프로그램밍(Programming)

C-언어를 하는 사람 중 프로그래밍이라는 단어를 이해 못하는 사람은 아마

없을 것이다. 그러나 먼저 프로그래밍 언어를 이해할 수 있도록 언어의 여

러가지 측면에 대해 알아 보자.

ⅵ@
새로운 용어

컴파일러(Compiler)	일시 번역기
인터프리터(Interpreter)	순차 번역기

ⅵ

1. 컴파일러와 인터프리터

우리가 프로그램을 작성하려고 한다면 우선 프로그램`소스(Source)`를

작성 해야한다. 이 작업은 PE2(Personal Editor II)나 Q(Quick Editor)

와 같은 에디터로 작성하거나 통합환경(IDE)에서 제공하는 Turbo-Series

의 컴파일러에서 작성해도 상관 없다.

어떻게 작성되었거나 소스파일은 우리가 도스의 `TYPE`명령으로 알아볼

수 있는 텍스트 파일의 형태로 되어 있다.

그런데 문제는 컴퓨터가 소스파일 자체로 프로그램을 실행할 수 없다.

소스파일은 사람이 알기 쉽게 작성한것이지 컴퓨터를 위한 것은 아니다.

소스파일을 컴퓨터가 알아들을 수 있도록 바꾸어 주는 것이 바로 `컴파`

`일러와 인터프리터`이다.

두가지로 나눈 이유는 소스파일을 컴퓨터가 알아듣게 하는데 방법상의

차이가 있기 때문이다. 먼저 컴파일러에 대해서 알아보기로 하자.

1.1 컴파일러(Compiler)란 ?

컴파일러는 소스파일을 컴퓨터가 바로 알아들을수 있도록 실행파일로 바

꾸어 주는 일을 한다. 여기서 실행파일이란 도스상에서 확장자가`*.EXE`

나 `*.COM` 으로된 파일을 말한다. 따라서 컴파일러로 작성된 프로그램은

컴퓨터가 바로 알아 들을 수 있는 형태로 변형되어 있으므로 속도가 빠르

고 거의 대부분의 언어가 여기에 속한다.

여러분이 앞으로 배우게 될 C-언어도 이 컴파일러의 부류에 속한다. 물론

뒤에 설명할 인터프리터로 되어 있는것도 있다. 하지만 여러분은 인터프리

터로 되어있는 C-언어를 볼수도 있을 것이다. 그러나 저자도 아직 이런 C-

컴파일러를 보지 못했다.

이외에 컴파일러로 되어있는 언어로는 PASCAL, FORTRAN, COBOL, LOGO, PR-

OLOG, ADA, MODULA2 등이다. 이들 언어에 대한 설명은 후에 있을것이다.

1.2 인터프리터(Interpreter)란 ?

사전을 찾아보면 알겠지만 인터프리터란`번역자`란 뜻이다. 인터프리터는 소스파일을 컴퓨터가 바로 알아들을 수 있는 형태로 변형하지 않고 한줄 한줄 말그대로 번역해서 컴퓨터에게 실행할 것을 요구한다. 따라서 프로그램을 실행할때 실제 프로그램 수행에 걸리는 시간 이외에 번역하는 시간이 필요하게 된다.

이렇게 됨으로 해서 프로그램이 느려짐은 물론 비효율적이게 된다. 게다가 인터프리터에서 작성한 프로그램은 그 인터프리터가 없이는 실행할 수 없게된다. 다른 프로그램들 처럼 도스상에서 단독으로 실행할 수 없다는 것이다.

이 인터프리터로 되어있는 언어로는 몇년전 까지만 하더라도 컴퓨터학원

에서 많이 가르치던`베이직(BASIC)을`들 수 있다. 하지만 이 베이직도 점차 컴파일러로 바뀌고 있는 추세여서 머지않아 인터프리터로 되어있는 언어는 찾아 보기 힘들것이다.

2. 구조적 프로그래밍(Structured Programming)

무엇이든 간에 잘 정리 되어있지 않은것은 알아보기 힘들다. 그림도 어지럽게 그려져있는 추상화 보다는 잘정리된 정물화가 이해하기 쉬운것과 같은 이치이다. 프로그램밍에서도 이런 사실이 그대로 적용된다.
구조적으로 안정적이기 위해서는 같은 역활을 하는 부분(Module)을 모아 하나로 묶어 관리하여야 한다. 이와같이 구조적으로 안정된 프로그래밍을 구조적 프로그램이라고 한다. C-언어도 이런 구조적 프로그래밍을 가장 잘 구현할 수 있는 언어이다. 그러나 C-언어를 사용한다고 해서 꼭 구조적 프로그래밍을 한다고는 볼 수 없다. C-언어로 만들어진 프로그램 중에서도 알아보기 힘든 복잡한 프로그램을 가끔 볼 수 있다.

프로그램은 무엇보다`알기쉽게 작성해야 한다.` 복잡한 프로그램이 좋은

프로그램일까 ? 우선 복잡한 프로그램은 프로그램을 작성한 본인도 알아 보기 힘들다. 프로그램을 작성하면서 만든 유용한 부분들을 다른 프로그램에 접목 시키려할 때에도 꽤 힘들게 될 것이다.

빠꿔 말해 프로그램의`유지,보수`가 힘들게 된다는 것이다.

3. 프로그래밍 언어의 역사

프로그램을 만들기 위한 언어는 최초의 컴퓨터인 애니악이 만들어질 때부터 있었다. 물론 이때는 컴파일러나 인터프리터가 없었으므로 기계가 곧바로 알아들을 수 있는`기계어(Machine Language)`를 사용하였다.

그리고 컴퓨터가 전쟁에 쓰이기 시작하였다. 모든 문화와 과학문명은 전쟁 이후에 발달한다. 이유는 적을 보다 손쉽게 많이 죽이기 위해 과학기술이 필요하기 때문이다.

이때는 대형 컴퓨터로 로켓을 발사하고 정확하게 목표물을 맞추기 위해서 컴퓨터가 사용되었는데 따라서 사용되는 언어도 수학계산을 손쉽게 할 수 있는 언어가 필요하게 되었다. 물론 그당시에도 어셈블리는 있었다. 그러나 어셈블리를 약간 이해하고 있는 사람은 알 수 있겠지만 어셈블리 언어로 정확한 계산을 하는 프로그램을 만드는것은 쉬운일이 아니다.

이래서`포트란(Fotran)`이 만들어졌다.   이런 이유로 지금도 공학계산용

으로 대학이나 연구소에서는 포트란을 사용하는 곳을 볼 수 있다.

전쟁이 끝나면 산업과 문명이 발달한다. 수많은 회사들이 생겨나고 각 회

사들 마다 처리해야할 업무량이 폭주하게 되었다. 이런 분야에 포트란이

사용될 수 있을까 ? 물론 업무용 프로그램에도 계산은 필요하다. 그러나

과학계산용 언어를 업무용에 사용하기에는 무리가 따랗다.

그래서 업무를 위한 프로그래밍 언어인 `코볼(COBOL)`이 나왔다.

이때부터 공학용 에는 포트란, 업무용에는 코볼이라는 등식이 만들어졌다.

이때에 사용되던 컴퓨터는 지금 우리가

쓰고 있는 그런 컴퓨터가 아니었다. 가

끔 TV화면에서 볼 수 있는 커다란 컴퓨

터(대형 컴퓨터)인 것이다.

illu0000.dat

그러다가 여러분도 알고 있는 컴퓨터의 천재 스티브잡스(현 NeXT사 사장)

가 개인용 컴퓨터인 APPLE을 만들어 냈다. 이런 이야기는 알고있는 사람

은 모두 알고 있을것이다.

지금으로서는 우스꽝스런 모양에 엄청나게 느린 8비트 컴퓨터 였지만 당시로서는 놀라운 일이었고 이 컴퓨터는 날개 돋힌듯이 팔려나갔다. APPLE 컴퓨터 에서도 포트란과 코볼이 개발 되었다. 그러나 포트란과 코볼은 APPLE 컴퓨터에서 사용되기에는 역부족 이었다. 너무 속도가 느린것이었다. 때문에 APPLE 컴퓨터용 으로 만들어진 대부분의 프로그램들은 어셈블리로 만들어졌다.

이쯤에 여러분들도 모두 아는 빌게이츠가 대형 컴퓨터의 베이직(BASIC)을 개인용 컴퓨터에서 돌아갈 수 있게끔 만든다. 이 베이직이 초보자 들에게 인기가 있자 APPLE이외에 다른 컴퓨터를 만들던 회사에서도 자기들의 컴퓨터에 BASIC을 내장하게 되었다.

내장이라는 표현이 이상하게 들릴지 모르지만 당시에는 베이직이 디스켓으로 공급된것이 아니고 컴퓨터를 키면 바로 베이직이 실행되는 (ROM 에 넣어진 형태) 그런 컴퓨터를 사용하였다.

APPLE사의 성공을 지켜본 업무용 컴퓨터를 만들던 IBM에서 IBM-PC를 발표한다. 이 IBM-PC에서 가장 많이 사용되는 언어는 어셈블리와 베이직 이었다. 베이직은 주로 초보자들 사이에서 배우기 쉬운 언어로 자리 잡았고 어셈블리는 일반 프로그램(어플리케이션)을 만드는데 사용되었다.

불과 5년쯤 전만 하더라도 IBM-PC용으로 발표되는 거의 모든 프로그램은 어셈블리로 작성된 것이다. 참으로 경이롭고 존경스러운 선배님(?)들이 아닐수 없다.

사실 C-언어는 20년의 역사를 가지는 아직은 어린(?) 언어이다. 40,50년 된 코볼이나 포트란에 비하면 말이다. 그런데 C-언어가 요즘에 와서 인기를 끄는데는 이유가 있다.
바로 속도 때문이었다. XT나 초창기AT 에서는 C-언어로 짜여진 프로그램은 느릴 수밖에 없다. 물론 어셈블리로 작성하더라도 느릴수가 있지만 같은 실력으로 프로그램을 작성한다면 어셈블리의 속도를 따라잡을수 는 없는 것이다.

그러나 이제는 많이 달라졌다. C-언어로 작성하여도 요즘의 386이나 486 기종에서는 충분히 빠르게 동작한다. 물론 AT에서도 제대로 사용할 수 있다. 이제는 엄청나게 쏟아져 나오는 프로그램의 알고리즘과 이론을 C-언어를 사용하여 알기쉽고 효율적으로 작성할 수있는 프로그래머가 진짜 능력있는 프로그래머 이다.

4. 파스칼에 대하여

C-언어를 공부하기 전에 파스칼을 배운 사람이 있을것이다. 이런 사람을 위해서 파스칼과 C-언어의 다른점과 그리고 비슷한 점을 알아보도록 한다

ㅎl@

[예] 파스칼과 C-언어

```
[C-언어]                              [파스칼]
main()                                program pascal;
{                                     begin
  int i,j,k;                        i,j,k : integer;
  for (i=0;i<=9;i++)                for i=0 to 9 do
  {                                   begin
    printf("%d",i);                   write(i);
    if (kbhit()) exit(0);           if keypressed then halt(0);
  }                                   end;
}                                     end.
```

파스칼과 C-언어는 원래는 같은 조상에서 나온 언어이다. 때문에 두 언어는 서로 비슷한 점이 많다. 그 예는 위를 보면 알수 있다.

파스칼을 배운 사람은 이미 C-언어를 반쯤은 알고 있는셈이 된다. 단지 파스칼에도 파스칼의 규칙이 있듯이 C-언어의 규칙을 익히면 되는것이다. 파스칼은 본래 교육용으로 만들어진 언어이다. 교육용 으로 만들어지다 보니 그 구조가 탄탄할 수 밖에 없으며 또 지나치다 할 만큼 안정적이다. 즉, 예외적인 상황이 발생되어 에러가 나는 확률이 비교적 적다는 뜻이된다. 이러한 것은 파스칼이 그만큼 표현의 자유를 박탈하기 때문에 가능하다. 저자도 C-언어를 배우기 전에 약 4년간 파스칼을 사용하였는데 이러

한 것이 될까 ? 하는 루틴은 파스칼에서는 되지않는다.

아예 컴파일러가 허용하지 않는다는 뜻이다. 물론 이러한 것이 프로그램 실행중에 예외적인 상황을 막기 위해서라는 취지는 좋다.

그러나 프로그래머 에게는 보다 자유로운 표현이 필요하다. C-는 바로 이

런 것을 충족 시켜주는 언어이다.   C-언어 보다 표현이 자유로운 언어는 아직없다. 물론 C-언어 보다 뛰어난 기능을 가진 언어가 없다는 뜻은 아니다.

그러나 가장많이 사용되고 있는데는 무슨 이유가 있는것이 아닐까 ? C-언어가 다른언어에 비해 뛰어난 점을 들라면 6가지 정도를 들 수가 있다.

4.1 다양한 연산자

C-언어의 가장 대표적인 특징으로는 다양한 연산자의 사용을 들 수 있다.
실제로 C-언어에서는 다른 언어에 비해 월등하게 많은 연산자를 사용하는데,
이에의해 C-언어는 다른 언어에서 몇행에 걸쳐 기술해야 하는 내용을 한 두행으로 간단히 기술할 수 있다.

illu0001.dat

바로 이러한 면이 초보자에게는 혼동의 대상이 되기도 한다. 그러나 일단 한번익숙해 지면 C-언어의 편리함에 다시한번 놀라는 부분이 바로 연산자에 관한 부분이다.

4.2 명쾌한 구조

C-언어는 고급언어와 저급언어의 양면성을 동시에 갖추고 있는데 고급언어로서의 대표적인 특징은 명쾌한 구조에 있다. C-언어에서 프로그램의 흐름을 제어할때는 if, while, for, case...등의 제어문을 사용한다. 이들 제어문의 기능은 베이직이나 파스칼등의 고급언어와 기본적으로 동일하다.

그런데도 저급언어로서의 기능도 동시에 발휘하는 것이 바로 C-언어이고 바로 이러한 점이 C-언어의 오묘한 점이기도 하다.

이러한 제어문을 사용하는 C-언어로 작성한 프로그램은 베이직이나 어셈블리어 프로그램 처럼 산만하지 않고, 산뜻하면서도 간결하다. 그런면에서는 파스칼과 비슷하다 할 수 있다.

그러나 파스칼과 다른점은 파스칼 보다는 간결하다는데 있다. 이는 위의 파스칼과 C-언어의 예제를 보면 알 수 있다.

4.3 함수에 의한 구성

C-언어에서는 특정 기능을 수행하는 단위는 모두 함수로 분류된다. 즉 C-언어에서의 함수는 일반 언어에서의 명령(Command)에 해당한다고 볼수 있다.

그러나 C-언어에서의 함수는 일반 언어의 명령보다는 한결 그 폭이 넓다.

C-언어의 프로그램은 함수를 단위로 하여 작성된다. 모든 프로그램은 최소한 한개 이상의 함수 (main()함수)로 구성되며, 프로그램의 각 부분은 함수의 형태로 분할하여 기술된다. C-언어에서는 이와같이 함수를 단위로 하여 프로그램을 작성함으로써 프로그램 전체를 논리화, 구조화 시킬 수 있다.

즉, 복잡한 프로그램도 각 기능별로 별도의 함수를 작성한 후, 복수개의 함수를 하나로 묶어줌으로써 전체 프로그램을 논리적으로 명쾌하게 작성해 나갈 수 있는것이다.

함수에는 C-언어가 자체적으로 제공하는`표준함수`와 사용자가 정의하여 사용하는`사용자 정의 함수`의 두가지가 있는데, 이중에서 특히 표준함수는 완성된 형태로 제공되기 때문에, 프로그래머는 별도의 작업없이 이를

직접 이용할 수 있다.

즉 여타 프로그래밍 언어의 명령을 사용하는것 처럼 이용할 수 있다.

이들 표준 함수의 수와 종류는 컴파일러에 따라 제각기 다른데, 어느 컴

파일러나 단른 어떤

프로그래밍 언어의 명령보다는 훨씬 다양하고 풍부

한 표준함수를 제공하여 주고 있다.

이는 프로그래밍의 작업이 보다 손쉬울 수도 있음을 의미한다.

4.4 간결하고 일관된 데이타 처리

C-언어에서 사용되는 데이타형(Data Type) 에는 문자형, 정수형, 실수형

등이 있고 이들 기본 데이타형의 합성에 의해, 구조체(Struct), 공용체(

Union)등의 복합 데이타형이 정의되어 사용되는데 어느 데이타형에 있어

서나 그 처리방식에서는 일관성이 있으며 상호 유기적으로 사용될 수

있다.

예들 들어 문자형 데이타의 경우, 이를 정수형처럼 생각하여, 각종

산술처리적인 방식을 통해 문자형 데이타를 처리할 수 있다.

4.5 동적이며 능동적인 데이타 관리

C-언어에서는 몇가지 경우를 제외하고는 메모리를 항상`동적(Dynamic)`으

로 관리한다. 즉, 메모리를 필요시에만 할당하며, 그렇지 않은 경우에는

메모리를 할당하지 않는다. 이와같이 함으로써 메모리관리 효율을 높이고

있다.

또`포인터(Pointer)`라는 개념을 도입하여 메모리를 능동적으로 관리하고

있다. 이 포인터는 C-언어의 대표적인 특징중에 하나인데, C-언어는 포인

터를 사용함으로써, 메모리를 보다 적극적으로 관리하며, 동시에 복합 데

이타형의 처리효율도 높이고 있다.

4.6 높은 이식성

C-언어는 비교적 저급언어의 특징을 띄고 있으면서도, 프로그램의 이식성

은 다른 고급언어 보다 오히려 뛰어나다. 즉, 프로그램을 다른 기종의 컴

퓨터용 으로 변환하기가 용이하다.

보통 저급언어 (어셈블리등)로 작성된

프로그램은 작성된 시점에서 사용된 기

종의 하드웨어와 밀착되어 있기 때문에

다른 기종으로 이식하는것은 거의 불가

능에 가깝다

illu0002.dat

그러나 C-언어로 작성된 프로그램은 프로그램을 구성하고 있는 함수중

하드웨어에 의존하는 함수만을 대상으로 비교적 용이하여 변환작업을 수

행할 수 있다.

이와같은 이유로 시스템 소프트웨어 개발자들이 C-언어에 관심을 가지고

있는것이고 실제로 C-언어로 시스템 소프트웨어를 작성하고 있는 것이다.
C-언어의 개요

이장에서는 C-언어로 프로그래밍 하는데에 필요한 사항을 배운다. 우선

컴파일러 의 사용법과 링커의 사용법 그리고 적당한 에디터 고르법,

간단한 예제 프로그램과 C-언어의 구성요소들을 배우게 된다. 실제로 C-

언어를 배우는 첫걸음이니 주의하여 살펴보기 바란다.

1. 프로그램을 만드는 과정

[1] 원시 프로그램 작성

프로그램을 만들기 위해서는 먼저 원시 프로그램(Source)를 만든다. 이과정은 에디터를 통하여 이루어 지는데 에디터는 여러분이 구하기 쉬운 것을 구하여 사용하기 바란다. 시중에 나와있는 에디터로는 PE2(PE3가 나왔다는 말이 있음)와 Q-에디터(Q.EXE)가 있다. 그외에도`워드스타 WordStar`나 edlin을 사용하는 사용자가 있을것이다.

사실 edlin은 도스에서 제공하던 라인에디터인데 사용이 불편해서 지금은 거의 사용되지 않고 가장 많이 사용되고 있는것은 PE2와 Q-에디터로 압축된다.

PE2는 IBM사에서 만든 프로그램이고, Q-에디터는 쉐어웨어 인데 사용하여보고 마음에들면 돈을 지불하는 방식이다. 만일 아직 에디터를 한번도 사용 해보지 않았다면 Q-에디터를 권한다. 사용이 간편하고, 무었보다 빠르고 저자도 이 프로그램을 사용하고 있다.

만일 이 프로그램을 가지고 있지 않다면 가까운 친구나 통신을 통해 구하기 바란다.

[2] 컴파일(Compile)

컴파일을 하기 위해서는 컴파일러가 필요하다.

가장 많이 사용되고 있는 컴파일러는 뭐니뭐니 해도 볼랜드사(BORLAND)의 Turbo-C 2.0과 Borland C++ 3.1이 있다. 이 둘중에 하나는 가지고 있어야 한다. 만일 가지고 있지 않다면 공부를 계속할 수 없다. 컴파일러를 가지고 있지 않다면 복사를 통해 구하기 보다는 정품을 구입하기 바란다.

어쨌든 컴파일이라는 것은 에디터로 작성된 확장자가 *.C인 파일을 *.OBJ 파일로 바꾸어 주는 것을 말한다.

이 과정에서 에러가 발생하면 컴파일러는 에러가 발생한 라인번호와 에러

의 종류를 알려준다. 그러면 다시 에디터를 사용하여 에러를 수정하고 컴파일을 계속하면 된다.

여기서 터보파스칼이나 터보베이직을 사용하던 사용자는 궁금증을 가질지 모른다. 터보파스칼이나 베이직 컴파일러는 *.OBJ 파일을 거치지 않고 곧바로 실행파일(*.EXE)을 만드는 것이다.

터보-C에서는 *.OBJ 파일을 *.EXE 파일로 만드는데`링크(Link)`라는 작업을 거치는데 이작업이 귀찮게 느껴질 수 있다.

그러나 이런 작업을 하는데는 이유가 있다. *.OBJ 파일은 다른 컴파일러에서도 공통적으로 사용하는 중간 파일이다. 즉 터보-C가 *.OBJ파일을 사용 함으로써 다른 언어들과 자유롭게 코드를 교환할 수 있는 것이다.

예를 들어 C-언어에서 만들어진 *.OBJ 파일은 파스칼에서도 사용될 수 있다. 그러나 파스칼에서 생성된 *.TPU 파일은 C-언어 에서는 사용될 수 없는 것이다.

[3] 링크(Link)

위에서 설명 했듯이 *.OBJ 파일을 *.EXE 파일로 변환하는 과정이다.

이 과정은 단순히 변환의 과정 이외의

다른 의미도 포함하는데 링크(사슬,연

계) 라는 의미가 말해주듯, 여러개의

*.OBJ 파일을 하나로 묶어 *.EXE 파일

로 만드는 것도 포함한다.

이과정은 TLINK.EXE파일로 하며 이 파

일은 컴파일러를 구입하면 포함되어

있다. ᆩillu0100.dat

2. 간단한 C-언어 프로그램(1)

아래는 가장 간단한 C-언어 프로그램으로서 화면에 "Hello everybody !"

를 출력하는 프로그램이다.

ᄚlsam0100.dat

```
[예제] 문장출력
#include <stdio>                          /* 헤더파일 삽입 */
main()
{
    printf("%s","Hello everybody");    /* 원하는 문장 출력 */
}
```

C-언어는 함수로 구성된 언어라고 하였다. 실제로 위의 프로그램을 보면

main()이라는 함수로 프로그램이 시작된것을 볼 수 있다. 프로그램의 첫

부분은 항상 main()함수이다.

```
#include <stdio.h>
```

include라는 말은 포함한다는 뜻이다. C-언어에서는 자체에 내장된 함수

를 사용하기 위해서 함수의 모양이 정의된 헤더파일이 필요하다.

include는 이러한 헤더파일을 포함 시켜주는 명령이다.

stdio.h는 C-언어의 기존 파일들을 제어하는 함수의 모양이 정의되어 있

다.

[참고] 이러한 함수의 모양을 프로토타입 이라고 하며 이제부터 이 용
어를 사용 하겠다.

이외에도 헤더파일은 여러개가 있으며 필요할 때마다 설명하겠다.

```
printf("%s","Hello everybody");
```

printf명령은 화면에 문자열을 표시 하여주는 함수로서 베이직의 "PRINT"

나 파스칼의 "write"함수와 비슷한 역할을 한다.

"%s"라고 표시된것은 뒤의 자료를 문자열의 형태로 화면에 표시하라는 뜻

이며 이와 같은것을`제어문자열`이라 한다. 제어문자열에는 이 밖에도 아

래와 같은 것들이 있다.

ㅎI
d...........10진수(Decimal)로 표시
f...........10진수 실수(float)로 표
x...........16진수(Hex)로 표시
c...........단일문자(Character)로 � s...........문자열(String)로 표시
ㅎ

3. 간단한 C-언어 프로그램(2)

다음은 C-언어 개발자인 Kernighan/Ritchie의 저서 [C-Programming Langu

age]에 가장 최초로 나오는 프로그램 이다. 이는 화씨 온도를 섭씨 온도

로 변환하여 출력하는 프로그램이다.
ㅎIsam0101.dat
[예제] 온도 단위변환

```
main()
{
  int     lower, upper, step;
  float   fahr, celsius;

  lower = 0;      /* 온도의 최소값 */
  upper = 300;    /* 온도의 최대값 */
  step   = 20;     /* 말 그대로 스텝 */
  fahr = lower;
```
ᄛ

ᄛIsam0101.dat

```
  while(fahr <= upper)
  {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%4.0f  %6.1f\n",fahr,celsius);
    fahr = fahr + step;
  }
}
[결과]
  0     -17.8
 20      -6.7
 40       4.4
 60      15.6
 --------------- 후략 ----------------
```
ᄛ

C-언어를 처음대하는 사람은 위의 예제가 매우 어렵게 느껴질것이다. 그러나 걱정할 필요는 없다. 위의 예제는 단지 C-언어로 짜여진 프로그램의 구성요소를 알아보기 위해 예를 든 것이므로...

```
তl
        lower = 0;        /* 온도의 최소값 */
        upper = 300;    /* 온도의 최대값 */
        step   = 20;      /* 말 그대로 스텝 */
ত
```

(1) 주석(Comment)

프로그램중에`/* 와 */`사이에 놓여진 부분은 주석으로 간주되어, 컴파일 시 컴파일 대상에서 제외된다. 그러면 컴파일도 되지 않는 내용을 왜 굳이 힘들게 타이핑하여 놓는것일까 ?

프로그램이 짧을때는 별 문제가 없지만 프로그램이 길어지게 되면 자신이 작성해 놓은 부분을 자신도 알아보지 못하는 경우가 생긴다. 이럴때를 대비 하여 프로그램의 부분 부분에 주석을 달아 놓으면, 나중에 프로그램을 수정하는 데에도 무척 편리하다.

여러분도 프로그램에 꼭 주석을 다는 습관을 들이도록 하자.

```
{
      int      lower, upper, step;
      float   fahr, celsius;
}
```

(2) 변수 선언문

파스칼을 사용해본 사용자는 위의 내용이 이해가 가겠지만 베이직만을 사용해본 사용자는 도무지 이해가 가지 않을것이다. 베이직에서는 변수를 사용할때 굳이 변수를 선언할 필요가 없다. 그러나 C-언어나 파스칼등과 같은 언어에서는 프로그램내에서 사용할 변수를 미리 선언해 주어야 한다 이러한 과정이 귀찮게 느껴질 수도 있으나 분명히 이유는 있다. 첫째 변수의 형을 확실히 지정해 줌으로써 프로그램상의 혼돈을 막고, 둘째 변수로 사용되는 영역을 미리 메모리 상에 확보하기 위해서 이고, 셋째 프로그램의 구조를 보다 명확하게 하기 위해서이다.

어렵게 설명했지만 다시말해 C-언어에서는 사용할 변수를 미리 선언해 주어야 한다.

위에서`int`는`단정도 정수형`으로 -32768 ~ 32767 사이의 값을 가진다.

int는 C-언어 프로그램에서 가장 많이 쓰이는 데이타형 이기도 하다.

`float`는`단정도 실수형`으로 실수형이란 간단히 말해 소숫점이 있는 수

이다. 이런 데이타형에 대한 자세한 내용은 다음장인"데이타"단원에서 자

세히 다룬다.

지금까지 보아서 알 수 있겠지만 선언문의 끝에는 반드시`;`를 적어야 한

다. 이는 비단 선언문 뿐만이 아니고 대입문이나 수행문의 경우에도 마찬

가지 이다.

ㅎI
```
   lower = 0;       /* 온도의 최소값 */
   upper = 300;    /* 온도의 최대값 */
   step  = 20;      /* 말 그대로 스텝 */
```
ㅎ

(3) 대입문

대입문은 변수에 값을 넣는 과정이다. 위의 예는 3라인에 걸쳐서 표현한

예이다. 다음과 같이 한라인에 해줄 수 도 있다.

ㅎI
```
   lower = 0;  upper = 300;  step  = 20;
```
ㅎ

둘중에 어떠한 표현을 쓰는가 하는 것은 프로그래머의 자유이다. 그리고

어떻게 표현해주든 달라지는것은 없다.   변수의 초기치는 변수의 선언과

동시에 해줄 수도 있다. 아래의 예를 보자

ㅎI
```
   int lower = 0,upper = 300,step  = 20;
```
ㅎ

변수의 선언과 동시에 초기값을 넣어주면 프로그램의 구조가 훨씬 명확하

고 간단히 진다. 대입문의 경우에도 끝에`;`를 붙였음을 확인하자.

(4) while루프

프로그램의 핵심이 되는 부분은 다음의 while루프이다.

ㅎl

```
while(fahr <= upper)
{
   celsius = (5.0/9.0) * (fahr-32.0);
   printf("%4.0f  %6.1f\n",fahr,celsius);
   fahr = fahr + step;
}
```

ㅎ

이는 변수 fahr의 값이 변수 upper의 값보다 작은한, 해당 루프를 반복하

여 실행한다. 프로그램 실행 초기에는 fahr의 값은 0, upper의 값은 300

으로 [fahr <= upper] 성립되므로 while루프의 각 수행문을 순서대로

실행한다.

우선 연산식 [celsius = (5.0/9.0) * (fahr-32.0);]으로 수행되어 화씨온

도 fahr에 따른 섭씨온도 celsius의 값을 계산한다. 여기서 수치를 5.0,

9.0, 32.0 과 같이 소숫점까지 지정하여

사용한 이유는, 이들을 실수로사용하기

위해서 이다.

이어 printf()함수는 특정 양식에 의거

fahr, celsius의 값을 표시한다.

이 printf()함수는 C-언어의 정의되어

있는 함수로 별다른 조치 없이 사용할

ᆩillu0101.dat

수 있다.

while루프 마지막의 [fahr=fahr+step;] 는 fahr의 값을 step의 값만큼

(20) 증가 시킨다.

4. C-언어의 데이타형

위에서 C-언어 프로그램 내에서 사용되는 자료는 미리 프로그램의 윗부분

에서 선언되어야 한다고 배웠다. 그러면 C-언어에서 사용될 수 있는 자료

형에는 어떤것이 있을까 ?

ᇙ

```
  자료형       허용되는값

  char         -128          ~ 127                  |
  int          -32768        ~ 32767                | 정수형
  long         -2147483648 ~ 2147483647             |
```

float　　단정도 실수　　　　　　　　| 실수형
double　　배정도 실수　　　　　　　　|
ㄷ

허용되는값 이라는 것은 각 자료형이 가질 수 있는 값의 범위를 말한다.

예를들어 char는 -128 ~ 127 사이의 값을 가지므로 1000이라는 값은 가질

수가 없다.

만약에 char로 선언된 데이타에 1000을 대입하면 에러가 발생할 수 밖에

없다. 이러한 에러를`오버플로우(Overflow)`라고 한다.

(1) char

char 형은 값의 범위가 적은 계산이나 문자하나를 다루기 위해 사용된다.

문자는 1~128 사이의 값을 가지므로 char형으로 충분이 표현할 수가 있다

(2) int

int 형은 일반 계산에 사용된다.

값의 허용 범위가 비교적 넓으므로(-32768 ~ 32767) 일반 계산에는 무리

없이 사용될 수 있다. 실제로 가장 많이 사용되는 데이타 형이다.

(3) long

만약에 게임 프로그램을 만든다고 해보자. 게임의 점수를 변수로 사용하

고자 할때 int형으로는 무리가 따른다. int의 범위는 -32768 ~ 32767 이

므로 10만점이나 100만점등의 숫자를 표시할 수 가 없다. 이럴때는 long

형 정수를 사용하면 된다. long형은 -2147483648 ~ 2147483647 의 값을 가지므로 표현에 제한이 없다.

(4) float,double

지금까지는 정수형 데이타에 관해서만 공부하였다. 실제로 정수형 데이타만 가지고도 왠만한 프로그램을 작성할 수 는 있다. 그러나 성적의 평균을 구한다던가, 실수형의 계산을 하여야 할 때는 실수형 데이타가 필요하다. 이럴때는 float나 double을 사용하면 된다.

float는 double보다 정밀도가 낮은 대신 메모리를 적게 차지한다. 그러나 지금까지는 굳이 실수를 사용할 필요가 없으므로 필요할때에 자세히 알아보도록 하자.

(5) 문자열형

파스칼에서는 string 이라는 형명칭으로 문자열형 데이타를 제공한다. 그러나 C-언어에서는 문자열형 이라는 형이 존재하지 않는다.

대신 문자형(char)을 배열로 정의하여 문자열로 사용한다. 이 내용은 상당히 중요하다.

예를들어 아래와 같은 표현은

`char name[13];`

name이라는 변수에 char형 데이타를 13개의 배열로 정의했다.

따라서 name에는 영문으로는 13자가 들어갈 수 있을까 ? 아니다. 영문으

로는 12자가 들어간다. 그 이유는 아래를 보면서 알아보자.

실제로 아래와 같이 정의했을 경우를 알아보자.

`char name[13] = "choi min suk";

문자열의 끝에는 문자열의 끝임을 알리는 0이(아스키 코드 0) 들어가므로

12자를 정의하여 사용하려면 문자열은 13개를 정의하여 사용하여야 한다.

"oh ki wook"이라는 문자열이 메모리에 저장되는 모습은 다음과 같다.

```
o  h     k  i     w  o  o  k  0
1  2  3  4  5  6  7  8  9  10 11
```

sam0102.dat

```
[예제] 문자열의 정의, 초기화
main()
{
  char st[12]="I am a boy";/* 11자 이므로 12개로 정의한다 */
                           /* 11자 뒤에는 자동으로 0이 붙는다. */
  printf(st);              /* st를 화면에 표시한다 */
}
[결과]
I am a boy
```

위의 예제는 문자열을 정의함과 동시에 일정한 값으로 초기화 함을 보여

준다.

여기서 한가지 중요한 것이 있다. 문자열은 정의와 동시에 초기화하면 굳

이 문자열의 길이를 명시할 필요가 없다는 것이다.

```
(1) char name[11] = "oh ki wook";
(2) char name[]   = "oh ki wook";
```

위의 (1)과 (2)는 완전히 동일하다는 것이다. 터보-C 컴파일러가 정

의와 동시에 초기화 시킨 문자열의 경우에는, 문자열의 길이를 자동으로

계산하여 처리하여 주기 때문에 가능하다. 이는 아주 중요하다.

그러면 아래와 같은 경우는 어떻게 될까 ?

```
char name[];    ............ 에러 발생
```

에러가 발생한다. 초기화를 하지 않았기 때문이다. 초기화를 하지 않으면

반드시 문자열의 길이를 명시해 주어야 한다.

5.printf()함수

위에서도 printf()함수에 대해서 간단히 설명했지만 printf()함수는 C-언

어의 표준 출력함수 이므로 사용이 빈번하다. 간단한 예와 함께 printf()

함수를 익혀보자.

```
      (1) printf("%s","hello !");
      (2) printf("hello !");
```

(1)과 (2)는 동일하다. 간단히 문자열 하나만 화면상에 나타내줄 때에는

제어 문자열을 사용할 필요가 없다.

printf()함수의 역할을 도식적으로 알아보자.

```
   printf("이름:%s 아이큐:%d","철수",60);
                 ---        ---   -----  --
                  +---------|-----+      |
                            +-----------+
```

5.1 특수문자 출력

printf()함수를 사용하다 보면 행을 바꾼다던가 삑 소리가 나게 한다든가

하는 일이 필요하다. 이를 특수문자가 해결해준다. 특수 문자는 제어문자

열 중에서`[/]`에 이어 지정한다. 아래는 대표적인 특수문자의 예이다.

ㄷI

```
    \n................행을 바꾼다.
    \t................1탭 간격만큼 띄운다. (보통 8칸)
    \b................한문자 만큼 앞으로 이동한다. (backspace)
    \7................"삑"소리를 울린다.
```

ㄷ

위의 특수문자중 가장 많이 사용되는 것은 행을 바꾸는`\n`이다. 이외에

도 삑 소리를 내는`\7`가 가끔 사용되기도 한다.

ㄷIsam0103.dat

```
    [예제] 특수문자 사용예
    #include <stdio.h>              /* 없어도 된다 */
    main()
    {
      printf("I am a boy\n\7");   /* 특수문자 \n과 \7을 동시에 사용 */
      printf("you are a girl");
    }
    [결과]
    I am a boy      ....... 행이 바뀌면서 "삑" 소리가 난다.
    you are a girl
```

ㄷ

6. scanf() 함수

scanf() 함수는 C-언어의 표준 입력 함수로써, 사용법은 printf()함수와 별로 다를것이 없다.

`scanf("%d",&age);`

그런데 인수인 age의 앞에 &를 붙이는것이 다르다.

여기서 그 이유를 생각해 보자. printf()함수의 인수는 함수 호출로 인수의 내용이 화면에 표시되는것 뿐이다. 그러나 scanf()함수는 인수가 입력된 값에 의해서 변해야 한다.

다시말해 printf()함수는 인수를 변화시키지 않기 때문에 인수의 값만 넘겨 주어도 된다. 그러나 scanf함수는 인수가 변화 되기 때문에 인수 자신의 주소를 인수로 넘겨 주어야 하는것이다. 따라서 추측할 수 있는 것은 `'&'`가 주소를 넘겨주는 연산자라는 것이다.

다시말해서 인수는`포인터`를 넘겨주는 연산자라는 것이다. 포인터에 대해서는 배열과 포인터 단원에서 자세히 다루도록 하겠다.

인수앞에`&`를 붙여 포인터를 만드는데도 일종의 규칙이 있다. 아래의 예를 통해 알아보자

ㅎl

(1) 기본 자료형에는 &를 붙인다.

```
int a;
float b;
char c;
scanf("%d %f %c",&a,&b,&c);

(2) 문자열 변수에는 &를 붙이지 않는다.
char ch[50];
scanf("%s",ch);
```
ত

위와 같이 문자열 변수에 &를 붙이지 않는 이유는 문자열 변수 자체가 일

종의 포인터 이기 때문이다.

তlsam0104.dat

```
[예제] scanf()함수 사용예
main()
{
  char name[30];
  int age;

  printf("Input your name\n");
  scanf("%s",name);         /* 이름을 입력받는다. */
  printf("Input your age\n");
  scanf("%d",&age);         /* 나이를 입력받는다. */
  printf("name %s age %d",name,age);
}
[결과]
배트맨 <enter>
38 <enter>
name 배트맨 age 38
```
ত

서론

데이타는 실제로 프로그램의 모든 정보가 들어가게 되는 부분이다. 데이

타 없는 프로그램은 존재할 수 없다. 바꿔 말하면 데이타를 얼마나 효율적으로 가공하고 사용하느냐에 따라서 프로그래머 의 능력이 평가된다고도 할 수 있다.

이장에서는 C-언어의 기본 데이타 형과 데이타를 사용하는데 필요한 내용을 공부하게 된다.

특히 정수형 데이타에 대한 내용과 포인터에 대한 기본적인 내용이 언급되므로 눈여겨 봐두기 바란다.

이장은 이론 중심이다. 좀더 실용적인 데이타의 활용법에 대해서는 진행 하면서 예가 나올때 마다 학습하도록 하겠다.

ѫillu0200.dat

1　데이터형의 분류

C 언어가 제공하는 데이터형(data type)의 종류는 타언어의 추종을 불허할 만큼 매우 다양하고 그 사용법을 모두 익히는데도 많은 시간이 든다.

그러므로 여기서는 기본형만을 집중적으로 알아보기로 하자.

기본형(bassic type) 이란 문자, 정수 또는 부동 소수점 수와 같이 각종

수치를 취급하는 데이터형과 특수한 void형 등을 통틀어서 일컫는 말이다

아래의 표중 void형을 제외한 나머지를 특별히`산술형(arithme-tic type)`

이라 함을 알아두기 바란다.

ƃI@
터보 C의 기본형

문자형(character type)
정수형(integral type)
열거형(enumerated type)
부동형(floating type)
void형(void type)
ƃ
1.1 비트, 바이트, 워드

위의 다섯가지 데이터형을 하나하나 알아보기 전에 우선 컴퓨터의 기억단

위인 비크, 바이트, 워드에 대해서 한번 정리해 보고 정수와 부동소수점

수 (실수)의 개념에 대해서도 간단히 살펴보자.

비트, 바이트, 워드라는 용어는 컴퓨터의 기억 단위(memory unit)를 나타

내는 말이며 적어도 컴퓨터 사용자라면 비트나 바이트라는 용어쯤은 익히

알고 있을 것이다. 그러나 "워드"에 대해서 모르는 사람이 간혹 있을지도

모르겠다. 자, 그러면 이 용어들의 의미에 대해 확인하도록 하자.

`비트(bit)`는 메모리의 최소 단위로서 1또는 0 중에 어느 하나를 나타낸

다. 비트의 개념은 흔히 on/off 또는 set/reset으로 표현한다. 컴퓨터 내

부의 모든 전자회로는 이 비트 개념을 바탕으로 하고 있다.

이렇듯 컴퓨터의 거의 모든 구성 요소가 비트라는 기억 단위에 바탕을 두

고 있지만, 비트 단위만을 가지고는 많은 정보를 표현할 수가 없다.

그래서 컴퓨터에서는 통상 여러 개의 비트를 하나로 묶어서 사용한다.

이 때 비트를 몇 개씩 묶느냐에 따라 새로운 기억 단위가 만들어 진다.

이렇게 묶는 방법에는 여러가지가 있을 수 있으나 컴퓨터에서는 대개 2의

제곱수 단위를 비트로 묶고 있다. 이들 각각의 묶음에 대해 붙이는 기억

단위명은 컴퓨터 시스템마다 제각기 다른 것도 있고 공통되는 것도 있다.
ᄒl@

여러가지 기억 단위(memory unit)

비트 수	바이트 수	단위명	컴퓨터 시스템
1	1/8	비트(bit)	공통
4	1/2	니트(nibble)	공통
8	1	바이트(byte)	공통
16	2	워드(word)	16비트 컴퓨터
32	4	더블워드(double word)	IBM-PC
128	16	패러그래프(paragraph)	IBM-PC

ᄒ

1.2 워드(WORD)

워드(WORD)라는 기억 단위의 크기는 비트나 바이트와는 달리 컴퓨터 시스

템마다 제각기 가르다. 그것은 워드라는 용어의 의미가`컴퓨터가 한꺼번

에 처리할 수 있는 데이터 처리 단위` 이기 때문이다. 그러므로 컴퓨터

의 데이터 처리 능력에 따라서 워드의 크기(워드의 비트 수)도 달라지게 된다.

각 컴퓨터가 다루는 워드의 크기를 가장 손쉽게 아는 방법은 그 컴퓨터가 `몇 비트 CPU`를 사용하고 있는가를 알아보면 된다.

여러분이 사용하고 있는 PC 인 IBM-PC/XT나 IBM-PC/AT 등은 각각 16비트 CPU로 인텔 8088/80286 CPU를 사용하므로 워드의 크기는 16비트 이다.

그리고 32비트 CPU인 80386에 기초한 386PC는 워드의 크기가 32비트. 즉 4바이트나 된다. 탁상전자출판(DTP)으로 유명한 매드의 크기가 64비트, 혹은 그 이상 되는 것도 있다.

우리가 사용하는 컴퓨터는 16비트 컴퓨터인 IBM-PC이므로 워드의 크기도 역시 16비트이다. 386PC에서도 286PC와의 호환성을 고려하여 16비트 워드를 사용하고 있다.

1.3 정수와 부동 소수점 수

정수(integer)는 다 아는 사실이지만, 소수부가 없는 수이다. 1989,-11, -629 등은 정수이다. 그러나 3.14592, 4.19, 6.102, -5.174 등의 수치는 정수가 아닌 부동소수점 수(floating point number)이다.

베이직 같은 언어에서는 정수나 부동 소수점 수를 메모리에 저장하는 방법 에 대해서 신경을 쓰지 않아도 프로그래밍하는 데 거의 지장이 없다.

그러나 C언어는 중간 수준 언어(middle level language)이기 때문에 수치의 저장 방법에 대해서 어느 정도 알고 있어야 후에 설명할 데이터형이나 비트 연산자 등을 완전하게 이해할 수 있다.

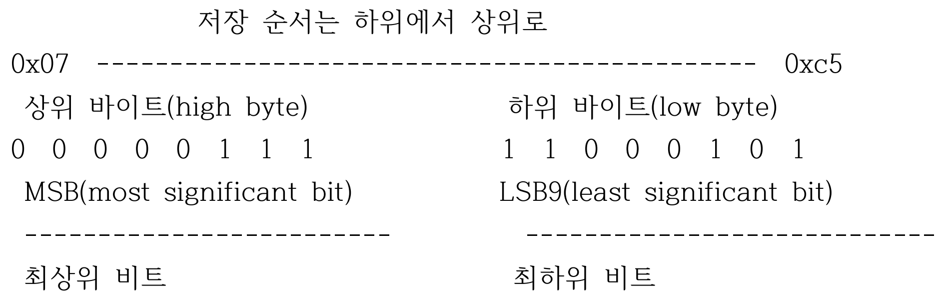
특히 정수형 데이터의 경우는 그 비트 구조에 대해 상세하게 알아둘 필요가 있다. 컴퓨터는 이러한 정수를 이진수의 형태로 메모리에 저장하는데, 특히 우리가 사용하는 IBM-PC에서는 대개 1워드, 즉 2바이트 크기로 저장하고 있다. 부호있는(signed) 정수인 경우에 그 비트 구조는 아래와 같다.

예를 들어 정수 1989는 16진과 2진표기법으로 나타내면 아래와 같다.

`16진 0x07c5      2진 00000111 11000101`

C-언어에서 접두사 `0x`는 16진수임을 나타낸다.

ㅎl@
 정수 1989를 메모리에 저장하는 방법



ㄷ

어셈블리어를 익히 알고 있는 독자라면 이해가 빠를텐데, 위의 정수가 메모리에 저장될 때에는 상위 바이트, 하위 바이트의 순으로 저장되는 것이 아니라 그 반대로 하위 바이트, 상위 바이트 순으로, 즉 역순으로 저장된다는 점을 잊지 말아야 한다.

위의 예에서 1989는 0x07과 같이 역순으로 저장된다. 이러한 저장 방식을`역 워드 방식`이라고도 한다. 하여간에 정수는 2바이트 크기로 메모리에 저장되기 때문에 그 표현할 수 있는 값에 당연히 한계가 생긴다.

만약 부호있는(signed) 정수라면 MSB를 부호 비트(sign bit)로 사용하기 때문에 표현할 수 있는 값들은 -32768 ~ 32767 범위를 가지는 65536개의 서로 다른 값들이 된다. 이 때 표현하고자 하는 값이 0 이상이면 MSB가 0으로 설정된다. 반대로 그 값이 음이면 MSB가 1로 설정된다. 다시 말해서 수치가 음일 때에는 2의보수 형태로 메모리에 저장된다는 뜻이다.

예를 들어 1212는 16진 표기법으로 표현하면 0x04bc이다. 따라서 음수 -1212를 2의보수 형태로 나타내면 65536 - 1212 ==0x10000 - 0x04bc == 0xfb44 == (11111011 01000100) 이 된다. 여기서 MSB가 1로 설정되어 있

음을 알수 있다.

파스칼의 interger형이 바로 C의 부호

있는 정수형에 해당된다. 부호없는(un

signed) 정수의 경우는 MSB가 부호 비

트로 사용되지 않기 때문에 0 ~ 65535

범위의 값을 취할 수 있다.

예를 들어, 양수 64324는 0xfb44 == (

1111011 01000100)의 형태로 메모리에

저장된다. 여기서 우리는 한 가지 사

실에 주목해야 한다.   illu0201.dat

위의 예에서 메모리에 저장되어 있는 2바이트의 0xfb44라는 값을 부호있

는 정수로 간주했을 경우 그 값은 -1212이지만, 반면에 부호없는 정수로

간주했을 경우는 64324라는 값으로 인식된다는 점이다.

쉽게 말해서 메모리에 저장되어 있는 동일한 수치라도 보는 관점(signed

, unsigned) 에 따라 그 값이 전혀 다르게 인식된다는 것이다. C 언어의

이러한 성질은 융통성이란 측면에서 장점이 될 수도 있지만, 잘못 오용하

게 되면 심각한 문제를 일으킬 수도 있으므로 어떻게 보면 단점이라고 할 수도 있다.

즉 다이너마이트처럼 잘만 활용하면 프로그래밍에 상당한 융통성을 주지만 실수로 오용하게 되면 크나 큰 문제점이 발생하는 그러한 양면성을 지니고 있다는 것이다. 이것은 C 언어의 전반적인 특징이기도 하다. 즉 표현의 자유를 대폭 허락하는 대신 그 결과에 대한 책임은 전적으로 프로그래머에게 지우는 것이다.

파스칼에서는 서로 다른 데이터형의 혼용을 철저하게 제한하기 때문에 이러한 문제가 거의 발생하지 않는다. 그러나 C에서는 이 문제가 심각한 논리 에러를 유발할 가능성이 있다. 구체적인 실예는 앞으로 계속해서 하나씩 언급해 나가겠다.

다음으로 부동 소수점 수(floating point number)에 대해서 알아보자. 먼저, 부동소수점 수를 표기하기 위한 부동 소수점 표기법은 과학적 표기법과 비슷한 표기법으로서 별것은 아니고 실수를 다소 색다르게 표현한 것뿐이다. 예를 들어, -1.9870629e3,7.7e-1, 6.26e+5 등과 같이 실수를 가수부(mantissa)와 지수부(exponent)로 나누어 표현하는 방식이 바로 부동소수점 표기법이다.

반면에 실수를 4.19, 6.6등 등으로 표기하는 방식은 고정 소수점 표기법(fixed point notation)이라고 한다. 이때 C에서는 고정 소수점 표기법으로 나타낸 수라 할지라도 컴파일할 때 자동적으로 부동 소수점 수로 변환

되어 메모리에 저장됨을 참고하기 바란다.

한편 부동 소수점 표기법의 특징은 광범위한 수를 다룰수 있다는 점이다.

예를들어 10의 -30제곱과 같이 매우작은 수를 고정 소수점 표기법으로 나

타내려면 0.000000000000000000000000000001이라고 표기해야 한다. 그러

나 부동 소수점 표기법을 쓰면 1.0e-30과 같이 짤막하게 나타낼 수 있다.

```
ठl@
     부동 소수점 수의 비트 구조

  float형 (4바이트 = 32 비트)
   31     30  29  28  27  26  25  24  23  21  20  ...   3  2  1  0
  ---    ------ 지수부 8비트 -----------------  -- 가수부 23비트 ---
 가수의 부호 비트<0: 양수, 1: 음수>

 double형 (8바이트 = 64비트)
  63     62  61  ...  55  54  53  52  51  50  49        3  2  1  0
 ----   ------ 지수부 11비트 ----------------  -- 가수부 52비트 ---
 가수의 부호 비트 <0: 양수, 1: 음수>
ठ
```

이러한 부동 소수점 수는 C에서 4바이트(float형) 혹은 8바이트(double형

)의 트기로 메모리에 저장된다. 그러나 부동 소수점 수식의 연산만은 언

제나 8바이트 크기(double형)로 행하여

진다는 점을 알아 두기 바란다.

그런데 부동 소수점 수의 비트 구조에

대해서는 굳이 관심을 가질 필요가 없

다고 본다.   다만 부동소수점 수가

4바이트 혹은 8바이트 크기로 메모리에

저장된다는 점만 확실하게 기억해 주기

illu0202.dat

바란다.

그리고 부동소수점 연산을 수행할 때에는 반올림 오차(round-off error)

가 발생한다는 점을 고려하여 수치의 정밀성에 주의를 기울여야 한다.

1.4 정수형(integral type)

1.4.1 정수형 데이터의 개론

C는 다양한 정수형을 제공한다. 원래 C에는 크게 나누어 세 가지의 정수

형이 있다. 먼저 파스칼의 integer형에 해당하는 int형이 있고, 그것을

기준을 크기가 같거나 작은 short형과, 크기가 같거나 큰 long형등 합해

서 모두 세 가지가 된다.

예를 들어 32비트 컴퓨터 등에서는 int형의 크기가 4바이트인 경우도 있

다. 그런데, 정수는 대개 2바이트 대개 2바이트 정도면 충분히 표

현할 수 있으므로 2바이트 크기를 가지는 데이터형을 마련해 두면 메모리

의 낭비를 줄일 수 있다. 그래서 C에서는 대개 (모두가 아님) 2바이트의

크기를 가지는 short int형(줄여서 short형)과 4바이트의 크기를 가지는

long int형(줄여서 long형)을 함께 제공한다.

이렇듯 정수형 데이터의 크기로만 보면 C에는 실질적으로 2종류의 정수형

만이 있는 셈이다.

그런데 C에는 따로 int형이라는 일반적인 정수형도 있다. 이 int라는 데

이터형은 프로그래머가 데이터형의 크기나 기타 여러가지 세부 사항에 대

해서는 신경쓰고 싶지 않을 때 즐겨 쓰는 정수형이다.

short int형이나 long int 형은 대형, 소형을 불문하고 컴퓨터 시스템마

다 거의 예외없이 그 크기가 각각 2바이트, 4바이트로 통일 되어있다.

그러나, int 형은 컴퓨터 시스템마다 제각기 그 크기가 다르다. 예를 들

어서, 대형컴퓨터나 DEC의 VAX 같은 32비트급 미니 컴퓨터는 int형의 크

기를 long int형과 같은 4바이트(32비트)로 잡는다.

그러나 16비트나 8비트급 컴퓨터는 int형의 크기를 short int 형과 같은

2바이트(32비트) 로 잡는다. 이렇게 컴퓨터 시스템마다 int형의 크기가

다른 이유는 각 컴퓨터의 데이터 처리 능력에 따라 가장 효율적으로 int

형의 크기를 선택하기 때문이다.

자, 그러면 우리의 IBM-PC에서는 int형의 크기가 몇 바이트일까?

답은 2바이트이다. 그 이유는 간단하다. IBM-PC는 16비트 CPU를 사용하고 있기 때문이다. 결론적으로 IBM-PC에서는 int형이 short int형과 같은 크기를 가진다고 할 수 있다.

그리고 long int형은 4바이트의 크기를 가진다.

한편 이상에서 설명한 int,short int, long int형 등은 별다른 제한을 가하지 않는한, 정확히 말하면 별다른 형지정자(tupe specifier)를 덧붙이지 않는 한, 부호있는(signed) 정수형이다. 이 때 특별히 부호있는 정수형임을 명확하게 밝히기 위하여 signed라는 형지정자를 덧붙이는 경우도 있다.

C는 이 외에도 부호없는 (unsigned) 정수형을 제공한다. 부호없는 정수형은 말그대로 음수가 아닌 0 이상의 정수값만을 표현할 수 있는 정수형이다. 1.3절에서 알아보았듯이 부호없는 정수는 그 비트 구조상 MSB를 부호비트(sign bit)로 사용하지 않는다.

따라서 동일한 바이트 크기의 부호있는 정수형은 부호있는 정수형의 명칭 바로 앞에 unsigned라는 형지정자를 붙임으로써 나타낸다. 그러므로 터보 C에는 unsigned short int, unsigned int. unsigned long int등 세 가지 부호없는 정수형이 있는 셈이다.

1.4.2 동의어와 약식 표기

다음과 같은 세 가지 이유 때문에 정수형 명칭 중에는 동의어가 생긴다.

ᇂI

[1] 터보 C 에서는 int형과 short int형이 완전히 동일하다. 따라서
unsigned short int 형과 unsigned int형도 완전히 동일하다.
[2] 형지정자 int 는 다른 형지정자가 있는 경우에 항상 생략 가능하다.
[3] signed 형 지정자는 붙이나마나이다.

ᇂ

이 세 가지 이유 때문에 완전히 동일한 형명(type name)이 생기게 된다.

이렇게 동의어들을 모조리 제외하고 나면 터보 C의 정수형은`int,`

`unsigned, long , unsigned long`등 단 4가지로 압축된다.

터보 C 2.0 에서는 이렇게해서 모두 14개의 동의어가 생긴다. 그러나 이

러한 동의어들은 뜻이 비슷한게 아니고 완전히 동일한 것이므로 안심하고

위의 4가지에만 신경쓰면 된다.

ᇂI@

[정리] 터보C의 정수형

int	부호있는 integer	2바이트
unsigned	부호없는 integer	2바이트
long	부호있는 long integer	4바이트
unsigned long	부호없는 long integer	4바이트

ᇂ

1.4.3 터보 C의 정수형

프로그램을 직접 작성하면서 가장 많이쓰이는 정수형은`int형`뿐이다.

long이나 unsigned long형은 별로 쓰이는 일이 없고 파일단원 에나 가야

쓰인다. 왜냐하면 그렇게 큰 정수값을 다룰 일이 없기 때문이다.

다만 unsigned형은 16진수를 다루는 데 매우 유용하기 때문에 비교적 자

주 쓰이는 편이다.

그러나 long형이 쓰여할 곳에는 반드시 long형이 쓰여야 한다. 예를 들어

게임을 만들때 게임의 점수를 생각해 보자.   int형의 한계는 고작 32767

이고 unsigned의 경우에도 65535가 한계이다. 이런 경우에는 반드시 long

형 정수를 사용하여야 하는 것이다.

그러나 int형 정도로 충분히 표현할 수 있는 곳에 long형 등을 쓰면 프로

그램의 실행속도를 떨어뜨리므로 유의하기 바란다. 실제로 모든 데이터형

중에서 가장 처리 속도가 빠른 형이 바로 int나 unsigned형이다. 심지어

문자형 보다도 근소한 차이로 빠르다.

ㅎl

정수형	바이트	부호	사용빈도
int	2	있음	가장 많이 쓰인다.
unsigned	2	없음	16진수나 2바이트값을 다루는 데 주로 쓰인다.
long	4	있음	대용량 화일 입출력과 산술연산에 쓰인다.
unsigned long	4	없음	원거리 포인터(far pointer)나 대용량 화일 입출력에 쓰인다.

ㅎ

이제 4가지 정수형이 가질수 있는 수의 허용 범위를 알아보자. 다른 언어

에서도 마찬가지지만 C 에서도 정수형 데이터에는 허용 범위 내의 값만을

사용하도록 주의해야 한다.   만약 허용 범위를 넘는 값을 사용하게 되면

오버플로우(overflow)가 발생 하여 원치 않느 결과를 얻게 된다.

예를 들어 unsigned 형에는 -1을 대입할 수 없다. 물론 거의 대부분 컴파

일러에게 적발(?) 되겠지만 만약 대입이 되었다 하더라도 올바른 값이 나

올 수 없다.

ъl@

정수형 데이터의 허용 범위

int	-32768	~ 32767
unsigned	0	~ 65535
long	-2147483648	~ 2147483647
unsigned long	0	~ 4294967295

ъ

ъlsam0200.dat

[예제]-오버플로우의 예

```
void main(void)
{
  unsigned su;    /* number를 부호없는 정수형으로 선언한다. */
  su = -1;        /* unsigned형에 음수의 값을 준다. */
  printf("%u\n", su);
}
```

[결과]

65535 -> 엉뚱한 값이 출력되었다.

ъ

1.5 문자형 (character type)

터보 C가 지원하는 문자형에는 모두 세 가지가 있다.

signed char형, unsigned char형과 char형이 그것이다. 문자형은 다른 언어에서와 마찬가지로 1바이트의 크기로 메모리에 저장된다. 그러나 C에서는 문자형 데이터가 수식 내에 쓰일 때 언제나 부호확장(sign extension)에 의해 int형으로 변환된다는 점을 알아두기 바란다.

우리가 사용하는 터보 C의 char형에는 부호가 있다. 다시 말해서 char와 signed char는 동의어이다 따라서 터보 C의 문자형은`char 형과 unsigned` `char형` 이 2가지만 있는 것으로 간주한다.

ចl@

문자형 데이터의 허용 범위

문자형	허용 범위	바이트	부호
char	-128~127	1	있음
unsigned char	0 ~255	1	없음

ច

1.6 부동형 소숫점형 (floating type)

지금까지 익힌 정수형이나 문자형 데이터만 가지고도 웬만한 프로그램은 작성할 수가 있다. 그러나 수학적인 계산이 필요할 때에는 부동형(실수형) 데이터가 더 필요하게 된다. 예를 들어 성적 처리를 위해 평균을 구한다든지, 바이오리듬의 곡선을 구한다든지 하는 경우이다.

이를 위해서 C는 부동형 (floating type)을 지원한다. C의 부동형은 10 진수로 나타냈을 때 가수부(manti-ssa)의 정밀도에 따라 2가지로 구분된다

ᄒl@

부동형의 허용 범위와 정밀도(precision)

	최대 지수	최소 지수	최대 정밀도	유효 정밀도	출력 정밀도
float	+38	-38	8자리	7자리	7자리
double	+308	-308	18자리	16자리	15~16자리

ᄒ

[참고] double형은`배정도형(double precision type)`, float는`단정도형`
`(single precision type)`이라고도 한다.

여기서의 정밀도란 유효 숫자의 자리수를 뜻한다.

수식에 부동 상수(floating constant) 를 사용할 때에는 가급적 유효 정밀도만큼의 유효 숫자를 지정해 주어야만 정밀한 부동 소수점 연산을 제대로 수행할 수 있다.

ꈰI@

[예] x, y가 float형일 때,

(1) x = 3.14;

(2) x = 3.141592;

(1) 보다는 (2)를 사용할때 보다 정확한 결과를 얻을 수 있다.

ꈰ

ꈰIsam0201.dat

[예제] double 형과 정밀도

```
#include <math.h>
void main(void)
{
      double pi, y;

      pi = asin(1. ) * 2.;       /* asin는 sin의 역함수이다. */
      printf("\npi = %.14f\n\n", pi);
      y = sin(pi / 6. );
      printf(" sin(pi / 6) = %14f\n" , y);
      pi = 3.141592;
      y = sin(pi / 6.);
      printf(" sin(3.141592 /6) = %.14f\n" , y);
}
```

ꈰ

ꈰIsam0201.dat

[결과]

```
    pi = 3.14159265358979
    sin(pi / 6) = 0.50000000000000          -> 정확한 결과
    sin(3.141592 / 6_ = 0.499999900566244 -> 오차가 있다.
```

위의 결과를 보면 가급적 최대의 유효 숫자를 써주어야만 정확한 결과를

얻을 수 있음을 알 수 있다.

여기서 한가지를 생각해 보아야 한다. 왜 부동소숫점 수에도 float와 do

uble 두가지가 있을까 ? 이유는 정밀도의 차이가 있기 때문이다. 다음의

예제는 10만을 만번더해 평균을 구한것이다. 결과는 당연히 10만 이여야

한다. 결과를 살펴보도록 하자.

sam0202.dat

```
    [예제] float와 double의 정밀도 비교
    void main(void)
    {
      int i; float x; double y;
      x = 0.;
      y = 0.;
      for(i = 0; i < 10000; i++)
      {
        x = x + 100000.;
        y = y + 100000.;
      }
      printf("%.6e\n", x / 100000.);
      printf("%15e\n", y / 100000.);
    }
    [결과]
    9.99852e+04                 -> float을 사용 하였더니 오차가 있다.
    1.00000000000000e+05     -> double을 사용한 경우 정확하다.
```

실행 결과에서 볼 수 있듯이 double형의 경우 정확한 결과를 보이나 flo

at형의 경우는 미미한 오차를 보이고

있다. 그러나 이러한 오차는 대개는

잘 일어나지 않는다. 만약에 이러한 오

차 조차도 용납할 수 없다면 double 형

을 사용하면 될것이다.

어차피 float형도 수식을 계산할때는

double형으로 변환되므로 수행시간에는

변화가 없다. 다만 double형이 float형

보다 4바이트의 메모리를 더 차지할 뿐

이다.

만약에 더 정밀한 연산을 해야 하는경

우 라면 10바이트 크기의 `long double`

`형` 을 사용할 수 있다. 이수의 지수

범위는 자그마치 -4916~+4932 나 되고

유효 정밀도와 출력 정밀도는 각각 20

, 19 이다.

illu0203.dat

1.7 상수 (constant)

지금까지 정수형, 문자형, 부동형 등의 세 가지 기본형에 대해서 알아보
았다. 이제부터 이 세가지 기본형에 대응하는 상수들에 대해서 살펴보자

터보 C의 상수는 문자 상수, 정수 상수, 부동 상수, 문자열 상수, 열거
상수(enumeration constant) 등 모두 5가지 종류가 있다. 이 상수들 중
열거 함수는 좀더후에 알아보도록 하고 나머지 4가지의 상수에 대해서 알
아 보도록 하자.

1.7.1 정수 상수(integer constant)

정수 상수는 그 값의 크기에 따라 4가지 정수형 중 하나의 데이터형으로 취급된다. 그 취급 기준은 아래의 표를 보기 바란다.

```
    0           ~ 32767                  int형으로 취급
    32768       ~ 65535             unsigned형으로 취급
    655369      ~ 2147483647            long형으로 취급
    2147483648 ~ 4294967295     unsigned long형으로 취급
```

위의 상수는 모두 10진수를 예로 들었다. 그러나 터보 C에서는 16진 상수와 8진 상수도 사용할 수 있다. 정수상수의 맨앞에`0`이 붙으면 8진수로 인식하고 `0x`가 붙으면 16진 상수로 인식한다.

```
    진수          사용숫자          예          비고
    10진수        0~9               123
    16진수        0~9 A~F(a~f)      0x123       앞에 0x를 붙인다.
     8진수        0~7               0123        앞에 0을 붙인다.
```

다음 예는 잘못된 상수 지정의 예이다.

```
  12a9      16진수가 분명한데 접두사 0x가 빠졌다.
  0128      8진수의 세상에는 8이라는 숫자가 존재하지 않는다.
  0x123g    16진수에는 0~9와 a, b, c, d, e, f만 숫자(digit)로 사용된다.
  0x123 2   상수 중간에 공백이 있으면 안 된다.
```

여기서 중요한것은 숫자를 표기하는 방법의 차이일뿐 실제로 메모리에 저장될 때에는 같은 2진수로 저장된다는 점이다.

sam0203.dat

```
  [예제]
    main()
    {
      printf("%d\n",10);
      printf("%d\n",012);
```

```
    printf("%d\n",0xa);
}
[결과]
10
10
10    -> 10, 012, 0xa 가 모두 같은 값 임을 알수 있다.
```
ট
그리고 10진 상수의 앞에 `0`을 붙이면 8진수로 인식되므로 주의 하여야 한다.
[예] 01212 -> 8진수로 인식된다.

이런경우를 생각해 보아야 한다. 상수의 값이 1인데 long형으로 지정해야 하는 경우에는 어떻게 해야 하는가 ? 위에서 상수의 값이 0~32767 사이이면 int형으로 인식된다고 배웠다.
이럴때는 각 형의 접미사를 붙이면 된다. long형의 값에는`L(l)`을 붙이면 되고 unsigned형의 값에는`U(u)`를 붙이면 된다. 상수의 접미사에 대한 사항은 앞으로 종종 언급하게 될것이므로 잘 알아두기 바란다.

টI@
```
[정리] 상수의 접미사

1234    -> int 값으로 인식된다.               2 바이트
1234L   -> long 값으로 인식된다.              4 바이트
1234U   -> unsigned 값으로 인식된다.          2 바이트
```
ট

1.7.2 문자 상수 (character constant)와 확장열 (escape sequence)

문자 상수라는 것은 작은따옴표(')로 둘러싸인 단일문자(하나의 문자)를 말한다. 예를 들면 다음과 같은 것들이 문자 상수이다.
টI
```
'A'    'B'   'a'    's'    '='    '['    '" '
```
ট
C-언어에는 이런 것들뿐만 아니라 다음과 같은것들을 문자 상수로 사용할 수가있다. 예를 들어,
টI
```
'\n'          '\''          '\a'          '\\'          '\b'
```

개행문자 (작은 따옴표) 벨소리 \(역슬래쉬) 백스페이스

ᄃ

이와 같은 문자 상수들을 따로 이름하여`확장열(escape sequence)`이라고 부른다. C에서는 작은따옴표 안에 역슬래쉬(\)가 사용되면 역슬래쉬 다음에 있는 문자를 확장열로 받아들인다.
확장열중 가장 많이 사용되는 것은 '\n'을 들수 있다. 이전의 예제들을 보면 printf문과 함께 사용되어 열을 바꾸는데 사용됨을 알 수 있다.

그 다음으로 알아두어야 할 확장열로는 \x???의 형태이다. 이 형태는 나타내려는 문자를 ASCII 코드 값으로 직접 나타내는 것이다.

ᄃI

```
'\x1b' -> ESC      '\0' -> NUL       '\x7f' -> DEL
'\x41' -> 'A'      '\xff' '\xb5' -> 그래픽 문자
```

ᄃ

여기서 중요한 것은 역슬래쉬(\)를 나타내는 방법이다. 이는 C-언어를 처음 배우는 사람 뿐만 아니라 C-언어에 도통한 사람도 가끔 실수하는 부분이기도 하다. 아래의 예제를 보자.

ᄃIsam0204.dat

```
[예제]
   main()
   {
      printf("%s","C:\UTIL\NORTON\SD.EXE");
   }
   [결과]
   C:UTILNORTONSD.EXE -> 아니 ! 이럴수가 !
```

ᄃ

위의 예제는 앞으로 배울 문자열 상수로 예를 들어 설명했지만 눈치가 빠른 사람들은 벌써 이해 했을것이다. 그렇다면 역슬래쉬(\)자체는 어떻게 나타내야 하는가 ? 정답은`'\\'`이다. 따라서 위의 예제의 올바른 표현은

ᄃI

```
printf("%s","C:\\UTIL\\NORTON\\SD.EXE");
```

ᄃ

인것이다.

같은예로 작은따옴표(') 를 나타낼 때는 ''' 라고 하면 않된다. 이럴때는

확장열을 사용하여 '\'' 라고 사용하여야 한다. 그렇다면 큰따옴표를 나타낼 때는 어떻게 해야 할까 ? 해답은 간단하다 '"'라고 쓰면 되는 것이다.

1.7.3 문자열 상수 (string constant)

위에서 배운 문자상수와 그 이름도 비슷하다. 무엇이 다를까 ? 문자열 상수란 큰 따옴표(")로 둘러싸인 문자열 상수를 말한다. 그렇다면 문자열이란 무엇인가. 문자열이란 문자의 모임을 말한다. 즉 문자가 한개 있으면 문자이고 2개 이상이 모여있으면 문자열이 되는 것이다. 아래의 예를 살펴 보도록 하자.

ъl@

[예] 문자와 문자열

```
 'a'   -> 문자              "abc"    -> 문자열
 '\n'  -> 문자              "abc\n" -> 문자열
 `"a" -> ?`
```

ъ

위의 예에서 물음표로 표시된 곳은 무엇일까 ? 비록 한글자로 표시되어 있긴 하지만 엄연히 문자열 이다. 왜일까 ? 사실은 두글자 이기 때문이다 문자열에는 문자열의 끝임을 나타내는 널문자(\0) 이 들어가므로 실제로는 두글자가 들어가는 것이다.

문자열 ?

이쯤에서 C-언어를 처음 배우는 초보자는 의아하게 생각할지 모르겠다. 이장의 처음에서 C-언어 에서 사용하는 기본 데이타 형에는 문자형(character type), 정수형(integral type), 열거형(enumerated type), 부동형(floating type), void형(void type) 이 있다고 배웠다.

이 다섯가지 기본형 중에 문자열형이 있는가 ? 분명히 없다. 그런데도 문자열 상수는 존재한다. 다른 언어를 이미 알고 있는 사람도 이상할 것이다. 저자도 C-언어를 배우기 전에 파스칼을 사용하였는데 문자열형이 없다고 생각했었다. 그러나 C-언어 에서도 분명히 문자열을 사용한다. 다만 문자열형 이라는 독립적인 형태가 아니고 문자형을 여러개 모아서 배열의

형태로 사용한다. 이에 대한 자세한 내용은 다음장인`배열과 포인터`편에 서 자세히 다룬다.

이 부분이 자세히 이해가 가지 않는다고 겁낼 필요는 없다. 당연하기 때 문 이다. 저자도 처음에는 이해하지 못했다. 만약에 C-를 처음대하는 사 람이 이부분은 이해할 수 있다면 그 사람은 C-언어를 쉽게 배울 수 있다.

1.7.4   문자열 상수 내의 확장열

문자열 상수의 요소로서 영숫자(alphanumeric), 각동 기호 문자와 더불어 확장열 (escape sequence)도 사용할 수 있다.

ϭIsam0205.dat

```
  [예제] 문자열에 확장열을 사용한 예
      main()
      {
           printf("%s","I'm a student\n");
                     /* 작은 따옴표는 그대로 사용한다. */
           printf("%s","C:\\TURBOC\\TCC.EXE");
                     /* 위에서도 설명한 내용       */
                     /* 역슬래쉬를 나타낼 때는     */
                     /* 두개의 역슬래쉬를 사용     */
                     /* 한다.                      */
       }
  [결과]
       I'm a student
       C:\TURBOC\TCC.EXE
```

ϭ

1.8 void 형

한가지 설명이 빠진것이 void형 이다. 사전을 찾아보면 "빈, 의미없는" 등의 뜻이 있다. 다시말해 아무형이나 지정하는것 이다. void 형의 사용 은 크게 3가지로 구분할 수 있다.

ϭI

[1] 함수가 void 형이면 이 함수는 귀한값을 주지 않는다.
[2] 함수가 void 형 인수(argument) 만을 가졌다면 함수는 인수가 없는것 과 같다.
[3] 포인터가 void 형이면 이 포인터는 어떤형의 포인터와도 같이 사용할 수 있다.

ϭ

당장은 이해하기 어려울 것이다.   대충 이런게 있다는 정도만 알아두자

[1],[2]는 함수에서 다루며 [3]은 배열과 포인터 단원에서 자세히 다룬다

2 보조형

보조형은란 기본형들과 함께 사용하여 기본형의 사용영역을 확대 하거나 축소한다. 터보 C 에서 제공하는 보조형에는 signed, unsigned, short, long 등이 있으며 보조형은 항상 기본형 앞에 기입된다.

ㅍ]2.1 signed

signed는 기본형이 음수와 양수의 모든 영역을 갖게 하는데 사용한다. 이 보조형은 기본으로 주어지므로 삭제해도 무방하다.
[예] char x -> signed char x로 간주된다.
int x -> signed int x로 간주된다.

ㅍ]2.2 unsigned

unsigned 는 형의 영역을 절대값화 할때 사용한다. 다시말해 음수를 사용하지 않도록 한다. 대신 양수의 범위는 두배로 늘어난다.
[예] unsigned로 인한 범위확장

int	-32768	~ 32767
unsigned int	0	~ 65535

2.3 short

이 보조형은 int형과 결합되어 사용되는 정수 영역이 단정도 임을 나타낸다. IBM-PC 이외의 다른 기종에서는 int가 4바이트 크기가 될 수도 있다고 배웠다. short은 이런 기종의 컴퓨터에서 2바이트 크기의 int형을 사용 하기위해 사용한다. 그러나 IBM-PC 기종에서는 int 형은 항상 2바이트 크기 이므로 short은 사용할 필요가 없다.

ㅍ]2.4 long

이 보조형은 int나 double형과 결합되어 사용되는 수의 배정도 임을 명시한다. long 보조형 자체만으로 사용되면 자동으로 long int 형으로 변환되어 4 바이트를 차지한다. double형과 결합되어 사용되면 10바이트의 double형을 사용할 수 있다. 터보 C 1.0과 1.5 에서는 long double 형이 일반 8바이트의 double형과 같다.

3. 결합형

결합형은 보조형과 기본형 앞에 접두어로 사용되며, 형의 사용범위, 유효 기간, 변수의 상수화 등의 작업에 사용된다. 터보 C에서 제공하는 결합형으로는 const, auto, static, extern, register 등이 있다. 결합형, 보조형, 그리고 기본형이 모두 사용될 때에는 아래와 같은 순서로 정의되어야 한다.

ঢI

```
결합형   +   보조형   +   기본형
static       unsigned     char
```

ঢ

ঢJ3.1 const

const는 어떠한 기본형과도 사용될 수 있으며 함께 사용된 데이타형을 상수화 시킨다. 즉, 일단 const와 함께 정의된 데이타는 프로그램에서 더이상 값을 변형 시킬 수 없게 된다. 이러한 경우는 프로그램의 서두에 데이타의 초기값을 지정하여 주어야 한다.

ঢI@

```
[예] const를 사용한 예
main()
{
  const int x=5;    /* const로 선언하고 초기값을 지정한다.    */

  x=10;             /* 이 부분이 에러처리 되어 컴파일이 되지 */
  x++;              /* 않는다.                                */
}
```

ঢ

위의 예제는 컴파일이 되지 않으므로 결과를 볼 수 없다. const는 데이타의 값을 프로그램 중간에 변할 수 없게 하는것 이므로 프로그램 실행 도중 변하지 않아야 할 데이타를 const로 지정하면 에러를 방지할 수 있다.

ꡕ3.2 auto

이 결합형은 static과 함께(둘이 동시에 쓰인다는 뜻은 아니다) 사용되어 변수의 사용 유효 범위를 결정하는데 사용된다.

ꡔ@

```
[예] auto 변수의 사용범위
void A()
{
  auto int i;         /*  auto 변수 i를 선언한다.           */
  i=88;               /*  i에 88의 값을 지정한다.           */
}
main()
{
  A();                /*  함수 A를 호출한다.                */
  printf("%d",i);     /*  이 부분에서 에러가 발생한다.      */
}
```

ꡔ

위의 예제는 에러가 발생되어 컴파일이 되지 않는 예제이다. 왜 main() 함수에서 함수 A 에서 정의된 i를 사용할 수 없는 것일까 ?
이유는 변수 i는 함수 A 에서만 사용할 수 있도록 auto변수로 선언되었기 때문이다. 여기에서 auto는 함수내의 변수 선언에는 자동으로 적용되기 때문에 생략할 수 있다.

여기서 한번 정리해 보자.
즉 함수내에서 선언된 모든 변수는 자동으로 auto로 선언되고 함수내부에서만 사용될 뿐 다른 함수에서는 사용될 수 없다. 이유는 함수 내부에서 선언된 함수는 그 함수가 호출될 때 만들어졌다가 함수가 종료하면 그 존재가 사라지기 때문이다.

ꡔ@

[요점] auto

[1] auto변수는 함수내에서 자동으로 선언된다. (굳이 auto로 선언해 줄 필요는 없다는 뜻이다.)
[2] auto변수는 함수내에서만 사용될뿐 함수 외부에서는 사용할 수 없다. 함수 내부에서만 사용된다는 뜻은 사용 지역이 국한 된다는 뜻이다 따라서 auto변수를`지역변수`라고도 한다.

ꡔ

ꠙIsam0206.dat

```
[예제] auto 변수의 사용예
void add()
{
  auto int i=0;      /* auto는 생략해도 된다. */
  i++;               /* i의 값 증가          */
  printf("%d ",i);   /* i의 값을 표시        */
}
main()
{
  add();
  add();
  add();
}
[결과]
1 1 1
```

ꠙ

위의 예제는 어떠한 사실을 알려주고 있는가 ? auto변수가 선언될때 초기화를 하고 있으면 그 함수가 호출될 때마다 초기화 함을 알려준다.

3.3 static

이 결합형도 auto와 함께 변수의 사용 유효 범위를 지정하는데 사용된다. 그러나 static형 변수는 auto형 변수와는 반대로 프로그램 전체에 유효 범위를 가진다.

ꠙI@

```
[예] static의 사용예
static int i=1,j=2,k=3;  /* 이와 같이 함수의 밖에서 선언되는 */
main()                   /* 변수가 static형 변수이다.        */
{
  printf("%d",i+j+k);
}
[결과]
6
```

ꠙ

위의 예제를 통해서 main()함수 밖에서 선언된 변수는 static변수라는 것을 알 수 있다. 그런데 왜 이제까지는 변수 선언에 static선언을 하지 않았을까 ? 이유는 static선언역시 auto선언과 마찬가지로 삭제해도 무방하

기 때문이다.

즉, 함수밖의 변수 선언은 굳이 알려주지 않아도 자동으로 static으로 인식 한다는 뜻이다. 아래의 예제 [1]과 [2]는 서로 동일하다.

```
    [예제1]                              [예제2]
    static int i,j,k;                  int i,j,k;
    void love()                          void love()
    {                                    {
      auto int i;                          int i;
      printf("%s","I am a boy");         printf("%s","I am a boy");
    }                                    }
    main()                               main()
    {                                    {
      love();                              love();
    }                                    }
```

참고로 static변수는 프로그램 전체에서 사용이 가능하므로`전역변수`라 한다. 이제부터 auto와 static변수라는 말 보다는`지역변수와 전역변수`라는 말을 사용하여 진행하겠다.

