
오브젝트 파스칼 입문

이 책을,

부족하다 싶을 정도로만

무엇인가 알 것 같다 싶을 정도로만

공부하시기 바랍니다.

나머지 부족한 부분은,

프로그래밍을 직접 작성하는 동안,

다른 사람의 프로그램 소스를 보는 동안,

보다 명확하게 채워집니다.

이 책에서는 기본적인 지식을 습득하는 것을 목표로 합니다.

현재 준비 중인, 다른 책들을 통해서 오브젝트 파스칼과 델파이 프로그래밍의 진수를 보여드릴 것을 약속드리겠습니다.

마치 영어를 공부할 때는

단어를 직접 외우는 것보다, 문장 속에서 외워야 하듯이,

프로그래밍을 배우실 때에도

단편적인 조각 지식을 습득하고 외우는데, 너무 많은 시간을 빼앗기지 마시기 바랍니다.

최소한의 문장을 만들고,

최소한의 문장을 이해할 수 있는,

상태!

바로 거기서부터가 시작입니다.

하지만, 주어진 예제는

반드시 이해를 하고

반드시 따라해 보고

반드시 스스로 작성해보고

가능하면, 자신만의 아이디어로 새로 만들어 보시기 바랍니다.

프로그래밍이란

프로그래밍에 대해서는 여러 가지 정의를 내릴 수 있겠지만, 필자의 경우에는 “프로그램이란 컴퓨터에게 하달하는 작업 스케줄”입니다. 즉, 프로그래밍이란 컴퓨터에게 일을 시키기 위해서 컴퓨터가 알아듣는 언어로 작업 스케줄을 작성하는 것입니다.

만약 여러분들이 자동으로 세차하는 프로그램을 만든다고 가정하겠습니다. 이때 작업 스케줄을 우리가 사용하는 언어로 표현하여 아래와 같이 작성해보겠습니다

[절차형 프로그래밍 방식]

- 1 : 자동차에 물을 뿌려라.
- 2 : 자동차에 비누칠을 해라.
- 3 : 자동차에 솔질을 해라.
- 4 : 자동차에 물을 뿌려라.
- 5 : 자동차를 건조기로 말려라.

만약 여러분들이 게임을 만들려고 한다고 가정하여 다른 작업 스케줄을 작성하겠습니다.

[이벤트 처리형 프로그래밍 방식]

- 1 : 왼쪽 방향키를 누르면 → 캐릭터를 왼쪽으로 옮겨라.
- 2 : 오른쪽 방향키를 누르면 → 캐릭터를 오른쪽으로 옮겨라.
- 3 : 적과 부딪치면 → 캐릭터의 생명포인트를 삭감해라.

위에서 예를 든 두 가지 작업 스케줄 중 첫 번째 형식은 절차형 방식이며, 본 강좌에서 사용할 방식의 프로그래밍 기법입니다. 두 번째 형식은 이벤트 처리 방식입니다. 주로 윈도우 프로그래밍을 할 때 사용하는 방식입니다. 절차형 프로그래밍 방식이 순서대로 실행되는 반면, 이벤트 처리 방식은 “어떤 일이 발생하면 무엇을 한다”와 같은 형식으로 프로그램이 작동됩니다.

아직까지는 절차형 방식이 무엇인지 또는 이벤트 처리 방식이 무엇인지 알아두실 필요는 없습니다.

정리하면, 프로그래밍이란 여러분들이 원하는 결과물을 얻기 위해 컴퓨터에게 작업을 지시하는 활동입니다. 이제부터 저자는 어떻게 작업 스케줄을 작성하면 컴퓨터가 알아듣고 작업을 시작하는가에 대해서 설명할 것입니다.

컴퓨터에게 작업을 지시하기 위해서는 컴퓨터가 알아들을 수 있는 언어로 이야기하여야 합니다. 컴퓨터 프로그래밍 언어에는 여러 가지 종류가 있으며, 본 강좌에서는 파스칼 언어를 통해서 컴퓨터와 대화를 시도하도록 하겠습니다.

프로그래밍의 구성요소

어떤 컴퓨터 프로그래밍 언어를 사용해도, 작성된 모든 프로그램을 크게 두 가지로 나눌 수 있습니다. 또한 이것을 각각 다시 세부 분류로 나눌 수 있으며, 대부분의 컴퓨터 언어는 대동소이한 구조를 가지고 있습니다.

프로그램 = 코드(명령어) + 데이터(변수)

예를 들어 컴퓨터 모니터에 글자를 표시하고 싶다면, “화면에 글자 표시”라는 명령어와 “원하는 글자”에 해당하는 데이터를 컴퓨터에게 알려주면 되는 것입니다. (여기서 명령어란 컴퓨터가 알아들을 수 있는 지시어들의 집합을 말하며, 변수라는 것은 데이터를 저장할 수 있는 컴퓨터 프로그램 내부의 공간입니다.)

좀더 컴퓨터 프로그래밍에 가까운 형식으로 변환해서 설명하겠습니다. “화면에 글자 표시”라는 명령어는 파스칼 언어에서는 “WriteLn()”라는 명령어가 있습니다. 정확하게는 함수라고 하지만, 개념을 이해하는데 별로 도움이 되지 않기 때문에 우선 이것에 대한 설명은 넘어가도록 하겠습니다. “화면에 표시할 글자”는 큰 따옴표에 넣어서 표시합니다. 아직 여러분들이 문법을 배우지 않은 관계로 이것을 그냥 아래와 같이 나열하겠습니다.

WriteLn(), "즐거운 프로그래밍의 세계로 나가자!"

사실 위에서 작성한 프로그램은 작동하지 않습니다. 왜냐하면 문법을 어겼기 때문입니다. 사람들끼리의 대화라면 상대가 문법을 다소 잘못 사용했다고 해도 어렵짐작으로 알아들을 수 있는 경우가 많지만, 컴퓨터는 문법이 틀리면 전혀 알아듣지 못합니다.

가끔은 데이터가 필요 없는 명령어도 사용됩니다. 하지만, 명령어 없이 데이터만 사용되는 경우는 없습니다.

현실 상황에서 우리가 원하는 결과물은 상당히 복잡한 것들이 많아서 위에서처럼 간단한 문법체계만으로는 원하는 내용을 쉽게 표현할 수는 없습니다. 그래서, 컴퓨터 언어의 문법에도 다양한 구성요소가 있습니다.

컴퓨터 언어의 구성요소를 간략하게 표현해보면 다음과 같습니다.

컴퓨터 프로그래밍 언어의 구성요소

- 변수
 - 숫자형 변수
 - 정수형 변수 : byte, integer, etc
 - 실수형 변수 : real, double, etc
 - 문자형 변수
 - 문자 변수 : char
 - 문자열 변수 : string
 - 기타 변수
 - 포인터 : pointer, etc
 - 불린 : boolean
- 명령어
 - 선언문 : 변수선언문, 상수선언문, type 선언문, etc
 - 식별자를 정의하고, 해당 식별자가 어떤 의미를 가지는 지를 선언한다.
 - 대입문
 - 변수에 데이터를 복사하여 저장한다.
 - 연산문 : 산술연산문, 논리연산문, 비트연산문, 포인트 연산문
 - 지정된 계산을 하여 결과값을 만들어 낸다.
 - 입출력문 : 표준입출력문, etc
 - 주변 장치에서 데이터를 받아들이거나, 주변 장치로 데이터를 전송한다.
 - 반복문 : for loop, while do, repeat until
 - 특정한 문장이나 블록을 반복한다.
 - 조건문 : if, case
 - 조건을 검사하여 조건에 따라 복수의 문장이나 블록 중에 하나를 선택하여 실행한다.
 - 제어문 : goto, break, continue, exit
 - 프로그램 흐름을 제어하여, 흐름을 변경한다.

컴퓨터언어의 필요성

[리스트 1] 기계어 코드를 16 진수로 표현한 것

```
B80200  
050300
```

[리스트 2] 기계어 코드를 어셈블리 언어로 표현한 것

```
MOV AX, 0002  
ADD AX, 0003
```

[리스트 3] 파스칼 문법으로 표현 한 것

```
X := 2 + 3;
```

위의 모든 코드(소스)는 "2 더하기 3" 을 구하는 프로그램입니다.

[리스트 1]의 경우에는 컴퓨터가 실제로 사용하고 있는 형태입니다. 컴퓨터는 1 과 0 으로 된 2 진수 밖에 인식하지 못하기 때문에 실제로는 더 복잡하게 보입니다.

[리스트 2]는 [리스트 1]의 기계어 코드를 좀더 인간이 보기에 편하도록 만들어진 어셈블리언 언어로 표현한 것입니다. **MOV** 는 데이터를 이동하라 는 뜻을 가진 명령어 이며, **ADD** 는 주어진 데이터와 메모리공간(레지스터)에 있는 값을 더하라는 명령어 입니다. [리스트 1]과 완벽하게 같은 뜻이지만, 조금 읽고 이해하기가 편해졌습니다.

[리스트 3]의 경우에는 파스칼 문법을 통해서 더하기를 구현한 것 입니다. **2 + 3** 이라는 이해하기 쉬운 문법 체계로 구성되어 있기 때문에 인간이 이해하고 작성하기 쉽게 되어 있습니다.

하지만, 아직까지 컴퓨터는 [리스트 3]과 같은 문법을 이해하지 못하고 있습니다. 따라서, 지금까지 인간은 두 가지 해법을 찾아냈습니다.

첫 번째는, 인간이 직접 기계(컴퓨터)가 알아들을 수 있는 언어로 프로그래밍을 하는 것 입니다. 이것은 너무나 비효율적일 뿐만 아니라, 거대한 규모의 프로그램을 작성하는 것은 거의 불가능에 가깝습니다.

두 번째 방법은, 인간이 이해하기 쉬운 언어형태의 프로그래밍 언어를 만들고, 이 언어를 컴퓨터가 알아들을 수 있는 해석기(컴파일러)를 만드는 것 입니다.

오늘날 프로그래밍은 두 번째 방법에 의존하여 진행되고 있습니다. 지금부터 우리는 인간이 이해하기 쉽게 작성된 프로그래밍 언어 중에서도 오브젝트 파스칼에 대해서 공부하도록 하겠습니다. 파스칼은 교육용으로 제작된 언어이기 때문에 이해하기 쉽고 배우기 쉬운 장점이 있습니다. 또한, 이 책의 최종 목적인 델파이는 쉽게 배울 수 있으면서도 다양하고 강력한 프로그램을 작성할 수 있는 환경을 제공합니다. 델파이를 통해서 여러분들은 게임, 유틸리티, 업무용 프로그램, 시스템 프로그램 등 여러분들이 PC에서 사용하는 대부분의 프로그램을 스스로 작성할 수 있습니다.

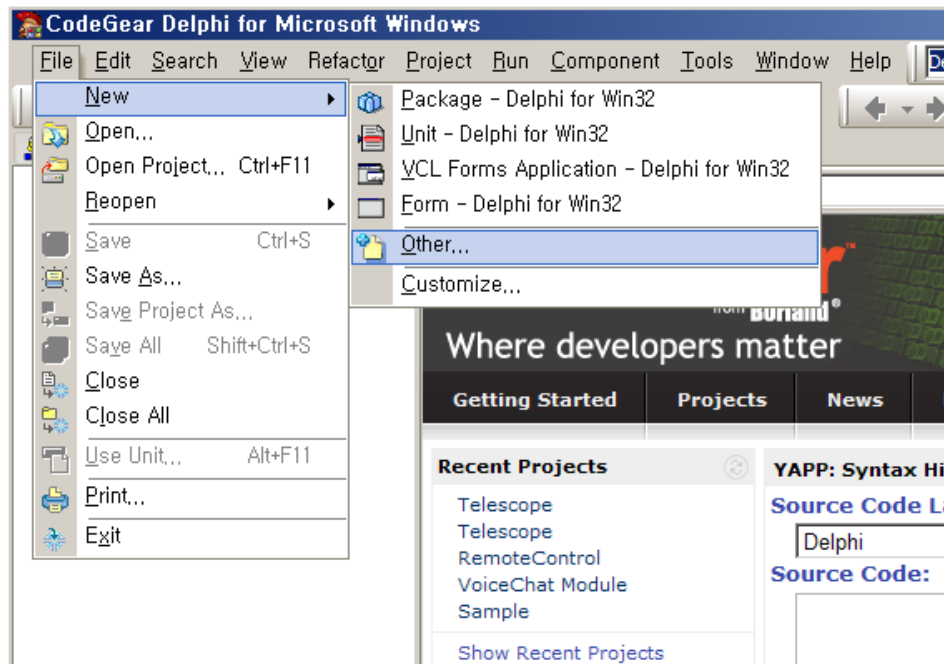
오브젝트 파스칼을 입문자를 위한 유용한 사이트

- 델파이 동영상 강좌 및 각종 자료 : <http://www.codeway.co.kr>
- 무료 델파이 다운받기 : <http://www.codegear.com/downloads/free/turbo>
- 무료 델파이 설치 설명서 : http://www.codeway.co.kr/board/bbs/tb.php/Delphi_Lecture/355
- 델파이 관련 사이트
 - 코드웨이 : <http://www.codeway.co.kr>
 - 델마당 : <http://www.delmadang.com>
 - 볼랜드 포럼 : <http://www.borlandforum.com>
 - 델파이 코리아 : <http://www.delphikorea.com>
 - 한델 : <http://www.delphi.co.kr>

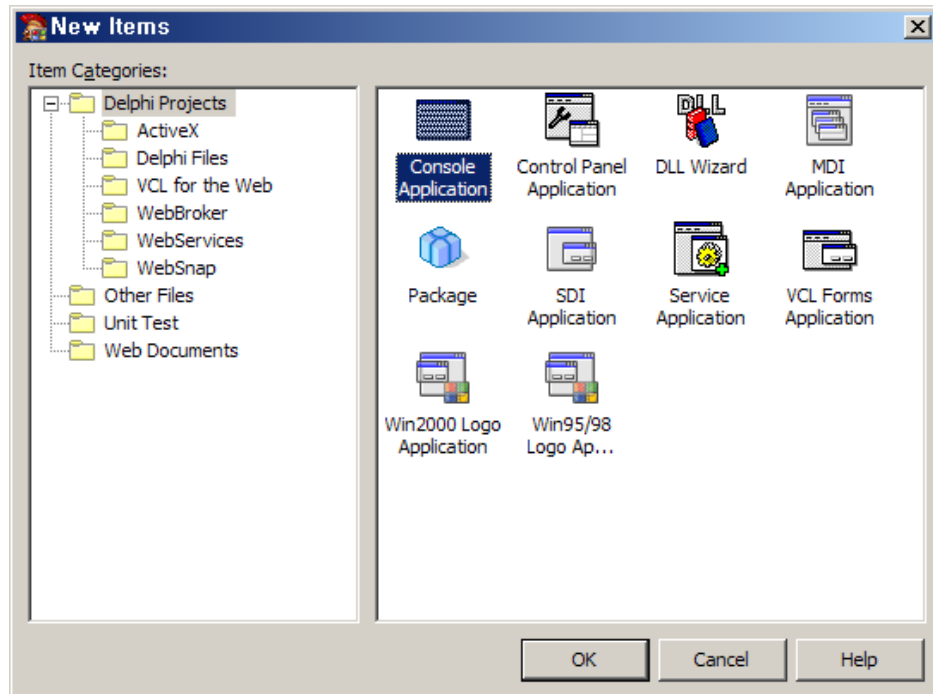
프로그램 소스 작성 및 실행

여러분들이 일반적으로 자주 사용하는 윈도우용 프로그램의 경우에는 구조가 다소 복잡하기 때문에, 문법을 배우시는 동안에는 콘솔에서 실행되는 프로그램을 작성하면서 연습을 하도록 하겠습니다. 콘솔용 프로그램은 예전의 도스용 프로그램과 비슷하다고 보면 됩니다.

우선 새로운 콘솔 프로그램을 시작하려면 [그림 1]과 [그림 2]와 같이 메뉴를 실행하시면 됩니다. 만약에 다른 파일들을 사용 중이었다면, 해당 파일을 모두 종료시켜 주시는 것이 좋습니다. 다른 파일이 열린 상태에서도 프로그래밍을 계속 하실 수 있지만, 초보의 경우에는 혼동되는 일이 자주 생기기 때문에 어느 정도 익숙해지시기 전까지는 기존의 파일은 닫는 것을 권장 합니다.



[그림 1] 메뉴실행



[그림 2] 콘솔 어플리케이션 선택하기

만약 프로그램 작성을 마치고 해당 소스를 저장하여 나중에 다시 사용하고 싶다면, [그림 3]과 같이 **Save All** 메뉴를 실행하시기 바랍니다. 간혹 여러분들이 작성하는 프로그램이 여러 개의 파일로 묶여질 수 있습니다. 이때, **Save** 와 같은 메뉴를 사용하면 저장되지 않는 파일이 생길 수 있습니다. 따라서, **Save All** 메뉴에 익숙해지실 것을 권장 합니다.

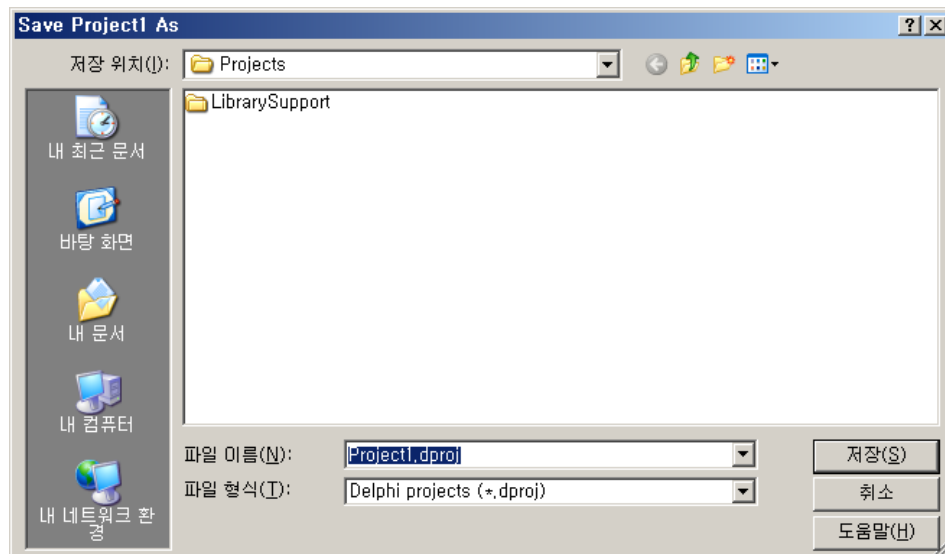
[그림 3] 에서 Project1.dproj 파일 이름 대신 여러분들이 원하시는 이름을 입력하시면 됩니다.

저장되는 파일은 델파이 프로젝트 파일이라고 부릅니다.

델파이로 프로그램을 작성하려면 우선 프로젝트 파일을 만드셔야 합니다.

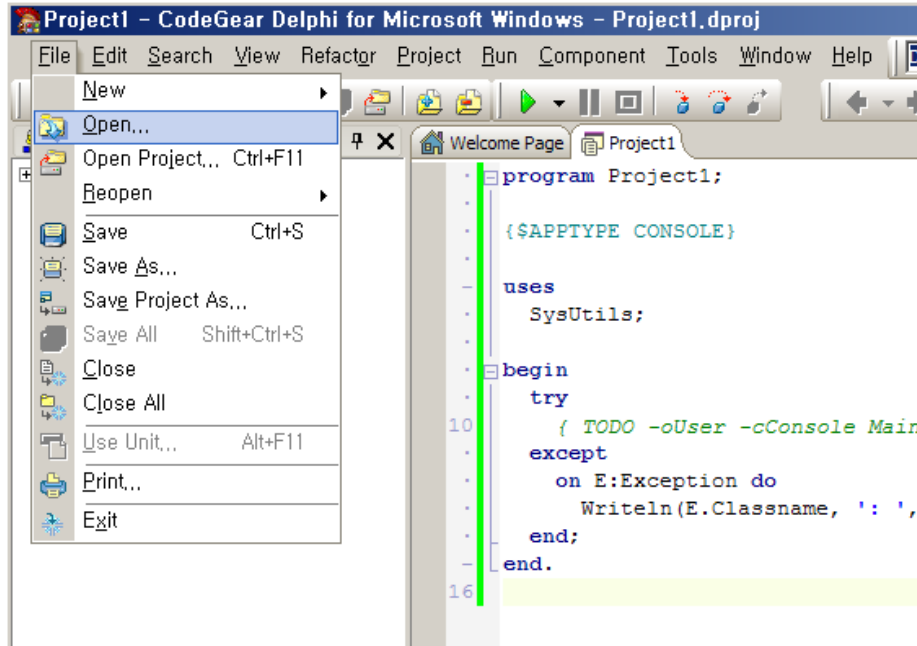
확장자가 dproj 또는 dpr 로 사용되는데 이것은 Delphi Project 의 약자로 사용된 것 입니다.

당분간 모든 예제는 프로젝트 파일 하나만을 이용하여 작성하게 될 것 입니다.



[그림 3] 소스 저장하기

이미 저장된 소스를 다시 불러와서 작업하고자 하실 때는 [그림 4] 와 같이 Open 메뉴를 실행하신 후 원하시는 파일을 선택하시면 됩니다.



[그림 4] 파일 읽기

프로그램의 실행은 소스가 열린 상태에서 **F9** 버튼을 누르시면 여러분들이 작성한 프로그램이 실행되어 결과를 확인하실 수 있습니다.

F9 버튼이 눌러지면 소스가 컴파일되어 실행할 수 있는 상태로 만들고 바로 컴파일된 프로그램을 실행하게 됩니다. 실행할 수 있는 상태로 변경된 것은 소스와 같은 폴더에 같은 이름을 가지고 확장자는 **exe** 인 실행파일로 저장이 됩니다. **Project** 옵션을 변경하면, 실행파일이 저장되는 위치를 변경하실 수 있습니다.

대부분의 프로그래밍 언어는 사람이 보기 쉽게 작성되어 있기 때문에, 컴퓨터가 직접 이해할 수는 없습니다. 따라서, 컴퓨터가 알아들을 수 있는 형태로 변환해야 하는데, 이것을 컴파일 이라고 부릅니다.

파스칼 프로그램의 구조

```
1 : program Begin01;  
2 :  
3 : {$APPTYPE CONSOLE}  
4 :  
5 : begin  
6 :  
7 : end
```

위의 소스가 일단 파스칼 프로그래밍의 가장 간단한 구조라고 생각하시면 됩니다.

파스칼의 모든 프로그램은 1: 라인과 같이 **program** 이라는 예약어로 부터 시작됩니다. 이제부터 프로그래밍을 시작하겠다는 뜻입니다. 바로 이어서는 프로그램 이름을 작성하시면 됩니다. 프로그램 이름은 파일이름과 동일하여야 하며, 텔파이가 스스로 변경하여 주시기 때문에 여러분들께서 수정하실 필요는 없습니다.

3: 라인에 있는 {} 안에 명령어는 우리가 작성하고 있는 프로그램이 콘솔에서 실행됨을 선언하기 위한 것입니다. 만약 {} 을 포함해서 3: 라인을 지워버리면, 일반적인 윈도우즈 프로그래밍으로 전환됩니다. 이부분은 사실 여러분들께서 프로그래밍을 하실 때 중요한 부분은 아닙니다. 하지만, 입문자가 윈도우즈 프로그래밍을 하는 것보다, 콘솔용 프로그램을 통해서 시작하는 것이 쉽기 때문에 선택한 것 일 뿐입니다.

5: - 7: 라인에서 **begin** 과 **end.** 는 프로그램의 시작과 끝부분을 선언하여 주는 예약어입니다. 프로그램은 **begin** 다음 라인에서 시작해서 **end.** 바로 이전 라인에서 중단됩니다. 또한, 파스칼 소스 파일의 마지막은 **end.** 끝나야 합니다. **end.** 이후의 글자들은 모두 없는 것으로 간주됩니다.

begin ... end 는 문장을 하나로 묶어주는 역할을 하며, 이것을 블록이라고 부릅니다.

블록 = begin + 문장들 + end;

예: Begin

```
WriteLn('1');
```

```
WriteLn('2');
```

```
end;
```

모든 문장 뒤에는 세미콜론(;)이 붙어 있는 것을 확인할 수 있습니다. 이는 컴퓨터(컴파일러)가 인식하는 한 문장입니다. 파스칼에서는 세미콜론(;) 문자가 나타나기 전에는 한 문장이 끝나지 않은 것으로 간주합니다. 한편, 세미콜론(;)은 문장 중간에 사용할 수 없습니다. 또한, 선언문과 블록의

시작을 알리는 문장에서도 세미콜론(;)을 사용하지 않습니다. 7: 라인에서처럼 파일의 마지막을 알리는 **end.**의 경우에서도 세미콜론(;) 대신 마침표(.)를 사용합니다.

주석

주석은 여러분들이 프로그램 소스에 설명을 달기 위해서 사용합니다. 컴퓨터(컴파일러)는 주석으로 처리된 문자열은 무시하게 됩니다. 당분간은 // 만을 사용할 것 입니다. 소스 중간에 // 이 나오면 그 뒤에 있는 문자열은 모두 무시하게 됩니다.

```
ReadLn;  
  
ReadLn;  // 잠시 대기
```

위의 두 문장은 같은 것으로 취급됩니다. // 를 포함해서 그 오른쪽에 있는 모든 문자열은 무시됩니다.

데이터 표시방법

데이터는 크게 숫자형 데이터와 문자형 데이터가 있습니다. 이것에 대해서는 나중에 변수를 설명드리면서 진행하도록 하겠습니다. 우선, 여기서는 데이터를 표시하는 방법에 대해서 알아보도록 하겠습니다.

```
1234 // 숫자형 데이터

'1234' // 문자형 데이터

abcd // 데이터가 아님

'abcd' // 문자형 데이터

1.234 // 숫자형 데이터

0.123E10 // 숫자형 데이터
```

숫자형 데이터는 숫자와 마침표(.)로 구성되어야 합니다. 단, 10의 제곱승으로 표현할 때에는 숫자 데이터 뒤에 붙여서 E10과 같이 사용하시면 됩니다. E10은 10의 10제곱승이라는 뜻입니다. 문자형 데이터는 작은 따옴표를 사용합니다. 숫자로 이루어졌다고 해도, 작은 따옴표 안에 있는 것들은 모두 문자형 데이터로 취급됩니다.

입출력문

이제 우리의 첫 번째 프로그래밍을 시작해보도록 하겠습니다. 우선은 화면에 "안녕하세요"라고 표시해보도록 하겠습니다. 앞서 설명한 것과 같이 **begin** 다음 라인에서 프로그램을 시작하여야 하기 때문에 화면에 '안녕하세요?' 라고 표시하라는 명령어를 사용해야 합니다. 이렇게 화면에 무엇인가 표시하는 명령어들을 우리는 출력문이라고 부릅니다.

출력문을 작성하는 문법형식은 아래와 같습니다.

문법

```
Write(출력할 데이터 또는 변수들의 배열);
```

```
WriteLn(출력할 데이터 또는 변수들의 배열);
```

사용의 예

```
WriteLn("문자열");
```

```
WriteLn(12345);
```

```
WriteLn("문자열", 123456, 7890);
```

```
WriteLn; // 아무것도 출력하지 않고 다음 줄로 넘어간다.
```

아직은 변수에 대해서 배우지 않았기 때문에 변수에 대한 사용법은 나중에 미루도록 하겠습니다. 출력문이 **Write** 와 **WriteLn** 두 가지가 있지만, 둘 사이의 차이점은 간단합니다. **WriteLn** 은 출력할 내용을 화면에 표시하고 엔터키를 친 효과를 나타냅니다. 즉, 다음 줄로 넘어가서 대기하게 됩니다. **Write** 는 출력한 데이터 바로 뒤에서 다음 출력을 위해 대기하게 됩니다.

```
1 : program Begin01;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : begin
6 :   WriteLn('안녕하세요?');
7 : end.
```

모두 작성이 끝났으면 텔파이 실행 상태에서 **F9** 키를 누르시면 해당 프로그램이 실행됩니다. 무엇인가 깜박하고 사라질 것입니다. 화면에 문자열을 출력하고 바로 프로그램이 **end.** 를 만나서 종료하기 때문에 프로그램이 화면에 보여지자마자 사라지기 때문입니다.

화면에 출력된 데이터를 확인하기 위해서는 어떻게 해야 할까요? 프로그램이 종료가 되기 전에 기다리게 하는 방법은 여러 가지가 있을 수 있겠지만, 그 중 한 가지는 바로 입력문을 사용하는 것입니다. 입력문을 사용하게 되면, 사용자가 입력을 마칠 때까지, 프로그램이 해당 라인에서 멈추게 됩니다.

입력문을 작성하는 형식은 아래와 같습니다.

문법

```
Read (데이터를 입력받을 변수들의 배열);
```

```
ReadLn (데이터를 입력받을 변수들의 배열);
```

사용의 예

```
ReadLn (Name);
```

```
ReadLn (Name, Age, Gender);
```

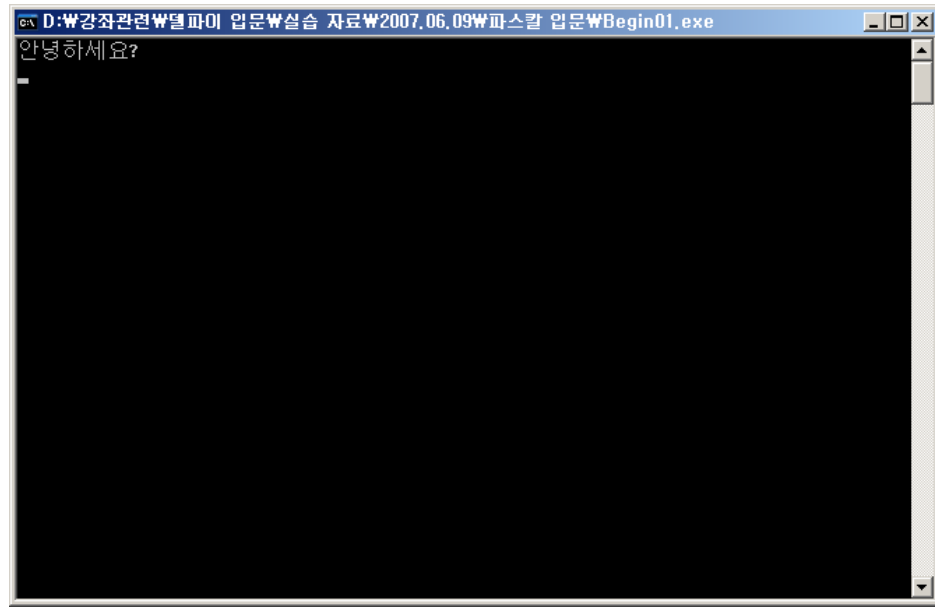
```
ReadLn; // 무엇을 입력받든지 무시하고 엔터키가 입력되면 멈춘다.
```

일단, 역시 변수에 대해서는 아직 배우지 않았기 때문에 그런 것이 있다 정도로 넘어가시기 바랍니다. 현재로서는 마지막에 있는 **ReadLn** 을 이용해서 프로그램을 잠시 멈추는 용도로만 사용하도록 하겠습니다. 파일에서 데이터를 읽어올 때가 아닌 이상, 화면상에서 데이터를 입력받을 때는, **Read** 와 **ReadLn** 의 차이점은 없습니다.

이제 아래와 같이 소스를 수정하고 **F9** 키를 누르시면 화면에 [그림 6] 과 같이 '안녕하세요?' 라는 문자열을 화면에 출력하고 엔터키가 입력될 때까지 기다리게 됩니다. 화면을 확인하셨으면, 엔터키를 치시기 바랍니다. 그러면, 텔파이로 되돌아 올 것입니다.

```
1 : program Begin01;  
2 :  
3 : {$APPTYPE CONSOLE}  
4 :  
5 : begin
```

```
6 :   WriteLn('안녕하세요?');  
7 :   ReadLn;  
8 : end.
```

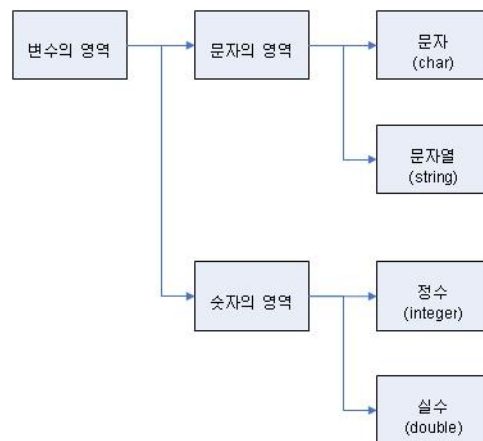


[그림 6] Begin01 의 실행 결과

변수

변수란 데이터를 저장하는 공간입니다. 이미 앞에서 거론했었던 것처럼, 컴퓨터 언어는 크게 명령어와 데이터로 나뉘집니다. 이 데이터를 처리하기 위한 저장 박스라고 생각하시면 됩니다. 또한, 데이터의 형태에 따라서 필요한 공간이 다르기 때문에 변수마다 크기가 서로 다릅니다. 만약 자신이 저장하고자 하는 데이터를 담을 수 있는 변수가 없다면, 스스로 새로운 변수를 만들어도 됩니다.

여기서는 파스칼에서 제공하는 기본적인 변수에 대해서만 알아보도록 하겠습니다.



[그림 7] 변수의 종류

일단 문자의 영역에는 영문자 기준으로 한 글자에 해당하는 **char** (문자) 형태(타입)이 있습니다. 즉, 문자(char) 타입 변수에는 영문자 하나만 저장할 수 있다는 뜻입니다. 한글의 한 글자는 영문자 두 자에 해당합니다. 한글은 모음과 자음이 혼합해서 수 만가지 글자를 만들어낼 수 있기 때문에 영문자보다 더 큰 공간이 필요합니다.

문자열(string) 타입의 변수는 여러 개의 문자로 이루어진 데이터를 처리하고자 할 때 사용합니다. 그 문자열의 크기는 기본적으로 영문자 한 개를 저장할 수 있는 1 바이트의 크기에서 2 기가 바이트까지 가능합니다. 문자열 타입의 변수 하나에 씨디의 내용 몇 장이 들어갈 정도이니, 충분한 공간이라고 할 수 있습니다.

물론, 문자열 변수를 사용하자 마자 그렇게 넓은 공간을 차지하지는 않습니다. 그렇게 되면 프로그램이 시작하자 마다 모든 메모리를 잠식하게 될 것입니다. 대신 문자열의 변수는 다른 변수와 달리 필요에 따라서 크기가 2 기가 바이트까지 늘어났다 줄었다 하는 것입니다. 하지만, 문자 타입의 변수는 그 크기가 항상 일정하게 1 바이트입니다.

분류	변수형태	크기(바이트)	범위
문자	char	1	#0 ~ #255
문자열	string	최대 2GB	

숫자의 영역에서는 크게 정수와 실수로 나누어 집니다. 숫자의 영역에서는 상당히 많은 분류의 변수 타입이 존재합니다. 하지만, 입문자가 그 모든 타입의 변수 종류를 외우실 필요는 없습니다. 여기서는 가장 많이 사용하는 몇 가지만 소개할 것입니다.

정수형 변수에서 가장 많이 사용하는 것은 단연 **integer** 타입입니다. 표현할 수 있는 크기는 약, 음수 20 억에서 양수 20 억 사이입니다. 정수 변수와 달리 소숫점 미만의 숫자도 표현하고자 할 때는 실수 타입의 변수를 사용하여야 합니다. 가장 많이 사용하는 것은 **double** 이며, 그 크기는 무려 10 의 308 제곱승 영역이나 됩니다.

분류	변수형태	크기(바이트)	범위
정수	integer	4	-2147483648 ~2147483647
실수	double	8	5.0*10E-324 ~ 1.7*10E308

소숫점 밑으로나 위로 10 의 몇 제곱승의 데이터를 표현한다고, 해도 표현할 수 있는 자릿수는 한정되어 있습니다. 이것을 **Significant digits** 라고 합니다. **double** 의 경우에는 그 크기가 15-16 자릿수입니다. 예를 들어 원주율의 경우 3.141592654... 이렇게 진행될 때 16 자리 수 미만의 숫자는 표현할 수 없다는 것입니다. 이것 때문에 실수의 계산에서는 오차를 가질 수밖에 없습니다.

변수 선언문

선언문에는 많은 종류가 있습니다. 여기서 우리는 변수 선언문부터 알아보도록 하겠습니다. 선언문이란 "무엇은 무엇이다" 라고 컴퓨터(컴파일러)에게 알려주는 문장입니다. 즉, 변수 선언문은 "이것은 변수다"라고 선언할 때 사용하는 문장입니다.

선언문을 사용할 수 있는 공간은 소스 내에서 한정되어 있습니다. 파스칼에서는 **begin** 과 **end** 밖에서 선언할 수 있다는 것을 알아두시기 바랍니다.

[리스트 Begin02]

```
1 : program Begin02;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : var
6 :   i : integer;
7 :
8 : begin
9 :   var // 컴파일 에러 발생
10 :     j : integer;
11 :
12 :   begin
13 :     var // 컴파일 에러 발생
14 :       k : integer;
15 :   end;
16 : end.
```

[리스트 Begin02]에서 5: - 6: 라인에서 선언한 정수형 변수 **i** 를 제외하고 나머지는 모두 문법적 오류이기 때문에 이 프로그램은 실행할 수 없습니다. 변수 **j** 의 경우에는 12: - 15: 라인의 **begin** 과 **end** 밖에 선언되어 있지만, 8: - 16: 라인의 **begin** 과 **end** 사이에 선언되어 있기 때문입니다.

end에는 **end.** 과 **end;** 가 있습니다. 위에서 마침표(.)를 **end** 뒤에 적지 않은 것은 두 가지의 경우 모두 해당하기 때문입니다.

단, **end.** 다음에 오는 모든 문자들은 무시되기 때문에 그 뒤에 선언할 수는 있지만, 아무런 의미는 없게 됩니다.

변수 선언문의 형식은 아래와 같습니다.

문법

```
var  
  
    변수이름 : 변수타입;  
  
    변수이름 1, 변수이름 2 : 변수타입;
```

사용의 예

```
var  
  
    Name : string;  
  
    Age, 몸무게 : integer;
```

변수이름은 식별자라고도 부릅니다.

식별자란 여러분들이 어떤 대상에게 지어준 이름을 뜻하는 것입니다.

함수 등의 이름도 식별자라고 부르며, 변수의 이름과 같은 규칙을 사용합니다.

텔파이 2005 버전 이상에서는 한글로도 변수이름을 지을 수 있습니다.

단, 식별자는 숫자와 특수문자로 시작해서는 안됩니다. 또한, 공백을 포함한 특수문자를 식별자에서 사용할 수 없습니다. (_ 는 사용할 수 있습니다)

이제 변수 선언문과 변수 타입을 이해하기 위해서 간단한 프로그램을 작성해보도록 하겠습니다. 프로그램의 목적은 여러문들의 이름과 나이, 그리고 몸무게를 입력하면, 그것을 다시 화면에 출력하는 것입니다.

[리스트 Begin03]

```
1 : program Begin03;  
2 :  
3 : {$APPTYPE CONSOLE}  
4 :
```

```

5 : var
6 :   Name : string;
7 :   Age, 몸무게 : integer;
8 :
9 : begin
10 :   Write(' 이름 : ');
11 :   ReadLn(Name);
12 :
13 :   Write(' 나이 : ');
14 :   ReadLn(Age);
15 :
16 :   Write(' 몸무게 : ');
17 :   ReadLn(몸무게);
18 :
19 :   WriteLn;
20 :
21 :   WriteLn('당신의 이름은 ', Name, '입니다. ');
22 :   WriteLn('나이는 ', Age, '살이고, 몸무게는 ', 몸무게, '입니다. ');
23 :
24 :   ReadLn;
25 : end.

```

5: - 7: 라인에서는 변수 선언이 이루어지고 있습니다. 6: 라인에서는 **Name** 이라는 문자열 변수가 선언되었습니다. 이 뜻은, 컴퓨터 메모리 영역 어딘가에 문자를 담을 수 있는 공간이 마련되어 있다는 뜻입니다. 여러분들은 **Name** 이라는 문자열 변수가 사용할 메모리 공간이 어디인지는 알 수 없지만, 이제부터는 **Name** 이라는 이름만 사용하면 컴퓨터가 알아서 해당 공간에 데이터를 저장하고 꺼내오게 됩니다.

변수라는 것은 메모리 공간의 주소의 별명과 같은 것입니다. 여러분들이 변수명을 사용하면, 컴퓨터는 그것을 숫자(메모리 주소)로 바꿔서 사용하게 됩니다.

10: 라인에서는 화면에 ' 이름 : ' 이라는 문자열을 표시하게 되며, 바로 문자열이 끝나는 다음에서 커서가 대기하게 됩니다.

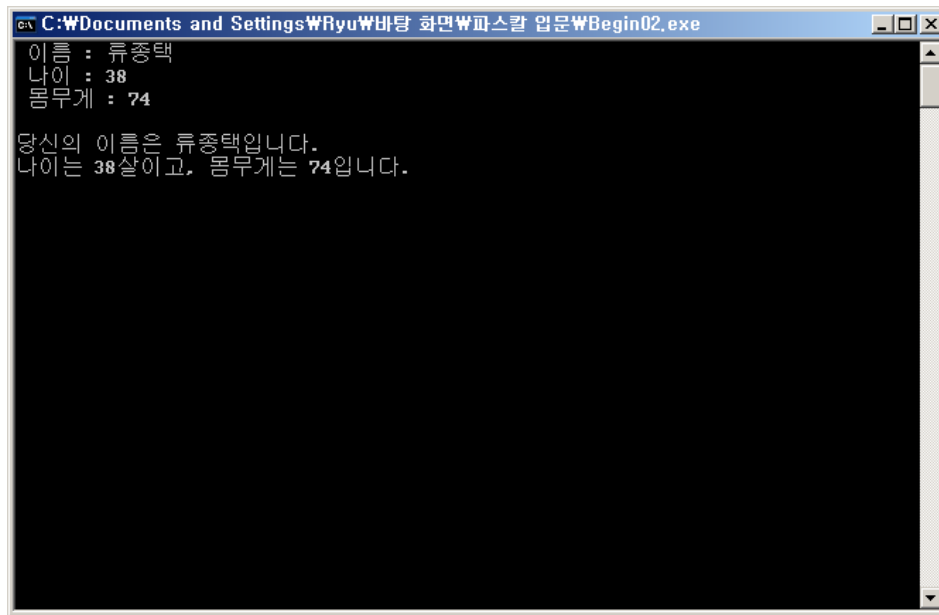
11: 라인에서는 커서의 위치에서부터 문자열을 읽은 후, **Name** 이라는 변수에 저장하게 됩니다.

14: 라인에서 **Age** 는 정수형 변수인 **integer** 타입이기 때문에, 만약 여러분들이 숫자 이외의 문자를 입력한다면 해당 라인에서 에러가 발생할 것입니다.

19: 라인은 다음 줄로 커서를 이동시키는 일만 하게 됩니다.

21: - 22: 라인에서는 각 변수에 있는 내용을 화면에 표시하되, 문자열로 설명을 덧붙여서 화면에 출력하고 있습니다.

24: 라인은, 이미 설명드렸던 것처럼, 잠시 화면을 대기상태로 두기 위해서 사용한 것입니다.



[그림 8] 리스트 Begin03 의 실행 결과

대입문

대입문이라는 것은 변수에 특정 데이터를 입력하기 위한 문장입니다. 형식은 아래와 같습니다.

문법

```
변수이름 := 데이터;  
변수이름 := 변수이름;  
변수이름 := 연산문;
```

사용법

```
Name := '류종택';  
Age := 38;  
Name := '류' + '종택'  
Name2 := Name;
```

대입문은 := 기호를 사용합니다. 해당 기호(대입문)의 의미는 "오른쪽의 데이터를 왼쪽에 있는 변수에 복사해 넣어라" 입니다. 대입문을 통해서 우리는 입력문 없이, 특정 변수 영역에 데이터를 입력할 수 있습니다. 연산문은 나중에 설명을 드리도록 하겠습니다.

이해를 돕기 위해서 간단한 프로그램을 작성해보도록 하겠습니다.

[리스트 Begin04]

```
1 : program Begin04;  
2 :  
3 : {$APPTYPE CONSOLE}  
4 :  
5 : var  
6 :   Name : string;  
7 :   Age, 몸무게 : integer;  
8 :
```

```

9 : begin
10 :   Name:= '류종택';
11 :   Age:= 38;
12 :   몸무게:= 74;
13 :
14 :   WriteLn('당신의 이름은 ', Name, '입니다.');
```

```

15 :   WriteLn('나이는 ', Age, '살이고, 몸무게는 ', 몸무게, '입니다.');
```

```

16 :
17 :   ReadLn;
18 : end.
```

[리스트 **Begin03**]에서는 키보드를 이용하여 사용자가 직접 **Name**, **Age**, 몸무게 라는 변수에 데이터를 입력하는 방식을 사용했었습니다. [리스트 **Begin04**]에서는 입력문을 통하지 않고 변수에 데이터를 입력하는 방식을 사용하고 있습니다. 실행하게 되면 [그림 8]과 같이 같은 결과를 얻을 수 있습니다.

산술 연산문

연산문은 데이터를 가공하는 문법입니다. 쉬운 예로는 더하기, 빼기, 곱하기, 나누기 등의 사칙연산을 예로 들 수도 있습니다. 연산문은 그 자체가 데이터 취급됩니다. 다만, 모든 연산(계산)을 마친 후 결과값을 데이터로 제공합니다.

```
5 // 숫자 데이터
```

```
2 + 3 // 산술 연산문
```

2+3은 연산 후에 결과값이 **5**이기 때문에, 위의 두 문장은 똑같이 **5**라는 숫자형 데이터로 취급됩니다.

여기서는 기본적인 산술 연산자 만을 설명하도록 하겠습니다.

+	더하기	
-	빼기	
*	곱하기	
div	정수 나누기	$3 \text{ div } 2 = 1$
/	실수 나누기	$3.0 / 2.0 = 1.5$
mod	나누기의 나머지	정수로 나누고 나머지를 반환 $5 \text{ mod } 3 = 2$

연 산문은 연산자들과 데이터 그리고 변수 등의 집합으로 이루어져 있습니다. 그리고 괄호 '()'를 통해서 어떤 연산을 먼저 수행할 것인지 결정할 수 있습니다. "곱셈이 덧셈보다 먼저 계산된다" 식의 연산자의 우선 순위 등은 우리가 수학시간에 배웠던 규칙이 그대로 적용됩니다.

이해를 돕기 위해서 간단한 프로그램을 작성해보도록 하겠습니다.

[리스트 05]

```
1 : program Begin05;  
2 :
```

```

3 : {$APPTYPE CONSOLE}
4 :
5 : var
6 :   i : integer;
7 :   d : double;
8 :
9 : begin
10 :   WriteLn('1 + 2 = ', 1 + 2);
11 :   WriteLn('2 * (1 + 2) = ' , 2 * (1 + 2));
12 :
13 :   d:= 3.14;
14 :   WriteLn('원주율 = ', d);
15 :
16 :   d:= 2 * d * 3;
17 :   WriteLn('지름이 3 인 원의 둘레 = ', d);
18 :
19 :   ReadLn;
20 : end.

```

10: 라인에서는 덧셈에 해당하는 연산자 + 를 사용한 결과를 표시하고 있습니다.

11: 라인에서는 좀더 복잡한 연산식으로 덧셈과 곱셈을 동시에 취급하고 있습니다. 일반적인 수학의 규칙에 따라 괄호는 다른 연산자보다 우선적으로 처리됩니다. 또한, 곱셈이 덧셈보다 먼저 처리되는 것도 일반적인 수학의 규칙과 같습니다.

```

C:\Documents and Settings\WRyu\바탕 화면\파스칼 입문\Begin04.exe
1 + 2 = 3
2 * (1 + 2) = 6
원주율 = 3.14000000000000E+0000
지름이 3인 원의 둘레 = 1.88400000000000E+0001

```

[그림 9] 리스트 Begin05 의 실행 결과

for loop 반복문

반복문 중에서도 가장 쉬운 **for loop** 반복문에 대해서 알아보도록 하겠습니다. **for loop** 반복문은 정해진 숫자만큼만 여러분들이 지정한 문장을 반복하는 문법입니다.

형식은 아래와 같습니다. 블록은 앞서 설명했었던 것과 같이 여러 문장을 하나로 묶어서 마치 하나의 문장인 것처럼 사용할 수 있는 문법입니다. **begin** 과 **end;** 를 사용하여, 묶고 싶은 문장들을 **begin** 과 **end;** 사이에 작성하시면 됩니다.

문법

```
for 변수이름 := 초기숫자 to 마지막숫자 do 문장;  
  
for 변수이름 := 초기숫자 to 마지막숫자 do 블록;  
  
for 변수이름 := 마지막숫자 downto 초기숫자 do 문장;  
  
for 변수이름 := 마지막숫자 downto 초기숫자 do 블록;
```

사용의 예

```
// 10 번 반복, i 숫자는 반복할 때마다 숫자 1 만큼 증가  
for i := 1 to 10 do WriteLn(i, '번째 반복 중');  
  
// 10 번 반복, i 숫자는 반복할 때마다 숫자 1 만큼 감소  
for i := 10 downto 1 do WriteLn(i, '번째 반복 중');
```

우선 **for loop** 문을 예제를 통해서 연습하도록 하겠습니다.

[리스트 Begin06]

```
1 : program Begin06;  
2 :  
3 : {$APPTYPE CONSOLE}
```



```

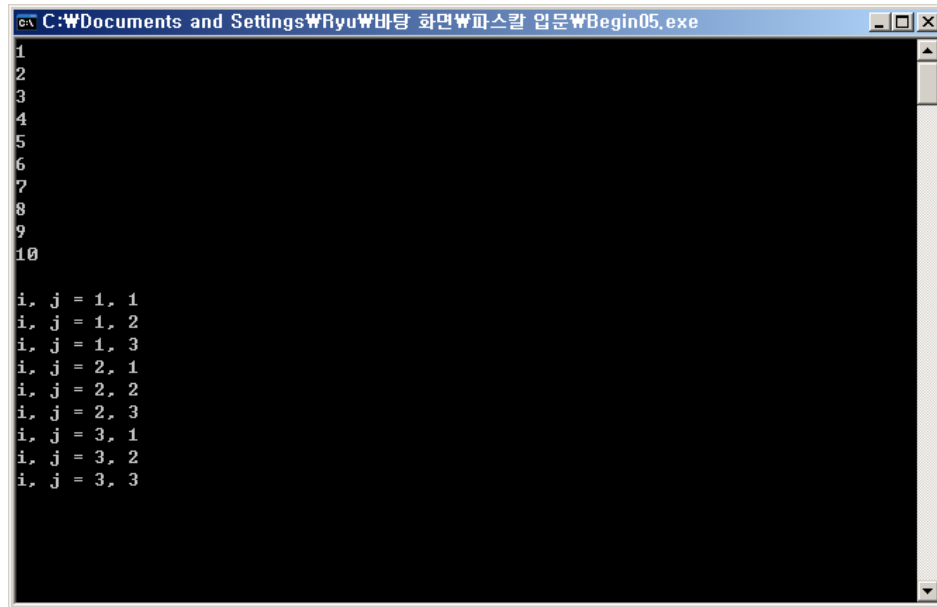
4 :
5 : var
6 :   i, j : integer;
7 :
8 : begin
9 :   for i := 1 to 10 do WriteLn(i);
10 :
11 :   WriteLn;
12 :
13 :   for i := 1 to 3 do
14 :     for j := 1 to 3 do
15 :       WriteLn('i, j = ', i, ', ', j);
16 :
17 :   ReadLn;
18 : end.

```

9: 라인에서는 정수형 변수 *i* 에 숫자 데이터 1 을 우선 입력한 후, `WriteLn(i)` 를 반복합니다. 한 번 반복할 때마다 *i* 값이 숫자 1 만큼 증가하게 되기 때문에 [그림 10]의 왼쪽과 같이 출력이 됩니다. 1 부터 10 까지 반복하기 때문에 [그림 10]에서도 1 부터 10 까지의 숫자가 한 줄에 하나씩 출력된 것을 확인할 수 있습니다. 즉, `WriteLn(i)` 가 10 번 반복하되, *i* 의 값은 1 부터 10 까지 순차적으로 증가하고 있는 것입니다.

13: - 14: 라인에서는 반복문이 두번 겹쳐 사용되는 예를 보였습니다. 13: 라인의 반복문이 *i* 의 값을 1 부터 3 까지 증가시키면서 세번 반복하게 됩니다. 그 세번 반복하는 대상이 14: 라인의 반복문이기 때문에 14: 라인의 반복문이 세번 반복하게 됩니다. 이때, 14: 라인의 반복문 또한 *j* 의 값을 1 부터 3 까지 증가시키면서 세번 반복합니다. 따라서, $3 * 3 = 9$, 9 번의 반복이 일어나게 되는 것입니다. 그 결과를 [그림 10]의 밑부분에서 확인할 수 있습니다.

14: - 15: 라인은 사람이 보기에는 두 줄이지만, 컴퓨터(컴파일러)가 생각하기에는 한 줄입니다. 파스칼에서는 세미콜론(;) 문자가 나타나기 전에는 한 줄이 끝나지 않은 것으로 간주합니다.



```
1
2
3
4
5
6
7
8
9
10
i, j = 1, 1
i, j = 1, 2
i, j = 1, 3
i, j = 2, 1
i, j = 2, 2
i, j = 2, 3
i, j = 3, 1
i, j = 3, 2
i, j = 3, 3
```

[그림 10] 리스트 Begin 06 실행 결과

조건문 if

프로그래밍에 있어서 조건문은 상당히 중요한 위치에 있습니다. 컴퓨터 프로그램이 마치 사람처럼 행동하기 위해서는 스스로 판단할 수 있는 능력이 필요하기 때문입니다. 자동으로 무엇인가를 처리한다는 것은 컴퓨터 또는 기계가 "어떠한 조건에서 어떻게 판단해야하는 지"를 알고 있어야 한다는 뜻입니다. 아직까지는 컴퓨터가 사람처럼 완전히 스스로 판단할 수는 없습니다. 따라서, 여러분들은 프로그램을 작성하면서, 어떠한 일이 벌어지면 어떻게 처리해야하는지 미리 프로그램을 통해서 컴퓨터에게 지시해 주어야 합니다.

```
if 조건연산문 then 문장;  
  
if 조건연산문 then 문장 else 문장;  
  
if 조건연산문 then 문장 else if 조건연산문 then 문장 else 문장;
```

위에서 문장은 블록으로 대체할 수 있습니다. 블록은 문장 취급한다고 이미 설명드린 것처럼, 문장이 오는 곳에는 블록도 같이 사용하실 수 있습니다.

if 문에서는 조건 연산문의 결과값이 참일 경우에만 then 다음의 문장이 실행되며, 조건 연산문의 값이 거짓인 경우에는 else 다음의 문장이 실행되게 됩니다. 사용법 및 예제에 대한 설명은 조건 연산문에 대한 이후에 하도록 하겠습니다.

조건 연산문

조 건 연산문은 어떠한 조건을 계산하여 그 값이 참(true)인지 거짓(false)인지를 파악하는 문법입니다. 조건 연산문도 연산문의 일종이기 때문에, 연산(계산)이 완료되면 특정한 데이터(값)로 취급됩니다. 수학의 명제와 유사하다고 생각하시면 됩니다.

우선 조건 연산문 중 하나인 관계 연산문을 살펴보면 다음과 같습니다.

=	같다	$a = b$, a 와 b 는 같다.
<	작다	$a < b$, a 는 b 보다 작다
>	크다	$a > b$, a 는 b 보다 크다
<=	작거나 같다	$a \leq b$, a 는 b 보다 작거나 같다
>=	크거나 같다	$a \geq b$, a 는 b 보다 크거나 같다
<>	같지 않다	$a \neq b$, a 와 b 는 같지 않다.

아래는 논리 연산문입니다.

not	아니다	not true = false
and	그리고	true and true = true true and false = false false and true = false false and false = false
or	혹은	true or true = true true or false = true false or true = true false or false = false
xor	배타선택	true xor true = false true xor false = true false xor true = true false xor false = false

조건 연산문은 주로 관계 연산과 논리 연산을 혼합해서 사용하게 됩니다. 논리 연산문이 없어도 연산이 가능합니다.

아래와 같이 간단한 예제를 통해서 살펴보도록 하겠습니다.

