제 0 장

테 스 트

▷ 테스트 1
▷ 테스트 2
▷ 테스트 3
▷ 테스트 4
▷ 테스트 5
▷ 테스트 6
▷ 테스트 7

이 장에서는 포인터에 관한 간단한 테스트를 해 보겠습니다. 자신이 얼마나 잘 풀 수 있는지 알아 보기 바랍니다.

포인터에 관해 잘 알지 못하는 사람은 전혀 맞추지 못할 수도 있습니다.

또 어느 정도 알고 있는 사람이라 할지라도 문제를 푸는 데 자신이 없을 수 있습니다. 그러나 걱정할 필요는 없습니다. 제 1장에서부터 제 9장까지 모두 공부한 후에는 이 테스트 문제들을 즉시 풀 수 있게 되기 때문입니다.

이 책을 서점에서 서서 읽고 계신 분들에게....

하나라도 잘 모르는 문제가 있다면 구입을 권하고 싶은데요(?)

테스트 1

다음의 함수

```
void f(int a[10])
{
    /* ... */
    a[2] = 0;
    /* ... */
}
```

에서 a는 배열이 아니라는 것과, a[10]의 10에는 의미가 없다는 것을 설명하십시오.

또, C 언어에서는 함수의 인자로 배열 그 자체를 주고 받을 수 없다는 것, 배열의 첨자가 0에서 시작하는(다른 언어에서처럼 1부터 시작된다든지 하한을 정할 수 있다든지 하지 않고) 이유를 설명하십시오.

테스트 2

다음 프로그램에서 각 문장의 의미를 설명하십시오.

```
int main(void)
{
  int a[10];

  a[0] = 1[a] = *(a+2) = *(3+a) = 0;
  4[a] = "ABC"[0];
  a[5] = 1["DEF"+1];
  return(0);
}
```

테스트 3

C 언어에는 1차원 배열만 있다는 것을 아래의 내용을 사용하여 설명하십시오.

```
int a[10][20];
```

테스트 4

다음 프로그램에서

```
void f(int a[][10])
{
  /* ... */
  a[2][3] = 0;
  /* ... */
}
int main(void)
{
    int x[5][10];

    f(x);
    /* ... */
    return(0);
}
```

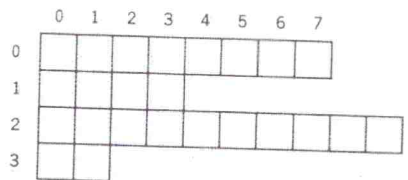
f 함수의 인자 a의 수형은 무엇입니까?

main 함수에서 호출된 함수 f(x)의 x는 &x, &x[0], 또는 &x[0][0] 중 어느 것이 되겠습니까?

테스트 5

동적 메모리 할당의 방법을 설명하십시오. 또, 아래의 그림처럼 행에 대한 열의 크기가 다른 배열을 동적으로 할당하는 프로그램을 작성하십시오.

단, 각 요소는 a[i][j]의 표현으로 액세스할 수 있어야 합니다.



테스트 6

표준 입출력의 화일 처리에서 반드시 사용하는

FILE *p;

선언의 의미를 설명하십시오. 또 fopen 함수가 하는 일에 대하여 구체적으로 설명하십시오.

테스트 7

다음 프로그램은 화일을 오픈하는 open_files 함수인데 제대로 동작하지 않습니다. 버그를 찾으십시오.

```
#include <stdio.h>

int open_files(FILE *fpi, FILE *fpo)
{
    if ((fpi = fopen("input.dat", "r")) == NULL) return(1);
    if ((fpo = fopen("output.dat", "w")) == NULL) return(1);
    return(0);
}

int main(void)
{
    int  flag;
    FILE *input, *output;

    flag = open_files(input, output);
    /* ... */
    return(0);
}
```

어떻습니까? 당신은 어느 정도입니까?

앞으로 여러분께서는 C 언어를 제일 잘 알고 있는 선생님과 초보자 학생과의 문답을 통하여 공부를 하게 될 것입니다.

이책에서 특별히 언급하지 않는 한 ANSI 규격의 C 언어를 대상으로 하겠습니다. 또 ANSI, K&R이라는 단어를

ANSI --- ANSI X3J11 위원회에서 정의한 C 언어의 규격
K&R --- Kernighan & Ritchie 저「프로그래밍 언어 C」

의 의미로 사용하겠습니다.

제 1 장

포인터의 기본

▷ 1-1 포인터란
▷ 1-2 함수의 호출과 포인터

C 언어에서는 빈번하게 포인터가 사용됩니다. 포인터란 객체(변수)나 함수 등의 어드레스를 값으로 갖는 것입니다. 포인터는

(1) 포인터를 사용하는 편이 다른 방법보다도 간결하고 효율적으로 표현할 수 있다.
(2) 포인터를 사용하지 않는다면 프로그램을 수행할 수 없는 경우가 있다

는 등의 이유 때문에 사용되고 있습니다.

C 언어의 초보자로서 포인터를 이해한다는 것은 어려운 일입니다. 그러나 포인터를 마스터하지 않으면 C 언어를 마스터할 수 없습니다. 이 장에서는 포인터에 대한 기본적인 사항을 설명하고 있습니다.

이 책과 비슷한 종류의 책에서는 흔히 포인터를 「변수를 가리킨다」라고 하고 이를 시각적으로 설명할 경우 →(화살표)를 사용하지만 이 책에서는 이때까지 사용하지 않았던 훨씬 쉬운 방법으로 설명하고 있습니다. 이 장을 완전히 이해하고 다음 장으로 넘어가기 바랍니다.

▷ 1-1 포인터란

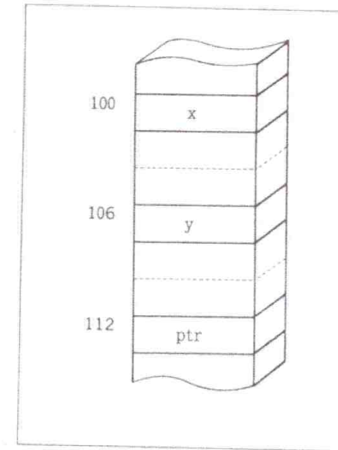
▷ …… 1-1-1 포인터 변수의 선언과 어드레스

— 포인터 변수에 대해 잘 모르겠습니다. 도대체 포인터 변수란 무엇입니까?

포인터 변수란 「변수를 가리키는」 변수라고 생각해 주십시오.

— 변수를 가리킨다구요?

변수는 일반적으로 메모리상의 어딘가에 저장되어 있습니다. 주소를 1614 번지라고 말하는 것처럼 메모리에도 주소가 있습니다. 이 주소를 어드레스(address)라고 합니다.



[그림 1-1] 변수와 어드레스

그림 1-1에서 변수 x와 y는 각각 100과 106번지에 저장되어 있습니다. 변수를 가리킨다는 것은 그 어드레스를 기억하는 것입니다.

— 그 설명으로는 전혀 감이 잡히지 않는데요.

좀더 구체적으로 설명해 주십시오.

int형의 변수 x와 y의 선언은 아래와 같이 합니다.

```
int x;
int y;
```

int형의 변수를 가리키는 포인터 변수 ptr은 다음과 같이 선언합니다.

```
int *ptr;
```

— *는 어떤 의미입니까?

ptr을 단순한 <int형>이 아닌, <int의 포인터형>이라고 선언하기 위한 지정자라고 생각해 주십시오.

1-1-2 간접 연산자, 어드레스 연산자

프로그램 중에 선언 이외의 곳에서 포인터 변수 앞에 *를 붙이면 그 포인터가 가리키는 변수의 실체—바꾸어 말하면 가리키고 있는 변수의 내용—를 나타내게 됩니다. 이 * 연산자를 간접 연산자(indirection operator)라고 부르겠습니다.

또, 여러분은 * 연산자와 관련되어 & 연산자를 꼭 기억해야 합니다. & 연산자를 변수 앞에 붙이면 그 변수의 어드레스 값을 나타냅니다. 이 & 연산자를 어드레스 연산자(address operator)라고 부릅니다.

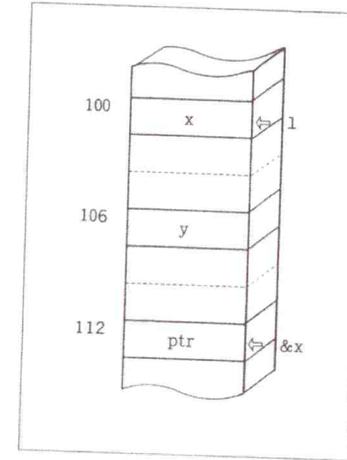
— 저, 머리가 혼란해지는 데요...

보다 구체적으로 설명해 나가겠습니다. 그림 1-1을 보면 x는 100번지에 저장되어 있기 때문에 &x는 100입니다. y의 어드레스는 106이기 때문에 &y는 106입니다.

여기에서

```
x = 1;
ptr = &x;
```

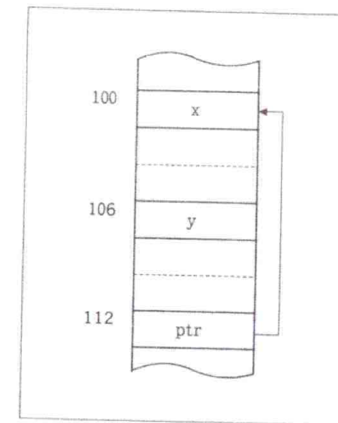
의 문장을 수행하면 그림 1-2와 같이 첫문장에서는 x에 1이 대입되고 둘째 문장에서는 ptr에 &x 즉, 100이 대입됩니다. 이것은 이해가 되겠죠?



[그림 1-2] 변수에 대입

— 예, 알겠습니다. 단지 값이 대입되어 있을 뿐이군요.

이처럼 포인터 변수(ptr)가 변수(x)의 어드레스를 가질 때, 「ptr은 x를 가리킨다」라고 하며 포인터 변수는 변수를 가리키므로 다른 책에서는 그림 1-3처럼 설명합니다.



[그림 1-3] 포인터 설명

int형의 변수 x와 y의 선언은 아래와 같이 합니다.

```
int x;
int y;
```

int형의 변수를 가리키는 포인터 변수 ptr은 다음과 같이 선언합니다.

```
int *ptr;
```

― *는 어떤 의미입니까?

ptr을 단순한 <int형>이 아닌, <int의 포인터형>이라고 선언하기 위한 지정자라고 생각해 주십시오.

1-1-2 간접 연산자, 어드레스 연산자

프로그램 중에 선언 이외의 곳에서 포인터 변수 앞에 *를 붙이면 그 포인터가 가리키는 변수의 실체―바꾸어 말하면 가리키고 있는 변수의 내용―를 나타내게 됩니다. 이 * 연산자를 간접 연산자(indirection operator)라고 부르겠습니다.

또, 여러분은 * 연산자와 관련되어 & 연산자를 꼭 기억해야 합니다. & 연산자를 변수 앞에 붙이면 그 변수의 어드레스 값을 나타냅니다. 이 & 연산자를 어드레스 연산자(address operator)라고 부릅니다.

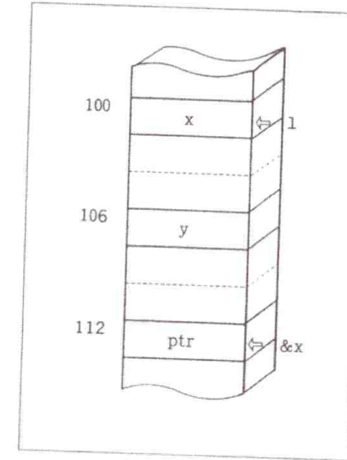
― 저, 머리가 혼란해지는 데요…

보다 구체적으로 설명해 나가겠습니다. 그림 1-1을 보면 x는 100번지에 저장되어 있기 때문에 &x는 100입니다. y의 어드레스는 106이기 때문에 &y는 106입니다.

여기에서

```
x = 1;
ptr = &x;
```

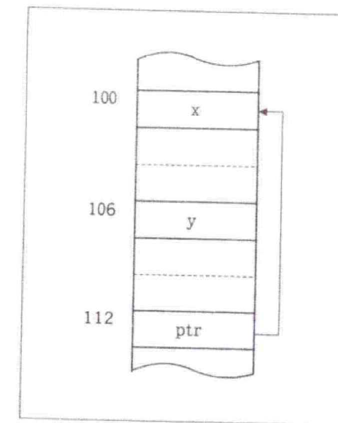
의 문장을 수행하면 그림 1-2와 같이 첫문장에서는 x에 1이 대입되고 둘째 문장에서는 ptr에 &x 즉, 100이 대입됩니다. 이것은 이해가 되겠죠?



[그림 1-2] 변수에 대입

― 예, 알겠습니다. 단지 값이 대입되어 있을 뿐이군요.

이처럼 포인터 변수(ptr)가 변수(x)의 어드레스를 가질 때, 「ptr은 x를 가리킨다」라고 하며 포인터 변수는 변수를 가리키므로 다른 책에서는 그림 1-3처럼 설명합니다.



[그림 1-3] 포인터 설명

— 이 그림으로는 전혀 모르겠습니다.

☆ 중 요 ☆

포인터 변수 ptr이 변수 x의 어드레스를 가질 때 「ptr은 x를 가리킨다」라고 한다.

거의 모든 C 언어에 관한 책들에서는 그림 1-3과 같은 방법으로 설명되어 있습니다. 이 그림은 확실하게 「포인터 변수는 다른 변수를 가리킨다」라고 하는 이미지를 잘 나타내고 있습니다. 그러나 학생의 말처럼 이 그림으로써는 잘 이해할 수가 없습니다. 그건 그렇다 하고, x 및 ptr에 & 연산자 및 * 연산자를 붙이면 표 1-1과 같이 됩니다.

[표 1-1] 변수와 포인터 변수의 값

x	1	&x	100	*x	-
ptr	100	&ptr	112	*ptr	1

여기에서 중요한 것은 포인터 변수라고 해서 특별하지는 않다는 점입니다.

ptr에 &x 즉, 100이 대입되었기 때문에, ptr은 단지 100이라고 하는 값을 갖는 것입니다.

— &ptr의 값이 어떻게 112가 됩니까? 또, 포인터 변수에 &를 붙여도 괜찮습니까?

포인터 변수는 변수를 가리킨다라고 하는 특징을 가진 변수입니다. 변수인 이상 메모리상의 어딘가에 저장되어 있습니다. 이 경우 그림 1-1에서 ptr은 112 번지에 저장되어 있습니다. 따라서 &ptr은 112가 되는 것입니다. 반복해서 말하지만 앞에서도 말했던 것처럼 포인터 변수라고 해서 특별하지는 않습니다.

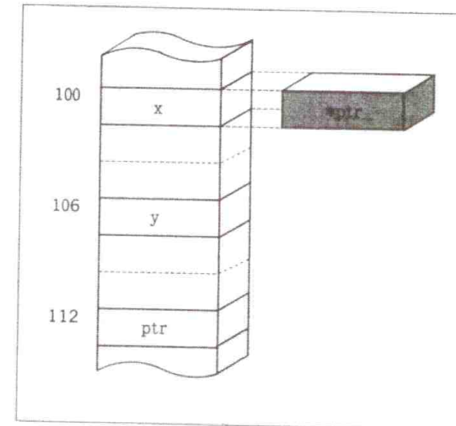
☆ 포인트 ☆

포인터 변수는 보통의 변수이다.

— *ptr의 값이 어떻게 1이 됩니까?

처음에서 말한 것처럼 포인터 변수에 * 연산자를 붙이면 그 포인터 변수가 가리키는 변수의 실제 내용을 나타내게 됩니다. 덧붙여서 말하면, * 연산자는 포인터에만 적용할 수 없기 때문에 *x는 의미가 없게 됩니다. 따라서 *ptr이라고 표기하면 ptr이 가리키고 있는—즉, 100 번지의—int형 정수를 간접적으로 나타내게 됩니다.

— 그러면 위에서 말씀하신 것은 ptr이 x를 가리키고 있을 때, *ptr과 x는 같은 것을 나타낸다는 것입니까?



[그림 1-4] 이 책의 포인터 변수 설명

그렇습니다. 좋은 지적을 해 주셨습니다. 일반적으로 ptr이 x를 가리킬 때, *ptr은 x의 앨리어스(alias : 별명)가 됩니다. 다시 말하면, x를 나타내는 변수인 기억 장소에 *ptr이라고 하는 「다른 명칭」이 붙어 있다고 생각하면 됩니다. 포인터 변수를 설명하는 데는 앞의 그림 1-3보다는, 그림 1-4를 보는 편이 이해하기 쉽습니다.

◎ 매우 중요 ◎

포인터 변수 ptr이 x를 가리키고 있을 때, *ptr은 x의 실제 내용을 나타낸다. 즉, *ptr은 x의 앨리어스(별명 : 다른 명칭)가 된다.

— 알겠습니다. *ptr과 x가 같은 것을 나타내고 있기 때문에 이 그림이 이해하기 쉽습니다.

다음의 설명도 모두 이 그림과 같은 표현을 사용하고 있기 때문에 완전히 이해하도록 하십시오. 여기에서 중요한 점은 ptr이라고 하는 변수는 존재하지만 *ptr이라고 하는 변수는 존재하지 않는다는 점입니다.

— 좀더 자세히 설명해 주시지 않겠습니까.

예를 들어 a라고 하는 int형 변수가 있고, 이 a의 값이 50이라고 해 봅시다. 그러면 -a의 값은 얼마입니까?

— 당연히 -50입니다.

그렇습니다. a라고 하는 변수가 있고 이것에 단항 연산자 —를 적용했을 뿐이지, -a라고 하는 변수가 어딘가에 존재하는 것은 아닙니다.

포인터 ptr에 관해서도 이와 같은 의미입니다. 포인터 변수 ptr에 * 연산자를 적용한 것이 x의 앨리어스가 될 뿐이지 실제로 *ptr이라고 하는 변수가 있는 것은 아닙니다.

— * 연산자는 포인터 변수가 가리키고 있는 변수를 간접적으로 액세스하기 위한 연산자이기 때문에 간접 연산자라고 불리워지는 것입니까?

그렇습니다. 그럼 다시 처음으로 돌아가서 포인터 변수의 선언 방법을 한 번 생각해 보겠습니다. int형의 변수 x와 int의 포인터형 변수 ptr의 선언은 어떻게 되겠습니까?

— 이제 알고 있습니다.

```
int x;
int *ptr;
```

입니다.

여기에서 아래와 같이 생각해 보면, int x;의 x 부분이 *ptr에 대응한다고 생각되지 않습니까?

```
int   x    ;
int  *ptr  ;
```


— *ptr은 int형 정수이다" 라고 해석할 수 있습니까?

ptr이 포인터 변수라는 것과 *ptr은 int형이라는 두 가지를 함께 생각하면 이해하기 쉽습니다.

