어플리케이션의 최적화와 세계화

(Considering Optimization and Internatinalization of Applications)

도스 시절에 C 를 이용해서 프로그래밍을 해본 경험이 있는 분들은 그 당시에 최적화를 위해서 얼마나 많은 신경을 썼는지 기억할 것이다. 최근의 프로그래밍 환경은 그 당시에 비해 최적화에 대한 고려를 그다지 하지 않아도 좋은 환경으로 바뀌었다. 그렇지만, 나름대로의 최적화는 항상 가능한 법이다.
최적화 기법은 대단히 어렵고 복잡한 것에서부터, 쉽고 단순한 것까지 다양하게 존재한다. 또한, 바야흐로 세계화 시대가 개막되면서 우리가 개발하는 각종 프로그램의 세계화를 고려하는 것도 자연스러운 일이 되었다. 이번 장에서는 델파이에서 손쉽게 고려할 수 있는 간단한 최적화 기법과 세계화를 하기 위해 알아야 할 방법들을 소개하고자 한다.

최적화 기법

- 리소스 절약

커다란 어플리케이션을 제작하다보면 리소스 부족으로 인해 에러가 발생하는 경우가 있다. 이럴 때에는 리소스를 절약하는 각종 테크닉을 사용해야 한다. 다음과 같은 방법으로 리소스를 절약할 수 있다.

1. 폼의 자동 생성(autocreation)을 피한다.

프로젝트에서 새로운 폼을 추가하게 되면, 자동적으로 어플리케이션의 시작 코드에 다음과 같은 형태로 폼을 자동 생성하는 코드가 추가 된다.

```
uses
  Form1 in 'Unit1.pas',
  Form2 in 'Unit2.pas',
begin
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  ...(중략)
```

```
end;
```

이렇게 하면 처음 어플리케이션이 시작할 때 느려질 뿐 아니라 많은 양의 메모리를 소모하게 된다. 그러므로, 메인 폼만 자동 생성하도록 하고, 다른 폼들은 Project|Options 메뉴에서 자동 생성을 하지 않도록 설정하고, 다음과 같은 형태로 필요할 때 만들어서 사용하도록 한다.

```
procedure MainForm.mnuChildFormClick(Sender: TObject);
begin
  ChildForm1 := TChildForm1.Create(Self);
ChildForm1.ShowModal;
ChildForm1.Release;
end;
```

2. 불필요한 OLE 지원 부분을 제거한다.

어플리케이션이 200KB 가 되지 않는 작은 크기라도, 델파이의 RTL 에서 기본적으로 OleAut32.dll 을 지원하기 위해서 약 1MB 의 RAM 을 사용하게 된다. 만약 어플리케이션에서 OLE 를 사용하지 않는다면 다음과 같은 코드로 이런 불필요한 메모리 낭비를 줄일 수 있다.

```
FreeLibrary(GetModuleHandle('OleAut32'));
```

메인 폼의 OnCreate 이벤트나 프로젝트 소스에서 이런 코드를 추가하면 OleAut32.dll 과 Ole32.dll 을 메모리에 적재하지 않게 되므로, 약 1MB 정도의 메모리를 절약할 수 있다.

3. 각각의 윈도우는 상당히 많은 양의 동적 리소스를 소모하므로 한번에 4~5 개 이상의 윈도우가 동시에 열리지 않도록 조치한다.

4. 같은 조건이라면 TPanel 보다는 TBevel 컴포넌트를 사용하는 것이 좋다. TPanel 컴포넌트는 각각의 윈도우에 대한 분리된 핸들이 필요한데 비해 TBevel 은 하나의 핸들만 사용한다. 그리고, TPanel 컴포넌트는 보기보다 많은 수의 프로퍼티와 이벤트가 연결되어 있기 때문에, dfm 파일의 크기를 많이 증가시킨다. 그 밖에 동일한 기능을 하는 컴포넌트 들이 있다면 되도록 적은 수의 프로퍼티와 이벤트를 가지고 있는 컴포넌트를 우선 사용하도록 한다.

5. 폼의 ParentFont 프로퍼티를 TRUE 로 설정하는 것이 좋다. 이렇게 하면 각각의 독립된 컴포넌트에 대해 독립된 폰트 인스턴스를 로드할 필요가 없다.

6. 컴포넌트를 만들 때에는 property 선언부에 default 값을 반드시 설정하도록 한다. default 값이 없으면 어플리케이션에서 dfm 파일에 무조건 값을 저장하게 되지만, deault 로 설정된 값이 있으면 변경되지 않는 한 저장되지 않기 때문에 dfm 파일의 크기를 줄일 수 있다.

7. 가능한 델파이의 폼의 자동 생성 기능은 꺼두는 것이 좋다. 메인 폼을 제외한 다른 폼들이 모두 자동 생성될 경우 로딩 시간이 길어지고, 힙 메모리의 낭비를 가져온다. 그러므로, 다소 번거롭더라도 폼은 동적으로 생성하고 해제하도록 한다.

8. 델파이의 폼을 사용하지 않는 프로그램을 개발하는 경우이면, uses 절에서 Forms.pas 유닛을 삭제하도록 한다. Forms.pas 유닛은 실행 파일의 크기를 100KB 이상 커지게 만든다.