[리스트 Begin07]

```
1 : program Begin07;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   x, y, z : boolean;
10 :
11 : begin
12 :   x:= 1 = 1;
13 :   y:= 2 > 3;
14 :   z:= 2 <> 3;
15 :
16 :   WriteLn('x : ', x);
17 :   WriteLn('y : ', y);
18 :   WriteLn('z : ', z);
19 :
20 :   WriteLn('(x and y) or z : ', (x and y) or z);
21 :   WriteLn('((1 = 1) and (2 > 3)) or (2 <> 3) : ', ((1 = 1) and
(2 > 3)) or (2 <> 3));
22 :
23 :   WriteLn;
24 :
25 :   if x and y then WriteLn('True')
26 :   else WriteLn('False');
27 :
28 :   WriteLn;
29 :
30 :   if (1 = 1) and (2 > 3) then WriteLn('True')
31 :   else WriteLn('False');
32 :
33 :   ReadLn;
34 : end.
```

5-6: 라인에서 **uses** 는 C 언어에서의 **include** 와 유사하다고 보면 됩니다. 이미 작성된 소스를 함께 포함시켜서 컴파일하고자할 때 **uses** 를 사용합니다. **uses** 에 관해서는 8 장의 **Unit** 에서 자세히 설명하겠습니다.

9: 라인에서는 조건 연산문을 연산하고 난 결과값(조건값)을 저장할 수 있는 불린변수를 선언하는 과정입니다. 지금까지 배운 정수형 변수 타입이나, 문자열 변수 타입 등과 같이 파스칼에서 기본적으로 제공하는 변수타입 중에 하나입니다. 불린변수는 **true** 와 **false** 중 하나의 값을 저장할 수 있습니다.

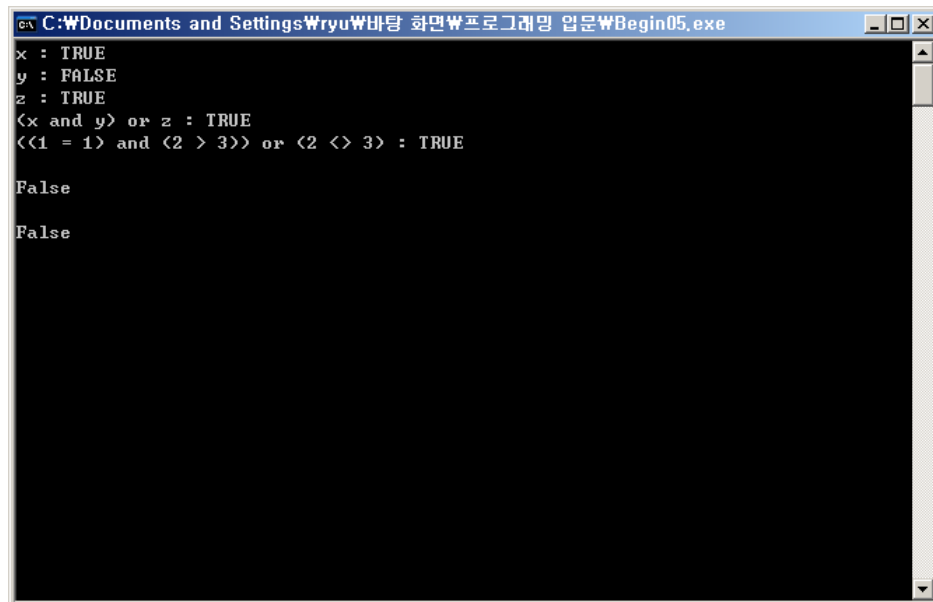
12: - 14: 라인에서는 각 변수에 조건(관계) 연산문을 통해서 데이터를 대입시키는 과정입니다.

16: - 18: 라인에서는 이들의 값을 화면에 출력하고 있습니다.

20: - 21: 라인에서는 같은 논리를 다르게 표현하면서, 그 결과값을 확인하기 위한 예제입니다. 두 라인은 결과값이 같습니다.

25: - 26: 라인에서는 조건 연산문을 if 문에서 어떻게 사용하는 지에 대한 간단한 예제를 보여주고 있습니다. 30: - 31 라인은 표현을 다르게 했을 뿐 같은 논리입니다. 결과값도 같습니다.

[리스트 Begin07]에서 보듯이 조건 연산문을 통하여 어떠한 조건들의 계산 값이 true 나 또는 false 나에 따라서 if 문의 then 다음의 문장이 실행되느냐 마느냐가 결정됩니다. 동반적으로 else 의 실행여부도 then 의 실행여부와 반대로 동작하게 됩니다.



```
C:\Documents and Settings\Wryu\바탕 화면\프로그래밍 입문\WBegin05.exe
x : TRUE
y : FALSE
z : TRUE
<x and y> or z : TRUE
<<1 = 1> and <2 > 3>> or <2 <> 3> : TRUE

False

False
```

[그림 11] 리스트 Begin07 의 실행 결과

Chapter

3

프로그래밍 예제 **#1**

프로그래밍 예제를 공부하시기 전에

영어를 공부할 때도, 우리는 가끔 듣는 말이 있습니다.

"단어를 문장 중에서 외워라."

필자가 권하는 입문자를 위한 프로그래밍 공부 방법도 마찬가지입니다. 너무 많은 것을 알기 위해 노력하기 보다, 조금씩 조금씩 예제를 통해서 기초를 다지시기 바랍니다. 처음에는 예제 소스를 이해하기 위해 반복하여 읽어보시기 바랍니다. 그리고, 따라하기를 한 번 진행해보시고, 자신이 이해한 것을 바탕으로 스스로 같은 예제를 작성해보시기 바랍니다.

적어도 입문자의 시기에서는 같은 내용이라고 해도 반복적으로 학습하는 것이 중요합니다.

외웠기 때문일지라도, 자신의 힘으로 예제의 내용을 스스로 작성할 수 없다면, 아직 그 지식은 여러분들의 것이라고 할 수 없습니다. 모든 단계의 공부를 그렇게 할 수는 없지만, 적어도 입문 공부만큼은 그렇게 하시기를 권해드립니다. 스스로 작성하실 때, 자신의 아이디어를 섞어서 변형시키신다면 금상첨화입니다.

아무리 간단한 것도 "알 것 같은 때"에 멈춰서면, 어느 한계점 이상 성장할 수 없는 쪽정이 프로그래머가 될 수 있습니다.

프로그래밍의 절차

프로그래밍을 하는 방법론은 상당히 많지만, 교재의 특성에 맞게 아래와 같은 순서로 프로그래밍을 설명 드리도록 하겠습니다.

- 기능분석
- 구현방법 설계
- 프로그램 작성
- 테스트 및 수정
- 완료

기능분석이란, 여러분들이 작성해야 할 프로그램이 가져야 할 기능을 나열하는 것입니다. 기능분석은 여러분들이 작성할 프로그램의 규모나 성격을 파악하는데 상당한 도움이 됩니다. 구현방법 설계는, 구체적으로 프로그램을 어떻게 동작하도록 할 것인가에 대한 동작 설계입니다. 다른 용어로는 동적분석이라고도 합니다. 본 교재에서는 플로우 차트를 사용하겠습니다. 테스트 및 수정에서는 작성된 프로그램을 동작시켜서, 여러분들이 원하는 결과를 얻을 수 있는 지 확인하고 문제점을 수정하는 과정입니다.

1 부터 100 까지 더하기

기능분석

1 부터 100 까지 더하기 프로그램이 가져야 할 기능은 다음과 같습니다.

- 1 부터 100 까지 합을 연산한다.
- 결과값을 화면에 출력한다.

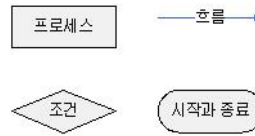
기능분석은 프로그램을 작성하기 전에 필요한 기능들을 나열함으로써, 작성하고자 하는 프로그램의 이해를 높이는 과정입니다. 본 예제는 간단한 편이기에 그 필요성이 느껴지지 않을 수 있지만, 실무에서는 종종 처음에 놓쳤던 요구사항을 나중에 알게 되어 개발이 힘들어지는 경우가 많습니다.

만약 이번 예제에서도 기능분석을 하지 않았다면, 1 부터 100 까지 더하기만 하고 결과값을 출력하는 것을 빼먹는 경우도 가정할 수 있습니다. 컴퓨터 내부에서는 계산이 되었지만 이것을 알 방법이 없다면, 프로그램은 별로 쓸모 없는 존재가 될 것입니다.

기능분석은 프로그램 작성 목표를 설정하는 과정이라고 생각하시면 됩니다.

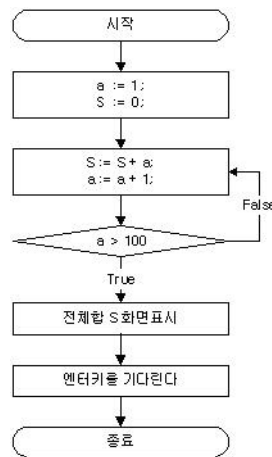
구현방법 설계

구현방법을 설계하는 가장 기본적이고 간단한 방법은 플로우 차트를 그리는 방법입니다.



[그림 12] 플로우 차트의 기본 구성 요소

[그림 12]은 플로우 차트의 기본 구성요소입니다. 출력과 같은 요소 등 다른 종류의 요소들도 존재하지만, 기본적인 것만을 표시했습니다. 그리고, 화면 출력과 같은 요소의 경우는 프로세스의 종류라고 생각할 수 있기 때문에 생략하였습니다.



[그림 13] 1 부터 100 까지 더하기의 플로우 차트

1 부터 100 까지 더하는 것에 대한 결과값은 순차수열에 관한 공식으로 쉽게 구할 수 있으나, [그림 13]에서는 실습의 목적상, 프로그램을 통하여 그것을 일일이 더하는 방법을 설명하고 있습니다.

우선 우리가 구하려는 합을 **S** 라고 하면,

$$\begin{aligned}
 a_n &= n \\
 S_n &= 1 + 2 + 3 \dots + n \\
 S_{100} &= 1 + 2 + 3 \dots + 100
 \end{aligned}$$

즉, a_n 항의 n 을 1 부터 100 까지 순차적으로 더해나가면 됩니다.

플로우 차트에서 보면 **S100**을 **S**라는 변수로, **an**을 **a**라는 변수로 지정하여 표현했습니다. 또한 변수 **a**는 1씩 증가하다가 100이 넘으면 반복되는 루프를 벗어나고 결과값을 화면에 표시한 다음 엔터키를 기다렸다가 프로그램이 종료되도록 되어 있습니다.

반복되는 동안에는

```
S := S + a;
```



와 같이 각 항이 계속 **S**라는 변수의 현재 값에 더해지게 되어 결과적으로는

```
S100 = 1 + 2 + 3 ... + 100
```

과 같은 결과값이 **S**라는 변수에 저장될 것입니다.

물론 순차순열의 합을 구하는 공식($S = n*(n+1)/2$)을 사용하면 쉽게 구할 수 있는 답입니다. 하지만, 여기서는 프로그램을 통해서 직접 더해보면서 반복문 등에 대한 복습을 하는 것을 목표로 하고 있습니다.

구현

텔파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin08** 로 저장하시기 바랍니다.

이후, 아래와 같이 코드를 작성하시기 바랍니다.

[리스트 Begin08]

```
1 : program Begin08;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Loop, Sum : integer;
10 :
11 : begin
12 :   Sum:= 0;
13 :
14 :   for Loop:= 1 to 100 do Sum:= Sum + Loop;
15 :
16 :   WriteLn('1 부터 100 까지의 합은 : ', Sum);
17 :
18 :   ReadLn;
19 : end.
```

12: 라인에서 **Sum** 변수에는 0 을 대입시켰습니다.

14: 라인에서는 **for loop** 문을 이용하여 1 부터 100 까지 반복하고 있습니다. 반복하면서 실행하는 문장은 **Sum:= Sum + Loop;** 입니다. 즉, **Sum** 은 자기 자신이 현재 가지고 있는 값에 **Loop** 의 값을 더한 값으로 대체됩니다. **Loop** 는 1 부터 100 까지 변하고 있기 때문에 아래와 같은 프로세스가 진행됩니다.

Loop = 1 일 때, **Sum:= Sum + 1;** 12: 라인을 통해서 **Sum** 에는 0 이 입력되어 있으므로, 이것은 결과적으로 **Sum:= 0 + 1;** 과 같습니다.

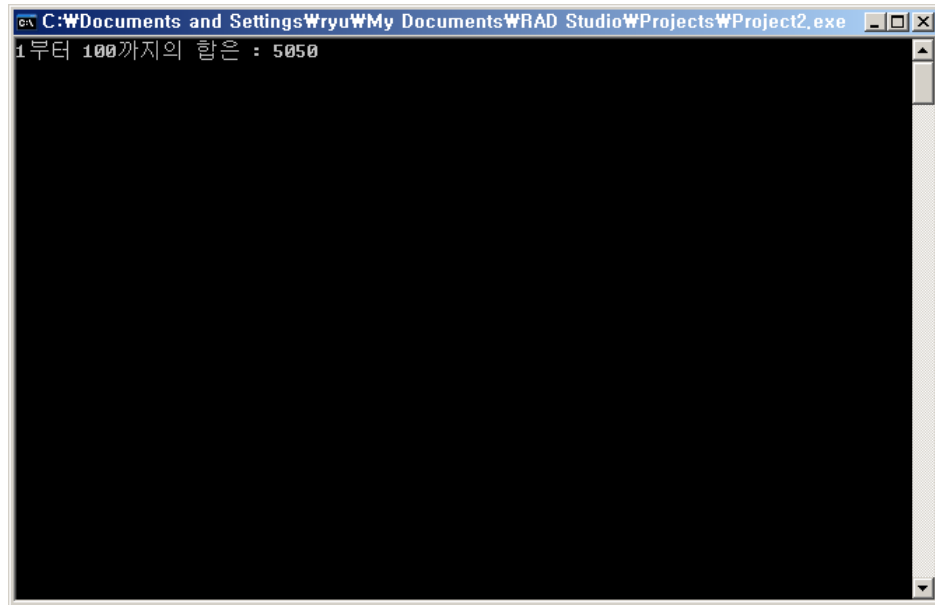
Loop = 2 일 때, **Sum:= Sum + 2;** 위에서 **Sum** 은 **Sum:= 0 + 1;** 이었으므로, 이것은 결과적으로 **Sum:= 0 + 1 + 2;** 과 같습니다.

Loop = 3 일 때, **Sum:= Sum + 3;** 위에서 **Sum** 은 **Sum:= 0 + 1 + 2;** 이었으므로, 이것은 결과적으로 **Sum:= 0 + 1 + 2 + 3;** 과 같습니다. 결과적으로, **Sum** 에는 1 부터 100 의 합이 저장되게 됩니다.

플로우 차트를 최대한 그대로 코드로 옮기면 아래와 같습니다. 하지만, **Goto** 문의 경우에는 프로그램을 복잡하게 만드는 근원으로 최대한 사용하지 않는 것이 좋습니다. 어쨌든, 두 프로그램 모두 같은 결과를 갖게 됩니다.

[리스트 Begin09]

```
1 : program Begin09;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Loop, Sum : integer;
10 :
11 : label
12 :   Ing;
13 :
14 : begin
15 :   Sum:= 0;
16 :   Loop:= 1;
17 :
18 :   Ing:
19 :     Sum:= Sum + Loop;
20 :     Loop:= Loop + 1;
21 :
22 :   if Loop <= 100 then Goto Ing;
23 :
24 :   WriteLn('1 부터 100 까지의 합은 : ', Sum);
25 :
26 :   ReadLn;
27 : end.
```



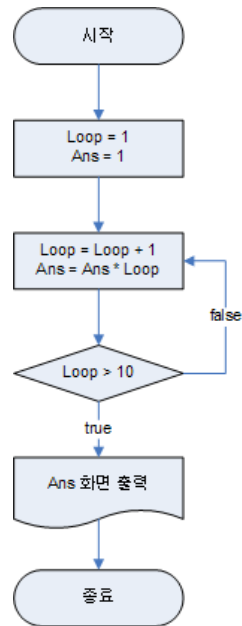
[그림 14] 리스트 Begin08 의 실행 결과

1 부터 10 까지 곱하기

기능분석



- 1 부터 10 까지 곱한다.
- 결과값을 화면에 출력한다.

구현방법 설계



[그림 15] 1 부터 10 까지 곱하기 플로우 차트

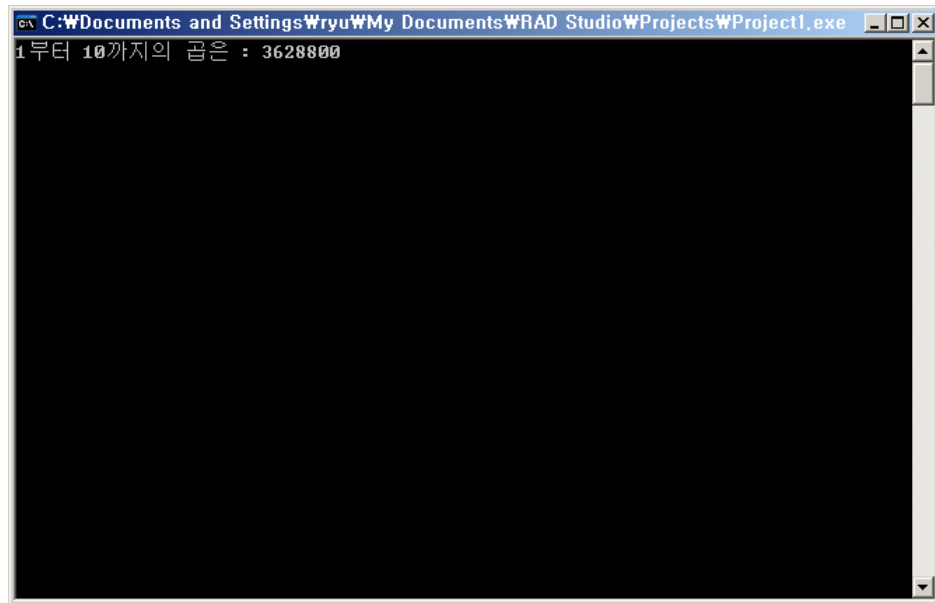
구현

텔파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin10** 로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다.

[리스트 Begin10]

```
1 : program Begin10;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Loop, Ans : integer;
10 :
11 : begin
12 :   Ans:= 1;
13 :
14 :   for Loop:= 1 to 10 do Ans:= Ans * Loop;
15 :
16 :   WriteLn('1 부터 10 까지의 곱은 : ', Ans);
17 :
18 :   ReadLn;
19 : end.
```

실행결과



[그림 16] 리스트 Begin10 의 실행 결과

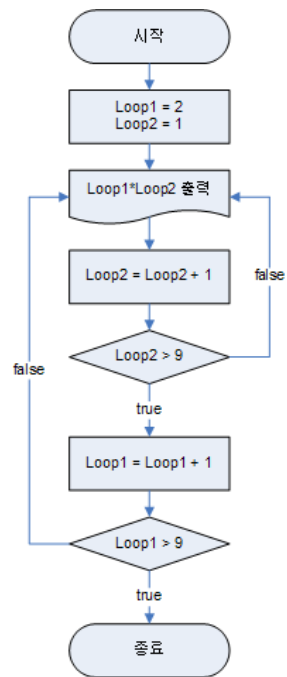
구구단 구하기

기능분석

- 구구단의 2 단 부터 9 단까지를 아래의 포맷으로 출력하는 프로그램을 작성하라.



```
2 X 1 = 2
...
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
...
9 X 9 = 81
```

구현방법 설계



[그림 17] 구구단 구하기 플로우 차트

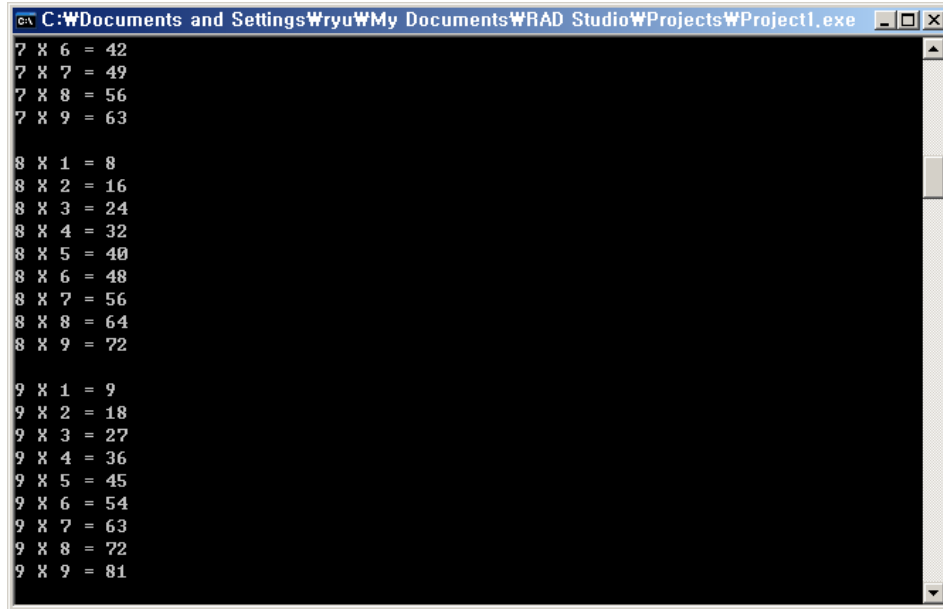
구현

텔파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin11** 로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다.

```
[리스트 Begin11]

1 : program Begin11;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Loop1, Loop2 : integer;
10 :
11 : begin
12 :   for Loop1:= 2 to 9 do begin
13 :     for Loop2:= 1 to 9 do
14 :       WriteLn(Loop1, ' X ', Loop2, ' = ', Loop1*Loop2);
15 :     WriteLn;
16 :   end;
17 :
18 :   ReadLn;
19 : end.
```

실행결과



```
C:\Documents and Settings\Wryu\My Documents\WRAD Studio\Projects\Project1.exe
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63

8 X 1 = 8
8 X 2 = 16
8 X 3 = 24
8 X 4 = 32
8 X 5 = 40
8 X 6 = 48
8 X 7 = 56
8 X 8 = 64
8 X 9 = 72

9 X 1 = 9
9 X 2 = 18
9 X 3 = 27
9 X 4 = 36
9 X 5 = 45
9 X 6 = 54
9 X 7 = 63
9 X 8 = 72
9 X 9 = 81
```

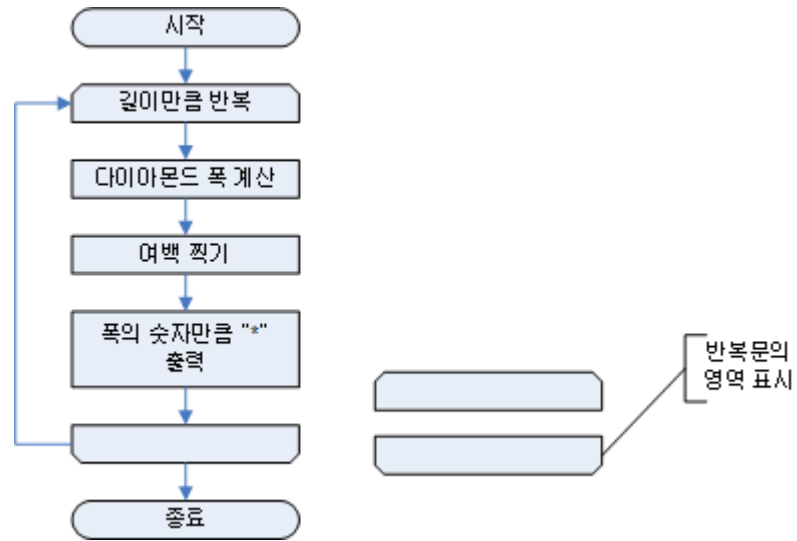
[그림 18] 리스트 Begin11 의 실행 결과

화면에 다이아몬드 표시하기 구구단 구하기

기능분석



- 별표 문자(*)를 이용해서, 화면에 [그림 20]과 같이 다이아몬드 모양을 출력한다.

구현방법 설계



[그림 19] 플로우 차트

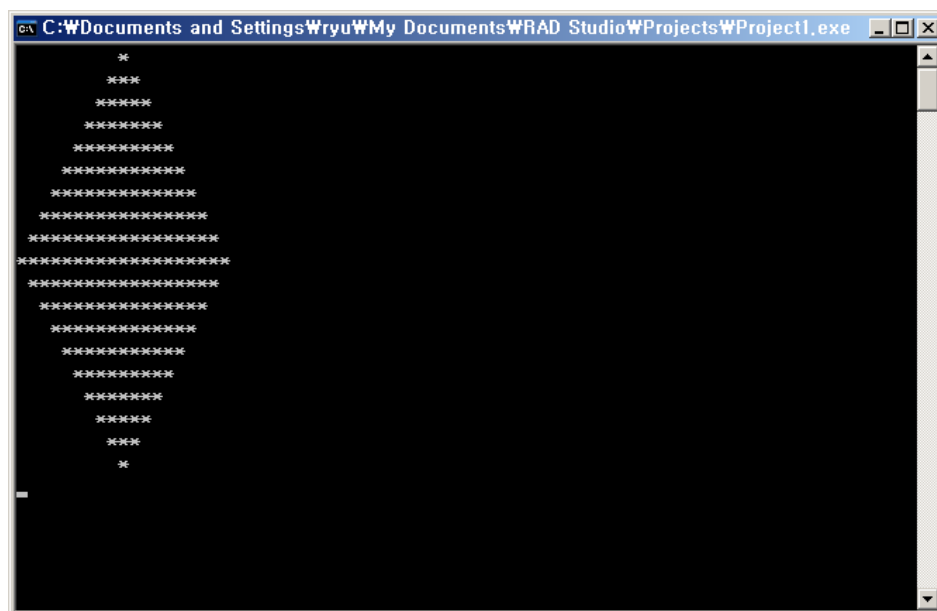
구현

델파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin12** 로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다.

[리스트 Begin12]

```
1 : program Begin12;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : const
9 :   Length = 19;
10 :
11 : var
12 :   Loop1, Loop2, Middle, Index, Width : integer;
13 :
14 : begin
15 :   Middle:= (Length div 2) + 1;
16 :
17 :   for Loop1:= 1 to Length do begin
18 :     Index:= Middle - abs(Middle-Loop1);
19 :     Width:= (Index-1)*2 + 1;
20 :
21 :     for Loop2:= 1 to Middle-Index do Write(' ');
22 :
23 :     for Loop2:= 1 to Width do Write('*');
24 :
25 :     WriteLn;
26 :   end;
27 :
28 :   ReadLn;
29 : end.
```

실행결과



[그림 20] 리스트 Begin12 의 실행 결과

2 장의 "기초적인 파스칼 문법 익히기"에서는 초보자들이 부담없이 공부를 할 수 있도록 최소한의 문법을 통해서 예제 프로그램을 작성할 수 있는 과정까지 설명하였습니다.

이번 장에서는 2 장에서 간단한게 지나친 것들을 보충하는 시간을 갖도록 하겠습니다.

while do 반복문

while do 반복문은 주어진 조건이 만족하는 동안 반복을 계속하고자 할 때 사용합니다.

문법

```
while 조건 do 문장;  
  
while 조건 do 블록;
```

사용의 예

```
i:= 0;  
  
while i < 10 do i:= i + 1;  
  
while i < 20 do begin  
  
    WriteLn(i);  
  
    i:= i + 1;0  
  
end;
```

[리스트 Begin13]은 while do 반복문의 사용에 대한 예제입니다. [리스트 Begin13]과 같은 경우에는 for loop 반복문을 이용하여 작성하는 것이 훨씬 간편합니다.

주로 while do 반복문은 반복 횟수보다는 반복할 조건을 검사할때 더욱 자주 사용하게 됩니다.

[리스트 Begin13]

```
1 : program Begin13;  
2 :  
3 : {$APPTYPE CONSOLE}  
4 :  
5 : var  
6 :     i : integer;
```

```

7 :
8 : begin
9 :   i:= 0;
10 :   while i < 10 do begin
11 :     WriteLn('i = ', i);
12 :     i:= i + 1;
13 :   end;
14 :
15 :   ReadLn;
16 : end.

```

[리스트 Begin14]은 while do 반복문을 이용하여 원주율을 구하는 예제 입니다.

[리스트 Begin14]

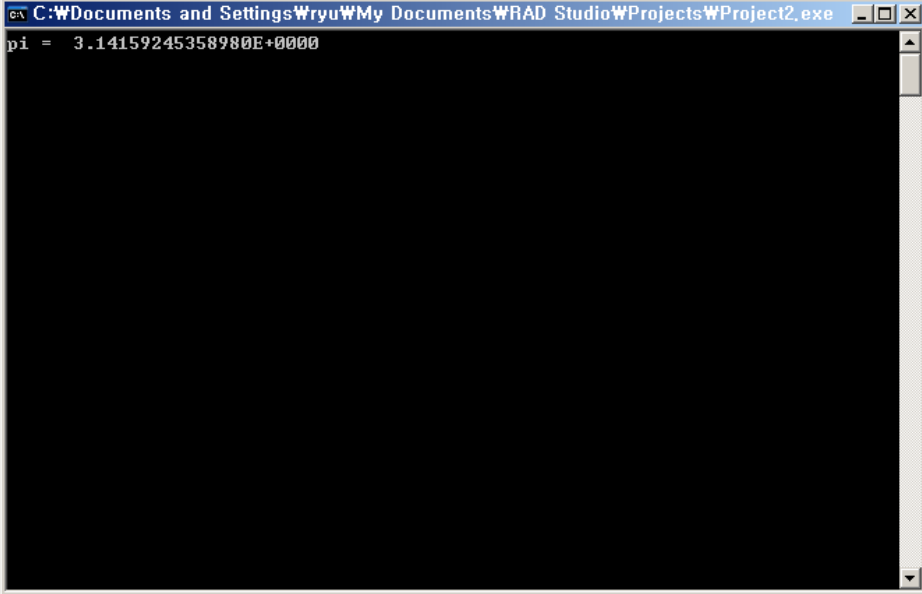
```

1 : program Begin14;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   pi, n, p : double;
10 :
11 : begin
12 :   // pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - ....
13 :
14 :   pi:= 0;
15 :   n:= 1;
16 :   p:= 1;
17 :
18 :   while Abs(1/n) > 0.0000001 do begin
19 :     pi:= pi + p/n;
20 :
21 :     n:= n + 2;
22 :     p:= -p;
23 :   end;
24 :
25 :   WriteLn('pi = ', pi * 4);
26 :
27 :   ReadLn;
28 : end.

```

원주율을 구하는 방법은 여러 가지가 있지만, 12: 라인의 주석과 같이 분수들의 무한급수의 합으로 구할 수도 있습니다.

18: 라인에서는 반복을 언제 멈출 것인지를 결정하는 것으로 개별 분수의 값이 0.0000001 보다 작거나 같을때 반복을 중단하게 됩니다.



```
C:\Documents and Settings\Wryu\My Documents\WRAD Studio\Projects\Project2.exe
pi = 3.14159245358980E+0000
```

[그림 21] 리스트 Begin14 의 실행 결과

repeat until 반복문

repeat until 반복문은 while do 반복문과 동일한 원리입니다. 반복 횟수가 아닌 조건이 만족할 때까지 반복을 합니다. while do 반복문과 다른 점은 두 가지 입니다. 첫째, 조건이 만족될 때 반복문을 중단합니다. 둘째, 문장이나 블록을 우선 실행하고 난 뒤에, 조건의 검사를 합니다.

두번째 차이점으로 인해 while do 반복문은 문장이나 블록이 한 번도 실행되지 않을 수도 있지만, repeat until 반복문은 문장이나 블록을 우선 한 번은 실행 한 뒤에 반복을 계속 할 것인지 결정하게 됩니다.

문법

```
repeat
    문장 또는 블록;
until 조건;
```

사용법

```
i := 0;

repeat
    WriteLn(i);
until i = 9;
```

[리스트 Begin15]는 [리스트 Begin14]을 그대로 repeat until 반복문을 이용하여 작성한 것입니다. 23: 라인에서 조건이 반대인 것에 유의하시기 바랍니다.

[리스트 Begin15]

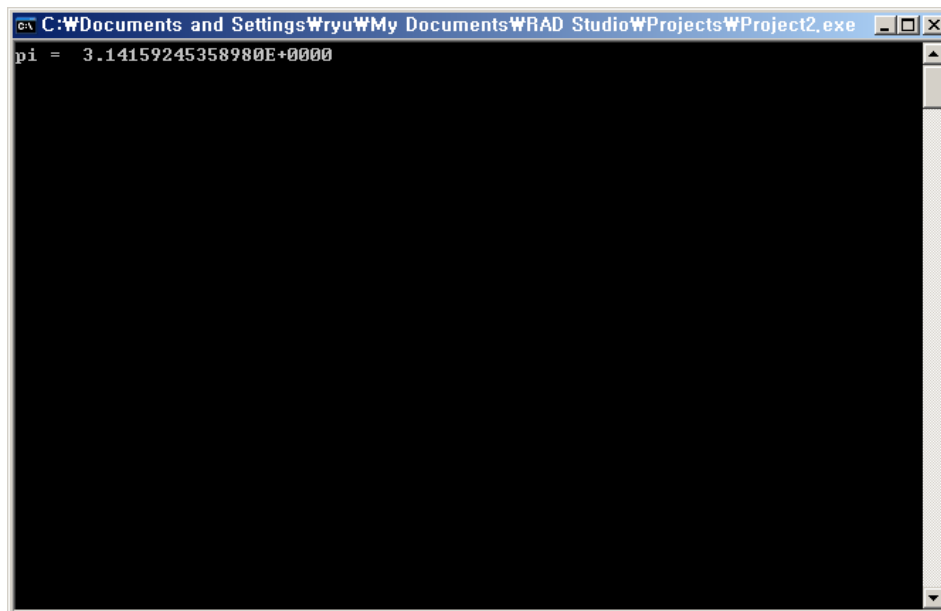
```
1 : program Begin15;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :     SysUtils;
7 :
```



```

8 : var
9 :   pi, n, p : double;
10 :
11 : begin
12 :   // pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - ....
13 :
14 :   pi:= 0;
15 :   n:= 1;
16 :   p:= 1;
17 :
18 :   repeat
19 :     pi:= pi + p/n;
20 :
21 :     n:= n + 2;
22 :     p:= -p;
23 :   until Abs(1/n) <= 0.0000001;
24 :
25 :   WriteLn('pi = ', pi * 4);
26 :
27 :   ReadLn;
28 : end.

```



[그림 22] 리스트 Begin15 의 실행 결과

goto 제어문

이동할 곳에 레이블 표시를 해두고, 해당 레이블로 이동하도록 합니다. `goto` 근래에 들어오면서 사장되어가는 문법입니다. `goto` 문을 사용하면 소스코드의 품질이 떨어지게 됩니다. 최대한 사용하지 않는 것이 좋습니다.

[리스트 Begin16]

```
1 : program Begin16;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : label
9 :   GotoHere;
10 :
11 : begin
12 :   goto GotoHere;
13 :
14 :   WriteLn('이것이 화면에 출력될까요?');
15 :
16 :   GotoHere;
17 :
18 :   ReadLn;
19 : end.
```

프로그램 11: 라인부터 실행됩니다. 12: 라인에서 `goto` 문을 만나면 뒤에 표시된 `GotoHere` 라는 레이블이 표기된 16: 라인으로 이동하게 됩니다. 따라서, 14: 라인은 실행되지 않습니다.

break 제어문

반복문을 빠져나가고 싶을 때 사용합니다.

[리스트 Begin17]

```
1 : program Begin17;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   i : integer;
10 :
11 : begin
12 :   for i:= 0 to 9 do begin
13 :     if i = 3 then Break;
14 :     WriteLn(i);
15 :   end;
16 :
17 :   ReadLn;
18 : end.
```

12: 라인에서보면 0 부터 9 까지 10 번의 반복이 이루어져야 합니다. 하지만, **13:** 라인에서 i 값이 3 일 때 **Break** 를 통해서 반복문을 탈출하게 됩니다. 따라서, 4 번의 반복이 있는 후에 **16:** 라인으로 이동합니다.

16: 라인에는 아무것도 없기 때문에 결과적으로 **17:** 라인이 실행됩니다.

continue 제어문

반복문의 반복을 한 번 지나치고 싶을 때 사용합니다.

[리스트 18]

```
1 : program Begin18;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   i : integer;
10 :
11 : begin
12 :   for i:= 0 to 9 do begin
13 :     if (i mod 2) = 0 then Continue;
14 :     WriteLn(i);
15 :   end;
16 :
17 :   ReadLn;
18 : end.
```

12:15: 라인은 10 번 반복이 이루어집니다. 다만 13: 라인으로 인해서 1 값이 짝수일 때는 13: 라인의 앞에 다른 문장이 있다면 이것은 실행되지만, 13: 라인 아래 부분은 무시됩니다.

조건문 **case**

문법

```
case 정수형변수 of
    값 1 : 문장 또는 블록;
    값 2 : 문장 또는 블록;
    값 3, 값 4 : 문장 또는 블록;
    값 5..값 6 : 문장 또는 블록;
    else 문장 또는 블록;
end;
```

사용의 예

```
i:= Random(10) + 1;
case i of
    1 : WriteLn('First');
    2 : WriteLn('Second');
    3 : WriteLn('Third');
    5 : WriteLn('Fifth');
    4, 6, 7, 8, 9 : WriteLn(i, 'th');
    else WriteLn('Ten');
end;
```

[리스트 **Begin19**]은 1 부터 10 까지의 숫자를 영어의 서수형으로 표현하는 예제 입니다.

[리스트 Begin19]

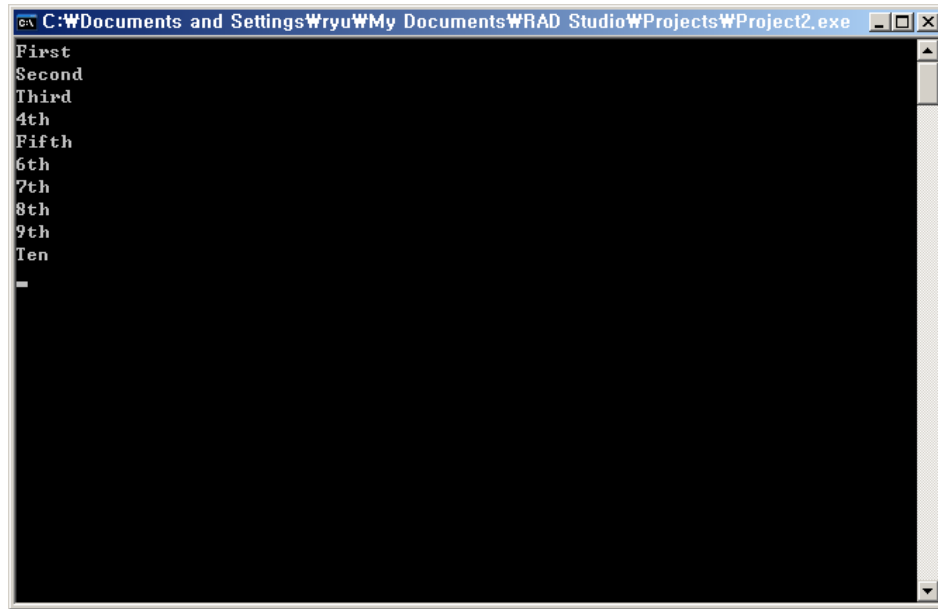
```
1 : program Begin19;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   i : integer;
10 :
11 : begin
12 :   for i := 1 to 10 do
13 :     case i of
14 :       1 : WriteLn('First');
15 :       2 : WriteLn('Second');
16 :       3 : WriteLn('Third');
17 :       5 : WriteLn('Fifth');
18 :       4, 6, 7, 8, 9 : WriteLn(i, 'th');
19 :       else WriteLn('Ten');
20 :     end;
21 :
22 :   ReadLn;
23 : end.
```

18: 라인에서는 여러 값을 한꺼번에 지정하는 방법을 보여주고 있습니다. 이것은 서브레인지(Sub Range)를 사용해서 다음과 같이 표현할 수도 있습니다.

```
18 :           4, 6..9 : WriteLn(i, 'th');
```

두 숫자 사이에 점을 두개 붙여서 사용하는 것을 서브레인지라고 부르며, 두 숫자를 포함하고 그 사이에 있는 모든 숫자를 의미하게 됩니다.

19: 라인에서 **else** 는 위의 조건이 하나도 맞지 않으면 디폴트로 실행할 문장이나 블록을 지정하게 됩니다.



[그림 23] 리스트 Begin19 의 실행 결과

비트연산

not	not	반전
and	and	논리곱
or	or	논리합
xor	xor	배타적 논리합
shl	Shift Left	비트 왼쪽 이동
shr	Shift Right	비트 오른쪽 이동

비트연산은 데이터를 비트 단위로 조작하고자할 때 사용합니다. 파스칼은 C 언어와 달리 논리연산자와 비트연산자를 동일하게 사용한다는 것에 유의하시기 바랍니다.

비트 연산에 대한 예제는 [리스트 Begin20]와 같습니다. 실행 결과는 [그림 4]를 참고하시면 됩니다. 결과를 확인하기 쉽도록 사용된 데이터 값을 모두 이진수로 표현하여 각각의 비트가 어떻게 변화하는지를 보여주고 있습니다.

[리스트 Begin20]

```

1 : program Begin20;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : function ByteToBinary(Value:integer):string;
9 : var
10 :   Loop : integer;
11 : begin
12 :   Result:= '';
13 :   for Loop:= 7 downto 0 do
14 :     if (Value and (1 shl Loop)) <> 0 then Result:= Result +
'1'
15 :       else Result:= Result + '0';
16 :   end;
17 :
18 : begin
19 :   WriteLn('240 = ', ByteToBinary(240));
20 :   WriteLn('255 = ', ByteToBinary(255));
21 :   WriteLn;
22 :
23 :   WriteLn('not 240 = ', ByteToBinary(not 240));
24 :   WriteLn;
25 :
26 :   WriteLn('240 and 255 = ', ByteToBinary(240 and 255));
27 :   WriteLn;
28 :

```



```

29 :   WriteLn('240 or 255 = ', ByteToBinary(240 or 255));
30 :   WriteLn;
31 :
32 :   WriteLn('1 = ', ByteToBinary(1));
33 :   WriteLn('1 shl 4 = ', ByteToBinary(1 shl 4));
34 :   WriteLn;
35 :
36 :
37 :   WriteLn('16 = ', ByteToBinary(16));
38 :   WriteLn('16 shr 2 = ', ByteToBinary(16 shr 2));
39 :   WriteLn;
40 :
41 :   ReadLn;
42 : end.

```

```

C:\Documents and Settings\Wryu\My Documents\WRAD Studio\Projects\Project2.exe
240 = 11110000
255 = 11111111

not 240 = 00001111

240 and 255 = 11110000

240 or 255 = 11111111

1 = 00000001
1 shl 4 = 00010000

16 = 00010000
16 shr 2 = 00001000

```

[그림 24] 리스트 Begin20 의 실행 결과

만약 여러분들이 복잡한 프로그램을 작성하다 보면, 프로세스의 요소들이 많아지고 프로그램은 점점 더 복잡해지게 됩니다. 이때, 프로그램의 부분 부분들을 작은 조각으로 나누어서 작성하게 되면, 좀더 쉽고 효율적인 프로그래밍을 할 수 있습니다. 함수란 이러한 경우에 프로그램을 작은 조각으로 나눌 수 있는 방법 중에 하나인 것입니다.

티브이나 라디오 등의 전자제품을 내부를 열어 보면, 수많은 부품들이 들어있는 것을 보실 수 있습니다. 대부분 이러한 부품들은 자신이 각자 맡은 임무가 있습니다.

이렇듯 프로그램을 작은 조각으로 나누고, 각자에게 임무를 부여하면서 분업화하고자 할 때 사용하는 것이 함수라고 생각하시면 되겠습니다.

procedure

파스칼 언어에서 함수는 `procedure` 와 `function` 으로 나뉩니다. `procedure` 는 실행 후 결과값을 반환하지 않는 함수이며, `function` 은 실행 후 결과값을 반환하는 함수입니다.

우선 `procedure` 의 문법부터 살펴보도록 하겠습니다.

문법

```
procedure 함수명 (인자 또는 인자블럭);  
  
const 선언문  
  
type 선언문  
  
var 선언문  
  
begin  
    문장 또는 블록;  
  
end;  
  
인자블럭 = 인자그룹 1; 인자그룹 2; ...  
  
인자그룹 =  
    변수명 : 변수타입;  
  
또는, 변수명 1, 변수명 2 ... : 변수타입;
```

사용의 예

```
procedure DoSomething1;  
  
begin  
  
end;  
  
  
procedure DoSomething2(a:integer; b,c:char; text:string);  
  
begin
```

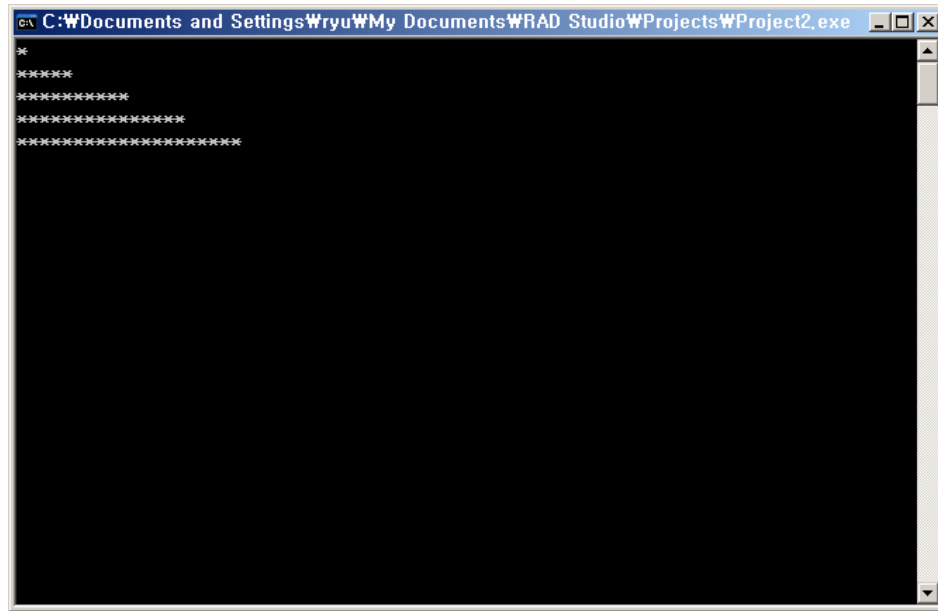
```
end;
```

[리스트 Begin21]

```
1 : program Begin21;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : procedure StarStick(Length:integer);
9 : var
10 :   i : integer;
11 : begin
12 :   for i := 1 to Length do Write('*');
13 : end;
14 :
15 : begin
16 :   StarStick(1);
17 :   WriteLn;
18 :
19 :   StarStick(5);
20 :   WriteLn;
21 :
22 :   StarStick(10);
23 :   WriteLn;
24 :
25 :   StarStick(15);
26 :   WriteLn;
27 :
28 :   StarStick(20);
29 :   WriteLn;
30 :
31 :   ReadLn;
32 : end.
```

8-13: 라인에서는 **StarStick** 이라는 함수를 선언하고 구현하고 있습니다. **StarStick** 은 8: 라인에 기술된 식별자와 인자를 이용하여 사용하는 함수이며, 함수를 호출하면 9-13: 라인처럼 동작한다는 의미입니다.

함수의 호출(사용)은 함수의 식별자를 표기하고, 필요한 인자를 넘겨주면 됩니다. 16:, 19:, 25:, 28: 라인은 함수 **StarSrick** 을 호출하면서 각각 다른 인자를 넘겨주는 장면입니다.



[그림 25] 리스트 Begin21 의 실행 결과

function

function 의 경우에는 리턴 값을 가진다는 것을 제외하고는 procedure 와 같습니다.

문법

```
function 함수명 (인자 또는 인자블럭) : 리턴타입;  
const 선언문  
type 선언문  
var 선언문  
begin  
    문장 또는 블록;  
    Result:= 리턴할 데이터 값;  
end;
```

사용의 예

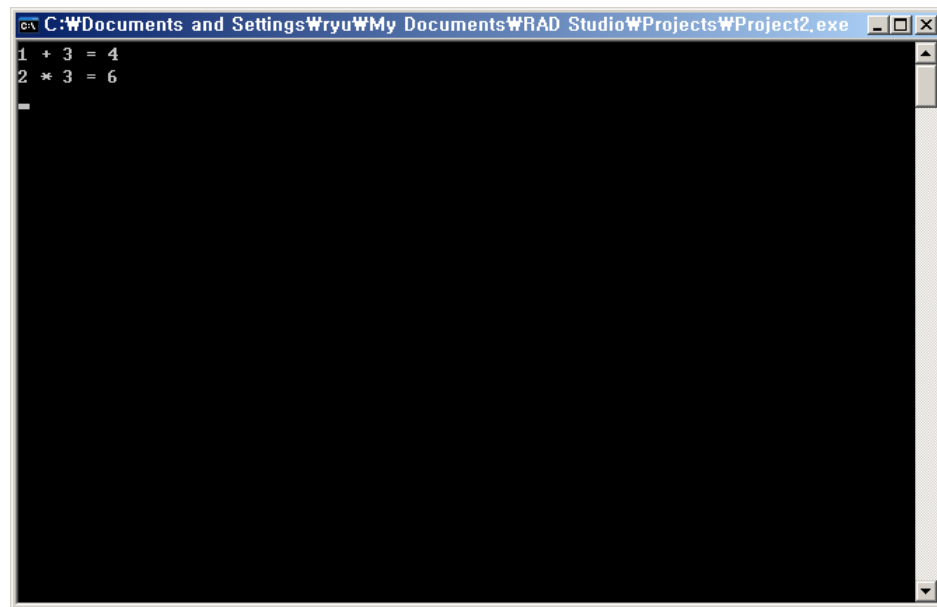
```
function DoubleValue(a:integer):integer;  
begin  
    Result:= a * 2;  
end;
```

[리스트 Begin22]는 두 정수형 인자를 받아서 합과 곱을 출력하는 각각의 함수 두개에 대한 예제를 보여주고 있습니다.

[리스트 Begin22]

```
1 : program Begin22;  
2 :  
3 : {$APPTYPE CONSOLE}  
4 :  
5 : uses  
6 :     SysUtils;  
7 :  
8 : function Sum(a,b:integer):integer;  
9 : begin  
10 :     Result:= a + b;  
11 : end;  
12 :  
13 : function Mul(a,b:integer):integer;
```

```
14 : begin
15 :   Result:= a * b;
16 : end;
17 :
18 : begin
19 :   WriteLn('1 + 3 = ', Sum(1, 3));
20 :   WriteLn('2 * 3 = ', Mul(2, 3));
21 :
22 :   ReadLn;
23 : end.
```



[그림 26] 리스트 Begin22 의 실행 결과

● Exit 제어문

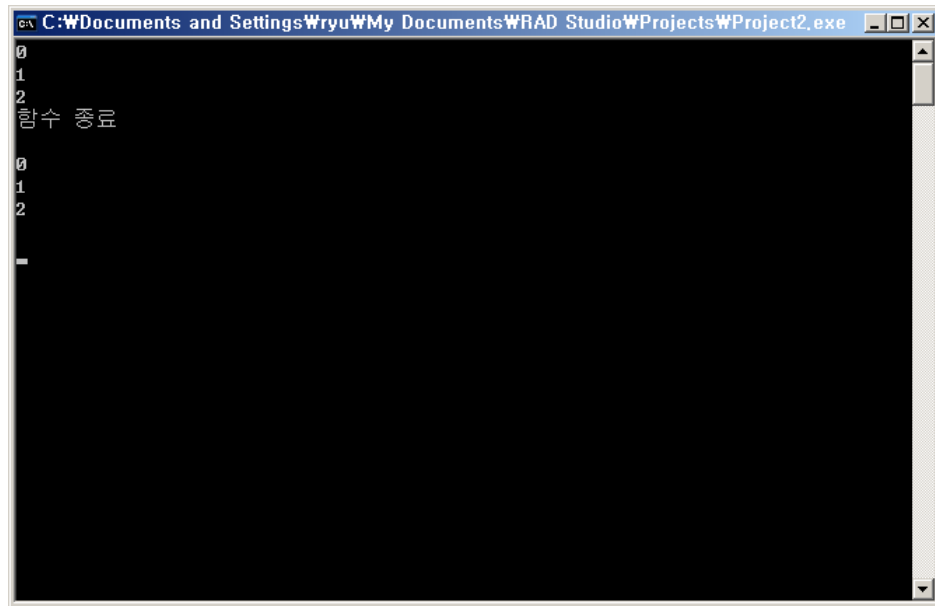
함수의 실행을 종료합니다.

[리스트 Begin23]

```
1 : program Begin0;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : procedure ForLoop1;
9 : var
10 :   i : integer;
11 : begin
12 :   for i := 0 to 9 do begin
13 :     if i = 3 then Break;
14 :     WriteLn(i);
15 :   end;
16 :
17 :   WriteLn('함수 종료');
18 : end;
19 :
20 : procedure ForLoop2;
21 : var
22 :   i : integer;
23 : begin
24 :   for i := 0 to 9 do begin
25 :     if i = 3 then Exit;
26 :     WriteLn(i);
27 :   end;
28 :
29 :   WriteLn('함수 종료');
30 : end;
31 :
32 : begin
33 :   ForLoop1;
34 :   WriteLn;
35 :
36 :   ForLoop2;
37 :   WriteLn;
38 :
39 :   ReadLn;
40 : end.
```


13: 라인에서 **Break** 는 해당 반복문만을 벗어나게 됩니다. 따라서, [그림 27]에서 보듯이 17: 라인이 실행되어, "함수 종료" 라는 문자열이 찍히게 됩니다.

하지만, 25: 라인의 **Exit** 는 해당 함수 자체를 종료하고 벗어나게 됩니다. 따라서, 29: 라인은 실행되지 않습니다.



[그림 27] 리스트 Begin23 의 실행 결과

함수의 인자처리 방식

함수의 인자는 아래의 두 가지 처리 방식이 있습니다.

- Call by value
 - procedure 함수명(변수명:변수타입);
 - function 함수명(변수명:변수타입) : 리턴타입;
- Call by reference
 - procedure 함수명(var 변수명:변수타입);
 - function 함수명(var 변수명:변수타입) : 리턴타입;

Call by value 는 함수를 실행하면서 인자를 넘길 때, 변수 안에 있는 값을 넘겨주는 방식이며, Call by reference 는 인자의 주소를 넘겨주는 방식입니다.

Call by reference 는 함수가 실행되면서 해당 인자의 값을 내부에서 변경하면, 함수를 실행할 때 인자로 사용했던 변수의 값이 변하게 됩니다.

이 두 가지 방식의 차이점은 아래의 소스를 통해서 설명하도록 하겠습니다.

[리스트 Begin24]

```
1 : program Begin24;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : procedure Test_A(a:integer);
9 : begin
10 :   a:= 1;
11 : end;
12 :
13 : procedure Test_B(var a:integer);
14 : begin
15 :   a:= 1;
16 : end;
17 :
18 : var
19 :   b : integer;
20 :
```

```

21 : begin
22 :   b:= 0;
23 :   WriteLn('b 의 값 : ', b);
24 :
25 :   Test_A(b);
26 :   WriteLn('b 의 값 : ', b);
27 :
28 :   Test_B(b);
29 :   WriteLn('b 의 값 : ', b);
30 :
31 :   ReadLn;
32 : end.

```

위에 소스에서는 **Test_A()** 함수와 **Test_B()** 함수 두 개를 가지고 인자전달 방식의 차이점을 테스트해보고 있습니다.

우선 23: 라인에서는 “0”이 출력됩니다.

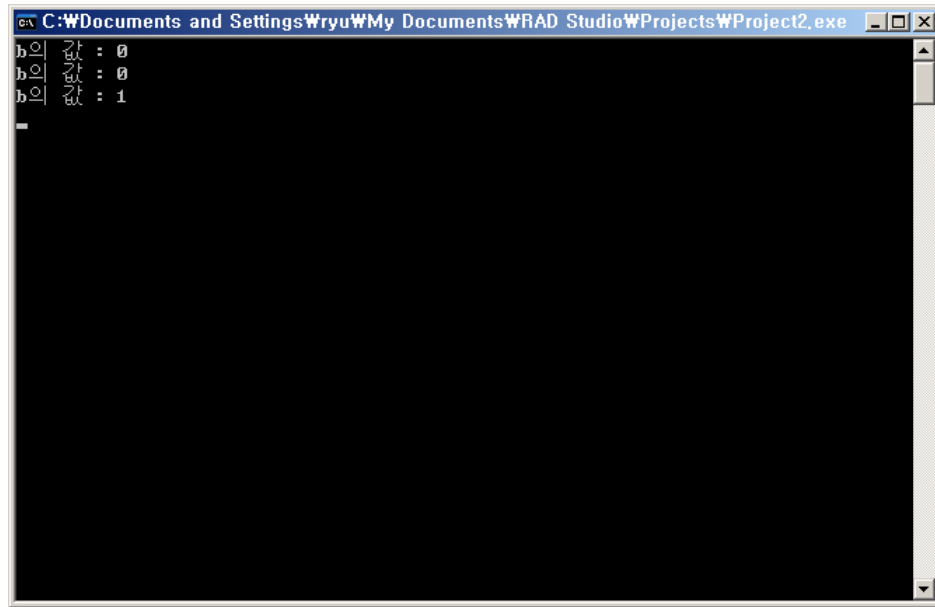
이후 26: 라인에서도 “0”이 출력됩니다.