그럼 다음 문제입니다. ptr, pc의 두 개 변수를 int의 포인터형으로 한 번에 선언하고 때 어떻 1 하면 되겠습니까?

— 이렇게 하면 됩니다.

```
int *ptr, pc;
```

유감이지만 이것은 다음과 같이 선언한 것입니다.

```
int *ptr;
int pc;
```

한 번에 기술하고 싶을 때에는 다음처럼 해야만 합니다.

```
int *ptr, *pc;
```

각 변수에 각각 *를 붙이는 것을 잊지 않도록 하십시오(컬럼 1-1 참조).

이것으로써 <어드레스 연산자>, <간접 연산자>라고 하는 말의 의미를 알았다고 생각합니다. 이제 다음 프로그램은 이해할 수 있을 것입니다.

```
int x = 1, y = 5;
int z[10];
int *p;

p = &x;        /* p는 x를 가리킨다(x의 어드레스가 대입됨) */
y = *p;        /* *p는 x의 앨리어스이기 때문에 y에 *p=x=1이 대입된다 */
*p = 0;        /* *p는 x의 앨리어스이기 때문에 *p 즉, x가 0이 된다 */
p = &z[2];     /* p는 z[2]를 가리킨다 */
```

— 배열에 & 연산자를 붙여도 괜찮습니까?

그 표현은 조금 틀립니다. 여기에서 &z[2]라고 한 표현은 배열에 & 연산자를 붙인 것이 아니고, 배열의 한 요소에 & 연산자를 적용한 것입니다. z[2]라고 하는 것은 단순히 int형의 변수, 즉, 하나의 변수입니다. 따라서 a의 어드레스를 &a라고 표현하듯이 z[2]의 어드레스를 &z[2]라고 표현할 수 있는 것입니다.

□ 컬럼 1-1 □

define ... typedef

어떤 범위의 정수(예를 들어 0에서 100,000까지)를 나타낼 수 있는 수형을 만들고 싶다고 하자.

Turbo C, MS-C 등 16비트 기종의 컴파일러의 경우에는 int형으로는 이와같은 범위의 수를 나타낼 수 없기 때문에 long형을 사용하기로 해 보자.

다음과 같은 매크로를 선언하여 사용하면 편리하다.

```
#define INTEGER     long
#define INTEGERptr  long *
```

위에서처럼 INTEGER라는 매크로 정의를 하여 원하는 수형을 갖는 변수나 그 수형을 가리키는 포인터의 선언 등에 사용할 수 있다. 예를 들면 다음과 같은 선언이 가능하게 된다.

```
INTEGER     a;
INTEGERptr  p;
```

이것은 다음과 같이

```
long a;
long *p;
```

직접 표현하는 방법과 비교해 보면 다음과 같은 장점이 있다.

(1) 프로그램을 한번 보고서도 다른 용도(보통의 long형)의 변수와 구분할 수 있게 된다.

(2) 예를 들면 int형으로써 0에서 100,000까지의 범위를 나타낼 수 있는 다른 컴파일러에 프로그램을 이식할 경우, 매크로 정의를

```
#define INTEGER     int
#define INTEGERptr  int *
```

라고 변경하기만 하면 되기 때문에 쉽고,프로그램 내용을 수정할 필요가 없게 된다.

C 언어의 매크로는 강력하기 때문에 이와 같이 편리하게 사용할 수 있다.

그러나 매크로의 사용 시기를 잘 알지 못하면 매우 곤란한 경우도 생긴다.

```
INTEGER     a, b;
INTEGERptr  p, q;
```

를 매크로 전개하면 어떻게 될까? 매크로는 단순히 치환만 하는 것이기 때문

에 위의 문자는 다음과 같이 된다.

```
long a, b;
long *p, q;
```

전혀 생각하지 않았던 결과가 된다. p는 long의 포인터 변수가 되지만 q는 단지 long형의 변수가 되어 버린다.

이런 경우에는 typedef를 사용한다.

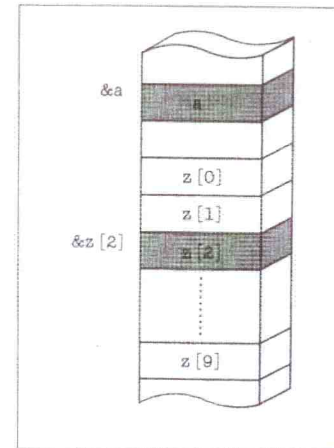
```
typedef long   INTEGER;
typedef long * INTEGERptr;
```

이 방법을 사용하여 다음과 같이

```
INTEGER     a, b;
INTEGERptr  p, q;
```

라고 선언하면 p, q는 모두 long의 포인터형으로 해석되어 원했던 결과를 얻을 수 있다.

그림 1-5를 보면 잘 이해할 수 있을 것입니다. 따라서 z[2]의 어드레스는 &z[2] 라고 표현할 수 있습니다(컬럼 1-2 참조).



[그림 1-5] 어드레스 연산

□ 컬럼 1-2 □

배열명에 &를 붙이면...

이 책의 문답에서 &z[2] 처럼 배열의 요소에 & 연산자를 적용하는 경우를 설명하였다. 그런데 &z라는 표현은 가능할까?
예전의 대부분의 C 컴파일러에서는 다음 두 가지로 해석된다.
(1) 문법적으로 옳지 않다. 즉, 컴파일시 에러로 나타난다.
(2) &z[0]와 같은 것으로 간주한다. 즉, 배열의 첫요소의 포인터이다.
어느쪽으로 해석되든지 컴파일러간의 호환성은 없게 된다. 지금까지 일부 컴파일러에서 사용되고 있던 &z의 해석이 ANSI 규격에서 정식으로 인정되었다. 즉,
(3) &z는 배열의 포인터다.
이것은 (2)의 해석과 같은 값을 갖지만 형이 틀린 것에 주의하기 바란다.
(2)에서는 배열 중의 하나인 한 요소의 포인터이기 때문에 「int의 포인터 <int *>형」을 가진다. (3)은 배열의 포인터이기 때문에 「int형의 크기 10인 배열의 포인터 <int (*)[10]>형」을 갖게 된다.
배열의 포인터에 관해서는 제 3장에서 설명하고 있다.

▶ 문제 1-1

다음 프로그램을 수행하여 동작을 확인하고 프로그램의 의미를 설명하시오.

```
#include <stdio.h>

int main(void)
{
    int  x,y;
    int  *p;

    p = &x;
    *p = 1;
    p = &y;
    *p = 2;
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    return(0);
}
```

— & 연산자나 * 연산자는 의미를 이해하면 의외로 간단하군요.

포인터 공부를 하면 할수록 점점 어려워지기 때문에 기본적인 내용을 정확히 파악해 두지 않으면 안됩니다. 완전히 이해해 주십시오.

& 연산자를 사용할 때에 꼭 알아 두어야 할 것이 있습니다. 예를 들어

```
register int i;
```

와 같이 **register 기억 클래스 지정자**(storage class specifier: 컬럼 1-3 참조)가 지정된 변수에 대해서는 & 연산자를 사용할 수 없습니다. 왜냐 하면 이와 같이 선언된 변수는 메모리상이 아닌 레지스터(register)라고 하는 CPU 내부의 특수한 장소에 저장되어 있기 때문에 주소를 가지고 있지 않습니다.

☆ 주 의 ☆

레지스터 변수에는 & 연산자를 사용할 수 없다.

— 레지스터를 지정하면 어떤 장점이 있습니까?

레지스터는 메모리와는 다르게 CPU 내부에 존재하기 때문에 고속 연산이 가능합다. 루프(loop)에 사용하는 변수나 자주 액세스하는 변수에 사용하면 좋겠지요.

컴파일러에 따라 레지스터에 저장하지 않고 보통 메모리 상에 저장하기도 하고, 또한 레지스터의 수가 한정되어 있기 때문에 실제로는 몇개 정도 밖에 레지스터에 저장되지 않습니다.

□ 컬럼 1-3 □

기억 클래스 지정자(storage class specifier)

기억 클래스 지정자라는 것은 기억 클래스(식별자가 표시하는 실제의 기억 기간이나 링크의 상태)를 지정하는 선언 지정자이다.
기억 클래스 지정자에는 다음과 같은 것이 있다.

```
typedef
extern
static
auto
register
```

여기에서 typedef는 문장 구성상 기억 클래스 지정자로 분류되어 있을 뿐이지 실제로 기억 클래스를 지정하는 것은 아니다.
기억 클래스 지정자는 사용되는 장소(블럭의 내부에 있는지, 함수의 외부에 있는지 등...)에 따라 여러가지로 의미가 변하는 등의, 사용 방법이 여러가지인 지정자이다.

요약해서 간단히 나타내면

- extern 외부 변수와 링크될 것임을 지정한다.
- static 블럭 내에서 지정할 경우 프로그램 수행중 정적으로 계속 유지됨을 지정한다. 블럭 밖에서 지정할 경우 화일 내부에서만 사용함을 지정한다.
- auto 지정한 블럭이 사용될 경우에만 자동으로 생성되어 유효하게 되는 변수임을 지정한다.
- register auto의 일종으로 고속 연산이 가능한, 자동으로 생성, 소멸되는 변수임을 정한다.

□ 컬럼 1-4 □

레지스터 변수

register 기억 클래스 지정자를 사용하여 선언한 객체를 일반적으로 레지스터 변수라고 부른다. 8086 CPU 상에서 동작하는 컴파일러의 경우 SI 및 DI 두 개의 레지스터가 레지스터 변수로 사용되는 경우가 많은 듯하다. 이들은 16비트(2바이트) 길이의 크기를 갖고 있기 때문에 2 바이트보다 큰 객체(long, float, double형 등)는 저장할 수 없다. MS-C (Ver. 6.0 이후)는 가능한한 많은 변수를 레지스터에 할당해 고속 연산이 가능하도록 최적화하게 된다.

— 다른 주의할 점은 없습니까?

포인터는 변수 뿐만 아니라 함수를 지정할 수도 있습니다. 이것에 관해서는 제6장에서 설명하도록 하겠습니다.

▷ 1-2 함수의 호출과 포인터

▷ ····· 1-2-1 값에 의한 호출(call by value)

— 그런데 여기까지의 설명에서는 포인터 변수가 부자연스러운 예를 통해서 사용되어졌지만 실제는 어떤 경우에 사용합니까?

그 이야기는 뒤로 돌리고 우선 2개의 int형 변수를 교환하는 함수와 그것을 이용한 프로그램을 작성해 보십시오.

— 예, 할 수 있습니다.

[리스트 1-1] 정수를 교환하는(틀린) 프로그램

```
/*
    List 1-1  정수를 교환하는(틀린) 프로그램
*/

#include <stdio.h>

void swap(int x, int y)
{
    int   temp;

    temp = x;
    x    = y;
    y    = temp;
}

int main(void)
{
    int   a = 5;
    int   b = 3;

    swap(a, b);
    printf("A = %d\n", a);
    printf("B = %d\n", b);
    return(0);
}
```

수행해 보십시오.

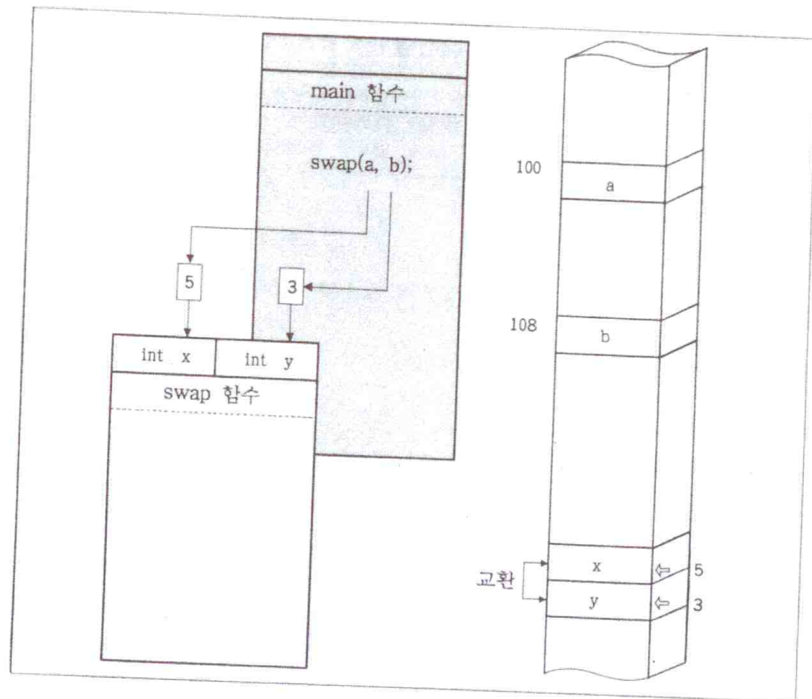
■ 수행 결과 ■

```
A = 5
B = 3
```

— 아! a가 5이고 b가 3 그대로인데요. 바뀌지 않았습니다.

C 언어에서는 함수의 호출을 값에 의한 호출(call by value)로 수행합니다. 이것을 설명하면, 호출하는 쪽은 실인자(argument)로 값을 넘겨 주고 호출되는 쪽은 형식 인자(parameter)로 받은 값의 복사본을 사용하는 것입니다.

— 복사본이라고요?



[그림 1-6] 값에 의한 호출

이 예에서 main 함수는 a, b라는 변수를 실제로 넘겨 주는 것이 아니고, 그것의 값 5, 3을 넘겨 줍니다. 호출된 swap 함수는 그 값을 x, y의 초기값으로 받습니다.

swap 함수 중에서 아무리 x와 y를 교환해도 받은 복사본을 교환할 뿐이지 원래의 a, b를 교환하는 것은 아닙니다. 따라서 main 함수의 a, b는 swap 함수를 호출한 후에도 각각 5와 3 그대로입니다.

— 호출하는 쪽의 인자와 호출되는 쪽의 인자는 실제로 아무관계가 없는 것입니까?

그렇습니다. 알기 쉽게 설명하면 호출하는 쪽은 강의 상류로부터 "값"을 방류한다고나 할까요.

1-2-2 값에 의한 호출과 포인터

— C 언어에서는 인자를 교환할 수 없는 것입니까?

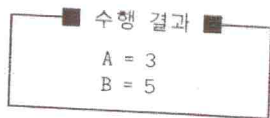
네, 맞습니다. C 언어에서는 모든 함수 호출이 <값에 의한 호출>이기 때문에 그렇습니다. 그러나 포인터를 사용하면 마치 인자를 교환한 것처럼 보일 수도 있습니다. 다음의 프로그램을 봐 주십시오.

[리스트 1-2] 정수를 교환하는 프로그램

```
/*
    List 1-2 정수를 교환하는 프로그램
*/

#include <stdio.h>

void swap(int *x, int *y)
{
    int   temp;

    temp = *x;
    *x   = *y;
    *y   = temp;
}

int main(void)
{
    int   a = 5;
```

```
        int    b = 3;

        swap(&a, &b);
        printf("A = %d\n", a);
        printf("B = %d\n", b);
        return(0);
}
```

수행해 보십시오.



— 이번에는 a와 b가 정말로 바뀌었습니다.

그림 1-7을 봐 주십시오. main 함수의 호출

```
swap(&a, &b);
```

의 &a, &b는 이제 알 수 있겠죠. 각각 a와 b의 어드레스입니다.

— 이경우 100과 108이 되겠군요.

호출된 swap 함수

```
void swap(int *x, int *y)
```

의 *x, *y는 무엇이 될까요. x, y는 int의 포인터형입니다. 각각 받은 값 즉, a, b의 주소가 복사됩니다. 따라서 x는 a를 가리키고 y는 b를 가리키게 됩니다.

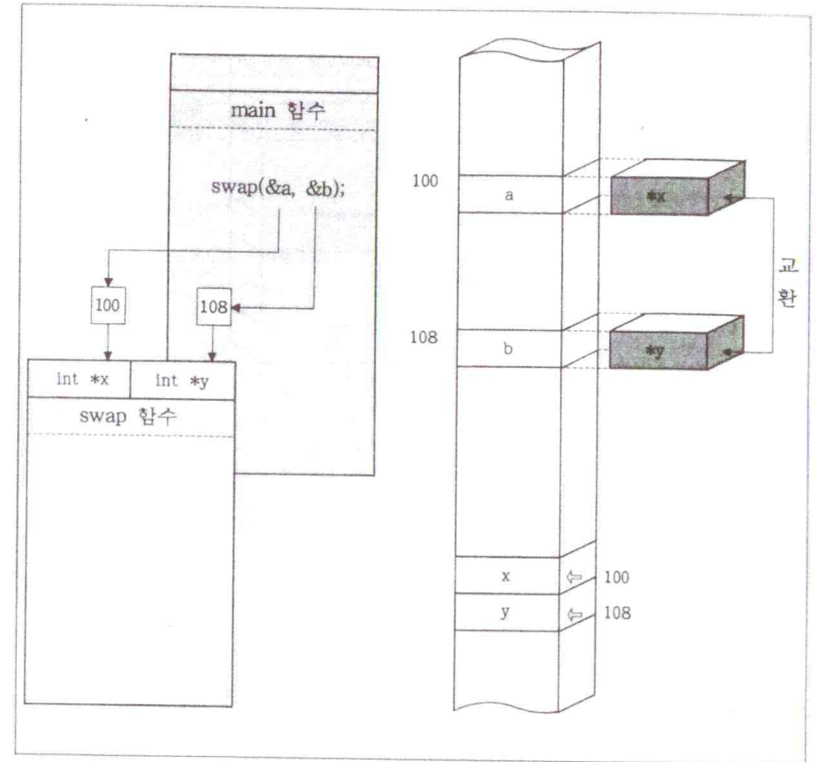
— 100과 108이라고 하는 값은 x, y에 복사되는 것입니까?

그렇습니다. 100과 108은 x, y에 복사되는 것이지 *x, *y에 복사되는 것이 아님에 주의해야 합니다.

— x, y는 각각 a, b를 가리키고 있기 때문에 *x, *y가 a, b의 앨리어스가 된다는 말씀이지요?

맞습니다. 그림을 보면 빨리 이해가 될 것입니다. 즉, *x와 *y를 교환하기 때문에 결국 a와 b를 교환하는 것과 같은 것이 됩니다.

— 이렇게 해서 인자가 교환되는 군요.