- 예외 처리

어플리케이션을 개발할 때 예외 처리 루틴을 이용해서 각종 리소스를 보호하고, 예외가 일어났을 때 사용자에게 보다 세련된 방법으로 이를 알릴 수 있게 만들 수 있다. 그렇지만, 예외 처리 자체가 실행 속도를 저하시키는 단점이 있다. 일단 예외가 발생하면 운영체제와 어플리케이션 코드는 예외를 처리하게 위해 수백 개에 이르는 명령어를 더 처리해야 한다. 다음의 코드를 살펴 보자.

```
function GetSquare(P: PInteger): Integer;
begin
  try
    Result := Sqr(P^);
  except
    ShowMessage('P was nil !');
    Result := -1;
  end;
end;
```

여기에서 P 가 nil 이면 예외가 발생한다. 이때 예외를 처리하는데 수행 속도가 저하되므로,

이를 위해서는 다음과 같이 if 문으로 검사하는 것이 효과적이다.

```
function GetSquare(P: PInteger): Integer;
begin
  if (P = nil) then
  begin
    ShowMessage('P was nil !');
    Result := -1;
  end
  else Result := Sqr(P^);
end;
```

또한, 예외 처리를 사용하면 블록에서 빠져나가기 위해 Exit 문을 사용할 경우가 생기는데, 이 역시 성능을 저하시킨다. 다음의 코드를 살펴보자.

```
var
  P: PChar;
begin
  GetMem(P, 1024);
  try
    ...
    if (P^ = 'Sample') then Exit;
    ...
  finally
    FreeMem(P, 1024);
  end;
end;
```

이렇게 try..finally 블록에서 빠져나가기 위해서 Exit 를 사용하는 것보다는 다음과 같이 하는 것이 더 효과적이다.

```
var
  P: PChar;
begin
  GetMem(P, 1024);
  try
```

```
    ...
    if (P^ <> 'Sample') then
    begin
      ...
    end;
  finally
    FreeMem(P, 1024);
  end;
end;
```

- 문자열 처리

델파이 2 이후 부터는 문자열의 255 자 길이 한계가 없어지면서 동적으로 메모리를 할당받는다. 그리고, 문자열 자체도 참조값으로 취급되기 때문에 비교적 빠르게 동작한다. 그렇지만, 아직도 문자열을 처리할 때 몇 가지 고려해야할 것들이 있다.
먼저 문자열에 필요 없는 내용을 가지고 있는 경우이다. 다음 코드를 살펴보자.

```
procedure SampleString;
var
  S: String;
begin
  S := '';
  ...
end;
```

여기서 S 변수에 대한 대입문은 전혀 필요 없는 부분이다.
이 경우는 문자열 처리에 대한 실행속도의 최적화에 해당되는 내용이다.
문자열을 다룰 때에는 코드의 크기를 최적화 하는 방법도 생각해야 한다. 예를 들어 다음과 같이 반복되는 문장에 대한 대입문이 있다고 하자.

```
S1 := '이것은 지구상에서 가장 크다';
S2 := '이것은 지구상에서 가장 무겁다';
S3 := '이것은 지구상에서 가장 빠르다';
```

이 경우에 '이것은 지구상에서 가장 ' 이라는 문자열이 반복되고 있다. 이렇게 해서 만들어진 실행파일을 살펴보면 반복된 문자열이 그대로 실행파일에 들어있는 것을 알 수 있다.

그러므로, 이를 다음과 같이 처리하면 실행파일의 크기를 줄일 수 있다.

```
const
  S = '이것은 지구상에서 가장 ';
begin'
  S1 := S + '크다';
  S2 := S + '무겁다';
  S3 := S + '빠르다';
end;
```

이 경우에는 상수가 컴파일 시에 정해지므로 그다지 큰 효과를 거두기는 어렵다. 그렇지만 상수의 선언을 다음과 같이 해보자.

```
const
  S: String = '이것은 지구상에서 가장 ';
```

이렇게 하면 S 가 문자열로 한번만 할당받고, 뒤쪽에 붙는 문자열 부분은 런타임에서 적절하게 처리된다. 그러므로, 중복되는 부분만큼 코드 크기를 줄여줄 수 있다. 물론 컴파일 시에 처리하는 것보다 속도의 저하는 가져올 수 있다. 그렇지만, 여러 번 중복되는 대입문이 있을 때에는 반드시 고려해야 할 테크닉이다.

- 순환문과 판단문의 최적화

어플리케이션의 수행 속도에는 순환문의 처리 방식이 큰 영향을 미친다. 이중에서도 특히 for 와 while 문을 잘못 사용하면 커다란 수행 속도의 손실을 가져올 수 있다. 예를 들어, 다음의 코드를 살펴 보자.

```
while i <= SomeFunction(AParameter) do
begin
  ...
end;
```

여기에서 SomeFunction 함수가 정수값을 돌려준다고 할 때, 이 함수는 루프를 도는 동안 계속해서 실행된다. 그러므로, 이 함수의 실행에 0.1 초가 걸린다고 가정하고 루프를 100 번을 돌 경우 10 초가 걸리는 셈이다. 이런 경우에는 다음과 같이 for 문을 이용하면 한 번만 실행되므로 수행 속도를 대폭 향상시킬 수 있다.

```
for I := 1 to SomeFunction(AParameter) do
begin
  ...
end;
```

판단문에 있어서는 가능한 else 문을 사용하지 않는 것이 요령이다. 예를 들어, 다음의 코드는 쉽게 else 를 떼어 버리도록 변경이 가능하다.

```
if Sample = False then A := x else A := y;
```