하지만 29: 라인에서는 “1”이 출력됩니다.

Test_A(b) 에서의 인자 **Call by value** 의 인자 전달 방식은 변수 **b** 의 데이터 값을 함수에 전달하게 됩니다. 또한, 함수 내부에서 사용하는 인자 **a** 의 경우에는 **b** 와 전혀 다른 변수가 됩니다. 따라서, **a** 의 값을 변경하는 것은 변수 **b** 에게 아무런 영향을 주지 않습니다.

Test_B(b) 함수의 경우에는 **Call by reference** 의 인자 전달 방식을 사용합니다.

13: 라인에서처럼 인자 선언부분에서 **var** 라는 예약어를 앞에 붙이면 **Call by reference** 방식으로 인자가 생성됩니다. 이제 **a** 라는 인자는 **b** 와 동일한 주소 공간을 갖는 변수가 됩니다. 즉, 동일한 변수라고 보면 됩니다. 따라서, **Test_B()** 함수 내부에서 **a** 값을 변경되면 해당 값이 그대로 **b** 값에도 적용되게 됩니다.



[그림 28] 리스트 Begin24 의 실행 결과

중복의 제거

함수의 중요한 사용법 중에서 중복 제거의 기능이 있습니다. 중복은 프로그램을 복잡하게 만들고 비효율적으로 만들기 때문에, 반드시 제거해야 할 대상입니다. 중복은 단순히 같은 문장이나 블록이 반복되는 것이 있을 수도 있지만, 특정한 패턴이 반복되는 경우도 있습니다.

[리스트 Begin25]

```
1 : program Begin25;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : begin
9 :   WriteLn(FormatDateTime('hh:nn:ss', Now));
10 :   ReadLn;
11 :
12 :   WriteLn(FormatDateTime('hh:nn:ss', Now));
13 :   ReadLn;
14 :
15 :   WriteLn(FormatDateTime('hh:nn:ss', Now));
16 :   ReadLn;
17 :
18 :   ReadLn;
19 : end.
```

[리스트 Begin25] 에서 반복되는 부분을 함수로 만들어서 [리스트 26]으로 표현하면 다음과 같습니다.

[리스트 Begin26]

```
1 : program Begin26;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : procedure DisplayTime;
9 : begin
10 :   WriteLn(FormatDateTime('hh:nn:ss', Now));
11 :   ReadLn;
12 : end;
13 :
```

```
14 : begin
15 :   DisplayTime;
16 :   DisplayTime;
17 :   DisplayTime;
18 :
19 :   ReadLn;
20 : end.
```

여기서 우리는, 소스가 짧아진 것보다 효율적으로 변모한 것에 집중해야 합니다. 만약 날짜를 표시하는 방법이 달라진다고 하면, 두 소스를 어떻게 수정해야 할까요?

[리스트 **Begin25**]의 경우에는 9:, 12:, 15: 라인 모두 수정해야만 날짜를 표시하는 포맷을 변경하실 수 있습니다. 하지만, [리스트 **Begin26**]에서는 10: 라인 한 곳만 수정하시면 됩니다.

[리스트 **Begin25**]의 경우에는 프로그램에 변경사항이 있을 때, 개발자의 실수로 빠트리고 변경사항을 적용하지 않을 수 있는 가능성이 생깁니다. 이것은 가끔 위험한 상황으로 번지는 경우가 있습니다.

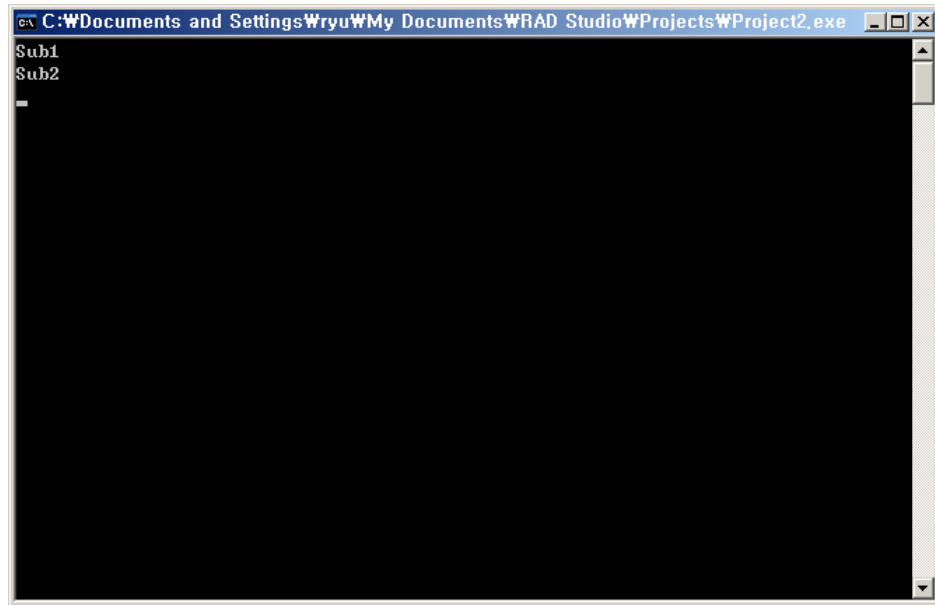
서브 함수의 선언

오브젝트 파스칼에서는 함수 내부에 다른 함수를 재귀적으로 선언할 수 있습니다. [리스트 Begin27]에서 **DoSomething** 이라는 함수 내부에 **Sub1** 과 **Sub2** 함수 두 개를 선언하고 호출하는 과정을 표현하고 있습니다.

서브 함수의 경우에는 서브 함수를 선언한 함수 이외의 곳에서는 사용할 수 없습니다.

[리스트 Begin27]

```
1 : program Begin27;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : procedure DoSomething;
9 :   procedure Sub1;
10 :    begin
11 :      WriteLn('Sub1');
12 :    end;
13 :   procedure Sub2;
14 :    begin
15 :      WriteLn('Sub2');
16 :    end;
17 :   begin
18 :     Sub1;
19 :     Sub2;
20 :   end;
21 :
22 :   begin
23 :     DoSomething;
24 :
25 :     ReadLn;
26 :   end.
```



[그림 29] 리스트 Begin27 의 실행 결과

1 차원 배열

문법

배열이름 : array [시작번호..종료번호] of 변수타입;

사용의 예

```
var  
  
    Scores : array [1..100] of integer;
```

배열은 같은 종류의 변수 여러 개를 하나로 묶는 방법을 제공합니다. 프로그램을 작성하다보면 동일한 용도로 사용하는 다수의 변수가 필요할 때가 있습니다.

만약, 그러한 변수가 1000 개가 있다고 가정하고 아래와 같이 코딩을 해야 한다면 어떨까요?

```
var  
  
    Score1, Score2, Score3, Score4 ..... : integer;
```

아마도, 프로그램을 작성하다가 정신이상이 생길지도 모르겠습니다. 이런 경우에는 배열의 문법을 이용해서 아래처럼 간단하게 다수의 변수를 한꺼번에 선언할 수 있습니다.

```
var  
  
    Scores : array [1..1000] of integer;
```

파스칼의 배열은 다른 언어와 달리 0 을 포함한 그 어떠한 정수값도 이용할 수 있습니다. 따라서, 아래의 선언은 모두 유효합니다.

```

var

    Array1 : array [1..10] of integer;

    Array2 : array [0..9] of integer;

    Array3 : array [-4..5] of integer;

```

위에서 배열 변수 **Array1**, **Array2**, **Array3** 모두는 10 개의 변수공간을 갖는 같은 크기의 배열로 선언됩니다. 배열에 데이터 값을 대입하기 위해서는 아래와 같은 방식을 취합니다.

```

begin

    Array1[1] := 0;

    Array1[2] := 0;

    ...

    Array2[0] := Array1[1];

    Array3[-4] := Array1[1];

    ...

end;

```

위의 소스는 배열의 요소들을 개별적으로 접근할 때의 방법을 보여줍니다. 하지만, 만약 전체 요소를 전부 0 또는 특정한 데이터 값을 대입시키고 싶을 때는 어떻게 해야 할까요? 물론 1000 개의 대입문을 작성하셔도 상관없습니다.

하지만, 좀더 효율적인 방법은 반복문을 이용하는 것 입니다.

[리스트 Begin28]

```

1 : program Begin28;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :     SysUtils;
7 :

```

```

8 : var
9 :   i : integer;
10 :   Array1 : array [1..10] of integer;
11 :   Array2 : array [0..9] of integer;
12 :   Array3 : array [-4..5] of integer;
13 :
14 : begin
15 :   for i:= 1 to 10 do Array1[i]:= 0;
16 :   for i:= 0 to 9 do Array2[i]:= 0;
17 :   for i:= -4 to 5 do Array3[i]:= 0;
18 :
19 :   for i := Low(Array1) to High(Array1) do Array1[i]:= 0;
20 :   for i := Low(Array2) to High(Array2) do Array2[i]:= 0;
21 :   for i := Low(Array3) to High(Array3) do Array3[i]:= 0;
22 :
23 :   FillChar(Array1, SizeOf(Array1), 0);
24 :   FillChar(Array2, SizeOf(Array2), 0);
25 :   FillChar(Array3, SizeOf(Array3), 0);
26 : end.

```

15-17: 라인이 일반적으로 배열의 데이터들을 초기화하기 위해서 사용하는 표현 방식입니다.

19-21: 라인의 경우에는 좀더 안전한 방식을 취하고 있습니다. 배열이 어디서 시작하고 어디서 종료하는 지 일일이 프로그래머가 지정하지 않고, **Low()** 함수와 **High()** 함수를 이용해서 배열의 시작과 끝 번호를 알아내는 방법입니다. 나중에 배열의 크기가 변경되더라도 코드를 수정하지 않아도 되며, 배열의 크기만 수정한 후 코드를 수정하지 않아서 생기는 에러를 방지할 수 있습니다.

23-25: 라인은 가장 빠른 방법 입니다. 다만, 0 부터 255 사이의 숫자를 배열의 요소에 저장할 때만 가능한 방법입니다.

FillChar() 함수를 이용하여 배열 데이터를 일괄 초기화하는 방법은 성능 면에서 아주 뛰어난 방법입니다. 다만, 다음과 같은 주의해야 할 점이 있습니다.

FillChar() 함수는 첫번째 인자로 넘어온 데이터의 타입을 구분하지 않고, 무조건 1 바이트 단위로 두번째 인자의 값을 채워넣습니다. 위의 예제에서 배열 데이터의 각 요소는 integer 타입인데, 이 배열 요소들에 1 바이트 단위로 값을 채워넣으면 예상치 못한 결과가 나올 수도 있습니다.

예제 코드에서는 채워넣을 값을 0 으로 지정했기 때문에 문제가 없습니다만, 0 이 아닌 값, 예를 들어 1 값을 초기화 값으로 넘기게 되면, 정수 4 바이트의 각 1 바이트마다 1 이라는 값을 채우게 되므로 배열요소 **Array1[1]**의 값은 1 이 아닌 **\$01010101** 이 됩니다. 따라서, **FillChar()**를 이용한 배열 데이터 초기화는 기본적으로 0 값으로 초기화할 때만 사용한다고 알아두시기 바랍니다.

다차원 배열

문법

배열이름 : array [범위 1, 범위 2, ...] of 변수타입;

범위 = 시작번호..종료번호

사용의 예

```
Pixels : array [1..100, 1..100] of integer;
```

다차원 배열은 배열의 요소를 접근할 수 있는 주소를 여러 개로 지정하고자 할 때 사용합니다. 마치, 우리가 살고 있는 집들이 통과 반 두개로 나누어서 지정되는 것과 같습니다. 때로는 이렇게 요소를 접근할 주소를 여러 개로 나누는 것이 효율적일 때가 있습니다.

화면상에 어떤 점의 위치를 표현할 때 우리는 **x** 축 좌표와 **y** 축 좌표 두개의 값을 이용해서 표현합니다. 이와 같은 것들을 표현할 때는 **2** 차원 배열이 필요합니다. 공간 속의 어떤 점의 위치를 표현할 때는 **x, y, z** 세 개의 좌표 값을 이용합니다. 이런 경우에는 **3** 차원 배열을 사용하면 됩니다.

다차원 배열은 필요한 차수만큼 범위를 만들어서 선언하시면 됩니다.

[리스트 Begin29]

```
1 : program Begin29;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   x, y, z : integer;
10 :   Pixels : array [1..10, 1..10] of integer;
11 :   Cube : array [1..10, 1..10, 1..10] of integer;
12 :
13 : begin
14 :   for x:= 1 to 10 do
15 :     for y:= 1 to 10 do Pixels[x, y]:= 0;
16 :
17 :   for x:= 1 to 10 do
18 :     for y:= 1 to 10 do
```

```
19 :      for z:= 1 to 10 do Cube[x, y, z]:= 0;  
20 : end.
```

배열의 차수만큼 반복문도 중첩해서 사용해야 합니다. 14: 라인은 15: 라인을 10 번 반복한다는 뜻이 됩니다.

15: 라인은 `Pixels[x, y]:= 0` 부분을 10 번 반복하게 됩니다.

동적배열

문법

배열이름 (변수이름) : array of 변수타입;

사용의 예

```
var  
  
    Scores : array of integer;
```

동적 배열은 배열의 크기를 정하지 않고 필요시에 배열의 크기를 늘리거나 줄일 수 있는 방법을 제시합니다. 동적 배열은 시작번호를 지정할 수 없이 항상 0 으로만 시작하는 점에서 일반 배열과 차이가 있습니다.

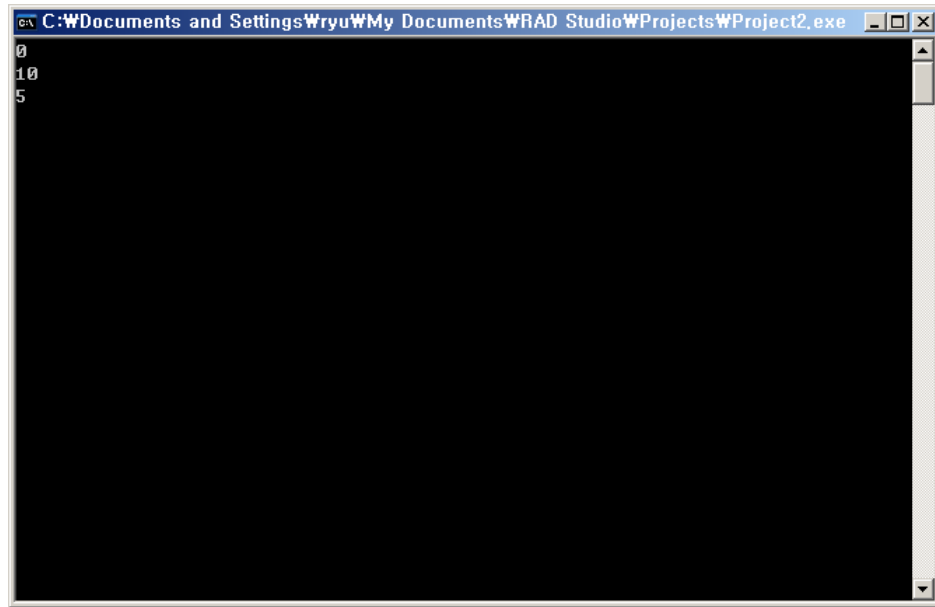
[리스트 Begin30]

```
1 : program Begin30;  
2 :  
3 : {$APPTYPE CONSOLE}  
4 :  
5 : uses  
6 :   SysUtils;  
7 :  
8 : var  
9 :   Chars : array of char;  
10 :  
11 : begin  
12 :   WriteLn (Length (Chars));  
13 :  
14 :   SetLength (Chars, 10);  
15 :   WriteLn (Length (Chars));  
16 :  
17 :  
18 :   SetLength (Chars, 5);  
19 :   WriteLn (Length (Chars));  
20 :  
21 :   ReadLn;  
22 : end.
```

12: 라인에서는 배열의 길이가 0 으로 출력됩니다. 동적배열은 선언하고 아무런 변경을 하지 않으면 길이가 0 이 됩니다.

14: 라인의 `SetLength()` 함수는 배열이나 문자열의 길이를 변경할 수 있습니다. [그림 1]에서 보는 것과 같이 길이가 10 으로 변경된 것을 알 수 있습니다.

18: 라인에서는 배열의 크기를 줄이는 것을 테스트 해 보았습니다. [그림 30]에서 보는 것처럼 동적배열의 크기는 늘일 수도 있지만, 줄이는 것도 가능하다는 것을 알 수 있습니다.



[그림 30] 리스트 `Begin30` 의 실행 결과

문자열

문자열은 배열의 한 종류입니다. 고전적인 파스칼의 문자열은 문자(char)형 변수의 배열로 사용하였습니다. 현재 델파이에서 기본적으로 사용하고 있는 문자열(string) 변수는 동적배열과 유사하다고 보면 됩니다.

아래의 예제를 보면 문자열도 배열처럼 각각의 요소를 **St[i]**와 같은 방식으로 접근이 가능하다는 것을 알 수 있습니다. 다만, 배열과 달리 문자열에서는 시작번호는 항상 1로 지정됩니다. 문자열과 동적배열은 길이가 변할 수 있기 때문에, 해당 요소를 접근하기 위한 첨자 i의 값이 문자열이나 동적배열의 범위 안에 있는지 확인하여야 합니다.

```
var
  i : integer;
  St : string;
begin
  St:= 'Test';
  for i := 1 to Length(St) do WriteLn(St[i]);
end;
```

문자열이 배열과 다른 점은 아래의 예제처럼 덧셈 연산이 가능하다는 점입니다.

```
var
  St : string;
begin
  St:= 'This is a' + 'Test';
  St:= St + '. ';
  St:= St + St;
  WriteLn;
end.
```

문자열을 다루기 위해 미리 준비된 여러가지 유용한 함수들이 있습니다. 그 중에서 자주 사용하는 것 몇 가지를 [리스트 31]을 통하여 설명하도록 하겠습니다.

[리스트 Begin31]

```

1 : program Begin31;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   St : string;
10 :
11 : begin
12 :   St:= 'This is a ' + 'Test';
13 :   St:= St + '. ';
14 :   St:= St + St;
15 :   WriteLn('테스트할 문자열 : ', St);
16 :
17 :   WriteLn('문자열의 길이 : ', Length(St));
18 :
19 :   SetLength(St, Length('This is a Test.'));
20 :   WriteLn('문자열의 길이 줄이기 : ', St);
21 :
22 :   WriteLn('문자열에서 Test 의 위치는 : ', Pos('Test', St));
23 :
24 :   WriteLn('소문자 변환 : ', LowerCase(St));
25 :
26 :   WriteLn('대문자 변환 : ', UpperCase(St));
27 :
28 :   WriteLn('문자열 일부 가져오기 : ', Copy(St, 1, 4));
29 :
30 :   Delete(St, 1, 5);
31 :   WriteLn('문자열 일부 삭제하기 : ', St);
32 :
33 :   St:= StringReplace(St, 'a', 'the', []);
34 :   WriteLn('문자열 중 특정 문자열을 다른 문자열로 치환하기 : ', St);
35 :
36 :   ReadLn;
37 : end.

```

17: 라인과 19: 라인의 **Length()** 함수와 **SetLength()** 함수의 사용법은 동적배열 설명과 동일합니다.

22: 라인의 **Pos**(찾을 대상 문자열, 대상을 찾고자 하는 원본 문자열) 함수는 문자열 내부에서 지정된 문자열을 찾는 함수입니다. 찾은 위치는 정수형 값으로 리턴합니다.

24: 라인과 26: 라인의 **LowerCase()** 함수와 **UpperCase()** 함수는 각각 문자열을 소문자로 또는 대문자열로 변환한 데이터(문자열)를 리턴합니다. 원본 문자열의 데이터는 변화가 없습니다. 원본 문자열 자체를 소문자로 변환하고자 한다면 다음과 같이 코드를 작성하시면 됩니다.

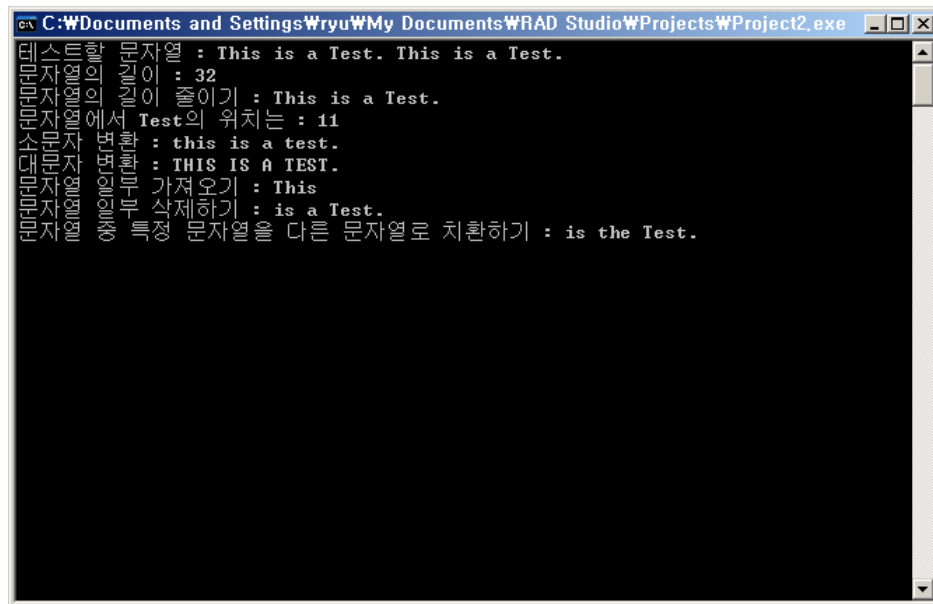
```
St:= LowerCase(St);
```

28: 라인의 **Copy**(문자열, 시작위치, 갯수) 함수는 문자열에서 시작위치에 해당하는 문자부터 갯수만큼의 문자를 복사한 데이터(문자열)를 리턴합니다.

30: 라인의 **Delete**(문자열, 시작위치, 갯수) 함수는 문자열의 시작위치에서부터 갯수만큼의 문자를 삭제하게 됩니다. 이때 문자열의 길이가 갯수만큼 줄어들게 됩니다.

33: 라인의 **StringReplace**(문자열, 원본, 목표, [옵션]) 함수는 문자열 내부에서 원본과 같은 문자열을 찾아서 해당 부분을 목표로 치환하고 결과 값(문자열)을 리턴합니다. 옵션의 경우에는 **rfReplaceAll**, **rfIgnoreCase** 두 가지를 사용할 수 있습니다. 두 옵션은 같이 사용할 수도 있습니다.

- **rfReplaceAll** : 발견되는 모든 원본 문자열을 목표로 변경한다.
- **rfIgnoreCase** : 원본을 찾을 때, 대소문자 구분을 하지 않는다.



```
C:\Documents and Settings\Wryu\My Documents\WRAD Studio\Projects\Project2.exe
테스트할 문자열 : This is a Test. This is a Test.
문자열의 길이 : 32
문자열의 길이 줄이기 : This is a Test.
문자열에서 Test의 위치는 : 11
소문자 변환 : this is a test.
대문자 변환 : THIS IS A TEST.
문자열 일부 가져오기 : This
문자열 일부 삭제하기 : is a Test.
문자열 중 특정 문자열을 다른 문자열로 치환하기 : is the Test.
```

[그림 31] 리스트 Begin31 의 실행 결과

구조체

문법

```
type
  구조체타입명 = record
    변수명 : 변수타입;
    변수명 1, 변수명 2 ... : 변수타입;
    ...
end;

var
  구조체변수명 : 구조체타입명;
```

사용의 예

```
type
  TPerson = record
    Name : string;
    Age : integer;
end;

var
  Person1, Person2 : TPerson;
```

배열이 같은 종류의 변수 여러 개를 하나로 묶는 방법을 제공하는 것에 비해, 구조체는 같은 목적을 가진 여러 변수를 하나로 묶는 방법을 제공 합니다.

TPerson 의 경우에는 사람의 정보를 저장하기 위한 같은 목적으로 사용되는 두 변수 이름과 나이를 하나로 묶은 것으로 보면 됩니다.

구조체의 경우에는 다른 종류의 변수를 묶을 수 있는 것이 배열과 다르며, 배열처럼 각각의 요소를 사용하기 위해서 첨자를 사용하지 않고, 요소들의 변수명 자체를 사용하게 됩니다.

또한 구조체는 **type** 선언문에서 구조체의 형태(**type**)가 어떻게 생겼는 지를 선언하여 주고, **var** 선언문에서, 새로 생겨난 변수타입(구조체타입)의 형태를 가진 데이터를 저장할 수 있는 변수를 선언해야 합니다.

[리스트 Begin32]

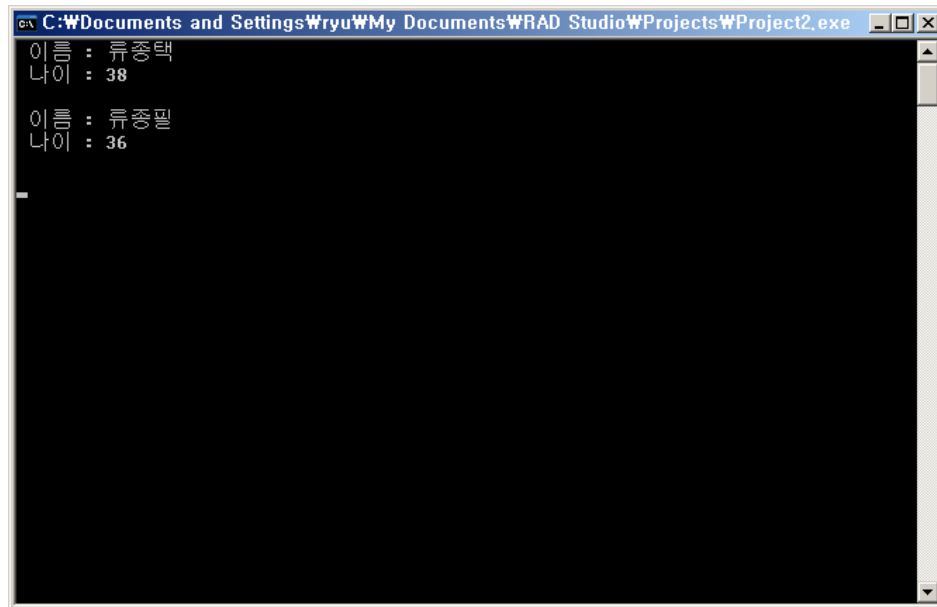
```
1 : program Begin32;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = record
10 :     Name : string;
11 :     Age : integer;
12 :   end;
13 :
14 : var
15 :   Person1, Person2 : TPerson;
16 :
17 : begin
18 :   Person1.Name:= '류종택';
19 :   Person1.Age:= 20 + 18;
20 :
21 :   Person2.Name:= '류종필';
22 :   Person2.Age:= Person1.Age - 2;
23 :
24 :   WriteLn(' 이름 : ', Person1.Name);
25 :   WriteLn(' 나이 : ', Person1.Age);
26 :   WriteLn;
27 :
28 :   WriteLn(' 이름 : ', Person2.Name);
29 :   WriteLn(' 나이 : ', Person2.Age);
30 :   WriteLn;
31 :
32 :   ReadLn;
33 : end.
```

9-12: 라인에서는 새로운 변수타입(구조체타입) **TPerson** 을 정의하고 있습니다. 변수타입은, 우리가 변수에 데이터를 집어넣거나 꺼내기 위해서, 해당 변수가 어떤 크기의 그리고 어떤 종류의 데이터를 저장할 수 있는 지를 결정하게 됩니다. 우리가 지금까지 사용해온 **integer** 등의 변수들도 이미 타입이 설정되어 있을 뿐, 컴파일러 내부에서는 해당 타입이 가지는 크기와 어떤 유형의 데이터를 사용할 수 있는 지가 정의되어 있습니다. 이렇게 이미 만들어둔 변수타입을 사용할 때는 **var** 선언문에서 바로 변수를 선언하면 되지만, 미리 준비된 타입과 전혀 다른 형태의 데이터를 사용하고 싶을 때는, **type** 선언문을 통해서 새로운 타입을 사용자 스스로가 만들어 내야 하는 것 입니다. 타입명 앞에 "**T**" 를 붙이는 것은 터보파스칼과 델파이의 전통입니다. 여러 가지 유래 설이 있지만, 저자의 경우에는 "**Type of**" 라는 의미로 생각하고 있습니다.

15: 라인에서는 새로 만든 **TPerson** 이라는 변수타입으로 만든 변수 **Person1** 과 **Person2** 가 선언되어 있습니다.

구조체의 각각의 요소에 접근하기 위해서는 **18:** 라인에서 보는 것처럼 "구조체 변수명.요소의 변수명" 형식으로 접근 합니다.

요소의 타입이 구조체타입일 수도 있습니다. 이런 경우에는 "a.b.c ..." 과 같이 점으로 구분지어서 계속
늘여 쓰시면 됩니다.



[그림 32] 리스트 Begin32 의 실행 결과

예외처리를 위한 **try except**

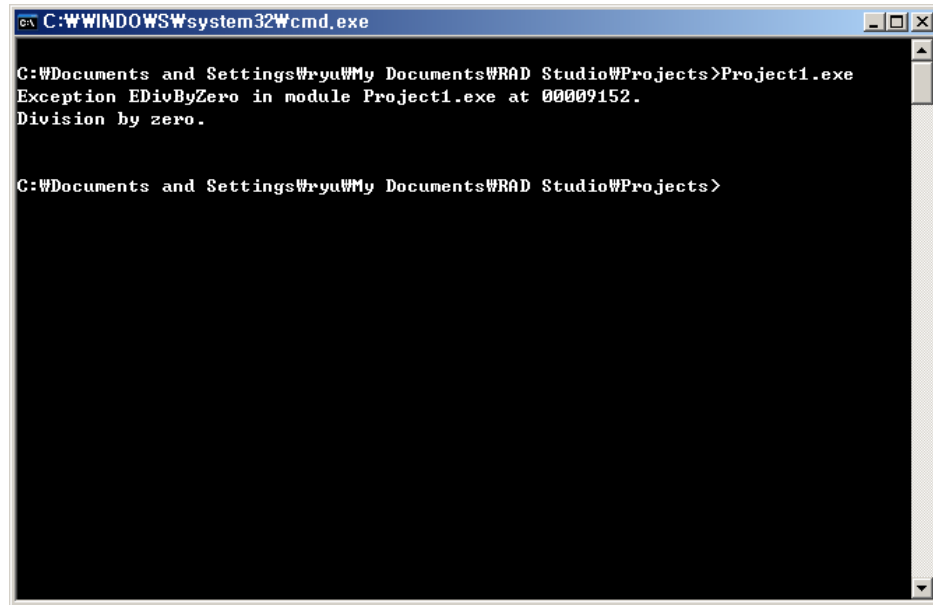
[리스트 Begin33]

```
1 : program Begin33;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   i : integer;
10 :
11 : begin
12 :   i:= 0;
13 :   i:= i div i;
14 :   WriteLn(i);
15 :
16 :   ReadLn;
17 : end.
```

프로그램이 실행되는 도중 에러가 발생하면 화면에는 에러가 표시되고 프로그램이 종료됩니다. 우선 [리스트 Begin33]의 코드를 실행하면 "Division by zero" 라는 에러가 발생하고 프로그램이 종료됩니다. 이유는 변수 *i* 의 값이 12: 라인에서 0 으로 입력되어 있기 때문입니다. 0 으로는 어떤 수도 나눌 수 없기 때문에 에러가 발생합니다. 따라서, 에러는 13: 라인에서 발생하며, 이후의 라인들은 모두 무시됩니다.

[리스트 Begin33]을 F9 등을 눌러서 실행한 경우에는 에러메시지를 보기도 전에 프로그램이 종료될 것입니다.

[그림 33]은 저자가 콘솔에서 실행파일을 직접 실행시킨 결과입니다.



[그림 33] 리스트 Begin33 의 실행 결과

만약에 여러분이 에러가 나더라도 프로그램의 실행은 그대로 진행되기를 바란다면 어떻게 해야 할까요? 해답은 **try except** 에 있습니다. **try except** 사이의 문장이나 블록이 실행되면서 에러가 발생하면, 에러를 표시하지 않고 **except end** 사이의 문장이나 블록을 실행하게 됩니다.

try except 의 일반적인 문법은 아래와 같습니다.

```
try
    문장 또는 블록;
except
    에러가 나면 실행할 문장 또는 블록;
end;
```

위에서 제시한 문법을 토대로 [리스트 Begin33]을 다시 작성하도록 하겠습니다.

[리스트 Begin34]

```
1 : program Begin34;
2 :
3 : {$APPTYPE CONSOLE}
```

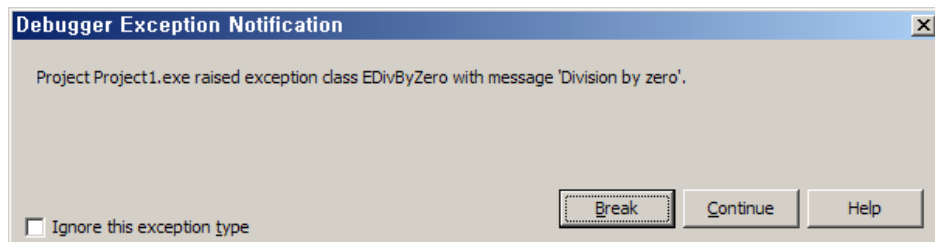
```

4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   i : integer;
10 :
11 : begin
12 :   i:= 0;
13 :   try
14 :     i:= i div i;
15 :     WriteLn(i);
16 :   except
17 :     WriteLn('에러가 발생했어요!');
18 :   end;
19 :
20 :   ReadLn;
21 : end.

```

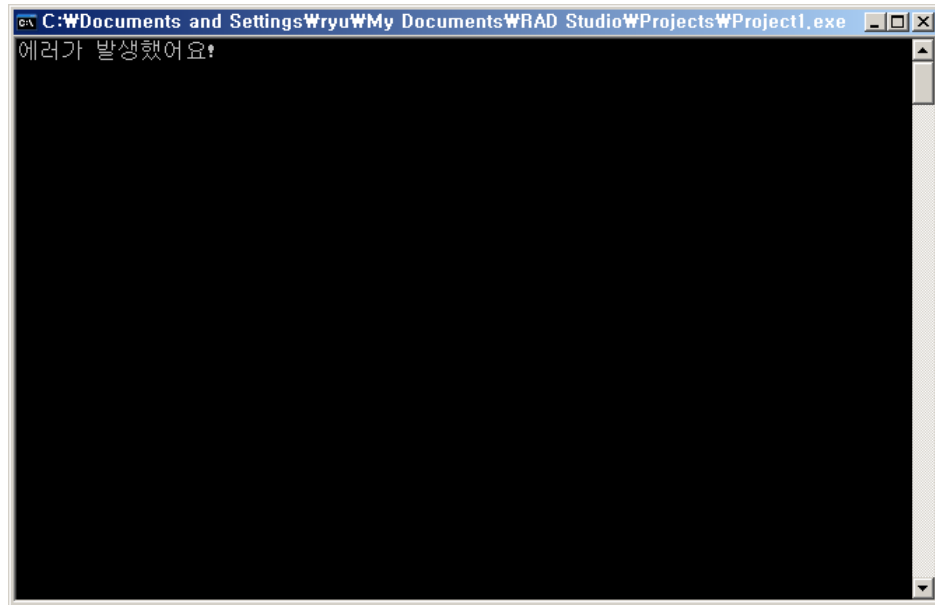
14: 라인에서 에러가 발생하게 되면 **except end** 사이를 실행하게 됩니다. 이때, **try except** 사이에 있는 다음 라인들은 모두 무시됩니다. 따라서, 17: 라인의 문자열이 화면에 출력될 것입니다.

하지만, 디버깅 모드 실행 중에는 [그림 34]와 같이 **Debugger Exception Notification** 창이 표시됩니다. 이것은 에러 표시창은 아닙니다. 단순히 개발자에게 이런 에러가 났었다는 것을 알려주는 역할일 뿐입니다. **Debugger Exception Notification** 창이 뜰 때, **Continue** 를 선택해주시면 프로그램이 계속 진행됩니다.



[그림 34] Debugger Exception Notification

F9 등을 눌러서 델파이를 통해서 프로그램을 실행할 경우에는 **try except** 로 막아둔 코드에서 나는 에러가 감지됩니다. F9 등을 눌러서 소스를 실행하는 것을 디버깅 모드라고 부릅니다. 디버깅 모드에서는 **try except** 구간 내에서 에러가 났다는 것을 개발자에게 알려주도록 되어 있습니다. 실행파일을 사용할 때는 에러가 표시되지 않습니다.



[그림 35] 리스트 Begin34 의 실행 결과

[그림 35]에서 보면 에러가 나긴 했지만 어떠한 에러인지 알 수 없습니다. 만약 어떠한 에러인지 확인하고 싶을 때 사용할 수 있는 간단한 코드는 아래와 같습니다.

```
try
  i:= i div i;
  WriteLn(i);
except
  on E : Exception do WriteLn('에러 : ', E.Message);
end;
```

try except 문법을 완전히 익히기 위해서는 좀더 많은 지식이 필요합니다. 하지만, 본 입문서에서는 지금까지 설명한 기본적인 문법에 대해서만 다루도록 하겠습니다. 보다 자세한 내용은 델파이에 포함된 문서를 참고하시기 바랍니다. 일단 입문자의 경우에는 **on E : Exception do** 를 앞에 붙이고 뒤에 문장 또는 블록을 사용할 수 있다고 까지만 기억해 두시기 바랍니다.

어떤한 악조건에서도 반드시 실행해야하는 코드를 위한 **try finally**

```
try
    문장 또는 블록;

finally
    try finally 구간을 실행하고 난 다음 실행할 문장 또는 블록;

end;
```

에러가 나더라도 반드시 실행해야 하는 코드가 있을 때, **try finally** 를 사용하면 됩니다. **try finally** 사이에서 에러가 나더라도, **finally end** 구간은 실행 됩니다. 또한, **try finally** 사이에서 에러가 나지 않더라도, **finally end** 구간은 실행 됩니다.

즉, **try finally** 가 실행되는 상황에서는, **finally end** 가 어떠한 경우라도 반드시 실행 됩니다.

```
try
    물건을 사용하면;

finally
    제 자리에 갖다 뒀다;

end;
```

try finally 는 주로 메모리 할당과 해제 그리고 멀티 쓰레드 사용 중에 자원 사용을 잠그거나 해제하는 부분에서 사용됩니다. 이에 대한 자세한 설명은 추후 포인터 및 오브젝트를 설명할 때 예제를 통해서 드리도록 하겠습니다.

Chapter

8

Unit

프로그램을 잘게 자르자

프로그램을 작성하다 보면 소스의 덩치가 커져서 소스를 관리하기 어려울 때가 있습니다. 이런 경우에는 프로그램을 잘게 잘라서 다른 파일에 보관하여 사용하는 방법을 사용하실 수 있습니다.

프로그램 소스를 다른 파일에 저장하고 난 후 다시 불러서 사용하는 방법은 큰 프로그램을 작성할 때 뿐만 아니라, 한 번 만들어 놓은 소스를 다른 곳에서 재사용하고자 할 때도 유용합니다. 유닛(Unit)을 사용하면, 언젠가 다시 사용할 가능성이 많은 소스를 따로 파일에 보관했다가, 필요 시 마다 불러서 사용할 수 있습니다.

일단 이미 작성된 파일을 불러오는 방법은 이 책의 처음부터 나오는 예제에서 자주 보아온 데로 `uses` 문법을 사용하시면 됩니다. `uses` 로 불러들인 유닛 파일들은 마치 여러분들이 불러들인 파일을 현재의 파일에 그대로 코딩한 것과 동일한 효과를 가지게 됩니다.

(유닛 파일의 구조가 프로젝트 파일과 다르기 때문에 약간의 변경이 필요합니다)

```
uses
```

```
    불러올 파일명 1, 불러올 파일명 2 ...;
```

`uses` 는 복수의 파일을 지정할 수 있으며, 불러올 파일의 경로나 확장자는 필요없이 파일 이름만 입력하시면 됩니다. `uses` 불러올 파일의 경로명을 지정하지 않은 경우에는, 불러올 파일들은 다음과 같이 특별히 지정된 폴더에 있어야지만 불러올 수 있습니다.

- 현재 작성 중인 Project 파일과 동일한 폴더
- 델파이가 설치된 폴더 밑에 Lib 폴더
- Project 옵션에서 지정한 Search Path 들
- 델파이 옵션 중에 Library Path 에 지정된 폴더들


또한, 현재 작성 중인 Project 파일에 묶여 있는 파일들은 위의 폴더 이외에 곳에 있더라도 사용할 수 있습니다. 델파이 프로젝트 파일을 사용 중에 델파이 메뉴에서 "New Items" 를 선택하여 추가한 파일들은 자동으로 프로젝트 파일에 묶여서 보관됩니다.

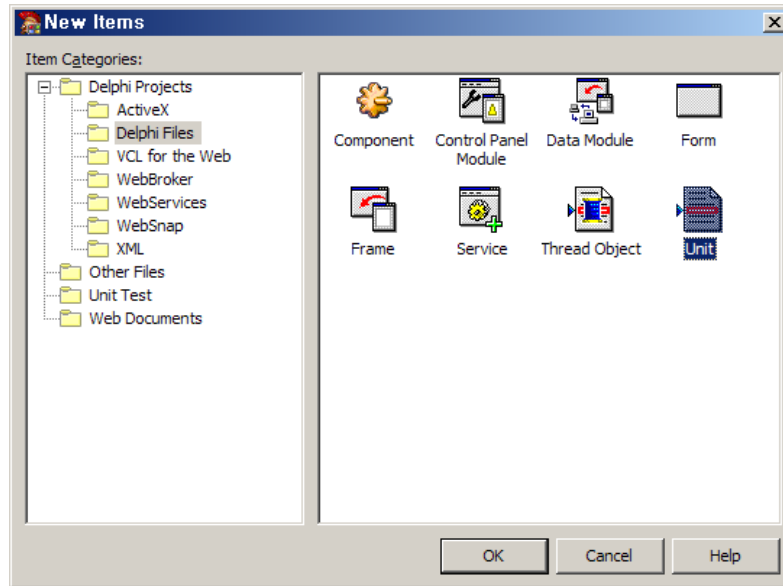
이해를 돕기 위해 [리스트 Begin35] 소스의 일부를 조각내어 [리스트 Begin36]으로 변경시켜보도록 하겠습니다.

[리스트 Begin35]

```
1 : program Begin35;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : function Sum(a,b:integer):integer;
9 : begin
10 :   Result:= a + b;
11 : end;
12 :
13 : function Mul(a,b:integer):integer;
14 : begin
15 :   Result:= a + b;
16 : end;
17 :
18 : function Divide(a,b:integer):integer;
19 : begin
20 :   Result:= a + b;
21 : end;
22 :
23 : begin
24 :   WriteLn('4 + 2 = ', Sum(4, 2));
25 :   WriteLn('4 * 2 = ', Mul(4, 2));
26 :   WriteLn('4 / 2 = ', Divide(4, 2));
27 :
28 :   ReadLn;
29 : end.
```

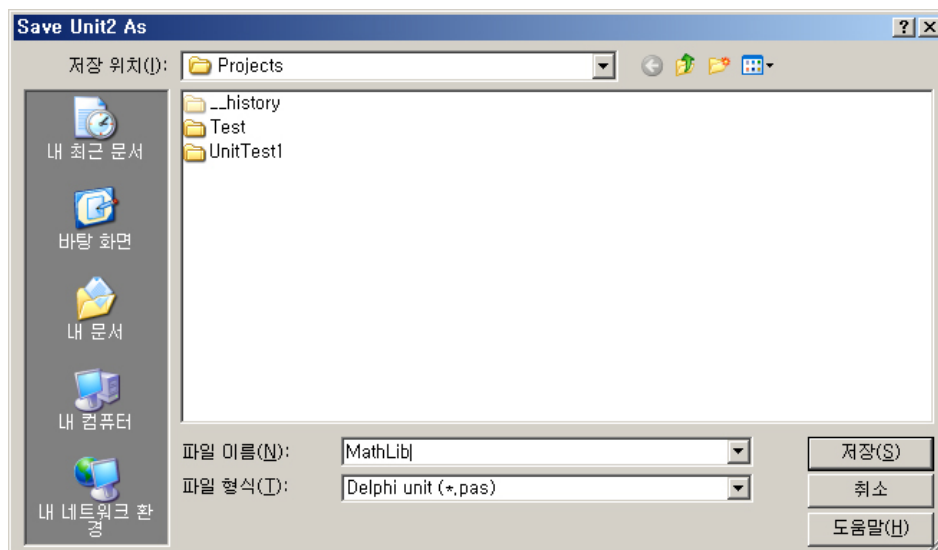
위의 소스에서 **Sum()**, **Mul()** 그리고 **Divide()** 함수를 나중에 다른 곳에서도 다시 사용하고자 한다고 가정하고, 유닛 파일을 새로 만들어 해당 유닛 파일에 [리스트 **Begin36**]과 같이 저장해 보도록 하겠습니다.

새로운 유닛 파일을 만들기 위해서 메뉴바에서  **New Items** 버튼을 클릭합니다. 이어서 [그림 **36**]과 같이 **Unit** 항목을 선택하고 **Ok** 버튼을 클릭 합니다. 이렇게 되면 새로운 유닛 파일이 생성되며 해당 유닛은 우선 현재 사용 중인 프로젝트 파일에 묶이게 됩니다.



[그림 36] 새로운 유닛 파일 추가하기

Ctrl+S 를 누르거나 Save 메뉴를 실행하여 우선 유닛 파일을 저장하도록 하겠습니다. [그림 36]처럼 저장할 유닛 파일의 이름을 물어오게 되면, MathLib 라고 적고 Ok 버튼을 클릭 합니다.



[그림 37] 유닛 파일의 저장

이제 유닛 파일에 [리스트 Begin36]과 같이 코딩하시면 유닛의 준비는 끝이 납니다.

[리스트 Begin36]

```
1 : unit MathLib;
2 :
3 : interface
4 :
5 : function Sum(a,b:integer):integer;
6 : function Mul(a,b:integer):integer;
7 : function Divide(a,b:integer):integer;
8 :
9 : implementation
10 :
11 : function Sum(a,b:integer):integer;
12 : begin
13 :   Result:= a + b;
14 : end;
15 :
16 : function Mul(a,b:integer):integer;
17 : begin
18 :   Result:= a * b;
19 : end;
20 :
21 : function Divide(a,b:integer):integer;
22 : begin
23 :   Result:= a / b;
24 : end;
25 :
26 : end.
```

유닛의 구조는 이어서 설명드리기로 하고, 우선은 작성된 유닛을 사용하는 방법부터 살펴보도록 하겠습니다.

[리스트 Begin37]

```
1 : program Begin37;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   MathLib in 'MathLib.pas';
8 :
9 : begin
10 :   WriteLn('4 + 2 = ', Sum(4, 2));
11 :   WriteLn('4 * 2 = ', Mul(4, 2));
12 :   WriteLn('4 / 2 = ', Divide(4, 2));
13 :
```

```
14 :   ReadLn;  
15 : end.
```

이제, [리스트 **Begin35**]를 [리스트 **Begin37**]처럼 바꾸면 동일하게 작동하는 소스가 됩니다.

7: 라인에서는 우리가 작성한 새로운 유닛 파일 **MathLib** 를 사용한다는 것을 선언하고 있습니다. **MathLib** in 'MathLib.pas' 를 **MathLib** 라고만 표현해도 됩니다. 파일의 경로를 확실하게 지정하고 싶을 때는 7: 라인처럼 in 을 이용하여 다음 문자열에 파일의 경로명을 포함한 파일명을 작성하시면 됩니다. 경로명의 루트(C:\ 등)가 없으면 현재 사용중인 프로젝트가 저장된 폴더가 자동으로 적용됩니다.

유닛을 잘 활용하면 소스를 간결하게 작성하여 효율적으로 관리할 수 있으며,

추후 재사용되는 부분을 이용하여 개발 시간을 단축하실 수도 있습니다.

Unit 의 구조

[리스트 Begin38]

```
1 : unit Begin38;
2 :
3 : interface
4 :
5 : uses
6 :   Classes, SysUtils;
7 :
8 : const
9 :   Limit = 1234567890;
10 :
11 : type
12 :   TPerson = record
13 :     Name : string;
14 :     Age : integer;
15 :   end;
16 :
17 : var
18 :   Person : TPerson;
19 :
20 : procedure DoSomething;
21 :
22 : implementation
23 :
24 : uses
25 :   MathLib;
26 :
27 : const
28 :   BufferSize = 1024;
29 :
30 : type
31 :   TBuffer = array [1..BufferSize] of integer;
32 :
33 : var
34 :   Buffer : TBuffer;
35 :
36 : procedure DoSomething;
37 : begin
38 :   // 이거 하고, 저것도 하고...
39 : end;
40 :
41 : end.
```

[리스트 Begin38]은 간단한 유닛의 구조를 예로 들어 본 것 입니다.

1: 라인에서 **unit** 이라는 예약어를 통해 해당 파일이 유닛임을 알리고 있습니다. 뒤에 이어서 **Begin38** 은 유닛명을 적어둔 것이며 파일명과 동일해야 합니다.

유닛은 크게 **interface** 와 **implementation** 으로 나뉩니다. **interface** 은 유닛외부에서 참조가 가능한 영역이며, 이곳에 선언된 것들은 모두 밖에서 사용할 수 있게 됩니다. "이 유닛에는 이러 이러한 것들이 있으니, 가져다 쓰시요" 라고 알려주는 역할을 하게 됩니다.

implementation 은 실제로 코딩하는 영역이며, 이곳에서 선언된 것들은 해당 유닛 밖에서는 사용할 수 없게 됩니다. **interface** 에서 "이러한 것들이 이 유닛에 있소" 라고 알려준 것들이 실제로는 어떻게 작동해야 하는 지를 구현하는 부분입니다.

interface 와 **implementation** 는 위에서 설명한 것 이외에는 거의 유사한 면이 많습니다.

uses 선언, 상수(**const**) 선언, **type** 선언, 변수(**var**) 선언, 함수(**procedure, function**) 선언 등을 할 수 있습니다.

상수란 변수와 달리 한 번 정해 두면 프로그램 내에서 끝까지 고정된 데이터 값을 가지게 되는 식별자입니다.

변동이 되지 않는다는 점을 빼고는 변수와 유사하게 취급됩니다.

다만, **interface** 에서는 함수에 대한 구현을 할 수 없습니다. **interface** 가 해당 유닛에 있는 요소들의 목록만을 관장하기 때문에 실제 동작하는 것은 언제나 **implementation** 의 몫입니다. 따라서, 20: 라인에서는 함수의 헤더 부분만 작성하고, 실제 동작은 **implementation** 에서 36-39: 라인과 같이 작성하게 됩니다.

uses 의 경우에는 위와 아래 어느 곳에 사용해도 크게 다를 것은 없으나, 다음과 같이 상호 참조는 불가능 합니다.

```
unit Unit1;
interface
uses
    Unit2;
implementation
end.

unit Unit2;
interface
uses
    Unit1;
implementation
end.
```

unit1 과 unit2, 둘 중 하나의 이상의 유닛에서 **uses** 가 **implementation** 의 밑에 선언되면 상호 참조가 가능 합니다.

begin end 에서의 end 와 달리 유닛 마지막을 알리는 end 의 경우에는 세미콜론(;) 대신 마침표(.)가 찍혀 있습니다. 파스칼(텔파이)에서 모든 소스 파일의 끝은 마침표로 끝나야 합니다.

이것은 프로젝트 파일도 마찬가지 입니다.

가위,바위,보 게임

기능분석

- 사용자는 가위, 바위, 보 중 하나를 선택한다.
- 컴퓨터는 무작위로 가위, 바위, 보 중 하나를 선택한다.
- 승리자를 화면에 표시한다.

구현방법 설계



[그림 38] 플로우 차트



가위, 바위, 보 게임의 승패를 구하는 부분은 입문자에게 어려울 듯 하여 간단한 설명을 하겠습니다. 일단 아래와 같이 가위, 바위, 보를 일렬로 반복해서 늘어 놓도록 하겠습니다.

가위, 바위, 보, 가위, 바위, 보, ...

우선, 사용자가 가위,바위,보 중 하나를 선택했다고 가정하겠습니다. 이때, 컴퓨터가 사용자가 선택한 것의 두 칸 뒤에 것을 선택하면 컴퓨터가 진 경우가 됩니다. 둘이 같은 것을 내면 비긴 것이며, 위의 두 가지 경우가 아니면 컴퓨터가 이긴 경우입니다. 이처럼 가위, 바위, 보가 순환하는 성질을 이용하면 쉽게 승패를 찾을 수 있습니다.

따라서, 사용자가 가위를 선택했다면 컴퓨터가 두 칸 뒤의 보를 선택했을 경우에는 컴퓨터가 진 것이 되고, 같은 것을 낸 경우는 비긴 것으로 생각하면 됩니다. 두 가지 경우가 아닌 경우에는 컴퓨터가 이긴 것입니다.

구현

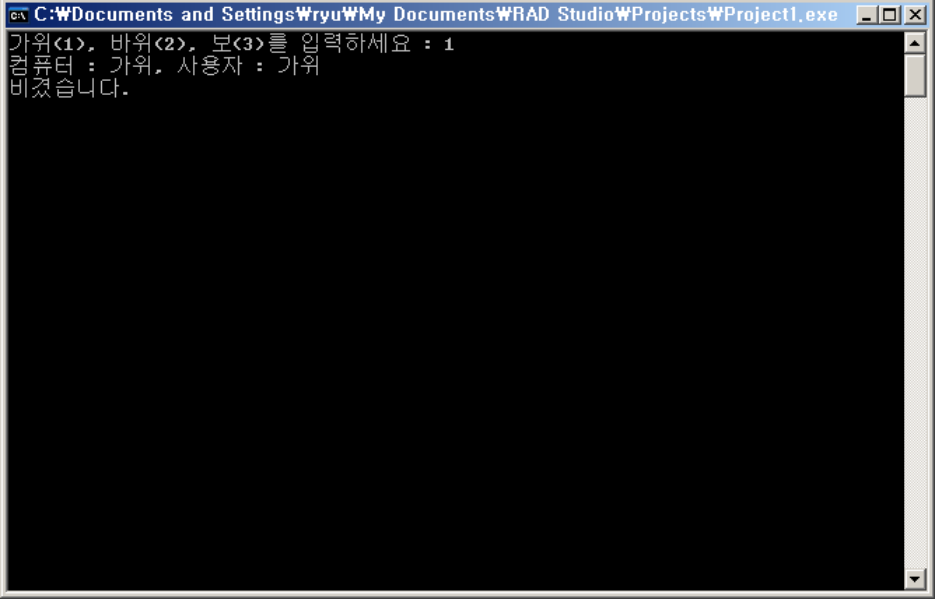
텔파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin39** 로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다.

[리스트 Begin39]

```
1 : program Begin39;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   PC, Player : integer;
10 :   stPC, stPlayer : string;
11 :
12 : begin
13 :   Randomize;
14 :
15 :   PC:= Random(3);
16 :
17 :   Write('가위 (1), 바위 (2), 보 (3)를 입력하세요 : ');
18 :   ReadLn(Player);
19 :   Player:= Player - 1;
20 :
21 :   case PC of
22 :     0 : stPC:= '가위';
23 :     1 : stPC:= '바위';
24 :     2 : stPC:= '보';
25 :   end;
26 :
27 :   case Player of
28 :     0 : stPlayer:= '가위';
29 :     1 : stPlayer:= '바위';
30 :     2 : stPlayer:= '보';
31 :   end;
32 :
33 :   WriteLn('컴퓨터 : ', stPC, ', 사용자 : ', stPlayer);
34 :
35 :   if PC = Player then WriteLn('비겼습니다.')
36 :   else if PC = ((Player + 2) mod 3) then WriteLn('이겼습니다.')
37 :   else WriteLn('졌습니다.');
```

```
38 :
39 :   ReadLn;
40 : end.
```

실행결과



```
C:\Documents and Settings\Wryu\My Documents\WRAD Studio\Projects\Project1.exe
가위<1>, 바위<2>, 보<3>를 입력하세요 : 1
컴퓨터 : 가위, 사용자 : 가위
비겼습니다.
```

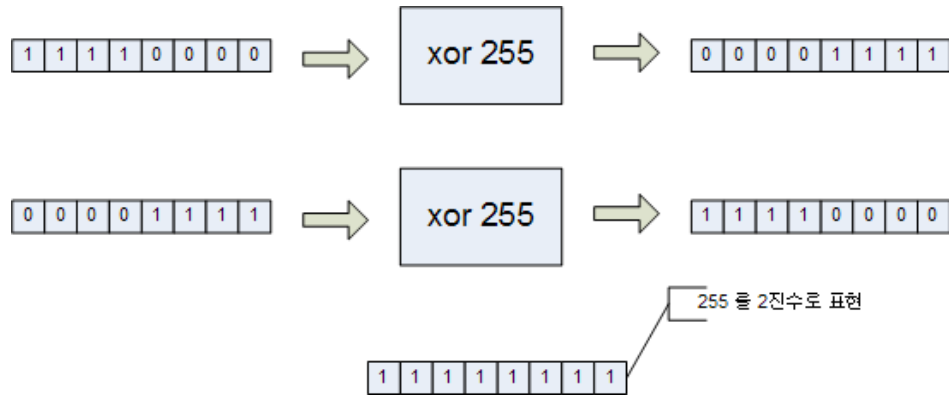
[그림 39] 리스트 Begin39 의 실행 결과

문자열 암호화

기능분석

- 주어진 문자열을 암호화 하여 화면에 표시한다.
- 암호화 된 문자열을 원래대로 복원하여 화면에 표시한다.