[그림 1-7] 포인터를 이용한 값에 의한 호출

□ 컬럼 1-5 □

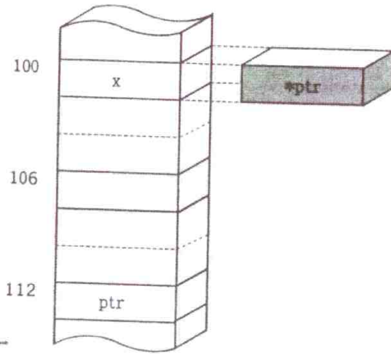
어드레스 연산자와 포인터

처음에 어드레스 연산자(&)는 그 객체의 어드레스를 갖기 위한 연산자라고 설명했다.

```
int  x;
int  *ptr;

ptr = &x;
```

이제는 이 프로그램을 이해할 수 있을 것이다. ptr에 x의 어드레스(이 경우 100)를 대입함에 따라 ptr이 x를 가리키게 된다. 이 책에서는 그림 처럼 100 등의 수치를 사용하고 있다.

그러나 이것은 어디까지나 가상적인 것이다. 여기에서 「어드레스 연산자는 객체의 어드레스를 갖기 위한 연산자」라는 설명을 자세히 하겠다.




```
ptr = &x;
```

는 int의 포인터형을 갖는 변수 ptr에 &x를 대입하고 있다. 그러므로 대입 &x가 int의 포인터형이 아니라고 한다면 말이안된다.

실제로 &x의 어드레스 연산자 &는 「그 객체형의 포인터를 생성하는 연산자」라고 할 수 있다. 예를 들어 이 경우 &x는 int의 포인터형을 가지며 그 값으로 x의 어드레스를 갖는 것이 된다. C 언어에서 단순하게 100이라고 한다면 int형의 정수가 되어 버린다. 그러나 여기서의 100은 어디까지나 int의 포인터형으로써의 100이다. 이것을 엄밀하게 표현하면(int*) 100이라고 해야 한다. swap 함수의 설명도, 본문에서는 「&a에서 a의 어드레스를 계산해 그 값을 포인터 x에 초기값으로 대입한다」라고 했다. 그러나 보다 엄밀히 말하면 「&a에서 a의 포인터를 생성해 그 값을 x의 초기값으로 대입한다」라는 것이 된다. 이 책에서는 독자들이 이해하기 쉽게 하기 위해서 어드레스는 (int *) 100이라고 표현하지 않고 그냥 100이라고 쓰고 있다. & 연산자에 대해서도 「어드레스를 갖는다」, 「포인터를 생성한다」 중 경우와 때에 따라서 (이해하기 쉽도록) 적절히 사용하고 있다.

아닙니다. 앞에서도 말했던 것처럼 C 언어에서는 값에 의한 호출밖에 할 수 없습니다. 이것은 값만을 넘겨주고 「마음대로 하십시오」라고 말하는 것과 같습니다. 즉, 이것으로써는 인자를 교환할 수 없습니다. 그래서, 어드레스라고 하는 "값"을 넘겨줍니다. 받는 쪽은 그것을 포인터로 받아서, *x, *y를 a, b의 앨리어스로 사용해, 호출하는 쪽의 a, b를 간접적으로 액세스할 수 있다는 것입니다. 여기에서 넘겨준 인자는 어드레스(100과 108)이고 이 인자는 교환되지 않습니다. 그럼 이 예를 통해서도 * 연산자가 간접 연산자로 불리워지고 있는 이유를 잘 알겠지요.

— 잘 알겠습니다. 그러나 좀 까다로운데요.

그렇습니다. 좀더 이해를 바르게 하기 위해서 C에는 없는 참조에 의한 호출(call by reference)을 공부해 보겠습니다.

리스트 1-3의 프로그램을 봐 주십시오.

[리스트 1-3] C++에서 정수를 교환하는 프로그램

```
/*
    List 1-3 C++에서 정수를 교환하는 프로그램
*/

#include <stdio.h>

void swap(int& x, int& y)
{
    int   temp;

    temp =  x;
    x    =  y;
    y    =  temp;
}

int main(void)
{
    int  a = 5;
    int  b = 3;

    swap(a, b);
    printf("A = %d\n", a);
    printf("B = %d\n", b);
    return(0);
}
```

■ 수행 결과 ■

```
A = 3
B = 5
```

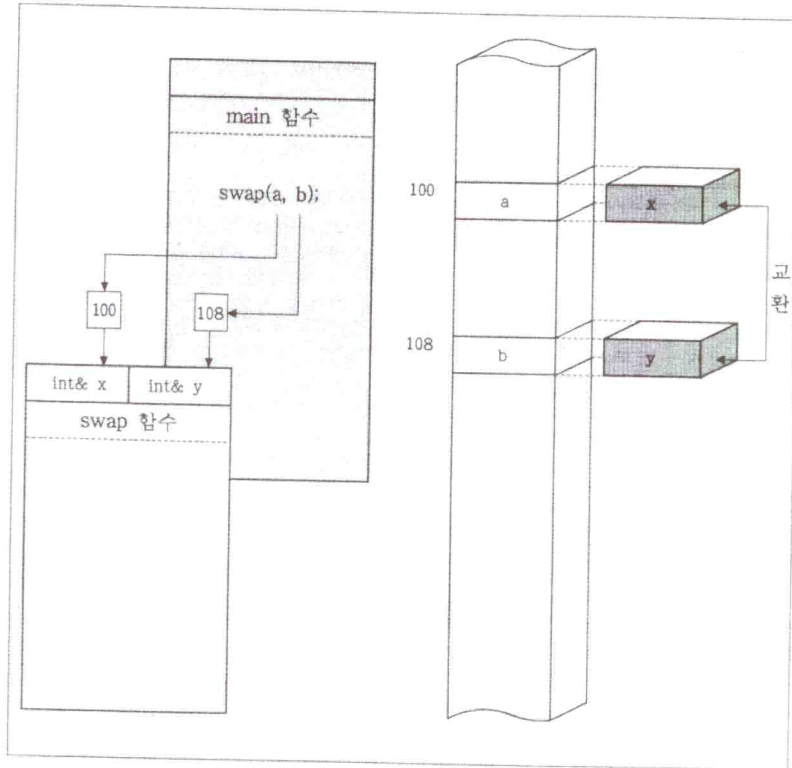
— 이 언어는 C 언어와 비슷한데요...

이 프로그램은 C++ 언어로 작성되어 있는데 상세한 설명은 생략하겠습니다.

swap 함수는 a, b의 번지를 받고 x, y는 그 번지를 사용하게 됩니다. 즉, x, y의 앨리어스가 됩니다. 그림 1-8을 보면 호출하는 쪽의 a, b와 swap 함수의 x, y가 일

대 일로 대응하고 있는 점이 C 언어와는 좀 다르지만 간결하게 표현되어 있습니다.

— C 언어의 경우보다도 C++이 간결하여 보기만 해도 기분이 좋습니다.



[그림 1-8] 참조에 의한 호출(C++ 언어)

그렇습니다. C 언어에서는 <참조에 의한 호출>이 불가능하기 때문에 「호출하는 쪽의 변수의 값을 바꾸려면 포인터를 간접적으로 사용해야 한다」는 것이 됩니다.

— 고작 변수를 1개 넘겨주는 데에 부르는 쪽도 불려지는 쪽도 포인터를 사용해 성가시게 처리를 하지 않으면 안된다는 바보스런 이야기군요.

✓☆ 체크 포인트 ☆

C 언어에서는 값에 의한 호출만 가능하고 참조에 의한 호출은 할 수 없다.

몇개의 언어에서 <값에 의한 호출>과 <참조에 의한 호출>의 가능 여부를 표 1-2에 정리해 보았습니다.

[표 1-2] 언어와 함수의 호출

	값에 의한 호출	참조에 의한 호출
C	○	×
C++	○	○
Pascal	○	○
FORTRAN	×	○

— FORTRAN은 참조에 의한 호출이 가능하니까 C 언어보다 발전된 언어가 아닐까요?

아닙니다. <값에 의한 호출>에 의해 각 모듈—C 언어에서의 함수라고 생각해 주십시오—의 독립성이 높아지게 되어 인자를 주고 받는 방법이 간단하게 되는 장점이 있습니다. 오히려 <참조에 의한 호출>은 호출하는 쪽과 호출된 쪽의 변수의 결합이 지나치게 강해져 모듈의 독립성이 낮아지게 되는 단점이 있습니다(컬럼 1-6 참조).

□ 컬럼 1-6 □

값에 의한 호출과 참조에 의한 호출

본문의 문답에서도 설명했듯이 프로그램에 있어서 함수의 인자를 전달하는 방법에는 〈값에 의한 호출〉과 〈참조에 의한 호출〉의 두 가지가 있다.
C 언어에서는 〈값에 의한 호출〉만이 가능하다. 값에 의한 호출이란 실인자로써 받은 식을 계산하여 그 결과로 얻은 값을 넘겨주는 방법이다.
값에 의한 호출의 장점은 다음과 같다.

(1-1) 함수의 독립성이 높다.

(1-2) 함수를 호출하는 데 있어서 인자의 교환 방법이 단순하다.

단점은,

〈1-1〉 인자의 변경이 호출하는 쪽에 반영되지 못한다는 점이다.

C 언어에서는 가능하지 않은 〈참조에 의한 호출〉은 인자의 어드레스를 넘겨주고 형식 인자를 같은 어드레스 상에 저장하는 방법이다. 〈참조에 의한 호출〉은 함수를 호출하는 쪽과 호출되는 쪽이 인자를 공유하는 것이다.
따라서 인자를 통해 함수가 서로 강하게 결합되기 때문에

〈2-1〉 함수의 독립성이 낮다.
〈2-2〉 함수 호출에 있어서 인자의 교환 방법이 복잡하다.

라는 단점이 있다. 예를 들면(이것은 C에 대한 설명이 아니고 가공의 언어에 대한 설명이다),

```
void func(double x);
```

라는 double형의 인자를 1개 넘겨 받는 함수 func가 있다고 하자.

```
func(a * 5.0);
```

이라는 함수 호출에서 a * 5.0은 식이지 변수는 아니다. 따라서〈참조에 의한 호출〉 방식을 사용하려고 해도 실인자의 「어드레스」가 존재하지 않는다.
사용하는 언어에 따라서

(I) 이와같이 함수를 호출하면 컴파일 불능(문법 에러)이 된다.
(II) 이와같이 함수를 호출하면 double형을 갖는 일시적인 변수를 생성해 그곳에 a*5.0을 계산한 값을 저장하고 그 어드레스를 넘겨준다.

위와 같은 두 가지 경우로 처리된다. 또, 참조에 의한 호출은

〈2-3〉 재귀 호출이 불가능하다.

라는 단점이 있다.
예를 들어

```
int factorial(int n)
{
    return((n==0) ? 1: n * factorial(n-1));
}
```

이라고 하는 함수를 생각해 보면 알 수 있다.
이상의 설명으로 이해할 수 있듯이 〈값에 의한 호출〉과 〈참조에 의한 호출〉은 그 방법이 전혀 다른 것이다. C 언어에서는 〈값에 의한 호출〉만이 가능하다. 간혹 현재 출판된 C 언어에 관한 책들 중 일부가 리스트 1-2의 swap 함수처럼 어드레스를 넘겨주고 그것을 포인터로써 받는 것을 〈참조에 의한 호출〉이라고 설명하고 있다.
그러나 이것은 잘못된 것이다. 포인터를 값으로써 넘겨주고 있을 뿐이지 참조에 의한 호출과는 방법이 다르다. 굳이 말하자면 「포인터의 값 전달에 의한 〈참조에 의한 호출〉」이라고나 할까.

▶ 문제 1-2

2개의 int형 변수 x, y를 받아 그 합과 차를 wa 및 sa가 가리키는 int형의 변수에 저장하는 함수

```
void calc(int x, int y, int *wa, int *sa)
```

를 작성하시오.

1-2-3 값에 의한 호출의 이점

― 값에 의한 호출은 어떤 이점이 있는가에 대하여 구체적인 예를 들어 주십시오.

리스트 1-4는 인자로 받은 n의 수 만큼 "string"을 출력하는 함수와 그것을 호출하는 프로그램입니다. print 함수는 처리를 n회 반복하기 위해서 i라는 변수를 for 문에서 사용하고 있습니다.

[리스트 1-4] 문자열을 n회 표시하는 프로그램(버전 1)

```
/*
  List 1-4  문자열을 n회 표시하는 프로그램(버전 1)
*/

#include <stdio.h>

void print(int n)
{
    int    i;

    for (i=0; i<n; i++)
        printf("string");
```

```
    }

    int main(void)
    {
        int    count = 5;

        print(count);
        return(0);
    }
```

— 어떤 의미입니까?

잘 생각해 보십시오. print 함수 중의 n은 실인자의 복사에 지나지 않습니다.

따라서, 그 값을 마음대로 변경해도 지장이 없다는 의미입니다. 리스트 1-5와 같이 루프를 구성하기 위해서 n의 값을 감소하여도 관계가 없는 것입니다.

```
while (n--)
```

는 루프를 제어하는 부분입니다. 이것은

```
while (n-- != 0)
```

이라는 의미로써 n이 0이 아닌 동안 처리를 반복합니다. n이 0일까, 0이 아닐까의 판단을 한 직후에 n을 1만큼 감소하는 것입니다.

[리스트 1-5] 문자열을 n회 표시하는 프로그램(버전 2)

```
/*
    List 1-5  문자열을 n회 표시하는 프로그램(버전 2)
*/

#include <stdio.h>

void print(int n)
{

    while (n--)
        printf("string");
}

int main(void)
{
    int    count = 5;

    print(count);
    return(0);
}
```


— 초 읽기 할 때처럼 5, 4, 3, 2, 1이 되는군요. i라고 하는 추가의 변수가 필요없네요. 참조에 의한 호출로 인자를 주고 받는 FORTRAN 등의 언어에서는 이와같이 할 수 없겠군요.

추가의 변수가 필요하지 않기 때문에 프로그램이 간단하고 효율도 높게 됩니다. print 함수를 빠져 나올 때 n의 값은 -1이 되겠죠.

— 그렇지만 호출하는 쪽의 카운트와 관계가 없다는 것이 좋은 것입니까?

그렇습니다. print 함수 쪽에서 말하면 n의 값을 마음대로 바꿔 사용하여도 좋기 때문에 효율이 높은 프로그램을 작성할 수가 있습니다.

반대로 호출하는 main 함수의 입장에서 말하면 count를 넘겨줘도 값이 변하지 않기 때문에 안심하고 호출할 수 있습니다.

— 정말로 그렇군요. sin(x)에서 sin 함수를 호출한 후에 x의 값이 마음대로 변한다면 어처구니 없는 일이 되어 버리겠군요.

1-2-4 포인터와 scanf 함수

그런데 이야기가 포인터로부터 벗어나 있군요. 이야기를 포인터로 돌리겠습니다.

C 언어에서 함수로부터 1개의 값을 돌려줄 때에는

```
return 식;
```

이라고 하면 됩니다. swap 함수처럼 2개 이상의 값을 돌려주고 싶을 때에는 여기에서 설명한 것처럼 반드시 포인터를 사용해야 합니다.

— 포인터의 사용 예에서 포인터를 함수의 인자로써 사용하는 것을 가장 중요하게 다루는 이유를 알겠군요.

그렇습니다. 이와같은 사용법에 대해서는 꼭 이해해 두셔야 합니다. 초보자가 제일 처음 만나는 것은 아마도 scanf 함수라고 생각됩니다.

— 아 생각났습니다. printf 함수에 표시하는 경우에는 &가 필요하지 않지만, scanf 함수일 때는 필요하기 때문에 의미를 잘 이해하지 못했습니다.

그렇습니다. printf 함수나 scanf 함수는 리스트 1-6에서와 같이 사용합니다.
말씀하신 대로 printf 함수에서는 &가 필요하지 않지만 scanf 함수에서는 &가 필요합니다.

[리스트 1-6]

```
/*
  List 1-6  printf와 scanf의 예
*/

#include <stdio.h>

int main(void)
{
    int   i;
    long  k;

    printf("input i : ");
    scanf("%d", &i);
    printf("input k : ");
    scanf("%d", &k);

    printf("i = %d\n", i);
    printf("k = %d\n", k);
    return(0);
}
```

— C 언어에서는 값에 의한 호출만을 사용하므로「인자의 값을 변경할 때에는 포인터형을 사용하여 주고 받음을 해야하기」때문에 &을 붙이지 않으면 안된다는 것이죠. 이제 겨우 의문이 풀렸습니다.

이제 포인터의 기본에 대해서는 이쯤에서 설명을 마치겠습니다.

— 그러면 scanf 함수에서 문자열 즉, %s를 지정할 때만은 &가 필요하지 않다고 들었습니다만 어떻게 된 것입니까?

이제 곧 알게 될 것입니다.

■ 컬럼 1-7 ■

main 함수를 작성하는 방법

main 함수가 돌려주는 형은 void일까? 그렇지 않으면 int가 될까? 결론부터 말하자면 main 함수에서 돌려주는 형은 int이다. 따라서

```
int main(void)
```

혹은

```
int main(int argc, char *argv[])
```

로 기술하는 것이 바른 형태이다.
이 책의 프로그램은 모두 이 형태로 작성되어 있다.

제 2 장

포인터와 배열

▷ 2-1 포인터와 배열
▷ 2-2 문자열과 포인터·배열
▷ 2-3 포인터의 연산

전장에서는 포인터의 기본에 대하여 설명하였습니다. 정확히 이해할 수 있었습니까? C 언어에서 포인터와 배열은 밀접한 관계가 있습니다. 포인터와 배열은 거의 같은 동작을 합니다. 포인터와 배열에 대해서 그 공통점·차이점을 정확히 알아두지 않으면 포인터와 배열을 혼동하여 이상한 프로그램을 작성하게 됩니다. 이들의 공통점·차이점을 정확히 이해한다면 포인터에 대한 이해를 보다 깊게 할 수 있습니다.

▷ 2-1 포인터와 배열

▷…… 2-1-1 도입

C 언어에서 포인터와 배열은 밀접한 관계가 있습니다. 포인터와 배열에 대한 공통점, 차이점 등을 공부하겠습니다.

– 여러가지 프로그램을 보면 인자로 배열을 받을 때 아래와 같이 3가지 방법을 사용하여 작성하고 있습니다. 이들의 차이점은 무엇입니까?

void func(int a[10])
{
 	
}
 	
(1) 배열(크기를 표시)
	void func(int a[])
	{
	
	}
	
	(2) 배열(크기가 없음)
		void func(int *a)
		{
		
		}
		
		(3) 포인터

실은 (1)~(3)은 같은 것입니다. 컴파일될 때 어느 것이든 a는 int의 포인터로 해석됩니다. 즉, (3)으로 해석되는 것입니다.

– 위의 말씀은 (1)처럼 10개이다 라고 크기를 지정해도 의미가 없다는 것입니까.

맞습니다. 10이라고 하는 숫자는 무시됩니다. 따라서, (1)과 같이 크기를 나타내도 그것은 안심시키기 위한 것에 지나지 않습니다. 다만 프로그램을 읽기 쉽게 하고 프로그램을 읽는 사람에게 크기가 10이라는 것을 알린다는 의미는 중요한 일입니다.