이 문장은 다음과 같이 변경할 수 있다.

```
A := x;
if Sample = True then A := y;
```

별것 아닌 것 같지만 이렇게 변경하는 것으로 jump, return 이라는 2 개의 기계어 코드를 절약할 수 있다.
최적화를 고려하면 판단문에서 중첩된 if...then...else 문장은 스택에 계속적인 push 를 하게 되므로 스택의 오버 플로우를 유발할 수도 있고, 효과적이지도 않다. 이런 경우에는 가능한 case 문장 등을 동원하여 변경하는 것이 좋다.
예를 들어 다음의 코드를 살펴보자.

```
if bps = 2400 then SomeFunc1
else if bps = 9600 then SomeFunc2
      else if bps = 1400 then SomeFunc3 ...
```

이런 경우에는 다음과 같이 변경하는 것이 바람직하다.

```
case bps of
  2400: SomeFunc1;
  9600: SomeFunc2;
  14400: SomeFunc3;
  ...
end;
```

- 간단한 최적화 기법

앞서 설명하지 않은 몇 가지 간단한 최적화 기법을 정리해서 소개하면 다음과 같은 것들이 더 있다.

1. DLL 에 커다란 파라미터를 넘기는 것은 금물이다. DLL 은 호출하는 프로그램과 스택을 공유하기 때문에, 에러가 발생하기 쉽다.

2. 당연한 이야기이지만, 순환 링크(circular link)는 사용하지 않도록 한다. 유지 보수가 어렵고 스마트 링크가 동작하지 않도록 한다.

3. 부득이한 경우가 아니면 Byte 데이터 형은 사용하지 않는다. CPU 의 최소 처리 단위가 Word 이기 때문에 저장 공간의 절약을 위해 Byte 를 사용할 경우 생각보다 많은 절약이 되지 않을 뿐더러, 컴파일러에 의해 이를 처리하기 위한 많은 코드가 생성되어야 하기 때문에 비효율적이다.

4. 마찬가지 이유로 Boolean 데이터 형도 Integer 로 변경하여 사용해도 되는 경우라면 Integer 형을 사용할 것을 권한다. Boolean 데이터 형도 비효율적인 데이터형 이다.

- 테이블과 데이터베이스 최적화 기법

테이블과 데이터베이스를 다룰 때에도 속도를 향상시킬 수 있는 수많은 테크닉들이 있다. 기본적인 테크닉으로는 다음과 같은 것들이 있다.

1. 테이블을 검색을 할 때에는 인덱스를 사용한다.

2. 한 테이블에 너무 많은 필드를 사용하지 않는다. 테이블 하나에 100 개 이상의 필드가 존재하면 메모리를 과도하게 사용하게 되고, 데이터 전송 시 무리가 오게 되므로 이런 경우 테이블을 열기 전에 몇 개의 테이블로 나눈다.

3. 테이블은 꼭 필요할 때에만 열어야 한다. 이는 테이블을 열 때 걸리는 시간이 많기 때문이며, 만약 테이블을 자주 열고 닫게 될 경우에는 테이블을 계속 열어 놓는 것이 좋다. 또한, 여러 개의 파트로 나누어진 어플리케이션을 만들 경우에는 (예를 들어 학생용, 교수용, 통계용 클라이언트가 따로 있는 경우) 테이블을 여는 작업을 폼의 OnCreate, OnClose 이벤트에서 작업하는 것이 좋다. 이렇게 하면, 잠금(locking)이나 업데이트 문제를 예방할 수 있다.

4. 서버 데이터베이스에 Range 를 사용할 때에는 서버가 레코드를 어떤 식으로 접근하는지 알지 못하기 때문에, 특정 범위의 레코드를 변경하고자 할 때에는 SQL 문장의 UPDATE 절을 사용하는 것이 좋다. 이렇게 UPDATE 절을 사용해야 레코드에 대한 모든 작업이 서버에서 작동하며 클라이언트로 변경할 레코드를 넘겨받아서 다시 넘겨주는 불필요한 작업을 거칠 필요가 없기 때문에 효과적이다.

5. 테이블에 대량으로 접근해서 조작을 할 때에는 TTable.DisableControls 를 사용해서 데이터 컨트롤과의 연결을 끊은 후 조작이 끝나고 TTable.EnableControls 를 호출한다.

6. 가능한 OnCalcFields 이벤트는 사용하지 않는다. 속도를 대단히 떨어뜨리는 요인이 된다. 대신 TDataSource.OnDataChange 이벤트를 활용하는 것이 요령이다.

7. DBase 테이블을 SQL 을 사용해서 접근하며 매우 느리다. 이는 서버의 작업이 이루어진 후 결과가 되는 세트 만을 전송해야 하는데, DBase 테이블의 경우에는 네트워크를 통해 데이터가 클라이언트로 모두 넘어온 후 처리되기 때문이다. 그러므로, 네트워크가 느리고 DBase 테이블이 큰 경우에는 심각한 네트워크 트래픽과 클라이언트 부담을 안을 수 밖에 없다. 이럴 때에는 빨리 파라독스로 전환하는 것이 요령이다.

- 실행파일의 크기를 줄이려면 ...

실행 파일의 속도도 중요하지만, 가능한 1 바이트라도 실행 파일의 크기를 줄이는 것도 중요하다. 실행 파일의 크기를 줄이는 기본적인 테크닉으로는 다음과 같은 것들이 있다.

1. 가능한 if..then..else 절보다는 case 문을 사용하는 것이 좋다.