3.3.1 함수내에서의 static 사용

위에서 함수 밖에서 선언된 모든 함수는 전역변수(static)라 하였다
그렇다면 함수내에서 억지로 전역변수를 선언하여 사용하면 어떠할까 ?

sam0207.dat

```
    [예제] 함수내에서의 전역변수 사용
    void add()
    {
      static int i=0;       /* 전역변수로 선언          */
      i++;                  /* i의 값 증가              */
      printf("%d ",i);      /* i의 값을 표시            */
    }
    main()
    {
      add();add();add();
    }
    [결과]
    1 2 3
```

위의 예제는 auto변수의 예제와 비슷하다. 무슨 의미가 있을까 ?

요점은 간단하다. 함수내에서 auto로 선언된 변수가 초기화 될때 그 변수는 함수가 호출될 때마다 초기화 되지만(i=0 이 수행되지만) static 변수로 선언되면 오직 한번만 초기화 된다는 것이다.

ㅋ3.4 extern

큰 프로그램을 작성하게 되면 한 개의 소스 파일로는 도저히 작성할 수 없게 된다. 이때는 여러개의 모듈(프로그램 조각)로 나누어 작성하면 된다.

이때 다른 모듈에서 선언된 변수를 사용할 때에는 extern을 사용하여야 한다. 프로그램을 여러개로 나누어 작성하는 방법과 extern변수의 사용법은 뒤에 전처리계 에서 다루게 된다. 이런게 있다고만 알아두자.

ᆿillu0204.dat

3.5 register

이 형은 auto형과 사용할 수 있다. auto형 대신에 사용할 수 있다. static 와는 사용할 수 없다는데 주의하자.
보통 변수는 메모리의 일부분을 할당받아 거기에 데이타를 저장하는데 이것보다 CPU 내부의 register 라는 것을 사용하여 데이타를 가공하면 훨씬 처리 속도가 빠르다.

그러나 register의 수가 한정되어 있고 사용이 가변적이기 때문에 static 으로는 사용할 수 없다. 때문에 사용과 삭제가 가변적인 auto형 대신에 사용하면 프로그램의 속도가 향상되고 프로그램의 크기가 작아진다.

그러면 이제부터 작성되는 모든 프로그램에 auto변수 대신에 register변수를 사용할까 ???

ᆿillu0205.dat

그렇게 할 수는 없다. 이유는 설명했듯이 register의 수가 한정되어 있기 때문이다. 일단 int형으로 4개만 있다고 알아두자. 어찌되었건 모든 auto 변수를 register변수로 교체할 수 는 없다는 것을 알았다.

또, 설사 모두 register변수로 선언한다 하더라도 register변수로 사용할 수 있는것을 제외하고는 모두 auto로 처리된다. 헛고생한 셈이된다.
사실 register변수는 별로 신경쓸 필요가 없다. 이유는 컴파일러가 알아서 필요한 변수를 register변수로 처리하기 때문이다. 즉, 알아서 해 주므로 프로그래머는 신경쓸 필요가 없다.

그러나 어떤 특별한 변수가 register변수로 쓰여지기를 바란다면 register라고 명시해 주면 register변수가 될 수 있는 우선권을 준다. 그러나 이러한 수고는 별로 할 필요가 없는것 같다. 옛날 XT나 8Mh AT시절에나 프로그램을 빠르게 하려고 노력을 했지만 386,486이 대중화되고 팬티엄(586) 이 나오는 이 마당에 굳이 속도에 매달리고 있을 필요는 없는 것이다.
이제는 보다 손쉽게 사용할 수 있고 기능이 많은 프로그램을 만드는 것이 일급 프로그래머의 관건인 것이다.

4 포인터형

이제부터 포인터의 맛뵈기 이다. C-언어가 가장 강력한 이유가 바로 이 포인터의 사용이 간편하기 때문이다. 물론 파스칼이나 다른 몇몇 언어 에도 포인터가 존재한다. 그러나 C-언어에서 처럼 자유롭게 사용할 수는 없다. 때문에 C-언어에서 포인터를 익히는 것은 매우 매우 중요하고 또한 까다롭다.

포인터형은 다른 데이타와는 다르게 데이타를 직접가지는 것이 아니고 데이타의 주소를 가진다. 포인터는 주소를 가지고 작업하므로 프로그램을 보다 빠르게 실행할 수 있도록 한다. C-언어의 가장 큰 특징중의 하나는 동적인 프로그램을 쉽게 작성할 수 있다는 것인데 이는 C-언어가 포인터를 효율적으로 사용할 수 있도록 하기 때문이다.

먼저 포인터를 선언하는 방법을 알아보자. 포인터를 선언하는 것은 지금까지 배운`기본형 앞에 "*"`를 추가하면 만들 수 있다. 여기서 주의 하여야 될것은 "*"는 항상 기본형 앞에만 사용되어야 하며 보조형이나 결합형 앞에는 사용할 수 없다는 것이다.

`char*`
이형은 char형 데이타에 대한 포인터를 의미한다.   다음예의 c는 문자에 대한 포인터이다.
[예] char* c; 또는 char *c;

`int*,short*,long*`
이형은 정수형 데이타에 대한 포인터를 의미한다.
[예] int* i;   또는 int *i;
short* i;   또는 short *i;
long* i;   또는 long *i;

`float*,double*`
이형은 실수형 데이타에 대한 포인터를 의미한다.
[예] float* x; 또는 float *x;
[예] double* x; 또는 double *x;

`void*`
이형은 void형 데이타에 대한 포인터를 의미한다.
[예] void* x; 또는 void *x;

void 형에대한 기억을 더듬어 보자. void형은 어느 특정한 형을 지정하는 것이 아니라 어떤 형이라도 상관없다는 의미로 사용되기 때문에 위의 예의 x는 어떤 포인터 형과도 사용이 가능하다.

이 장에서 포인터의 모든것을 다루는 것은 아니다. 포인터에 대해서는 배열과 포인터 단원에서 자세하고 전문적으로 다루게 된다. 단지 포인터라는 것이 이런것 이라는것 정도로만 알아두기 바란다.

요점정리

이장에서 배운 내용의 요점을 정리하여 보자.

(1) C-언어에는 5가지의 기본 데이타형이 있으며 이는 문자형(character type), 정수형(integral type), 열거형(enumerated type), 부동형(floating type), void형(void type) 이다.

* 이중 void 형을 뺀 나머지를`산술형`이라 한다.

(2) 정수형에는 4가지의 종류가 있는 것으로 본다. 동의어는 생략한다.

int	부호있는 integer	2바이트
unsigned	부호없는 integer	2바이트
long	부호있는 long integer	4바이트
unsigned long	부호없는 long integer	4바이트

(3) 문자형은 2가지 만 있는것으로 본다. 역시 동의어 생략

char	부호있는 char	1바이트
unsigned char	부호없는 char	1바이트

(4) const는 데이타를 상수화 하는것으로써 한번 초기화를 한 후에는 값을 바꿀 수 없다.
(5) auto는 함수내에서만 사용되는 지역변수로 함수내에서는 굳이 명시하지 않아도 된다.
(6) static은 함수 밖에서 프로그램의 모든 부분에서 사용되도록 하는 변수 이다. 이 역시 함수 밖에서는 명시하지 않아도 된다.
(7) 포인터형 변수는 직접 데이타를 가지는 것이 아니고 데이타를 주소를 가지고 작업한다. 각 데이타형 마다 대응되는 포인터 변수가 있으며 void 형 포인터는 모든 종류의 포인터와 작업할 수 있다.

서론

C-언어를 배움에 있어서 가장 먼저 배우고 넘어가야 할것이 바로`연산자`

이다. C-언어에서 사용되는 연산자는 기본적으로 수학에서 배운 +, -,

x, %와 비슷하나, 컴퓨터 언어의 특성상 몇가지 연산자가 추가되고 변형

된 것에 주목해야 한다.

베이직이나 파스칼의 경우에는 일반 수학에서 사용하는 연산자 정도만을

사용하고 있는데 반하여, C-언어에서는 상당히 다양한 종류의 연산자를 사용하고 있다. C-언어의 또다른 특징중의 하나가 바로 이 연산자 이며 프로그램을 효율적으로 작성하는 열쇠가 되기도 한다.

우선 C-언어의 연산자를 산술연산자, 논리연산자, 조건/나열연산자 순으로 알아보도록 하자. 각각의 분류는 연산자의 쓰임에 따른 분류이며 이 장이 끝날 무렵에는 C-의 모든 연산자를 이해할 수 있어야 한다.

1. 산술연산자

어찌보면 수학에서는 오직 산술연산 밖에 없으니 다른 컴퓨터 언어를 배우지 않은 사람은 이 산술연산자 밖에는 모를것이다.

ꡕ1.1 이항연산자 (Binary Operator)

이항연산자는 2개의 데이터를 대상으로 산술적인 처리를 지시, 처리하

는 연산자를 말한다. 다시말해 더하고, 빼고, 곱하고 나누는등의 작업이

바로 이항연산자가 하는일이다.

아니 그러면 그런것들 말고 다른 연산자가 있단 말인가 ?, 이 질문에 해

답은 이제 자연히 밝혀질 것이고 우선 C-언어에서 사용되는 이항 연산자

를 알아보자.

ㅎl

+	덧 셈	a=b+c;	b와 c의 합을 a에 대입
-	뺄 셈	a=b-c;	b에 c를 뺀값을 a에 대입
*	곱 셈	a=b*c;	b와 c의 곱을 a에 대입
/	나 눗 셈	a=b/c;	b를 c로 나눈 값을 a에 대입
%	나 머 지	a=b%c;	b를 c로 나눈 나머지를 a에 대입

ㅎ

보인 이항 연산자중 나머지를 계산하는`'%'`를 제외하고는 별 설명이 필

요 없을것이다. 왜냐하면 우리가 국민학교 때부터 줄곳 보아오던 친숙한

기호 들이기 때문이다.

곱셈을`'*'`로 표기하는 이유는 곱셈기호인`'x'`가 키보드에 없기 때문

다. 언어를 조금알고 있는 사람은 우습겠지만 의심나는 초보자는 자판을

유심히 찾아보기 바란다.

왜 곱셈 기호를 넣지 않았냐고 되묻는 사람이 있다면, 저자는 굳이 할말

이 없다. 왜냐하면 그냥 그렇게 만들었으니까.

추측하건데 알파벳'x'(엑스)와 혼동이 가지 않기 위해서 그렇게 한것 같

다.

나머지 기호인`'%'`대신에`'/'`를 사용하는데는 별 이유가 없으리라 본

다. 왜냐하면 원래부터 그렇게도 사용하니까.

설명한 연산자중 처음보는 연산자는`'%'`연산자 이다. 나머지 연산자라

고 미리 설명은 했지만 초보자는 얼른 이해가 가지 않을것이다.

천천히 설명을 하자면`'/'`는 나누기의 몫을 구하는 연산자이고,`'%'`는

나누기의 나머지를 구하는 연산자이다.

ᇙIsam0300.dat

```
[예제] /와 %연산자
main()
{
   int i=13,j=5;
   printf("%d\n",i/j);      /* i 나누기 j의 몫을 출력한다. */
   printf("%d\n",i%j);      /* i 나누기 j의 나머지를 출력한다. */
}
[결과]
2
3
```

ᇙ

ᇙIsam0301.dat

```
[예제]
main()
{
   int i;

   printf("수를 입력하세요\n");
   scanf("%d",&i);
   if ((i%2)==0) printf("짝수");
   if ((i%2)==1) printf("홀수");
}
[결과]
수를 입력하세요
1
```

홀수
ㆆ
위의 예제는 수를 입력받아 단순히 그 수가 짝수 인지 홀수 인지를 알려 주는 프로그램이다.

어떤수가 짝수인지 홀수인지를 알아보려면 그 수를 2로 나누어 나머지를 확인 하면된다. 즉 어떤수를 2로 나누어 나머지가 1이면 홀수이고 0이면 짝수 이다.

ㆆl
```
printf("수를 입력하세요\n");
scanf("%d",&i);
```
ㆆ

printf문으로 문자열만을 표시할 때는 위의 방식대로 하여도 된다. 확장열 "\n"를 사용하여 행을 바꾸는데 사용했는데 기억이 나지않는 사람은 전 단원을 참고하기 바란다. scanf문은 숫자를 키보드로 부터 읽어들이기 위해 사용되었다.

ㆆl
```
if ((i%2)==0) printf("짝수");
if ((i%2)==1) printf("홀수");
```
ㆆ

위의 내용이 이 예제의 핵심으로 수를 2로 나눈 나머지가 1 이면 "짝수"를 출력하고 0 이면 "홀수"를 출력하는 부분이다.

1.2 단항연산자(Unary Operator)

단항연산자는 1개의 데이타를 대상으로 산술적인 처리를 수행하는 연산자 이다. 이 단항연산자는 오직 C-언어에서만 있는 것인데, 단항연산자가 있음으로해서 프로그램이 더욱 간편하게 구성될 수 있다.

C-언어에서 사용되는 단항연산자에는 아래와 같은 것들이 있다.

연산자	기능	사용예	다른표현
++	1 만큼 증가	i++ 또는 ++i	i=i+1;
--	1 만큼 감소	i-- 또는 --i	i=i-1;
-	부호 바꿈	b=-i;	

다른 언어를 접해본 사람은 위의 내용을 보고 단항연산자를 과소 평가할 수 있다. 왜냐하면 어떤 수를 1만큼 증가 시키거나 감소시키는 것은 이항연산자를 사용해서도 표현할 수 있기 때문이다.

i++은 i=i+1로 표현될 수 있다. 그러나 이 두가지가 완전히 동일한것은 아니다. 왜냐하면 ++의 표현은 데이터의 앞에 올때와 뒤에 올때의 의미가 각각 다르기 때문이다. 말만으로 이해 하기 힘들다면 아래의 예를 보자

ᇹIsam0302.dat

```
[예제] ++i와 i++의 다른점
main()
{
   int i=2;

   printf("%d\n",i++);
   printf("%d\n",++i);
}
```
ᇹ

위의 예제에는 실행결과를 보이지 않았다. 그 이유는 여러분이 직접 생각할 기회를 주기 위해서이다. 결론적으로 말해 ++i와 i++은 다르다.

i++ 처럼 ++가 뒤에온 경우에는 먼저 데이타를 사용한 후 데이타를 증가시킨다. 다시말해 a=3이고 b=2인 경우 a=b++; 이라고 하면 a에는 2가들어가고 b는 3이 된다.

a=++b라고 하면 a는 3이 되고 b도 3이 된다. 이 차이를 명백하게 알아두어야 한다.

`printf("%d",i++);`

위에서 i에는 이미 2라는 값이 들어있다. 그런데 i++ 으로 데이타를 먼저 사용하므로 우선 2가 출력된후, 값이 증가한다. 따라서 i는 3이 되었다.

`printf("%d",++i);`

++i를 사용했으므로 i가 먼저 증가한 후 값이 출력된다. i가 3이므로 증가 하여 4가 된후 화면에 출력된다.

위의 예는 단지 ++연산자 뿐만 아니라 --연산자의 경우에도 그대로 적용된다.

1.3 대입연산자 (Assignment Operator)

연산결과를 변수의 값으로 대입하는 연산자이다. 다른 언어를 알고 있는 사람들은 의아할 것이다, 왜냐하면 다른 언어에서는 대입연산자는 오직 한 값을 다른값에 넣는 대입만이 존재하기 때문이다.

우선 C-언어에 존재하는 대입연산자에는 아래와 같은 것들이 있다.
ㅎl

연산자	기능	사용예	다른표현
=	좌변 = 우변	a=b	
+=	좌변 = 좌변 + 우변	a+=b	a=a+b
-=	좌면 = 좌변 - 우변	a-=b	a=a-b
*=	좌면 = 좌변 * 우변	a*=b	a=a*b
/=	좌면 = 좌변 / 우변	a/=b	a=a/b
%=	좌면 = 좌변 % 우변	a%=b	a=a%b

ㄷ

ㄷlsam0303.dat

```
[예제] 대입연산자의 사용
main()
{
   int i,j,k;

   printf("수를 입력하세요 ");
   scanf("%d",&i);
   printf("수를 입력하세요 ");
   scanf("%d",&j);
   i+=j;
   printf("합 %d",i);
}
[결과]
수를 입력하세요 4
수를 입력하세요 5
합 9
```

ㄷ

+= 연산자는 좌변에 우변을 더하여 그 값을 좌변의 값으로 대입(=)한다.

아래의 간단한 예를 살펴 보도록하자.

```
a = 18;
a += 3;
```

결과로 a는 21이 대입된다. "a+=3은 a의 값을 3만큼 증가시킨다." 라고

이해하여도 좋다.

여타 대입연산자 -=, *=, /=, %=도 기본적으로 +=과 동일하게 사용된다.

다만 수행하는 산술연산만이 다를 뿐이다.

ចI@

```
[예] a=4, b=2일때
a+=2    (a=6)
a-=b    (a=2)
a*=3    (a=12)
a/=b    (a=2)
a%=b    (a=0)
```

ច

*=, /=, %= 연산자를 사용할때 주의를 요할 필요가 있다.   예를 들어

[a *= b+1]는 [a = a*b+1]의 기능을 수행하는것이 아니고 [a = a*(b+1)]

의 기능을 수행한다.

대입연산자 에는 이외에도`비트 대입연산자`가 있는데 이들에 관해서

는 조금 뒤에 알아보기로 하자.

1.4 논리연산자

논리 연산자는 대상식이 성립되는가의 여부에 따라 "참" 또는 "거짓"을

결과로 돌리는 연산자이다. 파스칼에서는 참 또는 거짓을 불린(Boolean)

이라는 변수형으로 True(참),False(거짓)을 사용하여 나타내나, C-언어

에서는 참,거짓을 나타내는 특별한 변수형은 없고 0은 거짓 0이외의 다

른 값을 참으로 나타낸다.

이들 논리연산자는 보통 제어문의 조건수식을 구성할 때 사용된다. 이에

는 크게 관계 연산자와 논리연산자가 있다.

1.4.1 관계연산자

두 데이타가 서로 동일한가의 여부와 두 데이타를 대소관계를 평가하는

연산자 이다. 아래의 표에 관계연산자를 정의 하였다.

ㅎl@

관계연산자

`연산자	기능	사용예`	
>	보다 크다.	a=(b>c)	식이 성립되면 a에
<	보다 작다.	a=(b<c)	1(논리적 참), 성
>=	보다 크거나 같다.	a=(b>=c)	립되지 않으면 a에
<=	보다 작거나 같다.	a=(b<=c)	0(논리적 거짓),
==	같다.	a=(b==c)	이 각각 대입된다.
!=	같지 않다.	a=(b!=c)	

ㅎ

위에 보인 관계 연산자들은 사용된 수식이 성립되면 1(논리적 참), 성립

되지 않으면 0(논리적 거짓)을 되돌린다.

ㅎlsam0304.dat

```
[예제] 관계연산자의 사용예
main()
{
   printf("%d",3>2); /* 3이 2보다 크므로 1(참) 이 출력된다.
}
[결과]
1
```

ㅎ

한가지 주의할점은 두 개의 데이타가 같은지를 비교할때`'=='`를 사용한

다는 것이다. 파스칼에서는`'='`를 사용하므로 파스칼을 배운 사람은 혼

돈의 여지가 있다. 여기서 관계연산자를`'=='`로 사용한 이유는 어떤 값

을 비교하는일 보다, 단순히 값을 대입하는 대입연산쪽이 훨씬 빈번하기

때문이다.

ㅎl

	대입	비교
파스칼	a:=b	a=b
C-언어	a=b	a==b

ᄐ

같지않다 !=는 파스칼에서는``<>``로 사용되므로 역시 혼동의 여지가 있

다.

1.4.2 논리연산자

지금껏 설명한 조건을 복수개로 조합하여 사용할 때는 논리 연산자를 사

용 한다. 논리 연산자에는 다음과 같은 것들이 있다.

ᄐI

연산자	기능	사용예	
&&	논리곱(AND)	a = b && c	b와 c가 모두 참이어야 참
||	논리합(OR)	a = b || c	b와 c중 하나라도 참이면 참
!	논리부정(NOT)	a = !b	b가 참이면 거짓, 거짓이면 참

ᄐ

&&는 논리곱 연산자로, &&연산자 좌측의 수식과 우측의 수식이 모두 성

립되면(참이면) 결과로 1(참)을 돌리며, 그렇지 않을때는 결과로 0(거짓)

을 돌린다. 예를들어 b=3 일때

```
a = (b>2) && (b==3)
```

은 좌측과 우측의 조건이 모두 참이므로 a에 1을 돌린다.(대입한다.)

```
a = (b>2) && (b!=3)
```

은 수식 (b!=3)이 성립하지 않으므로 a에 0을 돌린다.

||는 논리합 연산자로, ||연산자 좌측의 수식과 우측의 수식중 어느 하

나라도 성립되면 논리적인 참의 값 1을 돌린다. 예를들어 b가 3일때

```
a = (b<2) || (b>=3)
```

은 수식 (b>=3)이 성립되므로 a에 1을 돌리며

```
a = (b<2) || (b>5)
```

는 두 수식이 모두 성립되지 않으므로 a에 0을 되돌린다.

!연산자는 논리부정 연산자로 대상수식이 성립되면 거짓의 값 0을, 성립

되지 않으면 참의 값 1을 되돌린다. 예를들어 b=3 일때

```
ᄒI
  a = !(b>2);        a에 0을 돌린다.
  a = !(b<2);        a에 1을 돌린다.
ᄒ
```

```
ᄒIsam0305.dat
    [예제] 논리합 연산자의 사용
    main()
    {
      int i;

      scanf("%d",&i);
      if ( (i>9) && (i<100) ) printf("2jarisu\n");
      if ( !(i%2) ) printf("Jjaksu");
    }
    [결과]
    35
    2jarisu
    Jjaksu
ᄒ
```

1.5 참과 거짓

논리 연산시 식이 성립되면 논리연산식은 결과로 1을 구하는데, 이는 참

을 의미하며 성립되지 않으면 0을 구하는데 이는 0을 의미한다. C-언어

에서는 논리수식 뿐만 아니라 일반 수식에서도 논리가 적용된다. 구체적

으로 수식의 결과가 0이면 이는 거짓으로 간주되며, 수식의 결과가 0이

아니면 참으로 간주된다.

ডI

수치 0..........................거짓

0이 아닌 수치...................참

ড

예를 들어 수식 [5+3]은 결과로 5를 구하는데, 이는 참으로 간주되며,

[3-3]은 결과로 0을 구하는데 이는 거짓으로 간주된다. 위의 예제 중에

서도`if ( !(i%2) )`부분이 이런 내용을 실제로 보여준 예이다.

이는 함수에 있어서도 마찬가지이다. 일부 함수는 함수의 실행이 성공적

이면 0을, 에러가 발생한 경우에는 0이 아닌값을 되돌리는데, 이는 각각

거짓과 참으로 인식된다.

2 기타연산자

C-언어에서는 산술연산자와 논리연산자 이외에도 여러가지 연산자가 사

용 되는데, 여기서는 그중에서도`조건연산자 와 나열연산자`에 대해서

알아보기로 한다.

�.1 조건연산자(3항연산자)

조건식은 결과가 참인가의 여부에 따라 별개의 값을 구하는 연산자 이다

조건연산자는`[?:]`로 표현되는데 그 사용방식은 다음과 같다.

```
ঢl
     조건식 ?      수식1 : 수식2
ঢ
```

이 전체 즉`[조건식 ? 수식1:수식2]`이 하나의 수식으로, 이는 조건식의

결과가 참(0이 아닌수치)이면 수식1의 결과를, 조건식의 결과가 거짓(0)

이면 수식의 결과로 수식2의 결과를 구한다.

설명이 다소 장황하게 느껴지는 사람들을 위해 예를들어 설명해 보도록

하겠다. a=3 일때 다음의 연산식은

```
ঢl
          이 전체가 하나의 연산식
        ------------+--------------
```

```
     a>2   ?   (a+2)   :   (a-2);
     -+-       --+--       --+--
      |          |           |
    조건식     수식1       수식2
```

연산식의 결과로 5(=a+2)를 구한다. 이는 조건식 a>2의 결과가 참이기

때문이다.

아래와 같이 이 연산식을 b에 대입하는 경우에는

```
   b = (a>2   ?   (a+2)   :   (a-2));
```

b에 5가 대입된다.

```
   [예1] 3항연산자를 사용한예
   b = (a>2   ?   (a+2)   :   (a-2));

   [예2] if문 으로 바꾸어 본예
   if (a>2) b=a+2;
        else b=a-2;
```

[예2]는 3항연산자를 사용한 [예1]을 if문으로 바꾸어 본 예이다. 둘 중

에 어떠한 것을 사용하겠냐는 질문을 하면 대부분의 사용자는 그냥 if문

을 사용하겠다고 할것이다.

그럴 수 밖에 없다. 무엇이든 새로운것은 처음에는 낯설게 느껴지기 마련

이다. 이런 희한한 연산자가 C-언어에만 있는만치 이런 표현방법들을 보다 열심히 익혀두는것도 C-언어를 보다 빨리 마스타할 수 있는 방법이 될 수 있다.

2.2 나열연산자(,)

이런 연산자도 있었나 ? 저자도 C-언어 관련서적을 보면서 이런 연산자는 처음 보았다. 아이러니칼 하지만 같이 공부하는 기분으로 배워 보도록 하자. 나열연산자는 각 연산자를 나열하고 좌측에서부터 차례대로 수행되게하는 연산자이다. 사용양식은 다음과 같다.

ㅎI
```
     연산식1, 연산식2, 연산식3 ....
```
ㅎ

프로그램 실행중에 위와 같은 연산식을 만나면 우선 연산식1을 실행하고, 이어 연산식2를 실행하고, 이어 연산식3을 실행하고... 같은 방식으로 실행해 나간다. 예를들어 보자. a=3일때 아래의 연산식은

ㅎI
```
     b=a+3, c=b*2, d=c-2;
```
ㅎ

최종적으로 d에 10을 대입한다. 이때 이 전체 즉`b=a+3, c=b*2, d=c-2;`가 하나의 연산식이다.

자세히 보았으면 알 수 있겠지만 지금까지 설명한 나열연산자는 단순히

산술식을 나열하여 표현하는 기능밖에 없는것 같다. [예1]과 [예2]는

동일하다.
ᄒI@

```
[예1]
b=a+3, c=b*2, d=c-2;
[예2]
b=a+3; c=b*2; d=c-2;
```

ᄒ

그러나 나열연산자 에도 단순히 나열하는 기능이외에 다른 기능이 있다.

다음과 같이 연산식의 결과를 새로운 변수에 대입할 수 있다.

ᄒI

```
e=(a+3, c=b*2, d=c-2);
```

ᄒ

a=3인 경우 위의 연산식의 마지막 연산의 결과인 d의 값 10을 변수 e에

대입한다.

ᄒIsam0306.dat

```
[예제] 나열연산자
main()
{
  int a=3,b=1,c,d,e;  /* 프로그램에서 사용할 변수를 선언한다. */

  e = a+3, c=b*2, d=c-2;
  printf("E-->%d",e);
}
```

ቴ

프로그램의 실행결과를 보면 변수 e의 값으로 10이 표시되어 있음을 알

수 있다.

3 비트연산자

ቭ3.1 비트란 ?

데이타 단원에서 컴퓨터에서 사용하는 모든 정보는 궁극적으로 비트단위

(0 또는 1 둘중의 하나)로 처리된다고

배웠다.

이는 다시말해 컴퓨터를 하드웨어적 견

지에서 보면 컴퓨터가 이해할 수 있는

것은 오직 전압의 높음과 낮음 두가지

인데, 사람들은 편이상 이를`2진수(Bin`

로 나타내고 있다. ঌillu0300.dat

예를들어 십진수로 5를 2진수로 어떻게 사용하는지 알아보자.

십진수 5를 2진수로 표시하면 `0101` 인데 이는 컴퓨터 내부에서

ቯ

0 ………… 전압의 낮음

1 ………… 전압의 높음

ቴ

의 형태로 변환되어 처리된다.

사실 이쯤에서 넘겨짚어야 할 문제가 있다. 지금까지 별 설명없이 십진 수나 이진수등 진수에 대한 이야기를 하였지만 진수의 개념을 이해 못하는 사람들이 있을줄로 안다. 사실 진수의 개념은 중학교 1학년 정규 교과에 포함되어 있다.

기억이 잘 나지 않아서 이해가 되지 않는 사람은 동생이나 아들의 교과서로 진수의 개념을 다시 잡시 바란다. 실제로 컴퓨터 에서는 10진수 보다 16진수의 개념을 사용할 때가 훨씬 많기 때문이다. 16진수는 2진수와 일맥상통 하면서 사람이 이해하기는 더 쉽기 때문이다.

이야기를 본론으로 돌려서 C-언어에는 데이타를 비트 단위로 처리할 수 있는 연산자가 구비되어 있다. 이에 의해 C-언어에서는 하드웨어와 밀접한 각종처리를 용이하게 수행할 수 있는 것이다.

C-언어에서 제공하는 비트연산자에는 아래와 같은것들이 있다.

ㅎl

<table>
<tr><th>연산자</th><th>기능</th><th>사용예</th><th></th></tr>
<tr><td><<</td><td>비트를 좌측으로 이동</td><td>a<<2</td><td>a의 각 비트를 2비트 만큼 좌측으로 이동</td></tr>
<tr><td>>></td><td>비트를 우측으로 이동</td><td>a>>2</td><td>a의 각 비트를 2비트 만큼 우측으로 이동</td></tr>
<tr><td>&</td><td>논리곱</td><td>a&b</td><td>a와 b의 논리곱을 계산</td></tr>
<tr><td>|</td><td>논리합</td><td>a|b</td><td>a와 b의 논리합을 계산</td></tr>
<tr><td>^</td><td>배타적 논리합</td><td>a^b</td><td>a와 b의 배타적 논리합을 계산</td></tr>
<tr><td>~</td><td>비트를 반전</td><td>~a</td><td>a의 각 비트를 반전시킴</td></tr>
</table>

ড

3.2 비트 이동 연산자

ট3.2.1 << 연산자

<<연산자는 대상데이타의 각 비트를 지정 비트만큼 좌측으로 이동 시킨다. 예를들어 a=14 일때 (이진수로 00001110 일때) `a<<2;` 는 a의 각 비트를 2비트만큼 모두 좌측으로 이동시킨다. 이동결과 우측에 공백으로 남는 비트는 0으로 채워진다. 옆의 그림에 이 메카니즘을 도식적으로 표시 하였다.

ঘillu0301.dat

비트 이동 결과 a에는 56이라는 값이 나왔다. 즉 a=14일때 연산식 a<<2 는 결과로 56을 구한다. 이 결과를 변수의 값에 대입할 수 도 있다.

```
b = a<<2;
```

3.2.2 >>연산자

대상 데이터의 각 비트를 비정된 수치만큼 우측으로 이동 시킨다. 예를 들어 a=14일때 (이진수로 00001110 일때)

`a>>2;`

는 a의 각 비트를 2바트 만큼 우측으로 이동 시킨다. 이동 결과 공백으로 남는 비트는 0으로 채워진다. <<연산자와 마찬가지로 변수의 값으로 대입할 수 있다.

3.2.3 곱셈 대신 천이

지금까지의 설명 만으로는 비트 이동 연산자가 어떻게 사용되는 것인지 발견하기가 힘들것이다. 비트 이동 연산자를 사용하기 위해서는 우선 이것들이 어떤곳에 쓰임새가 있는지를 알아보아야 할것이다. 단순이 비트를 이동 시킨다는 설명만으로는 부족할 것이다.

예를들어 a=3일때 (이진수로는 00000011 이다.)

```
    a<<1     00000110       = 6
    a<<2     00001100       = 12
    a<<3     00011000       = 24
    a<<4     00110000       = 48
ㄷ
```

위의 예로 어떤 사실을 발견 했는가 ? 위에서 알 수 있는 사실은 좌측으

로 비트 이동을 해주면 이동을 해준 수만큼`제곱`된 값을 구한다는 사실

이다.

간단하게 이야기 해서 한번 이동하면, (*2,두배)의 값을 구해주고 두번

이동하면 (*4,네배), 세번 이동하면 (*8,여덟배)의 값을 구해준다.

```
ㄷIsam0307.dat
 [예제] 곱셉대신 천이
  main()
  {
     int i;

     printf("input value\n"); /* 숫자를 입력하라는 메시지를 보낸다 */
     scanf("%d",&i);            /* 숫자를 입력받는다. */
     printf("Double value is %d",i<<1  /* 값을 두배로 하여 보여준다 */
  }
ㄷ
```

위의 간단한 예제는 곱셈대신 비트 이동 연산자를 사용한 간단한 예이다

그런데 한가지 궁금증이 생긴다. 바로 곱하기 명령을 사용하여도 되는데 왜 비트 이동 명령어를 사용할까 ? 이유는 간단하다. 곱하기 명령보다는 비트 이동 명령이 빠르기 때문이다.

대개 같은 비트 이동명령은(i<<1) 곱하기 명령(i*2)보다 약 50배가 빠르다.

요즘은 시스템들이 빨라져서 별차이가 없을것 같지만 이런 명령이 수천번 반복된다면 사정은 달라질 것이다.

위와같은 비트 이동 명령은 나눗셈을 하는데도 사용될 수 있다. 예를들어 a=48일때.

```
a>>1     00110000     = 48          a>>2     00011000     = 24
a>>3     00001100     = 12          a>>4     00000110     = 6
```

sam0308.dat

```
[예제] 나눗셈 대신 천이
main()
{
  int i;

  printf("input value\n"); /* 숫자를 입력하라는 메시지를 보낸다 */
  scanf("%d",&i);           /* 숫자를 입력받는다. */
  printf("Half value is %d",i>>1  /* 값을 반으로 하여 보여준다 */
}
```

3.3 비트 논리 연산자

3.3.1 &연산자

&연산자는 논리곱(AND)을 구한다. 이제 배우게 되는 비트 논리 연산자는

초보자들이 이해하기 가장 어려우면서 그냥 넘어갈 수 없는 아주 중요한

부분이다. 아래는 논리곱의 진리표 이다.

```
    X           Y           X & Y
    0           0             0
    0           1             0
    1           0             0
    1           1             1
```

즉 비트논리곱은 두 대응 비트가 모두 1일 때만 1을 구하고 그렇지 않을

때는 결과로 0을 구한다. 다음에 예를 표시하였다.

즉 a=219이고 b=85일때 다음 연산식은 결과로 81을 구한다.

```
    a         11011011 (219)
    b         01010101 ( 85)
    -------------------------
    a & b     01010001 ( 81)
```

연산식의 결과를 다음과 같이 변수에 대입할 수 도 있다.

`c = a & b`

&연산자는 보통 다음과 같은 용도로 사용한다.

[1] 데이터의 특정 비트를 0으로 바꿈 (마스크 오프:Mask off)

[2] 데이터의 최대값을 지정

ㅎ

아래에 몇가지 예를 보였다.

ㅎl

```
      10101010         11100011         01110000         11000011
  & | 00001111     & | 00001111     & | 00001111     & | 00001111
    +---------       +---------       +---------       +---------
      00001010         00000011         00000000         00000011
```

ㅎ

위의 예는 4가지 수치(10101010, 11100011, 01110000, 11000011)를

00001111(십진수로 15)로 &연산하는 예인데, 결과를 보면 0으로 해준 상

위 4비트는 항상 0으로 바뀌어져 있고 하위4비트는 원래의 값을 유지 함

을 알 수 있다.

또한 결과 수치가 모두 15(00001111)이하 임을 알 수 있다.

따라서 위의 예는 특정 비트를 0으로 바꾸는 (0으로 마스크 오프 하는)

예를 보여주는 것이다.

정리해보면 A&B 라고 했을때 얻어지는 값은 항상 B이하이다.

3.3.2 |연산자

|연산자는 비트의 논리합(OR)를 계산한다. 비트 논리합의 진리표는 아래

와 같다.

```
    X          Y          X | Y
    0          0            0
    0          1            1
    1          0            1
    1          1            1
```

즉 비트 논리합은 두 대응비트가 모두 0일때만 결과로 0을 구하고 대응

비트중 어느 하나라도 1이면 결과로 1을 구한다. 다음에 예를 표시하였

다.

```
    a          11011011 (219)
    b          01010101 ( 85)
    ------------------------
    a & b      11011111 (233)
```

즉 a=219이고 b=85일때 a|b식은 결과로 223을 구한다. |연산자는 보통

다음과 같은 용도로 사용된다.

```
    [1] 데이터의 특정 비트를 1로 바꿈 (마스크 온:Mask on)
    [2] 데이터의 최소값을 지정
```

다음에 몇가지 예를 표시하였다.

```
ㅎl
         10101010          11100011          01110000          11000011
    | | 00001111      | | 00001111      | | 00001111      | | 00001111
      +---------        +---------        +---------        +---------
         10101111          11101111          01111111          11001111
ㆆ
```

위의 예는 4가지 수치(10101010, 11100011, 01110000, 11000011)를

00001111(10진수로 15)로 |연산을 하는 예인데,   결과를 보면 1로 해준

하위 4비트는 모두 1로 바뀌어져 있고, 상위4비트는 원래의 값을 유지함

을 알 수 있다.

또 결과수치는 항상 15(00001111)이상이다. 위의 예는 |연산자를 사용하

여 대상데이터의 하위 4비트를 1로 바꾸는(마스크 온하는) 예이다.

a 와 b를 |연산 하는 경우(a|b) 얻어지는 결과는 항상 b이상이다. 앞의

예를 잘 관찰 해보기 바란다.

이와같은 기능을 잘 이용하면 데이터를 특정값 이상되게 할 수 있다.

3.3.3 비트논리 연산자 활용

지금까지 배운 비트 논리 연산자를 사용하면 하나의 바이트에 여러개의 자료를 저장할 수 있다. 이는 이 데이타가 작은 범위를 가지고 있을 때에만 가능한데 1바이트를 반으로 나누어 4비트로 사용한다 하더라도 0부터 15까지의 수밖에 나타낼 수 없기 때문이다.

사실 바이트 하나에 여러개의 데이타를 저장하는 방법은 이외로 많이 사용 된다. 둘로 나누어봐야 1바이트 절약인데, 굳이 이런 복잡한 방법을 사용해야 하는지 의문일지는 모르나 시스템 내부의 바이오스(BIOS)에 데이타를 저장하는데나 응용 프로그램의 데이타 파일 포맷 에는 이러한 방법이 많이 사용되고 있다.

ቼIsam0309.dat

```
[예제] 비트논리 연산자 활용
main()
{
 int i,j;

 printf("Input 1st value (0-15)\n");
 scanf("%d",&i);              /* 값을 i로 읽어들인다. */
 j=i;                         /* j의 상위비트에 값을 저장 */
 printf("Input 2rd value (0-15)\n");
 scanf("%d",&i);              /* 값을 i로 읽어들인다. */
 j=j|(i<<4);                 /* j의 하위비트에 저장 */
 printf("%d, %d",j&0x0f,(j&0xf0)>>4); /* 값을 각각 표시 */
}
```

ቼ

위의 예에서 핵심이 되는 부분은`(j&0x0f) 와 (j&0xf0)>>4`로 각각 j의 상위 4비트와 하위4비트를 구하는 곳이다.

```
     (j & 0x0f)            ......... j의 상위 4비트를 구함
     (j&0xf0) >> 4         ......... j의 하위 4비트를 구함
```

데이타 단원에서 배웠듯이`0x`는 16진수를 나타낸다. 0x0f는 10진수로는

15를 나타낸다. 따라서 (j & 0x0f) 하면 j의 상위 4비트 만이 남고 나머

지는 0으로 마스크 오프 된다.

```
 처음에 5, 나중에 1이 입력 되었을 경우
         하위    상위
       ------- -------
   j = 0 0 0 1 0 1 0 1

 상위를 구하는법 ........ (j & 0x0f)
                              00010101 & 00001111 = 00000101(5)
 하위를 구하는법 ........ (j&0xf0) >> 4
                              00010101 & 11110000 = 00010000(32)
                              00010000 >> 2        = 00000001( 1)
```

3.3.4 ^연산자

^연산자는 비트의 배타적 논리합을 계산한다. 비트의 배타적 논리합(xor

, Exclusive OR) 진리표는 아래와 같다.

ㅎI

X	Y	X | Y
0	0	0
0	1	1
1	0	1
1	1	0

ㅎ

즉 배타적 논리합은 두 대응비트가 서로 다를 때는 결과로 1을 구하고,

서로 같을때는 결과로 0을 구한다. 아래에 예를 표시하였다.

ㅎI

```
a        11011011 (219)
b        01010101 ( 85)
   --------------------
a^b      10001110 (142)
```

ㅎ

즉 a=219이고 b=85일때 아래의 연산식은 결과로 142를 구한다.

`a^b;`

연산식의 결과를 다음과 같이 변수의 값으로 대입할 수 도 있다.

`c=a^b;`

^연산자는 보통 다음과 같은 용도로 사용된다.

ㅎI

    [1] 데이터의 특정 비트를 반전 시키고자 할때

ㅎ

다음에 몇가지 예를 표시하였다.

ㅎI

```
     10101010          11100011          01110000          11000011
 || 00001111      || 00001111      || 00001111      || 00001111
   +---------        +---------        +---------        +---------
     10100101          11101100          01111111          11001100
```

ㅎ

결과를 보면 0으로 ^해준 상위4비트들은 모두 원래의 값을 유지하고 있

으나 1로 ^해준 하위 4비트들은 모두 원래비트의`반대`비트로 바뀌어져

있음을 알 수 있다.

위의 예는 대상데이터의 하위 4비트를 반대비트로 토글시키는 예이다.

3.3.5 ~연산자

~연산자는 대상 데이터의 모든 비크를 반대 비트로 바꾸어 준다. (비트의 반전) 비트 반전의 진리표는 아래와 같다.

```
    X       ~X
    1        0
    0        1
```

아래에 예를 표시하였다.

```
     | 11011011 (219)
   ~ +---------------
       00100100 (36)
```

즉 a=219일때`~a`는 결과로 36을 구한다. 연산식의 결과를 다음과 같이 변수의 값으로 대입할 수도 있다.

```
    b=~a;
```

4. 비트 대입 연산자

비트 대입 연산자에 대해서는 산술 연산자의 대입 연산자를 설명하면서

잠시 언급한바가 있는데 이는 비트 연산자와 관련이 깊기 때문에 설명을 뒤로 미룬것이다.

비트 대입 연산자는 비트 연산의 결과를 변수의 값으로 대입하는 연산자이다. 사실 비트 연산을 실제로 사용하는데에는 비트 연산을 하여 다른 변수에 넣기 보다는 자기 자신에게 할당하는 것이 일반적이다. 따라서 비트 대입연산자는 꽤 중요하다고 할 수 있다.

a=3일때 아래의 [예1]과 [예2]는 동일한 결과인 6을 a자신에 얻는다.

```
[예1]........... a = a << 1;
[예2]........... a <<= 1;
```

위의 예만 보고도 비트 대입연산자에 대해서 대충 감을 잡았다면 당신은 뛰어난 프로그래머가 될 재질이 있다. [예2]가 실제로 대트 대입연산자를 사용한 예인데 주의할 점은 아래와 같이 해서는 안된다는 점이다.

```
[잘못된 예] .......... a<=1; (a는 1보다 작거나 같다는뜻)
```

연산자	기능	사용예	다른표현
<<=	좌측 이동후 대입	a<<=2	a=a<<2;
>>=	우측 이동후 대입	a>>=2	a=a<<2;

&=	논리곱 대입	a&=10	a=a&10;
|=	논리합 대입	a|=10	a=a|10;
^=	배타적 논리합 대입	a^=10	a=a^10;

ҕ

이들 비트 대입연산자들의 기능은 기본적으로 전에 설명한 산술 대입 연산자와 같다. 다만 수행되는 연산이 다를 뿐이다. 산술 대입 연산자와 마찬가지로 비트 대입 연산자 사용시에도`대입연산은 변 단위로 수행`된다는 사실에 주의할 필요가 있다. 예를들어
`A <<= B + 2`
는 [A = (A<<B)+2)]의 기능을 수행하는 것이 아니라 [A=A<<(B+2)]의 기능을 수행한다는 것이다.

ҕIsam0310.dat

```
 [예제] 비트 연산자 사용예
   main()
   {
     unsigned int i;

     printf("Input value\n");
     scanf("%d",&i);

     printf("(i>>2) ---> %d\n",i>>2); /* 우측 이동 */
     printf("(i<<4) ---> %d\n",i<<4); /* 좌측 이동 */
     printf("(i&10) ---> %d\n",i&10); /* 10으로 논리곱 연산 */
     printf("(i|10) ---> %d\n",i|10); /* 10으로 논리합 연산 */
     printf("(i^10) ---> %d\n",i^10); /* 10으로 배타적 논리합 연산 */
     printf("(  ~i) ---> %d\n",~i);    /* 비트 반전 */
   }
```

ҕ

위 예제의 결과는 여러분이 직접 작성하여 보고 결과를 확인하기 바란다
지금까지 배운 비트 연산의 총복습 이라고 할 수 있다.
5. 캐스트 연산자 (형명)

캐스트 연산자의 특징

1. 단항 연산자이다.
2. 피연산자로 임의의 산술형 데이터와 수식을 취한다.

3. 단항 연산자이므로 결합순서는 <==이다.

캐스트 연산자는 명시적(explicit)인 형변환이 필요할 때 쓰인다. 캐스트 연산자의 일반적인 형태는 다음과 같다.

(형명)

이 때 형명(type name)의 앞뒤에 반드시 괄호가 필요하다. 터보 C의 모든 데이터형(포인터형 포함)을 형명으로 쓸 수 있다. 예를 들어 산술형의 캐스트 연산자는 다음과 같은 것이 있을 수 있다.

তI@
```
 터보-C 에서 가능한 캐스트 연산자

    (char)                    (unsigned char)
    (int)                     (unsigned)
    (long)                    (unsigned long)
    (float)                   (double)
    (void)
```
ত

그리고 포인터형의 캐스트 연산자가 있을 수 있다. 예를 들면,

তI
```
    (unsigned char *)    (int *)
    (char *)              (char (*)[])
    (char (*)())
```
ত

등등

그 밖에 프로그래머가 typedef문으로 정의한 데이터형이나 구조체, 공용체에 대한 캐스트 연산자도 가능하다.

`[예] (struct myblock *)`

다음 예제는 정수 640을 정수 7로 나누려는 것인데, 그 나눗셈을 부동 소수점 연산으로 수행하고자 한다.

ᄚIsam0311.dat
[예제]
```
void main(void)
{
   int m=640, n=7;

   printf("%f\n" , (float)m / (float)n);
}
[결과]
91.428571
```
ᄚ

C-언어의 캐스트 연산자는 모든 데이터형간의 형변환이 가능하고, 심지어는 포인터형간의 형변환도 가능하다. 이에 대해서는 나중에 관계있는 부분에서 다시 알아보도록 하자.

6. sizeof 연산자

sizeof 연산자의 특징

1. 단항 연산자이다.
2. 피연산자는 임의의 산술형 데이터나 수식이다. 그리고 괄호 안에 형명을 쓸 수도 있다. 결과는 정수 상수이다.

        서식: sizeof 수식
               sizeof (형명)
  3. sizeof 연산자라고 부른다.
  4. 단항 연산자이므로 결합 순서는 <==이다.

sizeof 연산자는 피연산자의 전체 바이트 크기를 알아낸다. 예를 들어 sizeof 연산자의 피연산자가 배열일 경우에는 배열의 전체 바이트 크기가 구해진다. 즉 아래의 두 수식은 같은 결과값을 가진다.

```
    sizeof 배열명 == 배열크기 * sizeof (배열요소형명)
```

sizeof 연산식의 결과가 "정수 상수"라 하는 것은, sizeof 연산식이 프로그램을 실행할 때가 아니라 컴파일하는 도중에 이미 계산이 끝니 버린다는 것을 뜻한다.

sizeof 연산자는 메모리 할당 루틴이나 입출력 시스템간의 정보 교환에 주로 쓰인다. 그리고 sizeof 연산자를 사용할 때에는 피연산자를 괄호 `'('`로 둘러싸는것이 좋다는것을 알아두자. 물론 꼭 그렇게 할 필요는 없지만 이러한 관례를 따르는 것이 알아보기가 쉽다.

참고로 포인터는 크게 2바이트 크기의 근거리 포인터(near pointer)와 4바이트 크기의 원거리 포인터(far pointer)로 나눠는데, 우리가 언급하는 포인터는 모두 근거리 포인터뿐이다. 따라서 아직까지는 포인터는 2바이트 크기를 가진다고 봐도 무방하다.

ᇹIsam0312.dat
[예제] sizeof 연산자의 사용 예

```
  void main(void)
  {
      char c, s[100]; int n, array[100];
      float x; double xx;

      printf("%d\n", sizeof(-1));            /* 2                */
      printf("%d\n", sizeof(0xffff));        /* 2, int type      */
      printf("%d\n", sizeof(65535));         /* 4, long type     */
      printf("%d\n", sizeof(32767));         /* 2, int type      */
      printf("%d\n", sizeof(c));             /* 1                */
      printf("%d\n", sizeof(s));             /* 100 * 1 = 100    */
      printf("%d\n", sizeof(n));             /* 2                */
      printf("%d\n", sizeof(array));         /* 100 * 2 + 200    */
      printf("%d\n", sizeof(x));             /* 4                */
      printf("%d\n", sizeof(xx));            /* 8                */
  }
```

ᇹ
7 산술 변환 규칙

C-언어 에서는 데이터형을 혼용할 경우에 자동적으로 어느 한 가지 형으

로 통일한 다음 연산을 수행한다. 즉 자동으로 데이터형을 변환해준다

는 말이다. 이처럼 데이터의 혼용이

자유로운 것은 C의 강력한 장점 (융

통성)인 종시에 꼼꼼하지 못한 초보자

에게는 무시 못할 문제가 된다. 다시

말해서,

C에서는 데이터형을 자유롭게 혼용하여

쓸 수 있는 대신에 데이터형을 실수로 illu0302.dat

잘못 혼용했을 때의 결과에 대한 책임이 전적으로 사용자에게 있다.

이 점을 프로그래머는 분명히 인식하고 있어야 한다.

여러분들이 먼저 알고 있어야할 산술 변환 규칙에는 다음과 같은 것들이

있다.

[1] 수식 내에서 정수형 또는 double형이 아닌 데이터형은 무조건 아래

의 표와 같이 일단 먼저 변환이 된다. (단, 별다른 컴파일러 옵션을

지정하지 않았을 경우).

l@

[표] 수식 내에서의 데이터형 변환

데이터형	변환되는 형	변환 방법
char	int	부호확장(sign extension)
unsigned char	int	상위 바이트를 0으로(zero-filled high byte)
enum	int	만약 부호없는 형이면 unsigned형으로 변환
float	double	확장되는 가수부(matissa)를 0으로

채운다.

ㄷ

위와 같이 변환이 끝나면 double, unsigned long, long, unsigned, int 형만이 남게 된다.

[2] 그 다음으로 이항 연산자 (또는 삼항 연산자)의 두 피연산자가 서로 데이터형이 일치하지 않을 때에는 두 피연산자의 데이터형을 비교해서 산술 변환 우선순위가 낮은 쪽을 우선순위가 높은 쪽의 데이터형으로 변환한다(이 때 우선순위는 연산자의 연산순위가 아니다. 혼동하지 말 것. 이러한 변환을 promotion이라고 한다.

[3] 대입식에서 =의 우변에 있는 수식의 최종 결과값은 좌변의 데이터형으로 변환된 후 대입된다. 이 과정에서는 앞서 말한 promotion이 일어날 수도 있고, 그 반대로 산술 변환 우선순위가 높은 데이터형이 우선순위가 낮은 형으로 변환되는 demotion이 일어날 수도 있다.

` (예: c가 char형 변수로 선언되어 있는 경우에`

` c = 0x3456라고 하면 상위 바이트 0x34가 잘려 나간다.)`

[4] 함수를 호출할 때에도 실매개변수가 함수로 전달되면서 형변환이 일어난다. 실매개변수가 수식이라면 앞의 규칙에 따라서 그 수식이 평가된 후 변환된다. 만약 실매개변수가 수수기이 아닌 하나의 변수이거나 상수라고 해도 항상 위의 [1]번 규칙에 따라서 먼저 변환된

다음에 함수로 전달된다.

그래서 함수의 실매개변수가 문자형이나 float형인 상태 그대로 함수에 전달되는 일은 결코 없다. 일단 실매개변수가 int형이나 double형 등으로 변환된 다음 함수로 전달된다.

그 후 형식매개변수로 쓰일 때 필요하면 다시 문자형이나 float형으로 재변환된다.

<참고> 변환되는 유형을 정리해 보면 모두 세 가지가 있음을 알 수 있다
첫째 promotion, 둘째 demotion, 세째 개념적인 변환.

1. promotion은 부호확장(sign extension)에 의한다. 즉 변환되기 전의 데이터형이 부호있는(signed) 형이고, 그 값이 음수(MSB가 1)이면 2의 보수 형태를 그대로 유지하기 위하여, 확장되는 비트를 1로 채운다. 부호있는 형이고, 0 이상의 수치(MSB가 0)이면 확장되는 비트를 0으로 채운다. 반대로 부호없는 형이면 확장되는 비트를 언제나 0으로 채운다. 한편 정수형이 부동형으로 변환될 경우에는, 정밀도에 극히 미미한 손실이 있을 수 있다.

2. demotion이 일어나면 변환되기 전 데이터의 상위 비트가 잘려나가서 (truncate) 원치않는 결과가 생길 수도 있으므로 상당히 주의해야 한다.

3. 부호 있는 형이 부호없는 형으로 변환되는 경우나 그 역과정은 다만

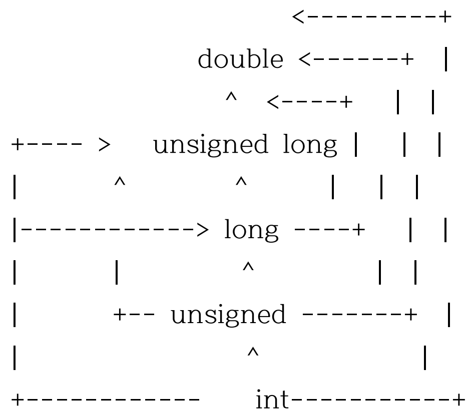
개념적인 것이며, 따라서 실제 비트 구조에는 아무런 변환도 없음에

유의하기 바란다. 그러나 대소비교를 할 때는 이런 변환이 커다란 문

제점이 될 수도 있다.

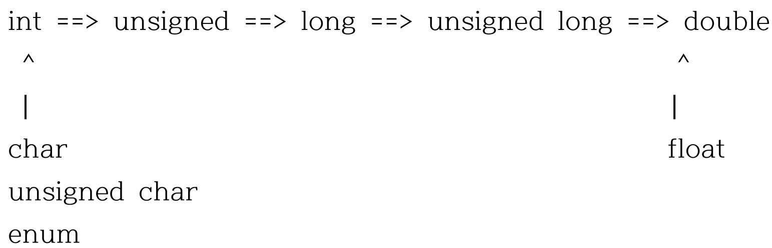
ㅎl@

데이터형의 산술 변환 우선순위



ㅎ

ㅎl



ㅎ

위의 그림에서 실선은 두 피연산자의 데이터형간의 모든 가능한 조합을

나타내는 것이고, 화살표 끝은 실선의 양끝에 연결되어 있는 두 데이터

형 중에서 산술 변환 우선순위가 더 높은 쪽을 가리킨다.

피연산자가 둘 이상일 때는 우선순위가 낮은 쪽 데이터형이 우선순위가

높은 쪽으로 자동 변환된다. 그러므로 데이터형을 혼용한 수식에서는 반드시 데이터형의 변환이 올바른 결과를 가져올 것인지에 대해서 미리 심사숙고해야만 한다.

ნl@

[정리] C-언어 연산자의 연산순위

	순위	터보 C
	1	() []
구조체 연산자	1	.
구조체 포인터 연산자	1	->
부호 연산자	2	+ -
논리부정 연산자	2	!
1의보수 연산자	2	~
증감 연산자	2	++ --
캐스트 연산자	2	(형명)
간접지정 연산자	2	*
번지 연산자	2	&
sizeof 연산자	2	szeof
산술 연산자	3	* / %
	4	+ -
쉬프트 연산자	5	<< >>
관계 연산자	6	< <= > >=
상등 연산자	7	== !=
비트 AND 연산자	8	&
비트 XOR 연산자	9	^
비트 OR 연산자	10	|
논리곱 연산자	11	&&
논리합 연산자	12	||
배타적 논리합 연산자		^
조건 연산자	13	? :
대입 연산자	14	=
	14	op=
쉼표 연산자	15	,

ნ

ნl

. 위의 표에서 숫자는 연산순위를 나타낸다.
. op=는 아래의 10가지 연산자를 총칭한다.
+= -= *= /= %= <<= >>= &= ^= |=

ㅎ

표에 나열된 45개의 연산자 외에도 연산순위라는 개념이 없는 #, ##,

defined등 3개의 전처리기 연산자가 더 있다.

전처리기 연산자는 기회가 있을때 설명하기로 한다.

서론

흔히들 C-언어를 저급언어와 고급언어의 양면성을 가지고 있는 언어라고

들 한다. C-언어의 저급언어적 특성에 관해서는 전 단원인 연산자에서의 비트 연산자등의 활용을 보면 알수있다.
이제 C-언어의 고급언어적 특성에 관해 공부할 차례이다. C-언어의 고급언어적 특성을 이야기하면 C-언어의 제어구조를 빠뜨릴 수 없다.

C-언어에서는 다른 고급언어(Pascal, Basic)등에서 사용하는 거의 모든 제어구조를 사용한다. 게다가 조금더 세련된 느낌으로 사용할 수 있는 점도 없지않다.

1. if 블럭

if-else문을 알아보기에 앞서 우선 복문(compound statement), 즉 블럭의 구조 부터 살펴보자. 나중에 다시 한 번 완전하게 설명하겠지만, 일반적인 블럭 구조는 아래와 같다.

ㅎl

블럭 구조

```
{
  문장1;
  문장2;
  ......
  문장n;
}
```
ট

제어문이나 루프는 너무나도 낯익은 것이므로 그 개념이나 원리에 대해서는 이미 여러분이 충분히 알고 있다고 가정하고 곧바로 본론으로 들어가겠다. 제어문(control statement)은 다음과 같이 모두 9개가 있으며 선택문, 순환문(반복문), 점프문 등 크게 3가지로 분류된다.

টI

```
C-언어의 3가지 제어문

               +-- 선택문 : if-else문, switch문
  제어문       |    (selection statement)
control        |
statement      |-- 순환문 : while문, do-while문, for문
               |     (iteration statement)
               |
               |    점프문 : break문, continue문, goto문, return문
               +-- (jump statement)
```
ট

C-언어에서는 세미콜론(;)이 문장의 끝임을 나타내기 때문에 각 문장의

뒤에 반드시 세미콜론을 붙여주어야 한다. 그러나 블럭의 시작과 끝을

나타내는 "{","}" 바로 뒤에 세미콜론이 전혀 쓰이지 않는다.

그 이유는 위에서 설명한 바와같이 C-언어 에서는 반드시 하나의 문장뒤

에만 세비콜론(;)을 붙이며 "{"나 "}"역시 문장의 일부로 간주되기 때문

이다.

if 문을 사용하는데 주의하여야 될 사항은 아래와 같다.
ㅎI
[1] C-언어에서는 if문의 조건식을 반드시 괄호로 묶어야 한다. 의심나
면 묶지않고 컴파일을 해보기 바란다.

`[예] if i==3 printf("NO");`

[2] 사실 위의 [1]번도 주로 파스칼을 사용하던 사람에게 해당되는 것이
고 또하나 파스칼을 배운 사람이 주의하여야할 것은 if문 뒤에
then을 붙이지 않는다는 것이다. 저자도 if문 뒤에 then을 쓰는 습
관이 아직도 가끔씩 발동(?) 하여 컴파일 에러를 내기도 한다.

[3] C-에서는 참과 거짓의 개념을 0은 거짓, 0이 아닌 수는 참으로 가지
고 있으므로 수식이나 변수를 직접 if문 조건식에 사용할 수 있다.
C-언어는 참 신기한 언어이다.
ㅎ

이미 다알고 있는 사실이지만 (모르는 사람도 있을까 ?) if 블럭은 참/

거짓 여부에 따라서 별개의 처리를 수행시킨다. 일반적인 사용방식은 옆

의 그림과 같다.

if문은 모든 컴퓨터 프로그래밍언어를

배울때 화면에 문자를 출력하는 방법

을 배운다음에 제일처음 대하는 것이

기도 하다.

그림에서 판별식은 (a>3), (i==5) 등의

수식이 될 수도 있으며 상수나 변수 자

체가 될 수도 있다.　　　　　　　　ঘillu0400.dat

`[예] if (i) printf("OK")`

위의 예는 i가 0이 아니면(참이면) OK

를 화면에 표시한다.

가장 간단한 if블럭은 아래와 같이 단순이 판별식에 따라 한가지 문장을

수행하는 블럭이다.

তI
```
    [1] if (판별식) 문장;

            또는

    [2] if (판별식)
            문장;
```
ত

위의 [1]번과 [2]번은 동일하다. [2]번은 단순이 문장을 아래줄에 배치

시켜 놓은것 뿐인데 이런예를 보인 이유는 초보자들이 범하기 쉬운 한가

지 실수를 보여주기 위해서 이다.

তI
```
    if (SU==3);
        printf("SU is 3");
```
ত

위의 예제가 올바르게 동작할까 ? 어디가 잘못되었는지 금방 눈에 띄는가 ? 위에서도 세미콜론은 (;) 한문장의 끝에만 붙여준다고 하였다. 우리들이 사용하는 언어의 감각을 사용하여도 "X하면 X한다"라는 if블럭은 한 문장이다.

그러나 위의 예는 if문과 수행문이 분리되었다고 해서 if문 판별식 뒤에 세미콜론(;)을 붙이는 잘못을 저질렀다.

먼저 아래를 보자.

```
ㅎI
      if (SU==3);
ㅎ
```

"만약에 SU가 3이면" 하고 문장이 끝났다.   물론 있을수 없는 일이지만 세미콜론이 붙었으므로 C는 한문장이 끝났다고 생각한다. 그러면 만약에 SU가 3이면 어떻게 하라는 것인가 ?

위의 문장은 어떻게 하라는것이 없으므로 있으나 마나한 것이된다.

그리고 아래의

```
ㅎI
      printf("SU is 3");
ㅎ
```

printf()함수는 if문과는 별도의 문장으로 해석되어 실행된다. 즉 if문이 있으나 마나하게 된다.

초보자는 이러한 실수를 몰라서 범할 수 있으나 어느정도 아는 사람들은 실수로 가끔 이런일을 보게된다.

그다음 볼 수 있는 if블럭은 if문 뒤에 블럭이 오는 경우이다.

```
ㅎI
    if (판별식)
    {
      문장1;
      문장2;
      문장n;
    }
ㅎ
```

사실 판별식 뒤에 단하나의 문장이 오는 경우는 드물므로 블럭을 사용하

여 if문의 판별식이 성립할때 실행할 문장을 블럭으로 지정한다.

1.1 else문

else문은 바로 전의 짝짓지 못한 if문과 짝지어져서 if문의 판별식이 거

짓일 경우에 실행된다.

```
ㅎIsam0400.dat
    [예제] if와 else를 사용한예
    main()
    {
      int i;

      scanf("%d",&i);                /* 키보드로 부터 i를 읽어들인다. */
      if (i>0) printf("yang-su");
      else      printf("um-su");
```

```
}
```

위의 예제는 키보드로부터 숫자를 읽어들여 양수인지 음수인지 표시하여

주는 예제인데 if와 else의 사용법을 가르켜준다.

위의 예제는 정상적으로 동작하는데 한가지 버그가 있다. i가 0보다 크

면 "yang-su"를 표시하고 그 이외에는 "um-su"를 표시한다. 언듯보면 이

상이 없는것 같지만 0을 입력하면 i가 0보다 크지 않으므로 else문이 실

행되어 화면에는 엉뚱하게 "um-su"가 표시된다.

```
    버그수정 ?

  if (i>0)  printf("yang-su");
  if (i==0) printf("zero");
        else printf("um-su");
```

0은 음수가 아니므로 예제를 고쳐보라고 한다면 초보자 10명중 5명은 위

와 같이 작성할것이다.

과연 올바르게 고쳐진 것일까 ?

결과를 알아보기 위해 컴퓨터가 하는일을 우리가 해서 한번결과를 알아

보도록 하자.

먼저 i=3일때 첫번째 if (i>0)은 성립된다. 그래서 화면에 "yang-su"가

표시된다. 여기까지는 문제가 없다. 그다음 아래줄의 if (i==0)은 성립

이 되지 않는다. 그래서 "zero"는 표시되지 않는다.

여기까지도 문제가 없다. 그런데 다음줄 else에서 문제가 생긴다.

তl

```
  if (i==0) printf("zero");
        else printf("um-su");
```

ত

else문이 가장 가까운 짝지워지지 않은 if문과 결합한다고 했으므로 위

의 if (i==0)가 성립되지 않으므로 화면에 "um-su"역시 표시된다. 따라

서 3을 입력하면 "yang-su","um-su"가 모두 표시된다. 아래는 버그수정

이 올바르게된 예이다.

তl

```
     버그수정

  if (i>0)  printf("yang-su");
  if (i==0) printf("zero");
  if (i<0)  printf("um-su");
```

ত

1.2 if문의 판별식

if문에서 사용하는 판별식은 수식, 변수, 상수등이 사용될 수 있다고 하였다. 그러면 if문에 사용되는 판별식의 종류와 사용법을 알아보도록 하자.

[1] 관계연산자

비교를 할때 쓰이는 관계연산자가 if문의 판별식에 사용될 수 있다.

তl

관계연산자	의미	사용예
<	작다.	if (i<5) i가 5보다 작으면
<=	작거나 같다.	if (i<=5) i가 5보다 작거나 같으면
==	같다.	if (i==5) i가 5이면
>=	크거나 같다.	if (i>=5) i가 5보다 크거나 같으면

> 크다. if (i>5) i가 5보다 크면
!= 같지않다. if (i!=5) i가 5가 아니면
ᇹ

ᇹIsam0401.dat
```
[예제] if문과 관계연산자 3
main()
{
  int age;

  printf("Input your age\n");
  scanf("%d",&age);
  if (age>60) printf("you are old man.");
}
```
ᇹ

위의 프로그램은 나이를 입력받아서 나이가 60이상이면 "you are old ma

n."을 출력시켜주는 프로그램이다.

위의 프로그램은 60세 이상이면 메시지를 "당신은 노인" 이라고 메시지

를 보내지만 10대일 경우 "당신은 10대"라고 메시지를 보내는 경우를 생

각해 보자. 나이가 10보다 크거나 같고 20보다 작으면 10대이다. 이런경

우 관계연산자가 두개가 사용되야 하고 두개의 판별식이 모두 만족되야

한다.

ᇹIsam0402.dat
```
[예제] if문과 관계연산자 2
main()
{
  int age;

  printf("Input your age\n");
  scanf("%d",&age);
  if ( (age>=10) && (age<20) ) printf("you are teenage.");
}
```

ნ

위 예제는 두개의 관계연산자를 논리연산자로 묶어서 사용하는 법을 보

여주고 있다. 논리연산자는 전단원인 "연산자"에서 배운바 있지만 복습

하는 의미에서 다시한번 공부해보자.

ნl

논리연산자	의미	사용예
&&	논리곱(AND)	if ((i>5) && (i<10)) 만약 i가 5보다 크고 10보다 작으면
||	논리합(OR)	if ((i>5) || (i<10)) 만약 i가 5보다 크거나 10보다 작으면
!	논리부정(NOT)	if (!(i>5)) 만약 i가 5보다 크지 않으면

ნ

논리곱은 두개의 수식이 모두 참이면(0이 아니면) 참을 돌리고 하나라도

거짓이면 (0이면) 거짓을 돌린다.

ნl

```
if ( (5>2) && (4<8) )    ............ 참
if ( (5>2) && (4>8) )    ............ 거짓
if ( (5>2) && (0) )      ............ 거짓
```

ნ

논리합(||)는 두수식중 하나라도 참이면(0이 아니면) 참을 돌리고 둘다
거짓이면 (0이면) 거짓을 돌린다.

ნl

```
if ( (5>2) || (4<8) )    ............ 참
if ( (5<2) || (4>8) )    ............ 거짓
if ( (5<2) || (0) )      ............ 거짓
```

ნ

논리부정(!)는 앞의 수식이 참이면 거짓을 돌리고, 거짓이면 참을 돌린다.

তl

```
if ( !(5>2) )      ..................... 거짓
if ( !(5<2) )      ..................... 참
if ( !(0) )        ..................... 참
```

ত

[2] 수식

if문에는 수식을 사용할 수도 있다고 하였다. 수식을 사용하면 수식의

결과를 판별식으로 사용한다. 즉 수식의 결과가 참이면(0이 아니면) 참

이되고 거짓이면(0이면) 거짓이된다.

তlsam0403.dat

```
[예제] 수식을 판별식으로 사용한 if문
main()
{
  int age;

  printf("Input your age\n");
  scanf("%d",&age);
  if (!(age-10))   printf("your age is 10.");
              else printf("your age is not 10.");
}
```

ত

위의 예제는 입력한 나이가 10이 아니면 "your age is not 10."을 출력

하고 10이면 "your age is 10."를 출력하는 프로그램이다.

if (!(age-10))에 11을 입력했다고 가정하자. 결과는 아래와 같이 표시

될 수있다.

`if (!(age-10)) ---> if (!(11-10)) ---> if (!(1)) ---> if (0)`

즉 (!(age-10))의 결과가 0이 아닌수를 돌리므로 이 수식은 참이 된다.

위의 예제에서 사용된 if블럭은 아래와 같이 고칠 수도 있다.

ㅎI

```
    if (age==10)   printf("your age is 10.");
                 else printf("your age is not 10.");
```

ㅎ

[3] 상수

if문의 판별식에는 상수가 사용될 수 도 있다.

ㅎI

```
    if (1)      ............ 항상 참
    if (0)      ............ 항상 거짓
```

ㅎ

0이 아니면 참이라는 것은 뒤에 못이 박히도록 설명했으므로 더 이상의 설명은 피하겠다. 다만 0이 아닌수가 참이라고 했으므로 음수도 참이된다.

ㅎI

```
    if (-5)     ............ 항상 참
```

ㅎ

2. switch 선택문

2.1 일반적인 switch문

if-else문을 가지고는 양자택일밖에 할 수가 없다. 다중택일을 하자면 if-else문을 여러개 사용하여야 한다. 그런데 if-else if-else문을 어느 한도 이상으로 많이 중첩하면 문장이 대단히 복잡해져서 이해하기가 힘들어진다. 물론 많은 if문을 중첩하는 것이 불가능하다는 것은 아니다.

```
복잡한 if문 구조
if (i='a') printf("animal");
if (i='b') printf("bird");
if (i='c') printf("cat");
if (i='d') printf("dog");
if (i='e') printf("energy");
if (i='f') printf("fly");
```

이럴 때는 다중택일 전용으로 마련된 선택문을 사용하는 것이 여러모로 편리하다. C-언어 에서는 switch 라는 다중택일만을 전용으로 담당하는 선택문이 있다.

switch문의 일반적인 구조는 다음과 같다.

```
switch (수식) {
   case 상수1: 문장;
               ......
               break;
   case 상수2: 문장;
               ......
               break;
   .
   .
   default:   문장;
               ......
               break;
}
```

switch문 에서 미리 알아두어야할 사항들은 다음과 같다.

* 수식은 주로 하나의 변수이다.
* 맨 마지막에 나오는 break문은 생략이 가능하나 관례상 생력하지 않는 것이 좋다.
* 그 외의 나머지 break문은 특별한 경우에 한해서 생략하기도 한다.
* 마지막의 default문은 필요에 따라 생략이 가능하다.
* default문을 꼭 마지막에 둘 필요는 없다
* case레이블은 goto문의 레이블과 마찬가지 역할을 한다.

switch문은 일단 수식을 평가한 뒤에 그 결과를 가지고 맨 처음의 case

레이블 부터 차례대로 비교해 나간다. (여기서는 상수1: 상수2: ......

식으로). 그러다가 수식의 결과값과 case 레이블이 일치할 경우, 그 레

이블 바로 옆에 딸려 있는 문장부터 실행한다.

실행하다가 break문을 만나면 즉각 switch 블럭을 탈출하고, break문을

만나지 못하면 (다른 case 레이블이 있거나 말거나) 계속해서 이어지는

문장들을 실행해 나가다가 결국에 switch 블럭의 끝"}" 에 다다르면 그

제서야 그 블럭을 벗어난다.

break문이 없으면 레이블 다음의 모든

문장이 연속적으로 실행되므로 이경우

에 원치 않는 결과가 발생할 수도 있으

므로 주의하기 바란다 (이 점을 역이용

하는 수도있기는 하다.

따라서 각각의 case 레이블이 나타내는 ☜illu0401.dat

범위의 끝에는 반드시 break문을 넣어

두도록 한다

(defalut: 레이블의 경우도 역시 넣어

둔다).

수식의 값을 모든 case레이블과 일일이 비교해 보아도 일치하는 case레이블이 없을 경우에는 default: 레이블에 딸린 문장을 실행한다. 그런데 만약 사용자가 default: 레이블을 아예 생략해 버린 경우에는 결국 switch문은 아무런 일도 하지 않고 종료된다.
그리고 switch문의 수식과 case 레이블에는 다음과 같은 제약이 따르고 있다는 사실도 주의해야 한다.

[1] 수식

수식은 반드시 정수형 데이터나 정수형으로 변환되는 데이터형의 값을가져 야만 한다. 즉 수식은 int, unsigned, char, unsigned char, enum형의 변수 또는 산술 변환규칙에 의해 정수형의 결과를 가질 수 있는 수식

이어야 한다.   물론 캐스트 연산자를 쓸 수도 있다.   그러므로 float, double, 문자열 상수, 포인터 등등을 사용해서는 안 된다.

[2] case 레이블

case 레이블 자체는 반드시 문자 상수, 정수 상수 등의 상수 수식 (cons tant ex-pression)이어야 한다. case 레이블에 변수는 안 된다. 부분범위를 지정하는 것도 불가능하다. 부분범위를 사용하고 싶을 때에는 편법을 사용하는것도 가능하나(나중에 설명), 부분범위의 갯수만큼 case레이블을 작성해야 하므로 아무때나 사용하기 에는 곤란하다.
이럴때는 어쩔 수 없이 눈물을 머금고 if-else문을 사용해야만 한다.

다음과 같은 if-else if-else문은 그 조건식이 범위를 조사하는 것이 아니라, 변수와 상수의 일치 여부만을 따지는 것이므로 switch문으로 변환하는 것이 가능하다.

```
ҕl
    if ((c = getch()) == '1') prmode(); /* 사용자가 만든 함수들 */
    else if (c == '2') linspc();
    else if (c == '3') {
 .............................. 계속 ............................
ҕ

ҕl
        miscel();
        fmtctl();
    } else if (c == '4') initpr();
    else if (c == '5') texpr();
    else if (c == '\x1b') quit();
    else {
        printf("Error\n");
        checkerror();
    }
ҕ
```

와 한눈으로 봐도 복잡하다. 이런 복잡한 프로그램의 구조는 단순히 프로그램을 작성한 프로그래머가 알아보기 힘들어서가 아니라 나중에 프로그램을 수정하거나 보강할 때 상당한 골치거리를 안겨주게 된다. 다시한

번 주지 하건데 프로그램을 가장 간단하게 작성할 수 있는 프로그래머가 최고의 프로그래머 이다. 이는 빈말이 아니다. 말뜻 그대로 이해해도 좋다.

저 복잡한 if블럭을 switch문을 사용하여 수정하면 아래와 같은 구조가 된다.

ㅎl

```
    switch (c = getch())
    {
      case '1':      prmode(c); break; <- 빠뜨리지 않도록 조심한다.
      case '2':      linspc(c); break;
      case '3':      miscel(c); fmtctl(c); break;
      case '4':      initpr(c); break;
      case '5':      textpr(c); break;
      case '\x1b': quit(c);     break;
      default:       printf("Error\n");
                      checkerror(c);
                      break;  <- 생략 가능하지만 그대로 두는 것이 좋다.
    }
```

ㅎ

위의 예를 보면 역시 switch문이 다중if문보다 한결 알아보기 쉽다는 것을 알 수 있다. 그리고 위의 예에서 맨 마지막에 나오는 default 레이블의 break문은 원래 불 필요한것 이지만 생략하지 말고 그대로 놔두는 것이 좋다. 그렇게 해두면 차후에 또다른 case 문을 첨가할 경우 앞에 나오는 case문과 서로 구분 짓기 위해서 꼭 필요한 break문 을 실수로 빠뜨릴 염려가 없어지기 때문이다.

ㅎl

[관례] switch문의 마지막 break문은 사실상 불필요한 것이지만 생략하지 말고 그대로 놔두는 것이 좋다.

ㅎ

ㅎl

[예제] switch문을 이용한 간단한 계산기

```
main()
{
  char operator;
  int value1, value2;

  while(1) /* 항상 참(1) 이므로 무한 루프 */
  {
    printf("\nEnter expression : ");
    scanf("%i %c %i" , &value1, &operator, &value2);
    printf("Result = ");
    switch (operator)
    {
       case '+': if (value1 == 0 && value2 == 0) {
                      printf("End");
                      return;  /* 프로그램의 실행을 즉시 끝낸다. */
                 } else printf("%d", value1 + value2); break;
```

.... 계속 ....

ㅎ

ㅎl

```
       case '_': printf("%d , value1 - value2);         break;
       case '*': printf("%d , value1 * value2);         break;
       case '/': if (value2 == 0) printf("\nDivision by zero\n");
                 else printf("%d , value1 / value2);   break;
       case '%': if (value2 == 0) printf("\nDivision by zero\n");
                 else printf("%d", value1 % value2);   break;
       default:  printf("Unknown operator\n");
                 return;
    }
  }
}
```

[결과] (0 + 0을 입력하면 실행이 끝난다)

```
Enter expression : -5 + -7 <Enter>
Result = -12
Enter expression : 0+0 <Enter> <== 띄어 써도 되고 붙여 써도 된다.
```

```
    Result = 200
```
ᄒ

위의 예제는 꽤나 복잡해 보인다. 물론 초보자가 아닌 사람은 간단해 보이겠지만......

ᄒl

```
    while(1) /* 항상 참(1) 이므로 무한 루프 */
```
ᄒ

위에 예제 에서 위와 같은 무한루프가 사용되었는데 무한 루프는 수식의 결과가 항상 참이므로, 루프를 빠져 나오지 않고 계속 실행되는 루프를 말한다. 실제로 저자도 큰 프로그램의 구조를 잡을때 위와 같은 무한루프를 즐겨 사용한다.

ᄒl

```
    scanf("%i %c %i" , &value1, &operator, &value2);
```
ᄒ

여지껏 scanf()함수로는 한가지의 변수만을 입력받는 작업만을 해왔다. 위의 scanf()함수호출은 여러개(위에서는 3개)의 변수를 입력받는 방법을 보여준다. printf()함수로 여러개의 변수를 출력하는 방법과 동일하므로 어려울것은 없을것 이다.

ᄒl

```
    switch (operator)
    {
       case '+': if (value1 == 0 && value2 == 0) {
                     printf("End");
                     return;  /* 프로그램의 실행을 즉시 끝낸다. */
                 } else printf("%d", value1 + value2); break;
       case '_': printf("%d , value1 - value2);        break;
       case '*': printf("%d , value1 * value2);        break;
       case '/': if (value2 == 0) printf("\nDivision by zero\n");
                 else printf("%d , value1 / value2);  break;
       case '%': if (value2 == 0) printf("\nDivision by zero\n");
                 else printf("%d", value1 % value2);  break;
       default:  printf("Unknown operator\n");
                 return;
    }
```
ᄒ

위에 보이는 switch블럭은 실제 프로그램의 심장부 인데 유심히 봐두기

바란다. 각 case레이블의 끝 부분에 break를 써주는 것도 잊어서는 않된다.

2.2 중첩된 switch문

switch문도 if문이나 다른 순환문과 마찬가지로 중첩될 수 있다. 그러나 이 때, 유의해야 할 사항이 하나 있다. 원래 case 레이블은 한 switch문 내에서는 유일해야 한다. 즉 중복되면 에러가 난다는 뜻이다.

ㅎl

```
    switch(n)
    {
      case 0:..... break;
      case 1:..... break;
      case 2:..... break;
      case 0:..... break;  <----- 이 부분에서 에러가 난다.
    }
```

ㅎ

그러나 한 switch문 내에 중첩되어 있는 안쪽 (inner) switch문의 case 레이블은 그 바깥쪽 switch문의 case 레이블과 중복되어도 무방하다. 예를 들어보자.

