파일 입출력과 인쇄 기능의 추가

(File I/O and Printing)

파일의 입출력과 프린터를 통한 출력은 어플리케이션을 개발할 때 중요하게 여기고 개발해야 하는 기능이다. 이번 장에서는 파일을 가지고 작업하는 방법과, 인쇄를 할 때 델파이에서 사용할 수 있는 여러가지 기법에 대해서 알아볼 것이다.

파일의 종류와 입출력 방법

파일 입출력 루틴을 통해 사용할 수 있는 파일 종류에는 3 가지가 있다. 전통적인 스타일의 파스칼 파일과 윈도우 파일 핸들, 파일 스트림 객체가 그것이다.
전통적인 스타일의 파스칼 파일은 F: Text 또는 F: File 의 형태로 사용되는 파일 변수를 사용하게 된다. 이들 파일에는 크게 나누어 type, text, untyped 의 3 가지로 종류를 나누어 볼 수 있으며, AssignPrn, WriteLn 을 비롯한 수많은 파일 처리 루틴이 이들을 지원한다.
윈도우 파일 핸들은 실제 운영체제에서 사용되는 직접적인 형태이다. 파스칼 언어 자체는 기본적으로 멀티 플랫폼을 대상으로 하기 때문에, 궁극적으로 보면 오브젝트 파스칼의 파일 처리 루틴 역시 윈도우 파일 핸들을 다루는 wrapper 에 불과하다. 결국에는 윈도우 API 를 사용하게 되는데, 예를 들어, FileRead 함수는 윈도우 API 의 ReadFile 함수를 호출한다. 그런데, 오브젝트 파스칼의 루틴을 사용하지 않고, 직접 윈도우 API 를 사용해서 파일 처리를 하려 하면 반드시 파일의 핸들을 사용해야 한다.
파일 스트림은 VCL 클래스인 TFileStream 에 기초한 객체로, 파일 입출력에 높은 수준에서 접근할 수 있게 해준다. TFileStrean 의 Handle 프로퍼티는 윈도우의 파일 핸들과 동일한 것이다. 파일 스트림을 사용하면 보다 고급스럽고, 세련된 프로그래밍을 할 수 있기 때문에 많이 사용되는 클래스이다.

고정길이 파일 (Fixed Length Files) 다루기

델파이의 비정형(untyped) 파일은 read, write, read write 모드로 열 수 있으며 파일의 레코드 위치를 자유롭게 할 수 있다. 또한, 한 번에 하나 이상의 고정길이 레코드를 읽고 쓸 수 있으며, 그 수행속도가 뛰어나다.
이러한 비정형 파일을 이용해서 파일을 복사하는 프로시저를 만들 수 있는데, 이 프로시저를 예로 들어서 비정형 파일의 사용법을 알아보도록 하자.

```
procedure CopyFile(SrcFileName, DestFileName: String);
```

```
var
  Buffer: array[1..8192] of Char;
  SrcFile, DestFile: File;
  ReadCount, WriteCount: Integer;
begin
  AssignFile(SrcFile, SrcFileName);
  Reset(SrcFile, 1);
  AssignFile(DestFile, DestFileName);
  Rewrite(DestFile, 1);
  repeat
    BlockRead(SrcFile, Buffer, SizeOf(Buffer), ReadCount);
    BlockWrite(DestFile, Buffer, ReadCount, WriteCount);
  until (ReadCount = 0) or (WriteCount < ReadCount);
  CloseFile(SrcFile);
  CloseFile(DestFile);
end;
```

일단은 AssignFile 프로시저를 이용해서 파일을 정의하고, 이를 읽어오는 파일은 Reset 으로 쓰는 쪽은 Rewrite 를 사용한다. Reset 프로시저는 기존에 파일이 존재할 때 이 파일의 제일 처음에 포인터를 위치시키는 프로시저이고, Rewrite 는 새로 파일을 생성하거나, 기존에 같은 이름의 파일이 있다면 이를 삭제하고 새로 파일을 만드는 프로시저 이다.
Reset, Rewrite 프로시저는 모두 두 개의 파라미터를 가질 수 있다. 첫번째 파라미터에는 대상이 되는 파일을 넘겨주게 되어 있고, 두번째 파라미터에는 레코드의 크기를 지정한다. 이때 두번째 파라미터는 생략할 수 있는데 생략할 경우 128 바이트가 디폴트로 지정된다. 파일을 복사하는 경우에는 1 바이트도 읽고, 쓸 수 있어야 하므로, 두번째 파라미터를 모두 1 로 지정한다.
실제로 복사가 이루어지는 부분은 다음 코드이다.

```
repeat
  BlockRead(SrcFile, Buffer, SizeOf(Buffer), ReadCount);
  BlockWrite(DestFile, Buffer, ReadCount, WriteCount);
until (ReadCount = 0) or (WriteCount < ReadCount);
```

비정형 파일을 읽고, 쓸 때 사용하는 프로시저들이 BlockRead 와 BlockWrite 이다. BlockRead 는 4 개의 파라미터를 가지는데 각각 파일 변수, 데이터에 대한 버퍼로 사용할 변수나 구조체의 이름, 그리고 읽을 레코드의 수를 첫번째에서 세번째 파라미터로 지정한다.

이때 읽을 레코드의 수에 레코드 크기를 곱한 값은 버퍼의 크기와 작거나 같아야 한다.
네번째 파라미터는 실제로 읽은 레코드의 수가 담겨지게 되는 변수이다. 보통 이 값은 세번째 파라미터의 값과 동일하게 되지만 파일의 끝부분에 도달하면 이 값이 더 작게 된다.
앞의 경우에 세번째 파라미터를 SizeOf(Buffer)로 지정했는데, SizeOf 함수는 변수의 바이트 크기를 돌려주는 함수이다. 이 경우에는 레코드의 크기가 1 바이트이므로 버퍼 배열의 크기와 같게 된다. BlockWrite 프로시저도 마지막 파라미터에 실제로 기록한 레코드의 수가 넘어온다는 것을 제외하면 BlockRead 와 같은 파라미터를 사용한다.
ReadCount 변수의 값이 0 이라는 의미는 파일의 끝에 도달했다는 뜻이며, WriteCount 변수가 ReadCount 변수보다 작다는 의미는 디스크가 꽉 차는 등의 에러로 인해 읽은 레코드를 모두 쓸 수 없었다는 의미가 되므로 여기까지 루프를 돌리면 파일의 복사가 종료된다.
파일을 복사하는 루틴에 대한 설명으로 비정형 파일의 사용법은 거의 익숙해졌을 것으로 생각된다. 그러면, 단순한 파일의 복사가 아니라 레코드를 비정형 파일 기법을 이용해서 관리하는 방법에 대해서 알아보도록 하자.
이렇게 비정형 파일 기법을 이용하면 대단히 빠른 속도로 레코드를 읽고 쓸 수 있다. 참고로 레코드를 파일로 관리하는 방법에는 레코드 파일(File of Record)을 이용하는 방법이 있는데, 여기에 대해서는 이장의 후반부에서 더 자세히 다룰 것이다.
먼저 레코드를 다음과 같이 선언하도록 한다.

```
type
  TPerson = record
    Name: array[1..10] of Char;
    ID: array[1..8] of Char;
    Password: array[1..8] of Char;
  end;
```

이렇게 정의한 TPerson 레코드의 크기는 28 바이트가 된다.
그러면, 이 레코드 형의 데이터 10 만개에 특정 값을 지정해서 저장했다가, 이를 읽어오고 읽은 레코드의 수를 TLabel 컴포넌트를 이용해서 나타내는 예제를 제작해 보자.
먼저 다음과 같이 폼에 버튼 2 개와 TLabel 컴포넌트 4 개를 올려 놓고, 버튼의 캡션을 각각 '파일 생성'과 '읽어 오기', 그리고 Label2 와 Label4 의 캡션을 '레코드', 'msec'로 설정한다.

여기서 Label1 에는 읽어온 레코드의 수를, 그리고 Label3 에는 Win32 의 GetTickCount 함수를 이용해서 10 만 레코드를 읽어오는데 걸린 시간을 백만 분의 1 초 단위로 나타낸다. 여기서 10 만 레코드를 사용한 이유는 필자가 테스트 해 본 결과 1 만 레코드 읽는데에는 백만 분의 1 초 밖에 걸리지 않아서, 의미가 없기에 그래도 조금은 큰 10 만 레코드를 읽고, 쓰는데 걸리는 시간을 사용하기 위한 것이다. 10 만 레코드를 저장하면 파일의 크기는 약 2.5MB 정도가 된다.
그리고, 비정형 파일을 다룰 때에는 레코드의 멤버가 파스칼 문자열로 존재하면, 이를 고정된 길이의 문자 배열로 바꾸어서 문자가 없는 부분은 공백으로 채우는 프로시저를 하나 만들어 주어야 한다. 이러한 역할을 하는 AssignData 라는 프로시저를 interface 섹션에 선언하고 다음과 같이 구현한다.

```
procedure AssignData(const Data: String; var ArrData: array of Char);
var
  i: Word;
begin
  for i := 0 to High(ArrData) do ArrData[i] := ' ';
  for i := 1 to Length(Data) do ArrData[i - 1] := Data[i];
end;
```

이 프로시저는 일단 ArrData 파라미터를 열린 배열(open array)로 선언하고, 구현부분에서 High 함수를 이용해서 크기를 정할 수 있도록 하는 것이 중요하다. 그리고, 일단은 문자 배열의 내용을 모두 공백문자(‘ ‘)로 채운 뒤, 파스칼 문자열의 길이 만큼 문자 배열의 앞에서부터 채워 나간다.
이제, 파일을 생성하는 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
const
  MaxRec = 100;
var
  Buffer: array[1..MaxRec] of TPerson;
```

```
  PersonFile: File;
  i, Count: Integer;
begin
  AssignFile(PersonFile, 'Person.dat');
  Rewrite(PersonFile, SizeOf(TPerson));
  for i := 1 to MaxRec do
  begin
    with Buffer[i] do
    begin
      AssignData('정지훈', Name);
      AssignData('ttolttol', ID);
      AssignData('pswrd1', Password);
    end;
  end;
  for i := 1 to 1000 do
    BlockWrite(PersonFile, Buffer, MaxRec, Count);
  CloseFile(PersonFile);
end;
```

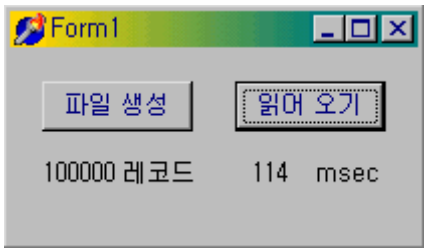
일단 레코드의 수를 100 개로 설정하고, TPerson 레코드 100 개의 레코드 배열을 버퍼로 선언한다. 앞에서 설명한 파일 복사 루틴과 비교해 보면 쉽게 이해할 수 있을 것이다.
Rewrite 프로시저를 사용해서 이미 존재하는 파일이 있으면 이를 삭제하고 새로 파일을 생성하도록 하며, 두번째 파라미터에는 레코드의 크기를 SizeOf(TPerson)을 이용해서 설정해 준다.
바로 다음의 for 루프에서는 버퍼에 100 개 까지의 레코드를 담을 수 있으므로, 1 부터 100 까지의 레코드 배열 버퍼에 문자열을 고정 길이의 문자 배열로 변환해서 Name, ID, Password 의 각 멤버에 저장한다. 이런 문자 배열로의 변환 작업에 앞에서 설명한 AssignData 프로시저를 이용한다.
그리고, 그 다음의 for 루프에서는 실제로 데이터를 쓰는 작업을 하는데 1000 번의 루프를 돌면서 버퍼의 레코드를 쓰게 되므로 총 10 만 레코드를 기록하게 된다.
그러면, 이번에는 Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
const
  MaxRec = 500;
type
```

```
  TBuffer = array[1..MaxRec] of TPerson;
var
  Buffer: ^TBuffer;
  PersonFile: File;
  Total, Count: Integer;
  StartTime, EndTime: LongInt;
begin
  try
    New(Buffer);
  except
    on EOutOfMemory do Exit;
  end;
  try
    AssignFile(PersonFile, 'Person.dat');
    Reset(PersonFile, SizeOf(TPerson));
    Total := 0;
    StartTime := GetTickCount;
    repeat
      Count := 0;
      BlockRead(PersonFile, Buffer^, MaxRec, Count);
      Total := Total + Count;
    until Count = 0;
    EndTime := GetTickCount;
    CloseFile(PersonFile);
    Label1.Caption := IntToStr(Total);
    Label3.Caption := IntToStr(EndTime - StartTime);
  finally
    Dispose(Buffer);
  end;
end;
```