– 머리가 혼란해지는데요. 포인터를 조금 알게 되어 좋아했었는데...

앞에서도 말씀드렸듯이 C 언어에서 포인터와 배열은 밀접한 관계가 있습니다. 이것을 정확히 이해하면 왜 C 언어에서는 배열의 첨자가 0에서만 시작하는 것일까—다른 언어에서는 첨자의 하한·상한을 지정할 수 있는 것이 있습니다—등을 이해하게 됩니다.

☆ 체크 포인트 ☆

C 언어에서 배열의 첨자는 반드시 0에서부터 시작된다.
그리고 여기에는 이유가 있다.

조금 복잡한 이야기가 되겠습니다만 (3)을 잘 봐 주십시오.

```
void func(int *a)
```

로 되어 있죠? 여기에서 a는 int의 포인터형을 갖고 있습니다. 리스트 1-2의 swap 함수에서는

```
void swap(int *x, int *y)
```

로 되어 있습니다. 이 int *x나 int *y와 어떻게 틀립니까? 실은 이것은 같은 것입니다.

— 네? 그렇군요. 1개의 변수를 포인터로 받을 때와 배열로 받을 때와의 선언은 같은 것이지만...

결론부터 말하자면 C 언어에서는 배열을 인자로 받을 수 없습니다.

☆ 체크 포인트 ☆

C 언어에서는 함수의 인자로써 배열을 받을 수가 없다.

— 하지만 아래의 리스트처럼 작성한다면 a는 배열이 아닙니까?

```
void func(int a[10])
{
    /* .... */
    a[3] = 0;
    /* .... */
}
```

배열이 아닙니다. 앞에서 말했던 것과 같이 a는 int의 포인터입니다.

— 그래도 이해가 잘 되지 않는데요.

함수의 인자에 관한 것은 뒤에서 설명하기로 하고 포인터와 배열에 대한 기본부터 공부하기로 하겠습니다.

▷ ····· 2-1-2 배열과 포인터

그림 2-1에 대해서 생각해 봅시다.

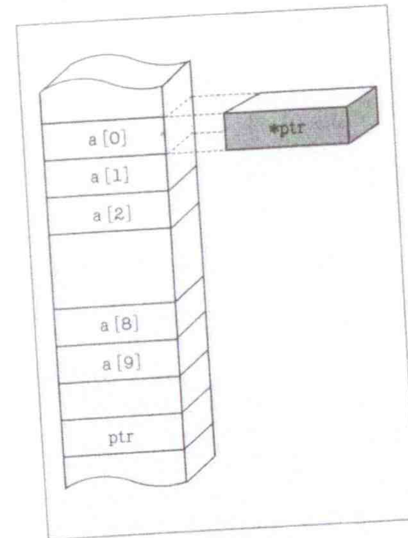
```
int a[10];
int *ptr;

ptr = &a[0];
```

를 수행한다면 어떻게 될까요?

— ptr에 a[0]의 어드레스를 대입하기 때문에 ptr은 a[0]을 가리키는 것이 되고 *ptr이 a[0]의 앨리어스가 됩니다.

그렇습니다.

일반적으로 포인터 변수 ptr에 대해서 ptr+i는 ptr이 가리키고 있는 요소의 i개 뒤의 요소를 가리키고 ptr-i는 ptr이 가리키고 있는 요소의 i개 앞의 요소를 가리킵니다.



[그림 2-1] 배열과 포인터

☆ 중 요 ☆

> 포인터 변수 ptr이 어떤 변수 x를 가리키고 있을 때
>
> ptr+i는 x의 i개 뒤의 요소를 가리킨다.
> ptr-i는 x의 i개 앞의 요소를 가리킨다.

– 이 경우 ptr+1은 a[1]을, ptr+2는 a[2]를 가리키고 있는 것이 됩니까?

그렇습니다. ptr+i에 간접 연산자 *를 적용하면 어떻게 되겠습니까? 예를 들어 ptr+1은 a[1]을 가리키고 있습니다. *는 그 포인터 변수가 가리키는 변수의 실제 내용을 나타내는 연산자이기 때문에 *(ptr+1)은 a[1]의 앨리어스가 됩니다. 따라서,

```
a[0]과 *ptr은 같은 내용을 나타낸다.
a[1]과 *(ptr+1)은 같은 내용을 나타낸다.
...
a[9]와 *(ptr+9)는 같은 내용을 나타낸다.
```

와 같이 됩니다. 즉, ptr+i는 a[i]의 어드레스이고 *(ptr+i)는 a[i]의 앨리어스가 됩니다. 실제로 C 언어에서는 아래 표와 같은 규칙이 있어 다른 언어에서는 생각할 수 없는 특이한 표기법을 사용합니다. 꼭 암기해 주십시오.

배열 a[n]에 대하여, a[i]를 *(a+i)라고 쓸 수 있다. &a[i]를 a+i라고 쓸 수 있다.
포인터 p에 대하여, *(p+i)를 p[i]라고 쓸 수 있다.

이는 a[2]는 *(a+2), *(ptr+2)는 ptr[2]로 기술할 수 있다는 의미입니다. 정리하면 다음의 표와 같으며 각 행에 나열된 4가지는 모두 같은 내용을 나타냅니다.

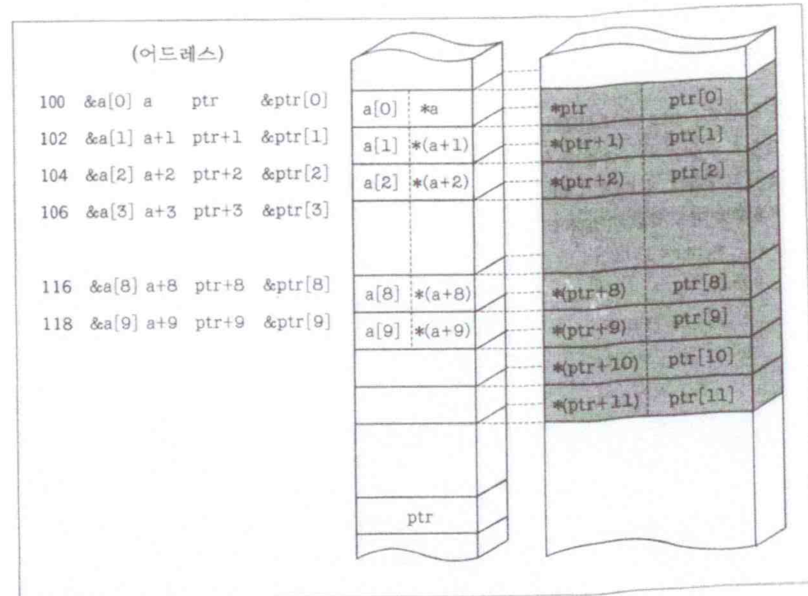
a[0]	*a	*ptr	ptr[0]
a[1]	*a	*(ptr+1)	ptr[1]
....			
a[9]	*a	*(ptr+9)	ptr[9]

– 무언지 조금 이상한데요.

또 a[0]에서부터 a[9]의 어드레스에 대해서도 4가지 종류로 기술할 수 있습니다.

&a[0]	a	ptr	&ptr[0]
&a[1]	a+1	ptr+1	&ptr[1]
....			
&a[9]	a+9	ptr+9	&ptr[9]

그림으로 나타내면 그림 2-2와 같습니다.



[그림 2-2] 배열과 포인터

– ptr[]이 10 이상 있는 것은 어떻게 된 것입니까?

ptr은 현재 a를 가리키고 있습니다. ptr+i는 그것으로부터 i개 뒤의 요소를 가리키

고 있을 뿐이고 어디까지라고 하는 상한이 있다는 의미가 아니기 때문에 우선 이와같이 썼습니다.

— 그러면 a[]은 배열이기 때문에 9까지 밖에 없는 것입니까?

메모리상에는 10개 밖에 확보되지 않기 때문에 그렇게 말할 수 있으나 프로그램 중에 배열명이 표시된 경우 그것은 배열의 선두 요소를 가리키는 포인터라고 해석되며 배열의 최종 요소의 다음 요소까지는 올바르게 액세스하도록 되어 있습니다.

— a[10]을 사용할 수 있다는 말씀입니까?

그런 의미가 아니고, &a[10]이라고 하는 식을 사용할 수 있다는 것입니다. 다만 &a[11] 이상의 액세스에 대해서 어떻게 동작하는지는 정의되어 있지 않습니다.

그림에서는 ptr의 상한이 없는 것처럼 쓰여 있지만 사실은 ptr이 바르게 가리킬 수 있는 범위도 ptr[0]에서 ptr[10]까지입니다.

— 이해가 잘 안되는데요. 왜 그렇습니까?

배열 a가 만약 데이터 영역의 맨 뒤쪽에 위치한 경우를 생각해 보면 ptr+50 등은 올바르지 않은 영역을 가리키는 것이 되어 버리며 따라서 포인터가 배열을 가리키고 있을 때 바르게 가리킬 수 있는 영역은 배열의 처음부터 제일 끝의 다음 요소까지가 됩니다.

컬럼 2-1

배열의 첨자 허용 범위

본문의 문답에서 설명하고 있는 것처럼 배열의 첨자는 배열의 최후 요소의 다음 요소까지를 액세스할 수 있다. 예를 들면

```
int a[10];
```

이라고 선언할 때에는 a[0]에서부터 a[9]까지의 10개의 영역을 갖게 된다. 즉, a[10]까지의 11개의 영역이 확보된다는 의미는 아니다. 이것은 배열의 제일 끝 요소 다음 요소의 포인터를 파수병(sentinel)으로 사용하는 프로그램 등에서 이전부터 사용했기 때문에(편리해서 자주 사용되어졌기 때문) 도입된 것이다. &a[10]은 &a[9]가 가리키는 다음 요소를 올바르게 가리킨다.
그러나 &a[11], &a[12]가 올바르게 다음 요소를 가리킨다고는 보장할 수 없기 때문에 주의하기 바란다.

— 이렇게 생각하면 포인터와 배열은 아주 똑같은 것이 아닙니까?

아닙니다. 배열과 포인터는 중요한 점이 틀립니다. 당연한 이야기이지만 변수를 선언하는 방법이 다릅니다. 이것에 관해서는 설명할 필요가 없겠죠?

— 네, 예를 들어 배열이라면

```
int a[10];
```

포인터는

```
int *ptr;
```

이라고 선언하기 때문에 전혀 다릅니다.

또, ptr과 a에는 본질적인 차이가 있습니다. 앞에서 설명했듯이 ptr은 int의 포인터형을 가지고 있습니다. 따라서 ptr은 가리키고 있는 변수—여기에서는 a[0]입니다—의 어드레스를 가집니다. 그러나 a는 어떻습니까? 정의에 의해서 a는 &a[0]이고 배열의 선두의 어드레스를 가집니다. 그러나 a는 변수가 아닌 상수입니다. 생각해 보면 알 수 있지만 a는 고정된 영역—즉, a[0], a[1],…,a[9]의 시작 어드레스이고 a의 값을 바꾸어 사용할 수 없습니다. 그림 2-2를 보아도 알 수 있듯이 ptr이라고 하는 영역은 메모리 상에 있지만 a라고 하는 영역은 메모리 상에 없습니다.

☆ 중 요 ☆

배열 int a[10];

배열은 a[0]~[9]의 고정된 영역을 갖는다. a는 처음 요소 a[0]의 포인터(&a[0])가 된다.

포인터 int *p;

int의 포인터 p는 임의의 int형 변수를 가리킬 수 있다. p가 가리키는 앞뒤 요소를 [] 연산자를 사용하여 p[i]와 같은 방법으로 액세스할 수 있다.

결론으로 말한다면 「포인터와 배열은 전혀 다른 것이지만 요소를 액세스하는 방법이 아주 비슷하다」고 할 수 있겠죠.

— 포인터도 배열도 a나 ptr을 기준으로 거기에서부터의 차를 첨자를 가지고 액세스하기 위하여 배열의 첨자는 0에서부터 시작할 수 밖에 없다는 뜻입니까?

그렇습니다. 다른 언어와 같이 첨자가 1에서부터 시작한다든가, 상한, 하한을 지정한다는 것은 C 언어에서는 없습니다.

왜냐하면 C 언어에서 첨자는 기준이 되는 포인터로부터의 차—즉, 얼마나 떨어져 있나—를 나타내는 수치이기 때문입니다.

▷…… 2-1-3 함수간 배열의 주고 받음, 첫번째

다시 함수의 인자 이야기로 돌아가겠습니다.

— 함수 인자로써의 배열 말씀이군요?

리스트 2-1의 프로그램을 생각해 보십시오. 그리고 나서 그림 2-3을 봐 주십시오.

main 함수는

```
func(x);
```

와 같이 실인자로써 x를 넘겨주고 있습니다. x는 &x[0]이기 때문에 배열의 선두 요소의 포인터 즉, 그 어드레스를 넘겨줍니다.

— 이 경우에는 200을 넘겨주겠군요?

그렇습니다. 함수의 머리 부분

```
void func(int *a)
```

의 *a는 무엇이겠습니까? a는 int의 포인터형을 갖는데 받은 값 즉, x[0]의 어드레스가 복사됩니다. 따라서 a는 x[0]을 가리킵니다.

— *a는 x[0]의 앨리어스가 됩니까?

그렇습니다. a[3]에 3을 대입했지만 a[3]은 *(a+3)입니다. 따라서 이것은 a가 가리키고 있는 3개 뒤의 요소 x[3]이 됩니다.

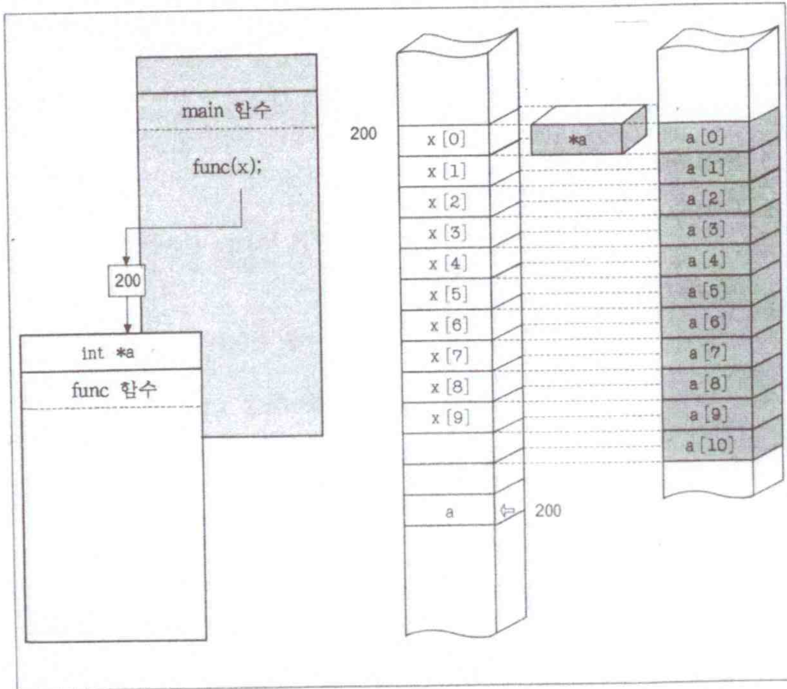
[리스트 2-1] 배열을 주고 받는 프로그램

```
/*
    List 2-1  배열을 주고 받는 프로그램
*/

void func(int *a)
{
    a[3] = 3;
}

int main(void)
{
    int   x[10];

    func(x);            /* func(&x[0])과 같음 */
    return(0);
}
```

— 이제 func 함수의 a가 배열이 아닌 포인터라고 하는 것을 알겠습니다.

변수의 어드레스라고 하는 <값>을 넘겨준다는 점에서 리스트 1-2의 swap 함수와 같은 것입니다. 호출된 함수는 포인터 변수로써 받은 것이지만 이 경우는 포인터 변수에 []의 첨자를 붙여서 배열처럼 액세스하고 있는 것에 지나지 않습니다.

> ☆ 중 요 ☆
>
> · 배열은 선두 요소의 포인터라고 하는 형으로 전달된다.
> · 포인터로 받은 인자 x는 리스트 1-2의 swap 함수처럼 * 연산자를 적용해 1개의 변수를 액세스하는 것으로 취급된다.
> · 리스트 2-1의 func 함수처럼 [] 연산자를 적용해 배열과 같이 취급한다.
> 위의 어느 방법이라도 사용 가능하며 x가 포인터라는 것은 의심할 여지가 없다.



[그림 2-3] 값에 의한 배열의 전달

그러면 리스트 2-2의 프로그램을 수행해 보십시오.

[리스트 2-2] 배열과 포인터의 차이

```
/*
    List 2-2  배열과 포인터의 차이
*/

#include <stdio.h>

void func(int *a)
{
    printf("SizeOf(a) = %d\n", sizeof(a));
}

int main(void)
```



```
{
    int   x[10];

    printf("SizeOf(x) = %d\n", sizeof(x));
    func(x);
    return(0);
}
```

– 네, 했습니다.

■ 수행 결과 ■

```
SizeOf(x) = 20
SizeOf(a) = 2
```

– 그런데 sizeof는 무엇입니까?

sizeof는 +나 *와 같은 연산자의 하나입니다. 다음과 같이

    sizeof 식

이라고 하면 그 식(객체)의 크기를 바이트 수로 나타내어 되돌려 줍니다. 덧붙여 말하면 문법상에서는 필요하지 않기 때문에

    sizeof x

라고 해도 관계 없지만 앞뒤의 식에 따라서 혼동하기 쉽기 때문에 반드시

    sizeof(x)

와 같이 ()를 붙여 주십시오.

프로그램을 수행한 환경에서는 int형이 2바이트, 포인터형도 2바이트입니다.

따라서 x의 크기는 2×10 즉, 20바이트가 되지만 a는 단순히 포인터이기 때문에 2바이트입니다.

■ 컬럼 2-2 ■

sizeof 식

아래의 형식

sizeof 식

은 그 식의 크기를 바이트 단위로 되돌려 준다.
int형이 2바이트인 컴파일러에서

```
int x;
printf("%d\n", sizeof(x));
```

를 수행하면 2라고 출력된다. 다음 문장

```
printf("%d\n", sizeof(1+2.0));
```

을 수행하면 어떤 결과가 될까?
만약 double형이 8바이트라면 8이라고 출력된다. 1은 int형의 상수 2.0은 double형의 부동 소수점 상수이다. int형과 double형의 연산에서 int형의 피연산자가 double형으로 승격되어 다음과 같이 double형 끼리의 연산이라고 간주된다.

```
(double)1 + 2.0
```

또, 이 식(연산 결과)은 double형을 갖는다. 따라서, double형의 바이트 수인 8이 표시된다. 다음의 프로그램

```
#include <stdio.h>

int main(void)
{
    int a;

    printf("sizeof a+1.0 : %d\n", sizeof a + 1.0 );
    printf("sizeof(a+1.0): %d\n", sizeof(a + 1.0));
    return(0);
}
```

에서 ()의 필요성을 알 수 있다고 생각한다. 직접 수행해 보고 의미를 생각해 보자. sizeof에는 sizeof(수형의 이름)이라고 하는 것도 있다.
이에 대해서는 제4장에서 설명하겠다.

