ nút đầu, vì thế trong những công việc thường xuyên phải truy cập đến những nút khác nhau, cài đặt này trở nên kém hiệu quả. Trong nhiều ứng dụng, người ta thấy việc dùng những thao tác truy cập cũng như các mối liên kết khác có thể tiện lợi hơn. Việc thay đổi này dẫn đến các biến thể của danh sách liên kết. Các bạn có thể tìm đọc chi tiết các biến thể này cũng như các ứng dụng của nó trong các giáo trình bàn về cấu trúc dữ liệu. Mục này chỉ giới thiệu sơ lược một số biến thể thường gặp nhất.

5.1. Ngăn xếp liên kết

Có thể cấp phát động để xây dựng một *ngăn xếp* (*stack*) theo mô hình LIFO. Các thao tác cơ bản của ngăn xếp là đẩy vào (Push) và lấy ra (Pop) một phần tử tại đầu ngăn xếp. Vị trí đầu ngăn xếp được gọi là đỉnh của nó. Việc khai báo giống như danh sách liên kết, ngoại trừ việc thay một số tên cho phù hợp:

```
Type
  DataType = ...
  PtrType = ^StackNode;
  StackNode = record
```

```
            Data: DataType;
            Next: PtrType;
          end;
Var
  StackPtr: PtrType;   {trỏ đỉnh ngăn xếp}
```

Các thao tác khởi tạo ngăn xếp rỗng, duyệt ngăn xếp được xây dựng như một danh sách liên kết. Thao tác đẩy một mục dữ liệu `Item` vào ngăn xếp được thực hiện theo các bước:

- Xin cấp phát một nút mới cho con trỏ `TempPtr` để lưu `Item`.
- Đẩy nút này vào đỉnh ngăn xếp bằng cách đặt `TempPtr^.Next` bằng `StackPtr` rồi sau đó ghi nhận lại `StackPtr` bằng `TempPtr`.

Thủ tục `Push` dưới đây đẩy mục `Item` vào ngăn xếp:

```
Procedure Push(Item: DataType);
var TempPtr: PtrType;
begin
  New(TempPtr);                  {xin cấp phát nút mới}
  TempPtr^.Data := Item;         {ghi nhận dữ liệu}
  TempPtr^.Next := StackPtr;     {đẩy vào đỉnh}
  StackPtr := TempPtr;           {ghi nhận lại đỉnh}
end;
```

Thao tác lấy một mục dữ liệu `Item` ra khỏi ngăn xếp được thực hiện theo các bước:

- Kiểm tra ngăn xếp xem có khác rỗng không, nếu ngăn xếp rỗng (`StackPtr = Nil`) thì gán cho `Item` một giá trị đặc biệt (mang tính quy ước, ký hiệu `Nul`) để báo hiệu danh sách rỗng và kết thúc. Trái lại thực hiện tiếp:
- Gán `Item` bằng `StackPtr^.Data`.
- Gán `TempPtr` bằng `StackPtr`.
- Đặt lại `StackPtr^` bằng `StackPtr^.Next`.
- Thu hồi vùng nhớ được trỏ bởi `TempPtr`.

Thủ tục `Pop` dưới đây lấy mục `Item` ra khỏi ngăn xếp:

```
Procedure Pop(var Item: DataType);
var TempPtr: PtrType;
begin
  if StackPtr = Nil then
    Item := Nul                  {báo hiệu ngăn xếp rỗng}
  else
  begin
    Item := StackPtr^.Data;      {ghi nhận Item}
    TempPtr := StackPtr;
    StackPtr := StackPtr.Next;   {ghi nhận lại đỉnh}
    Dispose(TempPtr);            {thu hồi vùng nhớ}
  end;
end;
```

5.2. Hàng đợi liên kết

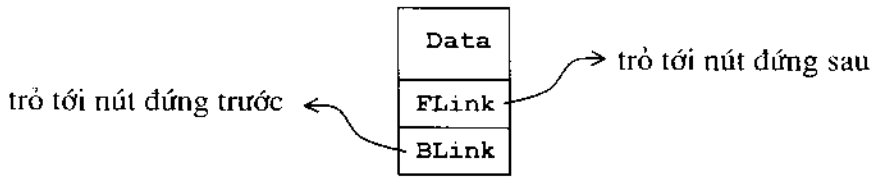
Hàng đợi (*queue*) được cấp phát động giống như một danh sách FIFO. Thao tác đẩy vào, lấy ra được thực hiện ở hai đầu: đẩy vào ở cuối hàng đợi và lấy ra ở đầu hàng đợi. Vì thế ta cần lưu trữ cả hai đầu để thuận tiện cho các thao tác này. Việc khai báo hàng đợi cũng giống như một danh sách liên kết, trong đó ngoài con trỏ trỏ vào đầu hàng đợi, có thêm một con trỏ trỏ vào cuối hàng đợi. Việc đổi một số tên trong khai báo nhằm phù hợp với cách gọi:

```
Type
  DataType = ...
  PtrType = ^QueueNode;
  QueueNode = record
                Data: DataType;
                Next: PtrType;
              end;
Var
  QueueFront,              {trỏ đầu hàng đợi}
  QueueRear: PtrType;      {trỏ cuối hàng đợi}
```

Các thao tác khởi tạo hàng đợi rỗng, duyệt hàng đợi, không có gì thay đổi so với danh sách liên kết. Các thao tác đẩy vào, lấy ra của hàng đợi rất giống các thao tác tương ứng của ngăn xếp. Chúng tôi dành cho các bạn như bài tập.

5.3. Danh sách liên kết đôi

Trong nhiều ứng dụng của danh sách, việc từ một nút đi đến hai nút cạnh nó cũng cần thường xuyên như nhau. Việc chỉ tổ chức một mối liên kết một chiều trong danh sách "tiêu chuẩn" rõ ràng là không đáp ứng tốt được đòi hỏi này. Mục này giới thiệu cách tổ chức danh sách có hai liên kết: Liên kết *cùng chiều* (*forward link*) trỏ đến nút đứng sau và liên kết *ngược chiều* (*backward link*) trỏ đến nút đứng trước.



Một danh sách liên kết tạo nên bởi những nút như thế gọi là *danh sách liên kết đôi*. Việc khai báo một danh sách liên kết đôi cũng giống như một danh sách liên kết "tiêu chuẩn", trong đó bổ sung thêm một mối liên kết ngược chiều. Vì lý do đối xứng, người ta cũng khai báo thêm con trỏ `Last` để trỏ tới nút cuối:

```
Type
  DataType = ...
  PtrType = ^Node;
  Node = record
           Data: DataType;
           FLink,            {trỏ tới nút sau}
           BLink: PtrType;   {trỏ tới nút trước}
```

```
        end;
Var
  First,              {trỏ nút đầu}
  Last: PtrType;      {trỏ nút cuối}
```

Các cài đặt cho các thao tác trên danh sách liên kết đôi được xem như bài tập.

5.4. Danh sách liên kết bội

Một số cài đặt danh sách liên kết dành cho những ứng dụng phức tạp, trong đó các nút cần được thiết kế nhiều hơn một liên kết. Chúng được gọi là những *danh sách liên kết bội*. Một thí dụ đơn giản về danh sách liên kết bội là danh sách có hai liên kết, mô tả hai thứ tự khác nhau cần được truy cập trong danh sách. Thực chất danh sách này có thể xem như hai danh sách liên kết dùng chung vùng dữ liệu. Khai báo dưới đây mô tả một danh sách gồm hai trường dữ liệu chứa tên và số hiệu, trong đó các tên được sắp theo thứ tự abc và các số hiệu được sắp theo thứ tự tăng dần:

```
Type
  IdType = string[10];
  PtrType = ^Node;
  Node = record
           Id: IdType;            {tên}
           Num: integer;          {số hiệu}
           IdLink,                {liên kết tên}
           NumLink: PtrType;      {liên kết số hiệu}
         end;
Var
  IdFirst,              {trỏ tên đầu}
  NumFirst: PtrType;    {trỏ số hiệu đầu}
```

Để tìm hiểu danh sách vừa được thiết kế, bạn hãy vẽ các mối liên kết (bằng các mũi tên) vào bảng dữ liệu sau:

IdLink	Id	Num	NumLink
	Smith	1537	
	Doe	1197	
	Adams	1932	
	Jones	2570	

Chương trình dưới đây tạo danh sách có hai liên kết như vừa mô tả. Dữ liệu được đọc từ file văn bản (file input), mỗi dòng dành 11 vị trí đầu cho tên, các vị trí tiếp theo dành cho số hiệu. Sau khi danh sách được tạo, chương trình ghi danh sách ra hai file văn bản khác (theo quy cách giống file input), một file ghi danh sách theo thứ tự tên còn file kia ghi danh sách theo thứ tự số hiệu. Thuật toán tạo danh sách cho mỗi liên kết giống như trong bài toán xây dựng bảng từ (xem chương trình `TextConcordance`, mục **4.3**)

```
Program MultLink;
```

```
Type
  IdType = string[10];
  PtrType = ^Node;
  Node = record
           Id: IdType;
           Num: integer;
           IdLink, NumLink: PtrType;
         end;
Var
  IdFirst, NumFirst: PtrType;

  procedure CreatListFrom(Name: string);
  {tạo danh sách từ file input có tên Name}
  var
    F: text;
    ITemp: IdType;
    NTemp: integer;
    IdPred, IdCurr, NumPred, NumCurr: PtrType;

    procedure SearchId(ITemp: IdType; var IdPred, IdCurr: PtrType);
    {tìm kiếm vị trí chèn phần tử có tên ITemp}
    begin
      IdPred := Nil; IdCurr := IdFirst;
      while (IdCurr <> Nil) and (IdCurr^.Id < ITemp) do
      begin
        IdPred := IdCurr;
        IdCurr := IdCurr^.IdLink;
      end;
    end;

    procedure SearchNum(NTemp: integer; var NumPred, NumCurr: PtrType);
    {tìm kiếm vị trí chèn phần tử có số hiệu NTemp}
    begin
      NumPred := Nil; NumCurr := NumFirst;
      while (NumCurr <> Nil) and (NumCurr^.Num < NTemp) do
      begin
        NumPred := NumCurr;
        NumCurr := NumCurr^.NumLink;
      end;
    end;

    procedure InsertList(ITemp: IdType; NTemp: integer; var IdPred,
                         IdCurr, NumPred, NumCurr: PtrType);
    {chèn phần tử có tên ITemp, số hiệu NTemp vào danh sách}
    var TempPtr: PtrType;
    begin
      New(TempPtr);
```

```
    with TempPtr^ do
    begin
      Id := ITemp;
      Num := NTemp;
      IdLink := IdCurr;
      NumLink := NumCurr;
    end;
    if IdCurr = IdFirst then IdFirst := TempPtr
      else IdPred^.IdLink := TempPtr;
    if NumCurr = NumFirst then NumFirst := TempPtr
      else NumPred^.NumLink := TempPtr;
  end;

begin
  Assign(F, Name); Reset(F);
  IdFirst := Nil; NumFirst := Nil;
  while not Eof(F) do
  begin
    Readln(F, ITemp, NTemp);
    SearchId(ITemp, IdPred, IdCurr);
    SearchNum(NTemp, NumPred, NumCurr);
    InsertList(ITemp, NTemp, IdPred, IdCurr, NumPred, NumCurr);
  end;
  Close(F);
end;

procedure WriteListTo(IdName, NumName: string);
{ghi danh sách ra các file kết quả có tên IdName và NumName}
var
  F: text;
  Curr: PtrType;
begin
  Assign(F, IdName); Rewrite(F);
  Curr := IdFirst;      {theo thứ tự tên}
  while Curr <> Nil do
  begin
    with Curr^ do
      Writeln(F, Id, ' ':11-Length(Id), Num:5);
    Curr := Curr^.IdLink;
  end;
  Close(F);
  Assign(F, NumName); Rewrite(F);
  Curr := NumFirst;     {theo thứ tự số hiệu}
  while Curr <> Nil do
  begin
    with Curr^ do
      Writeln(f, Id, ' ':11-Length(Id), Num:5);
```

```
      Curr := Curr^.NumLink;
    end;
    Close(F);
  end;

Begin
  CreatListFrom('DS.TXT');
  WriteListTo('DS1.TXT', 'DS2.TXT');
End.
```

Chương trình đọc dữ liệu từ file DS.TXT và ghi kết quả lên các file DS1.TXT (theo thứ tự tên) và DS2.TXT (theo thứ tự số hiệu). Dưới đây là nội dung các file dữ liệu và kết quả khi chạy chương trình:

DS.TXT		DS1.TXT		DS2.TXT	
Smith	1537	Adams	1932	Doe	1197
Doe	1197	Doe	1197	Smith	1537
Adams	1932	Jones	2570	Adams	1932
Jones	2570	Smith	1537	Jones	2570

Có thể thấy danh sách liên kết bội có mặt trong nhiều cài đặt phức tạp khác nhau như trong việc lưu trữ ma trận thưa, đồ thị, cây, ... Các mối liên kết được tổ chức sao cho phải đạt hiệu quả trong các thao tác thường xuyên của ứng dụng. Chẳng hạn trong việc cài đặt ma trận thưa, cần phải xây dựng các mối liên kết để từ một phần tử, ta có thể nhanh chóng đi đến các phần tử cùng hàng hoặc/và cùng cột với nó.

5.5. Danh sách liên kết lồng nhau

Một số danh sách liên kết có trường dữ liệu lại là một danh sách liên kết khác. Khi đó ta có *danh sách liên kết lồng nhau*. Số mức lồng nhau, về lôgic là không hạn chế. Danh sách liên kết lồng nhau thường được dùng để cài đặt những cấu trúc phức hợp, thay đổi nhiều kích thước. Chẳng hạn, một văn bản có thể xem như một danh sách liên kết giữa các dòng (số dòng không hạn chế) , trong đó mỗi dòng lại được xem như một danh sách liên kết giữa các ký tự (số ký tự trong dòng không hạn chế). Khai báo dưới đây xây dựng một danh sách như vậy:

```
Type
  {danh sách liên kết ký tự}
  CharPointer = ^CharNode;
  CharNode = record                    {nút chứa một ký tự của dòng}
               Ch: char;               {nội dung ký tự}
               ChNext: CharPointer;    {trỏ ký tự tiếp theo}
             end;
  {danh sách liên kết dòng}
  LinePointer = ^LineNode;
  LineNode = record                    {nút chứa một dòng của văn bản}
```

```
            ChFirst: CharPointer;    {trỏ ký tự đầu tiên của dòng}
            LineNext: LinePointer;   {trỏ dòng tiếp theo}
          end;
Var
  LineFirst: LinePointer;    {trỏ dòng đầu tiên của văn bản}
```

Những ứng dụng của danh sách lồng nhau (còn được gọi là danh sách tổng quát) rất đa dạng, chẳng hạn có thể gặp chúng trong việc cài đặt các đa thức nhiều ẩn, tập hợp, ma trận thưa, ... Danh sách tổng quát còn là cơ sở cho việc cài đặt ngôn ngữ lập trình LISP (LISt Processing), một ngôn ngữ có nhiều ứng dụng hiện nay. Các bạn có thể tìm hiểu thêm về những vấn đề này qua những tài liệu chuyên khảo về cấu trúc dữ liệu.

5

THƯ VIỆN LẬP TRÌNH

1. Thư viện lập trình và vai trò của thư viện lập trình

Thư viện lập trình là một số đối tượng trong chương trình như kiểu, hằng, biến, các chương trình con, ... được thiết kế sẵn, để có thể phục vụ lâu dài cho các chương trình khác nhau. Chúng thường được tổ chức thành những đơn vị riêng biệt, tùy theo mục đích sử dụng.

Từ những ý nghĩa đó, có thể thấy vai trò của thư viện lập trình qua mấy nét đại thể như sau:

- Cho phép tính kế thừa trong lập trình. Nhờ thư viện, người lập trình có thể sử dụng những thành quả cũ hoặc của những người khác, giảm nhẹ công sức lao động khi lập trình, từ đó kéo theo giảm giá thành của các sản phẩm phần mềm.
- Nâng cao độ tin cậy và ổn định của chương trình. Một chương trình viết theo thư viện sẽ đáng tin cậy hơn nhiều so với một chương trình viết từ A đến Z. Ngoài ra một chương trình viết theo thư viện sẽ dễ tìm lỗi để hiệu chỉnh hay nâng cấp.
- Nâng cao tính lôgic của công việc lập trình. Nhờ thư viện, người lập trình không phải bận tâm nhiều đến tính vật lý của thiết bị. Một chương trình viết theo thư viện sẽ dễ chuyển đổi từ hệ máy này sang hệ máy khác.

Cũng vì thế, thư viện lập trình là một công cụ không thể thiếu cho những phát triển ứng dụng của máy tính vốn đã rất đa dạng hiện nay.

Các cài đặt của những ngôn ngữ lập trình rất chú ý đến việc xây dựng và phát triển thư viện. Một mặt, chúng tổ chức những biện pháp kỹ thuật để người dùng có thể dễ dàng xây dựng thư viện của riêng mình, mặt khác, chúng cung cấp thêm những thư viện để người dùng khai thác. Ngoài những thư viện do các công cụ lập trình cung cấp, người dùng còn tìm thấy các thư viện ở những "gói" chương trình, do những người khác viết, hoặc dưới dạng những thương phẩm. Hiện nay có rất nhiều thư viện có tính chất chuyên biệt: Thư viện toán học, thư viện đồ họa, thư viện thiết kế (cơ khí, xây dựng ...), ... Kỹ thuật xây

dựng, phát triển và khai thác thư viện là một vấn đề quan trọng không thể thiếu được trong kỹ thuật lập trình.

Chương này nhằm thảo luận các khả năng xây dựng và khai thác thư viện trong Turbo Pascal, tuy nhiên nhiều vấn đề được mở rộng một cách tự nhiên cho các ngôn ngữ khác.

2. Unit (đơn vị chương trình)

Unit là một công cụ của Turbo Pascal nhằm giúp người lập trình xây dựng và phát triển các thư viện của mình.

2.1. Định nghĩa unit và câu lệnh liên kết

Unit là một tập hợp các kiểu, hằng, biến, hàm, thủ tục nào đấy được biên dịch sẵn để có thể liên kết với mọi chương trình.

Mỗi unit được tổ chức thành một file (dạng mã máy) lưu trên đĩa với phần mở rộng là .TPU (Turbo Pascal Unit). Các unit được phân biệt với nhau bởi tên. Tên unit được đặt theo quy tắc tên thông thường.

Giả sử đã có một số unit nào đó (các file đã được lưu trên đĩa để hệ thống có thể tìm thấy), một chương trình muốn liên kết với những unit này phải dùng lệnh khai báo có cú pháp:

```
USES unit1, unit2, ..., unitN;
```

trong đó `USES` là từ khóa, `unit1, unit2, ..., unitN` là dãy tên các unit (viết cách nhau bởi dấu phẩy và kết thúc bởi dấu chấm phẩy) cần liên kết với chương trình. Lệnh khai báo này phải là lệnh đầu tiên trong các lệnh khai báo khác (nghĩa là nó chỉ viết sau phần tiêu đề chương trình). Thí dụ các dòng lệnh:

```
Program Demo;
Uses Crt, Dos, Graph;
```

khai báo chương trình `Demo` được liên kết với các unit `Crt, Dos, Graph`.

Một chương trình sẽ dùng được tất cả những đối tượng đã được xây dựng trong các unit liên kết với nó như thể chúng được viết trong chương trình. Nếu gặp một đối tượng lạ, chưa được liên kết, sẽ gặp lỗi *Unknown identifier* khi biên dịch. Chẳng hạn, chương trình `Demo` ở trên có thể dùng các thủ tục `ClrScr` (unit `Crt`), kiểu `Registers` (unit `Dos`), hàm `GetColor` (unit `Graph`).

Một chương trình liên kết với những unit nào đấy còn được gọi là tham khảo (reference) đến những unit này.

2.2. Cách viết một unit

Viết một unit cũng như viết một chương trình, cần phải qua các bước: soạn thảo nguồn rồi sau đó dịch.

Nguồn của unit được soạn thảo như một file văn bản và được cất lên đĩa với phần mở rộng .PAS giống như nguồn của chương trình. Unit gồm những phần sau:

1. Tiêu đề unit: Gồm một dòng, mở đầu là từ khóa UNIT tiếp theo là tên unit và kết thúc là dấu chấm phẩy. Tên unit được đặt theo quy tắc như tên các đối tượng khác. Điểm đặc biệt là tên của unit phải trùng với tên file của nó (điều này là không bắt buộc khi soạn chương trình). Thí dụ:

```
UNIT Mat;
```

là tiêu đề của unit có tên là MAT. Khi cất nó lên đĩa, cũng phải đặt tên file là MAT (nghĩa là file MAT.PAS).

2. Phần giao tiếp: Bắt đầu bằng từ khóa INTERFACE. Phần này khai báo các kiểu, hằng, biến, hàm, thủ tục mà chương trình liên kết được dùng. Riêng các thủ tục, hàm chỉ khai báo phần tiêu đề. Thí dụ:

```
INTERFACE
  Type
    Matrix = record
               size: byte;
               elem: aray[1..50, 1..50] of real;
             end;
  function Invert(var A: Mattrix): boolean;
  procedure Show(var A: Matrix);
```

trình bày phần giao tiếp của unit `Mat`. Mọi chương trình tham khảo tới unit này đều được dùng kiểu `Matrix` , hàm `Invert` và thủ tục `Show` như trong unit đã xây dựng.

3. Phần nội tại: Bắt đầu bằng từ khóa IMPLEMENTATION. Trong phần này là nội dung các hàm, thủ tục đã khai báo trong phần giao tiếp. Thí dụ:

```
IMPLEMENTATION
  function Invert(var A: Mattrix): boolean;
  .....
  begin
    .....
  end;
  procedure Show(var A: Mattrix);
  .....
  begin
    .....
  end;
```

là phần nội tại của unit `Mat`, trong đó trình bày các lệnh xây dựng hàm `Invert` và thủ tục `Show`.

Chú ý rằng, tất cả các thủ tục, hàm đã khai báo trong phần giao tiếp, cần phải được xuất hiện trong phần này. Ngược lại, phần này có thể chứa các kiểu, hằng, biến, hàm, thủ tục không có mặt trong phần giao tiếp. Chúng là những đối tượng nội tại của unit, chương trình liên kết không dùng được. Những đối tượng không được trực tiếp truy cập bởi chương trình bên ngoài, cần đưa vào phần này để đảm bảo an toàn. Thông thường, những

đối tượng này là những thao tác hoặc tài nguyên nhằm hỗ trợ cho việc xây dựng các thủ tục và hàm trong phần giao tiếp.

4. Phần khởi động: Đây là phần các câu lệnh của unit, chúng được thực hiện trước khi thực hiện các lệnh của chương trình. Phần này mở đầu bằng từ khóa BEGIN và kết thúc bằng END kèm theo dấu chấm (như kết thúc của chương trình). Thường phần này là các lệnh khởi gán các giá trị ban đầu cho các biến hoặc lời gọi những thủ tục khởi tạo nào đấy để trao cho chương trình trước khi chương trình hoạt động. Vì thế phần khởi động là không bắt buộc. Nếu không có phần này, từ khóa BEGIN có thể bỏ, nhưng vẫn phải giữ lại từ khóa END cùng dấu chấm (báo hiệu hết unit).

Kết hợp các thí dụ đã nêu, toàn bộ văn bản của unit `Mat` có dạng như sau:

```
UNIT Mat;   {tiêu đề}
INTERFACE   {phần giao tiếp}
  Type
    Matrix = record
               size: byte;
               elem: aray[1..50, 1..50] of real;
             end;
  function Invert(var A: Mattrix): boolean;
  procedure Show(var A: Matrix);
IMPLEMENTATION   {phần nội tại}
  function Invert(var A: Mattrix): boolean;
  .....
  begin
    .....
  end;
  procedure Show(var A: Mattrix);
  .....
  begin
    .....
  end;
  {không có phần khởi động}
END.
```

Một unit, khi xây dựng, lại có thể tham khảo đến những unit khác. Câu lệnh khai báo USES được dùng sau các từ khóa INTERFACE hoặc IMPLEMENTATION, để có thể dùng các đối tượng của các unit được tham khảo trong những phần này. Thí dụ

```
UNIT Exam;
INTERFACE
Uses Dos, Crt, Mat;
  .....
IMPLEMENTATION
  .....
END.
```

mô tả một unit mà nội dung phần INTERFACE của nó được tham khảo đến các unit `Dos`, `Crt`, `Mat`.

Sau khi đã soạn xong nguồn của unit, bạn cần dịch nó ra đĩa theo cách thức như đối với chương trình. Turbo Pascal sẽ lưu ra đĩa bản dịch với tên giống như file nguồn và có phần mở rộng là .TPU (Turbo Pascal Unit). Nếu gặp lỗi biên dịch, hệ thống sẽ báo lỗi giống như khi dịch một chương trình.

Khi dịch một chương trình có liên kết với các unit, chỉ cần các file TPU của các unit đó, nghĩa là chỉ cần trao các TPU cho người sử dụng. Theo mặc định, khi dịch chương trình, hệ thống sẽ tìm các TPU có liên quan ở trong cùng một thư mục với chương trình nguồn, tuy nhiên các TPU có thể để ở chỗ khác với điều kiện phải đặt lại đường dẫn trong mục Options của hệ thống. Để tiện lợi cho người sử dụng, khi dịch chương trình (bằng Make, Build hoặc Run), trong trường hợp không tìm thấy các TPU, Turbo Pascal sẽ tự động dịch các file PAS liên quan ra các file TPU trước, rồi tiến hành dịch và thực hiện chương trình sau. Các file EXE được tạo đã bao gồm các mã của những unit được liên kết, vì thế khi chạy không cần có mặt các file TPU. Cũng giống như chương trình, các file PAS, nguồn của unit, là cần thiết để bảo trì thư viện. Mỗi khi muốn thay đổi nội dung thư viện, cần hiệu chỉnh lại nguồn và biên dịch lại.

Một unit được dùng, bao giờ cũng phải có các tài liệu kèm theo. Có hai loại tài liệu cho unit:

1. Tài liệu sử dụng: Đây là tài liệu dành cho người dùng, trong đó giới thiệu các chức năng chính của unit, ý nghĩa các kiểu dữ liệu, biến, hằng, các chức năng cũng như cú pháp khai báo của các hàm, thủ tục. Vì người dùng chỉ có trong tay các file TPU nên sẽ chẳng làm gì được nếu không có những tài liệu này. Một số thư viện lớn còn cung cấp thêm các tài liệu học tập, các chương trình mẫu để người dùng thuận tiện khi khai thác. Những thương phẩm thư viện hiện nay rất chú trọng biên soạn tài liệu cho người dùng và tiền tài liệu được tính chung với giá thành sản phẩm. Nhiều tài liệu còn đưa ra những khuyến cáo cho người dùng, chỉ rõ những mặt còn hạn chế của thư viện cũng như hướng phát triển trong tương lai của nó.

Unit `Mat` được xây dựng ở trên là một thí dụ đơn giản, chỉ bao gồm kiểu dữ liệu `Matrix` để mô tả một ma trận, hàm `Invert` để tính ma trận nghịch đảo và thủ tục `Show` để hiển thị ma trận lên màn hình. Kiểu `Matrix` gồm 2 trường: `size` (kiểu `byte`) để lưu cấp của ma trận (không quá 50) và `elem` (kiểu `array[1..50, 1..50] of real`) để lưu các phần tử của ma trận. Hàm `Invert(var A: Matrix): boolean` tính ma trận nghịch đảo của ma trận `A`. Nếu `A` có nghịch đảo, hàm trả về `True` và kết quả nghịch đảo được gửi lại vào `A`, trái lại hàm trả về `False`. Thủ tục `Show(var A: Matrix)` hiển thị nội dung ma trận `A` lên màn hình theo khuôn dạng dòng, cột quen thuộc.

2. Tài liệu thiết kế: Đây là tài liệu dành cho người viết unit, nhằm bảo trì những thư viện đã xây dựng. Một unit được viết xong, không phải là đã được hoàn thiện. Nhiều lỗi có thể được phát hiện trong quá trình dùng. Ngoài ra, luôn xuất hiện những vấn đề cần nâng cấp cho thư viện. Vì thế việc hiệu chỉnh, thậm chí viết lại, là những vấn đề luôn được đặt ra trong quá trình bảo trì thư viện. Nội dung của tài liệu thiết kế chủ yếu xoay quanh việc tổ chức các cấu trúc dữ liệu và các giải thuật được dùng, trong đó có đánh giá

các phí tổn cũng như hiệu quả. Một số hạn chế cũng như một số hoạch định cho tương lai của thư viện cũng được đề cập trong các tài liệu này. Nhờ các tài liệu thiết kế mà những người xây dựng thư viện tiết kiệm được rất nhiều công sức khi cần hiệu chỉnh hoặc phát triển sản phẩm của mình. Thông thường những tài liệu thiết kế là những tài liệu được giữ bản quyền, không được công bố cho người sử dụng.

Chẳng hạn, đối với unit `Mat` ở trên, tài liệu thiết kế cần chỉ rõ việc tổ chức dữ liệu kiểu `Matrix` để lưu ma trận và giải thuật được dùng để tính ma trận nghịch đảo. Nhờ tài liệu này, nếu cần thay đổi lại tổ chức dữ liệu hoặc thay đổi giải thuật, người thiết kế sẽ dễ dàng hiệu chỉnh lại được nội dung unit.

Việc viết một unit cho đúng quy cách cú pháp là một điều đơn giản, nhưng việc quyết định các nội dung của nó lại không đơn giản chút nào. Thông thường, một unit được xây dựng nhằm cung cấp các thao tác cơ bản (dưới dạng hàm, thủ tục) và những tài nguyên (dưới dạng cấu trúc dữ liệu) cho một lớp các bài toán nào đấy. Như thế việc hình thành một unit là hệ quả của việc phân tích cặn kẽ những yêu cầu cần giải quyết. Kỹ thuật viết unit bao gồm nhiều mặt, không những cần xây dựng những giải thuật và tổ chức dữ liệu sao cho phù hợp và có hiệu quả, mà còn phải để ý đến cách tổ chức các tham số và cách đặt tên để người đọc dễ nhớ, dễ dùng. Chẳng hạn cần tránh việc tổ chức quá nhiều tham số cho một thủ tục hoặc hàm bằng cách kết hợp các tham số liên quan nhờ các cấu trúc dữ liệu. Các tên kiểu, hàm, thủ tục phải mang tính chất gợi nhớ và phản ánh được mối liên hệ giữa chúng, ... Cũng cần nhớ rằng phạm vi ảnh hưởng của một unit là lớn hơn nhiều so với một chương trình, một sai sót nhỏ của unit có thể dẫn đến việc đổ vỡ hàng loạt các ứng dụng viết trên chúng và ngược lại một cải tiến của unit sẽ kéo theo hàng loạt chương trình được nâng cấp. Vì thế việc hoàn thiện các unit luôn là một yêu cầu cấp thiết trong công việc lập trình.

2.3. Cách dùng unit

Các đối tượng đã xây dựng trong unit có thể dùng trong các chương trình liên kết như chính các đối tượng này được khai báo trong chương trình. Chẳng hạn, chương trình trong thí dụ dưới đây, liên kết với unit `Mat`, sẽ thực hiện các công việc: Nhập một ma trận từ bàn phím, hiển thị ma trận đã nhập lên màn hình, tính ma trận nghịch đảo của ma trận này và hiển thị ma trận nghịch đảo tìm được (nếu có) lên màn hình:

```
Program Thidu;
Uses Mat;
Var A: Matrix;
    i, j: byte;
Begin
  with A do     {nhập ma trận A từ bàn phím}
  begin
    Writeln('Cap cua ma tran: '); Readln(size);
```

```
    Writeln('Vao cac phan tu cua ma tran:');
    for i := 1 to size do
      for j := 1 to size do
      begin
        Write('Phan tu dong ', i, ', cot ', j, ': ');
        Readln(elem[i, j]);
      end;
  end;
  Writeln('Ma tran da cho:');
  Show(A);
  if Invert(A) then
  begin
    Writeln('Ma tran nghich dao la:');
    Show(A);
  end else
    Writeln('Ma tran khong kha nghich');
  Write('An Enter de ket thuc ... ');
  Readln;
End.
```

Vì một chương trình có thể tham khảo tới nhiều unit nên tùy theo mối quan hệ giữa chúng mà có thể xảy ra những hiệu ứng khác nhau. Dưới đây là mấy điểm cần chú ý:

1. Một unit có thể tham khảo đến unit khác trong danh sách nhưng không được tham khảo vòng. Thí dụ chương trình:

```
  Program Test;
  Uses A, B, C;
  ............
```

trong đó unit A tham khảo tới B, unit B tham khảo tới C còn unit C lại tham khảo tới A, khi chạy sẽ gặp lỗi *Circular unit reference*.

2. Cho phép có tên trùng nhau trong các đối tượng được khai báo trong phần giao tiếp của các unit và trong chương trình tham khảo chúng. Trong những trường hợp như vậy, Turbo Pascal sẽ ưu tiên định nghĩa sau cùng. Trình tự định nghĩa tên tuân theo thứ tự các unit trong danh sách và cuối cùng là chương trình tham khảo đến chúng. Thí dụ dưới đây là hai unit A, B và chương trình AB tham khảo tới các unit này:

```
Unit A;
Interface
  Var x: integer;
  Function F(x: integer): integer;
Implementation
  Function F(x: integer): integer;
  begin
    F := x+1;
  end;
Begin
  x := 1;
End.
```

```
Unit B;
Interface
  Var x: integer;
  Function F(x: integer): integer;
Implementation
  Function F(x: integer): integer;
  begin
    F := x+2;
  end;
Begin
  x := 2;
End.

Program AB;
Uses A, B;
  Function F(x: integer): integer;
  begin
    F := x+3;
  end;
Begin
  Writeln('x = ', x);
  Writeln('F(x) = ', F(x));
  Readln;
End.
```

Trong thí dụ trên, tên biến `x` xuất hiện trong cả hai unit `A` và `B`, còn tên hàm `F` xuất hiện trong cả hai unit này và cả trong chương trình `AB`. Thứ tự tham khảo tới các unit `A`, `B` của chương trình `AB` là `A` trước `B` sau. Vì thế cả `x` và `F` đều được hiểu trong `AB` với những định nghĩa được xác định sau cùng. Với biến `x`, định nghĩa sau cùng thuộc về `B`, nghĩa là `x` có giá trị 2, còn đối với hàm `F`, định nghĩa sau cùng thuộc về `AB`, nghĩa là `F(x)` có giá trị 5.

Có thể phân biệt các tên trùng nhau này nếu chỉ tường minh chúng thuộc unit nào bằng cách viết tên unit trước, sau đó là dấu chấm và cuối cùng là tên đối tượng cần truy cập. Chẳng hạn, nếu viết lại các lệnh trong thân chương trình `AB` như dưới đây:

```
Program AB;
Uses A, B;
  Function F(x: integer): integer;
  begin
    F := x+3;
  end;
Begin
  Writeln('x = ', A.x);
  Writeln('F(x) = ', B.F(A.x));
  Readln;
End.
```

thì sẽ nhận được kết quả trên màn hình `x = 1` (`x` của `A`) và `F(x) = 3` (`F` của `B`).

Thí dụ trên cũng cảnh báo một tình huống có thể xảy ra là, một chương trình có thể phá hỏng unit được tham khảo, nếu như trong unit không che giấu những đối tượng không được truy cập trực tiếp bởi chương trình.

Việc dùng một unit không chỉ đơn thuần là nắm được cú pháp những lời gọi thủ tục hoặc hàm trong unit mà còn phải biết kết hợp chúng để thực hiện được mục đích của chương trình. Thông thường các đối tượng trong một unit có liên quan mật thiết đến nhau vì chúng được thiết kế để giải một lớp bài toán nào đấy. Việc nắm được những mối liên hệ này giúp cho việc hiểu unit được hệ thống và toàn diện hơn và như thế việc khai thác unit sẽ hiệu quả hơn.

Một biện pháp tốt để tìm hiểu các unit là tự viết các chương trình để thử chúng. Bất cứ một sự mơ hồ nào trong tài liệu sử dụng cũng cần được khẳng định lại bằng những chương trình test tương ứng. Nhờ kết quả của những khẳng định này mà việc dùng unit mới tránh được những lỗi "khó hiểu" về sau.

Nếu bạn dùng những unit dưới dạng thương phẩm để triển khai các ứng dụng của mình thì bạn cần theo dõi việc nâng cấp chúng để luôn có được trong tay phiên bản mới, nhất là trong những giai đoạn các máy tính được chuyển đổi thế hệ. Việc thay đổi thường xuyên này sẽ giúp cho các ứng dụng của bạn luôn thích nghi được với hoàn cảnh hiện tại.

Nói tóm lại, việc khai thác hiệu quả một unit là một việc làm công phu, đòi hỏi nhiều thời gian cho việc đọc tài liệu và làm máy. Việc làm quen với nhiều unit sẽ cung cấp cho bạn nhiều công cụ để phát triển ứng dụng, đồng thời cũng là bước khởi đầu để giúp bạn trở thành một nhà thiết kế thư viện tốt.

2.4. Những lý do của việc dùng unit

Unit là một bước mới của Turbo Pascal trong quá trình phát triển, được xây dựng bắt đầu từ phiên bản 4. Từ phiên bản 3 trở về trước, thư viện lập trình của Turbo Pascal được lưu trữ dưới dạng nguồn và được "nhúng" vào văn bản chương trình khi biên dịch.

Những lý do của việc dùng unit để tổ chức thư viện, có thể tóm tắt trong mấy điểm sau:

1. Unit là công cụ để tạo các chương trình lớn.

Do mỗi đơn thể chương trình phải được quản lý gọn trong một đoạn nên kích thước của một đơn thể không thể vượt ngưỡng 64 Kb. Nếu đưa tất cả các thủ tục, hàm phục vụ chương trình vào trong cùng một văn bản nguồn thì sẽ dẫn đến chương trình quá lớn. Đây cũng là lời giải thích tại sao phiên bản 3 của Turbo Pascal chỉ tạo được những file COM. Nhờ các unit, các thường trình được phân bố trên những đơn thể khác nhau, nghĩa là chương trình được liên kết từ nhiều đơn thể, có tổng kích thước vượt ngưỡng 64 Kb nhiều. Từ phiên bản 4 trở đi, nhờ unit, Turbo Pascal mới tạo ra được các file EXE.

2. Unit có khả năng che giấu nội tại.

Các chương trình bên ngoài chỉ có thể dùng các kết quả của các thường trình trong unit mà không thể truy cập được nội dung thiết kế của các thường trình này. Việc che giấu như vậy làm tăng độ an toàn của việc dùng thư viện, cũng như tăng khả năng trao đổi thư viện

giữa các người dùng. Đối với những người sử dụng thư viện, việc lưu trữ các TPU sẽ thuận tiện và đảm bảo hơn nhiều so với việc phải lưu trữ các văn bản nguồn.

3. Unit làm tăng tốc độ dịch chương trình.

Nhờ unit mà chương trình nguồn ngắn hơn nhiều. Khi dịch lại, chỉ có những lệnh trong chương trình nguồn mới được dịch, điều này kéo theo tốc độ dịch sẽ nhanh hơn.

2.5. Thí dụ xây dựng unit

Phần này trình bày các bước xây dựng một unit thông qua một bài toán cụ thể: thực hiện các phép toán cơ bản trên các đa thức gồm cộng, nhân và chia hai đa thức. Unit được đặt tên là POLY.

Trước hết, cần tổ chức kiểu dữ liệu để lưu trữ đa thức. Theo định nghĩa, một đa thức bậc n có dạng:

$$C_0x^n + C_1x^{n-1} + ... + C_n$$

trong đó $C_0, C_1, ..., C_n$ là các hằng số, $C_0 \neq 0$, được gọi là các hệ số của đa thức. Một cách tự nhiên, có thể cài đặt kiểu đa thức (với hạn chế bậc không quá 100) như sau:

```
Type
  Polynom = record
              deg: integer;
              coef: array[0..100] of real;
            end;
```

trong đó `deg` lưu trữ bậc của đa thức và mảng `coef` lưu trữ các hệ số của đa thức: `coef[0]`, `coef[1]`, ..., `coef[deg]` theo trình tự bậc giảm dần, `coef[0]` $\neq$ 0.

Để định nghĩa trên áp dụng cho cả đa thức 0 (có tất cả các hệ số đều bằng 0), ta quy ước rằng bậc của đa thức 0 là -1 (bậc không xác định).

Các phép toán trên đa thức được thực hiện nhờ các thủ tục tương ứng dưới đây:

Thủ tục

```
Procedure Add(a, b: Polynom; var c: Polynom);
```

thực hiện việc cộng đa thức `a` với đa thức `b` và gửi kết quả vào `c`.

Thủ tục

```
Procedure Multiply(a, b: Polynom; var c: Polynom);
```

thực hiện việc nhân đa thức `a` với đa thức `b` và gửi kết quả vào `c`.

Thủ tục

```
Procedure Divide(a, b: Polynom; var q, r: Polynom);
```

thực hiện việc chia đa thức `a` cho đa thức `b` (khác đa thức 0). Kết quả được gửi vào `q` (đa thức thương) và `r` (đa thức dư).

Như vậy phần INTEFACE của unit đã được thiết kế.

Phần IMPLEMENTATION là phần chi tiết của unit, trong đó phải thực hiện các thủ tục đã nêu ở phần giao tiếp. Ở đây, nếu cần có thể xây dựng thêm những hàm, thủ tục hỗ trợ cho các thủ tục này. Chúng tôi dành phần này cho các bạn, xem như bài tập.

Unit này không có phần khởi động.

Cuối cùng là việc biên tập những gì đã thiết kế theo quy cách của unit vào file có tên là POLY.PAS và dịch nó sang TPU.

Với unit POLY vừa xây dựng, bạn có thể viết một chương trình test nó như sau:

```
Program TestPoly;
Uses Crt, Poly;
Var
  a, b, c, d: Polynom;
  ch: char;

  Procedure Enter(var p: Polynom);
  {nhập đa thức p từ bàn phím, hệ số bậc cao nhất khác 0}
  var i: integer;
  begin
    with p do
    begin
      Write('  Bac (khong qua 100): ');
      Readln(deg);
      if (deg < 0) or (deg > 100) then
      begin
        deg := -1;
        exit;
      end;
      repeat
        Write('  He so bac ', deg, ' (khac 0): ');
        Readln(coef[0]);
      until coef[0] <> 0;
      for i := 1 to deg do
      begin
        Write('  He so bac ', deg-i, ': ');
        Readln(coef[i]);
      end;
    end;
  end;

  procedure Show(var p: Polynom);
  {hiển thị đa thức p lên màn hình}
  var i: integer;
  begin
    with p do
    begin
      if deg = -1 then Writeln('  0') else
      for i := 0 to deg do
        Writeln('  He so bac ', deg-i, ': ', coef[i]:1:2);
    end;
  end;
```

```
Begin
  ClrScr;
  Writeln('Nhap da thuc a:'); Enter(a);
  Writeln('Nhap da thuc b:'); Enter(b);
  repeat
    Writeln;
    Write('1. Cong / 2. Nhan / 3. Chia / 4. Ket thuc: ');
    ch := ReadKey; Writeln(ch);
    case ch of
      '1': begin
             Add(a, b, c);
             Writeln('Da thuc tong:');
             Show(c);
           end;
      '2': begin
             Multiply(a, b, c);
             Writeln('Da thuc tich:');
             Show(c);
           end;
      '3': begin
             Divide(a, b, c, d);
             Writeln('Da thuc thuong:');
             Show(c);
             Writeln('Da thuc du:');
             Show(d);
           end;
      '4': Halt;
    end;
  until false;
End.
```

3. Thư viện chuẩn của Turbo Pascal

Cũng giống như mọi cài đặt của ngôn ngữ lập trình, Turbo Pascal xây dựng sẵn một số thư viện cho người dùng, gọi là những thư viện chuẩn. Những unit cơ bản nhất được "gói" chung vào một file có tên là TURBO.TPL (Turbo Pascal Library), những unit khác được tổ chức thành những TPU riêng biệt.

Một số unit chuẩn sẽ được bàn kỹ trong chương sau, ở đây chỉ điểm qua những nét chính.

3.1. Các unit trong TURBO.TPL

TURBO.TPL chứa 5 unit cơ bản là: System, Crt, Dos, Printer và Overlay. Khi cài đặt, TURBO.TPL luôn ở cùng một nơi với TURBO.EXE (ngầm định trong thư mục BIN của hệ thống) để khi TURBO được gọi, TURBO.TPL cũng được tải vào bộ nhớ và các unit

chứa trong nó sẵn sàng được tham khảo. Vì được chứa trong bộ nhớ nên tốc độ tìm đến các unit này sẽ nhanh hơn các unit chứa trên đĩa. Turbo Pascal luôn ưu tiên tìm các unit cần thiết trong TURBO.TPL trước khi tìm chúng ở bên ngoài. Mặc dù ngầm định chứa 5 unit, nhưng người dùng có thể tùy ý thêm bớt danh sách này bằng cách dùng trình tiện ích TPUMOVER mà hệ thống Turbo Pascal cung cấp (xem mục **4** cùng chương).

Unit System

Đây là unit cơ bản nhất, nó chứa các tiện nghi thông dụng như các hàm sơ cấp, các thủ tục vào, ra, ... Nó là unit duy nhất tự động liên kết với mọi chương trình. Vì thế việc dùng unit này không cần câu lệnh khai báo USES.

Unit Crt

Unit Crt bao gồm các chức năng khai thác màn hình, bàn phím và âm thanh. Nó thường được dùng để thiết kế giao diện của chương trình nhờ việc điều khiển màn hình và bàn phím. Nó cũng là công cụ hỗ trợ cho việc xây dựng các chương trình mô phỏng.

Unit Dos

Unit này bao gồm các dịch vụ của Dos như quản lý file, đĩa, thiết lập ngày giờ., ... Ngoài ra có hai chương trình mức thấp là MsDos và Intr cho phép gọi trực tiếp một lệnh Dos hoặc một ngắt (interrupt). Unit Dos là công cụ thích hợp để lập trình hệ thống.

Unit Printer

Đây là unit ngắn nhất, đơn thuần nó chỉ định nghĩa tên cho máy in là LST và mọi kết xuất hướng tới tên này sẽ được trực tiếp gửi ra máy in.

Unit Overlay

Là unit cho phép tổ chức chạy các chương trình lớn khi mà bộ nhớ trong không đủ tải toàn bộ code của chương trình. Unit Overlay xây dựng các thao tác cho phép tải từng phần của chương trình vào bộ nhớ trong để chạy (gọi là chế độ overlay).

3.2. Các unit khác

Các unit còn lại của TURBO, mỗi unit được chứa trên một TPU riêng biệt. Lúc cài đặt, các unit này ngầm định được đặt trong thư mục UNITS của hệ thống.

Unit Graph

Là unit khai thác chế độ đồ họa của máy tính bao gồm một số các kiểu, thủ tục, hàm phục vụ cho các thao tác vẽ, tô mầu, xử lý bitmap, ... Trong unit có cài đặt bộ xử lý Graphics tùy thuộc thiết bị màn hình CGA, EGA, VGA, Hercules, AT&T400, MCCGA, 3270PC, ... Đây là một unit lớn, có nhiều hàm và thủ tục. Unit Graph là công cụ để xây dựng các chương trình đồ họa.

Unit Graph3, Turbo3

Là những unit nhằm tương thích với các chương trình viết trên Turbo Pascal version 3. Những unit này mang tính lịch sử trong thời kỳ chuyển tiếp từ phiên bản 3 sang các phiên bản cao hơn, hiện nay chúng trở nên không cần thiết.

Các unit của Turbo Vision

Bao gồm nhiều TPU, thiết kế theo hướng đối tượng, nhằm hỗ trợ khả năng giao tiếp của chương trình qua các công cụ như menu, khung hội thoại, cửa sổ, chuột, ... giống như môi trường của Turbo Pascal.

4. Trình tiện ích TPUMOVER

Trong các phát triển ứng dụng của Turbo Pascal, một chương trình thường cần tham khảo đến nhiều unit khác nhau. Việc để riêng rẽ chúng thành những TPU riêng biệt nhiều khi gây khó khăn cho việc quản lý và cài đặt. Turbo Pascal cung cấp một trình tiện ích tên là TPUMOVER (tên file có phần mở rộng là .EXE) để người dùng có thể tổ chức các unit thành những "gói", có thể thêm hoặc bớt dễ dàng.

Nhờ TPUMOVER, các TPU có thể gộp lại thành một file với một tên nào đấy, có phần mở rộng mặc định là .TPL (Turbo Pascal Library), khi cần có thể thêm, bớt hoặc trích các TPU từ TPL. Nói riêng, nhờ TPUMOVER, bạn có thể thêm bớt các unit trong file TURBO.TPL là thư viện luôn được mở kèm với TURBO.EXE, nhằm nhanh chóng phục vụ các ứng dụng của bạn. Cú pháp của TPUMOVER là:

TPUMOVER *filename operations*

trong đó *filename* là tên file TPL còn *operations* là một danh sách tùy chọn gồm một hay nhiều lệnh có dạng như sau:

+*unitname*	thêm unit có tên *unitname*
-*unitname*	bớt unit có tên *unitname* (unit bớt sẽ bị xóa)
**unitname*	trích unit có tên *unitname* (unit trích sẽ được sao ra ngoài)

(các *unitname* được ngầm định có phần mở rộng là .TPU).

Nếu thành phần *operations* không có mặt thì TPUMOVER sẽ hiển thị tên các unit hiện có trong *filename* ra màn hình.

Thí dụ bạn đang có ba unit là A.TPU, B.TPU, C.TPU (giả thiết chúng đang ở thư mục hiện hành cùng với TPUMOVER.EXE). Khi đó lệnh

TPUMOVER abc +a +b +c

sẽ tạo ra file ABC.TPL chứa các unit A, B, C.

Bây giờ đưa vào lệnh

TPUMOVER abc *c

nội dung của ABC.TPL vẫn như cũ, trong thư mục xuất hiện C.TPU vì C đã được sao ra ngoài. Đưa tiếp lệnh

TPUMOVER abc -a

nội dung của ABC.TPL lúc này chỉ còn hai unit B và C, trong thư mục không có A.TPU vì A đã bị xóa. Tại mỗi bước thử, bạn có thể kiểm tra lại nội dung của ABC.TPL bằng lệnh

TPUMOVER abc

6

MỘT SỐ UNIT CHUẨN

Chương này tìm hiểu một số unit chuẩn của Turbo Pascal với hai mục đích:

- Biết cách tiếp cận một thư viện để việc khai thác có hiệu quả.
- Học tập cách thiết kế một thư viện cho các phát triển ứng dụng.

Các unit được chọn là CRT, GRAPH và DOS vì chúng là những unit thông dụng (unit SYSTEM quá quen thuộc và đã được dùng trong suốt giáo trình, unit OVERLAY chỉ dùng khi cần tổ chức chạy những chương trình lớn). Một số các unit thuộc Turbo Vision cũng không được đề cập đến (chúng dành cho việc thiết kế hướng đối tượng, chưa có dịp bàn đến trong cuốn sách này). Mặc dù vậy, sau khi đã nắm vững các unit đã nêu, các bạn có thể tự tìm hiểu các unit khác theo các tài liệu trợ giúp của hệ thống.

Với mục đích đã đề ra, mỗi unit đều được trình bày thông qua các chức năng của nó, trong đó các thủ tục, hàm được phân thành từng nhóm. Thông thường, để thực hiện một yêu cầu của việc triển khai ứng dụng, cần một tổ hợp các hàm, thủ tục có liên quan đến nhau. Việc nắm vững ý nghĩa cách dùng cũng như các mối liên hệ giữa các thủ tục, hàm giúp bạn khai thác được triệt để chúng, ngoài ra bạn còn phát hiện được những điểm cần bổ sung cho các ứng dụng của mình.

Trong mỗi unit, chúng tôi chỉ nhấn mạnh những điểm cơ bản nhất, vì thế các thủ tục và hàm cũng không được giới thiệu đầy đủ, khi cần, các bạn có thể xem thêm trong các tài liệu trợ giúp của Turbo Pascal.

Cách tốt nhất để tiếp cận chương này là bạn làm việc luôn trên máy. Bạn có thể đưa vào máy những thí dụ đã nêu ra trong chương để chạy thử, qua đó bạn có thể thay đổi, phát triển những thí dụ này để hiểu rõ hơn thư viện đang dùng. Mặt khác, căn cứ vào những mục đích ứng dụng của riêng mình, bạn có thể xây dựng thêm các thư viện dựa vào nguồn tài nguyên mà Turbo Pascal đã cung cấp.

Bên cạnh việc khai thác những thư viện này một cách hiệu quả, bạn cũng cần để ý học tập cách viết thư viện của Turbo Pascal, từ cách xây dựng các chức năng của các hàm, thủ tục đến cách đặt tên, tổ chức tham số, xây dựng các kiểu dữ liệu, cũng như cách viết tài liệu sử dụng, ... Bạn cũng cần tận dụng những tiện lợi mà môi trường Turbo Pascal đã

dành cho bạn khi làm việc trên máy, đặc biệt cách dùng khả năng trợ giúp (Help) theo ngữ cảnh của Turbo Pascal. Khi muốn cần tra cứu một đối tượng nào khi đang soạn chương trình, bạn chỉ cần đưa con trỏ đến phạm vi tên của đối tượng đó trong trang soạn thảo (nếu chưa có bạn có thể gõ vào) rồi nhấn Ctrl-F1. Khi đó một cửa sổ với nội dung trợ giúp sẽ hiện ra. Bạn có thể tìm thấy trong nội dung này cú pháp của đối tượng cũng như cách dùng nó, những tên của các đối tượng liên quan mà bạn có thể tra cứu tiếp. Ngoài ra còn có những thí dụ minh họa mà bạn có thể dùng kỹ thuật cắt, dán để chép chúng vào trang soạn thảo và chạy thử ngay, ... Chẳng hạn, tại cửa sổ soạn thảo của Turbo Pascal bạn gõ vào từ `ReadKey`, sau đó nhấn tổ hợp phím Ctrl-F1, bạn sẽ được nội dung trợ giúp như sau:

```
 ReadKey (function)      (Crt unit and WinCrt unit)
Reads a character from the keyboard.

 Declaration:
 function ReadKey: Char;

 Target:
Windows, Real, Protected

 Return Value:
Returns a character or an extended scan code.

 Remarks:
The character is not echoed to the screen.

 Sample Code:

 {ReadKey.PAS}

 {Sample code for the ReadKey function.}

 { For Windows: }
 { uses WinCrt; }

 uses Crt;

 var
  C: Char;

 begin
  Writeln('Please press a key');
  C := Readkey;
  Writeln(' You pressed ', C, ', whose ASCII value is ', Ord(C), '.');
 end.
```

Bạn có thể xem tóm tắt môi trường làm việc của Turbo Pascal ở phụ lục cuối sách.