ㅎl

```
    switch (n)  <---- 바깥쪽 switch블럭
    {
      case 0: switch (value2)   <---- 안쪽 switch 블럭
              {
                case 0:      printf(:Division by zero\n");
                             break; <== 안쪽 switch문을 벗어난다.
                case 32768: exit(0);
                default:    printf("%d" , value1 / value2);
                             break;
              }
              break;  <==  여기서 바깥쪽 switch문을 벗어난다.
      case 1:
    }
```

ㅎ

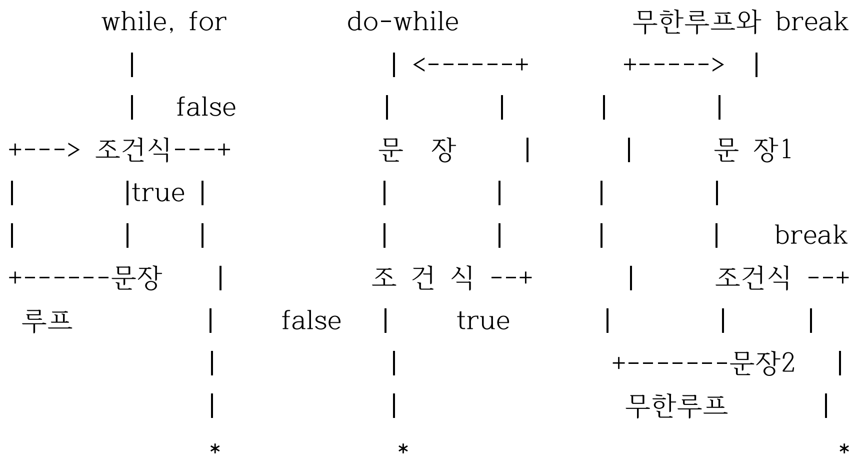
위의 예에서 case 레이블 0:가 바깥쪽과 안쪽 switch문에 중복되어 쓰이고 있음을 볼 수 있다.

2.3 break와 continue 점프문

루프의 처음이나 끝 부분에 있는 조건검사에 의하지 않고서도 루프를 즉시 빠져 나가야 할 때가 종종 생긴다. 이런 경우에도 break문을 쓰면 루프를 즉각 벗어날 수 있다. break문은 정확하게 다음과 같은 기능을 가지고 있다.
break문은 중첩된 while, for, do-while, switch등의 4가지 루프에서 안쪽 루프를 벗어나게 한다.

switch문에서 break문은 필수적이지만 다른 나머지 세 루프에서는 필요에 따라서 선택적으로 사용하면 된다. 대개 break문을 사용하게 되면 프로그램이 간결해지고, 또한 기존의 for, while, do-while 루프만으로는 구현이 불가능했던 특수한 루프도 만들 수 있게 된다.

ㅎl
전형적인 루프의 종류 3가지의 순서도



ㅎ
루프는 그 루프를 빠져나가느냐 마느냐를 결정하는 조건식의 위치에 따라 3가지 종류가 있을 수 있다. 그림에도 나와 있듯이 for, while문은 루프의 맨 선두에서 조건식을 검사해서, 그 값이 논리적으로 참이면 루프를 실행하고 거짓이면 루프를 빠져나간다.

이러한 루프를 일컬어서`진입조건(entry condition)`루프라고 부른다 이 루프는 실행에 앞서 먼저 조건을 따지기 때문에 그 결과가 어쩌다 거짓이라면 루프가 한 번도 실행되지 않는 경우도 생긴다. 반면에 do-while 문은 루프의 후미에서 조건식을 검사한다. 즉 일단 한번 덮어 놓고 실행부터 해놓고 나서 나중에 조건을 검사한다. 이러한 루프를`탈출조건` `(exit condition)`루프라고 부른다.

설명한 진입조건 루프와 탈출조건 루프 이외에 생각할 수있는 또 하나의 루프는 의 세번째에 그려놓은 종류의 루프인데. 이것은 루프의 중간에서 조건식을 검사하여, 그 결과에 의하여 계속 루프를 돌든지 아니면 그 자리에서 즉시 루프를 빠져나오는 그런 루프이다. 이러한 루프는 기존의 for, whil, do-while문만 가지고는 구현이 불가능하다. 그러면 어떻게 구현한단 말인가? 바로 break문을 써서 구현하는 것이다.

그것을 구현한 하나의 예를 들어보자.

टI

```
        do {
            ......
            if (조건) break;
            ......
        } while (1); <--- 무한루프를 만든다.
                        (꼭 무한루프가 아니어도 무방함)
```

ट

전형적인 루프는 이와같이 3가지 정도가 있을 수 있지만, C에서는 그 밖에도 사용자가 임의로 각자의 상황에 맞는 다양한 루프를 만들어 쓸 수 있다. 예를 들어 break문의 사용 횟수에는 제한이 없으므로 루프 탈출 조건을 루프 내의 여러 군데에 둘 수도 있는 것이다.

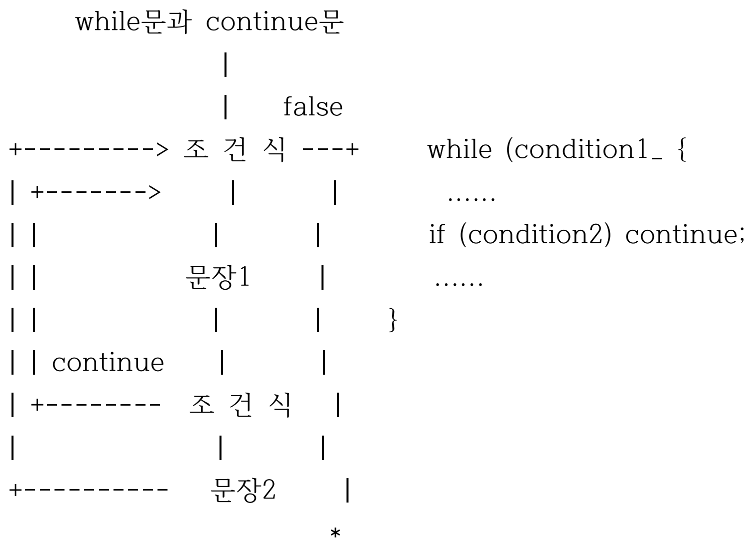
C-언어 에는 break문 이외에도 그와 비슷한 기능을 하는 제어문이 또하나 있다.` continue문`이라는 것이 그것인데, 이 continue문은 break문과 마찬가지로 루프에서 아직 실행되지 않은 채 남아 있는 뒷부분을 그냥 건너뛴다는 점에서는 동일하지만, break문은 건너뛰면서 아예 루프를 나몰라라하고 벗어나는데 반하여, continue문은 루프의 조건식을 검사하는 부분으로 다시 되돌아간다는 점이 다르다.

그래서 while 루프 내에서 continue문을 사용하면 프로그램의 제어가 루프의 맨 처음인 조건식으로 되돌아가고, do-while 루프 내에서는 루프의 맨 마지막에 있는 조건식으로 가게 된다. continue문은 이처럼 루프의 나머지 뒷부분을 싹 무시하고 처음부터 다시 루프를 시작하려 할 때 유용하게 쓰인다.

continue문을 사용하는 방법과 그 순서도는 아래의 그림과 같다.

ㅎl

continue문 사용 방법과 그 순서도



ㅎ

덧붙여서 continue문을 사용할 때는 다음과 같은 사항에 주의해야 한다.

contunue문은 for, while, do-while등 3가지 순환문에서만 쓰인다. 그러므로 continue문을 썼을때는 프로그램의 제어가 switch문의 처음으로 옮겨지는 일은 결코 일어나지 않는다. 즉, switch문 안에서 continue문을 사용했다면 이 때 continue문은 switch문의 조건식 검사부분으로 프로그램의 제어를 옮기는 것이 아니라 switch문 바깥쪽 어딘가에 있는 루프의 조건식 부분으로 제어를 옮겨준다.

`[중요] swtich문에는 continue문을 사용하지 않는다.`

ᇂlsam0404.dat

```
[예제] continue문의 사용예 - 홀수만 출력하기.
main()
{
  int i;
```

.... 계속 ....

ᇂ

ᇂlsam0404.dat

```
    for (i = 0; i < 100; i++)
    {
      if (i%2 == 0) continue; /* i가 짝수이면 for문으로 */
      printf("%4d " ,i);        /* 되돌아 간다. */
    }
  }
  [결과]
  1 3 5 7 9 11 .........
```

ᇂ

서론

전단원에서는 if문과 switch문등의 선택문을 배웠다. 이 단원도 전 단원과 같은 맥락의 반복문을 공부하게 된다.
이단원 에서는 지금까지 보다 조금 복잡하고 재미있는 몇가지 예제도 같이 다루려고 한다.

1 while 과 do-while 순환문

1.1 while 순환문

while문은 C-언어 이외의 언어에서도 거의 가지고 있는 반복문의 형태이며 파스칼이나 심지어 베이식에도 있다. 백번의 설명 보다는 간단한 예제를 통해 알아보도록 하자.

ㅎIsam0500.dat

```
[예제] 간단한 while문
main()
{
   int i=0,j,k;

   while(!kbhit())
   {
      printf("%d\n",i);
      i++;
   }
}
```

ㅎ

ㅎIsam0500.dat

```
[결과]
0 1 2 3 4 5 6 .......
```

ㅎ

위의 예제는 1부터 수를 증가하여 계속 화면에 수를 출력하는 프로그램이다.

ㅎI

```
  while(!kbhit())
```

ㅎ

예제에는 kbhit()함수가 사용되었는데 kbhit()는 키보드를 검사하여 눌려 졌으면 참(0 이외의 수)을 돌리고 않눌려졌으면 거짓(0)을 돌려주는 함수이다.

따라서 위와같은 while문은 키가 눌려질 때까지 계속반복하는 반복문이 된다.

1.2 쪽집게 프로그램

이번에는 조금큰 프로그램을 만들어 보자. 먼저 컴퓨터가 아무 숫자를 하나 선택한다. 그러면 사람은 그 숫자를 추측해 내야한다. 숫자를 사람이 입력하면 그 숫자보다 큰지 또는 작은지 알려준다. 가까운 친구나 가족과 함께 이 프로그램으로 내기를 해도 좋을것이다.

먼저 생각해야 될 사항들

ㅎI

[1] 먼저 컴퓨터가 아무 숫자나 만들어야 한다. 아무 숫자나 만들어내는 방법은 무었일까 ?

[2] 사람이 숫자를 입력하면 컴퓨터의 숫자와 비교하여 맞는지 틀리는지 알아 보아야 한다.

ㅎ

[1]번을 해결해주는 함수로 random()함수가 있다. 이 함수는 아무 숫자나 불규칙적으로 만들어 주는 함수인데, 파스칼에도 동일한 함수가 있고 베이식에도 있다. 컴퓨터에서 아무 수 나 만들어 내는것을`난수발생`이라고 하는데 만약 0~50 까지의 숫자를 변수 i에 저장하려면 다음과 같이 하면된다.

ㅎI

```
i=ramdom(51);
```

ㅎ

random()함수는 0보다는 크고 입력된 인자보다는 1작은 수 사이의 난수를 돌린다. 따라서 10~20 사이의 수를 얻으려면 다음과 같이 해야될 것이다.

ㅎI

```
i=random(11)+10; ----> 0~10 + 10 == 10~20
```

ㄷ
이제 대충 감이 잡힌것 같다. 예제를 확인해 보도록 하자.

ㄷl

[예제] 쪽집게 프로그램

```
#include <stdio.h>
#include <stdlib.h>
      /* random()함수를 사용하려면 stdlib.h가 있어야 한다. */
main()
{
  int i,j,k,count=0,value;
  int dap;
  retry:                  /* goto문을 위한 라벨 */
  clrscr();
  randomize();            /* 난수 발생기 초기화 */
  count=0;                /* 카운트 초기화 */
  value=random(100);      /* 0~99 까지의 난수를 발생시킨다. */
  while(1)                /* 무한 루프를 만든다. */
  {
    printf("Enter your value and press <Enter>\n");
```

.... 계속 ....
ㄷ

ㄷl

```
    scanf("%d",&dap);     /* 수를 입력받는다. */
    count++;              /* 카운트를 증가 시킨다. */
    if (value > dap) printf("bigger then %d\n",dap);
    if (value < dap) printf("smaller then %d\n",dap);
    if (value== dap)
    {
      printf("Congratulations ! you make it.\n");
      printf("you find at %d trial(s)\n",count);
      printf("one more ? (y/n)");
      i = getch();
      if ((i=='y') || (i=='Y')) goto retry;
      else break;
    }
```

```
        }
    }
```
ㄷ

예제를 실행하면 추측한 숫자를 치고 <Enter> 키를 치면 추측한 숫자가 원래의 숫자보다 작은지 크지 알려준다. 그러면 결과를 토대로 또다시 그 다음 숫자를 입력하고, 그러면 또다시 원래의 숫자보다 작은지 크지 알려주고 이러한 작업을 반복해서 답을 찾아내면 몇번 시도해서 맞추었는지를 알려준다.

ㄷI
```
    clrscr();
```
ㄷ
위의 함수는 처음 보는것인데 이는 화면을 지워주고 커서를 좌측 맨 위쪽에 위치 시키는 함수이다. 파스칼에도 같은 함수가 있으며 인자는 필요 없다.

ㄷI
```
    randomize();
```
ㄷ
randomize()함수는 random()함수전에 반드시 호출해 주어야 한다.

물론 randomize()함수 호출이 없으면 random()함수가 제대로 동작하지 않는 다는뜻은 아니다. randomize()함수는 컴퓨터 내의 난수 발생기를 초기화 시켜주는 역활을 하는데, 이 초기화가 없으면 random()함수는 프로그램을 실행 시킬때 마다 같은 결과를 보여준다.
ㄷI
```
    for (i=0;i<5;i++) printf("%d ",random(10));
```
ㄷ
위와 같은 예제가 있다고 하자. 만약 프로그램을 실행 시켜서 난수를 출력해서 "5 6 1 2 8" 이라는 결과를 얻었다고 하다. 그러면 다음에 실행 시킬때고 역시 "5 6 1 2 8"이라는 결과를 보여준다. 매번 실행때마다 같은 결과를 보여주므로 쪽집게 같은 프로그램에서는 사용하기가 곤란하다 답안이 유출된 시험과 같다고나 할까 ?

어쨌든 randomize()함수는 이런 현상을 막아준다. random()과 randomize

함수는 서로 단짝이라는 것을 기억하기 바란다.

ㅎl

```
        while(1)
        {
          printf("Enter your value and press <Enter>\n");
          scanf("%d",&dap); ............(1)
          count++;
          if (value > dap) .............(2);
          if (value < dap) .............(3);
          if (value== dap) .............(4);
          {
            i = getch();     .............(5)
            if ((i=='y') || (i=='Y')) goto retry; .........(6)
            else break;       .............(6)
          }
        }
```

ㅎ

위의 while문이 이 프로그램의 심장부이다. (1)번은 답을 입력받는 부분으로 더이상의 설명이 필요없는것 같고, (2)번은 입력한 수가 정답보다 작을때의 처리이다.

(3)번이 입력한 수가 정답보다 클때의 처리이다. (4)번은 정답일 경우의 처리인데 (2),(3)번 과는 달리 여러개의 문장이 들어가야 하므로 "{ }" 블럭을 사용하였다.
(5)번의 경우는 약간 설명이 필요할것 같다.`getch()`함수는 키보드에서 한 문자를 읽어들이는 함수로 지금까지 키보드로 부터의 입력을 맡아온 scanf()함수와는 약간 성질이 다르다.
scanf()함수는 여러가지 형태의 데이타 (문자열, 정수, 부동 소숫점)등을 받아들일 수가 있으고 입력후에 <Enter>키를 쳐 주어야 비로소 입력이 끝나지만`getch()`함수는 한문자(char 또는 int)만을 읽어들이며 한개의 키를 치면 바로 입력이된다.

getch()와 비슷한 일을 하는 함수로는`getchar()`과`getche()`가 있다.
getchar()과 getche()는 키가 눌러질때 눌려진 키를 화면에 보여주는 것이 getch()와는 다른점인데, 지금은 그냥 눌러진 키를 화면에 보이게 할때에는 getchar()이나 getche()를 쓰고 그렇지 않을때에는 getch()를 쓴

다는 정도만 알아두자. 사실 getchar()과 getche()사이에도 차이점이 있지만 이는 후에 알아보기로 하자.

1.3 do-while 순환문

do-while문은 위에서 배운 while문과는 매우 비슷하다. 다만 한가지 다른점은 while문은 순환문의 선두에서 조건식을 검사하여 루프를 돌것인지의 여부를 결정하지만 do-while문은 일단 루프를 돈다음 루프의 끝 부분에서 조건식의 검사하고 합당하면 다시 루프의 선두로 이동하는 것이 다르다.

즉 while문은 어쩌다가 처음부터 조건이 합당하지 않을 경우에는 루프를 한번도 실행하지 않을 경우도 있지만 do-while문은 일단 루프를 한번 실행 하고 조건식을 검사하므로 do-while문은 언제나 한번은 실행된다고 할 수 있다.

illu0500.dat

```
while문과 do-while문의 차이

while (조건식)  {                    do {
   문장;                                  문장;
   문장;                                  문장;
   ....;                                 ....;
}                                      } while (조건식);
```

사실 do-while문은 모든 프로그래머들에게 널리 쓰이는 순환문 이라고는 볼 수 없다. 전체 순환문중 do-while문이 차지하는 비중은 약5퍼센트 정도 밖에 되지 않는것만 보아도 알 수 있다. 아래의 예는 1부터 100까지의 수 를 화면에 출력하는 예제이다.

```
while (i<=100)              do
{                           {
   printf("%d",i++);         printf("%d",i++);
```

}   } while(i<100);

ㄸ

2. for 순환문

ㄲ2.1 for문 개론

여러가지 순환문 중에서 가장 사용 빈도수가 높은 것은 단연 for문이다. 특히 C에서는 단연 독보적인 존재이다. C의 for문은 파스칼이나 베이식의 for문과 비교해 볼 때 한 차원 높은 유연성을 가지고 있다.

파스칼의 for문은 비교적 제한적인 순환문이다. 그것은 for문을 while문으로 대치할 수는 있으나, while문이 할 수 있는 작업을 for문으로 대치할 수 없는 경우가 적지 않기 때문이다. 예를 들어서 루프 제어 변수(loop control variable, 관습적으로 i나 j 같은 변수)가 기하 급수적으로 증가해야 하는 경우를 들 수 있다.

따라서 파스칼의 for문은 while문의 하위 구조라 볼 수 있으며, 실행 속도를 높이기 위해서 주로 for문을 쓰게 된다.

그러나 C-언어 에서의 for문은 반대로 while문의 상위 구조이다. 왜냐하면 for문은 while문의 약식 표기법이라 할 수 있기 때문이다. for문은 분해하면 결국은 while문이 된다.
다른 언어와 비교했을때 C-언어의 for
문의 우수한 점은

첫째. 다른 언어에서는 꼭 제어변수가
1씩 증가 해야한다. 따라서 0.1 씩 증
가 한다던가 2씩 증가 한다던가 하는
것은 불가능 하다.

둘째. 표현 방법에 있어서 훨씬 유연함
을 제공한다.

illu0501.dat

for문의 일반적인 형태

```
for (초기식; 조건식; 증감식) ......;
또는
for (초기식; 조건식; 증감식)
{
}
```

. 초기식: 주로 루프 제어 변수에 초기값을 대입하는 대입식.
. 조건식: 주로 루프 제어 변수에 조건 범위를 검사하는 관계식 또는 논리식.
. 증감식: 주로 증감 연산자나 대입 연산자를 사용하여 루프 제어 변수를 제어한다.

전형적인 예

`for (i = 0; i < 10; i++) ......`

++i를 써도 되지만 관례적으로 i++를 쓴다.

`for (i = 10; i > 0; i--) ......`

루프 제어 변수 i를 10에서 0까지 감소시킨다.

`for (i = 1; i <= 99; i += 2) ......`

루프 제어 변수 i를 1부터 99까지 2씩 증가시킨다.

`for (i = 100; i >= 2; i -= 2) ......`

루프 제어 변수 i를 100에서 2까지 2씩 감소시킨다.

2.2 for문의 유연성

이 절에서는 하나씩 예를 들어 가면서 파스칼의 for문과는 구별이 되는 C-언어의 for문이 특별히 가지고 있는 특징에 대해서 이야기 하겠다.
C-언어의 for문이 이처럼 대단한 유연성을 가지고 있는 이유를 한 마디로 말한다면 C의 for문은 파스칼의 for문과 유사한 것이 아니라 오히려 while문과 동등하다는 점 때문이다.

그 점만 충분히 인식한다면 C의 for문도 실제로는 별 것이 아님을 알 수 있다. 단지 그 개념과 시각적인 이해도가 while문보다 매우 월등하다는 점이 다르다면 다르다.
그리고 이 절에서 언급하는 for문의 형태가 그리 자주 쓰이는 것은 아니지만 [4]번이나 [10]번의 형태와 같이 언젠가 한번쯤은 반드시 필요할 때가 있을 터이므로 "아! 이런 형태도 가능하구나"하는 정도로만 훑어 보기 바란다. 필요할 때마다 그때 그때 참조할 수 있으면 그만이다.

ㅎl

[1] 루프 제어 변수를 +-1 이외의 단위로 증감할 수가 있다. 이 경우에 +=나 -=와 같은 대입 연산자를 증감식으로 사용하면 된다.
`예) for (i = 1; i < 100; i + 2) ......`

[2] 루프 제어변수를 산술급수적이 아닌, 기하급수적으로 증감시킬 수도 있다. *= 또는 /= 대입 연산자를 증감식으로 사용하면 된다.
`예) for (i = 1024; i > 1; i /= 2) ......`

ㅎ

ㅎl

[3] [1]과 [2]를 일반적인 말로 한다면 증감식에 제한이 없다고 말하는 것과 같다. 루프 제어 변수가 반드시 일정한 단위로 증감할 필요는 없으며, 또한 산술급수적 이나 기하급수적으로만 증감할 필요도 없다.

예) x, y가 double형일 때

```
for (x = 1.2; x < 100.; x += y * 2.) {
    printf("Enter unit: ");
    scanf("%lf" , &y);
    printf("\nNext number: %d" , x);
}
```

[4] 루프 제어 변수의 데이터형에는 제한이 없다. 정수형 변수는 물론이고 float, double형 변수나 문자형 변수, 심지어는 포인터 변수도 사용이 가능하다.

예) x가 float형일 때

```
for (x = 0.; x > 180.3; x += 3.141592 / 2.) ......
```

* 파스칼에서도 루프 제어 변수로 integer나 byte형 변수만 쓸 수 있다는 사실과 비교하기 바람.

ㅎ

[5] 조건식에도 제한이 없다. 합당한 수식이기만 하면 된다. 따라서 관계식, 논리식, 또는 논리적 의미를 가지는 어떠한 임의의 수식이라도 모두 허용된다. 즉 for문의 조건식은 while문의 조건식과 동등하다는 뜻이다. 또한 초기식에도 제한이 없다.

```
예) for (i = 0; i * j < 300; i++) ......
     for (i = 0; (c = getch()) == '\r'; i++) ......
     for (c = getch(); c < '\x7f'; c++) putch(c);
```

[6] 초기식, 조건식 또는 증감식은 필요에 따라서 얼마든지 생략할 수가 있다. 특히 조건식을 생략하면 무한루프(infinite loop)가 형성된다는 점을 기억해 두기 바란다. 이 때 무한루프는 break문을 써서 탈출할 수가 있다. 그리고 증감식을 생략하는 경우에는 루프 안의 다른 적절한 장소에서 루프 제어 변수를 적당히 증감 시켜 주어야 할 것이다.

```
예) for (;;) ;   <== 무한 루프
    for (;;) {
         ......
         if (condition) break;
              ^----- 조건이 맞으면 무한루프를 탈출한다.
         ......
    }
    for (i = 0; i < 10;)    <== 증감식을 생략한 경우
    printf("5d\n" , i++);
                        ^---- 여기서 루프 제어 변수 i를 증가시킨다.
```

[7] [3]번의 형태와 같은 맥락에서 살펴볼 때, 루프의 번복 횟수를 프로그램의 실행도중에 임의로 변경할 수 있음을 알 수 있다.

.... 이 루프는 50회 반복한다. (100회가 아님) ....

```
for (i = 0; i< 100; i++)
{
     if (i % 10 == 0) i += 5;
     printf("%4d" , i);
}
```

[8] break문을 쓰면 남아 있는 반복 횟수와는 상관없이 언제든지 for 문을 탈출할 수 있다. 아래의 예는 문자를 20개 입력받는 도중에 <Ctrl-M)키를 누르면 루프를 빠져나오게 된다.

```
int i;
char c, s[20];
for (i = 1; i <= 20; i++)
{
      c = getch();
      if (c == '/') break;
      s[i] = c;
}
```

ᄒ

ᄒI

[9] continue문을 사용하면 루프가 보다 간결해질 수도 있다.

[10] 쉼표 연산자를 써서 여러 개의 루프 제어 변수를 동시에 사용할 수 있다. for문에서 쉼표 연산자를 사용하는 방법은 이제 설명하려고 한다.

ᄒ

ᄒJ2.3 쉼표 연산자와 for문

[3]번과 [5]번 형태에서도 연급했듯이 초기식과 증감식에는 어떠한 제약도 없다. 수식이기만 하면 되는 것이다. 이 말은 곧 쉼표 연산자를 써도 아무런 문제가 없다는 말이나 같다.

쉼표 연산자를 쓰면 초기식과 증감식에 두 개 이상의 루프 제어 변수를 쓸 수 있는 융통성이 생긴다. 예를 들어보자. 다음의 예는 두 개의 변수 i, j를 각각 증가 시켜서 출력하라는 것이다.

쉼표 연산자를 쓰지 않는 경우;

ㅎl

```
j = 1;
for (i = 0; i <9; i++){
      printf("[%d:%d]" , i, j);
      j *= 2;
}
[결과]
[0:1][1:2][2:4][3:8][4:16][5:32][6:64][7:128][8:256]
```

ㅎ

위의 예에 쉼표 연산자를 쓰면 단 한 줄로 나타낼 수 있다.

ㅎl

```
for (i=0,j=1 ; i<9 ; i++,j*=2) printf("[%d:%d]" , i, j);
```

ㅎ

여기서 한 가지 주의할 점은 쉼표 연산자로서의 쉼표 기호(,)와 구두점으로 사용하는 쉼표 기호(,)를 서로 혼동하지 말라는 것이다. 아래와 같이 여러 개의 변수를 선언할 때나, 함수의 실매개변수를 나열할 때 쓰이는 쉼표 기호는 연산자가 아니라 구두점이다.

3. goto 점프문과 레이블

goto문의 사용 여부에 대해 쏟아지는 논란에 대해서는 독자도 많이 들어보았을 것이다. 그러나 저자는 goto문을 전혀 사용하지 말라고는 하지는 않겠다. 왜냐하면 저자도 goto문을 자주 사용하는편에 속하기 때문이다. 비교적 goto문을 가장 많이 사용하는 언어는 베이식이다. 그래서 베이식 사용자에게 goto문을 사용하지 말라고 하면 상당히 의아애 할 것이다.

goto문을 사용하지 않고 어떻게 프로그램을 작성하는가 ?   물론 애당초부터 베이식이 구조적 프로그래밍(Structured programming)과는 너무도 먼 거리에 있는 언어이고 해서 베이식에서는 goto문 없이 프로그램을 작성 한다는 것은 거의 불가능한것이 사실이다. 그러나 goto문이 많이 들어가 있는 프로그램이 알아보기 힘들다는 것은 베이식 사용자들은 더욱 잘 알고 있을 것이다.

goto문은 프로그램의 구조를 파악하기 힘들고 유지,관리가 힘들다는 이유로 등한시 당하고 있다. 그리고 사실 C-언어와 같은 언어에서 goto문 없이 프로그램을 작성하는 것이 불가능 한 것도 아니다.

자신에 프로그램에 goto문을 사용할 것인지 아닌지는 어려분이 판단해야 할 문제이다. 다만 프로그램을 작성할 때 goto문을 사용하면 훨씬 작성

이 편리한 부분이 있다는 점만 알아두자.

ꈏ3.1 goto문의 사용

goto문을 사용하려면 레이블(label)을 써야한다. 레이블은 goto문으로 이동할 곳을 알리는 주소와 같은것이다. 주소 없이는 집을 찾아갈 수 없듯이 레이블 없는 goto문은 있을 수 없다. 파스칼에서 goto문을 사용해 본 사람은 goto문을 따로 배울 필요가 없다. 다만 레이블을 따로 선언할 필요가 없다. 와~~~ 저자도 전에 파스칼을 사용했었는데 goto문을 사용하기 위해서 레이블을 선언하는 것을 얼마나 귀찮게 어겼는지 모른다.

```
        goto label;   -----+
        ......             | 제어가 옮겨간다.
        label: <----------+
        ......
```

그리고 ANSI-C 표준을 보다 충실히 따르기 위해 터보-C 2.0 이상 에서는 레이블에 관한 문법이 추가되었다. 즉 레이블 다음에는 반드시 하나 이상의 문장이 존재해야 하며 레이블바로 다음이 블럭 끝이거나 하면 않된다는 것이다. 따라서 아래의 두 가지 예 중에서 왼쪽은 터보-C 2.0이상에서는(Borland c++ 도 마찬가지) 에러를 발생하므로 오른쪽과 같이 널문장(null statement)을 레이블 뒤에 추가해야 한다. 물론 보다 합리적인 방법은 break문이나 continue문을 쓰는 것이다.

```
    틀린 문법                          올바른 문법
    {                                  {
      ......                           ......
      if (cond) goto lable;             if (cond) goto label;
      ......                           ......
      label:                            label:; <--- 널문장(;)
    }                                  }
```

그건 그렇고 전문가들도 공인하는 goto문의 사용처가 딱 한 군데 있다. 그것은 바로 여러 번 중첩되어 있는 루프를 한꺼번에 벗어나는 일이다. 이것은 마치 겹겹이 둘러싸인 포로 수용소의 철조망을 단 한 번에 건너

뛰어 탈출하는 것과 같다.

```
    for (.....) {
        for (......) {
             for (......) {
                 ......
                 if (......) goto quit;-----+
                 .                          |
                 .                          | 단 한번에
                 .                          | 껑충 뛰어 탈출한다.
             }                              |
         }                                  |
     }                                      |
     quit:  <-------------------------------+
```

위의 경우에 한하여 goto문의 사용이 용인된다. 왜냐하면 goto문을 쓰지 않으면 오히려 프로그램이 더 복잡해지기 때문이다. 위의 예를 goto문을 쓰지 않고 break문을 써서 작성하면 아래와 같이 된다. goto문을 썼을 때보다 프로그램이 훨씬 더 복잡해졌음을 알 수 있다.
추가되는 if문으로 인해서 실행 속도도 물론 더 느려진다.

```
        #define true 1
        #define false 0
        quit = false;
        for (......) {
            for (......) {
                for (......) {
                    ......
                    if (......) {
                        quit = true;
                        break;    ----------+
```

.... 계 속 ....

```
                    }                   |
                    .                   |
                }                       |
                if (quit) break;  <---------+
```

```
            }                        |
            if (quit) break; <--+
        }                  |
        ......  <-------+
```

여기까지 제어문과 루프에 대해서 모두 익힌 셈이다. 이 제어문과 루프는 프로그래밍에 있어서 가장 기본이 되는 것이므로 철저하게 익혀 두어야 한다.

4. 금광을 찾아서

이제 루프와 제어문, 연산자 등 C-언어의 모든 부분을 대충 훑어 보았다 대충 훑어 보았다고 말할 수 밖에 없는 이유는 아직 배운것을 활용해 본 적이 없기 때문이다. 지금 소개하려고 하는 이 프로그램은 지금까지의 예제들 보다는 훨씬 크고 복잡하다.

이 게임은 금광을 찾아서 채굴하는 과정을 그렸다. 먼저 금광을 찾아낼 비용을 입력하면 금광을 찾기 시작한다. 이때 비용이 많을 수록 금광을 찾아낼 가능성은 높아진다. 만일 많은 비용을 들였는데도 금광을 찾지 못하면 당신이 들인 비용은 공중으로 날아가도 만다.
만일 금광을 찾았다면 다시 금을 챌 비용을 입력해야만 한다. 물론 많은 비용을 입력할수록 많은 금을 캘 수 있게 된다.

캐낸금은 창고에 보관된다. 금은 즉시 팔 수도 있으며 그때 그때 금값을 보아가며 팔 수도 있다. 처음에는 단 100만원을 가지고 시작하며 1억원 이상을 모으거나 무일푼(?)이 되면 끝난다.

```
#include <stdio.h>
#include <stdlib.h>

    int money=100,gold=0,price=50;
    int i,j,k,coin;
```

```
void display()
{
   int x,y;

   x=wherex();
   y=wherey();
   gotoxy(1,1); printf("돈        : %d       \n",money);
   gotoxy(1,2); printf("금        : %d       \n",gold);
   gotoxy(1,3); printf("금시세   : %d       \n",price);
   gotoxy(x,y);
.............................. 계 속 ............................
ㅎ

ㅎl
}

main()
{

   START:
   clrscr();
   price = random(60)+20;
   display();
   gotoxy(1,5);
   printf("선택하세요\n");
   printf("(1) 금광찾기    (2) 금팔기      (3) 종료\n\n");

   if ((money==0) && (gold==0)){
      printf("암됐군요 !\n");
      printf("당신은 무일푼이 됐습니다.\n");
.............................. 계 속 ............................
ㅎ

ㅎl
      exit(0);
   }
   if (money>1000){
      printf("축하합니다.\n");
      printf("당신은 억만장자가 됐습니다.\n");
      exit(0);
```

```
      }
      while(1){
        if (kbhit()){
          i = getch();
          if (i=='1'){
            REGET1:
            printf("금광을 찾을 비용을 입력하세요. (1-100)\n");
            scanf("%d",&coin);
            if ((coin<0) || (coin>100) || (coin>money)){
              printf("입력이 틀립니다 !\n");
```


.... 계 속 ....

ㄷ

ㄷl

```
              goto REGET1;
            }
            if (random(100)<coin){
              money-=coin;
              display();
              printf("축하합니다 !\n");
              printf("금광을 찾았습니다.\n");
              REGET2:
              printf("얼마를 들여 금광을 파겠습니까 ? (1-100)\n");
              scanf("%d",&coin);
              if ((coin<0) || (coin>100) || (coin>money)){
                printf("입력이 틀립니다 !\n");
                goto REGET2;
              }
              money-=coin;
              j = random((coin/10)+1)+1;
```


.... 계 속 ....

ㄷ

ㄷl

```
              printf("+-------------------------------------+\n");
              printf("|            결 과 보 고 서           |\n");
              printf("+-------------------------------------+\n\7");
              printf("%d개의 금덩어리를 찾았습니다.\n",j);
              gold+=j;
              printf("아무키나 누르세요......\n");
              display();
```

```
                getch();
                goto START;
              }
              else{
                money-=coin;
                printf("않됐군요 !\n");
                printf("금광은 아무나 찾는게 아닌가 봅니다.\n");
                printf("아무키나 누르세요......\n");
                getch();
```

.... 계 속 ....

ㄷ

ㄷl

```
                goto START;
              }
            }
            if (i=='2'){
              money+=gold*price;
              gold=0;
              goto START;
            }
            if (i=='3') exit(0);
          }
        }
      }
```

ㄷ

간단하게 작성하려고 하였는데 욕심을 부리다 보니 100라인이나 되는 큰 프로그램이 되었다. 사실 100라인 정도면 아주 작은 프로그램에 속하지만 초보자에게는 상당히 길게 느껴질 것이다. 참고로 대부분의 응용 프로그램의 라인수는 1만 ~ 10만 라인에 이른다.

프로그램을 실행 시키면 아래와 같이 현재 가지고 있는 돈과 금, 그리고 금의 시세가 나온다. 금의 시세는 금덩어리 한개당 가격을 말한다.

ㄷl

```
    돈      : 100
    금      : 0
    금시세  : 52

    선택하세요
```

(1) 금광찾기   (2) 금팔기   (3) 종료

ㆆ

위의 메뉴중에서 처음 시작할때는 금이 없으므로 금팔기는 필요가 없고 "1"을 눌러 금광찾기를 시행한다.   그러면 아래와 같이 금광을 찾는데 드는 비용을 요구한다.

ㆆI

금광을 찾을 비용을 입력하세요. (1-100)

ㆆ

1부터 100까지의 수를 입력하면 되는데 비용이 많을 수록 금광을 찾을 가능성은 높아진다.

100을 입력하면 반드시 금광을 찾을 수 있는데 처음에 100을 입력하면 돈이 다 떨어져서 금을 캐낼 비용이 없으므로 100보다는 작은 수를 입력해야 한다. 금을 발견했을 경우에는 아래와 같은 메시지가 나오고 금을 캘 비용을 요구한다. 1부터 100까지의 수를 입력해야만 하며 현재 가지고 있는 돈 보다 많은 양을 지정할 수는 없다.

ㆆI

```
축하합니다 !
금광을 찾았습니다.
얼마를 들여 금광을 파겠습니까 ? (1-100)
```

ㆆ

물론 많은 비용을 들일 수록 많은 금을 얻을 수 있다. 알맞은 양의 비용을 입력하면 금을 얼마나 캐냈는지 보여준다.

ㆆI

```
+--------------------------------------+
|              결 과 보 고 서             |
+--------------------------------------+
7개의 금덩어리를 찾았습니다.
```

ㆆ

금을 팔때에는 금시세를 보아가며 팔아야 한다. 시세는 빨리 변하고 그 차이도 크므로 언제 금을 파느냐에 따라 금액이 큰 차이가 난다.

ㆆJ4.1 텍스트 화면

위의 예제에서는 처음으로 텍스트 화면 좌표계에 관한 함수가 나왔는데 이들에 대해서 알아보도록 하자. 도스 실행 화면과 같은 텍스트 화면은 아래와 같이 x축으로 80행 y축으로 25열의 문자를 화면에 표시해 줄 수 있다 즉 한 화면에는 (80행 * 25열 = 2000문자) 2000개의 문자를 표시

해 줄수 있다.
그러나 이는 영문을 사용하였을때 얘기고 한글을 사용하였을 때는 한글 한자의 크기가 연문의 2배 이므로 (40행 * 25열 = 1000문자) 한 화면에 1000개의 문자를 표시할 수 있다.
ㅎl

```
한 화면에 표시할 수 있는 글자
영문 (80행 * 25열 = 2000문자)
한글 (40행 * 25열 = 1000문자)
```

ㅎ

화면에 문자가 쓰여질때 즉, printf()한수나 cprintf()등의 함수를 사용할때 문자는 항상 화면상에서 커서가 있는 위치에 쓰여지게 된다. 그리고 문자가 쓰여지고 나면 커서는 쓰여진 문자의 맨 끝으로 이동하게 된다. 때문에 "\n"과 같은 특수문자를 사용하지 않으면 글자가 연이어져 출력되기 때문에 알아보기 힘들게 된다.
ㅎl

```
main()
{
   printf("file will be deleted !");
   printf("are you sure ? (y/n)");
}
```

ㅎ
물론 위와 같은 예제에서 문자열의 끝에 "\n"만 붙여주어도 문자를 알아보는데 큰 지장은 없다. 그러나 문자를 화면상에 특정 위치에 출력하기 위에서는 gotoxy()함수의 사용이 불가피 해진다.

ㅎl

```
gotoxy( x좌표(1~80) , y좌표(1~25) );
```

ㅎ
위의 예는 gotoxy()함수를 호출하는 방법이다. 주의할 것은 좌표의 시작이 0이 아닌 1이라는 것이다. 그래픽을 먼저 배운 사람은 조금 의아하게 느낄것이다. 그래픽 좌표는 0부터 시작하는데 반해`텍스트 좌표는 반드``시 1부터 시작한다.`좌표값의 범위를 벗어난 값을 지정하면 gotoxy()함수는 아무일도 하지 않는다. 만일 gotoxy()함수가 성공적으로 수행되면 커서는 gotoxy()함수에서 지정한 위치로 이동하게 되는데 아까 설명했듯이 printf()등의 함수는 커서의 위치에 문자를 출력한다.
ㅎlsam0501.dat

```
#include <conio.h>    /* conio.h에는 gotoxy()등 텍스트 화면에 */
main()                /* 관계된 함수가 정의되어 있다.          */
{
  gotoxy(10,12); printf("Here is 10,12");
  gotoxy(20,15); printf("Here is 20,15");
}
```

컴퓨터에는 함수가 있으면 반드시 역함수가 있다. gotoxy()는 커서의 위치를 변경시키는 함수는 반면에 커서의 위치를 돌려주는 함수도 있다.

* 커서의 위치를 알려주는 함수

```
int wherex() ........ 현재의 x좌표를 알려준다.
int wherey() ........ 현재의 y좌표를 알려준다.
```

wherex(),wherey()함수를 이용하여 gotoxy()함수호출 이전의 커서위치로 이동할 수 도 있다. 아래와 같이 하면된다.

```
int x,y    /* 임의의 변수 x,y */

x=wherex();
y=wherey();
gotoxy(random(80)+1,random(25)+1) /* 아무데로나 이동한다. */
gotoxy(x,y); /* 원래의 위치로 복귀한다. */
```

서론

포인터를 빼어 놓고는 C-언어를 말할 수 없다. 당연한 이야기인지 모르지만 C-언어는 포인터 처리 중심의 언어이다. 따라서 C-언어를 배우는데 있어서 가장 중요한 부분이라 할 수 있다.   하지만 이 포인터라는 것이 이해하기 쉬운것이면 좋겠지만 불행하게도 C-언어를 배우려는 많은 사람들이 바로 이 포인터에서 좌절하게 된다.

참으로 안타까운 일이다. 이 장에서는 포인터를 낱낱히 분해하고 해집어서 완전히 배워보도록 하자.
다시한번 강조하건데 전체의 과정 중에서 가장 중요한 부분이 바로 이장이라는 것을 명심하기 바란다.

1. 배열(Array)

ŋ1.1 배열이란 ?

포인터를 배우려다가 왜 갑자기 배열을 배우는 걸까 ? 그것은 포인터와 배열이 서로 밀접한 관계에 있기 때문
이다. 서론은 일단접고, 배열이란 데
이타를 늘어놓은 것을 말한다.

예를들어 한 학급 학생들의 점수를 데
이타로 사용하려고 하면 어떨까 ? 한
학급의 학생수가 60명 이라면 데이터를
무려 60개를 선언해야 할 것이다.
아래와 같은 모양이 된다.

```
int score1,score2,score3,score4,
    score5,score6,score7,score8,
    score9,score10,score11, ....
```

ŋillu0600.dat

정말 끔찍한 일이 아닐 수 없다. 그러나 이정도는 인내심(?)이 뛰어난 프로그래머라면 해낼 수 있다. 하지만 이번에는 한 학급 학생이 아니고 한 학교, 아니 경기도나 전국의 학생의 점수를 처리하려면 어떻게 될까 이쯤까지 이야기가 전개되면 어떤 이들은 프로그래밍 언어 배우기를 포기 할 지도 모르겠다.

이러한 이유로 배열이 사용된다. 배열은 이와 같이 같은 용도로 여러개의 변수를 사용하고자 할때 유용하게 사용된다.

예를 들어 한 학급 학생의 점수를 데이타로 처리하기 위해 다음과 같이 정의 될 수 있다.

ㅎ|

```
    배열의 정의 방법

                +---------- 배열의 크기(필요한 자료의 수)
                |
             +--+--+
    int score[ 60 ];
        +----+
          |
          +----------- 배열변수 이름
```

ㅎ

위와 같이 정의하면 score란 배열변수의 이름으로 60개의 변수를 예약한 것과 같다.

첫 번째 배열변수는 score[0]이 되며, score[1], score[2], ... 이런 식으로 계속 진행되어서 마지막 배열변수는 score[59]가 된다. score[60]이 아님을 유의하기 바란다.
배열변수를 사용할때 "[", "]" 안에 들어가는 수를`첨자`라고 하는데 첨자는 항상 0부터 시작한다. 때문에 마지막 배열변수는 score[59]이다.
복잡한 설명보다는 간단한 예제로 배열의 사용법을 알아보도록 한다. 아래의 예제는 score[0] 부터 score[4]까지 5개의 점수를 읽어들이는 예제이다.

ㅎ|sam0600.dat

```
    [예제] 배열변수의 사용예

    int score[60],i; /* 배열변수 선언 */
```

```
     main()
     {
        scanf("%d %d %d %d %d",&score[1],&score[2],&score[3],
                                   &score[4],&score[5],);
ㆆ

ㆆIsam0600.dat
        for (i=0;i<5;i++) printf("student %d : score %d\n",i,score[i]);
        printf("Average : %d\n",(score[0]+score[1]+score[2]+
                                    score[3]+score[4])/5);
     }
     [결과]
     40 30 20 10 80 <Enter>  <- 이와같이 한줄에 5개의 점수를 입력한다.
     student : score 40
     student : score 30
     student : score 20
     student : score 10
     student : score 80
     Average : 36
ㆆ
```

지금까지 착실하게 공부한 사람은 별 설명이 필요 없겠지만 확실한 이해
를 위하여 예제를 분석해 보자.

```
ㆆI
   scanf("%d %d %d %d %d",&score[0],&score[1],&score[2],
                              &score[3],&score[4],);
ㆆ
```

위에 보이는 부분은 score[0]부터 score[4]까지의 점수를 한꺼번에 읽어
들이도록 되어있다. 단지 "40 30 20 10 80"과 같이 데이터를 한칸씩 띄
어서 입력하고 <Enter>를 치는것 만으로 5개의 자료가 입력된다.

```
ㆆI
   for (i=0;i<5;i++) printf("student %d : score %d\n",i,score[i]);
ㆆ
```

위의 부분은 별 설명이 필요 없을것이다. 단지 for문으로 5개의 점수를
하나하나 화면에 표시해주는것 뿐이다.

ㄷI

```
   printf("Average : %d\n",(score[0]+score[1]+score[2]+
                                  score[3]+score[4])/5);
```

ㄷ

5개의 데이타를 모두 합하여 다시 5로 나누어 평균을 구하는 부분이다. 데이타가 5개 밖에 되지 않아 일일이 더하는 무식한(?) 방법을 사용하였지만 데이타의 수가 많아지면 이런 방법은 통하지 않는다.

이쯤에서 평균을 구하는 예제를 한번 작성해 보도록한다. 이번에는 100개의 자료를 대상으로 한다. 자료의 수만 많을 뿐이지 기본원리는 같다.

ㄷIsam0601.dat

```
     [예제] 평균 구하기
     #include <stdlib.h>   ..... random()함수를 사용하기 위해서
     main()
     {
        int i,j,k;
        long temp;          ..... 수의 평균을 내기위한 임시변수
        int number[100]; ..... 100개의 배열을 선언한다.
```

ㄷ

ㄷIsam0601.dat

```
        for(i=0; i<100; i++)
        number[i]=random(1000);   ..... number에 0~999 까지의
                                        난수를 넣는다.
        for(i=0; i<100; i++)
        temp+=number[i];             ..... 수를 모두 더하고
        temp/=100;                   ..... 100으로 나누어 평균을 더한다.
        printf("Average : %d",temp); ..... 화면에 표시한다.
     }
     [결과]
     487
```

ㄷ

저자의 컴퓨터에서는 487 이라는 결과가 나왔으나 컴퓨터 기종마다 다른 결과를 보일 수 도 있다. 배열의 갯수를 늘려서 시험해 보면 점차 500에 가까운 결과를 보일것이다.

1.2 다차원 배열

지금까지 배운 배열은 모두 1차원 배열이다. 첨자가 "[" "]"가 하나밖에 들어 가지 않았으므로 1차원 배열인 것이다. 한편 한 학생이 여러 과목을 수강할 경우, 학생당 과목별로 성적을 나타내고 싶을 때는 다음과 같이 2차원 배열을 만들 수 있다.

`int score[학생수][과목수]`

이때, score[i][j]는 i+1 번째 학생의
j+1 번째 과목의 점수가 된다. 왜 i+1
인지는 첨자가 항상 0부터 시작하는것
을 잘 되새겨 보면 알 수 있다.

illu0601.dat

이때 student에 학생의 수가 있다고 하고, subject에 과목의 수가 있다고 하면 각 학생별 총점을 구하는 프로그램을 아래와 같이 작성할 수 있다.

```
for (i=0; i<student; i++)
{
   for (j=temp=0 ; j<subject ; j++)
   temp+=score[i][j]

   printf("%d",temp);
}
```

위의 예에서 j=temp=0;은 j=0,temp=0과 같은 의미이다. 따라서 위의 for 문은 아래와 같이 고쳐질 수 있다.

```
for (j=0,temp=0 ; j<subject ; j++)
```

1.3 배열의 초기화

여러분은 이미 단일 변수를 선언할때, 선언과 함께 변수에 초기치(initial value)를 줄 수 있다는 것을 이미 알고 있다.

```
[예]
int i = 32;
float PI = 3.1415;
```

그러나 배열에서는 모든 배열변수를 초기화 할 수는 없다.

[1] 외부배열(external array)과 정적배열(static)은 초기화가 가능하다
[2] 외부배열(external array)과 정적배열(static)은 초기치를 정해주지 않으면 자동으로 배열의 모든 원소가 0으로 초기화된다.

배열의 초기화는 "{"와 "}"를 사용하여 각 원소의 값을 ","로 분리하여 적어주면 된다.

```
int number[5] = {5,4,3,8,7};
char code[3] = {3,2,1};
```

문자형(char)배열은 C-언어에서는 타언어의 문자열 처럼 사용되므로 다음과 같이 초기화 될 수도 있다.

```
char name[5] = "choi"; <- 문자열의 끝에 널(0)이 추가되므로 5개
char name[5] = {'c','h','o','i',0};   <- 위의 문장과 같다.
```

재미있는 점은 원소의 초기값을 주면 원소의 갯수를 미리 적어줄 필요가 없다는 것이다.

ҕl

```
char name[] = "choi";    ...... 초기화를 하므로 갯수가 필요없음
char name[5] = "choi";  ...... 위의 문장과 같다.
```

ҕ

다차원 배열의 초기화는 아래와 같이 하면된다.

ҕl

```
int score[5][3] = {
    {1,2,3},{4,5,6},{7,8,9},{10,11,12},{13,14,15}
};
char name[3][5] = {"choi","kim","lee","jo","park"};
```

ҕ

ҕlsam0602.dat

```
[예제] 배열의 초기화
int days[12]={31,28,31,30,31,30,31,31,30,31,30,31};
int i;
main()
{
  for (i=0 ; i<12 ; i++)
  printf("Month %d has %d days.\n",i+1,days[i]);
}
[결과]
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
............  후략  ............
```

ҕ

단순한 프로그램 이므로 이해하는데 어려움은 없을것이다. 위에서 정적 배열은 초기화를 하지않으면 자동으로 0으로 초기화 된다고 하였다. 이번에는 그말이 맞는가를 시험해 보기로 한다.

ㆆIsam0603.dat

```
[예제] 배열의 초기화
int days[12]=
    {31,28,31,30,31,30,31,31,30,31}; ..... 10개만 정의해 본다.
int i;
main()
{
  for (i=0 ; i<12 ; i++)
  printf("Month %d has %d days.\n",i+1,days[i]);
}
[결과]
............  전략  ............
Month 10 has 31 days.
Month 11 has 0 days.      .....
Month 12 has 0 days.      ..... 정의되지 않았으므로 0이 출력된다.
```

ㆆ

2. 포인터(Pointer)

C-언어에서 포인터는 "*" 연산자를 사용하여 선언한다. 파스칼을 알고있는 사람들은 파스칼이 Pointer라는 형을 갖고 있는데서 C-언어에도 비슷한 방식의 포인터가 사용되지 않을까, 생각할지 모르지만 그런 생각은 잘못된 생각이다. C-언어의 포인터는 파스칼의 포인터 보다 한단계 발전된 형태이고 사용하기에 편리하며 훨씬 표현방법이 유연하다.

포인터(Pointer) 라는 말의 뜻을 좀더 생각해 보기 바란다. point라는 말은 "지점,점,한정된 장소"라는 의미외에 "가리키다". 라는 의미도 있다. 따라서 포인터를 "가리키는 것" 이라고 해석해도 좋을것이다.

사실 포인터는 배열과 밀접한 관계에 있으며 어떻게 보면 서로 비슷한점도 많다. 하지만 또 어떻게 보면 다른점도 많이 찾을 수 있다. 포인터는 말 자체가 의미하듯이 포인터 변수 자체가 데이터를 가지고 있는것이 아니다. 포인터 변수는 데이터의 주소만
을 가지고 작업하는 것이다.

배열은 정적인 작업에 사용된다. 다시
말해 한번 프로그램에 배열로 데이타가
정의되면 그 프로그램이 종료할 때 까
지 그 배열은 그대로 메모리를 차지 하
게 된다. 작은 프로그램의 경우에는 별
문제가 생기지 않을 수 도 있지만 프로
램이 커지고 처리해야할 데이타의 양이
많아지면 배열만으로 프로그램을 작성
하는것은 불가능 해진다.

ᆿillu0602.dat

포인터는 이러한것을 가능하게 해준다. 즉 데이타를 필요할때 할당하여 사용하고 필요 없어지면 메모리 상에서 없애버리는 것이다. 다시 자유로워진 메모리 공간은 다른 용도로 다시 사용할 수 있게 된다.

다음은 포인터를 이해하는데 필수적인 세그먼트(Segment)와 오프셋(Offset)와 관한 설명이다. 초보자에게는 어렵게 느껴지겠지만 꾹 참고(?) 끝까지 읽어보기 바란다.

ᄁ2.1 세그먼트(Segment)와 오프셋(Offset)

마이크로 프로세서 8088(XT), 8086, 80286(AT)는 기본적으로 모든 작업을 16비트로 행한다. 80386(386 기종)이 32비트 컴퓨터라고 해도 그 컴퓨터 내부에서 돌아가는 프로그램은 실은 거의 16비트로 움직이고 있다
따라서, 이론적으로 보면 정상적인 프로그램의 경우 2의 16승 비트까지 메모리를 사용할 수 있다. (흔히 사용하는 unsigned int의 크기이다.)

즉 2의 16승은 64킬로 바이트이며 0x0000 부터 0xffff까지의 메모리 주소를 가진다. 하지만 프로그램에 있어서 64킬로 바이트는 그리 많은것은 아니다. 아니 실제로는 터무니 없이 부족하다.
따라서 실질적으로 컴퓨터에서는`세그먼트와 오프셋`이라는 메모리 기법을 이용한다. 즉, 하나의 세그먼트는 64킬로 바이트의 영역을 가지며 이한 영역의 주소는 오프셋으로 결정하는것 이다.

일반적으로 세그먼트는 하나의 세그먼트에 작동하므로 불행히도 프로그래머가 64킬로 바이트 이상의 데이타(data) 에 포인터를 사용하면 이는 일반 데이타 구역을 초과하여 포인터의 코드를 재정립 하여야 한다.
ҕI
이러한 포인터를 near포인터라 한다.
ҕ

64킬로 바이트로 지정된 데이타(세그먼트) 메모리 구역밖의 메이타를 사용하는 경우 ANSI의 표준 범례에 어긋나므로 다른 컴퓨터와 프로그램의 호환성이 그만큼 떨어진다. 가장 일반적인 예로 화면의 인쇄속도를 증가시키기 위한 VIDEO 메모리의 사용을 들 수 있는데 VIDEO 메모리는 일반 데이터의 메모리 영역 64킬로 바이트 영역 밖에 위치하고 있다.
ҕI
이러한 포인터를 far포인터라 한다.
ҕ
실질적으로 프로그래머가 8088/8086/80286 구조를 가지고 있는 컴퓨터에 프로그래밍할 때 다른 컴퓨터와 최대의 호환성을 보장하기 위해서 포인터의 재정립 작업은 피하는게 좋다. C-언어 에서는 메모리의 구조를 사용하는데 사용자의 자유를 최대한 보장하기 위해서 6개의 메모리 모델을 제공한다. 이에 관해서는 관계되는 내용이 나올때 배워보기로 한다.

이제 이론은 대충 끝난것 같으니 실전에 돌입해 보자. 포인터를 사용하

기 위해서는 우선 포인터를 선언해야 한다. 포인터는 변수의 이름 앞에 `"*"`를 붙여서 선언한다고 하였다.

তl

```
int *NUM;           ....... 정수형 int를 가리키는 포인터
char *NAME;          ....... 문자형 char을 가리키는 포인터
floar *SCORE;       ....... 실수형 float를 가리키는 포인터
```

ত

각 데이타형 마다 각각을 가리키는 포인터가 위와 같이 정의될 수 있다. 다른 형 끼리도 병합하여 사용할 수 없는것은 아니나 에러를 유발하게 되므로 이점 미리 알아두기 바란다.

프로그래머가 포인터를 사용하게 되면 다음과 같은 2개의 데이타와 작업을 하는것을 의미한다.

তl

```
char *p,x;    ...... char을 가리키는 포인터 x와 char형 x를 선언
p = &x;
```

ত

[1] 포인터할 주소(Address)를 가지는 데이타(예를 들어 위의 포인터 p)
[2] 이 주소에 위치한는 객체(예를 들어 x)

포인터 p는 데이타 x의 주소를 가진다. 그리고 포인터 p는 x에 포인터 또는 데이타 x는 포인터 p에 의해 포인터되었다고 말한다. 즉,

[1] p는 x의 주소를 포함한다.
[2] p는 x에 포인터한다.
[3] x는 p에 의해 포인터 되었다.

포인터의 가장 핵심적인 사용용도는 다음과 같다.

[1] 함수 인수(function argument)로 주어지는 포인터

함수는 인수로 주어지는 포인터가 지정하는 데이타를 가지고 작업한다.

[2] 동적(Dynamic)프로그램에 사용된 포인터

동적 데이타는 heap이라는 메모리에 위치한다. 물론 메모리에 종류가 있는것은 아니고 편의상 그렇게 구분한다. 프로그래머는 주소를 이용해 동적 데이타를 조작할 수 있다. 그리고 동적 데이타는 언제든 필요할 때 만들어 져서, 필요 없을때 삭제될 수 있다.

포인터 정의는 곧 포인터하고 싶은 객체의 주소를 저장할 메모리의 예약을 의미한다. 프로그래머가 포인터를 사용하면서 객체에 대한 포인터의 초기와 작업을 하지 않아 발생하는 에러가 많으니 주의하기 바란다.

```
ㅎl
    {
      char *p;
      *p = 'C';
    }
ㅎ
```

위 예는 반드시 피해야하는 프로그램의 형태이다. *p='C'명령문은 메모리에 주소 p에 대한 공간을 예약하지 않았기 때문에 아무곳에나 문자'C'를 저장한다. 만일 문자를 저장 시킨곳이 다른 데이타가 있는 부분이라면 그 데이타는 파괴될 것이고 공교롭게도 프로그램 자체 코드(Code)가 있는 곳이라면 컴퓨터는 다운을 면치 못하게 된다. 위의 예제를 알맞게 고치면 아래와 같이 된다.

```
ㅎl
    {
      char *p,x;
      p = &x;    ...... 포인터가 x를 가리키게 함
      *p = 'C';  ...... p가 가리키는 곳이 'C'를 넣음 x='C'와 같다.
    }
ㅎ
```

예에서는 정보를 저장할 데이타의 주소를 지정한 후 여기에 포인터를 적용 했다. 포인터 p는 x의 주소를 가지며 여기에 문자 'C'를 입력한다. 포인터 정의(Pointer define)는 항상 포인터형(Pointer type) 지정과 함

께 이루어 져야 한다. 프로그래머는 주
소만 있는 포인터에 왜 포인터 형을 지
정해야 하는지 궁금할 것이다.

그 이유는 char형에 대한 포인터는 8비
트씩 이동하지만 int형은 16비트씩 이
동하는 차이점이 있기 때문이다.

옆의 그림에서 포인터를 동등하게 증가
시켰지만 int형 포인터가 char형 포인
터 보다 2배더 이동한 것을 알 수 있다

illu0603.dat

`char *p`
위의 포인터 p는 문자를 가지는 데이타에 대해서만 포인터해야 한다. 좀
더 간단히 말하면 p는 문자를 저장하고 있는 메모리하고만 작업해야 한
다는 것이다. p가 문장 "ABC"의 첫번째 문자 "A"를 포인터했다고 가정해
보자. 프로그래머가 p+2와 같이 스면 이는 문자 "A"에서 8비트씩 (문자
형 이므로) 우측으로 2번 이동한 것이므로 문자 "C"에 포인터를 이동시
킨다. 프로그래머가 포인터 p를 이용하여 데이타 x를 조작할 때

[1] p는 x에 포인터한다.
[2] indirect addressing(간접 주소지정 방법)으로 x에 액세스 한다.

표준 라이브러리 함수(Standard Library Function)의 인수는 작업 능률
과 효율성을 높이기위해 자주 포인터를 사용한다. 그리고 대량의 데이타
전체를 가지고 작업하는것 보다 필요한 데이타의 주소만을 가지고 작업
을 하는 포인터를 사용하면 프로그램의 실행속도 증가에 많은 도움을 준
다. 예를 들어 고속정렬(Quick sort) 작업에 포인터가 유용하게 사용된
다.

2.2 포인터형의 정의

t라는 형이 있다면, t형을 가지는 포인터는 "t*"와 같이 쓸 수 있으며,
"t*"의 포인터형을 가지는 데이타 x는 "t* x;"또는 "t *x;"와 같이 쓴다
좀더 정확하고 간다하게 설명하면
ㅎI

[1] x는 t형에 대한 포인터형(Pointer type) 이다.
[2] x는 t형 포인터 이다.
[3] x는 t의 포인터 이다.
[4] x는 t형에 대한 포인터 이다.
[5] x는 t를 향한 포인터 이다.

ᄒ

ᄒI

[예] int *x;

ᄒ

예에서 x는 정수형(int)형에 대한 포인터로 정의하였다. int*는 어떤 정수형에 대한 포인터이므로 x는 정수형 포인터라고 말할 수 있다. 명령문 "int *x"는 "int* x"로 바꾸어도 무방하다.

ᄒIsam0604.dat

```
[예제]
main()
{
  int x=10,*p;        <- 정수형 포인터 p와 정수 x를 선언

  p = &x;              <- p에 x의 주소를 넣음 따라서 *p는 x와 같다.
  printf("%d",*p)   <- p가 가리키는 값 즉, x를 표시한다.
}
[결과]
10
```

ᄒ

여기서 x는 정수형 이며, p는 정수형 포인터 이다. 명령문 p=&x;는 p를 x에 포인트할 수 있게 해준다. 이러므로써 *p는 p로 포인터한 x를 가진다. (완전하게 동일하게 사용)

ᄒI

p : 포인트하는 곳의 메모리 주소
*p : 포인트하는 곳의 메모리 주소에 있는 데이타

ᄒ

2.2.1 문자(charater)에 대한 포인터

문자에 대한 포인터는

`char*`

와 같이 명령을 사용하면 선언할 수 있다. 문자에 대한 포인터형 x가 있

다면

`char *x;   또는    char* x;`

와 같은 명령문을 사용한다. *x는 메모리 첫번째 문자값을 가진다.
ꠓIsam0605.dat

```
[예제] 문자에 대한 포인터
main()
{
  char *x;
  x = "This is Charater chain";
  printf("%s\n",x);
```

.... 계속 ....
ꠓ

ꠓIsam0605.dat

```
  printf("%c",*x);
}
[결과]
This is Charater chain
t
```

ꠓ
첫번째 printf()함수에 왜 "printf("%s\n",*x)"와 같이 포인터가 나타내는 데이터를 사용하지 않았는지 궁금할 지도 모른다. 괜히 이미 알고 있는 사람에게는 오히려 혼동이 될지도 모르지만 "%s"는 문자열을 표시한다. 그런데 문자열은 C-언어에서는 일종의 포인터 이다.

두번째 printf()함수 에서 *x는 포인터된 첫번째 문자를 가리키므로 "This is Charater chain"의 첫번째 문자인 'T'를 인쇄했다. 참고로 포인터를 n만큼 이동하고 싶으면 x+n;명령을 주면된다.   즉, 예제에서 x+1은 'h'를 나타내며 x+2는 i를 x+3은 s를 .... 가리킨다.

2.2.2 배열형 포인터

배열형 포인터란 데이타가 아닌 포인터로만 구성된 배열문을 말한다. 배열형 포인터는 여러 개의 문자열을 포인터할 때 유용하게 사용된다. (예를 들어, 팝업메뉴의 문자열 처리). 다음의 예를 살펴보자.
ꠓIsam0606.dat

```
[예제] 배열형 포인터의 사용
main()
```

```
{
   char *x[]={"chio","min","suk"};
   printf(x[0]);
   printf(x[1]);
   printf(x[2]);
}
```

ㅎ

예의 x[0]은 문자열 "choi"가 시작하는 곳의 주소를 가지며, x[1]은 문자열 "min"이 사작하는 곳의 주소를 가진다. 배열형 포인터는 이차 배열문과 완변한 호환성을 가진다. 따라서 *x[]의 선언을 x[3][4]와 같이 변형 해도 된다. 단 포인터는 동적인 메모리 할당을 하며 배열문은 정적인 베모리 할당을 한다.

2.2.3 문자형을 제외한 포인터형

ㅎl

```
float *x;
```

ㅎ

x는 float형(즉, 실수)에 대한 포인터이다. 따라서 포인터 x는 float형 포인터 이다.

ㅎl

```
double *x;
```

ㅎ

x는 double형(즉, 배정도 실수)에 대한 포인터이다. 따라서 포인터 x는 double형 포인터 이다.

ㅎl

```
t *x;
```

ㅎ

x는 t형에 대한 포인터이다. 따라서 포인터 x는 t형 포인터 이다.

ㅎl

```
void *x;
```

ㅎ

x는 void형에 대한 포인터이다. 따라서 x는 void형 포인터이다. 그리고 x는 모든형의 포인터와 작업이 가능하다.

ㅎl

```
char *x[3];
```

ꯍ
x는 문자형에 대한 포인터이다. 즉, x는 3개의 문자형 포인터를 가진 배열문이다.

ꯍI
```
short *f();
```
ꯍ
f는 short형에 대한 포인터형을 리턴값(return value) 으로 주는 함수이다.

2.3 배열과 포인터 (Pointer to arrays)

여러분은 이미 앞에서 포인터에 대해 배웠다. 이미 한번 배운바가 있지만 정수를 가리키는 포인터(정수형 포인터)와 문자를 가리키는 포인터(문자형 포인터)가 어떤 차이가 있는지 아직도 모르는 사람이 있을 것이다. 이제 배열과 포인터를 함께 사용하여 보면 그 차이를 확실히 알 수 있을것이다.

예를 들어 number[]라는 배열이 있다고 하자. 이때 배열 이름인 number는 사실 배열 number[]의 첫 번째 원소가 기억된 주소를 나타낸다.

`number == &number[0]`

단, number는 변수가 아니고 포인터 상수(Pointer constant)라는 점을 명심해야 한다. 즉,`배열의 이름은 포인터 상수가 된다.`다음의 프로그램을 보면 좀더 확실하게 이해할 수 있을 것이다.

ꯍIsam0607.dat
```
[예제] 배열과 포인터의 사용
main()
{
  int number[]={1,4,8,12};
  int i,*p;

  p = number;
  for (i=0;i<4;i++)
```

```
    printf("pointer %d value %d\n",p+i,*(p+i));
  }
  [결과]
  pointer 60200 value 1
  pointer 60202 value 4
  pointer 60204 value 8
  pointer 60206 value 12
```

"pointer"라고 나오는 부분은 실제로 배열 number[]가 수록되어 있는곳의 메모리 번지이다. 그 값은 시스템마다 틀리게 나오므로 이상하게 생각하지는 않아도 된다. 단지 중요한 것은 포인터가 다음 데이타를 나타내기 위해 증가할때 2바이트씩 증가한다는 것이다. 물론 예제에서 문자형 포인터를 사용하였다면 1바이트씩 늘어날 것이고 실수인 float형을 사용하였다면 4바이트씩 늘어 났을 것이다.

위의 예제에서 메모리에 데이타가 수록된 모양은 옆의 그림을 참고하기 바란다.

illu0604.dat

2.3.1 포인터변수와 포인터상수

앞에서 "*" 연산자를 사용하면 포인터가 가리키는 곳의 데이타를 취한다고 하였다. 아직까지 이 말의 의미를 모른다면 큰 일이므로 이해가 되지 않으면 뒤로 다시 가서 자세히 살펴보기 바란다. 어쨌든 배열의 경우도 마찬가지로 포인터를 증가시켜 각 원소의 값을 취할 수 있다.
즉, 다음과 같은 동등관계가 성립한다.

```
   dates+2   == &dates[2];
 *(dates+2) ==   dates[2];
```

여기까지 이해가 되는까 ? 된다면 당신은 장한 학생임에 틀림이 없다.

여기서 p = dates; 하여 date를 p에 포인터한 경우에는 dates 대신 p를 사용해도 된다.

```
   p+2   == &dates[2];
 *(p+2) ==   dates[2];
```

기본적으로 배열이름과 포인터는 같은 것이다. (중요함)   다만, 배열의 이름(date)은 상수(포인터)이고 포인터(p)는 포인터형 변수라는 점이 다르다. 실제로 컴파일러는 배열을 포인터로 취급한다. 배열이 포인터 상수라는 점이외에 또 하나의 차이점은 배열이 선언될 때 그 배열에 크기에 상당하는 기억장소가 할당된다는 점이다.

[확인]

포인터를 사용할때 아래의 두개를 혼동 하여서는 않된다.

```
[1] *(dates + 2 )
[2]  *dates + 2
```

연산자는 산술연산자(+,-,,/,%)보다 우선 순위가 높다.
따라서 *(dates+2)는 dates의 세번째 원소가 되며 *dates+2는 dates의 첫번째 원소에 2를 더한값이 된다.

배열과 포인터가 기본적으러 같다는 개념은 매우 중요하다. 한 예로 함수(Function)에 배열을 넘겨줄 때 인수로서 포인터가 이용된다. 즉, 부함수에 배열 그 자체를 넘겨주는 것이 아니라, 그 배열의 첫 번째 원소를 가리키는 포인터(주소)만을 넘겨준
다.

그러면 부함수 에서는 그 포인터를 이
용하여 배열의 각 원소들을 취할 수있
게된다.

illu0605.dat

2.4 함수에서의 배열과 포인터

배열은 여러개의 자료로 이루어져 있기 때문에 원소 모두를 부함수에 전달한다는 것은 비효율적인 일이며 사실상 불가능한 것이나 마찬가지 이다. 그래서 배열을 부함수에 전달할 때에는 배열의 각 원소대신 그 원소의 첫번째 포인터만을 부함수로 전달하여 그함수에서 배열을 전부 이용할 수 있도록 한다. 아래의 예는 배열을 전달하고 받는 예이다.

sam0608.dat

```
void print_array(int *age)
{
  int i,j,k;

  for (i=0;i<5;i++)
  printf("%d ",*(age+i)); ..... *age+i 가 아님
}

main()
{
```

.... 계속 ....

sam0608.dat

```
  int a[]={3,6,9,12,15};
  print_array(a);
}
[결과]
3,6,9,12,15
```

위 예제로 함수에 배열이 전달될 때에는 포인터를 사용한다는 사실이 명백히 밝혀졌다. 위 예제의 print_array()함수 에서는 "int *age" 정수형 포인터 변수를 사용하였지만 사실 "int age[]"와 같이 정수형 배열을 사용하여도 된다.

ㅎl

```
void print_array(int *age)
{
  int i,j,k;

  for (i=0;i<16;i++)
  {
    printf("%d ",*(age+i));
    *(age+i)='0';
  }
}

main()
{
  int a[]={3,6,9,12,15};
  print_array(a);
```

ㅎ

위의 예제는 잘못된 예제로서 컴파일을 해보려는 시도는 하지 않는게 좋다. 왜야하면 제대로 실행될 수 없는 예제이기 때문이다.

먼저 위의 예제는 두가지가 잘못되었음을 먼저 알린다.

ㅎl

첫째. 함수 main()에서는 a[]배열을 분명 5개로 선언하였는데 함수 print_array()에서는 "for (i=0;i<16;i++)"처럼 16개의 데이타를 사용하려는 실수를 저질렀다.
printf("%d ",*(age+i)); 호출 역시 i가 5보다 작을 때 에만 정상적으로 동작한다.

둘째. 16개의 데이타를 화면에 표시하는것 만으로는 큰 문제가 되지 않는다. 엉뚱한 값이 출력될 뿐이다. 그러나 "*(age+i)='0';"와 같이 값을 벋어난 변수에 엉뚱한 값을 집어 넣으면 자칫 시스템이 다운될 우려가 있다.

위의 예제 역시 억지로 컴파일하여 실행하려고 한다면 시스템 이 다운될 것이다.

ㅎ

실제로 이런 실수(?)는 초보자들 만이 범하는것은 아니다. 오히려 어느 정도 실력이 있는 프로그래머에게 자주 일어나는 실수인 것이다.

2.5 포인터를 이용한 배열의 처리

여기서는 배열을 함수에 전달하여 처리하는 예로서, 배열에 들어 있는 값들의 평균을 구하는 함수 mean()을 작성해 보자.
main() 함수 내에서 우리가 만들고자 하는 함수 mean(numbers,size); 는 다음과 같이 사용될 수 있을것이다.

`printf("mean value is %d",mean(numbers,size);`

여기서 numbers는 평균을 구할 자료가 들어 있는 정수배열 이름이고, size는 그 배열의 크기(원소의 수) 이다. 이제 보여줄 예에는 이제껏 우리가 배우지 않은 내용도 나오게 된다. 이제껏 배우지도 않은 함수에 대한 내용을 공공연하게 사용하였는데 이점에 대해서는 양해를 구한다. 예제를 작성하기 위해서는 어쩔 수 가 없었다. 어쨌든 지금껏 사용했던 함수들은 모두 값을 넘겨주지 않는 void형 함수 였는데, 지금은 배열의 평균을 넘겨주는 함수를 작성하려고 한다. 일단은 값을 넘겨주는 함수를 선언 할 때에는 void대신 넘겨줄 데이타를 대신 써준다는 정도만 알아두기로 하자.

일단은 예제를 이해하는게 목적이고 함수에 관한 전문적인 내용은 다음 단원에서 자세히 다루게될 내용이기 때문이다. 따라서 이 부분이 이해가 되지 않는다고 해서 걱정할 필요도 없는것이다.

ㅎl

```
int mean(int numbers,int size)
      ... int형을 넘겨주는 함수이므로 void
                    형 함수가 아닌 int함수로 선언
{
long sum=0;
              for ( i=0 ; i<size ; i++)
          sum += *(numbers+i);    <-  sum에 모든 원소를 더한후
          sum /= size;            <-  size로 나누어
```

```
return(sum);            <-  그값을 넘겨준다.
}
```
ㅎ

설명한 예제로 작성한 완벽한 프로그램을 예로 들었다.

ㅎIsam0609.dat

```
int mean(int numbers,int size)
{
long sum=0;
for ( i=0 ; i<size ; i++)
sum += *(numbers+i);
sum /= size; return(sum);

main()
{
  int num[]={5,7,4,8,3,9};
  printf("average %d",mean(num,6));
}
```

[결과]

6

ㅎ

위의 예제로 배열이 함수로 전달되어 포인터로 사용되는 예를 보았다. 지금 혹시 이 프로그램으로 공부하는 사람중에 C-언어를 배우기 전에 다른 언어를 사용해 보았다면 한번 대답해 보라. 다른 언어에서도 배열을 포인터로 받아 이처럼 명확한 구조로 사용할 수 있는지. 어쨌든 위의 예를 보면 아래와 같은 결론을 얻을 수 있다.

ㅎI

x[0]는 *x와 같다.
x[1]는 *(x+1)와 같다.
x[2]는 *(x+2)와 같다.

ㅎ

아직도 위의 내용이 이해가 되지 않는다면 당신은 다음단원으로 넘어가서는 않된다. 다시 한번 공부하기 바란다.

2.6 함수에 대한 포인터

수학적 표현으로 g.f(x)는 프로그램에서 g(f(x))와 같이 쓰여질까 ? 이런 형식으로 C-언어에서는 g(f,x)와 같이 표현하며 이때 f는 함수에 대한 포인터이다. 다음과 같은 프로그램을 연구해 보자.

ㆆI

인수값의 제곱승을 계산하는 square()함수
인수값의 삼승을 계산하는 함수 triple()

ㆆ

어떤수의 제곱승을 한 후 이 결과를 다시 삼승하는 식을 생각해 보자. 이를 수학적으로 표현한다면 triplesquare(x) 또는 triple(square())처럼 표기할 수 있겠다. 그리고 이를 프로그래밍할 때 square()함수와 triple()함수가 있음에도 불구하고 다시 triplesquare()함수를 만드는 것은 그리 바람직한 프로그램 방식이 아니다. 따라서 가장 효율적인 방식은 triple()함수에 square()함수를 삽입하는것 인데 이 경우에는 포인터를 이용하면 쉽게 문제를 해결할 수 있다.

즉, triple()함수가 square()함수의 결과값을 인수값으로 받을 수 있도록 만들면 되는데, 이때 triple()함수에 square()함수의 주소를 가지는 포인터를 사용하면 된다. 즉,

```
triple(square,value);
```

와 같은 방식의 명령문을 사용하면 된다. 어떻게 이런일이 가능한지 궁금한 사람도 있을것이다. 하지만 이런일을 가능하게 하는것이 바로 C-언어의 매력이다.

ㆆIsam0610.dat

```
int square(int i);
int triple(int(*i)() , int i);   <- 함수의 프로토타입 선언
                                          후에 배워보도록 합시다.

int square(int i)
```

```
      {
```

.... 계속 ....

ꂋ

ꂋIsam0610.dat

```
         return(i*i);
      }
      int triple(short(*f)(), int i)
      {
         return(((*f)(x))*3);
      }
      main()
      {
         int u,v;
         v=10;
         u=triple(square,v);
         printf("%d",u);
      }
```

ꂋ

위의 예제는 분명 상당한 수준의 예제이다. C-언어를 제대로 이해하는 사람들만이 이해할 수 있을 정도이다.

다만 위의 예제로서`함수의 포인터(함수를 카리키는 포인터)`만으로 함수를 호출할 수 있다는 것을 알 수 있다면 일단은 성공이다.
옆의 그림에서 triple()은 함수에 대한 포인터 (2번 선언)와 int형 수(3번 선언)를 인수로 가지며, int형의 수
(1번선언)를 리턴값(return value)으로
준다.

식별자(identifier) square()함수는 함
수 square()에 대한 포인터를 나타낸다

ᆨillu0606.dat

2.7 포인터의 연산작업

ᆩ2.7.1 포인터의 대입

분명히 포인터도 하나의 데이타이기 때문에 포인터값을 저장할 메모리를 예약한다. 그리고 메모리에 저장되는 포인터값은 포인터 하고 싶은 데이타의 주소를 의미한다. 상수형 포인터를 제외한 어떤 포인터라도 같은형의 포인터나 void형 포인터에 의해 대입작업이 이루어 질 수 있다.

ᆮlsam0611.dat

```
[예제] 포인터의 대입
main()
{
   char *x="OK";
   char *y="NO";
   x=y;
   printf("%s",x); ...... "NO"가 출력된다.
}
```

ᆮ

x와 y는 같은 형 포인터이므로 x가 y의 정보인 "NO"를 대입받았다. 예에서 실질적으로 x가 대입받은 값은 "NO" 문자열이 아니고 이 문자열이 시작하는 주소를 대입받았다(포인터 이므로).

ᆩ2.7.2 포인터의 증가(incremental)

어떤 형의 포인터에 n만큼 수를 더하면 이 형은 포인터를 n만큼 증가(즉, 우측으로)시켜 이동하라는 뜻이다. p가 문자형 포인터이며 문장 "Good morning"의 문자 "G"에 포인터하고 있다고 가정하다. p+11 명령은 포인터를 11만큼 증가시킨 것으로 "g"에 포
인터를 이동시킨다.

옆의 그림에서 p+3과 같은 명령은 정의된 메모리 영역밖에 위치하므로 프로그램 실행시 에러를 발생시킬 우려가 있다. (그러나 항상 에러를 유발하지는 않는다) . 위에서도 설명한 적이 있지만 단지 데이타를 화면에 출력시키거나 하는 일을 할 때에는 엉뚱한 값이 출력되는 것 뿐 이다. 하지만 이 역시 일종의 버그이고 때에 따라 심각한 문제를 야기 시킬 수 도 있다.