■ 컬럼 2-3 ■

배열의 이름

본문의 문답에서도 설명하고 있는 것처럼 프로그램 중에 배열의 이름이 나타나면 그 배열의 선두 요소의 포인터라고 해석된다.

```
int a[10];
```

라는 선언에서 a는 a[0]의 포인터가 되고 값으로는 &a[0]을 가진다. a가 포인터가 되어 버리기 때문에 *(a+1)과 같은 표현이 가능하게 된다. 이와같이 배열의 이름은 그 선두 요소의 포인터라고 해석되지만 예외가 있다.

(1) 어드레스 연산자 &의 피연산자가 될 때

컬럼 1-2 「배열명에 &를 붙이면」에서도 설명했듯이 a는 배열이지만 선두 요소의 포인터는 되지 않는다.

(2) sizeof 연산자의 피연산자가 될 때

여기에 대해서는 본문에서 설명하였으므로 분명하게 알 수 있다고 생각한다.

— 이 예로부터도 a는 배열이 아니고 포인터인 것을 알 수 있군요.

결국, C 언어에서는 배열을 전달할 수 없다는 것을 잘 알 수 있습니다.
그런데 3[a]라고 하는 것을 알겠습니까?

— 3[a]라고요? 그것은…

(ptr+i)를 잘 봐 주십시오. i와 ptr을 더하고 있습니다. 그럼 이것은 *(i+ptr)이라고 써도 좋다는 뜻입니다. 그리고 이것은 i[ptr]이라고 쓸 수 있습니다. 일반적으로 E1[E2]는 E2[E1]이라고 쓸 수 있습니다(컬럼 2-4 참조).

— 네?

□ 컬럼 2-4 □

a[3]과 3[a]

다른 언어에서는 배열의 첨자를 나타내는 []나 () 등의 기호는 범위를 나타내는 문자이다. 그러나 C 언어에서 []은 연산자이다. [] 연산자는 2개의 피연산자(operand)를 갖는다. 하나는 「어떤 수형 T의 포인터」이고 다른 하나는 정수형이다. 그리고 이 피연산자의 순서는 관계없다. a+b와 b+a가 같듯이 a[3]과 3[a]는 같은 의미이다.
물론 3[a]라는 표현은 혼동하기 쉽기 때문에 이와같은 표기는 피하는 편이 좋다.

포인터 ptr이 배열 a의 선두 요소 a[0]을 가리키고 있을 때 다음 4가지

```
a[i],    *(a+i),    ptr[i],    *(ptr+i)
```

가 같은 것을 나타낸다고 했지만 실은 다음 8가지

```
a[i],     i[a],      *(a+i)     *(i+a)
ptr[i],   i[ptr],    *(ptr+i)   *(i+ptr)
```

가 같은 것을 나타내고 있다는 의미입니다.

— 아마 친구가 이 프로그램을 보면 깜짝 놀라겠죠!

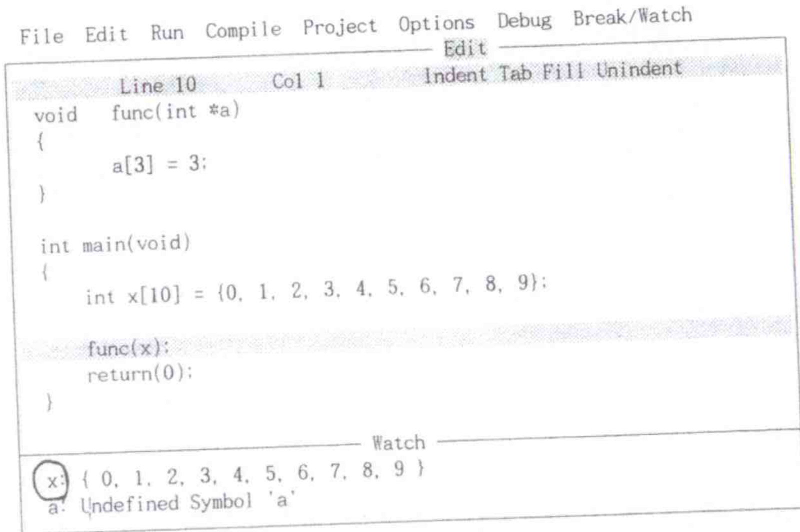
[리스트 2-3] 모두가 놀랄만한 프로그램

```
/*
    List 2-3  놀랄만한 프로그램
*/

#include <stdio.h>

int main(void)
{
    int    i, a[4];

    0[a] = a[1] = *(a+2) = *(3+a) = 0;
    for (i = 0; i < 4; i++)
        printf("%2d", i[a]);
    return(0);
}
```

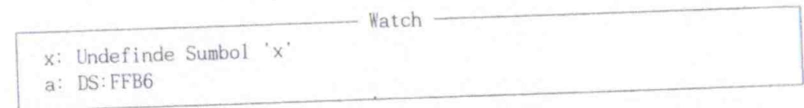

□ 컬럼 2-5 □

Turbo C에서 포인터를 배열로써 보는(watch) 방법

Turbo C(Ver. 2.0)나 Turbo C++의 통합 개발 환경에서는 프로그램 수행시에 변수의 값을 볼(watch) 수 있다.



위의 예는 main 함수를 수행중인 경우이다. 배열 x의 각 요소 값이 Watch 윈도우에 나타나 있다. a의 스코프(a라고 하는 식별자가 효과가 있는 범위)가 아니기 때문에 a는 선언하지 않은 심벌이 된다.
그러면 1행을 수행하여 func 함수 속으로 들어가 보자.



현재 func 함수 안에서 수행중이기 때문에 x의 스코프가 아니고 x는 선언하지 않은 심벌이 된다.
a는 포인터이기 때문에 세그먼트:옵셋의 형태로 포인터의 값이 표시된다.
(세그먼트, 옵셋에 관해서는 제7장에서 설명하고 있다).
여기에서 표시된 값 FFB6은 수행중의 메모리 상태 등의 환경에 따라 변하게된다. 여기에서도 a가 배열이 아니고 포인터라는 것을 쉽게 알 수 있다.
그러나 곤란한 점은 아무리 a가 포인터라고 해도 배열과 같이 사용하는 경우도 있다는 것이다. a[0]~a[9]의 값을 표시하고 싶을 경우 어떻게 하면

좋을까?
프로그램을 변경하여 func 함수를 선언하는 부분을

```
void func(int a[])
```

라고 변경해도 결과는 같다. 결국 a는 포인터인 것이다.
Watch 윈도우에 「*a」라고 하는 것을 추가할 경우를 보자

```
Watch
x: Undefined Symbol 'x'
a: DS:FFB6
*a: 0
```

이렇게 하면 선두 요소의 값(*a 즉 a[0])을 표시할 수는 있지만 배열 전체를 표시하지는 못한다. 「*a,10」이라는 것을 추가할 경우를 살펴보자.

```
Watch
x: Undefined Symbol 'x'
a: DS:FFB6
*a: 0
*a, 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

이제는 a[0]~a[9]의 값을 표시할 수 있다. a는 포인터이지만 위와같이 함에 따라 a가 가리키고 있는 곳으로부터 10개의 내용을 표시할 수 있고 배열처럼 그 값을 볼 수가 있다.

2-1-4 함수간 배열의 주고 받음, 두 번째 – 값에 의한 호출의 실현

– C 언어에서는 배열의 주고 받음을 수행할 수 없다는 뜻입니까?

그렇습니다. 선두 요소의 포인터라고 하는 형으로써 교환합니다.

– 「C 언어에서의 값에 의한 호출은 장점이다」라는 이야기를 들었습니다.

그렇습니다.

– 함수 중에서 형식 인자의 값을 변경해도 호출하는 쪽의 실인자와는 아무런 관련이

없다는 것이 장점이었습니다. 그러면 배열의 경우는 어떻게 되는 것입니까?

예리하게 지적해 주셨습니다. 지금까지 보아 왔듯이 배열은 <그 선두의 포인터>라는 형으로써 주고 받음을 수행합니다. 이것은 값에 의한 호출의 「값」으로써 포인터의 교환을 한다는 의미입니다. 그러나 이것은 사실상 참조에 의한 호출과 같은 것입니다. 이것은 리스트 2-4를 보아도 알 수 있습니다. main 함수의 x는 배열이고 func 함수의 a는 포인터라는 차이는 있지만 a[i]와 x[i]는 같은 것을 가리키기 때문에 func 함수로 a[0] 등의 요소를 변경하면 main 함수의 x[0]도 바뀌게 됩니다.

[리스트 2-4] 배열을 함수에 넘겨주기

```
/*
    List 2-4  배열을 함수에 넘겨주기
*/

#include <stdio.h>

void func(int *a)
{
    int   i;

    a[0] = 10;
    a[1] = 8;
    a[2] = 12;
    printf("\nfunc function \n");
    for (i = 0; i < 10; i++)
        printf("%4d", a[i]);
}

int main(void)
{
    int   i;
    int   x[10];

    for ( i=0; i<10; i++ )
        x[i] = i;
    printf("\nbefore the call of func function\n");
    for ( i=0; i<10; i++ )
        printf("%4d", x[i]);
    func(x);
    printf("\nafter the call of func function\n");
    for ( i=0; i<10; i++ )
        printf("%4d", x[i]);
    return(0);
}
```

■ 수행 결과 ■

```
before the call of func function
     0   1   2   3   4   5   6   7   8   9
func function
    10   8  12   3   4   5   6   7   8   9
after the call of func function
    10   8  12   3   4   5   6   7   8   9
```

— 배열을 그 자체로 넘겨줄 수는 없습니까?

할 수 없습니다. Pascal과 같은 언어라면 가능하겠죠.

— 정말로 할 수 없는 것입니까? 그래도 다른 방법이 있을 것만 같은데요.

정말 집요하게 질문하는군요. 실은 평범한 테크닉을 이용하여 배열을 그 자체로 주고 받을 수 있습니다.

리스트 2-5를 봐 주십시오. array형이라고 하는 구조체를 선언하고 있습니다. 구조체의 멤버로 int형의 크기 10인 배열을 갖고 있습니다. 이와 같이 배열을 전부 구조체에 넣어버림으로써 배열을 값으로 넘길 수 있습니다.

[리스트 2-5] 배열을 함수에 그대로 넘겨줌

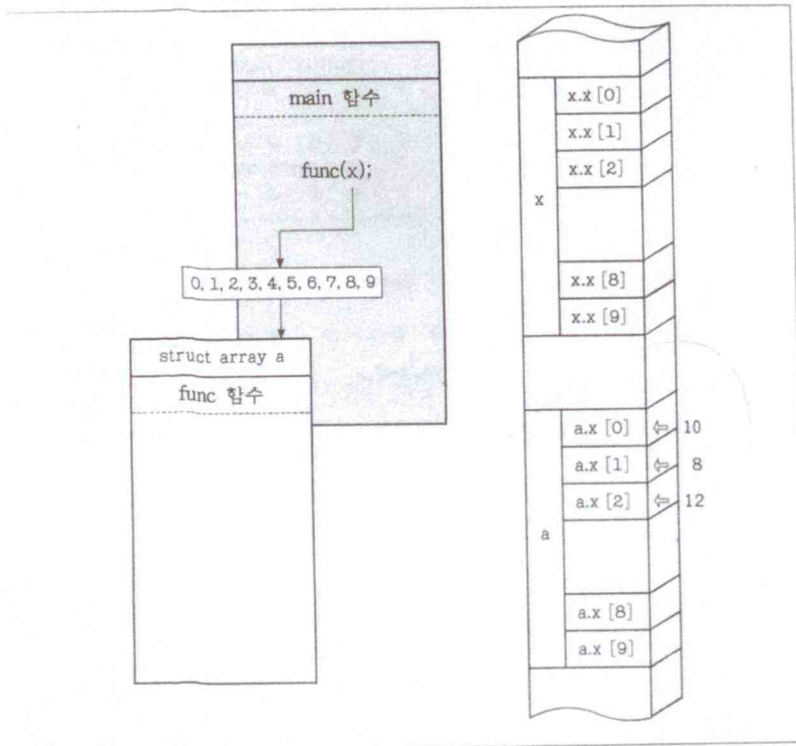
```
/*
   List 2-5 배열을 함수에 그대로 넘겨주기
*/

#include <stdio.h>

struct array {
    int   x[10];
};

void func(struct array a)
{
    int   i;

    a.x[0] = 10;
    a.x[1] = 8;
    a.x[2] = 12;
    printf("\nfunc function \n");
    for (i = 0; i < 10; i++)
        printf("%4d", a.x[i]);
}

int main(void)
{
    int   i;
    struct array x;

    for ( i=0; i<10; i++ )
        x.x[i] = i;
    printf("\nbefore the call of func function\n");
    for ( i=0; i<10; i++ )
        printf("%4d", x.x[i]);
    func(x);
    printf("\nafter the call of func function\n");
    for ( i=0; i<10; i++ )
        printf("%4d", x.x[i]);
    return(0);
}
```


— 값에 의한 호출이기 때문에 main 함수의 구조체 x와 func 함수쪽의 a는 관계가 없다는 뜻입니까?

그렇습니다. a는 x의 복사본이기 때문에 이 경우 func 함수에서 요소의 내용을 변경하여도 main 함수의 x는 전혀 영향을 받지 않습니다.

■ 수행 결과 ■

```
before the call of func function
     0   1   2   3   4   5   6   7   8   9
func function
    10   8  12   3   4   5   6   7   8   9
after the call of func function
     0   1   2   3   4   5   6   7   8   9
```



[그림 2-4] 배열을 구조체를 이용하여 그대로 전달

□ 컬럼 2-6 □

구조체의 값 전달

K&R에서는 구조체·공용체에 대하여 허용하는 것은

(1) 연산자를 사용해 멤버를 액세스한다.
(2) & 연산자를 사용해 어드레스를 얻는다.

이 두 가지 뿐이다.

구조체·공용체를 한 번에 대입하거나 인자로 넘겨주거나 하는 것은 K&R에서는 사용할 수 없고 이후에 사용 가능하게 될 것이라고 하였다.

그러나 ANSI에서는 이런 것들이 가능하다.

ANSI 표준이 아닌 컴파일러에서는 리스트 2-5의 프로그램이 동작하지 않

을 수도 있다.

구조체의 대입도 가능하기 때문에 리스트 2-5의 struct array의 수형이 주어졌을 때

```
sturct array a, b;
/* ... */
a = b;
```

라고 하는 것이 가능하다. 배열을 1개의 대입식을 사용하여 대입할 수 있는 것이다.

▷ 2-2 문자열과 포인터·배열

▷ ····· 2-2-1 문자 배열과 문자의 포인터

— 문자열과 포인터, 배열간의 관계를 잘 모르겠습니다.

문자열이란 문자의 배열입니다. 문자열이기 때문에 배열과 다르지 않습니다. 기본적인 이야기입니다만.

— 예를 들어 주십시오

```
printf("A=%2d", a);
```

위에서 ""로 둘러싸여 있는 부분이 문자열입니다.

이와같이 쌍따옴표로 둘러싸여 있는 문자열을 문자열 상수(string constant) 혹은 문자열 리터럴(string literal)이라고 합니다.

"A=%2d"라는 식의 값은 무엇이겠습니까?

— 네? 값이 있습니까?

일반적으로 "xxxx"라는 식은 char의 포인터형을 갖고 그 문자열의 선두 어드레스를 값으로 가집니다.

이야기가 바뀌었지만, 배열의 초기화 방법을 알고 있습니까?

— 물론 알고 있습니다. 초보자라고 해도 그 정도는 알고 있어요. 다음과 같습니다.

```
int x[] = {1, 2, 3, 4};
```

맞습니다. 문자 배열의 경우도 마찬가지로 다음과 같이 됩니다.

```
char c[] = {'a', 'b', 'c'};
```

배열의 초기화에서는 그 크기가 자동으로 계산되기 때문에 x의 크기는 4, c의 크기는 3이 됩니다.

C 언어에서는 문자열의 마지막에 널 문자('\0')를 붙여 처리합니다. 이것은 함수—물론 표준 함수 뿐만 아니라 직접 작성한 함수도 포함해서—가 문자열의 마지막을 찾

을 수 있도록 하기 위해서 입니다.

따라서 자신이 처리할 때는 직접 널 문자를 붙여야만 합니다. 만약 단순히 문자 3개의 배열로 c를 사용하는 것이라면 위와 같이 선언을 하면 되지만 문자열로 취급하고 싶은 경우에는 아래와 같이 널 문자를 명시적으로 추가해야 합니다.

```
char str1[] = {'a', 'b', 'c', '\0'};
```

— 굉장히 까다로운데요.

따라서 다음과 같이 쓸 수 있습니다.

```
char str2[] = "abc";
```

str1과 str2의 선언은 기능적으로 같습니다. str2의 선언이 str1의 선언보다 간편한 방법이 되겠지요(컬럼 2-7 참조).

— 아래와 같은 선언을 자주 보는데 이것은 다른 것입니까?

```
char *str3 = "def";
```

역시 아직 초보 단계를 벗어나지 못한 듯 하군요.

str2는 앞에서도 말했듯이 배열의 초기화에 지나지 않습니다. 결국 str2라고 하는 4문자 크기의 배열을 선언하고 그 요소에 선두로부터 'a', 'b', 'c', '\0'을 대입하는 것입니다.

한편 str3은 str3이라고 하는 포인터 변수를 한 개 선언하고 그 초기값으로 "def"를 대입하는 것입니다. 앞에서도 말했듯이 "xxxx"는 그 선두의 어드레스를 값으로 가집니다. 따라서 str3은 "def"라고 하는 문자열이 저장되어 있는 영역의 선두 문자를 가리키게 됩니다.

— 그림 2-5와 같이 된다는 의미입니까?

그렇습니다. 이 그림을 보면 차이를 분명히 알 수 있습니다.

str2는 배열입니다. 이 경우 100번지에서부터 str[0]~str[3]이 저장되어 있으며 각 칸에 'a', 'b', 'c', '\0'이 대입됩니다.

□ 컬럼 2-7 □

문자 배열의 초기화

문자열 리터럴 "xxxx"는 값으로 그 문자열의 선두 어드레스를 갖는다. 그러나 아래와 같이 문자 배열을 초기화 하는 경우는 예외이다.