Để tiện theo dõi, chúng tôi chia nội dung chương này thành 3 bài (đánh thứ tự **A**, **B**, **C**) tương ứng với 3 unit được giới thiệu là CRT, DOS và GRAPH.

A. Unit CRT

Unit CRT khai thác máy tính trên các khía cạnh: Màn hình (văn bản), bàn phím và âm thanh. Dùng các dịch vụ của unit này, bạn có thể điều khiển việc kết xuất thông tin ra màn hình, xây dựng các giao diện màn hình-bàn phím, dùng bàn phím điều khiển chương trình hoặc sử dụng âm thanh để xây dựng các chương trình mô phỏng.

Unit này chỉ được sử dụng trong các chương trình chạy trên các máy IBM PC, AT, PS/2s và các máy tương thích. Một ưu điểm của việc sử dụng CRT là xử lý nhanh và mềm dẻo việc đưa ra màn hình.

Hiện nay, với một số máy có tốc độ cao, unit này có thể gặp lỗi. Tuy nhiên điều này đã được khắc phục. Nếu bạn thấy khi chạy với CRT, dù chương trình chưa kịp làm công việc gì, mà đã gặp lỗi *Division by zero* thì cần thay CRT hiện có bằng bản mới.

1. Màn hình

Hệ thống cung cấp một vùng nhớ gồm 4000 byte, gọi là Video Ram, để quản lý những thông tin được đưa ra màn hình. Để có thể dùng màn hình như ý muốn, cần phải biết địa chỉ của vùng này và các cách thức tổ chức dữ liệu trong nó (tham khảo chương **QUẢN LÝ BỘ NHỚ**). Unit CRT hỗ trợ cho người dùng bằng cách thiết kế một số biến, hàm, thủ tục cho phép khai thác màn hình mà không cần quan tâm đến hệ thống. Khi khai thác màn hình cần để ý đến hai khía cạnh kỹ thuật: Độ phân giải và khả năng mầu của màn hình.

1.1. Tọa độ màn hình

Màn hình tiêu chuẩn của các máy IBM PC hiện nay (chế độ văn bản) gồm 80 cột và 25 dòng, mỗi ký tự được thể hiện trên màn hình tại một tọa độ cột, dòng. Các cột được đánh số từ 1 theo chiều từ trái sang và các dòng được đánh số từ 1 theo chiều từ trên xuống (như thế vị trí góc trên trái của màn hình sẽ có tọa độ cột, dòng là 1, 1).

Khi màn hình đang ở trạng thái chờ đợi, một điểm sáng nhấp nháy trên màn hình, gọi là con trỏ màn hình (cursor). Mọi sự kết xuất ra màn hình bắt đầu từ vị trí này. Mỗi khi kết xuất xong, cursor tự động dịch đến vị trí kế tiếp.

Để đưa cursor đến nơi mong muốn, unit CRT xây dựng thủ tục

```
procedure GotoXY(x, y: byte);
```

trong đó `x`, `y` là tọa độ cột, dòng của vị trí cần đến. Thủ tục này dùng để chỉ định vị trí kết xuất (ra màn hình) trước khi dùng `Write`, hoặc vị trí nhập liệu (từ bàn phím) trước khi dùng `Read`. Chú ý rằng thủ tục này không có tác dụng gì đối với các file văn bản. Nhờ thủ tục này, bạn có thể bố trí màn hình nhập/xuất thích hợp với công việc.

Thí dụ các lệnh

```
GotoXY(37, 12); Write('Turbo'); GotoXY(37, 13); Write('Pascal');
```

sẽ đưa ra hai dòng Turbo và Pascal tại khoảng giữa màn hình.

Thí dụ dưới đây xây dựng thủ tục **Frame** vẽ một khung chữ nhật trên màn hình:

```
procedure  Frame(x1, y1, x2, y2: byte);
var
   i: byte;
begin
  for i := x1+1 to x2-1 do
  begin
    GotoXY(i, y1); Write(#196);
    GotoXY(i, y2); Write(#196);
  end;
  for i := y1+1 to y2-1 do
  begin
    GotoXY(x1, i); Write(#179);
    GotoXY(x2, i); Write(#179);
  end;
  GotoXY(x1, y1); Write(#218);
  GotoXY(x2, y1); Write(#191);
  GotoXY(x1, y2); Write(#192);
  GotoXY(x2, y2); Write(#217);
end;
```

trong đó x1, y1 là tọa độ góc trên trái và x2, y2 là tọa độ góc dưới phải của khung.

Ngược với thủ tục GotoXY là các hàm

```
function WhereX: byte;
```

trả về tọa độ cột của cursor và hàm

```
function WhereY: byte;
```

trả về tọa độ dòng của cursor. Các hàm này dùng để ghi nhận những vị trí cursor để khi cần có thể truy cập lại.

Thí dụ

```
Write('Ho ten: ');
x1 := WhereX; y1 := WhereY;
Writeln;
Write('Dia chi: ');
x2 := WhereX; y2 := WhereY;
Writeln;
Write('So dien thoai: ');
x3 := WhereX; y3 := WhereY;
GotoXY(x1, y1); Readln(Name);
GotoXY(x2, y2); Readln(Address);
GotoXY(x3, y3); Readln(Tel);
```

1.2. Thuộc tính mầu

Mỗi ký tự thể hiện trên màn hình có hai giá trị mầu: Mầu của nét chữ (gọi là mầu chữ) và mầu của hình chữ nhật phía sau ký tự (gọi là mầu nền). Có 16 giá trị mầu được đánh số từ 0 đến 15, chia thành hai nhóm: Nhóm tối có giá trị từ 0 đến 7 và nhóm sáng có giá trị từ 8 đến 15. Để dễ nhớ, unit CRT định nghĩa các hằng với các tên mầu tương ứng:

nhóm mầu tối

giá trị	tên hằng	ý nghĩa mầu
0	Black	đen
1	Blue	xanh lam
2	Green	xanh lá cây
3	Cyan	xanh ngọc
4	Red	đỏ
5	Magenta	tím hồng
6	Brown	nâu
7	LightGray	xám sáng

nhóm mầu sáng

giá trị	tên hằng	ý nghĩa mầu
8	DarkGray	xám đậm
9	LightBlue	xanh lam sáng
10	LightGreen	xanh lá cây sáng
11	LightCyan	xanh ngọc sáng
12	LightRed	đỏ sáng
13	LightMagenta	tím hồng sáng
14	Yellow	vàng
15	White	trắng

Để thiết lập thuộc tính mầu, unit CRT cung cấp hai thủ tục:

```
procedure TextColor(Color: byte);
```

đặt mầu chữ có giá trị cho bởi `color`, và

```
procedure TextBackground(Color: byte);
```

đặt mầu nền có giá trị cho bởi `color`.

Thuộc tính mầu được cất trong một byte như sau: 4 bit thấp là mầu chữ, 3 bit cao tiếp theo là mầu nền, bit cao nhất là thuộc tính nhấp nháy. Vì thế mầu chữ có đủ 16 giá trị, mầu nền chỉ có 8 giá trị tương ứng với nhóm mầu tối và nếu cộng thêm 128 cho thuộc tính mầu (nghĩa là bit cao nhất được bật) thì sẽ tạo nhấp nháy cho các ký tự. Hằng Blink (nghĩa là nhấp nháy) được CRT định nghĩa bằng 128 cũng để làm việc này.

Các giá trị mầu được đặt sẽ có hiệu lực với các lệnh xuất sau đó cho đến khi gặp một lệnh đặt mầu mới. Giá trị ngầm định của mầu nền là 0 (đen) và mầu chữ là 7 (xám sáng).

Thí dụ

```
TextColor(Red); TextBackground(Cyan);   {đặt mầu chữ đỏ, mầu nền xanh ngọc}
Write(' Turbo ');
TextColor(Yellow); TextBackground(Blue);   {đặt mầu chữ vàng, mầu nền xanh lam}
Write(' Pascal ');
TextColor(LightGray); TextBackground(Black);   {trở về giá trị ngầm định}
```

Để biết được giá trị mầu hiện tại, bạn có thể tham khảo giá trị của biến `TextAttr` (kiểu byte) được CRT xây dựng để cất giữ thuộc tính mầu theo quy cách đã nêu. Chẳng hạn nếu biến `TextAttr` hiện có giá trị 23 thì có nghĩa mầu chữ là 7 (xám sáng) và mầu nền là 1 (xanh lam) vì 23 được phân tích thành 23 = 1*16+7. Ngược lại bạn có thể đặt lại giá trị cho biến này để thiết lập thuộc tính mầu thay cho lời gọi các thủ tục `TextColor` và `TextBackground`. Chẳng hạn lệnh gán

```
TextAttr := 3*16+14;
```

sẽ đặt mầu chữ là 14 (vàng) và mầu nền là 3 (xanh ngọc).

Unit CRT còn xây dựng 3 thủ tục dưới đây nhằm thay đổi độ sáng tối của mầu. Những thủ tục này đặt lại giá trị cho `TextAttr`:

```
procedure NormVideo;
```

đặt lại giá trị mầu nguyên thủy (trước lúc chạy chương trình) cho `TextAttr`,

```
procedure HighVideo;
```

bật bit mầu sáng (bit thứ tư tính từ bit thấp nhất) cho `TextAttr`,

```
procedure LowVideo;
```

tắt bit mầu sáng (bit thứ tư tính từ bit thấp nhất) cho `TextAttr`.

Thí dụ

```
Uses CRT;
Begin
  ClrScr;
  Writeln('Original Textattr = ', TextAttr);        {mầu nguyên thủy}
  TextAttr := 6;
  Writeln('Set Textattr = ', TextAttr);             {mầu nâu}
  HighVideo;
  Writeln('HighVideo, Textattr = ', TextAttr);      {mầu vàng}
  LowVideo;
  Writeln('LowVideo, Textattr = ', TextAttr);       {mầu nâu}
  NormVideo;
  Write('NormVideo, Textattr = ', TextAttr);        {mầu nguyên thủy}
  Readln;
End.
```

1.3. Cửa sổ

Giá trị ngầm định của cửa sổ làm việc là toàn bộ màn hình. Tuy nhiên, khi cần có thể quy định cửa sổ làm việc hẹp hơn, bằng lời gọi thủ tục:

```
procedure Window(x1, y1, x2, y2: byte);
```

trong đó `x1`, `y1` là tọa độ góc trên trái, `x2`, `y2` là tọa độ của góc dưới phải của cửa sổ.

Khi một cửa sổ được định nghĩa, mọi kết xuất ra màn hình đều nằm trong phạm vi cửa sổ và mọi giá trị tọa độ (trừ tọa độ của bản thân cửa sổ) đều là tương đối đối với cửa sổ hiện hành, chẳng hạn tọa độ 1, 1 lúc này là góc trên trái của cửa sổ. Cửa sổ hiện hành có hiệu lực cho đến khi một cửa sổ mới được định nghĩa.

Lời gọi

```
Window(1, 1, 80, 25);
```

sẽ quay về chế độ toàn màn hình (giá trị ngầm định). Kích thước nhỏ nhất của cửa sổ là một cột và một dòng (góc trên trái trùng với góc dưới phải).

Để lấy lại vị trí cửa sổ hiện hành, bạn tham khảo giá trị của hai biến `WindMin` và `WindMax` (kiểu `word`). `WindMin` cất giữ tọa độ góc trên trái và `WindMax` cất giữ tọa độ góc dưới phải, trong đó byte thấp cất tọa độ cột và byte cao cất tọa độ dòng. Chú ý ở đây các giá trị tọa độ được tính từ 0.

Để thực hiện thao tác sang trang, bạn dùng thủ tục không tham số

```
procedure ClrScr;
```

Thủ tục này sẽ xóa tất cả các ký tự trong cửa sổ hiện hành và đặt toàn cửa sổ có mầu nền được xác định bởi `TextBackground` trước đấy. Thí dụ nếu đang ở chế độ toàn màn hình, các lệnh

```
TextBackground(Blue);
ClrScr;
```

sẽ xóa tất cả các ký tự hiện có trên màn hình và đặt toàn bộ màn hình có mầu lam.

Thủ tục

```
procedure ClrEol;
```

sẽ xóa tất cả các ký tự kể từ vị trí con trỏ màn hình đến hết dòng (trong phạm vi cửa sổ đang hoạt động). Mọi vị trí bị xóa có mầu nền hiện hành. Vị trí con trỏ không thay đổi.

Các thủ tục dưới đây dùng kỹ thuật cuộn màn hình của BIOS:

```
procedure DelLine;
```

xóa dòng đang chứa con trỏ màn hình, các dòng ở phía dưới được dịch lên một dòng. Một dòng mới với mầu nền hiện hành được thêm vào ở phía dưới cùng cửa sổ.

```
procedure InsLine;
```

chèn thêm một dòng trống với mầu nền hiện hành vào vị trí con trỏ màn hình, các dòng ở phía dưới được đẩy xuống một dòng. Dòng ở đáy cửa sổ sẽ bị mất.

Các lệnh liên quan đến cửa sổ giúp chương trình khai thác tốt hơn màn hình, đặc biệt, với kỹ thuật cuộn màn hình và kỹ thuật dùng bàn phím, bạn có thể dễ dàng thiết kế việc xem đi xem lại nhiều dòng dữ liệu trên một cửa sổ (tham khảo thí dụ ở mục **2.2. Nhận diện phím được ấn**).

1.4. Các chế độ màn hình

Chế độ thông dụng của các màn hình mầu VGA hiện nay là 16 mầu với 80 cột, 25 dòng. Tuy nhiên CRT cho phép khai thác nhiều chế độ màn hình khác nhau bằng lời gọi thủ tục:

```
procedure TextMode(mode: word);
```

trong đó `mode` có thể nhận các giá trị dưới đây (các tên hằng được CRT định nghĩa cho dễ nhớ):

Tên hằng	Giá trị	Mô tả
BW40	0	40 cột, 25 dòng, đen/trắng trên màn hình mầu
BW80	2	80 cột, 25 dòng, đen/trắng trên màn hình mầu
Mono	7	80 cột, 25 dòng, đen/trắng trên màn hình đơn sắc
CO40	1	40 cột, 25 dòng, mầu trên màn hình mầu
CO80	3	80 cột, 25 dòng, mầu trên màn hình mầu
Font8x8	256	50 (hoặc 43) dòng cho VGA (hoặc EGA)

Giá trị ngầm định cho các màn hình mầu hiện nay là `CO80`. Giá trị `Font8x8` được dùng kết hợp với các giá trị khác. Chẳng hạn, lời gọi `TextMode(CO40+Font8x8)` sẽ đặt chế độ 40 cột, 50 dòng, 16 mầu cho màn hình mầu VGA. Vì giá trị `Font8x8` được ghi nhận bởi bit thấp nhất trong byte cao của `mode` nên để gỡ bỏ nó (mà không ảnh hưởng đến phần còn lại) bạn thực hiện `TextMode(Lo(mode))`.

Khi `TextMode` được gọi, toàn bộ màn hình cũ sẽ bị xóa, con trỏ màn hình được trả về góc trên trái, chế độ toàn màn hình được thiết lập (nghĩa là cửa sổ nếu đang có sẽ bị hủy bỏ) và `TextAttr` được đặt lại giá trị nguyên thủy (giống như gọi `NormVideo`). Unit CRT định nghĩa biến `LastMode` để lưu lại giá trị `mode` của lần gọi `TextMode` gần nhất. Chương trình dưới đây chuyển màn hình VGA sang chế độ 80 cột, 50 dòng sau đó lại trở về chế độ ban đầu:

```
Program DemoTextMode;
Uses CRT;
Var color: byte;
    OldMode: word;
Begin
  ClrScr;
  Writeln('Che do man hinh hien tai');
  Writeln('Last Mode = ', LastMode);
  Writeln('Text Attr = ', TextAttr);
  OldMode := LastMode;
  for color := 1 to 15 do
  begin
    TextColor(color);
    Writeln('  Text Color = ', color);
  end;
  Writeln;
  Writeln('Text Attr = ', TextAttr);
  Write('An phim Enter de chuyen che do man hinh ... ');
  Readln; TextMode(CO80+Font8x8);
  Writeln('Hien dang o che do CO80+Font8x8');
  Writeln('Last Mode = ', LastMode);
  Writeln('Text Attr = ', TextAttr);
  Writeln;
  for color := 1 to 15 do
  begin
    TextColor(color);
    Writeln('  Text Color = ', color);
  end;
  Writeln;
  Writeln('Text Attr = ', TextAttr);
  Write('An phim Enter de chuyen che do man hinh ... ');
  Readln; TextMode(OldMode);
  Writeln('Tro ve gia tri ban dau ...');
  Writeln('Last Mode = ', LastMode);
  Writeln('Text Attr = ', TextAttr);
  Write('An phim Enter de ket thuc ... ');
  Readln;
End.
```

2. Bàn phím

Ngoài việc nhập dữ liệu cho chương trình, bàn phím còn là công cụ điều khiển chương trình khi chương trình đang chạy. Nhờ bàn phím, người dùng có thể "lái" chương trình hoạt động theo những lựa chọn đã được thiết kế. Để làm được điều này, chương trình phải có khả năng nhận biết trong khi đang hoạt động, có phím nào được ấn hay không và phím được ấn là phím gì. Có thể dùng các ngắt bàn phím của DOS để thực hiện việc "bẫy phím", tuy nhiên unit CRT cung cấp cho người dùng các hàm khá tiện lợi để thực hiện chức năng này.

2.1. Nhận biết có phím được ấn

Unit CRT xây dựng hàm lôgic không tham số:

```
function KeyPressed: boolean;
```

trả về giá trị `True` nếu vùng đệm bàn phím khác rỗng và trả về `False` nếu trái lại. Thông thường vùng đệm bàn phím là rỗng vì các ký tự do các phím đưa vào được xử lý ngay, tuy nhiên, nếu chương trình đang bận, các ký tự này được giữ lại trong vùng đệm phím (theo quy tắc hàng đợi) cho đến khi gặp những lệnh đọc chúng. Nhờ hàm `KeyPressed`, bạn có thể kiểm tra khi chương trình đang chạy, có phím nào được ấn hay không để quyết định các lựa chọn cho chương trình.

Mô hình dưới đây xây dựng một chương trình hoạt động liên tục và kết thúc khi có một phím được ấn:

```
repeat
  ......
until KeyPressed;
```

Một dạng riêng hay dùng của mô hình trên là vòng lặp rỗng

```
repeat until KeyPressed;
```

sẽ dừng chương trình cho đến khi có một phím được ấn (phân biệt với `Readln;` chỉ chờ ấn phím Enter).

Chú ý rằng, hàm `KeyPressed` chỉ kiểm tra mà không làm thay đổi trạng thái vùng đệm phím. Vì thế `KeyPressed` chỉ phát hiện có phím được ấn trong lần gọi đầu tiên khi vùng đệm còn rỗng và phải làm sạch vùng này trước khi dừng lại `KeyPressed`.

2.2. Nhận diện phím được ấn

Để nhận diện phím được ấn, CRT xây dựng hàm ký tự không tham số

```
function ReadKey: char;
```

lấy ra ký tự đầu tiên trong vùng đệm phím và trả về giá trị của ký tự này. Nếu vùng đệm phím đang rỗng, hàm sẽ dừng chương trình, chờ một ký tự được đưa vào từ bàn phím (ký tự đưa vào không hiện lên màn hình). Thông thường vùng đệm phím ở trạng thái rỗng, vì thế khi gặp lời gọi `ReadKey`, chương trình sẽ dừng cho đến khi có một phím được ấn, khi đó hàm sẽ trả về ký tự ứng với phím này và chương trình lại tiếp tục. Nhờ giá trị trả về của hàm, bạn có thể thiết kế các lựa chọn tiếp theo của chương trình, tùy thuộc vào giá trị này.

Chẳng hạn mô hình dưới đây thiết kế một giao diện menu đơn giản, cho phép người dùng ấn phím để chọn các chức năng tương ứng:

```
repeat
  Menu;                {trình bày các chức năng trên màn hình}
  Write('Go phim tuong ung de chon:');
  ch := ReadKey;       {chờ ấn phím}
  case ch of           {rẽ nhánh theo phím được ấn}
    .....
    .....
  end;
until ch = #27;        {kết thúc khi ấn phím Esc}
```

Khác với KeyPressed, mỗi lần gọi hàm ReadKey, vùng đệm phím sẽ bớt đi một ký tự chừng nào vùng này còn khác rỗng. Vòng lặp dưới đây sẽ làm sạch vùng đệm phím nếu vùng này còn ký tự:

```
while KeyPressed do ch := ReadKey;
```

Việc làm sạch vùng đệm phím trước khi xử lý giúp chương trình tránh gặp những phím không mong muốn.

Thông thường, mỗi phím có một giá trị ký tự tương ứng với nó, thí dụ phím A tương ứng với ký tự mã 65, phím Enter tương ứng với ký tự mã 13, phím Esc tương ứng với ký tự mã 27, ... Tuy nhiên một số phím đặc biệt như các phím mũi tên, Insert, Delete, Home, End, PgUp, PgDn, F1, F2, ... khi được ấn, sẽ gửi vào vùng đệm phím một lúc 2 ký tự, đầu tiên là ký tự có mã 0, tiếp theo là một ký tự có mã khác 0. Sự phân biệt các phím này dựa vào giá trị của mã thứ hai (gọi là mã quét mở rộng), vì thế đối với chúng, cần phải gọi ReadKey hai lần để nhận diện:

```
if ReadKey = #0 then ch := ReadKey;
```

Dựa vào đặc điểm vừa nêu, bạn có thể mở rộng hàm ReadKey thành một hàm cho phép nhận diện cả những phím có mã quét mở rộng như đề nghị dưới đây:

```
function GetKey: integer;
var k: integer;
begin
  k := Ord(ReadKey);
  if k = 0 then k := -Ord(ReadKey); {gọi ReadKey 2 lần}
  GetKey := k;
end;
```

Hàm GetKey hoạt động giống như hàm ReadKey nhưng trả về một số nguyên tương ứng với phím được ấn. Nếu là phím thông thường thì giá trị trả về là mã ASCII của ký tự, nếu là phím có mã quét mở rộng thì giá trị trả về là mã quét này với dấu trừ để phân biệt với phím thông thường.

Chương trình minh họa dưới đây cho thấy các giá trị trả về của hàm GetKey khi có phím được ấn:

```
Program DemoGetKey;
Uses CRT;
```

```
  function GetKey: integer;
  var k: integer;
  begin
    k := Ord(ReadKey);
    if k = 0 then k := -Ord(ReadKey);
    GetKey := k;
  end;

Var k: integer;
Begin
  ClrScr;
  repeat
    k := GetKey;
    Writeln(k);
  until k = 27;
End.
```

Chương trình trên chờ và hiển thị lên màn hình mã của phím được ấn, chẳng hạn khi ấn phím Enter bạn nhận được giá trị 13. Với những phím có mã quét mở rộng, bạn nhận được những giá trị âm của mã này. Chương trình kết thúc khi ấn phím Esc (mã 27). Bạn có thể kiểm tra lại một số giá trị trả về dưới đây của hàm `GetKey` khi ấn những phím hay dùng:

Phím được ấn	Giá trị trả về
Left Arrow	-75
Right Arrow	-77
Up Arrow	-72
Down Arrow	-80
Page Up	-73
Page Down	-81
Home	-71
End	-79
Insert	-82
Delete	-83
F1	-59
Enter	13
Escape	27
Backspace	8

Hàm `GetKey` được dùng thay hàm `ReadKey` khi cần thiết. Để tiện dùng về sau, bạn nên lưu lại hàm `GetKey` như một thư viện, kèm theo định nghĩa một số tên hằng tương ứng với

những giá trị hay dùng cho dễ nhớ. Dưới đây là văn bản unit KEY được xây dựng với mục đích vừa nêu:

```
Unit KEY;
Interface
const
    LeftKey  = -75;
    RightKey = -77;
    UpKey    = -72;
    DnKey    = -80;
    InsKey   = -82;
    DelKey   = -83;
    HomeKey  = -71;
    EndKey   = -79;
    PgUpKey  = -73;
    PgDnKey  = -81;
    F1Key    = -59;
    F2Key    = -60;
    F3Key    = -61;
    F4Key    = -62;
    F5Key    = -63;
    F6Key    = -64;
    F7Key    = -65;
    F8Key    = -66;
    F9Key    = -67;
    F10Key   = -68;
    EnterKey =  13;
    EscKey   =  27;
    BackKey  =  8;
  function GetKey: integer;
Implementation
  Uses CRT;
  function GetKey: integer;
  var k: integer;
  begin
    k := Ord(ReadKey);
    if k = 0 then k := -Ord(ReadKey);
    GetKey := k;
  end;
End.
```

Chương trình dưới đây thiết kế một cửa sổ, trong đó hiển thị nội dung của file lưu trữ số tiền gửi của các khách hàng (xem chương **GIAO TIẾP FILE**, mục **3.1. File định kiểu**). Khác với các chương trình trước đây, chương trình này cho phép dùng các phím mũi tên lên, xuống để xem nội dung file theo các chiều xuôi, ngược. Trong chương trình có sử dụng unit KEY:

```
Program ShowFile;
Uses CRT, KEY;
Type
  BankRec = record
              HoTen: string[20];
              SoTaiKhoan: integer;
              SoTien: Real;
            end;
Var
  f: file of BankRec;
  name: string;
  i, x, y: byte;
  r: BankRec;
  k: integer;
  top, bottom, n: longint;

Begin
  ClrScr;
  GotoXY(28, 1); Write('Ten file tai khoan: '); Readln(name);
  Assign(f, name); Reset(f);
  n := FileSize(f);
  GotoXY(28, 2); Write('So cac tai khoan: ', n);
  GotoXY(12, 4);
  Write('Stt     Ho ten          So tai khoan     So tien gui');
  GotoXY(12, 24);
  Write('An cac phim Up, Down de xem / Esc de ket thuc ... ');
  Window(12, 6, 66, 22);
  TextBackground(blue);
  ClrScr;
  top := 0; bottom := 16;
  if n < bottom+1 then bottom := n-1;
  for i := 1 to bottom+1 do        {hiển thị một trang cửa sổ}
  begin
    Read(f, r);
    GotoXY(1, i);
    with r do
      Write(i:5, '. ', HoTen, ' ':21-Length(HoTen),
            SoTaiKhoan:6, SoTien:18:0);
  end;
  x := WhereX; y := WhereY;
  repeat                           {vòng lặp cuộn cửa sổ}
    GotoXY(x, y);
    k := GetKey;                   {chờ ấn phím}
    case k of                      {xử lý theo phím được ấn}
      UpKey: if top > 0 then             {phím được ấn là ↑}
             begin
```

```
            GotoXY(1, 1); InsLine;   {đẩy xuống 1 dòng}
            Dec(top); Dec(bottom);
            Seek(f, top); Read(f, r);
            with r do
              Writeln(top+1:5, '. ', HoTen, ' ':21-Length(HoTen),
                      SoTaiKhoan:6, SoTien:18:0);
          end;
    DnKey: if bottom < n-1 then        {phím được ấn là ↓}
          begin
            GotoXY(1, 1); DelLine;   {kéo lên 1 dòng}
            Inc(top); Inc(bottom);
            Seek(f, bottom); Read(f, r);
            GotoXY(1, 17);
            with r do
              Write(bottom+1:5, '. ', HoTen, ' ':21-Length(HoTen),
                    SoTaiKhoan:6, SoTien:18:0);

          end;
    end;
  until k = EscKey;                    {kết thúc khi ấn phím Esc}
  Close(f);
  Window(1, 1, 80, 25);
  TextBackground(black);
  ClrScr;
End.
```

2.3. Điều khiển chương trình bằng bàn phím

Để có thể dùng bàn phím can thiệp được hoạt động của chương trình khi chương trình đang chạy, người ta kết hợp việc dùng hai hàm `KeyPressed` và `ReadKey` (hoặc `GetKey`). Thoạt đầu, hàm `KeyPressed` nhận biết có phím nào được ấn hay không, tiếp theo hàm `ReadKey` (hoặc `GetKey`) sẽ nhận diện phím này. Sau đó cấu trúc `case` sẽ rẽ nhánh chương trình theo các giá trị mà hàm `ReadKey` (hoặc `GetKey`) trả về. Tất cả được bố trí trong một thân vòng lặp mô tả một nhịp hoạt động của chương trình:

```
repeat
  {các lệnh mô tả một nhịp hoạt động của chương trình}
  .........
  .........
  k := 0;              {khởi gán không có phím nào được ấn}
  if KeyPressed then k := GetKey;   {nhận diện phím được ấn}
  case k of            {rẽ nhánh theo giá trị của phím}
    .......
    .......
  end;
until k = 27;   {kết thúc khi ấn phím Esc}
```

Thông thường trong các chương trình mô phỏng hay trò chơi, hoạt động của chương trình được thể hiện qua những hình ảnh chuyển động trên màn hình, vì thế nhiều khi phải làm chậm chúng lại để thị giác con người kịp cảm nhận. Unit CRT thiết kế thủ tục:

```
procedure Delay(t: word);
```

tạo một khoảng thời gian trễ xấp xỉ `t` mili giây. Thủ tục này thường được cài vào giữa hai nhịp hoạt động của chương trình. Như vậy, qua mô hình trên, bạn có thể dùng bàn phím để điều khiển chương trình chạy nhanh lên hay chậm đi.

Dưới đây là một chương trình minh họa thiết kế đã nêu. Chương trình điều khiển một dòng chữ chạy trên suốt chiều rộng màn hình theo kiểu đèn kéo quân (nội dung dòng chữ cho trong chương trình). Khi dòng chữ đang chạy, có thể điều khiển nó bằng cách ấn các phím như sau:

- phím mũi tên trái: hướng chạy từ phải sang trái,
- phím mũi tên phải: hướng chạy từ trái sang phải,
- phím mũi tên lên: dòng chữ chạy chuyển lên trên một dòng,
- phím mũi tên xuống: dòng chữ chạy chuyển xuống dưới một dòng,
- phím PageUp: tăng tốc độ chạy,
- phím PageDown: giảm tốc độ chạy,
- các phím F1, F2, ..., F7: đặt lại mầu các ký tự trong dòng chữ như sau:
  - . phím F1: mầu ký tự xanh lam (1 - Blue),
  - . phím F2: mầu ký tự xanh lá cây (2 - Green),
  - . phím F3: mầu ký tự xanh ngọc (3 - Cyan),
  - . phím F4: mầu ký tự đỏ (4 - Red),
  - . phím F5: mầu ký tự tím hồng (5 - Magenta),
  - . phím F6: mầu ký tự nâu (6 - Brown),
  - . phím F7: mầu ký tự trắng (7 - LightGray),
- phím Escape: kết thúc chương trình.

Hướng chạy ngầm định của dòng chữ là từ phải sang trái, vị trí dòng chữ chạy ngầm định là dòng 1, mầu chữ ngầm định là mầu trắng (mầu số 7 - LightGray). Trong chương trình có dùng unit KEY.

```
Program MoveText;
Uses CRT, KEY;
Var k: integer;
    s: string;
    color, direct, row, t, n: word;
Begin
  ClrScr;
  direct := 1; row := 1; color := 7;
  t := 100; n := 79;
  s := '...... PROGRAM DEMO MOVE TEXT ......';
  while Length(s) < n do s := s+' ';
  repeat
```

```
    case direct of
      1: s := Copy(s, 2, n-1)+s[1];
      2: s := s[n]+Copy(s, 1, n-1);
    end;
    TextColor(color); GotoXY(1, row); Write(s);
    Delay(t);
    k := 0;
    if KeyPressed then k := GetKey;      {bẫy phím}
    case k of       {xử lý theo giá trị phím}
      LeftKey: direct := 1;          {từ phải sang trái}
      RightKey: direct := 2;         {từ trái sang phải}
      DnKey: begin          {xuống 1 dòng}
               GotoXY(1, row); ClrEol;
               if row < 25 then Inc(row) else row := 1;
             end;
      UpKey: begin          {lên 1 dòng}
               GotoXY(1, row); ClrEol;
               if row > 1 then Dec(row) else row := 25;
             end;
      PgUpKey: if t > 10 then Dec(t, 10);      {tăng tốc}
      PgDnKey: if t < 1000 then Inc(t, 10);    {giảm tốc}
      F7Key..F1Key: color := F1Key-k+1;        {đổi mầu}
    end;
  until k = EscKey;     {kết thúc nếu Esc được ấn}
  TextColor(7); ClrScr;
End.
```

3. Âm thanh

Loa của máy PC làm việc theo nguyên lý chuyển mạch bật/tắt. Khi được bật, loa sẽ phát ra âm thanh ở một tần số nào đó cho đến khi được tắt. Để khai thác khả năng này, unit CRT cung cấp thủ tục

```
procedure Sound(h: word);
```

làm cho loa phát ra một âm thanh có tần số là `h` (tính theo đơn vị Hertz), và thủ tục

```
procedure NoSound;
```

có tác dụng làm tắt âm thanh đang được phát bởi loa.

Mỗi khi gọi thủ tục `Sound` để phát một âm thanh nào đó, âm thanh này sẽ được kéo dài cho đến khi gặp lời gọi `NoSound`. Để ấn định thời khoảng này, bạn dùng thủ tục `Delay` với tham trị `t`, xác định thời gian trễ (tính bằng mili giây). Như vậy một âm thanh sẽ được xác định bởi các tham số `h` (cao độ) và `t` (trường độ) theo mẫu của thủ tục sau:

```
procedure CreatSound(h, t: word);
begin
  Sound(h); Delay(t); NoSound;
end;
```

Để không phải nhắc lại thủ tục này trong các thí dụ tiếp theo, ta giả thiết rằng nó đã được xây dựng trong unit có tên là SOUNDS.

3.1. Mô phỏng âm thanh

Có thể mô phỏng một âm thanh nhờ thủ tục `CreatSound(h, t)`. Vấn đề còn lại là tìm hiểu các giá trị thích hợp của `h` và `t`. Thí dụ dưới đây mô tả tiếng lộc cộc của vó ngựa phi:

```
Program DemoSound;
Uses CRT, SOUNDS;

  procedure Clockoc(h: word);
  begin
    CreatSound(h, 20);
    Delay(50);
    CreatSound(h, 20);
    Delay(100);
    CreatSound(h, 40);
  end;

Begin
  repeat
    Clockoc(500);
    Delay(200);
  until KeyPressed;
End.
```

Việc khai thác âm thanh thường được dùng để xây dựng các chương trình trò chơi hoặc mô phỏng, vì thế nếu bạn là người thường xuyên phải phát triển những ứng dụng loại này trên Turbo Pascal, thì bạn nên tổ chức riêng một thư viện, chứa một số âm thanh mẫu như tiếng các loại động cơ, tiếng còi, tiếng nổ, tiếng va chạm, ... để tiện dùng.

3.2. Mô phỏng nốt nhạc

Một trường hợp riêng của việc khai thác âm thanh là mô phỏng các nốt nhạc. Một nốt nhạc cao độ `h` (tính theo Hertz), trường độ `t` (tính theo mili giây) sẽ được thực hiện bởi một lời gọi `CreatSound(h, t)` tương ứng. Theo âm luật, từ một quãng 8 đến quãng 8 kế tiếp, thừa số nhân của cao độ là 2. Trong một quãng 8, có 12 khoảng nửa cung, vì thế thừa số nhân giữa hai khoảng này là căn bậc 12 của 2. Vì vậy chỉ cần biết cao độ của một nốt làm gốc, bạn có thể tính cao độ của các nốt khác. Chẳng hạn, xuất phát từ cao độ của nốt đô trung là 512, cao độ của các nốt trong quãng 8 trung lần lượt là (sau khi đã làm tròn): đô = 512, đô thăng = 542, rê = 575, rê thăng = 609, mi = 645, fa = 683, fa thăng = 724, son = 767, son thăng = 813, la = 861, la thăng = 912, si = 967. Về trường độ, lấy nốt móc đơn làm đơn vị là 150, bạn tính được các trường độ khác: nốt đen = 300, nốt trắng = 600,

nốt trắng chấm = 900, nốt tròn = 1200, ... Chương trình dưới đây thể hiện một bản nhạc ngắn của đồng hồ, tất cả các nốt đều ở quãng 8 trung:

```
Program DemoMusic;
Uses CRT, SOUNDS;
Begin
  CreatSound(967, 300);     {si đen}
  CreatSound(767, 300);     {son đen}
  CreatSound(861, 300);     {la đen}
  CreatSound(575, 900);     {rê trắng chấm}
  Delay(60);
  CreatSound(575, 300);     {rê đen}
  CreatSound(861, 300);     {la đen}
  CreatSound(967, 300);     {si đen}
  CreatSound(767, 900);     {son trắng chấm}
  Delay(60);
  CreatSound(967, 300);     {si đen}
  CreatSound(861, 300);     {la đen}
  CreatSound(767, 300);     {son đen}
  CreatSound(575, 900);     {rê trắng chấm}
  Delay(60);
  CreatSound(575, 300);     {rê đen}
  CreatSound(861, 300);     {la đen}
  CreatSound(967, 300);     {si đen}
  CreatSound(767, 900);     {son trắng chấm}
End.
```

Để dễ dùng, bạn có thể xây dựng một unit trong đó có các thủ tục thực hiện những nốt nhạc cơ bản với những tên gợi nhớ.

B. Unit DOS

Unit DOS chứa các chương trình con để gọi các phục vụ DOS và BIOS cũng như xây dựng một số kiểu dữ liệu để chứa những thông tin của những phục vụ này.

Nhờ unit DOS, bạn có thể viết dễ dàng một chương trình Pascal xử lý những vấn đề thuộc về hệ thống như quản lý các file, đĩa, truy cập ngày, giờ, can thiệp màn hình, phím, điều khiển chuột, ...

Có thể chia unit DOS thành hai phần: Một phần chứa những thường trình và những kiểu dữ liệu đặc biệt nhằm thực hiện những phục vụ phổ biến nhất của DOS và BIOS, phần còn lại chứa kiểu dữ liệu thanh ghi `Registers` và các thủ tục `MsDos`, `Intr` cho phép thực hiện tất cả các ngắt của DOS và BIOS. Để dùng phần thứ nhất, bạn không cần hiểu nhiều về hệ thống, trong khi để dùng phần thứ hai, bạn cần phải biết thêm về cấu trúc các thanh ghi cũng như nội dung của các ngắt thông dụng.

1. Các kiểu dữ liệu và các thường trình đặc biệt

1.1. Kiểm tra lỗi

Unit DOS thiết kế sẵn biến `DosError` kiểu nguyên để chứa mã lỗi mỗi khi gọi các thường trình của nó. Biến `DosError` sẽ có giá trị 0 khi không có lỗi, mỗi giá trị trong các giá trị còn lại được tương ứng với một loại lỗi. Dưới đây là bảng giá trị của `DosError` với những ý nghĩa tương ứng được tra cứu trong HELP của hệ thống Turbo Pascal:

Code	Meaning	
0	No error	Không có lỗi
2	File not found	Không tìm thấy file
3	Path not found	Không tìm thấy đường dẫn
5	Access denied	Không thể thâm nhập
6	Invalid handle	Handle không hợp lệ
8	Not enough memory	Không đủ bộ nhớ
10	Invalid environment	Môi trường không hợp lệ
11	Invalid format	Khuôn dạng sai
18	No more files	Không còn file nào nữa

1.2. Xem phiên bản của DOS

Mỗi khi muốn xem phiên bản đang làm việc của DOS, bạn có thể dùng hàm:

```
Function DosVersion: word
```

trả lại phiên bản của DOS dưới dạng một word, trong đó byte thấp chứa chỉ số chính còn byte cao chứa chỉ số phụ.

Thí dụ.

```
Uses DOS;
```

```
Begin
  Writeln('Dos dang dung la: MsDos ',
          Lo(DosVersion), '.', Hi(DosVersion);
  Readln;
End.
```

1.3. Các phục vụ file

Unit DOS xây dựng một số kiểu dữ liệu và chương trình con cho phép tìm kiếm file cũng như truy cập vào các thông tin thuộc về hệ thống của file.

Các thuộc tính của file được tổ chức lưu trữ trong một byte, trong đó mỗi bit của byte mô tả trạng thái (có hay không) của một thuộc tính. Unit DOS định nghĩa các hằng sau đây tương ứng với các giá trị của thuộc tính (các bit được đánh số từ 1 theo thứ tự từ thấp đến cao):

ReadOnly	$01	bit 1	chỉ đọc	
Hidden	$02	bit 2	ẩn	
SysFile	$04	bit 3	file hệ thống	
VolumeID	$08	bit 4	nhãn đĩa	
Directory	$10	bit 5	thư mục	
Archive	$20	bit 6	lưu trữ	
AnyFile	$3F		file bất kỳ	(là tổng của các giá trị thuộc tính)

Như vậy, mỗi giá trị của byte chứa thuộc tính sẽ xác định một tập hợp các thuộc tính nào đó. Chẳng hạn, giá trị 7 có các bit 1, 2, 3 bằng 1, các bit khác bằng 0, xác định các thuộc tính ReadOnly, Hidden và SysFile.

Để lấy và đặt thuộc tính cho file, bạn có thể dùng các thủ tục:

```
GetFAttr(var F; var Attr: word);
SetFAttr(var F; Attr: word);
```

trong đó `F` là biến file (bất kỳ). Thủ tục `GetFAttr(var F; var Attr: word)` trả về thuộc tính của file `F` cho biến `Attr` còn thủ tục `SetFAttr(var F; Attr: word)` thiết lập thuộc tính của file `F` từ giá trị `Attr`.

Chú ý rằng, để dùng các thủ tục trên, bạn không được mở file. Thí dụ, bạn có thể thiết lập thêm các thuộc tính ReadOnly và Hidden cho file RES.DOC như sau:

```
Assign(F, 'RES.DOC');
GetFAttr(F, Attr);
Attr := Attr or ReadOnly or Hidden;
SetFAttr(F, Attr);
```

hoặc loại bỏ các thuộc tính ReadOnly và Hidden cho file RES.DOC như sau:

```
Assign(F, 'RES.DOC');
GetFAttr(F, Attr);
Attr := Attr and (not ReadOnly) and (not Hidden);
SetFAttr(F, Attr);
```

Để truy cập các thông tin thuộc về hệ thống của file (thuộc tính, ngày giờ cập nhật, kích thước, tên), unit DOS xây dựng kiểu dữ liệu `SearchRec` dùng cho một số thường trình liên quan:

```
Type
  SearchRec = Record
                Fill: array[1..21] of byte;
                Attr: byte;
                Time: LongInt;
                Size: LongInt;
                Name: string[12];
              End;
```

trong đó trường `Attr` là thuộc tính, `Time` là ngày giờ cập nhật, `Size` là kích thước và `Name` là tên của file, riêng trường `Fill` được để dành.

Để duyệt các file, bạn có thể dùng cặp thủ tục sau đây:

```
FindFirst(Path: string; Attr: word; var DirInfo: SearchRec);
FindNext(var DirInfo: SearchRec);
```

trong đó `FindFirst(Path: string; Attr: word; var DirInfo: SearchRec)` trả về cho biến `DirInfo` thông tin của file đầu tiên tìm thấy có đặc tả `Path` và thỏa mãn thuộc tính `Attr`, còn `FindNext(var DirInfo: SearchRec)` trả về cho biến `DirInfo` thông tin của file kế tiếp. Việc duyệt file được xây dựng theo mô hình:

```
FindFirst(Path, Attr, DirInfo);
while DosError = 0 do
begin
  {xử lý file tìm thấy}
  .............
  FindNext(DirInfo);
end;
```

Thí dụ dưới đây là một chương trình tương tự như lệnh DIR của DOS. Khi chạy, chương trình thông báo lời nhắc DIR và chờ người dùng gõ vào xâu ký tự đặc tả các file cần liệt kê (chẳng hạn C:\TP*.*):

```
Uses DOS;
Var
  DirInfo: SearchRec;
  Path: string;
  CFile, CDir: word;
Begin
  Write('DIR '); Readln(Path);
  CFile := 0; CDir := 0;
  FindFirst(Path, AnyFile, DirInfo);
  while DosError = 0 do
  begin
    with DirInfo do
    if Attr = Directory then
    begin
      Inc(CDir);
```

```
      Writeln(Name, ' ':15-Length(Name), '<DIR>');
    end else
    begin
      Inc(CFile);
      Writeln(Name, ' ':15-Length(Name), Size:8);
    end;
    FindNext(DirInfo);
  end;
  Writeln('  Total ', CFile, ' file(s) and ', CDir, ' dir(s)');
  Readln;
End.
```

Các kiểu xâu dưới đây được định nghĩa trong unit DOS để chứa các tên đường dẫn, tên file, ... được dùng trong một số chương trình con:

`PathStr = string[79]`	đặc tả file (đường dẫn + tên file)
`DirStr  = string[67]`	đường dẫn thư mục
`NameStr = string[8]`	phần tên của tên file
`ExtStr  = string[4]`	phần mở rộng của tên file

Để tìm kiếm file trong một danh sách thư mục, bạn có thể dùng hàm:

```
FSearch(Path: PathStr; DirList: string): PathStr
```

Hàm sẽ tìm file có tên cho bởi `Path` bắt đầu từ thư mục hiện hành và lần lượt trong các đường dẫn cho bởi `DirList` (các đường dẫn trong danh sách này cần phân cách nhau bằng dấu chấm phẩy), nếu tìm thấy, hàm trả về tên file (cùng với tên đường dẫn chứa nó), nếu không tìm thấy, hàm trả về xâu rỗng.

Thí dụ, giả thiết file TURBO.EXE có trong thư mục C:\TP\BIN, khi đó lời gọi:

```
S := FSearch('TURBO.EXE', 'C:\DOS; C:\WINDOWS; C:\TP; C:\TP\BIN')
```

sẽ trả về cho xâu S nội dung 'C:\TP\BIN\TURBO.EXE'.

Để xử lý các tên file, unit DOS còn xây dựng hàm `FExpand(Path: PathStr)`, trả về một xâu kiểu `PathStr`, xác định tên đầy đủ của file có tên cho bởi `Path` và thủ tục ngược với nó là `FSplit(Path: PathStr; var Dir: DirStr; var Name: NameStr; var Ext: ExtStr)`, tách tên file đầy đủ cho bởi `Path`, thành các phần: Đường dẫn (gửi vào `Dir`), tên (gửi vào `Name`) và mở rộng (gửi vào `Ext`).

1.4. Các phục vụ ngày, giờ

Để lấy và đặt lại ngày của hệ thống, bạn có thể dùng hai thủ tục:

```
GetDate(var Year, Month, Day, DayOfWeek: word);
SetDate(Year, Month, Day: word);
```

Thủ tục `GetDate(var Year, Month, Day, DayOfWeek: word)` lấy ngày của hệ thống gửi vào các biến `Year` (năm), `Month` (tháng), `Day` (ngày), `DayOfWeek` (ngày trong tuần với quy ước 0-chủ nhật, 1-thứ hai, ...), còn thủ tục `SetDate(Year, Month, Day: word)` đặt lại ngày của hệ thống theo các giá trị tham số tương ứng. Thí dụ dưới đây là một chương trình giống như lệnh DATE của DOS. Chương trình hiển thị ngày của hệ thống và đặt lại giá trị ngày mới (theo khuôn dạng Việt nam ngày-tháng-năm):

Để truy cập các thông tin thuộc về hệ thống của file (thuộc tính, ngày giờ cập nhật, kích thước, tên), unit DOS xây dựng kiểu dữ liệu `SearchRec` dùng cho một số thường trình liên quan:

```
Type
  SearchRec = Record
                Fill: array[1..21] of byte;
                Attr: byte;
                Time: LongInt;
                Size: LongInt;
                Name: string[12];
              End;
```

trong đó trường `Attr` là thuộc tính, `Time` là ngày giờ cập nhật, `Size` là kích thước và `Name` là tên của file, riêng trường `Fill` được để dành.

Để duyệt các file, bạn có thể dùng cặp thủ tục sau đây:

```
FindFirst(Path: string; Attr: word; var DirInfo: SearchRec);
FindNext(var DirInfo: SearchRec);
```

trong đó `FindFirst(Path: string; Attr: word; var DirInfo: SearchRec)` trả về cho biến `DirInfo` thông tin của file đầu tiên tìm thấy có đặc tả `Path` và thỏa mãn thuộc tính `Attr`, còn `FindNext(var DirInfo: SearchRec)` trả về cho biến `DirInfo` thông tin của file kế tiếp. Việc duyệt file được xây dựng theo mô hình:

```
FindFirst(Path, Attr, DirInfo);
while DosError = 0 do
begin
  {xử lý file tìm thấy}
  .............
  FindNext(DirInfo);
end;
```

Thí dụ dưới đây là một chương trình tương tự như lệnh DIR của DOS. Khi chạy, chương trình thông báo lời nhắc DIR và chờ người dùng gõ vào xâu ký tự đặc tả các file cần liệt kê (chẳng hạn C:\TP*.*):

```
Uses DOS;
Var
  DirInfo: SearchRec;
  Path: string;
  CFile, CDir: word;
Begin
  Write('DIR '); Readln(Path);
  CFile := 0; CDir := 0;
  FindFirst(Path, AnyFile, DirInfo);
  while DosError = 0 do
  begin
    with DirInfo do
    if Attr = Directory then
    begin
      Inc(CDir);
```

```
Uses DOS;
Const
  Thu: array[0..6] of string[8] = ('Chu nhat', Thu hai', 'Thu ba',
       'Thu tu', 'Thu nam', 'Thu sau', 'Thu bay');
Var
  Ngay, Thang, Nam, Th: word;
Begin
  GetDate(Nam, Thang, Ngay, Th);
  Writeln('Ngay hien tai la:');
  Writeln('  ', Thu[Th], ' ', Ngay, '-', Thang, '-', Nam);
  Writeln('Dat lai gia tri ngay moi:');
  Write('  Ngay: '); Readln(Ngay);
  Write('  Thang: '); Readln(Thang);
  Write('  Nam: '); Readln(Nam);
  SetDate(Nam, Thang, Ngay);
  GetDate(Nam, Thang, Ngay, Th);
  Writeln('Ngay bay gio la:');
  Writeln('  ', Thu[Th], ' ', Ngay, '-', Thang, '-', Nam);
  Readln;
End.
```

Tương tự, unit DOS có hai thủ tục:

```
GetTime(var Hour, Minute, Second, Sec100: word);
SetTime(Hour, Minute, Second, Sec100: word);
```

cho phép lấy giờ hệ thống (thủ tục `GetTime`) và đặt lại giờ của hệ thống (thủ tục `SetTime`) qua các tham số: `Hour` (giờ), `Minute` (phút), `Second` (giây) và `Sec100` (phần trăm giây). Các bạn có thể viết một chương trình giống thí dụ trên để mô phỏng lệnh TIME của DOS.

Các thông tin ngày, giờ được unit DOS gộp chung vào một `LongInt` qua thủ tục nén:

```
PackTime(var TRec: DateTime; var TInt: LongInt);
```

trong đó `DateTime` là kiểu bản ghi được định nghĩa:

```
Type
  DateTime = Record
               Year, Month, Day, Hour, Min, Sec: word;
             End;
```

với các trường `Year` (năm), `Month` (tháng), `Day` (ngày), `Hour` (giờ), `Min` (phút), `Sec` (giây).

Ngược với thủ tục trên là thủ tục giải nén từ một `LongInt` sang một biến kiểu `DateTime`:

```
UnPackTime(TInt: LongInt; var TRec: DateTime);
```

Để lấy thông tin ngày, giờ của một file, bạn dùng cặp thủ tục:

```
GetFTime(var F; var TInt: LongInt);
UnPackTime(TInt: LongInt; var TRec: DateTime);
```

trong đó thủ tục đầu lấy ngày giờ của file xác định bởi `F` (biến file kiểu bất kỳ) gửi vào biến `TInt`, còn thủ tục sau giải nén `TInt` gửi vào các trường của bản ghi `TRec`.

Để đặt lại thông tin ngày giờ của một file, bạn dùng cặp thủ tục:

```
  PackTime(var TRec: DateTime; var TInt: LongInt);
  SetFTime(var F; TInt: LongInt);
```

trong đó thủ tục đầu nén các thông tin ghi trong các trường của bản ghi `TRec` vào biến `TInt` và thủ tục sau thiết lập ngày giờ của file `F` theo giá trị `TInt`. Chú ý rằng, các thủ tục `GetFTime` và `SetFTime` phải được dùng sau khi đã mở file.