레코드를 읽어올 때에도 버퍼를 사용하게 되므로 먼저 Buffer 변수를 선언한다. 이때 먼저 type 선언문에서 500 개의 레코드를 기억할 수 있는 레코드 배열인 TBuffer 를 선언하고, Buffer 변수는 이 레코드 배열에 대한 포인터로 선언하는 것이 메모리를 효율적으로 활용할 수 있는 방안이다.
이렇게 포인터로 선언한 Buffer 변수를 활용하려면 New 프로시저를 이용해서 메모리를 할

당받아야 하는데, 이 과정에서 에러가 발생하면 예외처리를 해준다. 그리고, 마지막에는 버퍼에 할당된 메모리를 Dispose 프로시저를 이용해서 해제해야 하므로 try-finally 블록을 사용한다.
나머지 부분은 그렇게 어렵거나 새로운 부분이 없으므로 따로 설명하지 않아도 쉽게 이해할 수 있을 것이다. 다만 레코드를 읽어오는데 걸리는 시간을 측정하는 GetTickCount 함수에 대해서 간단히 설명하면 이 프로시저는 백만 분의 1 초 단위로 시스템의 시각을 읽어오는 역할을 한다. 그러므로, 특정 작업을 시작하기 전에 이 함수를 호출해서 그 값을 LongInt 형의 변수에 저장하고, 작업이 끝나고 이 함수를 호출해서 그 값을 LongInt 형의 변수에 저장한 후 그 차를 계산하면 특정 작업에 소요된 시간을 백만 분의 1 초 단위로 알아낼 수 있다.
그러면, 이 프로그램을 실행해서 28 바이트 크기의 레코드 10 만개를 읽는데 걸리는 시간을 직접 알아보자. 다음 그림의 결과는 필자의 펜티엄-150MHz, HDD 2.1GB, RAM 80MB 의 노트북에서 실행한 결과이므로 시스템에 따라서 다소간의 차이가 있을 것이다.



즉, 10 만 레코드를 읽는데 소요되는 시간이 약 1/1000 초인 셈이다. 이 정도면 비정형 파일을 이용한 레코드 관리가 얼마나 빠른 속도로 이루어지는지 쉽게 알 수 있을 것이다.

텍스트 파일과의 호환성

앞에서 설명한 고정길이 파일의 경우에는 레코드 사이의 delimiter 를 사용하지 않았다. 그러므로, 편집기를 이용해서 파일을 열어보면 앞의 파일의 경우 '정지훈 ttolttolpswrd1 정지훈 ttolttolpswrd1 정지훈 ttolttolpswrd1 ...' 과 같이 저장되어 있는 것을 발견할 수 있을 것이다. 이러한 형태의 저장 방법은 과거의 메인 프레임이나 미니 컴퓨터에서 사용하던 방식으로 현재의 PC 환경과는 맞지 않는다.
대부분의 PC 용 프로그램 들은 고정길이 파일을 만들어낼 때 레코드의 마지막에 CR/LF(Carriage Return/Line Feed) 문자를 집어 넣어서 텍스트 파일을 만들어 낸다
그러므로, 이를 지원하기 위해서는 레코드에 다음과 같은 delimiter 멤버를 추가할 필요가 있다.

```
type
```

```
  TPerson = record
    Name: array[1..10] of Char;
    ID: array[1..8] of Char;
    Password: array of [1..8] of Char;
    Delimiter: array of [1..2] of Char;
  end;
```

그리고, 레코드를 쓰는 루틴에도 delimiter 를 추가하도록 수정해야 한다. 앞의 예제에서 파일을 생성하는 부분을 다음과 같은 형태로 수정하면 된다.

```
procedure TForm1.Button1Click(Sender: TObject);

... (중략)

  for i := 1 to MaxRec do
  begin
    with Buffer[i] do
    begin
      AssignData('정지훈', Name);
      AssignData('ttolttol', ID);
      AssignData('pswrd1', Password);
      Delimiter[1] := #13;
      Delimiter[2] := #10;
    end;
  end;

... (후략)
```

즉, 레코드 배열에 값을 대입할 때 CR/LF 문자를 추가해 주면 된다.

레코드 파일 시스템

델파이로 데이터베이스 어플리케이션을 제작하는데 단순한 레코드 형을 저장하기만 하면 되는 경우가 있다. 이럴 때에는 2MB 가 넘는 BDE 는 너무 무겁다고 할 수 있다. 이를 해결하기 위한 방편으로 BDE 를 사용하지 않는 새로운 DataSet 컴포넌트를 만들 수도 있지만, 이 역시 간단한 방법이라고는 할 수 없다. 여기에 대한 대안으로 레코드를 파일로 저장하

고 관리할 수 있는 방법이 있기에 이를 소개하고자 한다.

- 레코드 파일의 선언과 특성

레코드 파일은 레코드 형의 데이터를 순차적으로 저장하는 이진 파일을 말한다. 즉, 레코드의 배열을 파일에 저장하는 것으로 생각해도 무방하다. 이러한 레코드 파일은 다음과 같은 특성을 가지고 있다.

1. 레코드 파일은 메모리 상의 배열이 동적으로 크기 변화가 되면서 파일로 저장되는 것으로 생각할 수 있다.
2. 레코드 파일은 이진 파일이기 때문에 텍스트 에디터로 쉽게 편집할 없다. 그러므로, 모든 편집 과정은 프로그램이 담당하는 것이 보통이다.
3. Read 와 Write 함수를 사용할 수 있는데, 텍스트 파일에서 사용하는 ReadLn 과 WriteLn 함수와 마찬가지로 동작 후 다음 레코드로 포인터가 넘어간다. 참고로 텍스트 파일에서 사용하는 Read, Write 함수는 한 줄을 읽고 나면 다음 줄로 넘어가지 않는다.
4. 파일 내에 있는 각각의 레코드는 0 부터 시작되는 레코드 번호를 암묵적으로 부여받는다. Seek 함수를 이용해서 파일의 특정 위치로 바로 이동할 수 있다.

- 파일에 접근하기

텍스트 파일에서와 마찬가지로 AssignFile 함수를 이용해서 레코드 파일을 열게 된다. 이때 이미 존재하는 파일이라면 Reset 프로시저를, 새로 만드는 경우라면 Rewrite 프로시저를 사용한다. 레코드 파일을 닫을 때에는 CloseFile 프로시저를 사용한다.
이런 함수를 사용하려면 먼저 레코드 파일을 선언해야 하는데, 이는 레코드 데이터 형을 먼저 선언하고 이를 이용해서 다음과 같이 선언하면 된다.

```
type
  TMyRecord = record
    ID: Integer;
    Name: String;
  end;

var
  MyFile: File of TMyRecord;
```

파일을 처음 열 때에는 다음과 같이 한다.

```
AssignFile(MyFile, 'MyFile.dat');
if FileExists('MyFile.dat') then Reset(MyFile) else Rewrite(MyFile);
```

- 레코드 번호와 파일 포인터

레코드 파일에서의 레코드에는 순차적으로 레코드 번호가 주어진다. 만약에 10 개의 레코드가 있으면 각각 0 에서 9 까지의 번호를 가지게 된다. 일단 레코드 파일을 열면 파일 변수는 파일 포인터를 이용해서 레코드의 위치를 가리키게 된다. 파일을 열었을 때에는 파일 포인터가 파일의 제일 처음에 위치하게 되며, 0 번 레코드를 가리킨다. 이러한 파일 포인터의 위치를 설정하는데 Seek 함수가 사용된다. 예를 들어, 30 번째 레코드로 파일 포인터를 이동하고 싶으면 다음과 같이 한다.

```
Seek(MyFile, 30);
```

이렇게 일단 파일 포인터의 위치를 옮겨 놓고 Read, Write 함수를 이용해서 그 위치의 레코드를 읽거나 쓰게 된다. 그리고, Read 또는 Write 함수에 의해서 파일 포인터는 다음으로 이동하게 된다. 즉, 2 번 레코드를 읽었으면 파일 포인터는 3 번 레코드를 가리키게 된다. 이런 식으로 읽어나가다가 파일 포인터가 마지막 레코드를 읽고 나서, 다음 레코드를 읽으려고 하면 런타임 에러가 발생한다. 이런 에러를 방지하기 위해서 파일 포인터의 위치를 알아야 하는데, 이때 사용하는 함수가 FilePos 함수이다.
FilePos 함수는 현재 레코드의 번호를 반환하는데, FileSize 함수에서 알 수 있는 레코드의 수와 비교해서 현재의 레코드 위치가 마지막에 와있는지 알아낼 수 있다. 다음의 코드를 보면 쉽게 이해할 수 있을 것이다.

```
while (FilePos(MyFile) <= (FileSize(MyFile) – 1)) do
begin
  Read(MyFile, MyRecord);
  //Do something
end;
```

- 레코드 읽고 쓰기

레코드 파일에서 레코드를 읽을 때에는 Read 프로시저를 사용한다. 사용법은 앞에서도 몇 번 언급했지만 다음과 같다.

```
Read(MyFile, MyRecord);
```

앞의 문장에서 MyFile 은 파일을 담고 있는 파일 변수이며, MyRecord 는 레코드 변수이다. 레코드를 기록하는데 사용하는 Write 프로시저의 사용법 역시 비슷하다.

```
Write(MyFile, MyRecord);
```

레코드를 읽을 때에는 한 번에 여러 개의 레코드를 읽어올 수도 있다. 예를 들어 MyRecord1, MyRecord2, MyRecord3 라는 3 개의 변수에 3 개의 레코드를 다음과 같이 한꺼번에 읽어오게 할 수도 있다.

```
Read(MyFile, MyRecord1, MyRecord2, MyRecord3);
```

이들을 사용할 때 꼭 기억해야 할 것은 동작 후 파일의 포인터가 다음 레코드로 이동한다는 것이다. 별로 문제가 될 것 없는 사실인 것 같지만, 주의하지 않으면 레코드를 편집할 때 문제가 생길 수 있다. 예를 들어, 파일에서 여러 개의 레코드를 읽어서 이를 편집한다고 하자. 그리고, 이렇게 편집한 레코드를 다시 파일에 쓴다고 할 때에는 다음과 같은 형태의 프로시저를 사용 해야 한다.

```
procedure ChangeRecord(RecordNo: Integer; Value: String);
var
  Buffer: TMyRecord;
begin
  Seek(MyFile, RecordNo);
  Read(MyFile, Buffer);
  with Buffer do
    Name := Value;
  Seek(MyFile, RecordNo);              //이렇게 원래 레코드 위치로 복귀해야 한다 !
  Write(MyFile, Buffer);
end;
```

어려운 것은 아니지만 실수하기 쉬운 부분이므로 주의해야 한다. 즉, Read 프로시저에 의해 파일 포인터가 다음 레코드로 넘어갔기 때문에 다시 한번 Seek 프로시저를 이용해서 원래의 파일 포인터로 복귀 시키는 것이다.