자신이 열심히 작성한 프로그램이 실행되는 도중 갑자기 시스템이 다운된다면 약 10퍼센트 정도는 이런 이유에서 일 것이다. 따라서 포인터를 많이 사용하는 프로그램을 작성할 때 에는 그만큼 더 주의를 기울여야 한다는 뜻이 된다.

illu0607.dat

2.7.3 포인터의 감소(Decremental)

어떤 형의 포인터를 n만큼 수를 빼면 이 형의 포인터를 n만큼 감소(즉, 좌측으로) 시켜 이동하라는 뜻이된다. "Good morning"의 11번째 문자 "g"에 위치한 포인터에 p-6 명령을 주면 포인ㅌ터는 문자 "m"에 위치한 다.

sam0612.dat

```
main()
{
  char *x="Good morning",*p;
  p = x+3;
  printf("%s\n",p-3);
  printf("%s\n",p-2);
  printf("%s\n",p-1);
}
[결과]
Good morning
ood morning
od morning
```

2.7.4 두 포인터의 뺄셈

두 포인터의 합산값은 프로그램 상에서 아무런 의미도 지니지 않는다. 그러나 반대로 두 포인터의 차이값은 두 주소 사이에 존재하는 정보의 양을 알 수 있게 해준다. 두 포인터 차이값의 계산은 항상 같은 형 포인터 사이에서 이루어져야만 한다.

ᇹIsam0613.dat

[예제] 두 포인터의 뺄셈

```
main()
{
   char *s="Morning";
   char *p,*q;
   p=s;
   q=s+6;
   printf("%d",p-q);
}
[결과]
6
```

ᇹ

2.7.4 두 포인터의 비교

프로그래머는 같은 형의 포인터를 서로 비교하여 (특히 NULL 비교) 아주 유용한 작업을 할 수 있다.   특히 NULL값은 포인터 값이 없다는(즉, 0) 뜻이며, 어떤 형태의 포인터라도 NULL로 초기값을 부여할 수 있다. NULL은 상수형으로 프로그램 서두에`"stdio.h"`에 정의되어 있다.

ᇹIsam0612.dat

```
[예제] 두 포인터의 비교
#include <stdio.h>
char *p;

void print(char *x)
{
   if (x==NULL) printf("Pointer is NULL\n");
            else printf("%s\n",x);
}

main()
```

.... 계속 ....

ᇹ

ថsam0612.dat

```
{
   print(p);
   p="Program C";
   print(p);
}
[결과]
Pointer is NULL
Program C
```

ច
포인터 p는 함수 바깥에 위치하는 전역변수 이기 때문에 자동으로 NULL로 초기값이 설정되며, 프로그래머는 이를 이용하여 어떤 데이타에 대한 포인터 작업이 누락되었는 지를 확인할 수 있다.

2.7.5 포인터의 변환(Conversion)

포인터는 메모리 주소만을 가질 수 있으며 포인터에 대한 작업은 같은형(Type)일때만 가능하다. 따라서 예를들어 char형 포인터를 int형에 적용하려면 char형 포인터를 int형 포인터나 int형과 유사한 형으로 변환해야 한다. 다행히도 C-언어에서는 이런 포인터 변환 작업을 도와주는 피상적인 void형이 존재하고 있으며 이 void형 포인터는 어떠한 형의 포인터하고도 함께 작업할 수 있다.
ថ

```
char *p1;
void *p2;
   .
   .
p1 = p2;
```

ច
서론

C-언어의 가장큰 특징은 함수로만 구성되어 있다는 것이다. 함수에는 C-언어에서 제공하는`표준함수`(printf(), scanf() ....)가 있으며 컴파일러가 제공하는`컴파일러 함수`(turbo-C에는 대부분의 그래픽 함수), 그리고 사용자가 정의하여 사용하는`사용자 정의함수`가 있다. 사용자 정의 함수는 표준 함수나 컴파일러 함수가 제공하지 않는 기능들을 사용자

가 정의하여 사용하는 것을 말한다. 이 장에서는 사용자 정의 함수에 대해서 중점적으로 다룬다. C-언어의 표준함수나 컴파일러 함수는 예가 나올 때마다 설명을 할것이나, 설명이 부족할 때는 C-박사를 참조하기 바란다.

1. C-언어의 함수 개론

C-언어의 소스 프로그램(Source program)은 소스모듈(Source module)이라고 불리는 파일들로 구성된다. 소스 모듈은 쪼갤 수 없는 하나의 함수로 이루어져 있다. C-언어의 함수는 파스칼 또는 PL1언어의 포로시저(Procedure)나 FORTRAN의 서브루틴(Sub-routine)과 비슷하다고 볼 수 있다.

C-언어에서 함수를 조작할 때는 다음과 같은 점을 주의해야 한다.

তI

[1] 함수는 항상 "{"로 시작하여 "}"로 끝난다.

[2] 명령문 블록 선언부(declaration part)와 작업부(action part) 가 서로 뒤섞여 사용될 수 있다.

ত

함수에서 가장 중요한 것은 함수가 어떻게 작성되었느냐 보다는 함수가 어떤 인수를 가지며 어떤 리턴값을 돌려주느냐를 아는것이 중요하다.

তIsam0700.dat

```
[예제] 간단한 함수를 사용하는 예제
int f(int i);      ..... 함수의 프로토 타입
int f(int i)
```

```
{
   return(i*i);
}
main()
{
   printf("%d",f(43));
}
[결과]
1849
```

예의 함수 f()는 int형 인수를 가지며 인수의 제곱값을 리턴값으로 준다 또한 f는 함수 f()가 위치한 메모리 주소값을 가지는 포인터이다. 이렇게 함수의 주소를 포인터로 가지고 있게 됨으로써 다른 언어에서는 상상할 수 도 없는 여러가지 일이 가능해진다. 이들에 대한 것들은 후에 알아 보기로 한다.

[1] 함수는 항상 선언 명령문에 의해 선언되어야 하며 선언 명령문에는 함수의 형(type), 이름(name), 인수(argument) 등이 포함되어야 한다.
[2] 함수의 선언이 있은 후 프로그래머는 비로소 함수를 정의(define)할 수 있다.
[3] 함수의 선언과 정의가 있은 후에만 비로소 프로그래머는 이 함수를 호출(call)하여 사용할 수 있다.

2. 함수의 프로토타입

일반적으로 C-언어에서 사용되는 모든것들은 사용되기 전에 꼭 선언되어야한다. 함수도 예외는 아니어서 꼭 다음과 같은 단계를 거쳐야만 한다.

```
  +---- 선언 - 함수의 프로토타입       +---- 선언 - 함수의 프로토타입
  +---- 정의 - 함수 본체               +---- 사용 - 함수 호출
  +---- 사용 - 함수 호출               +---- 정의 - 함수 본체
```

ㅎ

함수의 선언은 일반적으로 함수의 프로토타입(prototype)화라고 말한다.
함수의 선언은

ㅎl

[1] 함수의 사용이 함수의 정의 전에 이루어지는지 알려준다.

[2] 컴파일러에 함수의 제어를 요청한다.

ㅎ

함수는 다음과 같은 방법으로 선언한다.

ㅎl

함수의 선언

```
  함수형   함수이름    인수형, 인수이름
      |    |               |
[예] char test(int x, char c.....);
                  |    |
            인수형 인수이름
```

ㅎ

함수의 선언은 함수의 동시적 사용문제에 대한 해결책을 준다.

ㅎl

```
main()
{
```

.... 계속 ....

ㅎ

ㅎl

```
int f();   ... 함수의 프로토타입
int g();   ... 함수의 프로토타입

int f();  {
   g();
}
int g();  {
   f();
```

```
        }
        main()    {
          f();
          g();
        }
      }
```
ъ

함수 두개가 서로 다른 함수를 호출하는것은 함수의 선언이 없으면 에러가 발생한다. 의심이 가면 시험해 보기 바란다.

어떠한 리턴값도 가지지 않는 함수를`void형 함수라`부르며 리턴값을 가지는 함수의 리턴값 형은 함수형이다. 함수의 선언은 컴파일러가 함수 정의를 확인할 수 있게 해 준다.

ъIsam0701.dat

```
      int f(double x);
      int f(double x)
      {
        return(x*x);
      }
      main()
      {
        double d;

        d=3.14159;
        printf("%d",f(d));
      }
```
ъ

ъIsam0701.dat

```
      [결과]
      9
```
ъ

예는 함수의 프로토타입(또는 선언)이 항상 함수 정의에 대한 모든 에러를 검출하지 못하는것을 잘 보여주고 있다. 즉, 함수 f()는 int형으로 선언되었으나 함수의 인수형과 제곱승을 하는 리턴값은 double형이다. 그러나 함수 f()가 int형이므로 리턴값은 3.14159를 함수 f()의 인수형으로 주면 정수 9라는 어처구니 없는 리턴값을 준다. 이런 경우 함수

f()는 int형이 아닌 double형으로 하는것이 원칙이다.

3. 함수 본체의 정의

프로그램 서두에 선언된 함수(프로토타
입화된 함수)는 반드시 프로그램에서
정의되어야 한다. 즉, 함수의 정의란
함수가 프로그램에서 어떤 역할과 작업
을 해야되는지를 지정해 주는 것이라고
말할 수 있다.

함수의 정의는 옆의 그림과 같은 형식
을 취한다.

illu0700.dat

4. 함수의 사용영역

함수는 다음 두가지 방법으로 사용될 수 있다.

전역적 함수 - 모든 프로그램에서 사용
지역적 함수 - 소스 파일(source file)에서만 사용

4.1 전역적 함수

아직까지는 그럴 필요성을 느끼지 못하겠지만 나중에 아주 큰 프로그램
을 작성하다 보면 하나의 소스 파일로는 프로그램 작성이 어려워지게 된

다. 예를들어 프로그램의 소스가 5000라인 가량되면 하나의 파일로는 도저히 감당할 수 가 없게 된다.

이럴때에는 프로그램을 여러개의 소스파일로 나누어 작성하면 되는데 이때 한소스에서 만들어진 함수를 다른 소스에서 쓸 수 있게 하려면 함수를 전역적 함수로 선언하여야 한다.

A와 B라는 두개의 소스파일을 가진 프로그램을 예로 들어보자. 소스파일 A에서 선언및 정의된 함수는 일반적으로 소스파일 A에서만 사용이 가능하다. 그러나 extern명령을 사용하여 함수를 선언하면 소스파일 B에서도 함수의 사용이 가능하다.

তI
 참고로 대부분의 C-언어 컴파일러는 extern명령을 초기값으로 지원한다.
 Turbo-C나 Borland C++도 예외는 아니다.
ত
그러나 프로그램의 판독력과 이해력을 높이기 위해서 프로그래머는 extern명령을 프로그램에 써넣는 것이 바람직하다. extern명령은 또한 함수의 프로토타입화와 컴파일러에 대한 함수 제어(Function control)에 일익을 담당한다.

তI

```
/* 두개의 소스파일중 p1.c 라는 첫번째 소스파일 */
long f(long i);
long f(long i)
{
   return(i*i);
}
```

```
/* 두개의 소스파일중 p2.c 라는 두번째 소스파일 */
extern long f(long i);
main()
{
```

```
    printf("%d",f(100));
}
[결과]
10000
```

예의 두 소스 프로그램을 각각 컴파일한 후 링크(프로그램 연결 편집)를 해야만 비로소 하나의 실행파일(*.EXE 파일)을 얻을 수 있다.

자신의 프로그램을 만들기 위해서 컴파일러와 링커를 사용하는 방법은 컴파일러 단원에서 자세히 배우게 된다.

함수 long f(long i); 는 첫번째 소스파일인 p1.c에서 선언및 정의 되었다. 따라서 프로그래머는 이 함수를 첫번째 소스파일인 p1.c 안에서 얼마든지 사용할 수 있다. 또한 함수 long f(long i); 는 두번째 소스파일인 p2.c에서 extern명령을 이용하여 재선언 되었다. 따라서 이 함수는 두번째 소스파일 p2.c 안에서도 사용이 가능하게 되었다.

만약에 프로그래머가 두번째 소스파일 p2.c 에서 함수 f()선언을 생략했다면 컴파일러는 이를 에러처리할 것이다. (함수 f()의 리턴값을 초기값인 int형으로 받아들이기 때문) 그러나 일반적으로 extern명령은 초기값으로 지정되어 있기 때문에 생략해도 컴파일 작업에는 무관하다.

4.2 지역적 함수

프로그래머는 함수가 어떤 특정한 소스파일에만 국한적으로 사용되도록 만들 수 있다. 이러한 함수를 지역적 함수라고 말하며, 다음과 같은 방법을 사용하여 만들 수 있다.

함수 선언문 서두에 static 명령을 삽입한다.

사실 함수를 굳이 지역적 함수로 만들어야 하는 일은 흔치 않다. 그냥 함수가 선언되지 않은 소스파일에서 함수를 사용하지 않으면 그 뿐이기 때문이다.

5. 함수의 인수

함수의 인수를 통하여 함수의 작업에 필요한 데이타나 수치 등이 제공된다. 물론 인수대신 프로그램 전체에서 사용되는 전역변수를 사용할 수도 있으나 프로그램의 판독성을 떨어뜨리고 버그의 원인이 된다.
인수는 절대로 전역변수와 같이 사용할 수 없으며 사용된 함수 안에서만 사용할 수 있다.

ㅎI
함수에서 작업 진행 상황에 따라 인수(Argument)값이 변형되더라도 인수에 전달된 값(즉, origin value)는 절대로 변하지 않는다. 단, 한가지 예외가 있는데 이는 주어진 인수값이 주소값(포인터 형)인 경우 이다.
ㅎ

ㅎIsam0702.dat
```
[예제] 인수와 지역변수
void f(int i);  ....... 함수 f()의 프로토타입
void g(int i);  ....... 함수 g()의 프로토타입
```
.... 계속 ....
ㅎ

ㅎIsam0702.dat
```
void f(int i) {
   i++; printf("%d\n",i);
}
void g(int i) {
   i++; printf("%d\n",i);
}
main() {
   f(3); g(3);
}
[결과]
4
4
```
ㅎ

예 프로그램은 함수 f()와 함수 g()의 인수가 함수 안에서 부분적으로만 사용되었음을 잘 보여주고 있다. 즉 함수 f()의 인수 i는 함수 f()에서만 사용되었으며 함수 g()의 인수 i 역시 마찬가지로 함수 f()의 인수 i와는 독립적으로 사용되었다. 결과적으로 함수에 사용되는 인수는`지역` `변수`이다.

5.1 전역변수와 인수와의 관계

아직 함수를 제대로 이해하지 못하는 초보자에게서 함수의 인수 대신 전역변수의 사용하는 현상이 가끔씩 나타난다. 그러나 프로그램의 판독력 정확성 효율성을 높이기 위해서는 전역변수를 함수의 인수 대신으로 사용하는 프로그램은 바람직 하지 않다.

그 이유는 전역 변수의 경우 모든 함수가 이 전역변수의 값을 변형시킬 수 있으므로 나중에 변수의 값이 정상적이지 않을때 에러의 근원을 찾기 힘들게 된다. 더욱이 전역변수를 사용하면 프로그램이 모듈화되지 않아 프로그램의 부품화 할 수 없다. 프로그램의 부품화란 프로그램을 작성할 때 이미 작성된 프로그램들을 부품과 같이 사용하는 것을 의미 한다.

illu0701.dat

sam0703.dat

```
[예제] 전역변수와 인수와의 관계
int i;
void f();   ....... 함수 f()의 프로토타입
void g();   ....... 함수 g()의 프로토타입

void f() {
  i++; printf("%d\n",i);
}
void g() {
  i++; printf("%d\n",i);
}
main() {
  i=3; f(); g();
}
```

[결과]
4
5
ㅎ

예제는 함수의 인수대신 전역변수인 i를 사용하였으며 프로그램의 정확성과 판독력이 떨어지는 것을 한눈에 알 수 있다. 또한 함수 f()에서 사용되는 변수 i와 함수 g()에서 사용한 변수 i는 동일한 변수로써 함수의 상호독립성이 상실되는것을 볼 수 있다.

즉, 함수 f()의 변수 i는 함수 g()의 변수 i에 영향을 미칠 수 있으며
함수 g()의 변수 i또한 함수 f()의 변수 i에 영향을 미친다. (알고보면
둘은 서로 같은 변수 이므로)

5.2 인수의 전달

함수 인수의 전달은 다음과 같이 두 가지 방법을 사용할 수 있다.
ㅎl
값에 의한 인수 전달
주소에 의한 인수 전달
ㅎ

ㅎ]5.2.1 값에 의한 인수 전달

값에 의한 인수 전달이란 인수에 특정한 값(데이타)를 직접 명시하는 것을 의미한다. 값에 의한 인수 전달에서는 함수의 인수로 주어지는 원래값(origin value)는 항상 보호된다.

ㅎlsam0704.dat
[예제] 값에 의한 인수 전달
void f(int i); .... 함수 f()의 프로토타입(선언)

void f(int i) ……. 함수 f()의 정의
……………………………. 계속 ………………………….
ᄃ

ᄃIsam0704.dat

```
{
  i++;
  printf("%d\n",i);
}
main()
{
  int x;
  x=0; f(x); printf("%d",x);
}
[결과]
1
0
```

ᄃ
예에서 프로그램 진행이 함수 f()에 들어서자 마자 함수의 인수로 주어진 변수 x의 값은 x의 원래값을 보호하기 위해서 함수 f()의 인수(argument)인 지역변수 i에 복사된다.

따라서 함수 f()는 x의 복사판인 변수 i를 가지고 작업을 진행하고 인수 i는 x가 가지고 있는 값(0)을 전달받는다. 예에서는 함수를 호출할 때 사용된 x와 인수값 i가 서로 독립적인 것을 알 수 있다. 값에 의한 인수 전달은 항상 일반형 인수를 사용한다.

5.2.2 주소에 의한 인수 전달

주소에 의한 인수 전달은 레퍼런스에 의한 인수 전달이라고도 말하며, 전달되는 인수의 원래값(original value)를 변형 시키는 성질을 가지고 있다. 주소에 의한 인수 전달은 항상 포인터형 인수나 주소형 인수를 사용 한다.

```
[예제] 주소에 의한 인수 전달
void f(int *p); .... 함수 f()의 프로토타입
void f(int *p)  .... 함수 f()의 정의
{
  (*p)++;                  .... 증가
  printf("%d ",*(p));
}

main()
{
```
.... 계속 ....

```
  int x,*a;

  x = 0;
  a = &x;
  f(a);
  printf("%d",x);
  f(a);
  printf("%d",x);
}
[결과]
0 0 1 1
```

함수 f()에 주어진 인수는 변수 x의 주소이며 함수 f()는 이 주소를 이용해 변수 x가 가지고 있는 정보를 변형 시켰다. 인수 p는 포인터 형으로 주소값을 가지는 인수이며 함수 f() 안에서만 제한적으로 사용되었다 포인터형 인수 p는 변수 x의 주소를 가지며 지역변수 p자체는 변형되어

도 x에 영향을 미치지 않는다.

`(*p)++`은 주소 p가 포인터한 곳의 데이타를 1만큼 증가시켰다. 결과적으로 최초의 p는 데이타 x의 들어있는 곳의 주소를 전달받으므로 p가 포인터한 곳의 데이타를 1만큼 증가시키는 것은 x를 1만큼 증가시킨것과 같다. 이렇듯 주소에 의한 인수전달은 데이타의 원래 값을 변형 시킨다.

C-언어 에서는 t&와 같은 명령어를 사용하여 t형에 대한 레퍼런스(reference)형을 정의한다. (여기서 t는 어떠한 형이라도 상과없음) 이는 C-언어의 커다란 장점이라고 할 수 있지만 프로그래머는 이를 주의하여 사용할 필요가 있다. 왜냐하면 함수의 인수를 형으로 주는 데이타의 원래값(origin value)는 더이상 보호 되지 않가 때문이다.(포인터형 인수와 동일 함)

ᄃI

```
void f(int i,int &j);   ..... 함수 f()의 프로토타입
void f(int i,int &j)    ..... 함수 f()의 정의
{
   i++;
   j++;
}

main()
{
   int x=1,y=1;
   f(x,y);
   printf("%d %d",x,y);
}
[결과]
1 2
```

ᄃ

x는 reference에 의한 인수부여가 아니기 때문에 함수 f()가 실행된 후에도 x의 원래값을 간직하고 있다.   그러나 y는 refernce에 의해 함수 f()에 인수로 부여되었기 때문에 함수 f()의 실행 후에 원래의 y값이 변형 되었다. 예에서는 함수 f()의 실행 후 y의 값이(초기값 정수 1) 2로 변형 되었음을 알 수 있다.
그러나 x는 함수 f() 실행 후에도 그 값에는 변함이 없다.

5.2.3 제 2의 함수를 인수로 가지는 함수

이 경우에 함수의 인수는 제 2 함수에 대한 포인터값 으로 간주한다. 따라서 엄밀한 의미에서 제 2의 함수를 인수로 전달하는 것은 포인터에 의한 인수 전달과 동일하다고 말할 수 있다.

이에 대한 예제는 이미 전에 소개된 바가 있는데 함수의 인수로 또다른 함수를 줄 수 있는것은 C-언어의 또다른 특징 이자 커다란 장점이기도 하다.

프로그래머는 이러한 C-언어의 유연함을 이용하여 보다 손쉽게 프로그램을 작성할 수 있다.

illu0702.dat

다음과 같은 함수가 있다고 가정하자.

`square() 함수 - 제곱승`
`cube() 함수    - 세제곱승`
`sum() 함수     - square()함수와 cube() 함수를 인수로 사용`

```
int square(int x);
int cube(int x);
int sum(int(*f)(),int(*g)(),int i);

int square(int x) {
  return (x*x);
}
int cube(int x) {
  return (x*x*x);
```

.... 계속 ....

```
}
int sum(int(*x)(),int(*x)(),int i) {
  return((*f)(i) + (*g)(i));
}

main() {
  printf("%d",sum(square,cube,2));
}
[결과]
12
```

예의`int(*f)();`명령은 f를 함수에 대한 포인터형 데이타(pointer type object)처럼 간주하라는 뜻이다.

5.3 함수 인수의 양

여기서 배우는 내용은 C-언어 초보자에게는 약간 어렵게 느껴질 수 있으므로 일단 넘겨두었다가 나중에 보아도 상관은 없다.

간혹 프로그래머가 얼마만큼의 인수를 사용해야 할지 모르는 일이 발생할 수 있다.   다행히도 ANSI(American National Standard Institude)의 표준 규범에는 이런 상황을 예측하고 있다.

잘 이해하지 못하겠으면 printf()함수나 scanf()함수를 생각해 보기 바란다.

```
    printf("%s %d %c %x",name, age, sex, hexnum);
    printf("%s %d %c",name, age, sex,);
    printf("%s %d",name, age,);
```

함수의 인수를 자유롭게 사용할 수 있음을 알 수 있다.   이러한 것을 `가변인수`라 한다. 가변인수를 사용하기 위해서는 가변인수가 정의되어 있는`stdarg.h`파일을 포함하여야 한다. 즉,

```
    #include <stdarg.h>
```

위와 같은 문장을 프로그램에 삽입해 주어야 한다. 문장이 삽입 되었으면 이제 배울 하나의 형과 세 개의 매크로(macro)를 정의하여 사용하여야 한다.

5.3.1 var_list

이는 프로그램 서두에 정의되는 형이다. 프로그램에서 사용되는 인수는

반드시 이형으로 선언되어야 한다.

ᄒl

[예] var_list v1; ..... v1 이라는 variable list 선언
variable list란 말 그대로 간변 리스트라는 뜻이다.

ᄒ

v1는 리스트형 변수이다. 이 리스트는`var_start, var_arg, var_end`에 의해 사용될 것이다.

5.3.2 var_start

var_start는 변수 리스트의 첫번째 인수(argument)에 포인터하면서 인수 리스트를 초기화 한다.

ᄒl

[예] var_start(v1, first_argument);

ᄒ

5.3.3 var_arg

이 매크로(macro)는 현재 사용중인 인수를 되돌려 준다. 첫번째로 사용한 var_arg는 인수 리스트의 첫번째 인수를 되돌려 주며, 두번째로 사용

한 var_arg는 인수 리스트의 두번째 인수를 되돌려 준다.

```
[예] var_arg(v1,type);
```

여기서 type은 macro가 되돌려 보내야 하는 어떠한 type을 나타낸다.

5.3.4 var_end

이 매크로는 모든 포인터를 차례대로 재정렬해 주며 함수의 맨 마지막에 사용되어야 한다.

```
[예] var_end(v1);
```

좀더 쉬운 이해를 위하여 실수 n의 곱셈을 하는 다음예를 살펴보자. (여기서 n은 변수이다.)

```
#include <stdio.h>
#include <stdarg.h>                ....... 가변인수 사용을 위해
double sum(char *begin,...);   ....... 함수의 프로토타입
double sum(char *begin,...)     ....... 함수의 정의
{
................................ 계속 .............................
```

```
double x,p=1;
var_list v1;
var_start(v1,begin);
```

```
        while(1)
        {
           x=var_arg(v1,double);
           if(x!=1.0) p*=x;
           else break;
        }
        var_end(v1);
        return(p);
     }
     main()
     {
        printf("%d", sum("p=",2.0,3.0,6.0,1.0));
     }
```
ত

예제에서 함수 sum()의 호출은 5개의 인자를 사용하였다. 프로그래머가

`x=var_arg(v1,double);`

명령을 사용할 때마다 차례대로 x는 인수(argument)리스트 안의 값을 가진다.

위의 내용이 이해가 되지 않더라도 당장은 크게 걱정할 필요가 없다. 당장 필요한 내용은 아니기 때문이다. 그러나 C-언어로 프로그램을 작성하다 보면 언젠가는 다시 보아야 할 것이다.

5.4 메인함수와 인수

C-언어로 프로그램을 짜다 보면 다음과 같은 의문이 생길것이다. 도스의 카피 (Copy)명령과 같은것을 만들고 싶은데 도스상에서 인자를 받는것은 어떻게 하는것일까 ?

아래의 예는 도스상에서 프로그램을 호출할 때 인자를 사용한 예이다.

তI
```
C:\NORTON>ncc /fast <Enter>
C:\DOS>copy *.bas a: <Enter>
C:\MUSIC>play romance <Enter>
```
ত

물론 C-언어에서도 위와같이 도스상에서 인자를 받아들이는 함수의 작성이 가능하다.

C-언어 프로그램의 처음 시작점이 되는 main()함수도 역시 함수라는 것을 기억하는가 ? main()함수도 함수인 이상 인수를 받아들일 수 있다. 이때는 main()함수의 선언을 아래와 같이 해주면 된다.

তI
```
main(int argc,char *argv)
```
ত

তJ5.4.1 int argc

인수 다음에 나오는 argv[]배열문의 입력된 인수 수를 나타낸다. 만약

`play romance`

와 같이 입력했다면 argc에는 2가 입력된다. 왜냐하면 프로그램의 이름인 play도 인자로 간주되기 때문이다.

5.4.2 char *argv[]

argv는 문장에 대한 문자열 포인터형 인수를 의미한다. 또한 동일한 의미로 사용될 수 있는 char **argv의 argv는 문자에 대한 포인터의 포인터를 의미한다.

어떤 경우라도 프로그램에서는

```
argc     - 항상 1보다 같거나 큰 수를 가진다. (적어도 프로그램 이름
            이 있기 때문)
argv[1] - 첫번째로 주어지는 인수를 가리킨다.
argv[2] - 두번째로 주어지는 인수를 가리킨다.
argv[n] - n번째로 주어지는 인수를 가리킨다.
```

따라서 paly.exe라는 프로그램을 아래와 같이 입력하여 실행하면

```
play -s romance -d 136
```

argc와 argv는 아래와 같이 된다.

```
argc == 5
argv[0] = play
argv[1] = -s
argv[2] = romance
argv[3] = -d
argv[4] = 136
```

```
[예제] 인수 출력 프로그램
main(int argc,char *argv)
{
  int i;
  for (i=0;i<argc;i++)
  {
```

```
        printf("%s ",argv[i]);
      }
    }
```
ㅎ

위 프로그램을 실행 시키면 인수를 입력할 것을 요구한다. "ok"라고 입력하면 "sample ok"라고 출력할 것이다. C-홀로서기 에서는 예제 파일의 이름을 자동으로 sample.exe로 만들기 때문이다.

6. 함수에 사용되는 데이타

함수에서 사용이 가능한 데이타는 다음과 같다.

ㅎl

[1] 프로그램 전체에 선언된 전역변수
[2] 함수 안에서 선언된 지역변수
[3] 함수에 사용된 인수(Argument)

ㅎ

만약 전역변수와 지역변수가 같은이름을 가지고 있으면 지역변수가 우선권을 가진다.`<- 중요함, 꼭 기억하기 바란다.`

ㅎl

```
int x=5;           <- 전역변수 (A)
void f()
{
  int x=10;        <- 지역변수 (B)
  printf("%d",x); <- 지역변수에게 우선권이 있으므로 10이 출력됨
}
main()
```

```
      {
         x=2;
         f();
         printf("%d",x); <- 2를 출력 함수 f()에서 변형되지 않았음 (C)
      }
```
ᄒ

(A)부분에서 선언된 x는 전역변수 이며 (B)부분에서 선언된 x는 지역변수 이다. 예 프로그램에서 x=10;은 (B)부분의 x를 이용하는것 이며 이순간 (A)부분의 x는 잠깐 쉬게된다.

전역변수(A)와 지역변수(B)가 독립적으로 작동하는 것은 (C)를 보면 알 수 있다.
전역변수와 지역변수에 대해서는 데이타 단원에서 자세히 언급한 바 있다.

ᆩ7. 함수의 리턴값

함수(Function)에서

히
```
      t f(argument);   ..... t는 어떠한 형이라도 무방
```
ᄒ

이럴 경우 함수 f()는 t형이라고 말한다. 다시 말해 함수 f()의 리턴값은 t형으로 이루어져야 한다는 뜻이다. 원래 리턴값 이란 호출당한 함수가 일정한 작업을 한 후에 작업의 결과를 호출한 함수로 결과를 돌려줄때 사용하는것을 의미한다.

함수에서 리탄값은 다음과 같은 형식으로 표현한다.

히
```
      return( 리턴값이나 리턴식 );
```
ᄒ

이 명령문은 함수가 작업을 마쳤을 경우`리턴값이나 리턴식`을 이 함수를 호출했던 함수로 인도한다. 여기서 함수를 호출했던 함수란 f(argum

ent);를 사용한 함수를 말한다.

참고로 여러개의 return명령을 한 함수내에 둘 수 있다. 이럴 경우에는
if, while... 등의 명령으로 필요한 return명령을 선별해야 한다.

7.1 리턴값이 없는 함수 - void 형

t f(argument);에서 t가 void형이면 리턴값을 가지지 않는 함수이다.

ㅎl
```
void f()
{
  명령문1 - .......

  명령문n - .......
}
```
ㅎ

만약 void형 함수에서 return명령이 사용되면 리턴값을 나타낸는`리턴값`
`이나 리턴식`이 없는 return문을 사용하여야 한다.
이때 return문은 단순히 함수를 빠져나가는 역활을 한다.

ㅎl
```
void f(int x)
{
  …
  if (!x) return;    ..... 인수없이 return명령 사용 - 함수종료
  x++
  …
}
```
ㅎ

7.2 char형, int형, double형을 리턴값으로 주는 함수

이는 가장 일반적으로 사용되는 함수 형태로서 다음 예와 같은 방법으로 사용된다.

ㆆl

```
long square(int i)
{
  return (i*i);
}
long cube(int i)
{
  return (i*i*i);
}
main() {
  printf("%d",square(2) + cube(3));
}
[결과]
31
```

ㆆ

7.3 포인터를 리턴값으로 주는 함수

프로그래머가 함수의 동적 메모리 분배(dymanic memory allocation) 작업을 한 후 여기에 입출력을 한다고 가정하자. 이럴때 프로그래머가 초기 메모리 주소를 잃어버리지 않도록 메모리 주소를 리턴값으로 줄 수 있도록 함수를 작성할 수 있다.

메모리의 동적 관리는 이단원의 범위를 넘어가므로 메모리를 동적으로 관리하는`그래픽 단원`에서 배워보도록 한다.

1. 구조체(structure)

C-언어는 서로 다른 자료형들의 항목으로 이루어진 복잡한 자료구조,

즉 Pascal의 레코드(record) 또는 COBOL의 집단 항목(group item)에 대응하는 자료 구조를 `구조체(struct)`라고 정의할 수 있다.

구조체란 서로 다른 몇개의 자료형을 단일 명칭하에 한데 묶은 `변수들의 모임 또는 형틀`이라고 한다.

초보자들중에는 간혹 구조체라는 것을 어렵고 쓰기 불편하다는 이유로 붙이면서 기피하는 경향이 있다. 그러나 `파일 처리` 부분에서는 구조체를 모르면 그 설명을 이해하는데 어려움이 많고, 또한 `그래픽 함수나` `시간,날짜`에 관계되는 함수에서는 제법 많이 사용되고 있다.
그러므로 절대 빼먹고 넘어 가서는 않되고 가벼운 마음으로 꼭 읽고 넘어 가기를 바란다.

구조체와 배열의 차이점 비교

- 구조체 : 개개의 변수들이 서로 다른 이질(다른)형 자료들의 집합이다.

- 배　열 : 개개의 변수들이 서로 같은 동질(같은)형 자료들의 집합이다.

구조체의 장점

1)연관된 자료(변수)들을 개별적이 아닌 하나의 단위로 취급할 수가 있다.

2)함수의 매개변수로서 통체로 넘겨줄 수 있고, 함수의 return 값으로 구조체를 통체로 return 받을 수 있다.

1.1 구조체 형식

구조체는 3가지 형식으로 다음과 같이 나눌 수가 있다.

[형식 1] struct 태그명 { 항목 선언 목록 };
[형식 2] struct 태그명 식별자명 [,식별자명 ...];
[형식 3] struct [태그명] { 항목 선언 목록 } 식별자명
[,식별자명 ...];

ᄒ

[설명]
- struct : 구조체 예약어이다.
- 태그명 : 구조체의 식별자로서 형틀(template)명이다.
- 항목 선언 목록 : 서로 성질이 다른 구조체의 원소를 나열한다.
- 식별자명 : 구조체의 실체를 표시하는 변수,배열,포인터를 쓴다.

1.2 구조체의 사용법

구조체 형틀 선언
[형식 1] struct 태그명 {
항목 선언 목록 };
[설명 1]
먼저 형틀(template)을 선언한 뒤에,
그 형틀을 사용하여 구조체 변수,
배열, 포인터 등을 정의한다.

```
[예] struct tag {
        char    a;
        int     b;
        float   c;
        double  d;
        char    e[3];
        };
```

ᆨillu0800.dat

[설명]

기억장소 할당없이 단순히 자료 구조의 형틀(template)만 선언이 되었으므로 기억 영역 확보 정의를 해야만이 실제로 사용할 수가 있다.

ㅎl

기억 영역 확보 정의

[형식 2] struct 태그명 식별자명[,식별자명...];

[설명 2] 실제 기억 영역을 확보한다.

ㅎ

[예제]

1) 일반 변수명 : struct tag a;

2) 배열명 : struct tag b[10];

3) 포인터 변수명 : struct tag *c;

[설명] 위의 형태로 이용하여 a,b,c 라는 변수명,배열명,포인터명으로
실제 기억 영역이 확보된다.

-------------------- 계 속 ---------------------

ㅎl

[형식 3] struct [형틀명] { 항목 선언 목록 } 식별자명
[,식별자명...];

[설명 3] 위 형식으로 바로 실제 기억 영역을 확보할 수 있다.

ㅎ

```
[예제] struct tag{
             int    a;
             float  b;
             char   c;
             char   d[4];
             } han;
```

[설명] han 이라는 변수명으로 실제 기억 영역이 확보된다.

ㅎlsam0800.dat

```
#include <stdio.h>
main()
{
  struct style {
         char name;
```

```
            char sex;
            int  age;
            }a;          /* 형식 3 */

    a.name='K';
    a.sex='F';
    a.age=18;
    printf("No  Name  Sex  Age\n");
    printf("1    %c     %c    %d\n",a.name,a.sex,a.age);
  }
```

[결과]

```
No  Name  Sex  Age
1    K      F     18
```

ㆅ

1.3 typedef문에 의한 구조체의 형정의

[선언]

```
struct date {         /* 구조체 형틀 선언 */
       int  da_year;
       char da_day;
       char da_mon;
       };
typedef struct data sdate;  /* 새로운 데이타형 정의 */
sdate today;  /* 새로 정의된 sdate형으로 구조체 변수 today를 정의 */
```

[선언]

```
typedef struct {    /* 구조체 형틀 생략 */
        int  da_year;
        char da_day;
        char da_mon;
        } sdate;
sdate today;  /* 새로 정의된 sdate형으로 구조체 변수 today를 정의 */
```

------------------------ 계 속 ---------------------------

[선언]

```
typedef struct date {
        int  da_year;
        char da_day;
        char da_mon;
        } sdate;
```

```
sdate today;  /* 새로 정의된 sdate형으로 구조체 변수 today를 정의 */
```

টl
    FILE형 구조체의 형정의
ট

```
typedef struct {
        short           level;     /* fill/empty level of buffer */
        unsigned        flags;     /* File status flags          */
        char            fd;        /* File descriptor            */
        unsigned char hold;      /* Ungetc char if no buffer   */

------------------------- 계  속 ----------------------------

        short           bsize;     /* Buffer size                */
        unsigned char *buffer;  /* Data transfer buffer        */
        unsigned char *curp;     /* Current active pointer      */
        unsigned        istemp;    /* Temporary file indicator    */
        short           token;     /* Used for validity checking */
        } FILE;                    /* This is the FILE object     */
```

[설명] FILE형을 typedef 문으로 형정의 했기 때문에 키워드 struct를
붙일 필요가 없고 붙여서도 않된다.

```
FILE *stream, *myfile;  /* FILE형 구조체를 가리키는 포인터 정의 */
```

1.4 구조체(struct) 초기화

다른 변수들과 마찬가지로 외부 또는 정적으로 선언된 구조체는
초기화가 가능하다. 그리고 구조체 멤버보다 적은 갯수의 값이 할당
되면 나머지는 `0`이 배정된다.

[예]
main()

```
{
 struct list {
         char ch;
         char sex;
         int   age;
         } a={'T','M','20'},
           b={'S','F','25'};    /* 형식 3 */

  printf("No   Name   Sex   Age\n");
  printf("1     %c      %c     %2d\n",a.ch,a.sex,a.age);
  printf("2     %c      %c     %2d\n",b.ch,b.sex,b.age);
}
```

ㅎl

```
    [결과]
    No   Name   Sex   Age
    1     T       M      20
    1     S       F      25
```

ㅎ

```
[예] struct study {
            int a;
            int b;
            } c[4][4]={ { 1,3 },
                        { 2,4 },
                        { 5,7 } };
```

[참고] c[3][]은 0으로 배정된다.

ㅎlsam0801.dat

```
    #include <stdio.h>
    main()
    {
      int i;
      static struct month {
                     char *mon;
                     int   day;
                     } month_tab[]={"January",   31,   "February",28,
```

```
                                        "March",     31,  "April",    30,
                                        "May",       31,  "June",     30,
                                        "July",      31,  "August",   31,
                                        "September",30,   "October",  31,
                                        "November", 30,   "December",31 };
    for(i=0; i<12; ++i)
    printf("%s   %02d\n",month_tab[i].mon,month_tab[i].day);
  }
```

ǂ

ǂIsam0801.dat

[결과]

```
January     31
February    28
March       31
April       30
May         31
June        30
July        31
August      31
September   30
October     31
November    30
December    31
```

ǂ

1.5 구조체형 포인터

ǂI

구조체 포인터의 장점

ǂ

(1)배열 자체보다 배열에 대한 포인터를 다루는 것이 쉬운 것처럼
포인터 변수를 다루는 것이 더 쉽다.

(2)시스템에 따라 인수로 전달할 수 없는 것을 포인터 변수로 전달할
수 있다.

(3)많은 자료 표현들이 다른 구조체형에 대한 포인터를 포함하는 구조형
이기 때문이다.(예, 트리,리스트)

ㅎI
구조체 포인터 선언
ㅎ

```
struct tag {
        int a;
        int b;
        double c;
        };            /* 형식 1 */
static struct tag tmp[]={ { 1, 5, 1.0001 },
                          { 2, 7, 3.5512 },
                          { 3, 9, 7,1221 } };
```

[설명] static은 정적변수로서 그 파일내에서는 어디서나 참조가 가능한
내부 변수이고, tag형의 배열 tmp[]를 선언하고 초기화 한다.

ㅎI

```
    struct tag *p;    /* p가 구조체 포인터 변수가 된다. */
    p=tmp;            /* 구조체형 배열 tmp[]의 주소가 p에 배정된다. */
    *p=&tmp;          /* 구조체형 배열 tmp[]의 주소가 p에 배정된다. */
```

ㅎ

[예제] 구조체 포인터 변수 p를 이용하여 다음과 같이 배열 tmp를 참조
할 수 있다.

```
p->a=tmp[0].a;
p->b=tmp[0].b;
p->c=tmp[0].c;

p++;              /* 포인터 증가 */

p->a=tmp[1].a;
```

p->b=tmp[1].b;
p->c=tmp[1].c;

로 되어 다음 자료를 가리킨다. 또 멤버에 값을 배정하는 p->a=3456;
의 기술이 가능하다.

```
ᇶIsam0802.dat
    #include <stdio.h>
    main()
    {
      int i;
      struct tag {
             int a;
             int b;
             double c;
             };           /* 형식 1 */
      struct tag tmp[]={ { 1, 5, 1.1001 },
                         { 2, 7, 3.5512 },
                         { 3, 9, 7.1221 } };
      struct tag *p;
      p=tmp;
ᇶ
```

------------------------- 계 속 ---------------------

```
ᇶIsam0802.dat
      for(i=0;i<3;++i)
      {
        printf("%d, %d, %1.4f\n",p->a,p->b,p->c);
        p++;
      }
    }

    [결과]
    1, 5, 1.0001
    2, 7, 3.5512
    3, 9, 7.1221
```

ㅎ

ㅎl
간접 연산자의 사용 예
ㅎ
[변수 선언]

```
struct study {          /* 형식 1 */
        long int id;
        char *name;
        char grade;
        };
```

struct study tmp, *p=&tmp; *p=&tm;를 다음 struct study *p; p=tmp;와 같이 바꿀 수가 있다.

[변수 초기화]

```
tmp.grade='A';
tmp.name="KOREA";
tmp.id=930915;
```

------------------------------ 계 속 -------------------------

수식	연산 우선 순위	결과값
tmp.name	p->name	KOREA
tmp.grade	p->grade	A
tmp.id	p->id	930915
*(p->name+2)	p->name[2]	R
*(p)->name+2	*((*p)->name)+2	(틀림)
*p->name+2	*(p->name)=2	(틀림)

[*(p->name+2)의 설명]

```
p->name     ----> KOREA
p->name[0] ----> K
p->name[1] ----> O
p->name[2] ----> R
p->name[3] ----> E
p->name[4] ----> A
```

ᄒIsam0803.dat

```
#include <stdio.h>
main()
{
  struct date {
         int month;
         int day;
         int year;
         } today;  /* 형식 3 */
  struct date *d;
  d=&today;

  today.month=9;    /* 방법 1 */
  d->day=1;          /* 방법 2 */
  (*d).year=1993;  /* 방법 3 */
  printf("방법 1 : %d\n",today.month);
  printf("방법 2 : %d\n",d->day);
  printf("방법 3 : %d\n",(*d).year);
}
```

ᄒ

ᄒI

```
[결과]
방법 1 : 9
방법 2 : 1
방법 3 : 1993
```

ᄒ

1.6 구조체형과 함수

이 단원에서는 구조체와 구조체형 멤버를 전달하는 방법과 구조체형

전체를 전달하는 방법을 공부르 하고, 이것을 바탕으로 1.6에서는

포인터 기반 함수와 구조체형 기반 함수에 대히 공부하도록 하자.

ㅎI
1.6.1 함수에 구조체형 멤버 전달
ㅎ

구조체형 멤버가 문자 배열등과 같이 복잡하지 않고 단순한 변수를 전달

하는 경우에는 그 구조체형 멤버의 값을 함수의 매개 변수로 전달하면

된다.

ㅎI
구조체 선언
ㅎ

```
struct friend {
        char   x;
        int     y;
        float  z;
        char   s[10];
        } map;
```

ㅎI
함수에 구조체형 멤버 전달
ㅎ

```
function0(map.x);        /* x의 문자값 전달 */
function1(map.y);        /* y의 정수값 전달 */
```

```
function2(map.z);         /* z의 부동 소수값 전달 */
function3(map.s);         /* 문자열 s의 주소 전달 */
function0(map.s[2]);    /* s[2]의 문자값 전달 */
```

함수에 구조체형 멤버의 주소 전달

```
function0(&map.x);       /* 문자형 x의 주소 전달 */
function1(&map.y);       /* 정수형 y의 주소 전달 */
function2(&map.z);       /* 부동 소수점형 z의 주소 전달 */
function3(map.s);         /* 문자열 s의 주소 전달 */
function0(&map.s[2]);   /* 문자형 s[2]의 주소 전달 */
```

[설명] 주소 연산자(&)는 멤버 앞이 아니라 구조체 이름에 붙이고 배열명 과는 달라 구조체명 자체는 구조체의 주소를 나타내는 것이 아니기 때문에 구조체형 주소를 얻기 위해서는 주소 연산다(&)가 필요하다. 그리고 문자열 s는 이미 주소를 의미하고 있기 때문에 주소 연산자(&)가 필요 없다. (call by reference - 내용을 넘긴다.)

1.6.2 함수에 구조체형 전체 전달

call by value(값 호출)의 기법을 사용하여 구조체형 전체를 전달할 수 있다. 이것은 내용 변경은 함수내에서만 이루어지고 호출 환경이 있는 구조체에는 전혀 영향을 주지 않는다. 이때 주의할 것은 `실매개 변수와` `형식 매개 변수`의 형이 일치되어야 한다.

[예제] call by value(값만 전달)

```
struct type {
        int   a,b;
        cahr ch;
        };
```

------------------------- 계 속 ---------------------

```
main()
{
   struct type map;
   map.a=100;
   functin(map);
}

function(parm)
struct type parm;
{
   printf("%d",parm.a);
}
```

[설명] 샐매개 변수(map)와 형식 매개 변수(parm)은 형식이 같다.

1.7 포인터 기반 함수와 구조체형 기반 함수

1.7.1 포인터 기반 함수

주소 연산자(&)를 사용하여 구조체의 주소를 찾아 그 멤버 중의 하나를 참조하는 방식에서는 구조체에 대한 포인터를 사용해야 구조체와 함수를 함께 사용할 수 있다.

ᄒI

[예제] call by reference(내용을 넘겨 준다)

ᄒ

```
main()
{
  int x=2;

  add(&x);
  ptintf("x=%d\n",x);
}

add(y)
int *y;
{
  *y=*y+5;
}
```

[결과] x=2;

ᄒIsam0804.dat

```
#include <stdio.h>
main()
{
  int x=5,y=500;                                   /* 초기값 설정 */

  printf("standard          : x=%d, y=%d\n", x,y);
  call_by_value(x,y);                              /* 값만 전달 */
  printf("call by value     : x=%d, y=%d\n", x,y); /* 자료 출력 */
  call_by_reference(&x,&y);                        /* 내용 전달 */
  printf("call by refererce : x=%d, y=%d\n", x,y); /* 자료 출력 */
}
```

------------------------- 계 속 -----------------------

ᄒ

ㅎIsam0804.dat

```
call_by_value (u,v)    /* 값만 전달 */
int u,v;
{
  int t;
  t = u;
  u = v;
  v = t;
}

call_by_reference(u,v)  /* 내용 전달 */
int *u, *v;
{
  int t;
  t = *u;
  *u = *v;
  *v = t;
}
```

------------------------ 계 속 -----------------------

ㅎ

ㅎIsam0804.dat

```
결과 ]  stendard          : x=5, y=500
        call by value     : x=5, y=500
        call by reference : x=500, y=5
```

ㅎ

7.2 구조체 기반 함수

구조체 형은 매개변수로 전달될 수 있을 뿐만 아니라 값으로 반환되거나 수식에 배정될 수 있다.

ㆅsam0805.dat

```
#include <stdio.h>

typedef struct {   /*  형 정의 선언자 */
        int a ;
        char b ;
        char *c;   /* 문자형 포인터 */
        } study;
```

------------------------ 계 속 -----------------------
ㆅ

ㆅsam0805.dat

```
study fuction(u,v)
int u;
char v;
{
  study c;        /* c는 구조체 지역 변수로 명시적반환된다 */

  c.a = u;
  c.b = v;
  c.c = "KOREA";
  return c;
}
```

------------------------ 계 속 -----------------------
ㆅ

ㅎl

```
main()
{
  int in=100;
  char ch='K';
  study d;        /* 구조체 지역 변수 */

  d = function(in,ch);
  printf("in=%d, ch=%c\n",in,ch);
  printf("d_a=%d\n",d.a);
  printf("d_b=%c\n",d.b);
  printf("d_c=%s\n",d.c);
}
[결과] in=100, ch=K
       d_a=100
       d_b=k
       d_c=KOREA
```

ㅎ

1.8 중첩된 구조체 (nosted structure)

구조체형은 새로이 유도되는 자료형이기 때문에 구조체 안에 들어있는 멤버로 또다시 구조체가 쓰일 수 있다. 다시 말해서 구조체로 중첩될(nested)수 있다.
이처럼 구조체 내에 또다른 구조체가 존재할 때 이를 `중첩된 구조체` `(nested structure)`라고 한다.

```
[예]  struct tag {
             int  x;
             int  y;
             };
      struct tag list1, list2;
```

ㅎl

중첩된 구조체 사용시 주의할 점

ㄷ

구조체가 자기자신을 다시 중첩시킬 수는 없다. 그 이유는 자기자신을 중첩할 경우에는 무한 중첩이 발생하여 실제로 멤버를 참조하는 것이 불가능해지기 때문이다. 따라서 아래와 같은 구조체 선언은 절대 불가능하다.

```
sturct list {
        char nam[20];
        int flag;
        struct list a;  /* Undefined structure 'list' 에러 발생 */
        };               /* Structure size too largo */
struct tag {
        int  x, y;
        struct cord p;  /* Undefined structure 'Cord' 에러 발생 */
struct cord {
        double x, y;
        struct tag pt;
        }
```

그러나 `자기참조 구조체(self-referential structure)`는 얼마든지 가능하다. 이 경우 무한 중첩을 일으키지 않으며 무한 중첩과 자기참조는 전혀 별개이다. 따라서 아래의 경우는 가능하며 구조체 안의 멤버 pt가 구조체 자신과 동일한 형의 구조체를 가리키는 포인터라는 점을 명심하라.

```
struct list {
        int  x, y
        struct list *pt;
        };
```

1.9 자기 참조 구조체(self-referential structure)

이것은 배열이나 단순한 구조체에서
한 차원 더 높게 발전한 여러 종류의
자료 구조를 구현하기 위한 일종의
`연계 구조체( linked structure)`
이다. 그 대표적인 예로 `선형 연계
리스트(linear linked list) 와 이진`
`나무구조(binary tree structure)`등
2가지를 들수 있다.
ঢI
선형 연계 리스트 예
ঢ

```
struct list {
        char *text;
        struct list *next;
        };
/* 개개의 구조체를 리스트(list)라고
   부른다. */
```

illu0801.dat

ঢI
이전 나무 구조의 예
ঢ

```
 struct tnode {
         char *text;
         int   count;
         struct tnode *left;
         struct tnode * right;
         };
```

[설명]
연계 구조체들의 형틀을 보면 구조체
자신의 선언이 완결되기도 전에 자기
자신을 참조하고 있으므로 에러가 날
것처럼 보이지만, 자기 자신을 멤버로
등록시키는 것이 아니라, 다만 자기
자신과 동일형을 가리키는 `포인터 `
`변수`를 멤버로 선언하므로 전혀 에러
가 나지 않는다. 그러나 형틀 선언시
반드시 구조체 태그를 필요로 한다. illu0802.dat
ঢI
연계 리스트 특징
1) 메모리를 동적으로 할당할 수 있다. (dynamic menory)

2) 자료의 추가(append), 삽입, 삭제가 간편하며 빠르다.
3) 자료를 효율적으로 빠른 시간내에 탐색(search)할 수 있다.

위 같이 연계 리스트는 연계 구조체와 배열의 단점만을 골라서 보완한 자료구조이기 때문에 처리해야 할 자료가 많을수록 그 위력이 나타난다.

Isam0806.dat

```
    #include <stdio.h>
    struct tag {
            char *a;
            char *b;
            struct tag *sp;
            }  u[] = {{"QWE", "RTY", u+1 },
                      {"UIO", "PAS", u+2},
                      {"DFG", "HJK", u }};
```

Isam0806.dat

```
 struct tag *ptr = u;
 main()
 {
  printf("1-%s\n",ptr->a);
  printf("2-%s\n",u[0].sp->a);
  printf("3-%s\n",ptr->sp->b);
  printf("4-%s\n",ptr->sp->sp->b);
  printf("5-%s\n",++(++ptr)->sp->a);
  /* 괄호앞 ++은 a에 적용됨 */
 }
    [결과]
    1 - QWE
    2 - UIO
    3 - PAS
    4 - HJK
    5 - FG
    /* ++(++ptr)->sp->a); - DFG */
```

illu0803.dat

1.9.1 선형 연계리스트

선형 연계리스트 (linear linked list)는 자료가 연속적으로 연결되어 있는 구조를 의미한다.

ㅎl

선형 연계리스트(헤더 파일 정의)

ㅎ

```
#define   NULL 0
struct linked_list;
        char d;
        struct linked_list *next;
        };
typedef struct linked_list ELEMENT;    /* 형 정의 */
typedef ELEMENT *LINK;                   /* 형 정의 */
LINK head;                              /* head는 LINK형 변수 */
head = (LINK)malloc(sizeof(ELEMENT)); /* 기억 영역 확보 */
```

ㅎl

[설명]

ㅎ

1) head는 시스템으로부터 ELEMENT를 저장하기 위해 malloc을 이용하여 기억영역(메모리)을 확보하고 그 주소를 할당받는다.
2) cast 연산자는 형변환을 하고, sizeof 연산자로 연산대상의 바이트 수를 구한다. malloc은 기억 영역을 할당하는 함수이다.
3) 만일 cast연산을 사용하지 않으면 에러가 난다. head는 char에 대한 포인터가 아니기 때문에 형 불일치가 일어난다.

ㅎl

선형 연계 리스트 생성(동적 메모리 저장 과정)

ㅎ

[그림 1]

/* 기억 영역 확보 */

```
head = (LINK) malloc (sizeof
        (ELEMENT));
head-d = 'n';
head->next=NULL;
[그림 2]
head->hext=(LINK) malloc
       (sizeof(ELEMENT));
head->next->d = 'e';
head->next->next = NULL;
[그림 3]
head->next->next = (LINK) malloc
       (sizeof(ELEMENT));
head->next->next - d = 'W';
head->next->next - next = NULL;
```

illu0804.dat

1.9.2 2진 트리

트리(tree)는 노드(node)라 부르는 원소들로 구성되어 있다. 또한 루트(root) 노드를 하나 갖는데 루트(root)를 제외한 나머지 노드들은 서로 별개인 부트리(subtree)를 갖을수 있다. 이런 부트리는 루트 노드의 자손(offspring)이 된다.

[설명]
B는 A의 자손이고 자손이 없는 E,F,G, C,D는 잎(leaf)노드이다. 그리고 2진 트리(binary tree)는 두개 이하의 자손을 원소로 갖는 트리이다.

illu0805.dat

ㅎI

2진 트리 방문 방법

1)중의법(inorder) : 왼쪽부트리 - 루트 - 오른쪽부트리
2)전위법(preorder) : 루트 - 왼쪽부트리 - 오른쪽부트리
3)후위법(postorder) : 왼쪽부트리 - 오른쪽부트리 - 루트

ᅙ

ᅙl

[예] 2진 트리

ᅙ

```
#define   NULL 0
struct node {
       char d;
       struct node *left;
       struct node *right;
       };
typedef struct node NODE;
typedef NODE    *BTREE;
------------------------ 계  속 ---------------------

inorder(root)     /* 중위법 */
BTREE root;
{
  if(root ! = NULL )  {
    inorder (root->left);
    printf("%c", root->d);
    inorder(root->right);
  }
}

preorder (root)   /* 전위법 */
BTREE root;
{
  if( root ! = NULL)  {
    printf("c",roo->d);
    preorder (root->left);
    preorder (root->right);
  }
} ----------------------------- 계  속 -------------------------

postorder (root) /* 후위법 */
BTREE root;
{
  if(root ! = NULL)
  {
    postorder(root->left);
```

```
        postorder(root->right);
        printf("%c", root->d);
    }
}
```

[결과]
inorder (중위법) - A B C D E F CT
H I J
preorder(전위법) - G D B A C F E
I H J
postorder(후위법) - A C B E F D H ☞illu0806.dat
J I G

2. 공용체 (union)

공용체(union)란 동일한 기억장소를 서로 다른 데이터형이 공동으로 사용할 수 있도록 하는 유도형 데이타이다. 즉, 가장 큰 데이타형의 메모리를 서로 `공유하는 구조`이다.

```
        [형식] union 태그명 {
                        형지정자 멤버 - 선언자;
                        };
```

`[주의]`

구조체와는 달리 멤버 선언자가 각각 멤버 선언문당 하나씩이다.

주의사항

1)공용제는 각 멤보가 저장되는 기억장소의 `선두번지가 일치`하는 구조체라고 볼수 있다.

2)공용체는 서로 다른 데이타(멤버)들이 `공유`하는 것이지 결코 함께 `공존`하는 것은 아니다.

3)하나의 공용체는 하나의 데이타만 저장할 수 있고 두개 이상의 데이타는 `동시`에 저장할 수 없다.

4)컴파일러는 공용체 멤버중 바이트 크기가 `가장 큰 멤버`가 들어갈 만큼의 기억장소를 공용체 변수에 할당한다.

5)공용체 변수에 저장되어 있는 값의 실제 데이타형은 `가장 최근`에 저장된 값의 데이타형이다.

ᄚIsam0807.dat

[실습내용] 구조체와 공용체의 크기를 바이트단위로 비교해 본다.

```
#include <stdio.h>
struct sb{              /* 구조체 선언 */
        char   a;
        int    b;
        float  c;
        } s_size;

union ub;               /* 공용체 선언 */
       char   a1;
       int    b1;
       float  c1;
       } u_size;
```

------------------------ 계 속 ---------------------

ᄚ

ᄚIsam0807.dat

```
main()
{
  int i, ;;

  i = sizeof(s_size);    /* 크기를 바이트로 구한다 */
  j = sizeof(u_size);
  printf ("struct = %d Byte \n", i);
  printf ("union  = %d Byte \n", j);
```

```
	}

	[결과] struct = 7 Byte
	        union  = 4 Byte
```

------------------------- 계 속 -----------------------
ᄄ

ᄁ 구조체와 공용체를 마치면서...

지금까지 `구조체(struct)와 공용체(union)`에 대해서 공부했다.

잘 이해가 가지 않는 부분이 있더라도 일단은 다음 `파일 단원`으로

넘어가기 바란다.

좋은 학습 방법은 한번에 이해를 하고서 다음에 보지 않는 것보다

여러번 자주 반복하므로서 보다 좋은 학습효과를 기대할 수 있으며

기억에도 오래 남는다.

파일 단원을 접하면서...

여기서는 `파일`에 대해서 다루도록 하겠다. 이 파일 단원은 매우 중요한

단원으로서 정신을 바짝 차리기 바란다.

C 언어에서는 파일 입출력 함수를 다른 언어보다 월등히 많은 표준 함수

를 제공하고 있다. 때문에 그것이 초보자들에게는 큰 부담이 되기도 한다.

그러나 이 함수들은 그 만큼 제각기 중요한 역할을 하고 있다. 그래서 그

냥 쉽게 넘길만 성질의 것들이 아니므로 차근 차근 풀어 나가면 `파일`을

정복할 수 있을 것이다.

I. 개론

ㅋJ1. 고수준 입출력 함수(high level file i/o functions)

fopen/fclose	파일 개방/종결 함수
fgetc(getc)/fputc(putc)	(단일문자)파일 입력/출력 함수
fgets/fputs	(문자열) 파일 입력/출력 함수(행단위)
fscanf/fprintf	(서식화) 파일 입력/출력 함수
fread/fwrite	(블럭) 입력/출력 함수
feof/ferror	파일끝(EOF)/에러 검출 함수
fseek	파일 임의 접근 함수
fsetpos/fgetpos	파일 포인터 위치 설정/파악 함수
setvbuf,setbuf	파일 입출력 버퍼 설정 함수
getw,putw,ungetc	그 밖의 스트림에 관계하는 함수들
fdopen,freopen	
fcloseall,flushall	
fflush,rewind,remove	
fileno,ftell,rename	

2. 저수준 입출력 함수(low level file i/o functions)

open,close,create	파일 개방/종결/생성 함수
read,write	파일 읽기/쓰기 함수
eof	파일끝(EOF) 검출 함수
lseek	파일 임의 접근 함수
	그밖에 몇가지 더 있다.

3. 스트림

`스트림(stream)`이라는 것은 물리적인 디스크상의 파일과 도스 장치

DOS devices)들을 하나의 통일된 방식으로 다루기 위한 추상화된 논리

적인 장치이다. 그리고 표준 헤더 `stdio.h에 들어 있는 FILE형 구조체`

에 의해 제어된다.

4. 파일(file)

파일이란 디스크 파일 및 터미널을 비롯한 모든 입출력 장치에 적용되는
논리적인 개념이다.

ㅠ]4.1 프로그램 실행과 동시 개방되는 표준파일

기술 형태	기술자	의미
stdin(standard input file)	0	키보드 접속(표준 입력)
stdout(standard output file)	1	화면(screen) 접속(표준 출력)
stderr(standard error file)	2	화면 접속(표준 에러)

[참고]`파일 입출력 버퍼 (file i/o buffer)`는 스트림에 읽고 쓸 데이타를

기억하는 임시기억장소이다.

4.2 파일과 스트림의 차이점

ϭl

4.2.1 파일

ϭ

디스크상의 파일을 포함하여 개개의 물리적인 실제 장치(actual device) 들을 총칭한다.

ϭl

4.2.2 스트림

ϭ

실제장치들을 하나의 통일된 방식으로 이룰수 있도록 마련된 그 실제 장치들과는 독립된 접속(interface)장치이다. 그 예로 디스크상의 파일, 키보드,화면,통신포트,프린터 등을 들수 있다.

4.3 FILE 구조체와 파일 포인터(FP,file pointer)

표준헤더 stdio.h에 정의된 FILE형 구조체

```
+-----------------------------------------------------------------+
|typedef struct {                                                  |
|         short level;           /* Fill / empty ievel of buffer */ |
|         unsigned flags;        /* File status flags           */ |
|         char fd;               /* File descriptor             */ |
|         unsigned char hold;    /* Ungetc char if no buffer    */ |
|         short bsize;           /* Buffer size                 */ |
|         unsigned char *buffer; /* Data transfer buffer        */ |
|         unsigned char *curp;   /* Current active pointer      */ |
|         unsigned istemp;       /* Temporary file indicator    */ |
|         short token;           /* Used for validity checking  */ |
|         } FILE ;               /* This is the FILE object     */ |
+-----------------------------------------------------------------+
```

ㅎl

4.3.1 중요사항

ㅎ

- buffer : 파일 입출력 버퍼의 선두번지를 가리키는 포인터.
- bsize : 파일 입출력 버퍼의 크기(통상 512 바이트)
- flags : 파일 접근모드, 파일끝, 에러등등에 관한 10비트의 상태
  정보를 담고 있다.
- curp : 현재 파일 포인터
- fd : 파일 기술자 (descirptor)

ㅎl

4.3.2 현재 파일 포인터 (current file pointer,FP)

ㅎ

편의상 파일 포인터(file pointer,FP)
또는 약칭으로 FP라고 부르는데, 개념
상으로는 "데이터를 입출력할 스프링
상의 현재 위치를 가리킨다."고 생각
하면 된다.

illu0900.dat

4.4 기정의 스트림의 특성

파일기술자(handle)	도스장치(dos device)	스트림(stream)
0	CON	stdin
1	CON	stdout
2	CON	stderr
3	AUX	stdaux
4	PRN	stdprn

4.5 파일 접근 모드(file access mode)

파일 접근 모드 (file access mode)란 개방하고자 하는 스트림에 어떤 작업을 할 것인지를 결정한다.

분류	모드	의 미
읽기	r	읽기전용으로 기존의 스트림을 개방한다.
쓰기	w	쓰기전용으로 새로운 스트림을 생성한다.
	a	추가로 쓰기 위해 스트림을 개방한다.
갱신	rt	기존의 스트일을 갱신하기 위해 개방한다.
	st	새로운 스트림을 갱신하기 위해 생성한다.
	at	스트림에 추가하여 갱신하기 위해 개방한다.

ㅎI

4.5.1 각 모드별 기능

ㅎ

1)읽기 전용(read-only)

특징은 개방할 스트림쪽으로 어떤 출력도 금지시키고 말그대로 읽기만 할 수 있다.

2)쓰기전용(write-only)

특징은 개방할 스트림쪽으로 어떤 입력도 금지시키고 말그대로 쓰기만 할 수 있다.

3)갱신(update) - + 표시

읽고 쓰기(reading and writing)를 뜻한다. 이것은 개방할 스트림에 입력과 출력을 모두할 수 있다.

4)추가(append)

스트림을 개방한 직후에 파일 포인터 FP를 파일을 (end-of-file)에 위치 시키도록 한다. 그러므로 데이타를 파일끝에 덧붙일 수 있도록 한다.

ㅎI

4.5.2 낱말의 해석

ㅎ

1)기존 스트림의 개방

이미 존재하는 스트림(디스크상의 파일 또는 도스장치)을 개방하는 작업을 뜻하다. 이 경우는 스트림이 존재하지 않으면 에러가 발생한다.
예) r, r+ 모드

2)새로운 스트림의 생성

디스크상의 파일이라면 지정하는 파일명으로 디스크상에 새로운 파일을 생성한 뒤에 개방한다. 이때 디스크상에 동일한 파일이 존재한다면 그 기존 파일의 내용은 삭제된다.
예) w, w+ 모드

3)스트림에 추가하기 위한 개방

지정한 파일명의 스트림이 존재하면 그 스트림을 개방한뒤 데이타를 추가하기 위해 파일 포인터 FP를 파일 끝으로 옮기고 존재하지 않는 다면 지정한 파일명의 스트림을 생성한다.
예) a, a+ 모드

4.6 문자 입출력 모드

문자 입출력 모드(charactor i/o mode)는 스트림에 문자를 입출력 할때 텍스트 모드(text mode)와 이전 모드(binary mode) 둘중 어느것으로 할 것 인지를 결정하는 것이다. 이런 문자 입출력 모드는 결정은 어디까지나 프로그래머가 주관적으로 앞에서 언급한 `파일 접근 모드`에 따라 결정 해야 한다.

기호	기 능	적용 예
t	텍스트 모드(text mode,텍스트 모드)	순차파일
b	이진모드(binary mode,이진 스트림)	임의 접근파일

ㄷl
[설명]
ㄷ
`텍스트 스트림`에서는 논리적인 파일끝을 나타내는 `\x1a(Ctrl-Z)`문자는 파일끝(EOF,end-of-file)로 변환되어 읽혀진다. EOF 는 "stdio.h"에 정의된 매크로 상수로서 `실제값은 -1`이다.

이것을 분석해 보다면 EOF의 실제값이 -1 이었다. 그렇다면 "\xff(십진" "수로 256)"문자와 EOF와의 구별이 어렵다. 그 이유는 -1 을 1 바이트 크기의 16진수값으로 표현하면 "0xff"이기 때문이다. 이런 문제점을 해결하기 위하여 파일끝이나 입출력 에러를 나타내기 위해 EOF를 리턴하는

------------------------- 계 속 -------------------------

fgetc 등과 같은 함수들은 `int 형(부호있는 2바이트형)`으로 처리한다.

이렇게 하면 16진수로 표현했을때 "\xff"는 부호확장 없이 "0x00ff(십진"

"수로 256)"이고, EOF는 부호확장으로 "0xffff(십진수로 -1)"가 되므로

확실하게 구별할 수 있다. 따라서 이진 스트림에서는 fgetc 함수등의 리

턴값은 반드시 char형이 아닌 int형에 저장해야 한다. 그러나 텍스트 스

트림에서는 "xff"문자가 일체 없다는 전제하에 char형을 사용이 가능하다.

4.7 입출력 모드의 선택 요령

ธI

4.7.1 스트림의 문자 입출력 모드

ธ

1)텍스트 모드
- 디스크상의 모든 텍스트와일(순차파일)일때
- 도스장치 CON으로부터 입력(키보드)과 출력(화면)일때

2)이진 모드
- 임의 접근파일일때
- 도스장치 AUX, COM1, COM2, COM3, COM4, PRN,LPT1,LPT2,LPT3 일때
- .COM, .EXE 파일을 포함하여 임의의 파일을 처리 대상으로 할때

ธI

4.7.2 파일접근 모드

ㄷ
1)순차파일 - rt, wt, at 모드중에서 하나

2)임의접근 파일
입력과 출력을 번갈아 해야 할때 : r+b, w+b, a+b 모드중에서 하나
입력과 출력 둘중 한 작업만 할때 : rb, wb, ab 모드중에서 하나

3)도스 장치
rt모드(CON0,) wt모드(CON),wb 또는 wt모드 (PRN, LPT1, LPT2, LPT3),
r+b모드 (AUX, COM1, COM2, COM3, COM4)를 선택한다.

ㄷl
4.7.3 자주 사용되는 예제
ㄷ
```
# include <stdio.h>
FILE * stream, *source, *destin, *lpt2, *com2;
```

[예]
1)stream = fopen ("data.in","rt);
data.in 파일이 이미 존재해야 하며 읽기 전용 텍스트 파일이다.

2)stream = fopen("data.out", "wt);
data.out 파일은 쓰기 전용 텍스트 파일이며 기존파일이 존재하면
기존 파일의 내용은 삭제된다.

3)stream = fopen ("data.out", "at");
data.out 파일에 이미 존재하면 파일을 개방하고 데이타를 추가하기
위해 파일 포인터를 파일의 맨끝으로 이동한다. 그러나 존지하지
않으면 2) "wt" 와 같은 역할을 한다.

------------------------ 계 속 ----------------------

4)source = fopen(argv[1], "rb");
destin = fopen(argv[2], "wb");

명령행 매개변수로 지정하는 파일 argv[1]을 argv[2]로 복사하고자 할때의 준비작업이다. 어떤 종류의 파일이 복사될지 모르므로 이진 모드로 입출력해야 한다.

5)stream = fopen ("database. dbf", "r+b");
database.dbf 파일이 이미 존재해야하고 임의 접근 파일이므로 이진 모드를 취한다.

6)lpt2 = fopen ("lpt2", "wb");
제 2프린터를 사용하기 위해 * 도스장치 lpt2를 개방한다.
출력 전용의 이진 모드이므로 입출력모드로 당연히 "wb"가 된다.

7)com2 = fopen("com2", "r+b");
모뎀이 연결되어 있는 통신포트 COM2를 개방하고 이진 모드이다.

[설명] 1)부터 5)까지 파일을 개방하려다 에러가 발생하면 fopen은 NLL을 돌려준다,.
II. 표준 입출력 함수

ㅠ1. 단일문자 입력 함수

[선언] int getch(void);
[설명] 키보드로부터 문자를 읽어들이고 화면에 표시(echo)하지 않는다.
[선언] int getche(void);
[설명] 콘솔로부터 문자를 읽어들이고 화면에 출력한다.
[선언] int getchar(void);
[설명] 표준 입력 스트림 stdin상에서 문자를 읽어 들인다.
#define getchar() getc(stdin)
로 정의된 매크로 함수이다.

[설명] 위의 단일문자 입력 함수는 입력받은 단일 문자를 리턴하며, getch와 getche는 0 - 255의 값을 getchar는 -1 - 255의 값을 리턴한다.

[참고] 단일문자를 입력받기 위해서는 getchar함수보다 getch와 getche 함수를 사용하는 것이 좋다.

ㅎI
getchar 함수의 문자 입력 방식
ㅎ
1)우선 행버퍼가 비어있는지 여부를 검사한다.

2)행버퍼에 문자가 하나라도 남아있으면 그 문자를 리턴하고 그 문자를 버퍼에서 삭제한다. 즉 키보드로부터 문자를 입력받지 않는다.
3)행버퍼가 비어있으면 키보드로부터 한 행을 입력 받아 행버퍼에 저장 한다.
4)위 사항은 내부사정이고 실제는 입력은 행 단위로 한꺼번에 하고 리턴은 한 문자씩 한다.

[예] getch, getche, getchar 사용 예.

```
#include <stdio.h>
main()
{
  int c;
  c = getch();      /* 입력 문자를 화면에 표시하지 않는다. */
  c = getche();     /* 입력 문자를 화면에 표시한다.        */
  c = getchar();    /* 입력 문자를 화면에 표시한다.        */
}
```

2. 단일문자 출력 함수

[선언] int putch(int c);
[설명] 문자를 화면에 출력한다.
[선언] int putchar(int c);
[설명] 표준 출력 스트림(stdout)상에 문자를 출력한다.
#define putchar(c) putc((c),stdout)
로 정의된 매크로 함수이다.

[설명] 위 함수는 성공에는 문자 c를 리턴하고 에러시에는 EOF를 리턴한다.
리턴한다. putchar은 매크로 함수이다.

[예] getchar와 putchar 사용

```
#include <stdio.h>
main()
{
 int ch;
 while((ch=getchar()) != EOF)   putchar(ch);
}
```

3. 문자열 입력 함수

[선언] char *gets(char *s);

[설명] 표준 입력 스트림(stdin)으로부터 1 문자열을 읽어들이고 문자형

`포인터 s`가 가리키는 곳에 저장한다. 개행문자(carriage return) 까지 읽어 들이고 저장될 때에 개행문자는 NULL문자로 변환한다. scanf와 다른 점은 `공백문자(space,tabs)`를 포함하도록 한다. 성공시에는 문자열 인수 s를 리턴하고 파일끝(EOF)나 에러시에는 NULL을 리턴한다.

[중요] 초기화되지 않은 포인터를 gets의 매개변수로 잘못 지정하면 컴파일러는 `Pissuble use of 'XXXXXXXX' before definition` 이라는 경고를 발생한다.

[참고 예]

```
char string[128]; 또는 char string[128],*ptr;
gets(string);          ptr=string; /* 반드시 이렇게 초기화해야 함 */
                        gets(ptr);
```

ᄚIsam0900.dat

[실습내용] gets와 puts를 이용한 문자열 입출력

```
#include <stdio.h>
void main(void)
{
  char string[128];

  do
  {
    gets(string);  /* 문자열을 입력 받는다. */
    puts(string);  /* 문자열을 출력한다.     */
  } while(*string);/* string[0] =='\0' && strlen(string) == 0) */
                    /* 문자열 입력 없이 리턴키를 치면 탈출한다. */
}
```

ᄚ

4. 문자열 출력 함수

[선언] int puts(const char *s);

[설명] 표준 출력 스트림(stdout)으로 1 문자열을 출력한다. 즉 NULL로 종료된 문자열 s를 표준 출력 스트림(stdout)으로 복사

하고 맨뒤에 개행문다('\0')를 추가한다.
성공적이면 음(-)이 아닌 값을 리턴하고 에러일 경우에는 EOF를
리턴한다.
문자열 출력후 자동으로 줄바꿈('\n')을 해 준다.

[참고] 문자열 입력 함수(gets) 실습예제를 참고 바란다.

ᄛI

문자열 출력 함수를 만들어 보자

ᄛ

```
#include <stdio.h>

int putstr(const char *s)
/* const는 문자열 s의 내용이 바뀔수 없는 변경자(modifier) */
{
  while(*s) putc(*s++);
  return *--s;            `/* 맨 마지막에 출력된 문자를 리턴한다. */`
}

int putstring(const char *s)
{
  while(*s) putchar(*s++);
  return putchar('\n'); `/* 맨 마지막에 줄을 바꾼다. */`
}
```

[참고] 위 예중 `putch와 putchar`를 사용한 점과 리턴값이 마지막 문자와
줄바꾼다는 점이 서로 다르다.

5. 서식화된 출력 함수

[선언] int printf(const char *format,...);
[방법] printf(서식문자열,인자1,인자2,...);

[설명] 포맷된 출력을 표준 출력 스트림(stdout)으로 출력한다.
즉 일련의 인수를 받아서 format에 의해 주어진 포맷 문자열의

형식에 따라 포맷된 데이타를 stdout호 출력한다.

이 함수는 출력한 문자의 갯수를 리턴하고 서식문자열(format string)에는 일반적으로 문자열 상수나 포인터 변수를 지정한다. 그리고 인자(argum-ent)들은 출력하고자 하는 문자나 숫자, 문자열 포인터,기타 등이 기술된다.

printf용 변환 문자

변환문자	의 미	자료형
%d	부호 있는 10진수로 출력	int or 문자형
%x(or %X)	16진수로 출력	unsigned 형
%u	부호 없는 10진수로 출력	unsigned 형
%e(or %E)	부동 소수점 표기범에 의해 출력	부동형
%f	고정 소수점 표기법에 의해 출력	부동형
%g(or %G)	부동 소수점 표기법에 의해 출력	부동형
%c	문자로 출력	문자 or 정수형
%s	널 종료문자를 만날 때까지 출력	문자열 포인터

[예] printf 사용 (^ : 공백)

```
printf("%d %o %x %u\n",10,10,10,10);
-> 10 12 A 10

printf("%e %f\n",15.5,15.3);
-> 1.530000e+001 15.30000
```

```
printf("%s\n","Turbo");
-> Turbo

printf("%-5s = %5d\n","Tur",18);
-> Tur^^ = ^^^18); /* -          : 왼쪽 정렬   */
                   /* + or 생략 : 오른쪽 정렬 */

printf("%.5s = %.3f\n","abcdefg",12.34567);
-> abcde = 12.345
```

6. 서식화된 입력 함수 - call by reference

[선언] int scanf(const char *format,...);
[방법] scanf(서식문자열,인자1,인자2,...);

[설명] 표준 입력 스트림(stdin)으로부터 입력을 검사하고 포맷한다.

즉 stdin 스트림에서 한번에 한 문자씩 읽고 검사해서 format이 가리키는 문자열에 들어있는 포맷 지정에 따라 내용을 바꿔서 format뒤의 인수로 주어진 주소에 그 내용을 집어 넣는다. 표시한 포맷수와 입력 필드 주소의 갯수는 동일해야 한다.

성공하면 입력받은 입력 필드의 갯수를 리턴하고, EOF일때는 -1을 리턴한다. 그러나 보통 리턴값을 무시한다.

------------------------ 계 속 -------------------------

[인자지정]

scanf(서식문자열,&변수명1,&변수명2,...);

여기에서 &는 번지 연산자로서 변수가 저장되어 있는 메모리의
번지를 알아 낸다.

[예] scanf 사용

```
int a,b,c;

scanf("%d %o %x",&a,&b,&c);
-> 입력 : 1234 4321 good
-> 반환 : 1234 04321 0xff
```

III. 고수준 파일 입출력 함수

ㅠ1. 파일의 개방과 종결

먼저 파일 입출력의 첫번째 관문인 fopen 함수부터 공부해 보도록 하자.

이 함수는 매우 중료한 위치를 차지하고 있다. 왜냐하면 fopen 함수를

이해하면 입출력을 전부 이해한 것과 다름없다고 할 수 있다.

여러번 반복하지만 모든 것을 한번에 전부를 얻으려고 하지 말고 처음에는

가볍게 읽은 다시 읽고 다시 읽고 하는 `반복 학습`이 더욱 더 좋은 학습

방법이며 좋은 효과를 기대할 수 있을 것이다.

[참고]
여기에서 언급되는 함수들은 모두 `stdio.h` 헤더 파일에 선언
되어 있다.

1.1 파일 개방 함수 - fopen

[선언] FILE *fopen(const char *filename,const char *mode);

[설명] 스트림을 개방한다.(파일의 오픈)

즉 filename으로 지정한 파일명을 개방한다. 이 함수는 성공하면
입출력 버퍼의 주소를 함수값으로 반환한다. 만약 에러 발생시에
는 NULL을 반환한다.