Chương trình dưới đây hiển thị ngày giờ cũ của file lên màn hình và cho phép thiết lập ngày giờ mới:

```
Uses DOS;
Var F: File;
    Name: string;
    TInt: LongInt;
    TRec: DateTime;
Begin
  Write('Ten File: '); Readln(Name);
  Reset(F);
  GetFTime(F, TInt);
  UnPackTime(TInt, TRec);
  Writeln('Ngay gio cu');
  with TRec do
  begin
    Writeln('  Ngay: ', Day, '-', Month, '-', Year);
    Writeln('  Gio: ', Hour, ':', Min, ':', Sec);
  end;
  Writeln('Thiet lap ngay gio moi');
  with TRec do
  begin
    Write('  Ngay: '); Readln(Day);
    Write('  Thang: '); Readln(Month);
    Write('  Nam: '); Readln(Year);
    Write('  Gio: '); Readln(Hour);
    Write('  Phut: '); Readln(Min);
    Write('  Giay: '); Readln(Sec);
  end;
  PackTime(TRec, TInt);
  SetFTime(F, TInt);
  Close(F);
  Write('Da thiet lap ngay gio moi cho file');
  Readln;
End.
```

1.5. Các phục vụ ổ đĩa

Unit DOS chứa hai hàm xử lý đĩa là

```
  DiskSize(Drive: byte): LongInt;
```

trả về kích thước của đĩa (tính theo byte) xác định bởi tham số Drive, và hàm

```
  DiskFree(Drive: byte): LongInt;
```

trả về số byte còn trống của đĩa xác định bởi tham số `Drive`. Các giá trị của `Drive` là 0 (ổ đĩa hiện hành), 1 (ổ đĩa A), 2 (ổ đĩa B), ...

Thí dụ.

```
Uses DOS;
Var Drive: byte;
    DriveName: char;
Begin
  Write('Cho biet ten o dia: '); Readln(DriveName);
  Drive := Ord(Upcase(DriveName))-64;
  Writeln('Kich thuoc cua dia (bytes): ', DiskSize(Drive));
  Writeln('So byte con trong cua dia: ', DiskFree(Drive));
  Readln;
End.
```

1.6. Thông tin môi trường

Trong quá trình làm việc, DOS thiết lập một số giá trị môi trường qua một số xâu ký tự có dạng `name=value`, tromg đó `name` là tên biến môi trường còn `value` là giá trị tương ứng (thí dụ `Path=C:\;C:\Dos;C:\Windows`, `Comspec=C:\Command.com`). Để lấy những thông tin này, unit DOS cung cấp các hàm:

```
EnvCount: integer;
```

trả về số xâu ký tự môi trường đã được thiết lập trong DOS. Nội dung các xâu này được trả về bởi hàm

```
EnvStr(i: integer): string;
```

trong đó chỉ số `i` được bắt đầu từ 1 cho đến `EnvCount`. Nếu `i` nằm ngoài phạm vi này thì hàm trả về xâu rỗng.

Để lấy phần giá trị của một biến môi trường, bạn có thể dùng hàm

```
GetEnv(Name: string): string;
```

trong đó `Name` là tên biến môi trường tương ứng. Nếu biến môi trường đưa ra không tồn tại, hàm trả về xâu rỗng. Thí dụ DOS đã có xâu môi trường `Comspec=C:\Command.com`, khi đó `GetEnv('Comspec')` sẽ trả về xâu `C:\Command.com`.

Chương trình dưới đây hiển thị một số thông tin môi trường mà DOS đã thiết lập:

```
Uses DOS;
Var i: integer;
Begin
  Writeln('Thong tin moi truong');
  for i := 1 to EnvCount do Writeln('  ', i, ') ', EnvStr(i));
  Writeln('Gia tri tra ve cua ham GetEnv(...)');
  Writeln('  GetEnv(''COMSPEC'') = ', GetEnv('COMSPEC'));
  Writeln('  GetEnv(''PATH'') = ', GetEnv('PATH'));
  Writeln('  GetEnv(''TEMP'') = ', GetEnv('TEMP'));
  Readln;
End.
```

1.7. Thực hiện chương trình ngoài

Unit DOS thiết kế thủ tục:

```
Exec(Path, CmdLine: string);
```

cho phép tạm thời thoát khỏi chương trình hiện hành để thực hiện một chương trình khác trong môi trường DOS, sau đó lại quay về chương trình đang thực hiện. Tham trị `Path` xác định tên chương trình cần thực hiện (kể cả đường dẫn), tham trị `CmdLine` là dòng lệnh cần chuyển tới cho chương trình này (có thể rỗng).

Để thực hiện được chương trình trong đó có dùng thủ tục `Exec`, cần giảm thiểu tối đa vùng nhớ Heap cung cấp cho chương trình (để dành chỗ cho chương trình ngoài) bằng cách dùng dẫn hướng biên dịch `{$M StackSize, HeapMin, HeapMax}` ngay đầu chương trình, trong đó `StackSize` là lượng vùng nhớ cho Stack, `HeapMin` là lượng vùng nhớ nhỏ nhất cho Heap và `HeapMax` là lượng vùng nhớ lớn nhất cho Heap. Nếu chương trình không dùng cấp phát động, bạn nên để các giá trị `HeapMin` và `HeapMax` bằng 0. Thí dụ dẫn hướng `{$M 1024, 0, 0}` cho 1 Kb stack và không byte nào cho Heap. Ngoài ra cần gọi thủ tục `SwapVectors` (cũng thuộc unit DOS) trước và ngay sau lời gọi thủ tục `Exec` để đảm bảo an toàn bảng vectơ ngắt. Nếu không đủ bộ nhớ để chạy `Exec`, biến `DosError` sẽ trả về mã lỗi 8.

Thí dụ dưới đây minh họa cách dùng `Exec`:

```
{$M 1024, 0, 0}
Uses DOS;
Var Path, CmdLine: string;
Begin
  Write('Ten chuong trinh can thuc hien (ke ca duong dan): ');
  Readln(Path);
  Write('Dong lenh chuyen toi chuong trinh: ');
  Readln(CmdLine);
  Writeln('Chay EXEC ...');
  SwapVectors;
  Exec(Path, CmdLine);
  SwapVectors;
  Writeln('Quay ve chuong trinh hien hanh ...');
  if DosError <> 0 then
    Writeln('DOS co loi ', DosError)
  else
    Writeln('Ok');
End.
```

Nếu tham trị `Path` là tên file COMMAND.COM thì tham trị `CmdLine` có thể là bất cứ lệnh nội trú nào. Thí dụ lời gọi:

```
Exec('C:\Command.com', '/C Dir *.*');
```

sẽ thực hiện lệnh `Dir *.*` của DOS. Khóa chuyển `'/C'` ở đây là cần thiết để hệ thống không nạp lại COMMAND.COM. Chú ý, có thể `Exec` không thực hiện được do đường dẫn đến COMMAND.COM không đúng, vì thế để chắc chắn tìm được file này, người ta thường truyền cho `Path` giá trị `GetEnv('Comspec')`. Tham trị `CmdLine` có thể rỗng, khi

đó, lệnh nội trú được chờ đọc vào tại dấu nhắc của DOS. Điều này cho phép bạn có thể thiết kế lời gọi DOS trong chương trình của mình như thí dụ dưới đây:

```
{$M 1024, 0, 0}
Uses DOS, CRT;
  function Menu: char;
  begin
    ClrScr;
    Writeln(' ': 31, 'MAIN MENU');
    Writeln;
    Writeln(' ':30, '(A)nything');
    Writeln(' ':30, '(D)OS shell');
    Writeln(' ':30, '(Q)uit');
    Writeln;
    Write(' ':28, 'Enter a choice: ');
    Menu := Upcase(ReadKey);
  end;
  procedure Anything;
  begin
    ClrScr;
    Writeln('This is Anything');
    Write('Press Enter to continue ... ');
    readln;
  end;
  procedure DosShell;
  begin
    ClrScr;
    Writeln('This is DOS. Type EXIT to return program ...');
    SwapVectors;
    Exec(GetEnv('Comspec'), '');
    SwapVectors;
    if DosError <> 0 then
      Writeln('Could not execute COMMAND.COM');
  end;
Begin
  repeat
    case Menu of
      'A': Anything;
      'D': DosShell;
      'Q': Halt;
    end;
  until False;
End.
```

2. Xử lý các ngắt

2.1. Ngắt và vai trò của ngắt

Ngắt là những thường trình của DOS và BIOS nhằm điều khiển và xử lý các thiết bị ngoại vi như màn hình, bàn phím, chuột, máy in, ... được thường trú trong bộ nhớ để có thể phục vụ bất cứ lúc nào khi được gọi. Mặc dù Turbo Pascal cung cấp khá nhiều các dịch vụ thông thường (như ghi các dữ liệu ra đĩa, màn hình, đọc các dữ liệu từ đĩa, bàn phím, nhận diện phím được ấn, truy cập thuộc tính màn hình ...) làm cho bạn hầu như không phải quan tâm trực tiếp đến các ngắt, nhưng các dịch vụ này thường không chi tiết, kém linh hoạt và chưa đầy đủ, khiến cho trong một số tình huống, bạn không thể thực hiện được ý định của mình qua các hàm, thủ tục sẵn có của Turbo Pascal (chẳng hạn truy cập dữ liệu tại một rãnh bất kỳ của đĩa, thay đổi kích thước con trỏ màn hình, điều khiển chuột, ...). Unit DOS cung cấp cho người dùng kiểu dữ liệu thanh ghi `Registers` và hai thủ tục `MsDos`, `Intr` cho phép thực hiện bất cứ các ngắt nào của DOS và BIOS.

Nhờ việc xử lý các ngắt, chương trình của bạn có thể can thiệp sâu hơn vào hệ thống, tăng được tốc độ cũng như hiệu quả của chương trình, đặc biệt, ngắt còn là một công cụ mạnh để tạo các thư viện lập trình có liên quan nhiều đến các chức năng của DOS, tiện dụng cho nhiều người không cần am hiểu nhiều về hệ thống.

2.2. Các thanh ghi

Các bộ vi xử lý của họ 8088 (kể cả 8086 và 80286) có 14 thanh ghi dùng để thực hiện các lệnh. Mỗi thanh ghi có chiều dài 16 bit (2 byte). Các thanh ghi này có thể chia thành 4 nhóm như sau:

+ Đầu tiên là 4 thanh ghi đa dụng AX, BX, CX, DX dùng để cất giữ tạm thời dữ liệu trong tiến trình xử lý. Chúng thường được dùng để gọi các phục vụ của DOS và BIOS. Mỗi thanh ghi này lại được chia thành 2 thanh ghi nhỏ, mỗi thanh 1 byte (tương ứng với byte thấp và byte cao), thí dụ AX gồm AL (byte thấp) và AH (byte cao):

AX	AL	AH	Accumulator (tích lũy)
BX	BL	BH	Base (cơ sở)
CX	CL	CH	Count (đếm)
DX	DL	DH	Data (dữ liệu)

+ Sau đó là 4 thanh ghi đoạn (segment) CS, DS, SS, ES, trong đó CS chứa đoạn chương trình, DS chứa đoạn dữ liệu, SS chứa đoạn stack và ES chứa đoạn tạm thời cho các thao tác đặc biệt:

CS	Code Segment (đoạn mã lệnh)
DS	Data Segment (đoạn dữ liệu)
SS	Stack Segment (đoạn ngăn xếp)
ES	Extra Segment (đoạn thêm)

+ Tiếp theo là 5 thanh ghi độ lệch (offset) IP, SP, BP, SI, DI. Chúng được dùng kèm với các thanh ghi đoạn để xác định một vị trí nhớ:

IP	Instruction Pointer (con trỏ lệnh)
SP	Stack Pointer (con trỏ ngăn xếp)
BP	Base Pointer (con trỏ cơ sở)
SI	Source Index (chỉ số nguồn)
DI	Destination Index (chỉ số đích)

+ Cuối cùng là một thanh ghi cờ FLAGS chứa các thông tin về trạng thái của lệnh thực hiện sau cùng, thường dùng để nhận diện lỗi:

FLAGS

2.3. Kiểu dữ liệu thanh ghi

Để có thể truy cập vào các thanh ghi, unit DOS định nghĩa kiểu dữ liệu `Registers` như sau:

```
Type
  Registers = Record
                case integer of
                  0: (AX, BX, CX, DX, BP, SI, DI, DS, ES: word);
                  1: (AL, AH, BL, BH, CL, CH, DL, DH: byte);
              End;
```

Kiểu dữ liệu `Registers` chỉ chứa các thanh ghi của CPU được các phục vụ của DOS và BIOS sử dụng. Các thanh ghi đa dụng AX, BX, CX, DX được dùng nhiều nhất. Trong các thanh ghi đoạn, các thanh ghi CS (đoạn mã lệnh) và SS (đoạn ngăn xếp) chứa các dữ liệu cực kỳ quan trọng nên Turbo Pascal không cho phép thâm nhập, còn các thanh ghi DS (đoạn dữ liệu) và ES (đoạn thêm) thỉnh thoảng vẫn được dùng để truyền địa chỉ đoạn. Trong các thanh ghi độ lệch, Turbo Pascal chỉ cho phép dùng các thanh ghi BP (con trỏ cơ sở), SI (chỉ số nguồn) và DI (chỉ số đích), còn các thanh ghi IP (con trỏ lệnh) và SP (con trỏ ngăn xếp) không bao giờ dùng đến. Khi dùng Turbo Pascal, thanh ghi cờ không cần thiết cho các lời gọi DOS và BIOS vì chúng trả về mã lỗi nằm ở một trong các thanh ghi đa dụng.

Để có thể truy cập vào từng byte của các thanh ghi đa dụng, kiểu `Registers` được tổ chức là một kiểu bản ghi biến thể, trong đó biến thể thứ nhất gồm các trường kiểu `word`, biểu diễn toàn bộ các thanh ghi, còn biến thể thứ hai gồm các trường kiểu `byte` biểu diễn byte thấp và byte cao của các thanh ghi đa dụng.

2.4. Các thủ tục gọi ngắt

Có tất cả 256 ngắt được đánh số từ 0, gọi là số hiệu ngắt. Mỗi ngắt lại gồm nhiều chức năng, được đánh số để phân biệt, gọi là số hiệu chức năng. Trong các tài liệu về DOS,

người ta thường dùng cách viết hexa (hệ 16) để biểu diễn các số hiệu này. Trong các thủ tục của Turbo Pascal, khi cần truyền các số hiệu này vào các tham số tương ứng, bạn có thể dùng cách viết hexa (có ký hiệu $ đằng trước) hoặc cách viết thập phân thông thường.

Unit DOS xây dựng hai thủ tục để gọi ngắt, đó là thủ tục `MsDos` và `Intr`. Thủ tục `MsDos` được dùng riêng cho các phục vụ DOS (có số hiệu ngắt 21 hexa hay 33 thập phân) và thủ tục `Intr` được dùng cho các phục vụ DOS và BIOS nói chung.

Thủ tục `MsDos` có cú pháp:

```
MsDos(var Regs: Registes);
```

trong đó tham biến `Regs` dùng để nhận các giá trị đầu vào (trước khi gọi thủ tục), cũng như dùng để gửi các giá trị đầu ra (sau khi gọi thủ tục).

Thủ tục `Intr` có cú pháp:

```
Intr(IntNo: byte; var Regs: Registes);
```

trong đó tham trị `IntNo` nhận số hiệu ngắt còn tham biến `Regs` được dùng giống như trong thủ tục trước. Có thể xem lời gọi thủ tục `MsDos` là trường hợp riêng của lời gọi thủ tục `Intr` với số hiệu ngắt 21 hexa.

Để dùng các ngắt, bạn cần chuẩn bị biến `Regs` kiểu thanh ghi rồi tiến hành các bước:

- Đặt số hiệu chức năng cần dùng của ngắt vào trường `AH` của biến thanh ghi `Regs`.
- Tùy theo từng chức năng, có thể cần nạp một số giá trị đòi hỏi vào một số trường tương ứng của biến thanh ghi `Regs`.
- Gọi ngắt bằng lời gọi `Intr(IntNo, Regs)`, trong đó `IntNo` là số hiệu ngắt.
- Kết quả trả lại, nếu có, sẽ được gửi vào một số trường của biến thanh ghi `Regs`.

Trong mọi trường hợp, nếu số hiệu ngắt là 21 hexa (phục vụ DOS) thì có thể thay lời gọi `Intr($21, Regs)` bằng lời gọi `MsDos(Regs)`.

Như vậy, để dùng các ngắt, bạn cần phải biết các chức năng của ngắt và các giá trị tương ứng cần nạp hay lấy ra của các thanh ghi. Điều này, bạn có thể tham khảo trong các tài liệu về ngắt của DOS. Trong các tài liệu này, mỗi lời giới thiệu về một chức năng đều chỉ rõ:

- Số hiệu ngắt.
- Số hiệu chức năng.
- Các giá trị cần nạp vào các thanh ghi.
- Các giá trị trả về ở các thanh ghi.

trong đó các số hiệu ngắt cũng như các số hiệu chức năng thường được viết dưới hệ hexa.

Chẳng hạn lời giới thiệu chức năng lấy và đặt ngày hệ thống như sau:

Lấy ngày hệ thống

Số hiệu ngắt: 21

Số hiệu chức năng: 2A

Giá trị cần nạp:

AH ← 2A (số hiệu chức năng)

Giá trị trả về:

AL → ngày trong tuần (0: chủ nhật, 1: thứ hai, ...)

CX → năm

DH → tháng

DL → ngày

Đặt ngày hệ thống

Số hiệu ngắt: 21

Số hiệu chức năng: 2B

Giá trị cần nạp:

AH ← 2Ah (số hiệu chức năng)

CX ← năm

DH ← tháng

DL ← ngày

Giá trị trả về: không có

Từ những ngắt trên bạn có thể viết lại thủ tục `GetDate` và `SetDate` mà unit DOS đã cung cấp (xem **1.4.**). Nói chung, các hàm và thủ tục của unit DOS đã được giới thiệu trong phần **1**, đều có thể trực tiếp viết từ các ngắt.

Dưới đây chúng tôi chỉ giới thiệu một số ngắt tiện ích mà chức năng của chúng chưa được thiết kế trong các thư viện thường dùng của Turbo Pascal. Bằng cách tương tự, bạn có thể dùng các ngắt khác với sự hỗ trợ của các tài liệu về ngắt, ngoài ra, bạn có thể dùng những ngắt này để xây dựng thêm các thư viện của riêng mình, nhằm tăng khả năng lập trình ứng dụng trên Turbo Pascal.

2.5. Một số ngắt tiện ích

Một số ngắt tiện ích được trình bày dưới đây nhằm tăng thêm các khả năng giao tiếp với hệ thống của chương trình Turbo Pascal, bạn có thể biến chúng thành các thư viện để tiện dùng lâu dài.

2.5.1. Lấy và đặt lại kích thước con trỏ màn hình

Khi chương trình chờ gõ phím, trên màn hình (văn bản) xuất hiện một điểm sáng nhấp nháy, gọi là con trỏ màn hình (cursor). Kích thước của con trỏ được ngầm định sẵn. Bạn có thể lấy thông tin kích thước của con trỏ hiện thời hoặc thay đổi kích thước này (dày lên, mỏng đi, hoặc biến mất) bằng cách dùng ngắt số hiệu 10 hexa.

Để lấy thông tin kích thước con trỏ hiện thời, bạn dùng chức năng 3 (gán cho AH giá trị 3) sau đó gọi ngắt 10 hexa. Khi đó thanh ghi CH sẽ chứa giá trị dòng quét trên còn thanh ghi CL sẽ chứa giá trị dòng quét dưới của con trỏ. Giá trị dòng quét càng nhỏ thì

càng sát đỉnh màn hình. Kích thước ngầm định của con trỏ đối với những màn hình thông dụng hiện nay thường là CH = 13 và CL = 14.

Để đặt lại kích thước con trỏ, bạn dùng chức năng 1 (gán cho AH giá trị 1), sau đó đặt các giá trị tương ứng cho CH (dòng quét trên) và CL (dòng quét dưới) rồi gọi ngắt 10h. Khoảng cách giữa dòng quét trên và dòng quét dưới càng lớn thì con trỏ càng dày. Một số giá trị của CH và CL sẽ làm con trỏ biến mất (chẳng hạn với màn hình VGA, khi đặt giá trị dòng quét trên bằng 32, con trỏ sẽ ẩn).

Chương trình dưới đây minh họa việc dùng các chức năng trên. Ban đầu chương trình thông báo kích thước con trỏ hiện thời, sau đó là vòng lặp đặt lại các kích thước khác nhau của con trỏ dựa vào việc nhập các giá trị dòng quét từ bàn phím. Vòng lặp (và chương trình) sẽ kết thúc khi các dòng quét nhận lại giá trị ban đầu:

```
Uses CRT, DOS;
Var Regs: Registers;
    High, Low, OldHigh, OldLow: byte;
Begin
  ClrScr;
  Writeln('Gia tri ban dau cua cursor');
  Regs.AH := 3;          {đặt chức năng 3}
  Intr($10, Regs);       {gọi ngắt 10h}
  OldHigh := Regs.CH; OldLow := Regs.CL;      {lấy thông tin cursor}
  Writeln('  dong quet tren: ', OldHigh);
  Writeln('  dong quet duoi: ', OldLow);
  repeat
    Writeln('Dat lai kich thuoc cursor');
    Write('  dong quet tren: '); Readln(High);
    Write('  dong quet duoi: '); Readln(Low);
    Regs.AH := 1;        {đặt chức năng 1}
    Regs.CH := High; Regs.CL := Low;          {đặt kích thước cursor}
    Intr($10, Regs);     {gọi ngắt 10h}
  until (High = OldHigh) and (Low = OldLow);
End.
```

Dựa vào kết quả của chương trình này, bạn có thể xây dựng một thư viện riêng (dưới dạng unit) về con trỏ màn hình để tiện dùng sau này. Dưới đây là một vài gợi ý về nội dung thiết kế (chúng tôi chỉ viết các khai báo của phần INTERFACE, còn phần cài đặt IMPLEMENTATION dành cho các bạn):

```
Unit CURSOR;
Interface
  Procedure GetCursor(var High, Low: byte);
  {lấy thông tin con trỏ, gửi dòng quét trên vào High và dòng quét dưới vào Low}
  Procedure SetCursor(High, Low: byte);
  {đặt con trỏ với dòng quét trên là High và dòng quét dưới là Low}
  Procedure BigCursor;
  {đặt con trỏ dày}
  Procedure SmallCursor;
  {đặt con trỏ mỏng}
```

```
  Procedure NormalCursor;
  {đặt con trỏ bình thường}
  Procedure HideCursor;
  {ẩn con trỏ}
Implementation
  Uses DOS;
  ............
  ............
  ............
End.
```

Để thử unit trên, bạn có thể dùng chương trình dưới đây:

```
Uses CURSOR;
Var High, Low: byte;
Begin
  GetCursor(High, Low);
  Write('Kich thuoc cursor hien thoi: ', High, ', ', Low, ' '); Readln;
  SetCursor(0, 3);
  Write('Dat lai kich thuoc cursor la 0, 3: '); Readln;
  BigCursor;
  Write('Cursor day: '); Readln;
  SmallCursor;
  Write('Cursor mong: '); Readln;
  HideCursor;
  Write('Cursor an: '); Readln;
  NormalCursor;
  Write('Cursor binh thuong: '); Readln;
End.
```

2.5.2. Lấy thông tin đĩa

Để đọc cấu trúc của đĩa, bạn dùng ngắt số hiệu 21 hexa (phục vụ DOS) với chức năng 36 hexa. Sau khi gán thanh ghi AH giá trị 36 hexa, bạn cần chỉ định ổ đĩa bằng việc gán giá trị tương ứng cho thanh ghi DL (0: ổ đĩa hiện hành, 1: ổ đĩa A, 2: ổ đĩa B, ...) và sau đó gọi ngắt bằng `Intr($21, Regs)` hoặc `MsDos(Regs)`. Kết quả trả về là các thông tin của đĩa được chứa trong các thanh ghi sau:

AX	số sector trong mỗi cluster
CX	kích thước một sector (bytes)
DX	tổng số cluster
BX	số cluster còn trống

Từ các thông tin này bạn có thể suy ra các thông tin khác, chẳng hạn kích thước của đĩa (bytes) hoặc kích thước vùng trống của đĩa (bytes). Nếu chỉ định ổ đĩa không hợp lệ, giá trị FFFFh sẽ được trả về cho thanh ghi AX.

Thí dụ dưới đây minh họa cách dùng ngắt vừa nêu:

```
Uses CRT, DOS;
var Regs: Registers;
    Drive: char;
```

```
Begin
  ClrScr;
  Write('Cho biet ten o dia: '); Readln(Drive);
  Regs.AH := $36;
  Regs.DL := Ord(Upcase(Drive))-64;
  Intr($21, Regs);    {hoặc MsDos(Regs)}
  if Regs.AX = $FFFF then
    Writeln('Invalid drive')
  else
  with Regs do
  begin
    Writeln('So sector tren mot cluster: ', AX);
    Writeln('Kich thuoc cua mot sector (bytes): ', CX);
    Writeln('Tong so cluster tren dia: ', DX);
    Writeln('So cluster con trong: ', BX);
    Writeln('Kich thuoc cua dia (bytes): ', LongInt(AX)*DX*CX);
    Writeln('So byte con trong cua dia: ', LongInt(AX)*BX*CX);
  end;
  Readln;
End.
```

2.5.3. Nhận diện các phím đặc biệt

Mặc dù Turbo Pascal đã cung cấp một số hàm nhận diện phím khá mạnh như `KeyPressed`, `ReadKey` (unit CRT) nhưng những hàm này không nhận biết được một số phím đặc biệt của bàn phím như Right Shift, Left Shift, Ctrl, Alt, Num Lock, Caps Lock, Scroll Lock. Muốn nhận biết các phím này, bạn phải dùng ngắt số hiệu 16 hexa với chức năng 2. Kết quả của ngắt trả về thanh ghi AL, trong đó mỗi bit trong byte này cho biết trạng thái của một phím (1- bật, 0 - tắt) theo thứ tự từ bit thấp đến bit cao: Right Shift (bit 1), Left Shift (bit 2), Ctrl (bit 3), Alt (bit 4), Scroll Lock (bit 5), Num Lock (bit 6), Caps Lock (bit 7) và Insert (bit 8). Chú ý rằng các phím Scroll Lock, Num Lock, Caps Lock và Insert mang tính chẵn lẻ (phím giữ nguyên trạng thái cho đến khi ấn lại), còn các phím khác chỉ được bật khi được ấn.

Chương trình dưới đây thông báo lên màn hình trạng thái của 8 phím trên theo thứ tự đã nêu. Một dấu X sẽ xuất hiện tại cột ứng với phím được bật. Chương trình kết thúc khi ấn tổ hợp phím Alt+Ctrl:

```
Uses CRT, DOS;
Const
  Code: array[1..8] of byte = (1, 2, 4, 8, 16, 32, 64, 128);
  KeyName: array[1..8] of string[9] = ('  RShift ', '  LShift ',
           '   Ctrl  ', '   Alt   ', ' ScrLock ', ' NumLock ',
           'CapsLock ', '  Insert ');
Var i: integer;
    BitOn: array[1..8] of boolean;
    Regs: Registers;
    St: string;
```

```
Begin
  ClrScr;
  Writeln('          Test cac phim dac biet (an Ctrl+Alt de ket thuc)');
  Writeln;
  for i := 1 to 8 do Write(KeyName[i]);
  Writeln;
  repeat
    Regs.AH := 2;
    Intr($16, Regs);
    St := '';
    for i := 1 to 8 do
    begin
      BitOn[i] := (Regs.AL and Code[i]) > 0;
      if BitOn[i] then St := St+'    X    '
                  else St := St+'         ';
    end;
    GotoXY(1, 4); Write(St);
    Delay(100);
  until BitOn[3] and BitOn[4];
  ClrScr;
End.
```

2.5.4. Điều khiển chuột (mouse)

Việc dùng chuột hiện nay là phổ biến cho mọi chương trình. Tuy nhiên Turbo Pascal không cung cấp một hàm hay thủ tục nào để điều khiển chuột. Dưới đây, chúng tôi trình bày cách dùng ngắt số hiệu 33 hexa với một số chức năng thông dụng để có thể điều khiển chuột trong chương trình.

Khác với những ngắt khác, khi dùng ngắt này, bạn phải đặt số hiệu chức năng vào toàn bộ thanh ghi AX. Dưới đây là một số chức năng thông dụng:

Khởi động chuột

Trước khi dùng chuột bạn phải khởi động nó, bằng cách dùng chức năng 0 của ngắt 33 hexa (gán AX bằng 0 rồi gọi `Intr` với số hiệu `$33`). Nếu chuột đã được cài đặt bởi hệ thống thì việc khởi động thành công và ngắt trả về giá trị FFFF cho AX và trả về số phím (button) của chuột cho BX, nếu chuột chưa được cài đặt bởi hệ thống thì việc khởi động là thất bại và ngắt trả về giá trị 0 cho AX. Chú ý rằng, dù khởi động xong, chuột vẫn ở trạng thái ẩn (không hiện trên màn hình). Hàm lôgic `InitMouse` dưới đây làm nhiệm vụ khởi động chuột. Hàm sẽ trả về `True` nếu chuột khởi động thành công và trả về `False` nếu trái lại:

```
Function InitMouse: boolean;
{khởi tạo mouse}
Var Regs: Registers;
Begin
  Regs.AX := 0;
```

```
  Intr($33, Regs);
  InitMouse := AX <> 0;
End;
```

Hiện chuột

Để hiện chuột (với điều kiện đã được khởi động), bạn dùng chức năng 1 của ngắt đã nêu. Thủ tục `ShowMouse` dưới đây dùng để hiện chuột:

```
Procedure ShowMouse;
{hiện mouse}
Var Regs: Registers;
Begin
  Regs.AX := 1;
  Intr($33, Regs);
End;
```

Khi chuột được hiện, hình dáng của nó được ngầm định tùy theo chế độ màn hình. Với màn hình văn bản chuẩn, chuột được hiện dưới hình dáng một hộp nhỏ có kích thước 8x8, với màn hình đồ họa, chuột được hiện với hình dáng mũi tên. Hình chuột trên màn hình được gọi là con trỏ chuột (mouse pointer). Vị trí con trỏ chuột trên màn hình được thay đổi tương ứng với việc di chuyển chuột trên bàn làm việc.

Ẩn chuột

Con trỏ chuột có thể không hiện trên màn hình dù chuột đang hiện diện. Để ẩn chuột, bạn dùng chức năng 2 của ngắt 33 hexa. Thủ tục `HideMouse` dưới đây dùng để ẩn chuột:

```
Procedure HideMouse;
{ẩn mouse}
Var Regs: Registers;
Begin
  Regs.AX := 2;
  Intr($33, Regs);
End;
```

Lấy trạng thái chuột

Trạng thái chuột bao gồm thông tin về tọa độ con trỏ chuột trên màn hình và trạng thái các phím của chuột (đang nhấn hoặc nhả). Để lấy các thông tin này, bạn dùng chức năng 3 của ngắt 33 hexa. Kết quả trả lại được gửi vào các thanh ghi đa dụng như sau:

BX chứa trạng thái các phím, mỗi bit (từ thấp đến cao) mô tả một phím:
+ Bit 1 → phím trái
+ Bit 2 → phím phải
+ Bit 3 → phím giữa
+ Các bit còn lại không dùng (0)

CX chứa hoành độ (X) chuột

DX chứa tung độ (Y) chuột

Trạng thái các phím của chuột được nhận biết từ giá trị của BX qua các tổ hợp của 3 bit thấp. Chẳng hạn nếu BX có giá trị 1 thì có nghĩa phím trái đang nhấn còn các phím kia đang nhả, nếu BX có giá trị 3 thì có nghĩa cả phím trái và phải đang được nhấn còn phím giữa đang nhả, ... Nếu chuột không có phím giữa thì bit 3 không dùng đến (có giá trị 0).

Tọa độ chuột liên quan đến tọa độ màn hình tùy thuộc chế độ hiện hành. Nếu màn hình đang ở chế độ đồ họa thì tọa độ chuột được hiểu là tọa độ của điểm đầu mũi tên của con trỏ chuột và được tính theo từng pixel của chế độ đó. Chẳng hạn nếu màn hình đang ở chế độ đồ họa VGA-Hi (phân giải cao 640×480) thì hoành độ chuột biến thiên từ 0 đến 639 và tung độ chuột biến thiên từ 0 đến 479. Nếu màn hình đang ở chế độ văn bản chuẩn 80×25 thì tọa độ chuột được hiểu là tọa độ góc trên bên trái của hộp sáng hiển thị nó. Vì kích thước của hộp này là 8×8 nên dễ dàng tính được hoành độ chuột trong khoảng từ 0 đến 80×8-8 = 632 và tung độ chuột trong khoảng từ 0 đến 25×8-8 = 192. Tại chế độ này, do con trỏ chuột di chuyển theo từng đơn vị cột, dòng mần hình nên tọa độ chuột sẽ thay đổi theo từng nhóm 8 đơn vị, từ đó ta nhận được công thức biến đổi giữa tọa độ chuột và tọa độ cột, dòng của màn hình trong chế độ văn bản như sau (ký hiệu U, V là tọa độ màn hình và X, Y là tọa độ chuột):

U = X div 8 + 1 X = (U-1)*8

V = Y div 8 + 1 Y = (V-1)*8

Thủ tục `GetMouse` dưới đây trả hoành độ chuột về tham biến `X`, tung độ chuột về tham biến `Y` và trang thái của phím trái về tham biến lôgic `Left` (`True` nếu phím trái đang được nhấn và `False` nếu trái lại):

```
Procedure GetMouse(var X, Y: integer; var Left: boolean);
{lấy vị trí mouse vào X, Y, lấy trạng thái nút trái mouse vào Left}
Var Regs: Registers;
Begin
  Regs.AX := 3;
  Intr($33, Regs);
  X := Regs.CX; Y := Regs.DX;
  Left := (Regs.BX and 1) > 0;
End;
```

Đặt con trỏ chuột vào một vị trí cho trước

Lúc mới khởi động, vị trí con trỏ chuột được ngầm định ở trung tâm màn hình sau đó được thay đổi tương ứng với việc di chuyển chuột bên ngoài. Tuy nhiên bạn có thể đặt con trỏ chuột vào những vị trí tùy ý bằng những câu lệnh trong chương trìmh. Để làm điều này bạn dùng chức năng 4 của ngắt 33 hexa và đặt vào thanh ghi CX giá trị hoành độ và thanh ghi DX giá trị tung độ mà con trỏ chuột cần chuyển đến.

Thủ tục `SetMouse` dưới đây sẽ đặt con trỏ chuột vào tọa độ được xác định bởi các tham trị `X` (hoành độ) và `Y` (tung độ):

```
Procedure SetMouse(X, Y: integer);
{đặt mouse vào vị trí X, Y}
Var Regs: Registers;
```

```
Begin
  Regs.AX := 4;
  Regs.CX := X;
  Regs.DX := Y;
  Intr($33, Regs);
End;
```

Giới hạn khoảng hoạt động của chuột

Bình thường, phạm vi hoạt động của chuột là toàn màn hình. Tuy nhiên, trong một số tình huống, bạn cần hạn chế khoảng hoạt động của nó. Chức năng 7 và 8 của ngắt 33 hexa cho phép bạn giới hạn khoảng ngang (chức năng 7) và khoảng dọc (chức năng 8) của con trỏ chuột trên màn hình.

Nếu dùng chức năng 7, bạn cần đặt giá trị hoành độ nhỏ nhất của chuột vào thanh ghi CX và giá trị hoành độ lớn nhất của chuột vào thanh ghi DX. Nếu dùng chức năng 8, thì giá trị các thanh ghi này được đặt tương tự với ý nghĩa tương ứng cho tung độ.

Thủ tục `RangeMouse` dưới đây đặt khoảng hoạt động của chuột vào một miền chữ nhật trên màn hình có tọa độ góc trên trái cho bởi các tham trị `X1`, `Y1` và tọa độ góc dưới phải cho bởi các tham trị `X2`, `Y2`:

```
Procedure RangeMouse(X1, Y1, X2, Y2: integer);
{đặt phạm vi mouse trong vùng X1, Y1, X2, Y2}
Var Regs: Registers;
Begin
  Regs.AX := 7;      {giới hạn ngang}
  Regs.CX := X1;
  Regs.DX := X2;
  Intr($33, Regs);
  Regs.AX := 8;      {giới hạn dọc}
  Regs.CX := Y1;
  Regs.DX := Y2;
  Intr($33, Regs);
End;
```

Các chức năng của ngắt 33 hexa khá phong phú. Bằng cách xem xét tương tự, bạn có thể tham khảo ngắt 33 hexa nói riêng và các ngắt khác nói chung trong các tài liệu liên quan của DOS. Điều quan trọng là từ các chức năng thông dụng của ngắt, bạn cần tổ chức thành các thư viện chuyên biệt để tiện dùng lâu dài về sau.

Chẳng hạn bạn có thể gói các hàm và thủ tục đã xây dựng ở trên thành một unit với tên là MOUSE. Khéo dùng thư viện này (mặc dù còn đơn giản), bạn cũng có thể viết những chương trình, trong đó có những giao diện đồ họa-chuột thú vị.

```
Unit MOUSE;
Interface
  Function InitMouse: boolean;
  Procedure ShowMouse;
  Procedure HideMouse;
  Procedure GetMouse(var X, Y: integer; var Left: boolean);
```

```
  Procedure SetMouse(X, Y: integer);
  Procedure RangeMouse(X1, Y1, X2, Y2: integer);
Implementation
  Uses DOS;
  {viết lại các hàm, thủ tục ở trên}
  ......................
End.
```

Dưới đây là một chương trình minh họa unit MOUSE. Chương trình thể hiện chuột và tọa độ tương ứng của nó trong các chế độ màn hình khác nhau. Ban đầu là chế độ đồ họa VGA-Hi (640×480). sau đó tùy vào người dùng ấn phím 'T', 'H', 'M', 'L' mà chương trình chuyển sang các chế độ tương ứng (T: Text 80×25, H: VGA-Hi 640×480, M: VGA-Med 640×350, L: VGA-Lo 640×200). Trong quá trình di chuyển chuột, tọa độ tương ứng của nó được hiện trên đỉnh màn hình. Mỗi khi nhấn phím trái chuột, nếu đang ở trong các chế độ đồ họa, thì một hình tròn nhỏ có tâm tại đúng tọa độ chuột sẽ được vẽ, nếu ở trong chế độ văn bản, một chữ 'M' sẽ được viết đúng tại vị trí cột, dòng mà chuột đang trỏ. Nếu vừa nhấn phím trái vừa kéo rê chuột, bạn có thể dùng chuột như một bút vẽ trên màn hình. Chương trình kết thúc khi ấn phím Esc. Ngoài việc dùng unit MOUSE, chương trình còn dùng các unit CRT, GRAPH (của Turbo Pascal), và dùng unit CURSOR (đã xây dựng).

```
Uses CRT, GRAPH, MOUSE, CURSOR;
Var
  Gd, Gm: integer;
  ModeStr: string;
  Ch: char;
  X, Y: integer;
  Left: boolean;
Begin
  if not InitMouse then
  begin
    Write('Mouse chua duoc cai dat. An phim Enter de ket thuc ... ');
    Readln; Halt;
  end;          {mouse đã được khởi động}
  DirectVideo := False;
  Gd := VGA; Gm := 2;    {ngầm định chế độ Graph VGA-Hi}
  InitGraph(Gd, Gm, 'C:\TP\BGI');
  if GraphResult <> 0 then
  begin
    Writeln('Duong dan den cac file BGI khong dung!');
    Write('An phim Enter de ket thuc ... ');
    Readln; Halt;
  end;
  ShowMouse;    {hiện mouse}
  ModeStr := GetModeName(Gm);
  GotoXY(1, 25); Write(ModeStr);
  GotoXY(36, 25);
  Write('H: VGA-Hi / M: VGA-Med / L: VGA-Lo / T: Text');
```

```
  Ch := #0;
  repeat
    GetMouse(X, Y, Left);   {lấy vị trí và trạng thái nút trái mouse}
    GotoXY(60, 1); Write(X:5, Y:5);
    if Ch = #0 then      {graph}
      if Left then       {nhấn nút trái}
      begin
        HideMouse;       {giấu mouse trước khi vẽ hình}
        FillEllipse(X, Y, 5, 5);
        ShowMouse;       {hiện lại mouse}
      end;
    if Ch = 'T' then     {text}
      if Left then       {nhấn nút trái}
      begin
        GotoXY(X div 8+1, Y div 8+1); Write('M');
      end;
    if Keypressed then Ch := Upcase(ReadKey);
    if Ch = 'T' then
    begin
      RestoreCrtMode;    {sang chế độ Text}
      ShowMouse;         {hiện mouse}
      HideCursor;        {giấu cursor}
      Gotoxy(1, 25); Write('Text');
      GotoXY(36, 25);
      Write('H: VGA-Hi / M: VGA-Med / L: VGA-Lo / T: Text');
    end else
    if Ch in ['L', 'M', 'H'] then
    begin
      case Ch of
        'L': Gm := 0;    {VGA-Lo}
        'M': Gm := 1;    {VGA-Med}
        'H': Gm := 2;    {VGA-Hi}
      end;
      Ch := #0;
      SetGraphMode(Gm);  {sang chế độ Gm của Graph}
      ShowMouse;         {hiện mouse}
      ModeStr := GetModeName(Gm);
      GotoXY(1, 25); Write(ModeStr);
      GotoXY(36, 25);
      Write('H: VGA-Hi / M: VGA-Med / L: VGA-Lo / T: Text');
    end;
  until Ch = #27;
  CloseGraph;        {về chế độ Text}
  HideMouse;         {giấu mouse}
  NormalCursor;      {hiện cursor}
  ClrScr;
End.
```

Chương trình dưới đây minh họa một ứng dụng khác của unit MOUSE, đó là khả năng dùng chuột để chọn và tác động lên một đối tượng hình học. Chương trình tạo một nền chữ nhật ở chế độ đồ họa, trong đó có chứa một hình tròn mầu trắng (tượng trưng cho một đối tượng hình học) ở trung tâm. Bạn có thể dùng chuột để chọn hình bằng cách di chuyển chuột sao cho mũi tên của con trỏ chuột chạm vào phạm vi của hình, rồi nhấn phím trái. Khi đó hình sẽ chuyển mầu đỏ báo hiệu đã nhận biết tín hiệu chọn. Nếu lúc này vừa nhấn phím trái, vừa kéo rê chuột, thì hình sẽ được kéo theo. Hình được kéo theo con trỏ chuột cho đến lúc phím trái được nhả, khi đó hình lại được chuyển sang mầu trắng báo hiệu trạng thái không được chọn nữa. Trên đỉnh màn hình là dòng thông báo tọa độ của hình và tọa độ của chuột tại mọi thời điểm tương ứng. Trong chương trình có sử dụng một số kỹ thuật xử lý ảnh bitmap, như cắt, dán hình, để việc kéo hình trên một nền ảnh không làm hỏng nền. Chương trình kết thúc khi ấn phím Esc. Các bạn có thể phát triển thêm các thí dụ loại này (cũng như phát triển thêm các hàm và thủ tục trong unit MOUSE) để thấy rằng chuột là một công cụ giao tiếp hết sức thuận tiện, đặc biệt trong đồ họa.

```
Uses CRT, GRAPH, MOUSE;
Var
  Gd, Gm: integer;
  Ch: char;
  X, Y, U, V, Xold, Yold: integer;
  Sel, Left: boolean;
  P: Pointer;
  Size: word;
Begin
  if not InitMouse then
  begin
    Write('Mouse chua duoc cai dat. An phim Enter de ket thuc ... ');
    Readln; Halt;
  end;            {mouse đã được khởi động}
  DirectVideo := False;
  Gd := 0;
  InitGraph(Gd, Gm, 'C:\TP\BGI');     {thiết lập chế độ Graph}
  if GraphResult <> 0 then
  begin
    Writeln('Duong dan den cac file BGI khong dung!');
    Write('An phim Enter de ket thuc ... ');
    Readln; Halt;
  end;
  SetFillStyle(11, 2);
  Bar(40, 40, 600, 440);      {tạo một nền hình chữ nhật}
  U := 320; V := 220;         {vị trí ban đầu của hình}
  Size := ImageSize(U-6, V-6, U+6, V+6);
  GetMem(P, Size);            {cất nền xung quanh hình và gửi vào P^}
  GetImage(U-5, V-5, U+5, V+5, P^);
  SetFillStyle(1, 15);
  FillEllipse(U, V, 5, 5);    {vẽ hình tròn mầu trắng bán kính 5}
```

```
  ShowMouse;      {hiện mouse}
  RangeMouse(40, 40, 600, 440);   {đặt phạm vi của mouse}
  Ch := #0; Sel := False;     {hình chưa được chọn}
  repeat          {vòng lặp chọn và kéo hình}
    GetMouse(X, Y, Left);
    GotoXY(6, 1); Write('Toa do hinh: ', U:5, V:5);
    GotoXY(52, 1); Write('Toa do chuot: ', X:5, Y:5);
    {kéo hình}
    if Sel and ((X<>Xold) or (Y<>Yold)) then
    begin
      HideMouse;      {trước khi vẽ cần giấu mouse}
      PutImage(Xold-5, Yold-5, P^, NormalPut);   {khôi phục nền ảnh cũ}
      GetImage(X-5, Y-5, X+5, Y+5, P^);        {cất giữ nền ảnh mới}
      SetFillStyle(1, 4);
      SetColor(4);
      FillEllipse(X, Y, 5, 5);     {hình có mầu đỏ khi được kéo}
      Xold := X; Yold := Y;        {giữ lại vị trí hình}
      ShowMouse;      {hiện lại mouse khi vẽ xong}
    end;
    {chọn hình}
    if not Sel and Left then     {nhấn nút trái}
      if (Abs(X-U) <= 5) and (Abs(Y-V) <= 5) then
      begin             {vị trí mouse trong phạm vi hình}
        HideMouse;
        SetFillStyle(1, 4);
        SetColor(4);
        FillEllipse(U, V, 5, 5);   {hình có mầu đỏ khi được chọn}
        Sel := True;
        SetMouse(U, V);      {điều chỉnh vị trí mouse vào đúng tâm hình}
        Xold := U; Yold := V;   {giữ lại vị trí hình}
        ShowMouse;
      end;
    if Sel and not Left then     {nhả nút trái}
    begin          {trở về trạng thái không được chọn}
      HideMouse;
      U := X; V := Y;     {ghi nhận vị trí mới của hình}
      SetFillStyle(1, 15);
      SetColor(15);
      FillEllipse(U, V, 5, 5);   {hình có mầu trắng khi không được chọn}
      Sel := False;
      ShowMouse;
    end;
    if KeyPressed then Ch := ReadKey;
  until Ch = #27;
  CloseGraph;       {trở về chế độ Text}
  HideMouse;
End.
```

Chương trình dưới đây giới thiệu một trò chơi dùng chuột trong chế độ văn bản. Khi chương trình bắt đầu, một con rắn (Snake) được thiết kế chạy xung quanh màn hình cho đến khi bạn click (nhấn rồi nhả) phím trái chuột. Khi đó, con trỏ chuột (một hộp sáng), tượng trưng cho chú chuột Mickey, sẽ xuất hiện trên màn hình và rắn sẽ bắt đầu đuổi bắt. Chương trình được xây dựng sao cho dù Mickey ở đâu, rắn cũng đều xác định được con đường gần nhất để tiếp cận. Nhiệm vụ của bạn là phải di chuyển chuột nhanh để tránh Mickey chạm phải rắn. Trong quá trình đuổi bắt, cứ khoảng 1 giây rắn lại dài thêm và tăng tốc một ít (lúc ấy rắn đổi mầu). Nếu Mickey chạm phải rắn thì xem như nó bị rắn bắt, hộp sáng sẽ biến mất và cuộc chơi được lặp lại với một Mickey khác. Khi rắn đạt được độ dài tối đa thì cuộc chơi kết thúc với những chuyển động ngẫu nhiên của rắn. Âm thanh được tạo ở những lúc cần thiết để tăng tính hấp dẫn của chương trình. Số Mickey bị bắt luôn được thông báo dưới đáy màn hình để tiện theo dõi và người chơi cần đạt được con số này càng nhỏ càng tốt. Chương trình sẽ kết thúc tại bất cứ lúc nào ấn phím Esc.

Chương trình dùng các unit CRT (của Turbo Pascal) và MOUSE (thiết kế ở trên). Phần chính của chương trình dùng 3 thủ tục:

Thủ tục `Init` khởi tạo các giá trị cần thiết cho chương trình. Thủ tục `Start` thực hiện chuyển động của rắn khi chưa có Mickey. Thủ tục `Play` thực hiện quá trình đuổi bắt Mickey của rắn. Các chương trình con khác dùng để hỗ trợ cho các thủ tục này, chẳng hạn thủ tục `DrawSnake` dùng để vẽ rắn, nó được gọi trong thân các vòng lặp tạo chuyển động của rắn, thủ tục `DefPath` xác định đường đi từ rắn đến Mickey, nó được gọi mỗi khi Mickey thay đổi vị trí. Thủ tục `Thank` thiết kế lời cám ơn của chương trình, được gọi khi cuộc chơi kết thúc. Thủ tục `Finish` kết thúc chương trình, được gọi mỗi khi có phím Esc được ấn, ... Về chi tiết, các bạn tìm hiểu thêm qua các lời chú giải để có thể thay đổi hay thêm bớt các lệnh theo ý thích của mình.

```
Program SnakeGame;
Uses CRT, MOUSE;
Const
  c1 = 1; d1 = 1; c2 = 80; d2 = 24;
  Top = d1+1; Bottom = d2-1; Left = c1+1; Right = c2-1;
  Hmid = (Left+Right) div 2; Vmid = (Top+Bottom) div 2;
  Tstart = 60;
  Nstart = 15;
  Duration = 18;
  Nmax = 195;
Type Cell = record
              Col, Row: integer;
            end;
      ArrCell = array[1..200] of Cell;
Var T, N, Nk: integer;
    Snake, Path: ArrCell;
    Head, Tail, Mickey: Cell;
    Status: byte;
```

```
  ColorS: word;
  Xm, Ym, Nm: integer;
  LeftOn: boolean;
  Tcurr: LongInt absolute $0:$46C;
  Tbegin, Dt: LongInt;
  Ch: char;

procedure DrawSnake;
{vẽ Snake từ các tọa độ cho bởi Snake[i], i = 1..N}
var i: integer;
begin
  TextColor(ColorS);
  for i := 1 to N-1 do
  with Snake[i] do
  begin
    GotoXY(Col, Row); Write(#219);
  end;
  with Head do
  begin
    GotoXY(Col, Row); Write('o');
  end;
end;

function Found(V: Cell): boolean;
{trả về True nếu Snake gặp tọa độ V}
var i: integer;
begin
  for i := 1 to N do
  if (V.Col = Snake[i].Col) and (V.Row = Snake[i].Row) then
  begin
    Found := True; Exit;
  end;
  Found := False;
end;

procedure SoundOn;
{phát âm thanh khi Snake gặp Mickey}
var i, h: integer;
begin
  h := 500;
  for i := 1 to 100 do
  begin
    Sound(h); Delay(5); NoSound;
    h := h+20;
  end;
end;
```

```
procedure DefPath;
{xác định đường đi Path[i] từ Snake đến Mickey, i = 1..Nk}
var DistRow, DistCol, SignCol, SignRow, StepRow, StepCol,
    Kmin, Kmax, i, c, d: integer;
begin
  DistCol := Mickey.Col-Head.Col;
  DistRow := Mickey.Row-Head.Row;
  if DistCol > 0 then SignCol := 1
  else if DistCol = 0 then SignCol := 0
       else begin SignCol := -1; DistCol := -DistCol; end;
  if DistRow > 0 then SignRow := 1
  else if DistRow = 0 then SignRow := 0
       else begin SignRow := -1; DistRow := -DistRow; end;
  Kmin := DistCol; Kmax := DistCol;
  if DistCol > DistRow then Kmin := DistRow;
  if DistCol < DistRow then Kmax := DistRow;
  StepCol := 0; StepRow := 0; c := Head.Col; d:= Head.Row;
  for i := 1 to Kmin+Kmin do
  begin
    if StepCol = 0 then
    begin
      StepCol := SignCol; StepRow := 0;
    end else
    begin
      StepCol := 0; StepRow := SignRow;
    end;
    c := c+StepCol; d := d+StepRow;
    Path[i].Col := c; Path[i].Row := d;
  end;
  if Kmin < Kmax then
    if Kmin = DistCol then
      for i := Kmin+Kmin+1 to Kmax+Kmin do
      begin
        d := d+SignRow;
        Path[i].Col := c;
        Path[i].Row := d;
      end
    else
      for i := Kmin+Kmin+1 to Kmax+Kmin do
      begin
        c := c+SignCol;
        Path[i].Row := d;
        Path[i].Col := c;
      end;
  Nk := Kmax+Kmin;
end;
```

```
procedure Message;
{thông báo số lượng Mickey đã bị bắt}
begin
  TextColor(7); GotoXY(c1+1, d2+1);
  Write('Mickey: ', Nm, '   ');
end;

procedure Finish;
{kết thúc chương trình}
var i, j, h: integer;
begin
  TextColor(14);
  GotoXY(36, d2); Write(' Good bye! ');
  TextColor(7);
  h := 100;
  for i := 1 to 200 do
  begin
    Sound(h); Delay(5); NoSound;
    h := h+20;
  end;
  for i := 1 to 200 do
  begin
    Sound(h); Delay(5); NoSound;
    h := h-10;
  end;
  for i := 1 to 50 do
  begin
    Sound(h); Delay(5); NoSound;
  end;
  Halt;
end;

procedure Thank;
{lời cảm ơn khi hết cuộc chơi}
var i, j: integer;
begin
  TextColor(14); GotoXY(30, d1);
  Write(' Thank You very much! ');
  repeat
    Mickey.Col := Left+Random(Right-Left+1);
    Mickey.Row := Top+Random(Bottom-Top+1);
    DefPath;
    if KeyPressed then Ch := ReadKey;
    if Ch = #27 then Finish;        {kết thúc nếu ấn phím Esc}
    for j := 1 to Nk do
    begin
      DrawSnake;
```

```
      if KeyPressed then Ch := ReadKey;
      if Ch = #27 then Finish;      {kết thúc nếu ấn phím Esc}
      Head := Path[j];
      for i := 1 to N-1 do Snake[i] := Snake[i+1];
      Snake[N] := Head;
      Delay(T);
      if not Found(Tail) then
      with Tail do
      begin
        GotoXY(Col, Row); Write(' ');
      end;
      Tail := Snake[1];
    end;    {for j}
  until False;
end;

procedure Init;
{khởi tạo các giá trị ban đầu}
var i: integer;
begin
  ClrScr;
  if not InitMouse then
  begin
    Write('Mouse chua duoc cai dat! An phim Enter de ket thuc ... ');
    Readln; Halt;
  end;
  TextColor(7);
  GotoXY(c1, d1); Write(#201);
  for i := c1+1 to c2-1 do Write(#205);
  Write(#187);
  for i := d1+1 to d2-1 do
  begin
    GotoXY(c1, i); Write(#186); GotoXY(c2, i); Write(#186);
  end;
  GotoXY(c1, d2); Write(#200);
  for i := c1+1 to c2-1 do Write(#205);
  Write(#188);
  GotoXY(63, d2+1); Write('Press Esc to quit');
  RangeMouse((Left-1)*8, (Top-1)*8, (Right-1)*8, (Bottom-1)*8);
  Randomize;
  N := Nstart;
  for i := 1 to N do
  with Snake[i] do
  begin
    Col := Left+i-1; Row := Top;
  end;
  Head := Snake[N]; Tail := Snake[1];
```

```
  Status := 1; T := Tstart; ColorS := 1+random(15);
  Nm := 0; Message;
end;

procedure Start;
{hoạt động của Snake khi không có Mickey}
var i: integer;
begin
  GoToXY(29, d2+1); TextColor(7); Write('Click left button to play');
  DrawSnake;
  repeat            {vòng lặp Snake chạy vòng quanh}
    GetMouse(Xm, Ym, LeftOn);
    if LeftOn then      {Mickey xuất hiện, chuyển sang Play}
    begin
      ShowMouse; Exit;
    end;
    DrawSnake;
    if KeyPressed then Ch := ReadKey;
    if Ch = #27 then Finish;        {kết thúc nếu ấn phím Esc}
    with Head do
      case Status of
        1: begin
             if Col < Right then Inc(Col);
             if Col = Right then
               if Row < Vmid then Status := 2 else Status := 4;
             end;
        2: begin
             if Row < Bottom then Inc(Row);
             if Row = Bottom then
               if Col < Hmid then Status := 1 else Status := 3;
           end;
        3: begin
             if Col > Left then Dec(Col);
             if Col = Left then
               if Row < Vmid then Status := 2 else Status := 4;
           end;
        4: begin
             if Row > Top then Dec(Row);
             if Row = Top then
               if Col < Hmid then Status := 1 else Status := 3;
           end;
      end;
    for i := 1 to N-1 do Snake[i] := Snake[i+1];
    Snake[N] := Head;
    Delay(T);
    if not Found(Tail) then
    with Tail do
```

```
    begin
      GotoXY(Col, Row); Write(' ');
    end;
    Tail := Snake[1];
  until False;
end;

procedure Play;
{Snake đuổi bắt Mickey}
var i, j: integer;
    NewPos: boolean;
    Temp: Cell;
begin
  GotoXY(29, d2+1); Write(' ':25);
  Tbegin := Tcurr;
  Mickey.Col := Xm div 8+1; Mickey.Row := Ym div 8+1;
  DefPath;
  repeat          {vòng lặp Snake đuổi bắt Mickey}
    if KeyPressed then Ch := ReadKey;
    if Ch = #27 then Finish;        {kết thúc nếu ấn phím Esc}
    NewPos := False;
    for j := 1 to Nk do
    begin
      DrawSnake;
      if Found(Mickey) then
      begin           {Mickey chạm phải Snake! chuyển sang Start}
        SoundOn; HideMouse; Inc(Nm); Message; Exit;
      end;
      GetMouse(Xm, Ym, LeftOn);
      Temp.Col := Xm div 8+1; Temp.Row := Ym div 8+1;
      if Abs(Temp.Col-Mickey.Col)+Abs(Temp.Row-Mickey.Row) >= 1 then
      begin           {nếu Mickey di chuyển thì phải tính lại Path}
        Mickey := Temp;
        DefPath;
        NewPos := True;
        break;    {for j}
      end;
      if KeyPressed then Ch := ReadKey;
      if Ch = #27 then Finish;      {kết thúc nếu ấn phím Esc}
      Head := Path[j];
      for i := 1 to N-1 do Snake[i] := Snake[i+1];
      Snake[N] := Head;
      Delay(T);
      if not Found(Tail) then
      with Tail do
      begin
        GotoXY(Col, Row); Write(' ');
```

```
        end;
        Tail := Snake[1];
        Dt := Tcurr-Tbegin;
        if Dt >= Duration then
        begin                          {sau khoảng 1 giây mà chưa gặp Mickey}
          Snake[N+1] := Head;
          Inc(N);                      {Snake dài thêm 1 đơn vị}
          if N = Nmax then
          begin                        {kết thúc chơi khi đạt độ dài tối đa}
            HideMouse; Thank;
          end;
          if T > 1 then Dec(T);        {nhanh hơn}
          ColorS := 1+Random(15);      {đổi mầu}
          Tbegin := Tcurr;
        end;
      end;    {for j}
    until not NewPos;
    {Snake bắt được Mickey}
    SoundOn; HideMouse;
    Inc(Nm); Message;
  end;

Begin    {chương trình chính}
  Init;
  repeat
    Start;
    Play;
  until False;
End.
```

C. Unit GRAPH

Lập trình đồ họa là một hướng đang được triển khai trong nhiều lĩnh vực ứng dụng hiện nay. Mặc dù Turbo Pascal không phải là một cài đặt chuyên về đồ họa, tuy nhiên với unit GRAPH mà nó cung cấp, người dùng có thể khai thác khả năng đồ họa của máy tính ở những mức độ thông dụng.