- 레코드 버퍼의 활용

이런 식으로 데이터를 레코드 파일과 사용자 인터페이스 사이에서 이동하려면, 레코드 변수를 사용해서 파일의 레코드와 실제 에디트 컨트롤 사이의 임시 버퍼로 활용할 필요가 있다. 이런 버퍼가 필요한 이유는 델파이에서 지원하는 각종 데이터 어웨어 컨트롤들과는 달리 TEdit, TComboBox 등의 표준 컨트롤 들에는 데이터 링크와 같이 파일의 레코드와 직접 연결할 수 있는 방법이 없기 때문이다. 이러한 연결을 위해서 레코드 변수를 활용한 접근 메소드가 필요하다. 다음에 소개할 예제에서는 ReadRecord 와 SetRecordValue 라는 메소드와 MyRecord (TMyRecord 데이터 형)라는 레코드 변수를 버퍼로 활용해서 이러한 문제를 해결할 것이다.

- 레코드의 삽입과 삭제

레코드 파일에서는 특정 위치에 레코드를 삽입하거나 삭제하는 것이 기본적으로는 불가능하다. 이를 구현하려면 레코드 버퍼를 이용해서 조작을 하고 나서, 이 데이터를 레코드 파일에 기록하는 방법을 사용해야 한다. 간단하게 구현 단계를 설명하면 다음과 같다.

1. 적절한 레코드 형으로 레코드 배열을 초기화한다.
2. 현재 파일 포인터의 위치를 얻어서, 이를 특정 변수에 저장한다.
3. 사용자 인터페이스에 있는 기록할 레코드 정보를 임시 레코드 변수에 저장한다.
4. Seek 프로시저를 사용해서 파일의 처음으로 이동한다.
5. Read 프로시저를 이용해서 삽입할 레코드 위치의 바로 앞 레코드까지 읽는다.
6. 삽입될 레코드를 배열에 저장한다.
7. 파일의 끝까지 읽어서 배열에 저장한다.
8. 레코드 배열의 내용을 파일에 저장한다.

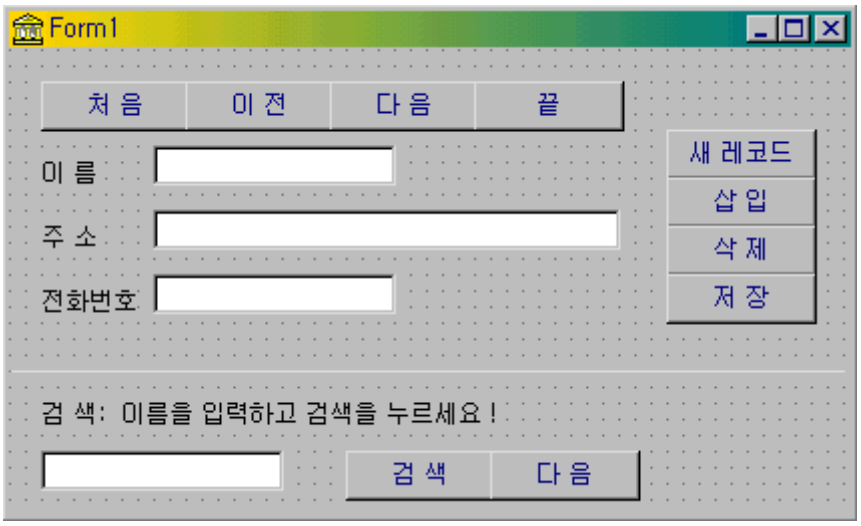
레코드를 삭제할 때에는 삽입할 때보다는 비교적 간단하게 구현할 수 있다. 삭제 과정의 구현 단계는 다음과 같다.

1. 삭제할 파일 포인터 위치 뒤에 있는 모든 레코드를 임시 레코드 버퍼에 읽어온다.
2. Truncate 를 사용해서 삭제할 위치에서 파일의 뒤쪽 부분을 모두 절단해서 버린다.
3. 버퍼에 저장된 레코드를 파일에 저장한다.

Truncate 프로시저는 현재 파일 위치에서 뒤쪽에 있는 모든 레코드를 제거하는 역할을 한다. 그리고, 파일 포인터는 파일의 맨 뒤에 위치하게 된다.

간단한 주소록 프로그램 만들기

이제 앞에서 설명한 레코드 파일을 이용해서 간단한 주소록 프로그램을 하나 만들어 보자. 이 프로그램은 새로운 레코드를 추가, 삭제할 수 있고 레코드를 처음부터 끝까지 검색할 수도 있으며, 이름으로 레코드를 검색할 수도 있다.
먼저 폼을 다음 그림과 같이 디자인 한다.



이번 프로그램에서 사용할 레코드에는 이름과 주소, 전화번호 필드로 구성되어 있다. 다음과 같이 TMyRecord, TMove 그리고, 레코드와 파일 변수를 선언한다.

```
type
  TMyRecord = record
    Name: String[20];
    Address: String[100];
    Phone: String[20];
  end;

  TRecMove = (rcFirst, rcLast, rcNext, rcPrev);

var
  Form1: TForm1;
  MyFile: File of TMyRecord;
  MyRecord: TMyRecord;
```

```
const
  MAX = 100;
```

MAX 상수는 레코드 상한선을 정한 것으로 개발자가 마음대로 정하면 된다. 배열의 크기를 잡을 때 사용된다.
먼저 폼이 생성되는 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  AssignFile(MyFile, 'MyFile.DAT');
  if FileExists('MyFile.DAT') then
  begin
    Reset(MyFile);
    if (FileSize(MyFile) > 0) then
    begin
      Seek(MyFile, 0);
      ReadRecord;
      NewRecord := False;
    end
  end
  else
  begin
    Rewrite(MyFile);
    NewRecord := True;
  end;
  bNew.Enabled := True;
  bInsert.Enabled := False;
end;
```

간단히 설명하면 'MyFile.DAT' 파일을 열고, 이미 존재한 파일이면 첫번째 레코드를 ReadRecord 라는 메소드를 이용해서 에디트 컨트롤에 읽어온다. 이때 NewRecord 라는 전역 변수를 사용했는데, 이 변수는 현재 에디트 컨트롤에 있는 내용이 새로운 내용인지를 가리키는 변수이다. ReadRecord 를 한 후에는 에디트 컨트롤의 내용이 레코드 파일에서 읽어온 내용이므로 NewRecord 를 False 로 설정한다.
그러나, 파일을 처음 생성하는 경우에는 레코드를 읽어오지 않으므로 NewRecord 를 True 로 설정한다.

폼의 OnClose 이벤트 핸들러에서는 사용한 파일을 닫아야 한다.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  CloseFile(MyFile);
end;
```

그러면 레코드 파일의 내용을 에디트 컨트롤로 읽어오는 ReadRecord 메소드와 에디트 컨트롤의 내용을 레코드 파일에 기록하는 SetRecordValue 메소드를 구현하자.

```
procedure TForm1.ReadRecord;
begin
  Read(MyFile, MyRecord);
  with MyRecord do
  begin
    Edit1.Text := Name;
    Edit2.Text := Address;
    Edit3.Text := Phone;
  end;
  Seek(MyFile, FilePos(MyFile) - 1);
end;

procedure TForm1.SetRecordValue;
begin
  with MyRecord do
  begin
    Name := Edit1.Text;
    Address := Edit2.Text;
    Phone := Edit3.Text;
  end;
end;
```

앞에서 설명했던 내용이므로 그다지 이해하기 어렵지는 않을 것이다.
이제 이동 버튼에 대한 작업을 시작하자. 제일 윗 줄의 버튼들이 이동 버튼인데 '처음', '이전', '다음', '끝' 버튼을 각각 bFirst, bPrevious, bNext, bLast 라고 명명한다. 실제로는 MoveToRecord 라는 메소드를 이용하는데, 이때 앞에서 선언한 TRecMove 형의 값을 파

라미터로 호출하여 사용한다. 이들 버튼의 OnClick 이벤트 핸들러를 다음과 같이 코딩한다.

```
procedure TForm1.bFirstClick(Sender: TObject);
begin
  MoveToRecord(rcFirst);
end;

procedure TForm1.bPreviousClick(Sender: TObject);
begin
  MoveToRecord(rcPrev);
end;

procedure TForm1.bNextClick(Sender: TObject);
begin
  MoveToRecord(rcNext);
end;

procedure TForm1.bLastClick(Sender: TObject);
begin
  MoveToRecord(rcLast);
end;
```

그러면, 실제로 레코드 이동을 담당하는 MoveToRecord 를 다음과 같이 구현한다.

```
procedure TForm1.MoveToRecord(Direction: TRecMove);
var
  Pos: Integer;
begin
  EnableButtons(True);
  Pos := FilePos(MyFile);
  if (FileSize(MyFile) = 0) then Exit;
  case Direction of
    rcFirst: Pos := 0;
    rcLast: Pos := FileSize(MyFile) - 1;
    rcNext:
      if (FilePos(MyFile) < (FileSize(MyFile) - 1)) then
```

```
        Pos := FilePos(MyFile) + 1
      else Exit;
    rcPrev:
      if (FilePos(MyFile) > 0) then Pos := FilePos(MyFile) - 1
      else Exit;
  end;
  Seek(MyFile, Pos);
  ReadRecord;
  NewRecord := False;
end;
```

다소 길어 보이지만, 4 가지 방향을 case 문을 이용해서 Seek 프로시저를 이용해서 이동한다. 이때 '이전' 이나 '다음' 버튼의 경우에는 파일의 제일 처음과 끝에 있는지에 대해서 검사하고, 그렇지 않은 경우 위치를 1 증감한다.
처음에 호출하는 EnableButtons 메소드는 새로운 레코드를 작성하거나, 여러가지 변동이 있을 때 버튼의 상태를 결정하기 위한 메소드이다. 이 메소드는 다음과 같이 구현한다.

```
procedure TForm1.EnableButtons(Enable: Boolean);
begin
  bNew.Enabled := Enable;
  bFirst.Enabled := Enable;
  bPrevious.Enabled := Enable;
  bNext.Enabled := Enable;
  bLast.Enabled := Enable;
  bInsert.Enabled := not Enable;
end;
```

간단히 설명하면 이동 버튼 들과 '새 레코드' 버튼은 같게 설정하고, '삽입' 버튼은 반대로 설정한다. 이는 레코드를 삽입할 때에는 반드시 '새 레코드'를 눌러서 '삽입' 버튼이 사용가능할 때에만 가능하도록 한 것이다. 이렇게 새 레코드를 눌렀을 때에는 레코드의 이동을 하지 못하도록 한다.
그러면, '새 레코드' 버튼을 눌렀을 때 호출하는 CreateNewRecord 메소드를 구현하자.

```
procedure TForm1.CreateNewRecord;
var
  i: Integer;
```

```
begin
  if NewRecord then
  LockWindowUpdate(Handle);
  for i := 0 to ComponentCount - 1 do
    if (Components[i] is TEdit) then
      TEdit(Components[i]).Clear;
  LockWindowUpdate(0);
  NewRecord := True;
  EnableButtons(False);
end;

procedure TForm1.bNewClick(Sender: TObject);
begin
  CreateNewRecord;
end;
```