2. 컴파일러 옵션의 활용

프로젝트에 연결되는 모든 .pas 파일과 .dpr 파일의 제일 위에 다음과 같은 줄을 추가한다.

```
{$D-,L-,O+,Q-,R-,Y-,S-}
```

{$D-}는 코드에 디버그 정보를 없애는 것으로 실행 파일을 배포할 때에는 이 옵션을 사용한다. {$L-}는 로컬 심볼을 코드에 추가하지 않는 것이며, {$O+}는 컴파일러에게 불필요한 변수를 제거해서 코드를 최적화하도록 한다.
{$Q-}는 정수 오버플로우 검사하는 코드를 제거하며, {$R-}는 문자열, 배열 등의 범위 검

사 코드를 삭제한다. 또한, {$S-}는 스택 검사에 대한 코드를 사용하며 {$Y-}는 심볼 정보를 코드에 추가하지 못하도록 한다.
이렇게 하면 실행 파일의 크기가 약 10~20% 정도가 작아진다.
참고로 이 작업은 실행 파일을 배포할 단계에서 마지막으로 빌드할 때 적용해야 한다.

3. Project|Options|Compiler 페이지에서 'Show Hints', 'Show Warnings' 옵션을 체크하고 프로젝트를 빌드하면 사용되지 않는 변수, 프로시저/함수를 보여준다. 이를 이용해서 불필요한 부분을 제거한다.

불필요한 유닛에 대한 uses 절을 제거한다. 델파이의 '스마트-링킹(Smart-Linking)이 우수하지만, 쓰이지 않는 코드를 모두 제거해주지는 않는다는 것을 알아야 한다.

- 옵티마이저(Optimizer)와 RTTI

델파이의 옵티마이저는 그 성능이 뛰어난 것으로 정평이 나있다. 옵티마이저가 메모리에 있는 변수에 대한 할당 부분을 CPU 레지스터를 활용하거나, 공통적인 표현식을 제거하여 최적화하는 등의 여러가지 일을 해준다. 그렇지만, 개발자가 신경을 써주면 옵티마이저의 이러한 특성을 더욱 잘 발휘하게 할 수도 있다. 다음의 코드를 살펴보자.

```
function DummyFunc(i: Integer): Integer;
begin
  Result := i;
end;

procedure Sample(i: Integer);
begin
  if (DummyFunc(i) > 1000) then ShowMessage('Thousands')
  else if (DummyFunc(i) > 100) then ShowMessage('Hundreds')
  else if (DummyFunc(i) > 10) then ShowMessage('Tens')
  else ShowMessage('A few');
end;
```

이 코드를 다음 코드와 비교해 보자.

```
procedure Sample(i: Integer);
var
  j: Integer;
```

```
begin
  j := DummyFunc(i);
  if (DummyFunc(i) > 1000) then ShowMessage('Thousands')
  else if (DummyFunc(i) > 100) then ShowMessage('Hundreds')
  else if (DummyFunc(i) > 10) then ShowMessage('Tens')
  else ShowMessage('A few');
end;
```

언뜻 보기에는 거의 달라진 것이 없어 보이지만, 여기에서는 j 변수의 값이 일단 CPU 레지스터에 저장되도록 최적화되기 때문에 코드가 빠르게 수행된다.

델파이의 RTTI(Run-time Type Information)를 활용할 때에도 최적화를 할 수 있는 경우가 있다. 예를 들어 다음과 같은 코드를 생각해보자.

```
uses MPlayer;
...

function Sample(Button: TMPBtnType): String;
begin
  case Button of
    btPlay: Result := 'btPlay';
    btPause: Result := 'btPause';
    ...
  end;
end;
```

이때 이렇게 나열한 부분이 실행 파일에 포함된다. 그러므로, 이로 인한 실행파일의 크기 증가와 코딩 시간의 낭비 등을 가져올 수 있는 것이다. 앞의 버튼 이름은 이미 RTTI 에 기록되어 있기 때문에 다음과 같이 RTTI 를 직접 이용하면 이러한 낭비를 막을 수 있다.

```
uses MPlayer, TypInfo;
...

function Sample(Button: TMPBtnType): String;
begin
  Result := GetEnumName(TypeInfo(TMPBtnType), Ord(Button));
```

end;

그 밖에도 수 많은 최적화 테크닉이 존재하겠지만 여기서는 이 정도로 줄이고자 한다. 최근과 같이 좋은 개발 환경에서도 신경을 쓰면 작게나마 코드의 크기를 줄이거나, 실행 효율을 향상시키는 최적화 기법은 존재하기 마련이다.
언제나 코딩을 할 때 내가 사용한 코드가 효율적인지를 다시 한 번쯤 되돌아 볼 줄 아는 습관을 들이는 것이 중요할 것이다.

어플리케이션의 세계화 (Internationalization)

세계화(internationalization)는 프로그램을 여러 나라에서 사용할 수 있도록 하는 과정이다. 이 과정에서 각 나라의 언어와 문화적인 요소(시간 표기법 등) 등의 사용자 환경을 locale 이라고 한다. 윈도우는 많은 세트의 locale 들을 지원하는데, 각각의 locale 에 맞도록 번역하는 과정을 지역화(localization)이라고 한다.
앞으로 설명할 내용에서는 세계화는 여러나라의 locale 을 쉽게 적용할 수 있도록 어플리케이션을 처음 디자인하고 코딩할 때부터 신경써야 할 내용에 대한 것을 설명하는 것이고, 지역화는 특정 locale 에 적용하기 위해서 어떤 것을 고려해야 되는지에 대해서 설명할 것이다.