1. Giới thiệu chung

Màn hình đồ họa có độ phân giải cao hơn nhiều so với màn hình văn bản. Mỗi phần tử của màn hình là một điểm sáng, gọi là pixel, thay vì một ký tự như trong chế độ văn bản. Lưới tọa độ đồ họa được chia thành hàng trăm (thậm chí hàng nghìn) pixel theo các chiều ngang dọc thay cho 80 cột, 25 dòng như lưới tọa độ văn bản. Cũng vì thế, chỉ ở chế độ đồ họa mới có thể mô phỏng những hình ảnh phức tạp bao gồm cả hình vẽ và chữ viết mà ở trong chế độ văn bản không thể thực hiện nổi. Không kể những ứng dụng chuyên về đồ họa, ngay cả với những ứng dụng thông thường, xu hướng dùng đồ họa để giao tiếp ngày càng được phát triển và thay thế dần các phương thức giao tiếp văn bản trước đây, nhất là trong tình huống cấu hình của máy tính ngày càng mạnh.

Khác với chế độ màn hình văn bản (ngầm định), khi sang chế độ màn hình đồ họa, hệ thống phải quản lý một vùng nhớ tương ứng lớn hơn nhiều. Việc tổ chức vùng nhớ này lại phụ thuộc vào từng loại màn hình. Vì thế, Turbo Pascal cung cấp một loạt các file BGI (Borland Graphics Interface) đóng vai trò là các trình điều khiển đồ họa, mỗi file tương ứng với một loại màn hình. Muốn hoạt động trong chế độ đồ họa, cần phải có file BGI thích hợp với màn hình đang dùng. Bảng dưới đây liệt kê các file BGI của Turbo Pascal:

Tên file BGI	Loại màn hình
ATT.BGI	AT&T 6300
CGA.BGI	CGA và MCGA
EGAVGA.BGI	EGA và VGA
HERC.BGI	HERCULES mono
IBM8514.BGI	IBM 8514
PC3270.BGI	3270 PC
VESA16.BGI	VESA 16 colors

Ngoài ra, chế độ đồ họa còn dùng các font chữ vectơ bên cạnh font bitmap của hệ thống. Với các font vectơ, chữ viết đa dạng và mềm mại hơn nhiều so với font hệ thống. Turbo Pascal cung cấp các file CHR, mỗi file là một font vectơ để khi cần thiết có thể dùng chúng trong các chương trình đồ họa. Với phiên bản 7, có 10 file như vậy: TRIP.CHR, LITT.CHR, SANS.CHR, GOTH.CHR (4 font cũ), SCRI.CHR, SIMP.CHR, TSCR.CHR, LCOM.CHR, EURO.CHR, BOLD.CHR (6 font bổ sung).

Các file BGI và CHR được Turbo Pascal ngầm định để trong thư mục con BGI của thư mục cài đặt nó. Khi khởi động đồ họa, cần chỉ rõ đường dẫn đến các file này để hệ thống tìm thấy. Trong các thí dụ minh họa, đường dẫn này được giả thiết là C:\TP\BGI.

Tương ứng với mỗi một trình điều khiển đồ họa, có nhiều mode đồ họa khác nhau, thể hiện qua độ phân giải màn hình và khả năng mầu sắc. Chẳng hạn với trình điều khiển VGA, có 3 mode đồ họa với 16 mầu: Phân giải thấp (VGALo) 320×200, phân giải trung bình (VGAMed) 640×350 và phân giải cao (VGAHi) 640×480.

Các giá trị tọa độ trong chế độ đồ họa được đánh số từ 0, cột được tính từ phải sang trái và dòng được tính từ trên xuống dưới và bao giờ cũng thuộc kiểu nguyên. Chẳng hạn với VGAHi, góc trên trái màn hình có tọa độ 0, 0 và góc dưới phải màn hình có tọa độ 639, 479. Các giá trị mầu cũng được unit GRAPH đặt tên cho dễ nhớ giống như unit CRT đã đặt (xem lại **A. Unit CRT**).

Một chương trình đồ họa bao giờ cũng mở đầu bằng việc khởi động chế độ đồ họa. Các thủ tục, hàm mà unit GRAPH thiết kế (trừ thủ tục khởi động) luôn ngầm định là chỉ hoạt động khi chế độ đồ họa đã được thiết lập. Khi kết thúc, chương trình đồ họa cần phải quay về chế độ ngầm định là màn hình văn bản, vì ở chế độ đồ họa, bộ nhớ bị dùng tốn kém hơn và tốc độ kết xuất ra màn hình cũng chậm hơn nhiều so với chế độ văn bản.

Unit GRAPH là một unit lớn, trong đó chứa khoảng 60 hàm và thủ tục cũng như một số kiểu, hằng. Vì thế để tiếp cận một cách có hiệu quả, bạn cần chú ý một số đặc điểm của GRAPH như sau:

- Các hàm, thủ tục được hình thành theo từng nhóm chức năng: Vẽ điểm, vẽ đường, tô miền, viết chữ, xử lý bitmap, ...
- Trong mỗi nhóm chức năng, hình thành hai khối ngược nhau: Một số có tên mở đầu là SET, dùng để xác lập một số giá trị nào đó, một số có tên mở đầu là GET, lấy lại các giá trị đã đặt.
- Một số kiểu, hằng được xây dựng để phục vụ cho lời gọi các hàm, thủ tục.
- Tên các hàm, thủ tục, kiểu, hằng, ... đều được đặt đủ dài để dễ gợi nhớ theo ngữ nghĩa tiếng Anh.

Để người đọc có thể dễ dàng vận dụng, chúng tôi cố gắng đưa ra những thí dụ minh họa, đặc biệt những thí dụ có tính chất mô hình. Tuy nhiên, bạn cần tham khảo thêm những thí dụ trong các tài liệu trợ giúp, đặc biệt, chương trình BGIDEMO.PAS của Turbo Pascal cung cấp, với những trình bày khá đầy đủ về các khía cạnh của unit GRAPH, sẽ có ích cho bạn trong việc tìm hiểu unit này.

2. Lỗi đồ họa

Trong quá trình thực hiện các thao tác đồ họa, có thể gặp lỗi và khi đó chương trình bị dừng. Để người dùng có thể chủ động trong việc xử lý lỗi, unit GRAPH cung cấp hàm:

```
function GraphResult: integer;
```

trả lại mã lỗi cho lời gọi đồ họa cuối cùng.

Nếu hàm trả lại giá trị 0 thì không có lỗi, các trường hợp có lỗi, hàm trả về một số âm. Các giá trị mã lỗi được đặt tên cho dễ nhớ (đều có phần đầu là gr). Dưới đây là bảng các giá trị trả lại của hàm `GraphResult`:

Mã lỗi	Tên hằng	Nghĩa của lỗi
0	grOK	không có lỗi
-1	grNoInitGraph	đồ họa chưa được cài đặt
-2	grNotDetected	không tìm thấy phần cứng đồ họa
-3	grFileNotFound	không tìm thấy file BGI
-4	grInvalidDriver	file BGI bị hỏng
-5	grNoLoadMem	không đủ bộ nhớ để nạp BGI
-6	grNoScanMem	không đủ bộ nhớ để scan fill
-7	grNoFloodMem	không đủ bộ nhớ để flood fill
-8	grFontNotFound	không tìm thấy font
-9	grNoFontMem	không đủ bộ nhớ để nạp font
-10	grInvalidMode	sai mode đồ họa
-11	grError	lỗi đồ họa
-12	grIOError	lỗi vào/ra đồ họa
-13	grInvalidFont	sai file font
-14	grInvalidFontNum	số font không đúng

Chú ý rằng `GraphResult` được đặt lại bằng 0 sau khi đã được gọi để dùng (giống như `IOResult`), vì thế nếu dùng lại giá trị này, bạn cần gửi nó vào một biến.

Để biết rõ lỗi là gì, bạn dùng hàm:

```
function GraphErrorMsg(ErrorCode: integer): string;
```

trả về xâu ký tự thông báo nghĩa của lỗi có mã lỗi cho bởi `ErrorCode` (bằng tiếng Anh).

Mặc dù mã lỗi có nhiều giá trị nhưng người ta thường quy chúng vào tình huống không tìm thấy file BGI (giá trị -3) vì đây là lỗi thường gặp do đặt đường dẫn sai và có thể điều chỉnh lại mà không cần ngắt chương trình.

3. Khởi động và giải phóng đồ họa

Khởi động đồ họa là bước bắt buộc trước khi dùng các thao tác mà GRAPH cung cấp. Nếu quên chưa khởi động đồ họa mà đã gọi một thủ tục hay hàm của unit GRAPH thì sẽ gặp lỗi *Graphics not installed* (mã lỗi -1) và chương trình kết thúc. Thủ tục để khởi động đồ họa là

```
procedure InitGraph(var driver, mode: integer; path: string);
```

trong đó `driver` là số hiệu của trình điều khiển BGI, `mode` là số hiệu của mode đồ họa khả dĩ của trình điều khiển và `path` là đường dẫn để hệ thống có thể tìm thấy các file BGI. Có hai cách dùng thủ tục khởi động đồ họa:

Cách thứ nhất là *khởi động áp đặt*, nghĩa là qui định cho hệ thống số hiệu driver và số hiệu mode. Với cách này phải đảm bảo số hiệu driver là phù hợp với màn hình hiện có và số hiệu mode phải là một giá trị cho phép của driver, nếu không phù hợp có thể dẫn đến treo máy. Dưới đây là bảng các giá trị số hiệu driver và các số hiệu mode tương ứng:

driver		mode		độ phân giải
số hiệu	tên	số hiệu	tên	(cột×dòng)
1	CGA	0	CGAC0	320×200
		1	CGAC1	320×200
		2	CGAC2	320×200
		3	CGAC3	320×200
		4	CGAHi	320×200
2	MCGA	0	MCGA0	320×200
		1	MCGA1	320×200
		2	MCGA2	320×200
		3	MCGA3	320×200
		4	MCGAMed	640×200
		5	MCGAHi	640×480
3	EGA	0	EGALo	640×200
		1	EGAHi	640×350
4	EGA64	0	EGA64Lo	640×200
		1	EGA64Hi	640×350
5	EGAMono	3	EGAMonoHi	640×350
6	IBM8514	0	IBM8514Lo	640×480
		1	IBM8514Hi	1024×768
7	HercMono	0	HercMonoHi	720×348
8	ATT400	0	ATT400C0	320×200
		1	ATT400C1	320×200
		2	ATT400C2	320×200
		3	ATT400C3	320×200
		4	ATT400Med	640×200
		5	ATT400Hi	640×400
9	VGA	0	VGALo	640×200
		1	VGAMed	640×350
		2	VGAHi	640×480
10	PC3270	0	PC3270Hi	720×350

Cách thứ hai là *khởi động tự động*. Theo cách này, biến `driver` được gán giá trị 0 (mà GRAPH định sẵn một tên là `Detect`) trước khi truyền cho `InitGraph`. Khi đó hệ thống sẽ tự động dò tìm trình điều khiển phù hợp với phần cứng và đồ họa sẽ được khởi động nhờ trình điều khiển này với mode phân giải cao nhất tương ứng. Số hiệu driver sẽ được trả về `driver` và số hiệu mode sẽ được trả về `mode`. Thông thường, khởi động tự động hay được dùng vì đơn giản và không phụ thuộc vào phần cứng.

Thí dụ, màn hình của bạn là VGA và các file BGI đang để ở thư mục C:\TP\BGI. Khi đó các lời gọi

```
driver := 0;
InitGraph(driver, mode, 'C:\TP\BGI');
```

sẽ khởi động đồ họa với chế độ VGAHi. Biến `driver` sẽ nhận giá trị 9 và biến `mode` sẽ nhận giá trị 2.

Sau lời gọi `InitGraph` bạn nên dùng `GraphResult` để kiểm tra việc khởi động đồ họa có thành công hay không. Đa phần lỗi rơi vào tình huống đường dẫn đến các BGI sai, và điều này cho phép người dùng đặt lại giá trị của `path` và khởi động lại. Vòng lặp sau đây sẽ hỏi lại đường dẫn đến các BGI cho đến khi vào đúng:

```
path := ....;   {giá trị ngầm định}
repeat
  driver := 0;
  InitGraph(driver, mode, path);
  err := GraphResult;
  if err <> 0 then
  begin
    Write('Vao lai duong dan den cac file BGI: ');
    Readln(path);
  end;
until err = 0;
```

Nếu bạn thường xuyên dùng cách tự động để khởi động đồ họa thì nên xây dựng hàm `DetectInit` như trình bày dưới đây để dùng lâu dài. Hàm `DetectInit` thực hiện việc khởi động đồ họa theo cách tự động, có tham trị `path` kiểu `string` xác định đường dẫn đến các file BGI và trả về giá trị kiểu `integer` xác định mã lỗi sau khi thực hiện:

```
function DetectInit(path: string): integer;
var drive, mode: integer;
begin
  drive := 0;
  InitGraph(drive, mode, path);
  DetectInit := GraphResult;
end;
```

Nếu bạn chỉ muốn GRAPH dò hộ `driver` để sau đó tự quyết định việc khởi động đồ họa thì dùng thủ tục sau đây trước khi gọi `InitGraph`:

```
procedure DetectGraph(var driver, mode: integer);
```

Thủ tục này sẽ trả về giá trị thích hợp cho `driver` và `mode` giống như `InitGraph` nhưng không khởi động. Điều này có nghĩa bạn có thể tự quyết định giá trị của `driver` và `mode` sao cho thích hợp để áp đặt cho `InitGraph`. Thí dụ dưới đây sẽ nhập trình điều khiển CGA với mode CGAHi thay cho các giá trị mà `DetectGraph` đề nghị, nếu phát hiện những giá trị này thuộc EGA hoặc EGA64:

```
.....
DetectGraph(driver, mode);
if (driver = EGA) or (driver = EGA64) then
begin
  driver := CGA;
  mode := CGAHi;
end;
InitGraph(driver, mode, path);   {áp đặt chế độ CGAHi}
.....
```

Sau khi khởi động đồ họa thành công, bạn có thể lấy hàng loạt các thông tin về chế độ đồ họa hiện hành nhờ các hàm hoặc thủ tục có tên mở đầu là `Get`. Dưới đây là một số các hàm và thủ tục như vậy (các hàm và thủ tục khác sẽ được giới thiệu trong những hoàn cảnh thích hợp):

```
function GetGraphMode: integer;
```

trả về số hiệu của mode đồ họa hiện hành (ý nghĩa của giá trị này tùy thuộc vào trình điều khiển đang hoạt động),

```
function GetMaxMode: integer;
```

trả về giá trị lớn nhất của số hiệu mode của trình điều khiển hiện hành,

```
function GetDriverName: string;
```

trả về tên của trình điều khiển hiện hành,

```
function GetModeName(mode: integer): string;
```

trả về tên của mode đồ họa cho bởi số hiệu `mode` của trình điều khiển hiện hành,

```
function GetMaxX: integer;
```

trả về giá trị lớn nhất của hoành độ X,

```
function GetMaxY: integer;
```

trả về giá trị lớn nhất của tung độ Y,

```
function GetMaxColor: word;
```

trả về giá trị lớn nhất của số hiệu mầu.

Khoảng giá trị mầu tùy thuộc chế độ đồ họa đang được sử dụng. Trong unit GRAPH, các mầu được đánh số từ 0 đến số hiệu mầu lớn nhất và được đặt tên giống như trong unit CRT. Với chế độ VGAHi, số hiệu mầu lớn nhất là 15 (gồm 16 mầu). Việc đặt mầu còn tùy thuộc vào các thao tác: Vẽ điểm, vẽ đường, viết chữ, tô miền, ... Chẳng hạn mầu của điểm được xác định trong thủ tục `PutPixel`, mầu của đường hoặc nét chữ được đặt bởi thủ tục `SetColor`, mầu của miền được tô được đặt bởi thủ tục `SetFillStyle`, ... (các thủ tục này sẽ được giới thiệu trong các mục tương ứng).

Khi sang chế độ đồ họa, toàn màn hình có mầu nền ngầm định là đen. Muốn đặt lại mầu này, bạn dùng thủ tục

```
procedure SetBkColor(color: word);
```

trong đó `color` là số hiệu mầu. Chú ý rằng, mầu của màn hình sẽ được đặt lại ngay sau lời gọi thủ tục này (khác với thủ tục `TextBackground` của unit CRT đặt mầu nền trong chế độ văn bản). Ngược với thủ tục `SetBkColor` là hàm

```
function GetBkColor: word;
```

trả về giá trị mầu nền hiện tại.

Giải phóng đồ họa là thao tác ngược với khởi động đồ họa, nó là câu lệnh kết của các chương trình đồ họa. Thủ tục giải phóng đồ họa là:

```
procedure CloseGraph;
```

Thủ tục này thu hồi tất cả những gì đã cấp phát cho chế độ đồ họa và đặt màn hình về trạng thái văn bản thông thường.

Thí dụ dưới đây trình bày một chương trình, trong đó khởi động đồ họa theo phương thức tự động và hiển thị các thông tin của chế độ đồ họa hiện hành. Cuối chương trình, đồ họa được giải phóng và chế độ làm việc ngầm định là văn bản lại được thiết lập:

```
Uses GRAPH;
Var i: integer;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

Begin
  Writeln('Hien dang o trong che do van ban');
  Write('An phim Enter de vao che do do hoa ... ');
  Readln;
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! An phim Enter de ket thuc...');
    Readln; Halt;
  end;
  Writeln('Hien dang o trong che do do hoa');
  Writeln('Ten trinh dieu khien: ', GetDriverName);
  Writeln('Cac che do tuong ung:');
  for i := 0 to GetMaxMode do
    Writeln('  Mode ', i, ': ', GetModeName(i));
  Writeln('Che do hien hanh: ', GetModeName(GetGraphMode));
  Writeln('Gia tri X cao nhat: ', GetMaxX);
  Writeln('Gia tri Y cao nhat: ', GetMaxY);
  Writeln('So hieu mau cao nhat: ', GetMaxColor);
  Write('An phim Enter de quay ve che do van ban ... ');
  Readln;
```

```
  CloseGraph;
  Writeln('Hien dang o che do van ban');
  Write('An phim Enter de ket thuc...');
  Readln;
End.
```

Trong một số thí dụ minh họa, có dùng một số giá trị hằng cho tọa độ hoặc mầu. Những giá trị này được chọn ngầm định phù hợp với chế độ VGAHi (640×480, 16 mầu), bạn có thể thay đổi lại cho phù hợp với chế độ hiện hành.

4. Chuyển đổi chế độ

Trong quá trình làm việc với đồ họa, có thể chuyển sang mode đồ họa khác (trong phạm vi thích hợp của trình điều khiển) mà không phải khởi động lại đồ họa, bằng cách dùng thủ tục:

```
  procedure SetGraphMode(mode: integer);
```

trong đó `mode` là số hiệu chế độ cần chuyển đến. Để tránh dùng những giá trị không thích hợp của `mode`, bạn nên dùng thủ tục:

```
  procedure GetModeRange(driver: integer; var LoMode, HiMode: integer);
```

trả về số hiệu mode thấp nhất cho `LoMode` và số hiệu mode cao nhất cho `HiMode` của trình điều khiển có số hiệu cho bởi `driver`.

Bạn cũng có thể chuyển tạm thời từ chế độ đồ họa sang chế độ văn bản (không phải đóng đồ họa bằng `CloseGraph`) để rồi quay lại đồ họa (như thế sẽ nhanh hơn vì không phải khởi động lại đồ họa) bằng thủ tục:

```
  procedure RestoreCrtMode;
```

Chú ý cần giữ lại mode đồ họa bằng `GetGraphMode` trước khi gọi `RestoreCrtMode` để có thể khôi phục lại mode này bằng `SetGraphMode`. Thí dụ dưới đây minh họa việc chuyển đổi giữa các chế độ đồ họa và văn bản:

```
Uses CRT, GRAPH;
Var driver, mode, LoMode, HiMode, CurrMode, i: integer;
    path: string;
    ch: char;
    stop: boolean;
Begin
  DirectVideo := false;   {tắt chế độ trực tiếp ghi lên Video Ram}
  driver := 0;
  path := 'C:\TP\BGI';
  InitGraph(driver, mode, path);
  GetModeRange(driver, LoMode, HiMode);
  repeat
    Writeln('Ten trinh dieu khien: ', GetDriverName);
    Writeln('Che do hien hanh: ', GetModeName(GetGraphMode));
```

```
    Writeln('Chon phim de chuyen che do: ');
    for i := LoMode to HiMode do
      Writeln('  ', i, ': ', GetModeName(i));
    Writeln('  ', i+1, ': Quay ve van ban');
    Writeln('Cac phim khac: Ket thuc');
    ch := ReadKey;
    i := Ord(ch)-48;
    stop := (i < LoMode) or (i > HiMode+1);
    if not stop then
      if i <= HiMode then SetGraphMode(i)
      else begin
             CurrMode := GetGraphMode;
             RestoreCrtMode;
             Writeln('Hien dang o che do van ban');
             Write('An phim Enter de tro lai che do do hoa ... ');
             Readln;
             SetGraphMode(CurrMode);
           end;
  until stop;
  CloseGraph;
End.
```

Chương trình có sử dụng unit CRT để nhận diện phím được ấn. Câu lệnh đặt lại giá trị `false` cho biến `DirectVideo` của unit CRT là cần thiết để có thể dùng được các thủ tục `Write` trong chế độ đồ họa.

5. Con trỏ đồ họa

Cũng như trong chế độ văn bản, có một cơ chế lôgic trong chế độ đồ họa quản lý vị trí bắt đầu của một số thao tác vẽ, gọi là con trỏ đồ họa (*current pointer*, viết tắt là CP). Chỉ có điều vị trí này không được nhìn thấy trên màn hình. Khi chế độ đồ họa mới được thiết lập, CP nằm ở tọa độ 0, 0 (góc trên trái màn hình). Một số thao tác vẽ được ngầm định bắt đầu từ vị trí CP và sau khi vẽ xong sẽ đẩy CP đến vị trí mới. Để định vị CP, GRAPH cung cấp thủ tục:

```
procedure MoveTo(x, y: integer);
```

chuyển CP đến tọa độ `x`, `y`. Ngược lại, để lấy lại các giá trị tọa độ của CP, GRAPH cung cấp hai hàm:

```
function GetX: integer;
function GetY: integer;
```

trả về các tọa độ hiện thời của CP. Các thủ tục `MoveTo` và các hàm `GetX`, `GetY` cũng giống như thủ tục `GotoXY` và các hàm `WhereX`, `WhereY` của unit CRT trong chế độ văn bản. Thí dụ:

```
Writeln('Vi tri hien thoi cua CP: x = ', GetX, ', y = ', GetY);
MoveTo(Random(GetMaxX+1), Random(GetMaxY+1));
Writeln('Vi tri hien thoi cua CP: x = ', GetX, ', y = ', GetY);
```

Một thủ tục khác cũng có tác dụng dịch chuyển CP, nhưng theo nghĩa tương đối so với vị trí cũ, là thủ tục:

```
procedure MoveRel(dx, dy: integer);
```

Thủ tục này di chuyển CP đến vị trí mới có độ lệch `dx`, `dy` so với vị trí cũ. Như vậy lời gọi `MoveRel(dx, dy)` là tương đương với lời gọi `MoveTo(GetX+dx, GetY+dy)`. Các lệnh vẽ tương đối và di chuyển tương đối là có ích khi phải vẽ kế tiếp các đối tượng mà độ lệch giữa chúng được xác định đơn giản.

6. Vẽ điểm

Thủ tục

```
procedure PutPixel(x, y: integer; color: word);
```

vẽ một điểm trên màn hình tại tọa độ `x`, `y` với mầu cho bởi `color`. Nếu giá trị `color` là mầu nền thì thủ tục này có tác dụng xóa điểm hiện có.

Ngược với thủ tục trên là hàm

```
function GetPixel(x, y: integer): word;
```

trả về số hiệu mầu của điểm tại vị trí `x`, `y`. Nếu mầu này trùng với mầu nền (nghĩa là không thấy điểm) thì hàm trả về `0`.

Chẳng hạn, lệnh

```
if GetPixel(x, y) = 0 then PutPixel(x, y, color)
else PutPixel(x, y, 0);
```

sẽ vẽ một điểm có mầu `color` tại vị trí `x`, `y` nếu tại đây không có điểm, trái lại sẽ xóa điểm hiện có.

Chương trình dưới đây thực hiện lặp lại việc vẽ ngẫu nhiên các điểm trên toàn màn hình đồ họa. Các điểm được vẽ theo quy tắc: nếu tại vị trí định vẽ đã có điểm thì chương trình lại xóa điểm đó đi. Chương trình kết thúc khi ấn phím Esc.

```
Program DemoPixel;
Uses CRT, GRAPH;
Var
  MaxX, MaxY: integer;
  MaxColor: word;
  ch: char;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure RanDomPixel;
```

```
  var x, y: integer;
      color: word;
  begin
    x := Random(MaxX+1);   {từ 0 đến MaxX}
    y := Random(MaxY+1);   {từ 0 đến MaxY}
    color := 1+Random(MaxColor);   {từ 1 đến MaxColor}
    if GetPixel(x, y) = 0 then PutPixel(x, y, color)
    else PutPixel(x, y, 0);   {nếu chưa có thì vẽ, nếu đã có thì xóa}
  end;

Begin
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! An phim Enter de ket thuc...');
    Readln; Halt;
  end;
  MaxX := GetMaxX; MaxY := GetMaxY;   {lấy độ phân giải}
  MaxColor := GetMaxColor;   {lấy giá trị mầu}
  ch := #0;
  Randommize;   {sinh dãy ngẫu nhiên}
  repeat
    RandomPixel;
    if KeyPressed then ch := ReadKey;
  until ch = #27;
  CloseGraph;
End.
```

7. Vẽ đường

Unit GRAPH xây dựng các thao tác cho phép vẽ các đường thẳng, đường cong, chu tuyến. Trước khi vẽ, có thể đặt mầu cũng như đặt thuộc tính cho nét vẽ.

7.1. Đặt mầu vẽ

Để đặt mầu cho các nét vẽ đường, bạn dùng thủ tục

```
procedure SetColor(color: word);
```

trong đó `color` là số hiệu mầu. Giá trị mầu được đặt sẽ có hiệu lực với các lệnh vẽ đường sau đó cho đến khi đặt một giá trị mầu mới.

Để lấy lại giá trị mầu đã thiết lập cho các nét vẽ đường, bạn dùng hàm

```
function GetColor: word;
```

trả về giá trị mầu đã được đặt bởi thủ tục `SetColor` trước đấy.

Giá trị mầu ngầm định của các nét vẽ đường là mầu trắng.

7.2. Đặt thuộc tính nét vẽ

Một số thuộc tính: dày hay mỏng, liền nét hay đứt đoạn, ... được đặt cho các nét vẽ đường để các thủ tục vẽ sau đó sử dụng.

Thủ tục

```
procedure SetLineStyle(style, pattern, width: word);
```

đặt nét vẽ có thuộc tính xác định bởi các tham trị với những ý nghĩa như sau:

`style`: Xác định kiểu của nét vẽ, lấy từ các giá trị (với tên kèm theo)

0	SolidLn	liền nét (ngầm định)
1	DotteLn	chấm chấm
2	CenterLn	chấm gạch
3	DashedLn	gạch gạch
4	UserBitLn	tự quy định

`pattern`: Xác định mẫu bit của nét vẽ trong trường hợp `style` tự quy định. Mỗi mẫu bit gồm 16 bit (2 byte), trong đó bit 1 tương ứng với điểm sáng. Thí dụ nếu `pattern` có giá trị $AAAA thì mẫu bit của nét vẽ là 1010101010101010 (chấm chấm mau), còn nếu `pattern` có giá trị $8888 thì mẫu bit của nét vẽ là 1000100010001000 (chấm chấm thưa).

`width`: Quy định độ dày mỏng của nét vẽ, có 2 giá trị (với tên kèm theo)

1	NormWidth	mỏng (ngầm định)
3	ThickWidth	dày gấp 3 lần NormWidth

Nếu không dùng thủ tục `SetLineStyle` trước khi vẽ thì nét vẽ được ngầm định là mỏng và liền nét.

Để lưu trữ thông tin nét vẽ hiện thời, GRAPH xây dựng kiểu

```
Type
  LineSettingsType = record
                       LineStyle: word;
                       Pattern: word;
                       Thickness: word;
                     end;
```

trong đó `LineStyle` là kiểu, `Pattern` là mẫu bit và `Thickness` là độ dày của nét vẽ.

Thủ tục

```
procedure GetLineSettings(var LineInfo: LineSettingsType);
```

lấy thông tin của nét vẽ hiện tại và gửi vào biến `LineInfo`.

7.3. Vẽ đoạn thẳng

Thao tác cơ bản để vẽ các loại đường là thủ tục vẽ đoạn thẳng. GRAPH cung cấp các thủ tục sau:

```
procedure Line(x1, y1, x2, y2: integer);
```

vẽ đoạn thẳng nối điểm có tọa độ `x1`, `y1` với điểm có tọa độ `x2`, `y2`. Thí dụ lời gọi

```
Line(0, 0, GetMaxX, GetMaxY);
```

vẽ một đường chéo trên màn hình, nối góc trên trái và góc dưới phải.

Thủ tục

```
procedure LineTo(x, y: integer);
```

vẽ đoạn thẳng nối điểm CP đang định vị với điểm có tọa độ x, y. Sau khi vẽ xong, CP được đẩy tới vị trí mới. Như thế lời gọi

```
Line(x1, y1, x2, y2);
```

sẽ vẽ một đoạn thẳng giống như lời gọi

```
MoveTo(x1, y1); LineTo(x2, y2);
```

ngoại trừ hiệu ứng đối với CP: lời gọi thứ nhất không ảnh hưởng đến CP, còn lời gọi thứ hai, CP được định vị đến `x2, y2`.

Thủ tục `LineTo` được dùng rất thuận tiện để vẽ các đoạn thẳng kế tiếp nhau tạo thành một đường gấp khúc. Giả sử tọa độ các đỉnh của đường này được cất trong các mảng nguyên `x[i], y[i], i = 0, ..., n`. Khi đó lệnh vẽ đường sẽ là:

```
MoveTo(x[0], y[0]);
for i := 1 to n do LineTo(x[i], y[i]);
```

Một thủ tục nữa được dùng để vẽ đoạn thẳng là

```
procedure  LineRel(dx, dy: integer);
```

Thủ tục này vẽ một đoạn thẳng kể từ vị trí CP đến điểm cách đó một khoảng tương đối là `dx, dy`. Sau khi vẽ xong, CP được đẩy tới vị trí mới. Như thế, lời gọi

```
LineTo(GetX+dx, GetY+dy);
```

là hoàn toàn tương đương với lời gọi

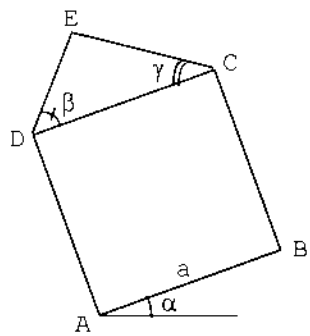
```
LineRel(dx, dy);
```

Để minh họa các thủ tục vẽ đoạn thẳng, bạn thử giải bài toán dựng hình dưới đây:

- Đầu tiên dựng hình vuông ABCD có đáy là AB với độ dài a cho trước.
- Sau đó trên cạnh CD (song song với đáy) dựng tam giác vuông CDE, có cạnh huyền là CD và các cạnh tuân theo tỷ lệ Pythagor DE:EC:CD = 3:4:5.
- Lấy các cạnh DE và cạnh EC làm các đáy và dựng tiếp các hình vuông theo quy tắc đã nêu.

Quá trình dựng các hình vuông được tiếp tục cho đến khi các cạnh đủ nhỏ. Bạn sẽ thu được một hình ảnh, trông như một cái cây. Người ta gọi nó là cây Pythagor.

Thao tác cơ bản để vẽ cây Pythagor là dựng một hình vuông như đã nêu, trong đó có thể thay đổi điểm đặt, kích thước cạnh và độ nghiêng. Vì thế, một giải pháp thích hợp là xây dựng một thủ tục đệ quy vẽ hình vuông với các tham số: x, y - tọa độ của điểm đặt (đầu mút A của đáy AB), a - kích thước của cạnh và α - góc nghiêng của đáy so với phương nằm ngang. Trong thủ tục này lại gọi chính nó hai lần với các giá trị tham số thích hợp để dựng hai hình vuông kế tiếp có đáy đặt tại DE và EC:



Các điểm B, C, D được suy từ A nhờ các đại lượng xác định độ lệch:

$dx = a.\cos\alpha$

$dy = a.\sin\alpha$

Hai hình vuông kế tiếp được đặt tại D và tại E với các đáy DE và EC. Từ tỷ lệ Pythagor 3:4:5, suy ra độ dài các cạnh của hai hình vuông này là:

$DE = 3a/5$

$ED = 4a/5$

Góc nghiêng của đáy DE so với phương nằm ngang là $\alpha+\beta$ và góc nghiêng của đáy EC so với phương nằm ngang là $\alpha-\gamma$, trong đó β và γ được tính trước như những hằng số:

$\beta = \mathrm{arctg}(4/3) = 0.927295$

$\gamma = \mathrm{arctg}(3/4) = 0.643501$

Tọa độ của điểm D (điểm đặt cho hình vuông thứ nhất) và tọa độ của điểm E (điểm đặt cho hình vuông thứ hai) được xác định từ tọa độ (x, y) của điểm A như sau:

$D = (x-dy, y-dx)$

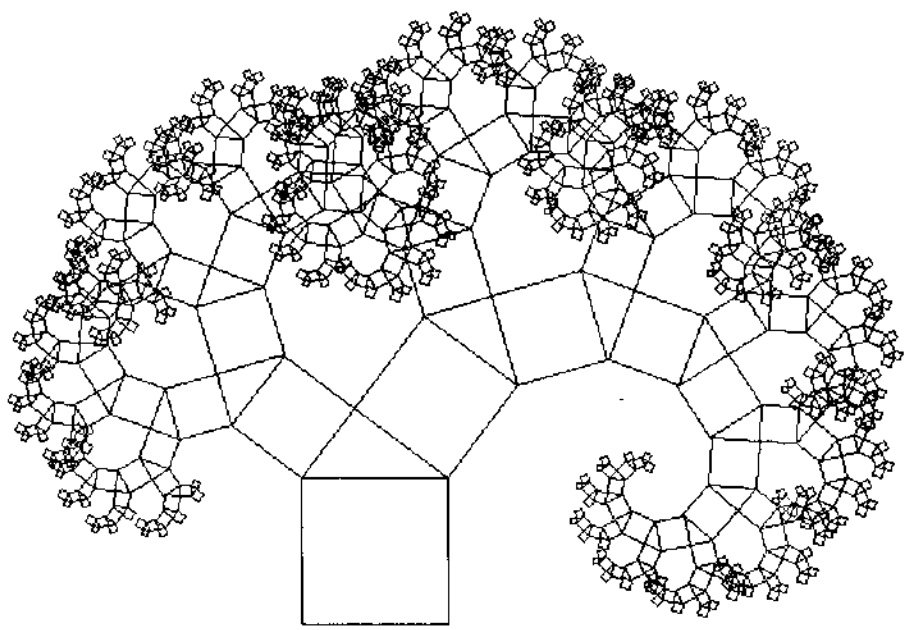
$E = (x-dy+3a/5.\cos(\alpha+\beta), y-dx-3a/5.\sin(\alpha+\beta))$

Dưới đây là toàn văn chương trình vẽ cây Pythagor theo nội dung đã thiết kế:

```
Program Pythagor;
Uses GRAPH;
Const
  beta = 0.927295;        {ArcTan(4/3)}
  gama = 0.643501;        {ArcTan(3/4)}
Var err: integer;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure Square(x, y: integer; a, anfa: real);
  {vẽ hình vuông tại x, y, cạnh a, góc nghiêng anfa}
  var dx, dy: integer;
  begin
    dx := Round(a*Cos(anfa));
    dy := Round(a*Sin(anfa));
    MoveTo(x, y);          {vẽ hình vuông}
    LineRel(dx, -dy);
    LineRel(-dy, -dx);
```

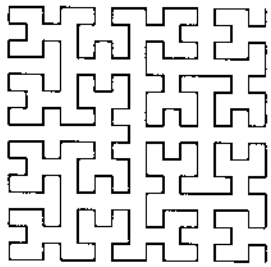
```
    LineRel(-dx, dy);
    LineRel(dy, dx);
    if a > 5 then          {dựng hai hình vuông kế tiếp}
    begin
      Square(x-dy, y-dx, 3*a/5, anfa+beta);
      Square(x-dy+Round(3*a/5*Cos(anfa+beta)),
             y-dx-Round(3*a/5*Sin(anfa+beta)),
             4*a/5, anfa-gama);
    end;
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err = 0 then
    Square(200, 450, 80, 0)
  else
    Write(GraphErrorMsg(err),'! Press Enter to quit...');
  Readln;
  CloseGraph;
End.
```

Thủ tục đệ quy `Square` kết thúc khi độ dài của cạnh hình vuông nhỏ hơn hay bằng 5 và nó được khởi động trong chương trình chính với tọa độ điểm đặt là 200, 450, kích thước cạnh là 80 và có góc nghiêng 0. Bạn có thể thay đổi lại các giá trị này theo ý thích. Dưới đây là hình ảnh cây Pythagor thu được khi chạy chương trình:

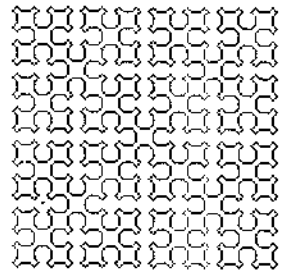


Bằng đệ quy, bạn có thể tạo nhiều hình vẽ thú vị từ những thao tác vẽ đoạn thẳng. Chẳng hạn, bạn có thể viết các chương trình dựng các đường Hilbert hoặc các đường Sierpinski như những hình vẽ dưới đây để làm thí dụ luyện tập:

Đường Hilbert cấp 4



Đường Sierpinski cấp 4



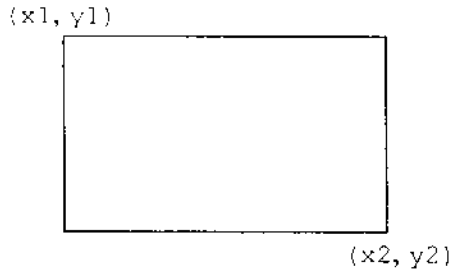
Mặc dù mọi đường đều có thể được vẽ từ các đoạn thẳng nhưng GRAPH vẫn cung cấp một số thủ tục vẽ một số đường đặc biệt nhằm nhanh chóng tạo các công cụ thuận tiện cho người sử dụng. Các mục tiếp theo từ **7.4** đến **7.6** dưới đây giới thiệu những thủ tục vẽ khung chữ nhật, đường tròn, đường ellip và đường gấp khúc.

7.4. Vẽ khung chữ nhật

Thủ tục

```
procedure Rectangle(x1, y1, x2, y2: integer);
```

vẽ một khung chữ nhật có các cạnh song song với cạnh màn hình, góc trên trái có tọa độ `x1`, `y1`, góc dưới phải có tọa độ `x2`, `y2`.

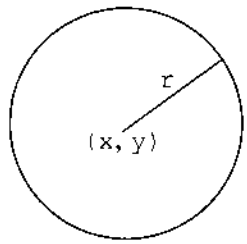


7.5. Vẽ đường tròn & ellip

Thủ tục

```
procedure Circle(x, y: integer; r: word);
```

vẽ một đường tròn tâm `x`, `y`, bán kính `r`.



Thủ tục

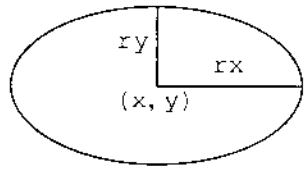
```
procedure Arc(x, y: integer; a1, a2, r: word);
```

vẽ một cung tròn tâm `x`, `y`, bán kính `r`, góc đầu `a1`, góc cuối `a2` (tính theo đơn vị độ).

Thủ tục

```
procedure Ellipse(x, y: integer; a1, a2, rx, ry: word);
```

vẽ một cung ellip tâm `x`, `y`, bán kính ngang `rx`, bán kính dọc `ry`, góc đầu `a1`, góc cuối `a2` (tính theo đơn vị độ). Toàn bộ ellip sẽ được vẽ với lời gọi `Ellipse(x, y, 0, 360, rx, ry);`



Thông tin về lần gọi cuối tới `Arc` hoặc `Ellipse` được lấy từ thủ tục:

```
procedure GetArcCoords(var ArcCoords: ArcCoordsType);
```

trong đó `ArcCoordsType` là kiểu dữ liệu được định nghĩa:

```
Type
ArcCoordsType = record
                  x, y,                  {tọa độ tâm}
                  xStart, yStart,        {tọa độ bắt đầu}
                  xEnd, yEnd: integer;   {tọa độ kết thúc}
                end;
```

7.6. Vẽ đường gấp khúc

Thủ tục

```
procedure DrawPoly(NumPoints: word; var PolyPoints);
```

vẽ một đường gấp khúc, trong đó `PolyPoints` là một tham biến không định kiểu lưu trữ tọa độ các đỉnh theo thứ tự được nối còn `NumPoints` là tham trị xác định số đỉnh của đường. Mỗi đỉnh được lưu trữ trong 4 byte: hai byte đầu (`integer`) lưu trữ tọa độ x, hai byte sau (`integer`) lưu trữ tọa độ y, vì thế kích thước của `PolyPoints` cần phải lớn hơn hay bằng 4 lần giá trị `NumPoints`. Thông thường `PolyPoints` được khai báo như một mảng gồm các phần tử có kiểu `PointType` được GRAPH định nghĩa như sau:

```
Type
PointType = record
              x, y: integer;
            end;
```

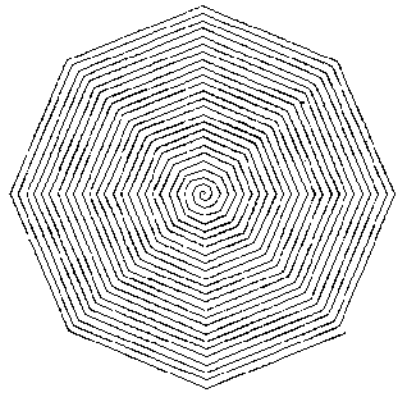
Nếu đường gấp khúc có quá nhiều đỉnh thì thủ tục `DrawPoly` có thể mắc lỗi do không đủ bộ nhớ (hàm `GraphResult` trả về mã lỗi -6).

Chương trình dưới đây dùng `DrawPoly` để vẽ một đường gấp khúc dạng xoắn ốc. Các đỉnh của đường này được lấy trên đường cong có phương trình trong tọa độ cực r = φ,

trong đó r là bán kính cực và φ là góc cực. Số đỉnh được xây dựng là 200, đỉnh xuất phát được đặt tại giữa màn hình:

```
Program Spiral;
Uses GRAPH;
Const
  NumPoints = 200;
Var
  x0, y0, err: integer;
  n: word;
  PolyPoints: array[1..NumPoints] of PointType;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure DefCoords;
  var i: word;
      t, dt: real;
  begin
    dt := Pi/4;
    t := 0;
    for i := 1 to NumPoints do
    begin
      with PolyPoints[i] do
      begin
        x := x0+Round(t*Cos(t));
        y := y0-Round(t*Sin(t));
      end;
      t := t+dt;
    end;
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  x0 := GetMaxX div 2; y0 := GetMaxY div 2; {lấy vị trí giữa màn hình}
  DefCoords;     {xác định tọa độ các đỉnh}
  DrawPoly(NumPoints, PolyPoints);     {vẽ đường gấp khúc}
```

```
  Readln;
  CloseGraph;
End.
```



Đường gấp khúc xoắn ốc được vẽ bởi chương trình

Nếu định vẽ một đa giác (đường gấp khúc khép kín) bằng `DrawPoly` thì phải thêm một tọa độ trùng với tọa độ của đỉnh đầu tiên. Thủ tục dưới đây được xây dựng nhờ `DrawPoly` nhằm vẽ một đa giác đều `n` đỉnh, có tâm tại điểm `x0`, `y0` và có bán kính `r`:

```
procedrue Polygon(x0, y0: integer; r, n: word);
var PolyPoints: array[1..100] of PointType;
    i: integer;
    t, dt: real;
begin
  dt := 2*Pi/n;
  t := 0;
  for i := 1 to n do                    {xác định tọa độ n đỉnh}
  begin
    with PolyPoints[i] do
    begin
      x := x0+Round(r*Cos(t));
      y := y0+Round(r*Sin(t));
    end;
    t := t+dt;
  end;
  PolyPoints[n+1] := PolyPoints[1];  {thêm đỉnh trùng với đỉnh đầu}
  DrawPoly(n+1, PolyPoints);         {vẽ đa giác}
end;
```

7.7. Vẽ đường cong bất kỳ

Về lý thuyết, mọi đường cong đều có thể mô tả xấp xỉ bằng một đường gấp khúc, vì thế chúng đều được thực hiện nhờ thao tác cơ bản là vẽ đoạn thẳng. Phương pháp chung để vẽ một đường cong là:

- Chia đường cong bởi những điểm chia và xác định tọa độ các điểm này. Số điểm chia càng nhiều thì mô tả càng chính xác.
- Vẽ các đoạn thẳng nối lần lượt các điểm chia theo đúng thứ tự của chúng.

Tọa độ các điểm chia được tính từ phương trình của đường cong. Tuy nhiên, do các lệnh vẽ chỉ xử lý các giá trị tọa độ nguyên và trong phạm vi màn hình nên cần phải biến đổi các tọa độ điểm chia (gọi là tọa độ thực) thành các tọa độ tương ứng trên màn hình (gọi là tọa độ màn hình) trước khi truyền chúng cho các lệnh vẽ để hình ảnh của đường cong trên màn hình có thể cảm nhận được.

Thông thường phương trình đường cong có 3 dạng: dạng hiện $y = F(x)$, dạng tham số $x = F_x(t)$, $y = F_y(t)$ và dạng tọa độ cực $r = F(\varphi)$. Ta lần lượt xét từng trường hợp.

a. Vẽ đường cong y = F(x)

Giả thiết F(x) là một hàm xác định và liên tục trên đoạn [a, b]. Cần biểu diễn đồ thị của nó trong đoạn đang xét trên một vùng nào đó của màn hình.

Chia đều đoạn a, b bởi n+1 điểm:

$$x_i = a+i.h \quad (i = 0, 1, ..., n)$$

trong đó h = (b-a)/n, từ đó tính được tung độ các điểm chia thuộc đường cong:

$$y_i = F(x_i) \quad (i = 0, 1, ..., n)$$

Bao quanh đoạn đường cong cần vẽ một miền chữ nhật có tọa độ góc dưới trái là x_{min}, y_{min} và tọa độ góc trên phải là x_{max}, y_{max} (tọa độ thực). Để miền này vừa đủ chứa tất cả các điểm chia của đường cong, ta chọn $x_{min} = a$, $x_{max} = b$, $y_{min} = \min \{y_i\}$, $y_{max} = \max \{y_i\}$. Giả thiết miền cần biểu diễn đường cong trên màn hình là hình chữ nhật có góc trên trái c_{min}, d_{min} và góc dưới phải c_{max}, d_{max} (tọa độ màn hình). Ánh xạ từ miền tọa độ thực (x_{min}, y_{min}, x_{max}, y_{max}) sang miền tọa độ màn hình (c_{min}, d_{min}, c_{max}, d_{max}) bảo toàn tính biến thiên của đường cong là một ánh xạ tuyến tính. Gọi x, y là tọa độ thực và c, d là tọa độ màn hình tương ứng, ta dễ dàng tính được mối liên hệ dưới đây:

$$c = p.(x-x_{min}) + c_{min}$$

$$d = q.(y-y_{min}) + d_{max}$$

trong đó $p = (c_{max}-c_{min})/(x_{max}-x_{min})$ và $q = (d_{min}-d_{max})/(y_{max}-y_{min})$. Chú ý rằng chiều biến thiên của y và d là ngược nhau nên hệ số $q < 0$. Công thức này dùng để chuyển tọa độ thực sang tọa độ màn hình trước khi chúng được truyền cho các thủ tục vẽ.