구현방법 설계



[그림 40] xor 를 이용한 데이터 암호화 및 해독

문자열의 정보를 알 수 없도록 변형시키는 일은 아주 간단합니다. 문자열도 배열의 하나이며 각 요소는 0 부터 255 의 숫자의 값을 가진 데이터를 가집니다. 이 데이터의 값을 1 씩 더하거나 하는 식의 간단한 산술 연산으로도 문자열은 원래의 의미를 알 수 없을 정도로 복잡하게 변합니다.



물론 간단한 변환일 수록 다른 사람에 의해서 해독될 가능성이 높습니다. 하지만, 지금 우리가 작성하는 예제는 수준 높은 암호화를 사용하는 것이 아니라, 간단히 암호화 하고 다시 원상태로 복구하는 것을 다룰 예정입니다.

여러가지 방법이 있을 수 있겠지만, 암호화 후 다시 원상태로 복구하는 것을 간단하게 할 수 있는 방법은 [그림 40]과 같이 xor 비트 연산을 이용하는 것입니다.

xor 비트 연산을 통해서 대상 데이터를 0 이 아닌 값으로 연산을 하게 되면, "1 → 0, 0 → 1" 처럼 대상 데이터의 모든 비트는 반전됩니다. 이렇게 반전된 문자열은 원래의 모습을 알아볼수 없을 정도로 변형이 됩니다. 이제 원상태로 변경하는 것은 다시 같은 값으로 xor 연산을 통해 데이터의 모든 비트를 반전시키는 것 입니다.

이렇게 xor 연산자는 비트의 반전을 일으키기 때문에 짝수로 반복하면 항상 원래의 데이터로 쉽게 복구 시킬 수 있습니다.

구현

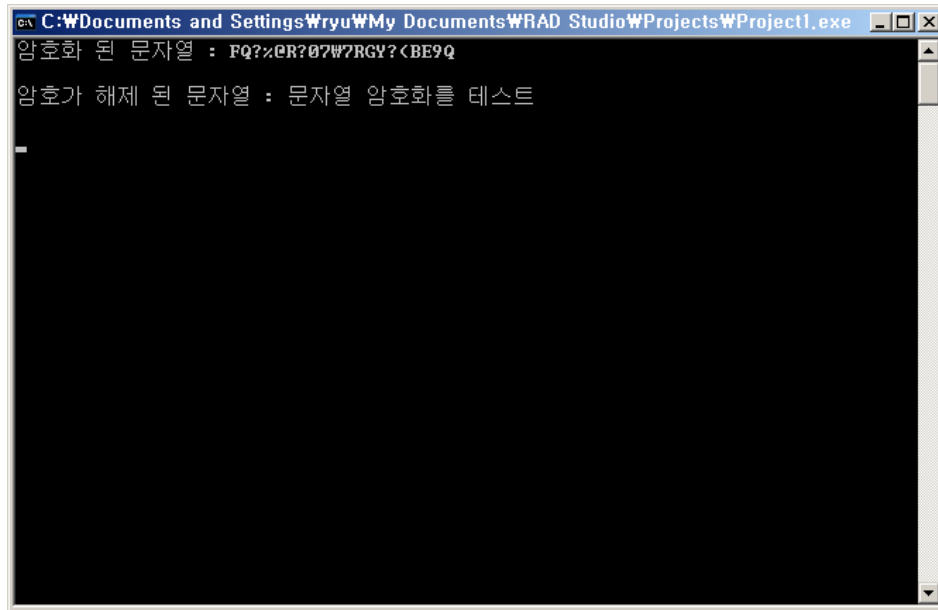
텔파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin39** 로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다.

[리스트 Begin40]

```
1 : program Begin40;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Text : string;
10 :
11 : function Encode(Text:string):string;
12 : var
13 :   Loop : Integer;
14 : begin
15 :   Result:= '';
16 :   for Loop := 1 to Length(Text) do
17 :     Result:= Result + Char( Byte(Text[Loop]) xor 255 );
18 : end;
19 :
20 : begin
21 :   Text:= '문자열 암호화를 테스트';
22 :
23 :   Text:= Encode(Text);
24 :   WriteLn('암호화 된 문자열 : ', Text);
25 :   WriteLn;
26 :
27 :   Text:= Encode(Text);
28 :   WriteLn('암호가 해제 된 문자열 : ', Text);
29 :   WriteLn;
30 :
31 :   ReadLn;
32 : end.
```

17: 라인의 255 대신 0 이 아닌 다른 숫자를 입력해보셔도 암호화는 동작합니다.

실행결과



```
C:\Documents and Settings\Wryu\My Documents\RAD Studio\Projects\Project1.exe
암호화 된 문자열 : FQ?%eR?07W7RGY?<BE9Q
암호가 해제 된 문자열 : 문자열 암호화를 테스트

```

[그림 41] 리스트 Begin40 의 실행 결과

미로 찾기

기능분석

- 미로를 화면에 표시한다.
- 방향키로 현재 위치를 이동한다.
- 벽을 통과하거나 지나칠 수 없다.

CRT unit 구하기

미로 찾기 예제에서는 콘솔 창에서 사용할 수 있는 몇 가지 유용한 함수가 미리 작성되어 있는 CRT unit 을 사용합니다. CRT unit 은 코드웨이(<http://www.codeway.co.kr>)의 델파이 자료실 게시판에서 다운 받으실 수 있습니다.

```
http://www.codeway.co.kr/board/bbs/tb.php/Delphi\_PDS/56
```

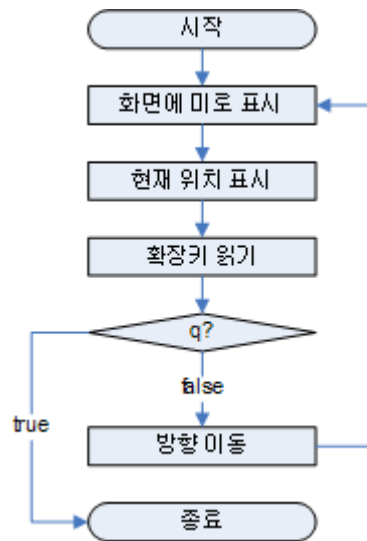
다운받으신 파일의 압축을 해제하고, 해당 파일을 여러분들의 프로젝트 파일이 있는 폴더에 복사해 주시거나, 델파이가 설치된 폴더 밑에 Lib 폴더에 두 파일 모두를 복사해주면 됩니다.

델파이 버전에 따라 아래의 폴더에 저장하시면 됩니다. 단, 설치시 폴더를 따로 지정하지 않은 경우에만 가능합니다.

```
델파이 7 이하 버전 사용 시 : C:\Program Files\Borland\Delphi7\Lib
```

```
델파이 2007 사용 시 : C:\Program Files\CodeGear\RAD Studio\5.0\lib
```




구현방법 설계



[그림 42] 플로우 차트

키가 눌리지면 해당 키가 **q** 일 때는 프로그램을 종료합니다. 방향키의 경우에는 해당 방향키가 뜻하는 좌표로 현재 위치를 이동합니다. 벽에 부딪히거나 미로의 범위를 넘어서면 이동을 중단합니다.

구현

텔파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin39** 로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다

[리스트 Begin41]

```
1 : program Begin41;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils, Windows, Crt;
7 :
8 : var
9 :   Key : Char;
10 :   Loop, X, Y, OldX, OldY : integer;
11 :   Miro : array [1..13] of string = (
12 :     '#####',
13 :     '#',
14 :     '##### #',
15 :     '#',
16 :     '# #####',
17 :     '#####',
18 :     '# #####',
19 :     '#',
20 :     '# #####',
21 :     '#####',
22 :     '# ##',
23 :     '#',
24 :     '#####'
25 :   );
26 :
27 : begin
28 :   X:= 1;
29 :   Y:= 1;
30 :
31 :   while true do begin
32 :     GotoXY(0, 0);
33 :     for Loop := 1 to 13 do WriteLn(Miro[Loop]);
34 :
35 :     GotoXY(X, Y);
36 :     Write('@');
37 :
38 :     OldX:= X;
39 :     OldY:= Y;
40 :
41 :     Key:= GetExtendedKey;
42 :     case Key of
```

```

43 :          #75 : X:= X - 1;
44 :          #77 : X:= X + 1;
45 :          #72 : Y:= Y - 1;
46 :          #80 : Y:= Y + 1;
47 :          'q' : Exit;
48 :      end;
49 :
50 :      if Copy(Miro[Y+1], X+1, 1) = '#' then begin
51 :          X:= OldX;
52 :          Y:= OldY;
53 :      end;
54 :  end;
55 : end.

```

32: 라인의 **GotoXY()** 함수는 콘솔 화면상에서 주어진 좌표 (x, y)로 커서를 이동시킵니다. 이어서, 36: 라인에서 @ 를 화면에 표시합니다.

41: 라인에서는 눌러진 키를 읽어 옵니다. **GetExtendedKey()** 는 확장키를 포함해서 읽어오는 함수입니다. 확장키라는 것은 일반적인 문자를 표시하는 키가 아닌 특수한 의미를 가진 키들로 방향키와 펄스키들이 확장키에 속합니다. 확장키 값이 읽혀졌을 때, 아래와 같은 숫자는 각각 방향키 중 하나를 뜻 합니다.

```

#75 : 왼쪽 커서 키
#77 : 오른쪽 커서 키
#72 : 윗쪽 커서 키
#80 : 아래쪽 커서 키

```

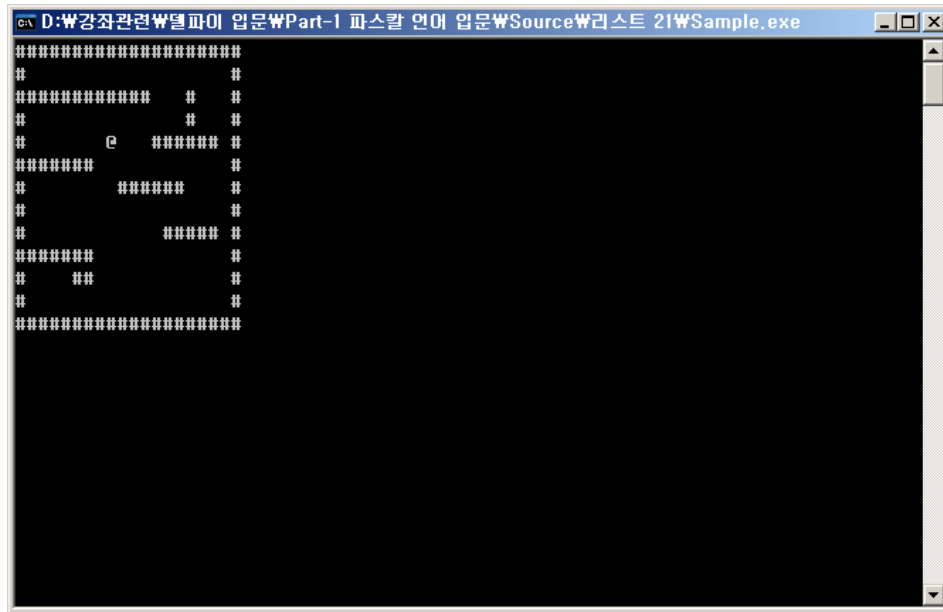
#65 와 'A' 는 같은 뜻이 됩니다.

다음에 숫자를 바로 붙여서 쓰면 해당 숫자가 의미하는 아스키 코드표의 문자가 됩니다.

42-48: 라인에서는 방향키의 경우 좌표를 수정하고, q 가 눌러졌을 경우에는 프로그램을 종료하게 됩니다.

50-53: 라인에서는 새로 이동한 위치에 # 문자가 있으면 벽이라고 인식하게 하기 위해서, 위치를 이동하기 이전의 위치가 저장된 (OldX, OldY) 좌표로 다시 복원하고 있습니다. 때문에, @ 표시가 # 으로 표시된 벽으로는 이동할 수 없게 됩니다.

실행결과



[그림 43] 리스트 Begin41 의 실행 화면

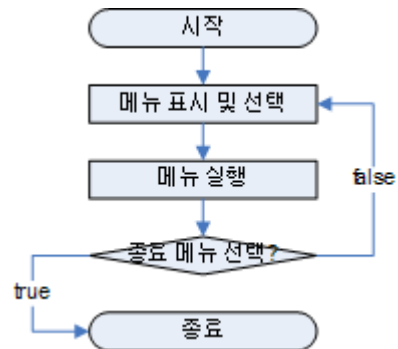
주소록 만들기

기능분석

- 주소를 입력한다.
- 이름으로 주소를 검색한다.
- 주소록 전체를 출력한다.

예제를 마치고 나면, 주소록 정보를 수정하거나, 파일에 저장하여 보존하는 것까지 연구해보시기 바랍니다.



구현방법 설계



[그림 44] 플로우 차트

- 미로찾기와 비슷한 방식으로 메뉴를 선택하고 선택된 메뉴의 내용에 따라 메뉴를 실행합니다.
- 종료 메뉴를 선택하면 반복을 무시하고 프로그램을 종료 합니다.
- 선택할 수 있는 메뉴는 다음과 같습니다.
 - 주소록 입력 하기
 - 이름으로 주소 검색 하기
 - 주소록 전체 출력 하기

구현

델파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **Begin39** 로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다

[리스트 Begin42]

```
1 : program Begin42;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils, Crt;
7 :
8 : type
9 :   TAddress = record
10 :     Name : string;
11 :     Phone : string;
12 :     Address : string;
13 :   end;
14 :
15 : var
16 :   iDataSize : integer = 0;
17 :   AddressList : array [1..100] of TAddress;
18 :
19 : function SelectMenu:integer;
20 : begin
21 : end;
22 :
23 : procedure do_InputAddress;
24 : begin
25 : end;
26 :
27 : procedure do_SearchAddress;
28 : begin
29 : end;
30 :
31 : procedure do_ListupAddress;
32 : begin
33 : end;
34 :
35 : begin
36 :   repeat
37 :     case SelectMenu of
38 :       1 : do_InputAddress;
39 :       2 : do_SearchAddress;
40 :       3 : do_ListupAddress;
41 :       9 : Break;
```

```

42 :     end;
43 :     until false;
44 : end.

```

우선 [그림 43]의 플로우 차트를 만족하는 소스부터 작성해보았습니다.

9-13: 라인에서는 주소록 데이터를 저장할 구조를 정의하고 있습니다. 이름과 전화번호 그리고 주소 세 가지의 요소로 주소정보를 저장하기로 하였습니다.

17: 라인에서는 주소록을 실제로 저장할 변수를 선언한 것 입니다. 한 명이 아닌 여러명의 주소를 저장하기 위해서 배열을 사용했습니다. 현재는 100 까지가 한계로 되어 있습니다.

16: 라인에서는 주소록 데이터의 갯수를 보관하기 위한 변수를 선언하고, 초기 값을 0 으로 지정하였습니다. 아무런 데이터가 저장되어 있지 않다는 뜻 입니다.

36-43: 라인을 반복하면서, **37:** 라인의 **SelectMenu()** 함수를 통해서 사용자가 메뉴를 선택할 수 있도록 할 예정입니다. **43:** 라인에서 조건을 무조건 **false** 로 주었기 때문에, 반복은 무한히 이루어지게 됩니다. 하지만, **41:** 라인이 실행된다면, **Break** 로 인해서 반복이 중단되고 프로그램은 종료될 것 입니다.

37-42: 라인을 통해서 선택된 메뉴의 번호에 따라 아래와 같은 함수들을 실행 할 것입니다.

- 선택된 번호가 1 일 때, **do_InputAddress()**
- 선택된 번호가 2 일 때, **do_SearchAddress()**
- 선택된 번호가 3 일 때, **do_ListupAddress()**
- 선택된 번호가 9 일 때는 반복을 벗어나도록 하였습니다.

이제 **SelectMenu()** 함수를 작성해보도록 하겠습니다. 이전의 모든 것은 다 잊어버리고 **SelectMenu()** 가 해야할 일만 신경쓰시기 바랍니다. 이런식으로 프로그램을 작성하는 것은 **Top-Down** 방식이라고 합니다. 동시에 여러 가지를 신경쓰지 않고 한곳에 집중할 수 있기 때문에 효율적인 프로그래밍이 가능합니다.

```

1 : function SelectMenu:integer;
2 : begin
3 :     ClrScr;
4 :
5 :     GotoXY(10, 6);
6 :     Write('  1. 주소입력');
7 :
8 :     GotoXY(10, 7);
9 :     Write('  2. 주소검색');
10 :
11 :     GotoXY(10, 8);

```



```

12 :   Write('   3. 주소 목록 보기');
13 :
14 :   GotoXY(10, 9);
15 :   Write('   9. 종료');
16 :
17 :   GotoXY(10, 11);
18 :   Write('메뉴를 선택하세요 : ');
19 :
20 :   ReadLn(Result);
21 : end;

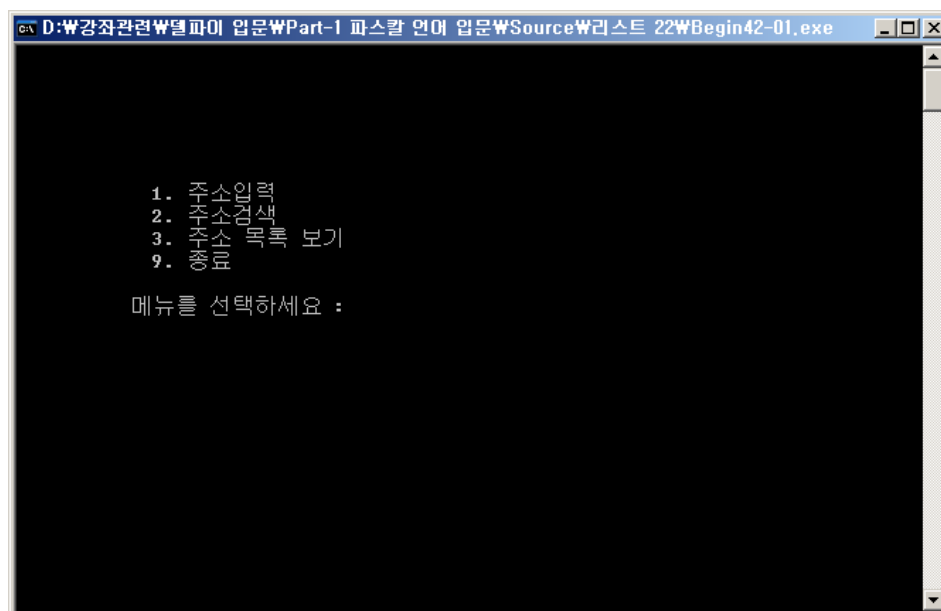
```

[리스트 Begin42]의 19-21: 라인을 위의 소스처럼 수정하고 실행해 보시기 바랍니다.

1, 2, 3 숫자 중 하나를 선택하면 해당 함수가 실행되고, 9 를 선택하면 프로그램이 종료됩니다. 다만, 1,2,3 숫자를 통해서 실행될 함수를 아직 완료하지 않았기 때문에 화면에는 아무런 변화가 없을 것입니다. 우선은 에러만 없다면 잘 작동하고 있는 것입니다.

위의 소스에서, 3: 라인에서 **ClrScr** 은 화면의 모든 것을 지우고 빈공간으로 만드는 함수입니다.

20: 라인은 숫자를 읽어서 함수의 결과 값(리턴 값)에 저장한다는 뜻입니다.



[그림 45] 메뉴 선택 화면면

이제 주소를 입력하는 함수를 작성해 보도록 하겠습니다.

```

1 : procedure do_InputAddress;
2 : begin
3 :   ClrScr;
4 :
5 :   iDataSize:= iDataSize + 1;
6 :
7 :   GotoXY(10, 6);
8 :   Write('  1. 이름 : ');
9 :   ReadLn(AddressList[iDataSize].Name);
10 :
11 :   GotoXY(10, 7);
12 :   Write('  2. 전화번호 : ');
13 :   ReadLn(AddressList[iDataSize].Phone);
14 :
15 :   GotoXY(10, 8);
16 :   Write('  3. 주소 : ');
17 :   ReadLn(AddressList[iDataSize].Address);
18 :
19 :   GotoXY(10, 11);
20 :   WriteLn('입력이 정상적으로 완료되었습니다. ');
21 :
22 :   ReadLn;
23 : end;

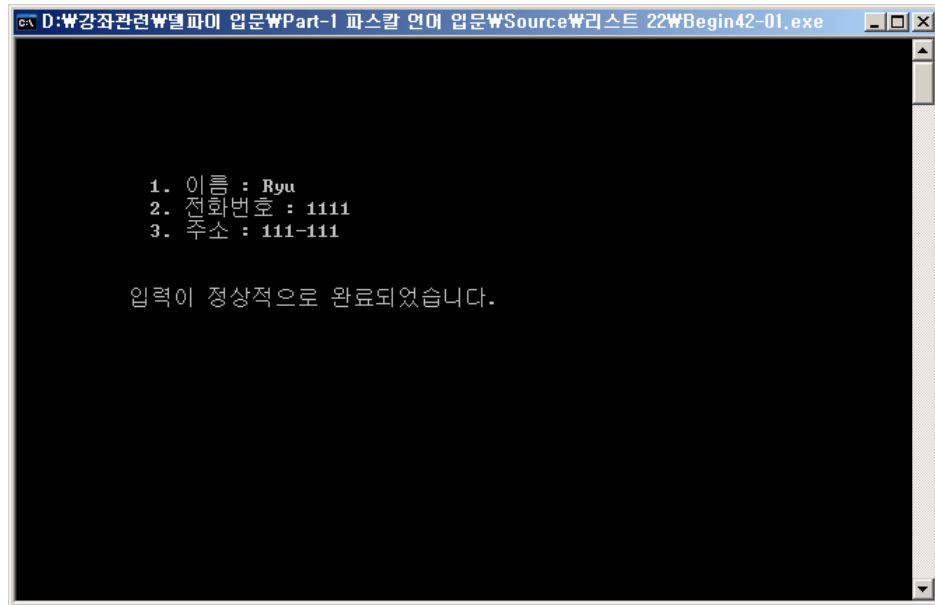
```

[리스트 Begin42]의 23-25: 라인을 위의 소스처럼 수정하고 실행해 보시기 바랍니다.

5: 라인에서는 새로 데이터를 입력할 것이기 때문에, 현재 데이터의 크기를 하나 늘여주고 있습니다.

이어서, 7-17: 라인을 통해서 각각 이름과 전화번호 그리고 주소를 입력받고 있습니다. 이때 주소 데이터를 저장할 공간은 `AddressList[iDataSize]` 로 지정되었습니다. `do_InputAddress` 함수가 실행될 때마다, `iDataSize` 가 증가되기 때문에, 새로운 입력이 발생할 때마다, 각각 다른 공간에 저장될 것입니다.

한글을 입력할 경우에는 오작동이 있을 수 있으니, 이점 유의하시기 바랍니다.



[그림 46] 주소 입력 장면

다음은 입력된 주소의 전체를 표시하는 `do_ListupAddress()` 함수를 작성해보도록 하겠습니다.

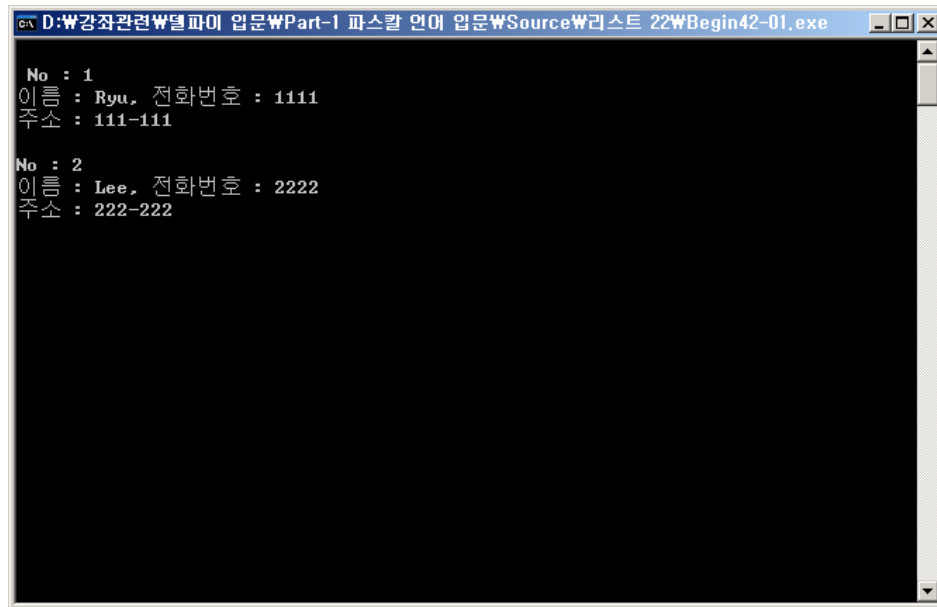
```

1 : procedure do_ListupAddress;
2 : var
3 :   Loop : integer;
4 : begin
5 :   ClrScr;
6 :
7 :   for Loop:= 1 to iDataSize do begin
8 :     WriteLn('No : ', Loop);
9 :     WriteLn('이름 : ', AddressList[Loop].Name, ', 전화번호 : ',
AddressList[Loop].Phone);
10 :    WriteLn('주소 : ', AddressList[Loop].Address, #13#10);
11 :   end;
12 :
13 :   ReadLn;
14 : end;

```

[리스트 Begin42]의 32-33: 라인을 위의 소스처럼 수정하고 실행해 보시기 바랍니다.

테스트를 위해서 "1. 주소입력" 메뉴를 몇 차례 실행하여, 몇 개의 주소록 정보를 입력하도록 하겠습니다. 주소록 입력을 마치고 나면, "3. 주소목록 보기" 메뉴를 실행하여 전체 목록을 살펴보도록 하겠습니다.



[그림 47] 주소목록 보기 실행 화면

이제 마지막으로 `do_SearchAddress()` 함수를 작성해 보도록 하겠습니다.

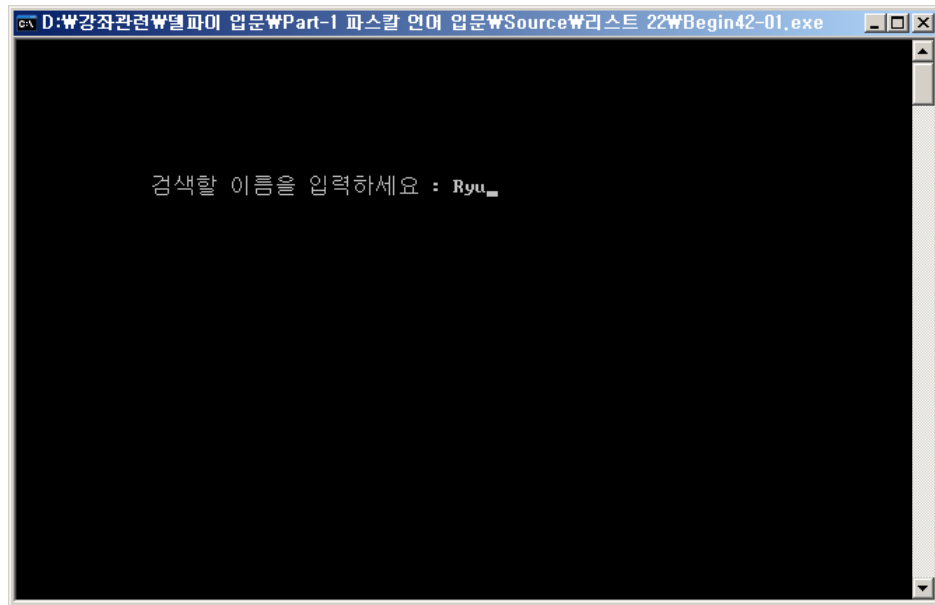
```

1 : procedure do_SearchAddress;
2 : var
3 :   Loop : integer;
4 :   stName : string;
5 : begin
6 :   ClrScr;
7 :
8 :   GotoXY(10, 6);
9 :   Write('  검색할 이름을 입력하세요 : ');
10 :   ReadLn(stName);
11 :
12 :   for Loop:= 1 to iDataSize do
13 :     if stName = AddressList[Loop].Name then begin
14 :       ClrScr;
15 :       WriteLn('No : ', Loop);
16 :       WriteLn('이름 : ', AddressList[Loop].Name,
17 :             ', 전화번호 : ', AddressList[Loop].Phone);
18 :       WriteLn('주소 : ', AddressList[Loop].Address, #13#10);
19 :     end;
20 :
21 :   ReadLn;
22 : end;

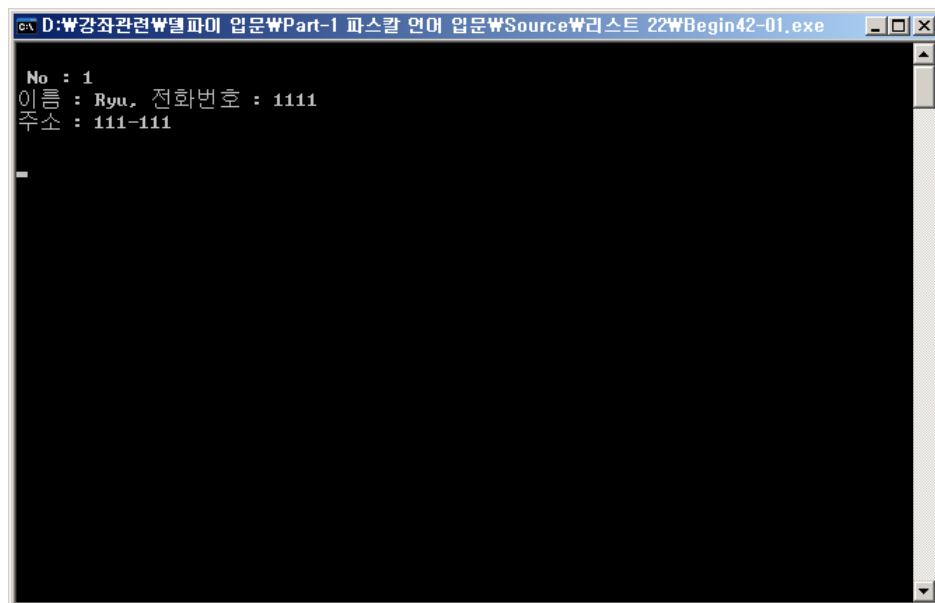
```

[리스트 Begin42]의 27-29: 라인을 위의 소스처럼 수정하고 실행해 보시기 바랍니다.

테스트를 위해서 "1. 주소입력" 메뉴를 몇 차례 실행하여, 몇 개의 주소록 정보를 입력하고, "2. 주소검색" 메뉴를 실행하여 원하는 이름을 입력해보시기 바랍니다.



[그림 48] 검색할 이름을 입력하는 장면

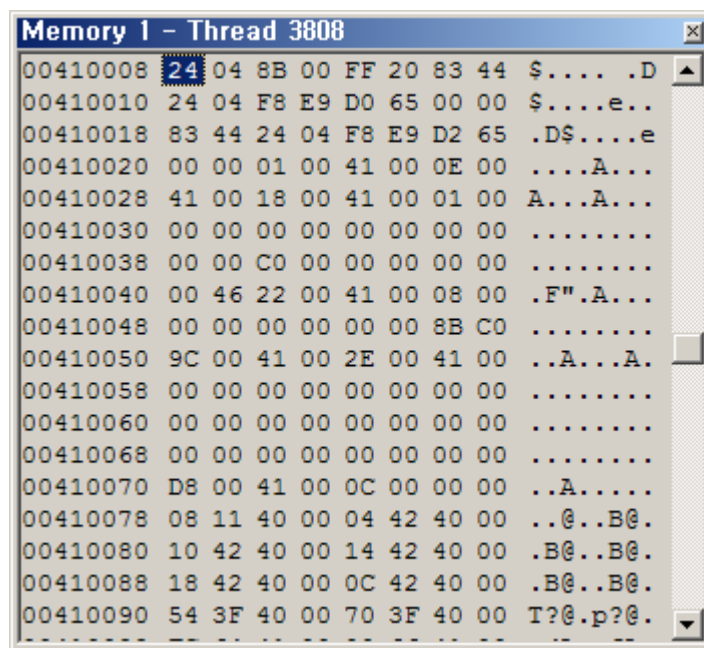


[그림 49] 검색된 결과 화면

변수란 무엇인가?

변수는 데이터를 담아둘 수 있는 일종의 박스입니다. 변수라는 박스는 컴퓨터의 메모리라는 진열장에 보관하게 됩니다. 또한, 박스에 넣을 수 있는 물건(데이터)의 종류는 박스(변수)의 종류마다 다릅니다. 이 때문에 박스는 그 종류에 따라서 크기도 다릅니다.

우리는 박스에 물건을 넣어두고, 해당 박스에 있는 물건을 확인하기 위해서는 진열장에서의 위치를 알고 있어야 합니다. 변수의 관점에서는, 이 위치를 주소라고 부릅니다.



[그림 50] 실행 중인 델파이 프로그램의 메모리 공간의 일부분

[그림 50]은 실행 중인 델파이 프로그램의 메모리 공간의 일부를 표시한 것입니다. 왼쪽에 보이는 00410008 과 같은 값들이 바로 메모리 상의 주소입니다. 변수의 내용을 확인하기 위해, [그림 50]과 같은 메모리 상의 주소를 외우거나 사용하는 것은 매우 번거롭습니다. 따라서, 프로그래밍에서는 주소를 이용해서 데이터를 저장하고 열람하는 대신, 변수 이름이라는 식별자를 사용합니다.

직장내에서 상사가 부하직원에게 서류를 누군가에게 전달하라는 지시를 한다는 가정을 하겠습니다.

- Case 1

- 3층 301 호 정문에서 두 번째 줄 다섯번째 책상에 있는 직원에게 이 서류를 가져다 주게.

- Case 2

- 김아무개 대리에게 이 서류를 가져다 주게.

"Case 1"의 경우는 주소를 직접 이용해서 프로그래밍을 하는 경우를 빗댄 것이며, "Case 2"의 경우에는 정확한 주소(위치) 대신 사람의 이름(식별자)를 통해서 프로그래밍 하는 경우를 비유적으로 설명한 것 입니다. 다만, 변수는 김아무개 대리처럼 여기 저기 옮겨다닐 수 없으며, 한 번 정해진 메모리 공간에 고정됩니다.

변수에 데이터를 입력하는 방법은 아래와 같습니다.

```
var  
  
  a : integer;  
  
begin  
  
  // 변수이름 := 데이터;  
  
  a := 3;
```

var a : integer; 선언을 통해서 여러분이 소스 상에서 **a** 라고 표기하면, 컴퓨터가 정수를 담을 수 있는 메모리 공간을 확보하고, 해당 메모리의 주소 정보를 변수 **a** 와 연결 시킵니다.

a := 3 은 "정수형 데이터 3 을 **a** 라는 이름이 실제로 가르키는 메모리 주소 공간에 저장하라" 라는 뜻 입니다. 실제의 상황에서는 박스에 담긴 물건을 빼내야 다른 물건을 담을 수 있지만, 변수에서는 새로운 데이터를 입력하게 되면, 기존에 데이터는 자동으로 사라지고, 새로운 데이터는 덮어쓰는 방식을 사용합니다.

변수는 생성될 때부터 언제나 데이터를 가지고 있습니다.

대입문을 통해서 데이터를 저장하지 않아도, 변수에는 이미 데이터가 있다는 점에 유의하시기 바랍니다.

하지만, 여러분들이 대입문을 통해서 데이터를 저장하지 않으면, 어떤 값이 있는 지 확신할 수 없습니다.

따라서, 한 번도 데이터를 저장하지 않은 변수를 사용하게되면 어떤 결과가 나올 지 예측할 수 없습니다.

이런 이유로 변수를 선언하자 마다 특정 데이터를 데이터에 저장하는 경우가 많습니다.

이런 경우를 **변수의 초기화**라고 합니다.

변수에서 데이터를 꺼내보는 방법에 대해서 설명하도록 하겠습니다. 우선, 변수에서 데이터를 꺼낸다는 표현은 맞지 않습니다. 예제와 설명을 통해서 자주 쓰이는 "데이터를 꺼낸다"라는 표현이 왜 틀리는지 알아 보겠습니다.

```
a := 3;
```

```
b := a;
```

위의 소스를 실행하면, **a := 3**에서 변수 **a**에는 3이라는 데이터가 들어가게 됩니다. **b := a**과정에서는 우선 **a**에 있는 데이터 값을 복사해 옵니다. 그리고, 복사된 데이터를 **b**에 입력하게 됩니다. 즉, 프로그래밍에 있어서 변수에서 데이터를 꺼내서 해당 변수에서 데이터는 사라지고, 데이터가 이동되는 일은 없습니다.

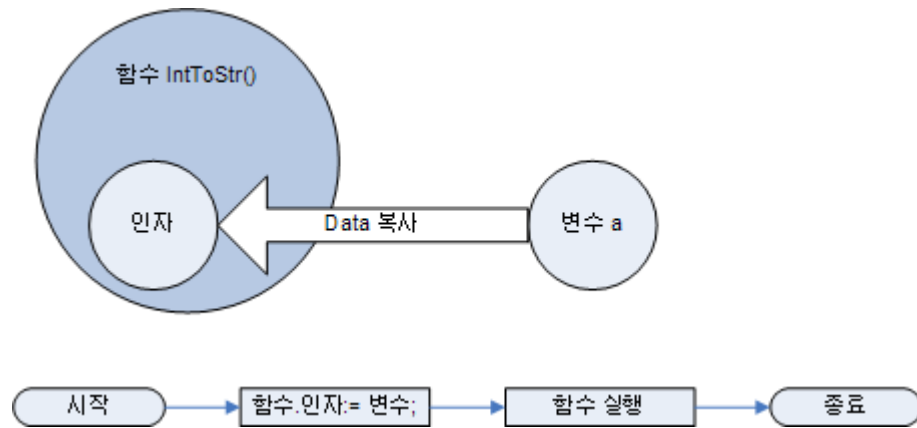
프로그래밍에서 변수를 사용하는 방법은 아래와 같은 절차를 사용합니다.

- 변수에 데이터를 입력
- 변수에 있는 데이터 값을 확인
- 변수에 있는 데이터를 복사해서 다른 변수나 메모리 공간으로 이동

이번에는, 함수의 인자에 변수를 입력하는 상황을 살펴보도록 하겠습니다.

```
var  
  
  St : string;  
  
  a : integer;  
  
begin  
  
  a:= 1234;  
  
  St:= IntToStr(a);
```

위와 같은 상황을 "IntToStr 함수에 변수 **a** 를 전달한다" 라고 표현하는 경우가 있습니다. 이 또한 틀린 표현입니다. 정확하게는 변수 **a** 안에 있는 데이터를 함수에 전달하는 것입니다. 전달되는 것은 변수가 아닌 변수 안에 있는 값을 복사한 사본 데이터 입니다.



[그림 51] IntToStr(a) 함수 실행 과정

포인터와 포인터 변수

대부분 "포인터"와 "포인터 변수" 라는 용어를 구별하지 않고 사용하고 있습니다. 하지만, 두 용어는 확연하게 다릅니다. 포인터는 메모리 상의 주소를 뜻하며, 포인터 변수는 메모리 상의 주소의 값을 저장할 수 있는 변수 타입입니다.

일반적으로 "포인터" 라고 하면, "포인터 변수"를 뜻하는 것으로 통용되고 있습니다.

프로그래밍에서 가장 까다로운 개념 중 하나인 포인터 변수는 일반 변수와 다를 것이 없습니다. 그저 정수형 변수가 정수만 다루듯이, 포인터 변수는 포인터(메모리 주소 값)만을 다룰 뿐입니다. 변수 또는 데이터 들의 형태에 따라 연산하는 방식이 다르듯이, 포인터의 경우에도 포인터 연산자가 따로 준비되어 있습니다.

주소 값(포인터)를 알아오는 @ 연산자

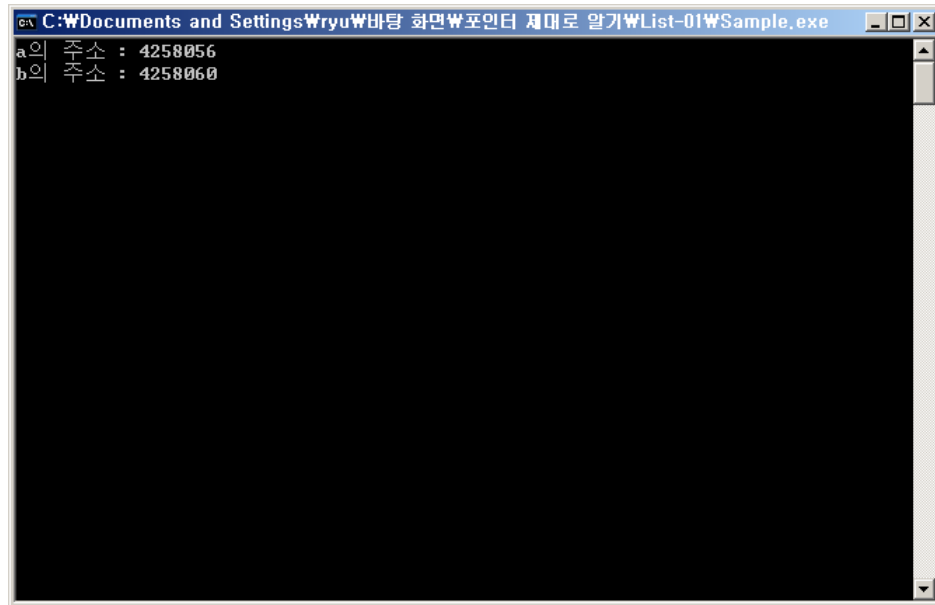
[리스트 Pointer01]

```
1 : program Pointer01;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   a, b : integer;
10 :
11 : begin
12 :   WriteLn('a 의 주소 : ', Integer(@a));
13 :   WriteLn('b 의 주소 : ', Integer(@b));
14 :
15 :   ReadLn;
16 : end.
```

[리스트 Pointer01]를 실행하면 [그림 52] 과 같은 결과를 얻을 수 있습니다. 앞서 설명드린 바와 같이 변수라는 것은 메모리 공간에 위치하는 박스와 같습니다. @ 연산자는 이 박스가 차지하고 있는 메모리 공간 중에서 제일 앞에 있는 메모리의 주소 값을 가져오게 됩니다.

@ 연산자는 변수뿐 아니라, 포인터 변수 및 함수와 같은 다른 식별자에도 사용할 수 있습니다.

파스칼(델파이) 문법은 형태에 대해서 엄격하며, WriteLn 함수에서 포인터 값을 출력할 수 없기 때문에, 12-13: 라인에서처럼 Integer() 를 통해서 정수 값으로 변경해서 출력하고 있습니다.



[그림 52] @ 연산자의 결과

[그림 52]에서는 두 변수의 메모리 주소 값이 4 바이트 크기로 차이가 납니다. 이것은 정수형 변수의 크기가 4 바이트이기 때문입니다.

연달아서 변수를 선언한다고 무조건 변수들이 메모리 공간에 바로 인접해서 생성된다는 보장은 없습니다.

포인터(주소 값) 변수(박스)로 변환해 주는 ^ 연산자

[리스트 Pointer02]

```
1 : program Pointer02;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   p : pointer;
```

```

10 :   a, b : integer;
11 :
12 : begin
13 :   a:= 3;
14 :
15 :   WriteLn('a 의 주소 : ', Integer(@a));
16 :   WriteLn('b 의 주소 : ', Integer(@b));
17 :
18 :   p:= @a;
19 :   WriteLn('p 의 값 = a 의 주소 : ', Integer(p));
20 :
21 :   WriteLn('a 의 값 : ', a);
22 :   WriteLn('p 에 저장된 주소에 있는 정수형태 박스의 값 : ',
Integer(p^));
23 :
24 :   ReadLn;
25 : end.

```

p 는 포인터 변수입니다. p:= @a 를 통해서 p 는 정수형 변수 a 가 있는 메모리 주소의 값을 가지게 됩니다. 이어서, p 의 값을 화면에 표시해보면, @a 를 통해서 얻어진 값과 일치하는 것을 아실 수 있습니다. 이때, p 의 경우에는 @ 연산자를 붙이지 않는다는 것을 유의해야 합니다. 즉, p 는 주소를 기억하는 변수이기 때문에 변수에 있는 값을 그대로 꺼내면 되는 것입니다.

이어서, 22: 라인에서는 ^ 연산자를 통해서 p 의 주소가 가르키는 박스로 변경하는 작업이 실행됩니다. 이때 박스의 크기를 알 수 없기 때문에 Integer() 를 통해서 주소로부터 얻어진 박스를 정수형으로 변형시키고 있습니다. 즉, p^ 는 자체가 일반 변수처럼 사용됩니다.

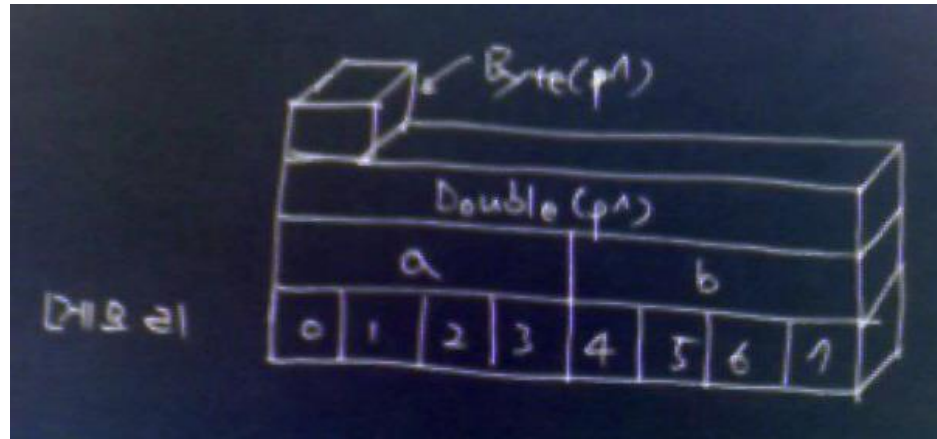
```

C:\Documents and Settings\Wryu\바탕 화면\포인터 제대로 알기\List-01\Sample.exe
a 의 주소 : 4258056
b 의 주소 : 4258060
p 의 값 = a 의 주소 : 4258056
a 의 값 : 3
p 에 저장된 주소에 있는 정수형태 박스의 값 : 3

```

[그림 53] ^ 연산자 사용의 예

Double() 과 같은 타입 캐스팅(타입 변환)을 실행하면, 원래 주소 값에 있는 박스보다 큰 변수가 되면서, 원하지 않는 에러가 생길 수 있습니다. 반대로 **Byte()** 과 같이 **Integer** 보다 작은 박스로 변경하면, **a** 가 차지하고 있는 메모리 공간에서 **1 byte** 만을 사용할 수 있는 임시 박스를 만들어 낼 수 있습니다.



[그림 54] $\text{Double}(p^1)$, $\text{Byte}(p^1)$ 를 통해 메모리에 저장된 값을 접근하는 경우

타입이 정해진 포인터의 활용

[리스트 Pointer03]은 [리스트 Pointer02]를 조금 수정한 소스입니다.

[리스트 Pointer03]

```
1 : program Pointer03;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   p : ^integer;
10 :   a, b : integer;
11 :
12 : begin
13 :   a:= 3;
14 :
15 :   WriteLn('a 의 주소 : ', Integer(@a));
16 :   WriteLn('b 의 주소 : ', Integer(@b));
17 :
18 :   p:= @a;
19 :   WriteLn('p 의 값 = a 의 주소 : ', Integer(p));
20 :
21 :   WriteLn('a 의 값 : ', a);
22 :   WriteLn('p 에 저장된 주소에 있는 정수형태 박스의 값 : ', p^);
23 :
24 :   ReadLn;
25 : end.
```

[리스트 Pointer02]와 달라진 것은 9: 라인에서 p 변수의 타입명을 pointer 대신 ^integer 를 사용한 것입니다. 이것은 포인터 변수 선언의 다른 형태 입니다.

모든 변수는 변수명(식별자)를 쓰고 콜론(:)을 붙인 다음 변수의 타입명을 적으면 됩니다. 이때, pointer 라고 타입명을 정하면 형태가 없는 포인터 변수라는 뜻입니다. 그리고, 변수 타입명 앞에 ^를 붙이면 해당 변수 타입을 유지하는 포인터 변수가 됩니다. 우선 p : pointer; 와 p : ^integer; 모두 포인터 변수 p 를 생성한다는 것을 알아두시기 바랍니다.

이제 그 차이점을 살펴해보도록 하겠습니다. [리스트 Pointer02]와 [리스트 Pointer03]은 거의 대부분이 똑 같은 코드로 되어 있습니다. 다른 곳을 찾아보면, 22: 라인에서 p^ 라고 표시된 부분이 [리스트 Pointer02]에서는 Integer(p^) 로 표현되었다는 것입니다.

이미 ^ 연산자를 포인터 변수 뒤에 붙여서 사용하면, 포인터 변수가 가르키고 있는 포인터(메모리 주소)의 위치에 있는 변수(박스)처럼 사용할 수 있다고 말씀드렸습니다. 그리고, 이때 포인터 변수의

타입이 지정되어 있으면, [리스트 Pointer02]처럼 포인터 변수를 통해서 얻어진 박스의 크기를 임의로 지정할 필요없이, 바로 지정된 타입의 변수가 되는 것입니다.

연산자는 포인터 변수를 일반 변수로 변환하는 힘이 있습니다.

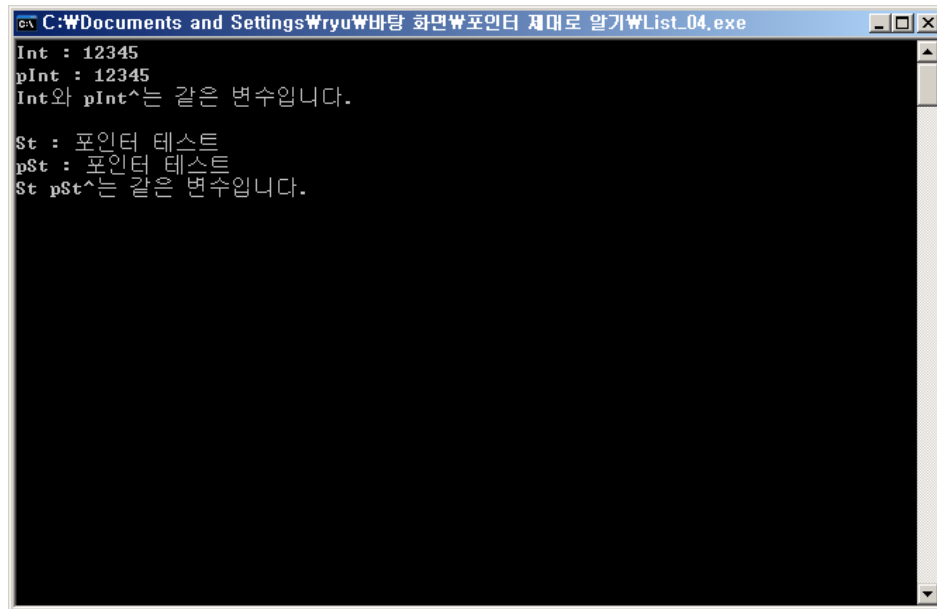
이때 pointer 처럼 타입이 없는 포인터 변수로 지정된 경우에는 변환된 변수의 타입을 강제로 지정해줘야 하지만,

^integer 처럼 형태를 지정한 경우에는 자동으로 해당 타입(변수 타입)의 변수가 되는 것입니다.

[리스트 Pointer04]

```
1 : program Pointer04;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Int : integer;
10 :   pInt : ^integer;
11 :
12 :   St : string;
13 :   pSt : ^string;
14 :
15 : begin
16 :   Int:= 12345;
17 :   pInt:= @Int;
18 :   WriteLn('Int : ', Int);
19 :   WriteLn('pInt^ : ', pInt^);
20 :   WriteLn('Int 와 pInt^는 같은 변수입니다. ');
21 :
22 :   WriteLn;
23 :
24 :   St:= '포인터 테스트';
25 :   pSt:= @St;
26 :   WriteLn('St : ', St);
27 :   WriteLn('pSt : ', pSt^);
28 :   WriteLn('St pSt^는 같은 변수입니다. ');
29 :
30 :   ReadLn;
31 : end.
```


[리스트 Pointer04]에서 보는 것처럼, 포인터 변수에 타입을 정해서 선언하고, 해당 포인터 변수에 ^ 연산자를 뒤에 붙이게 되면 정해진 타입의 변수로 변환됩니다.



```
C:\Documents and Settings\Wryu\바탕 화면\포인터 제대로 알기\List_04.exe
Int : 12345
pInt : 12345
Int와 pInt^는 같은 변수입니다.

St : 포인터 테스트
pSt : 포인터 테스트
St pSt^는 같은 변수입니다.
```

[그림 55] 리스트 Pointer04 의 실행 결과

메모리 할당 및 해제

포인터 변수는 주소를 지정해야 사용할 수 있습니다. 지금까지 보아온 방식은 이미 메모리 공간을 확보한 변수의 주소 값을 가져와서 포인터 변수에 대입하는 방법이었습니다. 이번에는 메모리 할당을 통해서 포인터 변수에 주소를 입력하는 방법을 사용해보도록 하겠습니다. 메모리 할당이란 메모리 공간을 확보하고, 해당 주소값을 포인터 변수에 입력하는 과정입니다.

우선 [리스트 Pointer05]를 통해서 설명을 이어가도록 하겠습니다.

[리스트 Pointer05]

```
1 : program Pointer05;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   n : integer;
10 :   i : ^integer;
11 :
12 : begin
13 :   n:= 1;
14 :   i:= @n;
15 :   WriteLn('n : ', n);
16 :   WriteLn('i^ : ', i^);
17 :   WriteLn;
18 :
19 :   i^:= 2;
20 :   WriteLn('n : ', n);
21 :   WriteLn('i^ : ', i^);
22 :   WriteLn;
23 :
24 :   New(i);
25 :   i^:= 3;
26 :   WriteLn('n : ', n);
27 :   WriteLn('i^ : ', i^);
28 :   WriteLn;
29 :
30 :   Dispose(i);
31 :
32 :   ReadLn;
33 : end.
```

13-17: 라인의 실행결과는 [그림 56]에서 윗 단란처럼 n 과 i^{\wedge} 값이 동일하게 1 이라는 값으로 표시됩니다. 이는 n 이 정수형 변수를 저장할 공간을 가지고 있으며, i 는 해당 주소를 가지는 포인터 변수이기 때문에 포인터를 해제하는 연산자 $^{\wedge}$ 를 이용하면 n 과 동일한 변수가 되기 때문입니다.

19: 라인에서는 i^* 의 값을 2로 변경하였습니다. 그리고 20-22: 라인의 실행 결과 역시 n 과 i^* 의 값이 동일하게 나옵니다. 변경한 것은 i^* 뿐이지만, i 가 n 의 주소를 가지고 있기 때문에 i^* 는 n 과 동일한 변수처럼 행동한 다는 것이 증명되었습니다.

24: 라인에서는 `New()` 함수를 통해서 i 크기의 메모리 공간을 새로 만들고 해당 주소를 i 에 입력하였습니다.

25: 라인에서는 i^* 값을 3으로 변경하였습니다. 이후 26-28: 라인을 통해서 출력된 값에서는 n 과 i^* 값이 서로 다르게 나타납니다. i 가 이제 새로운 주소를 가르키고 있기 때문에 두 변수 사이의 싱크는 깨진 것입니다. 이제 i^* 와 n 은 전혀 별개의 변수가 되었습니다.

30: 라인에서는 포인터 변수 i 에게 할당된 메모리를 해제합니다.

```

D:\강좌\관련\포인터 제대로 이해하기\Source\List_05.exe
n : 1
i^ : 1

n : 2
i^ : 2

n : 2
i^ : 3

```

[그림 56] 리스트 Pointer05의 실행 결과

[리스트 Pointer05]가 실행될 때 메모리 공간 속의 변화를 살펴보도록 하겠습니다.

13: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	1	n
@2	?	i

- i 의 주소가 초기화 되어 있지 않는 상태이기 때문에, i^* 로 접근할 수 없습니다.

14: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	1	n
@2	@1	i

- i^* 연산을 실행하면 i 가 가지고 있는 주소 값 @1 의 메모리 공간으로 이동합니다.
- 이후, i 의 포인터 타입인 정수형으로 형변환을 일으킵니다.
- 결과적으로 i^* 는 n 과 같은 공간을 가르키는 정수형 변수가 됩니다.