- 어플리케이션 세계화의 단계

어플리케이션을 세계화하기 위해서는 일반적으로 다음에 설명하는 몇 가지 단계를 거쳐야 한다. 일단 문자 세트를 여러 가지 활용할 수 있도록 문자열을 다루어야 할 것이며, 사용자 인터페이스를 쉽게 변경할 수 있어야 할 것이다. 그리고, 핵심적인 사항은 될 수 있는 한 지역화할 모든 리소스를 분리할 필요가 있다.

1. 어플리케이션 코드의 변경

세계화를 고려할 때 가장 중요한 점은 다양한 locale 에 적용할 수 있도록 문자열을 변경할 수 있어야 한다는 점이다. 각 나라의 문자 세트는 코드 페이지라고 불리는 각 나라 고유의 페이지를 이용한다. 경우에 따라서는 일반적으로 많이 사용되는 ANSI 윈도우 문자 세트를 어플리케이션 사용자 기계의 코드 페이지에 지정된 문자 세트(OEM 문자 세트)로 변환할 필요가 있다.
이때 우리나라를 비롯한 아시아 나라의 문자 세트의 경우 단순한 1:1 매핑으로 해결할 수가 없는 경우가 많다. 그렇기 때문에 2 바이트를 처리할 수 있는 문자열 처리 함수들이 많이 필요하게 되는데, 그렇기 때문에 문자열의 길이를 바이트로 처리하는 내용을 담고 있을

때에는 반드시 이를 고려해서 프로그래밍해야 한다.
이런 어려운 점을 해결하기 위한 방안으로 가장 좋은 것 중의 하나는 유니코드를 이용하는 것이다. 유니코드 문자 세트는 델파이에서 WideChar 데이터 형으로 지원하고 있다. 윈도우 NT 라면 이것으로 거의 문제를 해결할 수 있지만, 윈도우 95 를 사용할 경우에는 API 함수가 지원되는 것이 별로 없다는 단점이 있다. 그렇기 때문에, 델파이의 VCL 소스 코드를 보면 대부분의 문자열을 다루는 루틴이 ANSI 의 단일 바이트 문자 세트와 유니코드를 지원하는 DBCS 문자 세트를 모두 지원하도록 되어 있다. VCL 소스 코드를 자주 들여다보는 사람이라면 쉽게 알 수 있겠지만 각종 루틴의 말미에 'A'가 붙어 있으면 ANSI 를 지원하는 것이며 'W'가 붙어 있으면 유니코드를 지원하는 것이다.
그 밖에도 IME(input method editor)를 직접 제어하여 사용자의 입력을 제어할 수 있도록 하는 것도 좋은 지역화의 한 방법이다. 델파이 3 부터는 VCL 컴포넌트가 IME 를 이용할 수 있도록 제공되고 있다. 대부분의 윈도우 컨트롤은 텍스트 입력을 받을 때 ImeName 프로퍼티를 이용하여 특정 IME 를 선택할 수 있도록 지원하고 있으며, ImeMode 프로퍼티를 이용하여 IME 가 키보드 입력을 어떤 방식으로 전환할 것인지 지정할 수 있다.
또한, 전역 Screen 변수에서 사용자 시스템에서 이용가능한 IME 들에 대한 정보를 얻을 수 있으며, 그 밖에도 사용자 시스템에 설치된 키보드 매핑에 대한 정보를 얻을 수 있다.

2. 사용자 인터페이스 디자인

문자 세트에 대한 고려도 중요하지만, 여러 나라에 어플리케이션을 개발해서 배포하기 위해서는 사용자 인터페이스를 디자인하는 것도 중요하다. 이때 사용자 인터페이스는 각 나라의 환경에 맞도록 쉽게 수정할 수 있도록 해야 한다.
예를 들어, 컨트롤의 캡션 등은 여러나라의 문자로 바뀔 수 있으므로 해당되는 나라의 문자열로 바꾼 뒤에도 캡션이 나타날 수 있도록 넉넉한 공간이 확보되어야 할 것이다.
이렇게 번거로운 문자열 변환 과정을 최소한으로 줄이기 위해서는 이미지를 잘 활용하는 것이 요령이다. 그런데, 이미지에 문자를 같이 사용해야 되는 경우라면 하나의 이미지에 문자를 포함해서 사용하지 말고 투명 배경을 이용해서 텍스트 부분 만을 따로 이미지를 이용할 수 있도록 하는 것이 좋다.
그리고, 각 locale 별로 변경해서 사용해야 할 것으로는 날짜와 시간, 숫자, 화폐 단위 등이 있다. 윈도우 포맷만을 사용한다면 이러한 정보를 윈도우 레지스트리에서 얻을 수 있다.

3. 리소스의 분리

어플리케이션을 세계화할 때 코드를 특별히 변경하지 않고 쉽게 여러나라 언어로 지역화하기 위해서는 사용자 인터페이스의 문자열을 독립된 단일 모듈을 분리하여 사용하는 것이 좋다. 델파이는 메뉴, 대화상자, 비트맵 등에 대한 리소스를 포함한 .dfm 파일을 자동으로 생

성하므로, 여기의 내용을 텍스트 에디터로 직접 편집하는 것도 한 방법이다.
그 밖에 문자열을 따로 분리하여 사용할 수 있는 방법이 있는데 여기에 대해서 더욱 자세히 알아보도록 하자.