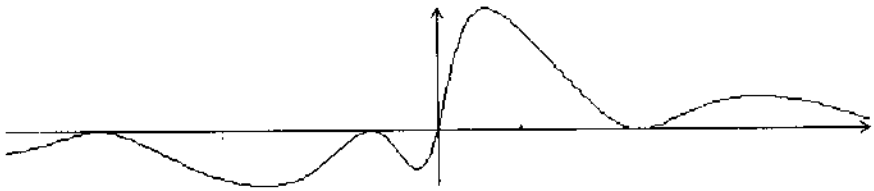
Chương trình dưới đây thực hiện việc vẽ đồ thị hàm số y = F(x) trên một đoạn cho trước. Hàm `F` được khai báo lên đầu để nhấn mạnh việc xử lý không phụ thuộc vào hàm và có thể thay thế bằng bất kỳ hàm nào (miễn là tính được). Các giá trị cho trước được khai báo dưới dạng hằng để bạn dễ dàng thay đổi theo ý thích. Trong chương trình có dùng các thủ tục `DefCoords` để xác định tọa độ các điểm chia và các hệ số tỷ lệ, hàm `DetectInit` để khởi động đồ họa theo phương thức tự động và thủ tục `Draw` để vẽ đồ thị của hàm trên đoạn đang xét.

Chương trình vẽ đồ thị hàm $y = x.\ln(2+\sin(x))/(x^2+1)$ trên đoạn [-10, 10]. Số đoạn chia là 200. Khung nhìn được đặt ở góc trên trái màn hình với chiều rộng 500 và chiều cao

100. Bạn có thể thử lại chương trình này với những hàm khác nhau và với những thông số khác nhau.

```
Uses GRAPH;

  Function F(x: real): real;
  begin
    F := x*ln(2+sin(x))/(sqr(x)+1);
  end;

Const
  n = 200;   {số đoạn chia}
  cmin = 1; dmin = 1; cmax = 500; dmax = 100;   {khung nhìn}
  xmin = -10.0; xmax = 10.0;   {đoạn được vẽ}

Var
  ymin, ymax, h: real;
  y: array[0..n] of real;
  p, q: real;
  err: integer;

  procedure DefCoords;
  var i: integer;
      xi, yi: real;
  begin
    h := (xmax-xmin)/n;
    ymin := 1e37; ymax := -ymin;
    for i := 0 to n do
    begin
      xi := xmin+i*h;
      yi := F(xi);
      if ymin > yi then ymin := yi;
      if ymax < yi then ymax := yi;
      y[i] := yi;
    end;
    p := (cmax-cmin)/(xmax-xmin);
    q := (dmin-dmax)/(ymax-ymin);
  end;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;
```

```
  procedure Draw;
  var i, c, d: integer;
      xi: real;
  begin
    c := cmin;          {xác định điểm đầu tiên}
    d := Round(q*(y[0]-ymin)+dmax);
    MoveTo(c, d);   {bắt đầu vẽ}
    for i := 1 to n do
    begin
      xi := xmin+i*h;
      c := Round(p*(xi-xmin)+cmin);
      d := Round(q*(y[i]-ymin)+dmax);
      LineTo(c, d);
    end;                {kết thúc vẽ đường cong}
    d := Round(q*(-ymin)+dmax);              {xác định vị trí trục hoành}
    if (d >= dmin) and (d <= dmax) then      {vẽ trục hoành}
    begin
      Line(cmin, d, cmax, d);
      Line(cmax, d, cmax-5, d-3);
      Line(cmax, d, cmax-5, d+3);
    end;
    c := Round(p*(-xmin)+cmin);              {xác định vị trí trục tung}
    if (c >= cmin) and (c <= cmax) then      {vẽ trục tung}
    begin
      Line(c, dmin, c, dmax);
      Line(c, dmin, c-3, dmin+5);
      Line(c, dmin, c+3, dmin+5);
    end;
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  DefCoords;
  Draw;
  Readln;
  CloseGraph;
End.
```



Đồ thị hàm số được vẽ bởi chương trình

b. Vẽ đường cong $x = F_x(t)$, $y = F_y(t)$

Giả sử đường cong cần vẽ được cho dưới dạng tham số:

$x = F_x(t)$

$y = F_y(t)$

trong đó t là tham số biến thiên từ t_{min} đến t_{max}. Chọn các điểm chia trên đường cong có các tọa độ xác định theo công thức:

$x_i = F_x(t_i)$

$y_i = F_y(t_i)$

trong đó $t_i = t_{min}+i.h$, $h = (t_{max}-t_{min})/n$, $i = 0, 1, ..., n$. Các bước tiếp theo giống như đã trình bày trong phần trên.

Chương trình dưới đây vẽ một đường xoắn dạng lò so có phương trình cho theo tham số:

$x = t+\cos(3t)$

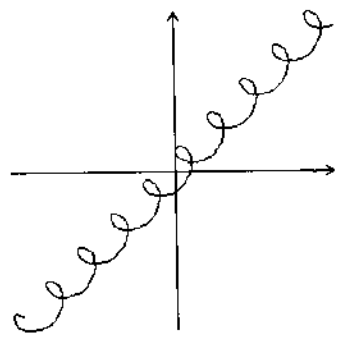
$y = t+\sin(3t)$

với t biến thiên trong khoảng từ -10 đến 10. Cấu trúc chương trình giống như chương trình trước.

```
Uses GRAPH;

  Function Fx(t: real): real;
  begin
    Fx := t+Cos(3*t);
  end;

  Function Fy(t: real): real;
  begin
    Fy := t+Sin(3*t);
  end;

Const
  n = 200;
  cmin = 1; dmin = 1; cmax = 200; dmax = 200;
  tmin = -10.0; tmax = 10.0;

Var
  xmin, xmax, ymin, ymax: real;
  x, y: array[0..n] of real;
```

```
p, q: real;
err: integer;

procedure DefCoords;
var i: integer;
    ti, xi, yi, h: real;
begin
  h := (tmax-tmin)/n;
  ymin := 1e37; ymax := -ymin;
  xmin := ymin; xmax := ymax;
  for i := 0 to n do
  begin
    ti := tmin+i*h;
    xi := Fx(ti);
    yi := Fy(ti);
    if xmin > xi then xmin := xi;
    if xmax < xi then xmax := xi;
    if ymin > yi then ymin := yi;
    if ymax < yi then ymax := yi;
    x[i] := xi;
    y[i] := yi;
  end;
  p := (cmax-cmin)/(xmax-xmin);
  q := (dmin-dmax)/(ymax-ymin);
end;

function DetectInit(path: string): integer;
var drive, mode: integer;
begin
  drive := 0;
  InitGraph(drive, mode, path);
  DetectInit := GraphResult;
end;

procedure Draw;
var i, c, d: integer;
    xi: real;
begin
  c := Round(p*(x[0]-xmin)+cmin);
  d := Round(q*(y[0]-ymin)+dmax);
  MoveTo(c, d);
  for i := 1 to n do
  begin
    c := Round(p*(x[i]-xmin)+cmin);
    d := Round(q*(y[i]-ymin)+dmax);
    LineTo(c, d);
  end;
```

```
    d := Round(q*(-ymin)+dmax);
    if (d >= dmin) and (d <= dmax) then
    begin
      Line(cmin, d, cmax, d);
      Line(cmax, d, cmax-5, d-3);
      Line(cmax, d, cmax-5, d+3);
    end;
    c := Round(p*(-xmin)+cmin);
    if (c >= cmin) and (c <= cmax) then
    begin
      Line(c, dmin, c, dmax);
      Line(c, dmin, c-3, dmin+5);
      Line(c, dmin, c+3, dmin+5);
    end;
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  DefCoords;
  Draw;
  Readln;
  CloseGraph;
End.
```



Đường xoắn lò so được vẽ bởi chương trình

Nói chung, dưới dạng tham số, các đường cong có dạng phức tạp hơn, chúng có thể tự cắt hoặc khép kín. Dưới đây là một vài dạng đường cong khép kín, đối xứng mà bạn có thể dùng chương trình trên để xem hình dáng của chúng:

$x = \cos^3(t)$, $y = \sin^3(t)$, $0 \le t \le 2\pi$ (đường hình sao - astroid)

$x = 2.\cos(t)-\cos(2t)$, $y = 2.\sin(t)-\sin(2t)$, $0 \le t \le 2\pi$ (đường hình tim - cardioid)

c. Vẽ đường cong r = F(φ)

Phương trình trong tọa độ cực là trường hợp riêng của phương trình theo tham số, trong đó góc cực φ đóng vai trò tham số t:

$r = F(\varphi)$

$x = r.\cos(\varphi)$

$y = r.\sin(\varphi)$

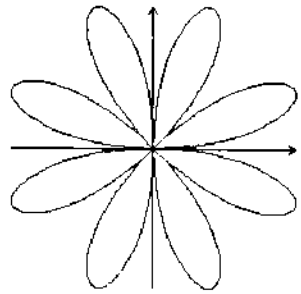
vì thế chúng được giải quyết giống như mô hình đã nêu.

Chương trình dưới đây vẽ đường hình hoa 8 cánh có phương trình trong tọa độ cực là $r = \sin(4\varphi)$, $0 \le \varphi \le 2\pi$:

```
Uses GRAPH;

  Function F(t: real): real;
  begin
    F := Sin(4*t);
  end;

Const
  n = 200;
  cmin = 1; dmin = 1; cmax = 200; dmax = 200;
  tmin = 0; tmax = 2*Pi;
Var
  xmin, xmax, ymin, ymax: real;
  x, y: array[0..n] of real;
  p, q: real;
  err: integer;

  procedure DefCoords;
  var i: integer;
      ti, xi, yi, h, r: real;
  begin
    h := (tmax-tmin)/n;
    ymin := 1e37; ymax := -ymin;
    xmin := ymin; xmax := ymax;
    for i := 0 to n do
    begin
      ti := tmin+i*h;
      r := F(ti);
      xi := r*Cos(ti);
      yi := r*Sin(ti);
      if xmin > xi then xmin := xi;
      if xmax < xi then xmax := xi;
      if ymin > yi then ymin := yi;
      if ymax < yi then ymax := yi;
```

```
      x[i] := xi;
      y[i] := yi;
    end;
    p := (cmax-cmin)/(xmax-xmin);
    q := (dmin-dmax)/(ymax-ymin);
  end;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure Draw;
  var i, c, d: integer;
      xi: real;
  begin
    c := Round(p*(x[0]-xmin)+cmin);
    d := Round(q*(y[0]-ymin)+dmax);
    MoveTo(c, d);
    for i := 1 to n do
    begin
      c := Round(p*(x[i]-xmin)+cmin);
      d := Round(q*(y[i]-ymin)+dmax);
      LineTo(c, d);
    end;
    d := Round(q*(-ymin)+dmax);
    if (d >= dmin) and (d <= dmax) then
    begin
      Line(cmin, d, cmax, d);
      Line(cmax, d, cmax-5, d-3);
      Line(cmax, d, cmax-5, d+3);
    end;
    c := Round(p*(-xmin)+cmin);
    if (c >= cmin) and (c <= cmax) then
    begin
      Line(c, dmin, c, dmax);
      Line(c, dmin, c-3, dmin+5);
      Line(c, dmin, c+3, dmin+5);
    end;
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
```

```
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  DefCoords;
  Draw;
  Readln;
  CloseGraph;
End.
```



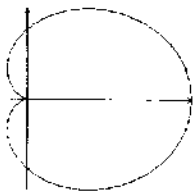
Đường hình hoa 8 cánh được vẽ bởi chương trình

Dưới đây là một số đường cong trong tọa độ cực để các bạn tham khảo:

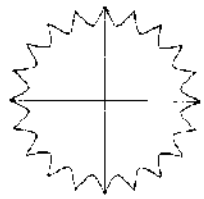
$r = 1+\cos(\varphi), 0 \le \varphi \le 2\pi$ (đường hình tim - cardioid)

$r = 5-|\sin(10\varphi)|, 0 \le \varphi \le 2\pi$ (đường hình bánh răng)

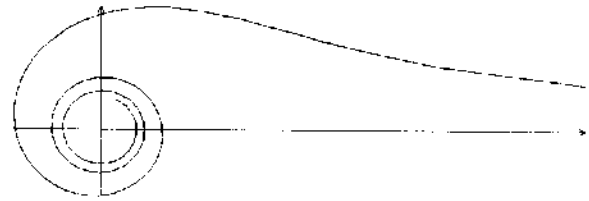
$r = \mathrm{sqrt}(\pi/\varphi), \varphi > 0$ (đường xoắn hình sên)



Đường hình tim



Đường bánh răng



Đường xoắn hình sên

8. Vẽ và tô miền

Bên cạnh các thủ tục vẽ đường, GRAPH còn xây dựng một loạt các thủ tục cho phép vẽ và tô các miền. Một miền được xác định bởi một chu tuyến bao quanh nó, thao tác tô miền là điền đầy vào miền đang xét một giá trị mầu tô và mẫu tô định trước.

8.1. Đặt thuộc tính tô miền

Để đặt thuộc tính cho các thao tác tô miền tiếp sau, GRAPH cung cấp thủ tục

```
procedure SetFillStyle(pattern: word; color: word);
```

trong đó `pattern` xác định mẫu tô và `color` xác định mầu tô. Các giá trị khả dĩ cho `color` là từ 0 đến số hiệu mầu lớn nhất của chế độ đồ họa hiện tại, trong đó giá trị ngầm định của `color` là số hiệu mầu lớn nhất. Các giá trị khả dĩ cho `pattern` gồm 13 giá trị được đặt tên theo ý nghĩa là:

0	EmptyFill	rỗng (tô với mầu nền)
1	SolidFill	trơn (ngầm định)
2	LineFill	gạch ngang
3	LtSlashFill	gạch chéo mảnh
4	SlashFill	gạch chéo đậm
5	BkSlashFill	gạch chéo ngược đậm
6	LtBkSlashFill	gạch chéo ngược mảnh
7	HatchFill	carô vuông
8	xHatchFill	carô chéo
9	InterleaveFill	xen
10	WideDotFill	chấm chấm thưa
11	CloseDotFill	chấm chấm mau
12	UserFill	tự thiết kế

Nếu không dùng `SetFillStyle` thì mẫu tô luôn được ngầm định là trơn (SolidFill).

Trong trường hợp dùng giá trị `UserFill` cho `pattern` thì mẫu tô và mầu tô được xác định bởi thủ tục:

```
procedure SetFillPattern(pattern: FillPatternType; color: word);
```

trong đó `FillPatternType` là một mảng gồm 8 byte đã được GRAPH định nghĩa sẵn:

```
Type
  FillPatternType = array[1..8] of byte;
```

dùng để chứa mẫu tô thiết kế.

Mỗi mẫu tô như vậy được xác định bởi một ma trận 8×8 bit với ý nghĩa bit 1 tương ứng với điểm sáng. Như vậy mẫu càng đặc nếu càng dùng nhiều bit 1. Bạn có thể thiết kế những hình vẽ khác nhau trên ma trận bit này để dùng làm mẫu tô bên cạnh những mẫu đã sẵn có. Chẳng hạn, bạn có thể thiết kế mẫu tô có đường gợn sóng như sau:

0	0	0	1	1	0	0	0	= $18	(byte thứ nhất)
0	0	1	0	0	1	0	0	= $24	(byte thứ hai)
0	1	0	0	0	0	1	0	= $42	(byte thứ ba)
1	0	0	0	0	0	0	1	= $81	(byte thứ tư)
0	0	0	1	1	0	0	0	= $18	(byte thứ năm)
0	0	1	0	0	1	0	0	= $24	(byte thứ sáu)
0	1	0	0	0	0	1	0	= $42	(byte thứ bảy)
1	0	0	0	0	0	0	1	= $81.	(byte thứ tám)

và gửi mẫu này vào một hằng kiểu `FillPatternType`:

```
Const
  WavePattern: FillPatternType =
               ($18, $24, $42, $81, $18, $24, $42, $81);
```

để sau này có thể truyền nó cho tham số `pattern` trong thủ tục `SetFillPattern`.

Để lấy lại thuộc tính tô miền hiện tại (nghĩa là mẫu tô và mầu tô được dùng lần cuối cùng trước đó bởi thủ tục `SetFillStyle`), bạn dùng thủ tục:

```
procedure GetFillSettings(var FillInfo: FillSettingsType);
```

trong đó `FillSettingsType` là kiểu đã được GRAPH định nghĩa:

```
Type
  FillSettingsType = record
                       pattern: word;
                       color: word;
                     end;
```

Lời gọi `GetFillSettings(FillInfo)` trả thuộc tính tô miền hiện đang dùng cho biến `FillInfo`, trong đó trường `pattern` giữ giá trị mẫu tô và trường `color` giữ giá trị mầu tô.

Để lấy lại bảng mẫu bit đã được thiết kế sau cùng bởi lời gọi `SetFillPattern`, bạn dùng thủ tục:

```
procedure GetFillPattern(var FillPattern: FillPatternType);
```

Thủ tục này sẽ trả về cho biến `FillPattern` 8 byte lưu trữ mẫu bit. Nếu trước đó không gọi thủ tục `SetFillPattern` thì thủ tục `GetFillPattern` sẽ trả về cả 8 byte có giá trị $FF (mẫu trơn).

Thuộc tính tô miền được dùng cho tất cả các thủ tục tô miền sau đó cho đến khi nó được đặt lại bởi `SetFillStyle` hoặc `SetFillPattern`. GRAPH xây dựng các thủ tục tô những miền chữ nhật, miền tròn, miền ellip, miền đa giác và đặc biệt có thủ tục tô một miền tổng quát, xác định bởi một chu tuyến bất kỳ.

8.2. Tô miền chữ nhật

Thủ tục:

```
procedure Bar(x1, y1, x2, y2: integer);
```

tô miền chữ nhật có `x1`, `y1` là tọa độ góc trên trái và `x2`; `y2` là tọa độ góc dưới phải.

Thủ tục `Bar` dùng để xây dựng các biểu đồ cột mô tả một dãy giá trị cho trước. Mỗi giá trị trong dãy được biểu diễn bởi một bar có độ rộng giống nhau và độ cao tỷ lệ với độ lớn của giá trị. Việc tính toán độ rộng và độ cao của mỗi bar trên màn hình phụ thuộc vào số lượng và độ lớn các giá trị trong dãy cũng như phụ thuộc vào kích thước khung nhìn bố trí trên màn hình.

Ngoài ra thủ tục `Bar` còn được dùng để xóa một vùng màn hình trong chế độ đồ họa, bằng cách dùng mẫu rỗng (0 - EmptyFill).

Để thí dụ, dưới đây xét hai chương trình dùng `Bar` để minh họa các mẫu tô và mầu tô được chọn.

Chương trình `DemoPattern` minh hoạ các mẫu tô (từ giá trị 0 đến 12). Một bar được trình bày trên màn hình đồ họa với mẫu tô tương ứng. Mầu tô cất trong biến `FillColor`, được cố định là mầu có số hiệu lớn nhất. Mẫu tô cất trong biến `FillPattern` và có giá trị

ban đầu là 1 (tô trơn). Các mẫu tô được thay đổi trên bar bằng cách gõ phím tương ứng để chọn. Quy ước các phím chọn mẫu tô (`PatternKey`) là các phím 0, 1, ..., 9, A, B, C (A-10, B-11, C-12). Chương trình kết thúc khi gõ phím Esc.

```
Program DemoPattern;
Uses GRAPH, CRT;
Const
  pattern: array[0..12] of string[14] =
         ('EmptyFill', 'SolidFill', 'LineFill', 'LtSlashFill',
          'SlashFill', 'BkSlashFill', 'LtBkSlashFill',
          'HatchFill', 'XHatchFill', 'InterleaveFill',
          'WideDotFill', 'CloseDotFill', 'UserFill');
  UserPatt: FillPatternType =
           ($18, $24, $42, $81, $18, $24, $42, $81);

Var FillPattern, FillColor: word;
    PatternKey: char;
    err: integer;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure DrawBar;
  var s: string;
  begin
    SetFillPattern(UserPatt, FillColor);
    SetFillStyle(FillPattern, FillColor);
    Bar(140, 100, 400, 310);
    Str(FillPattern, s);
    OutTextXY(140, 340,
              'FillPattern: '+s+' - '+Pattern[FillPattern]);
  end;

 Begin
   err := DetectInit('C:\TP\BGI');
   if err <> 0 then
   begin
     Write(GraphErrorMsg(err), '! Press Enter to quit...');
     Readln; Halt;
   end;
   FillPattern := 1; FillColor := GetMaxColor;
   OutTextXY(140, 70, 'DEMO FILLPATTERN');
```

```
  OutTextXY(140, 400, 'Pattern keys: 0, 1, ..., 9, A, B, C');
  OutTextXY(140, 420, 'Press Esc to quit.');
  PatternKey := #0;
  repeat
    DrawBar;
    PatternKey := UpCase(ReadKey);
    if PatternKey in ['0'..'9','A'..'C'] then
    begin
      case PatternKey of
        '0'..'9': FillPattern := Ord(PatternKey)-48;
        'A'..'C': FillPattern := Ord(PatternKey)-55;
      end;
      SetFillStyle(1, 0);
      Bar(140, 340, 400, 350);
    end;
  until PatternKey = #27;
  CloseGraph;
End.
```

Chương trình `DemoColor` minh hoạ các mầu tô (từ giá trị 0 đến 15). Một bar được trình bày trên màn hình đồ họa với mầu tô tương ứng. Mẫu tô cất trong biến `FillPattern` được cố định là mẫu trơn (mẫu 1). Mầu tô cất trong biến `FillColor` và có giá trị ban đầu là mầu có số hiệu cao nhất. Các mầu tô được thay đổi trên bar bằng cách gõ phím tương ứng để chọn. Quy ước các phím chọn mầu tô (`ColorKey`) là các phím 0, 1, ..., 9, A, ..., F (A-10, ..., F-15). Chương trình kết thúc khi gõ phím Esc.

```
Program DemoColor;
Uses GRAPH, CRT;
Const
  color: array[0..15] of string[12] =
       ('Black', 'Blue', 'Green', 'Cyan', 'Red', 'Magenta',
        'Brown', 'LightGray', 'DarkGray', 'LightBlue', 'LightGreen',
        'LightCyan', 'LightRed', 'LightMagenta', 'Yellow', 'White');

Var FillColor, FillPattern: word;
    ColorKey: char;
    err: integer;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;
```

```
  procedure DrawBar;
  var s: string;
  begin
    SetFillStyle(FillPattern, FillColor);
    Bar(140, 100, 400, 310);
    Str(FillColor, s);
    OutTextXY(140, 340, 'FillColor: '+s+' - '+Color[FillColor]);
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  FillPattern := 1; FillColor := GetMaxColor;
  OutTextXY(140, 70, 'DEMO FILLCOLOR');
  OutTextXY(140, 400, 'Color keys: 0, 1, ..., 9, A, B, ..., F');
  OutTextXY(140, 420, 'Press Esc to quit.');
  ColorKey := #0;
  repeat
    DrawBar;
    ColorKey := UpCase(ReadKey);
    if ColorKey in ['0'..'9','A'..'F'] then
    begin
      case ColorKey of
        '0'..'9': FillColor := Ord(ColorKey)-48;
        'A'..'F': FillColor := Ord(ColorKey)-55;
      end;
      SetFillStyle(EmptyFill, 0);
      Bar(140, 340, 400, 350);
    end;
  until ColorKey = #27;
  CloseGraph;
End.
```

Trong các chương trình trên có dùng thủ tục `OutTextXY` để đưa một thông báo lên màn hình đồ họa (xem mục **10.2.** ở dưới).

Unit GRAPH hỗ trợ thêm cho `Bar` bằng một thủ tục tương tự nhưng trong không gian 3 chiều, đó là thủ tục:

```
procedure Bar3D(x1, y1, x2, y2: integer; depth: word; top: boolean);
```

Thủ tục này tô khối hộp với mẫu tô và mầu tô hiện thời. Góc trên trái của khối hộp là `x1`, `y1`, góc dưới phải của khối hộp là `x2`, `y2`. Tham trị `depth` xác định bề dày của khối hộp. Có hai trạng thái được vẽ cho khối hộp: Nếu tham trị `top` là `True` (được đặt tên là

TopOn) thì khối hộp có nắp, trái lại (được đặt tên là TopOff) khối hộp không có nắp. Việc tổ chức khối hộp không có nắp nhằm tạo nét khuất khi vẽ hai khối hộp chồng lên nhau.

8.3. Tô miền tròn & ellip

Thủ tục

```
procedure PieSlice(x, y: integer; a1, a2, r: word);
```

tô mảnh quạt xác định bởi đường tròn tâm x, y, bán kính r, bắt đầu từ góc a1 đến góc a2 (đo theo đơn vị độ). Nếu a1 = 0 và a2 = 360 thì thủ tục này tô toàn bộ hình tròn.

Thủ tục PieSlice dùng để xây dựng các biểu đồ quạt, nhằm mô tả độ lớn của các thành phần so với toàn bộ.

Tương tự, thủ tục

```
procedure Sector(x, y: integer; a1, a2, rx, ry: word);
```

tô mảnh quạt xác định bởi đường ellip tâm x, y, bán kính rx, ry bắt đầu từ góc a1 đến góc a2 (đo theo đơn vị độ).

Mặc dù việc tô toàn bộ ellip là trường hợp riêng của thủ tục này nhưng GRAPH dựng riêng một thủ tục

```
procedure FillEllipse(x, y: integer; rx, ry: word);
```

để tô toàn bộ ellip tâm x, y, bán kính rx, ry. Việc tô toàn bộ hình tròn được xem là trường hợp riêng của FillEllipse khi chọn rx = ry.

Chương trình minh họa dưới đây mô tả một bầu trời đêm có nguyệt thực toàn phần với những ngôi sao nhấp nháy. Các ngôi sao được tạo bởi PutPixel với mầu ngẫu nhiên, trong đó một số ngôi nhấp nháy được tạo bởi FillEllipse với bán kính 1 và mầu luân phiên sáng tối. Mặt trăng được tạo bởi FillEllipse với mầu vàng và bán kính 20. Bóng trái đất được tạo bởi FillEllipse với mầu đen và bán kính 30. Cứ mỗi chu kỳ nhấp nháy các ngôi sao, bóng trái đất lại được dịch chuyển một bước để che dần dần mặt trăng. Bạn có thể tăng, giảm tốc độ dịch chuyển này bằng cách điều chỉnh giá trị hằng dt trong thủ tục Sky. Trạng thái của bầu trời được tạo lại một cách ngẫu nhiên mỗi khi có một phím được nhấn hoặc mặt trăng bị che khuất hoàn toàn. Trong chương trình có dùng thủ tục ClearDevice (xem mục **9** ở dưới) để xóa màn hình đồ họa hiện tại trước khi tạo một màn hình mới. Chương trình kết thúc khi nhấn phím Esc.

```
Program LunarEclipse;
Uses GRAPH, CRT;
Var MaxX, MaxY, err: integer;
    MaxColor: word;
    ch: char;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
```

```
  DetectInit := GraphResult;
end;

procedure Sky;
const
  nStar = 500;   {số ngôi sao}
  r1 = 20;       {bán kính mặt trăng}
  r2 = 30;       {bán kính bóng trái đất}
  dt = 0.001;    {bước dịch chuyển bóng trái đất}
type
  StarType = record
               col, row, color: integer;
             end;
var
  BlinkStar: array[1..nStar] of StarType;
  i, x1, y1, x2, y2, r, nBlink, x, y, c: integer;
  t, u1, v1, u2, v2: real;
begin
  {xác định tâm mặt trăng}
  x1 := r1+Random(MaxX-r1-r1);
  y1 := r1+Random(MaxY-r1-r1);
  {xác định tâm bóng trái đất}
  t := Random*2*Pi;
  r := r1+r2;
  x2 := Round(x1+r*cos(t));
  y2 := Round(y1+r*sin(t));
  {sinh và vẽ các sao}
  nBlink := 0;   {số ngôi sao nhấp nháy}
  for i := 1 to nStar do
  begin
    x := Random(MaxX+1);
    y := Random(MaxY+1);
    c := 1+Random(MaxColor);
    PutPixel(x, y, c);      {vẽ sao}
    {giữ lại ngẫu nhiên những ngôi sao nhấp nháy}
    u1 := x-x1; v1 := y-y1;
    u2 := x-x2; v2 := y-y2;
    if (random < 0.1) and (Sqrt(Sqr(u1)+Sqr(v1)) > 2*r1)
                      and (Sqrt(Sqr(u2)+Sqr(v2)) > 2*r2) then
    begin
      Inc(nBlink);
      with BlinkStar[nBlink] do
      begin
        col := x; row := y; color := c;
      end;
    end;
  end;
```

```
    {vẽ mặt trăng}
    SetColor(yellow);
    SetFillStyle(SolidFill, yellow);
    FillEllipse(x1, y1, r1, r1);
    {thực hiện di chuyển bóng trái đất và nhấp nháy các ngôi sao}
    i := 1; t := 0;
    while not KeyPressed do
    begin
      if i > nBlink then      {di chuyển bóng trái đất}
      begin
        x := Round(t*(x1-x2)+x2);      {xác định tâm mới bóng trái đất}
        y := Round(t*(y1-y2)+y2);
        SetColor(0);                   {vẽ bóng trái đất}
        SetFillStyle(EmptyFill, 0);
        FillEllipse(x, y, r2, r2);
        if t >= 0.9 then Break;        {thoát khi mặt trăng bị che hết}
        t := t+dt;
        i := 1;
      end;
      with BlinkStar[i] do     {nhấp nháy ngôi sao i}
      begin
        color := (color+(MaxColor+1) div 2) mod (MaxColor+1);
        SetFillStyle(SolidFill, color);
        SetColor(color);
        FillEllipse(col, row, 1, 1);
      end;
      Inc(i);
      Delay(5);
    end;
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  MaxX := GetMaxX; MaxY := GetMaxY; MaxColor := GetMaxColor;
  Randomize;
  repeat
    Sky;
    if KeyPressed then ch := ReadKey;
    ClearDevice;
  until ch = #27;
  CloseGraph;
End.
```

8.4. Tô miền đa giác

Với cách tổ chức tham số giống như `DrawPoly`, thủ tục

```
procedure FillPoly(NumPoints: word; var PolyPoints);
```

tô miền đa giác có số đỉnh `NumPoints` và có tọa độ các đỉnh lưu trong `PolyPoints`. Điểm khác `DrawPoly` là `FillPoly` không cần thêm một đỉnh trùng với đỉnh đầu tiên vì nó luôn hiểu biên đa giác là khép kín.

Để làm thí dụ, ta thử dựng một đa giác hình sao 5 cánh. Gọi tâm của đa giác này là O và 10 đỉnh theo thứ tự là P_0, P_1, ..., P_9. Các đỉnh chỉ số chẵn nằm cách đều nhau trên vòng tròn bán kính $r = OP_0$ (gọi là bán kính của sao) và các đỉnh chỉ số lẻ nằm cách đều nhau trên vòng tròn bán kính OP_1. Gọi góc nghiêng của OP_0 so với phương nằm ngang là α. Từ đó dễ dàng tính được tọa độ của các đỉnh P_i theo công thức:

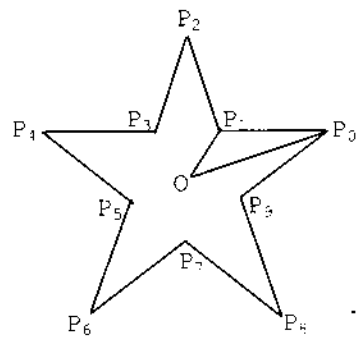
$$x = x_0+q.\cos(\alpha+i.\pi/5)$$

$$y = y_0+q.\sin(\alpha+i.\pi/5)$$

trong đó x_0, y_0 là tọa độ tâm O và q được xác định bởi:

$q = r$ với i chẵn (i = 0, 2, 4, 6, 8)

$q = r/(2.\cos(\pi/5)+1)$ với i lẻ (i = 1, 3, 5, 7, 9)



Từ phân tích trên, dễ dàng xây dựng được thủ tục `Star`, tô ngôi sao 5 cánh có tâm tại `x0`, `y0`, bán kính `r`, góc nghiêng `anfa` với mẫu tô trơn và mầu tô `color`.

Chương trình `DemoStar` dưới đây dùng thủ tục `Star` vẽ các ngôi sao lên màn hình. Tâm của sao được sinh ngẫu nhiên trong phạm vi màn hình, bán kính được sinh ngẫu nhiên trong khoảng từ 20 đến 100, góc nghiêng được sinh ngẫu nhiên trong khoảng từ 0 đến $2\pi/5$ và mầu tô được sinh ngẫu nhiên trong khoảng từ 1 đến số hiệu mầu cao nhất. Chương trình kết thúc khi ấn phím Esc.

```
Program DemoStar;
Uses GRAPH, CRT;
Var err, MaxX, MaxY: integer;
    MaxColor: word;
    ch: char;
```

```
  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure Star(x0, y0: integer; r: word; anfa: real; color: word);
  const beta = Pi/5;
  var i, q, rr: word;
      P: array[0..9] of PointType;
  begin
    rr := Round(r/(2*Cos(beta)+1));
    for i := 0 to 9 do   {xác định tọa độ các đỉnh P[i]}
    begin
      if Odd(i) then q := rr else q := r;
      with P[i] do
      begin
        x := x0+Round(q*Cos(anfa+i*beta));
        y := y0+Round(q*Sin(anfa+i*beta));
      end;
    end;
    SetFillStyle(1, color);
    SetColor(color);
    FillPoly(10, P);    {tô sao}
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  MaxX := GetMaxX; MaxY := GetMaxY; MaxColor := GetMaxColor;
  Randomize;
  repeat
    Star(Random(MaxX), Random(MaxY), 20+Random(100),
         2*Pi/5*Random, 1+Random(MaxColor));
    if KeyPressed then ch := ReadKey;
  until ch = #27;
  CloseGraph;
End.
```

Có thể cải tiến chút ít thủ tục `Star` để ngôi sao được tô nổi. Muốn vậy, thay vì tô toàn bộ ngôi sao, ta tô lần lượt các tam giác con OP_0P_1, OP_1P_2, ... xen kẽ nhau bằng các mẫu sáng (`SolidFill`), tối (`InterleaveFill`) như đề nghị dưới đây:

```
procedure Star(x0, y0: integer; r: word; anfa: real; color: word);
const
  beta = Pi/5;
type
  TriAngle = record      {kiểu tam giác}
               A, B, C: PointType;
             end;
var i, q, rr, pattern: word;
    P: array[0..9] of PointType;
    T: TriAngle;
begin
  rr := Round(r/(2*Cos(beta)+1));
  for i := 0 to 9 do     {xác định tọa độ các đỉnh P[i]}
  begin
    if odd(i) then q := rr else q := r;
    with P[i] do
    begin
      x := Round(x0+q*Cos(anfa+i*beta));
      y := Round(y0+q*Sin(anfa+i*beta));
    end;
  end;
  T.A.x := x0; T.A.y := y0;   {xác định và tô các tam giác T}
  for i := 0 to 9 do
  begin
    T.B := P[i mod 10];
    T.C := P[(i+1) mod 10];
    if Odd(i) then pattern := SolidFill   {chọn pattern sáng}
    else pattern := InterleaveFill;       {chọn pattern tối}
    SetColor(color);
    SetFillStyle(pattern, color);
    FillPoly(3, T);       {tô tam giác T}
  end;
end;
```

Bạn có thể thử lại chương trình `DemoStar` với nội dung `Star` vừa xây dựng.

8.5. Tô miền bất kỳ

GRAPH xây dựng một thủ tục tô miền tổng quát, trong đó miền được xác định bởi một chu tuyến bất kỳ:

```
procedure FloodFill(x, y: integer; border: word);
```

Thủ tục này đòi hỏi `x`, `y` là tọa độ của một điểm nằm bên trong miền và chu tuyến được vẽ bởi một mầu đồng nhất có số hiệu `border`. Nếu điểm `x`, `y` nằm ngoài miền thì miền

bên ngoài được tô. Nếu chu tuyến không kín hoặc không đảm bảo có mầu đồng nhất thì việc tô sẽ bị tràn ra ngoài.

Thủ tục `FloodFill` xuất phát từ điểm `x, y`, dùng thuật toán quét tổng quát để tô, vì thế nó không cần biết các đặc tính hình học của đường biên mà nó chỉ căn cứ mầu `border` của biên để làm giới hạn. Cũng vì vậy, tốc độ tô của thủ tục này chậm hơn nhiều so với các thủ tục tô những miền đặc biệt. Nó dành cho việc tô những hình có đường biên phức tạp, kết hợp của nhiều đường được vẽ.

Cũng cần chú ý rằng, nhiều khi `FoodFill` không tô đúng hình mong muốn vì bên cạnh chu tuyến của hình, còn có thể có nhiều đường vẽ sẵn có trên màn hình, giao cắt với chu tuyến và điều này thì `FloodFill` không thể phân biệt được. Ngoài ra `FloodFill` cũng đòi hỏi một vùng nhớ nhất định nên có thể không thực hiện được nếu khi đang tô mà bị thiếu bộ nhớ. Trong trường hợp này `GraphResult` trả về mã lỗi -7 (*grNoFloodMem*).

Thủ tục `DrawSaucer` trong chương trình minh họa dưới đây, mô phỏng hình một đĩa bay, có tâm tại điểm `x, y`, bán kính `r` và mầu `color`. Thân của đĩa bay được tạo bởi hai cung ellip và được tô trơn bởi thủ tục `FloodFill`. Chương trình dùng thủ tục `DrawSaucer` vẽ một đĩa bay ở vị trí giữa màn hình, bán kính 50 và có mầu trắng. Sau khi đĩa bay được vẽ, bạn có thể dùng các phím để điều khiển:

- Di chuyển theo 4 hướng — các phím mũi tên ← ↑ → ↓
- Phóng to, thu nhỏ — các phím '+', '-' .
- Chọn mầu từ 1 đến 15 — các phím '1', .., '9', 'A', .., 'F'
- Kết thúc chương trình — phím Esc.

```
Program DemoSaucer;
Uses GRAPH, CRT;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure DrawSaucer(x, y: integer; r, color: word);
  {vẽ đĩa bay tâm x, y, bán kính r, mầu color}
  var r1, r2, r3, r4, r5: word;
  begin
    r1 := Round(r/2); r2 := Round(r/3); r3 := Round(r/5);
    r4 := 2*r2; r5 := Round(r/10);
    SetColor(color);
    Ellipse(x, y, 0, 360, r, r1);
    Ellipse(x, y-r3, 194, 346, r, r2);
    Line(x+r2, y-r2, x+r4, y-r4);
```

```
    Circle(x+r4, y-r4, r5);
    Line(x-r2, y-r2, x-r4, y-r4);
    Circle(x-r4, y-r4, r5);
    SetFillStyle(SolidFill, Color);
    FloodFill(x, y+r2-r3+1, Color);
  end;

var MaxX, MaxY, x, y: integer;
    MaxColor, color, r: word;
    ch: char;

Begin
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! An phim Enter de ket thuc...');
    Readln; Halt;
  end;
  MaxX := GetmaxX; MaxY := GetMaxY; MaxColor := GetMaxColor;
  x := MaxX div 2; y := MaxY div 2; r := 50; color := MaxColor;
  DrawSaucer(x, y, r, color);
  repeat
    ch := #1;
    if KeyPressed then ch := UpCase(ReadKey);
    if ch = #0 then      {mã quét mở rộng}
    begin
      ch := ReadKey;
      case Ord(ch) of
        75: if x > r then          {dịch sang trái}
            begin
              DrawSaucer(x, y, r, 0);
              Dec(x, r div 5);
              DrawSaucer(x, y, r, color);
            end;
        72: if y > r then          {dịch lên trên}
            begin
              DrawSaucer(x, y, r, 0);
              Dec(y, r div 5);
              DrawSaucer(x, y, r, color);
            end;
        77: if x < MaxX-r then     {dịch sang phải}
            begin
              DrawSaucer(x, y, r, 0);
              Inc(x, r div 5);
              DrawSaucer(x, y, r, color);
            end;
        80: if y < MaxY-r then     {dịch xuống dưới}
            begin
```

```
            DrawSaucer(x, y, r, 0);
            Inc(y, r div 5);
            DrawSaucer(x, y, r, color);
          end;
      end;
    end else
      case ch of
          '+': if r < 200 then   {tăng kích thước}
               begin
                 DrawSaucer(x, y, r, 0);
                 Inc(r, r div 5);
                 DrawSaucer(x, y, r, color);
               end;
          '-': if r > 10 then    {giảm kích thước}
               begin
                 DrawSaucer(x, y, r, 0);
                 Dec(r, r div 5);
                 DrawSaucer(x, y, r, color);
               end;
      '1'..'9': begin
                 color := Ord(ch)-48;      {mầu từ 1 đến 9}
                 DrawSaucer(x, y, r, color);
               end;
      'A'..'F': begin
                 color := Ord(ch)-87;      {mầu từ 10 đến 15}
                 DrawSaucer(x, y, r, color);
               end;
      end;
  until ch = #27;
  CloseGraph;
End.
```

9. Khung nhìn

Một khung nhìn sẽ được thiết lập trên màn hình đồ họa bằng lời gọi thủ tục:

```
procedure SetViewPort(x1, y1, x2, y2: integer; clip: boolean);
```

trong đó `x1`, `y1` là tọa độ góc trên trái và `x2`, `y2` là tọa độ góc dưới phải của khung. Tham trị `clip` xác định các nét vẽ có được thực hiện bên ngoài khung nhìn hay không. Nếu `clip` là `True` (giá trị tương ứng được đặt tên là `ClipOn`) thì nét vẽ bị cắt, trái lại (giá trị tương ứng được đặt tên là `ClipOff`), nét vẽ vẫn được thực hiện bên ngoài khung nhìn.

Khi khung nhìn được thiết lập, các giá trị tọa độ đều được hiểu là tương đối so với khung nhìn này (nghĩa là 0, 0 trở thành tọa độ góc trên trái của khung nhìn). Nhờ vậy, các giá trị tọa độ khi làm việc với khung nhìn có thể nhận giá trị âm. Chẳng hạn lời gọi:

```
SetViewPort(GetMaxX div 2, GetMaxY div 2, GetMaxX, GetMaxY, ClipOff);
```

sẽ làm cho điểm giữa màn hình có tọa độ 0, 0 và tất cả các điểm nằm ở góc phần tư phía trên trái của màn hình đều có tọa độ cột, dòng âm.

Giá trị ngầm định của khung nhìn là toàn bộ màn hình. Có thể lập lại giá trị này bằng lời gọi:

```
SetViewPort(0, 0, GetMaxX, GetMaxY, ClipOn);
```

Các thủ tục `InitGraph`, `SetGraphMode` luôn luôn thiết lập lại khung nhìn là toàn bộ màn hình.

Thông tin về khung nhìn hiện tại có thể lấy ra bằng thủ tục:

```
procedure GetViewSettings(var ViewPort: ViewPortType);
```

trong đó `ViewPortType` được định nghĩa:

```
Type
  ViewPortType = record
                   x1, y1, x2, y2: integer;
                   clip: boolean;
                 end;
```

Để xoá tất cả các nét vẽ trong khung nhìn hiện tại, GRAPH xây dựng thủ tục:

```
procedure ClearViewPort;
```

còn để xoá toàn bộ màn hình đồ họa, dùng thủ tục:

```
procedure ClearDevice;
```

Chú ý không dùng thủ tục `ClrScr` của unit CRT để xoá màn hình đồ họa vì thủ tục này làm hỏng màn hình đồ họa đang dùng.

10. Viết chữ trong chế độ đồ họa

Có thể dùng các thủ tục `Write`, `Writeln` để đưa các thông báo lên màn hình đồ họa. Tuy nhiên các thủ tục này vẫn xem màn hình đồ họa như màn hình văn bản, nghĩa là mỗi ký tự được thể hiện trên màn hình tại một vị trí cột, dòng trong độ phân giải 80×25. Muốn các chữ viết được xử lý như một nét vẽ, có thể phóng to, thu nhỏ và dịch chuyển từng pixel, cần phải dùng các thủ tục viết chữ do GRAPH cung cấp.

Các thủ tục viết chữ của GRAPH, ngoài việc dùng font chữ hệ thống (là font bitmap), còn dùng một số các font chữ vectơ. Mỗi font này được Turbo Pascal cung cấp bởi một file có phần mở rộng là CHR. Với phiên bản 7, Turbo Pascal có 10 font như vậy.

10.1. Đặt thuộc tính nét chữ

Cũng giống mọi lệnh vẽ khác, trước khi viết chữ trong đồ họa, cần phải xác định thuộc tính cho nét chữ. Công việc này được thực hiện bởi thủ tục:

```
procedure SetTextStyle(font, direction, CharSize: word);
```

trong đó `font` xác định font chữ được dùng, `direction` xác định hướng viết, và `CharSize` xác định kích thước chữ.

Có 11 giá trị cho `font` (trong đó chỉ có các tên font từ số 0 đến số 4 là đã được định nghĩa, các font từ số 5 đến số 10 mới được phiên bản 7 bổ sung). Giá trị ngầm định của `font` là 0 (font chữ hệ thống). Mỗi font vectơ đều cần tương ứng một file CHR:

giá trị	tên	file tương ứng
0 (ngầm định)	DefaultFont	không có
1	TriplexFont	TRIP.CHR
2	SmallFont	LITT.CHR
3	SansSerifFont	SANS.CHR
4	GothicFont	GOTH.CHR
5	ScriptFont	SCRI.CHR
6	SimplexFont	SIMP.CHR
7	TriplexScriptFont	TSCR.CHR
8	ComplexFont	LCOM.CHR
9	EuropeanFont	EURO.CHR
10	BoldFont	BOLD.CHR

Tham trị `direction` nhận hai giá trị 0 và 1. Giá trị ngầm định của `direction` là 0:

giá trị	tên	hướng viết
0 (ngầm định)	HorizDir	ngang, từ trái sang phải
1	VertDir	dọc, từ dưới lên trên

Có 10 cỡ chữ (từ 1 đến 10) định sẵn cho `CharSize`. Giá trị của `CharSize` càng lớn thì kích thước chữ càng lớn. Với font hệ thống, cỡ chữ bình thường là 1, với các font vectơ, cỡ chữ bình thường là 4.

Có thể tự quy định cỡ chữ đối với các font vectơ bằng cách dùng thủ tục:

```
procedure SetUserCharSize(MultX, DivX, MultY, DivY: word);
```

trong đó `MultX` : `DivX` là tỷ lệ được nhân với độ rộng chuẩn và `MultY` : `DivY` là tỷ lệ được nhân với độ cao chuẩn của font chữ đang hoạt động. Nhờ thủ tục này, bạn có thể dãn rộng hoặc kéo cao cỡ chữ theo ý muốn. Chẳng hạn, lời gọi `SetUserCharSize(3, 1, 1, 1)` sẽ dãn rộng cỡ chữ gấp 3 lần bình thường còn lời gọi `SetUserCharSize(1, 1, 3, 1)` sẽ kéo cao cỡ chữ gấp 3 lần bình thường.

Thông tin về nét chữ hiện tại được lấy ra từ thủ tục:

```
procedure GetTextSettings(var TextInfo: TextSettingsType);
```

trong đó `TextSettingsType` được định nghĩa:

```
Type
  TextSettingsType = record
                       font: word;
                       direction: word;
                       CharSize: word;
```

```
    horiz: word;
    vert: word;
  end;
```

(các giá trị `horiz`, `vert` được giải thích trong thủ tục căn chữ `SetTextJustify` ở dưới).

Nếu không gọi `SetTextStyle` trước khi viết chữ thì các nét chữ sẽ lấy các giá trị ngầm định, nghĩa là font hệ thống được dùng, hướng viết nằm ngang từ trái sang phải và cỡ chữ là 1. Mầu của nét chữ được xác định bởi `SetColor` (giống như mầu của nét vẽ).

10.2. Viết xâu chữ lên màn hình

GRAPH cung cấp 2 thủ tục viết xâu chữ lên màn hình:

```
procedure OutTextXY(x, y: integer; s: string);
procedure OutText(s: string);
```

Thủ tục `OutTextXY` đưa xâu chữ `s` ra màn hình bắt đầu tại tọa độ `x`, `y`, không phụ thuộc vào vị trí CP cũng như không làm thay đổi vị trí CP, còn thủ tục `OutText` đưa xâu chữ `s` ra màn hình bắt đầu tại vị trí CP và đẩy CP tới vị trí mới (giống như `Line` so với `LineTo`).

Khác với các thủ tục `Write`, `Writeln`, các thủ tục `OutText`, `OutTextXY` xê dịch các nét chữ theo từng pixel và không có nền kèm theo vì thế các nét chữ sẽ đan xen nhau nếu chúng cùng được đưa ra tại một vị trí, ngoài ra các thủ tục này chỉ đưa ra được biểu thức `string`, không đưa ra được các biểu thức số hoặc lôgic như các thủ tục `Write`, `Writeln`. Điều này dẫn đến các chú ý dưới đây khi dùng `OutText`, `OutTextXY`:

- Trước khi đưa ra một xâu chữ tại vị trí đã có xâu chữ khác cần phải xóa xâu chữ hiện có như xóa một hình vẽ. Thường người ta xóa xâu chữ bằng việc tô một bar vừa đủ chứa xâu đó với mầu nền.
- Để đưa giá trị một biểu thức số hoặc lôgic ra màn hình phải đổi chúng thành biểu thức xâu tương ứng.
- Khoảng cách giữa các xâu chữ phải tính bằng đơn vị pixel sao cho phù hợp với font và cỡ chữ hiện dùng, không thể tính bằng đơn vị con chữ như trong chế độ văn bản.

Nói chung, việc viết chữ bằng `OutText`, `OutTextXY` phải xử lý phức tạp hơn so với việc dùng các thủ tục `Write`, `Writeln`, tuy nhiên chữ viết sẽ linh hoạt hơn nhiều vì nó có thể thay đổi to, nhỏ và dịch chuyển từng pixel theo hình vẽ.

Thí dụ dưới đây viết xâu chữ TURBO lên màn hình, tạo một bóng nổi cho nó bằng cách xê dịch xâu chữ một đoạn thích hợp. Xâu chữ được viết với font số 5 (ScriptFont), hướng ngang, cỡ chữ 8:

```
Uses GRAPH;
Var
  drive, mode, x, y, dx, dy, i: integer;
Begin
  drive := 0;
  InitGraph(drive, mode, 'C:\TP\BGI');
  SetTextStyle(5, 0, 8);
```

```
  x := 200; y := 200; dx := 2; dy := 1;
  for i := 1 to 8 do
    OutTextXY(x+(i-1)*dx, y-(i-1)*dy, 'TURBO');
  Readln;
  CloseGraph;
End.
```

Xâu chữ TURBO được viết bởi chương trình

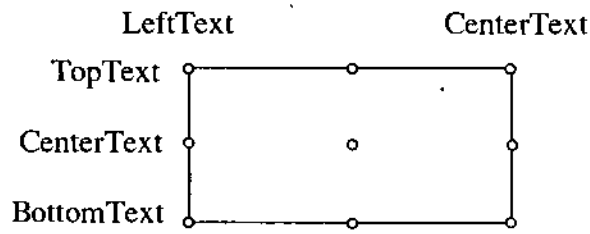
Để tính kích thước (theo pixel) một xâu chữ, GRAPH xây dựng hai hàm:

```
function TextHeight(s: string): word;
function TextWidth(s: string): word;
```

trả về chiều cao (`TextHeight`) và chiều rộng (`TextWidth`) theo pixel của xâu chữ `s`. Giá trị trả về được tính tùy thuộc vào font và cỡ chữ đang dùng. Những hàm này là có ích khi phải tính khoảng cách giữa các xâu chữ theo cột hoặc dòng để luôn phù hợp với font và cỡ chữ. Chẳng hạn, người ta thường lấy `TextHeight('H')` làm đơn vị cách dòng và `TextWidth('M')` làm đơn vị cách cột. Thủ tục `OutTextln(s)` dưới đây mô phỏng thủ tục `Writeln(s)`, viết xâu chữ `s` sau đó xuống dòng để chuẩn bị viết từ đầu dòng kế tiếp:

```
procedure OutTextln(s: string);
begin
  OutText(s);
  MoveTo(0, GetY+TextHeight('H'));
end;
```

Khi viết chữ tại tọa độ x, y, vị trí của xâu chữ còn phụ thuộc vào điểm đặt tọa độ này trong xâu chữ. Có 9 vị trí, tổ hợp của 3 giá trị theo chiều ngang và 3 giá trị theo chiều dọc, dùng để căn xâu chữ xung quanh điểm đặt x, y:



9 vị trí căn chữ có thể đặt trong xâu chữ

Việc chọn vị trí căn chữ được thực hiện bởi thủ tục:

```
procedure SetTextJustify(horiz, vert: word);
```

trong đó `horiz` xác định vị trí theo chiều ngang và `vert` xác định vị trí theo chiều dọc:

horiz		vert	
giá trị	tên	giá trị	tên
0	LeftText	0	BottomText
1	CenterText	1	CenterText
2	RightText	2	TopText

Giá trị ngầm định của `horiz` là 0 (LeftText) và của `vert` là 2 (TopText), nghĩa là vị trí căn chữ được đặt tại góc trên trái của xâu chữ.

Tùy các giá trị căn chữ (`horiz`, `vert`) mà vị trí của xâu chữ được phân bố 9 vị trí xung quanh điểm đặt, trong đó giá trị (1, 1) tương ứng với vị trí trung tâm (xem bản đồ ở dưới). Nhờ căn chữ, các xâu chữ được đặt chính xác vào những nơi mong muốn.