버튼의 이벤트 핸들러는 단순히 앞의 CreateNewRecord 메소드를 호출한다.
CreateNewRecord 메소드는 먼저 NewRecord 변수를 먼저 검사해서, 이 값이 True 이면 LockWindowUpdate 를 호출해서 에디트 컨트롤의 내용이 변하는 것을 막고 이들의 내용을 지운다.
그리고, EnableButton(False)를 호출해서 '삽입' 버튼이 사용 가능하도록 하고, 레코드 이동 버튼 들과 '새 레코드' 버튼을 사용할 수 없게 한다.
이번에는 이렇게 에디트 컨트롤에 기록된 내용을 실제로 저장, 삽입하는 부분을 구현하자.
'저장', '삽입' 버튼의 이름을 각각 bPost, bInsert 로 명명한다. '저장'의 경우는 현재의 파일 포인터에 있는 레코드의 값을 변경하는 것이므로 쉽게 구현이 되지만, '삽입'의 경우에는 앞에서 설명한 레코드 삽입의 구현 단계에 맞추어 구현한다. 앞의 설명을 자세히 읽어보면 코드를 이해하는데 커다란 어려움이 없을 것이다.

```
procedure TForm1.UpdateRecord;
var
  CurPos: Integer;
begin
  CurPos := FilePos(MyFile);
  SetRecordValue;
  if NewRecord then
  begin
```

```
    Seek(MyFile, FileSize(MyFile));
    CurPos := FileSize(MyFile) + 1;
  end;
  Write(MyFile, MyRecord);
  if (FileSize(MyFile) > 0) then
  begin
    Seek(MyFile, CurPos);
    NewRecord := False;
  end;
  EnableButtons(True);
end;

procedure TForm1.InsertRecord;
var
  CurPos, RecordNo, i: Integer;
  Buffer: array[0..MAX] of TMyRecord;
begin
  SetRecordValue;
  CurPos := FilePos(MyFile);
  RecordNo := FileSize(MyFile);
  if FilePos(MyFile) > 0 then
  begin
    i := 0;
    Seek(MyFile, 0);
    while FilePos(MyFile) < CurPos do
    begin
      Read(MyFile, Buffer[i]);
      Inc(i);
    end;
  end;
  Buffer[CurPos] := MyRecord;
  i := CurPos + 1;
  while not EOF(MyFile) do
  begin
    Read(MyFile, Buffer[i]);
    Inc(i);
```

```
  end;
  i := 0;
  Seek(MyFile, 0);
  while (i <= RecordNo) do begin
    Write(MyFile, Buffer[i]);
    Inc(i);
  end;
  Seek(MyFile, CurPos);
  ReadRecord;
  EnableButtons(True);
end;

procedure TForm1.bPostClick(Sender: TObject);
begin
  UpdateRecord;
end;

procedure TForm1.bInsertClick(Sender: TObject);
begin
  InsertRecord;
end;
```

레코드를 삭제하는 버튼인 '삭제' 버튼은 bDelete 로 명명한다. 레코드의 삭제를 구현하는 단계에 대해서도 앞에서 설명했으므로, 다음의 코드를 이해하는 것이 그다지 어렵지 않을 것이다.

```
procedure TForm1.DeleteRecord;
var
  CurPos, RecordNo, i: Integer;
  Buffer: array[0..MAX] of TMyRecord;
begin
  if MessageDlg('정말 지우시렵니까 ?', mtConfirmation, [mbYes, mbNo], 0) = mrNo
    then Exit;
  if NewRecord then
  begin
    ReadRecord;
```

```
    NewRecord := False;
    EnableButtons(True);
    Exit;
  end;
  CurPos := FilePos(MyFile);
  RecordNo := FileSize(MyFile) - CurPos - 1;
  if (FilePos(MyFile) < (FileSize(MyFile) - 1)) then
  begin
    Seek(MyFile, FilePos(MyFile) + 1);
    i := 0;
    while not EOF(MyFile) do
    begin
      Read(MyFile, Buffer[i]);
      Inc(i);
    end;
    Seek(MyFile, CurPos);
    Truncate(MyFile);
    for i := 0 to RecordNo - 1 do
      Write(MyFile, Buffer[i]);
  end
  else
  begin
    Truncate(MyFile);
    Dec(CurPos);
  end;
  Seek(MyFile, CurPos);
  ReadRecord;
end;

procedure TForm1.bDeleteClick(Sender: TObject);
begin
  DeleteRecord;
end;
```

마지막으로 검색 부분을 구현하자. 검색 버튼으로 '찾기'와 '다음' 버튼이 있다. 즉, 이름이 일치하는 첫번째 레코드를 찾을 때에는 '찾기'로 검색하고, 그 위치 이후에 계속 일치하

는 것을 찾을 때에 '다음' 버튼을 사용한다. 이들 버튼의 이름을 각각 bFind, bFindNext 라 명명하자. 그리고, 이들의 구현은 실제로 검색하는 방법이 동일하므로 LocateRecord 라는 동일한 메소드를 사용하되, 파라미터로 검색할 문자열과 '찾기' 버튼의 경우에는 파일의 처음부터, '다음' 버튼의 경우에는 현재 파일 포인터 이후부터 검색하도록 구현한다. 이 메소드는 다음과 같이 구현한다.

```
procedure TForm1.LocateRecord(Value: String; FromBOF: Boolean);
var
  CurPos, SearchPos: Integer;
  Found: Boolean;
begin
  CurPos := FilePos(MyFile);
  if FromBOF then SearchPos := 0
  else SearchPos := CurPos + 1;
  Found := False;
  while (SearchPos <= (FileSize(MyFile) - 1)) and (not Found) do
  begin
    Seek(MyFile, SearchPos);
    Read(MyFile, MyRecord);
    if MyRecord.Name = Value then
    begin
      Found := True;
      Seek(MyFile, SearchPos);
      ReadRecord;
    end;
    Inc(SearchPos);
  end;
  if not Found then ShowMessage('해당 레코드가 없습니다.');
end;

procedure TForm1.bFindClick(Sender: TObject);
begin
  if (Edit4.Text <> '') then
  begin
    if NewRecord then bPostClick(Self);
    LocateRecord(Edit4.Text, True);
```

```
  end;
end;

procedure TForm1.bFindNextClick(Sender: TObject);
begin
  LocateRecord(Edit4.Text, False);
end;
```

검색하는 방법은 소스를 보면 쉽게 이해할 수 있을 것이다. 파라미터에 따라 파일의 첫번째 레코드 또는 현재 레코드부터 순차적으로 검색해 나가는 방법이며, 파일 마지막에 도달하면 메시지 박스를 띄우고 종료하게 된다.
이로써 레코드 파일을 이용한 예제 프로그램이 완성되었다. 직접 사용해보면 알겠지만, 코딩하고 신경써야할 부분이 다소 많기는 해도 나름대로 빠르고 그럴 듯한 어플리케이션을 제작할 수 있다.
물론 이런 형태의 데이터베이스 어플리케이션은 BDE 를 사용하기 보다는, 클라이언트 데이터 세트나 사용자 정의 데이터 세트로 플랫 파일을 이용해서 접근하면 더욱 견고하고 관리가 편하게 만들 수 있다. 하지만, 이렇게 레코드 파일을 적절히 이용하는 것도 괜찮을 것이다.

오브젝트 파스칼의 파일 관련 루틴

오브젝트 파스칼의 런타임 라이브러리에는 흔히 쓰이는 파일 처리 루틴을 포함하고 있다. 이 때 파일 이름을 사용하거나, 파일의 핸들을 사용하게 된다. 기본적인 루틴은 앞에서 많이 설명했으므로, 유틸리티 루틴에 대해서 알아보도록 하자.

- 파일의 삭제

파일을 삭제하는 것은 비교적 간단하다. 단순히 DeleteFile 함수를 사용하면 되는데, Win32 의 SHFileOperation 기능처럼 복구는 지원하지 않는다. 코드는 다음과 같이 하면 된다.

```
DeleteFile(FileName);
```

DeleteFile 루틴은 파일을 삭제했으면 True 를 파일이 읽기 전용이거나, 존재하지 않아서 파일을 삭제하지 못했으면 False 를 반환한다.

● 파일 찾기

파일을 검색할 때에는 FindFirst, FindNext, FindClose 의 3 가지 루틴이 사용된다. FindFirst 는 지정된 디렉토리의 검색 조건에 맞는 첫번째 파일 이름을 가져오며, FindNext 는 일단 한번 검색된 이후에 맞는 조건의 파일을 가져온다. FindClose 는 FindFirst 에 의해 할당된 메모리를 해제한다. 파일이 존재하는지 알고 싶을 때에는 레코드 파일의 예제에서도 사용한 바 있는 FileExists 함수를 사용하는데, 이 함수는 파일이 있을 경우 True, 없을 경우 False 를 반환한다.
이 루틴 들은 TSearchRec 데이터 형의 파라미터를 가지는데, TSearchRec 구조체는 다음과 같은 파일 정보를 담고 있다.

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer;                  //파일에 기록된 시간
    Size: Integer;                  //파일의 크기 (byte)
    Attr: Integer;                  //파일의 속성
    Name: TFileName;                //DOS 파일 이름과 확장자
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;       //파일 생성시간, 마지막 접근 시간, 긴 파일 이름
  end;
```

파일을 찾으면, TSearchRec 파라미터의 필드가 찾은 파일의 내용으로 채워지게 된다.

● 파일의 속성 바꾸기

모든 파일은 비트맵 플래그로 다양한 파일의 속성을 가지고 있다. 파일의 속성을 바꾸기 위해서는 속성을 읽고, 변경하고, 설정하는 3 가지 단계를 거치게 된다.

1. 파일 속성 읽기:

운영체제는 파일 속성을 비트맵의 형태로 저장한다. 이런 파일 속성을 읽어오기 위해서는 FileGetAttr 함수를 이용한다. 이 함수에 파일 이름을 지정하면, 그 파일의 속성이 반환된다. 반환값은 Word 형으로 TSearchRec 구조체에 정의되어 있는 Attr 필드에서 사용될 수 있는 여러 가지 종류의 상수값과 and 연산을 해서 알아낼 수 있다. 이 값이 –1 이면 에

러가 발생한 것이다.

2. 파일 속성 변경

델파이에서 파일 속성은 세트로 나타난다. 그러므로, 4 장에서 설명한 것처럼 여러가지 논리 연산을 통해서 개별적 속성을 다룰 수 있다. 각 속성의 상수값은 SysUtils.pas 유닛에 정의되어 있으며, 도움말을 찾아보면 알 수 있다. 예를 들어, 파일의 읽기 전용 속성을 설정하려면 다음과 같이 하면 된다.

```
Attributes := Attributes or faReadOnly;
```

세트 데이터 형의 특성을 이용하여 한 번에 여러 개의 속성을 설정할 수도 있다. 예를 들어, 다음의 코드는 시스템 파일과 파일 숨김 속성을 제거한다.

```
Attributes := Attributes and not (faSysFile or faHidden);
```

3. 파일 속성 설정

이렇게 변경된 파일 속성은 FileSetAttr 함수를 이용해서 실제로 반영하도록 해야 한다. 이 함수에 파일의 이름을 설정하고, 속성을 넘겨주면 된다.

파일 스트림의 활용

TFileStream 은 고수준의 파일을 다루기 위해서 사용되는 VCL 클래스이다.
TFileStream 은 TStream 클래스에서 상속받으며, 그렇기 때문에 자동으로 지속성(persistence)을 지원한다. 스트림 클래스는 TFiler, TReader, TWriter 클래스를 가지고 작업할 수 있으며, 같은 코드로 VCL 스트림을 지원한다.
TFileStream 은 다른 스트림 클래스와 쉽게 상효작용할 수 있다. 예를 들어, 동적 메모리 블록을 디스크에 덤프하고자 하면, TFileStream 과 TMemoryStream 을 사용하면 된다.

- 스트림이란 ?

스트림이란 바이트 들의 연속이라고 말할 수 있다. 연속된 바이트 내에서 임의의 위치에 접근하여 이를 읽어들이거나 또는 임의의 바이트를 기록할 수 있다. 대개의 경우 데이터는 바이트나 문자열이지만 그 외의 모든 데이터 형의 데이터를 읽고, 쓰기위한 스트림 클래스도 작성할 수 있다. 스트림 클래스는 추상적 클래스인 TStream 으로부터 계승된다.

TStream 객체는 하나의 스트림을 현재 위치(position)를 기억하고 있는 바이트의 연속으로 취급한다. 스트림에서 위치란 입출력이 발생되는 곳을 의미하는데, 이 때 입출력이란 현재의 위치에서 다음 바이트 또는 연속된 바이트를 읽거나 쓰는 것을 의미한다.
TStream 클래스는 추상적 클래스일 뿐이므로 스트림의 프로토콜을 정의할 뿐이다. 파일을 읽고 쓸 수 있는 스트림을 생성하기 위해서는 TFileStream 을 이용해야 한다.

- 파일의 생성과 열기

파일을 생성하고, 이를 열어서 파일의 핸들을 얻으려면, TFileStream 객체의 인스턴스를 생성하기만 하면 된다. 파일이 열리지 않으면, TFileStream 은 예외를 발생시킨다.
생성자의 선언부는 다음과 같다.