ꠠIsam0901.dat

[실습] fopen과 fclose를 이용하여 autoexec.bat 파일을 백업 받는다.

```
#include <stdio.h>
main()
{
  FILE *in, *out;
  if((in=fopen("\\autoexec.bat","rt")) == NULL)
  {
    printf("Cannot open input file.\n");
    return 1;
  }
  if((out=fopen("\\autoexec.bak","wt")) == NULL)
  {
    printf("Cannot open output file.\n");
    return 1;
  }
```

------------------------ 계 속 ------------------------

ꠠ

ꠠIsam0901.dat

```
    while(!feof(in)) fputc(fgetc(in),out);
    fclose(in);
    fclose(out);
}
```

ꠠ

ꠓ1.2 fclose - 파일 종결 함수

[선언] int fclose(FILE *stream);

[설명] 스트림을 닫는다.
즉 지명된 스트림을 닫는다. 일반적으로 스트림과 관련된 모든 버퍼는 닫기전에 깨끗하게 지워진다. 시스템에 할당된 버퍼는 파일을 닫음으서 해제된다. 그러나 setbuf와 setvbuf로 지정된 버퍼는 자동적으로 해제되니 않는다.
이 함수는 성공하면 NULL을, 실패하면 EOF(-1)을 반환한다.

ㅎl
[참고]
ㅎ
모든 스트림은 프로그램의 종료와 함께 자동적을 종결된다. 그러나 파일

의 모드에 따라 예측 할수 없는(정전 등등) 불의의 사고를 예방하기 위해

입출력이 완료된 즉시 fclose함수를 사용하여 열린 파일을 종결시키는 것

이 바람직 하다. 단, 기정의 스트림이나 도스 장치는 제외한다.

fopen함수가 동시 개방할 수 있는 스트림의 최대 갯수는 20개이다. 이 값

은 `stdio.h` 에 정의되어 있는 매크로상수 OPEN-MAX의 값이다.

그러나 `config.sys`파일에 미리 설정해 높은 값에 의해 그 갯수가

제한된다.

------------------------ 계 속 ----------------------

[참고 예]

c:\>type config.sys

FILES = 20
BUFFERS = 15

와 같이 되어 있다면 동시에 개방할 수 있는 스트림의 갯수는 20개로

설정된다. 그 이유는 프로그램 실행과 동시에 기정의 스트림(stdiu,

stdout, stderr, stdaux, stdprn) 5개가 자동으로 개방되기 때문이다.

ㅎl

```
[실습] fopen과 fclose를 이용한 파일 복사 프로그램
#include <stdio.h>
void main(int argc, char *argv[])
{
  int c;  /* 이진 스트림에서는 반드시 int 형 이어야 한다.
             2.3 문자 입출력 모드 참조                   */
  FILE *source, *destin;
  if(argc <= 2)
  {
    puts("Usage : filecopy source destin");
    exit(1);
  }
  source = fopen (argv[1], "rb");    /* 읽기 전용 이진모드 */
  if(source == NULL)
  {
    perror("Source");
    exit(1);
  } -------------------------- 계  속 ----------------------
```

ㅎ

ㅎl

```
  destin = fopen (argv[2], "wb");
  if(destin == NULL)
  {
    perror("Destin");
     exit(1);
  }
  while((c=fgetc(source)) != EOF)
  if(fputc(c, destin) == EOF) break; /* 파일의 끝이거나 에러가
                                  발생하면 루프를 벗어난다. */
  if(ferror(source) || ferror(destin))
  {
    perror("Copying");
```

```
            exit(1);
        }

 ------------------------- 계  속 ----------------------
ㆆ



ㆆI
        else
        if (fclose(destin) == EOF)
        {
            perror("Destin");
            exit(1);
        }
        else
        {
            printf("%s copied to %s\n", argv[1],argv[2]);
            exit(0);
        }
    }
ㆆ
```

`[사용법]`

```
c:\> filecopy filecopy.c fcopy.c
     filecopy.c copied to fcopy.c    /* 성공하면 */
```

2. 버퍼형 파일(BF)의 입출력 함수

이 함수들은 모두 "stdio.h"표준 헤더파일에 선언되어 있고 파일끝(EOF) 이거나 또는 에러발생시에는 EOF(end-of-file)를 리턴한다.

함수 선언

```
+-------------------------------------------------+
| int fgetc(FILE *stream);                        |
| int fputc(int c, FILE *stream);                 |
| char *fgets(char *s, int n, FILE *stream);      |
| int fputs(const char *s, FILE *stream);         |
| int fscanf(FILE *stream, const char *foramt, ...);  |
| int fprintf(FILE *stream, const char *format, ...); |
+-------------------------------------------------+
```

2.1 단일 문자 입력 함수

[선언] int fgetc (FILE * stream);
[설명] 스트림으로부터 문자를 가져온다.(파일로부터 1문자를 읽어
들인다. 즉 스트림(stream)에서 한 문자를 읽어내고 FP(file
pointer)를 증가시킨다.

[참고] 이진 스트림에서는 EOF와 '\xff' 문자를 서로 구별하기 위하여 리턴
받는 변수는 int 형이어야 한다. (단 텍스트는 제외)

/* 문자 입출력 모드 [설명]부분을 참고 */

[예]
```
main()
{
  int c;          /* 이진 스트림에서는  반드시 int 형 이어야 한다.
                     unsigned char 형도 절대 않된다.              */
  FILE *stream;

  stream = fopen("data.in","rb");  /* 읽기 전용 이진 모드 */
  c = fgetc(stream);
  if(c == EOF && ferror(stream))
  {
    fputs("File reading error !\n", stderr);
    exit(1);
  }
```

```
  fclose(stream);
}
```

2.2 단일 문자 출력 함수

[선언] int fputc (intc, FILE *stream);
[설명] 스트림에 한개의 문자를 써 넣는다. 즉 문자 c 를 스트림(stream) 에 출력한다.
성공하면 문자 c 를 리턴하고 에러시에는 EOF를 리턴한다.
[예]

```
main()
{
  char c;    /* 입력이 아니므로 반드시 int 형일 필요는 없다.
                unsigned char 형도 무방하다.                  */
  FILE *stream;
  stream = fopen("data.out","wb");  /* 쓰기 전용 이진 모드 */
  if(fputc(c,stream) == EOF)
  {
    fputs("Disk full !\n",stderr);
    exit(1);
  }
  fclose(stream);
}
```

2.3 문자열 파일 입력 함수

[선언] char *fgets (char *s, int n, FILE *stream);
[설명] 스트림으로부터 문자열을 가져온다. 즉 파일로부터 행을 읽어들여 s가 가리키는 문자배열에 저장한다.
[참고] gets와 다른점은 '\n'문자를 제거하지 않고 그대로 읽어 들인다.
[예]

```
main()
{
  char name[12];
  FILE *stream;
  stream = fopen("data.in","rb");  /* 읽기 전용 이진 모드 */
  if(fgets(name,12,stream) == NULL && ferror(stream))
  {
    fputs("File reading error !\n", stderr);
    exit(1);
```

```
    }
    fclose(stream);
}
```

[설명] 위에서는 널(\0)을 포함하여 최대 11자를 읽어 들인다.

2.4 문자열 파일 출력 함수

[선언] int fputs (const char *s, FILE * stream);

[설명] 스트림에 한개의 문자열을 출력한다.
즉 NULL로 종료된 문자열 s를 주어진 출력 스트림(stream)에 복사해 온다. 이 함수는 개행문자(carriage return)를 추가하지 않으며 NULL 문자 자신도 복사되지 않는다.

따라서 한행을 출력하면 문자열 s의 맨끝이 개행문자 '\n'이어야 한다.(이진모드에서는 "\r\n")또한 널 문자로 스트림에 출력되지 않는다.

[예] 텍스트모드 - fputs (" Normal text line\n", stream);
이진모드 - fputs(" Normal binary line\r\n",stream);

2.5 서식화된 파일 입력함수

[선언] int fscanf (FILE *stream, const char *format, ...);

[설명] 서식지정의 갯수와 fscanf의 인자의 갯수가 같아야 하며, 각 인자는 포인터 수식어이어야 하고, 단순 변수일 경우에는 반드시 번지 연산자(&)를 붙여야 한다. 그러므로 fscanf 함수를 이용하여 한 행의 데이타를 읽어들일때에는 서식 문자열 맨 끝에 개행문자 '\n'을 붙이는 것이 좋다.

[예]

```
int a,b;
fscanf(stream, "%d %lf",&a,&b);
```

2.6 서식화된 파일 출력함수

[선언] int fprintf(FILE *stream,const char *format,....);

[설명] 포맷된 출력을 스트림에 보낸다.
즉 일련의 인수를 받아서 format이 가리키는 스트림 포맷지정에
적용하고, 포맷된 데이타를 스트림에 출력한다.
이 함수는 출력된 바이트를 리턴하고 에러시에는 EOF를 리턴한다.

3. 파일끝과 입출력 에러의 판별

FILE 구조체의 멤버인 flags도 unsigned형으로 해당 스트림에 대한 10
가지나 되는 각종 정보를 비트단위로 보유하고 있다.

`매크로 상수로 정의된 각 정보와 그 의미(stdio.h)`
- 해당 비트가 on 상태일때 그 정해진 의미를 갖는다.

```
+--------------------------------------------------------------+
| #define _F_RDWR 0x0003 /* Bit 0 - 1 : Update mode            */|
| #define _F_READ 0x0001 /* Bit 0     : Read-only mode         */|
| #define _F_WRIT 0x0002 /* Bit 1     : Write-only mode        */|
| #define _F_BUF  0x0004 /* Bit 2     : Malloc'ed buffer data  */|
| #define _F_LBUF 0x0008 /* Bit 3     : Line buffered stream   */|
| #define _F_ERR  0x0010 /* Bit 4     : Error indicator        */|
| #define _F_EOF  0x0020 /* Bit 5     : End-of-file indicator  */|
| #define _F_BIN  0x0040 /* Bit 6     : Binary stream indicator */|
| #define _F_IN   0x0080 /* Bit 7     : Data ins incoming      */|
| #define _F_OUT  0x0100 /* Bit 8     : Data is outgoing       */|
```

| #define _F_TERM 0x0200 /* Bit 9   : DOS device indicator   */|
+--+
ᄒI
FILE 구조체 멤버 flags의 비트 구조
ᄒ

illu0901.dat
비트4와 비트5는 매우 중요한 정보를 표시한다. 비트4는 에러 지시자로서 파일 입출력시 어떤 에러가 발생하면 on으로 설정 된다. 비트 5는 파일 끝지시자로서 입력시 파일끝이 검출되면 on 으로 설정된다.

3.1 에러 판별 매크로 함수

[선언] int ferror(FILE *stream);
[설명] 스트림상에서 에러를 탐지한다.(파일의 에러를 조사한다.)
즉 읽기 또는 쓰기 에러상태를 알기 위해 주어진 스트림을 조사하는 매크로이다. 스트림의 에러 지시자가 설정되면 clearerr 또는 rewind가 호출될때까지 아니면 스트림이 닫힐때까지 해제되지 않는다.
[선언] int feof(FILE *stream);
[설명] 스트림상에서 파일의 종료(EOF)를 탐지한다.
즉 EOF 지시자를 위해 주어진 스트림을 조사하는 매크로이다.
일단 지시자가 설정되면 rewind가 호출되거나 파일이 닫힐 때까지 읽기 작동은 이 지시자를 리턴하게 된다.

[정의] #define ferror(f) ((f)->flags & _F_ERR)
#define feof(f) ((f)->flags & _F_EOF)
[설명] 해당 비트가 on일때 0 이 이외의 값, off이면 0 을 갖는다.

[예]

```
int c;
FILE *stream;

c = fgetc(stream);
if(c == EOF && ferror(stream))
    fputs("File reading error !\n",stderr);
```

4. 블럭 입출력 함수(stdio.h에 정의)

fread와 fwrite 함수는 `이진 스트림` 전용으로서 일반 변수(산술형)나
구조체 변수 또는 배열 등과 같이 문자나 문자열이 아닌 `이진 데이타`를
스트림에 출력할 때 쓰인다.

ⅵ4.1 블럭 입력 함수 - fread

[선언] size_t fread(void *ptr,size_t size,size_t n,FILE *stream);

[설명] 데이타를 스트림으로부터 읽어 들인다. 즉 주어진 입력 스트림
(stream)으로부터 각각 size바이트의 길이로 데이타를 n개 읽어
서 그 데이타 포인터 ptr가 지정하는 블럭에 가져온다.

총 바이트 수는 n * size 가 되고 성공하면 실제로 읽혀진 실제
항목수(바이트 아님)를 리턴하며, 에러나 파일끝(EOF)일때는 0
이 리턴된다.

4.2 블럭 출력 함수 - fwrite

[선언] size_t fwrite(void *ptr,size_t size,size_t n,FILE *steam);

[설명] 데이타를 스트림에 써 넣는다. 즉 포인터 ptr가 가리키는 블럭을
스트림의 FP에 출력한다. 이때 블럭은 일반 변수,구조체,배열 등
이 될 수 있다.
총 바이트 수는 n * size 가 되고 성공하면 실제로 읽혀진 실제
항목수(바이트 아님)를 리턴하며, 에러시에는 0 이 리턴된다.

[참고] 입출력이 끝나면 FP는 블록의 크기만큼 증가하여 다음 블럭을
순차적으로 입출력할 수 있도록 한다.

ㆆl

[예] 배열에 대한 사용방법

ㆆ

```
A arrary[NUM];
fread(array sizeof(A), NUM, stream);
fwrite(array sizeof(A), NUM, stream);
```

[설명]

A : 배열요소형
NUM : 배열의 크기(또는 데이타항목의 깃수)
sizeof(A) : 데이타 항목의 크기 (요소령 A의 크기)
arrary : 배열의 선두번지로 가리키는 포인터.
NUM *sizeof(A) == sizeof(array)는 같음

ㆆl

[예] 일반 변수나 구조체에 대한 사용방법
ㆅ
A veriable ;
fread (&variable, sizeof(A), 1, stream);
fwrite (&variable, sizeof(A), 1, stream);

[설명]

A   : 데이타 (산술형, 구조체형 등등)
&variable : 변수를 가리키는 포인터 수식
sizeof(A) : 데이타형 A의 바이트 크기
1   : 데이타 항목의 갯수

5. 정수 입출력 함수

getw와 patw함수로 스트림에 2바이트 크기의 정수(-32768 ~ + 32767 또는 0 ~ 65535)를 입출력한다. 이 함수는 이진 모드에서만 쓰는 것이 정상이다.

ㅍ]5.1 정수 입력 함수 - getw

[선언] int getw (FILE *stream);
[설명] 스트림으로부터 정수를 읽어온다.
텍스트 모드에서 스트림이 개방 되어 있을 때는 사용할 수 없다.
파일끝(EOF)이나 에러가 발생하면 EOF를 리턴한다.

ㅍ]5.2 정수 출력 함수 - putw

[선언] int putw (int w, FILE *stream);
[설명] 스트림상에 정수를 출력한다. 즉 주어진 스트림에 정수 w를 출력한다. 성공하면 정수 w 를 리턴하고 에러시에는 EOF를 리턴한다.

[참고]

리턴값만 가지고는 정수 -1과 EOF를 구별할 방법이 전혀 없으므로 반드시 `feof와 ferror`함수를 아래 [예]와 같이 병용해야 한다.

[예] feof와 ferror 함수 사용

```
int w

if((w = getw(stream) == EOF && feof (stream))    ->  파일 끝검출
if((w = getw(stream) == EOF && ferror(stream))   ->  에러 발생
if(putw(w,stream) == EOF && ferror(stream))      ->  에러 발생
```

[예] getw 함수 사용

```
#include <stdio.h>
main()
{
  FILE *stream;
  int  a;
  if((stream=fopen("data.in","r")) == NULL)
  {
    printf("Can't open file\n");
    exit(1);
  }
  while((a=getw(stream)) != EOF)
  printf("%d",a);
}
```

[예] putw 함수 사용

```
#include <stdio.h>
main()
{
  FILE *stream;
  int   a;
  if((stream=fopen("data.out","wb")) == NULL)
  {
    printf("Can't open file\n");
    exit(1);
  }
  while(scanf("%d",&a) != EOF)
  putw(a,stream);
}
```

6. 파일 임의 접근 함수(fseek, rewind)

임의 접근 파일(random access file)이라고 해서 순차 파일(seguential file)과 특별하게 다른 것은 아니다. 다만 파일을 순차적으로 읽고 쓰는 대신 FP(file pointer)를 이리저리 옮겨가면서 입출력을 할 뿐이다.

ㅎJ6.1 fseek 함수

[선언] int fseek (FILE *stream, long offset, int whence);
[설명] 스트림상에 있는 파일 포인터 위치를 변경시킨다.
(파일내의 입출력 위치를 변경한다.)

즉 스트림과 관련된 파일 포인터에 whence에 의해 주어진 파일 위치로 부터 offset 바이트까지의 새로운 위치를 설정한다.
텍스트 모드 스트림에 대해서 offset은 0 이 되거나 ftell에 의해 리턴되는 값이어야 한다.
성공하면 0 을 리턴하고 실패하면 0 이외의 값을 리턴한다.

------------------------ 계 속 -------------------

`기준점 위치`

```
+---------------------------------------------------------+
|    whence     |          파일위치                 | offset 범위 |
```

```
|-------------|----------------------------|-------------|
| SEEK_SET(0) | 파일 선두(file beginning)  |    0L 이상    |
| SEEK_CUR(1) | 현재 파일 포인터 FP의 위치 |    임의       |
| SEEK_END(2) | 파일끝 검출위치 (P_EOF+1)  |    0L 이하    |
+------------------------------------------------------+
```

[설명]

1) 텍스트 모드에서는 반드시 offset 이 0L이어야 한다.
2) offset은 long형이어야 한다. 그러므로 접미사 L을 붙인다.
3) P_EOF는 물리적 파일의 끝이다.
4) SEEK -SET 에서는 offset이 항상 OL 또는 양수이어야 한다.

[예]

```
fseek(stream, 0L,  SEEK_SET); -> FP는 파일선두로 이동한다.
fseek(stream, +0L, SEEK_CUR); -> FP는 옮겨지지 않는다.
fseek(stream, 0L,  SEEK_END); -> FP를 P_EOF+1로 옮긴다.
fseek(stream, +1L, SEEK_CUR); -> FP를 1만큼 증가시키다.
fseek(stream, -1L, SEEK_CUR); -> FP를 1만큼 감소시킨다.
fseek(stream, -1L, SEEK_END); -> FP를 P-EOF에 위치시킨다.
```

ᇹlsam0903.dat

[실습내용] fseek를 이용하여 파일크기를 구한다.

```
#include <stdio.h>
long filesize(FILE *stream)
{
  long curpos, length;
```

--------------------- 계 속 -------------------

ᇹ

ᇹlsam0903.dat

```
  curpos = ftell(stream);       /* 현재 파일 포인터를 구한다.     */
  fseek(stream,0L,SEEK_END);   /* 파일 포인터를 파일끝으로 이동 */
  length = ftell(stream);       /* 현재 파일 포인터를 구한다.     */
  fseek(stream,curpos,SEEK_SET); /* 파일 포인터를 처음으로 이동*/
```

```
    return length;              /* 파일 포인터를 리턴( long 형 ) */
  }

  main()
  {
    long in;
    FILE *stream;
    stream=fopen("\\COMMAND.COM","rb");
    in = filesize(stream);
    printf("Filesize of COMMAND.COM is %u Byte\n",in);
  }
```
ㅎ

[결과] Filesize of AUTOEXEC.BAT is 00000 Byte

6.2 rewind

[선언] void rewind(FILE *stream);
[설명] `처음으로 되감는다`라는 뜻으로 fseek(stream,0L,SEEK_SET)와
같은 효과를 가져온다. 즉 스트림의 선두에 파일 포인터(FP)를
위치시킨다.
[예] fseek에 의한 FP의 위치(파일 크기는 10바이트라고 가정)

[그림]

[참고]
갱신모드에서 fseek나 rewind를 호출한 직후의 다음 작업은 입출력 모두
가능하다. 그러나 일단 한번 입력이 실시되면, 다시 fseek나 rewind를
호출하기 전까지는 계속 순차적인 입출력만 가능하다. fseek는 성공하면
0 을 실패하면 -1 을 리턴한다.
7. 파일 포인터 위치 파악 함수 - ftell

[선언] long ftell (FILE *stream);

[설명] 현재의 파일 포인터르 리턴한다.
즉 스트림(stream)에 대한 현재의 파일 포인터(FP)를리턴된다.

파일의 초기부터 현재의 오프셋을 바이트 단위로 측정하여 리턴
한다.

이 함수에 의해 리턴되는 값은 fseek를 호출하는데 사용한다.
성공하면 현재의 파일 포인터 위치를 리턴하고 에러시에는 -1L을
리턴한다.

ㅎIsam0904.dat

[실습내용] ftell 함수 사용

```
#include <stdio.h>
main()
{
  int in;
  FILE *stream;

  stream = fopen("\\AUTOEXEC.BAT","r");
  fseek(stream,0L,SEEK_END);
  in = ftell(stream);        /* 현재의 파일 포인터를 구한다. */
  printf("Filesize of AUTOEXEC.BAT is %1d byte\n",in);
  fclose(stream);
}
```

ㅎ

[결과] Filesize of AUTOEXEC.BAT is 00000 byte

7. 입출력 버퍼 사용자 설정함수 - setbuf

일반적으로 버퍼형 파일의 버퍼도 시스템이 동적으로 할당하나 betbuf
함수를 사용하면 입출력 버퍼를 사용자가 설정할 수 있다.

[선언] void *setbuf (FILE *stream,char *buf);

[설명] 스트림에 대해 버퍼를 지정한다.
즉 i/o 를 위해 자동적으로 할당된 버퍼 대신에 buf로 주어진 버퍼를 사용하도록 한다.
buf는 사용자가 설정하는 버퍼이다. 버퍼의 크기는 stdio.h에 정의되어 있는 BUFSIZE 이상 확보되어야 한다.

9. 강제 출력 함수 - fflush

[선언] int fflush (FILE *stream);

[설명] 파일용 출력 버퍼를 소거한다.
즉 파일 stream의 출력 버퍼자료를 강제로 파일 stream에 기록하는 함수이다. 적시에 버퍼의 내용을 강제적으로 옮기므로 안정성이 보장된다는 관점에서 문자장치의 스트림 입출력이 버퍼링되는 시스템에 많이 사용된다.

서론

여러분이 C-언어를 배워 직접 프로그램을 작성하려고 한다면 그래픽은 필수 과정 이라고 할 수 있다. 왜냐하면 요즘의 프로그램 추세를 보면 그래픽이 없는 프로그램은 별로 대접을 받지 못하고 있기 때문이다.

모양이 이쁜 음식이 먹기도 좋다는 말은 음식에만 해당되는 말은 아니다.
컴퓨터 프로그램도 조금이라도 멋있어 보이고 세련되어 보이면 그만큼 값어치가 나가게 된다.

illu1000.dat

1. 터보-C와 Borland C++ 의 그래픽

터보-C 계열의 컴파일러는 모두 동일한 그래픽 라이브러리를 가지고 있다. 바로`볼랜드 그래픽 인터페이스(BGI: Borland Graphics Interface)` 라는 것인데 다른 언어끼리도 사용법이 비슷하여 한가지만 배워두면 터보 계열의 컴파일러 그래픽은 모두 아는것이나 다를바가 없다.

그래픽을 사용하기 위해서는 아래의 4개의 파일이 필요하다.

```
 graphics.lib            ..... 그래픽 라이브러리
 graphics.h               ..... 그래픽 헤더
 *.bgi                   ..... 그래픽 카드 드라이버
 *.chr                    ..... 폰트 파일
```

`graphics.lib`

위의 파일은 그래픽 라이브러리로 실행파일을 만들기 위해 링크작업을 할때 같이 링크해 주어야 한다. (물론 그래픽을 사용하는 프로그램만)

`graphics.h`

위의 파일은 graphics.lib에 수록된 그래픽 관련 함수들의 프로토타입과 변수,상수 들이 선언되어 있는 파일이다. 다른 헤더파일 (예 : stdio.h stdlib.h) 와 마찬가지로 그래픽을 사용하는 프로그램에서는 반드시 사용 되어야 한다.

`*.bgi`

위의 파일은 각기 다른 그래픽카드를 위한 파일들이다. 예를 들어 VGA카드를 사용하는 사용자는 VGAEGA.BGI 가 필요하여 허큘리스 카드를 사용하는 사용자는 HERC.BGI가 필요하다.

아래의 표는 각 그래픽 카드별 드라이버 파일이다.

ठा

그래픽 카드(어댑터)	드라이버(BGI 파일)
CGA, MCGA	CGA.BGI
EGA, VGA	EGAVGA.BGI
Hercules	HERC.BGI

ठ

이외에서 AT&T, 3270PC, IBM-8514등이 있으나 저자는 구경도 해본적이 없으므로 생략한다.

`*.chr`

위의 파일은 BGI에서 쓰이는 스트로크 폰트 데이타 파일이다. 즉, 글자 모양을 담은 파일인데 영문만을 사용할
수 있어 아쉽지만, 그런데로 쓸만하다.

폰트의 종류는 옆의 그림에서 보여주고
있으며 이들 폰트를 사용하여 글자를
쓰는 방법은 차차 배워보기로 한다.

ヵillu1001.dat

2. 그래픽 시스템의 시작과 종료

우리가 그래픽을 사용하려면 우선 그래픽 모드로 들어가야 한다. 그리고 모든 그래픽 작업을 마치면 다시 텍스트 모드로 돌아가야 한다. 다음의 두 함수는 그래픽모드를 설정하고 텍스트 모드로 돌아가는 역할을 한다.

```
void initgraph(int   far *graphdriver,
               int   far *graphmode,
               char far *pathtodriver);
```

[기능] 그래픽 시스템을 초기화하고, 하드웨어에 따라 그래픽 모드를 바꾼다.

```
    void closegraph(void);
```

[기능] 그래픽을 끝내고 화면을 텍스트 상태로 돌려주며, 그래픽을 위하여 할당된 메모리를 풀어준다.

프로그램이 실행될 때 initgraph()함수는 그래픽 카드의 종류를 알아내 해당 그래픽 드라이버(BGI)를 로드 시켜서 화면을 그래픽 시스템으로 전환한다. 이와는 반대로 closegraph()를 실행하면 화면은 다시 텍스트 화면으로 돌아간다.

initgraph()함수 사용시 graphdriver와 graphmode는 둘다 int형으로 드라이버를 지정하는`graphdriver`는 graphics.h 파일에 상수로 정의되어 있다.

```
    enum graphics_drivers {
       DETECT,
       CGA, MCGA, EGA, EGA64, EGAMONO, IBM8514,    /* 1 - 6 */
       HERCMONO, ATT400, VGA, PC3270,              /* 7 - 10 */
       CURRENT_DRIVER = -1
    };
```
ㄷ

위의 그래픽카드 드라이버중 원하는 카드를 선택하여도 되지만 되도록이
면 DETECT를 선택하는 것이 좋다. DETECT는 initgraph호출시에 자동으로
그래픽 드라이버를 찾아주며 가장 좋은 해상도로 맞추어 준다.

graphdriver에 DETECT를 선택하면 graphmode는 선택할 필요없이 알아서
선택해 준다.

각 그래픽카드에 따라 지원할 수 있는 graphmode값에는 차이가 있으며
각 모드마다 해상도와 칼라수 에서 차이가 난다.

ㄷl
```
enum graphics_modes {
    CGAC0       = 0,  /* 320x200 palette 0; 1 page              */
    CGAC1       = 1,  /* 320x200 palette 1; 1 page              */
    CGAC2       = 2,  /* 320x200 palette 2: 1 page              */
    CGAC3       = 3,  /* 320x200 palette 3; 1 page              */
    CGAHI       = 4,  /* 640x200 1 page                         */
    MCGAC0      = 0,  /* 320x200 palette 0; 1 page              */
    MCGAC1      = 1,  /* 320x200 palette 1; 1 page              */
    MCGAC2      = 2,  /* 320x200 palette 2; 1 page              */
    MCGAC3      = 3,  /* 320x200 palette 3; 1 page              */
    MCGAMED     = 4,  /* 640x200 1 page                         */
    MCGAHI      = 5,  /* 640x480 1 page                         */
    EGALO       = 0,  /* 640x200 16 color 4 pages               */
```
ㄷ

ㄷl
```
    EGAHI       = 1,  /* 640x350 16 color 2 pages               */
```

```
    EGA64LO     = 0,  /* 640x200 16 color 1 page               */
    EGA64HI     = 1,  /* 640x350 4 color  1 page               */
    EGAMONOHI   = 0,
    HERCMONOHI = 0,  /* 720x348 2 pages                         */
    VGALO       = 0,  /* 640x200 16 color 4 pages              */
    VGAMED      = 1,  /* 640x350 16 color 2 pages              */
    VGAHI       = 2,  /* 640x480 16 color 1 page               */
    PC3270HI    = 0,  /* 720x350 1 page                        */
    IBM8514LO   = 0,  /* 640x480 256 colors                    */
    IBM8514HI   = 1   /*1024x768 256 colors                    */
};
```
ㄷ

그래픽 모드로 들어간후 graphresult()함수를 호출해 보면 그래픽으로의 전환이 성공적으로 이루어 졌는지 여부를 알 수 있다. 이때 graphresult 에서 결과에 따라 돌려주는 값은 아래와 같다.

ㄷI

```
enum graphics_errors {
  grOk              =   0,        그래픽 성공
  grNoInitGraph     =  -1,        initgraph 함수를 호출하지 않았음
  grNotDetected     =  -2,        그래픽카드 없음
  grFileNotFound    =  -3,        파일이 없음
  grInvalidDriver   =  -4,        드라이버 파일이 잘못됨
  grNoLoadMem       =  -5,        메모리 부족
  grNoScanMem       =  -6,        메모리 부족
  grNoFloodMem      =  -7,        메모리 부족
  grFontNotFound    =  -8,        폰트파일(*.chr) 이 없음
  grNoFontMem       =  -9,        폰트파일을 읽을 메모리 없음
  grInvalidMode     = -10,        모드가 잘못됨
```
ㄷ

ㄷI

```
  grError           = -11,        보통에러
  grIOerror         = -12,        입,출력 에러
```

```
  grInvalidFont        = -13,          폰트가 잘못됨
  grInvalidFontNum     = -14,          폰트번호가 잘못됨
  grInvalidVersion     = -18           버전이 잘못됨
};
```
ᄐ

ᄐIsam1000.dat

[예제] 그래픽 초기화와 종료

```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;

main()
```
.... 계 속 ....

ᄐ

ᄐIsam1000.dat

```
{
  initgraph(&GD,&GM,"");              .... 그래픽 초기화
  line(0,0,getmaxx(),getmaxy());      .... 선을 그린다.
  getch();                            .... 키입력을 기다린다.
  closegraph();                       .... 그래픽을 종료한다.
}
```
ᄐ

3. 그래픽 에러처리

ᄐI

```
int   graphresult(void);
char *grapherrormsg(int errorcode);
```

ㅎ

graphresult()함수는 맨 마지막 그래픽 조작의 상태에 따라 에러코드를 알려준다.

다음의 루틴들은 graphresult()의 결과값에 영향을 준다.

ㅎI

bar	imagesize	setfillpattern
bar3d	initgraph	setfillstyle
clearviewport	installuserdriver	setgraphbufsize
closegraph	installuserfont	setgraphmode
detectgraph	pieslice	setlinestyle

ㅎ

ㅎI

drawpoly	registerbgidriver	setpalette
fillpoly	registerbgifont	settextjustify
floodfill	setallpalette	settextstyle
getgraphmode		

ㅎ

graphresult()는 호출되어진 후에 0이 된다. 그러므로 사용자는 graphresult의 값을 임시로 변수에 저장한 다음 이 값을 테스트해야 한다. 그래픽 에러를 관리하는 함수로는 graphresult() 뿐만 아니라 grapherrmsg() 함수도 존재한다.

ㅎI

```
int    graphresult(void);
```

ㅎ

마지막 그래픽 동작의 에러 코드를 얻는다.

```
    char *grapherrormsg(int errorcode);
```

에러코드(Error Code)와 상관있는 메시지를 알려준다.

graphresult()함수의 결과와 해당 에러메시지는 아래와 같다.

```
-------------------------------------------------------------------
코드   상수              에러메시지
-------------------------------------------------------------------
```

```
  0 grOk                 No error
 -1 grNoInitGraph        (BGI)graphics not installed (use initgraph)
 -2 grNotDetected        Graphics hardwere not detected
 -3 grFileNotFound       Device driver file not found
 -4 grInvalidDriver      Invalid device driver file
 -5 grNoLoadMem          Not enough memory to load driver
 -6 grNoScanMem          Out of memory scan fill
 -7 grNoFloodMem         Out of memory flood fill
 -8 grFontNotFound       Font file not found
 -9 grNoFontMem          Not enough memory to load font
-10 grInvalidMode        Invalid graphics mode for selected driver
-11 grError              Graphics error
-12 grIOerror            Graphics I/O error
-13 grInvalidFont        Invalid font file
-14 grInvalidFontNum     Invalid font number
```

4. 뷰포트(Viewport)

```
    void setviewport(int left, int top, int right, int bottom,
                     int clip);
    void getviewsettings(struct viewporttype far *viewport);
    void clearviewport(void);
```

ㆆ

뷰포트를 처리하는 함수는 위의 세가지가 제공된다. 뷰포트는 텍스트 모드에서 텍스트 윈도우라고 불리는것과 아주 유사한다. 텍스트 모드에서 윈도우를 정의하기 위해 window()함수를 사용하는데 반해 그래픽 모드에서 뷰포트를 정의하는데에는 setviewport()함수가 이용된다.

ㆆI

```
void setviewport(int left, int top, int right, int bottom,
                 int clip);
```

ㆆ

그래픽 출력을 위하여 현재의 출력 뷰포트와 윈도우를 맞추어 준다.

ㆆI

```
void getviewsettings(struct viewporttype far *viewport);
```

ㆆ

현재의 뷰포트와 클리핑(Clipping) 여부를 알려준다.

ㆆI

```
void clearviewport(void);
```

ㆆ

현재 설정된 뷰포트 내부를 지운다.

setviewport()함수는 그래픽 출력을 나타내기 위한 새로운 뷰포트를 정의한다.

ㆆI

```
void setviewport(x1 , y1, x2 , y2 , 1);
```

ㆆ

위와 같이 사용하면 그래픽 화면 상에서 절대화면 좌표계로 (x1 , y1)을 좌상단으로, (x2 , y2)를 우하단으로 하는 직사각형 영역이 현재 뷰포트로 정의된다. 윈도우와 마찬가지로 이것을 정의한 후에는 그래픽 루틴의 모든 좌표값이 뷰포트에서 상대 좌표계를 기준으로 삼게된다.

그러므로 현재 설정된 뷰포트의 원점은 절대좌표로 (x1,y1), 상대좌표로는 (0,0)이 된다.

setviewport()함수의 마지막 인자인 clip는 새로 설정한 뷰포트에서 클리핑(Clipping)을 할것인지의 여부를 결정한다.

클리핑 이라는 것은 그래픽 출력이 현재 뷰포트의 경계 밖으로 나가려고 할 때 출력을 금지 시키는것을 말한다.
따라서 인자 clip 에 0 이 아닌값(참)을 넣으면 뷰포트경계를 넘는 모든 출력은 잘리게 된다.

illu1002.dat

getviewsetting()함수는 현재의 뷰포트를 돌려주는 역할을 하는데 인자로 사용되는 viewporttype은 다음과 같이 정의되어 있다.

```
struct viewporttype {
                int left, top, right, bottom;
                int clip;
};
```

ㅎ

ㅎIsam1001.dat

```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
{
```

.... 계 속 ....

ㅎ

ㅎIsam1001.dat

```
    initgraph(&GD,&GM,"");                    .... 그래픽 초기화
    setviewport(100,100,540,380,1);       .... 뷰포트를 정의한다.

    rectangle(0,0,getmaxx(),getmaxy());  .... 테두리를 그린다.
    line(0,0,getmaxx(),getmaxy());         .... 선을 그린다.
    getch();                                          .... 키입력을 기다린다.
    closegraph();                                   .... 그래픽을 종료한다.
}
```

ㅎ

5. 점관련 루틴

ㅎI

```
unsigned getpixel(int x, int y);
void       putpixel(int x, int y, int color);
```

ㅎ

ㅎI

```
void       putpixel(int x, int y, int color);
```

ㅎ

(x,y) 에 color의 색으로 점을 찍는다.

ꡔ
```
unsigned getpixel(int x, int y);
```
ꡕ
(x,y) 위치의 점의 색상을 알려준다.

ꡔsam1002.dat
```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
{
  int i,j,k;

  initgraph(&GD,&GM,"");                    .... 그래픽 초기화
  for (i=0;i<100;i++)
    putpixel(getmaxx(),getmaxy(),getmaxcolor());
  getch();                                  .... 키입력을 기다린다.
  closegraph();                             .... 그래픽을 종료한다.
}
```
ꡕ

6. 선과 CP, 색상 처리

ꡔ
```
void line(int x1, int y1, int x2, int y2);
void lineto(int x, int y);
void linerel(int dx, int dy);
void moveto(int x, int y);
void moverel(int dx, int dy);
int  getx(void);
int  gety(void);
void setbkcolor(int color);
```

```
     void setcolor(int color);
     int  getbkcolor(void);
     int  getcolor(void);
     void setlinestyle(int linestyle, unsigned upattern,
                       int thickness);
```
ᄒ

ᄒl

```
     void getlinesettings(struct linesettingstype far *lineinfo);
     void setwritemode( int mode );
     int  getmaxx(void);
     int  getmaxy(void);
```
ᄒ

먼저 선그리기에 대해서 설명하겠다. 터보-C 에서 사용할 수 있는 선을 그리는 함수는 세가지 이다.

ᄒl

```
     void line(int x1, int y1, int x2, int y2);
```
ᄒ

(x1,y1) 부터 (x2,y2) 까지 선을 긋는다.

ᄒl

```
     void lineto(int x, int y);
```
ᄒ

현재 위치(CP)에서 (x,y)까지 선을 긋는다.

ᄒl

```
     void linerel(int dx, int dy);
```
ᄒ

현재 위치(CP)에서 상대적 거리의 점 (x+dx, y+dy)까지 선을 긋는다.

위의 세가지 함수중 line()함수는 이미 주어진 인자들로 이루어진 두점

을 연결하는 선을 긋는 것이므로 다른 함수에 비해서 이해하기가 비교적 수월할 것이다.

그러나 함수 linerel()과 lineto()는 먼저`현재위치(CP)`라는 개념을 이해하여야 한다. 여기서 현재위치란 가장 최근에 그려진 전과 같은 의미로 쓰인다.

텍스트 모드에서는 `커서`라는 개념이 있는데 가령 printf("Turbo-C"); 라고 하면 커서는 "C"다음에 오게된다. 이와 비슷한 개념이 CP인데 화면의 첫째 줄 왼쪽 구석이 (0,0)이 된다.

그리고 우측으로 가면서 (1,0), (2,0), (3,0)식으로 증가하고 아래쪽으로 가면서 (0,1), (0,2), (0,3) 식으로 중가하게 된다.

위의 세 함수중 line()함수는 CP를 변형 시키지 않는다는데 주의하기 바란다
위의 함수중에 CP를 변형시키는 함수는 linerel()과 lineto()이다.

illu1003.dat

프로그램상에서 CP를 변형시키고 싶다면 굳이 linerel()이나 lineto()함수를 사용하지 않고도 방법이 있다. moveto()와 moverel()함수를 사용하는것 이다.

```
void moveto(int x, int y);
```

CP를 (x,y)로 옮긴다.

```
void moverel(int dx, int dy);
```

ㆆ
CP를 현재위치에서 (dx,dy)만큼 더한 위치로 옮긴다.

만약에 현재의 CP를 알고 싶다면 getx()와 gety()를 사용하면 된다.

ㆆl

```
int   getx(void);
```

ㆆ
CP의 x좌표를 얻는다.

ㆆl

```
int   gety(void);
```

ㆆ
CP의 y좌표를 얻는다.

ㆆlsam1003.dat

```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
{
  int i,j,k;

  initgraph(&GD,&GM,"");                    그래픽 초기화
  moveto(100,100);                          (100,100)으로 CP를 옮긴다.
  putpixel(getx(),gety(),WHITE);        현재 CP에 흰색점을 찍는다.
  getch();                                  키입력을 기다린다.
```

```
        closegraph();                          그래픽을 종료한다.
    }
```
ㆆ

선의 양식과 굵기도 마음대로 정의하여 사용할 수 있다. 다음의 두 함수가 그런일을 도와준다.

ㆆI
```
    void setcolor(int color);
```
ㆆ
그어질 선의 색을 color로 정한다. setcolor()에 의해서 색이 결정되면 이후의 선은 정해진 색으로 그려진다.

ㆆI
```
    void setlinestyle(int linestyle, unsigned upattern,
                      int thickness);
```
ㆆ
현재의 선의폭과 양식을 주어진 대로 맞춘다.

setlinestyle()에 의해서 정해진 선의 형태는 line(), lineto(), rectangle(), drawpoly(), arc(), circle()등의 함수에 영향을 준다.

선들은 보통선, 점선, 중심선, 또는 대쉬선으로 그려질 수 있다. 만일 적당하지 않은 입력이 setlinestyle()에 입력되면 graphresult()는 11을 리턴하며 선의 형태는 변하지 않는다.

인자`linestyle`에는 다음과 같은 상수를 사용할 수 있다.
ㆆI
```
    enum line_styles
    {
            SOLID_LINE    = 0,
            DOTTED_LINE   = 1,
            CENTER_LINE   = 2,
            DASHED_LINE   = 3,
            USERBIT_LINE = 4,
    };
```

ত

인자`thickness`는 선의 굵기를 말하며 다음과 같은 상수를 사용할 수 있다.

তI
```
    enum line_widths {
            NORM_WIDTH   = 1,
            THICK_WIDTH = 3,
    };
```
ত

C-언어에서 대부분의 그래픽 루틴은 set...형태의 함수가 있으면 get... 형태의 함수가 있다. 앞의 루틴에 관계된 함수는 다음과 같다.

তI
```
    int   getcolor(void);
```
ত

현재 쓰이고 있는 색상을 알려준다.

তI
```
    int   getbkcolor(void);
```
ত

현재의 바탕색을 알 수 있게 해준다.

তI
```
    void setbkcolor(int color);
```
ত

현재의 바탕색을 color로 바꾼다.

তIsam1004.dat
```
    [예제] 색상예제
    #include <stdio.h>
    #include <graphics.h>

    int GD=DETECT,GM;
```

main()
.... 계 속 ....
ㄷ

ㄷlsam1004.dat
```
{
   int x,y,k,color=0;

   initgraph(&GD,&GM,"");                  .... 그래픽 초기화
   for (y=0;y<4;y++)
   for (x=0;x<4;x++)
   {
      setfillstyle(SOLID_FILL,color); .... 다각형의 색상을 정한다.
      bar(x*50,y*50,x*50+40,y*50+40); .... 다각형을 그린다.
      color++;                             .... 색상을 바꾼다.
   }
   getch();                                .... 키입력을 기다린다.
   closegraph();                           .... 그래픽을 종료한다.
}
```
ㄷ
위의 예제중에 setfillstyle()함수와 bar()함수는 아직 배우지 않은것인데, 곧 배우게 될 것이다.

ㄷl
```
void getlinesettings(struct linesettingstype far *lineinfo);
```
ㄷ
setlinestyle()에 의해서 주어진 현재의 선의 양식, 패턴, 굵기 등을 얻는다.

인수로 쓰이는 linesettingstype은 다음과 같이 정의되어 있다.

ㄷl
```
struct linesettingstype {
               int linestyle;
               unsigned upattern;
               int thickness;
};
```
ㄷ

현재 쓰이고 있는 그래픽 화면의 크기를 알 필요가 있다.   많이 쓰이는 VGA모드는 640 x 480 의 해상도를 가진다.

ㅎl

```
int   getmaxx(void);
```

ㅎ

현재의 그래픽 모드에서 사용할 수 있는 최대의 X축값을 돌려준다.

ㅎl

```
int   getmaxy(void);
```

ㅎ

현재의 그래픽 모드에서 사용할 수 있는 최대의 Y축값을 돌려준다.

7. 다각형과 채색, 무늬

ㅎl

```
void rectangle(int x1, int y1, int x2, int y2);
void bar(int x1, int y1, int x2, int y2);
void bar3d(int x1, int y1, int x2, int y2,
           int depth, int topflag);
void drawpoly(int numpoints, int far *polypoints);
void fillpoly(int numpoints, int far *polypoints);
void getfillpattern(char far *pattern);
void getfillsettings(struct fillsettingstype far *fillinfo);
void setfillpattern(char far *upattern, int color);
void setfillstyle(int pattern, int color);
void floodfill(int x, int y, int border);
```

ㅎ

7.1 직사각형

```
void rectangle(int left, int top, int right, int bottom);
```

setlinestyle()에 의해서 정해진 선의 양식과 setcolor()에 의해서 정해진 선의 색으로 직사각형을 그린다.

(x1 ,y1)은 사각형 윈쪽 구석을 정의하고 (x2,y2)는 아래 오른쪽 구석을 정의한다.

다음의 예제는 화면에 마음대로(?) 선을 그릴것이다.

sam1005.dat
```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
```
.... 계속 ....

sam1005.dat
```
{
  int x,y,k,color=0;

  initgraph(&GD,&GM,"");                    .... 그래픽 초기화
  while(!kbhit())
  {
    setcolor(random(16));
    rectangle(random(640),random(480),
              random(640),random(480)); .... 사각형을 그린다.
  }
  getch();                                  .... 키입력을 기다린다.
  closegraph();                             .... 그래픽을 종료한다.
```

7.2 막대 그래프

ㅎI

```
void bar(int left, int top, int right, int bottom);
```

ㅎ

시작점 (x1,y1)과 끝점 (x2,y2)를 대각선으로 하는 막대를 그린다. rect angle과는 달리 안이 채색무늬와 선으로 채워진다.

ㅎI

```
void bar3d(int x1, int y1, int x2, int y2,
           int depth, int topflag);
```

ㅎ

bar()함수는 통계 도표등을 만들려고 할때 매우 유용하게 사용될 수 있으며 안이 채워져 있다.

bar()함수가 평면적인 것과 달리 bar3d()는 입체적인 감각을 느낄 수 있도록 해준다.

bar3d()는 입체감을 나타내기 위하여 테두리 선을 그리는데 이때 테두리 선은 setcolor()와 setlinestyle()의 영향을 받는다.

인자`depth`는 외부선의 점들의 수이다. 전형적인 depth는 막대폭의 25%로 계산되어 진다. 만일 인자`top`이 1이면 막대의 입체감이 느껴지도록 막대 위에도 외부선을 그려주고 top 이 0이면 막대 위에 아무것도 놓지 않는다.

ㅎIsam1006.dat

```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
{
  initgraph(&GD,&GM,"");          .... 그래픽 초기화
```

.... 계 속 ....

ㅎ

```
ᄒIsam1006.dat
        bar(10,10,50,50);                    .... bar를 그린다.
        bar3d(110,10,150,50,50,1);       .... 입체 bar를 그린다.
        getch();                                 .... 키입력을 기다린다.
        closegraph();                         .... 그래픽을 종료한다.
    }
ᄒ
```

7.3 다각형

다각형을 그리거나 채우는 데는 다음의 두 함수가 유용하게 쓰인다.

```
ᄒI
    void drawpoly(int numpoints, int far *polypoints);
ᄒ
```

최근 설정된 선의 양식과 색으로 다각형의 외형을 그린다.

```
ᄒI
    void fillpoly(int numpoints, int far *polypoints);
ᄒ
```

스캔 변환기를 이용하여 다각형의 내부를 채운다.

drawpoly()는 속이 빈 다각형을 그리고, 내부가 채워진 다각형을 그리는 함수는 fillpoly()이다. polypoints는 다각형의 각 좌표를 가지고 있는 정수형 포인터이다.

ᄐIsam1007.dat

```
     #include <stdio.h>
     #include <graphics.h>

     int GD=DETECT,GM;
     main()
     {
       struct PTS poly[ 6 ];
       int color;
       int i;

       initgraph(&GD,&GM,"");              .... 그래픽 초기화
       while( !kbhit() )
       {
         color = 1 + random( MaxColors-1 );
         setfillstyle( random(10), color );
         setcolor( color );
```

.... 계 속 ....

ᄐ

ᄐIsam1007.dat

```
         for( i=0 ; i<(6-1) ; i++ )
         {
           poly[i].x = random( MaxX );
           poly[i].y = random( MaxY );
         }

         poly[i].x = poly[0].x;
         poly[i].y = poly[1].y;

         fillpoly( 6, (int far *)poly );
       }
       closegraph();
     }
```

ᄐ

위의 예제는 fillpoly()함수를 이용하여 화면에 채원진 다각형을 마구 그릴것이다. 작업을 멈추기 위해서는 아무키나 누르면 된다.

7.4 채색과 무늬

```
void floodfill(int x, int y, int border);
```

현재 정해진 채색 양식과 색상으로 특정 영역 내부를 채운다.

```
void setfillstyle(int pattern, int color);
```

채색 무늬와 색상을 선택한다.

앞에서 몇 가지 다각형을 그리는 함수들에 대해서 알아 보았다.
여기서는 그러한 루틴들을 사용하여 그려 놓은 폐곡선의 내부를 채우는 `floodfill()`함수에 대해서 알아보자.

floodfill 함수는 아래와 같이 사용된다.

`floodfill(x,y,border);`

(x, y)는 채색하고자 하는 폐곡선 내의 임의의 점으로써, 여기서 처음으로 채색이 시작되기 때문에`씨점(seed point` 이라고도 한다.

이 씨점도 반드시 폐곡선 안의 점 이어야 하며 폐곡선 경계여서도 않된다.
border는 폐곡선의 경계선 색깔이다.
이 색은 정확하게 지정하여 주어야 한다. 그렇지 않으면 화면 전체가 칠 해지는 불행한(?) 사태가 발생한다.

illu1004.dat

폐곡선의 채우는 동안 에러가 발생한다면 함수 graphresult()는 -7을 리

턴한다. (grNoFloodMem) 에러가 발생하지 않도록 씨점과 border를 잘 정
하면 floodfill()은 씨점에서 출발하여 경계선 내부를 정해진 무늬로 채
운다.

이때 채색무늬와 색을 결정하는 함수로`setfillstyle()`가 사용될 수 있
다.

`void setfillstyle(무늬번호 , 무늬색상);`

무늬 색상은 허큘리스(단색)에서는 당연히 1이 되고 VGA사용자는 0~15까
지의 16가지 색을 사용할 수 있다.

무늬는 0부터 12까지의 13가지를 사용할 수 있는데 숫자를 직접 사용하
지 말고 graphics.h에 선언되어 있는 상수를 사용하면 된다. 사용할 수
있는 상수는 다음과 같다.

```
EMPTY_FILL        0
SOLID_FILL        1
LINE_FILL         2
LTSLASH_FILL      3
SLASH_FILL        4
BKSLASH_FILL      5
LTBKSLASH_FILL    6
HATCH_FILL        7
XHATCH_FILL       8
INTERLEAVE_FILL   9
WIDE_DOT_FILL     10
CLOSE_DOT_FILL    11
USER_FILL         12
```

ড l

```
void setfillpattern(char far *upattern, int color);
```

ড

사용자가 정의한 채색 무늬를 선택한다.

ㅎI
```
void getfillsettings(struct fillsettingstype far *fillinfo);
```
ㅎ
setfillstyle()이나 setfillpattern()에 의해서 선정된 현재의 채색 무
늬와 색상을 알려준다.

여기서 첫번째 나열됭 getfillsettings()는 setfillstyle() 함수에서 선
전하거나 미리 내정되어 있는 현재의 채색무늬와 색상에 대한 정보를 알
아내며, fillsettingstype은 graphincs.h에 다음과 같이 정의되어 있다.

```
struct fillsettingstype {
        int pattern;
        int color;
};
```

8. 원과 원호 그리고 화면의 종횡비

ㅎI
```
void arc(int x, int y, int stangle, int endangle,
         int radius);
void getarccoords(struct arccoordstype far *arccoords);
void circle(int x, int y, int radius);
void ellipse(int x, int y, int stangle, int endangle,
             int xradius, int yradius);
void getaspectratio(int far *xasp, int far *yasp);
void pieslice(int x, int y, int stangle, int endangle,
              int radius);
void fillellipse( int x, int y, int xradius, int yradius );
void sector( int X, int Y, int StAngle, int EndAngle,
                          int XRadius, int YRadius );
```
ㅎ

ㅎI
```
void circle(int x, int y, int radius);
```
ㅎ

(x,y)를 중심으로 반지름이 radius인 원을 그린다.

circle()함수는 중심이 (x,y)이고 반지름이 radius인 원을 그리는데 이를 점 단위를 그리면 정확한 원이 아니라 세로 방향으로 길쭉한 원이 그려지게 된다.

이는 화면을 이루고 있는 점이 정사각형이 아니고 y축이 약간 긴 직사각형이기 때문이다.

illu1100.dat

그래서 circle()함수는 이를 보정하기 위해서 특수한 방법을 사용하는데 x축은 그대로 두고 y축을 비율을 참작하여 축소시킨다. 그러면 실제로는 y축으로 약간 찌그러진 원이 되지만 눈으로 보기에는 완전한 원이 그려진다.

그러므로 y축을 얼마나 줄여야 하는가가 문제가 되는데 이때 그 비율을 `종횡비`라 하고 시스템마다의 종횡비를 알기위해서는 getaspectratio()를 호출하면 된다.

```
void getaspectratio(int far *xasp, int far *yasp);
```

화면의 종횡비를 구해준다.

```
void ellipse(int x, int y, int stangle, int endangle,
             int xradius, int yradius);
```

원과 유사한 것으로 타원이 있는데 이
타원을 그려주는 함수가 ellipse() 함
수 이다.

ellipse()는 (x,y)를 중심으로 stangle
에서 endangle까지, x축 반지름이 xrad
ius이고 y축 반지름이 yradius인 타원
을 그린다.

`[그림]`
` ellipse(400,200,0,360,200,50);`

ঘillu1101.dat

fillellipse()함수는 안이 채원진 타원을 그린다.

ঢl
```
void fillellipse( int x, int y, int xradius, int yradius );
```
ঢ
ellipse()와 같으나 안이 채워진 타원을 그린다. ellipse()와는 달리 타
원의 일부를 그리는것이 아니고 채워진 전체를 그리므로 각도는 필요없
다.

ঢlsam1100.dat
```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
{
```
.... 계속 ....
ঢ

ঢlsam1100.dat
```
    initgraph(&GD,&GM,"");                    .... 그래픽 초기화
    while(!kbhit())
    {
      fillellipse(random(640),random(480),
```

```
                    random(100),random(100));
        }
        getch();                              .... 키입력을 기다린다.
        closegraph();                         .... 그래픽을 종료한다.
    }
```
ᄚ

위의 예제는 화면에 여러개의 채워진 타원을 키보드가 눌려질때 까지 계속 그릴것이다.

ᄚl
```
    void sector( int X, int Y, int StAngle, int EndAngle,
                              int XRadius, int YRadius );
```
ᄚ

secter()함수는 circle()보다는 fillel
lipse()에 가까운 함수라고 말할 수 있
다.

secter()는 (x,y)를 중심으로 stangle
부터 endandle까지 x축 반지름이 xrad
ius이고 y반지름이 yradius인 속이 채
워진 타원을 그린다.

fillellipse()와 다른점은 원성되지 않
은 형태의 타원을 그린다는데 있다.

ᅏillu1102.dat

ᄚl
```
    void arc(int x, int y, int stangle, int endangle,
             int radius);
```
ᄚ
(x,y)를 중심으로, stangle에서 endangle까지 반지름이 radius인 원호를 그린다.

한편 마지막으로 그려진 원호에 한하여 그 원호의 시작좌표과 끝좌표를 알려주는 기능을 가진`getarccoords()`가 있다.

ㆆI
```
void getarccoords(struct arccoordstype far *arccoords);
```
ㆆ

getarccoords()는 arccoordstype값을 받는데 이는 다음과 같이 정의되어 있다.

ㆆI
```
struct arccoordstype {
     int x, y;
     int xstart, ystart, xend, yend;
};
```
ㆆ

이 함수를 통해 얻어지는 값은 원호의 두 끝점을 연결하는 직선을 그을 필요가 있는 경우 편리하다.

ㆆIsam1100.dat
```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
{
```

```
        struct arccoordstype ai;

        initgraph(&GD,&GM,"");                  .... 그래픽 초기화
        arc(300,200,0,270,100);
        getarccoords( &ai );
        line( ai.xstart , ai.ystart ,
              ai.xend , ai.yend );
        getch();                                .... 키입력을 기다린다.
        closegraph();                           .... 그래픽을 종료한다.
    }
```
ㅎ

9. 드라이버와 모드의 결정

ㅎI
```
    void   detectgraph(int far *graphdriver,int far *graphmode);
    char *getdrivername( void );
    int    getmaxmode(void);
    void   getmoderange(int graphdriver, int far *lomode,
                        int far *himode);
    char *getmodename( int mode_number );
    void   setgraphmode(int mode);
    int    getgraphmode(void);
```
ㅎ

ㅎI
```
    void   detectgraph(int far *graphdriver,int far *graphmode);
```
ㅎ

허드웨어를 검사하여 graphdriver와 graphmode를 결정한 후 initgraph에
그값을 보낸다. 그러면 옳바른 드라이버가 결정될 것이다.

initgraph()는 쓰기전에 detectgraph()를 직접 사용하는 경우가 있는데
이는 detectgraph()에서 얻어진 값을 무시하고자 하는 경우에 사용된다.

예를 들어 VGA카드에서 CGA모드를 사용하고 싶을 때가 그런경우 이다.

ㆆl

```
char *getdrivername( void );
```

ㆆ

현재의 드라이브명이 포함된 문자열을 돌려준다.

ㆆl

```
char *getmodename( int mode_number );
```

ㆆ

현재의 그래픽 모드명을 포함한 문자열을 돌려준다.

터보-C는 그래픽 모드에 관한 정보를 제공하기 위해서 그리고 다른 모드를 사용하기 위해서 다음의 몇가지 함수를 더 제공한다.

ㆆl

```
void   setgraphmode(int mode);
```

ㆆ

시스템을 그래픽 모드로 전환하고 화면을 지운다.

ㆆl

```
int    getgraphmode(void);
```

ㆆ

현재의 그래픽 모드를 알려준다.

ㆆl

```
int    getmaxmode(void);
```

ㆆ

현재 사용중인 드라이버에서 사용 가능한 최대의 모드수를 알려준다.

ㆆl

```
void   getmoderange(int graphdriver, int far *lomode,
                    int far *himode);
```

ㆆ

주어진 드라이버에 대하여 유효한 그래픽 모드의 최소값과 최대값을 얻
는다.

```
ㅎl
-----------------------------------------------------------------
   모드명        상수            해상도    팔레트/칼라
-----------------------------------------------------------------
    CGAC0        = 0,            320x200 palette 0; 1 page
    CGAC1        = 1,            320x200 palette 1; 1 page
ㅎ

ㅎl
-----------------------------------------------------------------
   모드명        상수            해상도    팔레트/칼라
-----------------------------------------------------------------
    CGAC2        = 2,            320x200 palette 2: 1 page
    CGAC3        = 3,            320x200 palette 3; 1 page
    CGAHI        = 4,            640x200 1 page
    MCGAC0       = 0,            320x200 palette 0; 1 page
    MCGAC1       = 1,            320x200 palette 1; 1 page
    MCGAC2       = 2,            320x200 palette 2; 1 page
    MCGAC3       = 3,            320x200 palette 3; 1 page
    MCGAMED      = 4,            640x200 1 page
    MCGAHI       = 5,            640x480 1 page
    EGALO        = 0,            640x200 16 color 4 pages
    EGAHI        = 1,            640x350 16 color 2 pages
    EGA64LO      = 0,            640x200 16 color 1 page
    EGA64HI      = 1,            640x350 4 color  1 page
    EGAMONOHI  = 0,              640x350
ㅎ

ㅎl
-----------------------------------------------------------------
   모드명        상수            해상도    팔레트/칼라
-----------------------------------------------------------------
    HERCMONOHI = 0,              720x348 2 pages
    ATT400C0     = 0,            320x200 palette 0; 1 page
    ATT400C1     = 1,            320x200 palette 1; 1 page
    ATT400C2     = 2,            320x200 palette 2; 1 page
    ATT400C3     = 3,            320x200 palette 3; 1 page
    ATT400MED   = 4,             640x200 1 page
```

```
    ATT400HI    = 5,           640x400 1 page
    VGALO       = 0,           640x200 16 color 4 pages
    VGAMED      = 1,           640x350 16 color 2 pages
    VGAHI       = 2,           640x480 16 color 1 page
    PC3270HI    = 0,           720x350 1 page
    IBM8514LO   = 0,           640x480 256 colors
    IBM8514HI   = 1            1024x768 256 colors
```
ㅎ

10. 그래픽에서의 텍스트 출력

ㅎI
```
    void gettextsettings(struct textsettingstype far *texttypeinfo);
    void outtext(char far *textstring);
    void outtextxy(int x, int y, char far *textstring);
    void settextjustify(int horiz, int vert);
    void settextstyle(int font, int direction, int charsize);
    void setusercharsize(int multx, int divx,
                         int multy, int divy);
    int  textheight(char far *textstring);
    int  textwidth(char far *textstring);
```
ㅎ

그래픽 화면에 텍스트를 출력하는 것은 두 가지 방법을 이용해서 한다. 첫번째 방법은 출력할 위치의 좌표를 지정하여 그 위치에 텍스트를 출력하는 것으로 outtextxy()가 그런일을 수행한다.

ㅎI
```
    void outtextxy(int x, int y, char far *textstring);
```
ㅎ

(x,y) 위치에 textstring을 출력시킨다. 여기서 주의할 점은 outtextxy는 printf() 와는 그 성질이 다르다는 것이다. printf() 한수는 정수형을 출력할 때도 그냥 printf("%d",i); 라고 하면 되었지만 outtextxy() 함수는 그런식으로 사용될 수 없다.

outtextxy()함수는 문자형 포인터형만을 받으므로 정수를 출력하기 위해서는 정수를 문자형 포인터로 바꾸어 주는 itoa()나 ltoa() 함수를 사용하여야 한다.

ъsam1101.dat

```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
```

.... 계속 ....

ъ

ъsam1101.dat

```
main()
{
  int i=52;
  char number[20];

  initgraph(&GD,&GM,"");          .... 그래픽 초기화
  itoa(i,number,10);              .... 정수를 문자형으로
  outtextxy(100,100,number);   .... 화면에 출력한다.
  getch();                            .... 키입력을 기다린다.
  closegraph();                     .... 그래픽을 종료한다.
}
```

ъ

outtextxy()의 중요한 특징은 전장에서 설명한 CP를 전혀 변형시키지 않는다는 점이다. 텍스트를 출력하는 두번째 방법은 현재의 CP의 위치에 outtext()함수를 사용하는 방법이다.

ъ

```
void outtext(char far *textstring);
```

ъ

현재점(CP) 에서 시작하여 문자를 출력시킨다.

outtext()함수는 outtextxy()함수와는 달리 좌표를 지정하지 않고 CP의 위치에다 문자열을 출력한다. 그리고 일단 문자열을 출력한 후에 CP를 그 문자열의 끝나는 다음 위치로 이동시킨다.

이 함수는 출력위치를 CP에 의존하기 때문에 원하는 위치에 문자를 출력시키려면 moveto()나 moverel()함수에 의존하여야 한다.
현재 CP에 문자를 출력하고 싶다면 outtext()함수를 사용할 수 있지만

화면 여기저기에 문자를 출력하려고 한다면 outttextxy()함수를 사용하는것이 간편할 것이다.

그런데 그래픽 화면에는 텍스트 화면과는 달리 줄 (line) 이라는 개념이 없기 때문에 한라인 문자를 쓰고 다음 라인으로 넘어가려고 하면 글자의 크기를 모르고 있는한 심란한(?) 문제가 될 수 밖에 없다.

그래서 문자를 크기를 알려주는 함수가 필요하다.

তI
```
int   textheight(char far *textstring);
```
ত
문자의 높이(세로 크기)를 점단위로 알려준다.

তI
```
int   textwidth(char far *textstring);
```
ত
문자의 넓이(가로 크기)를 점단위로 알려준다.

그러면 이제 출력되는 텍스트의 폰트(Font), 형태, 비율, 배열형태, 그리고 stroked폰트의 문자폭과 높이를 변화시키는 루틴에 대해서 알아보자.

তI
```
void settextstyle(int font, int direction, int charsize);
```
ত
이 후에 출력될 텍스트의 폰트(font), 형태, 문자 배율 등을 전해준다.

তI
```
void settextjustify(int horiz, int vert);
```
ত
outtext()와 outtextxy()에 사용되는 문자 배열의 형태를 정한다.

옆의 그림은 터보-C에서 제공하는 글자체이다.

터보-C 2.0 에서는 모두 4개의 스트로크 폰트를 사용할 수 있는데 반해 Borland C++ 3.0 에서는 모두 10개의 스트로크 폰트를 사용할 수 있다.

ꠑill10-1.flm

해당폰트를 선택하고 사용하려고 할때 각 폰트에 해당되는 폰트파일(*.CHR)이 디스크에 없으면 비트맵폰트인 default 폰트가 선택되는데 주의하기 바란다.

ꠓI

```
void gettextsettings(struct textsettingstype far *texttypeinfo);
```

ꠓ

settextstyle()과 settextjustify()에 의해서 결정된 현재의 텍스트 폰트, 방향, 크기, 그리고 문자 배열 형태를 얻는다.

ꠓI

```
void setusercharsize(int multx, int divx,
                     int multy, int divy);
```

ꠓ

사용자가 스트로크 폰트의 문자폭과 높이를 변화시킬 수 있도록 해준다.

settextstyle()함수는 outtext()와 outtextxy()를 사용하는 모든 텍스트 출력에 영향을 준다. 한개의 8x8형태의 비트맵 폰트와 4개의 스트로크

폰트가 사용 가능하다. (Borland C++ 에서는 10개)

ㅎl
```
DEFAULT_FONT         = 0,    /* 8x8 bit mapped font */
TRIPLEX_FONT         = 1,    /* "Stroked" fonts */
SMALL_FONT           = 2,
SANS_SERIF_FONT      = 3,
GOTHIC_FONT          = 4,
SCRIPT_FONT          = 5,
SIMPLEX_FONT         = 6,
TRIPLEX_SCR_FONT     = 7,
COMPLEX_FONT         = 8,
EUROPEAN_FONT        = 9,
BOLD_FONT            = 10
```
ㅎ

위의 상수중에 터보-C 사용자는 4번의 고딕폰트까지 밖에 사용할 수 없다. 5~10 까지는 Borland C++ 사용자 만이 사용할 수 있다.

폰트의 출력 방향은 왼쪽에서 오른쪽으로 진행되는 것이 보통이나 아래에서 위로 출력되는 수직적인 것이 있으며 settextstyle()함수의 direction 인자를 통해 정해진다.

ㅎl
```
#define HORIZ_DIR   0   /* left to right */
#define VERT_DIR    1   /* bottom to top */
```
ㅎ

각 글자의 크기는 charsize를 통해 전달되며 그 값 만큼의 배가 된다. 예를 들어 charsize값이 1이면 8x8비트 맵된 폰트가 화면상에 8x8의 형태로 표시되고 값이 2이면 16x16의 크기로 화면에 표시된다.

문자의 크기를 알기 위해서는 앞에서 설명한`texthight()와 textwidth()`

함수를 사용한다. 내정된 폰트에서의 일반적인 크기는 charsize=1 이고 스트로크 폰트에서는 4 이다.

스트로크 폰트들이 메모리에 로드될때 에러가 발생할 수 있다. 만일 에러가 발생하면 graphresult()는 다음값 중의 하나를 리턴한다.

ㅎI

```
 -8  : 폰트 파일을 찾지 못함
 -9  : 메모리가 불충분하여 파일을 읽을 수 없음
 -11 : 그래픽 에러
 -12 : 그래픽 입출력 에러
 -14 : 부적합한 폰트 파일
 -15 : 부적합한 폰트 번호
```

ㅎ

settextjustify()는 예를 통해서 이해하는 것이 가장 빠른 방법인것 같다. 다음을 보자.

ㅎI

```
    settextjustify(centertext , centertext);
    outtextxy(200,200,"Choi min suk");
```

ㅎ

위에서 centertext는 문자열 가운데에 (x,y) 위치가 오게끔 하는 인자값으로 graphics.h에 상수로 선언되어 있다.
그러므로 (100,100)에는 문자 "i"가 나타날 것이다. 문자 배열형태의 내정값은 settextjustify(lefttext,toptext) 이다. 다음의 상수가 정해져 있다.

ㅎI

```
        LEFT_TEXT          = 0,
        CENTER_TEXT         = 1,
        RIGHT_TEXT         = 2,

        BOTTOM_TEXT         = 0,
        CENTER_TEXT         = 1,   already defined above
        TOP_TEXT           = 2
```

ㅎ

gettextsetttings()함수는 settextstyle()과 settextjustify()함수에서
정한 텍스트 폰트, 방향, 크기, 문자배열 형태를 알아낸다. 이 함수 안
의 인자인 textsettingstype은 다음과 같이 정의되어 있다.

ㅎI

```
struct textsettingstype {
        int font;
        int direction;
        int charsize;
        int horiz;
        int vert;
};
```

ㅎ

setusercharsize()함수는 사용자가 스트로크 폰트를 사용할 때 문자의
폭과 높이를 바꾸어 줄 수 있게한다.

ㅎI

```
void setusercharsize(int multx, int divx,
                     int multy, int divy);
```

ㅎ

multx : divx는 사용중인 폰트의 일반적인 폭을 곱한 비율이고
multy : divy는 사용중인 폰트의 일반적인 높이를 곱한 비율이다.

ㅎIsam1102.dat

```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
```

.... 계속 ....
ᇂ

ᇂIsam1102.dat
```
     {
        int i=52;
        char number[20];

        initgraph(&GD,&GM,"");          .... 그래픽 초기화
        settextstyle(TRIPLEX_FONT,HORIZ_DIR,4);
        outtext("normal");
        setusercharsize(1,3,1,1);
        outtext("short");
        setusercharsize(3,1,1,1);
        outtext("wide");
        getch();                        .... 키입력을 기다린다.
        closegraph();                   .... 그래픽을 종료한다.
     }
```
ᇂ

11. 화면과 페이지 조작

ᇂI
```
     void cleardevice(void);
     void setvisualpage(int page);
     void setactivepage(int page);
```
ᇂ

텍스트 화면에서 화면을 지워주는 함수로는 clrscr()이 있다. 그래픽 모드에서도 화면을 지우는 함수가 있는데 바로 cleardevice()함수 이다.

ᇂI
```
     void cleardevice(void);
```
ᇂ
cleardevice()는 CP를 (0,0)으로 옮기고 화면을 setbkcolor()에서 정해진 색을로 깨끗이 지운다음 다음 출력을 준비한다.

EGA와 허큘리스 에서는 여러개의 페이지를 그래픽 출력을 위해서 사용할 수 있는데 안타깝게도 VGA (640x480)모드에서는 제공하지 않는다.
사실 페이지 관련 함수는 직접 프로그램을 작성하는데 에는 별로 사용되지 않을 것이다.

ㆆl

```
void setvisualpage(int page);
```

ㆆ

보이는 페이지를 정한다.

ㆆl

```
void setactivepage(int page);
```

ㆆ

그래픽 출력을 현재 사용중인 페이지로 맞춘다.

12. 비트 이미지 (Bit image)

ㆆl

```
unsigned imagesize(int x1, int y1, int x2, int y2);
void getimage(int x1, int y1, int x2, int y2,void far *bitmap);
void putimage(int x1, int y1, void far *bitmap, int op);
```

ㆆ

비트 이미지는 간단한 애니메이션과 화면상에 무었을 그렸다가 복구시키는 등의 일을 할때 기초가 되는 부분이다.
워낙에 속도가 느려서 응용 프로그램에서 사용하기 에는 무리가 따르지만 기초를 잘 닦아 두어야 하는 부분이다.

터보-C 에서 비트 이미지를 다루기 위한 함수는 다음의 3가지가 있다.

তl

```
void getimage(int x1, int y1, int x2, int y2,void far *bitmap);
```

ত

주어진 영역의 내부 (x1,y1,x2,y2의 내부)의 이미지를 버퍼에 저장한다.

তl

```
void putimage(int x1, int y1, void far *bitmap, int op);
```

ত

저장된 비트이미지를 화면상의 (x,y)에 출력한다.

তl

```
unsigned imagesize(int x1, int y1, int x2, int y2);
```

ত

화면의 4각을 저장하는데 필요한 메모리의 바이트 수 를 알려준다.

getimage()함수는 그래픽 화면이 좌상단이 (x1,y1), 우하단이 (x2,y2)로 설정된 사각형의 영역 내부의 상을 버퍼에 저장해 둔다. 이렇게 얻어진 상은 그래픽 화면에서`복사, 이동`등을 할 때 사용된다.

이때 void far *bitmap 의 첫번째 2바이트에는 이미지의 높이가 두번째 2바이트에는 이미지의 길이가 들어가고 나머지는 실제 이미지가 들어가게 된다.

putimage()함수는 getimage()함수에 의해서 얻어진 비트 이미지를 화면에 출력하는 역할을 한다. (x,y)는 화면에서 출력될 위치의 사각영역에서 좌상단 점이다.

출력은 getimage()에서 얻어진 이미지를 그대로 사용할 수도 있지만 약간의 가공을 해서 사용할 수 도 있다.

COPY_PUT ...... 어셈블리의 mov명령사용 원래의 이미지를 그대로 화면에 출력함
XOR_PUT ...... 어셈블리의 xor명령사용 에니메이션 기법에 종종 쓰인다.
OR_PUT
AND_PUT
NOT_PUT ...... 어셈블리의 not명령사용 반전 이미지가 나타난다.

만약 이미지의 일부가 화면 바깥에 나가게 되면 이미지 전체가 나타나지 않으므로 주의해야 한다. 또한 이미지를 저장하기 위한 공간은 반드시 64K 이하여야 한다.

64K라고 하면 어느정도 인지 의아해 할지 모르겠는데 VGA 640x480 모드 화면 전체를 저장하는데 150K정도가 필요하므로 64K면 화면의 1/3정도는 저장할 수 있다고 보면된다.

함수 imagesize()는 이미지를 저장할때 필요한 메모리를 계산하기 위해서 반드시 필요한 함수로 결과값은 바이트 값이다.

다라서 imagesize()는 getimage()로 이미지를 얻기 전에 먼저 사용되어야 한다. 만약 어떤이미지를 저장하는데 드는 메모리가 64K를 초과하면 graphresult()함수는 -11(GrError)을 리턴한다.

illu1103.dat

비트 이미지를 처리하려면 대개 다음과 같은 과정을 거친다.

ㅎI
```
    size = imagesize(x1,y1,x2,y2);    ..... 필요한 메모리를 얻음
    p = malloc(size);                 ..... 메모리 할당
    getimage(x1,y1,x2,y2,p);          ..... 이미지 입력
    putimage(x1,y1,p,COPY_PUT);       ..... 이미지 출력
```
ㅎ

다음의 예제는 움직이는 그림을 보여준다.

ㅎIsam1103.dat
```
    #include <stdio.h>
    #include <graphics.h>

    int GD=DETECT,GM;
```
.... 계속 ....

ㅎ

ㅎIsam1103.dat
```
    main()
    {
      int i;
      char number[20];
      void far *p;
      unsigned size;

      initgraph(&GD,&GM,"");        .... 그래픽 초기화
      circle(320,240,20);
      size = imagesize(300,220,240,260);
      p = malloc(size);
      getiamge(300,220,240,260,p);
      for (i=300;i<=500;i++)
      putimage(i,220,p,COPY_PUT);
      getch();                      .... 키입력을 기다린다.
      closegraph();                 .... 그래픽을 종료한다.
    }
```
ㅎ

13. 팔레트 (Palette)

ㅎI

```
void setallpalette(struct palettetype far *palette);
void setpalette(int colornum, int color);
void getpalette(struct palettetype far *palette);
int  getpalettesize( void );
struct palettetype *getdefaultpalette( void );
void setrgbpalette(int colornum,
                   int red, int green, int blue);
```

ㄷ

허큘리스 카드를 제외한 다른 그래픽 카드는 모든 그래픽 모드에서 다양한 색을 사용할 수 있다. 그런데 각 모드에서 사용할 수 있는 색의 수가 다르다는데에 주의하기 바란다.

PC화면의 출력은 색의 3원색과 강도라는 4가지 소자들의 보합에 의해 만들어 진다. 다음은 터보-C에서 그래픽을 위하여 정의하여 놓은 색의 상수로써 모든 그래픽 루틴에 사용된다.

ㄷl

```
BLACK       = 0     DARKGRAY     = 8
BLUE        = 1     LIGHTBLUE    = 9
GREEN       = 2     LIGHTGREEN   = 10
CYAN        = 3     LIGHTCYAN    = 11
RED         = 4     LIGHTRED     = 12
MAGENTA     = 5     LIGHTMAGENTA = 13
BROWN       = 6     YELLOW       = 14
LIGHTGRAY   = 7     WHITE        = 15
```

ㄷ

그래픽 모드에서는 색 선택을 미리 정의해 놓은 것이 있는데 이를 표준 팔레트 라고 한다. 표준 팔레트는 EGA나 VGA에서는 사용자 마음대로 바꿀 수가 있다.

다음은 표준 팔레트를 변형 시키기 위하여 제공되는 대표적인 함수이다.

ㄷl

```
void setallpalette(struct palettetype far *palette);
```

ᄒ

모든 팔레트(palette)의 색을 바꾼다.

ᄒl

```
void setpalette(int colornum, int color);
```

ᄒ

colornum과 color로 정해져 있는 팔레트(palette)의 한 색을 바꾼다.

ᄒl

```
int  getpalettesize( void );
```

ᄒ

현재 설정된 팔레트의 크기를 얻는다.

먼저 getpalette()함수에 대해 알아보자. getpalette()함수는 현재 쓰고 있는 팔레트와 그 크기를 알려준다. 인자 palette의 데이타 타입인 palettetype은 다음과 같이 정의되어 있다.

ᄒl

```
#define MAXCOLORS 15

struct palettetype {
    unsigned char size;
    signed char colors[MAXCOLORS+1];
};
```

ᄒ

여기서 size필드는 현재의 드라이브 모드에서 팔레트에 있는 색의 수를 나타낸다. color필드에 들어갈 수 있는 색의 수치는 size-1이다. setpalette()함수는 colornum으로 지적된 것을 color인자의 색으로 바꾸어 준다. 즉, 팔레트의 color 필드가 다음과 같이 된 것이다.

`color[colornum] = color`

그외에서 VGA사용자를 위해 좀더 진보된 팔레트 루틴으로 setrgbpalette()함수를 제공한다.

ᄒl

```
void setrgbpalette(int colornum,
                   int red, int green, int blue);
```

ㄷ
위의 팔레트 루틴은 색의 삼원색을 섞어서 우리가 원하는 색과 가장 가까운색을 사용할 수 있도록 해준다.
사용방법은 아래와 같다.

ㄷI

```
setrgbpalette(바꿀색 , 적색, 녹색, 청색);
```

ㄷ
들어갈 적색, 녹색, 청색은 0부터 63까지의 값이 될 수 있으며 값을 초과 하면 0으로 간주된다.

적색, 녹색, 청색의 모든 값이 0이면 검정색이고 모든 값이 63이면 흰색이 된다.

이 때 원하는 색을 만들기 위해서 원하는 색이 나올때 까지 계속 프로그램을 수정할 수 도 있다.

그러나 Deluxe Paint II와 같은 페인팅 프로그램을 이용하여 원하는 색을 찾아 내는 방법도 있다.

그러나 Deluxe Paint II는 0부터 100까지의 수를 사용하므로 0부터 63까지의 수로 나타내려면 중학교때 배운 비례식을 사용하여야 한다.

ㄱillu1104.dat

이런 방법 저런 방법이 모두 마음에 들지 않는다면 우리가 직접 원하는 팔레트를 만드는 프로그램을 만들 수 있다. 그리 어렵지 않다 한번 해 보자.