(0, 2)	(1, 2)	(2, 2)
(0, 1)	(1, 1)	(2, 1)
(0, 0)	(1, 0)	(2, 0)

Bản đồ các vị trí căn chữ theo các giá trị (`horiz`, `vert`)

Chương trình `DemoFonts` dưới đây minh hoạ các font chữ khác nhau trong chế độ đồ họa (gồm 11 font từ giá trị 0 đến 10). Các ký tự được trình bày trong một khung chữ nhật với font tương ứng. Các cỡ chữ được định sẵn trong mảng `CharSize[0..10]`. Font hiện hành được cất trong biến `CurrFont` (kiểu `word`). Giá trị ban đầu của `CurrFont` là 0 (font chữ hệ thống) và thay đổi tùy theo việc gõ phím chọn. Quy định các phím chọn font (`FontKey`) là 0, 1, ..., 9, A (A-10). Chương trình kết thúc khi gõ phím Esc.

```
Program DemoFonts;
Uses GRAPH, CRT;
Const
  Fonts : array[0..10] of string[17] =
  ('DefaultFont', 'TriplexFont', 'SmallFont', 'SansSerifFont',
   'GothicFont', 'ScriptFont', 'SimplexFont', 'TriplexScriptFont',
   'ComplexFont', 'EuropeanFont', 'BoldFont');

Var
  CurrFont: word;
  x1, x2, y1, y2, err: integer;
  FontKey: char;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;
```

```
  procedure TextDump;
  {hiển thị các ký tự lên màn hình}
  const
    CharSize: array[0..10] of word =
              (3, 4, 7, 4, 4, 4, 4, 4, 4, 2, 2);
  var
    s, t: string;
    ch : char;
  begin
    Rectangle(x1, y1, x2, y2);
    SetTextStyle(0, HorizDir, 1);
    Str(CurrFont, s); Str(CharSize[CurrFont], t);
    OutTextXY(180, 20, 'DEMO FONT '+s+' - '+
                       Fonts[CurrFont]+' size '+t);
    OutTextXY(100, y2+10,
             'Font keys: 0, 1, ..., 9, A        Press Esc to quit');
    SetTextStyle(CurrFont, HorizDir, CharSize[CurrFont]);
    MoveTo(40+TextWidth('M'), 53);
    if CurrFont = 0 then ch := #0 else ch := '!';
    repeat
      OutText(ch);
      if (GetX + TextWidth('M')) > (x2-x1) then
        MoveTo(40+TextWidth('M'), GetY + TextHeight('M')+3);
      ch := Succ(ch);
    until (ch >= #255);
  end;

Begin
  err := DetectInit('C:\TP\BGI');
  if err <> 0 then
  begin
    Write(GraphErrorMsg(err), '! Press Enter to quit...');
    Readln; Halt;
  end;
  CurrFont := 0;
  x1 := 40; x2 := 600; y1 := 50; y2 := 450;
  repeat
    TextDump;
    FontKey := Upcase(ReadKey);
    if FontKey in ['0'..'9', 'A'] then
    begin
      case FontKey of
        '0'..'9': CurrFont := Ord(FontKey)-48;
        'A'     : CurrFont := Ord(FontKey)-55;
      end;
      ClearDevice;
```

```
    end;
  until FontKey = #27;
  CloseGraph;
End.
```

Ghi chú. Trong chế độ đồ họa, nếu chương trình liên kết với unit CRT thì các thủ tục `Write`, `Writeln` có thể không đưa được các thông báo ra màn hình. Nguyên nhân do unit CRT nắm quyền trực tiếp truy cập vào vùng Video Ram mà không qua con đường BIOS (như thế tốc độ ra màn hình sẽ nhanh hơn). Muốn gỡ bỏ phương thức này để các thủ tục `Write`, `Writeln` hoạt động như bình thường, bạn đặt lại giá trị cho biến `DirectVideo` (là biến được định nghĩa trong CRT) bằng `False` (giá trị ngầm định là `True`), trước khi dùng các thủ tục này. Mặc dù đã có `OutText`, `OutTextXY`, các thủ tục `Write`, `Writeln` vẫn được dùng trong đồ họa khi muốn đưa ra nhanh chóng các kết quả tính toán mà không phải xử lý phức tạp như dùng `OutText` hay `OutTextXY`.

11. Vẽ ảnh chuyển động

Một ảnh chuyển động được cảm nhận từ một chuỗi đan xen hiện / ẩn ảnh theo quỹ đạo của nó. Giả sử thủ tục `ShowObj(x, y)` hiện ảnh của đối tượng tại vị trí `x, y` và ngược lại, thủ tục `HideObj(x, y)` ẩn ảnh của đối tượng tại vị trí `x, y`. Giả sử độ lệch của vị trí kế tiếp so với vị trí hiện thời là `dx, dy`. Khi đó mô hình chuyển động ảnh có thể xây dựng như sau:

```
x := x0; y := y0;        {vị trí bắt đầu}
repeat
  ShowObj(x, y);         {hiện ảnh đối tượng tại vị trí mới}
  {tính dx, dy theo quỹ đạo}
  ..................
  xx := x; yy := y;      {lưu vị trí cũ}
  x := x+dx; y := y+dy;  {tính vị trí mới}
  Delay(t);              {làm chậm lại với t thích hợp}
  HideObj(xx, yy);       {ẩn ảnh đối tượng tại vị trí cũ}
until stop;
```

Để ảnh đỡ bị nháy khi chuyển động, các thao tác `ShowObj` và `HideObj` cần phải thực hiện nhanh. Nếu từ `ShowObj` đến `HideObj` đòi hỏi phải có một khoảng thời gian thích hợp để mắt kịp cảm nhận, thì khoảng thời gian từ `HideObj` đến `ShowObj` càng nhanh càng tốt.

Thông thường, nếu đối tượng nhỏ và đơn giản, thủ tục `ShowObj` được thực hiện bằng cách vẽ đối tượng và thủ tục `HideObj` được thực hiện bằng cách xóa đối tượng. Việc xóa một đối tượng có thể thực hiện bằng việc vẽ lại đối tượng đó với mầu nền, hoặc tô một bar bao quanh nó với mầu nền.

Chương trình `DemoBall` dưới đây minh hoạ sự chuyển động của một hình ảnh. Một quả bóng mầu vàng (mầu số 14) được chuyển động bên trong một vùng chữ nhật có mầu xanh

da trời (mầu số 3). Khi đập vào biên của vùng, nó sẽ bật trở lại theo nguyên tắc phản xạ và phát ra một tiếng động. Trong quá trình chuyển động có thể cho thay đổi tốc độ của bóng bằng cách gõ phím + (tăng tốc) hoặc - (giảm tốc). Chương trình kết thúc khi gõ phím Esc. Chương trình xây dựng thủ tục `Ball(x, y, color)` tô trơn một hình tròn bán kính `r` = 5 (hằng số) có tâm `x`, `y` với mầu `color`. Thủ tục này vừa dùng để vẽ bóng (với `color` = 14) vừa dùng để xóa bóng (với `color` = 3).

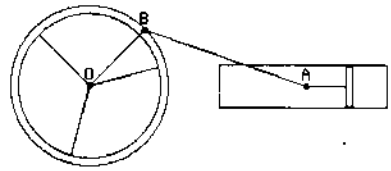
```
Program DemoBall;
Uses GRAPH, CRT;
Const
  r = 5;   {bán kính bóng}
Var
  c1, c2, d1, d2, t, x, y, xx, yy, dx, dy: integer;
  ch: char;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure Ball(x, y: integer; color: word);
  {vẽ bóng tại vị trí x, y, bán kính r với mầu color}
  begin
    SetFillStyle(1, color);
    SetColor(color);
    FillEllipse(x, y, r, r);
  end;

Begin
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! Nhan phim Enter de ket thuc...');
    Readln; Halt;
  end;
  c1 := 80; c2 := 560; d1 := 60; d2 := 400;    {kích thước}
  t := 10;                {độ trễ}
  dx := r; dy := r;       {bước dịch chuyển}
  x := c1+r+random(c2-c1-r-r);      {vị trí ban đầu}
  y := d1+r+random(d2-d1-r-r);
  SetBkColor(1);
  SetFillStyle(1, 3);
  Bar(c1, d1, c2, d2);
  OutTextXY(270, d1-20, 'DEMOBALL');
```

```
  OutTextXY(145, d2+29, '+: Faster      -: Slower      Esc: Quit');
  Randomize;
  repeat
    ch := #0;
    Ball(x, y, 14);          {vẽ bóng mầu vàng tại vị trí hiện tại}
    if (x-r+dx <= c1) or (x+r+dx >= c2) then   {xử lý chạm biên dọc}
    begin
      dx := -dx; Sound(2000); Delay(t); NoSound;
    end;
    if (y-r+dy <= d1) or (y+r+dy >= d2) then   {xử lý chạm biên ngang}
    begin
      dy := -dy; Sound(2000); Delay(t); NoSound;
    end;
    xx := x; yy := y;          {ghi nhận vị trí cũ}
    x := x+dx; y := y+dy;      {xác định vị trí mới}
    Delay(t);
    Ball(xx, yy, 3);           {xóa bóng}
    if KeyPressed then ch := ReadKey;
    case ch of
      '+': if t-1 >= 1 then t := t-1;
      '-': if t+1 <= 50 then t := t+1;
    end;
  until ch = #27;
  CloseGraph;
End.
```

Nhiều hình ảnh chuyển động được mô phỏng bằng cách vẽ chúng tại một vị trí nhưng với nhiều trạng thái khác nhau. Trong trường hợp này, thay cho các tham số tọa độ trong các thủ tục `ShowObj`, `HideObj` là các tham số trạng thái. Giả sử `ShowObj(s)` hiện ảnh với trạng thái `s` và `HideObj(s)` ẩn ảnh với trạng thái `s`. Mô hình cho tình huống này là:

```
s := s0;        {trạng thái ban đầu}
repeat
  ShowObj(s);              {hiện ảnh với trạng thái mới}
  ss := s;                 {lưu trạng thái cũ}
  {tính giá trị mới cho s}
  .............
  Delay(t);                {làm chậm lại với t thích hợp}
  HideObj(ss);             {ẩn ảnh với trạng thái cũ}
until stop;
```

Chương trình dưới đây mô phỏng hoạt động của cơ cấu biến chuyển động thẳng thành chuyển động quay (chẳng hạn trong động cơ tàu hỏa). Chuyển động thẳng của pittông trong xylanh được truyền qua tay quay để biến thành chuyển động tròn của bánh xe.



Pittông hoạt động liên tục gây cho bánh xe quay. Mỗi khi pittông chạm vào đầu xylanh, một âm thanh được phát ra để mô phỏng tiếng va đập. Có thể điều khiển tốc độ quay bằng cách nhấn các phím '+', '-'. Để tạm dừng, nhấn phím 'P'. Chương trình kết thúc khi ấn phím Esc.

```
Program DemoPiston;
Uses GRAPH, CRT;
Const
  x = 180; y = 150;       {điểm đặt ViewPort}
  r = 60;                 {bán kính ngoài bánh xe}
  rr = (r div 10)*9;      {bán kính trong bánh xe}
  l = 130;                {độ dài thanh trượt}
  k = 30;                 {độ dài piston}
  h = 15;                 {chiều cao piston}
  w = 5;                  {độ dày piston}
  d = 25;                 {tốc độ quay}
  n = 24;                 {số điểm chia}
Var
  da, co, si: real;
  x1, y1, x2, y2, x3, y3, xp, yp, xq, yq, xr, yr,
  xa, ya, xb, yb, xo, yo, t, i: integer;
  ch: char;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure SetValue;
  var a1, a2, a3, dd: real;
      hb: word;
  begin
    SetTextStyle(SmallFont, HorizDir, 5);
    hb := TextHeight('B')+1;
    SetTextJustify(CenterText, BottomText);
    SetViewPort(x, y, x+r+r+l+k+w+1+hb, y+r+r+hb, true);
    xo := r+hb; yo := r+hb;
```

```
  ya := yo;
  t := d;
  dd := 2*Pi/3;
  a1 := Pi/2;
  a2 := a1+dd;
  a3 := a2+dd;
  x1 := round(xo+rr*cos(a1));
  y1 := round(yo-rr*sin(a1));
  x2 := round(xo+rr*cos(a2));
  y2 := round(yo-rr*sin(a2));
  x3 := round(xo+rr*cos(a3));
  y3 := round(yo-rr*sin(a3));
  da := 2*Pi/n;
end;

procedure DrawPoint(x, y: integer; ch: char);
begin
  FillEllipse(x, y, 2, 2);
  OutTextXY(x, y-5, ch);
end;

procedure Draw(s: real);
{vẽ hình với góc quay s}
begin
  Circle(xo, yo, r);      {vẽ vành ngoài bánh xe}
  Circle(xo, yo, rr);     {vẽ vành trong bánh xe}
  co := Cos(s); si := sin(s);
  xp := Round(xo+(x1-xo)*co+(y1-yo)*si);
  yp := Round(yo-(x1-xo)*si+(y1-yo)*co);
  xq := Round(xo+(x2-xo)*co+(y2-yo)*si);
  yq := Round(yo-(x2-xo)*si+(y2-yo)*co);
  xr := Round(xo+(x3-xo)*co+(y3-yo)*si);
  yr := Round(yo-(x3-xo)*si+(y3-yo)*co);
  Line(xo, yo, xp, yp);     {vẽ 3 thanh nối tâm bánh xe với vành bánh}
  Line(xo, yo, xq, yq);
  Line(xo, yo, xr, yr);
  xa := Round(xo+r*co+Sqrt(l*l-r*r*Sqr(si)));
  xb := Round(xo+r*co);
  yb := Round(yo-r*si);
  Line(xo, yo, xb, yb);     {vẽ thanh OB}
  Line(xb, yb, xa, ya);     {vẽ thanh BA}
  Line(xa, ya, xa+k, yo);   {vẽ thanh nối A với piston}
  DrawPoint(xo, yo, 'O');
  DrawPoint(xb, yb, 'B');
  DrawPoint(xa, ya, 'A');
  Rectangle(xa+k, ya-h, xa+k+w, ya+h);   {vẽ piston}
  Rectangle(xo+l-r+k-1, yo-h-1, xo+l+r+k+w+1, yo+h+1);   {vẽ xylanh}
```

```
  end;

Begin
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! An phim Enter de ket thuc...');
    Readln; Halt;
  end;
  SetValue;
  repeat
    for i := 0 to n-1 do
    begin
      ch := #0;
      Draw(i*da);
      if i mod (n div 2) = 0 then
      begin
        Sound(500); Delay(5); NoSound;
      end;
      Delay(t);
      if KeyPressed then ch := UpCase(ReadKey);
      if ch = #27 then Break;
      case ch of
        '+': if t > 1 then Dec(t);
        '-': if t < 1000 then Inc(t);
        'P': ch := ReadKey;
      end;
      ClearViewPort;
    end;
  until ch = #27;
  CloseGraph;
End.
```

Tham số trạng thái được chọn trong chương trình là góc quay `s` của bánh xe. Thủ tục `Draw(s)` được xây dựng để vẽ hình tương ứng với góc quay `s`. Chương trình lặp lại việc vẽ 24 trạng thái tương ứng với 24 giá trị của `s` (vòng tròn bánh xe được chia đều thành 24 phần), trong đó thủ tục `ClearViewPort` được dùng để xóa hình cũ trước khi vẽ hình mới. Các giá trị khác trong chương trình, bạn đọc tự tìm hiểu.

Hai mô hình vừa được trình bày có thể kết hợp lại để tạo những chuyển động phức tạp, trong đó có cả sự thay đổi về tọa độ cũng như trạng thái của hình.

Các chương trình `DemoBall` và `DemoPiston` vừa được thiết kế có những hạn chế sau:

- Mỗi lần hiện ảnh phải vẽ lại ảnh, điều này làm chậm tốc độ, gây ra hiệu ứng nháy ảnh. Kỹ thuật này chỉ được dùng khi ảnh là đơn giản.
- Việc xóa ảnh bằng cách tô lại mầu nền, sẽ xóa luôn cả ảnh phía sau nó. Kỹ thuật này chỉ được dùng khi nền màn hình là đồng nhất.

Để khắc phục, người ta thường dùng các kỹ thuật xử lý bitmap, được trình bày trong mục sau.

12. Xử lý bitmap

Một ảnh được lưu trữ trong bộ nhớ bằng một tập hợp bit (chẳng hạn với ảnh 16 mầu, mỗi pixel tương ứng với 4 bit), người ta gọi chúng là bitmap. Việc xử lý các bitmap được tiến hành trên từng bit.

Unit GRAPH cung cấp một số thủ tục, hàm, cho phép lưu trữ một ảnh trên màn hình dưới dạng bitmap và đưa các bitmap ra màn hình dưới dạng ảnh tại những vị trí khác nhau. Việc đưa một bitmap ra màn hình có nhiều phương thức tương ứng với các toán tử bit tổ hợp mầu nền với mầu ảnh. Việc dùng các phương thức khác nhau dẫn đến những thay đổi khác nhau của ảnh trên màn hình. GRAPH xây dựng một số giá trị dưới đây (kèm với tên tương ứng) nhằm phục vụ cho việc đưa bitmap ra màn hình:

giá trị	tên	phép toán bit
0 (ngầm định)	CopyPut (hoặc NormalPut)	MOV (mầu vẽ ghi đè lên mầu nền)
1	XORPut	XOR
2	ORPut	OR
3	ANDPut	AND
4	NOTPut	NOT

Tùy theo phép toán bit mà mầu ảnh được tổ hợp với mầu nền. Chẳng hạn với CopyPut (ngầm định), các bit của ảnh sẽ thay thế các bit của nền, gây ra hiệu ứng ảnh được thể hiện trên màn hình giống như giá trị đã cất, còn với XORPut, các bit của ảnh sẽ được tổ hợp với các bit tương ứng của nền theo phép toán XOR, điều này dẫn đến mầu của ảnh bị thay đổi.

Để làm thí dụ, bạn có thể thực hiện một số các phép toán bit trên các số nguyên, từ đó suy ra các giá trị mầu tương ứng như dưới đây:

15 XOR 1 = 14 → White XOR Blue = Yellow,

4 OR 2 = 6 → Red OR Green = Brown,

4 AND 2 = 0 → Red AND Green = Black,

. . . . . .

GRAPH cung cấp thủ tục:

```
procedure GetImage(x1, y1, x2, y2: integer; var BitMap);
```

cho phép cất một ảnh trên màn hình trong miền chữ nhật có góc trên trái là `x1`, `y1`, góc dưới phải là `x2`, `y2` vào một biến không định kiểu `BitMap`. Kích thước của `BitMap` phải đủ chứa thông tin của ảnh. GRAPH tính hộ giá trị này bằng hàm:

```
function ImageSize(x1, y1, x2, y2: integer): word;
```

trả về kích thước của ảnh trong miền chữ nhật `x1, y1, x2, y2`.

Thông thường, `BitMap` được cấp phát động nhờ một con trỏ không định kiểu với kích thước cấp phát được xác định bởi hàm `ImageSize`.

Thủ tục:

```
PutImage(x, y, var BitMap; BitBlt: word);
```

đưa bitmap ra màn hình. Biến không định kiểu `BitMap` là biến lưu trữ ảnh, `x` và `y` là tọa độ góc trên trái của ảnh khi đưa ra màn hình, `BitBlt` là phương thức (phép toán bit) được dùng (xem bảng trên).

Dùng các kỹ thuật xử lý bitmap, ta có thể nhanh chóng đưa một ảnh ra màn hình hoặc xóa nó mà không ảnh hưởng đến nền phía sau ảnh. Hai phương thức hay được dùng nhất là CopyPut (ngầm định) và XORPut. Đặc biệt, phép toán XOR có tính chất a XOR a = 0 với mọi giá trị nguyên a nên XORPut được dùng rất tiện lợi trong việc hiện / ẩn ảnh mà không ảnh hưởng đến nền. Giả sử ảnh được tạo và cất vào biến động `p^` (`p` là một con trỏ không định kiểu nào đấy) nhờ thủ tục `CreatImage(p)`. Mô hình xây dựng chuyển động ảnh mà không làm hỏng nền phía sau ảnh bằng XORPut như sau:

```
CreatImage(p);    {tạo và cất ảnh vào biến động p^}
x := x0; y := y0;
repeat
  PutImage(x, y, p^, XORPut);      {hiện ảnh}
  {tính dx, dy theo quỹ đạo}
  ..................
  xx := x; yy := y;          {lưu vị trí cũ}
  x := x+dx; y := y+dy;      {tính vị trí mới}
  Delay(t);                  {làm chậm lại với t thích hợp}
  PutImage(xx, yy, p^, XORPut);   {ẩn ảnh}
until stop;
```

Chương trình minh họa dưới đây mô tả một đĩa bay mầu trắng chuyển động ngẫu nhiên trong bầu trời sao nhấp nháy. Trong khi chuyển động, kích thước của đĩa bay được thay đổi để tạo cảm giác lúc xa lúc gần. Các ảnh của đĩa bay tương ứng với những kích thước khác nhau được tạo và cất vào biến mảng `p[i]^` nhờ thủ tục `Saucers`. Quỹ đạo đĩa bay được sinh ngẫu nhiên theo kiểu zic-zac nhưng luôn đảm bảo trong vùng màn hình. Chương trình kết thúc khi ấn phím Esc.

```
Program MoveSaucer;
Uses GRAPH, CRT;
Const
  rmin = 5;              {bán kính nhỏ nhất của đĩa bay}
  rmax = 40;             {bán kính lớn nhất của đĩa bay}
  NumP = rmax-rmin+1;    {số ảnh cần lưu trữ}
  nStar = 500;           {số ngôi sao}
  iter = 10;             {khoảng lặp để thay đổi ảnh}
Type
  StarType = record
```

```
                col, row, color: integer;
              end;
Var MaxColor, count, nBlink, i, ii, j: word;
    MaxX, MaxY, x, y, dx, dy, xx, yy, d: integer;
    p: array[1..NumP] of pointer;
    r: array[1..NumP] of word;
    BlinkStar: array[1..nStar] of StarType;
    ch: char;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure CreatSaucer(r: word; var p: pointer);
  {tạo ảnh đĩa bay bán kính r và cất vào p^}
  var x, y, x1, y1, x2, y2: integer;
      r1, r2, r3, r4, r5: word;
  begin
    {vẽ đĩa bay bán kính r, tại giữa màn hình}
    x := MaxX div 2; y := MaxY div 2;
    r1 := Round(r/2); r2 := Round(r/3); r3 := Round(r/5);
    r4 := 2*r2; r5 := Round(r/10);
    Ellipse(x, y, 0, 360, r, r1);
    Ellipse(x, y-r3, 194, 346, r, r2);
    Line(x+r2, y-r2, x+r4, y-r4);
    Circle(x+r4, y-r4, r5);
    Line(x-r2, y-r2, x-r4, y-r4);
    Circle(x-r4, y-r4, r5);
    FloodFill(x, y+r2-r3+1, MaxColor);
    {cất ảnh đĩa bay vào p^}
    x1 := x-r; y1 := y-r4-r5;
    x2 := x+r; y2 := y+r1;
    GetMem(p, ImageSize(x1, y1, x2, y2));
    GetImage(x1, y1, x2, y2, p^);
    PutImage(x1, y1, p^, XORPut);          {xóa ảnh vừa vẽ}
  end;

  procedure Saucers;
  {tạo các đĩa bay từ nhỏ đến lớn và cất vào mảng p[i]^}
  var i: integer;
  begin
    for i := 1 to NumP do
    begin
```

```
      r[i] := rmin+i-1;
      CreatSaucer(r[i], p[i]);
    end;
  end;

  procedure Starry;
  {vẽ bầu trời sao}
  var i, x, y: integer;
      c: word;
  begin
    nBlink := 0;     {số các ngôi sao nhấp nháy}
    for i := 1 to nStar do
    begin
      x := Random(MaxX); y := Random(MaxY);
      c := 1+Random(MaxColor);
      PutPixel(x, y, c);          {vẽ các sao}
      if Random < 0.05 then       {ghi nhận các ngôi sao nhấp nháy}
      begin
        Inc(nBlink);
        with BlinkStar[nBlink] do
        begin
          col := x; row := y; color := c;
        end;
      end;
    end;
  end;

Begin
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! An phim Enter de ket thuc...');
    Readln; Halt;
  end;
  Randomize;
  MaxX := GetMaxX; MaxY := GetMaxY; MaxColor := GetMaxColor;
  x := MaxX div 2;          {xuất phát từ giữa màn hình}
  y := MaxY div 2;
  Saucers;                  {tạo và cất các ảnh đĩa bay}
  Starry;                   {vẽ bầu trời sao}
  i := NumP;                {bắt đầu từ đĩa bay lớn nhất}
  d := -1;                  {giảm dần kích thước}
  count := 0;
  x := x-r[i];              {x, y là góc trên trái của ảnh}
  y := y-2*round((r[i]/3))-round(r[i]/10);
  j := 1;
  repeat
    ch := #0;
```

```
    Inc(count);
    PutImage(x, y, p[i]^, XORPut);              {đưa ảnh ra màn hình}
    dx := random(r[i])*(-1+random(3));          {độ lệch ngẫu nhiên}
    dy := random(r[i] div 2)*(-1+random(3));
    xx := x; yy := y;                           {vị trí cũ}
    x := x+dx; y := y+dy;                       {vị trí mới}
    if (x < 0) or (x+2*rmax > MaxX) then x := xx;
    if (y < 0) or (y+rmax > MaxY) then y := yy;
    ii := i;                {ghi nhận chỉ số của ảnh cũ}
    if count > Iter then    {thay đổi ảnh}
    begin
      count := 0;
      i := i+d;             {chỉ số ảnh mới}
      if (d > 0) and ((random < 0.05) or (i > NumP)) then
      begin                 {giảm kích thước}
        d := -d; if i > NumP then i := NumP;
      end;
      if (d < 0) and ((random < 0.05) or (i < 1)) then
      begin                 {tăng kích thước}
        d := -d; if i < 1 then i := 1;
      end;
    end;
    with BlinkStar[j] do    {nhấp nháy ngôi sao j}
    begin
      color := (color+(MaxColor+1) div 2) mod (MaxColor+1);
      SetFillStyle(SolidFill, color);
      SetColor(color);
      FillEllipse(col, row, 1, 1);
    end;
    Inc(j);
    if j > nBlink then j := 1;
    Delay(20);
    PutImage(xx, yy, p[ii]^, XORPut);           {xóa ảnh cũ}
    if KeyPressed then ch := ReadKey;
  until ch = #27;
  CloseGraph;
End.
```

Hạn chế của phương thức XORPut là ảnh đưa ra không trung thực với ảnh gốc vì đã bị XOR với nền màn hình. Tuy nhiên, ưu điểm của nó là tốc độ nhanh và cài đặt đơn giản.

Trong một số tình huống cần đảm bảo ảnh không bị đổi mầu khi đi qua những nền không đồng nhất, người ta kết hợp việc vẽ ảnh với việc cất nền ở vị trí định vẽ để sau đó có thể đưa nền này về vị trí cũ bằng CopyPut như mô hình dưới đây:

```
  x := x0; y := y0;     {vị trí ban đầu}
  repeat
    GetImage(x, y, x+width, y+height, Bitmap);    {cất nền trước khi vẽ}
```

```
    DrawFig(x, y, width, height);      {vẽ ảnh vào vị trí đã cất nền}
    xx := x; yy := y;      {lưu vị trí cũ}
    {tính dx, dy theo quỹ đạo}
    ..................
    x := x+dx; y := y+dy;      {tính vị trí mới}
    Delay(t);      {làm chậm lại với t thích hợp}
    PutImage(xx, yy, Bitmap, CopyPut);      {đưa nền về vị trí cũ}
  until stop;
```

Trong mô hình trên, `DrawFig(x, y, width, height)` là thủ tục vẽ ảnh, được giới hạn trong một miền chữ nhật có chiều ngang `width`, chiều dọc `height` và tọa độ góc trên trái `x, y`. Tham biến không định kiểu `Bitmap` là một biến đủ lớn để chứa hình chữ nhật này (có thể xác định kích thước bằng hàm `ImageSize(x, y, x+width, y+height)`). Nhược điểm của mô hình này là tốc độ chậm vì phải liên tục vẽ lại ảnh, nó chỉ thích hợp khi ảnh được vẽ là đơn giản.

Đối với các lệnh vẽ đường, GRAPH cũng xây dựng các phương thức bitmap để đưa các nét vẽ ra màn hình nhờ thủ tục:

```
procedure SetWriteMode(BitBlt: word);
```

Thủ tục này đưa nét vẽ ra màn hình theo cách xác định bởi `BitBlt` (xem bảng ở trên), và có tác dụng đến các lệnh vẽ đoạn thẳng sau đó như `Line`, `LineTo`, `LineRel`, `Rectangle`, `DrawPoly`. Nó được dùng để tạo sự chuyển động của các hình gồm những nét vẽ của các thủ tục đã nêu. Giá trị ngầm định của `SetWriteMode` là CopyPut.

Chương trình minh họa dưới đây tạo một nền màn hình gồm một loạt các `Bar` xếp cạnh nhau với những mầu xen kẽ (thủ tục `DrawBack`), sau đó cho chuyển động các `Line` và `Rectangle` trên nền này (thủ tục `MoveLines`) bằng việc dùng `SetWriteMode` với giá trị XORPut. Chú ý rằng, mầu của các đoạn thẳng bị thay đổi khi đi qua các `Bar` khác nhau do hiệu ứng của phép toán XOR trên các giá trị mầu.

```
Program DemoWriteMode;
Uses GRAPH, CRT;
Var
  MaxColor: word;
  MaxX, MaxY: integer;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure DrawBack;
  {vẽ nền màn hình}
  const
```

```
  xBars = 15;
  yBars = 10;
var
  i, j, x, y: integer;
  color, width, height: word;
begin
  width := MaxX div xBars; height := MaxY div yBars;
  x := 0; y := 0; color := 0;
  for j := 1 to yBars do
  begin
    for i := 1 to xBars do
    begin          {tô các Bar kề nhau với các mầu luân phiên}
      SetFillStyle(SolidFill, color);
      Bar(x, y, x+width, y+height);
      Inc(x, width+1); Inc(color);
      color := color mod (MaxColor+1);
    end;
    x := 0;
    Inc(y, height+1);
  end;
end;

procedure MoveLines;
{chuyển động các Linc và Rectangle}
var
  left, top, right, bottom, step: integer;
  ch: char;
begin
  left := 0; top := 0; right := MaxX; bottom := MaxY;
  step := bottom div 100;
  SetWriteMode(XORPut);
  ch := #0;
  repeat
    {vẽ các Linc và Rectangle}
    Line(left, top, right, bottom);
    Line(left, bottom, right, top);
    Rectangle(left, top, right, bottom);
    MoveTo((left+right) div 2, top);
    LineTo(right, (top+bottom) div 2);
    LineTo((left+right) div 2, bottom);
    LineTo(left, (top+bottom) div 2);
    LineTo((left+right) div 2, top);
    Delay(50);
    {xóa các Line và Rectangle vừa vẽ}
    Line(left, top, right, bottom);
    Line(left, bottom, right, top);
    Rectangle(left, top, right, bottom);
```

```
      LineTo(right, (top+bottom) div 2);
      LineTo((left+right) div 2, bottom);
      LineTo(left, (top+bottom) div 2);
      LineTo((left+right) div 2, top);
      if (left+step < right) and (top+step < bottom) then
      begin      {cập nhật lại kích thước}
        Inc(left, step); Inc(top, step);
        Dec(right, step); Dec(bottom, step);
      end
      else
      begin      {trở về kích thước ban đầu}
        left := 0; top := 0; right := MaxX; bottom := MaxY;
      end;
      if KeyPressed then ch := ReadKey;
    until ch = #27;;
    SetWriteMode(CopyPut);
  end;

Begin
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! An phim Enter de ket thuc...');
    Readln; Halt;
  end;
  MaxX := GetMaxX; MaxY := GetMaxY; MaxColor := GetMaxColor;
  DrawBack;
  MoveLines;
  CloseGraph;
End.
```

7

MỘT SỐ THÍ DỤ

Trong chương này, chúng tôi đưa ra một số thí dụ nhằm cụ thể hóa những vấn đề đã được đề cập khi thảo luận về kỹ thuật lập trình.

Nói chung, để giải một bài toán lập trình, cần qua những bước:

- *Xác định khuôn dạng (quy cách) vào/ra*: Một số bài toán đã quy định rõ yêu cầu vào/ra (chẳng hạn các đề thi). Tuy nhiên, một số bài toán không nêu cụ thể (chẳng hạn các mô hình). Với dạng thứ nhất, cần phải hiểu thật chính xác những yêu cầu này để có thể kiểm tra kết quả của chương trình bằng những chương trình test. Với dạng thứ hai, phải tự tổ chức khuôn dạng vào/ra (sao cho hợp lý và thuận tiện) trước khi bắt tay thiết kế chương trình. Nhiều khi việc tổ chức tốt khuôn dạng vào/ra làm giảm nhẹ việc xử lý của chương trình. Trong các thí dụ được trình bày dưới đây, đa phần khuôn dạng vào/ra đã được phát biểu cụ thể.
- *Xác định giải thuật và tổ chức dữ liệu*: Đây là bước chủ yếu của lập trình. Việc tìm một giải thuật đúng đắn và hiệu quả để giải bài toán không phải là một công việc đơn giản, nó đòi hỏi một sự phân tích kỹ càng bài toán đã cho. Bên cạnh một số giải thuật cơ bản mà người lập trình không thể thiếu như tích lũy, sắp xếp, tìm kiếm, ..., người lập trình cần phải trang bị một số mô hình thuộc những lĩnh vực khác nhau như đồ thị, tổ hợp, hình học, tối ưu, quy hoạch, ... nhằm hỗ trợ cho việc định hướng lời giải. Cấu trúc dữ liệu là phương thức để có thể biến giải thuật thành chương trình, vì thế nó cần được tổ chức gắn liền với giải thuật. Một sự lựa chọn cấu trúc dữ liệu phù hợp sẽ giảm nhẹ các thao tác tính toán cho giải thuật. Chương trình, theo định nghĩa của N.Wirth, là kết quả của việc kết hợp hữu cơ giữa cấu trúc dữ liệu và giải thuật. Ngoài sự phụ thuộc vào giải thuật, việc lựa chọn tổ chức dữ liệu còn phụ thuộc vào nguồn tài nguyên cung cấp cho bài toán như công cụ lập trình, công cụ quản lý hệ thống, kích thước bài toán, ... Hiện nay, các ngôn

ngữ lập trình đều cho phép tổ chức các cấu trúc dữ liệu rất phong phú nhằm phản ánh được sự đa dạng của dữ liệu. Bên cạnh các kiểu dữ liệu cơ bản như số nguyên, số thực, ký tự, lôgic, mảng, bản ghi, file, ..., người lập trình còn phải nắm được các cấu trúc trừu tượng như các danh sách liên kết, cây, đồ thị, ... cũng như các thao tác cơ bản trên chúng.

- *Xây dựng chương trình*: Sau khi đã xác định được giải thuật và tổ chức dữ liệu để thực hiện, có thể bắt tay vào việc xây dựng chương trình. Việc viết chương trình dựa trên nguyên lý Top-Down, đó là xây dựng từ tổng thể đến chi tiết. Để xây dựng các khối tương ứng, cần phải vận dụng các kỹ thuật tổ chức chương trình con, trong đó đặc biệt chú ý đến việc tổ chức các tham số. Việc tổ chức tốt các khối giúp cho chương trình sáng sủa, ý đồ của giải thuật được thể hiện rõ ràng, dễ tìm lỗi và đặc biệt dễ mở rộng và phát triển chương trình. Trong quá trình thiết kế các chương trình con, cần nhận diện những chương trình con có tính chất thường xuyên để lưu chúng lại thành những thư viện, nhằm phục vụ lâu dài.
- *Kiểm chứng chương trình*: Một chương trình đã chạy thông cần phải được chạy thử trên những thí dụ để kiểm chứng kết quả. Thường người ta chú ý đến hai khía cạnh cần kiểm chứng, đó là tính đúng đắn và tính hiệu quả. Để kiểm tra tính đúng đắn, cần phải soạn những bộ test mà ta đã biết trước kết quả đúng hoặc phải viết một chương trình kiểm tra các tính chất cần thỏa mãn của kết quả. Những test loại này cần đa dạng bao gồm cả những tình huống thông thường và những tình huống đặc biệt. Những test này, trong trường hợp chương trình chạy không đúng, sẽ như những phản ví dụ giúp cho người lập trình phát hiện những lỗi lôgic của chương trình (đặc biệt những lỗi thuộc về thuật giải). Để kiểm tra tính hiệu quả, người ta thường dùng những bộ test có kích thước lớn để đo thời gian chạy cũng như không gian mà chương trình cần sử dụng. Những bộ test này thường được sinh ngẫu nhiên từ một chương trình, trong đó đầu vào là những giá trị ràng buộc của dữ liệu.

Vì số thí dụ là có hạn, nên chúng tôi cố gắng đưa ra những mô hình có thể áp dụng cho một lớp các bài toán để các bạn có điều kiện suy ngẫm và phát triển. Mặc dù các thí dụ đưa ra đều được giải quyết trên Turbo Pascal, nhưng các bạn đều có thể dễ dàng chuyển đổi chúng sang các công cụ lập trình khác. Ngoài ra cũng cần chú ý rằng, những lời giải cho các thí dụ này đều mang dấu ấn cá nhân, vì thế chúng chỉ có tính chất tham khảo. Các bạn có thể thay đổi hoặc phát triển chúng theo suy nghĩ của mình. Cuối mỗi thí dụ, chúng tôi có nêu một số vấn đề dành cho các bạn luyện tập.

Một số chương trình được cài đặt trong môi trường đồ họa, trong đó dùng thư viện GRAPH của Turbo Pascal. Các giá trị hằng tọa độ cũng như mầu trong những chương trình này, được ngầm định cho chế độ VGAHi (640×480, 16 mầu), vì thế nếu cần, có thể thay đổi chúng cho phù hợp với chế độ đồ họa hiện hành.

1. Thống kê các từ khóa

Một câu hỏi thường đặt ra khi đọc một chương trình nguồn viết từ một ngôn ngữ nào đó là chương trình này dùng bao nhiêu từ khóa của ngôn ngữ được viết và số lần lặp của mỗi từ khóa là bao nhiêu? Đây là một trong số những bài toán thống kê các từ ngữ khi xử lý các văn bản.

Mục này nhằm xây dựng một chương trình, cho phép trả lời câu hỏi trên. Tuy nhiên, trước hết cần phải làm rõ quy cách các dữ liệu vào/ra của chương trình.

Đầu tiên, chương trình cần phải biết bộ từ khóa của ngôn ngữ. Điều này dễ dàng đáp ứng bằng việc giả thiết các từ khóa của ngôn ngữ đã được soạn sẵn trên một file văn bản (gọi là file từ khóa) theo quy cách mỗi dòng một từ và tất cả các chữ cái đều được viết hoa. Ngoài ra ta giả thiết rằng có không quá 100 từ khóa và mỗi từ khóa dài không quá 15 ký tự (các giả thiết này khi cần có thể nới rộng). Việc tiếp theo là cần một định nghĩa cho các ký tự phân cách từ. Thông thường, mỗi từ đều được viết trọn vẹn trên một dòng và phân biệt với nhau bằng một số ký tự phân cách. Các ký tự phân cách này sẽ được định nghĩa bằng một hằng tập hợp ký tự trong chương trình.

Với những giả thiết như vậy, chương trình sẽ đọc văn bản nguồn (file vào) và thống kê các từ khóa đọc được. Kết quả thống kê được ghi sang file văn bản khác (file ra) theo quy cách: mỗi dòng ghi một từ khóa, trong đó 16 vị trí đầu dành cho nội dung từ, tiếp theo 3 vị trí dành cho số lần xuất hiện từ khóa đó.

Các từ khóa lần lượt được lấy ra từ văn bản nguồn. Tuy nhiên kết quả thống kê không thể có ngay khi đọc xong một từ khóa mà cần phải tích lũy dần. Một cách thích hợp cho việc tích lũy kết quả thống kê này là tổ chức một danh sách dưới dạng mảng mà mỗi phần tử của nó là một bản ghi với hai trường: một trường lưu trữ nội dung từ khóa còn một trường lưu trữ số lần xuất hiện hiện thời của từ khóa đó. Danh sách được tạo như sau:

- Ban đầu danh sách được khởi tạo rỗng.
- Cứ mỗi từ khóa đọc được từ văn bản nguồn, ta kết nạp nó vào danh sách theo một trong hai cách:
  - Nếu từ khóa không thuộc danh sách thì nó được thêm vào cuối danh sách với số lần xuất hiện là 1.
  - Trái lại, tăng số lần xuất hiện của nó lên 1 đơn vị.

Khi đọc hết các từ khóa của văn bản cũng là lúc danh sách lưu trữ đầy đủ kết quả đã thống kê. Cuối cùng chỉ là thao tác ghi kết quả này lên file ra theo quy cách mà bài toán đã yêu cầu.

Vấn đề còn lại là thao tác lấy các từ khóa từ văn bản nguồn. Việc này được giải quyết theo từng dòng. Các từ lần lượt được tách ra khỏi dòng nhờ việc nhận biết các ký tự phân cách. Việc kiểm tra xem từ được tách có phải là từ khóa không được tiến hành theo mô hình tìm kiếm tuần tự trên tập hợp các từ khóa đã cho. Các từ khóa này sẽ được đọc trước từ file từ khóa vào một mảng gọi là bảng từ khóa.

Từ những phân tích trên, các bước của lời giải có thể viết lại như sau (trong cách viết có mượn các cấu trúc lập trình của Pascal, các thao tác được mô tả bằng lời trong cặp ngoặc nhọn):

```
{đọc các từ khóa trong file từ khóa vào bảng từ khóa}
{khởi tạo danh sách rỗng}
while {còn file văn bản nguồn} do
begin
  {đọc một dòng của file vào xâu ký tự s}
  While {còn tách được từ w ra khỏi s} do
  begin
    If {w là từ khóa} then
    begin
      If {w thuộc danh sách} then
        {tăng số lần xuất hiện của w trong danh sách lên 1 đơn vị}
      else
        {thêm w vào cuối danh sách với số lần xuất hiện là 1}
    end
  end
end
{ghi danh sách vào file kết quả}
```

Bây giờ chuyển sang bước xây dựng chương trình.

Về dữ liệu, bảng từ khóa được tổ chức là mảng `KeyWord` gồm các thành phần kiểu `string[15]`, có số phần tử là `nKeyWord`. Danh sách được tổ chức là mảng `List`, gồm các thành phần kiểu bản ghi:

```
Type
  RecType = record
              term: string[15];
              freq: integer;
            end;
```

trong đó `term` chứa nội dung và `freq` chứa số lần xuất hiện của từ khóa. Số phần tử của danh sách được lưu trong `nList`.

Từ các bước của lời giải, chương trình có thể chia làm 3 khối lớn:

Khối thứ nhất là đọc các từ khóa vào bảng từ. Khối này được thực hiện nhờ thủ tục `ReadKeyWord(name: string)` có nhiệm vụ đọc các từ khóa trong file có tên cho bởi tham trị `name` vào mảng `KeyWord` và lưu trữ số từ khóa vào biến `nKeyWord`.

Khối thứ hai là xây dựng danh sách thống kê các từ khóa trong văn bản nguồn. Khối này được thực hiện nhờ thủ tục `CreatList(name: string)`. Thủ tục `CreatList` có nhiệm vụ đọc văn bản nguồn có tên cho bởi tham trị `name` rồi lần lượt kết nạp các từ khóa đọc được vào danh sách `List`, với số phần tử được lưu trữ trong biến `nList`, theo quy cách đã trình bày.

Khối thứ ba là ghi danh sách thống kê vừa tìm lên file kết quả. Khối này được thực hiện nhờ thủ tục `WriteResult(name: string)`, trong đó `name` là tham trị xác định tên file kết quả.

Với các thủ tục được xây dựng, chương trình chính bao gồm 3 lời gọi tương ứng đến 3 thủ tục này:

```
ReadKeyWord(KeyWordName);
CreatList(InputName);
WriteResult(OutputName);
```

trong đó `KeyWordName` là tên file từ khóa, còn `InputName` và `OutputName` là các tên file vào và ra tương ứng.

Các thủ tục `ReadKeyWord` và `WriteResult` là đơn giản, chúng bao gồm các lệnh đọc và ghi file. Riêng thủ tục `CreatList` bao gồm nhiều thao tác, cần được xây dựng thêm chi tiết. Các thao tác trong `CreatList` là:

- Khởi tạo danh sách rỗng.
- Kiểm tra file nguồn còn hay hết.
- Đọc từng dòng của file nguồn.
- Tách lần lượt các từ của dòng được đọc.
- Kiểm tra một từ có phải là từ khóa.
- Tìm vị trí của một từ khóa trong danh sách để cập nhật lại danh sách.

Việc khởi tạo danh sách rỗng được thực hiện bởi lệnh gán `nList` bằng 0. Việc kiểm tra còn file `f` được thực hiện bởi hàm chuẩn `Eof(f)`. Việc đọc từng dòng của file `f` vào xâu `s` được thực hiện bởi thủ tục chuẩn `Readln(f, s)`. Để tách lần lượt các từ của xâu `s` cho đến hết, ta dựng hàm `GetWord(s, i)` có nhiệm vụ trả về từ được tách tại vị trí `i` của `s`, sau khi tách xong hàm trả lại vị trí kế tiếp cho biến `i` để chuẩn bị cho lần tách sau (vì thế `i` phải là tham biến). Hàm này cũng kiêm luôn nhiệm vụ kiểm tra xem việc tách từ đã hết chưa bằng cách trả về từ rỗng nếu không còn từ nào được tách. Các chữ cái trong từ được tách cũng được hàm đổi luôn thành chữ cái lớn để tiện so sánh với các từ khóa sau này. Việc kiểm tra từ `w` có phải là từ khóa hay không được thực hiện nhờ hàm lôgic `IsKeyWord(w)` và việc tìm vị trí từ khóa `w` trong danh sách được thực hiện bởi hàm `InList(w)`. Hàm này trả về giá trị nguyên ≤ `nList` xác định chỉ số của phần tử được tìm thấy trong danh sách, nếu không thấy, hàm trả về giá trị > `nList` . Việc kết nạp `w` vào danh sách được thực hiện nhờ các thao tác truy cập danh sách: nếu `j` là chỉ số tìm thấy `w` trong danh sách (`j` ≤ `nList`) thì tăng `List[j].freq` lên 1 đơn vị, trái lại (`j` > `nList`) thì tăng `nList` lên 1 đơn vị và ghi nhận giá trị `w` cho `List[nList].term` và giá trị 1 cho `List[nList].freq`. Từ đó nhận được nội dung của `CreatList` là:

```
nList := 0;
while not Eof(f) do        {f là file nguồn đã mở}
begin
  Readln(f, s);            {đọc một dòng của f vào s}
  i := 1;
```

```
    w := GetWord(s, i);      {w là từ đầu tiên của dòng s}
    while w <> '' do
    begin
      if IsKeyWord(w) then   {nếu w là từ khóa}
      begin
        j := InList(w);
        if j > nList then    {nếu w không có trong danh sách}
        begin
          Inc(nList);
          List[nList].term := w;
          List[nList].freq := 1;
        end else Inc(List[j].freq);   {nếu w đã có trong danh sách}
      end;
      w := GetWord(s, i);    {w là từ kế tiếp}
    end;     {hết các từ trong dòng s}
  end;     {hết file f}
```

Các hàm `IsKeyWord`, `InList` được viết nhờ kỹ thuật tìm kiếm tuần tự. Thủ tục `GetWord` được viết nhờ kỹ thuật tích lũy. Đây là những mô hình cơ bản mà nếu bạn nào chưa thạo thì cần phải xem lại.

Dưới đây là toàn văn chương trình thống kê các từ khóa. Các ký tự phân cách được định nghĩa trong hằng tập hợp ký tự `DelimitSet`, bao gồm các dấu trắng, dấu chấm, dấu hai chấm, dấu phẩy, dấu chấm phẩy, dấu bằng và các dấu ngoặc. Các ký tự này có thể điều chỉnh thêm hoặc bớt. Chương trình đọc bộ từ khóa cất trong file KEYWORD.TXT và hỏi tên các file vào/ra từ bàn phím.

```
Program Statistics;
Const
  max = 100;   {số từ khóa tối đa}
  DelimitSet: set of char = [' ', '.', ':', ',', ';', '=',
                             '(', ')', '[', ']', '{', '}'];
Type
  RecType = record
              term: string[15];     {nội dung từ khóa}
              freq: integer;        {số lần xuất hiện}
            end;
Var
  nKeyWord, nList: integer;
  KeyWord: array[1..max] of string[15];
  List: array[1..max] of RecType;
  InputName, OutputName: string;

  procedure ReadKeyWord(KeyWordName: string);
  {đọc các từ khóa cất trong file có tên KeyWordName vào bảng từ khóa KeyWord}
  var f: Text;
  begin
```

```
  Assign(f, KeyWordName);
  Reset(f);
  nKeyWord := 0;
  while not Eof(f) do
  begin
    Inc(nKeyWord);
    Readln(f, KeyWord[nKeyWord]);
  end;
  Close(f);
end;

function GetWord(s: string; var i: integer): string;
{trả về một từ được tách của xâu s tại vị trí i, sau đó trả về i vị trí kế tiếp}
var w: string;
    l: integer;
begin
  l := Length(s);
  w := '';
  {bỏ qua ký tự ngăn cách}
  while (i <= l) and (s[i] in DelimitSet) do Inc(i);
  {tích lũy w}
  while (i <= l) and not (s[i] in DelimitSet) do
  begin
    w := w+UpCase(s[i]);   {đổi ra chữ lớn}
    Inc(i);
  end;
  GetWord := w;
end;

function IsKeyWord(w: string): boolean;
{kiểm tra w có phải là từ khóa không}
var i: integer;
begin
  i := 1;
  while (i <= nKeyWord) and (w <> KeyWord[i]) do Inc(i);
  IsKeyWord := i <= nKeyWord;
end;

function InList(w: string): integer;
{tìm kiếm w trong List}
var i : integer;
begin
  i := 1;
  while (i <= nList) and (w <> List[i].term) do Inc(i);
  InList := i;
end;
```

```
procedure CreatList(name: string);
{tạo List chứa các từ khóa của file có tên name}
var f: text;
    s, w: string;
    i, j: integer;
begin
  Assign(f, name); Reset(f);
  nList := 0;
  while not Eof(f) do
  begin
    Readln(f, s);
    i := 1;
    w := GetWord(s, i);     {w là từ được tách}
    while w <> '' do
    begin
      if IsKeyWord(w) then     {w là từ khóa}
      begin
        j := InList(w);
        if j > nList then      {w không thuộc List}
        begin
          Inc(nList);          {nối w vào cuối List}
          List[nList].term := w;
          List[nList].freq := 1;
        end else               {w thuộc List}
          Inc(List[j].freq);   {tăng số lần xuất hiện lên 1}
      end;
      w := GetWord(s, i);    {tách từ kế tiếp}
    end;
  end;
  Close(f);
end;

procedure WriteResult(name: string);
{ghi List lên file văn bản có tên name theo quy cách yêu cầu}
var f: text;
    i: integer;
begin
  Assign(f, name); Rewrite(f);
  for i := 1 to nList do
  with List[i] do
    Writeln(f, term, ' ':16-Length(term), freq:3);
  Close(f);
end;

Begin
  Write('Input name: '); Readln(InputName);
  Write('Output name: '); Readln(OutputName);
```

```
  ReadKeyWord('KEYWORD.TXT');
  CreatList(InputName);
  WriteResult(OutputName);
End.
```

Bạn có thể thử chương trình trên bằng chính văn bản nguồn của nó với file KEYWORD.TXT là bảng từ khóa của Turbo Pascal. Dưới đây là kết quả nhận được:

```
PROGRAM          1
CONST            1
SET              1
OF               3
TYPE             1
RECORD           1
STRING          11
END             14
VAR              8
ARRAY            2
PROCEDURE        3
BEGIN           13
WHILE            7
NOT              3
DO               9
FUNCTION         3
AND              4
IN               2
IF               2
THEN             2
ELSE             1
FOR              1
TO               1
WITH             1
```

Vấn đề dành cho bạn. Hãy viết chương trình giải bài toán trên với giả thiết không biết trước giới hạn trên của số từ khóa và yêu cầu các từ khóa trong file kết quả phải được sắp theo thứ tự abc.

2. Phân lớp tương đương

Một số bài toán có mô hình: Cho trước những đối tượng trong một quần thể nào đó, cần lấy ra những đối tượng này và phân loại chúng. Việc phân loại dựa vào việc so sánh hai đối tượng: chúng được gọi là cùng loại nếu thỏa mãn những điều kiện nào đấy. Tùy thuộc từng bài toán mà định nghĩa phân loại có khác nhau, tuy nhiên việc phân loại cần phải đảm bảo tính chất tương đương để việc chia lớp theo loại là một phân hoạch trên các đối tượng (trong toán học, các lớp này được gọi là các lớp tương đương). Có hai thao tác cơ bản để giải bài toán:

- Lấy ra lần lượt các đối tượng trong quần thể.
- Đánh dấu lớp tương đương cho mỗi đối tượng được lấy ra.

Thao tác thứ nhất dựa vào các thông tin nhận biết đối tượng trong quần thể. Thông thường chúng được tiến hành tuần tự nhờ một số kỹ thuật tìm kiếm. Thao tác thứ hai dựa vào định nghĩa phân loại để xác định lớp tương đương của đối tượng. Các đối tượng đã đánh dấu cần được lưu lại để so sánh với các đối tượng tiếp theo. Thông thường, danh sách là công cụ để lưu những đối tượng này.

Dưới đây chúng tôi giới thiệu bài toán Starry Night, nguyên là một đề thi Olympic quốc tế về tin học, làm thí dụ cho mô hình vừa nêu.

Starry Night (IOI Setubal 1998)

Bản đồ bầu trời sao ban đêm là một miền chữ nhật được chia lưới toạ độ, trong đó tại mỗi nút của lưới, có thể có hoặc không, một ngôi sao. Các ngôi sao họp thành từng chùm. Mỗi chùm gồm những ngôi sao kề nhau theo chiều ngang, chiều dọc và theo đường chéo. Một chùm không thể là một phần của một chùm lớn hơn.