```
char str2[] = "abc";
```

문자열의 선두 어드레스를 초기값으로 대입한다는 의미가 아니라 어디까지나 {'a', 'b', 'c', '\0'}의 다른 표현 방식으로 생각해야 한다.
배열이나 구조체 등을 **집합체**(aggregate)라고 한다. 집합체의 선언과 동시에 초기값을 설정할 때에는

```
int x[] = {1, 2, 3};
```

과 같이 {}가 필요하지만 문자 배열의 초기값으로 문자열 리터럴을 지정하는 경우에 한해서는 필요하지 않다. 따라서

```
char string[] = {"abc"};
```

에서 처럼 {}를 쓸 필요는 없다(단, 써도 틀린 것은 아니다).

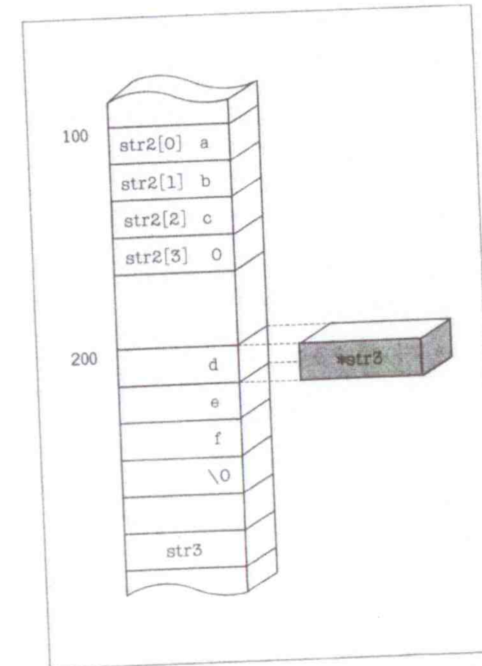
str3은 포인터 변수입니다. 이 경우 "def"라고 하는 문자열은 200번지에 저장되어 있습니다. str3의 초기값으로 이 200이라고 하는 값이 대입됩니다.

— 전혀 다른 것이군요.

그렇습니다. 정리해 보면

- str2의 선언은 배열의 각 요소에 초기값을 설정하고 있습니다.
- str3의 선언은 포인터 변수가 가리키는 주소를 이 변수의 초기값으로 설정하고 있습니다.

이렇게 생각해 보면 전혀 틀리다는 것이 이해가 될 것입니다.



[그림 2-5] 문자열의 포인터와 문자의 배열

☆ 중 요 ☆

문자 배열

```
char str[] = "ABC";
```

배열의 각 요소 str[0] ~ str[3]에 초기값 'A', 'B', 'C', '\0'을 설정한다.

문자의 포인터

```
char *str = "ABC";
```

포인터의 초기값으로 문자열 리터럴 "ABC"의 선두 문자의 어드레스를 설정한다.

— 그러나 여러가지 프로그램을 보면 양쪽 모두를 사용하고 있는데 어느쪽을 사용해도 괜찮다는 의미입니까?

그것은 잘못된 생각입니다. str2(배열), str3(포인터)에 대해서

```
(1) str2 = "ABC";  … X
(2) str3 = "DEF";  … O
```

라고 썼을 때(2)는 바르게 컴파일 할 수 있지만 (1)은 컴파일할 수도 없습니다.

왜 그런지 아십니까?

— 글쎄요

오른편에 문자열을 써, 그것을 대입한다고 하는 것은 그 문자열의 선두 문자 어드레스를 대입하는 것이 됩니다.

여기에서 포인터와 배열의 차이를 생각해 주십시오. str2는 배열명－값으로는 그 배열의 선두 어드레스를 갖습니다－이지만 str3은 포인터입니다. 전자는 변경이 불가능한 상수이고 후자는 변경이 가능한 변수입니다.

— 아 그렇군요.

앞에서 말했듯이 문자열은 기본적으로 배열입니다. 정수의 배열을 생각해 봅시다.

```
int x[10];
int a;

x = &a;
```

여기에서 x = &a라고 쓸 수 없다는 것을 알겠지요. 문자열의 경우도 같습니다. str2(배열), str3(포인터)의 두 번째 요소에 각각 문자 'Z'를 대입해 봅시다.

```
<1> str2[1] = 'Z';
<2> str3[1] = 'Z';
```

□ 컬럼 2-8 □

문자열에서 포인터가 자주 사용되는 이유

K&R에서는 자동 변수 배열을 초기화 하는 것이 불가능했다. 따라서 함수 내부에서 다음과 같은

```
char str[] = "ABCD";            (1)
```

선언은 할 수 없지만 static 배열의 초기화는 가능하기 때문에 다음과 같은

선언이 가능했다.

```
static char str[] = "ABCD";     (2)
```

한편 포인터 변수는 자동 방식과 정적인 방식이 모두 허용되기 때문에 아래와 같은

```
char *str = "ABCD"              (3)
static char *str = "ABCD"       (4)
```

선언이 가능했다.

(1)은 ANSI에서 새롭게 인정한 방법이지만 널리 사용되지는 않는다.

실제 함수가 호출될 때에 배열 영역의 확보 및 초기화가 되기 때문에 비효율적이라고 볼 수 있다(우선 문자열 상수 "ABCD"의 영역이 어딘가에 확보된다. 또, 함수가 호출될 때에는 이것과 별도로 str이라는 배열의 영역이 할당되어 그 각 요소가 'A', 'B', 'C', 'D', '\0'으로 초기화 된다).

(2), (3), (4)의 의미는 각기 다르며 이중에 static을 붙이지 않은 (3)의 방법이 가장 많이 사용된다.

배열이나 포인터가 정적일 필요가 있는지를 잘 생각하여 상황에 따라 적절하게 사용하는 것이 현명한 방법이다.

<1>이 옳다는 것은 알겠지요. 문자열이라 해도 문자의 배열입니다. 배열의 한 요소에 값을 대입하는 것이기 때문에 당연한 것입니다.

<2>는 옳다고 생각합니까?

— str3은 메모리상의 어딘가에 "def"라고 저장되어 있는 영역을 가리키고 있습니다. 이것은 상수와 같은 것이기 때문에 틀린 것이 아닐까요?

ANSI에서 이와같은 문장을 수행할 경우에 대한 결과를 정의하지 않았습니다.

즉, 문자열 상수의 변경이 가능한지의 여부는 컴파일러에 따라 다르다는 의미입니다. 따라서 현재 사용하는 컴파일러가 문자열 상수의 변환을 허용한다고 해도 될 수 있는대로 이와같은 방법을 사용하지 않는 것이 좋다고 말할 수 있습니다.

다음의 선언이 옳다고 생각합니까?

```
char str[3] = "abc";
```

— 틀렸습니다. 왜냐하면 문자열 상수에는 널 문자가 자동으로 붙여지기 때문에 초기치 값은 4 문자를 나타냅니다. 따라서 배열의 크기보다 초기값의 크기가 크기 때

문에 컴파일 에러입니다.

아닙니다. 배열의 크기와 널 문자를 포함하지 않은 문자열의 크기가 같은 경우에 한해서는 널 문자를 붙이지 않은 문자열로 초기화합니다. 즉, str[0]~[2]에 각각 'a', 'b', 'c'가 초기값으로 대입됩니다. '\0'은 어디에도 대입되지 않습니다.

따라서

```
char str[3] = {'a', 'b', 'c'};
```

라고 해석됩니다. 단, 다음과 같이

```
char str[3] = "abcd";
```

문자열 상수의 크기가 배열의 크기보다 클 경우에는 컴파일시 에러가 됩니다.

☆ 중 요 ☆

문자 배열의 초기화에 대하여

```
char str[3] = "ABC";
```

와 같이 배열의 크기와 문자열의 길이가 같은 경우 '\0'이 첨가되지 않고 초기화된다.

그러면

```
char str[3] = {'a', 'b', 'c', '\0'};
```

은 맞는 것 같습니까?

— 이것은 틀렸다고 생각합니다.

그렇습니다. 이것은 틀립니다. 따라서 처음에

```
char str1[] = {'a', 'b', 'c', '\0'};
```

과

```
char str2[] = "abc";
```

이 같다고 설명을 했습니다만 배열의 크기가 지정되어 있고 '\0'를 포함하지 않은 문자수와 배열의 크기가 같을 경우에만 예외로 처리되는 것에 주의하십시오.

□ 컬럼 2-9 □

"abc"[2] ··· 문자열에 첨자를 붙이면....

컬럼 2-4 「a[3]과 3[a]」에서 설명했듯이 []의 피연산자의 한쪽은 「어떤 수형 T의 포인터」이고 다른 한쪽은 정수형이면 된다.
"abc"와 같은 문자열 상수는 「문자열 포인터」형을 가진다는 사실에서 다음과 같은 프로그램을 생각할 수 있다.

```
int main(void)
{
    char c;

    c = "ABC"[0];      ··· (1)
    c = 1["DEF"+1];    ··· (2)
}
```

문자열 상수의 값 → string pointer type.

구체적으로 보면 "ABC"는 선두 문자 'A'를 가리키는 포인터이다.
(1)에서는 c에 'A'가 대입된다. (2)에서 "DEF+1"은 문자열 "DEF"의 선두 다음 문자 'E'를 가리키는 포인터이다. 1["DEF"+1]은 ("DEF"+1)[1]과 같은 것이기 때문에 결국 c에는 'F'가 대입되게 된다. 가지고 있는 컴파일러에서 이와 같은 프로그램을 수행시킬 수 있나 확인해 보기 바란다(수행시킬 수 없다면 ANSI나 K&R의 표준조차 따르지 않는 것이다).

▷····· 2-2-2 문자열의 복사, 첫번째

— 문자열과 포인터, 배열에 관한 기본적인 내용은 이제 알겠습니다. 간단한 예를 하나 들어 설명해 주시면 좋겠습니다.

리스트 2-6의 프로그램은 문자열을 복사하는 함수입니다.

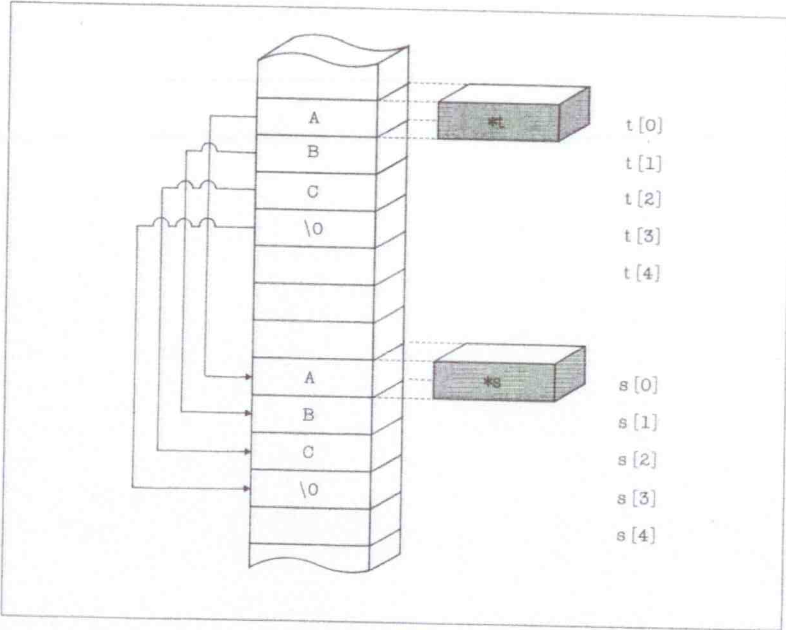
이 프로그램은 이해가 가겠지요. s, t는 포인터이지만 []의 첨자를 붙여 배열처럼 액세스할 수 있습니다.

— 널 문자가 나올 때까지 복사가 됩니까?

그렇습니다. 포인터 s, t를 포인터답게 사용해 다시 작성하면 리스트 2-7과 같이 됩니다.

[리스트 2-6] 문자열을 복사하는 함수, 첫번째

```
/*
    List 2-6  문자열을 복사하는 함수, 첫번째
*/

void strcpy(char *s, char *t)
{
    int   i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

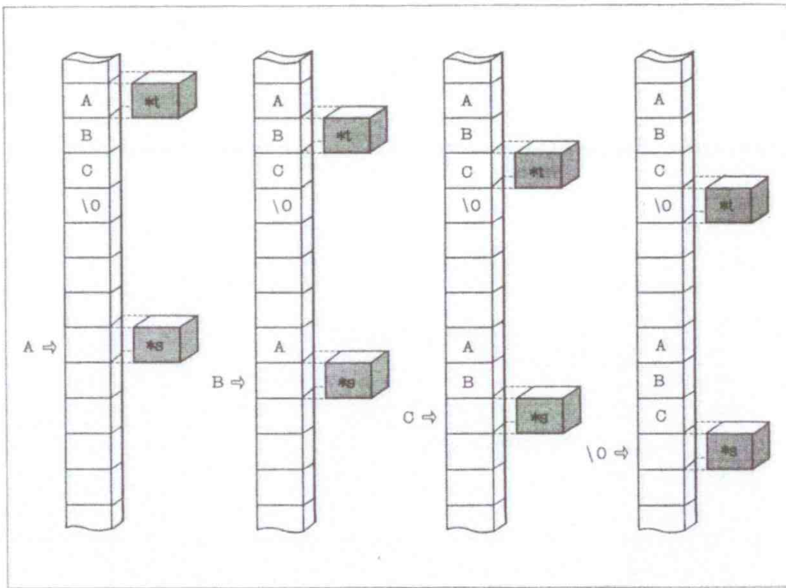


[그림 2-6] 포인터를 배열처럼 사용한 문자열의 복사

- 잘 모르겠는데요.

[리스트 2-7] 문자열을 복사하는 함수, 두 번째

```
/*
    List 2-7  문자열을 복사하는 함수, 두 번째
*/

void strcpy(char *s, char *t)
{
    while (( *s = *t ) != '\0') {
        s++;
        t++;
    }
}
```



[그림 2-7] 포인터를 포인터답게 사용한 문자열의 복사

우선

```
*s = *t
```

로 t가 가리키고 있는 문자가 s가 가리키고 있는 곳에 대입됩니다. 그 후 s 및 t를 증가시킵니다. 그것에 의해 s 및 t는 각각 다음 문자를 가리키게 됩니다. 그리고 '\0'를 만날 때까지 같은 처리를 반복합니다.

— 초보자로서는 이해하기 어렵습니다만 두 번째 프로그램의 장점은 무엇입니까?

우선 첫번째 프로그램에서 사용된 i라는 변수를 사용할 필요가 없습니다.

— 그래서 메모리를 절약할 수 있다는 의미입니까?

아닙니다. 그것은 본질적으로 다릅니다. 첫번째 프로그램에서 사용한 s[i]나 t[i]는 포인터 변수 s와 t가 가리키고 있는 곳으로부터 i개 뒤의 요소를 나타내는 것입니다. 따라서 루프를 돌 때마다 매회 첨자의 계산—즉, i개 뒤의 요소의 값을 갖거나 대입하는 등—을 하지 않으면 안됩니다.

— 알겠습니다.

배열의 첨자는 내부에서 일일이 계산되어진다는 뜻이군요.

두 번째 프로그램을 보십시오. 매회 s 및 t를 증가시키고 있지만 *s 및 *t를 사용하여 포인터 변수가 가리키고 있는 변수를 그대로 액세스할 수 있기 때문에 첫번째와 같이 까다로운 내부 처리는 필요하지 않게 됩니다.

결국 두 번째 프로그램이 프로그램도 작아지고 또, 수행 속도도 약간 빠르게 될 가능성이 높습니다.

두 번째 프로그램을 좀더 간단하게 작성하면 리스트 2-8과 같이 됩니다.

[리스트 2-8] 문자열을 복사하는 함수, 세 번째

```
/*
    List 2-8  문자열을 복사하는 함수, 두 번째
*/

void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```


— 이렇게 짧게 될 수 있습니까?

어떻게 이와같이 짧게 작성할 수 있는가는 잘 생각해 보면 알 수 있기 때문에 따로 설명하지 않겠습니다.

▶ 문제 2-1

다음 프로그램

```
int main(void)
{
    char *p = "ABCD";
    char *q;

    q = p;
    return(0);
}
```

이 하는 일이 무엇인가를 설명하십시오. 문자열을 복사하는 것입니까?

▷ …… 2-2-3 문자열의 복사, 두 번째 - 포인터의 리턴

— 그런데 제가 사용하고 있는 컴파일러의 매뉴얼에서는 strcpy 함수가

```
char *strcpy(char *s, const char *t)
```

로 되어 있습니다.

함수의 리턴 값이 char *인 것과 인자의 선언 중 const char *를 잘 모르겠군요?

— 선생님은 제 마음을 잘 알고 계시군요. 제가 어떻게 알겠습니까.

초보자들이 잘 모르는 부분이 대개 비슷 비슷하지요.

그럼 먼저 const char *t부터 설명하겠습니다. 이것은 const char의 포인터라는 선언입니다. 상수의 포인터이기 때문에 t가 가리키는 내용을 바꿔 쓸 수 없게 됩니다.

— 이 함수의 내부에서 바뀔 것 같지 않은데 그래도 const가 필요한가요?

그렇게도 말할 수 있습니다. 그러나 strcpy 함수를 사용하는 사용자의 입장에서는 어떨까요. 포인터를 주고 받기 때문에 어쩌면 변경되는 것이나 아닐까 하는 불안이 생기겠지요. 그러나 이와같은 프로토타입을 사용자에게 제공하여 사용자가 안심하고 strcpy 함수를 호출할 수 있습니다.

— 잘 알겠습니다.

덧붙여서 말하면 const * char t라고 하면 어떻게 되겠습니까?

— 네? 그런 것도 있습니까?

이것은 char의 const 포인터라고 하는 선언입니다. 따라서, 이렇게 하면 포인터 t의 값을 변경할 수 없게 됩니다.

앞의 const char *t은 상수의 포인터이지만 이번의 const * char t는 포인터 쪽이 상수로 되어 있습니다. 따라서, t가 가리키는 내용이 바뀔 수 있습니다.

— strcpy 함수는 루프 내에서 t를 증가시키기 때문에 그와같은 선언은 할 수 없겠군요.

맞습니다.

다음에는 함수의 리턴 값이 char *라는 것을 설명하겠습니다. 간단하게 말하자면 함수가 되돌려주는 수형이 char의 포인터형이라는 것입니다.

컬럼 2-10

const 수형 수식자

ANSI에서는 K&R에 없는 키워드 const가 도입되었다. 따라서 선언에서의 const가 무엇에 걸리는지를 충분히 이해해야 한다(이 책은 포인터 편이기 때문에 선언에 관한 상세한 설명은 하지 않겠다).

아래에 간단한 선언과 그 의미를 열거하였으므로 참고하기 바란다.

(배열이나 함수의 포인터 등이 나오면 좀더 복잡하게 된다).