- 리소스 DLL 의 제작

리소스를 분리하면 지역화를 매우 편하게 진행할 수 있다. 리소스 분리를 할 때 편리하게 사용할 수 있는 방법의 하나가 리소스 DLL 을 생성하여 사용하는 것이다. 리소스 DLL 을 작성하면 리소스 DLL 을 변경하는 것으로 간단하게 번역 작업을 대신할 수 있고, 배포 작업 역시 간단하게 만들 수 있다.
델파이 4 에서 제공되는 리소스 DLL 위저드를 이용하면 쉽게 리소스 DLL 을 생성할 수 있다. 리소스 DLL 위저드는 현재의 저장된 프로젝트에 대해서 RC 파일과 프로젝트에서 사용된 resourcestring 지시어로 지정된 문자열 테이블을 포함한 RC 파일을 생성한다. 그리고, 적절한 폼과 RES 파일의 내용을 포함한 리소스 DLL 에 대한 프로젝트를 생성한다.
지원하고자 하는 locale 이 있다면, 그 수만큼의 리소스 DLL 을 생성하면 된다. 이때 각각의 리소스 DLL 은 해당 locale 에 대한 확장자를 지정하는데, 보통 처음 2 자는 해당 언어를 가리키며, 마지막 문자는 나라를 지시한다. 한국어의 경우는 언어와 사용하는 나라가 일치하므로 kor 로 표시할 수 있지만, 영어의 경우에는 영국, 미국, 캐나다 등 여러 나라를 지정할 수 있다. 이 경우 미국의 경우 enu(english + US), 캐나다의 경우 enc(english + Canada)가 확장자로 사용된다.

- 리소스 DLL 의 활용

리소스 DLL 을 이용하면 실행파일과 다른 DLL, 패키지에 포함된 리소스를 쉽게 지역화된 버전으로 만들 수 있다. 어플리케이션을 시작할 때 로컬 시스템의 locale 을 검사하고, EXE, DLL, BPL 파일과 같은 이름의 리소스 DLL 을 발견하게 되면 확장자를 검사해서 확장자가 시스템 locale 의 나라와 언어에 부합하는지 알아보고, 그렇다면 실행 파일이나 DLL, 패키지에 포함된 리소스 대신 그 리소스 모듈의 리소스를 사용한다. 그리고, 해당되는 언어와 나라에 대한 리소스 모듈이 없을 경우에는 기본적으로 언어에 해당되는 것을 찾고 그조차도 없을 경우에는 실행 파일이나 DLL, 패키지에 포함되어 컴파일된 리소스를 사용한다.
어떤 경우에는 어플리케이션에서 로컬 시스템의 locale 과 매치되는 리소스 모듈 이외의 리소스 모듈을 사용하고 싶을 때가 있다. 이럴 때에는 윈도우 레지스트리의 엔트리를 설정하면 된다. HKEY_CURRENT_USER₩Software₩Borland₩Locales 키에 어플리케이션의 패스와 파일 이름을 문자열 값으로 추가하고, 데이터 값을 리소스 DLL 의 확장자를 데이터 값으로 설정하는 것이 그 방법인데, 어플리케이션이 시작할 때 시스템 locale 에 해당되는 리소스 모듈을 찾기 전에 어플리케이션이 이 키의 값을 이용하여 리소스 모듈을 찾는다.

예를 들어, 다음의 프로시저는 프로그램을 처음 설치하거나 설정할 때 레지스트리 키 값을 설정할 수 있다.

```
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    if Reg.OpenKey('Software₩Borland₩Locales', True) then
      Reg.WriteString(LocalOverride, FileName);
  finally
    Reg.Free;
end;
```

이렇게 설정하고 나면, 다음과 같이 어플리케이션에서 FindResourceHInstance 함수를 이용하여 현재의 리소스 모듈의 핸들을 얻을 수 있다.