Hai chùm sao được gọi là giống nhau nếu chúng có cùng hình dạng và cùng số lượng các ngôi sao, không phụ thuộc vào hướng của chúng. Nói chung, có 8 hướng cho một chùm sao như trong hình vẽ dưới đây:

```
* * *         * * *               *         *
*     *           *       *       *       *
*                 *         * * *         *
                  *                       * * *

  * * *         * * *       *               *
*     *         *           *       *         *
      *         *           * * *             *
                  *                       * * *
```

Tám chùm sao giống nhau

Thông tin về bản đồ bầu trời được cho bởi mảng 2 chiều, với số cột W (chiều rộng bản đồ) và số dòng H (chiều cao bản đồ). Mỗi phần tử của mảng ghi nhận trạng thái của nút lưới tương ứng: có giá trị 1 nếu có sao, có giá trị 0 nếu trái lại.

Bạn cần phải phân lớp các chùm sao giống nhau trên bản đồ bầu trời đã cho bằng cách đánh dấu các ngôi sao (vị trí có chữ số 1) bằng các chữ cái thường (26 chữ cái tiếng Anh, từ a đến z). Các chùm sao giống nhau cần được đánh dấu bằng cùng một chữ cái, các chùm sao khác nhau cần được đánh dấu bằng các chữ cái khác nhau.

Dữ liệu vào cho trong file STARRY.IN, gồm 2 dòng đầu ghi tương ứng chiều rộng W và chiều cao H của bản đồ bầu trời. H dòng tiếp theo mô tả bản đồ bầu trời, mỗi dòng gồm W ký tự 0 hoặc 1 (viết liền nhau)

Thí dụ, file STARRY.IN dưới đây:

```
23
15
10001000000000010000000
01111100011111000101101
01000000010001000111111
00000000010101000101111
00000111010001000000000
00001001011111000000000
10000001000000000000000
00101000000111110010000
00001000000100010011111
00000001110101010100010
00000100110100010000000
00010001110111110000000
00100001110000000100000
00001000100001000100101
00000001110001000111000
```

mô tả bản đồ bầu trời có chiều rộng 23 và chiều cao 15, với 19 chùm sao trong hình vẽ sau:

```
*       *                       *
  * * * * *       * * * * *       *   * *   *
  *               *       *       * * * * * *
                  *   *   *       *   * * * *
          * * *   *       *
        *     *   * * * * *
*               *
    *   *               * * * * *     *
        *               *       *     * * * * *
                *  *  *  *   *   *   *   *       *
          *     *  *  *   *       *
      *         *  *  *   * * * * *
    *           *  *  *   .           *
        *         *         *       *     *   *
                *  *  *       *       * * *
```

Kết quả đưa ra file STARRY.OUT, gồm H dòng, chứa chính bản đồ bầu trời trong file STARRY.IN, ngoại trừ các chùm được đánh dấu như đã mô tả trong yêu cầu.

Dưới đây là một file STARRY.OUT có thể có của file STARRY.IN trong thí dụ trên:

```
a000a0000000000b0000000
0aaaaa000ccccc000d0dd0d
0a0000000c000c000dddddd
000000000c0b0c000d0dddd
00000eee0c000c000000000
0000e00e0ccccc000000000
b000000e000000000000000
00b0f000000ccccc00a0000
0000f000000c000c00aaaaa
0000000ddd0c0b0c0a000a0
00000b00dd0c000c0000000
000g000ddd0ccccc0000000
00g0000ddd0000000e00000
0000b000d0000f000e00e0b
0000000ddd000f000eee000
```

File này mô tả các chùm sao được đánh dấu tương ứng với hình vẽ sau:

```
a       a                       b
  a a a a a       c c c c c       d   d d   d
  a               c       c       d d d d d d
                  c   b   c       d   d d d d
          e e e   c       c
        e     e   c c c c c
b               e
    b   f               c c c c c     a
        f               c       c     a a a a a
                d d d   c   b   c   a       a
          b       d d   c       c
      g         d d d   c c c c c
    g           d d d               e
        b         d         f       e     e   b
                d d d       f       e e e
```

Giới hạn kích thước:

Chiều rộng và chiều cao của bản đồ không quá 100, số các chùm sao không quá 500, số các chùm sao không giống nhau không quá 26 (từ a đến z), số các sao trong mỗi chùm không quá 160.

Trong bài toán, quần thể là bản đồ bầu trời, được cho dưới dạng một bảng bit, trong đó bit 1 là vị trí có sao. Đối tượng ở đây là một chùm sao, gồm một tập hợp các bit 1, được xác định bằng cách tìm kiếm các bit này kề cận theo 8 hướng. Hai chùm sao là tương đương (giống nhau) nếu chúng nhận được từ nhau bởi các phép quay 90^0 và phép đối xứng trục. Việc đánh dấu các lớp tương đương được thể hiện bằng việc hiển thị lại bản đồ bầu trời, trong đó các bit 1 được thay bằng chữ cái tương ứng với lớp mà chùm sao chứa bit đó thuộc vào. Các giá trị giới hạn trong bài gợi ý có thể dùng mảng để lưu trữ dữ liệu.

Để chứa bản đồ bầu trời, một cách thích hợp là dùng một mảng 2 chỉ số có các phần tử ký tự `a[i,j]`, trong đó giá trị '1' mô tả bit 1 và giá trị '0' mô tả bit 0. Để dễ dàng thay thế các bit 1 trong bản đồ bầu trời bằng chữ cái tương ứng, ta ghi nhận tại những vị trí này số hiệu chùm sao mà nó thuộc vào. Điều này dẫn đến việc dùng một mảng 2 chỉ số kiểu nguyên `temp[i,j]` lưu lại giá trị số hiệu các chùm sao tại những vị trí tương ứng (tại những vị trí không có sao, `temp[i,j]` bằng 0). Mảng này được xác định dần trong quá trình tìm kiếm các chùm sao. Để xác định chùm sao thuộc lớp tương đương nào, ta dùng mảng `code[i]` ghi nhận số hiệu của lớp tương đương chứa chùm sao `i`, giá trị này được xác định sau khi so sánh chùm sao `i` với danh sách `class` lưu các lớp tương đương đã được tìm thấy trước đấy. Nhằm tiết kiệm không gian lưu trữ mỗi chùm sao và thuận tiện cho việc so sánh chúng, kiểu chùm sao được định nghĩa:

```
Type
  cluster = record
              num, nRow, nCol: byte;
              row, col: array[1..160] of byte;
            end;
```

trong đó `num` lưu trữ số ngôi sao trong chùm, `nRow`, `nCol` là số dòng và số cột của hình chữ nhật nhỏ nhất chứa các ngôi sao của chùm, `row[i]`, `col[i]` là tọa độ dòng, cột của ngôi sao thứ `i` trong chùm, được đánh lại tương đối theo hình chữ nhật (dòng từ 1 theo chiều trên xuống, cột từ 1 theo chiều trái sang), các ngôi sao trong chùm được đánh số từ 1 theo thứ tự từ trái sang phải và từ trên xuống dưới. Số các chùm sao (không quá 500) được lưu trong biến `nCluster`.

Danh sách `class` được tổ chức là một mảng, với phần tử thuộc kiểu `cluster`. Mỗi phần tử của `class` lưu chùm sao đại diện cho một lớp tương đương. Số phần tử của `class` (không quá 26) được lưu trong biến `nClass`.

Quá trình xác định các lớp tương đương được tiến hành trong quá trình tìm kiếm các chùm sao. Với chùm sao `i` tìm thấy, nó sẽ được so sánh với các phần tử trong danh sách `class`. Nếu nó giống (tương đương) với `class[j]` thì `code[i]` được ghi nhận bằng `j`, trái lại cần thêm nó vào cuối danh sách và ghi nhận `code[i]` bằng số phần tử của danh sách mới. Khi các chùm sao được tìm hết, kết quả đánh dấu các lớp tương đương sẽ nhận được từ các mảng `temp[i, j]` và `code[i]`. Việc thay các `code[i]` bởi các chữ cái tương ứng chỉ là việc chuyển các số 1, 2, 3, ... tương ứng thành các chữ cái 'a', 'b', 'c', ...

Chương trình có thể chia thành 3 khối, tương ứng với 3 thủ tục:

Thủ tục `ReadInp` đọc dữ liệu từ file STARRY.IN vào các biến `w` (số cột), `h` (số dòng) và `a[i, j]`.

Thủ tục `Find` thực hiện quá trình tìm kiếm các chùm sao và phân lớp chúng. Kết quả tìm kiếm được lưu trong các mảng `temp[i, j]` và `code[i]`.

Thủ tục `WriteResult` ghi kết quả lên file STARRY.OUT theo quy cách đã yêu cầu.

Với các thủ tục trên, nội dung chương trình chính sẽ là:

```
ReadInp; Find; WriteResult;
```

Các thủ tục `ReadInp`, `WriteResult` không có gì đáng bàn, chúng chỉ gồm những lệnh

đọc từ file và ghi lên file. Nội dung chủ yếu của chương trình nằm ở thủ tục `Find`. Thủ tục này cần hai thao tác cơ bản sau:

- Lần lượt lấy ra được các chùm sao từ bản đồ sao.
- Với mỗi chùm sao, cần biết được nó thuộc lớp tương đương nào?

Để thực hiện thao tác thứ nhất, ta xây dựng thủ tục

```
procedure Expand(u, v: byte; var x: cluster)
```

có nhiệm vụ tìm kiếm chùm sao `x` bắt đầu từ vị trí `i`, `j` và xác định số hiệu của chùm sao này. Thủ tục cũng ghi nhận vào mảng `temp` số hiệu này tại những vị trí tìm thấy. Thủ tục `Expand` dùng kỹ thuật tìm kiếm theo chiều rộng: Từ vị trí có bit 1, phát triển sang các bit 1 kề cạnh (chưa được xét) theo 8 hướng. Các hằng mảng `dr`, `dc` gồm 8 giá trị được định nghĩa trong chương trình là 8 giá trị độ lệch theo dòng, cột tương ứng với 8 hướng đã nêu.

Để thực hiện thao tác thứ hai, hàm

```
function InList(var y: cluster): byte;
```

được xây dựng nhằm trả về chỉ số của phần tử trong danh sách `class` tương đương với `y`. Nếu `y` không tương đương với phần tử nào trong `class` thì hàm trả về 0. Nhờ hàm này, việc ghi nhận lớp tương đương của chùm sao số hiệu `i` sẽ được tiến hành bằng việc gán `code[i]` một giá trị như đã trình bày.

Hàm `InList` được thực hiện nhờ thuật toán tìm kiếm tuần tự. Việc so sánh hai đối tượng dựa vào hàm lôgic `Equiv(var x, y: cluster)`. Hàm này trả về `True` nếu `x` và `y` là tương đương và trả về `False` nếu trái lại. Việc kiểm tra tính tương đương dựa vào định nghĩa cho trong bài toán: Hai chùm sao là tương đương (giống nhau) nếu chúng nhận được từ nhau bằng một số phép biến đổi lấy từ phép quay 90^0 và đối xứng qua trục. Vì thế các thủ tục `Rot(var x: cluster)`, quay `x` một góc 90^0, `Sym(var x: cluster)`, đối xứng `x` qua trục và hàm lôgic `Same(var x, y: cluster)`, kiểm tra tính trùng khít của `x` và `y`, được xây dựng như những thư viện để phục vụ hàm `Equiv`. Về chi tiết, các bạn có thể xem thêm các chú thích trong chương trình.

```
Program StarryNight;
Type
  cluster = record
              num, nRow, nCol: byte;
              row, col: array[1..160] of byte;
            end;
  direc = array[1..8] of integer;
Const
  InpName = 'STARRY.IN';
  OutName = 'STARRY.OUT';
  dr: direc = (0, -1, -1, -1,  0,  1, 1, 1);
  dc: direc = (1,  1,  0, -1, -1, -1, 0, 1);
  ch: array[1..26] of char = 'abcdefghijklmnopqrstuvwxyz';
Var
  a: array[0..101, 0..101] of char;
  temp: array[1..100, 1..100] of integer;
```

```
class: array[1..26] of cluster;
code: array[1..500] of byte;
w, h, nClass: byte;
nCluster: integer;
f: text;

procedure Sym(var x: cluster);
{lấy đối xứng x qua trục giữa, kết quả trả về x}
var i: byte;
begin
  with x do
    for i := 1 to num do col[i] := nCol+1-col[i];
end;

procedure Rot(var x: cluster);
{quay x một góc 90⁰, kết quả trả về x}
var i, t: byte;
begin
  with x do
  begin
    for i := 1 to num do
    begin
      t := row[i];
      row[i] := nCol+1-col[i];
      col[i] := t;
    end;
    t := nRow; nRow := nCol; nCol := t;
  end;
end;

function Same(var x, y: cluster): boolean;
{trả về True nếu x và y trùng khít}
var i, j: byte;
    b: array[1..100, 1..100] of boolean;
begin
  FillChar(b, SizeOf(b), false);
  with x do
    for i := 1 to num do b[row[i], col[i]] := true;
  with y do
    for i := 1 to num do
    if not b[row[i], col[i]] then
    begin
      Same := false; exit;
    end;
  Same := true;
end;
```

```
function Equiv(var x, y: cluster): boolean;
{trả về True nếu x tương đương y}
var i: byte;
begin
  for i := 1 to 4 do      {thử các phép quay}
  begin
    if Same(x, y) then
    begin
      Equiv := true; exit;
    end;
    Rot(y);
  end;
  Sym(y);                 {lấy đối xứng}
  for i := 1 to 4 do      {thử các phép quay}
  begin
    if Same(x, y) then
    begin
      Equiv := true; exit;
    end;
    Rot(y);
  end;
  Equiv := false;
end;

procedure ReadInp;
{đọc dữ liệu từ file input}
var i, j: byte;
    s: string;
begin
  FillChar(a, SizeOf(a), '0');
  Assign(f, InpName); Reset(f);
  Readln(f, w); Readln(f, h);
  for i := 1 to h do
  begin
    Readln(f, s);
    for j := 1 to w do a[i, j] := s[j];
  end;
  Close(f);
end;

procedure Expand(u, v: byte; var x: cluster);
{tìm kiếm theo chiều rộng bắt đầu từ u, v, kết quả trả về x}
var i, k, uu, vv, imin, jmin, imax, jmax: byte;
begin
  Inc(nCluster);      {ghi nhận số chùm sao}
  with x do           {ghi nhận kết quả vào x}
  begin
```

```
    num := 0; i := 1;    {bắt đầu tìm kiếm}
    Inc(num); temp[u, v] := ncluster;
    row[num] := u; col[num] := v;
    imin := u; imax := u;
    jmin := v; jmax := v;
    while num >= i do    {vòng lặp tìm kiếm theo chiều rộng}
    begin
      u := row[i]; v := col[i]; Inc(i);
      for k := 1 to 8 do
      begin
        uu := u+dr[k]; vv := v+dc[k];
        if (a[uu, vv] = '1') and (temp[uu, vv] = 0) then
        begin
          Inc(num); temp[uu, vv] := nCluster;
          row[num] := uu; col[num] := vv;
          if imin > uu then imin := uu;
          if imax < uu then imax := uu;
          if jmin > vv then jmin := vv;
          if jmax < vv then jmax := vv;
        end;
      end;
    end;
    nRow := imax-imin+1; nCol := jmax-jmin+1;
    for k := 1 to num do    {chuyển các tọa độ sang tương đối}
    begin
      row[k] := row[k]-imin+1;
      col[k] := col[k]-jmin+1;
    end;
  end;
end;

function InList(var y: cluster): byte;
{trả về vị trí trong danh sách mà y tương đương, nếu không thấy trả về 0}
var k: byte;
    x: cluster;
begin
  for k := 1 to nClass do
  begin
    x := class[k];
    if x.num = y.num then
    if (x.nRow = y.nRow) and (x.nCol = y.nCol) or
       (x.nCol = y.nRow) and (x.nRow = y.nCol) then
    if Equiv(x, y) then
    begin
      InList := k; exit;    {tìm thấy y trong danh sách tại vị trí k}
    end;
  end;
```

```
    InList := 0;   {không tìm thấy y}
  end;

  procedure Find;
  {xác định và phân loại các chùm sao}
  var i, j, k: byte;
      x: cluster;
  begin
    FillChar(temp, SizeOf(temp), 0);
    nClass := 0; nCluster := 0;
    for i := 1 to h do      {vòng lặp tìm kiếm các chùm sao}
      for j := 1 to w do
      if (a[i, j] = '1') and (temp[i, j] = 0) then
      begin
        Expand(i, j, x);    {tìm thấy chùm sao x}
        k := InList(x);     {so sánh x với danh sách}
        if k = 0 then       {không thấy phần tử tương đương với x}
        begin               {nối x vào cuối danh sách}
          Inc(nClass);
          class[nClass] := x;
          k := nClass;
        end;
        code[nCluster] := k;   {ghi nhận lớp tương đương cho x}
      end;
  end;

  procedure WriteResult;
  {ghi kết quả lên file output theo quy cách yêu cầu}
  var i, j: byte;
  begin
    Assign(f, OutName); Rewrite(f);
    for i := 1 to h do
    begin
      for j := 1 to w do
        if temp[i, j] = 0 then Write(f, '0')
        else Write(f, ch[code[temp[i, j]]]);
      Writeln(f);
    end;
    Close(f);
  end;

Begin
  ReadInp; Find; WriteResult;
End.
```

Bạn có thể thử lại chương trình trên với thí dụ đã cho trong đầu bài. Chú ý rằng, kết quả của bài toán có thể sai khác một hoán vị trên tập chữ cái đánh dấu. Trong chương trình, tập này cho bởi hằng mảng ký tự:

```
ch: array[1..26] of char = 'abcdefghijklmnopqrstuvwxyz';
```

trong đó các chữ cái được xếp theo thứ tự tự nhiên. Bạn có thể thay đổi thứ tự này và thử chạy lại chương trình.

Vấn đề dành cho bạn. Hãy viết lại chương trình trên sao cho mỗi lần chạy lại (trên cùng một bộ dữ liệu), chương trình lại cho những kết quả sai khác nhau một hoán vị trên tập chữ cái đánh dấu.

3. Tìm dãy biến đổi

Trong nhiều bài toán, cần tìm một dãy biến đổi để đưa một cấu hình từ trạng thái này sang trạng thái khác. Mặc dù dãy biến đổi được tạo từ một số hữu hạn các phép biến đổi, nhưng tổ hợp của chúng lại khá phức tạp. Nói chung, việc duyệt tất cả các dãy biến đổi để kiểm tra các trạng thái tương ứng của cấu hình là không khả thi, mặc dù ý tưởng này là tự nhiên và đơn giản. Tuy nhiên trong một số trường hợp đặc biệt, do tính đơn giản của các phép biến đổi mà số các dãy tạo nên từ chúng là hữu hạn và có thể thực hiện được ý tưởng này.

Một trong các tính chất quý báu cần khai thác của các phép biến đổi là tính giao hoán và tính lũy đẳng. Tính giao hoán cho phép dồn các phép biến đổi giống nhau trong dãy thành một cụm, và tính lũy đẳng làm cho mỗi phép biến đổi chỉ cần xuất hiện trong dãy một số hữu hạn lần. Gọi $p_1, p_2, ..., p_n$ là các phép biến đổi như thế, tương ứng với các số mũ lũy đẳng là $k_1, k_2, ..., k_n$ (nghĩa là thực hiện k_i lần liên tiếp p_i ta lại được phép đồng nhất). Khi đó mọi dãy biến đổi đều có thể biểu diễn dưới dạng một bộ có thứ tự gồm n thành phần nhận giá trị nguyên:

$(m_1, m_2, ..., m_n)$

với $0 \le m_i \le k_i-1$ (i =1, 2, ..., n). Bộ trên biểu diễn dãy biến đổi trong đó p_i được gọi m_i lần. Số dãy biến đổi vì thế không vượt quá giá trị của tích $k_1k_2...k_n$. Việc duyệt tất cả các bộ đã nêu là có thể thực hiện được với những giới hạn kích thước thích hợp.

Để làm thí dụ, dưới đây chúng tôi giới thiệu bài toán về những chiếc đồng hồ (Clocks), một đề thi Olympic quốc tế khác về Tin học.

Clocks (IOI Stockholm 1994)

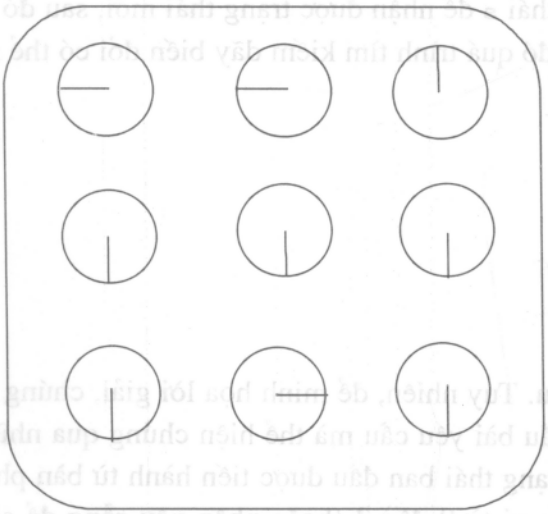
Trên hình 1 trình bày 3×3 chiếc đồng hồ có một kim, có thể vặn để được một trong bốn trạng thái: 12 giờ, 3 giờ, 6 giờ và 9 giờ. Có 9 phép biến đổi được đánh số từ 1 đến 9 tác động tương ứng lên các đồng hồ thuộc miền được tô đậm trên hình 2, trong đó kim của mỗi đồng hồ được vặn đi một góc 90 độ theo ngược chiều lượng giác. Vấn đề là cần tìm một dãy ngắn nhất từ 9 phép biến đổi đã cho để đưa cả 9 đồng hồ từ một trạng thái ban đầu cho trước về trạng thái tất cả đồng hồ đều chỉ 12 giờ.

Dữ liệu vào được đọc từ file văn bản có tên là INPUT.TXT gồm 9 số ghi nhận trạng thái kim của mỗi đồng hồ tương ứng: 0 = 12 giờ, 1 = 3 giờ, 2 = 6 giờ, 3 = 9 giờ. Chẳng hạn hình 1 là file có dạng:

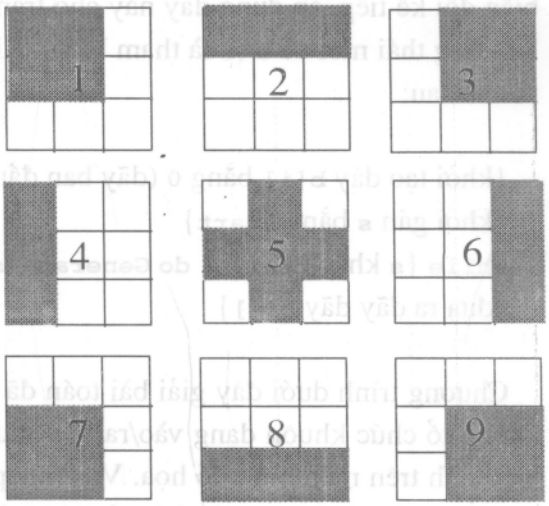
```
3 3 0
2 2 2
2 1 2
```

Dữ liệu ra cần ghi lên file văn bản tên là OUTPUT.TXT dãy ngắn nhất các phép biến đổi để đưa các đồng hồ từ trạng thái ban đầu về trạng thái kết thúc. Chẳng hạn với file trên, kết quả sẽ là:

```
5 8 4 9
```



Hình 1



Hình 2

Trước hết nhận xét rằng các biến đổi đã cho giao hoán được với nhau và có số mũ lũy đẳng là 4. Từ đấy, theo như cách đã trình bày, có thể biểu diễn một dãy biến đổi như là một chỉnh hợp lặp chập 9 của 4 giá trị 0, 1, 2, 3:

`b[1], b[2], ..., b[9]`

trong đó `b[i]` (lấy giá trị trong tập 0, 1, 2, 3) là số lần xuất hiện của phép biến đổi `i` trong dãy. Mặt khác có thể chứng minh được trong bài toán này, mọi dãy biến đổi đều được biểu diễn duy nhất qua dạng trên và vét cạn các trạng thái của các đồng hồ. Số lượng của chúng bằng $4^9 = 262144$, là hoàn toàn khả thi cho việc duyệt toàn bộ với những máy tính thông thường hiện thời. Như thế, với bất kỳ trạng thái ban đầu nào cho trước của các đồng hồ, ta đều tìm được dãy biến đổi để đưa các đồng hồ về trạng thái kết thúc. Chú ý rằng với cách biểu diễn đã nêu, số các phép biến đổi trong dãy tìm được cũng đồng thời thỏa mãn là ít nhất.

Đánh số các đồng hồ từ 1 đến 9 theo chiều từ trái sang phải và từ trên xuống dưới. Khi đó mỗi trạng thái sẽ được tổ chức như là một xâu ký tự `s` độ dài 9, trong đó `s[i]` ghi nhận trạng thái của đồng hồ `i`. Trạng thái ban đầu của các đồng hồ được lưu tại biến xâu `start`. Trạng thái kết thúc của các đồng hồ được biểu diễn bởi hằng xâu `finish` gồm 9 chữ số 0. Mỗi phép biến đổi cũng được biểu diễn bằng một xâu, trong đó mỗi ký tự ghi nhận số hiệu của đồng hồ bị sự tác động của phép biến đổi này. Các phép biến đổi được tổ chức dưới dạng hằng mảng xâu `act` gồm 9 phần tử, trong đó `act[j]` là phép biến đổi số hiệu `j`.

Việc duyệt các dãy `b[1], b[2], ..., b[9]` được thực hiện giống như một sự liệt kê các số gồm 9 chữ số trong hệ đếm 4 theo thứ tự tăng dần. Cứ mỗi dãy nhận được, dãy biến đổi tương ứng với nó lại được áp dụng cho các đồng hồ để nhận được một trạng thái mới. Quá trình duyệt hoàn thành khi nhận được trạng thái kết thúc của các đồng hồ. Khi đó dãy

biến đổi cuối cùng sẽ là kết quả của bài toán. Giả sử `Generate(s)` là thủ tục sinh dãy biến đổi kế tiếp, áp dụng dãy này cho trạng thái `s` để nhận được trạng thái mới, sau đó trả lại trạng thái mới về `s` (`s` là tham biến). Khi đó quá trình tìm kiếm dãy biến đổi có thể mô tả như sau:

{khởi tạo dãy `b[i]` bằng `0` (dãy ban đầu)}
{khởi gán `s` bằng `start`}
`while` {`s` khác `finish`} `do Generate(s);`
{đưa ra dãy dãy `b[i]`}

Chương trình dưới đây giải bài toán đã nêu. Tuy nhiên, để minh họa lời giải, chúng tôi không tổ chức khuôn dạng vào/ra file như đầu bài yêu cầu mà thể hiện chúng qua những hình ảnh trên màn hình đồ họa. Việc nhập trạng thái ban đầu được tiến hành từ bàn phím nhờ việc nhập một xâu 9 chữ số trong phạm vi từ 0 đến 3 (hoặc nhập xâu rỗng để sinh ngẫu nhiên). Kết quả được biểu diễn bằng hình ảnh chuyển động kim của các đồng hồ tương ứng với dãy các biến đổi tìm được. Chương trình được tổ chức lặp lại với những trạng thái ban đầu khác nhau và kết thúc khi ấn phím Esc. Chương trình chạy trong chế độ graph, có dùng các font chữ vectơ, vì thế phải đảm bảo có các file GRAPH.TPU, *.BGI và *.CHR. Nội dung của chương trình, ngoài nhiệm vụ chủ yếu là tìm dãy biến đổi, còn có một số thủ tục và hàm phục vụ cho các thao tác đồ họa. Không kể các thủ tục khởi động và đóng đồ họa, chương trình có những thủ tục cơ bản sau:

- Thủ tục `ReadInp`, nhập trạng thái ban đầu theo quy cách đã trình bày.
- Thủ tục `DrawBack`, vẽ nền màn hình, mô tả trạng thái ban đầu của các đồng hồ.
- Thủ tục `Find`, là thủ tục chủ yếu, thực hiện việc tìm kiếm dãy biến đổi theo mô hình đã nêu.
- Thủ tục `Display`, thực hiện việc chuyển động các kim đồng hồ theo dãy biến đổi đã tìm.

Như vậy, quá trình tìm dãy biến đổi để đưa các đồng hồ về trạng thái kết thúc cho mỗi lần đưa vào trạng thái ban đầu và minh họa dãy biến đổi tìm được qua chuyển động kim của các đồng hồ sẽ được thực hiện bởi các lời gọi:

```
ReadInp;
DrawBack;
Find;
Display;
```

Các thủ tục và hàm dưới đây được xây dựng trong chương trình để phục vụ các thủ tục đã nêu:

- Thủ tục `DrawClock(i, t)`, vẽ đồng hồ số hiệu `i` đang ở trạng thái `t`. Thủ tục này được gọi trong `DrawBack`.

- Hàm `Status(s, m)`, trả về trạng thái mới của các đồng hồ khi áp dụng phép biến đổi `m` cho trang thái hiện thời `s`. Hàm này được gọi trong thủ tục `Generate`.
- Thủ tục `Generate(s)` sinh dãy biến đổi kế tiếp, áp dụng dãy này cho trạng thái hiện thời `s` và trả về cho `s` trạng thái mới. Thủ tục này được gọi trong thủ tục `Find`.
- Thủ tục `Change(s, m)`, thực hiện việc chuyển động kim của các đồng hồ khi áp dụng phép biến đổi `m` vào trạng thái `s` hiện thời. Thủ tục này được gọi trong thủ tục `Display`.

Về chi tiết các bạn xem thêm các chú thích trong chương trình và có thể thay đổi lại theo ý muốn.

```
Program Clocks;
Uses CRT, GRAPH;
Type
  st5 = string[5];    {kiểu mô tả phép biến đổi}
  st9 = string[9];    {kiểu mô tả trạng thái các đồng hồ}
Const
  r = 50;             {bán kính đồng hồ}
  dist = 10;          {khoảng cách giữa các đồng hồ}
  x0 = 60;            {tọa độ tâm đồng hồ thứ nhất}
  y0 = 60;
  pi1 = pi;           {góc kim vị trí 9 giờ (trạng thái 3)}
  pi2 = pi1/2;        {góc kim vị trí 12 giờ (trạng thái 0)}
  pi3_2 = 3*pi2;      {góc kim vị trí 6 giờ (trạng thái 2)}
                      {0: góc kim vị trí 3 giờ (trạng thái 1)}
  pi30 = pi1/30;      {bước dịch chuyển của kim}
  {các góc kim ứng với các trạng thái}
  angl: array['0'..'3'] of real = (pi2, 0, pi3_2, pi1);
  {các phép biến đổi}
  act: array[1..9] of st5 = ('1245', '123', '2356', '147', '24568',
                             '369', '4578', '789', '5689');
  finish = '000000000';     {trạng thái kết thúc của các đồng hồ}
Var
  start: st9;               {trạng thái ban đầu của các đồng hồ}
  b: array[1..9] of byte;   {dãy biến đổi}
  ch: char;

  function DetectInit(path: string): integer;
  var drive, mode : integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;
```

```
procedure ReadInp;
{nhập trạng thái ban đầu của các đồng hồ}
var i: byte;
begin
  Write('Cho trang thai ban dau (neu rong la ngau nhien): ');
  Readln(start);
  if start = '' then
  begin
    Randomize;
    for i := 1 to 9 do
      start := start+Chr(Random(4)+48);
  end else
  for i := 1 to 9 do
  if not (start[i] in ['0'..'3']) then
  begin
    Writeln('Can vao mot chuoi 9 ky tu lay tu cac chu so 0..3');
    Write('An phim Enter de ket thuc...');
    Readln; CloseGraph; Halt;
  end;
  ClearDevice;
end;

procedure DrawClock(i: byte; t: char);
{vẽ đồng hồ số hiệu i, trạng thái t}
var cot, dong: byte;
    d, x, y, x1, y1: integer;
begin
  Dec(i);
  cot := i mod 3; dong := i div 3;
  d := 2*r+dist;
  x := x0+cot*d;
  y := y0+dong*d;
  Circle(x, y, r);
  case t of
    '0': begin  x1 := x; y1 := y-r; end;
    '1': begin  x1 := x+r; y1 := y; end;
    '2': begin  x1 := x; y1 := y+r; end;
    '3': begin  x1 := x-r; y1 := y; end;
  end;
  Line(x, y, x1, y1);
end;

procedure DrawBack;
{vẽ nền màn hình, biểu diễn trạng thái ban đầu của các đồng hồ}
var i, j, k: byte;
begin
```

```
  for i := 1 to 9 do DrawClock(i, start[i]);
  SetTextStyle(GothicFont, HorizDir, 6);
  OutTextXY(400, 30, 'CLOCKS');
  SetTextStyle(TriplexFont, HorizDir, 2);
  OutTextXY(405, 95, 'th');
  SetTextStyle(TriplexFont, HorizDir, 3);
  OutTextXY(390, 100, '6   IOI - Stockholm');
  OutTextXY(460, 130, 'Sweden');
  SetTextStyle(SmallFont, HorizDir, 6);
  OutTextXY(390, 220, 'Trang thai ban dau:');
  k := 0;
  for i := 1 to 3 do
    for j := 1 to 3 do
    begin
      Inc(k);
      OutTextXY(450+(j-1)*20, 250+(i-1)*20, start[k]);
    end;
end;

function Status(s: st9; m: byte): st9;
{trả lại trạng thái các đồng hồ khi áp dụng phép biến đổi m cho trạng thái s}
var i, j: byte;
    p: st5;
begin
  p := act[m];
  for i := 1 to Length(p) do
  begin
    j := Ord(p[i])-48;
    s[j] := Succ(s[j]);
    if s[j] = '4' then s[j] := '0';
  end;
  Status := s;
end;

procedure Generate(var s: st9);
{sinh dãy biến đổi kế tiếp để thay đổi trạng thái s}
var i: byte;
begin
  i := 9;
  while (i > 0) and (b[i] = 3) do
  begin
    s := Status(s, i);
    b[i] := 0;
    Dec(i);
  end;
  if i > 0 then
  begin
```

```
      s := Status(s, i);
      Inc(b[i]);
    end;
  end;

  procedure Find;
  {tìm dãy biến đổi}
  var i: byte;
      s: st9;
  begin
    for i := 1 to 9 do b[i] := 0;
    s := start;
    while s <> finish do Generate(s);
  end;

  procedure Change(var s: st9; m: byte);
  {vẽ chuyển động kim các đồng hồ khi áp dụng phép biến đổi m cho trạng thái s}
  var i, j, k, dong, cot: byte;
      t: char;
      p: st5;
      g0, g: array[1..5] of real;
      x, y, c, d, x1, y1: integer;
  begin
    SetFillStyle(SolidFill, 8);   {chọn mầu xám đậm cho các đồng hồ bị biến đổi}
    SetWriteMode(XORPut);
    p := act[m];
    for j := 1 to Length(p) do    {vẽ và tô các đồng hồ bị biến đổi}
    begin
      i := Ord(p[j])-48;          {i là số hiệu đồng hồ bị biến đổi}
      t := s[i];                  {t là trạng thái của đồng hồ i}
      g0[j] := angl[t];
      Dec(i);
      cot := i mod 3; dong := i div 3;
      d := 2*r+dist;
      x := x0+cot*d;
      y := y0+dong*d;
      FillEllipse(x, y, r, r);    {tô đồng hồ i}
      case t of
        '0': begin  x1 := x; y1 := y-r; end;
        '1': begin  x1 := x+r; y1 := y; end;
        '2': begin  x1 := x; y1 := y+r; end;
        '3': begin  x1 := x-r; y1 := y; end;
      end;
      Line(x, y, x1, y1);         {vẽ kim đồng hồ i}
    end;
    g := g0;
    for k := 1 to 15 do    {vẽ chuyển động kim của các đồng hồ}
```

```
begin
  for j := 1 to Length(p) do      {xóa kim của các đồng hồ ở vị trí cũ}
  begin
    i := Ord(p[j])-48;            {i là số hiệu đồng hồ}
    Dec(i);
    cot := i mod 3; dong := i div 3;
    d := 2*r+dist;
    x := x0+cot*d;
    y := y0+dong*d;
    c := Round(x+r*Cos(g[j]));
    d := Round(y-r*Sin(g[j]));
    Line(x, y, c, d);             {xóa kim đồng hồ i}
  end;
  for j := 1 to Length(p) do      {vẽ kim của các đồng hồ tại vị trí mới}
  begin
    i := Ord(p[j])-48;            {i là số hiệu đồng hồ}
    Dec(i);
    cot := i mod 3; dong := i div 3;
    d := 2*r+dist;
    x := x0+cot*d;
    y := y0+dong*d;
    g[j] := g[j]-pi30;
    c := Round(x+r*Cos(g[j]));
    d := Round(y-r*Sin(g[j]));
    Line(x, y, c, d);             {vẽ kim đồng hồ i}
  end;
  Delay(100);
end;                              {kết thúc chuyển động kim}
SetWriteMode(CopyPut);
s := Status(s, m);                {cập nhật trạng thái mới của các đồng hồ}
SetFillStyle(SolidFill, 0);       {chọn lại mầu nền cho các đồng hồ bị biến đổi}
for j := 1 to Length(p) do        {và vẽ lại các đồng hồ này}
begin
  i := Ord(p[j])-48;
  t := s[i];
  Dec(i);
  cot := i mod 3; dong := i div 3;
  d := 2*r+dist;
  x := x0+cot*d;
  y := y0+dong*d;
  FillEllipse(x, y, r, r);
  Circle(x, y, r);
  case t of
    '0': begin  x1 := x; y1 := y-r; end;
    '1': begin  x1 := x+r; y1 := y; end;
    '2': begin  x1 := x; y1 := y+r; end;
    '3': begin  x1 := x-r; y1 := y; end;
```

```
      end;
      Line(x, y, x1, y1);
    end;
  end;

  procedure Display;
  {vẽ chuyển động kim của các đồng hồ tương ứng với dãy biến đổi tìm được}
  var i, j, k: byte;
      st: string;
      s: st9;
  begin
    MoveTo(0, 400); OutText('Bat dau bien doi: ');
    s := start; k := 0;
    for i := 1 to 9 do
    begin
      Inc(k, b[i]);
      for j := 1 to b[i] do
      begin
        OutText(chr(i+48));
        Change(s, i);
        Delay(100);
      end;
    end;
    Str(k, st);
    OutText(' ('+st+' buoc)');
    OutTextXY(0, GetY+20, 'An mot phim de tiep tuc, Esc de ra...');
    ch := ReadKey;
    ClearDevice;
  end;

Begin
  if DetectInit('C:\TP\BGI') <> 0 then
  begin
    Write('Loi do hoa! An phim Enter de ket thuc...');
    Readln; Halt;
  end;
  DirectVideo := false;
  repeat
    ReadInp;
    DrawBack;
    Find;
    Display;
  until ch = #27;
  CloseGraph;
End.
```

Bạn có thể thử chương trình vừa viết với những trạng thái ban đầu khác nhau của các đồng hồ. Việc minh họa dãy biến đổi tìm được bằng các chuyển động kim của các đồng hồ là một sự kiểm nghiệm kết quả: Sau tác động của dãy biến đổi được tìm, tất cả các kim của các đồng hồ đều chỉ 12 giờ.

Vấn đề dành cho bạn. Hãy viết một chương trình giải bài toán tương tự dưới đây:

Có 9 căn phòng (đánh số từ 1 đến 9) đã được quét vôi với mầu trắng, xanh hoặc vàng. Có 9 rôbôt (đánh số từ 1 đến 9) phụ trách việc quét vôi các phòng. Mỗi rôbôt chỉ quét vôi một số phòng nhất định. Việc quét vôi được thực hiện nhờ một chương trình cài sẵn theo qui tắc:

- Nếu phòng đang có mầu trắng thì quét mầu xanh.

- Nếu phòng đang có mầu xanh thì quét mầu vàng.

- Nếu phòng đang có mầu vàng thì quét mầu trắng.

Cần phải gọi lần lượt một số các rôbôt ra quét vôi (mỗi lần một rôbôt, một rôbôt có thể gọi nhiều lần và có thể có rôbôt không được gọi. Rôbôt được gọi sẽ quét vôi tất cả các phòng mà nó phụ trách) để cuối cùng các phòng đều có mầu trắng. Hãy tìm một phương án như vậy sao cho lượng vôi phải quét là ít nhất. Giả thiết rằng lượng vôi cho mỗi lượt quét đối với các phòng là như nhau.

Dữ liệu: *Đọc từ file văn bản ROBOT.INP gồm các dòng:*

- 9 dòng đầu, mỗi dòng mô tả danh sách các phòng được quét vôi bởi một rôbôt theo thứ tự từ rôbôt 1 đến rôbôt 9. Mỗi dòng như vậy gồm các số hiệu phòng viết sát nhau. Chẳng hạn dòng thứ 2 có nội dung:

3578

mô tả rôbôt 2 phụ trách việc quét vôi các phòng 3, 5, 7, 8.

- Dòng cuối mô tả mầu vôi ban đầu của các phòng. Dòng gồm 9 ký tự viết sát nhau, ký tự thứ i biểu diễn mầu vôi của phòng i với quy ước: Ký tự T chỉ mầu trắng, ký tự X chỉ mầu xanh, ký tự V chỉ mầu vàng.

Kết quả: *Đưa ra file văn bản ROBOT.OUT gồm một dòng dưới dạng:*

- Nếu không có phương án thì ghi một chữ số 0.

- Trái lại ghi dãy thứ tự các rôbôt được gọi (các số hiệu rôbôt viết sát nhau).

Thí dụ:

ROBOT.INP
159
123
357
147
5
369
456
789
258
XVXVXVTXT

ROBOT.OUT
2455688

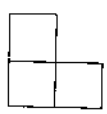
4. Kỹ thuật đệ quy

Khả năng đệ quy của ngôn ngữ là một điểm mạnh mà người lập trình cần khai thác. Mặc dù trong một số tình huống, việc cài đặt đệ quy tỏ ra kém hiệu quả trong việc sử dụng các tài nguyên như bộ nhớ, thời gian (chẳng hạn việc tính các số hạng của dãy Fibonaci mà cuốn sách này đã bàn đến trong chương **CHƯƠNG TRÌNH CON**, mục **6**), nhưng trong một số tình huống khác, việc cài đặt đệ quy lại tỏ ra rất hiệu quả, đặc biệt trong việc tiết kiệm công sức lập trình (chẳng hạn lời giải bài toán Tháp Hà nội trong cùng chương đã dẫn). Khéo dùng đệ quy, bạn có thể tận dụng được cách giải quy nạp trong toán học để cài đặt những chương trình bằng một lời giải đệ quy đơn giản, mà nếu không dùng nó, bạn sẽ tốn rất nhiều công sức. Hiện nay, có rất nhiều giải thuật được cài đặt bằng đệ quy như quick-sort, quay lui, tìm kiếm theo chiều sâu, ... Một số hình vẽ mà cuốn sách này đã giới thiệu trong phần unit GRAPH (chương **MỘT SỐ UNIT CHUẨN**), như cây Pythagor, các đường Hilbert, Sierpinski, ..., nếu không được thực hiện đệ quy thì chắc chắn sẽ rất phức tạp.

Để thiết kế một chương trình đệ quy, về mặt toán học, bạn phải có một lời giải quy nạp. Căn cứ vào lời giải này, chương trình phải xây dựng những chương trình con đệ quy tương ứng. Các chương trình con đệ quy có thể gọi chính chúng hoặc gọi lẫn nhau nhiều lần với những giá trị tham số thích hợp. Vì thế việc tổ chức các tham số cho các chương trình con này là một kỹ thuật quan trọng. Thông thường các tham số được tổ chức dựa vào các thông tin xác định đối tượng được đệ quy. Khéo chọn các thông tin này, lời giải đệ quy sẽ sáng sủa và hiệu quả. Cần chú ý các tham số truyền theo trị sẽ kéo theo việc cấp phát stack tỷ lệ với độ sâu của lời gọi, còn các tham số truyền theo biến sẽ gây ra các hiệu ứng lề (xem mục **4**, chương **CHƯƠNG TRÌNH CON**).

Dưới đây chúng tôi giới thiệu một đề thi quốc gia chọn học sinh giỏi phổ thông môn Tin học (bảng B - 1995) để làm thí dụ:

Cho bàn cờ vuông n×n ô, trong đó có một ô bị loại bỏ. Cần phủ kín bàn cờ bằng các thanh chữ L, mỗi thanh đặt lên bàn cờ sẽ phủ kín 3 ô vuông của bàn cờ (xem hình vẽ dưới đây):



Thanh chữ L

1	1	2	2
1	3	×	2
4	3	3	5
4	4	5	5

Bàn cờ 4 × 4 bị loại bỏ ô dòng 2 cột 3 (có dấu ×) được phủ bởi 5 thanh chữ L đánh số là 1, 2, 3, 4, 5

Giải bài toán trên với n có dạng 2^k, $1 \le k \le 7$.

Yêu cầu. *Dữ liệu đọc từ file văn bản có tên L.INP gồm 2 dòng: Dòng đầu ghi giá trị k, dòng sau ghi 2 giá trị u, v (cách nhau ít nhất một dấu trắng) là tọa độ dòng, cột của ô bị loại bỏ. Kết quả ghi ra file văn bản có tên L.OUT một cách phủ bàn cờ dưới dạng bảng số có cấu trúc: Mỗi dòng của file ghi chỉ số của các thanh chữ L phủ các ô trong dòng tương ứng của bàn cờ, trong đó ô bị loại được đánh dấu bởi số 0. Các số trong mỗi dòng được ghi cách nhau ít nhất một dấu trắng. Trong thí dụ trên, bảng kết quả có dạng:*

```
1 1 2 2
1 3 0 2
4 3 3 5
4 4 5 5
```

(so với nguyên văn của đề k ≤ 5, chúng tôi có tăng kích thước k ≤ 7 mà không ảnh hưởng đến lời giải và thay đổi chút ít quy cách vào ra để tiện cho việc test chương trình).

Đây là một bài toán có lời giải quy nạp rất đơn giản. Tuy nhiên, việc cài đặt nó bằng một thủ tục đệ quy đòi hỏi có một sự phân tích kỹ càng. Việc quy nạp được tiến hành theo kích thước k của bảng (để tiện, ta quy ước gọi bảng kích thước k là bảng $2^k \times 2^k$). Có thể bắt đầu từ k = 0 (bảng 1×1) với thao tác rỗng (không làm gì cả). Giả thiết việc phủ đã được giải quyết với các bảng kích thước k-1, khi đó việc phủ bảng kích thước k được tiến hành như sau:

- Chia bảng hiện hành thành 4 bảng con kích thước k-1 tương ứng nằm ở 4 góc.
- Xác định xem ô loại bỏ (u, v) thuộc góc nào thì thực hiện phủ bảng con ở góc đó.
- Trong 4 ô giữa bảng có 1 ô bị loại do đã được phủ, phủ nốt 3 ô còn lại bằng một thanh chữ L.
- Mỗi trong số các bảng con còn lại đều có 1 ô bị loại do đã được phủ. Tiến hành phủ nốt các bảng này.

Để chứa kết quả, một cách thích hợp là dùng mảng 2 chiều `a[i, j]` trong đó mỗi phần tử ghi nhận số tương ứng trong mỗi ô của bảng. Vì mỗi lần bảng chia 4, giá trị k giảm đi 1 đơn vị nên các số $2^0, 2^1, 2^2, ..., 2^k$ (k ≤ 7) sẽ luôn được gọi đến. Để đỡ phải tính lại, các số này được lưu trong một hằng mảng `two[i]`, 0 ≤ i ≤ 7:

```
two: array[0..7] of integer = (1, 2, 4, 8, 16, 32, 64, 128);
```

Các góc phần tư của bảng được đánh số từ 0 đến 3 theo hình vẽ sau:

0	3
1	2

Lấy vị trí 0 làm gốc, vị trí các góc được tính nhờ việc dịch chuyển 1 đơn vị theo dòng hoặc cột. Các giá trị này được lưu trong các hằng mảng `dx[i]` (độ dịch dòng), `dy[i]` (độ dịch cột), 0 ≤ i ≤ 3:

```
dx: array[0..3] of integer = (0, 1, 1, 0);
dy: array[0..3] of integer = (0, 0, 1, 1);
```

Để tổ chức lời giải đệ quy tương ứng với một bảng, cần phải biết các thông tin sau:

- Kích thước của bảng.

- Vị trí của bảng.
- Vị trí của ô bị loại.

từ đó gợi ý việc xây dựng thủ tục đệ quy:

```
procedure Over(size, x, y, u, v: integer);
```

thực hiện việc phủ bảng với các tham số: `size` - kích thước bảng, `x`, `y` - tọa độ dòng, cột góc trên trái của bảng và `u`, `v` - tọa độ dòng, cột ô bị loại thuộc bảng.

Nội dung của thủ tục `Over` được xây dựng theo các bước đã phát biểu trong lời giải quy nạp. Chúng chỉ được thực hiện khi `size` > 0:

- Xác định vị trí giữa bảng. Vị trí này được chọn là ô góc dưới phải của góc phần tư số hiệu 0. Tọa độ `xc`, `yc` của nó được tính theo công thức `xc = two[size-1]+x-1`, `yc = two[size-1]+y-1`.
- Xác định góc phần tư chứa ô bị loại `u`, `v` bằng việc so sánh `u`, `v` với `xc`, `yc`:

  `u` ≤ `xc`, `v` ≤ `yc` → `u`, `v` thuộc góc 0,

  `u` ≤ `xc`, `v` > `yc` → `u`, `v` thuộc góc 1,

  `u` > `xc`, `v` ≤ `yc` → `u`, `v` thuộc góc 2,

  `u` > `xc`, `v` > `yc` → `u`, `v` thuộc góc 3.
- Tọa độ 4 ô giữa bảng tương ứng với 4 góc phần tư `i` được tính từ `xc`, `yc` là `xc+dx[i]`, `yc+dy[i]`, 0 ≤ `i` ≤ 3.
- Tọa độ 4 ô góc trên trái tương ứng với 4 góc phần tư `i` được tính từ `x`, `y` là `x+dx[i]*two[size-1]`, `y+dy[i]*two[size-1]`, 0 ≤ `i` ≤ 3.
- Gọi `s` là số hiệu góc phần tư chứa ô `u`, `v`. Việc phủ góc phần tư này được thực hiện bằng lời gọi `Over(size-1, x+dx[s]*two[size-1], y+dy[s]*two[size-1], u, v)`.
- Việc phủ 3 ô giữa còn lại được thực hiện bằng các phép gán chỉ số `count` của thanh L hiện tại (chỉ số này được khởi động 0, trước mỗi lúc gán, nó được tăng lên 1) cho `a[xc+dx[t], yc+dy[t]]` với `t` lần lượt là các số hiệu của các góc phần tư khác `s`.
- Cuối cùng là việc phủ nốt 3 góc phần tư còn lại nhờ 3 lời gọi `Over(size-1, x+dx[t]*two[size-1], y+dy[t]*two[size-1], xc+dx[t], yc+dy[t])` với `t` lần lượt là các số hiệu của các góc phần tư khác `s`.

Chương trình chính gồm các lời gọi `Init` - đọc các giá trị `k`, `u`, `v` từ file và khởi gán các giá trị ban đầu, `Over(k, 1, 1, u, v)` - phủ bảng kích thước `k` với ô bị loại `u`, `v` và `WriteResult` - ghi bảng kết quả `a[i, j]` ra file. Dưới đây là toàn văn chương trình:

```
Program OverL;
Const
  two: array[0..7] of integer = (1, 2, 4, 8, 16, 32, 64, 128);
  dx: array[0..3] of integer = (0, 1, 1, 0);
  dy: array[0..3] of integer = (0, 0, 1, 1);
```

```
Var
  a: array[1..128, 1..128] of integer;
  k, u, v, n, count: integer;

  procedure Init;
  {đọc dữ liệu và khởi gán các giá trị}
  var i, j: integer;
      f: text;
  begin
    Assign(f, 'L.INP'); Reset(f);
    Readln(f, k); Read(f, u, v);
    Close(f);
    n := two[k];      {kích thước n = 2^k}
    count := 0;       {đếm thanh L}
  end;

  procedure Over(size, x, y, u, v: integer);
  {phủ bảng kích thước size, góc trên trái x, y, ô bị loại u, v}
  var xc, yc, s, i, t: integer;
  begin
    if size > 0 then
    begin
      {xác định ô giữa}
      xc := two[size-1]+x-1;
      yc := two[size-1]+y-1;
      {xác định góc s của u, v}
      if v <= yc then
        if u <= xc then s := 0 else s := 1
      else
        if u <= xc then s := 3 else s := 2;
      {phủ góc s}
      Over(size-1, x+dx[s]*two[size-1], y+dy[s]*two[size-1], u, v);
      {phủ 3 ô giữa}
      Inc(count);
      for i := 1 to 3 do
      begin
        t := (s+i) mod 4;
        a[xc+dx[t], yc+dy[t]] := count;
      end;
      {phủ 3 góc còn lại}
      for i := 1 to 3 do
      begin
        t := (s+i) mod 4;
        Over(size-1, x+dx[t]*two[size-1], y+dy[t]*two[size-1],
             xc+dx[t], yc+dy[t]);
      end;
    end;
```

```
  end;

  procedure WriteResult;
  {ghi kết quả ra file}
  var i, j: integer;
      f: text;
  begin
    Assign(f, 'L.OUT'); Rewrite(f);
    a[u, v] := 0;
    for i := 1 to n do
    begin
      for j := 1 to n do Write(f, a[i, j]:5);
      Writeln(f);
    end;
    Close(f);
  end;

Begin
  Init;
  Over(k, 1, 1, u, v);
  WriteResult;
End.
```

Bạn có thể thử lại chương trình với những kích thước k cho phép. Dưới đây là kết quả khi chạy chương trình với các giá trị k = 3, u = 5, v = 7:

```
 15   15   14   14   11   11   10   10
 15   13   13   14   11    8    8   10
 16   13   12   12    7    7    8    9
 16   16   12    6    6    7    9    9
 19   19   17    6    3    3    0    1
 19   18   17   17    3    2    1    1
 20   18   18   21    4    2    2    5
 20   20   21   21    4    4    5    5
```

Vấn đề dành cho bạn. Hãy viết một chương trình kiểm tra tính đúng đắn của các kết quả của bài toán đã nêu.