19: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	2	n
@2	@1	i

- 결과적으로 n 의 값을 2 로 변경하였습니다.
- i^* 를 통해서 @1 의 주소값을 가지는 정수형 변수로 변환하면 n 과 같은 2 의 값을 갖는 정수형 변수가 됩니다.

24: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	2	n
@2	@3	i
@3	?	식별자 없음

- $New(i)$ 를 통해서 새로운 메모리 공간 @3 을 확보하였습니다. 이 공간에 해당하는 식별자는 없습니다.
- $New(i)$ 를 통해서 i 의 값이 @3 을 가지게 됩니다.

25: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	2	n
@2	@3	i
@3	3	식별자 없음

- i^* 를 통해서 우선 i 가 가르키는 메모리 공간 @3으로 이동하여 해당 공간에 해당하는 정수형 변수로 변환합니다.
- $i^* = 3$ 을 통해서 @3 공간에 3이라는 정수형 값을 입력합니다.

30: 라인까지 진행되었을 때,

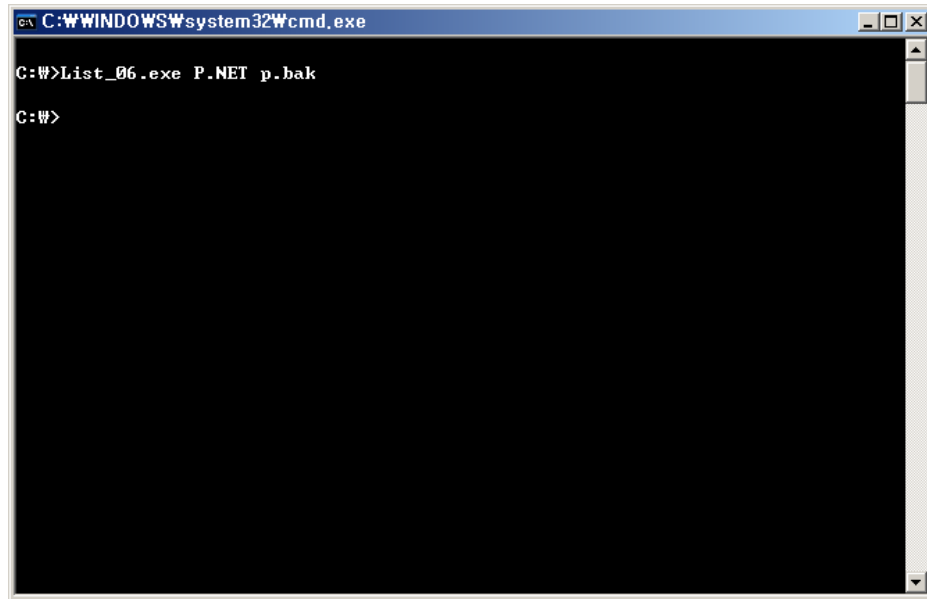
주소	데이터 (값)	식별자
@1	2	n
@2	@3	i

- **Dispose(i)**를 통해서 메모리 공간 @3은 더이상 해당 어플리케이션에서 사용할 수 없습니다.
- 하지만, i 는 여전히 @3을 가리키고 있다는 것을 유의해야 합니다.
- 이 같은 상황에서 i^* 와 같은 작업을 실행하면 **Access Violation** 에러가 발생할 수 있습니다.
- 무조건 발생할 것이라고 예측할 수 있는데, @3이 다른 곳에 사용하지 않는 순간까지는 에러가 발생하지 않을 수 있습니다.
- 하지만, 메모리가 해제된 공간에 접근하는 것은 상당히 위험한 상황을 초래할 수 있으니 조심하여야 합니다.

델파이에서 메모리를 할당하고 해제하는 방법은 주로 두 가지 방법을 사용하게 됩니다.

첫 번째는 위의 예제를 통해서 본 **New()**와 **Dispose()** 함수를 이용하는 것입니다. 이 방법은 주로 타입이 정해진 포인터 변수를 사용할 때 사용합니다.

두 번째 방법은 **GetMem()**와 **FreeMem()** 함수입니다. 이 방법은 주로 타입이 정해지지 않은 포인터 변수에 원하는 크기만큼 메모리를 할당하고 싶을 때 사용하게 됩니다.



[그림 57] 파일 복사 프로그램 실행 결과

GetMem() 와 FreeMem() 함수의 사용법을 설명하기 위해 간단한 예제를 List_06.dpr 작성하였습니다. [리스트 Pointer06]은 설명을 위한 프로그램입니다. 코드가 효율적이지 못한 부분이 있습니다. 유의하시기 바랍니다.

[리스트 Pointer06]은 [그림 57] 과 같이 콘솔 명령을 통해서 첫 번째 파라미터에 해당하는 파일을 복사하여 두 번째 파라미터에 해당하는 파일을 만들어 냅니다.

[리스트 Pointer06]

```
1 : program Pointer06;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   BufferSize : integer;
10 :   FileIn, FileOut : File;
11 :
12 : procedure do_CopyFile;
13 : var
14 :   iRead : integer;
15 :   Buffer : pointer;
16 : begin
17 :   BufferSize:= FileSize(FileIn);
18 :   GetMem(Buffer, BufferSize);
19 :   try
```

```

20 :      BlockRead(FileIn, Buffer^, BufferSize, iRead);
21 :      BlockWrite(FileOut, Buffer^, iRead);
22 :      finally
23 :          FreeMem(Buffer);
24 :      end;
25 : end;
26 :
27 : begin
28 :     Assign(FileIn, ParamStr(1));
29 :     Assign(FileOut, ParamStr(2));
30 :
31 :     Reset(FileIn);
32 :     try
33 :         Rewrite(FileOut);
34 :         try
35 :             do_CopyFile;
36 :             finally
37 :                 Close(FileOut);
38 :             end;
39 :         finally
40 :             Close(FileIn);
41 :         end;
42 : end.

```

18: 라인에서 포인터 변수 **Buffer** 에 메모리를 파일 크기만큼 할당하고 있습니다. 파일 크기가 대단히 큰 경우라면 이부분에서 심각한 문제를 발생합니다. 따라서, 이 소스는 **GetMem()** 와 **FreeMem()** 함수의 이해를 위해서만 사용하시기 바랍니다. 버퍼 크기를 작게 하고 파일 크기만큼 읽고 쓰기를 반복해서 사용하는 것이 보다 일반적인 방법입니다.

23: 라인에서는 포인터 변수가 가르키는 주소에 할당된 메모리를 해제하고 있습니다. 이제 해당 부분은 다른 곳에서 재사용할 수 있게 됩니다.

19-24: 라인에서 **try finally end** 는 "try finally 사이의 코드가 실행되다가 에러가 나더라도 finally end 사이의 코드는 반드시 실행해라" 라는 뜻 입니다. 즉, 어떠한 경우에도 23: 라인이 실행되어 일단 할당 받은 메모리가 에러 등의 이유로 해제 되지 않고 남아있지 않도록 합니다. 메모리를 할당하는 부분에서 자주 사용하는 방법입니다. 논리적 결함이나 에러 등으로 메모리가 계속해서 낭비되는 것을 방지하기 위함 입니다.

메모리 할당과 해제가 일어나는 또 다른 상황 중 하나는 클래스를 실체화 시키는 과정입니다. 클래스로 선언한 변수의 경우에는 포인터 변수처럼 행동합니다. 실제로는 거의 포인터 변수라고 해도 무방합니다. 클래스로 선언한 변수를 오브젝트 레퍼런스 변수라고 부릅니다. 클래스에 대한 자세한 설명은 다음 장을 통해서 드리도록 하겠습니다.

클래스에 대한 지식이 없는 분들은 다음 장을 먼저 읽고 아래 내용을 읽는 것도 무방합니다.

[리스트 Pointer07]을 통해서 클래스를 실체화 시키는 과정에 대하여 설명하겠습니다.

[리스트 Pointer07]

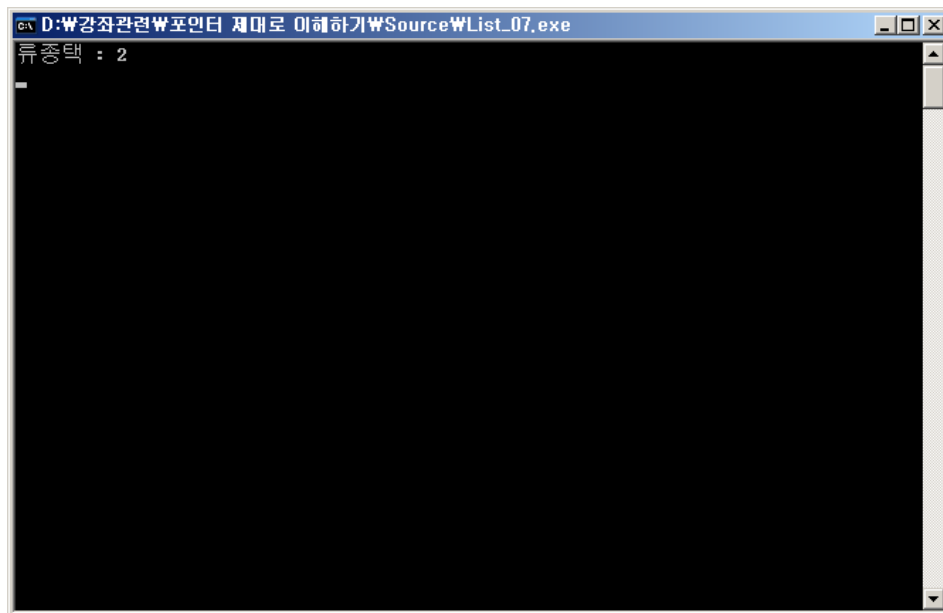
```
1 : program Pointer07;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     Name : string;
11 :     Age : integer;
12 :     procedure GrowUp;
13 :   end;
14 :
15 : procedure TPerson.GrowUp;
16 : begin
17 :   Age:= Age + 1;
18 : end;
19 :
20 : var
21 :   a : TPerson;
22 :
23 : begin
24 :   a:= TPerson.Create;
25 :   try
26 :     a.Name:= '류종택';
27 :     a.Age:= 1;
28 :     a.GrowUp;
29 :
30 :     WriteLn(a.Name, ' : ', a.Age);
31 :   finally
32 :     a.Free;
33 :   end;
34 :
35 :   ReadLn;
36 : end.
```

21: 라인에서는 오브젝트 레퍼런스 변수 **a**를 **TPerson** 클래스 타입으로 설정하였습니다.

24: 라인에서는 클래스 **TPerson**의 생성자를 통해서 **TPerson**의 크기만큼 메모리를 할당 받아서 해당 주소를 **a** 변수에 대입하는 과정입니다. 즉, 오브젝트 레퍼런스 변수도 포인터와 같이 주소 값을 저장하는 변수입니다. 포인터와도 호환됩니다. 이 과정은 **New**(포인터 변수)와 같은 절차를 따르게 됩니다. 다만, 클래스 타입으로 선언된 변수의 경우에는 해당 클래스의 생성자(**Create()** 메소드)를 통해서 메모리를 할당받게 됩니다.

26: 라인에서는 **a**가 가르키는 객체 내부의 **Name**이라는 멤버 변수를 접근하는 과정입니다. 포인터 변수의 경우는 해당 포인터가 가르키는 메모리 공간으로 이동하여 포인터 변수의 타입의 변수로 전환하는 **^** 연산자를 사용했었습니다. 하지만, 레퍼런스 변수의 경우에는 **^** 연산자를 사용하지 않습니다. 하지만, 동작 원리는 포인터 변수에 **^** 연산자를 사용한 것과 동일 합니다.

32: 라인에서는 변수 **a** 에 할당된 메모리를 해제하는 과정입니다. **Dispose**(포인터 변수) 를 하는 것과 마찬가지로 절차를 가지게 됩니다. 다만, 오브젝트 레퍼런스 변수의 경우에는 클래스로부터 물려받은 소멸자 **Destroy()** 메소드를 통해서 메모리를 반환합니다. 하지만, 일반적으로 **Destroy()** 메소드를 직접 호출하지 않고 **Free()** 메소드를 통해서 **Destroy()** 메소드를 실행 합니다.



[그림 58] 리스트 Pointer07 의 실행 결과

클래스가 포인터 변수 타입과 유사한 행동하는 것에 대한 이해를 돕기 위해 아래와 같은 예제를 통해서 메모리 변화를 살펴보도록 하겠습니다.

```

1 : begin
2 :   a:= TPerson.Create;
3 :   try
4 :     a.Name:= '류종택';
5 :     a.Age:= 1;
6 :     a.GrowUp;
7 :
8 :     a:= TPerson.Create;
9 :
10 :    WriteLn(a.Name, ' : ', a.Age);
11 :  finally
12 :    a.Free;

```

```

13 : end;
14 :
15 : ReadLn;
16 : end.

```

2: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	@2	a
@2	?	식별자 없음

- TPerson.Create() 메소드에 의해서 @2 주소의 메모리 공간을 확보합니다.
- a:= TPerson.Create 대입문을 통해서 해당 주소 값을 a 변수의 공간에 저장합니다.
- 결과적으로 a 는 주소 값을 가지는 포인터 변수가 됩니다.

8: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	@3	a
@2	?	식별자 없음
@3	?	식별자 없음

- TPerson.Create() 메소드에 의해서 @3 주소의 메모리 공간을 확보합니다.
- a:= TPerson.Create 대입문을 통해서 해당 주소 값을 a 변수의 공간에 저장합니다.
- 결국 @2 는 아무도 가르키고 있지 않기 때문에 해당 주소로 이동하여 사용할 방법이 사라졌습니다.
- @2 에 저장된 내용(객체)은 이제 미아가 되어, 메모리만 차지하고 있습니다. 이러한 경우를 메모리 누수 현상이라고 합니다.

12: 라인까지 진행되었을 때,

주소	데이터 (값)	식별자
@1	@3	a
@2	?	식별자 없음
@3	X	메모리 해제

- `Free()` 메소드에 의해서 `@3` 주소에 있는 메모리가 해제되어 다른 곳에서 재사용할 수 있는 상태가 됩니다.
- 이때, `a` 는 여전히 `@3` 을 가리키고 있다는 점을 유의하시기 바랍니다.
- 포인터에서처럼 이런 경우에 `a` 를 계속 사용하면 심각한 에러가 발생할 수 있습니다.
- `@2` 에 해당하는 부분은 해제할 방법이 없기 때문에 프로그램 종료 시까지 공간을 소비하게 됩니다.

함수의 인자에 포인터 적용

이번에는 함수의 인자에서 포인터를 활용하는 방법을 살펴보도록 하겠습니다.

[리스트 Pointer08]

```
1 : program Pointer08;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   a, b : integer;
10 :
11 : procedure SwapIntegersFirstTry(a,b:integer);
12 : var
13 :   c : integer;
14 : begin
15 :   c:= a;
16 :   a:= b;
17 :   b:= c;
18 : end;
19 :
20 : procedure SwapIntegersSecondTry(var a,b:integer);
21 : var
22 :   c : integer;
23 : begin
24 :   c:= a;
25 :   a:= b;
26 :   b:= c;
27 : end;
28 :
29 : procedure SwapIntegersThirdTry(a,b:PInteger);
30 : var
31 :   c : integer;
32 : begin
33 :   c:= a^;
34 :   a^:= b^;
35 :   b^:= c;
36 : end;
37 :
38 : begin
39 :   a:= 0;
40 :   b:= 1;
41 :
```

```

42 :   SwapIntegersFirstTry(a, b);
43 :   WriteLn('SwapIntegersFirstTry');
44 :   WriteLn('a = ', a);
45 :   WriteLn('b = ', b);
46 :   WriteLn;
47 :
48 :   SwapIntegersSecondTry(a, b);
49 :   WriteLn('SwapIntegersSecondTry');
50 :   WriteLn('a = ', a);
51 :   WriteLn('b = ', b);
52 :   WriteLn;
53 :
54 :   SwapIntegersThirdTry(@a, @b);
55 :   WriteLn('SwapIntegersThirdTry');
56 :   WriteLn('a = ', a);
57 :   WriteLn('b = ', b);
58 :   WriteLn;
59 :
60 :   ReadLn;
61 : end.

```

다음은 42: 라인의 `SwapIntegersFirstTry(a, b)` 를 실행할 때 생기는 메모리 내의 변화를 추적한 것입니다.

14: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	0	a
@2	1	b
@3	0	함수.인자.a
@4	1	함수.인자.b
@5	?	함수.지역변수.c

- 함수의 인자 `a, b` 가 각각 메모리 공간을 차지하는 새로운 변수로 생성되었습니다.
- 42: 라인에서 넘겨 받은 값으로 각각 초기화 되어 있습니다.
- 9: 라인에서 선언한 변수들의 이름과 같지만, 함수 내에서는 `a, b` 는 함수의 인자로써만 인식이 됩니다.

15: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	0	a
@2	1	b
@3	0	함수.인자.a
@4	1	함수.인자.b
@5	0	함수.지역변수.c

16: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	0	a
@2	1	b
@3	1	함수.인자.a
@4	1	함수.인자.b
@5	0	함수.지역변수.c

17: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	0	a
@2	1	b
@3	1	함수.인자.a
@4	0	함수.인자.b
@5	0	함수.지역변수.c

18: 라인을 마치고 함수가 종료되었을 때,

- 함수의 인자를 포함해서 함수 내부의 지역변수가 소멸되고 메모리를 반환 합니다.
- 결국 스왑 작업은 함수 실행으로 새로 생겨난 변수들에서 일어났기 때문에 9: 라인에서 선언한 변수들의 값에는 아무런 변화가 없습니다.
- 이 결과는 [그림 59] 에서 확인하실 수 있습니다.

다음은 48: 라인의 SwapIntegersSecondTry(a, b) 를 실행할 때 생기는 메모리 내의 변화를 추적한 것입니다.

23: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	0	a
@2	1	b
@3	@1	함수.인자.a
@4	@2	함수.인자.b
@5	?	함수.지역변수.c

- 새로 생긴 인자(변수)들은 레퍼런스라고 하는 포인터 변수로 생성 됩니다.
- 레퍼런스 변수는 포인터 변수와 거의 동일하지만, 포인터 해제 연산자를 사용하지 않는 것이 다릅니다. 자동으로 포인터를 해제하기 때문에 일반 적인 변수와 사용하는 방법이 똑 같습니다. 다만, 레퍼런스 변수는 자신이 값을 가지고 있는 것이 아니고, 값을 가지고 있는 다른 변수의 주소 값을 가지고 있는 포인터 변수 입니다.

24: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	0	a
@2	1	b
@3	@1	함수.인자.a
@4	@2	함수.인자.b
@5	0	함수.지역변수.c

- 변수 **c** 에 함수의 인자 **a** 가 가르키고 있는 변수의 값, 즉, 9: 라인에서 선언한 변수 **a** 의 값이 입력됩니다.
- 이때, **c:= a^** 와 같이 표현하지 않아도 함수의 인자 **a** 의 메모리 주소를 따라가서 해당 변수의 값을 가져오는 것을 유의하여야 합니다.

25: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	1	a
@2	1	b
@3	@1	함수.인자.a
@4	@2	함수.인자.b
@5	0	함수.지역변수.c

26: 라인까지 실행되었을 때,

주소	데이터 (값)	식별자
@1	1	a
@2	0	b
@3	@1	함수.인자.a
@4	@2	함수.인자.b
@5	0	함수.지역변수.c

27: 라인을 마치고 함수가 종료되었을 때,

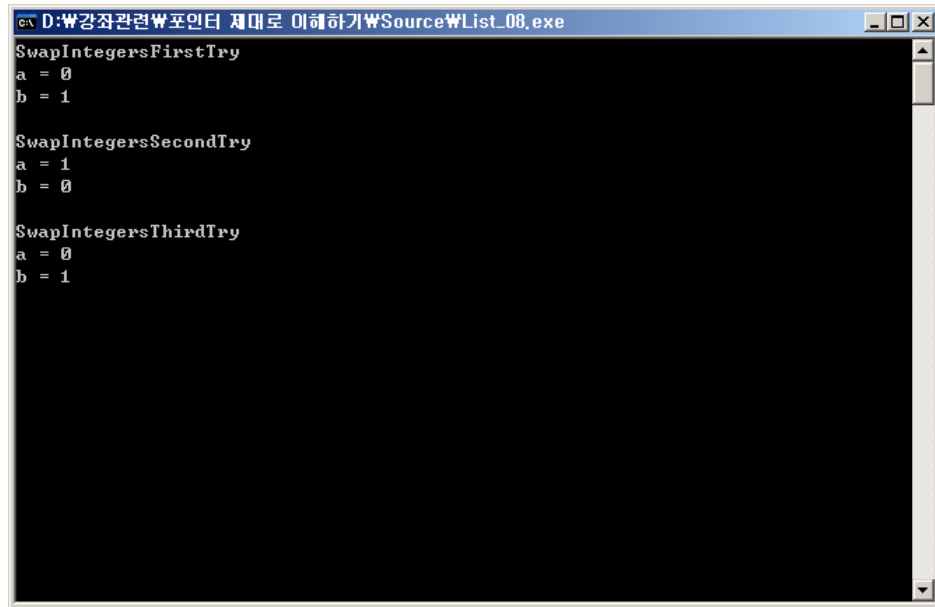
주소	데이터 (값)	식별자
@1	1	a
@2	0	b

- 함수의 인자를 포함해서 함수 내부의 지역변수가 소멸되고 메모리를 반환 합니다.
- [그림 59]에서 확인하실 수 있듯이, 9: 라인에서 선언한 변수들의 값이 서로 바뀌어져 있는 것을 확인하실 수 있습니다.

29-36: 라인의 `SwapIntegersThirdTry()` 함수의 경우는 `SwapIntegersSecondTry()` 를 포인터로 표현한 것입니다. 동작원리는 똑 같기 때문에 같은 결과를 [그림 59]에서 확인할 수 있습니다. 변수 a 와 b 의 값이 다시 뒤바뀌어 원래대로 되어 있습니다.

레퍼런스 변수를 사용할 때와 다른 점은, 33-35: 라인에서 보듯이 ^ 연산자를 통해서 작업하고 있다는 점 입니다. 포인터 변수는 이렇듯 원거리의 함수나 객체에게 자신을 복사해 주는 효과를 일으킬 수 있습니다. 마치 분신술처럼 양쪽에서 같은 변수의 값을 변경하거나 참조할 수 있는 것 입니다.

`SwapIntegersFirstTry()` 함수의 경우를 **Call by value** 방식이라 부르며, `SwapIntegersSecondTry()` 함수와 `SwapIntegersThirdTry()` 함수의 경우를 **Call by reference** 방식이라고 합니다.



```
D:\강좌\강좌관련\포인터 제대로 이해하기\Source\List_08.exe
SwapIntegersFirstTry
a = 0
b = 1

SwapIntegersSecondTry
a = 1
b = 0

SwapIntegersThirdTry
a = 0
b = 1
```

[그림 59] 리스트 Pointer08 의 실행 결과

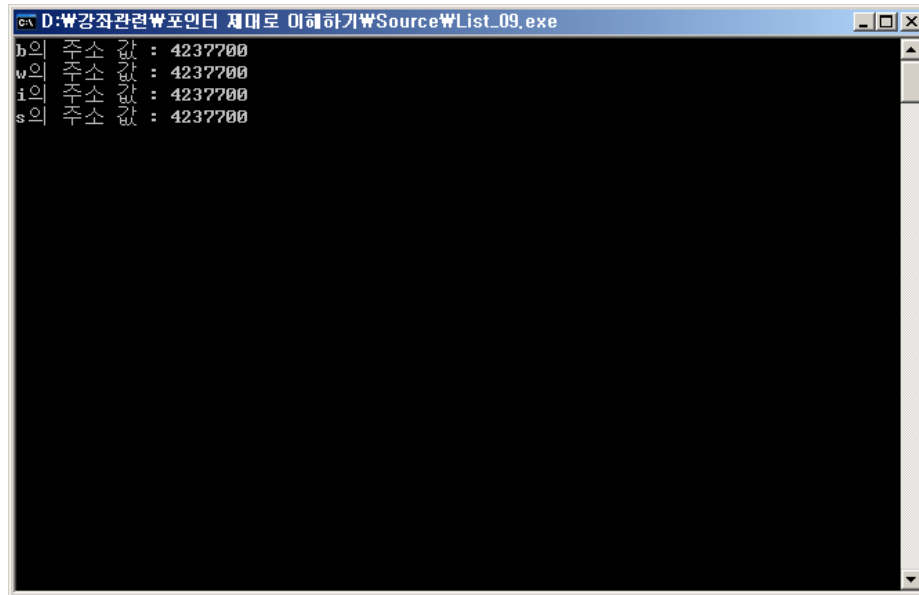
포인터의 호환성 문제

타입이 정해진 포인터 변수끼리의 호환성

[리스트 Pointer09]

```
1 : program Pointer09;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   b : ^byte;
10 :  w : ^word;
11 :  i : ^integer;
12 :  s : ^string;
13 :
14 :   Data : integer = 12345;
15 :
16 : begin
17 :   b:= @Data;
18 :   w:= @Data;
19 :   i:= @Data;
20 :   s:= @Data;
21 :
22 :   WriteLn('b 의 주소 값 : ', Integer(b));
23 :   WriteLn('w 의 주소 값 : ', Integer(w));
24 :   WriteLn('i 의 주소 값 : ', Integer(i));
25 :   WriteLn('s 의 주소 값 : ', Integer(s));
26 :
27 :   ReadLn;
28 : end.
```

17-20: 라인에서는 정수형 변수의 주소 값을 다양한 타입의 포인터 변수에 대입 시키는 것을 보실 수 있습니다. **Data** 자체는 정수형 변수이지만, 해당 변수의 메모리 주소는 그저 포인터일 뿐입니다. 따라서, 포인터(메모리 주소 값)은 어떠한 형태의 포인터에도 대입시킬 수 있습니다.



[그림 60] 리스트 Pointer09 의 실행 결과

하지만, 다음의 경우에는 컴파일이 되지 않습니다. **b, w, i, s** 변수들은 각각의 타입이 서로 다르기 때문입니다.

```

1 : var
2 :   b : ^byte;
3 :   w : ^word;
4 :   i : ^integer;
5 :   s : ^string;
6 :
7 :   Data : integer = 12345;
8 :
9 : begin
10 :   b:= @Data;
11 :   w:= b;
12 :   i:= w;
13 :   s:= i;

```

파스칼은 문법이 엄격하게 설계되어 있습니다.

파스칼이 교육용 언어의 목적으로 개발되었기 때문에,

초보자가 실수로 프로그램을 위험하게 작성하는 것을 방지하기 위함입니다.

포인터는 작은 실수로 인해서 치명적인 버그를 만들 수 있기 때문에, 조심해서 사용해야 합니다.

이제 다음과 같이 [리스트 Pointer10]을 통해서 서로 다른 타입을 가진 포인터 변수를 호환하여 사용하는 예제를 살펴보도록 하겠습니다.

[리스트 Pointer10]

```
1 : program Pointer10;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   b : ^byte;
10 :   w : ^word;
11 :   i : ^integer;
12 :   s : ^string;
13 :
14 :   Data : integer = 1234567890;
15 :
16 : begin
17 :   b:= @Data;
18 :   w:= Pointer(b);
19 :   i:= Pointer(w);
20 :   s:= Pointer(i);
21 :
22 :   WriteLn('b 의 주소 값 : ', Integer(b));
23 :   WriteLn('w 의 주소 값 : ', Integer(w));
24 :   WriteLn('i 의 주소 값 : ', Integer(i));
25 :   WriteLn('s 의 주소 값 : ', Integer(s));
26 :   ReadLn;
27 :
28 :   WriteLn('b^의 값 : ', b^);
29 :   ReadLn;
30 :
31 :   WriteLn('w^의 값 : ', w^);
32 :   ReadLn;
33 :
34 :   WriteLn('i^의 값 : ', i^);
35 :   ReadLn;
36 :
37 :   try
38 :     WriteLn('s^의 값 : ', s^);
```

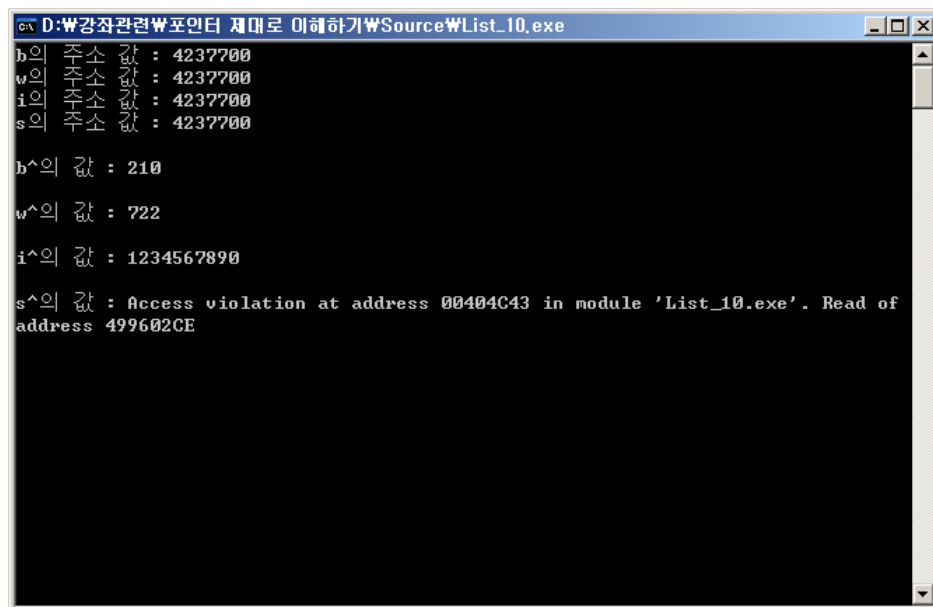
```

39 :   except
40 :       on E : Exception do WriteLn(E.Message);
41 :   end;
42 :   ReadLn;
43 : end.

```

타입이 다른 포인터 변수끼리 호환하여 사용하려면 각각의 타입으로 타입 캐스팅을 하셔도 됩니다. 하지만, 간단하게 **Pointer()** 로 통일하여 타입을 포인터로 전환하셔도 됩니다. 이것은 **List_09.dpr** 에서 17-20: 라인을 통해 주소 값을 지정하는 것과 동일한 효과를 나타냅니다.

하지만, [그림 61]에서처럼, 타입을 변화시킨 후에 출력되는 결과값을 보면, 원래 변수 **Data** 와 같은 타입의 포인터 변수 **i** 를 제외하고는 모두 다른 값을 출력하게 됨을 알 수 있습니다. 특히, 문자열의 경우에는 **Access Violation** 에러가 발생하는 것을 보실 수 있습니다. 이것은 정수형 변수가 차지하는 메모리 크기와 구조가 각각의 타입의 변수와 다르기 때문에 일어나는 현상입니다.



[그림 61] 리스트 Pointer10 의 실행 결과

Data	01001001	10010110	00000010	11010010
b^	01001001			
w^	01001001	10010110		
i^	01001001	10010110	00000010	11010010
s^	01001001	10010110	00000010	11010010

[그림 62] 리스트 Pointer10 이 실행될 때의 메모리 상태

b^{\wedge} 와 w^{\wedge} 의 값이 원래의 값에 형태가 전혀 나타나지 않는 것을 의아해하실 수도 있습니다. [그림 62]에서 보듯이 b^{\wedge} 이 가지는 메모리 공간은 **Data** 변수의 맨 앞의 1 바이트(8 비트) 입니다. 원래 데이터가 사실 이진수로 저장되어 있기 때문에, 이진수 형태에서는 원래 데이터의 형태가 그대로 남아있게 됩니다. 이것을 10 진수로 변환해서 출력하게 되면, [그림 2]와 같음을 확인하실 수 있습니다.

그런데, **Data** 와 똑 같은 크기를 가진 문자열 변수를 접근할 때는 왜 에러가 나는 것 일까요? 그것은 문자열 변수가 처리하는 값 역시 포인터이기 때문입니다. 즉, s^{\wedge} 를 실행하면, 변수의 **Data** 의 값이 가르키는 1234567890 번째 메모리 주소를 참조하게 됩니다. (정확하게는 4 바이트 차이가 나는 주소 값을 참조하면서 에러가 납니다) 우리는 단 한 번도 해당 주소의 메모리를 사용하겠다고 선언한 적이 없기 때문에 에러가 발생하게 됩니다.

예전 파스칼 문법에서 문자열 변수는 배열과 동일 시 취급되었습니다. 하지만, 현재 델파이에서는 포인터로 취급됩니다. 델파이에는 예전 파스칼 문법처럼 되돌릴 수 있는 컴파일 옵션이 있기 때문에, 예전 파스칼 코드와 호환성을 유지할 수도 있습니다. 하지만, 예전 파스칼 문법에서는 문자열의 길이가 255 까지 제한되어 있습니다. 현재는 2GB 까지 사용할 수 있습니다.

New()-Dispose(), GetMem()-FreeMem() 함수들끼리의 호환성

[리스트 Pointer11]은 New() 함수와 GetMem() 함수 그리고, Dispose() 함수와 FreeMem() 함수가 서로 호환되는 것을 예제로 보여드리기 위한 소스입니다.

[리스트 Pointer11]

```
1 : program Pointer11;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   i1, i2 : ^integer;
10 :
11 : begin
12 :   New(i1);
13 :   GetMem(i2, SizeOf(integer));
14 :
15 :   i1^:= 123456;
16 :   i2^:= i1^;
17 :
18 :   WriteLn('i1^의 값 : ', i1^);
19 :   WriteLn('i2^의 값 : ', i2^);
20 :
21 :   FreeMem(i1);
22 :   Dispose(i2);
23 :
24 :   ReadLn;
25 : end.
```

12: 라인의 New(i1) 은 실제 코드 상에서는 다음과 같습니다. 결국 New() 함수는 GetMem() 함수를 호출하게 되며, 13: 라인이 호출되는 것과 동일한 프로세스를 거치게 됩니다.

```
List_11.dpr.12: New(i1);
0040914E B804000000      mov eax,$00000004
00409153 E85C9BFFFF      call @GetMem
00409158 8BD8              mov ebx,eax
```

22: 라인의 Dispose(i2) 함수는 아래와 같이 컴파일 됩니다. 결국 21: 라인의 FreeMem() 함수와 같은 프로세스를 거치게 됩니다.

```
List_11.dpr.22: Dispose(i2);  
004091B6 BA04000000      mov edx,$00000004  
004091BB 8BC6              mov eax,esi  
004091BD E80E9BFFFF          call @FreeMem
```


오브젝트 래퍼런스 변수와 포인터 변수의 호환성

다음의 코드는 클래스가 포인터 타입처럼 행동한다는 것을 보여드리기 위한 예제입니다.

```
1 : procedure TForm1.Button1Click(Sender: TObject);
2 : var
3 :   Button : TButton;
4 : begin
5 :   Button:= TButton(Sender);
6 :   Button:= Sender as TButton;
7 :   Button:= Pointer(Sender);
8 :
9 :   Button.Visible:= false;
10 : end;
```

버튼을 클릭하면 해당 버튼을 보이지 않게 **Visible** 속성을 **false** 로 변경하는 것이 목적입니다. 이벤트가 발생하면 **Button1Click()** 이벤트 핸들러가 실행되면서, 어떤 버튼 객체가 클릭되었는 지 **Sender** 라는 인자를 통해서 알려주게 됩니다. 하지만, **Sender** 의 타입은 **TObject** 로 되어 있기 때문에, **Visible** 이라는 속성이 없습니다. 따라서, **Sender** 를 원래의 타입으로 타입 캐스팅 하는 것이 필요합니다.

5-7: 라인의 모두는 똑같은 결과를 가지게 됩니다. 모두 **Sender** 의 타입을 **TButton** 으로 전환하고, 변수 **Button** 은 해당 주소를 가르키게 됩니다.

포인터 이동

포인터의 이동은 타입이 정해진 포인터 변수와 타입이 없는 포인터 변수의 경우가 다르게 처리됩니다. 우선, 타입이 정해진 경우에는 **Inc()** 와 **Dec()** 함수를 사용할 수 있습니다. 그리고, 타입이 정해지지 않은 경우에는 [리스트 Pointer12]의 32: 라인처럼 다소 복잡한 연산이 필요합니다.

32: 라인의 경우에는 타입이 정해진 포인터 변수에도 동일하게 적용할 수 있습니다.

[리스트 Pointer12]

```
1 : program Pointer12;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   b : ^byte;
10 :   w : ^word;
11 :   i : ^integer;
12 :   p : pointer;
13 :
14 : begin
15 :   GetMem(p, 1024);
16 :   b:= p;
17 :   w:= p;
18 :   i:= p;
19 :
20 :   WriteLn('b 의 값 : ', Integer(b));
21 :   WriteLn('w 의 값 : ', Integer(w));
22 :   WriteLn('i 의 값 : ', Integer(i));
23 :   WriteLn('p 의 값 : ', Integer(p));
24 :   WriteLn;
25 :
26 :   Inc(b, 1); // Inc(b) 와 동일
27 :   Inc(w);
28 :   Inc(i);
29 :
30 : // 컴파일 할 수 없음
31 : // Inc(p);
32 :   p:= Ptr(Integer(p) + 1);
33 :
34 :   WriteLn('b 의 값 : ', Integer(b));
35 :   WriteLn('w 의 값 : ', Integer(w));
36 :   WriteLn('i 의 값 : ', Integer(i));
37 :   WriteLn('p 의 값 : ', Integer(p));
38 :   WriteLn;
```

```

39 :
40 :   ReadLn;
41 : end.

```

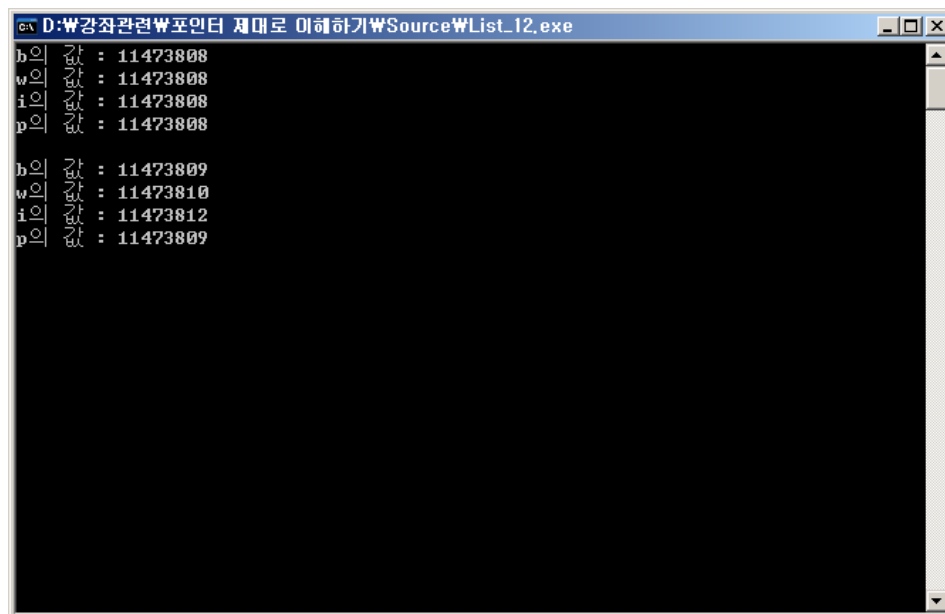
32: 라인에서는 우선 포인터 변수 **p** 에 있는 주소 값을 정수형으로 변환하고, 정수 **1** 을 더합니다. 그리고, 그 결과값을 포인터 형으로 변환하는 함수 **Ptr()** 을 이용해서 형태를 포인터로 변환하여 변수 **p** 에 다시 대입합니다.

[그림 63]에서 보듯이 26-32: 라인의 수행 결과는 각각 자신의 변수타입이 차지하는 메모리 공간의 크기 만큼 늘어났음을 알게 됩니다.

26: 라인의 **byte** 는 1 바이트 크기의 메모리 공간이 필요, 따라서 1 만큼의 주소 값이 변경되었습니다.

27: 라인의 **word** 는 2 바이트 크기의 메모리 공간이 필요, 따라서 2 만큼의 주소 값이 변경되었습니다.

28: 라인의 **integer** 는 4 바이트 크기의 메모리 공간이 필요, 따라서 4 만큼의 주소 값이 변경되었습니다.



[그림 63] 리스트 Pointer12 의 실행 결과

TColor 를 R+G+B 형태로 조작하는 예제

TColor 는 4 바이트로 구성된 데이터 타입 입니다.

그중에서도 앞의 3 바이트는 RGB 형태의 색상 정보를 저장하고 있습니다.

따라서, 이 3 바이트의 값을 혼합하면 어떠한 색상도 표현할 수 있습니다.

이것을 포인터 변수를 통해서 제어하는 예제 두 가지를 통해서 설명드리도록 하겠습니다.

첫 번째 예제

[리스트 Pointer13]

```
1 : program Pointer13;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils, Forms, Graphics;
7 :
8 : var
9 :   MainForm : TForm;
10 :
11 :   Color : TColor;
12 :   R, G, B : ^byte;
13 :
14 : begin
15 :   MainForm:= TForm.CreateNew(nil);
16 :   MainForm.Show;
17 :
18 :   R:= @Color;
19 :
20 :   G:= R;
21 :   Inc(G);
22 :
23 :   B:= G;
24 :   Inc(B);
25 :
26 :   R^:= $12;
27 :   G^:= $34;
28 :   B^:= $56;
29 :
30 :   MainForm.Color:= Color;
31 :   MainForm.Repaint;
32 :
```

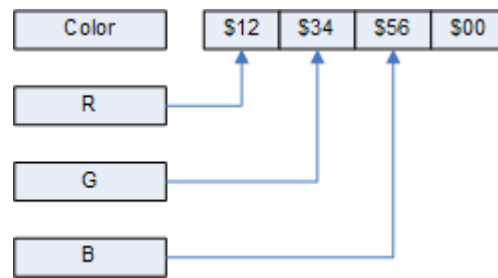
```
33 :   ReadLn;  
34 : end.
```

15-6: 라인에서 출현하는 **TForm** 과 그에 대한 설명은 생략하기로 하겠습니다. 우선 이 예제에서는 화면에 표시되는 색상 확인하기 위해서 임시적으로 사용한 것 입니다.

18: 라인에서는 변수 **Color** 의 메모리 주소 값이 포인터 변수 **R** 에 지정 됩니다.

20: 라인에서 **G** 는 **R** 과 같은 주소를 가리키게 되고, 이어서 21: 라인에서 **Inc()** 함수를 통해서 자신의 타입인 **byte** 크기만큼 이동합니다. 따라서, **Color** 가 차지하는 메모리 공간의 두 번째 바이트를 가리키게 됩니다.

23-24: 라인에서 **B** 도 같은 원리로, **Color** 가 차지하는 메모리 공간의 세 번째 바이트를 가리키게 됩니다.



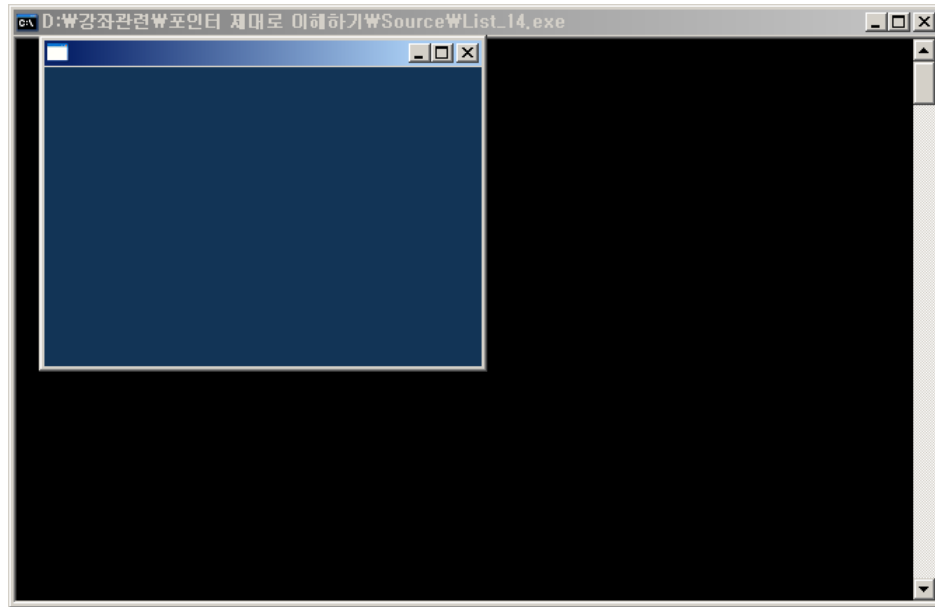
[그림 64] 포인터 변수 R, G, B 가 각각 가르키고 있는 변수 **Color** 의 내부

두 번째 예제

[리스트 Pointer14]

```
1 : program Pointer14;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils, Forms, Graphics;
7 :
8 : type
9 :   TRGB = record
10 :     R, G, B : byte;
11 :   end;
12 :
13 : var
14 :   MainForm : TForm;
15 :
16 :   Color : TColor;
17 :   RGB : ^TRGB;
18 :
19 : begin
20 :   MainForm:= TForm.CreateNew(nil);
21 :   MainForm.Show;
22 :
23 :   RGB:= @Color;
24 :   RGB^.R:= $12;
25 :   RGB^.G:= $34;
26 :   RGB^.B:= $56;
27 :
28 :   MainForm.Color:= Color;
29 :   MainForm.Repaint;
30 :
31 :   ReadLn;
32 : end.
```

[리스트 Pointer13], [리스트 Pointer14] 의 두 예제 모두, 같은 원리로 똑같은 결과 값을 [그림 2]와 같이 출력하게 됩니다.



[그림 65] TColor 를 R+G+B 형태로 조작하는 예제의 실행 결과

여러 개의 데이터를 순서대로 보관 하기

정수 데이터를 연속해서 받으면서,

현재 입력된 변수가 순서대로 정렬되면서 배열 또는 리스트에 저장되도록 하는 간단한 예제를 통해서,

배열을 사용할 때와 포인터 변수를 사용할 때의 차이점을 설명드리도록 하겠습니다.

정적 배열을 사용한 예제

[리스트 Pointer15]

```
1 : program Pointer15;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Data, Index : integer;
10 :   Count : integer = 0;
11 :   Datas : array [1..1024] of integer;
12 :
13 : function FindIndexOf(Value:integer):integer;
14 : var
15 :   Loop : Integer;
16 : begin
17 :   Result:= Count + 1;
18 :   for Loop := 1 to Count do
19 :     if Value < Datas[Loop] then begin
20 :       Result:= Loop;
21 :       Break;
22 :     end;
23 : end;
24 :
25 : procedure MakeVacancy(Value:integer);
26 : var
27 :   Loop : integer;
28 : begin
29 :   for Loop := Count downto Value do Datas[Loop+1]:= Datas[Loop];
30 : end;
31 :
32 : procedure DisplayDatas;
33 : var
34 :   Loop : integer;
35 : begin
```

```

36 :   for Loop := 1 to Count do
37 :       WriteLn(Loop, ' : ', Datas[Loop]);
38 :   end;
39 :
40 :   begin
41 :       Write('Data : '); ReadLn(Data);
42 :       while Data > 0 do begin
43 :           // 자신보다 큰 숫자가 있는 Index 를 찾는다.
44 :           Index:= FindIndexOf(Data);
45 :
46 :           // 해당 위치에 빈 공간을 마련한다.
47 :           MakeVacancy(Index);
48 :
49 :           Datas[Index]:= Data;
50 :           Count:= Count + 1;
51 :
52 :           Write('Data : '); ReadLn(Data);
53 :       end;
54 :
55 :       DisplayDatas;
56 :
57 :       ReadLn;
58 :   end.

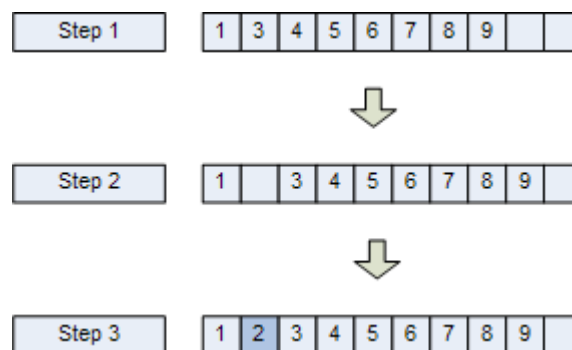
```

[리스트 Pointer15] 의 소스는 [그림 66]과 같이 세 단계를 통해서 작동합니다.

우선 **Step 1** 처럼 기존의 데이터가 저장된 경우를 생각해보겠습니다.

이 상황에서 2 라는 새로운 데이터가 입력되었다는 가정으로 **Step 2** 로 넘어가보겠습니다. 2 보다 큰 수를 가진 배열의 요소들이 전부 한 칸씩 이동하여야 합니다. 2 보다 큰 수를 가진 요소의 갯수 만큼의 이동이 필요하게 되는 것 입니다. 이 부분은 **MakeVacancy()** 함수가 처리하는 내용 입니다.

이제 마지막 단계에서는 비워진 공간에 2 라는 데이터를 저장하는 것뿐 입니다.



[그림 66] 배열을 이용한 소스의 처리 과정

포인터를 사용한 예제

[리스트 Pointer16]

```
1 : program Pointer16;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   PData = ^TData;
10 :   TData = record
11 :     Value : integer;
12 :     Prior : PData;
13 :     Next : PData;
14 :   end;
15 :
16 : var
17 :   Data, Index : PData;
18 :   Head : PData = nil;
19 :   Tail : PData = nil;
20 :
21 : function FindIndexOf(Value:integer):PData;
22 : var
23 :   Loop : Integer;
24 : begin
25 :   Result:= Head.Next;
26 :   while Result <> Tail do begin
27 :     if Value < Result.Value then Break;
28 :
29 :     Result:= Result.Next;
30 :   end;
31 : end;
32 :
33 : procedure DisplayDatas;
34 : var
35 :   Node : PData;
36 :   Count : integer;
37 : begin
38 :   Count:= 0;
39 :   Node:= Head.Next;
40 :   while Node <> Tail do begin
41 :     Count:= Count + 1;
42 :     WriteLn(Count, ' : ', Node.Value);
43 :
44 :     Node:= Node.Next;
45 :   end;
46 : end;
47 :
48 : begin
49 :   New(Head);
50 :   New(Tail);
51 :   Head^.Prior:= nil;
52 :   Head^.Next:= Tail;
```

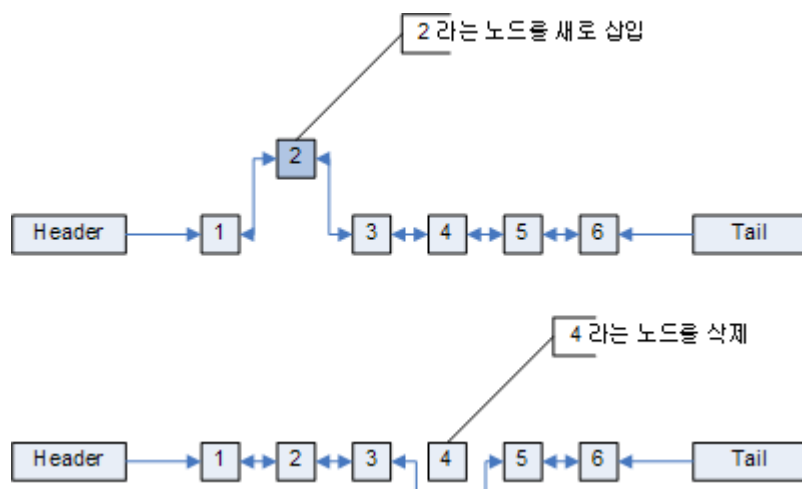
```

53 :   Tail^.Prior:= Head;
54 :   Tail^.Next:= nil;
55 :
56 :   New(Data);
57 :   Write('Data : '); ReadLn(Data^.Value);
58 :   while Data.Value > 0 do begin
59 :       // 자신보다 큰 숫자가 있는 Index 를 찾는다.
60 :       Index:= FindIndexOf(Data^.Value);
61 :
62 :       Data.Next:= Index;
63 :       Data.Prior:= Index.Prior;
64 :       Index.Prior.Next:= Data;
65 :       Index.Prior:= Data;
66 :
67 :       // 해당 위치에 빈 공간을 마련할 필요가 없다.
68 :
69 :       New(Data);
70 :       Write('Data : '); ReadLn(Data^.Value);
71 :   end;
72 :   Dispose(Data);
73 :
74 :   DisplayDatas;
75 :
76 :   ReadLn;
77 : end.

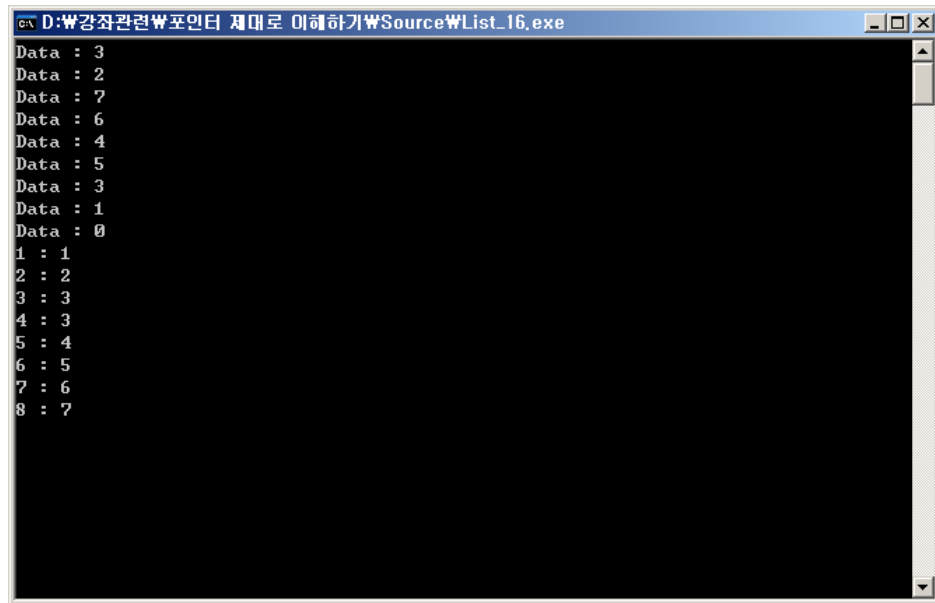
```

[리스트 Pointer16]은 [그림 68]와 같이 [리스트 Pointer15]과 같은 결과를 출력합니다. 하지만, 내부 프로세스는 포인터 변수를 활용하여 좀더 효율적인 면이 있습니다.

[그림 67]는 [리스트 Pointer16]에서 사용하는 Linked List 의 동작을 도식으로 표현한 것입니다. Linked List 의 경우에는 새로운 노드의 삽입 이나 삭제를 위해서 다른 노드의 데이터를 이동 시킬 필요가 없기 때문에, 그만큼 효율적인 작동이 가능해 집니다.



[그림 67] Linked List 의 작동 원리



[그림 68] 여러 개의 데이터를 순서대로 보관하기

함수에 대한 포인터 활용

[리스트 Pointer17]

```
1 : program Pointer17;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Choice : integer;
10 :   Func : pointer;
11 :
12 : procedure Hello;
13 : begin
14 :   WriteLn('안녕하세요?');
15 : end;
16 :
17 : procedure Bye;
18 : begin
19 :   WriteLn('안녕히 가세요!');
20 : end;
21 :
22 : begin
23 :   repeat
24 :     Write('선택하세요 (0:종료, 1:만남인사, 2:작별인사) : ');
25 :     ReadLn(Choice);
26 :     case Choice of
27 :       1 : Func:= @Hello;
28 :       2 : Func:= @Bye;
29 :     end;
30 :     asm
31 :       Call Func
32 :     end;
33 :     until Choice = 0;
34 :   end.
35 :
```

26-29: 라인을 통해서 선택된 값에 따라 포인터 변수 Func 에 Hello() 함수와 Bye() 함수 둘 중에 하나의 주소 값을 입력하게 됩니다.

31-33: 라인에서는 이전에 선택된 함수가 실행됩니다.

[리스트 Pointer18]의 경우에는 [리스트 Pointer17]과 동일한 동작을 하는 코드입니다.

다만, 포인터 변수 대신 함수에 대한 레퍼런스를 사용하고 있습니다.

[리스트 Pointer18]

```
1 : program Pointer18;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Choice : integer;
10 :   Func : procedure;
11 :
12 : procedure Hello;
13 : begin
14 :   WriteLn('안녕하세요?');
15 : end;
16 :
17 : procedure Bye;
18 : begin
19 :   WriteLn('안녕히 가세요!');
20 : end;
21 :
22 : begin
23 :   repeat
24 :     Write('선택하세요 (0:종료, 1:만남인사, 2:작별인사) : ');
ReadLn(Choice);
25 :
26 :     case Choice of
27 :       1 : Func:= Hello;
28 :       2 : Func:= Bye;
29 :     end;
30 :
31 :     Func;
32 :   until Choice = 0;
33 : end.
```


구조체에서 클래스로 확장하기

객체지향적 프로그래밍을 이해하기 위해서는 우선 클래스가 무엇인지 알아야 합니다.