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
```

- 리소스 DLL 의 동적 변경

리소스 DLL 은 어플리케이션이 시작될 때 설정하는 것이 보통이지만, 런타임에서 동적으로 이를 변경하는 것도 가능하다. 이를 위해서는 델파이의 데모 어플리케이션과 함께 제공되는 ReInit.pas 유닛을 사용하면 된다. ReInit.pas 유닛은 델파이 4 의 Demos 디렉토리의 RichEdit 디렉토리에 있으므로 이를 복사해서 사용하면 된다.
언어를 변경하려면 LoadNewResourceModule 함수에 새로운 언어에 대한 LCID 를 넘겨주고, ReinitializeForms 프로시저를 호출하면 된다. 예를 들어, 다음의 코드는 인터페이스 언어를 프랑스어로 변경한다.

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;

if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

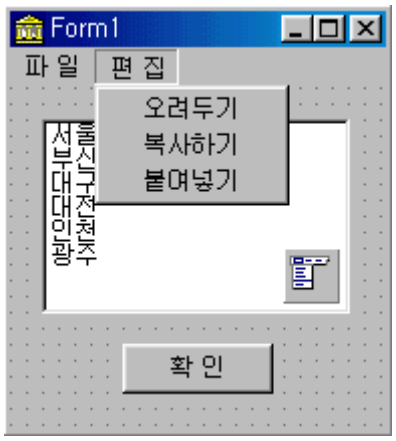
이 방법의 장점은 레지스트리의 설정을 변경하거나, 어플리케이션을 특별히 다시 시작할 필요 없이 리소스를 변경할 수 있다는 점이다.
이때 주의할 점은 런타임에서 폼에 대한 프로퍼티를 변경한 내용이 효력을 잃게 되므로, 일단 새로운 DLL 이 로드되면 디폴트 값이 재설정되지 않는다. 그러므로, 경우에 따라서는 폼 객체의 프로퍼티를 재설정 해주어야 한다.
리소스 DLL 을 동적으로 변경하는 예제는 델파이 4 의 Demos 디렉토리의 RichEdit 서브 디렉토리에서 제공되는 RichEdit.dpr 프로젝트 예제를 참고하기 바란다.

- 리소스 DLL 위저드

리소스 DLL 프로젝트를 생성하기 위해서 리소스 DLL 위저드를 사용하는 방법에 대해서 알아보자. 리소스 DLL 에는 어플리케이션에서 사용하는 모든 폼과 문자열에 대한 정보를 포함한다. 보통 리소스 DLL 은 어플리케이션을 배포할 준비가 모두 끝났을 때 작성하는 것이 좋다.
리소스 DLL 위저드를 이용하려면 프로젝트를 저장하고 빌드한 뒤에 File|New 메뉴에서 Resource DLL wizard 아이콘을 더블 클릭하면 된다. 이렇게 하면 델파이는 변환이 필요한 모든 폼과 문자열을 포함한 파일의 리스트를 보여준다. 여기에서 다른 폼을 추가하거나 삭제하려면 Add, Remove 버튼을 이용하면 된다.
폼들이 모두 결정되었으면 Next 버튼을 클릭한다. 이렇게 하면 위저드가 추가적인 DRC 파일을 추가할 것인지 물어본다. 만약에 다른 DRC 파일을 사용하는 프로젝트라면 여기에서 Add 버튼을 클릭하여 추가할 수 있다.
마지막으로 Next 버튼을 클릭하면 리소스 DLL 에 대한 언어를 선택하면 된다.
각각의 윈도우 locale 은 해당되는 파일 확장자를 가진다. 그리고, 리소스 DLL 에 대한 프로젝트는 언어 확장자와 같은 이름의 서브 디렉토리에 저장된다.
어플리케이션을 배포하려면, DLL 을 서브 디렉토리에서 프로젝트 디렉토리로 복사한 뒤에 같이 배포하면 된다.

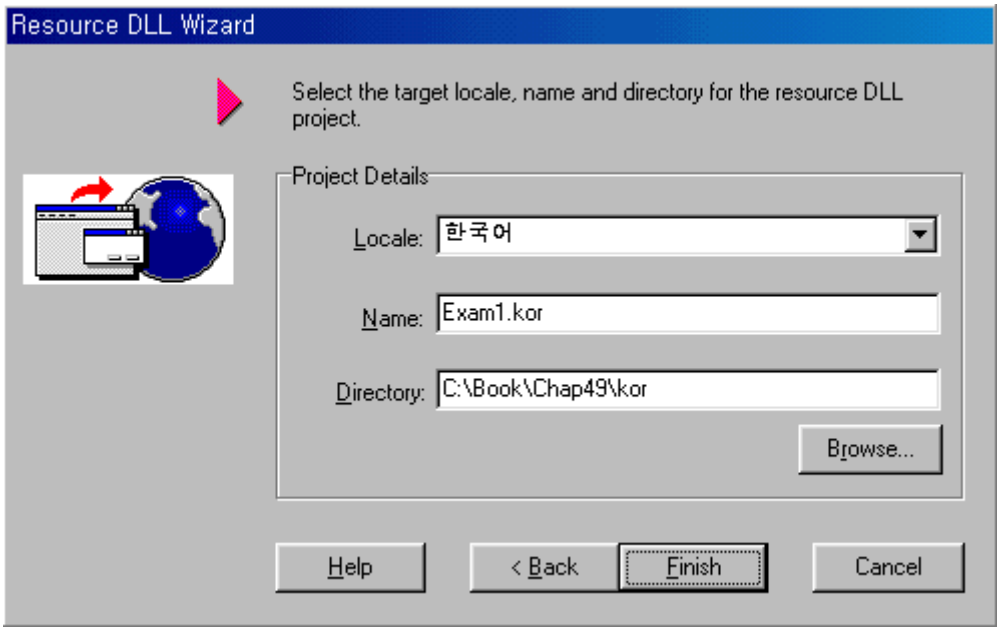
예제 어플리케이션의 제작

그러면, 한국어와 영어(미국과 캐나다)를 지원하는(아마도 우리나라 개발자는 이렇게 개발하는 경우가 가장 많을 것이다) 간단한 예제 어플리케이션을 개발해보도록 하자. 먼저 한국어를 지원하도록 다음과 같이 리스트 박스와 버튼, 메뉴 컴포넌트를 하나씩 얹고 폼을 디자인하도록 하자.

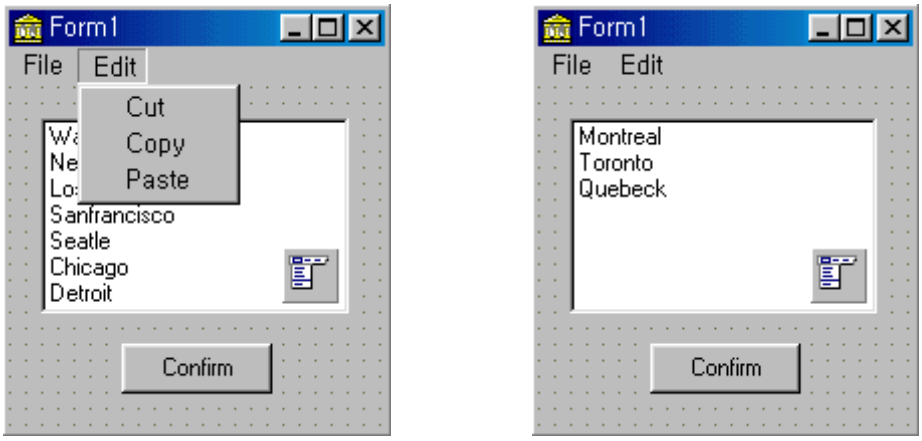


이번에 제작하는 어플리케이션의 목적은 2 개 국어와 3 개 나라를 지원하도록 리소스 DLL 위저드를 이용하는 방법을 익히는 것이므로, 특별한 코딩은 하지 않는다. 리소스 DLL 위저드를 사용하기 위해서는 프로젝트를 저장하고, 컴파일을 일단 해주어야 한다.