```
constructor Create(const Filename: string; Mode: Word);
```

Mode 파라미터는 파일 스트림을 생성할 때 파일을 어떤 식으로 접근할 것인지를 결정한다. Mode 파라미터에는 다음과 같은 것이 있다.

값	의 미
fmCreate	주어진 파일을 생성한다. 파일이 존재하면, 파일을 write 모드로 연다.
fmOpenRead	파일을 읽기 전용으로 연다.
fmOpenWrite	파일을 쓰기 전용으로 연다. 내용을 쓰면 현재의 내용을 완전히 대치한다.
fmOpenReadWrite	파일의 내용을 수정하고자 할 때 많이 사용하는 열기 모드이다.
fmShareCompat	FCB 가 열리는 방법으로 공유 방법을 결정
fmShareExclusive	다른 어플리케이션은 열린 파일에 접근할 수 없다.
fmShareDenyWrite	다른 어플리케이션이 읽기 위해 파일을 열 수 있으나, 쓰지는 못한다.
fmShareDenyRead	다른 어플리케이션이 쓰기 위해 파일을 열 수 있으나, 읽지는 못한다.
fmShareDenyNone	다른 어플리케이션이 마음대로 파일을 읽고 쓸 수 있다.

- 파일 핸들

TFileStream 객체를 인스턴스화하면 파일 핸들에 접근할 수 있게 되는데, 파일 핸들은 Handle 프로퍼티에서 얻을 수 있다. 핸들은 읽기 전용이다. 참고로, 파일 핸들의 속성을 바꾸고자 한다면, 새로운 파일 스트림 객체를 생성해야 한다. 윈도우 API 함수를 사용할 때에는 핸들을 파라미터로 사용할 경우가 많이 있다.

- 파일 읽기와 쓰기

TFileStream 은 파일에 데이터를 읽고, 쓰는데 몇가지 다른 메소드를 제공한다. 이들을 특징에 따라 구분하면 기록하거나 읽은 바이트 수를 반환하는지 ?, 에러가 발생할 때 예외를 발생시키는지 ?, 기록하거나 읽을 바이트 수를 요구하는지 ? 등을 생각할 수 있다.
Read 메소드는 Count 파라미터에 지정한 바이트 수만큼 현재의 위치에서 버퍼로 읽어들이는 메소드이다. 파일을 읽고 나면, 읽은 바이트 수만큼 현재 위치를 이동하게 되며 실제로 전송된 바이트 수를 반환한다. 이 메소드의 선언부는 다음과 같다.

```
function Read(var Buffer; Count: Longint): Longint; override;
```

Read 메소드는 파일의 크기를 모를 때 유용하다.
Write 메소드는 Count 파라미터에 지정한 바이트 수 만큼 버퍼에서, 파일 스트림의 현재 위치부터 기록을 한다. 그리고, 실제로 기록한 바이트 수를 반환한다. 이 메소드의 선언부는 다음과 같다.

```
function Write(const Buffer; Count: Longint): Longint; override;
```

ReadBuffer 와 WriteBuffer 메소드는 Read, Write 메소드와는 달리 읽거나, 기록한 바이트 수를 반환하지 않는다. 이들 프로시저는 구조체를 읽을 때와 같이 읽거나 쓸 데이터의 바이트 수를 정확히 알 때 많이 사용한다. 또한, ReadBuffer 와 WriteBuffer 메소드는 에러가 발생할 경우 EReadError, EWriteError 예외를 발생시킨다. 이들 메소드의 선언부는 다음과 같다.

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

- 문자열 읽기와 쓰기

Read, Write 함수에서 문자열을 사용하려면 문법적으로 다소 고려해야 할 부분들이 있다. Read 메소드의 Buffer 파라미터는 var 로 선언되어 있고, Write 메소드의 Buffer 파라미터는 const 로 선언되어 있다. 이들은 데이터 형이 정해져 있지 않기 때문에, 루틴은 변수의 주소를 사용하게 된다. 가장 흔히 사용하게 되는 데이터 형은 긴 문자열인데, 긴 문자열에는 문자열의 길이와 참조 계수 등의 내용이 담겨 있기 때문에 단순히 역참조(dereference)로 해결이 되지 않는다. 제대로 사용하려면 일단 문자열을 포인터나 PChar 로 형변환을 하고나서 역참조를 해야 한다. 예제 코드를 소개하면 다음과 같다.

```
procedure UsingStrings;
var
  FS: TFileStream;
  S: string = 'Hello';
begin
  FS := TFileStream.Create('foo', fmCreate or fmOpenRead);
  FS.Write(s, Length(s));                    //이렇게 하면 쓰레기 값이 저장된다.
  FS.Write(PChar(s)^, Length(s));            //이 방법이 옳다.
end;
```

- 파일 찾기

TFileStream 은 Seek 메소드를 통해서 검색 기능을 제공한다. Seek 메소드의 선언부는 다음과 같다.

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

Origin 파라미터를 이용해서 Offset 파라미터를 어떻게 해석할 지를 결정하게 된다. Origin 파라미터가 가질 수 있는 값에는 다음과 같은 것들이 있다.

값	의 미
soFromBeginning	오프셋이 리소스의 처음부터 시작된다. 오프셋은 0 보다 커야 하며, Seek 메소드에 의해 Offset 파라미터에 지정된 위치로 이동한다.
soFromCurrent	오프셋이 리소스의 현재 위치에서 시작된다. Seek 메소드에 의해 현재 위치 + Offset 파라미터의 위치로 이동한다.
soFromEnd	오프셋이 리소스의 끝에서 시작된다. 오프셋은 0 보다 작아야 하며, 파일의 끝에서 오프셋에 지정된 바이트 수만큼 이전의 위치에서 시작된다.

Seek 메소드는 스트림의 현재 위치를 재설정하고, Position 프로퍼티의 새로운 값을 반환하게 된다.

- 파일 위치와 크기

TFileStream 은 파일의 현재 위치와 크기를 Position, Size 프로퍼티에서 얻을 수 있다. 이 프로퍼티 들은 Seek, Read, Write 메소드에 의해 사용된다. 이들 프로퍼티의 값은 바이트

단위로 나타난다.
Size 프로퍼티를 설정하면 파일의 크기를 변경하게 된다. 파일의 크기가 바뀔 수 없는 경우에 이 프로퍼티를 바꾸려고 하면, 예외가 발생한다. 예를 들어, fmOpenRead 모드로 열린 파일의 Size 프로퍼티를 변경하려고 하면 예외가 발생할 것이다.

- 파일 스트림의 복사

파일 스트림을 복사할 때에는 CopyFrom 메소드를 사용한다. 이 메소드의 선언부는 다음과 같다.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

CopyFrom 메소드를 사용하면 사용자가 간단하게 데이터를 사용할 수 있다. 이 메소드는 Source 파라미터에 지정된 스트림에서 Count 바이트 만큼 복사해 오는데, 실제로 복사한 바이트 수를 반환한다. Count 파라미터를 0 으로 설정하면 Source 의 전체 내용을 복사해 온다. Count 값이 0 이 아니면 Source 스트림의 현재 위치에서 지정한 수만큼 복사하게 된다.

- 파일 스트림을 이용한 파일 복사

그러면, 지금까지 설명한 파일 스트림을 이용해서 파일을 복사하는 루틴을 소개한다. 이 루틴을 살펴보면 구체적인 파일 스트림의 사용 방법을 알 수 있을 것이다.

```
procedure FileCopy(const SrcFile, DestFile: String);
const
  BufSize = 8192;
var
  SrcStream, DestStream: TFileStream;
  Buffer: Pointer;
  ReadCount: Integer;
begin
  SrcStream := TFileStream.Create(SrcFile, fmOpenRead or fmShareDenyWrite);
                    //읽기 전용으로 소스 파일을 연다.
  try
    DestStream := TFileStream.Create(DestFile, fmCreate or fmShareExclusive);
                    //기록할 파일이 존재하지 않으면 새로 생성하고, 존재하면 덮어 쓴다.
```

```
                    //그리고, 다른 사용자는 이 파일에 접근하지 못한다.
    try
      GetMem(Buffer, BufSize);
      try
        repeat
          ReadCount := SrcStream.Read(Buffer^, BufSize); //BufSize 만큼 읽는다.
          DestStream.WriteBuffer(Buffer^, ReadCount);     //버퍼의 내용을 읽은 만큼 기록한다.
        until ReadCount = 0;                              //소스 파일의 끝까지 읽었음
      finally
        FreeMem(Buffer, BufSize);
      end;
    finally
      DestStream.Free;
    end;
  finally
    SrcStream.Free;
  end;
end;
```

이 파일 복사 루틴은 소스 파일을 읽을 때에는 Read 메소드를 사용하였다. 이 메소드를 이용하면 실제로 읽어들인 바이트 수를 알 수 있기 때문에 이를 사용해야 한다. 대부분의 파일의 크기가 버퍼의 크기(8192)의 배수가 아니기 때문에, Read 메소드를 이용해서 실제로 읽은 바이트 수를 알아야 한다. 그리고, 기록을 할 때에는 Write, WriteBuffer 중 아무거나 사용해도 된다.
물론 이 파일 복사 루틴은 앞에서 설명한 CopyFrom 메소드를 이용하면 훨씬 더 짧고 간단하게 작성할 수 있다. 그렇지만, 지금까지 설명한 것을 예제로 하기에는 너무 단순해서 조금 복잡한 방법을 소개하였다. CopyFrom 메소드를 이용하면 다음과 같이 간단하게 작성이 가능하다.

```
procedure FileCopy(const SrcFile, DestFile: string);
var
  SrcStream, DestStream: TFileStream;
begin
  SrcStream := TFileStream.Create(SrcFile, fmOpenRead);
  try
    DestStream := TFileStream.Create(DestFile, fmCreate);
```

```
    try
      DestStream.CopyFrom(SrcStream, 0);
    finally
      DestStream.Free
    end;
  finally
    SrcStream.Free;
  end;
end;
```

Win32 Shell API 의 활용

Win32 기반의 운영체제인 윈도우 95/98/NT 4.0 에서는 기본적으로 파일을 다룰 때 과거의 도스에서 쓰던 디렉토리와 파일의 방식으로 다루는 것이 아니라, 네트워크 환경과 컴퓨터에서 접근할 수 있는 모든 논리적인 객체를 가상 폴더(virtual folder)를 중심으로 접근하도록 변경되었다.
윈도우 3.1 을 쓰다가 윈도우 95 의 탐색기를 써보면, 이 차이를 잘 느낄 수 있을 것이다. 윈도우 95 의 탐색기에는 파일이 아닌 '내 컴퓨터'나 '휴지통'과 같은 가상 폴더에 파일에 접근하는 것과 같은 방식으로 접근하게 된다.
그렇기 때문에, 윈도우 95 는 과거에 도스에서 쓰던 방식으로 파일에 접근하지 않고 Win32 에서 제공되는 쉘의 파일 처리 API 를 이용하여 이들을 처리하게 된다.
Win32 Shell 에 관한 내용은 방대하면서도 복잡하지만, 여기서는 파일 처리에 관한 부분 만을 다루도록 한다.

- SHFileOpStruct 구조체와 SHFileOperation API

ShellAPI.pas 유닛에 정의되어 있는 SHFileOpStruct 구조체와 SHFileOperation API 함수를 사용하면 파일을 삭제할 때 휴지통의 사용을 지정할 수도 있으며, 그 밖에 복사, 이동, 이름 변경 등의 여러가지 조작을 가할 수 있다. 다음의 프로시저는 파일을 삭제하는 예이다.