클래스는 구조체의 확장된 문법이다.

객체지향적 프로그래밍을 제대로 이해하고 있는 사람이라면, 이러한 표현에 대해서는 반발을 할 것입니다. 하지만, 저는 "클래스는 구조체의 확장된 문법이다" 에서부터 객체지향적 프로그래밍을 시작하려고 합니다. 그것이 입문자가 클래스를 쉽게 이해할 수 있는 길이라고 믿기 때문입니다.

클래스도 구조체와 같이 같은 목적을 가진 요소들을 묶는 기술입니다.

```
type
  TPerson = record
    Name : string;
    Age : integer;
  end;

var
  Person1, Person2 : TPerson;
```

구조체를 설명하는 부분에서 위의 같은 소스를 보신 적이 있습니다. 이부분을 클래스로 변환하기 쉬운 형태로 변경하기 위해서 포인터를 적용해보도록 하겠습니다.

```
[리스트 OOP_01]

1 : program OOP_01;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
```

```

7 :
8 : type
9 :   TPerson = record
10 :     Name : string;
11 :     Age : integer;
12 :   end;
13 :
14 : var
15 :   Person1, Person2 : ^TPerson;
16 :
17 : begin
18 :   New(Person1);
19 :   Person1^.Name:= '류종택';
20 :   Person1^.Age:= 20 + 18;
21 :
22 :   New(Person2);
23 :   Person2^.Name:= '류종필';
24 :   Person2^.Age:= Person1^.Age - 2;
25 :
26 :   WriteLn(' 이름 : ', Person1^.Name);
27 :   WriteLn(' 나이 : ', Person1^.Age);
28 :   WriteLn;
29 :
30 :   WriteLn(' 이름 : ', Person2^.Name);
31 :   WriteLn(' 나이 : ', Person2^.Age);
32 :   WriteLn;
33 :
34 :   Dispose(Person1);
35 :   Dispose(Person2);
36 :
37 :   ReadLn;
38 : end.

```

6 장의 [리스트 Begin32]와 동일한 동작과 결과를 갖는 프로그램이 완성되었습니다.

다른 점은 아래와 같습니다.

- 15: 라인에서 **Person1, Person2** 두 변수가 포인터 변수로 선언되었습니다.
- 18: 라인과 22: 라인에서 **Person1, Person2** 두 변수가 메모리 공간을 할당 받고 있습니다.
- **Person1, Person2** 를 사용하기 전에 포인터 해제 연산자 **^**를 사용하고 있습니다.

포인터 변수는 주소만을 가지는 변수이기 때문에, 확보되어있는 메모리 공간을 가리키기 전까지는 사용할 수 없습니다. **New()** 함수를 통해서 **TPerson** 이 들어갈 수 있는

크기의 메모리 공간을 확보하고 주소를 Person1, Person2 두 변수에 저장하는 과정입니다.

이제 [리스트 OOP_01]을 아래와 같이 클래스로 변경하여 [리스트 OOP_02]처럼 작성해 보도록 하겠습니다.

[리스트 OOP_02]

```
1 : program OOP_02;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     Name : string;
11 :     Age : integer;
12 :   end;
13 :
14 : var
15 :   Person1, Person2 : TPerson;
16 :
17 : begin
18 :   Person1:= TPerson.Create;
19 :   Person1.Name:= '류종택';
20 :   Person1.Age:= 20 + 18;
21 :
22 :   Person2:= TPerson.Create;
23 :   Person2.Name:= '류종필';
24 :   Person2.Age:= Person1.Age - 2;
25 :
26 :   WriteLn(' 이름 : ', Person1.Name);
27 :   WriteLn(' 나이 : ', Person1.Age);
28 :   WriteLn;
29 :
30 :   WriteLn(' 이름 : ', Person2.Name);
31 :   WriteLn(' 나이 : ', Person2.Age);
32 :   WriteLn;
33 :
34 :   Person1.Free;
35 :   Person2.Free;
36 :
37 :   ReadLn;
38 : end.
```

[리스트 OOP_01]과 동일한 동작과 결과를 갖는 프로그램이 완성되었습니다.

다른 점은 아래와 같습니다.

- 9: 라인에서 TPerson 이 record 가 아닌 class 로 선언되었습니다.
- 15: 라인에서 Person1, Person2 두 포인터 변수가 아닌 일반 변수로 선언되었습니다. (실제로는 일반 변수가 아닌 레퍼런스 변수 입니다.)
- 18: 라인과 22: 라인에서 Person1, Person2 두 변수가 메모리 공간을 할당 받고 있습니다. 레퍼런스 변수도 포인터 변수와 같이 주소만을 가지는 변수이기 때문에, 확보되어있는 메모리 공간을 가리키기 전까지는 사용할 수 없습니다. 포인터와 달리 New() 함수가 아닌 클래스의 Create 생성자를 통해서 TPerson 이 들어갈 수 있는 크기의 메모리 공간을 확보하고 주소를 Person1, Person2 두 변수에 저장하고 있습니다.
- Person1, Person2 를 사용하기 전에 포인터 해제 연산자 ^ 를 사용하지 않습니다. 레퍼런스 변수는 포인터 연산자를 사용할 수 없습니다. ^ 연산자를 사용하지 않아도 주소가 가르키는 곳의 객체를 사용할 수 있습니다. ^ 연산자가 생략되었다고 생각하셔도 됩니다.

현재까지는 “클래스는 구조체의 포인터 타입이다” 라고 기억해두시면 되겠습니다. 실제로 클래스로 생성된 변수들은 모두 포인터처럼 행동합니다.

클래스 = 함수들 + 변수들

클래스도 구조체와 같이 같은 목적을 가진 요소들을 묶는 기술이라고 이미 말씀을 드렸습니다. 일반적으로 클래스는 변수들의 묶음이지만, 클래스는 변수뿐 아니라 함수도 함께 묶을 수 있다는 것이 차이점입니다.

텔파이가 발전해 오면서 구조체에도 함수를 함께 묶을 수 있도록 텔파이 2006 부터 문법이 변경되었습니다.

프로그래밍언어는 점점 무엇인가 묶어 가면서 발전해 왔습니다.

문장의 묶음 → 블록 : 구조적 프로그래밍 언어

함수와 변수의 묶음 → 클래스 : 객체지향적 프로그래밍 언어

[리스트 OOP_03]은 클래스 TPerson 에 GrowUp() Method 를 추가하는 예제 입니다. 클래스에 속한 함수는 Method 라고 부릅니다.

[리스트 OOP_03]

```
1 : program OOP_03;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     Name : string;
11 :     Age : integer;
12 :     procedure GrowUp;
13 :   end;
14 :
15 : var
```

```

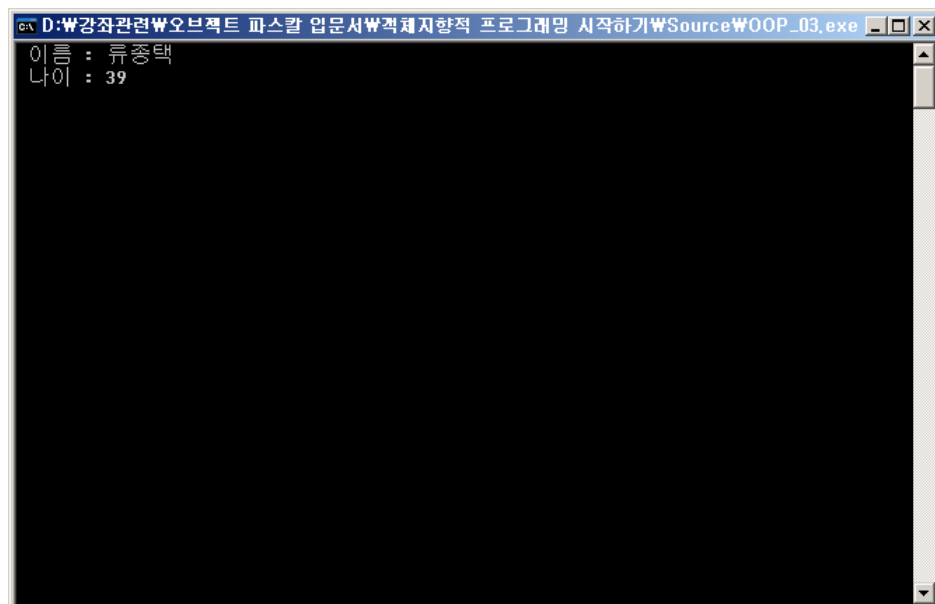
16 :   Person1 : TPerson;
17 :
18 : procedure TPerson.GrowUp;
19 : begin
20 :   Age:= Age + 1;
21 : end;
22 :
23 : begin
24 :   Person1:= TPerson.Create;
25 :   Person1.Name:= '류종택';
26 :   Person1.Age:= 20 + 18;
27 :   Person1.GrowUp;
28 :
29 :   WriteLn(' 이름 : ', Person1.Name);
30 :   WriteLn(' 나이 : ', Person1.Age);
31 :   WriteLn;
32 :
33 :   ReadLn;
34 : end.

```

12: 라인에서 **TPerson** 클래스에 **GrowUp()** 함수를 추가하고 있습니다. 이때, 구현코드를 클래스 선언 내에서는 할 수 없다는 것을 유의하시기 바랍니다.

GrowUp() 함수의 구현코드는 18-21: 라인에서 이루어지고 있습니다. 함수 이름이 **GrowUp** 이 아닌 **TPerson.GrowUp** 으로 작성되었습니다. 이는 **TPerson** 클래스 내에 속해있는 함수라는 뜻으로 사용된 것 입니다. **Age** 를 접근할 때, **TPerson.Age** 라고 하지 않는 것에 유의하시기 바랍니다.

클래스에 소속된 함수를 사용하는 것은 27: 라인에서처럼 앞에 클래스로 선언된 변수 **Person1** 과 마침표를 연결해서 사용하시면 됩니다. 이제부터, **GrowUp()**과 같은 함수의 경우에는 **Method** 라고 부르며, **Person1** 과 같은 변수들을 객체 레퍼런스 변수라고 부르겠습니다.



[그림 69] 리스트 OOP_03 의 실행 결과

[리스트 OOP_03]의 실행결과는 [그림 69]과 같이 38 (20+18, 스물 열여덟 살)이 아닌 39 로 출력되었습니다. 27: 라인의 실행으로 나이를 저장하는 변수 Age 값이 1 증가되었기 때문입니다.

객체란 무엇인가?

간혹 입문자들은 객체지향적 프로그래밍에서 사용되는 클래스와 객체(오브젝트) 두 용어를 혼동하는 경우가 있습니다.

```
1 : type
2 :   TPerson = class
3 :     Name : string;
4 :     Age : integer;
5 :   end;
6 :
7 : var
8 :   Person1 : TPerson;
9 :
10 : begin
11 :   Person1:= TPerson.Create;
12 :   Person1:= TPerson.Create;
```

위의 소스에서 클래스는 **TPerson** 입니다. 새로운 변수타입이라고 보면 됩니다. 클래스는, 마치 우리가 정수형 변수를 선언하기 위해 사용하는 변수타입 **integer** 처럼, 실제로 생성될 객체의 크기나 정보만을 가지고 있습니다.

그럼 객체는 어디에 있는 것 일까요? 어떤 분들은 8: 라인의 **Person1** 이 객체라고 생각하시는 경우가 있습니다. 하지만, 이미 설명 드렸듯이, **Person1** 은 객체를 가르키고 있는 포인터형 변수입니다. 정확하게는 오브젝트 레퍼런스 변수 입니다.

그러니까, 객체는 **Person1** 이 가르키고 있는 메모리 공간에 있다는 것 입니다. 결론적으로 말씀을 드리면, **객체는 클래스를 이용하여 확보한 메모리 공간 영역 자체를** 뜻 하는 것 입니다. 클래스 **TPerson** 은 11: 라인에서 **TPerson** 의 **Create** 생성자로 메모리 공간을 확보하고, 그곳에 객체를 생성하여 보관합니다.

이제 12: 라인이 실행되면 어떻게 될까요? 새로운 객체가 붕어빵처럼 생겨나고 **Person1** 이라는 레퍼런스 변수는 새로 생긴 객체가 있는 곳을 가르키게 됩니다. 하지만, 이미 만들어둔 객체가 있는 주소를 보관하는 곳은 이제 더이상 아무 곳에도 없습니다. 따라서, 11: 라인에서 생성한 객체는 이제 영영 사용할 수 없는 것 입니다. 객체는 이제 미아가 되어 버렸고, 이런 경우를 메모리 누수현상이라고 합니다.

포인터를 활용하여 메모리를 할당 받을 때, 같은 이유로 메모리 누수현상이 발생합니다.

이렇듯 객체를 생성하기 위해서 객체의 형태(속성과 기능)를 정의한 클래스라는 틀을 먼저 만들어야 합니다. 그리고, 해당 클래스를 메모리에 찍어내면, 메모리 공간에 객체가 생성되는 것입니다. 엄밀하게 말하면 클래스가 필요한 공간만큼 메모리 영역을 확보하고, 클래스에 정의된 변수나 함수의 묶음이 해당 메모리에 들어가게 되며, 이 메모리 영역을 객체라고 부릅니다.

붕어빵을 먹어 본적이 있으신가요?

클래스가 붕어빵 틀이라고 하면, 찍혀 나온 붕어빵들이 모두 객체입니다.

붕어빵의 크기와 생김새가 붕어빵 틀에 의해 좌우되는 것처럼,

객체의 크기와 생김새 그리고 기능은 클래스를 어떻게 선언하느냐에 달려있습니다.

생성자와 소멸자

객체를 생성하고 삭제하기 위해서 클래스는 생성자와 소멸자라는 특수한 함수를 가지고 있습니다. 생성자는 **Create**, 소멸자는 **Destroy** 로 이미 클래스 내부에 정의되어 있기 때문에 여러분들께서는 그저 사용하기만 하시면 됩니다.

```
1 : begin
2 :   Person1:= TPerson.Create;
3 :   Person1.Free;
```

객체를 생성하는 방법은 2: 라인과 같으며 이미 여러 차례 보아왔습니다.

객체를 삭제하는 방법은 주로 3: 라인과 같은 방식을 사용합니다. 생성된 객체에 대한 삭제이기 때문에 클래스 식별자가 아닌, 객체의 레퍼런스 변수에서 **Free** 를 실행해야 합니다.

Free Method 대신 소멸자 Destroy 를 직접 사용하셔도 상관없습니다. 하지만, 특별한 이유가 없는 한 Free 메소드를 사용하는 것이 더욱 안전한 방법입니다.

Free 메소드의 소스가 정의된 **TObject** 의 소스를 살펴보면 아래와 같습니다. **TObject** 는 델파이의 최상위 클래스입니다. 어떠한 클래스도 **TObject** 에서부터 상속받아서 생성됩니다. 즉, 델파이의 객체는 **Free** 라는 메소드를 포함하게 되며, 이는 **TObject** 에서 이미 정의된 대로 실행이 됩니다.

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

조금더 안전한 객체의 삭제 방법은 아래와 같습니다. **FreeAndNil()** 함수를 사용하면 객체를 삭제하고, 객체의 레퍼런스 변수에 **nil** 이라는 가상의 주소를 저장합니다. **nil** 은 "아무곳도 가르키고 있지 않음" 이라는 뜻입니다.

```
1 : begin
2 :   Person1:= TPerson.Create;
3 :   FreeAndNil(Person1);
```

상속

객체지향적 프로그래밍의 특징에서 두드러지는 것 중 하나가 상속이라는 기능입니다. 상속은 기존의 클래스를 확장하여 새로운 클래스는 정의하는 방법입니다.

기존의 클래스에서 상속을 받아서 새로운 클래스로 확장하는 문법은 아래와 같습니다.

```
클래스명 = class(기존의 클래스명)

    클래스 정의;

end;
```

위의 문법처럼 클래스를 상속 받아오면, 기존의 클래스의 모든 기능과 요소를 그대로 사용할 수 있습니다.

[리스트 OOP_04]

```
1 : program OOP_04;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     Name : string;
11 :     Age : integer;
12 :   end;
13 :
14 :   TMan = class(TPerson)
15 :   end;
16 :
17 :   TWoman = class(TPerson)
18 :   end;
19 :
20 : var
21 :   Ryu : TMan;
22 :   Lee : TWoman;
23 :
24 : begin
```

```

25 :   Ryu:= TMan.Create;
26 :   Ryu.Name:= '류종택';
27 :   Ryu.Age:= 20 + 18;
28 :
29 :   Lee:= TWoman.Create;
30 :   Lee.Name:= '이미정';
31 :   Lee.Age:= Ryu.Age - 1;
32 :
33 :   WriteLn(' 이름 : ', Ryu.Name);
34 :   WriteLn(' 나이 : ', Ryu.Age);
35 :   WriteLn;
36 :
37 :   WriteLn(' 이름 : ', Lee.Name);
38 :   WriteLn(' 나이 : ', Lee.Age);
39 :   WriteLn;
40 :
41 :   ReadLn;
42 : end.

```

14-15: 라인과 17-18: 라인을 통해서 TPerson 을 상속 받아서, 각각 TMan 과 TWoman 이라는 클래스 두 개를 정의하였습니다.

이후 라인들을 보면 TMan 과 TWoman 에는 어떠한 요소도 포함시킨 적이 없지만, TPerson 클래스의 요소들을 사용할 수 있는 것을 보실 수 있습니다. 결국 TPerson 과 TMan 그리고 TWoman 은 같은 클래스라고 보면 됩니다.

델파이에서 모든 클래스는 TObject 에서 자동으로 상속받게 됩니다.

TPerson = class 라고 선언하는 순간 이미 TPerson = class(TObject) 라는 의미를 가지게 됩니다.

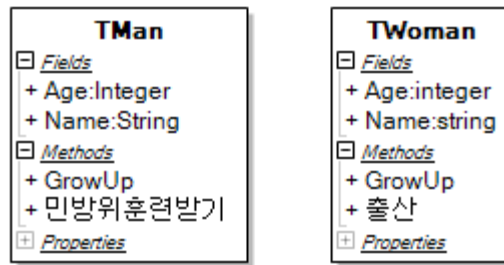
TPerson 을 기준으로 하면, TMan 과 TWoman 을 자식 클래스라고 합니다.

TMan 과 TWoman 을 기준으로 하면, TPerson 을 부모 클래스라고 합니다.

자식 클래스는 하위 클래스라고 불리기도 합니다.

부모 클래스는 수퍼 또는 상위 클래스라고 불리기도 합니다.

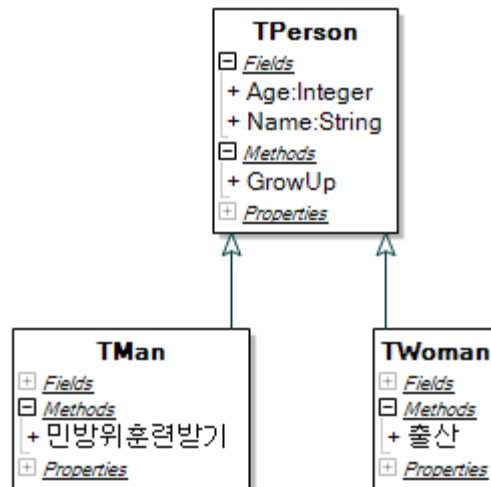
만약 남자를 뜻하는 TMan 클래스와 TWoman 이라는 두 클래스가 유사하거나 같은 요소를 포함하고 있지만, 서로 다른 요소도 포함하고 있다는 가정을 해보겠습니다.



[그림 70] TMan 과 TWoman 의 클래스 다이어그램

두 클래스가 가져야할 요소들을 한눈에 볼 수 있도록 [그림 70]와 같이 클래스 다이어그램을 사용하였습니다. **Fields** 라는 항목에 있는 것들은 변수들의 묶음입니다. **Methods** 라는 항목에 있는 것들은 말 그대로 **Method** 들의 묶음입니다.

두 클래스가 **Age**, **Name**, **GrowUp** 이라는 세 가지 동일한 요소를 가지고 있고, 각각 민방위훈련받기와 출산이라는 다른 요소를 포함하고 있습니다.



[그림 71] 상속을 통해서 중복을 제거한 상태의 클래스 다이어그램

중복은 프로그래밍의 칠거지악 중에서도 최악의 상태입니다. 결국, **Age**, **Name**, **GrowUp** 이라는 세 가지 요소처럼, 프로그래밍 코드 중에서 중복되는 부분으로 프로그램의 효율을 떨어뜨리고 위험한 버그들을 양산하는 원인이 되기도 합니다.

그리고, 클래스에는 이러한 중복을 제거하기 위해서 데이터베이스 설계 시에 정규화를 하듯이, 클래스도 정규화를 하는 방법이 있습니다. 그것을 바로 상속이라고 부릅니다.

데이터베이스 설계시에도 중복을 제거하기 위해 새로운 테이블을 생성하고 중복되는 것들은 몰아넣은 다음, 방금 중복된 요소를 제거한 테이블을 조인하여 사용하는 것처럼, 클래스의 경우에도 [그림 71]과 같이 중복되는 것은 다른 클래스에 몰아넣고 상속이라는 조인을 이용하게 됩니다.

이것을 코드로 표현하면 [리스트 OOP_05]와 같습니다.

[리스트 OOP_05]

```
1 : program OOP_05;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     Name : string;
11 :     Age : integer;
12 :     procedure GrowUp;
13 :   end;
14 :
15 :   TMan = class(TPerson)
16 :     procedure 민방위훈련받기;
17 :   end;
18 :
19 :   TWoman = class(TPerson)
20 :     procedure 출산;
21 :   end;
22 :
23 : var
24 :   Ryu : TMan;
25 :   Lee : TWoman;
26 :
27 : procedure TPerson.GrowUp;
28 : begin
29 :   Age:= Age + 1;
30 : end;
31 :
32 : procedure TMan.민방위훈련받기;
33 : begin
34 :   WriteLn('겨우 비디오나 틀어주려고 사람을 괴롭히냐!');
35 : end;
36 :
37 : procedure TWoman.출산;
38 : begin
39 :   WriteLn('둘째는 생각지도 마라!');
40 : end;
41 :
42 : begin
43 :   Ryu:= TMan.Create;
44 :   Ryu.Name:= '류종택';
```

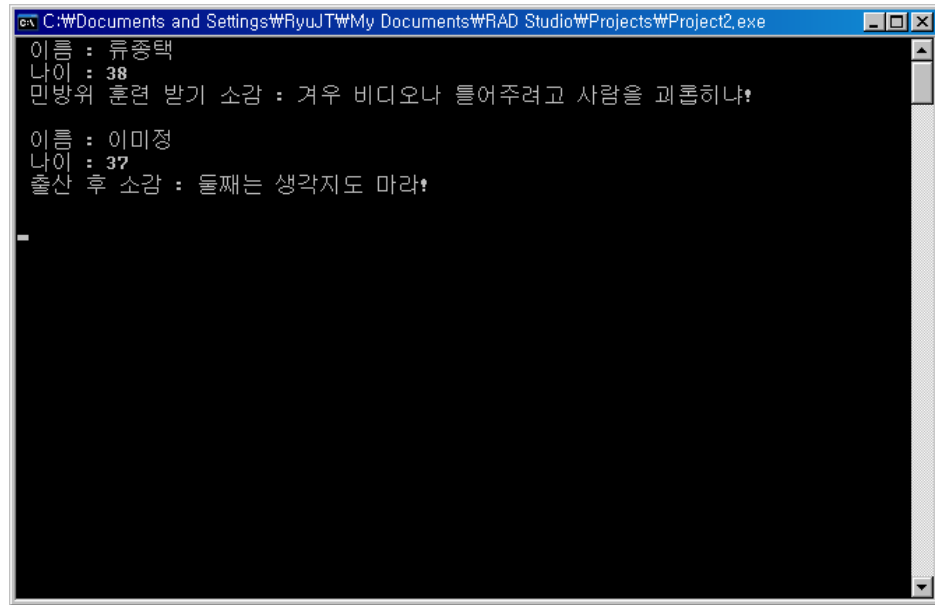
```

45 :   Ryu.Age:= 20 + 18;
46 :
47 :   Lee:= TWoman.Create;
48 :   Lee.Name:= '이미정';
49 :   Lee.Age:= Ryu.Age - 1;
50 :
51 :   WriteLn(' 이름 : ', Ryu.Name);
52 :   WriteLn(' 나이 : ', Ryu.Age);
53 :
54 :   Write(' 민방위 훈련 받기 소감 : ');
55 :   Ryu.민방위훈련받기;
56 :
57 :   WriteLn;
58 :
59 :   WriteLn(' 이름 : ', Lee.Name);
60 :   WriteLn(' 나이 : ', Lee.Age);
61 :
62 :   Write(' 출산 후 소감 : ');
63 :   Lee.출산;
64 :
65 :   WriteLn;
66 :
67 :   ReadLn;
68 : end.

```

중복되는 요소(멤버)들은 모두 **TPerson** 에서 상속을 받았기 때문에, **Ryu** 와 **Lee** 가 가르키고 있는 두 객체 모두 **Age**, **Name**, **GrowUp** 요소를 사용할 수 있으며, 56: 라인과 64: 라인처럼 각 클래스에 고유한 요소들을 개별적으로 사용할 수 있습니다.

Lee 가 가르키고 있는 객체는 민방위훈련받기와 같은 **Method** 를 사용할 수 없습니다. **TWoman** 클래스가 해당 **Method** 를 가지고 있지 않기 때문이며, 상속을 받아온 **TPerson** 에도 없기 때문입니다.



```
C:\Documents and Settings\RyuJT\My Documents\RAD_Studio\Projects\Project2.exe
이름 : 류종택
나이 : 38
민방위 훈련 받기 소감 : 겨우 비디오나 틀어주려고 사람을 괴롭히냐!

이름 : 이미정
나이 : 37
출산 후 소감 : 둘째는 생각지도 마라!
-
```

[그림 72] 리스트 OOP_05 의 실행 결과

상속은 중복을 제거하는 데에 효과적으로 사용할 수 있습니다.

객체지향적 프로그래밍의 가장 큰 장점은 코드(프로그램)의 분업화에 있습니다.

분업화를 위해서 중복의 제거는 필수 조건입니다.

멤버들의 가시성

클래스의 모든 요소(멤버)는 전자제품이나 기계들의 부품과 유사한 면이 있습니다. 그중에 하나가 바로 가시성입니다.

전화기를 예로 들어 보겠습니다. 전화기에는 수 많은 부품들이 들어 있습니다. 그 중에서 우리 눈에 보이는 요소는 수화기와 버튼이 있습니다. 그 이외의 부품들은 모두 내부에 감춰져 있기 때문에 우리가 직접 만지거나 조작하는 일이 거의 없습니다. 이렇게 클래스의 요소들도 외부에서 보이지 않게하고 접근할 수 없도록 정의할 수 있습니다.

외부 즉, 클래스 밖에서 사용할 수 없도록 감추는 것을 캡슐화라고 합니다.

캡슐화를 사용하는 이유는 아래와 같습니다.

- **변화의 요소를 감추기 위해서.** 앞으로 변할 수 있는 부분이 밖으로 나오면 객체의 동작을 이해하는데 어려울 수 있습니다. 이런 변화의 요소를 안에 감추고자할 때 사용합니다.
- **객체 사용자(개발자)의 오류를 방지하기 위해서.** 잘못 사용하면 심각한 에러가 발생할 것으로 예상되는 부분을 감춰서 에러의 방지하고자 사용할 수 있습니다.
- **간편하게 보이기 위해서.** 너무 많은 것이 노출되어 있으면, 한 번에 신경 써야 할 범위가 넓어져서 능률이 떨어집니다. 클래스의 구조와 동작을 이해하는데 필요없는 부분을 감춰서 이해하기 쉽도록 하기 위함입니다.

텔파이에서는 아래와 같은 네 가지의 가시성이 존재합니다.

```
1 : type
2 :   TMyClass = class
3 :     // 텔파이가 사용하는 영역
4 :   private
5 :     // private 영역
6 :   protected
7 :     // protected 영역
8 :   public
9 :     // public 영역
10 :   published
11 :     // published 영역
12 :   end;
```

- 델파이가 사용하는 영역
 - 이 부분은 **public** 혹은 **published** 취급을 받습니다.
 - 주로 델파이가 컴포넌트 관련 코딩을 자동으로 추가하고 관리하는 영역으로 사용합니다.
- **private** 영역
 - 이 영역에서 선언한 멤버들은 밖에서 볼 수 없습니다.
 - 같은 소스 파일 내에서는 예외적으로 볼 수 있으며, 때문에 사용이 가능합니다.
 - 클래스 자신 이외의 영역에서 절대 못보게 하기 위해서는 델파이 8 버전부터 새로 추가된 문법인 **strict** 를 붙여서 사용하면 됩니다. (**strict private**)
- **protected** 영역
 - **private** 와 거의 같은 의미를 가지고 있습니다.
 - 같은 소스 파일 내에서는 예외적으로 볼 수 있으며, 때문에 사용이 가능합니다.
 - 자신을 상속받은 자식 클래스에서는 볼 수 있으며, 때문에 사용이 가능합니다.
 - 같은 파일 내의 자식이 아닌 클래스의 참조가 불가능하도록 하기 위해서는 델파이 8 버전부터는 **strict** 를 붙여서 사용하면 됩니다. (**strict protected**)
- **public** 영역
 - 이 영역에서 선언한 멤버들은 어디서든 참조할 수 있습니다.
- **published** 영역
 - **public** 과 거의 같은 의미를 가지고 있습니다.
 - **Object Inspector** 에 나타나는 **Property** 를 선언하거나할 때는 반드시 **published** 영역에서 멤버를 선언해야 합니다.
 - 입문서에서는 자세한 설명을 하지 않겠습니다.

[리스트 OOP_06]

```
1 : program OOP_06;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
```

```

6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     strict private
11 :       procedure GrowUp;
12 :     public
13 :       Name : string;
14 :       Age : integer;
15 :     end;
16 :
17 : var
18 :   Person1 : TPerson;
19 :
20 : procedure TPerson.GrowUp;
21 : begin
22 :   Age:= Age + 1;
23 : end;
24 :
25 : begin
26 :   Person1:= TPerson.Create;
27 :   Person1.Name:= '류종택';
28 :   Person1.Age:= 20 + 18;
29 :   Person1.GrowUp;
30 : end.

```

11-12: 라인을 통해서 **GrowUp()** 함수가 **strict private** 영역에 선언이 되었습니다. 따라서, 클래스 밖에서 사용이 불가능하기 때문에 **30:** 라인을 컴파일되지 않고 에러가 발생합니다. 만약, **GrowUp()** 함수가 **private** 영역에 선언된다면, 같은 소스 파일 내에서는 사용되기 때문에 컴파일 됩니다.

하지만, 유닛을 사용해서 클래스 선언을 다른 파일로 옮기면 **30:** 라인은 컴파일이 되지 않습니다.

Method 의 유전자 변형 장치, Virtual 과 Override

부모 클래스로부터 받아온 Method 는 자식 클래스에서 변형하거나 확장 및 축소할 수 있습니다. 다만, 부모 클래스에서 해당 Method 를 자식 클래스에서 변형할 수 있도록 허가를 해주어야 합니다. 자식 클래스에서 Method 를 변형할 수 있도록 허가를 내주려면 Virtual 예약어를 사용하시면 됩니다.

자식 클래스는 해당 Method 를 변형하고자 할 때, Override 예약어를 사용하여 알려주어야 합니다.

[리스트 OOP_07]

```
1 : program OOP_07;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     Name : string;
11 :     Age : integer;
12 :     procedure GrowUp; virtual;
13 :   end;
14 :
15 :   TMan = class(TPerson)
16 :     procedure GrowUp; override;
17 :   end;
18 :
19 :   TWoman = class(TPerson)
20 :     procedure GrowUp; override;
21 :   end;
22 :
23 : var
24 :   Ryu : TMan;
25 :   Lee : TWoman;
26 :
27 : procedure TPerson.GrowUp;
28 : begin
29 :   Age:= Age + 1;
30 : end;
31 :
32 : procedure TMan.GrowUp;
33 : begin
34 :   inherited;
35 :
36 :   Age:= Age + 1;
37 : end;
```

```

38 :
39 : procedure TWoman.GrowUp;
40 : begin
41 :   Age:= Age - 1;
42 : end;
43 :
44 : begin
45 :   Ryu:= TMan.Create;
46 :   Ryu.Name:= '류중택';
47 :   Ryu.Age:= 38;
48 :   Ryu.GrowUp;
49 :
50 :   Lee:= TWoman.Create;
51 :   Lee.Name:= '이미정';
52 :   Lee.Age:= 37;
53 :   Lee.GrowUp;
54 :
55 :   WriteLn(' 이름 : ', Ryu.Name);
56 :   WriteLn(' 나이 : ', Ryu.Age);
57 :   WriteLn;
58 :
59 :   WriteLn(' 이름 : ', Lee.Name);
60 :   WriteLn(' 나이 : ', Lee.Age);
61 :   WriteLn;
62 :
63 :   ReadLn;
64 : end.

```

12: 라인에서는 **GrowUp Method** 를 자식 클래스에서 변경할 수 있도록 **virtual** 로 선언되어 있습니다.

이어서, 16: 라인과 20: 라인에서는 부모의 **Method** 를 상속 받아서 다시 정의하여 사용하겠다는 의미로, **override** 가 선언되어 있습니다.

34: 라인에서 보면 새로운 예약어인 **inherited** 를 보실 수 있습니다. 이것은 부모의 코드를 그대로 실행해라 라는 의미 입니다. 부모의 코드는 29: 라인에서 보듯이 **Age** 를 하나씩 추가하는 것입니다.

이어서 36: 라인에서 **Age** 를 하나 더 추가했기 때문에, **TMan** 클래스로 생성한 객체는 **GrowUp Method** 가 실행될 때마다 나이가 2 살씩 먹게 될 것 입니다.

32-37: 라인은 아래와 같이 작성하셔도 됩니다. 즉, **inherited GrowUp** 는 **inherited** 처럼 생략해서 사용할 수 있습니다.

```

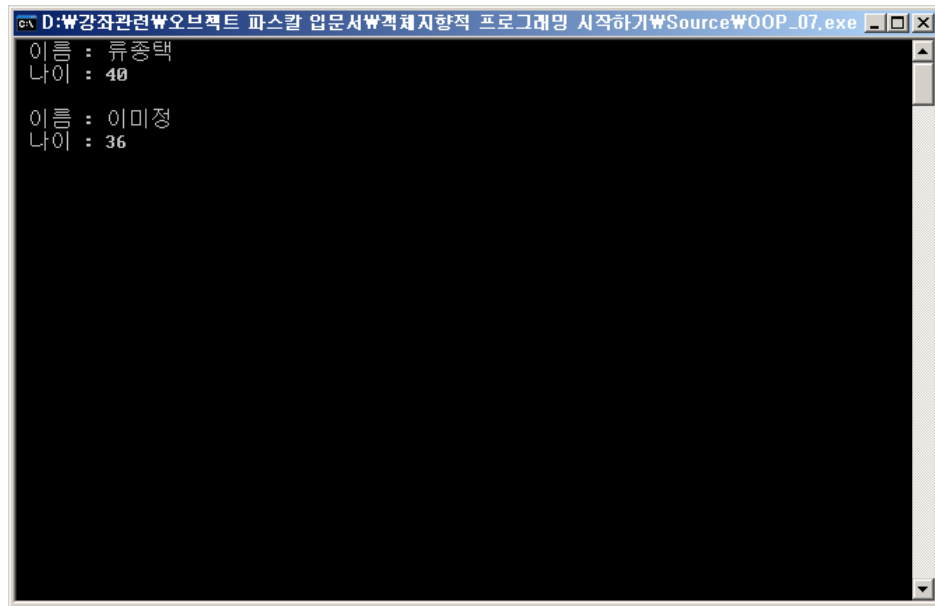
procedure TMan.GrowUp;
begin
  inherited GrowUp;

  Age:= Age + 1;
end;

```

39-42: 라인에서 구현하고 있는 TWoman.GrowUp Method 의 경우에는 inherited 를 보실 수 없습니다. 이 경우에는 부모 클래스에서 정의한 Method 의 모든 코드를 무시하고 새로운 Method 를 만드는 경우가 됩니다.

41: 라인에서 보듯이 TWoman.GrowUp Method 의 경우에는 실행될 때마다 1 살씩 나이를 꺼꾸로 먹게 됩니다. [그림 73]에서 보면 류종택의 나이는 애초의 38 에서 2 살 많은 상태로 출력되었으며, 이미정의 나이는 37 살에서 36 으로 줄어든 것을 확인하실 수 있습니다.



```
이름 : 류종택
나이 : 40

이름 : 이미정
나이 : 36
```

[그림 73] 리스트 OOP_07 의 실행 결과

클래스 연산자

[리스트 OOP_08]

```
1 : program OOP_08;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     Name : string;
11 :     Age : integer;
12 :   end;
13 :
14 :   TMan = class(TPerson)
15 :   end;
16 :
17 :   TWoman = class(TPerson)
18 :   end;
19 :
20 : var
21 :   Person : TPerson;
22 :   Ryu : TMan;
23 :   Lee : TWoman;
24 :
25 : begin
26 :   Ryu:= TMan.Create;
27 :   Lee:= TWoman.Create;
28 :
29 :   WriteLn('Ryu is TMan = ', Ryu is TMan);
30 :   WriteLn('Ryu is TPerson = ', Ryu is TPerson);
31 :
32 :   Person:= Lee as TPerson;
33 :   Person.Name:= 'Mrs. Lee';
34 :   WriteLn('Person 의 이름 : ', Lee.Name);
35 :
36 :   ReadLn;
37 : end.
```

클래스 연산자에는 **is** 와 **as** 두 가지가 존재 합니다.

29-30: 라인에서 **object is class** 는 객체가 뒤에 오는 클래스로 생성되거나 해당 클래스의 자식 클래스로 생성이 되었는 가를 검사하는 연산자입니다. 객체가 뒤에 오는 클래스나 자식 클래스로 생성된 경우에는 **true** 가, 아닐 경우에는 **false** 가 반환 됩니다.

32: 라인에서 **as** 는 해당 객체를 뒤에오는 클래스 타입으로 형변환을 하는 연산자 입니다. **32:** 라인의 경우에는 아래와 같이 일반적인 형변환 문법을 사용하여도 무방합니다. 일반적인 타입변환 문법은 객체나 클래스가 아닌 일반적인 변수에도 적용이 됩니다.

서로 호환되지 않는 타입으로 변환할 경우에는, 객체를 접근할 때, 에러가 발생한다는 것에 유의하시기 바랍니다.

일반적인 타입변환(typecasting)

타입명(변수)

사용의 예

```
Person:= TPerson(Lee);
```

as 를 사용할 경우에는, 일반적인 타입변환과 달리, 변환할 수 없는 타입을 사용하면 컴파일이 되지 않습니다.

클래스 함수와 클래스 변수

클래스에 속한 모든 멤버는 객체로 생성되기 전에는 사용할 수 없습니다.

예를 들어 아래와 같은 소스에서는 4: 라인에서 에러가 발생합니다.

```
1 : var
2 :   Ryu : TMan;
3 : begin
4 :   Ryu.Name:= '류종택';
```

하지만, 우리는 클래스를 객체화 하지 않고 사용하는 클래스 멤버가 필요할 경우가 있습니다. 그중에서 가장 자주 쓰이는 것은 바로 생성자입니다. 생성자는 객체를 생성해주는 함수인데, 만약 이것을 객체를 생성하기 이전에 사용할 수 없다면, 객체는 영원히 생성할 수 없을 것 입니다. 이런 필요성이 있을 때 사용할 수 있는 것이 바로 클래스 함수와 클래스 변수입니다.

[리스트 OOP_09]

```
1 : program OOP_09;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TPerson = class
10 :     class var HowManyTimeDidYouAskMyName : integer;
11 :     class function MyName:string;
12 :   end;
13 :
14 : class function TPerson.MyName: string;
15 : begin
16 :   HowManyTimeDidYouAskMyName:= HowManyTimeDidYouAskMyName + 1;
17 :   Result:= 'My Name is TPerson';
18 : end;
19 :
20 : begin
21 :   TPerson.HowManyTimeDidYouAskMyName:= 0;
22 :
```

```

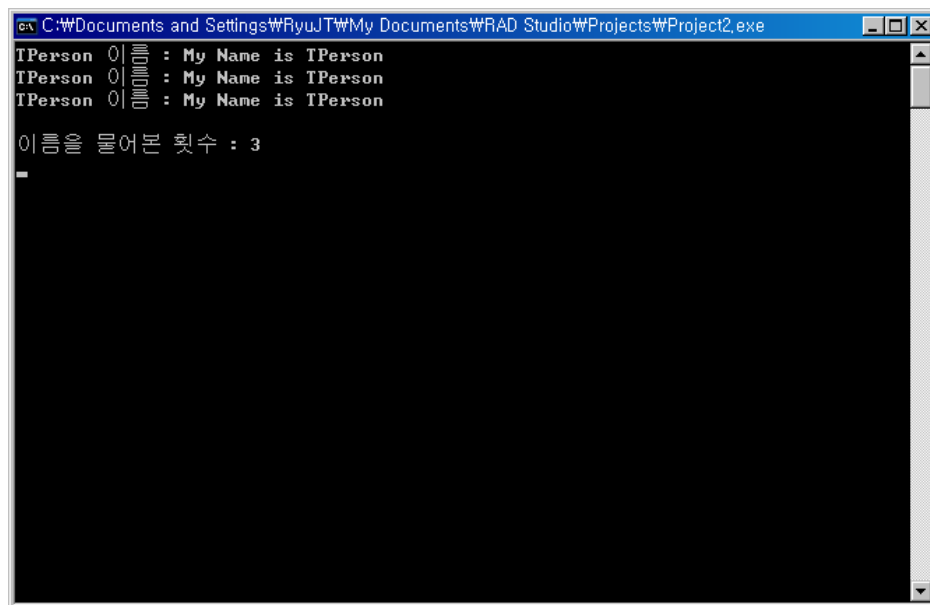
23 :   WriteLn('TPerson 이름 : ', TPerson.MyName);
24 :   WriteLn('TPerson 이름 : ', TPerson.MyName);
25 :   WriteLn('TPerson 이름 : ', TPerson.MyName);
26 :   WriteLn;
27 :
28 :   WriteLn('이름을 물어본 횟수 : ',
TPerson.HowManyTimeDidYouAskMyName);
29 :
30 :   ReadLn;
31 : end.

```

10: 라인에서는 어떻게 클래스 변수를 선언하는 가를 보여주고 있습니다.

11: 라인에서는 어떻게 클래스 함수를 선언하는 가를 보여주고 있습니다.

21-28: 라인에서 보듯이 클래스 함수와 클래스 변수는 객체를 생성시키기 전에 사용할 수 있다는 것을 알 수 있습니다.



[그림 74] 리스트 OOP_09 의 실행 결과

바이오리듬 만들기

이번 장에서는 바이오리듬 프로그램을 만들면서 객체지향적 프로그래밍을 연습하고자 합니다.

객체지향적 프로그램의 진정한 힘과 의미를 보여드리기 위해서는 조금 큰 규모의 프로그램이 필요합니다. 소규모의 프로그램에서는 객체지향적 프로그래밍 설계가 오히려 부담이 되어서 효과가 없을 수 있습니다.

하지만, 입문자 과정에서 규모가 큰 프로그램을 작성하는 것은 무리가 있기 때문에, 아쉽지만 바이오리듬 프로그램을 통해서 최대한 객체지향적 프로그래밍의 흐름을 보여드릴 수 있도록 노력하겠습니다.

객체지향적 프로그래밍에 어느 정도 익숙해지는 것은, 프로그래밍 입문과정을 거친 개발자의 경우에도 대개 1년 에서 2년 정도 걸린다고 합니다. 그것은 객체지향적 프로그래밍이 어느 정도의 프로그래밍 경험과 능력을 필요로 하기 때문 입니다.

너무 조급해 할 필요 없이 하나씩 배워나가다 보면, 자신도 모르는 사이에 객체지향적 프로그래밍에 익숙해져 있을 겁니다.

객체지향적 프로그래밍은 상당히 중요한 요소이니만큼, 끈기를 가지고 공부하시기 바랍니다.

분석 및 설계 과정

개발자는 프로그래밍을 시작하기 전에 분석과 설계 과정을 꼼꼼히 살펴보아야 합니다. 분석이란 우리가 작성해야 할 프로그램이 어떤 목적으로 프로그램이 사용되어야 하는가 등에 대한 조사과정입니다.

분석과 설계는 뚜렷하게 구분되는 작업이라고 보기에는 어렵습니다.

분석은 사용자의 요구사항과 현재 상태를 조사하는 과정이고,

설계는 개발자가 어떻게 분석된 내용을 만들어갈 것인지를 연구하는 과정입니다.

분석을 하다보면 자연히 설계자료가 생성되기 마련입니다.

가장 쉬운 분석 방법은 기능분석입니다. 우리가 만들어야 할 프로그램이 갖춰야 할 기능을 나열하는 것입니다. 이것은 우리가 예제를 통해서 계속해오던 것입니다.

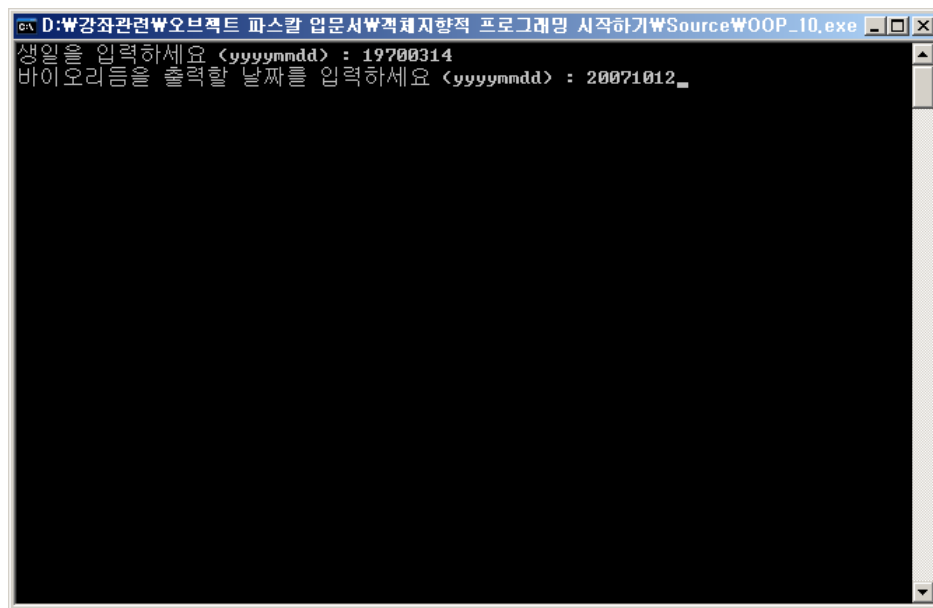
바이오리듬 프로그램이 갖춰야 할 기능

- 생일을 입력 받는다.
- 출력할 기준일을 입력 받는다.
- 어떤 날짜의 바이오리듬을 보고 싶은 지, 사용자가 입력하도록 합니다.
- 화면에 바이오리듬을 출력한다.

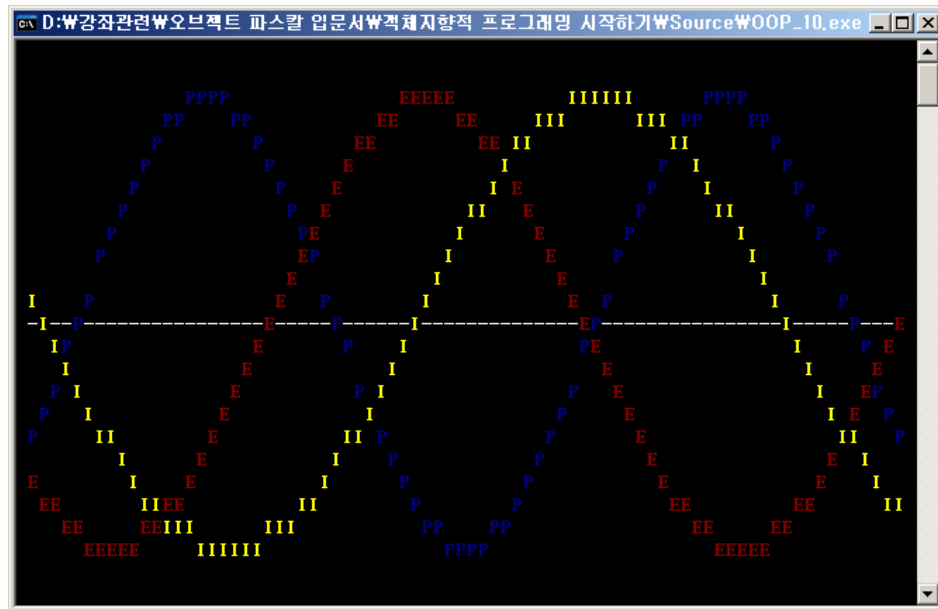
입력 및 출력의 기능은 사용자와 밀접한 관계가 있는 부분이기 때문에, 사용자가 어떠한 스타일을 원하는 지, 미리 세심한 분석이 있어야 합니다. 이러한 사용자들의 요구사항을 분석하는 것을 UI(User Interface) 분석이라고 합니다.

기능분석은 작성해야 할 프로그램의 크기 및 범위 그리고 개발 일정을 산출하는 근거자료로 사용되게 됩니다.

입문자과정에서 화려한 UI 를 작성하는 것은 무리가 따르기 때문에 [그림 75]과 [그림 76]와 같이 단순한 UI 를 사용하도록 하였습니다. 분석과정에서는 아직 프로그램이 나오지 않은 상태이기 때문에, 종이에 스케치하거나 그림을 그릴 수 있는 프로그램을 사용해서 대략적인 모양만 설명할 수 있는 자료를 만드시면 됩니다.



[그림 75] 생일과 출력할 날짜 입력 받는 장면



[그림 76] 바이오리듬을 화면에 출력한 장면

지금까지의 예제에서는 기능분석 이후에는 플로우차트를 이용하여 구현방법을 설계해왔습니다.

하지만, 객체지향적 프로그래밍을 할 때에는 기능분석 이후에 구조설계가 필요합니다.

구조설계의 목적은 효율적인 소스코드를 생산해내는 것입니다.

객체지향적이든 아니든, 어떻게 짜든지 서로 비슷하게 돌아가는 프로그램을 만들 수는 있습니다.

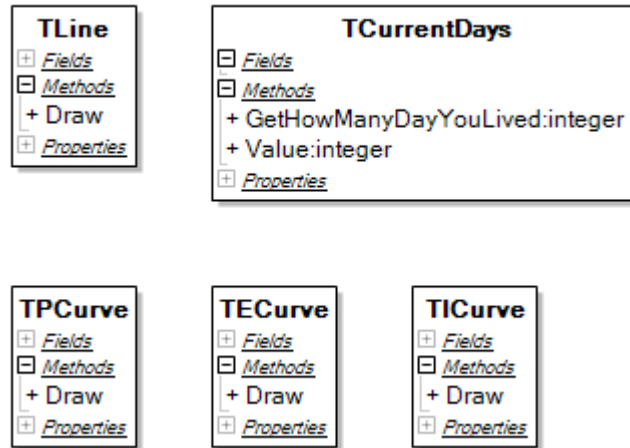
하지만, 효율적이지 못한 소스코드의 경우에는 프로그램이 점점 커져갈 때, 그것을 감당하지 못한다는 점에 있습니다.

초보 개발자들이 가장 많이 실수하는 부분이기도 합니다.

사용자의 요구사항 또는 자신의 아이디어를 통해서 즉흥적으로 프로그램을 작성하기 시작하는 경우가 많습니다. 이런 경우에도, 대부분, 처음에는 쉽게 결과물이 나오고 제대로 프로그램이 완성되는 듯 합니다. 하지만, 대수롭게 생각되지 않는 기능들이나 요구사항을 추가하면 프로그래밍은 영원히 끝나지 않는 터널 속에 빠지게 됩니다. 그러한 점에서 구조설계는 매우 중요한 위치에 있습니다.

하지만, 지금 우리가 만들고 있는 바이오리듬 정도의 크기의 프로그램에서는 구조설계가 큰 위력을 발휘하지 못 합니다. 이미 말씀드린대로 입문자용 예제이기 때문에 너무 복잡하지 않을 수준까지만 만들 것이기 때문입니다.

[그림 77]은 바이오리듬 프로그램을 작성하기 위한 구조설계 자료입니다.



[그림 77] 바이오리듬을 만들기 위한 클래스 다이어그램

객체지향적 프로그래밍의 가장 큰 장점은 분업화에 있다고 이미 말씀드렸습니다. 우리가 기능분석을 통해서 분석한 기능들을 각 클래스 마다 고유의 임무로 부여해주기 위해서 [그림 3]과 같이 분업화를 시도하였습니다.

- TLine 클래스
 - Draw Method : 화면에 X 좌표 축을 그리는 일만 담당합니다. 제일 한가한 클래스입니다.
- TCurrentDays 클래스
 - GetHowManyDayYouLived Method : 사용자가 입력한 생일에서부터 오늘까지 살아온 일수를 계산합니다. 생각보다 복잡합니다.
 - Value Method : 한 번 계산된 값은 객체 내부에 저장하도록 되어 있습니다. 두 번째부터는 다시 계산하지 않고 이미 계산된 데이터를 사용합니다. 이 부분은 Method 를 이용하지 않고, property 를 이용할 수도 있습니다. 입문자에게는 어려운 일이라고 생각하여 Method 로 사용했습니다. 자바의 경우에는 Method 이름앞에 get 을 붙여서 사용하기도 합니다.
- TPCurve 클래스

- Draw Method : 화면에 신체적 상태를 그린다.
- TECurve 클래스
 - Draw Method : 화면에 감성적 상태를 그린다.
- TICurve 클래스
 - Draw Method : 화면에 지성적 상태를 그린다.

이제 마지막으로 동적분석을 진행해보도록 하겠습니다. 지금까지의 예제에서는 플로우차트를 사용했습니다. 하지만, 객체들의 동작은 플로우차트로 설명하기가 어렵습니다. 따라서, UML 의 시퀀스 다이어그램 등을 사용하게 됩니다.

동적분석은 프로그램을 완성하기 이전에 검증을 하는 도구입니다.

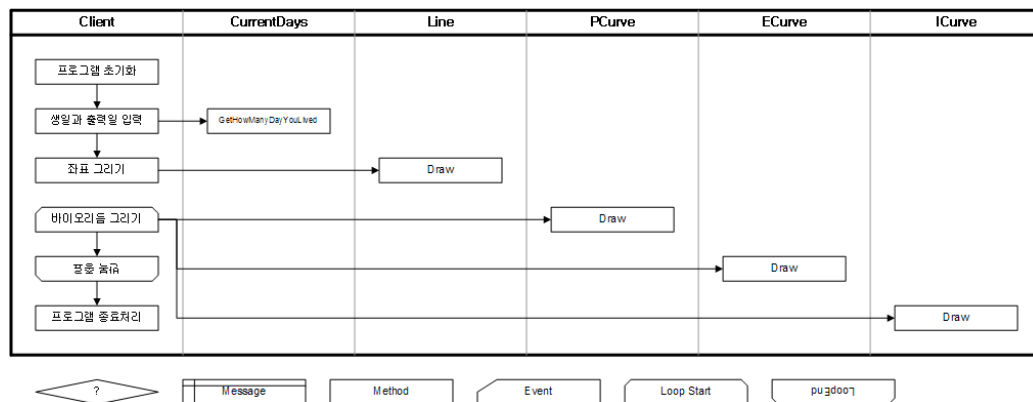
자동차를 출시 이전에 운행 테스트를 하는 것과 마찬가지입니다.

최대한 초기에 적은 노력으로 잘못된 부분을 파악하고 제거하는데 목적이 있습니다.

동적분석을 통해서 필요없거나 중요하지 않는 기능을 찾아낼 수도 있고,

분석과정에서 찾아내지 못한 요구사항을 발견할 수도 있습니다.

여기서는 저자가 1995 년 객체지향적 프로그래밍 설계를 위해서 만든 Job Flow 라는 형식을 사용하도록 하겠습니다. 플로우차트를 객체 마다 분리해서 그릴 수 있도록 확장한 것 입니다.



[그림 79] Job Flow - 바이오리듬의 동적분석



Client 는 우리가 설계한 객체들을 사용하는 대상(객체)를 뜻 합니다. [그림 79]에서는 프로젝트 소스의 코드들을 **Client** 라고 의인화하여 설명하고 있습니다.

[그림 79]의 동적분석의 경우에도 우리가 작성할 프로그램의 규모가 작다보니 크게 효용가치는 없습니다. 우리가 원하는 바이오리듬이 동작하는 방식을 이해하는데, 크게 무리가 없어 보이기 때문에 동적분석을 마지막으로 분석과 설계 과정을 마무리하겠습니다.

기능분석과 구조분석 그리고 동적분석은 순차적으로 이뤄져야하는 것은 아닙니다.