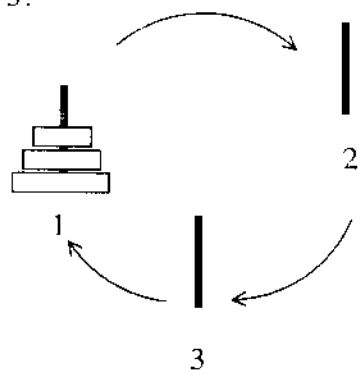
Bài toán Tháp Hà nội với các biến thể của nó thường là những bài tập để rèn luyện kỹ năng đệ quy. Dưới đây chúng tôi giới thiệu bài toán Chuyển đĩa, một biến thể của bài toán Tháp Hà nội, được chọn làm đề thi quốc gia chọn học sinh giỏi phổ thông môn Tin học, bảng B, năm 2000:

Có 3 cọc trên vòng tròn, được đánh số theo thứ tự chiều kim đồng hồ là 1, 2, 3. Có N đĩa kích thước từ lớn đến nhỏ, được xếp chồng đĩa nhỏ nằm trên đĩa lớn tại cọc 1.

Yêu cầu: *Cần chuyển N đĩa từ cọc 1 sang cọc 2 theo các quy tắc:*

- *Mỗi bước chỉ chuyển một đĩa sang cọc sát cạnh nó theo chiều kim đồng hồ (1 sang 2, 2 sang 3, 3 sang 1),*
- *Trong quá trình chuyển đĩa, chỉ cho phép đĩa nhỏ đặt lên trên đĩa lớn.*

Hình vẽ dưới mô tả với N = 3:



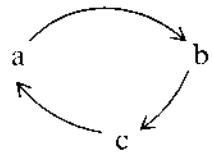
Dữ liệu: *Vào từ file văn bản DISK.INP chứa số N (N ≤ 15).*

Kết quả: *Ghi ra file văn bản DISK.OUT, trong đó mỗi dòng mô tả một bước chuyển đĩa gồm bắt đầu là số hiệu cọc xuất phát, tiếp theo là số hiệu cọc chuyển đến, ghi cách nhau ít nhất một dấu trắng.*

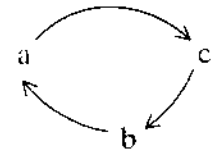
Thí dụ:

DISK.INP	DISK.OUT
3	1 2
	2 3
	1 2
	3 1
	2 3
	1 2
	2 3
	1 2
	3 1
	1 2
	3 1
	2 3
	1 2
	3 1
	1 2

Gọi a, b, c là các số hiệu cọc. Lời giải quy nạp chuyển N đĩa từ a sang b lấy c làm trung gian cần phải xét cả hai tình huống: b cạnh a theo chiều kim đồng hồ và ngược lại (xem hình vẽ dưới).



b cạnh a



b không cạnh a

Các thao tác chỉ được thực hiện khi N > 0.

Nếu b cạnh a theo chiều kim đồng hồ thì thực hiện:

- Chuyển N-1 đĩa từ a sang c, lấy b làm trung gian.
- Chuyển 1 đĩa từ a sang b.
- Chuyển N-1 đĩa từ c sang b, lấy a làm trung gian.

trái lại, thực hiện:

- Chuyển N-1 đĩa từ a sang b, lấy c làm trung gian.
- Chuyển 1 đĩa từ a sang c.
- Chuyển N-1 đĩa từ b sang a, lấy c làm trung gian.
- Chuyển 1 đĩa từ c sang b.
- Chuyển N-1 đĩa từ a sang b, lấy c làm trung gian.

Lời giải được thực hiện bởi thủ tục đệ quy:

```
procedure MoveD(N, a, b, c: integer);
```

mà các lệnh trong nó không khác gì lời giải quy nạp, trong đó thao tác chuyển 1 đĩa được thực hiện bằng một lệnh ghi một dòng lên file theo quy cách yêu cầu, còn thao tác chuyển N-1 đĩa được thực hiện bằng lời gọi chính nó với các giá trị tham số tương ứng. Sau khi đọc giá trị N, và mở file chuẩn bị ghi, chương trình bắt đầu thực hiện việc chuyển đĩa bằng lời gọi `MoveD(N, 1, 2, 3);`

Dưới đây là toàn văn chương trình giải bài toán Chuyển đĩa:

```
Program MoveDisk;
Var
  N: integer;      {số đĩa}
  f: text;

  procedure MoveD(N, a, b, c: integer);
  {chuyển N đĩa từ a sang b, lấy c làm trung gian}
  begin
    if N > 0 then
    begin
      if b = a mod 3+1 then      {b cạnh a}
      begin
        MoveD(N-1, a, c, b);
        Writeln(f, a, ' ', b);
```

```
        MoveD(N-1, c, b, a);
      end else                    {b không cạnh a}
      begin
        MoveD(N-1, a, b, c);
        Writeln(f, a, ' ', c);
        MoveD(N-1, b, a, c);
        Writeln(f, c, ' ', b);
        MoveD(N-1, a, b, c);
      end;
    end;
  end;

Begin
  Assign(f, 'DISK.INP'); Reset(f);
  Read(f, N);
  Close(f);
  Assign(f, 'DISK.OUT'); Rewrite(f);
  MoveD(N, 1, 2, 3);
  Close(f);
End.
```

Bạn có thể cài thêm một biến đếm vào trong chương trình trên để đếm số bước phải chuyển đĩa. Khi chạy thử chương trình, bạn sẽ thấy số này tăng rất nhanh theo N. Với N = 15, số bước phải chuyển đĩa bằng 2 781 183.

Vấn đề dành cho bạn.

1. Hãy viết một chương trình kiểm tra tính đúng đắn của các kết quả của bài toán đã nêu.

2. Hãy tìm công thức (dưới dạng truy hồi) đếm số bước phải chuyển đĩa trong chương trình `MoveDisk`. Từ đó xây dựng hàm

```
function F(N: integer): LongInt;
```

trả về số bước phải chuyển đĩa của chương trình `MoveDisk`. Sau đó cho chạy thử hàm này và so sánh kết quả của nó với kết quả đếm được trong `MoveDisk`.

Dưới đây là một số giá trị của `F(N)` tính trong phạm vi `LongInt` để các bạn tham khảo (chúng tôi viết thành từng nhóm 3 chữ số để các bạn dễ đọc):

N	15	16	17	18	19	20	21
F	2 781 183	7 598 335	20 759 039	56 714 751	154 947 583	423 324 671	1 156 544 511

5. Chơi cờ với máy tính

Việc thiết kế các chương trình trò chơi trên máy tính điện tử hiện nay đang là một lĩnh vực được quan tâm và phát triển. Chương trình chơi cờ với máy tính là một trong những chương trình trò chơi có tính trí tuệ được nhiều người ham thích. Nội dung thiết kế chương trình loại này có thể mô tả như sau:

- Trình bày thế khởi đầu của ván cờ.
- Xây dựng giao diện thân thiện để người chơi có thể chọn nước đi của mình. Cần kiểm soát tính đúng luật của nước đi.
- Xây dựng một chiến lược hiệu quả chọn nước đi cho máy.
- Cứ xong một lượt đi, cần kiểm tra tính kết thúc của ván cờ và kết quả cuộc chơi.

Về mặt dữ liệu, cần chuẩn bị các kiểu dữ liệu thích hợp để có thể lưu trữ trạng thái của ván cờ sau mỗi lượt đi.

Giả sử thủ tục `Person` thực hiện việc ghi nhận nước đi của người chơi và thủ tục `Computer` thực hiện một nước đi của máy. Khi đó mô hình của chương trình là:

```
{khởi động ván cờ}
while {ván cờ chưa kết thúc} do
  if {đến lượt người} then Person
  else Computer;
{thông báo kết quả}
```

Khó khăn khi thiết kế các chương trình loại này không nằm ở phần giao diện mà ở phần tìm kiếm một chiến lược hiệu quả. Các loại cờ thông dụng hiện nay như cờ tướng, cờ vua có độ phức tạp rất lớn, việc tìm kiếm một chiến lược hiệu quả không phải là đơn giản và còn tùy thuộc vào trình độ những người tham gia thiết kế. Tuy nhiên có một số loại cờ có tính xác định: Bắt đầu từ một thế cờ nào đó, luôn có một cách đi thắng cuộc, đồng thời có một chiến lược hiệu quả để đưa ván cờ về thế có lợi đó.

Dưới đây chúng tôi trình bày một chương trình chơi cờ, thuộc loại cờ xác định, gọi là cờ dồn. Nội dung của luật chơi là như sau:

Bàn cờ được chia ô, gồm n cột, m dòng. Mỗi bên có n quân, mỗi quân nằm vào một ô nào đấy sao cho mỗi cột của bàn cờ có đúng một quân của mỗi bên. Đến lượt đi, mỗi bên được chọn một quân của mình, tiến hay lùi một số ô tùy ý trong phạm vi cột đang đứng và không được vượt qua đầu quân đối thủ. Bên nào đến lượt mình mà không đi được thì thua.

Gọi S là tổng XOR của các số $d_1, d_2, ..., d_n$ trong đó d_i là số ô nằm giữa hai quân trên cột i. Nếu tổng này bằng 0 thì bất cứ một nước đi nào tiếp theo cũng dẫn đến tổng trở nên khác 0, trái lại nếu tổng này khác 0 thì luôn tìm được một nước đi quân cờ ở một cột k

nào đó, thu hẹp khoảng cách giữa hai quân và đẩy tổng S trở về 0. Từ đó chương trình xây dựng chiến lược chơi của máy như sau:

- Nếu đến lượt máy đi mà tổng S khác 0 thì chương trình tìm nước đi như đã trình bày để đưa tổng S trở về 0. Bằng cách này người chơi không thể đảo ngược thế cờ và máy chắc chắn thắng cuộc.
- Trong trường hợp trái lại, chương trình xây dựng một nước đi nào đó có tính chất chờ thời. Nếu người chơi sai lầm đi một nước làm cho tổng S khác 0 thì tình thế đảo ngược: Máy lại quay về trường hợp chắc chắn thắng.

Chương trình dùng khả năng sinh dãy ngẫu nhiên để xây dựng những nước đi khác nhau trong mỗi lần chơi lại. Điều này làm ván cờ đa dạng và không nhàm chán. Chương trình dùng giao diện màn hình đồ họa, chuột, bàn phím vì thế nó cần tham khảo đến các unit GRAPH, CRT, MOUSE, trong đó unit MOUSE đã được thiết kế trong chương **MỘT SỐ UNIT CHUẨN**, phần giới thiệu unit DOS.

Trong chương trình, các biến `n` và `m` được dùng để lưu trữ kích thước bàn cờ (`n` là số cột, `m` là số dòng). Độ rộng của mỗi ô và bán kính quân cờ (tính theo pixel) được cho bởi các hằng `d`, `r`. Do khuôn khổ màn hình nên chương trình giới hạn `n` ≤ 40 và `m` ≤ 29. Vì tọa độ cột của các quân cờ không thay đổi nên mỗi quân cờ được đánh số theo thứ tự cột mà nó đứng. Như vậy, vị trí của mỗi quân cờ sẽ được xác định bởi tọa độ dòng của nó. Các mảng nguyên `pers` và `comp` được dùng để lưu trữ các vị trí quân cờ, trong đó `pers[i]` là vị trí quân cờ `i` của người chơi và `comp[i]` là vị trí quân cờ `i` của máy. Chúng luôn được cập nhật sau mỗi lượt đi. Biến lôgic `PersTurn` dùng để ghi nhận lượt đi: Nếu nó là `true` thì đến lượt người, trái lại là lượt máy. Biến lôgic `stop` dùng để kiểm tra việc kết thúc ván cờ, nó nhận giá trị `true` nếu nếu ván cờ đã ngã ngũ hoặc chương trình bị ngắt giữa chừng do ấn phím Esc.

Vì chương trình được giả thiết chạy trong môi trường đồ họa và có chuột nên thủ tục `CheckDevice` được gọi ngay đầu chương trình để kiểm tra điều kiện làm việc. Thủ tục này cũng kiêm luôn việc ấn định kích thước bàn cờ trong cuộc chơi. `CheckDevice` sẽ kết thúc chương trình khi có một điều kiện không thỏa mãn.

Các thủ tục dùng để xây dựng một ván cờ là:

- Thủ tục `Init` khởi động ván cờ bằng cách sinh ngẫu nhiên vị trí các quân cờ. Nó có nhiệm vụ trình bày thế cờ đầu tiên lên màn hình cho người chơi quan sát và cho phép người chơi chọn lượt đi.
- Thủ tục `Person` thiết kế giao diện cho phép người chơi chọn nước đi bằng chuột khi đến lượt. Nó cũng có nhiệm vụ kiểm tra luật đi để người chơi không thể đi sai quy định.
- Thủ tục `Computer` xây dựng nước đi của máy theo chiến lược đã trình bày và thể hiện nước đi này trên bàn cờ để người chơi quan sát.
- Thủ tục `MessResult` thông báo kết quả khi ván cờ kết thúc.

Tại đầu các thủ tục `Person` và `Computer` có kiểm tra việc kết thúc: Nếu không còn nước đi thì biến `stop` được gán giá trị `true` và thoát khỏi các thủ tục này.

Như vậy đoạn chương trình mô tả một ván cờ là:

```
Init;
while not stop do
begin
  if PersTurn then Person
  else Computer;
end;
MessResult;
```

Để phục vụ các thủ tục vừa nêu, trong chương trình có xây dựng những thủ tục và hàm dưới đây:

- Thủ tục `Piece(i, j, color)` vẽ quân cờ dưới dạng một hình tròn nằm ở cột `i` và dòng `j` bàn cờ với mầu được tô là `color`. Thủ tục này được gọi trong `Init`. Các quân của người chơi được tô mầu đỏ còn các quân của máy được tô mầu trắng.
- Hàm lôgic `SelOk(x, y, u, v, k)` kiểm tra việc chọn quân của người chơi tại tọa độ `x`, `y` của chuột có đúng không. Nếu đúng hàm trả về `true` và trả về thông tin quân cờ được chọn cho các biến `u`, `v`, `k`, trong đó `u`, `v` là tọa độ tâm của quân cờ và `k` là số hiệu của quân cờ. Hàm này được gọi trong `Person` mỗi khi người chơi nhấn phím trái của chuột để chọn quân đi.
- Hàm lôgic `PosOk(k, u, v, j)` kiểm tra khi người chơi nhả chuột, vị trí mới của quân cờ `k` được chọn có hợp lệ không. Nếu hợp lệ, hàm trả về `true` và trả về thông tin vị trí mới của quân cờ cho các biến `u`, `v`, `j`, trong đó `u`, `v` là tọa độ tâm của quân cờ và `j` là số hiệu dòng bàn cờ mà nó đứng. Hàm này được gọi trong `Person` mỗi khi người chơi nhả phím trái của chuột để ấn định vị trí mới của quân được chọn.
- Thủ tục `RandP(p)` sinh ngẫu nhiên hoán vị `p` trên tập 1, 2, ..., n. Nó được gọi trong `Computer` để thủ tục này có thể duyệt các quân cờ theo một thứ tự ngẫu nhiên trong quá trình tìm kiếm nước đi.
- Hàm lôgic `Lose(p)` trả về `true` nếu các quân cờ `p` hết nước đi. Nó được gọi trong các thủ tục `Person` (với giá trị tham số là `pers`) và thủ tục `Computer` (với giá trị tham số là `comp`) để kiểm tra việc kết thúc của ván cờ.

Trong chương trình có dùng một số kỹ thuật đồ họa như kéo ảnh bằng chuột, tạo ảnh chuyển động, đã được bàn đến trong chương **MỘT SỐ UNIT CHUẨN** trong phần giới thiệu unit GRAPH. Chương trình được tổ chức lặp lại để có thể chơi nhiều ván liên tiếp (với cùng kích thước) và kết thúc khi ấn phím Esc.

Dưới đây là toàn văn chương trình, bạn đọc có thể tìm hiểu thêm chi tiết qua các lời chú giải trong chương trình.

```
Program Chess;
Uses MOUSE, CRT, GRAPH;
```

```
Const
  d = 16;          {độ rộng ô}
  r = d div 2;     {bán kính quân cờ}
Type
  arr = array[1..40] of integer;
Var
  n, m: integer;
  ch: char;
  pers, comp: arr;
  PersTurn, stop: boolean;
  pt: pointer;

  function DetectInit(path: string): integer;
  var drive, mode: integer;
  begin
    drive := 0;
    InitGraph(drive, mode, path);
    DetectInit := GraphResult;
  end;

  procedure CheckDevice;
  {kiểm tra các thiết bị và ấn định kích thước}
  begin
    if not InitMouse then
    begin
      Write('Mouse khong duoc cai dat! An phim Enter de ket thuc...');
      Readln; Halt;
    end;
    if DetectInit('C:\TP\BGI') <> 0 then
    begin
      Write('Do hoa co loi! An phim Enter de ket thuc...');
      Readln; Halt;
    end;
    DirectVideo := false;
    GotoXY(1, 1);
    Writeln('Chon kich thuoc ban co');
    Write('  so cot (khong qua 40): '); Readln(n);
    Write('  so dong (khong qua 29): '); Readln(m);
    if (n > 40) or (m > 29) then
    begin
      Write('Vuot qua kich thuoc! An phim Enter de ket thuc...');
      Readln; CloseGraph; Halt;
    end;
  end;

  procedure Piece(i, j, color: integer);
  {tô mầu quân cờ nằm ở cột i, dòng j bàn cờ với mầu tô color}
```

```
var x, y: integer;
begin
  SetFillStyle(SolidFill, color);
  SetColor(color);
  x := (i-1)*d+r;
  y := (j-1)*d+r;
  FillEllipse(x, y, r-1, r-1);
end;

procedure Init;
{khởi tạo ngẫu nhiên và trình bày thế cờ ban đầu}
var i, xmin, ymin, xmax, ymax: integer;
begin
  ClearDevice;
  {vẽ bàn cờ}
  xmin := 0; xmax := n*d; ymin := 0; ymax := m*d;
  SetFillStyle(SolidFill, 1);
  Bar(xmin, ymin, xmax, ymax);
  SetLineStyle(SolidLn, 0, 1);
  SetColor(LightGray);
  for i := 0 to n do Line(i*d, ymin, i*d, ymax );
  for i := 0 to m do Line(xmin, i*d, xmax, i*d);
  {cất nền ảnh cũ}
  GetMem(pt, ImageSize(1, 1, d-1, d-1));
  GetImage(1, 1, d-1, d-1, pt^);
  {sinh ngẫu nhiên và vẽ các quân cờ}
  Randomize;
  for i := 1 to n do
  begin
    repeat
      pers[i] := 1+Random(m);
      comp[i] := 1+Random(m);
    until (pers[i] <> comp[i]);
    Piece(i, pers[i], red);
    Piece(i, comp[i], white);
  end;
  {đặt vùng giới hạn và hiển thị Mouse}
  ShowMouse;
  RangeMouse(xmin+r, ymin+r, xmax-r, ymax-r);
  SetTextStyle(SmallFont, HorizDir, 4);
  SetTextJustify(LeftText, BottomText);
  OutTextXY(500, 475, 'Person');
  OutTextXY(570, 475, 'Computer');
  Piece(35, 30, red); Piece(40, 30, white);
  OutTextXY(0, 475,'Ban di truoc (y/n)? '); ch := ReadKey;
  if ch = #27 then
  begin
```

```
    stop := true; exit;
  end;
  PersTurn := Upcase(ch) = 'Y';
  stop := false; ch := #0;
end;

function SelOk(x, y: integer; var u, v, k: integer): boolean;
{kiểm tra việc chọn quân đi của người chơi tại tọa độ x, y của chuột có thích hợp không? nếu thích
 hợp, hàm trả về kết quả chọn: u, v là tọa độ tâm và k là số hiệu của quân cờ được chọn}
var i: integer;
begin
  for i := 1 to n do
  begin
    u := (i-1)*d+r;
    v := (pers[i]-1)*d+r;
    if (Abs(x-u) <= r) and (Abs(y-v) <= r) then
    begin
      k := i;
      SelOk := true;
      exit;
    end;
  end;
  SelOk := false;
end;

function PosOK(k: integer; var u, v, j: integer): boolean;
{kiểm tra việc đặt quân số hiệu k của người chơi có đúng luật không? nếu đúng luật,
 hàm trả về vị trí đặt: u, v là tọa độ tâm và j là giá trị dòng của quân được đặt}
begin
  if (u <= (k-1)*d) or (u >= k*d) then
  begin
    PosOk := false; exit;
  end;
  if v mod d = 0 then
  begin
    PosOk := false; exit;
  end;
  j := v div d+1;
  if pers[k] < comp[k] then
    if (j = pers[k]) or (j >= comp[k]) then
    begin
      PosOk := false; exit;
    end;
  if pers[k] > comp[k] then
    if (j = pers[k]) or (j <= comp[k]) then
    begin
      PosOk := false; exit;
```

```
    end;
  u := (k-1)*d+r;
  v := (j-1)*d+r;
  PosOk := true;
end;

function Lose(var p: arr): boolean;
{trả lại true nếu các quân cờ p hết nước đi}
var i: integer;
begin
  Lose := false;
  for i := 1 to n do
    if (Abs(comp[i]-pers[i]) > 1) or
       ((p[i] > 1) and (p[i] < m)) then exit;
  Lose := true;
end;

procedure Person;
{thực hiện lượt đi của người chơi: quân được chọn k, dòng được đi đến j}
var u, v, k, x, y, a, j: integer;
    left, sel, ok: boolean;
begin
  if Lose(pers) then    {người chơi thua}
  begin
    stop := true; exit;
  end;
  SetFillStyle(EmptyFill, 0);
  Bar(0, 465, 495, 479);
  SetColor(white);
  OutTextXY(0, 475, 'Moi ban di ....');
  a := r-1;
  ok := false; sel := false;
  repeat
    GetMouse(x, y, left);
    if sel and ((x <> u) or (y <> v)) then    {kéo quân cờ}
    begin
      HideMouse;
      PutImage(u-a, v-a, pt^, NormalPut);
      GetImage(x-a, y-a, x+a, y+a, pt^);
      SetFillStyle(SolidFill, LightRed);
      SetColor(LightRed);
      FillEllipse(x, y, a, a);
      u := x; v := y;
      ShowMouse;
    end;
    if not sel and left then    {chọn quân để đi}
      if SelOk(x, y, u, v, k) then
```

```
      begin
        HideMouse;
        SetFillStyle(SolidFill, LightRed);
        SetColor(LightRed);
        FillEllipse(u, v, a, a);
        sel := true;
        SetMouse(u, v);
        ShowMouse;
      end;
    if sel and not left then    {đặt quân vào vị trí mới}
    begin
      SetFillStyle(SolidFill, red);
      SetColor(red);
      if PosOk(k, x, y, j) then
      begin
        HideMouse;
        PutImage(u-a, v-a, pt^, NormalPut);
        GetImage(x-a, y-a, x+a, y+a, pt^);
        FillEllipse(x, y, a, a);
        sel := False;
        ShowMouse;
        pers[k] := j;
        ok := true;
        PersTurn := false;
      end else
      begin
        HideMouse;
        PutImage(u-a, v-a, pt^, NormalPut);
        x := (k-1)*d+r;
        y := (pers[k]-1)*d+r;
        GetImage(x-a, y-a, x+a, y+a, pt^);
        FillEllipse(x, y, a, a);
        sel := False;
        ShowMouse;
      end;
    end;
    if KeyPressed then ch := ReadKey;
    stop := ch = #27;
  until ok or stop;
end;

procedure RandP(var p: arr);
{sinh ngẫu nhiên hoán vị p}
var i, j, nq: integer;
    q: arr;
begin
  for i := 1 to n do q[i] := i;
```

```
  nq := n;
  for i := 1 to n do
  begin
    j := 1+Random(nq);
    p[i] := q[j];
    q[j] := q[nq];
    Dec(nq);
  end;
end;

procedure Computer;
{thực hiện lượt đi của máy: quân được chọn k, số ô di chuyển h, hướng di chuyển t}
var s, t, i, j, k, h, a, min, max, x, y: integer;
    p: arr;
begin
  if Lose(comp) then    {máy thua}
  begin
    stop := true; exit;
  end;
  SetFillStyle(EmptyFill, 0);
  Bar(0, 465, 495, 479);
  SetColor(white);
  OutTextXY(0, 475, 'Computer ...');
  s := 0;    {tính tổng XOR các khoảng cách}
  for i := 1 to n do
    s := s XOR (Abs(comp[i]-pers[i])-1);
  RandP(p); {sinh ngẫu nhiên thứ tự duyệt}
  h := 0;    {tìm quân cờ cần đi}
  for j := 1 to n do
  begin
    i := p[j];
    a := Abs(comp[i]-pers[i])-1;
    t := s XOR a;
    if a > t then
    begin
      k := i;
      h := a-t;
      Break;
    end;
  end;
  if h > 0 then    {tìm được quân cờ k có nước đi để dẫn về thế thắng}
    if comp[k] < pers[k] then t := 1 else t := -1
  else             {tìm nước đi chờ thời}
  begin
    for j := 1 to n do
    begin
      i := p[j];
```

```
    if comp[i] < pers[i] then
    begin
      min := 1; max := pers[i]-1;
    end else
    begin
      min := pers[i]+1; max := m;
    end;
    if min < max then
    begin
      k := i;      {k là quân được chọn}
      if comp[k] = min then
      begin
        t := 1;
        h := 1+random(max-min);
      end else
      if comp[k] = max then
      begin
        t := -1;
        h := 1+random(max-min);
      end else
      begin
        if random < 0.5 then
        begin
          t := 1;
          h := 1+random(max-comp[k]);
        end else
        begin
          t := -1;
          h := 1+random(comp[k]-min);
        end;
      end;
      Break;
    end;
  end;   {end for j}
end;   {end if h}
{di chuyển quân k, h ô, theo hướng t}
x := (k-1)*d+r;
y := (comp[k]-1)*d+r;
a := r-1;
SetFillStyle(SolidFill, white);
SetColor(white);
HideMouse;
for i := 1 to 2*h do
begin
  PutImage(x-a, y-a, pt^, NormalPut);
  y := y+t*r;
  GetImage(x-a, y-a, x+a, y+a, pt^);
```

```
      FillEllipse(x, y, a, a);
      Delay(100);
    end;
    ShowMouse;
    comp[k] := comp[k]+t*h;
    PersTurn := true;
  end;

  procedure MessResult;
  {thông báo kết quả}
  begin
    SetFillStyle(EmptyFill, 0);
    Bar(0, 465, 495, 479);
    SetColor(white);
    MoveTo(0, 475);
    if ch = #27 then OutText('Chuong trinh bi ngat! ')
    else
      if PersTurn then OutText('Ban da thua! ')
      else OutText('Ban da thang! ');
    OutText('An mot phim de tiep tuc, an Esc de ket thuc ...');
  end;

Begin
  CheckDevice;
  repeat
    Init;
    while not stop do
    begin
      if PersTurn then Person
      else Computer;
    end;
    MessResult;
    ch := ReadKey;
    if ch = #0 then ch := ReadKey;
    HideMouse;
  until ch = #27;
  CloseGraph;
End.
```

Bạn thử chơi với máy qua chương trình trên bằng cách nâng dần kích thước bàn cờ để thấy rằng khả năng thắng cuộc của máy là rất lớn. Nếu máy đang ở thế thắng thì bạn không thể đảo ngược tình thế dù bạn cố gắng đi bất cứ cách nào. Trái lại, bạn chỉ cần đi sai một nước trong khi đáng lẽ bạn có cách đi thắng, thì bạn vẫn trở thành người thua. Ngay cả với những ván bạn đã thắng và ghi nhớ nước đi, nếu chơi lại, máy có thể đi những nước khác làm cho bạn gặp khó khăn rất nhiều trong việc vận dụng trí nhớ của mình.

Chương trình trên có thể hoàn thiện hơn trong giao tiếp, chẳng hạn có thể xây dựng một hệ thống menu cho phép người chơi chọn lại các tham số của ván cờ, hoặc khởi tạo ván mới tại bất cứ lúc nào. Bạn cũng có thể thêm khả năng trong việc xây dựng thế cờ ban đầu bằng cách đọc thế cờ đã cất trên file hoặc cho phép người chơi xếp đặt theo ý muốn bằng chuột. Trong quá trình chơi, bạn cũng có thể thêm khả năng cất thế cờ hiện tại lên file để lần sau chơi tiếp, v, v, ...

Vấn đề dành cho bạn. Hãy viết một chương trình chơi cờ với máy sau đây:

Một bàn cờ kích thước 8x8. Trên bàn cờ có đặt một quân mã tại một ô nào đấy. Đến lượt mình, mỗi bên có thể đi quân mã đến một ô tùy thích trên bàn cờ (theo luật đi của mã), miễn là ô đó mã chưa đi qua. Bên nào đến lượt mà không đi được thì xem như thua cuộc.

Bạn cần xây dựng một chiến lược đi cho máy sao cho nếu người chơi chọn đi trước thì máy luôn có cách đi thắng cuộc. Trong trường hợp máy phải đi trước thì cố gắng xây dựng một cách đi chờ thời hiệu quả nhất để khi người chơi sai lầm, máy vẫn có thể thắng lại. Cần phải sử dụng việc sinh ngẫu nhiên để máy có thể đi những cách khác nhau khi ở những thế cờ giống nhau, nhằm che giấu chiến lược đi của máy. Chúc các bạn thành công.

PHỤ LỤC

MÔI TRƯỜNG TURBO PASCAL

Turbo Pascal (viết tắt là TP) là sản phẩm của hãng Borland (Mỹ), nhằm cài đặt Pascal chuẩn vào các máy PC-IBM và chạy dưới hệ điều hành MS-DOS.

TP cung cấp một môi trường làm việc với chương trình dịch hết sức thuận tiện, nó bao gồm các chức năng cơ bản: biên soạn chương trình nguồn, biên dịch chương trình nguồn ra mã máy và thực hiện chương trình. Ngoài ra, để hỗ trợ các công việc này, TP còn cung cấp một hệ thống menu cho phép giao diện bằng bàn phím và chuột, một hệ thống thông tin trợ giúp theo ngữ cảnh, các tiện nghi gỡ rối, ..., vì thế môi trường làm việc của TP còn được gọi là *môi trường phát triển kết hợp* (Integrated Development Environment), gọi tắt là IDE.

TP tuân theo rất chặt chẽ những định nghĩa của Pascal chuẩn đã được K. Jensen và N. Wirth trình bày trong cuốn *"Pascal User Manual and Report"*. Tuy nhiên nó cũng có một số ít biến đổi và cài đặt thêm một số thủ tục và hàm nhằm làm tăng thêm tính mềm dẻo của ngôn ngữ.

Từ khi ra đời, TP đã được đông đảo người dùng hưởng ứng vì tốc độ dịch cực nhanh và các tiện nghi mà IDE cung cấp. Hiện nay, TP thường được dùng với các phiên bản (version) 6 và 7. Bài này giới thiệu cách dùng IDE của TP qua phiên bản 7, về cơ bản, nó cũng giống như phiên bản 6.

1. Cài đặt

Tốt hơn cả là cài đặt đầy đủ TP bằng chương trình INSTALL của nó. Theo ngầm định, chương trình này sẽ tổ chức thư mục C:\TP, trong đó có nhiều thư mục con như BIN, BGI, ... để đặt các file của TP. File chính (được gọi bởi người sử dụng) của TP là TURBO.EXE, nó được cất (ngầm định) trong thư mục C:\TP\BIN. Bằng cách này, bạn có đầy đủ các tiện ích mà TP cung cấp.

Tuy nhiên, bạn có thể làm việc với TP qua 2 file tối thiểu là TURBO.EXE (403655 bytes) và TURBO.TPL (48432 bytes). Các file này cần chép vào một thư mục. Để có thể dùng được các thông tin trợ giúp, bạn cần chép thêm file TURBO.TPH (700786 bytes) vào cùng thư mục với các file trên. Chú ý rằng, cần phải chép các file của TP thuộc cùng một phiên bản để đảm bảo tính tương thích bởi vì các file của TP vẫn giữ nguyên tên trong khi nội dung của chúng đã được sửa đổi (có thể nhận ra điều này qua sự thay đổi của kích thước các file).

2. Hệ thống menu của TP

Tại dấu nhắc của DOS, trong thư mục có chứa các file TURBO.EXE và TURBO.TPL (ngầm định là thư mục C:\TP\BIN), khi gõ lệnh TURBO, bạn sẽ được màn hình của TP:

```
File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
+-[_]---------------------------- NONAME00.PAS --------------------------1-[_]-+
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
+-------- 1:1 -----_____________________________________________________________-+
 F1 Help  F2 Save  F3 Open  Alt+F9 Compile  F9 Make  Alt+F10 Local menu
```

Hình 1. Màn hình khi vào TP

Trên đỉnh cửa sổ này là các chức năng chính của TP được trình bày dưới dạng một dòng ngang. Để vào hệ thống menu của TP, bạn có thể dùng chuột để chọn hoặc gõ phím F10, sau đó dùng các phím mũi tên để di chuyển hộp sáng đến mục cần chọn và nhấn phím Enter. Mỗi chức năng chính của TP lại có một loạt chức năng con trong nó, được trình bày dưới dạng một cột dọc và cũng được chọn theo thể thức như chọn chức năng chính. Quy ước cách viết *chức năng chính / chức năng con* để mô tả việc chọn một chức năng con trong chức năng chính chứa nó. Thí dụ cách viết *Run / Run* mô tả việc chọn chức năng con *Run* trong chức năng chính *Run*. Ngoài việc chọn bằng chuột và di chuyển hộp sáng bằng các phím mũi tên, TP còn cho phép vào nhanh menu bằng cách gõ những *phím nóng* (hot key) như sau: để vào một chức năng chính, bạn gõ phím Alt kèm với phím

tương ứng với chữ cái đầu của chức năng này (có mầu phân biệt với những chữ cái khác). Thí dụ gõ Alt+F để vào chức năng *File*. Khi đang ở một chức năng chính, bạn có thể chọn một chức năng con của nó bằng cách gõ phím tương ứng với chữ cái có mầu tương phản với những chữ khác của chức năng con này (thường là chữ cái đầu). Thí dụ đang ở chức năng chính *File*, bạn gõ phím O để chọn chức năng *File / Open*. Ngoài ra, một số chức năng con hay dùng được TP bố trí vào nhanh bằng cách gõ phím tương ứng, gọi là *phím tắt* (short key). Thí dụ việc gõ phím F2 tương đương với việc chọn *File / Save*. Trong hệ thống menu, các phím tắt được trình bày bên cạnh các chức năng con tương ứng. Khi đang ở hệ thống menu, muốn thoát ra để về chế độ soạn thảo, bạn gõ phím Escape.

Nói chung, bước đầu bạn chỉ cần quan tâm đến những chức năng thường dùng nhất. Những chức năng này sẽ được giới thiệu trong các mục tiếp theo. Để đơn giản, chúng tôi chủ yếu trình bày cách chọn chức năng bằng phím tắt. Việc tìm hiểu thêm hệ thống menu, các bạn có thể tiến hành dần trong các giờ trên máy.

3. Chương trình soạn thảo văn bản của TP

Khi bắt đầu vào IDE, TP cung cấp ngay cho người dùng một cửa sổ soạn thảo (chưa có nội dung). Tại cửa sổ này, bạn có thể soạn thảo một văn bản bất kỳ với các lệnh soạn thảo tương tự như các hệ soạn thảo thông dụng.

Trong cửa sổ soạn thảo có một điểm nháy (con trỏ soạn thảo), nó đóng vai trò như một đầu viết, nơi mà các ký tự gõ vào sẽ được hiển thị. Có hai chế độ soạn thảo: viết chèn và viết đè. Khi viết chèn, ký tự mới gõ sẽ chèn vào nơi viết (các ký tự sau nó sẽ được đẩy sang phải), khi viết đè, ký tự mới gõ sẽ thay thế (đè lên) ký tự cũ ở nơi viết. Chế độ ngầm định là viết chèn. Để chuyển chế độ chèn / đè, bạn gõ phím Insert.

Các phím mũi tên ←, →, ↑, ↓ dùng để di chuyển điểm nháy trong trang soạn thảo. Phím Home chuyển điểm nháy về đầu dòng, phím End chuyển điểm nháy về cuối dòng. Tổ hợp phím Ctrl+Home chuyển điểm nháy về đầu trang, Ctrl+End chuyển điểm nháy về cuối trang. Phím PageUp chuyển lên trang trước, phím PageDown chuyển xuống trang sau. Tổ hợp phím Ctrl+PageUp chuyển điểm nháy về đầu file, Ctrl+PageDown chuyển điểm nháy về cuối file.

Các phím Backspace và Delete dùng để xóa các ký tự. Để chèn dòng trắng, bạn nhấn phím Enter. Để xóa nguyên một dòng (nơi có điểm nháy đang đứng), bạn gõ tổ hợp phím Ctrl+Y.

Các lệnh thao tác khối giúp cho việc soạn thảo nhanh và chính xác. Để đánh dấu khối, TP có 2 cách:

Cách 1. Đưa điểm nháy đến vị trí đầu khối rồi gõ tổ hợp phím Ctrl+K, B, sau đó đưa điểm nháy đến vị trí cuối khối rồi gõ tổ hợp phím Ctrl+K, K.

Cách 2. Nhấn giữ phím Shift và dùng các phím mũi tên để quét phạm vi khối (giống như soạn thảo trong Microsoft Word).

Khi khối được đánh dấu, nó sẽ có mầu tương phản với mầu nền của cửa sổ soạn thảo. Nếu đánh dấu một khối mới thì khối cũ sẽ không còn tác dụng. Một khối đã được đánh dấu có thể bỏ việc đánh dấu bằng cách gõ tổ hợp phím Ctrl+K, H.

Khi một khối đã được đánh dấu, bạn có thể dùng các thao tác trên khối sau đây:

1. Sao chép khối: Đưa điểm nháy đến vị trí định sao chép khối đã đánh dấu rồi gõ tổ hợp phím Ctrl+K, C.
2. Di chuyển khối: Đưa điểm nháy đến vị trí định di chuyển khối đã đánh dấu rồi gõ tổ hợp phím Ctrl+K, V.
3. Xóa khối: Để xóa khối đã đánh dấu ra khỏi văn bản đang soạn thảo có thể gõ tổ hợp phím Ctrl+K, Y hoặc tổ hợp phím Ctrl+Delete.
4. Chép khối vào vùng đệm: Khối đánh dấu có thể chép vào một vùng nhớ trung gian (gọi là *clipboard*) để có thể dán nó vào bất cứ một văn bản nào đang soạn thảo (giống như Microsoft Word). Có hai kiểu chép:
   - Chép khối vào clipboard, đồng thời xóa khối ra khỏi văn bản đang soạn thảo, gọi là thao tác cắt (cut): Gõ tổ hợp phím Shift+Delete.
   - Chép khối vào clipboard, nhưng không xóa khối ra khỏi văn bản đang soạn thảo, gọi là thao tác chép (copy): Gõ tổ hợp phím Ctrl+Insert.
5. Dán khối từ clipboard vào văn bản đang soạn thảo: Khối được chép sau chót vào cipboard có thể chèn vào bất cứ văn bản nào đang soạn thảo. Thao tác này được gọi là thao tác dán (paste). Đưa điểm nháy đến nơi muốn chèn và gõ tổ hợp phím Shift+Insert.

Chương trình soạn thảo TP còn cung cấp các thao tác tìm kiếm và thay thế nhằm giúp người dùng dễ dàng hiệu chỉnh lỗi. Để tìm kiếm, bạn gõ tổ hợp phím Ctrl+Q, F. Để tìm kiếm và sau đó thay thế, bạn gõ tổ hợp phím Ctrl+Q, A.

Trong khi đang soạn thảo, bạn có thể chèn nội dung một file văn bản khác vào vị trí điểm nháy bằng cách gõ tổ hợp phím Ctrl+K, R cũng như ghi một khối đã đánh dấu sang một file văn bản khác bằng cách gõ Ctrl+K, W.

TP cho phép mở nhiều cửa sổ soạn thảo. Muốn mở một file mới để soạn thảo bạn gõ phím F3. Để đóng cửa sổ đang soạn thảo bạn gõ tổ hợp phím Alt+F3.

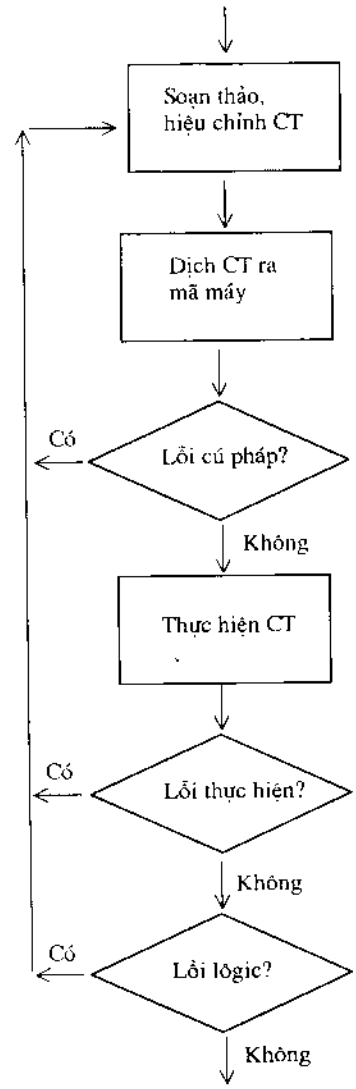
Chú ý rằng, nội dung đang soạn thảo được lưu trữ trong RAM, vì thế nó sẽ mất khi nguồn điện bị cắt. Để cất nội dung đang soạn thảo lên đĩa, bạn gõ phím F2. Nói chung, khi đang soạn thảo, thỉnh thoảng bạn nên gõ phím F2 để lưu lên đĩa những thông tin mới nhất.

Mặc dù về nguyên tắc, chương trình soạn thảo TP cho phép biên soạn một văn bản bất kỳ, tuy nhiên ngầm định của nó là phục vụ biên soạn các file chương trình của Pascal. Vì thế, TP luôn tự động gắn phần mở rộng là PAS vào tên file (đặc trưng cho các file nguồn Pascal) trong các thao tác đọc, ghi nếu tên file đưa vào không có phần mở rộng. Muốn TP hiểu được là tên file không có phần mở rộng, bạn phải đưa thêm dấu chấm vào sau tên file.

4. Quá trình làm việc với TP

Sau khi vào hệ thống TP, công việc chủ yếu là biên soạn chương trình và cho chạy. Quá trình đó có thể trình bày qua những bước sau:

1. *Biên soạn và hiệu chỉnh chương trình:* Đây là bước dùng chương trình soạn thảo của TP để biên soạn một chương trình mới hoặc hiệu chỉnh một chương trình cũ. Nếu soạn thảo file mới, bạn có thể dùng ngay trang soạn thảo mà TP cung cấp lúc ban đầu (trang trắng), hoặc mở file bằng cách gõ phím F3 (*File / Open*), rồi đưa tên file vào. Nếu tên này đã có (file cũ) thì nội dung của file sẽ được hiển thị trong cửa sổ soạn thảo để bạn hiệu chỉnh hoặc gõ thêm, nếu tên này là không có (file mới) thì TP sẽ mở ra một trang trắng cho bạn soạn thảo. Bạn cũng có thể lựa chọn tên trong bảng tên file mà TP cung cấp để mở file. TP cho phép mở nhiều cửa sổ soạn thảo để có thể soạn nhiều file cùng một lúc. Để chuyển từ một cửa sổ soạn thảo sang cửa sổ soạn thảo kế tiếp, bạn gõ phím F6 (*Window / Next*) hoặc Shift+F6 (*Window / Previous*), nếu theo chiều ngược lại. Để đóng cửa sổ soạn thảo hiện hành, bạn gõ tổ hợp phím Alt+F3 (*Window / Close*). Khi kết thúc, bạn nhớ gõ phím F2 (*File / Save*) để cất nội dung đã soạn thảo lên đĩa (trừ khi bạn cố ý không muốn lưu lại). Tất cả các tên file chương trình của TP đều được ngầm định với phần mở rộng là PAS.



Hình 2.
Quá trình biên soạn và chạy chương trình

2. *Dịch chương trình:* Để dịch chương trình đang có trong cửa sổ soạn thảo hiện hành, bạn gõ tổ hợp phím Alt+F9 (*Compile / Compile*). Chương trình dịch của TP sẽ lần lượt dịch từng lệnh viết trong chương trình nguồn ra mã máy. Trong quá trình dịch, nếu gặp *lỗi cú pháp* (syntax error) thì hệ thống báo lỗi của TP sẽ hiển thị thông báo lỗi lên màn hình, đồng thời điểm nháy trong cửa sổ soạn thảo sẽ được định vị tại nơi mà TP tìm ra lỗi để người dùng quay về bước 1 hiệu chỉnh lại chương trình. Bước dịch chương trình sẽ được thực hiện lặp lại cho đến khi không còn gặp một lỗi cú pháp nào nữa.

3. *Thực hiện chương trình:* Sau khi dịch xong, chương trình được chạy (thực hiện) bằng cách gõ tổ hợp phím Ctrl+F9 (*Run / Run*). Tại bước này có thể gặp lỗi, gọi là *lỗi thực hiện* (run-time error). Khi gặp lỗi loại này, hệ thống báo lỗi của TP cũng làm việc giống như khi gặp lỗi dịch (lỗi cú pháp) và người dùng lại quay về bước 1 để hiệu chỉnh

chương trình. Một chương trình được gọi là chạy thông khi không còn gặp lỗi thực hiện. Khi đó, người dùng sẽ nhận được kết quả của chương trình.

4. *Sửa lỗi kết quả:* Khi nhận được kết quả của chương trình, rất có thể kết quả này không đúng. Lỗi này gọi là *lỗi lôgic*. Lỗi này nằm ngoài phạm vi quản lý của TP và do rất nhiều nguyên nhân, có thể chỉ đơn giản là gõ nhầm các biểu thức nhưng cũng có thể là do cách giải sai! Để sửa lỗi này người lập trình phải tự tìm lấy bằng cách khoanh vùng dần để loại trừ và phải tự quay về bước 1 để hiệu chỉnh chương trình.

Nói chung, nếu chương trình được thiết kế tốt thì ít mắc lỗi lôgic và nếu có mắc thì cũng dễ phát hiện và sửa chữa, trái lại việc viết chương trình một cách tùy tiện sẽ dẫn đến rất khó phát hiện và sửa chữa những lỗi loại này, nhiều khi thiết kế và viết lại chương trình còn đỡ tốn công sức hơn là sửa lỗi!

Mặc dù TP không có trách nhiệm phát hiện các lỗi lôgic, nhưng nó cung cấp một hệ thống gỡ rối (debug) khá linh hoạt, cho phép người dùng theo dõi từng bước thực hiện chương trình để có thể lần ra dấu vết của những lỗi này.

Để kết thúc công việc đối với TP, bạn gõ tổ hợp phím Alt+X (*File / Exit*). Khi đó IDE của TP sẽ được đóng lại để trả về môi trường DOS. Quá trình biên soạn và chạy thử một chương trình có thể tóm tắt trong sơ đồ hình 2.

Chú ý.

1. Để tiện cho người dùng, TP cho phép khi biên soạn xong chương trình, có thể gõ tổ hợp phím Ctrl+F9 để thực hiện luôn. Thực chất, nếu thấy lúc này chương trình chưa được dịch thì TP sẽ tự động thực hiện luôn bước dịch.

2. TP ngầm định việc dịch ra mã máy là lưu trong RAM (nghĩa là lần sau chạy lại chương trình thì phải dịch lại). Muốn lưu bản đã dịch (mã máy) ra đĩa, trước khi dịch, cần vào chức năng *Compile / Destination* đổi từ chế độ *Memory* sang chế độ *Disk* (mỗi lần chọn, chế độ này lại được chuyển đổi). Khi đó, bản dịch sẽ được cất dưới dạng file có tên trùng với tên chương trình nguồn và có phần mở rộng là EXE và file này được thực hiện trong môi trường DOS. Nói chung, nếu không phải bàn giao file EXE cho người sử dụng, bạn nên để việc dịch ở chế độ *memory* (ngầm định). Chú ý rằng, nếu mất file EXE, ta có thể dịch lại từ file PAS, nhưng nếu mất file PAS, ta không thể tạo lại từ file EXE.

3. Việc chương trình không có lỗi (kể cả lỗi lôgic) sau vài lần chạy thử chưa chắc đã có thể kết luận là chương trình đúng, vì với bộ dữ liệu này, chương trình có thể không có lỗi nhưng với bộ dữ liệu khác chương trình lại có thể có lỗi. Vì thế, một chương trình chỉ được xem là tin cậy nếu nó đã được thử thách sau một thời gian đủ dài hoặc tính đúng đắn của nó được chứng minh chặt chẽ bằng toán học.

5. Sử dụng trợ giúp

TP cung cấp một hệ thống thông tin trợ giúp khá đầy đủ theo ngữ cảnh. Khi đang ở một công việc nào đấy trong TP mà muốn nhận được một sự chỉ dẫn về công việc đó, bạn gõ

phím F1. Khi đó, một cửa sổ (gọi là *cửa sổ trợ giúp*), trong đó có ghi những thông tin liên quan đến vấn đề bạn hỏi, được hiển thị lên màn hình. Cửa sổ trợ giúp có thể cuộn theo mọi hướng để bạn có thể đọc hết những thông tin ghi trong đó. Trong các lời chỉ dẫn, lại xuất hiện những đề mục, có mầu đặc biệt, để bạn có thể tra cứu tiếp (bằng cách đưa điểm nháy đến và nhấn phím Enter). Thí dụ, khi đang ở chức năng *File / Open*, bạn sẽ nhận được cửa sổ trợ giúp khi gõ phím F1. Trong cửa sổ này, những lời chỉ dẫn về chức năng mở file được trình bày. Bạn có thể tra cứu tiếp cụm từ *Open a File* trong đó để tìm hiểu chi tiết về hộp thoại mở file.

Một cách tra cứu khác là bạn đưa điểm nháy đến một từ được viết trong cửa sổ soạn thảo mà bạn muốn tra cứu rồi gõ tổ hợp phím Ctrl+F1. Khi đó, bạn sẽ nhận được lời chỉ dẫn về từ này. Đây là cách tra cứu cú pháp các thủ tục, hàm, kiểu, ... của TP rất thuận tiện. Thí dụ để tra cú pháp của thủ tục *Write* của TP, bạn gõ từ này trong cửa sổ soạn thảo (hoặc đưa điểm nháy đến từ này nếu nó đã có) rồi gõ Ctrl+F1. Bạn cũng có thể vào thẳng chức năng *Help* của TP để tìm hiểu các vấn đề về thông tin trợ giúp.

Nội dung trong cửa sổ trợ giúp có thể gửi vào clipboard để có thể dán nó vào những văn bản khác. Bạn đánh dấu khối văn bản trong cửa sổ trợ giúp bằng cách nhấn giữ phím Shift rồi dùng các phím mũi tên để kéo vệt sáng ghi nhận khối, sau đó bạn cất khối này vào cipboard bằng cách gõ tổ hợp phím Ctrl+Insert. Khối này sẽ được dán vào vẩn bản hiện hành bằng cách gõ tổ hợp phím Shift+Insert. Bằng cách này bạn có thể chép những thí dụ mà TP cung cấp vào trong văn bản chương trình của mình để thử.

```
File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
 +-----------------+ ----------- NONAME00.PAS -------------------------1-[_]-+
 ¦ New             ¦                                                          _
 ¦ Ope+-[_]---------------------- Turbo Help ---------------------------+      _
 ¦ Sav¦+-----------------------------------------------+                ¦      _
 ¦ Sav¦¦                                                -                ¦      _
 ¦ Sav¦¦  _ File¦Open  (F3)                             _   Cross ref _  ¦      _
 +----¦¦  ________________                              _   ___________  ¦      _
 ¦ Cha¦¦ The Open command displays the Open a File      _                ¦      _
 ¦ Pri¦¦ dialog box. In this dialog box, you select the _   Previous  _ ¦      _
 ¦ Pri¦¦ program file you want to open in an Edit       _   ___________  ¦      _
 ¦ DOS¦¦ window.                                        _   Contents  _ ¦      _
 ¦ Exi¦¦                                                _   ___________  ¦      _
 +----¦¦ The Open a File dialog box contains:           _   Index     _ ¦      _
¦     ¦¦                                                _   ___________  ¦      _
¦     ¦¦  _ an input box with a history list            _                ¦      _
¦     ¦¦  _ a list box to traverse                      _                ¦      _
¦     ¦¦  _ the standard Cancel and Help action buttons _   Cancel    _ ¦      _
¦     ¦¦  _ Open and Replace action buttons             _   ___________  ¦      _
¦     ¦¦  _ an information panel that describes the                     ¦      _
¦     ¦+_______________________________________________+                ¦      _
¦     +-----------------------------------------------------------------+      _
¦                                                                              _
+------- 1:1 -----_____________________________________________________________-+
 F1 Help on help  Alt+F1 Previous topic  Shift+F1 Help index  Esc Close help
```

Hình 3. Cửa sổ trợ giúp chức năng *File / Open*

Chú ý rằng, để có thể dùng được trợ giúp, bạn cần có file TURBO.TPH cất trong cùng thư mục với các file TURBO.EXE và TURBO.TPL.

Chịu trách nhiệm xuất bản:

Giám đốc: PHÙNG QUỐC BẢO
Tổng biên tập: PHẠM THÀNH HƯNG

Người nhận xét: : PHẠM PHÚ TRIÊM

Biên tập: PHẠM PHÚ TRIÊM

Trình bày bìa: NGỌC ANH

LẬP TRÌNH NÂNG CAO TRÊN NGÔN NGỮ PASCAL

Mã số: 1L-01004-04105
In 1000 cuốn, khổ 19x27 tại Xí nghiệp in Nhà xuất bản Lý luận Chính trị
Số xuất bản: 77/56/XB-QLXB, ngày 17/1/2005.
Số trích ngang: 36 KH/XB
In xong và nộp lưu chiểu quý I năm 2005.

02020800000101

D 34.000 V.

Giá: 34.000đ