```
procedure DeleteFiles;
var
  T: TSHFileOpStruct;
  X: Integer;
begin
```

```
  with T do
  begin
    Wnd := 0;
    wFunc := FO_DELETE;
    pFrom := 'E:₩Temp₩Test₩*.*';
    fFlags := FOF_ALLOWUNDO or FOF_FILESONLY or FOF_SILENT or
          FOF_NOCONFIRMATION;
  end;
  SHFileOperation(T);
end;
```

이 프로시저는 E:₩Temp₩Test 디렉토리에 있는 모든 파일을 지운다. 이때 지정한 플래그에 따라 동작을 달리 하는데, FOF_ALLOWUNDO 플래그를 지정 했으므로 휴지통에 파일을 버리게 되며 FOF_NOCONFIRMATION 플래그는 파일을 지울 때 확인 대화상자를 띄우지 않는 것을 의미한다. 또한, FOF_SILENT 플래그를 지정 했으므로 파일을 삭제할 때 파일이 삭제되는 진행 상황을 나타내는 애니메이션을 보여주지 않는다.
TSHFileOpStruct 구조체를 사용할 때에는 몇 가지 주의할 사항이 있다.
우선 파일의 패스를 지정할 때에는 항상 완전한 패스 이름을 적어주어야 한다. 현재의 디렉토리를 이용해서 작업을 할 경우 보통 때에는 문제가 없으나, FOF_ALLOWUNDO 플래그를 지정해서 파일을 휴지통에 버릴 경우, 이를 복구할 때 원래의 디렉토리를 제대로 찾지 못하는 문제가 생기게 된다.
또한, pFrom 멤버에 파일을 여러 개 지정할 때에는 각 파일 이름 사이는 #0(Null) 문자로 분리되어야 하며, 마지막 파일 이름 뒤에는 Null 문자를 2 개 연달아 배치해야 한다. 그러므로, 델파이에서는 아래와 같은 식으로 문자열을 조작해줄 필요가 있다.

```
var
  FileList: String;
  FOS: TSHFileOpStruct;
begin
  FileList := 'c:₩delete.met' + #0 + 'c:₩windows₩temp.$$$'+ #0 + #0;
  FileList := Filename1 + #0 + Filename2 + #0#0;
  FOS.pFrom := PChar(FileList);
end;
```

- 시스템에서 정의한 아이콘 나타내기

이 작업을 하려면 SHGetFileInfo 함수와 TSHFileInfo 구조체를 이해해야 한다. 이들은 ShellAPI.pas 유닛에 선언되어 있다.
SHGetFileInfo API 함수는 파일의 패스를 지정해주면 그 파일에 대한 속성을 TSHFileInfo 구조체에 담아서 돌려 준다. 선언부는 다음과 같다.

```
function SHGetFileInfo(pszPath: PAnsiChar; dwFileAttributes: DWORD;
  var psfi: TSHFileInfo; cbFileInfo, uFlags: UINT): DWORD; stdcall;
```

파라미터를 설명하면, pszPath 파라미터에는 알아볼 파일의 패스를, dwFileAttributes 파라미터는 잘 사용하지 않으며, psfi 파라미터에는 파일의 정보를 담아올 구조체의 변수를 담아야 하며, cbFileInfo 와 uFlags 파라미터에는 각각 구조체의 크기와 플래그를 지정한다.
이렇게 호출하고 나면, TSHFileInfo type 의 구조체 변수에 파일에 대한 정보가 담기게 되는데, 이 구조체의 선언부는 다음과 같다.

```
TSHFileInfoA = record
    hIcon: HICON;                                    //아이콘의 핸들
    iIcon: Integer;                                  //아이콘의 인덱스
    dwAttributes: DWORD;                             //파일의 속성 플래그
    szDisplayName: array [0..MAX_PATH-1] of  AnsiChar;   //디스플레이 이름이나 패스
    szTypeName: array [0..79] of AnsiChar;           //type 의 이름
  end;
```

즉, hIcon 멤버에는 파일을 대표하는 아이콘의 핸들, iIcon 에는 아이콘의 인덱스가 들어있게 된다.
그럼 해당하는 디렉토리의 파일들을 보여줄 때에는 이들의 아이콘을 TImageList 에 담아서 보여주면 된다. 다음 코드를 살펴 보자.

```
Images := TImageList.Create(Self);
Images.ShareImages := True;
Images.DrawingStyle := dsTransparent;

var
  sfi: TSHFileInfo;
  ...

Images.Handle := SHGetFileInfo(Path, 0, sfi, SizeOF(sfi), SHGFI_SYSICONINDEX
```

```
        + SHGFI_SMALLICON);
Images.Draw(Canvas, 0, 0, sfi.iIcon);
```

TImageList 컴포넌트의 ShareImages 프로퍼티는 TImageList 컴포넌트가 파괴될 때, 이미지의 핸들도 같이 해제할 것인지 여부를 결정짓는 프로퍼티이다. 이 값을 True 로 설정하면 TImageList 컴포넌트가 파괴되어도 핸들이 해제되지 않는다.
DrawingStyle 프로퍼티는 이미지를 그리는 스타일을 지정하는 것으로 dsFocus, dsSelected, dsNormal, dsTransparent 가 있다. 각각의 값은 다음의 의미를 가진다.

dsFocused	시스템의 highlight 색상과 25% 섞어서 이미지를 나타낸다. 이 값은 마스크를 가진 이미지 리스트에만 영향을 미친다.
dsSelected	시스템의 highlight 색상과 50% 섞어서 이미지를 나타낸다. 역시 마스크를 가진 이미지 리스트에만 영향을 미친다.
dsNormal	BkColor 프로퍼티에 지정된 색상을 이용해서 이미지를 그린다. BkColor 프로퍼티에 clNone 이 지정되어 있으면 dsTransparent 와 같게 동작한다.
dsTransparent	BkColor 설정과 관계 없이 마스크를 이용해서 이미지를 그린다.

인쇄 기능의 추가

보통 델파이를 가지고 어플리케이션을 만들 때, 인쇄가 필요한 경우에는 델파이 3 부터 기본 제공되는 QuickReport 를 사용하게 된다. 그런데, QuickReport 를 사용하지 않고도 직접 인쇄 기능을 수행할 수 있는 방법이 있다.
QuickReport 보다는 사용 방법도 어렵고, 이쁘게 만들기도 어렵지만 직접 인쇄를 제어하는 방법을 익혀 놓으면 좀더 정교하면서도 깨끗한 인쇄를 가능하게 할 수도 있다.

- 인쇄의 기본

가장 원시적인 방법으로 인쇄하는 것은 파일 변수를 생성해서 문자열을 인쇄하는 것이 가능하다. 도스 시절의 표준 파스칼에서부터 사용하던 방식이다.
다음의 코드를 살펴 보자.

```
var
  P: TextFile;
begin
  AssignPrn(P);
  ReWrite(P);
```

```
  WriteLn(P, '테스트 입니다.');
  CloseFile(P);
end;
```

여기서 P 라는 변수를 TextFile 로 선언하고 AssignPrn 루틴을 사용하면 프린터 포트를 변수 P 에 대입하게 되며, ReWrite 문을 통해 프린터 포트를 연다. 그리고, WriteLn 프로시저를 사용하면 프린터로 텍스트를 전송할 수 있다. 마지막으로 사용이 끝난 후에는 반드시 CloseFile 을 이용하여 프린터 포트를 닫아주어야 한다.

- TPrinter 객체의 사용

델파이에서 윈도우의 프린터 인터페이스에 접근하기 위해서는 TPrinter 객체를 사용한다. 델파이의 Printer.pas 유닛에는 Printer 변수가 선언되어 있는데, 이것은 TPrinter 클래스로 선언되어 있다. 그러므로, TPrinter 객체를 사용하기 위해서는 Printers.pas 유닛을 uses 절에 포함시키고 사용해야 한다.
TPrinter 객체의 프로퍼티와 메소드에 대해 다음에 정리하였다.

프로퍼티/메소드	설 명
Canvas	TCanvas 클래스로 선언된다. Canvas 는 인쇄되기 전에 메모리에 페이지나 문서를 생성되는 곳이다. Canvas 에는 Pen, Brush 등의 속성을 가지고 있는데, 이들은 그림을 그리거나 텍스트를 추가하는데 사용한다.
TextOut	TCanvas 클래스의 메소드로서, Canvas 에 텍스트를 전달하는 역할을 한다.
BeginDoc	인쇄 작업을 시작할 때 사용한다.
EndDoc	인쇄 작업을 종료할 때 사용한다. EndDoc 이 호출되어야만 실제 인쇄 작업이 시작된다.
PageHeight	페이지 높이를 픽셀 단위의 값으로 반환한다.
PageWidth	페이지 폭을 픽셀 단위의 값으로 반환한다.
NewPage	강제로 새로운 페이지로 이동하도록 하며, Canvas 의 Pen 속성 값을 (0, 0)으로 초기화 시킨다.
PageNumber	현재 인쇄 중인 페이지의 수를 반환한다.
Aborted	읽기 전용으로 이 값이 True 이면 출력이 사용자에 의해 중지된 것이다. 그러므로, 그리기 명령을 오랫동안 수행할 때에는 이 프로퍼티를 자주 확인하여 사용자의 입력에 빠르게 반응해야 한다.
Copies	현재 출력하는 문서의 매수를 지정할 수 있다.
Fonts	읽기 전용으로 프린터가 지원하는 모든 폰트의 리스트이다.
Handle	읽기 전용으로 현재 선택된 프린터의 핸들을 반환한다. 출력 중에는 이 프로퍼티

	의 값이 Canvas.Handle 프로퍼티와 같다. 이 프로퍼티는 다양한 디바이스 핸들을 사용하는 GetDeviceCaps 같은 API 함수들에게 정보를 제공할 때 사용한다.
Orientation	poPortrait 로 설정되면 문서는 일반적인 방향으로 출력되며, poLandscape 로 선택된 경우 문서를 넓게 찍는다.
Printing	읽기 전용으로 현재 출력 중인지 여부를 알려 준다.

TPrinter 클래스를 사용할 때에는 먼저 Title 프로퍼티를 설정하고, BeginDoc 프로시저를 호출한 후 Canvas 프로퍼티에 마음대로 그리고, 한 페이지를 넘는 경우에는 NewPage 를 호출하고 다시 Canvas 프로퍼티에 그리게 된다. 출력이 끝났으면 EndDoc 을 호출하면 된다.
다음의 코드는 첫번째 페이지에는 타원을 하나 그리고, 다음 페이지에는 사각형을 하나 그린다.