ㄷIsam1103.dat

```
#include <stdio.h>
#include <graphics.h>

int GD=DETECT,GM;
main()
{
  int i,R=20,G=20,B=20;
  char number[20];
  void far *p;
  unsigned size;

  initgraph(&GD,&GM,"");        .... 그래픽 초기화
```

ᇂ

ᇂlsam1103.dat

```
  setfillstyle(SOLID_FILL,BLUE);
  bar(220,140,420,340);

  DRAW:
  gotoxy(1,23);
  printf("Key q:red++ w:green++ e:blue++     SPACE:quit\n");
  printf("Key a:red-- s:green-- d:blue--\n");
  printf("RED %d    GREEN %d    BLUE %d   ",R,G,B);
  setrgbpalette(EGA_BLUE,R,G,B);

  while(1)
  {
    if kbhit()
    {
      switch(getch())
      {
        case 'q':case 'Q':
```

ᇂ

ᇂlsam1103.dat

```
                 R++; if (R>63) R=0; goto DRAW;
        case 'w':case 'W':
                 G++; if (G>63) G=0; goto DRAW;
        case 'e':case 'E':
                 B++; if (B>63) B=0; goto DRAW;
        case 'a':case 'A':
```

```
                R--; if (R<0) R=63; goto DRAW;
            case 's':case 'S':
                G--; if (G<0) G=63; goto DRAW;
            case 'd':case 'D':
                B--; if (B<0) B=63; goto DRAW;
            case ' ':goto ending;
          }
        }
      }
      ending:
      closegraph();                    .... 그래픽을 종료한다.
    }
```

그리고 현재 사용중인 그래픽 모드에서 사용가능한 색의 수와 내정된 하드웨어 팔레트를 알려주는 루틴은 다음과 같다.

```
    int   getpalettesize( void );
```

현재 로드되어 있는 그래픽 모드가 사용할 수 있는 색상의 수를 알려준다.

```
    struct palettetype *getdefaultpalette( void );
```

palettetype에서 현재의 팔레트를 알 수 있게 해준다.

14. 그밖의 함수들

```
    int registerbgidriver(void (*driver)(void));
    int registerfarbgidriver(void far *driver);
    int installuserdriver( char far *name, int huge (*detect)(void));
    int installuserfont( char far *name );
    void graphdefaults(void);
```

void restorecrtmode(void);
unsigned setgraphbufsize(unsigned bufsize);

ড

ডI

int registerbgidriver(void (*driver)(void));

ড

그래픽 시스템에 적당한 BGI 드라이버 파일을 기록해 둔다.

ডI

int registerbgifont(void (*font)(void));

ড

그래픽 시스템에 적당한 BGI 폰트 파일을 기록해 둔다.

ডI

int installuserdriver(char far *name, int huge (*detect)(void));

ড

vender-added 디바이스 드라이버를 사용할 수 있게 해준다. name은 디바이스 드라이버의 파일 이름이고 autodetectptr은 autodetect()함수의 포인터 이다.

ডI

int installuserfont(char far *name);

ড

BGI 시스템에 내장되어 있지 않은 새로운 폰트를 내장시킨다.

ডI

void graphdefaults(void);

ড

현재점 CP를 원점으로 되돌려 주며 다음에 열거 해 놓은 값들을 원래의 값으로 변경시킨다.

ডI

뷰포트	팔레트
문자와 배경색	선의 양식과 패턴
채색양삭, 색상, 무늬	

사용중인 폰트, 텍스트 양식, 문자 크기

ᄒ

히

unsigned setgraphbufsize(unsigned bufsize);

ᄒ

스캔이나 floodfill을 위해사용되는 버퍼의 크기를 변경하기 위해 사용 한다. 버퍼의 크기는 4K로 정해져 있는데 650개의 꼭지점을 가진 다각형을 채우기에 충분한 크기이다.

버퍼가 모자라면 늘릴 수 있으며 필요없을 때에는 작게하여 메모리를 절약할 수 있다.

히

void restorecrtmode(void);

ᄒ

그래픽상태를 텍스트 상태로 바꾼다. 다시 그래픽 모드로 돌아가기 위해서는 initgraph()를 사용하는것이 아니고 getgraphmode()를 사용한다.

서론

우리가 프로그램을 작성할 때 파일을 만든다던가 파일의 속성을 알아 본다던가 하는 일이 필요할 때 가 있다.

이럴때는 C-언어에서 제공하는 함수를 사용하면 되는데 사실 파일을 만든 다던가 지우는등의 실질적인 일은 함수가 하는것이 아니다. 실질적인 적은`도스`가 맡아서 하게 되며 C-언어에서 제공하는 함수들은 단지 도스와 사용자를 이어주는 끈에 불과한 것이다.

이제 도스가 제공하는 함수들을 배우려고 한다. 다소 어려운 면도 있지만 프로그램을 스스로 작성해 보려고 한다면 반드시 거쳐야 할 관문이다

1. 도스명령어 만들기

도스를 이해하기 위해서 가장 좋은 방법은 도스에서 제공하는 명령어들

을 직접 우리가 만들어 보는 것일 것이다.
도스를 자체를 만들어 보자는 것이 아니고 명령어 해석기(COMMAND.COM)
에서 제공하는 COPY명령이나 DIR명령등을 만들어 보자.

1.1 COPY 명령

copy명령을 모르는 사람은 아마 없을 것이다. copy명령은 파일을 복사하
는 명령이다.
copy명령을 만들기 위한 순서를 알아보자.

তl

  읽을 파일을 연다.
  읽을 파일을 읽는다.
  쓸 파일을 연다.
  쓸 파일에 내용을 쓴다.

ত

읽을 파일을 여는 경우에는 open(O_RDONLY | O_BINARY); 를 사용하면
될 것이고 파일을 읽어들일 때는 read(f , 읽어들일곳 , 읽어들일양);
과 같이 사용하면 될 것이다.

그런데 한가지 문제가 있다. 파일을 읽는다고 했는데 어디에다 읽을 것
인가 ? 읽어서 어딘가는 보관해야 한다. 아래와 같이 해볼까 ?

```
ㅎl
      char buf[512];
      int f;

      f = open("source",O_RDONLY | O_BINARY);
      read(f,buf,512);
ㄷ
```

잘 않될것이다. 아니 프로그램상에 버그가 있는 것은 아니지만 문제가 있다. 파일을 512바이트씩 읽어들이므로 만약에 1메가 바이트의 큰 파일을 복사하려고 한다면 (1,048,576 / 512 = 2,048) 2048번을 읽고 쓰고 해야 한다.

2048번을 읽고 쓰고 하려면 상당한 시간이 걸린다. 그런데 실제로 일부의 프로그램에서 (copy명령이 필요한 인스톨 프로그램) 이런 복사방식을 사용하는 경우도 있다.

대개 인스톨이란 플로피디스크에서 하드디스크로 옮기는 것이 대부분 인데 이런 방식을 사용하면 엄청나게 느려지게 된다.

물론 프로그램이 느리게 돌아가서 좋은점은 아무것도 없다. 게다가 한가지 문제점이 또 있다. 위에서 buf로 선언한 공간은 함수 안에서 auto형 변수로 선언되었는데 auto형 변수면 사용이 끝난후 메모리에서 해제되어서 안심이지만 만약에 static으로 선언 되었다면 사용이 끝난 후에도 메모리에 그대로 남아있게 된다.

즉 프로그램의 다른 부분이 사용할 수 있는 메모리의 양이 그만큼 줄어들게 된다는 뜻이다.

이 모든 문제를 메모리를 동적으로 할당하여 사용 함으로써 해결할 수 있다.

```
    void malloc   (size_t size);
    void free      (void *block);
```

위의 두 함수가 가장 메모리의 동적 사용을 위해서 기본적으로 사용되는 함수이다.
실제로 메모리를 할당하는 것은 다음과 같이 하면된다.

`포인터 = malloc(필요한 바이트 수);`

필요한 바이트 수는 64K를 넘으면 않된다. 정 64K를 넘는 공간을 사용하고 싶으면 메모리를 여러번 나누어서 사용하는 것이 좋다. 도스상에서 프로그램에서 돌려지는 메모리가 대개 600K이하 인걸보면 한번에 할당할 수 있는 64K는 결코 작은양은 아니다.

메모리의 사용이 끝나면 메모리를 해제 시켜주어야 한다.

`free(포인터)`

따라서 메모리의 동적 할당을 사용하는 프로그램은 다음과 같은 형태가 될 것이다.

```
     p = malloc(1024); ........ 메모리 할당
     명령 1 .................. 메모리를 사용하는 작업
     명령 1 .................. 메모리를 사용하는 작업
     free(p); ................ 메모리 해제
```

이제 copy명령에서 동적 할당을 사용하면 속도를 증가시킬 수 있다. 그런데 문제는 아직도 있다.

파일이 64K를 넘는 경우에는 어떻게 해 주어야 하는가 ?

이럴때는 파일을 나누어서 여러번 일고 쓰면 된다. 즉 버퍼(할당한 메모리)가 32K일 때는 복사할 파일의 크기를 32K로 나누어서 몫의 수 만큼 32K를 쓰고 나머지 만큼 파일에 쓴 나머지를 쓰면된다.

이야기가 어려운가 ?

즉 복사하고자 하는 파일의 크기가 fsize이고 버퍼의 크기가 bufsize일 때 다음과 같이 작성하면 된다.

```
  for (i=0 ; i < (fsize/bufsize) ; i++)
  {
     read(source,buf,bufsize);
     write(dest,buf,bufsize);
  }
  read(source,buf,fsize%bufsize);
  write(dest,buf,fsize%bufsize);
```

위의 방법이 최상의 방법이라고는 말하고 싶지 않다. 생각하기에 따라 얼마든지 쉬운 방법이 나올 수가 있고 바로 그런일이 프로그래머가 하는 일 이기 때문이다.

이제 실제로 프로그램을 작성해 보자.

```
     #include <fcntl.h>
     #include <stdio.h>

     main()
     {
```

```
        int source,dest,i,j,k;
        void *buf;
        long bufsize,fsize;
        char sourcef[64],destf[64];

        printf("Input file name to COPY\n");
        printf("(Example) c:\\himem.sys a:\\himem.sys\n");
        scanf("%s %s",sourcef,destf);
        source = open(sourcef,O_RDONLY | O_BINARY);
        dest = open(destf,O_WRONLY | O_BINARY);

        if (source==-1)  {
ㅎ

ㅎI
          printf("can't open SOURCE %s",sourcef);
          exit(1);
        }
        if (dest==-1)  {
          printf("can't open TARGET %s",destf);
          exit(1);
        }

        malloc(buf,32 * 1024);
        for (i=0 ; i < (fsize/bufsize) ; i++)
        {
          read(source,buf,bufsize);
          write(dest,buf,bufsize);
        }
        read(source,buf,fsize%bufsize);
        write(dest,buf,fsize%bufsize);
        free(buf);
ㅎ

ㅎI
        close(source);
        close(dest);
        printf("Success !!\n");
      }
ㅎ
```

한가지 집고 넘어가야 할 것이 있는데 무었이든지 끝 마무리가 중요하다는 것이다.

만일 프로그램에서 malloc()함수를 사용하며 메모리를 할당 받았다면 사용이 끝난 후에는 반드시 free()함수로 메모리를 해제 시켜주어야 한다. 그리고 open()함수로 파일을 열었으면 파일 작업이 끝난 후에는 반드시 close()함수로 파일을 닫아야 한다.

이것을 잊어서는 않된다. 간혹 이작업을 잊었을 경우 프로그램이 다운되는 현상이 생기는데 이런 이유에서 생기는 버그는 찾기가 매우 힘들다.

1.2 DEL 명령

del명령은 파일을 지우는 명령이다. copy명령에 비해서는 비교적 구현이 쉬운편 인데, 왜냐하면 파일을 열 필요도 메모리를 할당할 필요도 없기 때문이다. 단지 파일의 이름을 받아서 지우기만 하면된다.

ㅎI

```
    int unlink  (const char *path);
```

ㅎ

위의 함수 unlink()는 파일을 지우는 역할을 한다. 사용법도 간단하여 그냥 unlink(지울파일); 하면 된다. 그러나 읽기전용 파일이나 와일드카드가 있으면 않된다.

읽기전용 파일을 지우려면 chmod()나 _chmod()를 호출하여 파일에서 읽기전용 속성을 제거한 후 작업을 진행하면 된다.

unlink()는 파일이 성공적으로 삭제되면 0을 리턴한다. 에러가 발생하면 -1을 리턴한다.

자 대충 설명이 끝난것 같으니 프로그램을 작성해 보자.

ㅎI

```
    #include <fcntl.h>
```

```
#include <stdio.h>
#include <dos.h>            ..... unlink()를 사용하기 위해 필요함

main()
{
  int i,j,k;
  char f[64];

  printf("Input file name to DELETE\n");
  printf("(Example) c:\\temp.bak\n");
```

ㅎ

ㅎl

```
  scanf("%s",f);
  if (unlink(f)==-1)
  {
    printf("can't DELETE %s",f);
    exit(1);
  }
  printf("Success !!\n");
}
```

ㅎ

프로그램은 무척 간단하다. 주의할 것은 이 프로그램을 시험한다고 아무 파일이나 지워서는 않된다는 것이다.

그리고 unlink()함수를 사용하기 위해서는`dos.h`파일이 필요하다는 것도 잊지 말자.

1.3 REN 명령

ren명령은 파일의 이름을 변경하는 명령어이다. 이 역시 del명령과 같이 무척 간단하다.

ㅎl

```
int rename(const char *oldname, const char *newname);
```

ㅎ

함수의 사용법도 간단해서 rename(바꿀 파일이름, 새이름); 하고 해주면 된다.

에러가 없으면 rename()은 0을 리턴한고 에러가 있으면 -1을 리턴한다.
실제 프로그램도 무척 간단하다.

```
#include <fcntl.h>
#include <stdio.h>

main()
{
```

```
   int source,dest,i,j,k;
   char sourcef[64],destf[64];

   printf("Input file name to RENAME\n");
   printf("(Example) c:\\tree.com a:\\tree.exe\n");
   scanf("%s %s",sourcef,destf);
   i = rename(sourcef,destf);

   if (i==-1)  {
     printf("Dupliced file name or file not found.");
     exit(1);
   }
   printf("Success !!\n");
}
```

unlink()함수와는 다르게 remove()함수는`stdio.h`파일에 정의되어 있다

1.4 TYPE 명령

type명령은 지금까지의 다른 명령들과는 조금 다르다. 왜냐하면 지금까지는 모든 명령을 대신할 함수가 있었지만 type명령은 그렇지가 않기때문이다.

하지만 그렇게 어렵지 않다. 사실 간단하다. 파일을 읽어들여서 화면에 표시만 해주면 되는 것이다.

```
#include <fcntl.h>
```

```
#include <stdio.h>

main()
{
   int source,dest,i,j,k;
   char f[64];
   FILE *F;


   printf("Input file name to TYPE\n");
   printf("(Example) c:\\tree.com\n");
   scanf("%s",f);
   F = fopen(f,"rb");

   if (F==NULL)  {
      printf("file not found.");
      exit(1);
   }

   while(!feof(F)) putch(fgetc(F));
   fclose(F);
}
```

1.5 DIR 명령

dir명령은 지금까지 배운것중 가장 어려운 관문이 될것이다. 실제로 저자도 프로그래밍을 배우면서 처음 배운지 1년이 지나서야 파일의 목록을 다룰 수 있게 되었다.

물론 여러분은 더 빠르게 배우길 바란다. 사실 알고보면 별로 어려운 것도 아니다.

우선 파일목록을 제어하기 위해서는`dir.h`파일이 필요하다. dir.h에는 파일 목록을 제어하기 위해 다음과 같은 함수와 데이타가 정의되어 있다

ㅎI

```
    int findfirst(const char *path, struct ffblk *ffblk,
                  int attrib);
```

ㅎ

findfirst()함수는 path에 해당하는 첫번째 파일을 찾는 역할을 한다.
path는 파일의 경로로 쓰여질 수 있으며 파일이름으로 입력될 수 도 있으나 대개 와일드 카드문자를 사용한다.
찾은 파일은 인자로 사용하는 ffblk에 입력되게 되는데 ffblk는`dir.h`에 다음과 같이 정의되어 있다.

ㅎI

```
    struct   ffblk    {
            char              ff_reserved[21];  예약공간
            char              ff_attrib;        속성
            unsigned          ff_ftime;         생성시간
            unsigned          ff_fdate;         생성날자
            long              ff_fsize;         파일크기
            char              ff_name[13];      파일이름 };
```

ㅎ

attrib는 찾으려고 하는 파일의 속성을 적는다. 파일속성 상수는`dos.h`에 다음과 같이 정의되어 있다.

ㅎI

```
    #define FA_RDONLY        0x01   읽기전용
    #define FA_HIDDEN        0x02   숨김속성
    #define FA_SYSTEM        0x04   시스템 속성
    #define FA_LABEL         0x08   레이블
    #define FA_DIREC         0x10   디렉토리
    #define FA_ARCH          0x20   저장속성
```

ㅎ

attrib는 특별한 속성의 파일을 찾는다던가 하는데 사용될 수 있다.
예를들어

ᇹ
```
findfirst("*.*",ffblk,FA_DIREC | FA_RDONLY);
```
ᇺ
위의 명령은 디렉토리와 읽기전용 파일만을 찾는다. 모든 파일을 찾고자 할 때에는 '|'연산자를 사용할 수 도 있으나 속성상수를 모두 합한 값인 '0x3f`를 사용하여도 된다.

findfirst()는 path와 일치하는 파일을 찾으면 0을 리턴하고 파일을 더 이상 찾지 못하거나 에러가 발생하면 -1을 리턴한다.

그러나 findfirst()만으로는 모든 파일을 찾을 수가 없다. findfirst()는 첫번째 파일만을 찾는것일 뿐 모든 파일을 찾기 위해서는 findnext()와 함께 사용하여야 한다.

ᇹ
```
int findnext(struct ffblk *ffblk);
```
ᇺ
보다시피 인자에는 path나 attrib가 없다. 단지 파일의 정보를 받아들일 ffblk만 써주면 된다.
path나 attrib는 바로 전에 사용된 findfirst()의 것을 사용한다.

findnext()는 path와 일치하는 파일이 있으면 0을 리턴한다. 더이상의

파일이 발견되지 않거나 에러가 있을경우 -1이 리턴되며 전역변수인 `errno`은 다음값 중의 하나가 된다.

`ENOENT`   파일 이름이 합당치 않음
`ENMFILE` 더이상의 파일이 없음

```
#include <stdio.h>
#include <dir.h> ... findfirst(),findnext()함수를 사용하기 위해
main()
{
  struct ffblk ffblk;
  int done;
  printf("Directory list of *.*\n");
  done = findfirst("*.*",&ffblk,0);
  while(!done)
  {
    printf("%s\n",ffblk.ff_name);
    done = findnext(&ffblk);
  }
}
```

1.6 디스크 빈공간 구하기

파스칼을 보면 디스크의 총공간과 빈공간을 알려주는 함수가 있다. 물론 C-언어에도 있지만 그대로는 쓸 수 없고 약간 손을 보아야 한다.

```
void getdfree(unsigned char drive, struct dfree *dtable);
```

위의 함수 getdfree()는 드라이브를 지정하여 주면 드라이브의 총공간과 빈공간을 알려준다. 인자로 주어지는 dtable은 다음과 같이 정의 되어있다.

```
struct  dfree  {
        unsigned df_avail;     사용가능 클러스터
```

```
            unsigned df_total;      총 클러스터
            unsigned df_bsec;       섹터당 바이트 수
            unsigned df_sclus;      클러스터당 섹터 수
    };
```
ㅎ

따라서 df_avail * df_bsec * df_sclus 하면 빈 공간이 나오고 df_total * df_bsec * df_sclus 하면 총 공간이 나온다. getdfree()에 인자로 주어지는 drive는 0이번 사용하고 있는 드라이브, 1=A, 2=B, 3=C 순으로 나간다.

ㅎI
```
    #include <stdio.h>
    #include <dir.h>
    #include <dos.h>
    main()
    {
      struct dfree dtb;
      int done;
      long totals,frees;

      getdfree(0,&dtb);
      totals = (long)dtb.df_total * dtb.df_bsec * dtb.df_sclus;
```
ㅎ

ㅎI
```
      frees = (long)dtb.df_avail * dtb.df_bsec * dtb.df_sclus;

      printf("Total : %ld MB\n" , totals/1048576);
      printf(" Free : %ld MB\n" , frees/1048576);
    }
```
ㅎ

왜 수를 1,048,576으로 나누었는지 의아해 할 사람이 있을지 모르겠다. 1,048,576은 바이트를 메가 바이트로 바꾸기 위해 나누어 주는 값이다.

ㅎI
```
  1 MByte  =   1024 KByte   =   1048576 Byte
```
ㅎ

2. system()함수

system() 함수는 도스 명령어를 C-언어 프로그램에서 직접 쓰기 위한 함수 이다.
사용법은 사용하고자 하는 도스명령어를 system()에 넣어주기만 하면 된다. 즉, dir/w명령을 사용하고자 한다면,

`system("dir/w");`

와 같이 해주면 된다.
지금까지 배운 모든것을 이 system()함수로 해결할 수 있다. 심지어 프로그램 내에서 다른 프로그램을 실행시키는 것도 가능하다. 이 system() 명령을 사용하여 간단한 쉘 프로그램을 만들어 보도록 하자.

ㅎI

```
#include <stdio.h>
#include <dos.h>

char *command[]=
{
        "dir/w","tree c:",
        "format A:","format B:",
        "diskcopy A: A:","diskcopy A: B:",
        "quit"
};
void printfcommand()
{
  int i;
  for (i=0;i<7;i++)
  printf("<%d> %s\n",i,command[i]);
```

```
    }
ㅎ


ㅎl
    main()
    {
      int i;
      while(1)  {
        printfcommand();
        i=getch();
        switch(i)
        {
          case '0':case '1':
          case '2':case '3':
          case '4':case '5':
                    system(command[i-'0']);
                    break;
          case '6': exit(0);
        }
      }
    }
ㅎ
```

위의 예제는 system()함수로 도스 내부 명령어는 물론 다른 프로그램도 실행 시킬 수 있다는 것을 보여준다.
자신이 많이 사용하는 도스명령어를 모아 프로그램을 개조해도 좋겠다.

system()함수가 이렇게 많이 사용되지만 되도록 이면 램상주 프로그램은 실행 시키지 않는것이 좋다.

쉘 프로그램에서 램상주 프로그램을 실행시키는 않는것은 일종의 묵계인데 이 묵계를 깨뜨리면 컴퓨터가 조용하게 반항을 한다. (다운됨)

ཀillu1200.dat

서론

바이오스(BIOS)란 컴퓨터의 롬(ROM)칩에 내장되어 있는 일종의 프로그램으로서 과거에 어셈블리로 프로그램을 작성하던 시절에는 없어서는 안될 것이었다.

프로그래머가 프로그램의 만들때 모든 것을 다 작성하여야 한다고 생각해 보자. 화면에 문자열을 표시해 주는 루틴이라던가 키보드를 읽어들이는 루틴, 그리고 화면모드를 바꾸는 루틴, 귀찮다고 생각하기 이전에 하드웨어 전문가가 아니면 아무도 프로그램을 작성하려 하지 않을 것이다.

그래서 바이오스를 사용하였다. 바이오스는 프로그래머가 필요한 기본적인 루틴들을 제공한다. 때로 너무 기본적이어서 사용이 불편하기도 하지만.

하여튼 C-언어와 같은 좋은 언어를 사용하는 지금에도 바이오스에 대해서를 알아둘 필요가 있다.
바이오스에서 제공하는 대부분의 루틴을 C-언어에서 제공하지만 사실 그렇지 않은것이 더 많고 하드웨어에 대
해서 공부할 수 있는 좋은 기회가 되
기 때문이다.

사실 전 단원인 도스와 바이오스를 잘
이해하지 못한다면 뛰어난 프로그래머
라고는 말할 수 없다.

왜냐하면 소프트웨어에만 의존하는 프
로그램은 반드시 한계가 있기 때문이다
좀더 나은 프로그램을 만들기 위해서도
하드웨어에 대한 이해는 필수이다.

ཀillu1300.dat

1. 인터럽트(Interrupt)

인터럽트란 말은 "끼어들다." "가로채다." 라는 말이다. 인터럽트는 여러가지 프로그래밍 테크닉 중에서도 고급 테크닉에 속하는데 이유는 그만큼 이해하기도 어렵고 또 실제로 사용하는데도 애로가 따르기 때문이다.

인터럽트를 사용하는 가장 대표적인 프로그램들로는`램상주 프로그램`을 들수 있다.

램상주 프로그램은 다른 프로그램으로 작업을 하는 도중에 인터럽트가 걸린(Hooked) 바이오스 서비스가 호출되면, 예를 들어 키보드 서비스인 9번에 걸려 있다면 키보드의 특정한 키가 눌려지면 램에 상주하고 있던 프로그램이 활동을 시작하게 된다.

illu1301.dat

바이오스를 설명하는 도중에 왜 인터럽트를 설명하는지 이해가 되지 않는 사람이 있을지 모르겠다. 이런 사람들을 위해 설명하자면 바이오스의 서비스는 각기 번호로 기능에 따라 분리되어 있으며 몇몇 서비스는 인터럽트에 관련된 서비스 이다.

예를 들어 인터럽트 9번은 키보드 인터럽트이고 8번은 시계 인터럽트 이다. 키보드가 눌려질 때마다 인터럽트 9번이 실행되며 이 서비스는 키보드에서 문자를 읽어들여 메모리에 저장하고 메모리가 꽉 차면 "삐-익"소리를 내보낸다.

시계 인터럽트는 1초에 약 18번 자동으로 실행된다. 따라서 자신의 램상주 프로그램을 8번 인터럽트에 걸어놓으면 램상주 프로그램은 1초에 18번 실행된다,

어쨌든 우리가 배우려는 바이오스의 서비스는 각기 번호가 매겨져 있다. 이번호는 1바이트로 구성되므로`(0 ~ 255)`이론적으로 256개의 서비스가 존재한다.

그러나 시스템에 예약되어 있는 서비스도 있고, 사용하지 않고 비어있는 서비스도 있다.

이중에 우리가 배우려고 하는 서비스는 몇개 되지 않는다. 그러나 한개의 번호가 매겨져 있는 서비스라 해서 기능이 한가지 인것은 아니다. 예를 들어 인터럽트 10번은 비디오 시스템에 관련된 서비스인데 화면의 모드를 바꾸고, 커서의 위치를 바꾸고, 화면에 점을 찍는등 여러가지 일을 한다.

바이오스를 배우는 방법으로는 기본적으로 어셈블리를 기초로 하면 좋겠지만, 어셈블리에 익숙하지 않는 사람들을 위해 C-언어에서 바이오스를 인터페이스 하는 방법을 먼저 알아보도록 하겠다.

C-언어 내부에서 바이오스를 사용하는데는 보통`int86()`함수를 사용한다.

ㅎl

```
int int86(int intno, union REGS *inregs, union REGS *outregs);
```

ㅎ

인수 intno에 의해 지정된 소프트웨어 인터럽트를 실행한다. 그런데 아직 배우지 않은것이 나왔다. 공용체인`REGS`에 관한 것이다.

공용체`REGS`는 dos.h에 다음과 같이 정의되어 있다.

ㅎl

```
    struct WORDREGS {
            unsigned int     ax, bx, cx, dx, si, di, cflag, flags; };
    struct BYTEREGS {
            unsigned char    al, ah, bl, bh, cl, ch, dl, dh; };
    union    REGS     {
        struct   WORDREGS x;
        struct   BYTEREGS h;
    };
```

ㅎ

위에서는 구조체와 공용체를 사용하여 일반 데이타로 선언되어 사용되지만 사실 REGS란 하드웨어에서 데이타를 운반하는 레지스터를 의미한다. 가장 기초적인 정보 운반 수단이므로 인터럽트 서비스와 같이 하드웨어에 밀착된 곳에서는 이 레지스터를 사용한다.

그러나 C-언어가 어셈블리가 아닌이상 프로그램 내에서 직접 레지스터를 다룰 수는 없고 레지스터 처럼 데이타를 선언하여 사용하는것 이다.

사실 레지스터에 관한 내용은 어셈블리 전문 서적을 아무리 찾아보아도 쉽게 찾아볼 수 없을것이다. 왜냐하면 그 만큼 전문적인 내용이기 때문이다. 물론 어셈블리 단원에서 자세히 다루게 되겠지만 지금까지 설명한 내용만 이라도 어느정도 이해한다면 일단은 성공이다.

IBM-PC (8088, 80286, 80386)에서 공통적으로 사용되는 범용 레지스터에

는 다음과 같은것들이 있다.

ㅎI
워드형 레지스터 AX, BX, CX, DX, SI, DI
바이트형 레지스터 AH,AL, BH,BL, CH,CL, DH,DL
ㅎ

그외에도 세그먼트, 포인터 레지스터와 플래그 레지스터라는 것이 있으나 이 장의 범위를 벗어나므로 어셈블리 단원에서 자세히 배워보도록 한다.

바이트형 레지스터는 공용체라는 말이 의미하듯이 각 워드형 레지스터를 2분하여 사용하는 레지스터이다. 즉 AX는 AX로 사용할 수 도 있으며 AH와 AL로 사용할 수 도 있다. 여기서 AH는 AX의 상위 바이트 이며 AL은 하위 바이트 이다.

BX나 CX, DX도 같은 방법으로 상위 바이트 레지스터와 하위 바이트 레지스터를 모두 사용할 수 있다. 그러나 SI, DI 레지스터는 그렇게 사용할 수 없다.

어쨌든 인터럽트 서비스를 사용하는데 정보 전달 수단으로는 레지스터를 사용한다.

C-언어에서 함수가 정보 전달 수단으로 메모리(인수)를 사용하는데 비하면 꽤 까다롭다 할 수 있다.

illu1301.dat

2. 10번 인터럽트 (비디오 서비스)

10번 인터럽트는 비디오에 관한 서비스로 그 덩치가 상당히 크다.
10번 인터럽트가 거의 대부분의 그래픽 루틴을 포함하고 있는데도 프로

그래머들이 즐겨 사용하지 않는 이유는 속도가 느리다는데 있다.

이것은 롬의 속도가 램보다는 필연적으로 느리기 때문에 당연한 현상이지만 많은 프로그래머들이 이 이유 때문에 왠만하면 바이오스를 쓰려하지 않는다.

그러나 꼭 써야만 하는 경우도 있다. 즉, 바이오스에만 존재하는 기능을 사용하려고 하는 경우이다.

지금 부터 배우는 인터럽트 10번의 서브함수들에 대해서 너무 집착할 필요는 없다.   왜냐하면 C-언어를 배워서 텍스트 모드에서 돌아가는 명함관리 프로그램이나 주소관리 프로그램 등을 만들려고 한다면 굳이 어려운 인터럽트를 배울 필요가 없기 때문이다. (쓰지 않으므로)

그러나 자신의 프로그램에서 멋진 스크롤을 하고 싶다거나 하드웨어에 대해서 몹시 배우고 싶다면 배우는 것이 좋을것이다.

당연한 이야기지만 인터럽트 10번은 어셈블리로 되어있으며 AH레지스터에 함수 번호를 입력 시켜서 BIOS가 제공하는 여러 비디오루틴 중의 하나를 선택한다.

BIOS루틴으로 매개 변수를 넘겨주려면 INT 0x10를 수행하기 전에 그 변수의 값을 정해진 레지스터에 입력시킨다.
BIOS루틴에서 리턴되는 값 역시 레지스터 담겨서 리턴된다.

ㅎI

[주의]

일반적으로 BIOS 비디오 입출력 루틴은 데이타의 타당성 검사를 하지 않으며 상태 코드나 에러를 리턴하지도 않는다. 그러므로 유효하지도 않은 비디오 버퍼의 번지를 엑세스 한다던지, 비디오 모드가 제공하지 않은 비디오 페이지를 선택한다던가 하는 프로그램을 작성하는것은 절대 금물이다. BIOS루틴은 이러한 오류를 전혀 감지해 내지 못하기 때문이다.

ㅎ

2.1 0번 서비스 (화면모드 선택)

이 서비스는 화면 모드를 선택하는 서비스 이다. BGI함수의 initgraph() 처럼 그래픽 모드로만 전환하는것도 아니고 restorecrtmode()처럼 텍스트모드 로만 변환하는것도 아니다.

0번 서비스에서 사용할 수 있는 화면모드는 다음과 같다.

ㅎI

AH = 0
AL = 비디오 모드 번호

모드번호	해상도	칼라 수	모드/특성
0	40 x 25	16칼라	텍스트 모드
1	40 x 25	16칼라	텍스트 모드
2	80 x 25	16칼라	텍스트 모드
3	80 x 25	16칼라	텍스트 모드
4	300 x 200	4칼라	그래픽 모드

ㅎ

ㅎI

모드번호	해상도	칼라 수	모드/특성
5	300 x 200	4칼라	그래픽 모드

6   640 x 200   2칼라   그래픽 모드
7   80 x 25   단색   허큘리스 텍스트 모드
8   160 x 200   16칼라   그래픽 모드 (PCjr 전용)
9   300 x 200   16칼라   그래픽 모드 (PCjr 전용)
0x0A   640 x 200   4칼라   그래픽 모드 (PCjr 전용)
0x0D   320 x 200   16칼라   그래픽 모드 (EGA,VGA 전용)
0x0E   640 x 200   16칼라   그래픽 모드 (EGA,VGA 전용)
0x0F   640 x 350   단색   그래픽 모드 (EGA,VGA 전용)
0x10   640 x 350   16칼라   그래픽 모드 (EGA,VGA 전용)
0x11   640 x 480   단색   그래픽 모드 (MCGA,VGA 전용)
0x12   640 x 480   16칼라   그래픽 모드 (VGA 전용)
0x13   320 x 200   16칼라   그래픽 모드 (MCGA,VGA 전용)
ㅎ

위에 나열된 모드중 우리가 익히 알고 있는 모드는 몇개 되지 않는다.

ㅎl
[1]   3번   80 x 25   16칼라   텍스트 모드
[2] 0x12번   640 x 480   16칼라   그래픽 모드 (VGA 전용)
[3] 0x13번   320 x 200   16칼라   그래픽 모드 (MCGA,VGA 전용)
ㅎ
[1]번은 우리가 가장 흔히 볼 수 있는 도스화면과 같은 16칼라 텍스트모드이다.
[2]번은 VGA그래픽 카드에서 가장 많이 사용되는 그래픽 모드이다. 우리가 initgraph(&GD,&GM,"");을 사용하면 자동으로 이 모드로 선택되며 우리의 C-홀로서기도 이 모드를 사용한다.
[3]번은 정겨운 모드로 게임에서 가장많이 사용하는 모드이다. 해상도는 떨어지지만 게임에서는 많은 칼라가 필요하므로 이모드를 주로 쓴다.

서비스 0번을 사용하여 화면 모드를 선택하는 방법은 레지스터 AH에 서비스 번호 0을 넣고 바꾸고자 하는 화면모드 번호를 AL에 넣어 인터럽트 10번을 실행하는 것이다.

ㅎl

```
void set_to_text()
{
  union REGS r;
  r.h.ah=0;
  r.h.al=3;          /* 80x25 16칼라 텍스트 모드 */
  int86(0x10,&r,&r);
}
```

ᄒ

위의 함수 set_to_text()는 화면을 텍스트 모드로 변환할 것이다.

BGI에는 게임에서 볼 수 있는 320x200 모드에 대한 루틴이 없다. 만약 게임을 작성하려고 한다면 이 서비스 0번이 없이는 불가능 할 것이다.

2.2 1번 서비스 (텍스트 커서 크기 조정)

ᄒI

```
레지스터 사용

AH = 1  /* 서비스 번호 */
CH = 커서의 상부
CL = 커서의 하부
```

ᄒ

텍스트 모드에서 프로그래밍을 하다보면 커서의 크기를 바꿔야 할 필요성을 느낀다. 예를 들어 커서를 화면에서 보이지 않게 한다거나 커서를 좀더 크게 하는 등의 일이다.

일반적으로 텍스트 모드는 한글자의 높이가 16이므로 CH와 CL의 범위는 0에서 16까지의 값이 될 수 있다.

예를 들어 CH의 값을 2로 하고 CL의 값을 15로 하면 옆의 그림과 같이 한글자를 거의 다 채우는 커다란 커서가 만들어지게 된다.

만일 커서를 화면에서 없어지게 하려면
CH에 0x20의 값을 주면된다.

상당히 많은 프로그램에서 커서의 모양을 변형시키거나 화면에서 보이지 않게 하는데, 주의할 점은 프로그램이 종료한 후에는 원래의 커서모양으로 복귀시켜주어야 한다는 점이다.

만일 프로그램에서 커서를 없애고, 원래의 모양으로 복귀하지 않고 도스상태로 돌아가게 되면 커서없는 도스를 보게 될 것이다.

illu1303.dat

```
#include <stdio.h>
#include <dos.h>
main()
{
  union REGS r;
  r.h.ah = 1;
  r.h.ch = 2;
  r.h.ch = 15;
  int86(0x10,&r,&r);
}
```

위의 예제는 커서모양을 커다랗게 만들어 준다. 만약 커서모양이 마음에 들지 않는다면 도스상에서 다음과 같이 하기 바란다.

`C:\DOS>mode co80 <Enter>`

2.3 2번 서비스 (커서의 위치 조정)

```
레지스터 사용

AH = 2  /* 서비스 번호 */
BH = 비디오 페이지
DH = 커서의 상부
DL = 커서의 하부
```

ㄸ

위의 함수는 터보-C의 gotoxy()함수를 생각하면 이해하기 쉬울 것이다.
텍스트 화면상의 커서를 옮기는 서비스 이다.

BH에 입력되는 비디오 페이지는 VGA 텍스트 모드 에서는 거의 사용되지 않으며 따라서 신경쓸 필요가 없다.

2.4 3번 서비스 (커서의 상태 리턴)

ㄸl

```
    레지스터 사용

    AH = 3  /* 서비스 번호 */
    BH = 비디오 페이지
    [출력]
    CH = 커서의 상부 크기
    CH = 커서의 하부 크기
    DH = 커서위치(행)
    DL = 커서위치(열)
```

ㄸ

위의 함수는 커서의 크기와 위치를 알려주는 함수이다. 한번에 여러가지 정보를 주므로 유용한 함수라 할 수 있다.

터보-C 에서는 커서의 크기를 알려주는 함수가 없고 커서의 위치를 알기 위해서`wherex()와 wherey()`함수를 각기 사용하여야 하는점을 생각하면 참 편리한 서비스라 할 수 있다.

ㄸl

```
    int get_cursor_status(int *hsize , int *lsize ,
                          int *xpos , int *ypos)
    {
      union REGS r;
      r.h.ah=3;
      r.h.bh=0;
      int86(0x10,&r,&r);
      *hsize = r.h.ch
```

```
        *lsize = r.h.cl
        *xpos = r.h.dh
        *xpos = r.h.dl
    }
```
ㅎ
3번 서비스를 이용하여 커서의 상태를 알려주는 함수를 만들어 보았다.
자신의 프로그램에 한번 사용해 보는것도 좋을듯 하다.

ㅋ]4번 서비스 (라이트 펜)
ㅋ]5번 서비스 (비디오 페이지 선택)

여러분은 라이트 펜을 본적이 있는가 ?

아마 본적이 없을것이다. 디스켓 낭비
를 막기위해서 4번 서비스는 설명을 피
하겠다.

하까도 설명했듯이 텍스트 모드에서는
비디오 페이지가 별 의미가 없다. 따라
서 5번 서비스도 생략

ㅋillu1304.dat

2.5 6번 서비스 (위로 스크롤)

ㅎl

레지스터 사용

AH = 6 /* 서비스 번호 */
AL = 스크롤될 문자행의 수
BH = 속성
CH = 좌상귀 행
CL = 좌상귀 열
DH = 우하귀 행

DL = 우하귀 열
ট

BH에 저장되는 속성은 스크롤된 구역의 아랫 부분을 공백으로 채울때 사용된다. 텍스트 모드에서는 이 속성이 일반적인 형식을 취한다. 즉, 상위 바이트는 배경속성을 하위 바이트는 전경속성을 나타낸다. 그래픽 모드에 따라서 BH에 입력되는 속성이 다르다.

MCGA의 640x200 2칼라모드, 320x200 4칼라모드, 640x480 2칼라 모드에서는 BH에 입력되는 값이 1 바이트 픽셀 패턴을 나타낸다.
2칼라모드 에서는 1바이트에 8개의 픽셀이 들어가며 4칼라모드 에서는 4개 16칼라모드 에서는 2개의 픽셀이 들어간다.

AL의 스크롤될 행 수를 0으로 입력시키면 전체 화면이 공백으로 나타난다. 즉, 화면을 지우는 효과를 나타낸다.

টI
```
#include <stdio.h>
#include <dos.h>

main()
{
  union REGS r;
  r.h.ah=6;
  r.h.al=1;
```
ট

টI
```
  r.h.bh=7;         /* 속성 */
  r.h.ch=0;         /* x1 */
  r.h.cl=0;         /* y1 */
  r.h.dh=24;        /* x2 */
  r.h.dl=79;        /* y2 */
  int86(0x10,&r,&r);
}
```
ট

위의 예제는 현재의 화면을 위로 한칸 스크롤 시킨다. 응용 프로그램에

서 볼수 있는 텍스트 화면이 갈라지거나 하는 것도 이 예제 프로그램을 변형하면 가능하다.

2.6 7번 서비스 (아래로 스크롤)

ᄒI
```
    레지스터 사용

    AH = 7  /* 서비스 번호 */
    AL = 스크롤될 문자행의 수
    BH = 속성
    CH = 좌상귀 행
    CL = 좌상귀 열
    DH = 우하귀 행
    DL = 우하귀 열
```
ᄒ

아래로 스크롤 한다는 점만 다를뿐 위에 설명한 6번 서비스와 모든 사항이 같다.

2.7 8번 서비스 (커서위치의 문자와 속성읽기)

ᄒI
```
    레지스터 사용

    AH = 8  /* 서비스 번호 */
    BH = 비디오 페이지
    [리턴값]
    AH = 속성
    AL = ASCII 코드
```
ᄒ

이 서비스는 현재 커서가 위치하고 있는 위치의 있는 문자의 속성과 ASC

II코드를 리턴한다.

2.8 0x0B번 서비스 (잉여주사 칼라 지정)

ъI

```
레지스터 사용

AH = 0x0B  /* 서비스 번호 */
BH = 0
BL = 잉여주사 칼라값
```

ъ

CGA와 MCGA에서 BH=0 이면 BIOS는 BL에 저장되어 있는 값 중 하위 다섯 비트를 컬러선택 레지스터(Color Select) 레지스터(0x03D9)로 옮겨준다

우리가 보는 화면에는 모이는 부분 밖의 칼라도 지정해 줄 수 있는데,
디폴트로 검정색이 지정되어 있기 때문에 마치 없는것 같아 보인다.

0x0B번 서비스를 이용하면 이 보이는 부분밖의 칼라를 지정해 줄 수 있
다.

ъI

```
#include <conio.h>

main()
{
  union REGS r;
  r.h.ah = 0x0b;
  r.h.bh = 0
  r.h.bl = BLUE;
  int86(0x10,&r,&r);
```

```
    }
```
ㄷ

위의 예제는 잉여주사 칼라를 파란색으로 바꾸어 줄 것이다.

2.8 0x0C번 서비스 (putpixel())

ㄷI
```
    레지스터 사용

    AH = 0x0C  /* 서비스 번호 */
    AL = 픽셀 칼라
    BH = 비디오 페이지
    CX = x 좌표
    DX = y 좌표
```
ㄷ

이 서비스는 putpixel()함수를 생각하여 보면 쉽게 이해가 될 것이다.
화면의 좌표와 칼라를 입력하면 화면에 점을 찍는 서비스이다.
어찌보면 BIOS차원에서는 유일하게 지원하는 그래픽 서비스이다.

320 x 200, 256칼라모드를 제외한 다른 그래픽 모드에서는 AL에 입려된 값중 상위 1 비트가 셋트되어 있으면 AL의 값은 XOR연산에 의해 버퍼에 그려진다.

이 서비스는 어떤 그래픽 모드에서도 사용될 수 있다.

ㄷI
```
    void put_pixel(int x,int y,int color)
    {
      union REGS r;
      r.h.ah = 0x0c;
      r.h.al = color;
      r.x.cx = x;
      r.x.dx = y;
```

```
    int86(0x10,&r,&r);
  }
```

위의 함수는 화면에 점을 찍는 함수를 만들어 본 것이다.

2.9 0x0D번 서비스 (getpixel())

```
  레지스터 사용

  AH = 0x0D  /* 서비스 번호 */
  AL = 픽셀 칼라
  CX = x 좌표
  DX = y 좌표
  [리턴값]
  AL = 픽셀값
```

이 서비스는 getpixel()함수를 생각하면 쉽게 이해할 수 있다.
EGA의 320 x 200 4칼라 모드에서는 BH에 지정된 비디오 페이지
가 무시된다.

IBM의 EGA BIOS(9/13/84 버전)의 0x0D서비스에는 버그가 숨어있다. IBM-EGA의 IBM EGA의 350 주사선 그래픽 모드에서는 AL에 리턴되는 값이 부정확하다.

BIOS루틴의 짝수 번지는 짝수 비트플레이인에 연결되고 홀수 번지는 홀수 비트플레인에 연결된 매핑구조를 계산에 넣지 않은 상태에서 비디오 버퍼내의 바이트 오프셋을 계산하고 있는것이 분명하다.

```
  int get_pixel(int x,int y)
  {
    union REGS r;
    r.h.ah = 0x0d;
    r.x.cx = x;
    r.x.dx = y;
    int86(0x10,&r,&r);
```

```
    return (r.h.al);
  }
```
ᄒ

2.10 0x0F번 서비스 (현재 비디오 상태 리턴)

ᄒl
```
  레지스터 사용

  AH = 0x0F  /* 서비스 번호 */
  [리턴값]
  AH = 화면에 보이는 한 행 내의 문자 갯수(80 또는 40)
  AL = 비디오 모드 번호
  BH = 액티브 비디오 페이지
```
ᄒ

이 서비스는 유용하게 사용될 수 있는 BIOS서비스의 하나이다. 바로 비디오의 상태를 알려주는 것이다.

AH에 입력되는 한 행 내의 문자 갯수는 텍스트 모드에서만 사용될 수 있다. 우리가 사용하는 텍스트 모드는 대개 80x25 또는 40x25이므로 80 또는 40의 값이 리턴된다.

AL에 입력되는 값은 현재 사용되고 있는 비디오 모드를 알려준다.
BH에 입력되는 현재 사용되고 있는 액티브 페이지를 리턴하는 것으로서 BGI함수에도 없는 것이다.
ᄒl
```
  int video_info(int *col,int *mode,int *active)
  {
    union REGS r;
    r.h.ah = 0x0f;
    int86(0x10,&r,&r);
    *col    = r.h.ah;
    *mode   = r.h.al;
    *active = r.h.bh;
```

```
    }
```

이상으로 개괄적인 인터럽트 10번 서비스에 대한 설명이 끝났다. 여기에 나와있는 내용도 초보자를 대상으로 하는 C-홀로서기의 범위에서 벗어난 것이지만 사실 시스템 BIOS에 대한 이해는 전문 프로그래머 에게는 필수라 할 수 있다.

예를들어 여러분 주위에 프로그래밍을 잘 아는 사람이 있으면 한번 물어보라 "인터럽트 10번 서비스의 서브함수 0번이 뭐하는 거더라 ?" 만약 그 사람이 "그게 뭔데 ?" 하고 되묻는다면 이제 당신은 그 사람을 앞설 수 있다.

illu1305.dat

3. 0x13번 인터럽트 (디스크 서비스)

0x13번 서비스는 디스크 서비스 이다. 그렇다고 해서 파일을 만들거나 디렉토리를 이동시키는등 고차원적인 서비스가 아니고 디스크의 섹터를 읽어들이거나 포맷을 하는등 아주 저차원적이고 기본적인 서비스 들이다

인터럽트 13번 역시 10번 못지 않게 상당히 크다. 서브 함수들을 하나하나 모두 설명할 수는 없으므로 대충 중요하다고 생각되는 부분만 집고 넘어가겠다.

3.1 디스크란 무엇인가 ?

어지껏 하드웨어에 대해 그리 자세히 배워본 적이 없으므로 13번 인터럽트를 배우는 이 참에 디스크에 대해서 자세히 알아보기로 하자.

하드 디스크이건 플로피 디스크이건간에 모든 디스크들은 똑같은 방식으로 구성되어 있다. 디스크의 표면에는 `트랙(Track)`이라 불리우는 여러개의 동심원이 있는데 그 트랙은 다시 원을 따라서`섹터(Sector)`라는 단위로 나누어 진다.

예를 들어 5.25인치 2HD 플로피 디스크의 경우 80트랙을 가지며 트랙당 15섹터를 가진다. 그리고 모든 디스크는 한 섹터에 512바이트를 가지므로 다음의 계산으로 지스크의 용량을 쉽게 구할 수 있다.

```
512바이트 * 15섹터 * 80트랙 * 2면 = 1.2메가 바이트
```

보통 하나의 파일은 여러개의 섹터로 나누어져 기록된다. 디스크에 바깥쪽에 위치한 몇 개의 섹터들은 특수한 목적으로 사용되도록 지정되어 있다.

그외에는 비어있는 바깥쪽 섹터부터 먼저 사용된다. 이 말은 디스크에 데이타가 들어 찰수록 디스크의 중심쪽을 향하여 섹터들이 점진적으로 채워짐을 의미한다.

파일이 지워지면 그에 할당되었던 섹터들은 자유롭게 된다. 따라서 디스크를 오래 사용하다 보면, 사용 가능한 섹터들이 디스크의 여기저기에 흩어져서 존재 하게 된다.

illu1306.dat

이렇게 되면 새로이 만드는 파일의 섹터도 여기저기로 흩어지게 되어 이에 따라 디스크에 읽고 쓰는 시간이 더 걸리게 된다.

하드 디스크는 몇가지 특별한 특징을 가지고 있다. 하드 디스크는 보통 2개 이상의 평평한 디스크들로 구성되는데, 각각의 디스크는 양면을 읽을 수 있는 한 쌍의 헤드를 가지고 있다.
이때 디스크 전체에 대해 중심으로 부터 같은 거리에 위치하는 모든 트랙을`실린터(Cylinder)`라고 부른다.

이 디스크들의 헤드는 모두 나란이 움직이므로 헤드가 디스크의 안쪽으로 움직이기 전에 한 실린더 내의 모든 트랙들을 채우는 것이 작동거리면에서 보다 경제적이다.

한편 실린더의 모임은 서로 다른 오퍼레이팅 시스템(OS)을 위해 할당될 수 있다. DOS의`FDISK`프로그램이 이 기능을 수행하는데 이 프로그램은 하드 디스크를 각각 크기를 조절할 수 있는 4개의 부분으로 변환한다.

이러한 이유로 디스크의 내용(Specification)은 매우 크게 변할 수 있다

디스크 섹터는 디스크를 포맷하는 유틸리티에 의해 쓰여진 자기적인(magnetic)정보에 의해 정의된다.
이 정보는 각 섹터에대한 등록 번호를 가지고 있다. 섹터들에 대한 BIOS 번호는 디스크의 용량에 따라 1~8, 1~9, 1~15 등이 있다.

트랙은 자기적으로 표시되지 않는 대신, 헤드가 읽고 쓰기 작동을 하는 디스크 바깥쪽 가장자리로 부터 옵셋으로서 기계적으로 정의된다.

3.2 int86x()함수

13번 서비스에 대해서 배우기 전에 먼저 int86x()함수에 대해서 배워야 한다. 10번 서비는 int86()함수를 사용하여도 충분히 사용이 가능하였으나 13번 서비스는 그렇지가 못하다.

int86x()함수의 사용법은 아래와 같다.

```
    int int86x(int intno, union REGS *inregs, union REGS *outregs,
               struct SREGS *segregs);
```

처음의 3개의 인자는 같은데 나머지 한개의 인자가 추가 되었다. 이것이 int86()함수와 다른 점이다.

나머지 하나의 인자형인 SREGS는 dos.h에 다음과 같이 정의되어 있다.

```
    struct   SREGS   {
        unsigned int     es;
        unsigned int     cs;
        unsigned int     ss;
        unsigned int     ds;
    };
```

사실 이 장의 윗부분에서 레지스터에 대한 설명을 할 때 세그먼트 레지스터에 대한 설명은 빼고 진행했는데 이 SREG는 세그먼트 레지스터 이다

세그먼트 레지스터는 프로그램에서 사용되는 메모리를 대표하고 있는 레지스터인데 각각의 레지스터 마다 사용되는 곳이 다르다.

ㅎl

```
ES (Extre segment)          DS와 함께 데이타를 처리하는데 쓰인다.
CS (Code segment)           프로그램의 코드에 관계한다.
SS (Stack segment)          스택에 관계한다.
DS (Data segment)           프로그램의 데이타에 관계한다.
```

ㅎ

물론 어셈블리 단원으로 가면 좀더 자세히 배우게 되겠지만 일단은 각
레지스터들의 쓰임만 알아두자.

3.3 2번 서비스 (섹터읽기)

ㅎl

```
레지스터 사용

AH = 0x02  /* 서비스 번호 */
DL = 드라이브 번호
DH = 헤드 번호
CH = 실린더(하드 디스크), 트랙(플로피 디스크)
CL = 섹터번호
AL = 읽어들일 섹터의 수
```

ㅎ

디스크를 섹터 단위로 읽는다. 여러 개의 섹터를 읽으려면 그 섹터들은
같은 트랙, 같은 면에 있어야 한다. 그 이유는 BIOS가 한 트랙에 몇개
의 섹터가 있는지 모르기 때문이다.

BIOS는 몇개의 섹터를 읽고나서 다음 트랙을 읽어야 할 지를 모른다.
보통 이 기능은 한 섹터를 읽는데 쓰인다.

3.4 3번 서비스 (섹터쓰기)

তা

레지스터 사용

AH = 0x03 /* 서비스 번호 */
DL = 드라이브 번호
DH = 헤드 번호
CH = 실린더(하드 디스크), 트랙(플로피 디스크)
CL = 섹터번호
AL = 기록할 섹터의 수

ত

사용 방법및 사용 레지스터는 2번 서비스와 동일하다.

3.5 4번 서비스 (섹터확인)

তা

레지스터 사용

AH = 0x04 /* 서비스 번호 */
DL = 드라이브 번호
DH = 헤드 번호
CH = 실린더(하드 디스크), 트랙(플로피 디스크)
CL = 섹터번호
AL = 검사할 섹터의 수

ত

디스크의 섹터를 점검한다. 디스크의 내용과 메모리의 내용이 같은지 검사하는게 아니고 섹터가 있는지 검사하고 읽어보는 방법과 CRC라는 검사를 한다. CRC는 데이타에 대해 패리티 검사를 해서 에러가 있으면 찾아내는 작업이다.

3.5 5번 서비스 (디스크 포맷)

তা

레지스터 사용

AH = 0x04 /* 서비스 번호 */
DL = 드라이브 번호
DH = 헤드 번호
CH = 실린더(하드 디스크), 트랙(플로피 디스크)
AL = 트랙당 섹터의 수

ত

디스크 한면의 한 트랙을 포맷한다. 기능 3과 동일한 레지스터를 사용하나 CL은 사용하지 않는다. 포맷은 한 트랙씩 하기 때문에 섹터단위의 포맷은 할 수 없지만 섹터에 대해 어떤지정을 해줄 수 있다.

4. 0x16번 인터럽트 (키보드 서비스)

0x16번 서비스 역시 가장 많이 사용되는 BIOS 서비스 중의 하나이다. 터보-C의 입력함수인 getch()계열의 함수에 별로 만족하지 못하는 사람은 직접 바이오스를 이용해서 프로그램에 이용하는 것도 좋은 방법이 될 수 있다.

0x16번 함수에도 몇가지 용도에 따라 종류가 있는데 하나하나 살펴보기로 한다.

4.1 0번 서비스 (키보드 읽기)

তা

레지스터 사용

AH = 0 /* 서비스 번호 */
[출력]
AH = 스캔코드
AL = 아스키코드

ত

스캔코드를 사용하는 프로그램은 이 서비스를 사용하면 좋다. 스캔코드는 아스키 코드와는 달리 각 키보드마다 하나씩만 설정되어 있기 때문에 각 키를 모두 따로 정의하여 사용하려면 스캔키를 이용하여야 한다.

그리고 키패드에는 아스키 코드가 정의되어 있지 않으므로 키패드의 키를 사용할 때도 스캔코드를 사용하여야 한다.

4.2 1번 서비스 (키보드 검사)

তI

레지스터 사용

AH = 1 /* 서비스 번호 */
[출력]
읽혀진 키가 있을때
AH = 스캔코드
AL = 아스키코드

ত

4.3 2번 서비스 (상태 바이트 얻기)

তI

레지스터 사용

AH = 2 /* 서비스 번호 */

[출력]
AL = 상태 바이트

대부분의 키보드가 스캔코드를 가지고 있지만 Shift, Ctrl, Alt 키등은 스캔코드가 없다. 즉 키가 눌리워도 알아낼 방법이 없다.

이럴때는 2번 서비스를 사용하면 된다. AL에 넘어오는 상태 바이트의 각 비트를 검사하여 이런 키들이 눌리웠는지 알 수 있다.

비트	내용	상태
7	<Ins> 키가 눌리워 졌는지의 상태	1=눌림
6	<CapsLock> 키가 눌리워 졌는지의 상태	1=눌림
5	<NumLock> 키가 눌리워 졌는지의 상태	1=눌림
4	<ScrollLock> 키가 눌리워 졌는지의 상태	1=눌림
3	<Alt-Shift> 키가 눌리워 졌는지의 상태	1=눌림
2	<Ctrl-Shift> 키가 눌리워 졌는지의 상태	1=눌림
1	왼쪽 <Shift> 키가 눌리워 졌는지의 상태	1=눌림
0	오른쪽 <Shift> 키가 눌리워 졌는지의 상태	1=눌림

서론

우리가 아무리 프로그래밍을 잘 안다 하더라도 컴파일러 사용법을 모르면 프로그램을 작성할 수가 없다.

이말은 프로그래밍 전문자라면 컴파일러에 대해서도 어느정도 지식을 갖추고 있어야 한다는 이야기가 된다. 이 장에서는 저자가 주로 사용하는 에디터의 사용법과 컴파일러와 링커의 사용법을 배워보도록 하겠다.

1. 컴파일러의 종류

아직까지도 자신이 사용할 컴파일러를 정하지 못했는가 ? 대개는 터보-C 2.0이나 Borland C++ 아니면 터보-C++을 사용할 것이다.

아직도 컴파일러를 결정하지 못한 사람은 새로운 컴파일러를 구입할 때 자신의 시스템 사양을 참작해야 한다. Turbo-C 2.0은 XT에서도 잘 돌아가지만 Borland C++ 이나 Turbo C++ 컴파일러는 자체가 386이상이 아니면 실행이 되지 않거나 2메가 이상의 메모리를 필요로 한다.

또 자신의 주머니 사정도 고려해야 한다. Borland C++은 많은 기능을 자랑 하지만 가격이 조금 비싸서(60 만원대) 주머니 사정이 빈약한 사람에게는 권하고 싶지 않다.

새로운 객체지향 프로그래밍을 지원하면서도 터보 C++ 2.0 ,3.0 은 비교적 저렴한 가격에 구입할 수 있다. (20만원대)

만약 학생신분 이라면 아카데미 버전을 구하면 보다 저렴한 가격에 컴파일러를 구할 수 있을것이다.

터보 C 2.0은 옛날 버전이라 기능은 약간 빈약하나 사용에 불편은 없고
아주 싼가격에 구할 수 있다. 저자도 이 버전과 터보 C++ 3.0을 사용하
고 있으니 참고 바란다.

이 외에서 마이크로 소프트사의 MS-C 6.0 이나 Quick C 컴파일러를 사용
하는 사람도 있으리라 본다.
마이크로 소프트사의 이들 컴파일러는 만들어내는 코드의 질 면에서 볼
랜드사의 컴파일러를 앞서나 많은 사람에게 사용되지는 않는다.

어찌보면 하나의 프로그램을 만들때 모두 한 사람이 만드는것은 아니다.
다른 프로그래머가 작성해 놓은 라이브러리나 소스를 사용해야 할 경우
가 많은데 MS-C를 사용할 경우 이런 점에서 제약이 따른다.

결국 최후의 선택은 여러분 손에 달려 있으므로 신중히 생각해서 현명한
선택을 하기 바란다.

1. 프로그래밍의 순서

하나의 프로그램을 만드는 데는 모두 3가지의 과정을 거친다.

첫째 : 소스파일을 작성한다. 이 작업
은 에디터나 텍스트파일을 제공하는 워
드 프로세서에서 하면 된다.

둘째 : 컴파일을 한다. 만들어진 소스
파일을 가지고 컴파일러를 사용해서 컴
파일을 한다.
대개의 초보자들은 이 과정 만으로 프
로그램이 만들어 지는줄 알고 있는데

사실은 하나의 과정이 더 있다.

링크 : 컴파일러에 의해서 만들어진 중
간 코드를 역어서 하나의 프로그램을
완성한다.

illu1400.dat

1.1 소스 파일 작성

위에서 소스파일은 에디터로 한다고 했는데, C-언어를 배우기 전에 베이
식 만을 배운 사람은 에디터가 무었인지도 모를것이다.
만일 아직 사용해본 에디터가 없거나 사용하고 있는 에디터가 아직 손에
익지 않았다면 Q-에디터의 사용을 권한다.

Q-에디터는 에디터가 갖추어야 할 기능을 모두 갖추 었으면서도 속도가
상당히 빠르다. `전문 프로그래머는 곧 에디터 전문가`라는 말이 있는데
이는 프로그래밍에 있어서 에디터의 중요성을 대변한다 할 수 있다.

Q-에디터는 쉐어웨어 이므로 통신이나 주위의 사람들에게서 쉽게 얻을수
있다.

1.2 Q-에디터 사용법

Q-에디터의 실행파일은`Q.EXE`이다. 도스상에서 실행시킬 때에는 "q" 라
고만 해주면 된다.
Q-에디터는 기본적으로 팝업 풀다운 방식의 메뉴를 사용하고 있으므로
사용상의 어려움은 없을 것이다. 팝업 풀다운 메뉴를 부르는 방법은 Q-에
디터를 사용하는 도중에 <Esc>키를 누르면 된다.

팝업 풀다운 메뉴는 기본적으로 아래와 같이 구성되어 있다.

File   Window   Block   Search   Print   Macro   Editing   Other   Quit

1.2.1 File-메뉴

Load - 파일을 로드한다.
메뉴선택 후 파일의 이름을 타이핑 하거나 와일드 카드
를 입력한다음 파일을 골라도 된다.
File - 파일을 저장하고 메모리에서 지운다.
Save - 파일을 저장한다.
Quitfile - 파일을 저장없이 메모리에서 지운다.
Next - 메모리에 있는 다음파일을 편집한다.
Prev - 메모리에 있는 전파일을 편집한다.
Read - 파일을 읽어들여 삽입한다.
Changename - 파일의 이름을 바꾼다.
Writeblock - 블록으로 정해진 구간을 파일로 저장한다.
Globalfile - 모든 파일을 저장하고 메모리에서 지운다.
Dos - 도스명령어를 수행한다.
프로그램도 실행 시킬 수 있다.

1.2.2 Window-메뉴

Close - 열려진 윈도우를 닫는다.
Split - 파일을 윈도우로 나눈다.
Next - 다음 윈도우의 파일을 편집한다.
One - 나눠진 윈도우를 하나의 윈도우로 만든다.
Prev - 전 윈도우의 파일을 편집한다.
Zoom - 화면상에 작게 보이는 윈도우를 확대한다.
Resize - 현재 화성화 되어있는 윈도우의 크기를 변경한다.

1.2.3 Block-메뉴

Mark Line - 라인단위로 블록을 설정한다.
Mark Character - 문자단위로 블록을 설정한다.
Mark Kolumn - 컬럼단위로 블록을 설정한다.
cOpy - 블록을 복사한다.
cUt - 블록을 잘라둔다.
Delete - 블록으로 설정된 구간을 지운다.
Unmark - 블록 설정을 해제한다.

1.2.4 Search-메뉴

Find - 편집중인 파일 안에서 찾고자 하는 문자열을 찾는다.
Replace - 편집중인 파일 안에서 찾고자 하는 문자열을 찾아 대치 하고 싶은 문자열로 대치한다.
Again - Find나 Replace작업을 계속한다.

ҕJ 1.2.5 Print-메뉴

Print all - 파일의 모든 부분을 프린트 한다.
Print block - 블록으로 설정된 구간만을 프린트 한다.
send Formfeed - 프린터로 폼 피드를 보낸다.
set print Left margin - 왼쪽 여백을 설정하다.
set print Right margin - 오른쪽 여백을 설정하다.
set print Top margin - 위쪽 여백을 설정하다.
set print Bottom margin - 아래쪽 여백을 설정하다.

1.2.6 Macro-메뉴

Macro record - 매크로를 정의한다.
Read macro - 파일에 저장되어 있는 매크로를 불러들인다.
Write macro - 정의된 매크로를 파일로 저장한다.

ᇹ1.2.7 Editing-메뉴

Addline - 현재 커서의 밑에 한 라인을 삽입한다.
Deleteline - 한 라인을 삭제한다.
Insertline - 현재 커서의 위치에 한 라인을 삽입한다.
Joinline - 두 라인을 합친다.
Splitline - 현재 위치에서 라인을 끊는다.

1.3 명령어 라인 컴파일러, 통합환경(IDE)

볼랜드사의 컴파일러의 특징을 들라면 통합환경의 지원을 들 수 있다. 볼랜드사에서 나온 컴파일러들은 모두 통합환경을 지원하는데 이는 초보자들에게는 매우 반가운 일이 아닐 수 없다.

그러나 통합 환경 자체를 배우는데도 상당한 시간이 걸릴뿐 아니라, 통합 환경이 가지는 여러가지 문제 (메모리, 호환성 결여)들로 인하여 대부분의 프로그래머들은 통합환경 대신 명령어 라인 컴파일러를 사용한다.

여러분도 통합환경만 사용하였다면 지금이라도 명령어 라인 컴파일러를 사용하기 바란다.

처음 배우기에는 약간 황당해 보이지만 일단 익숙해지면 훨씬 편하고 모든 작업이 자유롭다.

어떻게 보면 이 단원 자체가 명령어행 컴파일러의 사용법을 배우기 위한 단원이고 통합환경을 배우더라도 언젠가는 명령어행 컴파일러를 사용해야 하므로 시간절약 측면에서도 처음부터 명령어 행 컴파일러를 사용하는 것이 현명한 일이다.

1.3 컴파일 작업

에디터로 소스파일을 만들면 컴파일 작업을 하여야 한다. 컴파일은 터보-C 사용자는 TCC.EXE 파일을 볼랜드 C++ 사용자는 BCC.EXE 를 사용해야 한다.
한가지 주의해야 할 일은 터보-C 에서는 소스파일의 확장자가 *.C 이지만 볼랜드 C++ 에서는 *.CPP 라는 점이다. 이점 착오 없기 바란다.

তl
    TCC <소스파일>
ত

소스파일을 명시하지 않고 그냥 TCC만을 쳐주면 아래와 같이 컴파일 작업에 필요한 옵션들이 나온다.

```
Turbo C  Version 2.0  Copyright (c) 1988 Borland International
Syntax is: TCC [ options ] file[s]

-1      80186/286 Instructions
-A      Disable non-ANSI extensions
-B      Compile via assembly
-C      Allow nested comments
-Dxxx   Define macro
-Exxx   Alternate assembler name
-G      Generate for speed
-Ixxx   Include files directory
-K      Default char is unsigned
-Lxxx   Libraries directory
```

```
-M      Generate link map
-N      Check stack overflow
-O      Optimize jumps
-S      Produce assembly output
-Uxxx   Undefine macro
-Z      Optimize register usage
-a      Generate word alignment
-c      Compile only
-d      Merge duplicate strings
-exxx   Executable file name
-f     *Floating point emulator
-f87    8087 floating point
-gN     Stop after N warnings
-iN     Maximum identifier length N
-jN     Stop after N errors
-k      Standard stack frame
-lx     Pass option x to linker
```

ㄸ

ㄸㅣ

```
-mc   Compact Model      -mh   Huge Model
-ml   Large Model        -mm   Medium Model
-ms  *Small Model        -mt   Tiny Model

-nxxx  Output file directory
-oxxx  Object file name
-p     Pascal calls
-r    *Register variables
-u    *Underscores on externs
-v     Source level debugging
-w     Enable all warnings
-wxxx  Enable warning xxx
-w-xxx Disable warning xxx
-y     Produce line number info
-zxxx  Set segment names
```

ㄸ

위의 옵션들중 자주 사용되는 옵션들에 대해서 알아보자

`-1     80186/286 Instructions`
위의 옵션은 AT를 위한 코드를 발생시키는 명령이다. 보통은 XT에서도 실행 시킬 수 있는 코드를 만들어 내는데 이 옵션을 사용하면 실행파일의 크기도 작어지고 속도도 조금 빨라진다.

`-G     Generate for speed`
속도를 증가시킨 코드를 만들어 낸다. 이 옵션을 사용하면 실행속도면에도 조금 보강된 코드를 만들어 낸다. 반면 실행파일의 크기는 조금 커지게 된다. 디폴트는 실행파일을 작게 만들도록 되어있다.

`-Ixxx  Include files directory`
헤더파일이 들어 있는 디렉토리를 지정한다. 이것이 지정되어 있지도 않으면 헤더파일 include문에서 에러가 발생된다.

`-N        Check stack overflow`

프로그램을 작성하다보면 작성한 프로그램이 스택이 넘쳐서 다운되는 버그가 많다. 이럴때는 이 옵션을 사용하면 스택이 넘치는 시기를 알려준다.   그러나 평상시에 이 옵션을 사용하면 실행파일의 크기가 늘어나고 속도도 떨어지게 되므로 사용하지 않는것이 좋다.

`-mc      Compact Model             -mh      Huge Model`
`-ml      Large Model               -mm      Medium Model`
`-ms     *Small Model               -mt      Tiny Model`

위는 메모리 모델을 설정하는 옵션이다. 메모리 모델은 프로그램이 사용할 데이터의 양과 프로그램의 크기를 생각해서 설정해야 한다.
대부분의 프로그램은 large 모델을 사용하니 참고하기 바란다.

`-p        Pascal calls`

함수의 호출방식과 인자전달 방식에 파스칼 식을 사용한다. 속도가 빨라지고 실행파일의 크기가 감소하는 대신 C-언어의 자유로운 함수 사용은 제약을 받게 된다.

만약 컴파일시 아래와 같이 에러가 발생한다면,

ㅎl

```
C:\TC>TCC -ml exam <Enter>

Turbo C  Version 2.0  Copyright (c) 1988 Borland International
exam.c:
Error exam.c 6: For statement missing ; in function main
Error exam.c 6: Statement missing ; in function main
*** 2 errors in Compile ***
Available memory 353308
```

ㅎ

6번째 라인에 에러가 발생했음을 알 수 있다. 그러면 exam.c 파일을 살펴 보도록 하자.

ㅎl

```
 1: #include <stdlib.h>
 2: main()
 3: {
 4:    int i,j,k;
 5:
 6:    for (i=0:i<5;i++)        ----> 이 부분에 에러가 있음
 7:    printf("%d",i);
 8: }
```

ㅎ

6번째 라인의 for문에서 ";"가 사용될 부분에 ":"가 사용 됬음을 알 수 있다. 그러면 에러가 발생한 부분을 수정한 다음 다시 컴파일을 하여보자.

ㅎl

```
  C:\TC>TCC exam <Enter>
  Turbo C  Version 2.0  Copyright (c) 1988 Borland International
  exam.c:
          Available memory 349678
ㆆ
```

위에서 처럼 에러가 발생하지 않으면 아무런 메시지도 나타나지 않는다. 그러면 실행파일이 만들어 졌는지 dir명령으로 확인해 보자.

```
ㆆl
    C:\TC>dir exam.* <Enter>

    exam.c                 91    9-09-93   9:30p
    exam.obj              273    9-09-93   9:30p
             364 bytes in 2 file(s)
      11,091,968 bytes free
ㆆ
```

어 ! 이게 어찌된 일인가 실행파일인 *.EXE 파일은 만들어 지지 않고 처음보는 exam.obj 파일이 생성되었다.

여기서 확장자가 *.OBJ인 파일은 중간코드 파일로서 컴파일을 하면 생기능 파일이다. 이 파일을 사용하여`링크`라는 작업을 하면 드디어 실행파일이 만들어 지게 된다.

이제 링크(Link)라는 작업에 대해서 알아보도록 하자.

1.4 링크(Link)

소스파일을 컴파일 하면 확장자가 *.OBJ인 중간 코드가 생성 된다고 하

였다. 그런데 실제로 커다란 프로그램을 만들때는 한개의 파일로만 만드는 것은 아니다. 왠만한 프로그램은 1만 라인을 훨씬 넘으므로 한 파일에 담아두기에는 역부족이다.

그래서 크 프로그램을 작성할 때는 파일을 여러개로 나누게 되는데 여러개로 나누어진 파일을 각자 컴파일 하면 각 *.OBJ 파일이 생긴다.

이때 이 여러개의 *.OBJ파일을 모아서 한개의 실행파일을 만드는 것이 링커가 하는 일이다.

illu1401.dat

즉 모듈화된 여러개의 *.OBJ파일을 묶어주는 것이 링커가 하는일이다.
사전을 찾아보면 링크는 "잇다, 연결고리"의 뜻을 가지고 있다.

이 링커는 어셈블리나 다른 컴파일러에서는 거의 대부분 사용되는데 파스칼에서는 링크의 개념이 없다. 그래서 전에 파스칼을 배웠던 사람들은 다소 혼동스러울 것이다.

또 링커에서는 라이브러리 라는 개념을 알아 두어야 하는데 라이브러리란 여러개의 *.OBJ파일을 하나의 파일에 담아 놓은것이다.

터보-C로 작성된 프로그램은 링크시에 자체 라이브러리를 꼭 포함시켜야 한다.

자체 라이브러리에는 다음과 같은것들이 있다.

ㅎl
```
   matht, maths, mathc,
   mathm, mathl, mathh 각 메모리 모델별 산술 라이브러리
   emu                  8007 에뮬레이션, 코프로세서 없이 부동소숫점을
                       사용하려면 반드시 필요하다.
   graphics             그래픽 라이브러리, 그래픽을 사요하려면 반드시
                       필요하다.
   ct, cs, cc,
   cm, cl, ch           각 메모리 모델별 일반 라이브러리
```
ㅎ

터보-C로 작성된 프로그램을 링크할 때는 터보-C에 포함된`TLINK.EXE`를 사용한다. 도스상에서 TLINK의 사용법은 다음과 같다.

ㅎl
```
   TLINK <*.OBJ 파일>, <실행파일 이름>, <*.MAP 파일>, <*.LIB 파일>
```
ㅎ

*.MAP 파일이란 프로그램의 각 데이타와 함수의 번지를 알려주는 아스키 포맷으로 된 파일으로서 파일명을 입력하지 않으면 만들어 지지 않는다.

위에서 컴파일한 EXAM.OBJ 파일을 실행파일로 만들려면 다음과 같이 하면 된다.

ㅎl
```
      tlink exam+c0l, exam,, mathl+emu+graphics+cl
            ----        ----
```
ㅎ

링크될 때 같이 포함될 *.OBJ 파일과 *.LIB 파일을`"+"`로 묶어 줌을 알 수 있다. 위에서 c0l은 프로그램이 시작될때 각 프로그램의 초기화를 담당하는 부분이다.

이 역시 각 모델별 일반 라이브러리와 마찬가지로 항상 포함시켜 주어야 한다.

2. 배치파일(Batch)의 이용

지금까지 본 바와같이 exam.c 파일을 만들어서 exam.exe 파일을 만들려면 도스상에서 다음과 같이 타이핑 하여야 한다.

```
ㅎl
          large 모델
          _|_
     tcc -ml exam <Enter>
     tlink exam+c0l, exam,, mathl+emu+graphics+cl
                  -+-                                     -+-
                   |                                       |
               large 모델                              large 모델
ㅎ
```

프로그램을 한번 컴파일하고 링크할 때마다 위와 같이 타이핑 해 주어야 한다면 여간 귀찮은 일이 아닐 수 없다.
이때는 배치파일을 사용하면 간단하게 할 수 있다.

```
ㅎl
     C:\DOS>type cp.bat <Enter>
     tcc -ml %1
     tlink %1+c0l, %1,, mathl+emu+graphics+cl
```

ট

위는 배치파일의 의사변수`"%"`를 사용하여 실제로 배치파일을 작성한 예이다.
일단 배치파일이 만들어 지면 도스사에서`cp exam <Enter>`만 해주면 컴파일과 링크작업이 자동으로 진행된다.
서 론

객체 지향형 프로그램이란 `Object Oriented Program` 을 우리 말로 바꾸어 놓은 것으로서 약자로 `OOP`라고도 한다. 객체란 말은 아직까지는 우리들에게 생소한 낱말로서 간단하게 말해서 `자동차의 부품`과도 같다고 할 수가 있다.

여기에서는 `객체 지향 프로그램이란 무엇인가`에 대한 개략적인 것만 다루겠다. 좀더 자세한 사항은 관련 서적을 참고하기 바란다.

1. 객체 지향형 프로그램(Object Oriented Program)

우리가 일반적으로 알고 있는 `COBOL, PASCAL, FORTRAN, C 언어` 등과 같은 프로그래밍 언어들은 `프로세스 중심의 언어`라고 할 수가 있다. 이러한 프로세서 중심의 언어들은 데이타 중심의 언어와는 상반된 개념으로 어떠한 작업을 `프로세서 중심`으로 운영된다. 그러나 객체 지향형 프로그램은 `데이타(즉, 객체 중심)`으로 운영된다.

쉽게 말해서 프로세서 중심의 언어는 한 과정에서만 적용시킬 수 있다는

단점을 가지고 있으나, 객체 지향형 언어는 서로 다른 작업을 하는 프로그

이라도 동일한 데이타를 사용하면 모두 동일시 할 수 있다.

객체 지향형 언어로는 `C++, Objective-c, Smalltalk` 를 들 수 있다.

তl

객체 지향형 프로그램의 특징

ত

(1) 객체의 데이타를 집약하여 데이타의 산벌화를 방지한다(정보 은익)

(2) 상위 계층에서 사용한 것을 하위 계층에서도 전수 받아 사용할 수 있다(정보 계승).

- 코드의 재 사용과 분배
- 변수와 함수의 전수

(3) 한 연산자에 여러 가지 연산 기능을 수행할 수 있게 하여 프로그램 코드를 최소화한다(다양성).

2. C++ 언어의 개념

C++ 언어는 객체 지향 프로그램의 객체들을 정의할 수 있는 강력한 언어로 다음과 같은 많은 장점을 가지고 있다.

(1) C 언어의 모든 주위 환경을 수용하고 전달한다.

(2) 프로그램을 모듈적이고 구조적으로 강력하게 작성할 수 있도록 한 C 언어를 확장한 것이다.

(3) C 언어의 판독력 향상과 간단하고 명료한 프로그램을 작성할 수 있다.

(4) ANSI 의 표준 범례를 준수함으로써 발생되는 프로그램의 정확성을 향상 시킨다.

(5) 간단하고 명료한 프로그램으로 프로그래머의 작업 능률을 증진하고

프로그램의 수정과 보완 기능이 뛰어나다.

------------------------------- 계 속 -----------------------------

(6) 간단하고 명료한 코드들에 의한 프로그램의 실행이 뛰어나다.

(7) GUI(Graphic User Interface)를 지원한다(예 윈도우).

(8) 프로그램의 호환성과 이식성이 높다.

(9) 동적(dynamic)으로 프로그램이 운영된다.

(10) 혼합된 언어의 프로그래밍이 쉽다.

(11) 함수문으로만 구성되어 있다.

2.1 C++의 대표적인 구조에 대한 규칙

ㅎl

포함문(include)

ㅎ

```
#include<stdio.h>
#include<iostream.h>
#include"user.h"
```

ㅎl

매크로 정의

ㅎ

```
#define ESC  27
#define BYTE char
```

ㅎl

형 정의(typedef)

ㅎ

```
typedef struct complex COMPLEX;
```

------------------------------ 계　속 ----------------------------

তI

구조체 선언(struct)

ত

```
struct complex {
        double : real;
        double : image;
        };
```

তI

계층 선언

ত

```
class order {
       public :
       double order;
       };
```

তI

전역 변수 선언

ত

```
short x;
```

------------------------------ 계　속 ----------------------------

তI

함수 선언

ত

```
void f(short);
```

তI

기타

ত

```
void f(short x)              //함수 정의
{
  short i;                   //지역 변수 선언
  cout<<"Sub function..";  //함수 작업부
  printf("\n argument value is %d",x);
}
main()                        //메인 함수
{
  x=1;                        //전역 변수 사용
  cout<<"C++";
  printf("Function call...");
```

```
   f(x);
}
```

3. 객체의 정의

객체는 `형(type)`의 도움으로 정의할 수 있고, 객체를 정의하는 동안 관련되는 형은 메모리에 필요한 공간을 예약한다. 그러므로써 프로그래머는 객체와 관련된 메모리를 마음대로 조작할 수 있다.

ㅠ]3.1 객체의 정의(대표적인 형태)

(1) int x;
x 는 정수형 객체이므로 x 는 정수값 다체를 나타낸다.

(2) int x, y, x;
int x; int y; int x; 와 같은 객체 정의문의 반복을 줄인다.