```
int i;                      /* int */
int *p;                     /* int의 포인터 */
const int i;                /* int 상수 */
const int *p;               /* int 상수의 포인터 */
int const *p;               /* int 상수의 포인터 */
int * const p;              /* int의 상수 포인터 */
const int * const p;        /* int 상수의 상수 포인터 */
```

프로토타입 선언 중의 포인터를 수식하는 const는 그 인자가 가리키는 객체가 그 함수에 의해 변경되지 않는 것을 나타낸다. 여기에서 실인자로 넘겨주는 것은 상수 객체의 포인터가 아니어도 된다. 상수가 아닌 객체의 포인터를 넘겨주어도 관계없다.

예를 들어 strcpy 함수의 두 번째 인자가 const char * 수형의 포인터이어야 하는 것은 아니며 char * 수형의 포인터를 넘겨줄 수 있다.

일반적으로 인자의 교환은 「대입」과 같은 규칙이 적용된다. 상수의 포인터에 상수가 아닌 객체의 포인터를 대입하는 것은 가능하지만 반대로 상수가 아닌 객체의 포인터에 상수 포인터를 대입하는 것은 불가능하다.

이유는 각자 생각해 보기 바란다.

— 단지 그것 뿐입니까?

그렇습니다. int func(...)라고 한다면 이것은 int형을 되돌려 준다는 것입니다. 이 int가 char *로 되었을 뿐입니다.

— 알 것 같기도 하고 모를 것 같기도 하고... 그럼 구체적으로 무엇을 되돌려주는 것입니까?

이 함수는 2개의 char의 포인터형 변수 s, t를 받고 문자열을 복사합니다. 그런 후 s와 같은 값을 되돌려 줍니다.

따라서 리스트 2-8을 라이브러리 사양에 따라 바꿔 작성하면 리스트 2-9와 같이 됩니다.

변수의 선언

```
char *p = s;
```

에서 p를 char의 포인터형이라고 선언하고 초기값으로 s의 값이 대입됩니다.

— s가 p에 대입되는 것입니까?

그렇습니다. *p에 대입되는 것이 아닙니다. 따라서 이것은

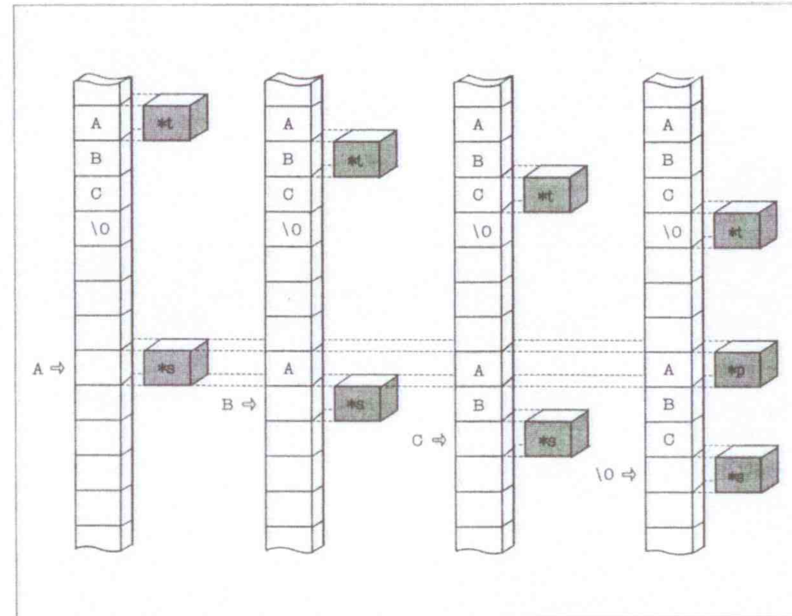
[리스트 2-9] 문자열을 복사하는 함수, 네 번째

```
/*
   List 2-9 문자열을 복사하는 함수, 네 번째
*/

char *strcpy(char *s, const char *t)
{
    char  *p = s;

    while (*s++ = *t++)
      ;
    return(p);
}
```

```
char *p;
p = s;                /* *p = s가 아님 */
```

과 같게 됩니다.

— s는 프로그램 속에서 증가하지만 p는 그대로이기 때문에 처음에 s가 가리키고 있는 문자를 가리키게 됩니까?



[그림 2-8] 포인터를 포인터답게 사용한 문자열의 복사

그렇습니다. 그림 2-8을 보면 알 수 있습니다.

함수의 끝에서 p를 그대로 되돌려 줍니다. 따라서 복사전 문자열 선두 문자의 어드레스를 돌려주게 됩니다.

— 프로그램의 동작은 알겠지만 무엇 때문에 그런 것을 되돌려 주는 것입니까?

간단히 말하자면 「문자의 포인터」가 여러가지로 편리하기 때문입니다.

— 편리하다고요?

예를 들면 printf 함수의 처음 인자는 const char의 포인터형입니다. 따라서 s에 "ABC"를 복사하고 그것을 출력하는 아래의 프로그램을

```
char s[10];

strcpy(s, "ABC");
printf(s);
```

보다 간단하게 다음과 같이 기술할 수 있습니다.

```
char s[10];

printf(strcpy(s, "ABC"));
```

— 아 예, 참 편리하군요.

s, t의 양쪽에 "ABC"를 복사하는 아래의 프로그램을

```
char s[10], t[10];

strcpy(s, "ABC");
strcpy(t, s);
```

위에서와 마찬가지로 다음과 같이 작성할 수 있습니다.

```
char s[10], t[10];

strcpy(t, strcpy(s, "ABC"));
```

s에 t를 복사하고 그 길이를 표시하는 프로그램을 다음과 같이 간단하게 작성할 수 있습니다.

```
char s[10], t[10];

printf("%d", strlen(strcpy(s, t)));
```

— strlen 함수는 문자열의 길이를 되돌려 주는 함수입니까?

그렇습니다. 뒤에서 실제로 strlen 함수를 작성해 보도록 합시다.

▷ ····· 2-2-4 문자열과 기억 수명 - 없어진 것을 가리키는 포인터

그러면, 문자열 "ABC"의 포인터를 되돌려 주는 함수를 작성해 보십시오.

— 포인터를 되돌려 주면 됩니까? 할 수 있습니다.

[리스트 2-10] 포인터를 되돌려 주는(틀린) 프로그램

```
/*
    List 2-10  포인터를 되돌려 주는(틀린) 프로그램
*/

char *abc(void)
{
    char  p[] = "ABC";

    return(p);
}

int main(void)
{
    printf("%s\n", abc());
    return(0);
}
```

수행해 보십시오.

— 어, 이상한 문자가 표시되는데요.

abc 함수의 배열 p는 자동 변수입니다. 결국 abc 함수 내에서만 <살아 있다>는 말이 됩니다. 그 배열 선두 요소의 포인터를 돌려주어도 의미가 없게 됩니다. 왜냐하면 main 함수에 되돌아 간 때에는 배열 p는 <죽게> 되고 배열의 내용은 없어졌기 때문입니다.

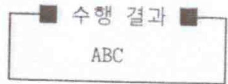
— 정말 그렇군요. 잘 생각해 보았으면 알 수 있었을텐데, 함수 내부에서만 살아 있는 데이터의 어드레스를 되돌려 주는 것은 의미가 없다는 말씀이지요.
그러면 어떻게 하면 됩니까?

배열 p에 static 기억 클래스 지정자를 붙여서 선언합니다. 따라서 p라는 이름은 함수 abc 내에서만 의미를 갖지만 배열 p는 함수 abc가 수행을 마친 이후에도 없어지지 않습니다. 즉, 프로그램 수행의 시작부터 끝까지 <살아> 있을 수 있습니다.

리스트 2-11과 같이 고치면 바르게 수행됩니다.

[리스트 2-11] 포인터를 되돌려 주는 프로그램

```
/*
    List 2-11  포인터를 되돌려 주는 프로그램
*/

char *abc(void)
{
    static char   p[] = "ABC";

    return(p);
}

int main(void)
{
    printf("%s\n", abc());
    return(0);
}
```



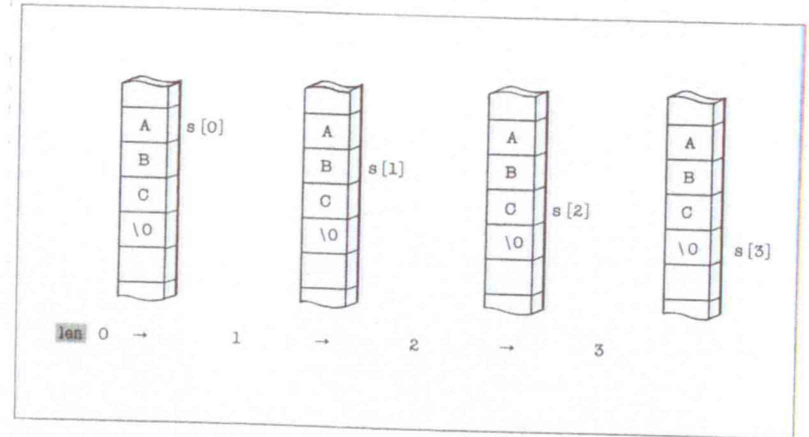
▷ ····· 2-2-5 문자열의 길이를 구하기

다음에는 문자열의 길이를 구하는 함수를 작성해 보겠습니다. 우선 문자열의 복사 때와 마찬가지로 포인터를 배열처럼 사용한 예를 나타내보면 리스트 2-12가 됩니다.

[리스트 2-12] 문자열의 길이를 구하는 함수, 첫번째

```
/*
    List 2-12  문자열의 길이를 구하는 함수, 첫번째
*/

int strlen(char s[])
{
    int   len;

    for (len = 0; s[len] != '\0'; len++)
        ;
    return(len);
}
```

— 이 프로그램에서는 매회 내부에서 첨자의 계산을 하기 때문에 비효율적이군요.



[그림 2-9] 포인터를 배열처럼 사용한 문자열 길이의 계산

맞습니다. 그러면 포인터를 포인터답게 사용해서 프로그램을 바꾸어 보십시오.

— 했습니다. 리스트 2-13입니다.

[리스트 2-13] 문자열의 길이를 구하는 함수, 두 번째

```
/*
    List 2-13 문자열의 길이를 구하는 함수, 두 번째
*/

int strlen(char *s)
{
    int   len;

    for (len = 0; *s != '\0'; s++)
        len++;
    return(len);
}
```

맞습니다만 실은 이 프로그램도 효율적이지 못합니다.

— 이것 이상으로 효율이 높은 프로그램을 작성할 수 있습니까?

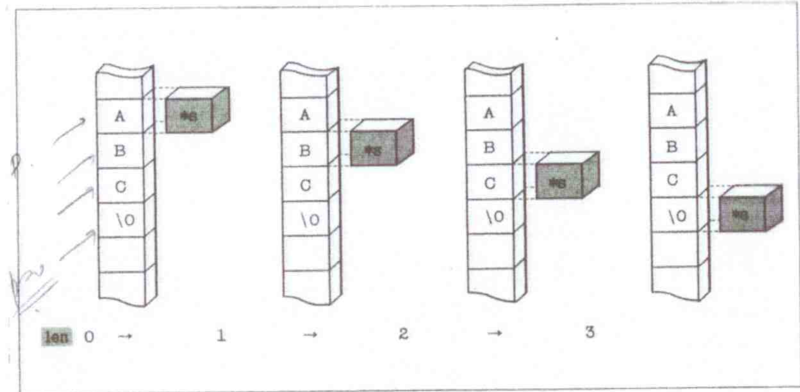
리스트 2-13을 봐 주십시오. 루프를 반복할 때 포인터 s와 문자의 길이를 나타내는 len을 둘다 증가시키고 있습니다. 이 점이 비효율적입니다.

— 그렇게 말씀하시니까 그런 느낌이 듭니다... 따로 따로 1개씩 늘린다고 하는 것이 왠지 이상하군요.

그러한 비효율적인 점을 없앤 프로그램이 리스트 2-14입니다.

[리스트 2-14] 문자열의 길이를 구하는 함수, 세 번째

```
/*
    List 2-14 문자열의 길이를 구하는 함수, 세 번째
*/

int strlen(char *s)
{
    char  *p = s;

    while (*p != '\0')
        p++;
    return(p - s);
}
```

[그림 2-10] 포인터를 포인터답게 사용한 문자열 길이의 계산(1)

□ 컬럼 2-11 □

문자열 길이와 sizeof(문자열)

표준 라이브러리로 제공되는 strlen 함수는 문자열의 논리적 길이를 돌려주는 함수이다. '\0'을 제외한 문자열의 길이를 되돌려 준다.
문자열의 길이를 구하는 방법에는 sizeof 연산자로 구하는 방법이 하나 더 있다.
단, 이 방법은 '\0'을 포함한 크기(바이트 수)를 구하게 된다.
예를 들면 다음과 같다.

```
strlen("")        -> 0        sizeof("")        -> 1
strlen("ABCD")    -> 4        sizeof("ABCD")    -> 5
```

여기에서 주의하지 않으면 안될 사항이 있다.

(1) ANSI 표준이 아닌 컴파일러의 경우

sizeof의 동작이 올바르게 되지 않을 수가 있다.
sizeof 연산자는 적용되는 식의 값보다는 수형이 중요하다.
오래된 컴파일러에서는 sizeof("ABCD")를 sizeof(char *)라고 간주하는 것이 있다. 만약 포인터가 2바이트라면 아무리 긴 문자열을 적용해도 이 값은 항상 2가 된다.

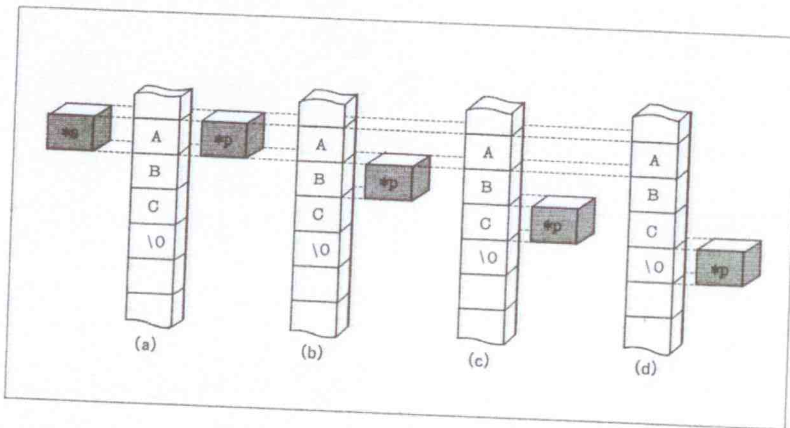
(2) strlen 함수는 논리적 길이를, sizeof는 기억 장소의 크기를 돌려 준다.

이미 설명했듯이 strlen 함수는 '\0'의 앞 문자까지의 길이를 되돌려 준다. sizeof 연산자는 기억 영역에 차지한 크기를 되돌려 준다.

따라서 다음과 같은 차이점이 있다.

```
strlen("ABC\0DEF") -> 3          sizeof("ABC\0DEF") -> 8
```

문자열 리터럴 "ABC\0DEF"는 도중에 '\0'를 포함하고 있기 때문에 strlen은 [ABC]의 3문자라고 간주한다. 한편 sizeof는 기억 장소에 할당된 영역의 바이트 수를 되돌려주기 때문에 [ABC\0DEF\0]의 8바이트가 되는 것이다.



[그림 2-11] 포인터를 포인터답게 사용한 문자열 길이의 계산(2)

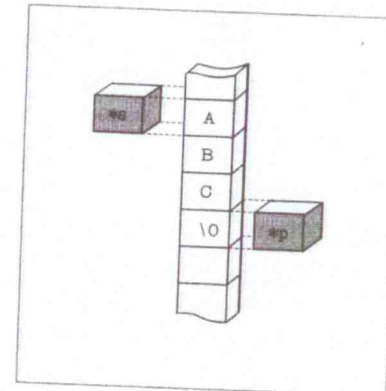
먼저 p가 char의 포인터형으로 선언하였고 초기값으로 s가 대입됩니다.

― p에 s가 대입되기 때문에 p, s 모두 배열의 선두 문자를 가리키는 것이 되는군요.

다음에 *p가 '\0'과 같아질 때까지 p가 증가됩니다. 마지막으로 그림 2-12와 같이 됩니다. 프로그램에서

```
return(p - s);
```

를 보십시오.



[그림 2-12] 루프 종료시 *s와 *p

포인터와 포인터 사이에서는 뺄셈이 가능하기 때문에 p-s를 계산하여 p가 가리키고 있는 요소와 s가 가리키고 있는 요소가 몇개의 요소만큼 떨어져 있는지를 계산할 수 있습니다. 이 예에서는 3이 됩니다.

― 포인터간에 뺄셈이 가능한 것을 이용해 문자열의 길이를 계산한 것이군요. 리스트 2-14에서 증가된 것은 p뿐이기 때문에 리스트 2-13보다 효율이 높다는 말씀인가요?

그렇습니다.

― 그런데 제가 사용하고 있는 컴파일러의 매뉴얼에는 strlen 함수가

```
size_t *strlen(const char *s)
```

라고 나와 있습니다. size_t는 무엇입니까?

size_t는 sizeof 연산자의 리턴형으로 부호없는 정수입니다. 이것은 <stddef.h> 등의 헤더 화일에서 아래와 같이 선언되었습니다.

```
typedef unsigned int size_t;
```

물론 unsigned int 이외의 수형인 경우도 있으며 컴파일러에 따라 다릅니다.

■ 컬럼 2-12 ■

size_t와 ptrdiff_t

리스트 2-14에서는 포인터간의 뺄셈에 의해 문자열의 길이를 구하였다. 두 개의 포인터간의 뺄셈 결과를 나타내는 수형이 <stddef.h>에 ptrdiff_t로 정의되어 있다.