```
Printer.Title := '테스트 입니다 !';
Printer.BeginDoc;
Printer.Canvas.Ellipse(50, 50, 200, 200);
Printer.NewPage;
Printer.Canvas.Rectangle(50, 50, 200, 20);
Printer.EndDoc;
```

그러면 간단한 예제로 비트맵 파일을 용지의 한가운데에 인쇄하는 예제를 하나 만들어 보자. 폼을 다음과 같이 버튼 2 개와 이미지 컴포넌트와 TOpenPictureDialog 컴포넌트를 하나씩 올려 놓자.

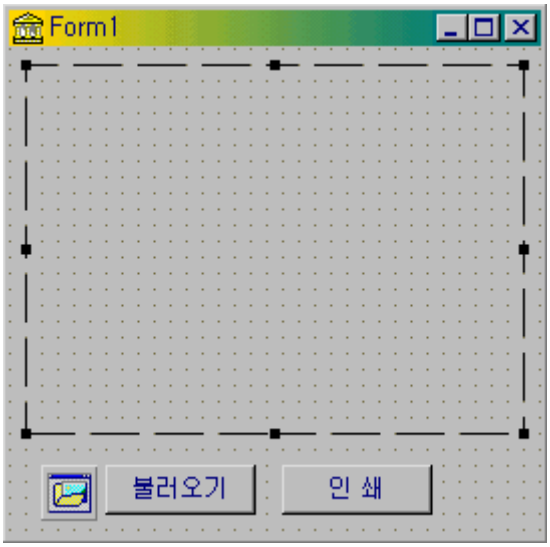


Image1 의 Stretch 프로퍼티를 True 로 설정하여 그림이 크기에 맞도록 한다.
Button1 을 클릭하면 그림을 불러올 수 있도록 다음과 같이 OnClick 이벤트 핸들러를 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

그리고, 실제 인쇄 루틴을 Button2 를 클릭하면 실행해야 하므로 Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  PrnRect, ImgRect: TRect;
  XCenter, YCenter: Integer;
begin
  with Printer do
  begin
    BeginDoc;
    XCenter := PageWidth div 2;
    YCenter := PageHeight div 2;
    PrnRect := Rect(XCenter div 2, YCenter div 2,
      PageWidth - (XCenter div 2), PageHeight - (YCenter div 2));
    ImgRect := Rect(Image1.Left, Image1.Top,
      Image1.Left + Image1.Width, Image1.Top + Image1.Height);
    Canvas.CopyRect(PrnRect, Form1.Canvas, ImgRect);
    EndDoc;
  end;
end;
```

그림을 페이지의 중앙에 위치시키며, 구역을 용지 전체 크기로 설정하여 인쇄하는 루틴이다. 이때 그래픽을 프린터의 캔버스에 위치시키기 위해서 CopyRect 메소드를 사용했는데, 이 메소드를 이용하면 화면의 캔버스 영역을 프린터의 캔버스 영역으로 복사하게 된다.

CopyRect 메소드의 선언부는 다음과 같다.

```
procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);
```

목적지와 원본의 사각형이 TRect 형의 변수로 전달되는데, 이 경우에 원본 캔버스는 Form1 의 캔버스이다. CopyRect 문에서 Image1 이 아닌, 폼의 캔버스로부터 이미지를 복사하도록 하기 위해 ImgRect 의 구역을 Image1 의 전체가 들어가도록 설정하였다.
TRect 데이터 형은 4 개의 값이 필요하며, 4 개의 정수값을 이용하여 TRect 데이터 형으로 변환시킬 때 Rect 함수를 사용한다. Button2 의 OnClick 이벤트 핸들러에서 PrnRect 는 프린터의 캔버스의 구역을 저장하며, ImgRect 는 폼의 캔버스의 구역을 저장한다. 즉, 다음의 코드 들은 PrnRect 는 프린터 페이지의 중앙 1/2 영역을 지정하는 것이며, ImgRect 는 폼의 Image1 컨트롤의 전체 크기를 지정하는 것이다.

```
PrnRect := Rect(XCenter div 2, YCenter div 2,
  PageWidth - (XCenter div 2), PageHeight - (YCenter div 2));
ImgRect := Rect(Image1.Left, Image1.Top,
  Image1.Left + Image1.Width, Image1.Top + Image1.Height);
```

나머지 코드는 그다지 어려운 내용이 아니므로, 설명을 생략하겠다.
이것으로 간단한 이미지 인쇄 프로그램을 작성하였다. 컴파일하고 실행해서 직접 인쇄를 해보도록 하자.

- GetDeviceCaps 함수의 활용

지금까지는 인쇄를 할 때 픽셀 단위로 모든 처리를 해왔다. 이 방법은 프린터의 픽셀 크기가 제각기 다르기 때문에, 어떤 프로그램에서 페이지에 인치나 밀리미터 단위로 그리기를 원하게 된다. 이를 해결하기 위해서는 먼저 GetDeviceCaps 함수를 호출하여, 선택한 유닛당 픽셀의 수를 결정해야 한다. GetDeviceCaps 함수는 다음과 같이 선언되어 있다.

```
function GetDeviceCaps(DC: HDC; Index: Integer): Integer;
```

이 함수는 디바이스 컨텍스트의 핸들과 디바이스 특성 중에서 원하는 자료를 상수로 입력받는다. 그리고, 실제 입력에 해당되는 값을 반환하게 된다. TPrinter 의 경우에는 Handle 이나 Canvas.Handle 과 LOGPIXELSX 와 LOGPIXELSY 를 파라미터로 입력하면 논리적 인치당 픽셀 수가 반환된다. 하지만 화면에서의 논리적 인치라는 개념은 실제 인치보다 조금 큰데, 그 원인은 8 이하의 작은 폰트를 눈에 보이도록 하기 위해 개발된 것이 논리

적 인치이기 때문이다. 그러나, 프린터의 경우에는 논리적 인치와 실제 인치의 차이가 거의 없다.
각각의 축에 대해 상수 값으로 LOGPIXELSX 는 수평, LOGPIXELSY 는 수직을 나타낸다. 스크린에서 두 축은 서로 다른 값을 반환한다. 즉, LOGPIXELSX 로 GetDeviceCaps 를 호출한 경우와 LOGPIXELSY 로 호출한 경우 다른 값이 반환된다. 이는 화면 상에서 수직 방향 단위당 픽셀수와 수평 방향 단위당 픽셀 수가 다르기 때문인데, 프린터의 경우에는 픽셀이 수평, 수직의 크기가 같기 때문에 GetDeviceCaps 함수를 HORSIZE 나 VERTSIZE 로 호출하여도 같은 값이 반환된다.
TPrinter.Canvas 는 일반적인 Canvas 와 다를 것이 없기 때문에, 출력의 미리보기를 매우 쉽게 구현할 수 있다. 그러면, 실제로 GetDeviceCaps 함수를 사용해서 작업을 시작해보자. 먼저 각각의 축에 대한 함수를 다음과 같이 호출해야 한다.

```
XPPI := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
YPPI := GetDeviceCaps(Printer.Handle, LOGPIXELSY);
```

이 값은 프린터의 종류에 따라 다르지만 보통 300~360 정도가 될 것이다. 이 값을 이렇게 변수에 저장해 두었다가, 그리고자 하는 대상을 정확한 크기와 위치에 그릴 수 있다. 다음 코드는 2 인치 크기의 원 하나를 오른쪽으로 2 인치, 아래로 2 인치 떨어진 곳에 그리는 것이다.

```
Printer.BeginDoc;
XRadius := XPPI div 2;
YRadius := YPPI div 2;
XCenter := XPPI * 2 – 1;
YCenter := YPPI * 2 – 1;
Printer.Canvas.Ellipse(XCenter – XRadius, YCenter – YRadius,
  XCenter + XRadius, YCenter + YRadius);
Printer.EndDoc;
```

(0, 0)은 사실 종이의 모서리에서 다소 떨어진 지점이다. 이때 원점이 (0, 0)이기 때문에 XCenter 나 YCenter 같은 위치를 계산할 때 1 씩 뺀 것이다.
그러면, 밀리미터 단위로 계산을 하려면 어떻게 해야 할까 ? 1 인치는 25.4 밀리미터이기 때문에, 1 밀리미터는 XPPI, YPPI 의 25.4 분의 1 이다. 그렇다고, 이를 덥석 25.4 로 나누면 오차가 나게 된다.
이 문제를 해결하려면 XPPI, YPPI 값은 유지하고, 필요할 때마다 스케일해서 사용하는 것이다. 예를 들어, 2cm 지름의 원을 오른쪽으로 10cm, 아래로 10cm 떨어진 곳에 그린다고

하면 다음과 같이 하면 된다.

```
Printer.BeginDoc;
Printer.Canvas.MoveTo(0, 0);
XRadius := MulDiv(XPPI, 100, 254);
YRadius := MulDiv(YPPI, 100, 254);
XCenter := MulDiv(XPPI, 1000, 254) – 1;
YCenter := MulDiv(YPPI, 1000, 254) – 1;
Printer.Canvas.Ellipse(XCenter – XRadius, YCenter – YRadius,
  XCenter + XRadius, YCenter + YRadius);
Printer.EndDoc;
```

참고로, MulDiv 루틴은 처음 두 개의 파라미터를 곱한 값을 마지막 파라미터의 값으로 나누는(정수형 나누기) 루틴이다. 즉, 100 을 곱하고 254 로 나눈 것은 인치를 밀리미터 단위로 변환하기 위한 것이다.

- 메모 인쇄하기

앞에서 간단하게 WriteLn 과 TextOut 을 이용해서 인쇄하는 방법에 대해서 알아보았지만, 보다 쓸모 있게 메모 컴포넌트의 내용을 인쇄하는 프로시저를 소개한다. 이 프로시저는 원래 Peter Below(100113.1101@compuserve.com)에 의해 작성된 프로시저를 바탕으로 하였다. TPrinter 를 이용해서 인쇄하는 방법을 배우기에는 꽤 좋은 예제라고 생각되어 소개하는 것이다.
이 프로시저는 메모의 내용을 메모의 폰트를 이용해서 선택된 프린트로 인쇄하는 역할을 한다. 출력은 수평/수직 방향으로 가운데 정렬을 할 수 있고, 여백은 디폴트 값을 사용하게 할 것이다. 텍스트가 한 페이지가 넘을 경우에는 수직 방향 가운데 정렬은 무시된다.
먼저 사용할 유닛의 uses 절에 Printers.pas 유닛을 추가한다. 그리고, 프로시저의 선언을 해보자. 프로시저의 이름은 PrintMemo 로 하고, 파라미터로 인쇄할 메모, 수평과 수직 방향에서 가운데 정렬을 할 지 여부를 결정하는 논리값을 사용하도록 한다.

```
procedure PrintMemo(aMemo: TMemo; centerVertical, centerHorizontal: Boolean);
```

로컬 변수로는 여백을 나타내는 4 개의 변수와 x, y 축 라인의 좌표를 다용도로 사용할 2 개의 면수와 줄의 최대 길이, 줄수, 줄의 높이를 대표한 변수를 사용한다.

```
var
```

```
  topMargin, bottomMargin, leftMargin, rightMargin: Integer;
  x, y: Integer;
  maxLineLength: Integer;
  linecounter: Integer;
  lineheight : Integer;

begin
```

먼저 디폴트 여백으로 위, 아래로는 1 인치, 좌우로는 0.75 인치를 설정한다. 이때 앞에서도 설명한 GetDeviceCaps API 함수를 이용해서 인치 단위를 사용할 수 있도록 한다.

```
  topMargin := GetDeviceCaps( Printer.Handle, LOGPIXELSY );
  bottomMargin := Printer.PageHeight - topMargin;
  leftMargin := GetDeviceCaps( Printer.handle, LOGPIXELSX ) * 3 div 4;
  rightMargin := Printer.PageWidth - leftMargin;
```

이제 인쇄를 시작한다. 먼저 메모의 폰트를 Printer 의 캔버스 객체에 대입한다.

```
  Printer.BeginDoc;
  try
    Printer.Canvas.Font.PixelsPerInch :=
      GetDeviceCaps( Printer.Canvas.Handle, LOGPIXELSY );
    Printer.Canvas.Font:= aMemo.Font;
```

한 줄의 높이를 결정한다. 보통 'Ay'를 사용하는데, 이는 가장 높은 위치('A')와 낮은 위치('y')를 포함했기 때문이다.

```
    lineHeight := Printer.Canvas.TextHeight('Ay');
```

centerHorizontal 파라미터가 설정됐으면, 일단 메모의 한 줄 전체를 검사해서 가장 긴 줄의 길이를 픽셀로 알아낸다. 이를 이용해서 좌측 여백을 결정하게 된다.

```
    if centerHorizontal Then
    begin
      maxLineLength := 0;
      for linecounter := 0 to aMemo.Lines.Count - 1 do
```

```
    begin
      x := Printer.Canvas.TextWidth(aMemo.Lines[linecounter]);
      if x > maxLineLength then maxLineLength := x;
    end;
    x := leftMargin + (rightMargin – leftMargin – maxLineLength) div 2;
                    //이 줄의 좌측 여백을 결정한다.
    if x < leftMargin then
    begin
                    //줄의 폭이 용지의 폭을 넘는 경우다.  그냥 잘리게 된다.
    end
    else
      leftMargin := x;
  end;
```

centerVertical 파라미터가 설정됐으면, 인쇄할 모든 줄의 공간을 일단 계산하고, 이를 topMargin 을 결정하는데 참고로 한다.