저장과 컴파일이 끝났으면 File|New 메뉴를 선택하고 Resource DLL Wizard 아이콘을 더블 클릭하여 리소스 DLL 을 생성하도록 하자. 리소스 DLL 위저드가 처음 실행되면 현재 프로젝트의 폼이 나타날 것이다. 이때 리소스 DLL 에 포함시킬 추가적인 폼이 있다면 Add 버튼을 클릭하여 추가하도록 한다. 그렇지만, 대부분의 경우 프로젝트 내부에 리소스가 포함되도록 디자인하므로 특별한 폼의 추가 없이 Next 버튼을 클릭하면 된다. 그 다음 화면에는 리소스 파일을 추가하도록 하는 화면이 나타나는데, 마찬가지로 추가할 .rc 파일이 있으면 여기에서 추가하도록 한다. 보통은 Next 버튼을 클릭하면 된다. 마지막으로 다음과 같이 Locale 과 DLL 파일의 이름, 디렉토리를 지정하는 대화상자를 완성하고 Finish 버튼을 클릭하면 리소스 DLL 프로젝트가 만들어진다.



리소스 DLL 프로젝트가 만들어 졌으면, 이를 컴파일하면 해당 디렉토리에 확장자가 .kor 인 DLL 파일이 생성된다. 나중에 이를 이용하여 한국어 버전을 배포하는 방법을 설명할 것이다. 다시 이전의 프로젝트 파일을 열고, 미국 버전을 위해 폰트를 MS San Serif 의 8 포인트로 설정한다 (필자는 한국어 어플리케이션을 제작할 때 폰트를 굴림, 9 포인트를 사용한다). 그리고, 폼을 다음과 같이 디자인한다. 참고로 캐나다 버전의 폼 디자인도 옆에 같이 나타내었다. 캐나다 버전을 만드는 방법은 미국 버전과 동일하므로 생략한다.



프로젝트 파일을 저장하고, 컴파일을 한 뒤에 다시 리소스 DLL 위저드를 실행한다. Next 버튼을 2 번 클릭하여 추가할 폼과 리소스 파일이 없다는 것을 나타낸 뒤에 마지막 대화 상자의 Locale 로 '영어(미국)'을 선택한다. 이렇게 하면 아마도 Name 은 'Exam1.enu', Directory 는 'c:₩Book₩Chap49₩enu'로 설정될 것이다. 참고로 캐나다 버전의 경우에는 Locale 은 '영어(캐나다)', Name 과 Directory 는 'Exam1.enc, c:₩Book₩Chap49₩enc'로 각각 지정하면 된다. Finish 버튼을 클릭하면 리소스 DLL 프로젝트가 생성되며, 이를 컴파일해서 .enu, .enc DLL 파일을 생성하도록 한다.

이것으로 리소스 DLL 의 제작이 완료되었다. 그러면, 이들을 직접 만들어 보자. 미국 버전과 캐나다 버전을 나중에 제작했으므로 현재의 Exam1 실행 파일은 영어를 지원할 것이다. 이 실행파일을 한국어를 지원하도록 하려면, ₩kor 디렉토리의 Exam1.kor 파일을 실행파일 디렉토리로 옮기면 된다. 실행파일을 실행하면 처음에 제작한 한국어 버전의 폼이 나타나는 것을 볼 수 있다. 배포하는 방법은 이와 같이 각 나라의 로컬 버전에 맞는 리소스 DLL 파일을 실행파일 디렉토리에 포함해서 제공하면 된다.

정 리 (Summary)

델파이 4 를 이용하여 각종 프로그램을 제작하는 여러가지 단계가 모두 중요하겠지만, 마무리와 뒷처리도 그에 못지 않게 중요한 부분이다. 이번 장에서 다룬 최적화와 세계화라는

이슈는 이런 마무리에 해당되는 부분인 만큼 최대한 깔끔하고 깨끗하게 처리하는 것이 좋을 것이다.

이제 개발자들도 세계를 무대로 프로그램을 제작하는 시대가 되었다. 델파이 4 에서 제공하는 리소스 DLL 위저드는 우리나라의 프로그래머들이 외국 시장을 쉽게 공략할 수 있는 터를 열어 놓았지만, 다른 측면에서 보면 미국이나 유럽 또는 인도 등의 소프트웨어 강국에서 쉽게 한국어 버전의 프로그램을 만들어낼 수 있는 장이 열렸다고도 할 수 있다.

그러므로, 개발자들도 우물안 개구리처럼 한국 시장에서만 통할 수 있는 프로그램을 개발하는데 열을 올릴 것이 아니라 바야흐로 시야를 넓혀서 세계 시장을 적극적으로 공략하는 발상의 전환이 필요한 시대이다.