(3) int* x; int *x;
x 는 정수형 객체에 대한 포인터형을 가지는 객체이다. 즉 정수형 포인터이다. 그러나 int *x, y;에서 x 는 정수형 포인터 객체이고, y는 일반 객체이다.

------------------------------ 계 속 ------------------------------

(4) int x[5];
x 는 5개의 정수형 객체들의 모임으로 이루어진 배열형 객체이다.
즉 x 는 5개의 정수를 가지는 배열문이다.

(5) int* x[5]; int *x[5]; int *(x[5]);
x 는 5개의 정수형 포인터를 가지는 배열형 객체를 의미한다.
int *(x[5]);에서 연산자 []는 연산자 * 보다 우선한다.

(6) int (*x)[5];
x 는 5개의 정수를 가지는 배열문에 대한 포인터형 객체이다. 즉 x 는 5 개의 정수에 대한 포인터이다.

(7) char x[10];
x 는 10개의 문자형 객체를 가지는 배열문형 객체이다. 즉 x 는 10개 문자를 가지는 배열문을 의미한다.

------------------------------ 계 속 ------------------------------

(8) char* x[10];   char *x[10];   char *(x[10]);

x 는 10개의 문자형 객체에 대한 포인터들로 구성된 배열문형을 가지는 객체이다. 즉, x 는 10개의 문자형 포인터를 가지는 배열문을 의미한다.

(9) char x[5][10];   char (x[5])[10];

x 는 10개의 문자로 이루어진 5개의 배열형을 가지는 객체를 의미한다. 즉, 5 * 10 구조를 가진 2 차원 배열문을 가지는 객체를 의미한다.

(10) int x();

x() 는 정수형 객체를 귀환값으로 주는 함수이다. 즉, x() 는 정수를 리턴값으로 가지는 함수이다. 그리고 x 는 함수 x()를 나타내는 포인터이다.

(11) char x();

x() 는 문자형 객체를 귀환값으로 주는 함수이다. 즉, x() 는 문자를 리턴값으로 가지는 함수이다. 그리고 x 는 함수 x()를 나타내는 포인터이다.

------------------------------ 계 속 ----------------------------

(12) int *x(); int* x();

x()는 정수형 포인터를 귀환값으로 주는 함수이다.

(13) float* x();   float *x();   float *(x());

x() 는 실수형 포인터를 리턴값으로 주는 함수이다. 연산자 "()"는 "*"에 우선한다.

(14) float (*x)();

(*x)() 는 실수를 귀환값으로 주는 함수이다. 그리고 x 는 실수를 귀환값으로 주는 함수에 대한 포인터이다.

(15) int x, *y;

이는 int x; 와 int *y;를 합해 놓은 것이다.

(16) void* x;

x 는 어떤 알 수 없는 형에 대한 포인터를 의미한다.

------------------------------ 계 속 ----------------------------

(17) void x();
x 는 어떠한 형이라도 귀환값으로 줄 수 있는 함수이다.

(18) char** x;
*x 는 문자형 포인터이며, x 는 문자형 포인터에 대한 포인터이다.

(19) int x(int x, char* y) { 함수 본체 }
x() 는 정수형 객체를 리턴값으로 주는 함수이며, 첫번째 인수로는 정수형 객체를, 두번째 인수로는 문자형 객체를 가진다.

(20) const char *x;
x 는 문자 상수 포인터형이다.

(21) const char *const x="constant chain";
x 는 상수형 문자 배열문에 대한 상수형 포인터이다. 즉, x 는 프로그램상에서 항상 "constant chain"만을 가지고 변경될 수 없다.

(22) const char* x="1234";
(21)과 그 내용이 같다.

[예제]// C++ Programming

```
      // 상수형 포인터를 가지는 struct 형 객체이다.
#include<stdio.h>
char* f();       // 함수 선언
char* g();
char* f();       // 함수 정의
{
  char* p;
  p="C++ Programming";
  return(p);
}

char* g()        // 함수 정의
{
  char* p;
  p="OK";
  return(p);
}
```

------------------------------ 계 속 ----------------------------

```
main()             // 메인 함수
{
   char *p1, *p2;
   p1=f();
   p2=g();
   printf("\n%s",p1);
   printf("\n%s",p2);
}
```

[결과]

```
Programming
OK
```

3.2 형 정의를 이용한 객체 정의

```
struct person {        //형 정의
          char name[30];
          int   age;
          };
```

(1) person x;
   x 는 person 형을 가지는 객체이다.

(2) person *x;
   x 는 person 형 객체에 대한 포인터이다.

[참고] struct 문이나 union 문이 일반 객체일 경우에는 직접 연산자(.),
포인터형일 경우에는 간접 연산자(->)로 메버를 참조한다.

[예제] // C++ Programming

```
#include<iostream.h>
struct person {             // 형 정의
        char name[20];
        int  age;
        };
void printing(person *); // 함수 선언
void printing(person *x);
{
  count<<(x->age);
}
main()                        // 메인 함수
{
  person p;
  p.age=30;
  printing(&p);             // 화면에 나이 30 출력
}
```

3.3 계층형 정의

```
class cl {
            .....   // class member ....
           };
```

(1) cl x; ---> x 는 cl 계층형을 가지는 객체이다.
(2) cl* x; ---> x 는 cl 계층형에 대한 포인터이다.

4. 객체의 선언

ᆩ4.1 객체의 선언

객체의 선언이란 이미 정의된 형(type)을 객체의 식별자와 연결시켜 주는 작업이다.

```
[예] char ch;  // char : 형
               // ch   : 객체의 식별자
```

이미 정의된 객체를 또 다른 소스 파일에서 사용할 수도 있다.

```
extern int x;
```

4.2 함수의 선언(function declaration)

(1) 함수 선언 : 함수의 프로토타입을 만드는 것을 의미한다.
(2) 함수의 프로토타입 : 함수형 및 함수 인수형을 선언하는 것이다.
(3) 함수의 정의 : 함수가 실행해야 하는 작업 명령문을 지정해 주는 것으로 함수 본체를 만드는 것이다.

ㅎl

함수의 선언을 이용하여 컴파일러가 하는 작업

ㅎ

(1) 함수 인수양을 제어하고 확인한다.
(2) 함수 인수형을 제어하고 확인한다.
(3) 함수 선언과 정의 사이의 동일 구조를 제어하고 확인한다.

[예제]

```
#include<stdio.h>
int f(float x)        //함수의 정의
{                     //인수의 자승을 계산하는 함수이다.
  return(x*x);
}
------------------------------ 계  속 ----------------------------

main()                //메인 함수
{
  float x;
```

```
   x=f(1.5);                //함수 호출
   printf("\n%f",x);   //1.5 * 1.5 의 계산값을 인쇄한다.
}
```

[설명] 함수의 선언(프로토타입)이 없이 작성된 것으로 컴파일은 아무런
에러없이 진행된다. 그러나 리턴하는 형과 리턴받는 형이 다르다.
그러므로 올바른 작업결과를 얻을 수 없을 수도 있다.

5. 객체의 사용 영역

(1) 소스 파일 또는 소스 모듈에서의 객체 사용(전역 객체)
  - 이 객체를 선언, 정의한 모듈만이 사용할 수 있다.
(2) 함수 또는 프로그램 블럭에서의 객체 사용(지역 객체)
  - 이 객체는 모든 프로그램에서 사용될 수 있다.
(3) 계층(class) 객체 사용 영역

5.1 전역적 객체(global object)

5.1.1 한 소스 파일에서만 사용이 가능한 객체

```
#include<iostream.h>
int x=1;                  //전역 객체
int f();                 //함수 프로토타입
int f()                  //함수 정의
```

------------------------------ 계 속 ----------------------------

```
{
   return (x+y);       //전역 객체 y 가 사용된 후에 선언되었으므로 에러
}                          //전역 객체 y 가 사용전에 정의되어야 한다.
```

```
int y=2;                //전역 객체
main()                  //메인 함수
{
   count<<f();
}
```

ㅎl
 5.1.2 여러 소스 파일에서 사용이 가능한 객체
ㅎ

[조건] 전역 객체의 조건
(1) 소스 파일에서 static 이 아니 automatic으로 정의할 것.
(2) 소스 파일에서 이 객체를 extern으로 선언할 것.

------------------------------ 계 속 ----------------------------

```
//첫번째 소스 파일 first.cpp
#include<stdio.h>
int x;
extern void f();
main()
{
   x=4;
   printf("\n this is first file 1%d",x);  //4 인쇄
}
//두번째 소스 파일 second.cpp
#include<stdio.h>
void f();
extern int x;
void f()
{
   x++;
   printf(" this is second file 1%d",x);   //5 인쇄
}
```

ㅎl
 5.1.3 한 소스 파일에서만 전역적으로 사용된 객체
ㅎ

이때는 첫번째 소스 파일에서 static형 전역 객체로 선언하면 된다.

```
//첫번째 소스 파일 first.cpp
```

```
#include<stdio.h>
static int x;
extern void f();
main()
{
   x=4;
   printf("\n this is first file 1%d",x);   //4 인쇄
}
```

------------------------------ 계 속 ----------------------------

```
//두번째 소스 파일 second.cpp
#include<stdio.h>
void f();
int x;                  //첫번째 소스 파일의 객체와는 이름만 같을
                         //뿐 아무런 관계가 없다.
void f()
{
   x++;
   printf(" this is second file 1%d",x);    //5 인쇄
}
```

5.2 지역적 객체(local object)

C++ 에서는 함수 안에서 지역적 객체와 같은 이름을 가지고 있는 전역적 객체를 연산자 '::'의 도움으로 사용할 수 있다.

```
#include<iostream.h>
int x=10;          //전역 객체
```

```
void f();          //함수의 프로토타입
main()
{
   f();
}
void f();          //함수 정의
{
   int x=1;         //지역 객체
   x++;             //지역 객체  1 증가
   cout<<x;         //지역 객체  2 출력
   ::x++;           //전역 객체  1 증가
   cout<<x;         //전역 객체 11 출력
}
```

5.3 동적 객체(dynamic object)의 사용 영역

객체는 언제든지 메모리에 동적으로 할당될 수 있다. 이 할당된 메모리 구역은 동적 메모리나 heap 또는 tas 라고 부른다.
이 동적 객체는 주소를 이용하여 조작하고 전역 객체에 속한다. 동적 객체의 정의는 연산자 "new" 나 malloc() 함수를 이용한다.

```
#include<iostream.h>
void f();
main()
{
   char *q;
   f();
   q=p;         //에러 발생 : p 는 함수 f()에서만 사용된다.
   cout<<q;
}
```

------------------------------ 계 속 ----------------------------

```
void f()
{
   char *p;
   p=new char[4];    //4byte 할당
   strcpy(p,"C++");
   cout<<p;
}
```

ㅋ5.4 계층형 객체의 사용 영역

C++ 의 계층은 `public 멤버와 private 멤버 그리고 friend 함수`로 구성된다.

(1) 계층 멤버(class member)는 데이타이거나 함수일 수 있다.
(2) friend 는 계층 멤버에 속하지 않는다. 그러나 어떠한 계층 멤버도
    사용할 권리를 가진다.

ㅎI
5.4.1 계층의 이중 연산자(overloadint operator)의 간주
ㅎ
(1) 데이타의 함수 멤버로 간주한다.
(2) friend 함수로 간주한다.

계층의 private 멤버는 계층안에서만 사용되는 `지역 객체`이다. 따라서 계층의 privte 메버는 계층 안의 멤버 함수들에 의해서만 사용이 가능하다(예외적으로 friend 함수에 의해서는 사용이 가능).

[예] private 멤버는 반드시 public 멤버 전에 정의되어야만 한다.

```
#include<iostream.h>
#include<math.h>           //수학 함수를 사용하기 위한 헤더 파일

class point
{
  int x,y;                  //private 멤버
  public:                   //public 멤버
------------------------------ 계  속 ----------------------------

  double distance(double a, double b)
  {
    x=a;
    y=b;
    return(sqrt(x+y*y));
  }
}                             //end of class
```

```
main()
{
  point p;                  //point 계층형 객체인 p
  cout<<p.distance(5,4);
  p.x=2;                    //x 는 지역 객체이므로 에러 발생
}
```

------------------------------ 계 속 ----------------------------

[예] 계층 객체안에서 public 으로 정의된 객체들은 `전역 객체`이다.
그리고 public 형만을 가지는 계층은 struct 구조문과 동일하다.

```
#include<iostream.h>          //수학 함수를 사용하기 위한 헤더 파일
#include<math.h>
struct point
{
  int x, y;
  double distance(double a, double b)
  {
    x=a;
    y=b;
    return(sqrt(x*x+y*y));  //수학 함수
  }
}                              //end of class
```

------------------------------ 계 속 ----------------------------

```
main()
{
  point p;
  cout<<p.distance(5,4);
  p.x;
}
```

[설명]

위의 프로그램은 프로그램의 모듈화를 방해한다. 마치 전역 객체만 사용하는 프로그램을 작성하는 것과 같은 결과를 초래할 수 있다.

6. 객체의 사용 유효 기간

ㄶ]6.1 전역 객체나 static 형 객체

(1) 컴파일러에 의해 일정하게 지정된 메모리를 예약한다.
(2) 이렇게 지정된 구역을 데이타 구역이라고 한다.
(3) 자동적으로 0 이 초기값(initialize)`으로 지정된다.
(4) 프로그램 실행 기간과 같은 사용 유효 기간을 가진다(즉, 시작과 끝)
(5) static 형 객체는 지역적 객체이다. 그러나 그 작업 결과는 계속 가지고 있으면서 다음에 다시 호출시 그 값을 가진다.

[예]

```
include<iostream.h>
void f();                    //함수 선언

void f()                     //함수 정의
{
  static int x=1;            //지역 객체
```

------------------------------- 계 속 -----------------------------

```
  ++x;
  cout<<x;
}

main()
{
  f();          //2 를 인쇄
  f();          //바로 전 결과를 가지고 작업하므로 3 을 인쇄
  f();          //바로 전 결과를 가지고 작업하므로 4 을 인쇄
}
```

ㄲ6.2 지역 객체

(1) 프로그램이 실행될 때의 상황에 따라 필요한 메모리를 예약한다.
(2) automatic 형으로 절대로 초기값이 자동적으로 주어지지 않는다.
(3) 관련 함수나 프로그램 모듈의 실행 시간과 같은 사용 유효 기간을 가진다.
(4) 작업이 끝나면 작업 결과를 버리고 항상 새로운 값으로 작업한다.

------------------------------ 계 속 ----------------------------

[예]

```
#include<iostream.h>
short* f()
{
   short x=10;
   return(&r);
}
short g()
{
   cout<<"OK";
}
main()
{
   short* p;
   p=f();
   cout<<*p;          //10 을 인쇄
   g();
   cout<<*p;          //어떤 값을 인쇄할지 모름
}
```

6.3 동적 객체

(1) C++ 의 연산자 "new"의 도움으로 heap 이나 tas 메모리가 할당된다.
(2) 사용 범위는 전역 객체와 같으나 사용 유효 기간은 이 객체를 메모리에서 제거할 때 까지 유효하다.
(3) 전역 객체와 같이 사용되면서 원할 때 언제든지 없앨 수 있다.

[예]

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char *x, *y;
```

```
void f();
void g();

main()
{
  f();
  printf("%s",x);            //STUDY 인쇄
```

------------------------------ 계 속 ----------------------------

```
  g();
  printf("%s",y);            //STUDENT 인쇄
  printf("%s",x);            //STUDENT 인쇄
}
void f()
{
  x=(char *) malloc(5);     //x에 동적 메모리 할당
  strcpy(x,"STUDY");
}
void g()
{
  free(x);                  //x에 할당된 메모리 해제
  y=(char *) malloc(8);     //y에 동적 메모리 할당
  strcpy(y,"STUDENT");
}
```

6.4 객체의 요약

(1) 객체는 항상 사용 전에 반드시 선언과 정의되어야 한다.

(2) 객체의 정의는 곧 객체를 위한 메모리 예약을 의미한다.

(3) 객체는 식별자와 형(type)을 조합하여 만든다.

(4) C++ 에서 객체는 변수나 상수, 데이타등 프로그램이 작업할 요소이다.

(5) 객체는 사용 범위에 따라 전역과 지역 객체로 구분되며, 유효 범위에

따라 단정도형 객체와 배정도형 객체가 있다.

I. 고급언어와 어셈블러의 인터페이싱에 관한 기초

ㅎI

고급언어(Turbo C, Pascal, ...)와

어셈블리 언어 서브루틴을 인터페이스할 때 유의 사항(3가지)

ㅎ

(1) 서브루틴을 어떻게 호출하고 시키는가 ?

(2) 파라미터를 어떻게 넘겨주고, 넘겨 받는가 ?

(3) 반환값이 어떻게 반환되는가 ?

1. 파라미터 넘겨줌(Passing parameter)

ㅋJ1.1 스택을 이용한다.

데이타를 서브루틴으로 파라미터를 사용하여 전송하기 위해서 파라미터를 임시로 담아두는 영역으로 스택을 사용한다.

ㅋJ1.2 CPU 의 레지스터 혹은 메모리 영역(파라미터 블럭)을 사용한다.

파라미터들은 CPU 내의 레지스터를 이요해서 넙겨줄 수도 있고, 어떤 다른 메모리 영역을 이용하는 방법이 있다(DBMS 언어).

ㅋJ1.3 서브루틴 호출 전에 파라미터를 컴파일러가 스택을 삽입해 주고
서브루틴이 액세스, 수정한다.

고급언어 각각이 메모리를 서로 다른 방식으로 사용하기 때문에 서브루틴 인터페이스를 위해서 `공동영역(common area)`이 필료하게 된다. 이런 영역을 위해서 표준적으로 선택하는 것이 스택이다. 우리가 서브루틴을 호출하기 전에 파라미터를 스택에 삽입해 놓게 되면 서브루틴은 스택 속의 데이타를 액세스하여 수정하게 된다.

2. 스텍에 파라미터가 어떤 식으로 쌓이는가 ?

보통 각각의 파라미터는 `데이타값이나 데이타를 가리키는 포인터`로서 넘겨준다.

(1) 파라미터가 짧은 수치정수일때는 그 데이타값을 스택에 넣게 된다.
(2) 문자열일때는 그 문자열에 대한 포인터만 넘겨준다.

ЋЈ3. 파라미터들은 어떤 긱으로 포맷하는가 ?

개별적 파라미터 포맷은 언어마다 다르다.
파라미터는 스택상에 적어도 하나의 완전한 워드(16 비트)를 차지해야 되지만, 고급언어에 따라서 추가적인 워드가 사용되기도 한다.

4. 스택상의 파라미터를 어떻게 액세스하는가 ?

스택을 통하여 넘겨준 정보를 액세스하기 위해서, 대개의 경우 이책의 각종 루틴들을 베이스 포인터인 BP 레지스터를 사용한다. 이 레지스터는 2차 스택 포인터로서 주어진 위치와 관련하여 `스택상에 있는 데이타를` `액세스`하는데 주로 사용된다.

ЋЈ4.1 제 2 의 스택 포인터 BP 레지스터

대부분 어셈블리 언어 프로그래머는 스택의 내용을 액세스하는데 BP를 사용한다.
BP를 사용하면 넘겨진 파라미터나 서브루틴 내에서의 파라미터 등도 액세스 사능하게 된다.

ЋЈ4.2 BP 는 색인 간접번지 지정방식에 적합하다.

스택상에 넘겨준 파라미터를 액세스하는 방법은 색인, 간접번지 지정방식을 사용하는데 BP가 적합하다.
그런데 SP 레지스터는 색인, 간접번지 지정방식에 사용할 수 없다.

5. 서브루틴으로부터 고급언어로 값을 어떻게 반환하는가 ?

어셈블리어 서브루틴으로부터 처리된 값을 고급언어의 호출 프로그램쪽으로 결과 값을 반환하는 방식은 다음과 같다.

(1) 스택을 통하여 반환한다.
(2) 레지스터를 통하여 반환한다
(3) 메모리를 통하여 반환한다.

II. 터보 C 와 어셈블러의 인터페이스

대부분의 C 프로그래머들은 에셈블리어 서브루틴과 C 언어와의 인터페이스 작업에 직면하지 않을 수 없다. 어셈블리어 서브루틴은 프로그램 개발 과정의 맨 마지막 단계로서 링크할 수 있는 것이다. 이들 인터페이스상에서 `변수들의 합당한 넘겨줌을 허용하는 일`이 중요하다.

1. 메모리 모델의 종류

메모리 모델은 인터페이싱 절차에 커다란 영향을 준다.
Intel 8088 이나 Intel 8086 CPU 는 세그먼트 단위로 메모리 관리 하도록 되어 있고, 각 세그먼터 내에서는 최대 64K(65,636 byte)까지 번지지정할 수 있다.

তI
1.1 Tiny 모델
ত
tiny 모델이란 COM 화일이 사용하는 포맷이다.
프로시저에 대한 포인터 및 데이타 포인터는 near 이다.
-------------------------------- 계 속 ----------------------------
তI
1.2 Small 모델

small 모델은 code 와 data 를 각각 별개의 64K 세그먼트로 분리시켜 저장하게 되어 있다.
프로시저에 대한 포인터 및 데이타 포인터는 near 이다.

1.3 Medium 모델

code들의 모듈들은 각시 64K까지의 자기 자신의 코드 세그먼트를 가질 수 있다. 그러나 모든 데이타는 하나의 세그먼트에 들어가야 한다.
프로시저 포인터는 보통 far 포인터이고, 에이타 포인터는 near 이다.

1.4 Compact 모델

medium 모델의 반대이다. 즉 각각의 데이타 구조는 각기 자체의 세그먼트를 갖는다. 그러나 코드는 64K내에 들어가야 한다.
코드 포인터는 near 이고, 데이타 포인터는 far 이다.

------------------------------- 계 속 ---------------------------

1.5 Large 모델

large 모델은 코드 포인터, 데이터 포인터 모두 far 포인터를 사용한다.
그러나 정적 데이타 저장은 64K 크기로 제한한다. medium 과 compact 모델의 조합이다.

1.6 Huge 모델

이 모델은 정적 데이타상에 과해지는 60K 제한을 제거하여 하나의 데이타 구조를 한 세그먼트보다 크게 확장을 허용한다. 그러나 내부적인 번지 계산이 갑자기 복잡해진다.

2. C 언어에 어셈블이어 서브루틴을 인터페이스하는 절차

(1) 어셈블리어 서브루틴의 코드 세그먼트에 특정한 세그먼트 이름을

주어야만 한다(모델별로 지정된 형식의 이름이다).

- Turbo C, Microsoft C, Quick C 에서 tiny, small, compact 메모리 모델에서는 코드 세그먼트 이름이 `_TEXT(Microsoft C 만 tiny 모델` `을 지원하지 않음)`이고, 다른 메모리 모델에서는`<세그먼트 이름>` `_TEXT_`와 같은 포맷이 지정된다.

(2) 어셈블리어 서브루틴의 데이타가 세그먼트에 대해서는 특정한 이름을 필료로 한다.(만일 데이타 코드가 코드 세그먼트의 외부에서 참조될 경우)

- Turbo C, Microsoft C, Quick C 는 데이타 세그먼트 이름이 _DATA_ 이다.

-------------------------------- 계 속 ----------------------------

(3) 어셈블리어 서브루틴으로 스택을 통해서 어떻게 변수들이 넘겨지는가 확실히 알아야 한다.

- function_name(arg1, arg2,...,arg n);

(4) C (Microsoft C, Quick C) 로부터 호출되는 어셈블이어 루틴 이름이 어셈블리어 소스 화일속에서 밑줄부터 시작해야 된다.
(Turbo C 는 디폴트 옵션이 그렇고, OFF 로 지정하면 밑줄 프리픽스를 없앨 수 있다).

- _kchar --> kchar();

(5) 어셈블리어 서브루틴속에서 변경시키게 되는 특수 목적 레지스터들은 보존하라.

- 어셈블리어 서브루틴속에서 사용하게 되는 CS, DS, SS, BP, SI, DI 같은 특별 목적으로 사용되는 레지스터들을 스텍에 삽입해서 담아두어라. 그렇지 않으면 서브루틴 수행이 끝나고 제어가 C 프로그램으로 복귀되었을때 레지스터값이 변경되어 원치 않는 결과를 초래하게 된다.

3. C 언어와 어셈블리어 루틴과 인터페이싱

- 호출받는 어셈블리어 서브루틴의 생김새.

(1) 베이스 포인터 레지스터(BP)를 저장한다.

```
PUSH BP   ...  베이스 포인터 레지스터를 저장한다.
```

(2) 스택 포인터(SP)의 내용을 BP 에 저장한다.

```
MOV DP, SP   .. 스텍 포인터를 BP 에 담는다.
```

(3) 인덱스 레지스터(SI, DI) 와 세그먼트 레지스터(DS, ES)를 대피시킨다.

```
PUSH   DS
PUSH   ES
PUSH   SI
PUSH   DI
```

-------------------------------- 계 속 ----------------------------

(4) 어셈블리어 서브루틴의 내용을 수정한다.
(5) 스택으로부터 전 페이지 (3)번에 저장했던 레지스터들을 복원하낟.

```
POP   DI
POP   SI
POP   ES
POP   DS
```

(6) 스택 포인터(SP)를 복원한다.

```
MOV   SP, BP
```

(7) 베이스 포인터 레지스터(BP)의 내용을 복원한다.

```
POP   BP
```

(8) 제어를 호출 프로그램 C 로 넘긴다.

```
RET
```

ㅎl
매크로 기능 활용
ㅎ
스택에 레지스터들을 저장(save)과 로드(load)하는 과정이 되풀이 되고,

길기 때문에 읽기 쉽게 만들기 위해서 아래와 같이 두 개의 MACRO를 정의해서 사용한다. Prefix라는 매크로눈 서브루틴의 진입점 다음에 첫 명령으로 놓을 수 있다.

```
[화일명]    macro.asm
Prefix
PUSH BP           ;save the base pointer
MOV  BP, SP       ;copy and save the stack pointer
PUSH DS           ;save the data segment
PUSH ES           ;save the extra segment
PUSH SI           ;save the source segment
PUSH DI           ;save the destination index
endm              ;end of macro
```

-------------------------------- 계 속 ----------------------------

```
Postfix macro
POP  DI           ;restore the destination index
POP  SI           ;restore the source index
POP  ES           ;restore the extra segment
POP  DS           ;restore the data segment
MOV  SP, BP       ;restore the stack pointer
POP  BP           ;restore the base pointer
RET               ;return to the calling "C" routine
endm              ;end of macro
```

4. C 루틴으로부터 파라미터를 어셈블리어 서브루틴에 넘겨준다.

- 파라미터를 넘겨주는 과정

ㅠ4.1 스택 작동을 훤히 알고 있어야 한다.

C 언어로부터 어셈블리어 서브루틴으로 파라미터들을 넘겨줄 때는 스택을 사용한다.

তI
C 언어에서 어셈블리어 서브루틴 호출하는 표준 형태
ত
Do_something(parameter1, parameter2, ..., parameter N);
서브루틴의 화일 이름은 Do_something이고, 파라미터들은 선택이어서 전혀 없는 경우도 있지만, 여기서는 N기이 파라미터들과 함께 호출되고 있다.

4.1 파라미터(Parameter)

তI
C 언어로부터 어셈블리어 서브루틴으로 파라미터를 넘겨주는 단계
ত

```
+--------------------------------------------------------------+
|단계번호|          기              능                         |
|--------|-----------------------------------------------------|
|   1    |스택에 파라미터들을 삽입한다(파라미터 N부터 파라미터 순)|
|   2    |스택에 복귀번지를 삽입한다.                           |
|   3    |어셈블리어 서브루틴으로 제어를 넘긴다.                |
+--------------------------------------------------------------+
```

5. 어셈블리어 서브루틴에서는 파라미터들을 어떻게 가져다 쓰는가 ?

5.1 파라미터들을 가져다 쓰기 위해 필요한 자전 지식

(1) 파라미터들의 갯수
(2) 각 파라미터의 데이터형(어떤 값인가 포인터인가)
(3) 어떤 메모리 모델로 C 언어가 컴파일되었는가 ?

5.2 파라미터들을 어셈블리어 서브루틴속에 액세스하는 방식은 간접, 색인
번지지정방식이다.

5.3 스택 포인터(SP)는 색인, 간법번지지정방식에는 사용할 수 없는 레지
스터이다. 베이스 포인터 레지스(BP)를 파라미터로 사용한다.

5.4 어셈블리어 서브루틴속에서 C 루틴이 삽입해 놓은 파라미터들을 POP
해서는 절대로 안 된다.

5.5 서브루틴속에서 파라미터를 꺼내지 않고 BP를 이용하여 색인, 간접번지
지정방식으로 파라미터들을 액세스한다.

-------------------------------- 계 속 ----------------------------

5.6 Small C 컴파일러 모델에서는 한 워드의 복귀번지가 스택에 저장된다.

5.7 Medium 과 Large C 컴파일러 모델에서는 두 개의 16 비트 워드(세그먼
트와 오프셋)를 삽입한다.

5.8 여기서는 서브루틴에게 파라미터 형으로서 데이타값과 포인터가 있다.

5.8.1 변수를 어셈블리어 서브루틴에 넘겨줄 때(passing variable) 특징

- 변수가 C 언어로부터 어셈블리어 루틴으로 넘겨주게 되면 그 변수의
  현재의 값이 복가된 것이 스택상에 놓인다.

(1) 어셈블리어 루틴속에서는 C 자체 속에 있는 변수(파라미터)를 변경할
수 없다.
(2) C 프로그램내의 변수를 보호하는 기능이다.

-------------------------------- 계 속 ----------------------------

5.8.2 포인터를 어셈블리어 서브루틴에 넘겨줄 때의 특징

(1) 포인터 서브루틴에게 넘긴다는 사실은 그 파라미터를 서브루틴에서 변
경할 수 있게 허용한다.

(2) 변수 데이타에 대한 포인터와 배열에 대한 포인터를 넘긴다.

ㅠ6. 함수값(argument)을 반환하는 경우

6.1 함수를 C 언어로 작성하는 경우 함수값 반환

```
return(argument);
```

6.2 함수를 어셈블리어로 작성한 경우 함수값 반환

- AX 레지스터를 통해 C 호출 루틴으로 반환한다.

-------------------------------- 계 속 ----------------------------

```
MOV AX, argument        ;return "argument" to the caller
RET
```

(1) 2 바이트 크기 이하의 데이타형 - AX 에 담아 돌려준다.
- char, unsigned char, int, unsigned, enum, near pointer

(2) 4 바이트 크기의 데이타형 - DX : AX에 담아 돌려준다.
- DX 에 상위 2 바이트를, AX 에 하위 2 바이트가 담긴다.
- long, unsigned long, far pointer, huge pointer

(3) Floating pointer 형 - 80x87 의 TOS 레지스터, 즉 ST(O) 에 담아 돌려준다.
- 단, 8087을 emulate 하는 경우는 에뮬레이터내의 TOS 레지스터에 담는다.
- float, double, longdouble

(4) 구조체, 공용체의 복귀
- 따로 메모리에 결과를 저장하고 그 영역을 가리키는 포인터를 AX (near 인 경우) 또는 DX : AX(far 인 경우)에 담아 돌려준다.

7. C 루틴에서 호출하는 전형적 어셈블리어 루틴은 어떻게 생겼나 ?

ㅠ7.1 어셈블리어 서브루틴 설계 단계

(1) 필요한 매크로 확장이 있으면 선언한다.

```
if1
INCLUDE macros.asm
ENDif
```

(2) 어셈블리어 서브루틴 진입점을 PUBLIC 이라 선언한다.

```
PUBLIC _example
```

(3) 세그먼트를 정의한다.

```
codeseg SEGMENT byte public 'code'
ASSUME cs : codeseg
```

-------------------------------- 계 속 ----------------------------

(4) 서브루틴의 진입점을 정의한다.

```
_example   PROC   far
```

(5) 서브루틴의 본체를 수행한다.

(6) 에셈블리어 서브루틴을 닫는다.

```
_example   ENDP      ;프로시저의 끝
```

(7) 코드 세그먼트를 닫는다.

```
codeseg   ENDS
```

8. C 언어와 어셈블리어 서브루틴의 실제 접속

8.1 어셈블러를 이용하는 방법

(1) TASM 의 이용

```
tasm /mx filename
```

(2) MASM의 이용

```
masm /MX filename
```

8.2 tcc를 이용한 간접 어셈블

만일 프로그램이 C 모듈인 main.c 와
어셈블리어로 된 부속 모듈 sub.asm으로 구성시

```
tcc main sub.asm
tcc -c sub.asm          /* 어셈블만 */
tcc main.lbj sub.asm    /* 링크 */
```

[예제] 어셈블리어 모듈의 어셈블 및 링크

```
/* test.c */

extern void sub(void);

main()
{
  for(;;)
  {
    sub()'
  }
}
```

```
/* t4sub.asm */

public _sub
```

-------------------------------- 계 속 ----------------------------

```
t3sub SEGMENT byye public 'code'
        assume  cs : t3sub

_sub      proc    near

        push    bp          ;진입부
        mov     bp, sp

        mov ah,02h          ;기능번호 호출
        mov dl,31h          ;출력문자
```

```
          int 21h

          pop bp              ;탈출부
          ret                 ;복귀
```

-------------------------------- 계 속 ----------------------------

```
_sub      endp
t3sub     ends

dsub      segument word public 'DATA'
dsub      ends

          end
```

[설명]

(1) tcc 를 이용한 test.c를 컴파일하여 test.obj를 만든다.

    tcc -c test.c

(2) t4sub.asm 인 어셈블리어 루틴을 TASM 을 이용 어셈블하여
    t4sub.obj 를 만든다.

-------------------------------- 계 속 ----------------------------

(3) TLINK를 이용하여 아래외 같이 링크한다.

    TLINK

    그러면 TEST.EXE가 생성된다.

    tcc testc.sub

    TESTC.EXE 가 생성된다.

1. 전처리계란 ?

전처리계(preprocessor)란 원시 파일을 컴파일하기 이전에 그 원시 화일에 대해 일련의 작업을 행하는 것으로서 원시 화일을 컴파일하기 좋도록 `가공`하는 작업이다.

다시 말하자면 전처리계는 원시 화일 내의 모든 전처리계 지시자(pre-pro-cessing directive)를 컴파일할 수 있는 C 의 문장으로 전개(expansion)하는 작업을 한다.

2. 전처리 기능과 지시자

1) 매크로 전개(macro expansion) - #define

2) 외부 화일 포함(file inclusion) - #include

3) 조건부 컴파일(condifional compilation)

- #if, #ifdef, #ifndef, #else, #elif, #endifet defined 연산자

4) 기타 : #error, #line, #, #undef, #pragma

3. #include 문에 의 외부 화일 포함 기능

- 외부화일을 포함하는 기능을 한다.

[형식] : #include <화일명>

[방법] : #include<standard.h>

[설명] : 표준 헤더화일을 포함시킬때 주로 사용된다.
이 형식은 화일명에 경로(path)가 생략되었을때는 표준 헤더
화일 디렉토리에서 헤더 화일을 찾는다.

[형식] : #include "화일명"

[방법] : #include "user.h"

------------------------ 계 속 ----------------------

[설명] : 사용자가 작성한 헤더화일을 포함시킬때 주로 사용된다.
이 형식은 화일명에 경로(path)가 생략되었을때는 현재 디렉
토리(current directory)에 찾은 다음 없으면 표준 헤더 화일
디렉토리에서 헤더 화일을 찾는다.

[형식] : #include 매크로 상수

[방법] : #define HEADER "bcheader.h"
#include HEADER

[설명] : 이것은 매크로 상수를 이용하여 헤더화일을 지정할 수 있다는
점 외에는 별 다른 것이 없다.

[참고] 헤더화일 경로명(path)은 문자열 상수가 아니므로 "\\"이 아니고
'\'로 해야 된다.
예] #include "sys\\stat.h" ------- (X)
#include "sys\stat.h" -------- (0)

4. #define 문에 의한 매크로 정의

ㅠ]4.1매크로 상수 정의시 주의사항

1) 문자상수나 문자열상수내에서는 매크로가 전개되지 않는다.

2) #define 문의 치환 리스트가 수식일때는 반드시 수식 전체를 괄호()로 묶어야 한다.

3) 매크로 상수명은 되도록 대문자를 사용하도록 한다.

```
[예] #define SQUSRE(y * y)
       x = SQUARE + 1;
       ----> x = (y * y) + 1 로 전개 된다.
```

------------------------ 계 속 ----------------------

```
[예] #define square(x) (x * x)
       z = square(y + 1);
       ----> z = (y + 1) * (y + 1);  /* 원하는 결과 */
       ----> z = y + y + 1;          /* 실제 결과   */
```

이것은 부작용으로서 아래 [예]와 같이 재정의 해야 한다.

```
[예] #define square (x) ((x)*(x))
       z = square(y + 1);
       ----> z = ((y + 1) * (y + 1)) 로 전개된다.
```

[참고] #define SQUSRE(y * y)은 매크로 상수 정의한 것이고 다른 것은 매크로 함수를 정의한 것이다.

ㅎlsam1700.dat

[실습내용] 이것은 월급을 계산하는 프로그램이다. 기본급(base)이 20 만원이면 기본급에 근무시간당(rate) 1천원씩 추가하여 지급하는 방법에 따라 월 근무시간 150시간(LOW)부터 170시간(UP)까지에 해당하는 월급을 계산한다.

```
#define RATE 1
```

```
#define BASE 200
#define LOW   150
#define UP     170
#define PR(x) printf("Your total x is %u\n", x)
main()
{
  unsigned income, hours;
  for (hours = LOW; hours <=UP; hours++)
  {
    income = hours * RATE + BASE;
    PR(hours);
    PR(income);
  } }
```
ㅎ
4.2 매크로 함수의 문제점

```
#define square(x) (x * x)
z = square(y + 1);
```

이와 같이 정의하면 보통 그 결과를

`z = (y+1 * y+1);`으로 오판하기 쉽다.
그러나 항상 연산자 우선순위를 고려해야만 한다.
연산자 +보다 *가 우선하므로 결과는

`z = y + y + 1;`이 된다. 그래서 square 를 아래와 같이 재정의 해야만 한다.

`#define square (x)  ((x) * (x))`

그 결과는 `z = ((y + 1) * (y + 1));`와 같이 원하는 결과를 얻을 수 있다.

----------------------- 계 속 ----------------------

[참고] 증감연산자 + +, - -는 어쩔수 없이 부작용이 있다.

```
z=square(y+ +); ->  z=((y+ +)*(y+ +));
z=square(y- -): ->  z=((y- -)*(y- -));
```

ㅋ4.3 매크로 함수 정의시 주의사항

1) #define는 반드시 첫열에서 시작해야 한다.

2) 매크로 함수명과 식별자 리스트 사이에는 공백이 없어야 한다.

3) 식별자 리스트내에는 공백이 있어야 상관없다.

4) 식별자 리스트는 생략할 수 있으나 식별자 리스트를 묶는 괄호()
   는 생략할 수 없다. 생략시는 매크로 상수가 되어 버린다.

---------------------- 계　속 -------------------

5) #define는 2행에 걸쳐서 기술할 수 없다.
   그러나 '\'를 이용하여 아래와 같이 2행에 걸쳐서 기술할 수는 있다.

```
+------------------------------------------+
|        틀린 예         |       바른 예            |
|-------------------|---------------------|
| # define square(x) | #define square(x) \ |
|             ((x)*(x)   |            ((x)*(x)      |
+------------------------------------------+
```

6) 매크로 상수와는 달리 실재함수와 구별없이 사용하기 위해서 소문자
   로 쓴다.

7) 실매개변수로 + +,- -,=,op=등의 연자자를 사용해서는 안된다.

4.4　매크로 함수의 활용

ƃl
4.4.1 매크로 함수의 특징
ƃ
1)매개변수를 함수에 전달하는 시간을 줄이므로 실행 속도를 증가시킨다.

2)매크로 함수는 그것이 사용되는 곳마다 치환리스트로 전개된다. 따라서

실제 함수를 사용할 때보다 프로그램이 길어질 수 있다.

ㅎl
4.4.2 매크로 함수의 용도
ㅎ
1) 중복되는 수식을 매크로 함수는 간결하게 기술할 수 있다.

[정의] #define leap year(year) (year % 4==0 && year % 100!=0 \
                              || year % 400==0)

----------------------- 계 속 ----------------------

[예] if(leap year(year)) printf("%d is the leap year.\n",year);
     else   printf(%"d is the common year.\n",year);

2) 표준 라이브러리 함수의 특정 실매개변수를 특정값으로 지정할 경우 사용할 수 있다. 프로그램을 작성하다보면 표준 라이브러리 함수의 특정 매개변수를 항상 특정한 값으로 지정하여 사용하는 경우가 많다.

[정의] #define paint (x,y) floodfill(x,y,1)

3) 매개변수를 특정 데이타형에만 제한하고 싶지 않은 경우에 사용할 수 있다.

[예] #define max(a,b) (((a) > (b))? (a) ; (b))
     #define min(a,b) (((a) < (b))? (a); (b))

5. 조건부 컴파일(넘어가도 상관없다)

조건부 컴파일(conditional compilation)이란 말 그대로 특정한 조건이

성립될때나 성립되지 않을 때에만 지정한 범위내의 문장을 컴파일하거나

또는 그냥 무시라고 컴파일하지 않는 것을 말한다. 주목적은 프로그램의

호환성을 높이려는데 있다.참고로 표준헤더화일에서 찾아보면 많이 볼수

있으므로 자세하게 알고자 한다면 그것을 분석해 보면 가장 빠를 것이다.

ㅎl
#if, #elit문의 상수 수식에 사용할 수 없는 연산자

+ +, - -, *(간접 연산자), &(번지 연산자), =, op=(10개),
[], ->, ., #, ##
ㅎ
ㅎl
조건부 컴파일 3가지 형식
ㅎ

```
+-------------------------------------------------------------+
| #if   상수 수식 1   | #ifdef 매크로명    | #ifndef 매크로명    |
|       문장 1        |        문장 1      |         문장 1      |
| [#elif 상수 수식 2 | [#elif 상수 수식 2 | [#elif  상수 수식 2 |
|         문장 2]     |        문장 2]     |         문장2]      |
| [#else              | [#else             | [#else              |
|         문장 3]     |        문장 3]     |         문장3]      |
| #endif              | #endif             | #endif              |
+-------------------------------------------------------------+
----------------------------- 계  속 ----------------------------
```

[설명]
C의 선택문 if ~ else문을 쓰듯이 하면 된다. 다만 다른점은 괄호()로
묶지 않는다는 것이다.그리고 #endif문을 빠뜨릴 경우에는 조건에 따라
#if...문 이후의 모든 문장이 컴파일되지 않을 수도 있다.

```
+------------------------------------------+
|   옳은 예              |   틀릴 예         |
|---------------------|--------------------|
| #define MACRO        | #if defined(MACRO) |
```

| #if defined (MACRO) | #endif |
| #endif | #define MACRO |
+--+

6. 기타 전처리계 지시자

#error, #(널 지시자), #line, #undef, #pragma등 6가지의 지시자가 더

있다. 여기에서는 거의 사용되지 않으므로 undef와 pragma에 대해 간단

하게 다루고 넘어가도록 하자.

ㅁ6.1 #undef문

#undef은 해당되는 매크로가 이미 정의되어 있을 때, 그중에서 가장 최근

에 만들어진 매크로 정의를 취소시킨다. 즉, #undef는 이미 정의된 어떤

것을 취소하거나, 아니면 필요에 의하여 매크로나 상수를 중복 정의한 다

음에 다시 원래의 정의로 되돌리 때 사용된다.

------------------------ 계 속 ----------------------

```
[예] #define BIG 3
     #define HUGE 5
     #undef BIG        /* BIG의 정의를 취소                 */
     #define HUGE 10   /* HUGE를 10으로 재정의              */
     #undef HUGE       /* HUGE는 전에 정의된 5의 값으로 환원 */
     #undef HUGE       /* HUGE의 정의를 취소                */
```

ㅁ6.2 #pragma문

#pragma문은 #pragma inline 문과 #pragma warn문 2가지로 구분할 수

있다. 여기서 #pragma warn문의 예를 다루겠다.

ㄶI
#pragma warn
ㄶ
이 문은 경고 메시지에 관계된 컴파일러 옵션을 원시 화일내에서 제어할 수 있도록 지시한다.

----------------------- 계 속 ---------------------
ㄶI
#pragma warn -dup
ㄶ
"Rede finition of 'xxxxxxxx' is not idenical"경고가 전혀 발생하지 못하게 한다. 이 지시자를 쓰면 모든 사용자 정의 매크로를 마음대로 재정의하여 쓸 수 있다.

ㄶI
#pragma +cln
ㄶ
"Constant is long"경고를 발생하도록 한다. 이지시자를 쓰면 32768같은 long형 상수(unsigned 형이 아님)에 접미사 L을 붙이지 않으면 경고가 발생한다.

ㄶI
pragna +sig
ㄶ
산술 변환에 의해 demotion이 일어났을 때, "Conversion may lose significant digits"경고를 발생하도록 한다. demotion 에 의해 논리 에러를 점검할때 유용하다.
7. typedef문에 의한 사용자 정의 데이타형(user-defined data type)

ㄷJ7.1 형정의 (type definition)를 위한 절차"

1) 형정의 하고자 하는 원래의 데이타형을 결정하고 그 데이타형의 변수를 임시로 하나만 선언(정의) 한다. (A:임시변수위치)

```
[예] unsigned char A;
     char *A;
     int A[100];
```

2) 1)의 선언문에서 A위치에 형정의할 새로운 데이타형의 형명을 놓는다.

3) 1) 2) 를 따라 typedef를 사용형 정의

```
[예] typedef unsigned char byte;  /* byte형 정의 */
      typedef char *string;          /* string형 정의 */
      typedef int array[100];        /* aray 형 정의 */
```

--------------------------- 계 속 --------------------------

4) 선언문과 수식에 사용

```
+----------------------------------------+
|      선언문          |      수  식         |
|--------------------|-------------------|
| byte a, b, c;        | a = (byte) 1, 2;  |
| string S1, S2, S3; | s3 = "All right !"|
| array x,  y;         | x[10] = y[10];     |
+----------------------------------------+
```

ㅠ7.2 다음 typedef문에 의한 형정의를 하면 좋다.

1) 기본형 중에서 void 형을 제외한 산술형의 전부,그 중에서도 특히
   열거형의 경우에는 여러가지면에서 좋다.

```
typedef unsigned char byte;      /*byte형 정의  */
typedef enum {false, true} boolean;    /* byte형 정의 */
```

------------------------ 계 속 ----------------------

2) 구조체형 (structure type)과 공용체형(union type)에 사용하면 좋다.

[참고] typedef문을 포이터형과 배열형에 사용하는 것을 절대 좋지 않다.
        그이유는 실제 사실을 망각하는 경우가 많다.

ㅠ8. 열거형 (enumerated type)

열거형은 열거상수들의 모임이다. 그리고 모든 열거형 데이타는 수식내에서
항상 int 형으로 자동 변환이 된다.

[형식] enum {멤버1, 멤버2,...} 변수명1, 변수명2,...;
        enum 태그명 {멤버1, 메버2,...} 변수명1, 면수명2,...;

[예] enum {a, b, c, d, e} my, your;

[설명] 컴파일러는 괄호{...}에 나열된 식별자 목록에 0 부터 1씩 증가된
값을 할당한다.

----------------------- 계 속 ----------------------

[결과] a=0, b=1, c=2, d=3, e=4 가 된다.

[예2] enum {a, b=0, c, d, e } my, your;

[설명] a에는 묵시적으로 0이 할당되나 b에 명시적으로 0이 지정
되어 C부터 e에는 1부터 4가 지정된다.

ꪸl
typedef에 의한 형 정의
ꪸ
typedef {멤버1,......}형명;

예) typedef enum {a, b, c, d} ABC;

ꪸlsam1701.dat
[실습내용] 열거형의 사용예

```
void main (void)
{
   typedef enum {sun, mon, tue, wed, thu, fri, sat} days;
   enum { dog, boy, girl } friend;
   days today;
   int itoday, ifriend;
   friend = girl;
   today = sat;
   printf("Friend = %d, Today = %d\n", friend, today);
   ifriend = friend;
   itoday = today;
   printf("iFriend = %d,iToday = %d\n", ifriend, itoday);
```

```
   today = girl;
     printf("Today = %d\n", today);
}
```

------------------------ 계 속 ----------------------

ᇹ

ᇹIsam1701.dat

[결과]

```
Friend = 2, Today = 6
iFriend = 2, iToday = 6
Today = 2
```

ᇹ

1. 서론

마우스는 위치를 표시할 수 있는 대화식 입력 장치로서 컴퓨터 뒷면에는 `RS-232C`라는 표준 직렬 인터페이스 카드에 연결 코드가 접속되어 있으면, 마우스의 모든 동작 상태는 이러한 직렬 전송선과 통신으로 이루어진다. IBM-PC와 그 호환기종에서는 사용되는 마우스들은 직렬 포트를 이용하는 것과 병렬 포트를 이용하는 것이 있고, 국내의 대분분 마우스는 직렬 포트를 이용하고 있다.

IBM-PC와 그 호환기종에서는 마우스 제어 루틴을 지원하지 않는다. 따라서 램 상주 프로그램과 마우스 드라이버에서 지원하는 `INT 33H`를 이용한다. 그러므로 마우스를 사용하기 위해서는 마우스 드라이브를 램 상주시켜 놓고 마우스 제어 함수를 구동시켜야 한다.

2. 마우스 인터럽트 함수

마우스에 대한 정보 입력과 출력의 제어는 `INT 33H`를 이용한다.

ᇹI

내부 레지스터

ᇹ

AX　　: 마우스 함수종류 선택 및 마우스 상태 변환

BX : 버튼의 상태에 관한 정보
CX : X축 방향에 관한 관련 정보
DX : Y축 방향에 관한 관련 정보
SD, DI : 마우스 확장 정보에 쓰인다.
ㆆl
마우스의 버튼을 조작할 때 쓰이는 용어
ㆆ
Press : 버튼을 누른 채로 가만히 있는 상태
Release : 버튼에서 손을 뗀 상태
Click : 버튼을 재빨리 눌렀다 뗀 상태
Double Click : 정지 상태에서 클릭 동작을 연속 두 번 수행한 상태
Drag : 버튼을 누른 채로 이동하는 상태가 있다.

마우스 버튼을 누르면 마우스의 버튼 상태를 나타내는 2 바이트 변수가 값을 달리한다. 가령 `왼쪽과 맨 오른쪽 버튼`을 동시에 누르면 리턴되는 BX 가 3 으로 반환되고, 버튼이 세개인 마우스에서 가운데 버튼을 누를 경우에는 4 가 된다.

`마우스 함수 번화와 기능`

```
+-------------------------------------------+
|함수번호| 기    능                          |
|--------|-----------------------------------|
|   0    | 마우스 드라이버 초기화            |
|   1    | 커서 보이기                       |
|   2    | 커서 숨기기                       |
|   3    | 커서 위치와 버튼의 상태 알아내기  |
|   4    | 커서의 위치 지정                  |
|   5    | 버튼이 눌러진 횟수 구하기         |
|   6    | 버튼이 떼어진 횟수 구하기         |
|   7    | 수평 최대/최소 위치 지정          |

------------------------------ 계  속 ----------------------------

|   8    | 수직 최대/최소 위치 지정          |
|   9    | 그래픽 커서 모양 정하기           |
|   10   | 텍스트 커서 지정                  |
|   11   | 마우스 카운터의 읽기              |
|   12   | 특수 상태에 따른 인터럽트 실행    |
|   13   | 라이트 펜 에뮬레이션 ON           |
|   14   | 라이트 펜 에뮬레이션 OFF          |
|   15   | Mickey/Pixel 비율 지정            |
```

| 16 | 조건에 따른 커서 숨기기 |
| 19 | 속도 조절 |
+--+
[참고] Mickey란 마우스가 이동해 다닐 때 그 움직임의 단위이다.

(1) 함수 0 : 마우스 드라이버의 설치 유무를 판단한다.

    입력 : AX = 0
    출력 : AX = 설치된 상태   0 : 설치되지 않았음
                                    -1 : 설치 되었음
------------------------------- 계 속 -----------------------------

            BX = 버튼의 갯수   2button : 2
                                    3button : 3
    - 마우스를 활성화시킨다.
    - 마우스 커서는 나타나지 않으며 성공하면 -1 을
                                       실패하면 0 을 리턴한다.

(2) 함수 3 : 커서 위치와 버튼의 상태를 알아낸다.

    입력 : AX = 03
    출력 : BX = 버튼의 상태
            CX = 수평 커서 위치
            DX = 수직 커서 위치
    - 비트 0 은 좌측 버튼, 비트 1 은 우측 버튼, 비트 2 는 가운데 버튼
    - 커서의 수평, 수직 위치는 화면에서 이동 가능한 커서의 최소, 최대
      치안에 있게 된다.
    - CX, DX 는 현재 마우스 커서의 X, Y 좌표를 얻는다.

(3) 함수 4 : 커서의 위치를 지정한다.

    입력 : AX = 04
            CX = 수평 커서 위치
            DX = 수직 커서 위치
    출력 : 없음
    - 마우스 커서를 (X, Y) 좌표로 이동시킨다.

(4) 함수 5 : 버튼 누르기에 관한 정보를 얻는다.

입력 : AX = 05
BX = 버튼
출력 : AX = 버튼 상태
BX = 버튼이 눌러진 횟수
CX = 최후에 버튼이 눌러졋을 때 커서 수평 위치
DX = 최후에 버튼이 눌러졌을 때 커서 수직 위치

- 입력측에서는 BX 가 체크할 버튼의 값을 나타내는데 이 값이 0 이면 왼쪽 버튼, 1 이면 오른쪽 버튼, 2 이면 가운데 버튼을 의미한다.

------------------------------ 계 속 ------------------------------

- 출력측에서는 AX는 버튼의 상태를 나타내는 비트가 기억되는데 비트 0 번에는 왼쪽 버튼, 비트 1 번에는 오른쪽 버튼, 비트 2 번에는 가운데 버튼의 상태가 기억되며 버튼이 눌러진 횟수는 BX 에서 받는데 1 - 32767 사이의 값을 가진다.

(5) 함수 6 : 버튼 떼기에 관한 정보를 얻는다.

입력 : AX = 06
BX = 버튼
출력 : AX = 버튼 상태
BX = 버튼이 떼어진 횟수
CX = 최종 떼어졌을 때의 수평 위치
DX = 최종 떼어졌을 때의 수직 위치

------------------------------ 계 속 ------------------------------

(6) 함수 7 : 수평의 커서 최대, 최소 위치를 지정한다.

입력 : AX = 07
CX = 최소 위치
DX = 최대 위치
출력 : 없음

- 커서가 수평으로 움직일 때 그 범위를 제한한다.

(7) 함수 8 : 수직의 커서 최대, 최소 위치를 지정한다.

입력 : AX = 08
BX = 최소 위치
CX = 최대 위치
출력 : 없음
- 커서가 수직으로 움직일 때 그 범위를 제한한다.

------------------------------ 계 속 ----------------------------

(8) 함수 9 : 그래픽 커서를 지정한다.

입력 : AX = 09
BX = 수평 Hot Spot
CX = 수직 Hot Spot
DX = 스크린 및 커서 마스크의 포인터
출력 : 없음
- 함수 9 번은 커서의 모양, 색, 커서의 중심을 지정한다.
(함수 1 번 호출)
- 여기서는 스크린 마스크와 커서 마스크의 내용을 가지고 커서의 모양을 결정하는데, 이 두 마스크의 내용을 마우스 루틴에 보내기 위해 그 값을 배열에 등록한 후 그 배열의 첫번째 내용을 DX 대신 넣으면 된다.

------------------------------ 계 속 ----------------------------

(9) 함수 10 : 텍스트 커서를 지정한다.

입력 : AX = 10
BX = 커서 선택
CX = 스크린 마스크값
DX = 커서 마스크값
- 텍스트 커서를 소프트웨어 또는 하드웨어적인 커서 상태로 선택한다.
- CX 와 DX 는 각 마스크가 들어간다.

(10) 함수 11 : 마우스의 카운터를 읽는다.

입력 : AX = 0b
출력 : BX = 가로축 이동간격 카운트
DX = 세로측 이동간격 카운트
- 가로측과 세로측의 이동간격 카운트를 보고한다.
- 마우스의 기종에 따라 100 미키/인치 또는 200 미키/인치 단위로, 320 미키/인치 단위로 보고한다.

------------------------------ 계 속 ----------------------------

- 세로축과 가로축의 스탭 카운트는 좌 -> 우 방향, 그리고 상 -> 하의 방향에 대한 이동을 지정하는 변수의 값으로 -32768 ~ 32767 범위로 보고된다.
- X 의 값이 음수이면 왼쪽으로 이동한 것이고, Y 의 값이 음수이면 위쪽으로 이동한 것이다.
- X, Y의 값이 0 이면 마우스의 이동이 없었음을 나타낸다. 키보드와 함께 마우스를 지원할때 유료하다.

(11) 함수 15 : 커서 움직임의 속도를 정한다.

입력 : AX = 0f
CX = 수평 비율
DX = 수직 비율
- 이 함수는 미키(움직임의 단위)대 점의 비를 정한다.
- 즉, 하나의 이동단위에 몇 개의 점을 할당하느냐의 문제이며, 보통 표준값(디폴트)은 수평으로 8 이고 수직으로는 16 이다.
- X, Y 의 값이 작을수록 커서가 크게 움직인다.

ᄒI
마우스 인터 페이스 프로그램 리스트

```
/* FILE NAME : [mus_driv.c]                */
/* FUNCTIONS : Mouse Interface Funtions */
ᄒ
#include <dos.h>

#define MOUSE 0x33
#define ON 1
#define OFF 0

/* 마우스 드라이버의 설치 유무 테스트 */
```

```
tst_mous(mus_stas)
  int       *mus_stas;
{
  union     REGS     inregs, outregs;

------------------------------ 계  속 ----------------------------


  inregs.x.ax = 0;
  int86(MOUSE, &inregs, &outregs);
  if(outregs.x.ax)
    *mus_stas = 1;
  else
    *mus_stas = 0;
}

/* 커서 보이기 */
mus_show(mus_cusr)
  int     *mus_cusr;
{
  union     REGS     inregs, outregs;

  inregs.x.ax = 1;
  int86(MOUSE, &inregs, &outregs);
  *mus_cusr = ON;
}
------------------------------ 계  속 ----------------------------

/* 커서 숨기기 */
mus_hide(mus_cusr)
  int     *mus_cusr;
{
  union     REGS     inregs, outregs;

  inregs.x.ax = 2;
  int86(MOUSE, &inregs, &outregs);
  *mus_cusr = OFF;
}

/* 커서 위치와 버튼의 상태 알아내기 */
mus_gett(hor_posx, ver_posy, but_left, but_rigt, but_cntr)
```

```
  int      *hor_posx,
           *ver_posy,
           *but_left,
           *but_rigt,
           *but_cntr;
------------------------------ 계  속 ----------------------------

{
  union      REGS     inregs, outregs;

  inregs.x.ax = 3;
  inregs.x.bx = 0;
  int86(MOUSE, &inregs, &outregs);

 *hor_posx = outregs.x.cx;
 *ver_posy = outregs.x.dx;
 *but_left = (outregs.x.bx & 0x0001);
 *but_rigt = (outregs.x.bx & 0x0002)/2;
 *but_cntr = (outregs.x.bx & 0x0004)/4;
}

/* 커서의 위치 지정 */
mus_sett(hor_posx, ver_posy)
  int      hor_posx,
           ver_posy;
------------------------------ 계  속 ----------------------------

{

  union      REGS     inregs, outregs;

  inregs.x.ax = 4;
  inregs.x.bx = 0;
  inregs.x.cx = hor_posx;
  inregs.x.dx = ver_posy;
  int86(MOUSE, &inregs, &outregs);
}

/* 버튼이 눌러진 횟수 구하기 */
get_pres(mus_butt, mus_stas, mus_numb, mus_posx, mus_posy)
  int      mus_butt,
```

```
        *mus_stas,
        *mus_numb,
        *mus_posx,
        *mus_posy;
------------------------------ 계  속 ----------------------------

{
  union     REGS    inregs, outregs;

  inregs.x.ax = 5;
  inregs.x.bx = mus_butt;
  int86(MOUSE, &inregs, &outregs);

  *mus_stas = outregs.x.ax;
  *mus_numb = outregs.x.bx;
  *mus_posx = outregs.x.cx;
  *mus_posy = outregs.x.dx;
}

/* 버튼이 떼어진 횟수 구하기 */
get_rels(mus_butt, mus_stas, mus_numb, mus_posx, mus_posy)
  int     mus_butt,
        *mus_stas,
        *mus_numb,
------------------------------ 계  속 ----------------------------

        *mus_posx,
        *mus_posy;
{
  union     REGS    inregs, outregs;

  inregs.x.ax = 6;
  inregs.x.bx = mus_butt;
  int86(MOUSE, &inregs, &outregs);

  *mus_stas = outregs.x.ax;
  *mus_numb = outregs.x.bx;
  *mus_posx = outregs.x.cx;
  *mus_posy = outregs.x.dx;
}
```

------------------------------ 계 속 ----------------------------

```
/* 수평 수직 최대, 최소 위치 구하기 */
mus_area(hor_minx, hor_maxx, ver_miny, ver_maxy)
  int     hor_minx,
          hor_maxx,
          ver_miny,
          ver_maxy;
{
  union     REGS     inregs, outregs;

  inregs.x.ax = 7;
  inregs.x.cx = hor_minx;
  inregs.x.dx = hor_maxx;
  int86(MOUSE, &inregs, &outregs);
  inregs.x.ax = 8;
  inregs.x.cx = ver_miny;
  inregs.x.dx = ver_maxy;
  int86(MOUSE, &inregs, &outregs);
}
```

------------------------------ 계 속 ----------------------------

```
/* 마우스 카운터 일기 */
mus_move(hor_posx, ver_posy)
  int     *hor_posx,
          *ver_posy;
{
  union     REGS     inregs, outregs;

  inregs.x.ax = 0x0b;
  int86(MOUSE, &inregs, &outregs);
  *hor_posx = outregs.x.cx;
  *ver_posy = outregs.x.dx;
}

/* Mickey / Pixel 비율 지정 */
set_rato(hor_posx, ver_posy)
  int     hor_posx,
```

```
          ver_posy;
------------------------------ 계  속 -----------------------------

{
  union      REGS     inregs, outregs;

  inregs.x.ax = 0x0f;
  inregs.x.cx = hor_posx;
  inregs.x.dx = ver_posy;
  int86(MOUSE, &inregs, &outregs);
 }
```

3. 마우스 커서의 종류

(1) 그래픽 커서

그래픽 커서는 그래픽 모드에서만 나타나며, 그 크기는 16 * 16 도트로서 256 개의 도트로 구성되며 일반적으로 화살표 모양을 하고 있다.이 모양은 32 바이트의 스크린 마스크와 32 바이트의 커서 모양을 정의하고 사용자 임의로 모양을 바꿀 수 있다. `인터럽트 33H 9 번` 기능을 이용하면 그래픽 커서를 만들 수 있다. 그러나 이 기능은 EGA 나 CGA 등의 IBM 표준 그래픽 카드인 경우에만 적용된다. 따라서 비표준 허클레스 그래픽 보드인 경우에는 프로그램 상에서 마우스 커서를 만들어야 한다.

(2) 소프트웨어 텍스트 커서

화면이 텍스트 모드 상태에서 사용자가 원하는 모양의 커서를 지정한다.
그래픽 상태처럼 다양한 모양을 만들지 못한다. 그 이유는 문자의 속성을 가지고 만들기 때문이며 ASCII코드의 256개 문자 중 한개가 되기 때문이다.

(3) 하드웨어 텍스트 커서

하드웨어 텍스트 커서는 DOS 상태에서 나오는 커서 타입이며, 이는 램 상주 드라이버에서 미리 정의된 모양이 셋업된 화면 모드와 카드에 맞춰서 나타난다.

ᄒl
마우스 헨들링 방법 구현
ᄒ

```
/* FILE NAME : [mus_cntl.c]            */
/* FUNCTIONS : Control Mouse Events */

#include <graphics.h>
#include <bios.h>
#include <conio.h>
#define LEFT_BUTT 1
#define RIGT_BUTT 2
#define RSHT_FLAG 0x01
#define LSHT_FLAG 0x02
#define SHFT_FLAG 0x07
#define RIGHT_     77
#define LEFT_      75
#define DOWN_       80
#define UP_        72
```

------------------------------ 계 속 ----------------------------

```
#define _RIGHT (RIGHT_ + 256)
#define _LEFT  (LEFT_  + 256)
#define _DOWN  (DOWN_  + 256)
#define _UP    (UP_    + 256)
#define CLIP_ON 1
#define MIN_X 0
#define MIN_Y 0
#define MAX_X 719
#define MAX_Y 347
#define _LSHIFT (2+512)
#define _RSHIFT (1+512)
#define SHOW 5

extern
char      brush[],
          box[];
```

------------------------------ 계 속 ----------------------------

```
extern
int     key_code,
        mod_flag,
        cur_posx,  /* current mouse position X */
        cur_posy,  /* current mouse position Y */
        pox_step,  /* move : X  step */
        poy_step,  /* move : Y  step */
        mnu_coll,  /* collected menu */
        mnu_flag,  /* menu flag on/off */
        mus_bton,  /* mouse button flag */
        mus_drag,  /* mouse drag flag */
        min_posx,  /* min mouse pos_X */
        min_posy,  /* min mouse pos_Y */
        max_posx,  /* max mouse pos_X */
        max_posy;  /* max mouse pos_Y */
```

------------------------------ 계 속 ----------------------------

```
/* 그래픽 초기화 */
ini_grph()
{

  int          grp_driv,
               grp_mode;
  grp_driv = DETECT;
  initgraph( &grp_driv, &grp_mode, "");
  cleardevice();
  setviewport(MIN_X, MIN_Y, MAX_X, MAX_Y, CLIP_ON);
}

/* 마우스 드라이버 초기화 *.
ini_muse()
{
  int          mus_stus;
```

------------------------------ 계 속 ----------------------------

```
    tst_mous(&mus_stus);
    min_posx = 135;
    min_posy = 56;
    max_posx = 530;
    max_posy = 287;

    if(mus_stus)
      set_rato(8,6);
}

icn_muss()
{
  int        mus_curs,
             mus_butt,
             mus_stus,
             mus_numb,
             mus_posx,
```

------------------------------ 계 속 ----------------------------

```
            mus_posy,
            but_left,
            but_rigt,
            but_cntr;

  while(1){
  while(!kbhit()){       /* No key pressed */
    und_brsh();
    switch(key_code) {
      case _LSHIFT:      /* mouse click */
      case _RSHIFT:
        mus_drag = 0;
        ctl_area();
        break;
      default:           /* check to mouse area */
        mus_drag = -1;
        mnu_flag = 0;
```

------------------------------ 계 속 ----------------------------

```
        cur_posx += pox_step;
        cur_posy += poy_step;
        if(cur_posx < min_posx)
           cur_posx = min_posx;
        else if(cur_posx > max_posx)
           cur_posx = max_posx;
        if(cur_posy < min_posy)
           cur_posy = min_posy;
        else if(cur_posy > max_posy)
           cur_posy = max_posy;
        break;
    }
    drw_brsh();
    key_code = get_muse();
  }
  bioskey(0);
  }
}
```

------------------------------ 계 속 ----------------------------

```
drw_brsh()
```

```
{
     putimage(cur_posx-4, cur_posy-4, brush, XOR_PUT);
}

und_brsh()
{
     putimage(cur_posx-4, cur_posy-4, brush, XOR_PUT);
}

get_muse()
{
    static
    int       old_butt = 0;
    int       mus_butt,
    ch,
```

------------------------------ 계 속 ----------------------------

```
              mov_posx,
              mov_posy,
              but_numb,
              ret_valu;
    do {
      if(kbhit()) {
        bioskey(0);           /* Keep out Key press */
        continue;
      }
      mus_move(&mov_posx, &mov_posy);
      if( !mov_posx & !mov_posy ) { /* No move + Click => Put Image */
        mus_drag = -1;
        get_pres(0, &mus_butt, &but_numb, &mov_posx, &mov_posy);
        if(old_butt == LEFT_BUTT && !mus_butt){
          mus_bton = 0;
          mus_drag = mod_flag = old_butt = 0;
          return(512+LSHT_FLAG);
```

------------------------------ 계 속 ----------------------------

```
        } else
        if(old_butt == RIGT_BUTT && !mus_butt){
```

```
        mus_bton = 1;
        mus_drag = mod_flag = old_butt = 0;
        return(512+RSHT_FLAG);
      } else {
        mnu_flag = 0;
        mus_bton = mus_drag = -1;
        old_butt = mus_butt;
        continue;
      }
    }
    else {       /* Move */
        pox_step = mov_posx;
        poy_step = mov_posy;
        get_pres(0, &mus_butt, &but_numb, &mov_posx, &mov_posy);
```

------------------------------ 계 속 ----------------------------

```
        if(abs(pox_step) > abs(poy_step))
          /* X : +(right moved) */
          ret_valu = (pox_step > 0 ? _RIGHT : _LEFT);
        else
          /* X : -(left moved ) */
          /* Y : +(down moved ) */
          /* Y : -(up moved    ) */
          ret_valu = (poy_step > 0 ? _DOWN : _UP);

        switch(mus_butt){
          case LEFT_BUTT:    /* move + drag */
            mus_drag = 1;
            mus_bton = 0;
            ctl_area();
            break;
```

------------------------------ 계 속 ----------------------------

```
          case RIGT_BUTT:
            mus_drag = 1;
            mus_bton = 1;
```

```
                ctl_area();
                break;
             default:
                mus_bton = -1;
                break;
          }
          return(ret_valu);
       }
   }while(1);
}
```

I. 유틸리티(Uitility)

তI
터보 C 에 부속된 유틸리티(7 가지)
ত

(1) CPP(터보 C 프리프로세서)

(2) MAKE(TOUCH 유틸리티를 포함 : 별도의 프로그램 관리기)

(3) TLINK(터보 링크)

(4) TLIB(터보 라이브러리 관리기)

(5) GREP(파일 탐색 기능)

(6) BGIOBJ(그래픽 드라이버와 폰트를 위한 변환 유틸리티)

(7) OBJXREF(오브젝트 모듈 관련 참조)

1. CPP 터보 C 프리프로세서

인클루드 파일이나 매크로로 정의를 정개한 C 소스 프로그램의 리스팅

파일을 출력한다. CPP는 디폴트 옵션으로 똑같은 TURBOC.CFG를 읽어들

이며 TCC와 같은 명령행 옵션을 받아들인다.

CPP에 의해 처리된 각 파일의 출력은 현재의 디렉토리(또는 -n 옵션에

의해 출력 디렉토리)에 있는 파일로 작성되며 이때 파일명은 소스명과

같고 확장자 .I 가 붙는다. 이 출력 파일은 소스파일에 있는 모든 행

과 인클루드 파일을 갖는 텍스트 파일이 된다.

1.1 마크로 프리프로세서 CPP

CPP에 `-P` 옵션을 부가하면 앞에 소스 파일명과 행번호가 첨가되고,

그러나 `-P-`가 주어지면 이 행번호를 첨가하지 않는다. 이 옵션을

off로 하면 CPP는 마크로 프리프로세서로 사용될 수 있다. 그러면

.I 파일은 TC 또는 TCC로 컴파일할 수 있게 된다.

[예]
소스 파일 : source.c

```
#define FILENAME "Turbo C"
#define BEGIN {
#define END    }
```

----------------------- 계 속 ---------------------

```
main()
BEGIN
   printf("%s\n",FILENAME);
END
c:\>cpp -P source.c    /* `-P` 사용 */
결과 : source.c 2:
        source.c 3:
        source.c 4:
        source.c 5:
        source.c 6: main()
        source.c 7: {
```

```
        source.c 8:     printf("s\n","Turo C");
        source.c 9: }

c:\>cpp -P- source.c    /* `-P-` 사용 */
결과 : main()
        {
          printf("s\n","Turo C");
        }
```

2. MAKE 유틸리티

터보 C의 MAKE 유틸리티는 적당한 명령을 제공하는 지능적인 프로그램

관리자로 사용자의 프로그램을 가장 최근의 상태로 유지하는데 필요한

모든 작업을 수행한다.

2.1 MAKE 수행

1) 작성한 특정 MAKEFILE을 읽는다. 이 파일은 실행파일(.EXE)를 만들기 위해서 어느 (.OBJ)와 라이브러리 파일을 링크(LINK)할 것인가를 알려준다. (.OBJ)파일을 생성하기 위해서 어떤 소스와 헤더파일을 컴파일 할 것인가를 알려준다.

------------------------ 계 속 ----------------------

2) 소스와 헤더 파일의 시간과 날짜를 가지고 각 (.OBJ) 파일의 시간과 날짜를 검사한다. 만일 이들 시간과 날짜가 (.OBJ) 파일의 그것보다 나중인 경우에 MAKE는 이 파일이 수정된 것을 알고 (.OBJ) 파일이 다시 컴파일되어야 하는 것을 안다.

3) (.OBJ) 파일을 다시 컴파일 하기 위해서 TCC를 호출한다.

4) 일단 모든 (.OBJ) 파일의 종속성이 검사되고 나면 실행파일의 시간과 날짜에 대해서 각 (.OBJ) 파일의 시간과 날짜를 검사한다.

5) 만일 (.OBJ) 파일이 (.EXE) 파일보다 늦은 경우에는 (.EXE) 파일을 다시 생성하기 위해서 터보 링커인 TLINK를 호출한다.

[참고] MAKE는 사용자의 프로그램을 현재의 상태로 유지시키는 것 이외 백업을 작성하고, 파일을 각기 다른 서브 디렉토리로부터 가져오며, 사용할 데이터 파일이 변경되도록 하는 프로그램을 자동으로 실행시켜 준다.

[예제]

.c 파일	헤더파일
STARLIB.C	None
GSPARSE.C	STARDEFS.H
GSCOMP.C	STARDEFS.H,STARLIB.H
GETSTARS.C	STARDEFS.H,STARDEFS.H,GSPARSE.H,GSCOMP.H

টI
GETSARS.EXE(medium 데이터 모델 가정)를 생성 과정
ট

```
tcc -c -mm -f starlib
tcc -c -mm -f gsparse
tcc -c -mm -f gscomp
tcc -c -mm -f getstars
tlink lib\com starlib gsparse gscomp getstars, \
    getstars, getstars, lib\emu lib\mathm lib\cm
```

[주의]
DOS는 TLINK 명령행이 한 행에 맞추도록 요구한다. 앞 페이지에서와 같이 너무 긴 경우에는 그 행 마지막에 백스래쉬(\)를 사용하여 다음 행으로 계속할 수 있다.

টI
앞 페이지 예제의 파일 종속성 분석
ট

1) GSPARSE,GSCOMP,GETSTARS : STARDEFS.H에 의존한다.
   만약 STARDEFS.H를 변경되면 앞의 세개 파일을 재컴파일해야 한다.
2) 마찬가지로 STARLIB.H가 변경되면 GSCOMP와 GETSTARS, GSPARSE.H 또는 GSCOMP.H가 변경되면 GETSTARS가 재컴파일 되어야 한다.
3) 물론 소스코드 파일에 변경되면(가령, STARLIB.C, GSPARSE.C 등) 그

파일은 재컴파일되어야 한다.

4) 재컴파일이 수행되면 링크도 다시 해야 한다.

2.2 MAKE를 이용한 단순화 작업

ㅠJ3.2.1 Makefile의 작성

Makefile은 파일의 종속성과 이때에 필요한 명령어 리스트를 결합한다.

[예제]

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
    tlink lib\com starlib gsparse gscomp getstars, getstars, \
            getstars, lib\emu lib\mathm lib\cm
getstars.obj: getstars.c stardefs.h starlib.h gscomp.h gsparse.h
    tcc -c -mm -f getstars.c
gscomp.obj: gscomp.c stardefs.h starlib.h
    tcc -c -mm -f gscomp.c
gsparse.obj: gsparse.c stardefs.h
    tcc -c -mm -f gsparse.c
starlib.obj: starlib.c
    tcc -c -mm -f starlib.c
```

[해석] 앞 페이지 예제 해석

1) 파일 GETSTARS.EXE는 4개의 파일(즉, GETSTARS.OBJ, GSCOMP.OBJ, GSP-ARSE.OBJ)과 STARLIB.OBJ에 종속된다. 이 4 개중 어느 하나가 변경되면, GETSTARS.EXE는 TLINK 명령을 사용해서 재링크해야 한다.

2) 파일 GETSTARS.OBJ는 5개의 파일(즉, GETSTARS.C STARDEFS.H, STARLIB.H, GSCOMP.H)과 GSPARSE.H에 종속된다. 이들 중 어느 하나가 변경되면 GETSTARS.OBJ는 제공되는 TCC 명령을 사용해서 재컴파일해야 한다.

3) 파일 GSCOMP.OBJ는 3개의 파일(즉, GSCOMP.C, STARDEFS.H 그리고 STAR-LIB.H)에 종속된다. 이들 중 어느 하나가 변경되면 GSCOMP.OBJ는 제공되는 TCC 명령을 사용해서 재컴파일해야 한다.

-------------------------- 계　속 ----------------------------

4) 파일 GSPARSE.OBJ는 2개의 파일(즉, GSPARSE.C,STARDEFS.H)에 종속된다.
이 2개의 파일 중 어느 하나가 변경되면 제공되는 TCC 명령을 사용해서
재컴파일해야 한다.

5) 파일 STARLIB.OBJ는 오직 1 개의 파일(즉, STARLIB.C)에만 종속된다.
만일 STARLIB.C가 변경되는 경우에는 TCC를 통해 재컴파일시켜야 한다.

ㅎJ2.2.2 Makefile의 사용

앞에서 만든 Makefile을 가지고 사용해 보자.(소스 파일이 있다고 가정)

```
make
```

1) MAKE는 Makefile을 찾는다.

2) GETSTARS.EXE의 종속성을 기술하면서 첫째행에 읽어들인다.

-------------------------- 계　속 ----------------------------

3) 2)는 GETSTARS.EXE가 최근의 상태로 있는 것인지의 여부를 검사하는
것이다.

3) 또한 GETSTARS.EXE가 종속되는 파일인 GETSTARS.OBJ,GSCOMP.OBJ,GSP
-ARSE.OBJ 그리고 STARLIB.OBJ에 대해 각각 똑같은 것을 검사하도록
요구한다.

4) .OBJ 파일을 수정하기 위해서 필요한 TCC에 대해 다양한 호출이 수행
되고 GETSTARS.EXE의 최신 버전을 생성하기 위해서는 TLINK 명령의
실행으로 끝낸다.

5) GETSTARS.EXE와 모든 .OBJ 파일이 이미 존재하는 경우에 MAKE는 각
.OBJ 파일의 가장 최근에 변경된 시간과 날짜를 종속 파일의 시간 및
날짜와 비교한다.

--------------------------- 계 속 ----------------------------

6) 만일 종속파일이 .OBJ 파일보다 최근의 것일 경우에 MAKE는 최근의
   .OBJ 파일이 생성된 이래 변경이 발생한 것을 알고 TCC 명령을 실행
   시킨다.

7) MAKE가 .OBJ 파일을 변경하면 이것이 GETSTARS.EXE의 시간과 날짜를
   .OBJ 파일과 비교할 때 변경된 GETSTARS.EXE를 만들기 위해서 TLINK
   명령을 실행해야 함을 알게 된다.

ㅠ2.2.3 단계별 예제 설명

지금까지 설명한 내용을 명확하게 이해하기 위해서 단계별로 예제를 들어
설명한다.

1) GETSTARS.EXE와 모든 .OBJ 파일이 있고, GETSTARS.EXE는 어떤 .OBJ 화
   일 보다도 최신의 것이고, 마찬가지로 각각의 .OBJ 파일은 그것의 종속
   파일보다 최신의 것이라 가정한다.

--------------------------- 계 속 ----------------------------

```
make
```

아무것도 일어나지 않는다. 그 이유는 아무것도 변경할 필요가 없기
때문이다.

2) 이제 STARLIB.C와 STARLIB.H를 변경해서 몇개의 상수값을 바꾸었다고
   가정하자. MAKE는 STARLIB.C가 STARLIB.OBJ보다 최근의 것임을 알기
   때문에 다음 명령을 행한다.

```
tcc -c -mm -f starlib.c
```

3) 이때 STARLIB.H가 GSCOMP.OBJ보다 최근의 것임을 알기 때문에 다음 명
   령을 행한다.

```
tcc -c -mm -f gscomp.c
```

-------------------------- 계 속 ----------------------------

4) STRLIB.H 역시 GETSTARS.OBJ 보다 최근의 것이기 때문에 다음 명령을 행한다.

```
tcc -c -mm -f getstars.c
```

5) 마지막으로 이들 3개의 명령 때문에 파일 STARLIB.OBJ, GSCOMP.OBJ 그리고 GETSTARS.OBJ는 모두 GETSTARS.EXE보다 최근의 것이므로 최종적으로 MAKE는 다음 명령을 행한다.

```
tlink lib\com starlib gsparse gscomp getstars, getstars, \
      getstars, lib\emu lib\mathm lib\cm
```

6) 모든 것을 링크시키고 새로운 버젼의 파일 GETSTARS.EXE를 생성한다.