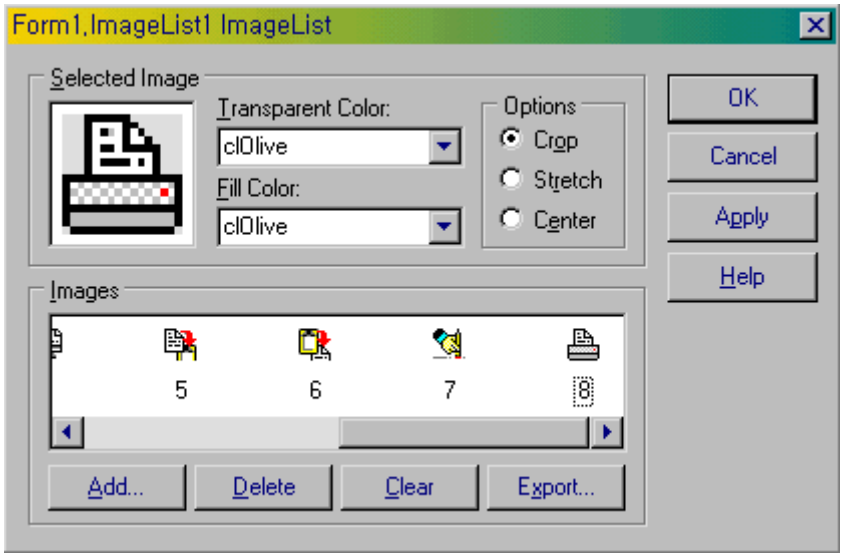
```
  if centerVertical then
  begin
    y := lineHeight * aMemo.Lines.Count;
    if y < (bottomMargin – topMargin) then
      topMargin := topMargin + (bottomMargin – topMargin – y) div 2;
                    //y 가 더 큰 경우에는 페이지 전체 크기보다 내용이 더 많은 경우이다.
                      이때에는 수직 방향의 정렬은 무시하고, 그냥 놔둔다.
  end;
```

여백이 모두 계산 되었으므로, 실제로 인쇄를 하면 된다. 다음 코드에서 x, y 변수는 다음 줄의 시작하는 지점의 좌표를 담고 있다.

```
  x:= leftMargin;
  y:= topMargin;
  for linecounter := 0 to aMemo.Lines.Count-1 do
  begin
    Printer.Canvas.TextOut(x, y, aMemo.Lines[linecounter]);
    y := y + lineHeight;
    if y >= bottomMargin then
```

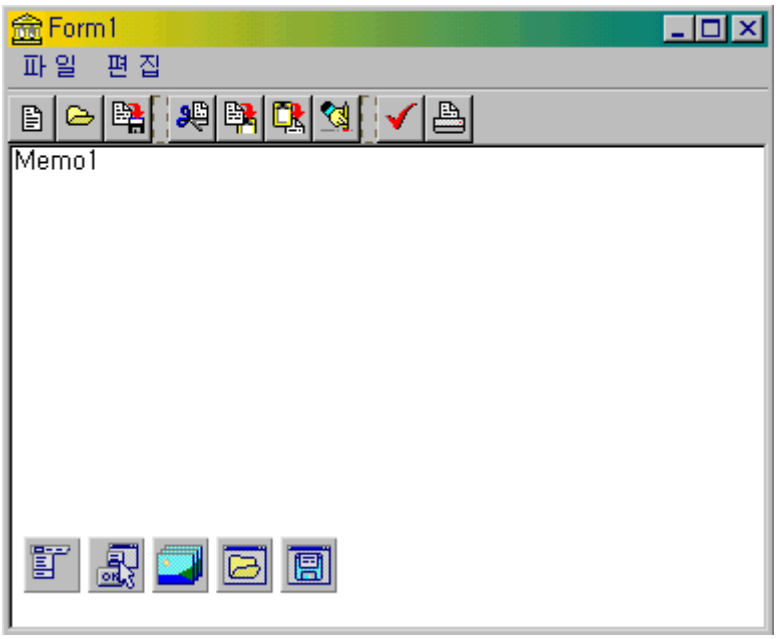
```
        begin
          //페이지 끝이므로, 페이지를 넘긴다.
          if linecounter < (aMemo.Lines.Count - 1) then
          begin
            Printer.NewPage;
            y:= topMargin;
          end;
        end;
      end;
    finally
      Printer.EndDoc;
    end;
end;
```

그럼, 이 프로시저를 이용해서 메모를 인쇄하는 간단한 어플리케이션을 제작해 보자. 여기에서는 8 장에서 제작했던 에디터에 인쇄기능을 추가하기로 하겠다. 먼저 8 장의 소스 디렉토리의 Exam3.dpr 파일을 읽어온 후, 유닛과 dpr 파일을 다른 이름으로 저장하자. 그리고, ImageList1 을 더블 클릭한 후 다음과 같이 인쇄 버튼에 해당되는 비트맵을 추가한다.



그리고, ActionList1 컴포넌트를 더블 클릭한 후, New Action 메뉴를 선택하여 액션 아이템을 하나 추가한다. 이 아이템의 Name 프로퍼티를 PrintFile, Caption 프로퍼티를 '인쇄하기', 그리고 ImageIndex 프로퍼티를 8 로 설정한다. 또한, Toolbar1 컴포넌트를 클릭한 후 오른쪽 버튼을 눌러서 New Button 명령을 선택하면 새로운 툴바 버튼이 추가되는데, 이 버튼의 Action 프로퍼티를 PrintFile 로 설정한다. 마찬가지로 MainMenu1 컴포넌트를 더블

클릭한 후 '파일저장'과 '-' 메뉴 아이템 사이에 커서를 위치시키고 Insert 키를 눌러서 메뉴 아이템을 하나 삽입한 후, 이 아이템의 Action 프로퍼티를 PrintFile 로 설정한다.
이렇게 하면 인쇄 명령을 위한 기본적인 인터페이스는 완료 되었다.
그리고 uses 절에 Printers 를 추가하고, 앞에서 작성한 PrintMemo 프로시저를 유닛의 implementation 섹션의 처음에 위치시킨다. 이 프로시저의 위치가 액션 아이템의 OnClick 이벤트 핸들러의 위치보다 앞에 있어야 하는데 주의한다. 물론 interface 섹션에 PrintMemo 프로시저의 선언부를 적어주면, 구현부분은 어디에 위치해도 상관없다. 완성된 폼의 형태는 다음과 같다.



마지막으로 PrintFile 액션 아이템의 OnExecute 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.PrintFileExecute(Sender: TObject);
begin
  PrintMemo(Memo1, True, True);
end;
```

이제 간이 에디터에 인쇄 기능이 추가되었다. 필요한 파일을 불러서 인쇄를 해보길 바란다.
이 어플리케이션은 9 장의 Exam4.dpr 파일을 로드하면 실행할 수 있을 것이다.

- 프린터 제어 코드 직접 보내기

프린터를 이용해서 도스에서 작업을 많이 한 경우, 프린터에 직접 제어 코드를 보내는 경우

가 있다. 이러한 프린터 제어 코드는 보통 Esc 키를 시작으로 나가게 되는데, 일반적인 방법으로는 윈도우 프로그램에서 이를 직접 사용할 수 없다.
그냥 생각하기에는 WriteLn 을 사용하면 될 것 같지만 제대로 동작하지 않는다. 여기서 이를 해결하기 위한 방법을 간단하게 소개한다.
먼저 TPrinter 를 사용해야 하므로, 제어 코드를 보내려고 하는 유닛의 uses 절에 Printers.pas 를 추가한다.
그리고, 다음과 같이 직접 프린터로 데이터를 보낼 수 있도록 레코드를 하나 선언한다.

```
type
  TPassThroughData = Record
     nLen : Word;
     Data : array[0..255] of Byte;
  end;
```

이 레코드에는 제어 코드로 보낼 데이터를 Data 필드에 저장하게 된다.
그리고, 실제로 데이터를 프린터로 보내는 프로시저를 다음과 같이 작성한다.

```
procedure DirectPrint(s: string);
var
  PTBlock : TPassThroughData;
begin
  PTBlock.nLen := Length(s);
  StrPCopy(@PTBlock.Data, s);
  Escape(Printer.Handle, PASSTHROUGH, 0, @PTBlock, nil);
end;
```

이 프로시저는 필자가 인터넷의 뉴스 그룹에서 발견한 것인데, 정리를 해 두었던 것으로 David Block(dblock@vdn.com)이란 사람이 소개한 것이다.
이 프로시저에서 핵심이 되는 부분은 Escape 라는 API 함수이다. 이 함수는 어플리케이션이 특정 디바이스에 GDI 를 거치지 않고 접근할 수 있도록 허용하는 역할을 하며, 지정된 드라이버에 데이터가 직접 날아간다.
WriteLn 으로 제어 코드를 날릴 때 이것이 작동하지 않는 이유는 언제나 이 문자를 GDI 가 중간에서 가로채기 때문이기 때문이므로, 이 함수를 사용하면 제어 코드를 날릴 수 있는 것이다. 이 함수의 첫번째 파라미터에는 디바이스 컨텍스트의 핸들을 지정한다. 여기서는 당연히 TPrinter 객체의 Handle 을 지정하면 된다. 그리고, 두번째 파라미터에 실행될 제어 코드 함수를 지정한다. 직접 제어코드를 날릴 때에는 PASSTHROUGH 를 지정하면 된다.

세번째 파라미터에는 날아갈 데이터의 크기를 지정하는데 0 을 지정하면 네번째 파라미터의 레코드 데이터가 모두 날아간다. 네번째 파라미터에 실제 데이터가 담긴 레코드의 포인터를 넘긴다. 마지막 파라미터에는 제어 코드를 날린 뒤에 돌아올 데이터를 담을 구조체의 포인터를 지정하면 되는데, 돌아올 데이터가 없을 때에는 nil 을 지정하면 된다.
사용법은 간단하다. 다음과 같이 DirectPrint 프로시저를 이용해서 제어 코드를 써 주면 된다.

```
procedure PrintTest;
begin
  Printer.BeginDoc;
  DirectPrint(CHR(27) + '&l1O' + 'Hello, World!');
  Printer.EndDoc;
end;
```

이렇게 DirectPrint 를 호출할 때에도 TPrinter 의 BeginDoc 과 EndDoc 메소드를 호출해야 하는데, BeginDoc 을 호출하면 프린터 드라이버가 프린터를 초기화하게 되며, EndDoc 메소드를 호출하면 프린터 드라이버가 페이지를 출력하게 된다.
참고로, 프린터 드라이버에 따라서는 PASSTHROUGH 를 지원하지 않을 수도 있다. 이를 알아보기 위해서는 다음과 같이 QUERYESCSUPPORT 를 이용하면 된다.

```
var
  TestInt: Integer;
begin
  TestInt := PASSTHROUGH;
  if Escape(Printer.Handle, QUERYESCSUPPORT, SizeOf(TestInt), @TestInt, nil) > 0 then
  begin
  ... (후략)
```

정 리 (Summary)

이번 장에서는 어플리케이션을 작성할 때 파일을 다루는 여러가지 테크닉과 델파이에서 제공하는 TPrinter 를 이용하여 인쇄를 하는 방법을 알아 보았다. 비교적 자주 쓰이면서도 잘 모르는 것이 파일의 입출력과 인쇄에 대한 방법 들이다.
지면 관계상 깊은 곳까지 다루지 못한 것이 아쉽지만, 델파이용으로 공개된 많은 VCL 소스를 살펴 보면 많은 것을 알 수 있을 것이다.
다음 장에서는 델파이를 이용해서 그래픽과 멀티 미디어 기능을 구현하고, 사용하는 방법에

대해서 다룰 것이다.