설계자(개발자)의 머릿속에서 동시에 일어나기도 하고, 다른 분석과정에 의해서 서로 수정 및 보완되기도 합니다.

구현과정

델파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, **Console** 어플리케이션을 선택하시기 바랍니다. 이어서,  **Save All** 버튼을 클릭하고 프로젝트 파일 이름을 **OOP_10** 으로 저장하시기 바랍니다. 이후, 아래와 같이 코드를 작성하시기 바랍니다.

[리스트 OOP_10]

```
1 : program OOP_10;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : procedure CreateObjects;
9 : begin
10 : end;
11 :
12 : procedure DestroyObjects;
13 : begin
14 : end;
15 :
16 : procedure do_GetBirthday;
17 : begin
18 : end;
19 :
20 : procedure do_DrawBiorhythm;
21 : begin
22 : end;
23 :
24 : procedure Run;
25 : begin
26 :   do_GetBirthDay;
27 :   do_DrawBiorhythm;
28 : end;
29 :
30 : begin
31 :   CreateObjects;
32 :   try
33 :     Run;
34 :   finally
35 :     DestroyObjects;
36 :   end;
37 :
38 :   ReadLn;
39 : end.
```

우선 동적분석을 토대로 전체적인 흐름에 대해서만 코드로 작성해 보았습니다.


31: 라인에서 초기화를 하면서 필요한 객체를 생성합니다.

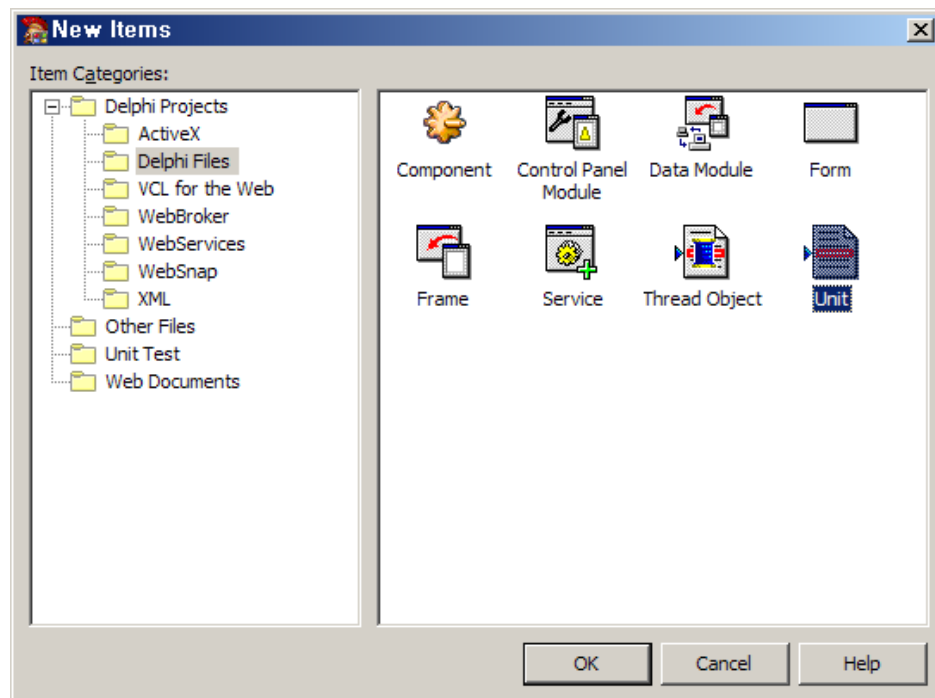
33: 라인에서는 본격적인 바이오리듬 프로그램의 기능을 동작합니다.

35: 라인에서는 생성된 모든 객체를 해제합니다. **try finally** 문장이 적용되어 있기 때문에, 객체가 해제되지 않는 일은 발생하지 않습니다. (예제의 경우에는, 어차피 프로그램이 종료될 것이기 때문에 해제를 하지 않는다고 해도 큰 상관은 없습니다)


24-28: 라인에서는 **Run** 함수가 실제로 동작하는 과정을 구현하였습니다. 동적분석을 토대로 생일을 입력받고 바이오리듬을 출력하도록 되어 있습니다. 출력할 날짜는 생일을 입력받을 때, 한꺼번에 받도록 하겠습니다.

이제 구조설계를 통해서 설계한 클래스를 구현해보도록 하겠습니다. 클래스에 대한 구현은 유닛을 추가하여 만들도록 하겠습니다.

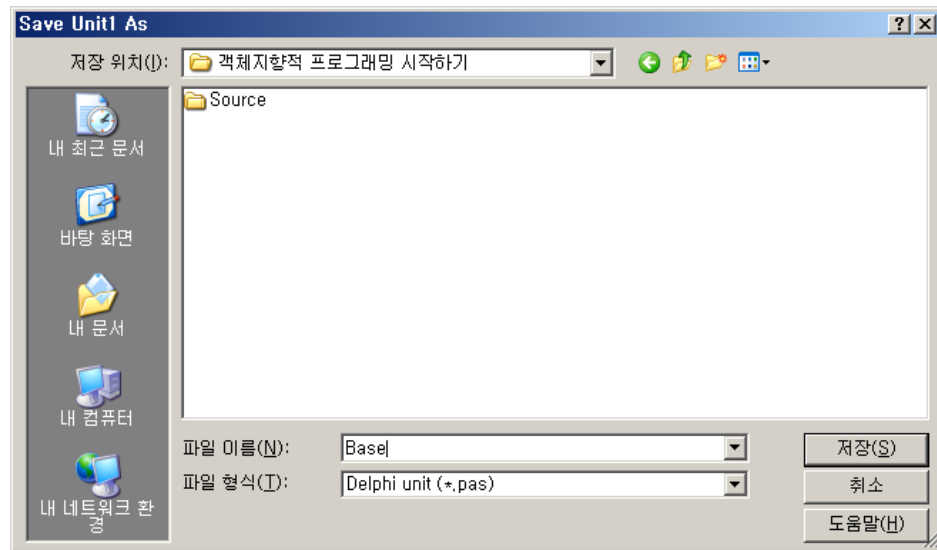
델파이의 왼쪽 상단의  **New Items** 버튼을 클릭하고, [그림 80]과 같이 유닛을 선택하신 후 **Ok** 버튼을 클릭하면 됩니다.



[그림 80] 새로운 유닛 추가

이제  **Save All** 버튼을 클릭하여 소스 전부를 저장합니다. 이미 프로젝트 파일을 저장하신 분께서는 유닛 파일만을 저장하시면 됩니다. 유닛의 이름은 **Base** 라고 하겠습니다. 프로젝트 소스인 [리스트 OOP_10]의 6: 라인의 **uses** 절이 자동으로 다음과 같이 변경되어 있을 것 입니다.

```
uses
  SysUtils,
  Base in 'Base.pas';
```



[그림 81] 유닛의 이름 입력 및 저장

이후 [리스트 OOP_11]과 같이 작성하도록 하겠습니다. 구조분석을 통해서 설계한 **Method** 들의 껍데기만 정리한 상태입니다. **9-34: 라인의 클래스 인터페이스 영역을 정의하신 후 각각의 클래스 영역에서 Ctrl+Shift+C 를 클릭하시면, implementation 영역은 델파이가 자동으로 생성하여 줍니다.**

[리스트 OOP_11]

```
1 : unit Base;
2 :
3 : interface
4 :
5 : uses
6 :   Classes, SysUtils, Crt;
7 :
8 : type
9 :   TLine = class
10 :   public
11 :     procedure Draw;
```

```

12 :   end;
13 :
14 :   TPCurve = class
15 :   public
16 :       procedure Draw(Index,CurrentDays:integer);
17 :   end;
18 :
19 :   TECurve = class
20 :   public
21 :       procedure Draw(Index,CurrentDays:integer);
22 :   end;
23 :
24 :   TICurve = class
25 :   public
26 :       procedure Draw(Index,CurrentDays:integer);
27 :   end;
28 :
29 :   TCurrentDays = class
30 :   private
31 :   public
32 :       function Value:integer;
33 :       function
GetHowManyDayYouLived(Birthday,Today:integer):integer;
34 :   end;
35 :
36 : implementation
37 :
38 : procedure TLine.Draw;
39 : begin
40 : end;
41 :
42 : procedure TPCurve.Draw(Index, CurrentDays: integer);
43 : begin
44 : end;
45 :
46 : procedure TECurve.Draw(Index, CurrentDays: integer);
47 : begin
48 : end;
49 :
50 : procedure TICurve.Draw(Index, CurrentDays: integer);
51 : begin
52 : end;
53 :
54 : function TCurrentDays.GetHowManyDayYouLived(Birthday, Today:
integer): integer;
55 : begin
56 : end;
57 :
58 : function TCurrentDays.Value: integer;
59 : begin
60 : end;
61 :
62 : end.

```

6: 라인에는 **Crt** 유닛을 추가했습니다. **Classes** 와 **SysUtils** 유닛은 자주 사용하는 것이기에 일단 넣어 두었습니다. 델파이의 경우에는 개발자가 **uses** 에 다양한 유닛을 넣더라도, 사용하지 않으면 컴파일에서 자동으로 삭제되어 실행파일에 포함되지 않습니다.

바이오리듬 프로그램에서 가장 복잡한 것이 살아온 일 수를 구하는 것입니다. 바이오리듬이라는 것이 태어난 날짜로부터 몇 일인가를 계산해서 해당 위치에서의 사인함수의 값이 현재의 상태로 보고 있기 때문입니다. 우선은 다른것부터 작성하면서 테스트할 수 있도록 살아온 날짜를 계산하지 않고 무조건 1000 일이라고 보겠습니다.

54-60: 라인에 구현된 두 **Method** 를 아래와 같이 수정합니다.

```
function TCurrentDays.GetHowManyDayYouLived(Birthday, Today: integer):
integer;
begin
    Result:= 1000;
end;

function TCurrentDays.Value: integer;
begin
    Result:= 1000;
end;
```

다음은 비교적 쉬운 좌표 그리기를 해보겠습니다. 38-40: 라인에 구현된 **TLine.Draw Method** 를 아래와 같이 수정합니다.

```
1 : procedure TLine.Draw;
2 : var
3 :   i : integer;
4 : begin
5 :   ClrScr;
6 :
7 :   for i := 1 to 78 do begin
8 :     GotoXY(i, 12);
9 :     TextColor(White);
10 :    Write('-');
11 :   end;
12 : end;
```

반복문을 사용하면서 변수 **i** 의 값이 1 부터 78 까지 변할 때, (i, 12) 좌표로 이동하여 '-' 문자를 찍도록 되어 있습니다.

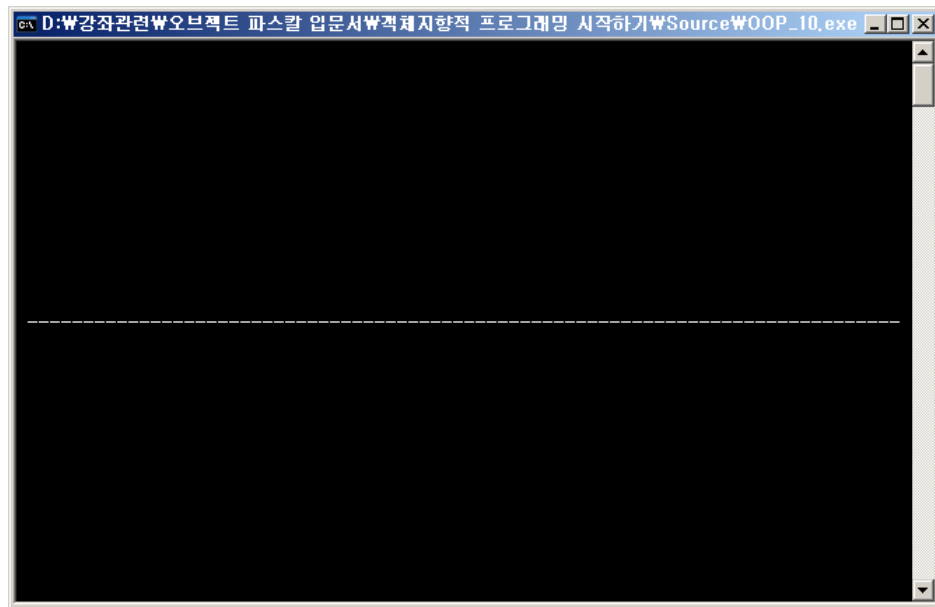
9: 라인의 **TextColor** 는 좌표와 바이오리듬 곡선들을 서로 구별하기 위해서 화면의 출력되는 문자열의 색상을 지정기 위해서 작성하였습니다.

테스트를 위해서 [리스트 OOP_10]의 30-39: 라인을 임시로 아래와 같이 수정한 후 F9 를 눌러서 프로그램을 실행해 보시기 바랍니다. [그림 82]과 같은 결과가 나오면 성공한 것입니다.

```
var
  Line : TLine;

begin
  Line:= TLine.Create;
  Line.Draw;

  ReadLn;
end.
```



[그림 82] TLine.Draw Method 의 테스트 장면

프로그램이 제대로 작성되고 있는 지 계속 테스트를 하는 것은 상당히 중요한 일입니다. 테스트를 통해서 버그를 초기에 잡을 수록 얻어지는 이익은 크기 때문입니다.

이제 신체리듬에 해당하는 TPCurve.Draw Method 를 구현해보도록 하겠습니다. 42-44: 라인을 아래의 소스와 같이 변경하시기 바랍니다.

```

1 : procedure TPCurve.Draw(Index, CurrentDays: integer);
2 : var
3 :   iY : integer;
4 : begin
5 :   iY:= Round( Sin(CurrentDays*Pi*2/23)*10 ) + 12;
6 :
7 :   GotoXY(Index, iY);
8 :   TextColor(Blue);
9 :   Write('P');
10 : end;

```

1: 라인의 **index** 는 **x** 축의 좌표의 값을 밖에서부터 얻어오기 위한 인자입니다. **CurrentDays** 는 생일에서부터 현재 신체리듬을 찍어야할 날짜 사이의 일 수 입니다.

5: 라인에서는 신체리듬의 현재의 **y** 축 값을 구하고 있습니다. 신체리듬은 23 일마다 반복이 되기 때문에 $Pi*2/23$ 을 통해서 23 일마다 원주율의 두 배가 되도록 하였습니다.

테스트를 위해서 [리스트 OOP_10]의 30-39: 라인을 임시로 아래의 소스와 같이 수정한 후 **F9** 를 눌러서 프로그램을 실행해 보시기 바랍니다. [그림 83]와 같은 결과가 나오면 성공한 것입니다.

```

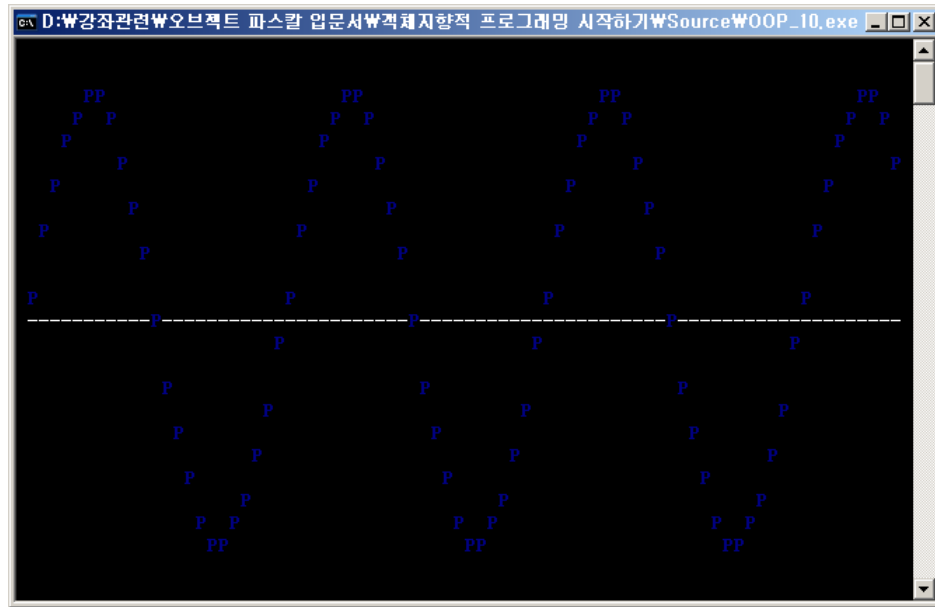
var
  Line : TLine;
  PCurve : TPCurve;
  i : integer;

begin
  Line:= TLine.Create;
  Line.Draw;

  PCurve:= TPCurve.Create;
  for i := 1 to 78 do PCurve.Draw(i, i + 1000);

  ReadLn;
end.

```



[그림 83] TPCurve.Draw Method 의 테스트 장면

감성리듬에 해당하는 **TECurve.Draw Method** 를 구현해보도록 하겠습니다. 46-48: 라인을 아래의 소스와 같이 변경하시기 바랍니다. 신체리듬과 달리 5: 라인에서 반복 주기가 $\text{Pi} \times 2 / 28$ 을 통해서 28일로 지정되어 있습니다.

```

1 : procedure TPCurve.Draw(Index, CurrentDays: integer);
2 : var
3 :   iY : integer;
4 : begin
5 :   iY:= Round( Sin(CurrentDays*Pi*2/28)*10 ) + 12;
6 :
7 :   GotoXY(Index, iY);
8 :   TextColor(Red);
9 :   Write('E');
10 : end;

```

테스트를 위해서 [리스트 OOP_10]의 30-39: 라인을 임시로 아래의 소스와 같이 수정한 후 F9 를 눌러서 프로그램을 실행해 보시기 바랍니다. [그림 84]과 같은 결과가 나오면 성공한 것입니다.

```

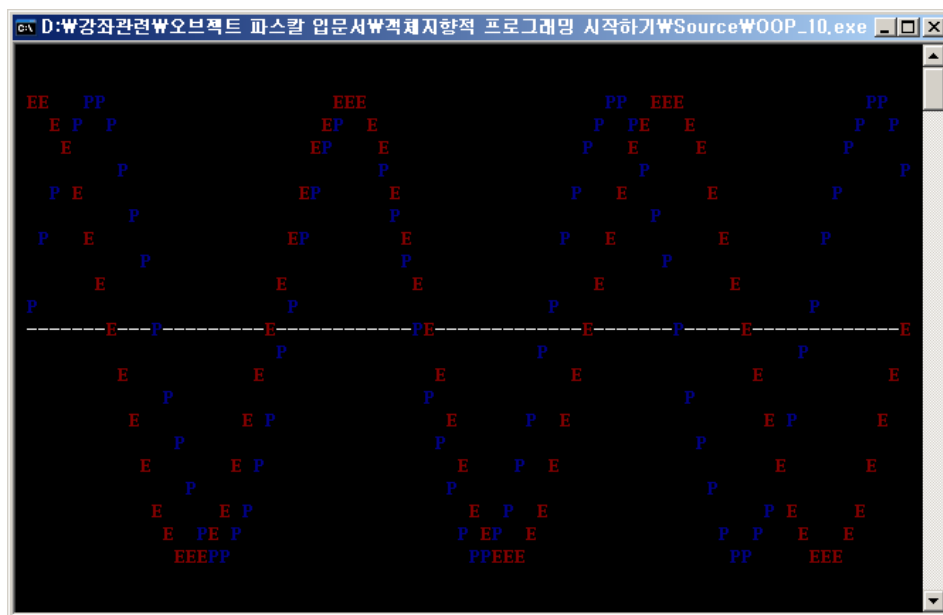
var
  Line : TLine;
  PCurve : TPCurve;
  ECurve : TECurve;
  i : integer;

begin
  Line:= TLine.Create;
  Line.Draw;

  PCurve:= TPCurve.Create;
  ECurve:= TECurve.Create;
  for i := 1 to 78 do begin
    PCurve.Draw(i, i + 1000);
    ECurve.Draw(i, i + 1000);
  end;

  ReadLn;
end.

```



[그림 84] TECurve.Draw Method 의 테스트 장면

마지막으로 지성리듬을 구하기 위해서 TICurve.Draw 를 구현해 보겠습니다. 33 일 주기에 맞추기 위해서 5: 라인에서 $\text{Pi} \cdot 2/33$ 으로 구현되어 있습니다.

```

1 : procedure TICurve.Draw(Index, CurrentDays: integer);
2 : var
3 :   iY : integer;
4 : begin
5 :   iY:= Round( Sin(CurrentDays*Pi*2/33)*10 ) + 12;
6 :
7 :   GotoXY(Index, iY);
8 :   TextColor(Yellow);
9 :   Write('I');
10 : end;

```

테스트를 위해서 [리스트 OOP_10]의 30-39: 라인을 임시로 아래의 소스와 같이 수정한 후 F9 를 눌러서 프로그램을 실행해 보시기 바랍니다. [그림 85]과 같은 결과가 나오면 성공한 것입니다.

```

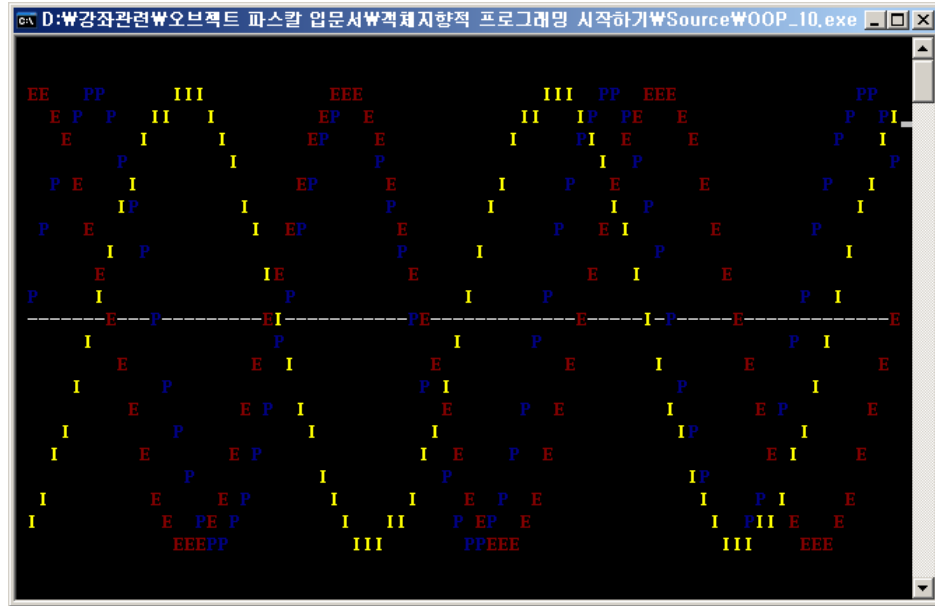
var
  Line : TLine;
  PCurve : TPCurve;
  ECurve : TECurve;
  ICurve : TICurve;
  i : integer;

begin
  Line:= TLine.Create;
  Line.Draw;

  PCurve:= TPCurve.Create;
  ECurve:= TECurve.Create;
  ICurve:= TICurve.Create;
  for i := 1 to 78 do begin
    PCurve.Draw(i, i + 1000);
    ECurve.Draw(i, i + 1000);
    ICurve.Draw(i, i + 1000);
  end;

  ReadLn;
end.

```



[그림 85] TlCurve.Draw Method 의 테스트 장면

이제 생일과 출력할 날짜를 입력받기 위해서 테스트 코드를 모두 원상태로 복구하시고, [리스트 OOP_10]의 내용을 다음과 같이 수정하여 주시기 바랍니다.

우선 [리스트 OOP_10]의 5: 라인의 **uses** 밑에 추가 및 수정정될 내용들 입니다.

```

1 : uses
2 :   SysUtils,
3 :   Base in 'Base.pas';
4 :
5 : var
6 :   Line : TLine;
7 :   PCurve : TPCurve;
8 :   ECurve : TECurve;
9 :   ICurve : TICurve;
10 :   CurrentDays : TCurrentDays;
11 :
12 : procedure CreateObjects;
13 : begin
14 :   Line := TLine.Create;
15 :   PCurve := TPCurve.Create;
16 :   ECurve := TECurve.Create;
17 :   ICurve := TICurve.Create;
18 :   CurrentDays := TCurrentDays.Create;
19 : end;
20 :
21 : procedure DestroyObjects;

```

```

22 : begin
23 :   FreeAndNil (Line);
24 :   FreeAndNil (PCurve);
25 :   FreeAndNil (ECurve);
26 :   FreeAndNil (ICurve);
27 :   FreeAndNil (CurrentDays);
28 : end;

```

5-10: 라인을 통해서 사용될 객체의 레퍼런스 변수들을 선언하였습니다.

14-18: 라인은 객체들을 생성하는 과정입니다.

23-27: 라인은 객체들을 삭제하고 객체의 레퍼런스 변수에 nil 값을 대입하는 FreeAndNil() 함수를 사용하는 과정입니다.

[리스트 OOP_10]의 16-18: 라인을 아래와 같이 수정합니다.

```

1 : procedure do_GetBirthday;
2 : var
3 :   Birthday, Today : integer;
4 : begin
5 :   Write('생일을 입력하세요 (yyyymmdd) : ');
6 :   ReadLn (Birthday);
7 :
8 :   Write('바이오리듬을 출력할 날짜를 입력하세요 (yyyymmdd) : ');
9 :   ReadLn (Today);
10 :
11 :   CurrentDays.GetHowManyDayYouLived(Birthday, Today);
12 : end;

```

11: 라인에서 생일과 출력할 날짜 사이의 날짜 수를 계산합니다. 계산된 결과를 전역변수에 담아서 바이오리듬을 구하는 do_DrawBiorhythm 함수에서 사용할 수 있도록 할 수도 있습니다. 하지만, 전역변수는 사용하지 않는 것이 좋기 때문에 사용하지 않기로 하였습니다. 더욱이 CurrentDays 는 날짜를 계산하는 임무를 가지는 객체이기 때문에, 계산된 날짜 수를 보관하는 임무를 맡기는 편이 보다 효율적입니다.

예제 수준에서는 do_GetBirthday 와 do_DrawBiorhythm 함수에 나뉜 코드를 하나로 합쳐서 사용하는 것이 더 효과적일 수 있습니다.

[리스트 OOP_10]의 20-22: 라인을 아래와 같이 수정합니다.

```

1 : procedure do_DrawBiorhythm;
2 : var
3 :   i, Days : integer;

```

```

4 : begin
5 :   Line.Draw;
6 :
7 :   Days:= CurrentDays.Value;
8 :   for i := 1 to 78 do begin
9 :     PCurve.Draw(i, Days + i - 1);
10 :    ECurve.Draw(i, Days + i - 1);
11 :    ICurve.Draw(i, Days + i - 1);
12 :   end;
13 : end;

```

9-11: 라인에서 각각의 바이오리듬 그리기 객체의 **Draw() Method** 를 실행할 때, 현재 그려야할 날짜는 **Days + i - 1** 로 구현되었습니다. 출력 기준으로 부터 1 일씩 증가하면서 그래프를 출력하기 위함입니다. 반복문을 아래와 같이 수정할 수도 있습니다.

```

for i := 0 to 77 do begin
  PCurve.Draw(i, Days + i);
  ECurve.Draw(i, Days + i);
  ICurve.Draw(i, Days + i);
end;

```

마지막으로 날짜 계산에 대한 구현에 대해서 설명드리겠습니다. 우선 **TCurrentDays** 클래스에 다음과 같은 것들을 추가 합니다. 이부분에 대한 설계와 분석 과정은 생략하도록 하겠습니다.

```

1 :   TDateRec = record
2 :     Year, Month, Day : word;
3 :   end;
4 :
5 :   TCurrentDays = class
6 :   private
7 :     FValue : integer;
8 :     function DayOfYear(Year:Integer):Integer;
9 :     function DayOfMonth(Year,Month:integer):Integer;
10 :    function IntToDate(Value:integer):TDateRec;
11 :   public
12 :     function Value:integer;
13 :     function
GetHowManyDayYouLived(Birthday,Today:integer):integer;
14 :   end;

```

1-3: 라인에서는 날짜 계산에 필요한 세 정수형 변수를 하나로 묶어서 사용하기 위해서 만들어진 **TDateRec** 구조체를 정의하고 있습니다.

7: 라인에서 **FValue** 는 **GetHowManyDayYouLived()** Method 를 통해서 계산된 날짜를, 계산없이 다음에 다시 사용할 수 있도록 보관하기 위한 변수입니다. **private** 영역에 정의한 것은 해당 날짜를

임의로 변경하면 에러가 발생할 수 있기 때문입니다. **FValue** 는 **GetHowManyDayYouLived()** **Method** 에 의해서만 값이 변경되는 변수입니다. **FValue** 가 **private** 영역에 정의되어 있기 때문에 외부에서 해당 변수의 값을 읽기 위하여 12: 라인에서 **Value** 라는 **Method** 를 사용하고 있습니다.

Value Method 는 [리스트 OOP_11]의 58-60: 라인을 아래와 같이 변경하시면 됩니다.

```
function TCurrentDays.Value: integer;
begin
    Result:= FValue;
end;
```

8-10: 라인은 날짜 수를 계산하기 위해 필요한 함수들입니다. 날짜 수를 계산하는 과정은 전체적인 흐름과는 상관없기 때문에, 캡슐화를 사용하여 **private** 영역에 감춰버렸습니다.

아래는 **TCurrentDays** 클래스의 **Method** 들에 대한 설명입니다.

- **DayOfYear**
 - 해당 년도에 총 몇 일이 존재하는가?
 - 윤년에 대한 계산을 포함
 - 4 로 나뉘떨어지는 해이면서 100 으로 나뉘떨어지면 안된다.
 - 4 로 나뉘떨어지면서, 100 으로 나뉘떨어지더라도, 400 으로 나뉘떨어지면 윤년이다.
- **DayOfMonth**
 - 지정된 년도의 지정된 달에 총 몇 일이 존재하는가?
 - 윤달에 대한 계산을 포함
- **IntToDate**
 - **yyyymmdd** 방식으로 입력된 정수형을 **TDateRec** 구조체 타입으로 변형시키는 함수

위의 **private** 영역의 **Method** 와 **GetHowManyDayYouLived()** **Method** 는 [리스트 OOP_11]의 58-60: 라인을 아래와 같이 변경하시면 됩니다.

```
1 : function TCurrentDays.DayOfMonth(Year,Month:integer):Integer;
2 : begin
3 :     Result:= 0;
```

```

4 :   case Month of
5 :       1, 3, 5, 7, 8,10, 12 : Result:= 31;
6 :       4, 6, 9, 11           : Result:= 30;
7 :       2                     : if DayOfYear(Year) = 365 then
Result:= 28
8 :       else Result:= 29;
9 :       end;
10 : end;
11 :
12 : function TCurrentDays.DayOfYear(Year:Integer):Integer;
13 : begin
14 :     if ((Year Mod 4) = 0) then begin
15 :         if ((Year Mod 100) = 0) then begin
16 :             if ((Year Mod 400) = 0) then DayOfYear:=366
17 :             else DayOfYear:=365;
18 :         end else
19 :             DayOfYear:=366;
20 :         end else
21 :             DayOfYear:=365;
22 :     end;
23 :
24 : function TCurrentDays.IntToDate(Value:integer):TDateRec;
25 : begin
26 :     Result.Year:= Value div 10000;
27 :     Result.Month:= (Value - Result.Year*10000) div 100;
28 :     Result.Day:= Value - Result.Year*10000 - Result.Month*100;
29 : end;
30 :
31 : function TCurrentDays.GetHowManyDayYouLived(Birthday, Today:
integer): integer;
32 : var
33 :     Loop : Integer;
34 :     StartDate, EndDate : TDateRec;
35 : begin
36 :     StartDate:= IntToDate(Birthday);
37 :     EndDate:=   IntToDate(Today);
38 :
39 :     Result:= 0;
40 :
41 :     for Loop:= StartDate.Year to EndDate.Year-1 do
42 :         Result:= Result + DayOfYear(Loop);
43 :
44 :     for Loop:= 1 to StartDate.Month-1 do
45 :         Result:= Result - DayOfMonth(StartDate.Year, Loop);
46 :
47 :     Result:= Result - StartDate.Day;
48 :
49 :     for Loop:= 1 to EndDate.Month-1 do
50 :         Result:= Result + DayOfMonth(EndDate.Year, Loop);
51 :
52 :     Result:= Result + EndDate.Day;
53 :     FValue:= Result;
54 : end;

```

날짜를 계산하는 원리는 아래와 같습니다.

(년도의 차이)*365 + (윤달에 의해 추가된 날짜) - (시작일의 1 월 1 일에서
시작일까지의 날짜수) + (출력일자의 1 월 1 일에서 출력일까지의 날짜수)

41-52: 라인은 위의 원리를 코드로 표현한 것들입니다.

53: 라인에서는 나중에 계산된 값을 계산없이 사용하기 위하여 FValue 에 저장해두고 있습니다.

이번 예제의 경우에는 날짜를 계산하는 부분을 제외하고는 입문자에게도 어려움이 없다고 생각합니다.
바이오리듬 자체나 날짜계산에 연연하지 마시고, 객체지향적으로 프로그래밍 하는 감각을 익히는데
집중하여 주시기 바랍니다. 또한, 전체적인 흐름이 익숙해질 때까지 반복하시기를 권장 합니다.

끝으로 완성된 코드 전체는 다음과 같습니다.

[리스트 OOP_10]

```
1 : program OOP_10;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   Base in 'Base.pas';
8 :
9 : var
10 :   Line : TLine;
11 :   PCurve : TPCurve;
12 :   ECurve : TECurve;
13 :   ICurve : TICurve;
14 :   CurrentDays : TCurrentDays;
15 :
16 : procedure CreateObjects;
17 : begin
18 :   Line := TLine.Create;
19 :   PCurve := TPCurve.Create;
20 :   ECurve := TECurve.Create;
21 :   ICurve := TICurve.Create;
22 :   CurrentDays := TCurrentDays.Create;
23 : end;
24 :
25 : procedure DestroyObjects;
26 : begin
27 :   FreeAndNil(Line);
28 :   FreeAndNil(PCurve);
29 :   FreeAndNil(ECurve);
30 :   FreeAndNil(ICurve);
31 :   FreeAndNil(CurrentDays);
```

```

32 : end;
33 :
34 : procedure do_GetBirthday;
35 : var
36 :   Birthday, Today : integer;
37 : begin
38 :   Write('생일을 입력하세요 (yyyymmdd) : ');
39 :   ReadLn(Birthday);
40 :
41 :   Write('바이오리듬을 출력할 날짜를 입력하세요 (yyyymmdd) : ');
42 :   ReadLn(Today);
43 :
44 :   CurrentDays.GetHowManyDayYouLived(Birthday, Today);
45 : end;
46 :
47 : procedure do_DrawBiorhythm;
48 : var
49 :   i, Days : integer;
50 : begin
51 :   Line.Draw;
52 :
53 :   Days:= CurrentDays.Value;
54 :   for i := 1 to 78 do begin
55 :     PCurve.Draw(i, Days + i - 1);
56 :     ECurve.Draw(i, Days + i - 1);
57 :     ICurve.Draw(i, Days + i - 1);
58 :   end;
59 : end;
60 :
61 : procedure Run;
62 : begin
63 :   do_GetBirthDay;
64 :   do_DrawBiorhythm;
65 : end;
66 :
67 : begin
68 :   CreateObjects;
69 :   try
70 :     Run;
71 :   finally
72 :     DestroyObjects;
73 :   end;
74 :
75 :   ReadLn;
76 : end.

```

[리스트 OOP_11]

```

1 : unit Base;
2 :
3 : interface

```

```

4 :
5 : uses
6 :   Classes, SysUtils, Crt;
7 :
8 : type
9 :   TLine = class
10 :   public
11 :     procedure Draw;
12 :   end;
13 :
14 :   TPCurve = class
15 :   public
16 :     procedure Draw(Index,CurrentDays:integer);
17 :   end;
18 :
19 :   TECurve = class
20 :   public
21 :     procedure Draw(Index,CurrentDays:integer);
22 :   end;
23 :
24 :   TICurve = class
25 :   public
26 :     procedure Draw(Index,CurrentDays:integer);
27 :   end;
28 :
29 :   TDateRec = record
30 :     Year, Month, Day : word;
31 :   end;
32 :
33 :   TCurrentDays = class
34 :   private
35 :     FValue : integer;
36 :     function DayOfYear(Year:Integer):Integer;
37 :     function DayOfMonth(Year,Month:integer):Integer;
38 :     function IntToDate(Value:integer):TDateRec;
39 :   public
40 :     function Value:integer;
41 :     function
GetHowManyDayYouLived(Birthday,Today:integer):integer;
42 :   end;
43 :
44 : implementation
45 :
46 : procedure TLine.Draw;
47 : var
48 :   i : integer;
49 : begin
50 :   ClrScr;
51 :
52 :   for i := 1 to 78 do begin
53 :     GotoXY(i, 12);
54 :     TextColor(White);
55 :     Write('-');
56 :   end;
57 : end;
58 :

```

```

59 : procedure TPCurve.Draw(Index, CurrentDays: integer);
60 : var
61 :   iY : integer;
62 : begin
63 :   iY:= Round( Sin(CurrentDays*Pi*2/23)*10 ) + 12;
64 :
65 :   GotoXY(Index, iY);
66 :   TextColor(Blue);
67 :   Write('P');
68 : end;
69 :
70 : procedure TECurve.Draw(Index, CurrentDays: integer);
71 : var
72 :   iY : integer;
73 : begin
74 :   iY:= Round( Sin(CurrentDays*Pi*2/28)*10 ) + 12;
75 :
76 :   GotoXY(Index, iY);
77 :   TextColor(Red);
78 :   Write('E');
79 : end;
80 :
81 : procedure TICurve.Draw(Index, CurrentDays: integer);
82 : var
83 :   iY : integer;
84 : begin
85 :   iY:= Round( Sin(CurrentDays*Pi*2/33)*10 ) + 12;
86 :
87 :   GotoXY(Index, iY);
88 :   TextColor(Yellow);
89 :   Write('I');
90 : end;
91 :
92 : function TCurrentDays.DayOfMonth(Year,Month:integer):Integer;
93 : begin
94 :   Result:= 0;
95 :   case Month of
96 :     1, 3, 5, 7, 8,10, 12 : Result:= 31;
97 :     4, 6, 9, 11          : Result:= 30;
98 :     2                    : if DayOfYear(Year) = 365 then
Result:= 28
99 :       else Result:= 29;
100 :   end;
101 : end;
102 :
103 : function TCurrentDays.DayOfYear(Year:Integer):Integer;
104 : begin
105 :   if ((Year Mod 4) = 0) then begin
106 :     if ((Year Mod 100) = 0) then begin
107 :       if ((Year Mod 400) = 0) then DayOfYear:=366
108 :       else DayOfYear:=365;
109 :     end else
110 :       DayOfYear:=366;
111 :   end else
112 :     DayOfYear:=365;
113 : end;

```

```

114 :
115 : function TCurrentDays.IntToDate(Value:integer):TDateRec;
116 : begin
117 :   Result.Year:= Value div 10000;
118 :   Result.Month:= (Value - Result.Year*10000) div 100;
119 :   Result.Day:= Value - Result.Year*10000 - Result.Month*100;
120 : end;
121 :
122 : function TCurrentDays.GetHowManyDayYouLived(Birthday, Today:
integer): integer;
123 : var
124 :   Loop : Integer;
125 :   StartDate, EndDate : TDateRec;
126 : begin
127 :   StartDate:= IntToDate(Birthday);
128 :   EndDate:=   IntToDate(Today);
129 :
130 :   Result:= 0;
131 :
132 :   for Loop:= StartDate.Year to EndDate.Year-1 do
133 :     Result:= Result + DayOfYear(Loop);
134 :
135 :   for Loop:= 1 to StartDate.Month-1 do
136 :     Result:= Result - DayOfMonth(StartDate.Year, Loop);
137 :
138 :   Result:= Result - StartDate.Day;
139 :
140 :   for Loop:= 1 to EndDate.Month-1 do
141 :     Result:= Result + DayOfMonth(EndDate.Year, Loop);
142 :
143 :   Result:= Result + EndDate.Day;
144 :   FValue:= Result;
145 : end;
146 :
147 : function TCurrentDays.Value: integer;
148 : begin
149 :   Result:= FValue;
150 : end;
151 :
152 : end.

```

본 책은 입문자가 오브젝트 파스칼을 쉽게 이해할 수 있도록, 중요도에 상관없이 모르더라도 크게 문제가 되지 않을 것 같은 것들에 대한 설명을 생략하였습니다. 하지만, 생략된 것들 중에서도 중요하다고 생각되는 몇 가지를 주려서 설명을 드리고자 합니다.

변수의 보충 수업

아래는 이전 내용에서 설명을 생략했던 변수 종류에 대한 설명입니다.

정수형 변수타입

타입	범위	타입
정수	-2147483648..2147483647	부호있는 32비트
기수	0..4294967295	부호없는 32비트
Shortint	-128..127	부호있는 8비트
Smallint	-32768..32767	부호있는 16비트
Longint	-2147483648..2147483647	부호있는 32비트
Int64	-263..263-1	부호있는 64비트
Byte	0..255	부호없는 8비트
Word	0..65535	부호없는 16비트
Longword	0..4294967295	부호없는 32비트

실수형 변수타입

타입	범위	유효 자릿수	바이트 크기
<i>Real48</i>	$2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$	11-12	6
<i>Single</i>	$1.5 \times 10^{-45} \dots 3.4 \times 10^{38}$	7-8	4
<i>Double</i>	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	15-16	8
<i>Extended</i>	$3.6 \times 10^{-4951} \dots 1.1 \times 10^{4932}$	19-20	10
<i>Comp</i>	$-2^{63}+1 \dots 2^{63}-1$	19-20	8
<i>Currency</i>	-922337203685477.5808..922337203685477.5807	19-20	8
Real	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	15-16	8

문자열 변수타입

타입	최대길이	필요 메모리	용도
ShortString	255 문자	2에서 256바이트	역 호환성
AnsiString	~231 문자	4바이트에서 2기가 바이트	8비트(ANSI) 문자
WideString	~230 문자	4바이트에서 2기가 바이트	유니코드 문자, 다중 사용자 서버 및 다중 랭귀지 애플리케이션

불린 변수타입

불린의 경우에는 **Boolean**, **ByteBool**, **WordBool**, **LongBool** 이라는 타입명이 존재합니다. 하지만, **Boolean** 이외의 타입은 다른 언어나 OS와의 호환성을 위해서 존재할 뿐, 일반적으로는 **Boolean** 만을 사용합니다.

모든 불린형 변수의 경우에는 저장할 수 있는 값이 **true** 와 **false** 두 가지 밖에 없습니다.

```
1 : procedure TForm1.Button1Click(Sender: TObject);  
2 : var  
3 :   OK : Boolean;  
4 : begin  
5 :   OK:= 1 <> 2;  
6 :   if OK then ShowMessage('두 숫자는 서로 달라요. ');  
7 :   if 1 <> 2 then ShowMessage('두 숫자는 서로 다르다니까요 ー.ー+');  
8 : end;
```

5: 라인에서는 $1 \neq 2$ 라는 논리연산 결과 값이 **OK** 라는 불린 변수에 저장됩니다. $1 \neq 2$ 는 서로 다르기 때문에 **OK** 에는 참의 값인 **true** 가 입력됩니다.

6: 라인과 7: 라인은 **OK** 와 $1 \neq 2$ 연산 결과값이 서로 같기 때문에 같은 결과를 보여줍니다.

Overload

가끔 같은 용도로 사용되는 함수가 약간은 다른 사용법이 존재해야할 경우가 생깁니다. 이럴 때 **Overload** 는 아주 유용하게 사용될 수 있습니다.

[리스트 OP_01]

```
1 : program OP_01;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   i, j : integer;
10 :   x, y : double;
11 :
12 : function AddIntegers(a,b:integer):integer;
13 : begin
14 :   Result:= a + b;
15 : end;
16 :
17 : function AddDoubles(a,b:double):double;
18 : begin
19 :   Result:= a + b;
20 : end;
21 :
22 : begin
```

```

23 :   i:= 1;
24 :   j:= 2;
25 :   x:= 1.1;
26 :   y:= 2.2;
27 :
28 :   WriteLn('i + j = ', AddIntegers(i, j));
29 :   WriteLn('x + y = ', AddDoubles(x, y));
30 :
31 :   ReadLn;
32 : end.

```

12: 와 17: 라인에서는 두 개의 인자를 받아서 그 합을 출력하는 함수 두 개가 선언되어 있습니다. 이들은 목적은 같지만 12: 라인의 함수는 정수형 변수의 합을 구하고, 17: 라인의 함수는 실수형 변수의 합을 구하는 것이 서로 다릅니다.

이제 [리스트 OP_01] 소스를 **Overload** 를 사용하여, 아래와 같은 방식으로 수정하면 함수 이름을 같게하면서 서로 다른 동작을 할 수 있도록 할 수 있습니다.

```

1 : function Add(a,b:integer):integer; overload;
2 : begin
3 :   Result:= a + b;
4 : end;
5 :
6 : function Add(a,b:double):double; overload;
7 : begin
8 :   Result:= a + b;
9 : end;
10 :
11 : begin

```

```

12 :   i:= 1;
13 :   j:= 2;
14 :   x:= 1.1;
15 :   y:= 2.2;
16 :
17 :   WriteLn('i + j = ', Add(i, j));
18 :   WriteLn('x + y = ', Add(x, y));
19 :
20 :   ReadLn;
21 : end.

```

1: 과 6: 라인의 함수 선언은 원칙적으로는 불가능한 것입니다. 이름이 같은 함수가 프로그램 내에 하나 이상 존재할 수는 없기 때문입니다. 하지만, **Overload** 를 사용하면 같은 이름의 함수를 여러 개 만들어 낼 수 있습니다.

17:과 18: 라인에서 **Add** 라는 동일한 식별자로 함수를 호출하고 있습니다. 하지만, 17: 라인은 1: 라인에서 선언한 함수가 실행되며, 18: 라인에서는 6: 라인에서 선언된 함수가 실행됩니다.

이처럼 같은 이름으로 만들어진 함수는 파라미터에 의해서 어떤 함수가 실행이 될지 결정됩니다.

배열에 대한 보충 수업

배열에는 아래와 같이 범위를 지정하는 다양한 방법이 제시되어 있습니다. 이미 배우신 문법으로도 충분하기 때문에 배열을 설명하는 곳에서는 설명을 생략한 방법입니다. 자주 거론하지만, 입문자의 경우에는 너무 많은 것을 알기 보다 알고 있는 것을 바탕으로 응용하는 방법에 익숙해지는 것이 급선무입니다.

변수타입을 이용하여 범위를 지정하는 방법

```
1 : var
2 :    // array [변수타입] of 변수타입;
3 :    Arr1 : array [Byte] of integer;
4 :    Arr2: array [0..255] of integer;
```

Arr1 과 Arr2 는 똑 같은 형태의 배열로 선언됩니다.

다차원 배열을 정의할 때 표현 방법

```
1 : var
2 :    ArrA : array [1..10, 1..10] of integer;
3 :    ArrB : array [1..10] of array [1..10] of integer;
```

ArrA 과 ArrB 는 똑 같은 형태의 배열로 선언됩니다.

Self

클래스를 정의하고 있는 동안에는 객체가 생성되기 전입니다. 하지만, 자신이 객체화 되면 자신의 레퍼런스를 참조하도록 코드를 작성할 필요가 생깁니다. 이때, 자기 자신의 주소가 어디에 있는 지를 알려주는 지시자가 바로 **Self** 입니다.

Self 는 클래스 입장에서 자신이 메모리에 생성될 때, 해당 주소를 가르키는 일종의 변수입니다.

이해를 돕기 위해서 아래와 같은 예를 들어보겠습니다.

```
1 : constructor TList.Create;
2 : var
3 :   Loop : Integer;
4 : begin
5 :   for Loop := 1 to 10 do begin
6 :     Items[Loop] := TItem.Create;
7 :     Items[Loop].Parent := Self;
8 :     WriteLn(Loop, '번 객체가 생성되었습니다.');
```

위의 소스는 [리스트 OP_02]의 일부분입니다. **TList** 로 생성된 객체는 **TItem** 클래스로 생성된 객체를 내부배열 **Items** 로 보관하는 하게 됩니다. 그리고, **TItem** 으로 생성된 객체가 해제되면 **TList** 내부의 배열에서도 해당 객체가 해제되었다는 것을 알아야한다는 가정입니다.

즉, **TItem** 클래스로 생성된 객체는 자신을 관리하고 있는 **TList** 로 생성된 객체에게 자신이 메모리상에서 해제되는 것을 알려줘야 하는 책임이 있는 것입니다. 하지만, **TList** 를 정의하고 있는 동안에는 **TList** 로 생성된 객체가 없기 때문에 **Self** 가 없다면, 이를 클래스 내부에서 지정해줄 수 없습니다.

6: 라인에서는 **TItem** 클래스를 이용해서 객체를 생성한 후, 해당 주소를 **Items[Loop]**에 저장하고 있습니다.

7: 라인에서는 이미 생성된 객체 **Items[Loop]**의 내부변수 **Parent** 에 **Self** 를 저장하고 있습니다. 여기서 이미 설명드린 것처럼, **Self** 는 **TList** 로 생성된 객체의 주소입니다.

[리스트 OP_02]

```
1 : program OP_02;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TList = class;
10 :
11 :   TItem = class
12 :     Parent : TList;
13 :     destructor Destroy; override;
14 :   end;
15 :
16 :   TList = class
17 :     Items : array [1..10] of TItem;
18 :     constructor Create;
19 :     procedure DeleteItem(Index:integer);
20 :     procedure DeletedNotify(Child:TItem);
21 :   end;
22 :
```



```

23 : { TClassA }
24 :
25 : constructor TList.Create;
26 : var
27 :   Loop : Integer;
28 : begin
29 :   for Loop := 1 to 10 do begin
30 :     Items[Loop] := TItem.Create;
31 :     Items[Loop].Parent := Self;
32 :     WriteLn(Loop, '번 객체가 생성되었습니다. ');
33 :   end;
34 : end;
35 :
36 : procedure TList.DeletedNotify(Child: TItem);
37 : var
38 :   Loop : Integer;
39 : begin
40 :   for Loop := 10 downto 1 do begin
41 :     if Items[Loop] = Child then begin
42 :       Items[Loop] := nil;
43 :       WriteLn(Loop, '번 객체가 해제되었습니다. ');
44 :     end;
45 :   end;
46 : end;
47 :
48 : procedure TList.DeleteItem(Index: integer);
49 : begin

```

```

50 :   Items[Index].Free;
51 : end;
52 :
53 : { TClassB }
54 :
55 : destructor TItem.Destroy;
56 : begin
57 :   Parent.DeletedNotify(Self);
58 :
59 :   inherited;
60 : end;
61 :
62 : var
63 :   List : TList;
64 :
65 : begin
66 :   List:= TList.Create;
67 :   List.DeleteItem(3);
68 :
69 :   ReadLn;
70 : end.

```

9: 라인은 **TList** 라는 클래스가 다음에 정의될 것이라는 예고문입니다.

예고문이 필요한 이유는 12: 라인에서 **TItem** 클래스가 **TList** 클래스를 사용하고 있지만, 아직 **TList** 클래스가 정의되지 않은 상태이기 때문에 예고문없이는 컴파일할 수 없기 때문입니다. 이러한 경우에는 **TList** 를 **TItem** 이전에 정의하면 됩니다.

하지만, **TList** 클래스는 17: 라인에서 **TItem** 클래스 사용하고 있기 때문에 어느 한쪽을 앞에서 선언하면 같은 상황만 반복하게 됩니다. 따라서, “한쪽이 저 뒤에 정의될 것니까 미리 사용해도 되”라고 예고문을 사용하게 되는 것입니다.

67: 라인에서는 **TItem** 을 이용해서 3 번째 생성된 객체를 삭제하고 있습니다.

50: 라인에서는 해당 객체의 **Free Method** 를 이용하여 메모리에서 삭제하고 있습니다.

57: 라인에서는 원래의 의도대로 **TItem** 자신이 삭제된 것을 **TList** 로 생성된 객체에게 알려주고 있습니다. **TItem.Parent** 는 31: 라인을 통해서 **TList** 의 **Self** 의 값으로 지정되어 있습니다. 이제 해당 주소에 있는 객체의 **DeletedNotify()** Method 를 실행하고 있습니다. 여기서도 자신의 주소를 **Parent** 에게 알려주어야 하기 때문에 **Self** 를 사용했습니다.

36-56: 라인에서는 삭제된 **TItem** 로 생성된 객체를 배열 중에서 찾아서 삭제되었음을 표시하고 있습니다.

“**Self** 는 클래스 입장에서 자신이 메모리에 생성될 때, 해당 주소를 가르키는 일종의 변수입니다.”라고 간단히 설명하고 끝날 수도 있지만, 이렇게 다소 복잡한 예제를 통해서 설명한 이유가 있습니다. 그것은 클래스와 객체에 대한 정확한 이해는 그렇게 쉽지 않기 때문입니다. 클래스와 객체를 정확하게 구별해야지만 **Self** 의 진정한 의미를 파악할 수 있기 때문입니다.

만약 아직 클래스와 객체를 정확하게 구별하지 못하는 입문자라면 [리스트 OP_03]과 같은 소스를 생각할 수 있습니다.

[리스트 OP_03]

```
1 : program OP_03;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :     SysUtils;
7 :
8 : type
9 :     TItem = class
10 :         destructor Destroy; override;
11 :     end;
12 :
13 :     TList = class
```

```

14 :     Items : array [1..10] of TItem;
15 :     constructor Create;
16 :     procedure DeleteItem(Index:integer);
17 :     procedure DeletedNotify(Child:TItem);
18 :     end;
19 :
20 : var
21 :     List : TList;
22 :
23 : { TClassA }
24 :
25 : constructor TList.Create;
26 : var
27 :     Loop : Integer;
28 : begin
29 :     for Loop := 1 to 10 do begin
30 :         Items[Loop]:= TItem.Create;
31 :         WriteLn(Loop, '번 객체가 생성되었습니다. ');
32 :     end;
33 : end;
34 :
35 : procedure TList.DeletedNotify(Child: TItem);
36 : var
37 :     Loop : Integer;
38 : begin
39 :     for Loop := 10 downto 1 do begin
40 :         if Items[Loop] = Child then begin

```

```

41 :      Items[Loop] := nil;
42 :      WriteLn (Loop, '번 객체가 해제되었습니다. ');
43 :      end;
44 :  end;
45 : end;
46 :
47 : procedure TList.DeleteItem (Index: integer);
48 : begin
49 :   Items[Index].Free;
50 : end;
51 :
52 : { TClassB }
53 :
54 : destructor TItem.Destroy;
55 : begin
56 :   List.DeletedNotify (Self);
57 :
58 :   inherited;
59 : end;
60 :
61 : begin
62 :   List := TList.Create;
63 :   List.DeleteItem (3);
64 :
65 :   ReadLn;
66 : end.

```

9-11: 라인에서 보면 TItem 클래스에 Parent 를 지정하는 변수가 사라졌습니다.

29-32: 반복문에서도 **Parent** 지정은 하지 않고 있습니다.

56: 라인에서는 **Parent** 대신 **List** 변수를 직접이용하여 **TList** 로 생성된 객체를 호출하고 있습니다.

우선 [리스트 OP_03]은 [리스트 OP_02]와 같은 동작을 합니다. 하지만, [리스트 OP_03]의 61-66: 라인을 아래와 같이 수정해보도록 하겠습니다.

```
1 : var
2 :   List2 : TList;
3 :
4 : begin
5 :   List:= TList.Create;
6 :   List.DeleteItem(3);
7 :
8 :   List2:= TList.Create;
9 :   List2.DeleteItem(3);
10 :
11 :   ReadLn;
12 : end.
```

9: 라인에서는 객체가 삭제된 후 이것이 화면에 표시되지 않습니다. 이유는 **TList.Destroy** 에서 호출한 것은 **List** 객체이기 때문입니다. 즉, **List2** 에 배열로 관리되는 **TItem** 로 생성된 객체가 해제되도 호출은 **List** 의 **DeletedNotify()** Method 가 호출되는 것입니다. 그리고, 6: 라인을 통해서 같은 3 번 인덱스에 해당하는 객체가 이미 삭제되었음을 알렸기 때문에 9: 라인에서 삭제된 것은 화면에 표시되지 않습니다.

만약, [리스트 OP_02]에서 위의 소스를 같이 변경을 가하면, 3 번 객체가 삭제된 것을 두 번 알리게 됩니다.

이름이 정해진 **Named type** 과 익명의 **type** 선언

[리스트 OP_04]을 통해서 이름이 정해진 **Named type** 과 익명의 **type** 선언에 대해서 설명해보도록 하겠습니다.

[리스트 OP_04]

```
1 : program OP_04;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TBuffer = array [1..1024] of byte;
10 :
11 : var
12 :   Buffer1, Buffer2 : TBuffer;
13 :   Buffer3 : array [1..1024] of byte;
14 :   Buffer4 : array [1..1024] of byte;
15 :
16 : begin
17 :   Buffer2:= Buffer1;
18 :
19 :   // 컴파일되지 않음
20 :   Buffer3:= Buffer1;
21 :   Buffer3:= Buffer4;
```

```
22 : end.
```

17: 라인의 **Buffer1** 과 **Buffer2** 는 같은 타입이기 때문에 잘 컴파일이 됩니다.