2.3 Makefile의 구성요소와 그 작성

makefile은 MAKE가 사용자의 파일을 최신의 상태로 유지할 수 있도록 하는데 필요한 정의와 관계를 구성하고 있다.

ㅋJ2.3.1 Makefile의 구성요소

(1) 주석(comments)
(2) 명시적 규칙(explicit rules)
(3) 묵시적 규칙(implicit rules)
(4) 마크로 정의
(5) 지시명령(파일 포함, 조건실행, 에러 탐색, 마크로 해제)

이 중에서 (2) 명시적 규칙만 이해하면 MAKE를 사용할 수가 있다. 여기서는 (1)과 (2)에 대해서만 다루도록 하겠다. 좀더 깊게 알고 싶으면 여러 참고서적을 보기 바란다.

ㅎl
1) 주석(comments)
ㅎ
(1) 주석은 # 문자로 시작된다.
(2) # 뒤에 있는 행의 나머지는 MAKE에 의해 무시 된다.
(3) 어느 곳에나 위치할 수 있고 특정 열에서 시작할 필료는 없다.
(4) 백슬래쉬(\)를 사용해서 주석을 다음 행으로 계속할 수 없다.
(5) (4)대신 각 행에 #을 사용하여 계속할 수 있다.
(6) # 앞에 오는 경우 더 이상 그 행에서 마지막 문자가 될수 없다.
(7) # 뒤에 오는 경우 주석의 일부가 된다.

[예제]

```
# makefile for GETSTARS.EXE
# does complete project maintenance
getstars.exe: getstars.obj gscomp. obj gsparse.obj starlib.obj
```

--------------------------- 계 속 ----------------------------

```
# can't put a comment at the end df the next line
   tlink lib\com starlib gsparse gscomp getstars, getstars, \
         getstars, lib\emu lib\mathm lob\cm
# legal comment
#  can't put a comment between the next two lines
getstars.obj:  getstars.c stardefs.h starlib.h gscomp.h gsparse.h
   tcc -c -mm -f getstars.c    # you can put a comment here
```

[설명]

(1) 주석은 샤프(#) 기호로 쓰기 시작한다.
(2) 샤프(#) 기호 이하의 부분은 주석이므로 MAKE의 실행시에는 무시된다.
(3) 행단위로 되어 주석 자체를 백슬래쉬(\)를 사용해서 다음 행에 계속할 수 없다.
(4) 2행 이상에 걸쳐 있는 긴 주석의 경우에는 각 행마다 샤프(#)기호를 사용하지 않으면 안된다.

--------------------------- 계 속 ----------------------------

(5) 백슬래쉬(\)뒤에 샤프(#)기호가 오면 본래의 백슬래쉬(\)의 의미가 없어지므로 이와 같은 주석을 기술할 수 없다.
(6) 샤프(#)기호 뒤에 있는 백슬래쉬(\)는 본래의 백슬래쉬로서의 의미는 없게 되고 단순히 주석의 일부로 해석된다.
(7) 기본적으로 어느 행에 기술해도 상관이 없지만 백슬래쉬(\)로 2행으로 나누어져 있는 행 사이에 들러갈 수 없다.

ㅎl
2) 명시적 규칙(Explicit Rules)
ㅎ
명시적 규칙을 이해하면 makefile을 작성할 수가 있다.

```
[형식] target [target ...]: [source source ...]
              [command]
              [command]
              ...
```

-------------------------- 계 속 ---------------------------

[설명]

(1) target은 변경될 목적파일이다.
(2) souce는 target을 종속시키는 파일이다.
(3) command는 DOS 명령(.BAT 파일의 호출과 .EXE 파일의 실행 등)이 된다.
(4) target은 반드시 그 행의 시작 지점(1컬럼)에 있어야 한다.
(5) command는 최소한 하나의 공백 혹은 탭이 앞에 오면서 들여쓰기를 해야 한다.
(6) 명시적 규칙에는 뒤에 콜론(:)이 오는 target [target...] 만으로 구성될 수 있다.
(7) MAKE가 명시적 규칙을 만나면, 먼저 소스파일이 makefile 의 어딘가에 target으로 되어 있는가 아닌가를 체크한다.

(8) 명시적 규칙은 한개 이상의 목적파일명과 몇개의 소스파일(없을 수도 있다.) 그리고 수행될 임의의 명령을 정의한다.

-------------------------- 계 속 ---------------------------

[고려사항]

(1) 명령이 있는 명시적 규칙이 존재하는 경우, target이 종속되는 파일
만이 명시적 규칙에 리스트 된다.

(2) 명령이 없는 명시적 규칙의 경우, target은 명시적 규칙에서 주어진
파일에 종속되며 또한 이들은 그 target에 대한 묵시적 규칙에 일치
하는 파일에 종속된다.

[예제]
```
myprog.obj: myprog.c
    tcc -c    myprog.c
prog2.obj: prog2.c include\stdio.h
    tcc  -c -k prog2.c
prog.exe: myprog.c prog2.c include\stdio.h
    tcc -c myprog.c
    tcc -c -k prog.c
    tlink lib\cos myprog prog2, prog, lib\cs
```

-------------------------- 계 속 ---------------------------

(3) 처음 명시적 규칙은 MYPROG.OBJ가 MYPROG.C에 종속하며 주어진 TCC
명령을 실행함으로써 MYPROG.OBJ가 생성됨을 나타낸다.

(4) 두번째 규칙은 PROG2.OBJ가PROG2.C와 INCLUDE 디렉토리에 있는 STDI
-O.H에 종속하며, TCC 명령에 의해 생성됨을 나타낸다.

(5) PROG.EXE가 MYPROG.C, PROG2.C 그리고 STDIO.H에 종속하며 이 3개중
하나가 변경되면 PRO.EXE는 주어진 일련의 명령에 의해 재작성됨을
나타낸다.

(6) (5)는 MYPROG.C만 변경되는 경우에도 PROG2.C가 재컴파일되어야 하기
때문에 불필요한 작업을 하므로 아래와 위치시키면 된다. 그러면 이
파일들만 재컴파일될 필요가 있게 된다.

```
prog.exe: myprog.obj prog2.obj
     tlink lib\cos myprog prog2, prog, , lib\cs
```

2.4 MAKE의 사용

ㆆl

2.4.1 명령행 문법

ㆆ

```
make
```

DOS 프롬프트상에서 make를 입력하면 MAKEFILE을 찾는다. 만일 찾지 못하면 MAKEFILE.MAK를 찾고 이것도 없으면 에러 메시지를 내고 중단한다.

```
make -fstars.mak
```

위와 같이 파일을 상용할 경우에는 MAKE에 파일 옵션(-f)을 주고 일반적인 문법은 아래와 같다.

```
make option option ... target target ...
```

위에서 option은 MAKE 옵션이며 target은 명시적 규칙에 의해 처리되는 target 파일명이 된다.

[문법 규칙]

(1) make 단어 뒤에는 공백이 오고 다음에 옵션 리스가 온다.
(2) 각각의 옵션은 공백으로 인접한 옵션돠 분리된다.
(3) 옵션 리스트 뒤에 공백이 오고 다음에 임의의 target 리스트가 온다.
(4) 각각의 target은 공백으로 분리된다.

ㆆl

2.4.2 MAKE의 중단

ㆆ

MAKE는 ctrl+Break와 ctrl+C 를 통해서 실행명령이 중지된다.

ㆆl

2.4.3 BUILTINS.MAK 파일

ᄐ
이것은 자주 사용되는 마크로나 규칙을 일일이 정의하기가 불편할때 사용
된다. 이 파일이 존재하면 MAKE는 우선 MAKEFILE을 평가하기 전에 이 BUI
-LTINS.MAK를 평가한다. 이 파일은 컴퓨터상의 어디에서도 파일에서 일반
적으로 사용될 수 있는 규칙이나 마크로를 위해 만들어진 것이다. 그러므
로 BUILTINS.MAK 파일이 존재해야 하는 요구 조건은 없다.

ᄐI
2.4.4 MAKE가 BUILTINS.MAK 또는 Makefile을 탐색하는 방법 순서
ᄐ
(1) 현재의 디렉토리 탐색한다.
(2) MAKE.EXE가 들어 있는 터보 C 디렉토리 탐색한다.(이 파일은 MAKE.
EXE와 같은 디렉토리에 넣어야 한다.)

[설명] MAKE는 현재의 디렉토리에서 !include 파일을 탐색한다. 이때 -I
옵션(include)을 사용하면 지정 디렉토리를 탐색하게 된다.

ᄐI
2.4.5 MAKE 명령행 옵션
ᄐ
여기서 주의할 것은 대문자와 소문자가 각기 서로 다른 의미를 갖는다.

(1) -a : 자동 종속성 검사를 수행한다.
(2) -Didentifier : 이름에 대해서 1을 정의한다.
(3) -Diden=string : 이름에 대해서 지정된 문자열을 정의한다.
(4) -Idirectory : 인클루드하는 파일을 탐색할 디렉토리를 지정한다.
(5) -Uidentifier : 마크로 이름의 정의를 취소한다.
(6) -s : MAKE파일에 기술되 명령실행시에 명령행에 나타나지
않는다. 디폴트 지정은 명령행에 출력한다.
(7) -n : 명령의 실행은 하지 않고 makefile을 해석한다.
이것은 makefile의 디버깅시 유효하다.
(8) -ffilename : makefile 을 디폴트인 MAKEFILE 이외의 파일명으로
한다.
(9) -? 또는 -h : makefile 의 help 메시지를 출력한다.

3. 터보 LINK

터보 C 통합환경(TC)에서는 링커가 내장되어 있다. 터보 C 의 명령행 버젼

(TCC)에 대해서도 링커는 별도의 분리된 프로그램으로 호출한다. 이 분리

된 프로그램인 TLINK 는 `독리된 링커`로서 사용될 수있다.

디폴트로 TCC는 컴파일이 성공적일 경우에 TLINK를 호출하는데 이때 TLINK

는 오브젝트 모듈과 라이브러리 파일을 결합해서 실행가능한 파일을 생성한

다.

3.1 TLINK의 호출

DOS 명령행에서 TLINK(인수르 갖거나 또는 인수없이)를 입력하므로서 호
출할 수 있고 인수와 옵션의 요약 리스트는 다음과 같다.

```
Turbo Link  Version 4.0 Copyright (c) 1991 Borland International
Syntax: TLINK objfiles, exefile, mapfile, libfiles, deffile
@xxxx indicates use response file xxxx
Options: /m = map file with publics
         /x = no map file at all
         /i = initialize all segments
         /l = include source line numbers
         /L = specify library search paths
         /s = detailed map of segments
         /n = no default libraries
         /d = warn if duplicate symbols in libraries
         /c = lower case significant in symbols
         /3 = enable 32-bit processing
```

------------------------- 계 속 -----------------------

```
         /v = include full symbolic debug information
         /e = ignore Extended Dictionary
         /t = create COM file (same as /Tc)
```

/o = overlay switch
/P[=NNNNN] = pack code segments
/A=NNNN = set NewExe segment alignment factor
/ye = expanded memory swapping
/yx = extended memory swapping
/C = case sensitive exports and imports
/Txx = specify output file type
/Tdx = DOS image (default)
/Twx = Windows image
 (third letter can be c=COM, e=EXE, d=DLL)

[설명] 앞 페이지에서
Syntax: TLINK objfiles, exefile, mapfile, libfiles, deffile
은 사용자가 파일의 형태를 컴마로 분리하면서 주어진 순서대로
파일명을 주어야 하는 것을 지정한다.

------------------------- 계 속 -----------------------

[예] tlink /c mainline wd ln tx,fin,mfin,lib\comm lib\support

ㆆI
TLINK의 의미 해석
ㆆ
(1) 링크하는 동안 활자체(소문자,대문자)는 중요하다.(/c)
(2) 링크될 .OBJ 파일은 MAINLINE.OBJ, WD.OBJ, LN.OBJ, TX.OBJ이다.
(3) 실행 가능한 프로그램명은 FIN.EXE가 된다.
(4) 맵파일은 MFIN.MAP이다.
(5) 링크되는 라이브러리 파일은 COMM.LIB와 SUPPORT.LIB가 되고 이들은
   모두 서브 디렉토리 LIB에 있다.

ㆆI
확장자를 갖지 않은 파일명에 대한 확장자
ㆆ
(1) 오브젝트 파일을 위해 .OBJ
(2) 실행가능한 파일을 위해 .EXE
(3) 맵파일을 위해 .MAP
(4) 라이브러리 파일을 위해 .LIB

ㆆI
옵션(options)에 대한 설명
ㆆ
(1) /t : 실행가능한 파일의 `디폴트(.EXE)`가 아닌 `.COM`으로 된다.
(2) /x : 맵파일을 생성하지 않도록 한다.

(3) /m : 맵파일에 public 기호를 포함한다.
(4) /s : 맵파일에 상세한 세그먼트 맵이 된다.

ㅎI
맵파일명의 결정 참고사항
ㅎ
.MAP 파일에 지정되지 않은 경우에는 `실행가능한 파일(.EXE)`에 .MAP확장자를 붙임으로서 맵파일을 만든다. 만약 .EXE 명이 주어지지 않은 경우에는 `첫번째 .OBJ 파일명`으로부터 맵파일을 만든다.

ㅎI
3.1.1 응답파일의 사용
ㅎ
(1) 플러스기호(+)의 사용은 오브젝트나 라이브러리 파일의 긴 리스트를 여러 행에 걸쳐서 기입하고자 할때 한행의 끝에 기호(+)를 붙인다.

(2) 오브젝트 파일, 실행파일, 맵파일, 라아브러리 등을 각각의 행에 분리하고자 할때는 구성요소 분리용으로 사용되는 컴마(,)를 생략해야 한다.
ㅎI
응답파일(FINRESP) 만든다.
ㅎ
방법 1]

```
/c mainline wd+
ln tx,fin
mfin
lib\comm lib\support
```

그리고 TLINK 명령은 다음과 같이 한다.

--------------------- 계 속 ---------------------

```
tlink @finresp
```

여기서 alt문자(@)는 응답파일을 나타낸다.

[방법 2] 링크 명령을 복수의 응답파일에 기입할 수 있다.

```
+------------------------+
| LISTOBJS | mainline+    |
|          | wd+          |
|          | ln  tx       |
| LISTLIBS | lib\comm+    |
|          | lib\support  |
+------------------------+
```

그리고 TLINK 명령은 다음과 같이 한다.

```
tlink /c @listobjs,fin,mfin,@listlibs
```

3.1.2 TLINK와 터보 C모듈의 사용

터보 C는 6개의 다른 메모리 모델(tiny,small,compact,medium,large,huge)를 지원한다. TLINK를 사용해서 실행가능한 파일을 생성할 때 반드시 사용되는 메모리 모델에 대한 초기화 모듈과 라이브러리를 포함해야 한다.

[형식] tlink COx <myobjs>,<exe>,[map],<mylibs> [emu fp87 mathx] Cx
[설명] <myjobs> : 링크될 .OBJ 파일
<exe> : 실행가능 파일로 주어진 이름
[map] : 맵파일로 주어진 이름(임의 지정)
<mylibs> : 링크시에 포함될 파일

- COx : 메모리 모델 t,s,c,m,l 또는 h에 대한 초기화 모듈
- emu/fp87 : 부동 소숫점 라이브러리 (하나를 선택)
- mathx : 메모리 모델 s,c,m,l 또는 h 에 대한 math 라이브러리
- Cx : 메모리 모델 s,c,m,l 또는 h에 대한 런타임 라이브러리

[주의] tiny 모델을 사용하는 경우 .COM파일을 생성하기 위해서 TLINK를 원할때도 역시 /t 옵션을 지정해야 한다.

(1) 초기화 모듈

초기화 모듈은 C0x.OBJ 이름을 갖고 여기서 x 는 단일 문자로 대응되는 모델 (t,s,c,m,l,h)을 나타낸다.

적절한 초기화 모듈과 링크되지 못했을 때는 어떤 식별자를 해석하지 못

하거나 아무 스택도 생성되지 않았음을 알리는 긴 에러 메시지 리스트가 발생한다.

이 모듈은 리스트에서 첫번째 오브젝트 파일로 나타나야 한다. 그리고 이 모듈은 프로그램의 다양한 세그먼트의 순서를 정렬시킨다.

이 모듈이 맨처음이 아닌 경우 프로그램 세그먼트는 올바르게 메모리상에 위치하지 않도록 프로그램 버그를 야기시키게 된다.

사용자는 TLINK 명령행상에서 명시적인 .EXE 파일명을 주어야 하다. 그렇지 않으면 사용자 프로그램명이 COx.EXE가 되어 원치 않는 것이 된다.

(2) 라이브러리

링크 명령에서는 또한 사용자 자신의 라이브러리 뒤에 대응되는 메모리 모델의 라이브러리 들이 포함되어야 한다.

이들 라이브러리는 반드시 특정한 순서로 나타나야 한다. 즉 부동소수점 라이브러리, math 라이브러리, 그리고 대응되는 런타임 라이브러리가 그것이다.

만일 사용자가 어떤 터보 C 그래픽 함수를 사용한다고 하면, 이때는 반드시 graphics.lib를 링크시켜야 한다.

터보 C의 두개의 부동소숫점 라이브러리는 프로그램의 메모리 모델과는 무관하다.

---------------------- 계 속 ---------------------

- 프로그램이 math coprocessor(8087 또는 80287)칩이 있거나 없는 기계에서도 모두 작동할 수 있도록 부동소숫점 에뮬레이션 로직을 포함하려는 경우 반드시 EMU.LIB를 사용해야 한다.

- 프로그램이 math corprocessor 칩을 갖는 기계에서만 항상 실행될 경우 FP87.LIB 라이브러리는 더 작고 더 빠른 실행가능 프로그램을 생성한다.

[주의]

부동소숫점 연산을 사용할 때는 반드시 C 런타임 라이브러리 앞에 에뮬레
이터와 math 라이브러리를 포함해야 한다.

ᄒᆝ
3.1.3 TLINK와 TCC의 사용
ᄒ
TLINK를 준비로서 독립된 터보 C 컴파일러인 TCC 를 사용할 수 있다.
TCC 는 명령행상에서 .OBJ와 .LIB 확장자를 명시적으로 갖는 파일명을
아래 예와 같이 주어야 한다.

```
tcc -mx mainfile.obj subl.obj mylib.lib
```

TCC는 COx.OBJ, EMU.LIB,MATHx.LIB 그리고 Cx.LIB 파일(이들은 각각 메모
리 모델 x 에 대한 초기화 모듈, 디폴트 887 에뮬레이션 라이브러리, math
라이브러리, 그리고 런타임 라이브러리가 된다.)과 함께 TLINK를 호출하게
된다.

이때 TLINK는 이들을 사용자 모듈(MAINLINE.OBJ,SUBI.OBJ)과 사용자 라이
브러리(MYLIB.LIB)와 함께 링크시킨다.

[주의] TCC가 TLINK를 호출할 때는 -1-c로 무효화되지 않는 한 항상 옵션
/c(문자체 인식 링크)를 사용한다.

3.2 TLINK 옵션

TLINK 옵션은 명령행상의 어디에서도 발생할 수가 있다. 옵션은 슬래쉬(/)
와 옵션을 지정하는 문자 (m,x,i,l,s,n,d,c,3,v,e 또는 t)로 구성된다.

하나 이상의 옵션을 줄 경우에 공백은 의미를 갖지 않는다.(즉, /m/c는
/m /c와 똑같다.)

ᄒᆝ

3.2.1 /x, /m, /s 옵션

ড

[기능] 항상 디폴트로 실행가능 파일의 맵을 생성한다.

[설명] 이 디폴트 맵은 프로그램에서 세그먼트 리스트, 프로그램 시작 어드레스 그리고 링크하는 동안에 발생되는 경고나 에러 메시지만을 포함한다.

- /m 옵션 : 어드레스의 증가순으로 소트된 맵 파일에 public 심볼 리스트를 추가한다.
- /s 옵션 : /m 옵션이 하는 것처럼 세그먼트, public심볼 그리고 프로그램 시작 어드레스를 갖는 맵파일을 생성하고 여기에다 상세한 세그먼트 맵을추가한다.

ডI

3.2.2 /I 옵션

ড

[기능] 소스코드 행번호를 위해 .MAP 파일에 섹션을 생성한다. 그러므로 행번호를 포함하게 된다.

[설명] 이 옵션을 사용하기 위해서는 -y 옵션(Line munbers...On)으로 컴파일 하여 .OBJ 파일을 만들어 놓아야 한다. 그러나 /x 옵션을 사용해서 TLINK로 하여금 맵파일의 생성을 억제하는 경우 이 옵션은 무시된다.

ডI

3.3.3 /i 옵션

ড

[기능] 세그먼트가 데이터 레코드를 포함하고 있지 않은 경우에도 초기화되지 않은 빈 세그먼트가 실행가능 파일에 출력되도록 하게 한다.

ডI

3.3.4 /n 옵션

ড

[기능] 링커가 몇몇 컴파일러에 의해 지정된 디폴트 라이브러리를 무시하도록 한다.

[설며] 이 옵션은 디폴트 라이브러리가 또다른 디렉토리에 있는 경우에 필요로 한다. 그 이유는 TLINK가 라이브러리 탐색을 지원하지 않기 때문이다.

ডI

3.3.5 /c 옵션

[기능] public과 외부 심볼의 대문자와 소문자를 구별한다.
[설명] 예를 들면 디폴트로 TLINK는 fred, Fred 그리고 FRED를 같은것으로 간주하지만 /c 옵션은 이들을 다른 것으로 한다.

3.3.6 /d 옵션

[기능] 심볼의 중복을 경고한다. 디폴트는 OFF이다.

3.3.7 /e 옵션

[기능] 터보 C의 라이브러리에 포함되어 있느 확장된 사전 (Extended dic-tionary)을 무시한다.

[설명] 터보 C 에 포함되어 있는 라이브러리 파일은 TLINK가 보다 고속으로 이들 라이브러리를 링크하게 해주는 정보를 가지고 있는 확장 사전을 포함한다.
이 확장사전은 TLIB와 함께 /E 옵션을 사용해서 다른 라이브러리 파일에도 추가될 수가 있다.

/e 옵션을 사용하는 이유 2 가지

(1) 확장사전이 사용될 때는 프로그램은 링크하기 위해 좀더 많은 메모리를 필요로 한다.
(2) /e가 사용되지 않는 경우 TLNIK 는 확장사전을 갖는 라이브러리에 포함된 어떠한 디버깅 정보도 무시하게 된다.

3.3.8 /t 옵션

[기능] tiny모델의 프로그램에서 .COM 파일을 생성한다.

[설명] tiny 메모리 모델에서 파일을 컴파일하고 이 스위치가 On으로 토글 되어 링크하면 TLINK는 통상의 .EXE 파일 대신에 .COM 파일을 생성한다. 그 실행파일에 대한 디폴트 확장자는 .COM이 된다.

[주의] .COM 파일은 크기가 64를 초과할 수가 없고 어떠한 상대적인 세그먼트 정돈(segment-relative fixups)도 없다. 또한, 스택 세그먼트도 정의하지 않으며 반드시 0:100H와 같은 시작 어드레스를 갖는다. 실행가능 파일에 대해 .COM이 아닌 다른 확장자 가령.BIN)가 사용될 때 그 시작 어드레스는 0:0 도는 0100H 중에 하나가 된다.

তI

3.3.9 /v 옵션

ত

[기능] TLINK 에게 실행가능 파일에 디버깅 정보를 포함하도록 지시한다.

তI

3.3.10 /3 옵션

ত

[기능] 80386 CPU의 32비트 코드를 포함하고 있는 하나 이상의 옵젝트 모듈의 링크를 지시한다.

[설명] 이러한 오브젝트 모듈은 TASM 이나 MASM 혹은 유사한 어셈블러로 어셈블된 것이다. 이 옵션은 TLINK의 메모리 요구를 메모리 중대시키고 링크 속도를 줄이기 때문에 필요할 때만 사용해야 한다.

তI

TLINK에 대한 중대한 제약사항

ত

(1) 중복(overlays)은 지원되지 않는다.

(2) 변수(Common variables)는 부분적으로만 지원된다. 이것을 해결하기 위해 public이 지원되어야 한다.

(3) 논리 세그먼트를 최대 약 4000개만 가질 수 있다.

(4) 같은 이름과 분류의 세그먼트들은 모두 결합되거나 아니면 모두 결합할 수 없다.(이는 어셈블러 프로그래머에게만 문제가 된다.)

(5) Microsoft C 나 Microsoft Fortran 으로 컴파일된 코드는 종종 TLINK 와 함께 링크될 수가 없다. 이것은 Microsoft 언어가 이들의 .OBJ 화일에서 도큐멘트 되지 않은 오브젝트 레코드 형식을 갖기 때문인데

TLINK는 이를 지원하지 않고 있다.

[참고] TLINK는 TASM. 터보 프롤로그 그리고 기타 다른 컴파일러 뿐만이
아니라 통합환경과 명령행 버젼의 터보 C와도 함께 사용되도록
설계되어 있다. 그러나 일반적인 MS-Li-nk를 대체하지는 않는다.

4. 터보 라이브러리 관리기(TLIB)

TLIB는 볼랜드의 터보 라이브러리 관리기이다. 이것은 개별 .OBJ 파일의

라이브러리리를 관리하는 유틸리티이다. 라이브러리는 오브젝트 모듈의

집합을 단일체로서 처리해주는 매우 편리한 방식이다. 터보 C에 포함된

라이브러리들은 TLIB로 작성되어있다. 따라서 사용자는 TLBI를 사용해서

자신의 라이브러리를 작성하거나 또는 터보 C 라이브러리, 사용자 자신의

라이브러리, 다른 프로그래머에 의해 제공된 라이브러리 및 기타 라이브

러리를 수정할 수가 있다.

TLIB의 사용 범위

(1) 오브젝트 모듈의 그룹으로부터 새로운 라이브러리의 생성할때
(2) 현재의 라이브러리에 오브젝트 모듈이나 다른 라이브러리를 추가할때
(3) 현재의 라이브러리로 부터 오브젝트 모듈을 제거할때
(4) 현재의 리아브러리로 부터 오브젝트 모듈의 추출할때
(5) 새로운 또는 현재의 라이브러리의 내용을 열거할때

TLIB의 부연 설명

(1) 현재 존재하는 라이브러리를 수정할때는 항상 .BAK 확장자를 갖는

원래 라이브러리의 복사본을 생성한다.

(2) 확장사전을 생성할 수 있다. 이것은 고속의 링크를 수행하는데 사용
될수 있다.

---------------------- 계 속 ---------------------

(3) 터보 C로 실행가능 프로그램을 생성하는데 필수적인 것은 아니다.
그러나 프로그래머 생산성의 유용한 도구가 된다.

(4) 사용자는 TLIB가 대형 개발 프로젝트에 있어서 없어서는 안될 필수적
인 것임을 알게 될 것이다.

(5) 다른 사람들에 의해 개발된 오브젝트 모듈 라이브러리로 작업을 하는
경우에도 필요할 때는 TLIB 를 사용해서 이들 라이브러리를 유지보수
할 수가 있다.

ㄲ4.1 오브젝트 모듈 라이브러리의 사용 장점

(1) 사용자가 사용하고 있는 파일이 정확하게 어느 것인가를 알아낼 수
있다.
(2) 사용자가 항상 모든 소스 파일을 포함하는 경우 발생되는 프로그램이
커지는 것을 막을 수 잇다.
(3) 링커의 수행을 고속으로 해준다.
(4) 각각의 오브젝트 파일들이 작을 때는 유용하다.

4.2 TLIB 명령행의 구성

TLIB는 DOS 프롬프트상에서 TLIB명령행을 입력함으로써 샐행할 수 있다.
[형식] tlib libname [/C] [/E] [operations] , [listfile]
[설명]
(1) tlib : TLIB를 호출하는 명령어이다.
(2) libname : 작성하거나 관리하려는 라이브러리의 DOS 패스명이다.
모든 TLIB 명령은 libname으로 주어져야 한다. 와일드
카드는 사용할 수 없고 TLIB는 아무것도 주어지지 않
으면 확장자를 .LIB로 간주한다. TCC와 TC의 project
make기능은 라이브러리 파일을 인지하기 위해서, .LIB
확장자를 요구하므로 사용자는 `.LIB`이외의 확장자를
사용하지 않는 것이 좋다.
(3) /C : 문자체 인식 플래그. /C를 지정하면 함수이름의 대문자

와 소문자를 같은 것으로 간주한다.

(4) /E : 확장된 사전을 생성한다.

(5) operations : TLIB가 수행하는 연산의 리스트이다.

(6) listfile : 라이브러리 내용을 리스트하는 listfile명이 주어질 경우, 반드시 컴마가 앞에 와야 한다.

ㅎl

4.2.1 연산 리스트(Operation List)

ㅎ

TLIB 가 어떻게 작동하는가를 기술하는 것으로서 항상 아래와 같이 특정한 순서로 연산을 적용한다.

- 모든 extract(*) 연산이 제일 먼저 수행되고
- 모든 remove(-) 연산이 다음에 수행되며
- 모든 add(+) 연산이 제일 나중에 수행된다.

(1) 파일과 모듈명

TLIB가 오브젝트 모듈 파일을 라이브러리에 추가할 때는 이 파일은 단순히 모듈로 호출된다. TLIB는 주어진 파일명을 가지고 드라이브,패스 그리고 확장자 정보를 분석해서 모듈의 이름을 찾는다.

TLIB는 항상 합리적인 디폴트를 가정한다. 예를 들면, 현재 디렉토리로부터 .OBJ 확장을 갖는 모듈을 추가하기 위해 사용자는 오직 모듈명만으 주면 된다(패스와 .OBJ는 줄 필요가 없다.). 파일 또는 모듈명에서 와일드카드는 사용할 수 없다.

(2) TLIB 연산

기호	이름	내 용
+	Add	지정된 파일을 라이브러리에 추가한다. 확장자가 없으면, 확장자를 .OBJ로 간주한다. 파일자체가 확장자를 .LIB를 갖는 경우에는 연산은 지정된 모든 모듈을 타겟 타겟에 추가한다. 추가된 모듈이 존재할 때는 메시지를 표시하고 새로운 모듈을 추가하지 않는다.
-	Remove	TLIB가 라이브러리로부터 지정된 모듈을 제거한다. 모듈이 존재하지 않으면 TLIB 는 메시지를 표시한다.
*	Extract	TLIB가 라이브러리로부터, 대응되는 모듈을 파일에 복

```
|      |              |사함으로써 지정된 파일을 생성한다. 그 모듈이 존재하|
|      |              |지 않을 때는 메시지를 표시하고 파일을 생성하지 않는|
|      |              |다. 지정된 파일이 이미 존재할때는 겹쳐 쓰기를 한다.|

---------------------------- 계  속 -------------------------------

+------------------------------------------------------------------+
| -+ | Replace  |지정된 모듈을 대응하는 파일로 대체한다. 이것은 단지|
| +- |              |remove 뒤에 add 연산이 오는 것에 대한 속기이다.     |
+------------------------------------------------------------------+
| -* | Extract &|지정된 모듈을 대응하는 파일명에 복사하고 라이브러리|
| *- | Remove    |로부터 그것을 제거한다. 이것은 단지 extract 다음에 |
|      |              |remove 연산이 오는 것에 대한 속기이다.              |
+------------------------------------------------------------------+
```

ㅎl

4.2.2 라이브러리의 생성

ㅎ

라이브러리를 생성하기 위해서는 사용자가 단지 모듈들을 아직 존재하지 않는 라이브러리에 추가하기만 하면 된다.

4.3 확장된 사전의 생성(/E 옵션)

표준 Cx.LIB와 같은 대형 라이브러리 파일과의 링크를 고속화하기 위해서 사용자는 TLIB가 확장사전을 생성하고, 이를 라이브러리 파일에 추가하도록 지시할 수 있다.

ㅎl

4.3.1 확장된 사전

ㅎ

(1) 매우 압축된 형대로 표준 라이브러리 사전에 포함되지 않는 정보를 포함한다.
(2) TLINK 가 라이브러리 파일을 더 빠르게 처리할 수 있게 해준다.
(3) 특히 플로피 디스크나 느린 하드 디스크상에 위치해 있을 때 빠르게 처리할 수 있도록 한다.
(4) 배포된 터보 C 모든 라이브러리들은 확장된 사전은 갖는다.

ㅎl
4.3.2 확장된 사전의 생성
ㅎ
[방법 1] 수정된 라이브러리의 확장된 사전을 생성할때
- remove 또는 replace 모듈을 라이브러리에 추가할때 /E 옵션을 사용한다.

[방버 2] 수정하지 않은 기존 라이브러리의 확장된 사전을 생성할때
- /E 옵션을 사용하여 라이브러리로부터 존재하지 않는 모듈을 제거하도록 요구한다.
이때 TLIB 는 지정된 모듈이 라이브러리에서 찾을 수 없다는 경고를 표시하지만 지정된 라이브러리에 대해 확장된 사전을 생성한다.

[예제] tlib /E mylib -bogus

4.4 진보된 연산(/C 옵션)

이 옵션은 TLIB가 그 라이브러에 이미 존재해 있는 symbol 을 갖는 모듈을 받아들이도록 한다.

[예] 어떤 모듈이 라이브러리에 추가될 때 이미 그 라이브러리에 있는데 문자체만 다른 symbol인 경우(소문자와 대문자)에는 /C 옵션을 사용해야 한다.

[주의] 다른 링커와 함께 라이브러리를 사용하는 경우에는 보호를 위해서 사용하지 않는 것이 좋다.

4.5 TLIB로 할 수 있는 여러가지 예제

[방법] tlib mylib +x +y +z
[설명] 모듈 X.OBJ(+x), Y.OBJ(+y), Z.OBJ(+z)를 갖는 MYLIB.LIB 라는 라이브러리를 생성한다.

[방법] tlib mylib +x +y +z, mylib.lst
[설명] 라이브러리를 생성하고 MYLIB.LST 리스트를 갖는다.

[방법] tlib cs, cs.lst
[설명] 현재 라이브러리 CS.LIB 의 CS.LST 리스트를 갖는다.

[방법] tlib mylib -+x +a -z
[설명] 모듈 X.OBJ(-+x)를 새로운 복사본으로 대체하고, A.OBJ(+a)를 추가하며, Z.OBJ(-z)를 MYLIB.LIB로부터 삭제한다.

---------------------------- 계 속 -------------------------------

[방법] tlib mylib *y, mylib.lst
[설명] 모듈 Y.OBJ(*y)를 MYLIB.LIB 로부터 추출하고, MYLIB.LST의 리스트 갖는다.

[방법] 응답파일을 사용해서 모듈 A.OBJ, B.OBJ ...,G.OBJ를 갖는 ALPHA 라는 새로운 라이브러리를 생성한다.
먼저 아래와 같이 ALPHA.RSP(텍스트 파일)를 생성한다.

```
+a.obj +b.obj +c.obj &
+d.obj +e.obj +f.obj &
+g.obj
```

그리고 TLIB 을 이요하여 아래와 같이 한다.

```
tlib alpha @alpha.rsp, alpha.lst
```

@ 은 응답파일을 의미하고 ALPHA.LST 의 리스트를 갖는다.

5. 파일 탐색 유틸리티(GREP)

GREP 는 한번에 여러개의 파일에 있는 텍스트를 탐색할 수 있는 막강한 탐색 유틸리티이다.

[형식] grep [options] searchstring [filespec...]

[예제] 사용자가 setupmodem 함수를 어느 소스 파일에서 호출하는지를 알
알아 보자.

grep setupmodem *c

사용자의 디렉토리에 있는 모든 .C 파일의 탐색하여 문자열 setup
-modem 을 찾는다.

5.1 GREP 옵션

명령행에서 옵션은 앞에 대쉬(-)가 오며 하나 이상의 단일 문장가 된다.

(1) 각각의 개별문자는 사용자가 On 또는 Off할 수 있는 스위치이다.

옵션 On : 문자뒤에 플러스 기호(+)를 넣는다.
옵션 Off : 문자뒤에 대쉬(-) 를 넣는다.
디폴트는 On(+의 의미)이다.
[예] -r 은 -r+ 와 같다.

(2) 각각 여러개의 옵션을 줄 수가 있다(-i -d -l).
(3) 옵션을 묶을 수도 있다(-ild 혹은 -il -d).

ㄶl
옵션 문자의 리스트와 그 내용
ㄶ

옵션	기 능	내 용
-c	Count only	지정된 문자열과 일치하는 행수만을 표시하며, 그 행의 내용은 출력되지 않는다.
-d	Directories	파일을 참색하려는 디렉토리를 표시한다. 서브 디렉토리도 탐색의 대상으로 한다.
-i	Ignore case	대문자/소문자의 구별을 무시한다.
-l	List match files	일치하는 행을 포함하는 파일명을 출력한다.
-n	Numbers	일치하는 행에 행번호를 붙여 출력한다.

---------------------------- 계 속 -------------------------------

옵션	기 능	내 용
-o	UNIX output formal	UNIX 형식의 출력을 수행한다.
-r	Regular expression search	문자열 탐색에 의해 정의된 텍스트는 문자열 대신에 정상적인 표현으로 처리된다.
-u	Update options	지정된 모든 옵션을 기동하는 GREP.COM파일에 이들 디폴트 옵션을 써넣어 자기 자신을 수정 한다(새로운 디폴트를 갖는 GREP를 작성)
-v	Non-match	지정된 문자열과 일치하지 않는 행을 출력한다
-w	Word search [regular express-ion: 정상적인 표현]	워드단위로 정상적인 표현과 일치하는가를 조사한다.(워드를 구성하는 문자를 정상적인 표현으로 지정한다.)

---------------------------- 계 속 -------------------------------

```
|----|-------------------|----------------------------------------|
| -z |Vervose            |보다 상세하게 일치하는 정보를 출력한다. 즉 |
|    |                   |탐색된 모든 파일의 파일명을 출력하는데 각  |
|    |                   |일치하는 행번호와 행 수가 0 이라도 일치되는|
|    |                   |행의 수를 표시한다.                       |
+-----------------------------------------------------------------+
```

ㅋ]5.2 탐자문자열

탐색문자열(searchstring)의 값은 GREP가 탐색하는 형태를 정의한다. 이 탐색문자열은 정상적인 표현이나 또는 문자열 중 어느 것이나 될 수 있다.

ㄷl
정상적인 표현식에서의 연산자
ㄷ
사용자가 -r 옵션을 사용할 때 탐색문자열은 정상적인 표현식(문자 표현식 이 아님)으로 처리되며 다음의 문자들은 특별한 의미를 갖는다.

---------------------------- 계 속 -------------------------------

(1) ^ : 표현식의 선두에 있는 ^은 행의 시작을 표시한다.

(2) S : 표현식의 끝에 S은 행의 끝을 나타낸다. (S 는 달러표시 임)

(3) . : 임의의 어떠한 문자를 나타낸다.

(4) * : 표현식뒤에 * 와일드 카드가 오면 그 표현식과 일치하는 것이 o
또는 o 이 여러개 있다는 것을 나타낸다.

(5) + : 그 표현식의 발생에 있어서 바로 앞의 문자를 1개 이상 반복하는
것을 나타낸다.

(6)[] : 괄호로 둘러싸인 문자열이 다른 문자가 아닌 바로 그 문자열에
있는 어느 문자와도 일치한다.

(7) \ : GREP에게 그 뒤에 오는 문자를 탐색하도록 지시한다. 예를들어
\.는 어떤 문자 대신에 피리어드(.)와 일치한다.

5.3 파일 명세(filspec)

이것은 GREP에게 탐색할 파일이나 파일의 그룹을 지시한다. filespec 은 명시적 파일명이 되거나 DOS의 와일드 카드인 ? 과 * 를 혼합한 일반적 (generic) 파일명이 될 수 있다.

여기에다 사용자는 filespec의 일부로서 패스(드라이브와 이렉토리 정보) 를 입력할 수 있다. 패스 없이 filespec을 주면 GREP는 현대의 디렉토리만 을 탐색한다. 어떠한 파일 명세도 지정하지 않는 경우 GREP에 대한 입력은 stdin을 리디렉트 하거나 파이프에 의해서 지정되어야 한다.

6. 그래픽 드라이버와 폰트 변환 유틸리티(BGIOBJ)

BGIOBJ는 그래픽 드라이버와 폰트 파일을 `.OBJ 형식`으로 바꿔서 사용자의 `실행파일(.EXE)에 링크`할 수 있도록 해준다. 이 기능은 실행시에 디스크에서 드라이버와 폰트를 로드하는 개념에 대해 추가로 제공되는 것이다.
사용자 프로그램에 드라이버와 폰트를 링크하는 것은 그 프로그램이 일단 실행되기 시작하면, 필요한 드라이버와 폰트를 자신이 가지고 있는 관계로 디스크를 필요로 하지 않는다는 점에서 매우 유요하다. 그러나 실행파일에 드라이버와 폰트를 링크하면 `파일의 크기가 증가`한다.

ㆆl
드라이버나 폰트 파일을 링크 가능한 .OBJ 파일로 변환
ㆆ
[형식] BGIOBJ <source file>
[설명] <source file>은 오브젝트 파일로 변환될 드라이버나 폰트파일을 말한다. 생성되는 오브젝트 파일은 원래의 파일과 같은 이름을 가지며 단지 확장자가 .OBJ가 된다.
[예제] EGAVGA.BGI를 변환하면 EGAVGA.OBJ가 생성된다.
SANS.CHR을 변환하면 SANS.OBJ가 생성된다.
6.1 GRAPHICS.LIB에 새로운 .OBJ 파일 추가

GRAPHICS.LIB에 새로운 드라이버와 폰트 오브젝트를 `추가`하여야 한다.
이렇게 함으로써 링커가 그 파일들을 실행 파일속에 배치할 수 있기 때문
이다.

만일 GRAPHICS.LIB에 이 파일들을 추가하지 않는다면, TC의 프로젝
트(.PRJ)파일이나 TCC의 명령행, TLINK의 명령행에 이 파일을 일일이 나열
해야만 한다.

[방법] tlib graphics + <object file name> [+<object file name> ...]

[설명] <object file name>은 BGIOBJ,EXE(CGA, EGAVGA, GOTH 등과 같음)에
의해서 생성된 .OBJ 파일을 말한다. OBJ 확장자는 이미 암시되어
있으므로 굳이 붙일 필요는 없다. 사용자는 시간을 절약하기 위해
단 한줄에 여러개의 파일을 처리할 수 있다.

6.2 드라이버와 폰트의 등록

원하는 오브젝트를 GRAPHICS.LIB에 추가한 뒤에는 필히 링크해야될 드라이

버나 폰트를 등록하여야 한다. 이 일은 initgraph를 호출하기 전에 regist

-erbgidriver나 registerbgifont를 호출함으로써 행해진다. 이러한 함수를

호출함으로써 그래픽 시스템에서 이들 파일의 존재를 알리고 링커에 의해

실행파일이 생성될 때 랭크하게 해준다.

등록 루틴은 GRAPHICS.H에서 정의된 기호 이름을 인수로 받고, 드라이버나

폰트가 제대로 등록되면 음수가 아닌 값을 리턴한다.

ㅎl
Turbo C에 포함된 드라이버와 폰트의 이름
(registerbgidriver나 registerbgifont와 함께 사용)

ㄷ

드라이버 파일 (*,BGI)	registerbgidriver 기호이름	폰트파일 (*.GHR)	registerbgifont 기호이름
CGA	CGA_driver	TRIP	triplex_font
EGAVGA	EGAVGA_driver	LITT	small_font
HERC	HERC_driver	SANS	sansserif_font
ATT	ATT_driver	GOTH	
PC3270	PC3270_driver		
IBM8514	IBM8514_driver		

[예제] CGA그래픽 드라이버, 고딕 폰트, 그리고 3중(triplex)폰트를 위한 파일을 오브젝트 모듈로 바꿔서 프로그램에 링크하려 한다고 가정하자.

(1) BGIOBJ.EXE를 사용하여 이진 파일을 오브젝트 파일로 변환한다.

```
bgiobj cga
bgiobj trip
bgiobj goth
```

생성 파일: CGA.OBJ, TRIP.OBJ, GOTH.OBJ

(2) GRAPHICS.LIB에 위(1)의 파일을 추가한다.

```
tlib graphics +cga +trip +goth
```

---------------------------- 계 속 --------------------------------

(3) 만일 GRAPHICS.LIB에 오브젝트 파일을 추가하지 않으면 TC의 프로젝트 리스트나 TCC의 명령행, TLINK의 명령행에 위의 파일들을 나열해야 한다.

예를 들어 TCC 경우에는 아래와 같이 나열해야 한다.

```
tcc niftgraf graphics.lib cga.obj goth,obj
```

(4) 다음과 같이 사용자의 그래픽 프로그램에 이들 파일들을 등록한다.

```
/* Header file declares CGA_driver, triplex_font, and gothic_font */
#include <graphics.h>
/* Register and check fof errors (one never knows ...) */
if (registerbgidriver(CGA_driver) <0) exit (1);
if (registerbgifont(triplex_font) <0) exit (1);
if (registerbgifont(gothic-font)  <0) exit (1);
/* ... */
initgraph(...); /* initgraph should be called after registering */
/* ... */
```

6.3 /F 옵션

여기에서는 링커의 에러 메시지 `Segment exceeds 64K`가 나올 경우 어떻게 하는지를 설명한다.

보통 BGIOBJ.EXE는 동일한 세그먼트 이름(_TEXT)을 사용한다. 따라서 메모리 모델이 small 이나 compact 인 경우 혹은 여러개의 드라이버나 폰트를 링크할 경우에 문제가 발생할 수 있다. 이 옵션은 `세그먼트 이름을 지정` 할 수 있어 문제를 해결할 수 있다.

이 옵션을 사용하면 세그머트의 이름이 <filename>_TEXT로 생성되어 다른 드라이버나 폰트화의 세그먼트 중복이 발생하지 않는다.

---------------------------- 계 속 -------------------------------

예를 들어 아래와 같이 하면 각각의 오브젝트는 EGAVGA_TEXT와 SANS_TEXT 라는 세그먼트 이름을 가진다.

```
bgiobj  /F  egamga
bgiobj  /F  sans
```

이 옵션을 사용하면 생성되는 오브젝트 파일은 이름 끝에 F가 붙게 되며

registerfarbgidriver 와 registerfarbgifont 에 사용될 이름에는 -far가

붙는다(예를 들면, EGAVGA-driver가 EGAVGA-driver-far가 된다). /F 옵션

을 사용하여 생성된 파일을 사용하는 경우 정상적인 registerbgidriver나

registerbgifont대신에 far 등록 루틴을 사용하여야 한다. 예를 들면

```
if(registerfarbgidriver(EGAVGA-dirver-far) < O) exit(1);
if(registerfarbgifont(sansserif-font-far) < 0) exit (1);
```

6.4 BGIOBJ의 고급 기능

ㅎI

BGIOBJ.EXE 유틸리티를 쓰는 완전한 형식

ㅎ

```
[형식] BGIOBJ [/F] <source> <destination> <public naem> <seg-name>
                 <seg-class>
+-----------------------------------------------------------------+
|  구성요소   |              설              명                   |
|-------------|---------------------------------------------------|
|/F 또는 -F   |디폴트(-TEXT)가 아닌 세그먼트명을 사용하게 하고 퍼 |
|             |블릭명과 결과 파일명을 변경하도록 한다(자세한 내용 |
|             |은 앞의 /F 옵션에 대한 내용을 참조).               |
|-------------|---------------------------------------------------|
|<source>     |변환될 드라이버 또는 파일이다. 만일 변환할 파일이  |
|             |Turbo C 와 함께 주어진 드라이버나 폰트가 아니라면  |
|             |그 파일의 완전한 이름이 주어져야 한다(확장자 포함).|

---------------------------- 계  속 -------------------------------

|-------------|---------------------------------------------------|
|<destination>|생성될 결과 파일명이다. 결과 파일명은 디폴트로 <sor |
|             |-tce>.OBJ가 된다. 또는 /F 옵션을 주면 <source>F.OBJ |
|             |가 된다.                                           |
|-------------|---------------------------------------------------|
```

```
|<public name>|프로그램에서 오브잭트 모듈을 링크하도록 registerbgi |
|             |-driver나 registerbgifont(또는 각각의 far버젼)를 호 |
|             |출하는데 사용될 퍼블릭 명이다. 이 이름은 링커에 의해|
|             |외부명으로 사용되므로 밑줄(-)로 시작해야 한다. 만일,|
|             |프로그램이 파스칼의 호출방식을 사용한다면 대문자만을|
|             |사용해야 하며, 밑줄을 추가해서는 안된다.            |
|-------------|----------------------------------------------------|
|<seg-name>   |임의의 세그먼트 명으로 디폴트는 -TEXT(또는 /F가 지정|
|             |되면 <filename>-TEXT가 된다.                        |
|-------------|----------------------------------------------------|
|<seg-class>  |임의의 세그먼트 클래스로 디폴트는 CODE가 된다.      |
+------------------------------------------------------------------+
```

<source>를 제외한 다른 매개변수들은 생략해도 된다.

ㅎI

메모리에 폰트 파일을 로드하는 예제가 아래에 있다.

ㅎ

```
/* example of loading a font file into memory */
#include <graphics.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <process.h>
#include <alloc.h>

main()
(
   void     *gothic-fontp; /* points to the font buffer in memory */
   int       handle;       /* file handle used for I/O        */
   unsigned fsize;         /* size lf file (and buffer)      */
```

---------------------------- 계 속 -------------------------------

```
   int errorcode;
   int graphdriver;
   int graphmode;

/* open font file */
   handle = open ("GOTH.CHR", O-RDONLY O-BINARY);
```

```
   if (handle == -1
   (
      printf("unable to open font file 'GOTH.CHR'\n");
      exit(1);
   )
/* find out size of the file */
   fsize = filelength(handle);
/* allocate buffer */
   gothic fontp = malloc(fsize);
   if (fothic-fontp == NULL)
   )
```

-------------------------------- 계 속 -----------------------------------

```
      printf("unable to allocate memory for font file 'GOTH.CHR'\n");
      exit(1);
   )
/* read font into memory */
   if (read(handle, gothic-fontp, fsize) != fsize)
   (
      printf("unable to read font file 'GOTH.CHR'\n");
      exit(1);
   )
/* closs font file */
   close (handle);
/* register font */
  if (registerfarbgifont (gothic-fontp) !=GOTHIC-FONT)
  (
      printf("unable to register font file 'GOTH.CHR'\n");
      errorcode = graphersult();
```

-------------------------------- 계 속 -----------------------------------

```
      if (errorcode !=grOk)
      (
         printf("graphics error: %s\n".grapherformsg(errorcode));
         exit(1);
      )
      settextjustify(CENTER-TEXT,CENTER-TEXT);
      settextstyle(GOTHIC-FONT, HORIZ-DIR, 4);
```

```
        outtextxy( getmaxx() / 2, getmaxy() / 2,
                   "Borland Graphics Interface (BGI)");
/* hit a key to terminate */
   getch();
/* shut down graphics system */
   closegraph();
   return(0);
```

7. 오브젝트 모듈 교차-참조 유틸리티(OBJXREF)

OBJXREF는 오브젝트 파일과 라이브러리 파일의 리스트를 검사해서 그 내용에 대해 보고를 작성하는 유틸리티이다.

[보고]
퍼블릭명과 그 참조에 대한 것과 오브젝트 모듈이 정의한 세그먼트 크기에 관한 것으로 나눌 수 있다.

[퍼블릭명의 범주]
전역변수와 함수로 나눌 수 있다.

[오브젝트 모듈]
TC.TCC 또는 TASM에 의해 생성되는 오브젝트(.OBJ) 파일이다.
라이브러리(.LIB)파일은 여러개의 이름을 가진다. TCC에 의한 파일도 -o TCC 명령행 옵션을 써서 지정하지 않으면 이름을 가진다.

7.1 OBJXRET 명령행

[형식] OBFXREF < options > fioename < filename ... >

ㅋ7.1.1 OBJXREF 명령행 옵션

이 옵션은 제어옵션과 보고옵션으로 나눌 수 있다.
ㅎl
제어옵션(Control Options)

ㆆ
이 옵션은 OBJXREF의 디폴트 작동을 수정한다(OBJXREF의 디폴트 작동은 아래의 어느 옵션도 지정되지 않은 것이다).

(1) /I : 퍼블릭명의 대소문자를 구분하지 않는다. 이 옵션은 TLINK를 대 소문자를 구별하게 하는 /C 옵션없이 쓸 경우에 사용한다.
(2) /F : 라이브러리를 모두 포함한다. 라이브러리(.LIB)의 모든 .OBJ 모 듈에 대해 그 내부의 퍼블릭명ㅇ에 대한 참조여부를 가리지 않고 작업한다. 이 기능은 .LIB파일의 전체 내용을 알고자 하는 경우 에 매우 유용하다.

----------------------------- 계 속 --------------------------------

(3) /V : 읽은 파일의 이름을 나열하고, 퍼블릭명, 모듈, 세그먼트, 클래 스를 전부 표시한다.
(4) /Z : 길이가 0 인 빈 세그먼트의 정의를 포함한다. 때때로 공백을 전 혀 할당하지 않는 세그먼트가 오브젝트 모듈에 있을 수 있다. 이러한 빈 세그먼트를 나열하게 하는 것은 세그먼트 크기 보고를 사용하는데는 거추장스럽지만 세그먼트 정의를 제거하려 하는 경 우에는 매우 유용하다.

ㆆI
보고옵션(Report Options)
ㆆ
이 옵션은 어떤 내용이 보고되며, 그 상세함이 어느 정도인지를 정한다.

(1) /RC(Report by class Type)
 - 세그먼트의 클래스 순서에 따른 세그먼트 크기
(2) /RM(Report by Module)
 - 모듈 순서에 따른 퍼블릭명

----------------------------- 계 속 --------------------------------

(3) /RP(Report by Public Names)
 - 모듈명을 정의하는 순서대로의 퍼블릭명
(4) /RR(Report by Reference)
 - 퍼블릭명과 참조를 이름 순서에 따라
(5) /RS(Report by Module Sizes)
 - 모듈 크기를 세그먼트 순서에 따라
(6) /RU(Report of Unreferenced Symbol Name)
 - 참조되지 않는 퍼블릭명을 정의된 모듈의 순서에 따라
(7) /RV(Verbose Reporting)

- 모든 종류의 보고를 산출

(8) /RX(Report by External Reference)

- 외부 참조를 참조하는 모듈의 순서에 따라

.C 파일에 정의된 퍼블릭명은 컴파일시에 - U- 옵션을 써서 컴파일하지 않으면 앞에 밑줄(_)이 붙어서 보고된다(main이 _main로 됨).

7.1.2 응답파일

DOS의 명령행에는 최대 128자 밖에는 쓸 수 없다. 만일 옵션이나 파일명을 모두 표시하는데 이보다 더 길다면 응답파일을 써야 한다.
응답파일은 텍스트 에디터로 만든 파일이다. Turbo C 프로그램을 만들기 위하여 필요한 파일의 표를 이미 만든 경우가 있기 때문에 OBJXREF는 두세 개의 다른 형식을 응답파일로 받아들인다.

ㅎl
(1) 자유형식 응답파일
ㅎ
자신의 .EXE 파일을 만드는데 필요한 모든 .OBJ와 .LIB파일을 단순히 나열하기만 하면 된다.
이 응답파일을 사용하려면 OBJXREF의 명령행에 그 파일 앞에 @을 붙여 표시하고 다른 명령의 입력과 구분하기 위해 공백이나 탭으로 분리하면 된다.

[형식] @filename   @filename
[주의] 응답파일내에 있는 파일을 확장자가 없이 표시하면 .OBJ 파일로 간주 된다.

ㅎl
(2) 프로젝트 파일
ㅎ
프로젝트 파일명 앞에 /P를 붙여서 응답파일로 쓸 수 있다.

[형식] /Pfilename
[설명] 파일 명에 확장자를 붙이지 않으면 .PRJ로 간주한다.
프로젝트 파일내에 .C가 붙은 파일이나 화장자가 없는 파일은 그 이름을 가진 .OBJ 파일로 간주된다. 괄호안에 표시된 파일간의 의존관계를 제거할 필요는 없다. 그 내용들을 OBXREF는 무시한다.

[주의] 프로젝트 파일은 그 자체로서 .EXE 파일을 만드는데 필요한 모든 파일명을 가지고 있지 않다. 따라서 기동파일(COx.OBJ)과 Turbo C의 라이브러리 파일(예:mathx.lib, emu.lib, Cx.lib 등)을 명시햐야만 한다. 또한 0BJXREF가 필요한 .OBJ 파일을 찾을 디렉토리를 알려주기 위하여 /O 명령을 쓸 필요가 있을 때도 있다.

ᄐI
(3) 링커 응답파일
ᄐ
링커 응답파일을 쓰려면 명령행에서 /L을 파일명 앞에 붙여 사용한다.

[형색] Lfilename

ᄐI
(4) /O 명령
ᄐ
OBJXREF로 하여금 현재의 디렉토리가 아닌 다른 디렉토리에서 .OBJ 파일을 찾도록 하려면 명령행에서 그 디렉코리명 앞에 /O를 붙여서 표시한다.

[형식] /Omyobjdif

ᄐI
(5) /N 명령
ᄐ
/N 명령을 어떤 이름의 앞에 붙여 명령행에 표시함으로써 OBJXREF가 보고할 모듈, 세그먼트, 클래스 또는 퍼블릭명에 제한을 가할 수 있다.
다음 형식은 OBJXREF 가 CO 란 이름을 가지는 모듈에 한해 보고할 것을 지시한다.

[형식] OBJXREF <filelist> /RM /NCO

II. TCC 명령행 옵션

ডI
옵션의 구분(3가지)
ড
(1) 컴파일러 옵션
(2) 링커 옵션
(3) 환경 옵션

ডI
컴파일러 옵션의 범주
ড
(1) 메모리 모델
(2) #defines(마크로 정의)
(3) 코드 생성 옵션
(4) 최적화 옵션
(5) 소스 코드 옵션
(6) 에러 옵션
(7) 세그먼트명 제어 옵션

ডI
TCC 옵션
ড
DOS 프롬프트상에서 TCC 명령을 다음과 같은 리스트를 볼수가 있다.

```
Turbo C  Version 2.0  Copyright (c) 1987, 1988 Borland International
Syntax is: TCC [options] file[s] * = default; -x- = turn switch x off
+-------------------------------------------------------------------+
|-1      80186/286 Instructions  |-A      Disable non-ANSI extensions|
|-B      Compile via assembly     |-C      Allow nested comments      |
|-Dxxx   Define macro             |-Exxx   Alternate assembler name   |
|-G      Generate for speed       |-Ixxx   Include files directory    |
|-K      Default char is unsigned|-Lxxx   Libraries directory         |
```

```
|-M       Generate link map        |-N       Check stack overflow        |
|-O       Optimize jumps           |-S       Produce assembly output     |
|-Uxxx    Undefine macro           |-Z       Optimize register usage     |
|-a       Generate word alignment |-c       Compile only                |
|-d       Merge duplicate strings |-exxx    Executable file name        |

------------------------------ 계  속 ------------------------------

|-f     * Floating point emulator |-f87     8087 floating point          |
|-gN      Stop after N warnings    |-iN      Maximum identifier length N|
|-jN      Stop after N errors      |-k       Standard stack frame        |
|-lx      Pass option x to linker |-mc      Compact Model               |
|-mh      Huge Model               |-ml      Large Model                 |
|-mm      Medium Model             |-ms    * Small Model                 |
|-mt      Tiny Model               |-nxxx    Output file directory       |
|-oxxx    Object file name         |-p       Pascal calls                |
|-r     * Register variables       |-u     * Underscores on externs      |
|-v       Source level debugging  |-w       Enable all warnings         |
|-wxxx    Enable warning xxx       |-w-xxx  Disable warning xxx          |
|-y       Produce line number info|-zxxx    Set segment names           |
+------------------------------------------------------------------+
```

1. 옵션의 ON과 OFF

(1) 옵션의 ON : -A   (예)
(1) 옵션의 OFF : -A- (예)

ㅎ]2. 문법

[형식] tcc [option option ...] filename filename ...

ㅎI
2.1 컴파일 규칙
ㅎ
(1) filename   : filename.c를 컴파일 한다.
(2) filename.c   : filename.c를 컴파일 한다.

(3) filename.xyz : filename.xyz를 컴파일 한다.
(4) filename.obj : 링크시에 오브젝트로서 포함된다.
(5) filename.lib : 링크시에 라이브러리로서 포함된다.
(6) filename.asm : OBJ로 어셈블하기 위해서 TASM을 호출한다.

ㆆl
[예제] tcc -a -f -C -o -z -emyexe oldfile1.c oldfile2.c nextfile.c
ㆆ
[설명]
(1) OLDFILE1.C, OLDFILE2.C, NEXTFILE.C를 .OBJ로 컴파일 한다.
(2) 단어정렬(-a), 부동소숫점 에뮬레이션(-f), 네스트된 주석문(-C),
최적화 점프(-O), 레지스터 최적화(-Z)옵션 등이 선택되어 있는
실행프로그램 파일인 MYEXE.EXE를 작성한다.
(3) 사용자가 명령행상에서 .ASM 파일에 TASM을 제공하거나 a.C 파일이
인라인 어셈블리를 포함한다면 TCC는 TASM을 호출하게 된다.
(4) TCC 가 TASM에 제공하는 스위치

/mx /D_mdl_

mdl : tiny, small, medium, compact, large, huge 중의 하나가 된다.
/mx : TASM에게 case 민감도(case-sensitiy)를 ON으로 어셈블하도록
알린다.

3. 컴파일러 옵션

ㆆl
메모리 모델
ㆆ
사용자가 어떤 메모리 모델하에서 프로그램을 컴파일할 것인가를 지정하
게 한다(모델은 `tiny, small, medium, compact, large, huge`).

ㆆl
#defines(마크로 정의)
ㆆ
사용자가 명령행상에서 마크로(manifest 나 symbolic 상수로 알려진)를
정의하도록 한다. 디폴트로 정의될 때는 단일 공백문자이다.

ㆆl

코드생성 옵션
ट
이것은 부동소수점 모드, 규칙 호출, 문자형 또는 CPU 명령어 등과 같이 실행시에 사용되는 생성된 코드의 특성을 제어한다.

टI
최적화 옵션
ट
사용자가 오브젝트 코드가 최적화 될 수 있는 방법을 지정할 수있게 한다. 여기서는 크기와 속도, 레지스터 변수의 사용 유무, 과도한 부하 작동의 유무등이 포함된다.

टI
소스코드 옵션
ट
이것은 컴파일러가 소스코드의 어떤 특성을 인식하거나 또는 무시하게 하는 것으로 특정 수행(ASNI가 아님) 키워드, 네스트된 주석문, 그리고 식별자 길이 등이 포함된다.

टI
에러 옵션
ट
사용자로 하여금 컴파일가 표시하는 경고 메시지와 컴파일 중단되기 전에 발생할 수 있는 경고 또는 에러의 최대수를 조정할 수 있게 한다.

टI
세크먼트명 제어 옵션
ट
사용자가 세그먼트를 다시 명명할 수 있게 하고, 그들의 그룹과 문류를 재지정하도록 한다.

टI
컴파일 제어 옵션
ट
이것은 사용자가 컴파일러에게 다음과 같이 하도록 지시한다.
(1) 어셈불리 코드로 컴파일한다(오브젝트 모듈이 아님).
(2) 인라인 어셈블리를 포함하는 소스파일을 컴파일한다.
(3) 링크시키지 않고 컴파일한다.

3.1 메모리 모델

(1) -mc : compact 메모리 모델을 사용해서 컴파일한다.
(2) -mh : huge 메모리 모델을 사용해서 컴파일한다.
(3) -ml : large 메모리 모델을 사용해서 컴파일한다.
(4) -mm : medium 메모리 모델을 사용해서 컴파일한다.
(5) -ms : small 메모리 모델을 사용해서 컴파일한다.
(6) -mt : tiny 메모리 모델을 사용해서 컴파일한다. small 메모리 모델과
같은 코드를 생성하지만 tiny 모델 프로그램을 생성하기 위해서
링크 수행시에 C0T.OBJ를 사용한다.

ъ]3.2 #defines

(1) -Dxxx
지정된 식별자 xxx를 단일 공백문자로 구성되는 문자열로 정의한다.
(2) -Dxxx=string
지정된 식별자 xxx를 등호기호 뒤에 문자열인 string으로 정의한다.
string은 어떠한 공백이나 탭을 포함하지 않는다.
(3) -Uxxx
이전에 정의된 식별자의 이름인 xxx를 취소한다.
ъl
복수의 define 입력 방법
ъ
(1) 사용자는 단일 -D 옵션뒤에 복수개의 입력 옵션을 포한할 수 이때 세미
콜론으로 분리한다(이것은 ganging 옵션으로 알려져 있다).

tcc -Dxx;yyy=1;zzz=NO myfile.c

(2) 사용자는 명령행상에서 하나 이상의 옵션을 위치시킬 수가 있다.

tcc -Dxxx -Dyyy=1 -Dzzz=NO myfile.c

(3) 사용자는 복수의 -D를 나열하고 집단으로 혼용할 수 있다.

tcc -Dxxx -Dyyy=1;zzz=NO myfile.c

3.3 코드 생성 옵션

(1) -l : 터보 C가 확장된 80186명령을 생성하도록 한다.

(2) -a : 데이타의 할당을 워드 경계로 한다.

(3) -d : 하나의 문자열과 동일한 있는 경우에 문자열을 합병한다.
(디폴트는 off).

(4) -f87 : 인라인 8087(80287) 명령을 생성한다. 부동 소숫점의 연산을
하기 위해서 이다(부동소숫점 칩이 내장되어 있지 않은 기계상
에서는 실행되지 않는다).

(5) -f : 실행 시스템이 8087을 내장하고 있지 않은 경우 실행시에 8087
호출을 에뮬레이트 한다.

(6) -f- : 부동소숫점 연산을 행하지 않으며 따라서 부동소숫점 라이브러
리를 링크하지 않는다.

------------------------------ 계 속 ------------------------------

(7) -K : 컴파일러가 모든 char 선언을 unsigned char 형으로 취급한다.
(다른 컴파일러와의 호환성을 갖도록 하기 위한 것이다)

(8) -k : 표준 스택 프레임(standard stack frame)을 생성한다.
(디폴트는 on 이다)

(9) -N : 각 함수의 입력에 스택 오버플로우 로직을 생성한다.
이것은 스택 오버플로우를 탐지할때 사용한다.

(10) -p : 함수의 호출과 종료코드를 파스칼 형식으로 한다.

(11) -u : 선언된 함수명 앞에서 밑선(_)을 붙여서 오브젝트 파일에 출력
한다(초보자는 사용하지 않는 것이 좋다).

(12) -y : 기호 처리 디버거용의 오브젝트 파일에 행번호를 포함한다.

(13) -v : 터보 C 통합 디버거용의 디버그 정보를 오브젝트 파일에 출력

한다.

3.4 최적화 옵션

(1) -G : 크기보다 실행속도에 중점을 두어 최적화 한다.

(2) -O : 불필요한 점프를 삭제하거나 루프 혹은 스위치문을 재편성으로
써 최적화를 행한다.

(3) -r- : 레지스터 변수의 사용을 금지한다.
-r- 옵션 혹은 O/C/Optimization/Use register variables Off를
사용하면, 컴파일러는 레지스터 변수를 사용하지 않으며 또한
어떠한 호출로부터도 레지스터 변수 SI, DI 를 보유하거나 고려
하지 않게 된다.

반면에 SI, DI 를 보유하고 있지 않은 기존의 어셈블리어 코드
와 인터페이스하는 경우 -r- 옵션은 사용자가 터보 C 로 부터
호출할 수 있도록 허용한다(초보자는 사용하지 않는 것이 좋다).

(4) -r : 디폴트로 레지스터 변수의 사용을 허용한다.

------------------------------ 계 속 ------------------------------

(5) -Z : 레지스터의 내용을 기억하고, 가능한한 자주 그들을 재사용함으
로써 불필요한 작동부하를 줄인다.
(주의 : 레지스터가 포인터에 의해 간접적으로 무효화되는 경우
컴파일러는 이를 탐지할 수 없기 때문이다.)

ㅠ]3.5 소스코드 옵션

(1) -A : ANSI 호환성의 코드를 컴파일 한다. 터보 C 확장 키워드는 모두
무시되고 정상적인 식별명으로서 사용될 수 있다.

[키워드] near, far, huge, cdecl, asm, pascal, interrupt,
_es, _ds, _cs, _ss 과 의사변수(pseudo-variables)
의 _AX, _BX, _SI 등도 포함된다.

(2) -C : 주석문(comment)의 네스트를 허용한다. 보통 주석문의 네스트는
허용되지 않을 수도 있다.

(3) -i# : 컴파일러가 식별자의 첫번째 # 문자만을 인식하게 한다.

3.6 에러옵션

(1) -g#
　　#에 해당하는 갯수 만큼의 경고와 에러가 발생하면 컴파일을 중지한다.

(2) j#
　　#에 해당하는 갯수 만큼의 에러가 발생하면 컴파일을 중지한다.

(3) -wxxx
　　xxx 에 의해 지정된 경고 메시지를 출력한다. 그러나 `-w-xxx`는 wxxx
　　에 의해 지정된 경고 메시지를 방지한다.

ᄒ]-wxxx 에 의해 가능한 값들

ᄒl
ANSI 규정 위반
ᄒ
(1) -wbig* : 8진수 혹은 16진 상수가 너무 크다.
------------------------------ 계　속 ------------------------------

(2) -wdup* : 똑같은 이름 'XXXXXXXX'이 다른 형식으로 재정의되어 있다.
(3) -wret* : 값이 있는 리턴과 없는 리턴이 양쪽 모두 사용되고 있다.
(4) -wstr* : 구조체의 멤버가 아닌 'XXXXXXXX'이 사용되고 있다.
(5) -wstu* : 미정의된 구조체 'XXXXXXXX'이 사용되고 있다.
(6) -wsus* : 의심스러운 포인터의 변환이 있다.
(7) -wvoi* : void형의 함수로 값을 리턴하도록 하고 있다. 이 함수는 값을
　　　　　　리턴하지 않을 수 있다.
(8) -wzst* : 구조에의 길이가 0 이다.

ᄒl
일반적인 에러
ᄒ
(1) -waus* : 'XXXXXXXX'에 사용되지 않는 값이 지정되고 있다.
(2) -wdef* : 정의하기 전에 'XXXXXXXX'이 사용되고 있다.
(3) -weff* : 코드의 영향이 없다.
(4) -wpar* : 인수 'XXXXXXXX'이 사용되고 있지 않다.
(5) -wpia* : 잘못된 지정이 있다.
(6) -wrch* : 제어가 도달할 수 없는 코드가 있다.
(7) -wrvl　: 함수가 값을 리턴하지 않고 있다.

ㅎl
특수한 에러
ㅎ
(1) -wamb : 연산이 애매모호하기 때문에 괄호가 필료하다.
(2) -wamp : 함수나 배열에 여분의 &가 붙어 있다.
(3) -wnod : 함수 'XXXXXXXX'의 선언이 없다.
(4) -wpro : 프로토타이프가 없는 함수를 호출하고 있다.
(5) -wstv : 구조체를 값으로 전달하고 있다.
(6) -wuse : 'XXXXXXXX'이 선언되었지만 사용되고 있지 않다.
ㅎl
이식성에 관한 경고
ㅎ
(1) -wapt* : 이식성이 없는 포인터으 지정이 있다.
(2) -wcln : 상수길이가 길다.
(3) -wcpt* : 이식성이 없는 포인터의 비교이다.
(4) -wrng* : 지정한 범위를 넘는 상수의 비교이다.
(5) -wrpt* : 이식성이 없는 리턴값의 형변환이 있다.
(6) -wsig : 변환에 의해서 유효숫자의 자릿수가 적어졌다.
(7) -wucp : 포인터가 signed 와 unsigned char 형으로 혼용되었다.
[참고] *는 디폴트로 ON을 의미하며 나머지는 디폴트가 OFF를 나타낸다.
3.7 세그먼트명 제어옵션

(1) -zAname : 코드 세그먼트의 분류(class)명을 name으로 변경시킨다.
디폴트로 코드 세그먼트는 CODE로 분류 지정된다.

(2) -zBname : 초기화 되지 않은 데이타 세그먼트의 분류명을 name으로
변경시킨다. 디폴트로 초기화되지 않은 데이터 세그먼트는
BSS로 분류 지정된다.

(3) -zCname : 코드 세그먼트명을 name으로 변경한다. 디폴트로 코드 세
그먼트명은 TEXT가 되고 medium, large, huge모델에서는
filename_TXT가 된다(filename은 소스 파일명이다).

(4) -zDname : 초기화되지 않은 데이타 세그먼트 명을 name으로 변경한다.
디폴트로 초기화되지 않은 데이타 세그먼트명은 _BSS로 된
다. hege 모델에서는 초기화가 안된 세그먼트가 생성되지
않는다.

------------------------------ 계 속 ------------------------------

(5) -zGname : 초기화 되지 않은 데이타 세그먼트의 그룹명을 name으로
변경한다. 디폴트로 데이타 그룹명은 DGROUP이 된다. huge
모델에서는 데이타 그룹이 없다.

(6) -zPname : 출력파일을 name으로 지정된 코드 세그먼트를 위해 코드
그룹으로 생성되게 한다.

(7) -zRname : 초기화된 데이타 세그먼트명을 name으로 설정한다. 디폴트
로 데이타 세그먼트명은 _DATA가 된다. huge 모델에서는
세그먼트명이 filename_DATA가 된다.

(8) -zSname : 초기화된 데이타 세그먼트 그룹명을 name으로 변경한다.
디폴트로 데이타 그룹명은 DGROUP이 된다. huge 모델에서는
데이타 그룹이 없다.

(9) -zTname : 초기화된 데이타 세그먼트 분류명을 name으로 설정한다.
디폴트로 초기화된 데이타 세그먼트 분류명은 DATA이다.

------------------------------ 계 속 ------------------------------

(10) -zX* : X 에 대해 디폴트명을 사용한다. 예를 들어 -zA*는 디폴트
분류명 CODE 를 코드 세그먼트에 지정한다.

`[주의]`사용자가 `8086 프로세서 상에서 세그멘테이션`에 대해 충분히
알고 있지 못한 경우에는 이들 스위치를 사용하지 않는 것이 좋다.
정상적인 환경하에서는 세그먼트명을 지정할 필요가 없다.

ㅠ]3.8 컴파일 제어 옵션

(1) -B : 인라인 어셈블리 코드를 처리하기 위해서 어셈블러를 호출하고
컴파일한다(통합환경 TC.EXE 에서는 사용할 수 없다).

(2) -c : 지정된 .C 와 .ASM 파일을 컴파일하고 어셈블한다. 그러나 링크
명령을 실행하지 않는다.

(3) -ofilename : 지정된 파일을 filensme.OBJ로 컴파일한다.

------------------------------ 계 속 ------------------------------

(4) -S : 지정된 소스 파일을 컴파일하고 어셈블리어로 출력파일(.ASM)을

생성한다(통합환경 TC.EXE 에서는 사용할 수 없다).

(5) -Efilename : 사용하는 어셈블러의 파일명으로서 filename을 사용한다.
디폴트로 TASM이 사용된다.

ᆩ4. 링커 옵션

(1) -efilename
파일 확장자 .EXE 를 부가함으로써 filename으로부터 실행 프로그램명을 이끌어 낸다(FILENAME.EXE). 이 옵션이 없는 경우 링커는 파일명 리스트에서의 첫번째 소스나 오브젝트 파일명으로부터 .EXE 를 이끌어 낸다.

(2) -M
링커에 대해서 링크맵을 출력하도록 지시한다. 디폴트는 링크맵을 생성하지 않는다.

------------------------------- 계 속 -------------------------------

(3) 1x
옵션 x 를 링커에 전달한다. 스위치 -1-x는 옵션 x 를 금지한다. -1 뒤에는 하나 이상의 옵션이 올 수 있다(TLINK 편을 참조).

ᆩ5. 환경 옵션

(1) -Idirectory
`인클루드 디렉토리`의 지정한다.
인클루드 파일을 위해 디렉토리, 드라이브 지정자 또는 서브 디렉토리의 패스명을 탐색한다(복수 지정이 가능).

(2) -Ldirectory
`라이브러리 디렉토리`의 지정한다.
링커가 지정된 디렉토리로부터 C0x.OBJ 기동 오브젝트 파일과 터보 C 라이브러리 파일(Cx.LIB, MATHx.LIB, EMU.LIB, FP87.LIB)을 불러 오도록 한다(디폴트는 현재 디렉토리를 탐색하게 한다).

------------------------------- 계 속 -------------------------------

(3) -nxxx
`출력 파일의 디렉토리`를 지정한다.

컴파일러에 의해 생성된 .OBJ나 .ASM 파일을 패스 xxx 가 지정하는 디
렉토리 또는 드라이브에 위치시킨다.

[형식] Library directories : -Ldirname[;dirname;...]
Include directories : -Idirname[;dirname;...]

[예제]

(1) 단일의 -L 또는 -I 옵션으로 복수개의 입력을 행할 수 있다.

tcc -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc2 myfile.c

(2) 명령행상에서 하나 이상의 옵션을 위치시킬 수 있다.

tcc -Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 myfile.c

------------------------------ 계 속 ------------------------------

(3) 복수개의 리스트와 혼합해서 사용할 수가 있다.

tcc -Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 myfile.c

[설명] 명령행상에서 복수의 -L 또는 -I 옵션을 리스트할 경우에 그 결과는
누적된다. 컴파일러는 나열된 모든 디렉토리를 탐색하거나 지정된
상수들을 왼쪽에서 오른쪽순으로 정의하게 된다.

ᄒI
Implicit 라이브러리 파일
ᄒ
터보 C가 `자동으로 링크`하는 것으로서 Cx.LIB파일, EMU.LIB 또는 FP87
.LIB, MATHx.LIB 그리고 기동 오브젝트 파일(C0x.OBJ)등이 있다.

ᄒI
Explicit(User-specified) 라이브러리 파일
ᄒ
사용자가 명령행이나 프로젝트 파일에서 명시적으로 리스트해야 한다.
.LIB 확장자를 갖는 파일명들이 속한다.
6. 인클루드와 라이브러리 파일의 탐색 알고리즘

ᄏJ6.1 인클루드(include)

(1) #include<somefile.h> : 지정된 인클루드 디렉토리에서만 SOMEFILE
.H 를 탐색하게 된다.

(2) #include "somefile.h" : 먼저 현재 디렉토리를 탐색한 다음 없으면
명령행의 지정된 인클루드 디렉토리에서
탐색한다.

ㅠJ6.2 라이브러리 탐색 알고리즘 - 인클루드 파일에 대한 것과 같다.

(1) Implicit 라이브러리 : 지정된 라이브러리에서만 탐색한다.

(2) Explicit 라이브러리 : 사용자가 라이브러리 파일명을 어떻게 리스트
하는가에 달려 있다(6.1 인클루드 참조).

7. 명령행 예제

ㅎI
[예제 1] tcc -IB:\include -LB:\lib -etest start.c body.obj end
ㅎ

[설명 1] 명령행 옵션

(1) 인클루드 디렉토리 : B:\INCLUDE(-IB:\include)이다.
(2) 라이브러리 : B:\LIB(-LB:\lib)이다.
(3) 실행가능한 결과 : TEST.EXE(-etest) 파일에 놓여져야 한다.

[설명 2] 리스트된 파일

(1) 컴파일해야 될 소스 파일 : STATIC.C
(2) 링크시 포함되어야 할 목적 파일 : BODY.OBJ
(3) 컴파일해야 될 다름 소스 파일 : END.C

ㅎI
[예제 2] tcc -IC:\include\ -LC:\lib2 -mm-c -k s1 s2.c z.asm mylib.lib
ㅎ

[설명 1] 명령행 옵션

(1) C:\INCLUDE(-IC:\include) 디렉토리에서 인클루드 화일을 찾는자.
(2) C:\LIB2(-LC:\lib2) 디렉토리에서 라이브러리 화일을 찾는다.
(3) Medium 메모리 모델(-mm)을 사용한다.
(4) 중첩된 커맨드(-c))를 허용한다.
(5) chars를 unsigned(-k)로 한다.

[설명 2] 리스트된 파일

(1) S1.C 와 S2.C 소스 파일이 컴파일된다.
(2) TASM을 사용하여 Z.ASM 파일이 어셈블된다.
(3) 실행가능 파일의 이름을 S1.EXE로 명명한다.
(4) MYLIB.LIB 라이브러리 파일이 링크시 링크된다.