하지만, 똑 같은 배열로 선언했지만, 오브젝트 파스칼에서 20-21: 라인은 전혀 다른 타입으로 생각합니다.

여기서, **TBuffer** 는 타입의 이름이 **TBuffer** 로 확정된 **Named type** 이라고 합니다. 그리고, 뒤에 오는 **array [1..1024] of byte** 는 이름이 없는 익명의 타입이라고 합니다. 기억하시겠지만, 변수의 선언은 아래와 같습니다.

```
var  
  
    변수명 : 변수타입;
```

즉, **array [1..1024] of byte** 는 변수타입이라고 합니다. 그런데, 21: 라인은 같은 변수타입으로 선언된 것처럼 보이지만, 오브젝트 파스칼은 다르다라고 생각하는 것입니다. 익명으로 선언된 타입은 그 정의가 같더라도 다르다고 판단합니다.

type 에서 선언하지 않은 **of** 예약어를 포함한 변수타입을 익명의 타입선언이라고 합니다. 배열 이외에는 아래와 같이 집합 선언에서도 가능합니다.

```
var  
  
    ByteSet : set of Byte;
```


16 진수의 표현 방법

오브젝트 파스칼에서 16 진수를 표시하려면 **\$숫자** 와 같은 포맷을 사용합니다. 아래 소스를 실행하면 WriteLn 문이 실행되면서 화면에 해당 문자열이 표시될 것입니다.

```
if $FF = 255 then
    WriteLn('16 진수 $FF 는 십진수 255 에 해당합니다.');
```

Set

오브젝트 파스칼에는 집합연산을 할 수 있는 **Set** 이라는 변수타입이 존재합니다. 설명을 위해 아래의 소스를 살펴보도록 하겠습니다.

```
1 : var
2 :   Set1, Set2, Set3, Set4 : set of byte;
3 :
4 : begin
5 :   Set1:= [1, 2, 3];
6 :   Set2:= [1, 2, 3];
7 :   if Set1 = Set2 then WriteLn('Set1 = Set2');
8 :
9 :   Set3:= Set1 - [1, 2, 4];
10 :   Set4:= [3];
11 :   if Set3 = Set4 then WriteLn('Set3 = Set4');
12 : end.
```

3: 라인에서는 집합 변수 **Set1, Set2, Set3, Set4** 를 선언하고 있습니다. 집합 타입으로 선언할 수 있는 변수타입은 0..255 의 값을 가지는 변수들로 한정되어 있습니다. 따라서, integer 등을 집합의 원소로 사용할 수 없습니다.

9: 라인에서는 집합연산의 차집합을 설명하고 있습니다. 일반적인 뺄셈과 달리, 집합의 차집합 연산에서처럼 **Set1** 에 없는 원소인 4 는 뺄셈에서 무시됩니다.

따라서, 11: 라인에서는 **Set3** 과 **Set4** 가 같은 집합으로 취급됩니다.

합집합연산에서도 마찬가지로 작동하게 됩니다.

주석과 주석 아닌 것들

```
1 :    // 한줄 짜리 주석입니다.
2 :    {
3 :        여러줄을
4 :        사용할 수 있는 주석입니다.
5 :    }
6 :    (*
7 :        여러줄을
8 :        사용할 수 있는 주석입니다.
9 :    *)
10 :
11 :    {$R *.RES}
12 :    {$IFDEF MSWINDOWS}
13 :        WriteLn('MS 윈도우에서 컴파일합니다. ');
14 :    {$ENDIF}
15 :    {$IFDEF LINUX}
16 :        WriteLn('리눅스에서 컴파일합니다. ');
17 :    {$ENDIF}
```

1-9: 라인은 주석의 여러 가지 종류를 설명하고 있습니다. 1: 라인을 제외하면 나머지는 여러줄에 걸쳐서 사용할 수 있는 주석입니다.

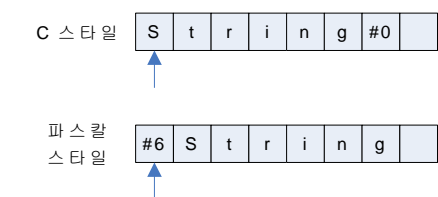
11-17: 라인에 작성되어 있는 코드는 주석이 아닌 컴파일러 지시자입니다. 주석은 있으나 없으나 상관없는 존재이지만, 컴파일러 지시자의 경우에는 컴파일할 때 컴파일러가 사용하기 때문에 중요한 코드입니다. 입문자에게 필요한 컴파일러 지시자는 델파이가 자동으로 생성하기 때문에 특별히 알아야 할 것은 없습니다. 컴파일러 지시자에 대해 좀더 알고 싶으신 분들은 델파이에 함께 제공되는 pdf 문서 또는 도움말을 참고하시기 바랍니다.

이번 장에서는 기초적인 알고리즘을 통해서 오브젝트 파스칼 문법에 대한 연습을 하도록 하겠습니다. 여기서 훌륭한 알고리즘을 소개하기 보다, 쉽게 이해할 수 있는 것들을 간추려보았습니다.

대소문자 변환

문자열의 구조

우선 우리가 문자열 변수에 문자열 데이터를 입력하면 컴퓨터는 이것을 어떻게 처리할 까요? 이에 대해서 [그림 86]에서는 대표적인 두 가지 방법에 대한 예를 보여 주고 있습니다.



[그림 86] 문자열을 저장하는 2 가지 방법

일단 'String' 이란 문자열을 변수(메모리)에 저장했다고 가정하겠습니다. 그 과정을 C 언어의 경우에는 [그림 86]에서의 위에 있는 그림처럼 문자열이 끝나는 부분에 #0 (아스키 코드 0 번에 해당하는) 문자를 집어 넣게 됩니다. 이러한 방식으로 문자열을 표시하는 것을 “null terminated string”이라고 부릅니다.

메모리 공간은 연속적으로 이어져 있기 때문에, 사용자가 입력한 문자열이 끝나는 위치를 표시할 수 있어야 하기 때문입니다. 화살표는 변수가 저장되어 있는 메모리 공간의 위치를 표시한 것입니다.

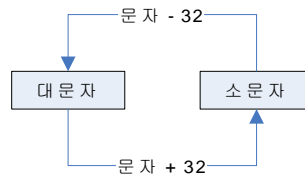
파스칼과 같은 언어는 맨 앞에 공간을 문자열의 길이를 표시하도록 되어 있습니다. 따라서, 맨 앞에 있는 길이 정보만 변경하면, 바로 인식할 수 있는 문자의 길이가 달라집니다. 현재, 델파이에서는 [그림 86]과는 다른 형식으로 문자열을 관리합니다. [그림 86]은 고전적인 방식으로 길이를 저장하는 곳이 1 바이트이기 때문에, 길이가 255 를 넘어가는 문자열을 표시할 수 없습니다. 이전에 파스칼에서는 그러한 제약이 있었습니다.

두 방식은 장단점이 있습니다. 후자의 경우에는 문자열을 다루기 쉬워서 “+” 연산도 가능합니다. C 스타일의 언어는 함수를 통해서 문자열을 붙여나갑니다. 파스칼 스타일의 문자열은 문자의 길이를 줄이거나 알아내는 과정이 비교적 효율적입니다. 또한, C 스타일의 문자열에서는 #0 문자를 포함할 수 없지만, 파스칼에서는 #0 문자를 표현할 수 있다는 것이 다릅니다.

두 문자열의 이러한 차이점 때문에 파스칼(델파이)로 작성된 dll 과 C 계열 언어로 작성된 dll 간에 문자열 교환 시 주의해야 합니다. dll 의 문자열은 C 스타일로 된 문자열을 취급하기 때문에, 델파이에서 dll 을 작성하실 때에는 “null terminated string”을 지원하는 PChar 타입을 사용하여야 합니다.

대소문자 변환 방법

이미 파스칼 문법 강좌에서 아스키 코드 표에 대해서 설명한 적이 있습니다. 각 문자는 고유한 번호를 가지고 있다는 것을 도표로 설명했었던 것을 기억하시는지요?



[그림 87] 대소문자의 변환 방법

아스키 코드 표에는 대문자가 먼저 등장합니다. 대문자 “A”와 소문자 “a”는 숫자로 32 정도 차이를 두고 떨어져 있습니다. 그리고, 모든 문자들은 순서대로 번호가 주어져 있기 때문에, 대문자에서 소문자로 변환하려면, 해당 문자에다가 32 라는 숫자만 더해주면 됩니다. 반대로 소문자에서 32 를 빼주면 대문자가 됩니다.

문자열 전체를 대문자로 변환

[리스트 A1_01]

```
1 : program A1_01;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Loop : integer;
10 :   sTemp : string = 'This is a temporary variable.';
```

```

11 :
12 : begin
13 :   for Loop := 1 to Length(sTemp) do
14 :     Write(''+sTemp[Loop]+'=', Byte(sTemp[Loop]), ', ');
15 :   WriteLn;
16 :
17 :   for Loop := 1 to Length(sTemp) do
18 :     if sTemp[Loop] in ['a'..'z'] then
19 :       sTemp[Loop] := Char( Byte(sTemp[Loop]) - 32 );
20 :   WriteLn('Result : ', sTemp);
21 :
22 :   ReadLn;
23 : end.

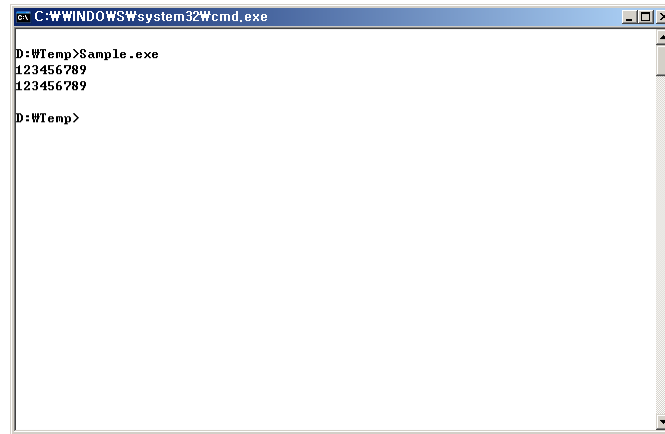
```

숫자와 문자열 간의 변환

숫자 변수와 문자 변수의 차이점

[리스트 A1_02]

```
1 : program A1_02;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   iTemp : integer = 123456789;
10 :   sTemp : string = '123456789';
11 :
12 : begin
13 :   WriteLn(iTemp);
14 :   WriteLn(sTemp);
15 : end.
```



[그림 88]

[그림 88]은 [리스트 AI_02]의 실행결과입니다. [그림 88]에서 보면 정수형태로 저장한 데이터나 문자열로 저장한 데이터나 화면에 표시되는 것은 전혀 틀리지 않은 것을 확인할 수 있습니다.

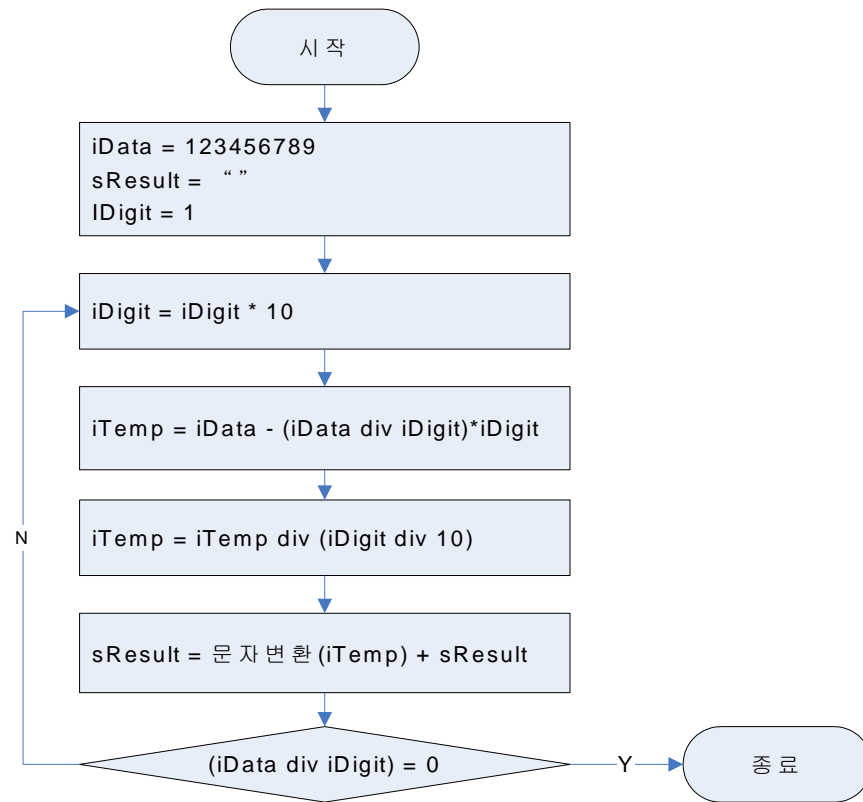
하지만, 저장된 두 데이터가 차지하고 있는 공간의 크기는 서로 확연한 차이를 보입니다. 정수 변수인 “iTemp”의 크기는 4 바이트로 고정입니다. 대신 4 바이트가 표현할 수 없는 숫자의 범위(파스칼 문법 입문 강좌 참조)를 표시할 수는 없습니다.

문자열은 각 숫자처럼 보이는 문자열 마다 1 바이트씩 차지하게 됩니다. 또한, 문자열이 저장되어 있는 공간 정보를 위해서 4 바이트를 추가로 사용하게 됩니다. 따라서, [리스트 AI_02]의 소스에서는 총 13 바이트의 메모리 공간을 차지하게 됩니다.

또한, 정수는 덧셈을 비롯한 사칙연산과 같은 수학적 계산이 가능하지만, 문자열은 그러한 연산을 할 수 없습니다.

따라서, 얼핏 비슷해 보이는 두 데이터에는 커다란 차이가 존재한다는 것을 알 수 있습니다.

문자를 문자열로 변환 하기



[그림 89] 숫자를 문자열로 변환하기

[리스트 A1_03]

```

1 : program A1_03;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
  
```

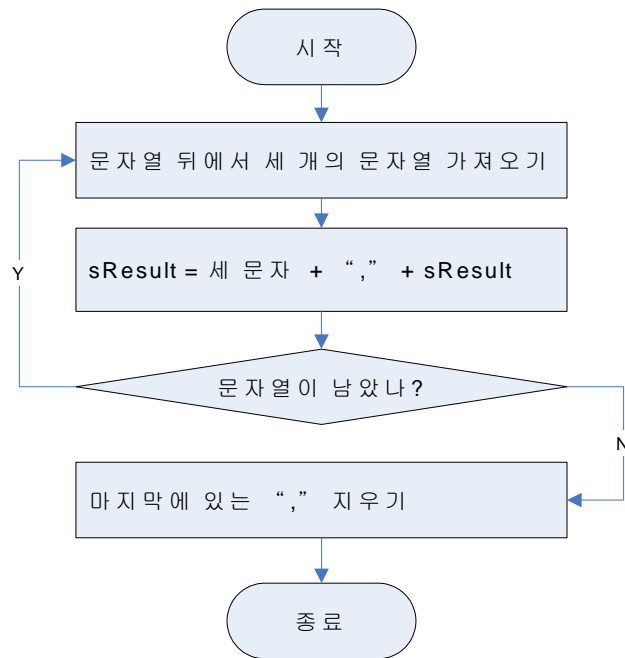
```

 9 :   iData : integer = 123456789;
10 :   sResult : string = '';
11 :   iDigit : integer = 1;
12 :   iTemp : integer;
13 :
14 : begin
15 :   repeat
16 :     iDigit:= iDigit * 10;
17 :     iTemp:= iData - (iData div iDigit) * iDigit;
18 :     iTemp:= iTemp div (iDigit div 10);
19 :     sResult:= Char(iTemp+48) + sResult;
20 :   until (iData div iDigit) = 0;
21 :
22 :   WriteLn('Result : ', sResult);
23 :
24 :   ReadLn;
25 : end.

```

19: 라인의 “iTemp + 48”에서 48 은 아스키 코드 값에서 “0” 문자에 해당하는 숫자입니다.

세 자리 수마다 콤마 찍기



[그림 90] 세 자리 수마다 콤마 찍기

[리스트 A1_04]

```

1 : program A1_04;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   iData : integer = 123456789;
10 :   sTemp : string;
11 :   sResult : string = '';
  
```

```

12 :
13 : begin
14 :   sTemp:= IntToStr(iData);
15 :   sResult:= '';
16 :
17 :   while Length(sTemp) > 3 do begin
18 :     sResult:= Copy(sTemp, Length(sTemp)-2, 3) + ',' + sResult;
19 :     SetLength(sTemp, Length(sTemp)-3);
20 :   end;
21 :
22 :   sResult:= sTemp + ',' + sResult;
23 :
24 :   SetLength(sResult, Length(sResult)-1);
25 :
26 :   WriteLn('Result : ', sResult);
27 :
28 :   ReadLn;
29 : end.

```

문자를 숫자로 변환 하기

이번 소스는 너무나 쉽기 때문에 플로우 차트를 그리지 않겠습니다.

[리스트 A1_05]

```
1 : program A1_05;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   sData : string = '123456789';
10 :   iResult : integer = 0;
11 :   Loop : integer;
12 :   iDigit : integer = 1;
13 :
14 : begin
15 :   for Loop := Length(sData) downto 1 do begin
16 :     iResult:= iResult + (Byte(sData[Loop])-48) * iDigit;
17 :     iDigit:= iDigit * 10;
18 :   end;
19 :
20 :   WriteLn('Result : ', iResult);
21 :
22 :   ReadLn;
```

```
23 : end.
```

9: 라인에 있는 **sData** 라는 문자열 변수 안에 있는 숫자를 정수형 변수인 **iResult** 에 변환해서 입력하는 과정입니다.

문자열 검색과 치환

문자열 검색 하기

[리스트 A1_06]

```
1 : program A1_06;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   sSource : string = 'Find xxx from string!';
10 :   sFind : string = 'xxx';
11 :   iResult : integer = 0;
12 :   Loop : integer;
13 :
14 : begin
15 :   for Loop := 1 to Length(sSource) do
16 :     if Copy(sSource, Loop, Length(sFind)) = sFind then begin
17 :       iResult:= Loop;
18 :       Break;
19 :     end;
20 :
21 :   WriteLn('Result : ', iResult);
```

```
22 :  
23 :   ReadLn;  
24 : end.
```

[리스트 AI_06]은 문자열 내부에서 원하는 문자가 어디쯤에 있는 검색하는 소스입니다. 원본 문자열인 **sSource** 안의 문자를 처음부터 끝까지 하나씩 찾아가면서, 찾고자 하는 문자열만큼 잘라온 후에 비교하는 간단한 소스입니다. 델파이에서 기본적으로 제공하는 **Pos()** 함수와 같은 작업을 진행하게 됩니다.

문자열 치환 하기

[리스트 A1_07]

```
1 : program A1_07;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   sSource : string = 'Replace xxx from string!';
10 :   sFind : string = 'xxx';
11 :   sTarget : string = 'Ryu';
12 :   sResult : string = '';
13 :   iPos : integer = 0;
14 :   Loop : integer;
15 :
16 : begin
17 :   for Loop := 1 to Length(sSource) do
18 :     if Copy(sSource, Loop, Length(sFind)) = sFind then begin
19 :       iPos:= Loop;
20 :       Break;
21 :     end;
22 :
23 :   sResult:= Copy(sSource, 1, iPos-1);
24 :   sResult:= sResult + sTarget;
```

```
25 :   sResult:= sResult + Copy(sSource, iPos+Length(sFind),  
Length(sSource));  
  
26 :  
  
27 :   WriteLn('Result : ', sResult);  
  
28 :  
  
29 :   ReadLn;  
  
30 : end.
```

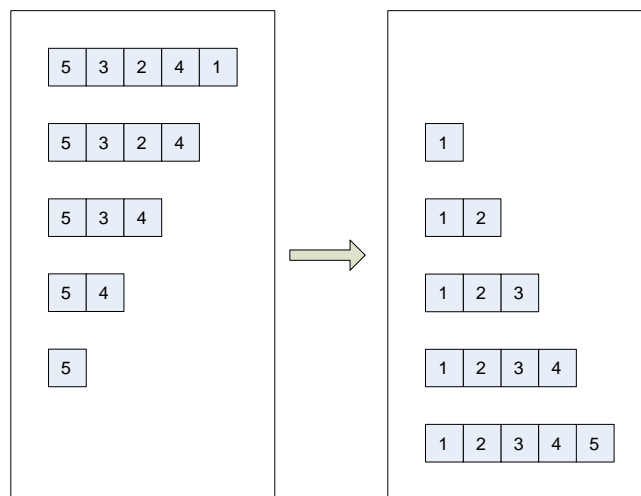
[리스트 AI_07]은 [리스트 AI_06]에서 찾아낸 문자를 다른 문자로 치환하는 작업을 설명하고 있습니다.

정렬

가장 단순한 소트

만약 “5 3 2 4 1”과 같은 데이터가 준비되어 있다고 가정하겠습니다. 이 데이터는 순서가 더욱 엉망으로 되어 있어도 상관없습니다. 이제 이 데이터를 “1 2 3 4 5”와 같은 순서로 정렬(소트)하고자 합니다.

가장 쉽게 생각할 수 있는 것은 데이터들을 저장할 수 있는 또 다른 변수(박스)를 하나 선언합니다. 이후, “5 3 2 4 1”에서 가장 작은 숫자를 찾아서 준비된 박스에 넣어둡니다. 그리고, 이러한 과정을 반복하면, 모든 데이터는 순서대로 정렬될 것입니다.



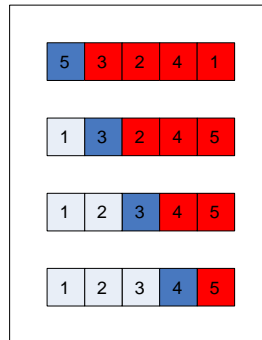
[그림 91] 단순한 정렬

[그림 91]과 같은 정렬 방법에는 두 가지 문제가 있습니다. 첫 째는, 정렬을 하고자 하는 데이터의 크기와 같은 크기의 다른 메모리 공간이 필요하다는 것입니다. 두 번째는 데이터의 이동이 많다는 것입니다. 왼쪽의 원래 데이터들도 오른쪽으로 데이터가 이동할 때마다 자리가 변동되어야 함을 알 수 있습니다.

따라서, 메모리의 효율이나 속도 면에서 다소 떨어지는 방법이라고 할 수 있습니다.

선택정렬의 소개

선택정렬은 간단한 소트보다는 한 단계 발전한 정렬방법입니다. 하지만, 이것 역시 그렇게 효율이 좋은 편은 아닙니다. 다만, 퀵소트와 같은 경우에는 코드가 다소 복잡하기 때문에 입문자에게 적합한 알고리즘으로 선택한 것입니다. 버블소트 정도는 직접 도전해보시는 것도 좋을 듯 합니다.



[그림 92] 선택정렬

[그림 92]에서처럼 데이터의 가장 첫 번째부터 마지막의 이전까지 반복합니다. 그림에서는 5 개의 데이터이기 때문에, 우선 4 번을 반복하는 것입니다.

첫 번째 반복에서는 5 라는 데이터를 기준으로 그 다음에 있는 모든 데이터 중에서 자신(5)을 포함해서 가장 숫자가 작은 숫자를 찾아내고, 서로 그 위치를 바꿔버립니다. 이후 두 번째 반복에서도 3 이후의 데이터와 자신(3)을 포함해서 가장 작은 숫자와 서로 그 위치를 바꾸어 버립니다. 자신이 가장 작은 숫자라면 바꿀 필요가 없겠지요.

이렇게 반복해 나가다 보면 모든 데이터가 순차적으로 정렬되는 것을 확인하실 수 있습니다.

선택정렬의 구현

[리스트 A1_08]

```
1 : program A1_08;
2 :
3 : {$APPTYPE CONSOLE}
4 :
```

```

5 : uses

6 :   SysUtils;

7 :

8 : var

9 :   Loop1, Loop2 : integer;

10 :   Datas : array [1..5] of integer = (5, 3, 2, 4, 1);

11 :

12 : procedure Swap(var a,b:integer);

13 : var

14 :   iTemp : integer;

15 : begin

16 :   iTemp:= a;

17 :   a:= b;

18 :   b:= iTemp;

19 : end;

20 :

21 : begin

22 :   for Loop1 := 1 to 5 - 1 do

23 :     for Loop2 := Loop1+1 to 5 do

24 :       if Datas[Loop1] > Datas[Loop2] then Swap(Datas[Loop1],

25 : Datas[Loop2]);

26 :     for Loop1 := 1 to 5 do

27 :       WriteLn(Loop1, ' : ', Datas[Loop1]);

28 :

29 :   ReadLn;

30 : end.

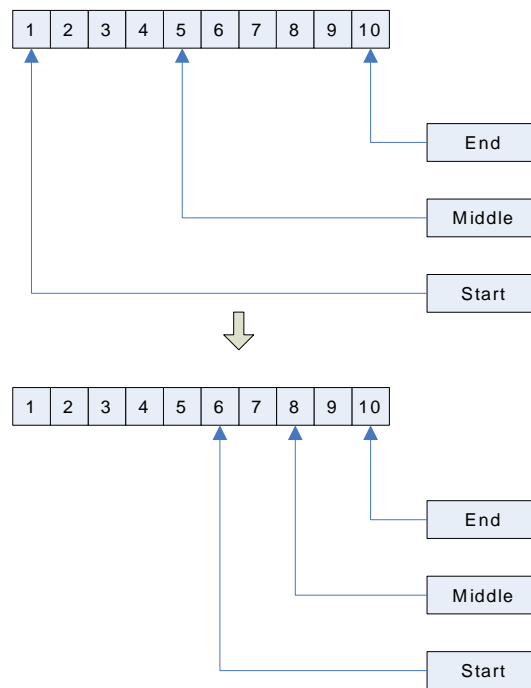
```

검색

이분 탐색법 소개

만약에 데이터가 순서대로 정렬되지 않았다면, 원하는 데이터를 찾기 위해서 전체의 데이터를 다 뒤져야만 합니다. 하지만, 우리가 영어사전을 사용할 때는 처음부터 원하는 단어를 찾기 보다 대략적인 위치를 선택하고, 이후 찾고자 하는 단어보다 순서가 앞이나 뒤냐를 따져서 쉽고 빠르게 찾아갈 수 있습니다.

이처럼 순서대로 정렬되어 있는 데이터의 경우에는 좀더 빠르고 효율적으로 데이터를 찾아낼 수 있는데, 이번 강좌에서는 이분법을 이용해서 데이터를 찾는 방법을 소개하겠습니다.



[그림 93] 이분 탐색법

[그림 93]은 이분 탐색법을 통해서 8 이라는 데이터를 찾아가는 과정입니다. 단 두 번의 동작으로 8 을 찾아낸 것입니다. 만약 다른 데이터였다면 (7 또는 9), 한 번의 과정을 더 밟으면 됩니다.

원리는 데이터의 시작과 끝 그리고 중간 값을 찾습니다. 중간 값이 만약 찾고자 하는 값보다 크다면, 중간 값 다음의 숫자를 시작 값으로 변경하고 다시 중간 값을 찾아나가는 것을 반복합니다. 이러다 보면 원하는 데이터가 포함되지 않는 반 토막을 잘라서 버려나가는 형태를 취하게 됩니다.

이분법은 상당히 빠른 시간 내에 데이터를 찾을 수 있습니다. 예를 들어 데이터의 개수가 수백만 건으로 늘어난다고 해도, 검색 시간이 수백만 배로 늘어나지는 않습니다. 초기 데이터가 크면 그만 한 번에 버려지는 반 토막의 크기도 크기 때문에 검색 시간의 증가가 지극히 완만하게 이루어집니다.

이분 탐색법 구현

[리스트 A1_09]

```
1 : program A1_09;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   TDatas = array [1..15] of integer;
10 :
11 : var
12 :   Loop : integer;
13 :   Datas : TDatas;
14 :
15 : procedure RandomizeDatas(var Values:TDatas);
16 : var
17 :   Loop : integer;
18 : begin
19 :   Randomize;
```

```

20 :   for Loop:= Low(Values) to High(Values) do
21 :       Values[Loop] := Round(Random(10));
22 : end;
23 :
24 : procedure Swap(var a,b:integer);
25 : var
26 :     iTemp : integer;
27 : begin
28 :     iTemp:= a;
29 :     a:= b;
30 :     b:= iTemp;
31 : end;
32 :
33 : procedure Sort(var Values:TDatas);
34 : var
35 :     Loop1, Loop2 : integer;
36 : begin
37 :     for Loop1 := Low(Values) to High(Values) - 1 do
38 :         for Loop2 := Loop1+1 to High(Values) do
39 :             if Values[Loop1] > Values[Loop2] then
40 :                 Swap(Values[Loop1], Values[Loop2]);
41 :             end;
42 :
43 : function BinarySearch(Values:TDatas; Data:integer):integer;
44 : var
45 :     iStart, iMiddle, iEnd : integer;
46 : begin

```



```

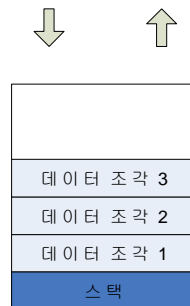
47 :   Result:= -1;
48 :   iStart:= Low(Values);
49 :   iEnd:=   High(Values);
50 :   iMiddle:= (iStart + iEnd) div 2;
51 :
52 :   while iStart < iEnd do begin
53 :       iMiddle:= (iStart + iEnd) div 2;
54 :       if Data > Values[iMiddle] then
55 :           iStart:= iMiddle + 1
56 :       else if Data = Values[iMiddle] then begin
57 :           Result:= iMiddle;
58 :           Exit;
59 :       end else
60 :           iEnd:= iMiddle - 1;
61 :   end;
62 :
63 :   if Data = Values[iStart] then Result:= iStart;
64 : end;
65 :
66 : begin
67 :   RandomizeDatas(Datas);
68 :   for Loop:= Low(Datas) to High(Datas) do
69 :       WriteLn(Loop, ' : ', Datas[Loop]);
70 :   ReadLn;
71 :   WriteLn;
72 :
73 :   Sort(Datas);

```

```
74 :   for Loop:= Low(Datas) to High(Datas) do  
75 :       WriteLn(Loop, ' : ', Datas[Loop]);  
76 :   ReadLn;  
77 :   WriteLn;  
78 :  
79 :   // Result = -1 : Can't find the data.  
80 :   WriteLn('Position = ', BinarySearch(Datas, 2));  
81 :  
82 :   ReadLn;  
83 : end.
```

스택

스택이란?



[그림 94] 스택

[그림 94]와 같이 스택은 데이터를 저장할 때, 마치 책을 쌓아 올리듯이 저장하는 방식을 취합니다. 따라서, 마지막에 쌓은 데이터가 가장 위에 있기 때문에, 데이터를 다시 꺼낼 때는, 마지막에 넣은 데이터부터 가져오게 하는 방식입니다. 이것을 **FILO(First In Last Out)**이라고 부르기도 합니다.

이때, 데이터를 집어넣는 동작을 **Push** 라고 하며, 데이터를 하나 꺼내는 동작을 **Pop** 이라고도 부릅니다.

스택은 작업을 잠시 미루다가 최근에 미뤘던 일부터 다시 시작하는 알고리즘에 자주 등장하게 됩니다. 이러한 방식에 대해서는 깊이 우선 탐색법 등에 대해서 찾아보시기 바랍니다. 윈도우의 파일 탐색기에서 하위 폴더를 찾아가는 과정이 바로 이러한 깊이 우선 탐색법을 사용하게 됩니다. 대부분 이러한 경우 재귀호출을 사용하게 됩니다. 재귀호출을 사용하면 사용자도 모르게 스택을 이용하게 되며, 이에 대해서는 재귀호출을 설명하는 **Chapter**에서 다루도록 하겠습니다.

문자열을 뒤집기

[리스트 Al_10]

```
1 : program Al_10;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   Loop : integer;
10 :   sSource : string = '123456789';
11 :   sReverse : string = '';
12 :   StackSize : integer = 0;
13 :   Stack : array [1..100] of char;
14 :
15 : procedure Push(Value:char);
16 : begin
17 :   StackSize:= StackSize + 1;
18 :   Stack[StackSize]:= Value;
19 : end;
20 :
21 : function Pop:char;
22 : begin
23 :   Result:= Stack[StackSize];
24 :   StackSize:= StackSize - 1;
```

```
25 : end;

26 :

27 : begin

28 :   for Loop := 1 to Length(sSource) do

29 :     Push(sSource[Loop]);

30 :

31 :   while StackSize > 0 do

32 :     sReverse:= sReverse + Pop;

33 :

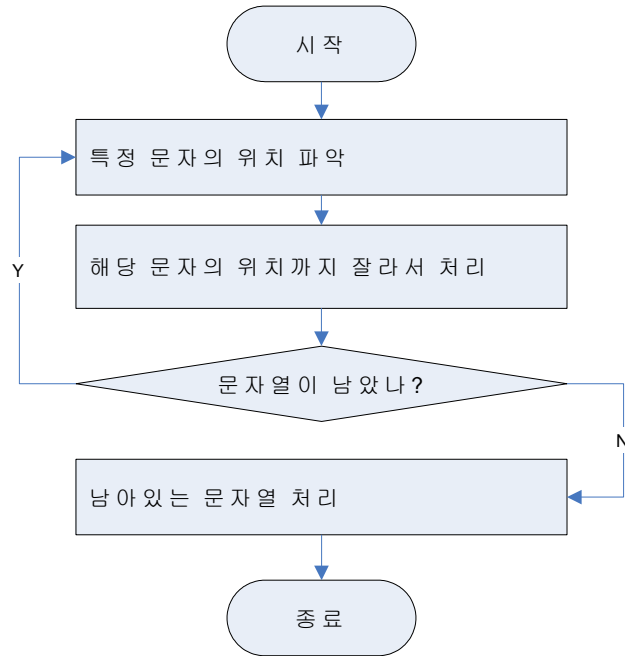
34 :   WriteLn('Result : ', sReverse);

35 :

36 :   ReadLn;

37 : end.
```

특정 문자로 문자열 자르기



[그림 95]

특정 문자로 문자열을 잘라서 사용하는 쉬운 방법 중에 하나는 [그림 95]처럼 원본 문자열을 해당 문자의 위치를 파악해서 그만큼씩 잘라내면서 처리하는 것입니다. 하지만, 이 방법의 문제는 원본 문자열의 앞부분을 잘라오면서 원본 문자열 내부의 문자들이 그만큼 지속적으로 이동해야 하기 때문에 효율이 너무 떨어진다는 것입니다.

[리스트 A1_11]은 스택을 활용해서 특정 문자가 나타날 때까지 버퍼에 쌓아두었다가, 특정 문자가 나타나면 이를 한꺼번에 처리하는 방식을 취하고 있습니다.

[리스트 A1_11]

```

1 : program A1_11;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses

```

```

6 :   SysUtils;

7 :

8 : var

9 :   Loop : integer;

10 :   sSource : string = '123|456|789';

11 :   StackSize : integer = 0;

12 :   Stack : array [1..100] of char;

13 :

14 : procedure Push(Value:char);

15 : begin

16 :   StackSize:= StackSize + 1;

17 :   Stack[StackSize]:= Value;

18 : end;

19 :

20 : function Pop:string;

21 : var

22 :   Loop : integer;

23 : begin

24 :   Result:= '';

25 :   for Loop := 1 to StackSize do

26 :     Result:= Result + Stack[Loop];

27 :   StackSize:= 0;

28 : end;

29 :

30 : begin

31 :   for Loop := 1 to Length(sSource) do

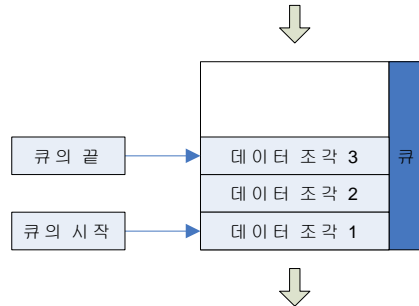
32 :     if sSource[Loop] <> '|' then Push(sSource[Loop])

```

```
33 :      else WriteLn(Pop);  
34 :  
35 :      if StackSize > 0 then WriteLn(Pop);  
36 :  
37 :      ReadLn;  
38 : end.
```


큐

큐에 대한 간략한 소개

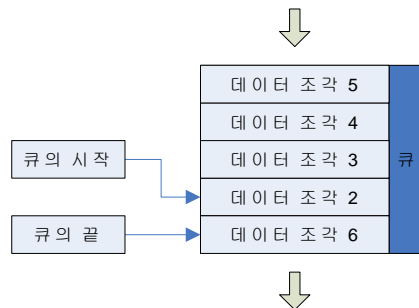


[그림 96]

큐는 스택과 달리 먼저 집어 넣은 데이터를 제일 먼저 꺼내는 방식을 취합니다. 순서대로 데이터가 입력되고, 순서대로 데이터가 빠져나갑니다. FIFO(First In First Out)

만약에 통신을 통해서 데이터가 들어오고 있지만, 처리 속도가 데이터 수신속도보다 느리다면 통신에 지장을 줄 수 있습니다. 이때는 큐를 이용해서 데이터를 쌓아두고, 먼저 들어온 데이터부터 순서대로 처리하는 방식을 취할 수 있습니다.

큐의 경우에는 앞에서부터 데이터를 빼내기 때문에, 큐의 크기가 한정적일 때 (메모리 공간이 한정적이니 무한대로 사용할 수만은 없습니다), 원형 큐를 사용합니다.



[그림 97] 원형 큐

[그림 97]는 “데이터 조각 1”을 사용하고 계속 데이터가 쌓인 경우를 도식화 하였습니다. “데이터 조각 5”가 들어오고 난 뒤, 더 이상 뒤로 공간이 없기 때문에, “데이터 조각 6”을 맨 앞으로 옮겨서 저장하는 방식을 취했습니다.

그리고, 큐의 시작과 큐의 마지막을 가리키는 포인터(또는, 위치를 저장한 정수)의 위치가 서로 반대로 되어 있는 것을 보실 수 있습니다. 즉, 큐는 2 번부터 시작해서 다섯 번째 칸을 지나면 다시 첫 번째로 복귀해서 다시 위로 증가하도록 하여, 시작과 끝이 반복되는 원형의 모양을 흉내 내고 있습니다.

예제

큐를 이용해서 문자열 데이터를 순서대로 입력하고, 다시 입력된 순서대로 문자열 데이터를 꺼내오는 예제를 [리스트 A1_12]에서 참고하시기 바랍니다.

[리스트 A1_12]

```
1 : program Sample;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : const
9 :   QueueSize = 5;
10 :
11 : var
12 :   Head : integer = 0;
13 :   Tail : integer = 0;
14 :   Queue : array [0..QueueSize-1] of string;
15 :
16 : procedure do_AddData;
17 : begin
18 :   if ((Tail+1) mod QueueSize) = Head then begin
19 :     WriteLn('* 큐가 가득 찼습니다. ');
20 :     ReadLn;
21 :   end else begin
```

```

22 :      Write('* 데이터를 입력하세요 : ');
23 :      ReadLn(Queue[Tail]);
24 :
25 :      Tail:= Tail + 1;
26 :      Tail:= Tail mod QueueSize;
27 :  end;
28 :
29 :  WriteLn;
30 : end;
31 :
32 : procedure do_GetData;
33 : begin
34 :   if Tail = Head then begin
35 :     WriteLn('* 큐가 비어있습니다. ');
36 :     ReadLn;
37 :   end else begin
38 :     WriteLn('* 큐의 첫번째 데이터 : ', Queue[Head]);
39 :     Head:= Head + 1;
40 :     Head:= Head mod QueueSize;
41 :   end;
42 :
43 :   WriteLn;
44 : end;
45 :
46 : procedure do_ListData;
47 : var
48 :   iCount, iIndex : integer;

```

```

49 : begin
50 :   WriteLn('* 큐에 있는 목록을 출력합니다. ');
51 :
52 :   iCount:= 1;
53 :   iIndex:= Head;
54 :   while ((iIndex+1) mod QueueSize) <> Head do begin
55 :     WriteLn('  ', iCount, ': ', Queue[iIndex]);
56 :
57 :     iIndex:= iIndex + 1;
58 :     iIndex:= iIndex mod QueueSize;
59 :   end;
60 :
61 :   WriteLn;
62 : end;
63 :
64 : function get_MenuNo:integer;
65 : begin
66 :   WriteLn('* 메뉴를 선택하세요. ');
67 :   WriteLn('  1. 데이터 입력하기, 2. 데이터 가져오기, 3. 데이터 목록
출력하기 ');
68 :   WriteLn('  0. 종료 ');
69 :   Write ('  번호를 입력하세요 : ');
70 :   ReadLn(Result);
71 :
72 :   WriteLn;
73 : end;
74 :

```

```

75 : procedure do_SelectMenu;
76 : var
77 :   iMenuNo : integer;
78 : begin
79 :   repeat
80 :     iMenuNo:= get_MenuNo;
81 :     case iMenuNo of
82 :       1 : do_AddData;
83 :       2 : do_GetData;
84 :       3 : do_ListData;
85 :       else iMenuNo:= 0;
86 :     end;
87 :   until iMenuNo = 0;
88 : end;
89 :
90 : begin
91 :   do_SelectMenu;
92 : end.

```

```
D:\₩강좌관련₩델파이 입문₩Part-3 알고리즘 입문₩Source₩리스트 19. 큐에 대한 다른 예제₩Sample.exe
* 메뉴를 선택하세요.
1. 데이터 입력하기, 2. 데이터 가져오기, 3. 데이터 목록 출력하기
0. 종료
번호를 입력하세요 : 1
* 데이터를 입력하세요 : 류종택
* 메뉴를 선택하세요.
1. 데이터 입력하기, 2. 데이터 가져오기, 3. 데이터 목록 출력하기
0. 종료
번호를 입력하세요 : 1
* 데이터를 입력하세요 : 이미정
* 메뉴를 선택하세요.
1. 데이터 입력하기, 2. 데이터 가져오기, 3. 데이터 목록 출력하기
0. 종료
번호를 입력하세요 : 2
* 큐의 첫번째 데이터 : 류종택
* 메뉴를 선택하세요.
1. 데이터 입력하기, 2. 데이터 가져오기, 3. 데이터 목록 출력하기
0. 종료
번호를 입력하세요 : 2
* 큐의 첫번째 데이터 : 이미정
```

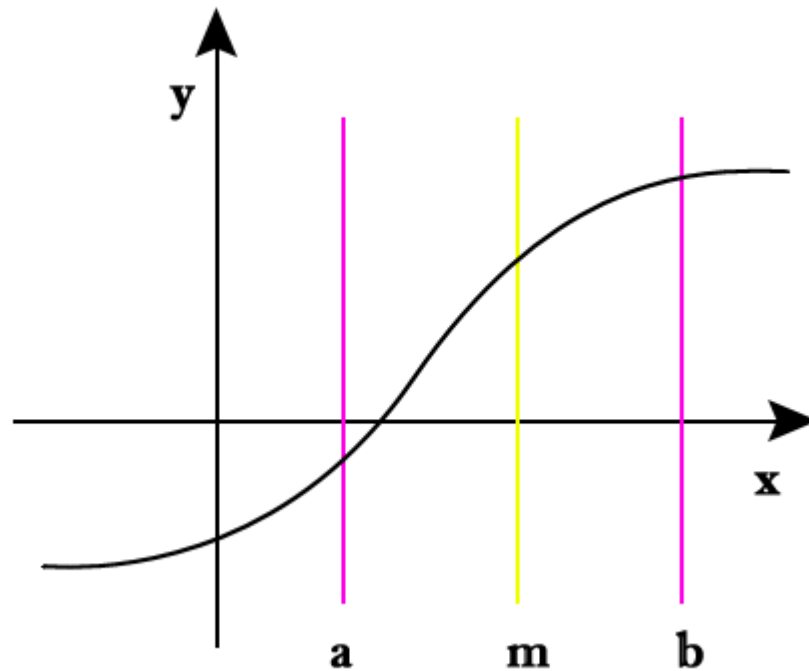
[그림 98] 리스트 AI_12 실행 결과

다차원 방정식의 해 구하기

$f(x)$ 의 해를 구하는 원리

우선 해를 구하려는 구간의 x 축의 범위를 지정해 줘야 합니다. 예를 들어 x 축의 시작 범위를 a 라고 하고, 끝나는 범위 값을 b 라고 하겠습니다.

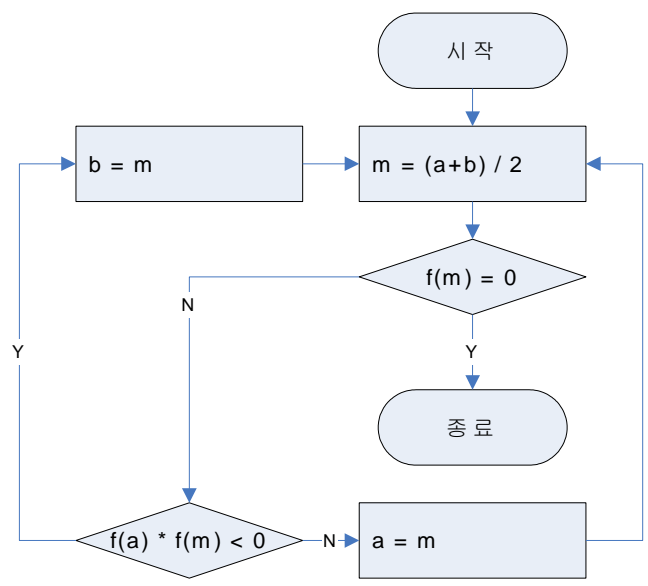
이때, 두 범위 안에 해가 있으려면 반드시 " $f(a) * f(b) < 0$ "을 만족해야 합니다. 이것을 기반으로 이분법을 통해서 해를 구하는 방법은 다음과 같습니다.



[그림 99]

[그림 99]와 같이 " $m = (a+b) / 2$ "와 같이 m 값을 취합니다. 그리고, " $f(a) * f(m) < 0$ "의 조건을 검사합니다. 만약, true 일 경우에는 중간에 해가 존재하는 경우이지만, " $f(m) * f(b)$ "와 같이 양수의 결과가 나온다면, m 과 b 사이에는 해가 없습니다.

이제 m 을 마지막 구간으로 설정하고, a 와 m 의 중간 값을 통해서 같은 작업을 반복합니다. 이렇게 반복하다 보면 언젠가는 “ $f(x) = 0$ ”을 만족하는 x 값에 가까운 값에 접근하게 됩니다. 이때, 허용되는 오차범위를 입력하여, 오차범위를 만족하면 반복을 멈추게 합니다. [그림 100]는 위의 과정을 플로우 차트로 표현한 것입니다.



[그림 100]

다차원 방정식의 해 구하기

[리스트 A1_13]

```
1 : program A1_13;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   a : double = 0.0;
10 :   b : double = 100.0;
11 :   m : double;
12 :
13 :   // 허용 오차
14 :   EPS : double = 1e-300;
15 :
16 : function f(x:double):double;
17 : begin
18 :   Result:= x*x - 4;
19 : end;
20 :
21 : begin
22 :   if f(a) * f(b) > 0 then begin
23 :     WriteLn(' 해가 존재하지 않습니다. ');
24 :     Exit;
```

```

25 :   end;

26 :

27 :   repeat

28 :       m:= (a + b) / 2;

29 :       if f(m) = 0 then Break;

30 :

31 :       if (f(a) * f(m)) < 0 then b:= m

32 :       else a:= m;

33 :   until abs(f(m)) < EPS;

34 :

35 :   WriteLn(Format('f(%f) = %f', [m, f(m)]));

36 :

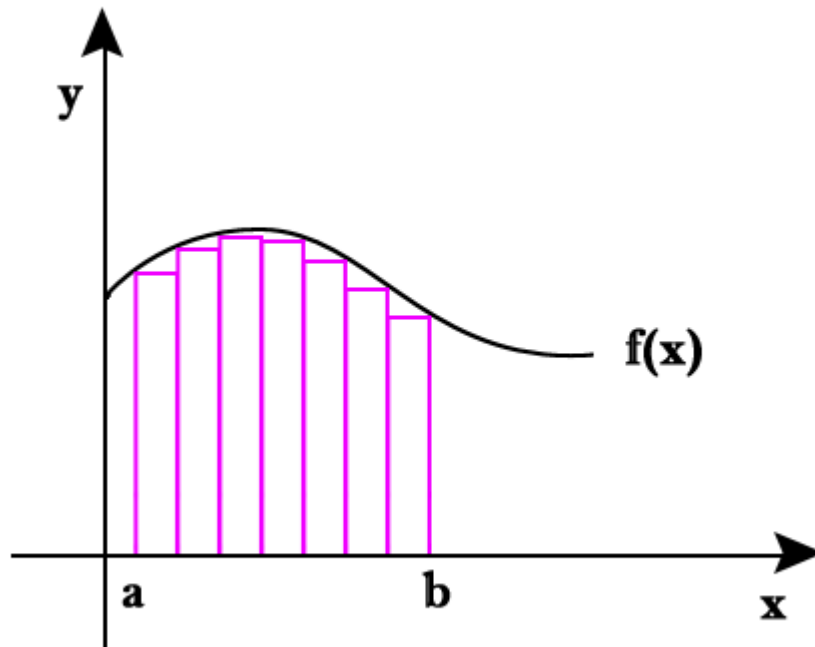
37 :   ReadLn;

38 : end.

```

적분

기본적인 적분의 원리



[그림 101]

적분의 단순한 방법 중 하나를 살펴보기 위해 [그림 101]과 같은 함수의 곡선을 예로 들어 보겠습니다. 구하고자 하는 면적은 a 에서 b 구간까지의 면적입니다.

이제 a 와 b 의 구간을 짧게 잘라내고 곡선에 맞춰서 직사각형을 그리도록 하겠습니다. 그러면 짧게 자른 두께를 dx 라고 할 때, 첫 번째 직사각형의 면적은 " $f(a) * dx$ "와 같습니다.

그 다음 직사각형의 면적도 같은 방식으로 구해서 전체를 합하면, [그림 101]의 함수가 그리는 곡선에 대한 면적을 구할 수 있습니다.

이때, dx 값이 작을수록 오차가 줄어들 것이며, 0 에 가까운 순간에 면적과 일치할 것입니다. 하지만, 컴퓨터로는 작은 수치를 입력하는 데에 한계가 있기 때문에 정확한 값을 구하기는 어렵습니다.

더구나 위의 직사각형과 곡선 사이에 공간 때문에 오차는 더욱 커지게 됩니다. 이에 대한 오차를 좀 더 줄일 수 있는 방법으로는 심슨 적분법 등이 있습니다.

간단한 적분에 대한 구현

[리스트 A1_14]

```
1 : program A1_14;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   a : double = 0.0;
10 :   b : double = 2.0;
11 :   dx : double = 0.001;
12 :   s : double = 0;
13 :
14 : function f(x:double):double;
15 : begin
16 :   Result:= x*x;
17 : end;
18 :
19 : begin
20 :   repeat
21 :     s:= s + f(a)*dx;
```

```
22 :      a:= a + dx;  
23 :      until a > b;  
24 :  
25 :      WriteLn('S = ', s);  
26 :  
27 :      ReadLn;  
28 : end.
```

재귀 호출에 대한 이해

자신이 자신을 호출하는 함수

재귀 호출은 자신이 자신을 호출하는 함수입니다. 우선 간단한 예를 들기 위해서 [리스트 A1_15]을 참고하여 설명하도록 하겠습니다.

[리스트 A1_15]

```
1 : program A1_15;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : function Factorial(n:integer):integer;
9 : begin
10 :   if n = 1 then Result:= 1
11 :   else Result:= n * Factorial(n-1);
12 : end;
13 :
14 : begin
15 :   WriteLn('3! = 1 x 2 x 3 = ', Factorial(3));
16 :
17 :   ReadLn;
18 : end.
```

8: - 12: 라인에 구현된 Factorial() 함수를 살펴보겠습니다.

11: 라인에 보면 함수 자신이 다시 자신을 호출하는 장면이 나옵니다. **Factorial()** 함수 내부에서 다시 자기 자신인 **Factorial()**를 호출하는 것입니다. 이렇게 되면 컴퓨터는 “n *”까지 계산하다가 말고 해당 상태를 스택에 저장합니다.

[리스트 A1_15]에서는 3!을 구하고 있기 때문에, “3 *”가 스택에 저장됩니다. 이제, 함수는 **Factorial(3-1)**을 호출하였고, 이제 n 값은 2로 호출이 되었기 때문에, “2 *”가 스택에 저장됩니다.

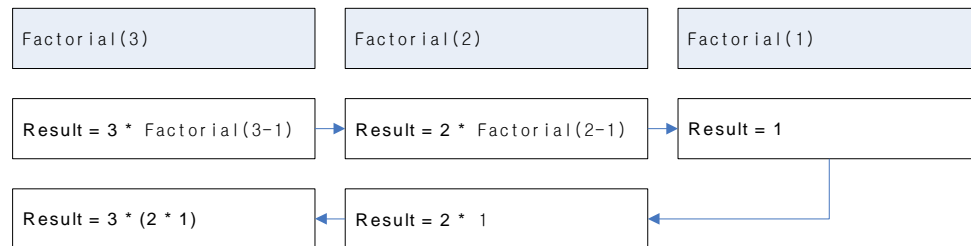
마지막으로 **Factorial(2-1)**이 호출되고 이는 10: 라인만 실행하여 1을 되돌려 줍니다. 그럼, 제일 마지막에 스택에 저장된 곳으로 돌아갑니다. 그리고 합쳐보면 “2*1”이 됩니다. 그 다음 스택에 저장된 곳으로 가서 합쳐봅니다. “3*2*1”이 되어서 우리가 구하고자 하는 3! 값과 같은 값이 출력될 것입니다.

재귀 호출을 주의 깊게 살펴보지 않으면 해당 코드가 단순히 10: 라인과 11: 라인을 반복하는 것처럼 보입니다.

하지만, 자기 자신을 호출할 때마다, 자신의 복사본을 만들고, 복사본을 움직이게 합니다. 그리고, 자신은 멈춥니다. 멈출 때 현재의 상태를 스택에 저장합니다. 만약 조건이 성립되지 않아서 자신이 복사해서 실행한 함수 호출이 또 다른 호출을 만들면 자신을 다시 스택에 저장해서 멈추고 다른 복사본을 실행합니다.

그러한 동작을 반복하다가, 조건이 성립되어서 반복이 멈추면 가장 최근에 호출된 함수의 복사본부터 결과값을 내놓으면서 차례로 사라져 갑니다.

재귀 호출은 일종의 분신술이라고 볼 수 있습니다.



[그림 102] 3! 결과가 나오는과정

함수 호출이라는 것은 현재의 위치의 정보를 스택에 저장했다가, 함수 실행이 종료되면 저장되었던 곳으로 다시 되돌아오도록 되어 있습니다.

3! 계산에서는 **Factorial(3)** 실행 중에 잠시 **Factorial(2)**로 이동하고 이후 **Factorial(1)** 이동했다가, **Factorial(1)**이 결과값 1을 내놓고 종료한 후 **Factorial(2)**의 **Factorial(1)** 호출하던 부분으로 되돌아와서 “2 * **Factorial(1)**”의 결과값, 즉, “2 * 1”을 결과값으로 내놓고 다시 종료합니다.

이후 **Factorial(3)**에서도 마찬가지로 방법을 통해서, 최종적으로 “3 * 2 * 1”의 결과를 내놓게 되는 것입니다.

하위 폴더 검색

재귀 호출이 자주 사용되는 경우 중 하나가 하위 폴더를 검색하는 곳입니다. 백신 등의 프로그램들이 전체 폴더를 찾아 다니면서 파일을 검사하려고 할 때 자주 사용되는 알고리즘입니다.

[리스트 Al_16]

```
1 : program Al_16;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils, SearchDir;
7 :
8 : procedure SearchDirectory(Path:String);
9 : var
10 :   Found : Integer;
11 :   SearchRec : TSearchRec;
12 : begin
13 :   if Copy(Path, Length(Path), 1) <> '\' then Path:= Path + '\';
14 :
15 :   WriteLn(Path);
16 :
17 :   Found:= FindFirst(Path+'*.*', faAnyFile, SearchRec);
18 :   while Found = 0 do begin
19 :     if (SearchRec.Name = '.' ) or (SearchRec.Name = '..') then
begin
20 :       Found:= FindNext(SearchRec);
21 :       Continue;
22 :     end;
23 :
24 :     if (SearchRec.Attr and faDirectory) = faDirectory then
25 :       SearchDirectory(Path+SearchRec.Name);
26 :
27 :     Found:= FindNext(SearchRec);
28 :   end;
29 :
30 :   FindClose(SearchRec);
31 : end;
32 :
33 : begin
34 :   SearchDirectory('C:\Program Files');
35 :
36 :   ReadLn;
37 : end.
```