```
&p[0] - &p[100]
```

과 같이 낮은 위치를 가리키는 포인터로부터 높은 위치를 가리키는 포인터를 빼는 경우를 고려한다면 ptrdiff_t는 부호가 있는 정수를 나타내는 수형이어야 한다는 것을 쉽게 알 수 있다. 8086 CPU용 컴파일러의 대부분이

```
typedef unsigned int size_t;
tyepdef signed   int ptrdiff_t;
```

라고 선언하고 있다. 거의 모든 컴파일러에서 size_t- 즉, unsigned int-는 0~65535까지를, ptrdiff_t-즉, signed int-는 -32768부터 32767까지를 갖는다. 다음과 같이 정의된 크기 35000 문자의 배열 s를 생각해 보자

```
char s[35000U];
```

ptrdiff_t형은 32767까지 밖에 나타낼 수 없기 때문에 다음의 프로그램을 실행해 보면

```
int main(void)
{
    static char s[35000U];

    printf("%d\n", &s[33000U] - &s[0]);
    printf("%d\n", &s[0] - &s[32536]);
    return(0);
}
```

둘 다 -32536이라고 표시된다. 큰쪽에서 작은쪽을 빼도 음수가 되어 버린다.

— 왜 그렇게 정의된 것입니까?

K&R에서는 sizeof 연산자의 결과 수형은 기본적으로 정수와 같다고 되어 있으며 컴파일러에 따라 int이거나 unsigned int이거나 unsigned long이거나 해 각각 다를 수 있습니다. 한편 ANSI에서는 size_t라는 수형을 정하고 있습니다. 따라서 ANSI 표준 컴파일러라면 sizeof 연산자의 결과 수형이나 strlen 함수의 결과 수형은 반드시 size_t라는 것이 보증됩니다.

▷ ····· 2-2-6 문자열의 결합, 문자열 처리에서 하기 쉬운 실수

다음 프로그램을 보십시오.

[리스트 2-15] 문자열을 결합하는(틀린) 프로그램

```
/*
    List 2-15  문자열을 결합하는(틀린) 프로그램
*/

#include <stdio.h>
#include <string.h>

int main(void)
{
    char  *s = "I love ";

    strcat(s, "you");
    printf(s);
    return(0);
}
```

— strcat라는 함수는 어떤 기능을 합니까?

strcat 함수는 두 개의 문자열을 결합해 주는 함수입니다.

— cat란 고양이입니까?

하하, cat는 결합한다는 의미의 영어 단어 concatenate의 약자입니다. strcat의 개요는 다음과 같습니다.

strcat

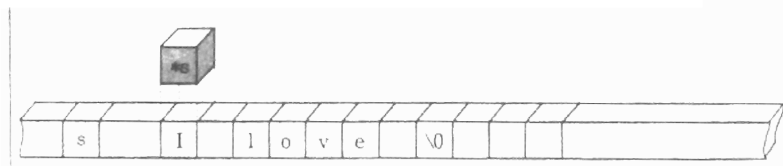
형　　식	:	char *strcat(char *s1, char *s2);
프로토타입	:	<string.h>
기　　능	:	문자열 s1의 끝에 문자열 s2를 복사한다
리 턴 값	:	s1의 값을 돌려준다.

"I love "라는 문자열에 "you"를 결합하여 "I love you"라는 문자열을 표시하는 프로그램이군요.

맞습니다만 이 프로그램에는 중대한 버그가 있는데 알 수 있겠습니까?

– 전혀 모르겠는데요! strcat라는 함수조차도 처음 들었습니다.

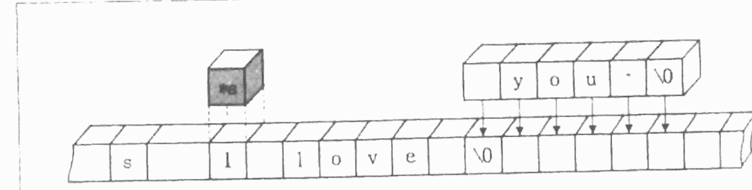
포인터 s와 문자열이 그림 2-13과 같이 되는 것은 알 수 있겠죠



[그림 2-13] 포인터와 문자열

네. 지금까지의 그림과는 달리 가로로 되어 있군요.

렇습니다. strcat 함수는 그림 2-14와 같이 문자열의 끝인 '\0'으로부터 1문자씩 사를 합니다.

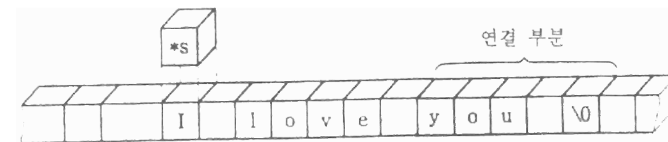


[그림 2-14] 문자열의 결합

– 결합 후는 그림 2-15와 같이 됩니까?

네, 맞습니다.

– 그러면 어디에 버그가 있다는 말씀입니까?



[그림 2-15] 결합 후의 문자열

림 2-15를 잘 봐 주십시오. <you\0>이 결합되는 검게 칠한 부분이 비어 있다 보증은 어디에도 없습니다. 이 영역에는 다른 변수나 시스템의 중요한 정보가 시 장되어 있을 가능성이 있습니다.

그러니까 다른 변수의 값을 바꿔 쓰거나 프로그램을 파괴하거나 할 가능성이 있다는 의미입니까? 정말 뜻밖입니다.

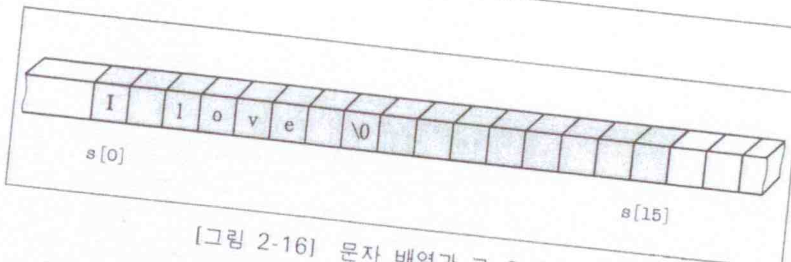
맞습니다. 프로그램이 폭주할 위험성도 있습니다.

올바 프로그램을 작성하려면 어떻게 하면 됩니까?

이 프로그램은 메모리 상의 어딘가에 저장되어 있는 문자열을 가리키는 포인터 변수를 사용하고 있지만 이제까지 사용하였던 이러한 방법으로는 안됩니다. 따라서 포인터 변수가 아닌 배열을 사용합니다. 올바른 프로그램은 리스트 2-16과 같이 됩니다.

[리스트 2-16] 문자열을 결합하는 프로그램

```
/*
   List 2-16 문자열을 결합하는 프로그램
*/

#include <stdio.h>
#include <string.h>

int main(void)
{
    char  s[16] = "I love ";

    strcat(s, "you");
    printf(s);
    return(0);
}
```



[그림 2-16] 문자 배열과 그 초기값

– s를 배열로 선언하면 그림 2-16과 같이 16 문자분의 영역이 메모리 상에 반드시 할당되기 때문에 안심할 수 있겠네요.

그렇습니다. 필요한 기억 영역은 스스로 할당해야만 합니다.

☆ 교 훈 ☆

필요한 기억 영역은 스스로 할당해야 한다.

▶ 문제 2-2

표준 라이브러리에 있는 strcat 함수

char *strcat(char *s1, const char *s2)

를 작성하십시오.

▷ 2-3 포인터의 연산

리스트 2-6에서부터 리스트 2-9까지의 문자열 복사 프로그램을 참고로 int형 배열을 복사하는 함수를 작성해 보십시오. 단, 배열의 크기도 인자로 주어졌다고 합시다.

— 어째서 배열의 크기를 넘겨줄 필요가 있습니까?

배열의 교환은 선두 요소의 포인터라는 수형을 주고 받음을 통하여 수행합니다. 따라서 받는 쪽은 배열의 크기를 알 방법이 없습니다.

— 그렇습니까? 문자열의 경우에는 마지막에 '\0'이 있기 때문에 끝날 곳을 알 수 있습니다만 문자열 이외에서는 그와같은 것을 알 수 없군요.

그럼, 프로그램을 작성해 주십시오.

— 포인터를 배열처럼 첨자를 붙여 작성했는데요.... 리스트 2-17입니다.

[리스트 2-17] 정수 배열을 복사하는 함수, 첫번째

```
/*
    List 2-17  정수 배열을 복사하는 함수, 첫번째
*/

void intncpy(int s[], int t[], int n)
{
    int    i;

    for (i = 0; i < n; i++)
        s[i] = t[i];
}
```

그러면 포인터를 포인터답게 사용하여 다시 프로그램을 작성해 보십시오.

— 네, 문자수를 세어서 했습니다. 리스트 2-18과 같이 됩니다.

복사할 때에 포인터를 증가시켰군요. char형이라면 1바이트이기 때문에 단순하게 증가시키면 됩니다만 대부분의 PC용 컴파일러에서는 int형은 2바이트입니다. 포인터를 한 개 증가시켜 다음 요소를 가리키기 위해서는


```
s += 2;
t += 2;
```

[리스트 2-18] 정수 배열을 복사하는 함수, 두 번째

```
/*
    List 2-18  정수 배열을 복사하는 함수, 두 번째
*/

void intncpy(int *s, int *t, int n)
{
    while (n--) {
        *s = *t;
        s++;
        t++;
    }
}
```

라고 해야 하는 것이 아닐까요?

— 그렇군요.

하하. 프로그램을 작성하기 전에 속으셨군요. 포인터에 대한 연산은 그 요소의 크기를 단위로 하여 수행하게 됩니다. 따라서 s++와 t++가 맞는 것입니다.

▶ 문제 2-3

다음 프로그램을 수행하십시오.

```
int main(void)
{
  int  x[200];
  int  *p, *q;

  p = &x[0];
  q = p + 200;
  printf("q - p = %4d\n", q - p);
  printf("(unsigned)q - (unsigned)p = %4d\n", (unsigned)q - (unsigned)p);
  return(0);
}
```

200 ; pointer difference
400 ; integer difference
200 × 2
2byte

또 수행 결과에 대하여 생각해 보십시오.

□ 컬럼 2-13 □

포인터의 연산

이 책에서 설명하고 있듯이 포인터에 대한 연산은 그 포인터가 가리키는 수형의 크기를 단위로 하여 수행된다. 따라서

```
char    *cp;
int     *ip;
double  *dp;
```

에 대해

```
c++;        /* 1개 뒤의 요소를 가리킨다. cp의 값은 sizeof(char) 만큼
               증가한다 */
ip++;       /* 1개 뒤의 요소를 가리킨다. ip의 값은 sizeof(int) 만큼
               증가한다 */
dp += 5;    /* 5개 뒤의 요소를 가리킨다. dp의 값은 sizeof(double)*5
               만큼 증가한다 */
*(ip+6)     /* 6개 뒤의 객체 - 즉 ip + sizeof(int)*6번지의 int형
               변수 */
```

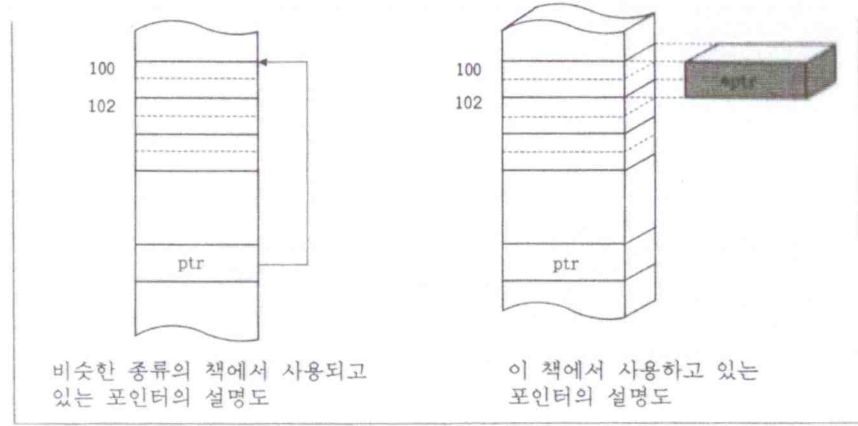
와 같이 된다. 포인터(물론 배열도 포함해서)의 연산에서는 그 포인터가 가리키는 객체의 크기에 따라 연산이 행해진다.

int를 가리키는 포인터인 ptr이 100번지에 저장되어 있는 int형 객체를 가리키는 경우를 생각해 보자.

여기에서 int형은 2바이트라고 하면 이 책과 비슷한 종류의 책에서 사용되고있는 설명도(다음 그림의 왼쪽 그림)에서는 ptr이 100번지를 가리키기 때문에

```
ptr++;
```

을 수행하면 ptr이 101이 된다고 착각할 수 있으나 이 책의 설명도(다음 그림의 오른쪽 그림)에서는 *ptr이 2바이트의 크기를 갖는 상자와 같이 표현되기 때문에 이와 같은 착각은 하지 않게 된다.



비슷한 종류의 책에서 사용되고 있는 포인터의 설명도

이 책에서 사용하고 있는 포인터의 설명도

제 3 장

포인터와 다차원 배열

▷ 3-1 다차원 배열
▷ 3-2 포인터의 배열과 다차원 배열
▷ 3-3 커맨드 라인 인자

C 언어에는 엄밀한 의미로 다차원 배열은 존재하지 않으며, 일차원 배열만이 존재합니다.

따라서 이차원 배열은 배열의 배열이라고 하는 형이 됩니다. 이것을 단지 그럴싸한 표현 정도로 생각해서는 안 됩니다.

다차원 배열이 배열의 배열이라고 하는 사실을 알게됨에 따라 C 언어의 다차원 배열과 포인터와의 관계를 보다 잘 알 수 있게 되는 것입니다.

▷ 3-1 다차원 배열

▷ ····· 3-1-1 다차원 배열은 배열의 배열이다

배열과 포인터와의 관계는 이제 대체로 안다고 생각되기 때문에, 이제는 다차원 배열에 대해서 공부해 보도록 하겠습니다.

— C 언어의 다차원 배열이란 좀 번거로워요. 예를들면 3차원 배열의 경우, a[i,j,k]라고 쓰지 않고 a[i][j][k]라고 써야 하기 때문입니다. 괄호가 너무 많아 성가셔요.

C 언어에 a[i,j]로 쓰지 않고 a[i][j]로 써야 하는 데에는 그럴만한 이유가 있습니다. 그 이유에 대한 설명은 뒤로 미루기로 하고, 일차원 배열에서 그것을 인자로 받는 함수를 생각했던 것과 같이 우선 이차원 배열을 인자로 받는 함수를 생각해 봅시다. 4*3 크기의 배열을 인자로 받는 경우를 예로 들겠습니다. 여기에도 여러가지 표현 방법이 있지만 정리하면 다음과 같은 세 가지 경우로 됩니다.

void func(int a[4][3])
{
}
(1) 배열 (크기를 정함)
	void func(int a[][3])
	{
	}
	(2) 배열 (크기 없음)
		void func(int (*a)[3])
		{
		}
		(3) 포인터

— (3)의 방법은 잘 모르겠습니다. 모든 경우에서 3은 생략할 수 없다고 하는 규칙이 있다는 것은 알고 있는데 왜 그런지 이해가 되지 않습니다.

세 가지 모두 일차원 배열과 마찬가지로 int의 포인터라는 것은 알겠지요?

— 네.

하하 아닙니다. a는 int의 포인터가 아닙니다.

— 그러면 무엇입니까?

a는 <크기 3인 int형의 배열>의 포인터입니다. 그것을 명시하기 위하여 3이 생략

되지 않는 것입니다.

— 까다롭군요. 이해가 잘 안됩니다.

처음부터 C 언어에는 「일차원 배열」만이 존재하고 엄밀한 의미에서의 「다차원 배열」은 존재하지 않습니다.

▨ 컬럼 3-1 ▨

a[i,j]라고 쓴다면...

다음과 같이 선언한 이차원 배열

```
int a[3][4];
```

가 있다고 하자. a[1,2]라고 쓰면 무엇을 의미하는 것이 될까?

여기에서 쉼표는 연산자로 간주된다. 쉼표 연산자는 좌측의 피연산 수식을 계산한 후에 이를 무시하고 우측의 피연산 수식을 계산하여 그 값을 가지게 된다. 따라서 1,2라고 하는 식은 2라는 값을 가지게 된다. 결국 a[1,2]는 a[2]와 같은 의미가 되어 버린다. 뒤에서 설명하듯이 a[1,2]는 크기가 4인 int형 배열의 포인터이다. 문맥에 따라서는 a[1,2]라고 써도 된다.

따라서 a[1,2]라는 표현 방법이 완전히 틀렸다고 딱 잘라서 말할 수는 없지만 a[1][2]와는 의미가 다르다는 것에 주의해야 한다.

☆ 체크 포인트 ☆

C 언어에서는 일차원 배열 만이 존재하고 다차원 배열은 존재하지 않는다.

— 아! 그렇군요.

예를들면 다음의 선언

```
int c[4];
int x[4][3];
```

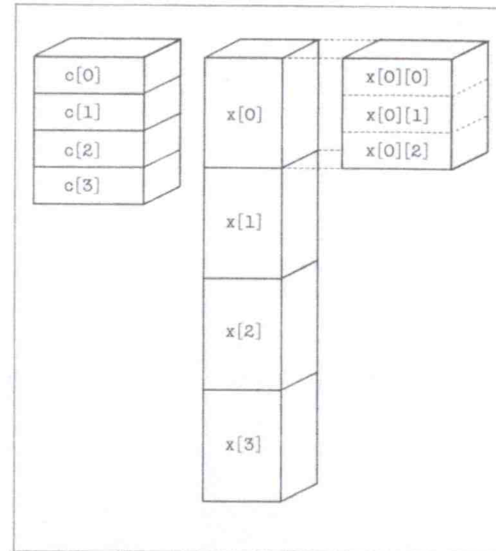
에서 전자는 <int형 변수>를 한 개의 요소로 하는 크기 4개의 배열 c를 선언하고 있습니다. 후자는 엄밀히 말하면 4*3의 배열을 선언하고 있다는 뜻이 아닙니다. 이것은 <int형의 크기 3의 배열>을 한 개의 요소로 하는 크기 4개의 배열을 선언하는 것입니다. 그림 3-1에서도 알 수 있듯이 c[0]-c[3]과 x[0]-x[3]은 모두 단순한 일차원의 배열로 간주할 수 있습니다.

— 위의 말은 int x[4][3]은 x[0] 부터 x[3] 이라는 네 개의 요소로 된 배열을 선언하고 있다는 것입니까?

그렇습니다. 자, c[3]과 같이 한 개의 요소만 생각할 때, 이것은 int형의 변수입니다. 그럼 x[2]는 무엇일까요.

— int형의 크기 3의 배열입니까?

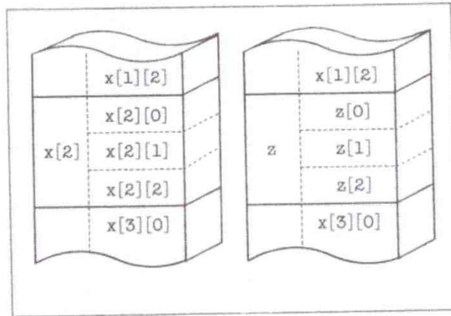
네 맞습니다. 여기서는 x[2]를 단순히 z라고 해 보겠습니다. 설명할 필요가 없지만 그림 3-2와 같이 되겠지요?



[그림 3-1] 일차원 배열과 이차원 배열

— 당연하지 않습니까? 단순히 바꿔 놓은 것 뿐이니까요.

이 그림을 잘 보면 직감적으로 중요한 것을 하나 발견할 수 있습니다. 자, 앞의 이야기에서 나왔던 a[i,j]라 쓰지 않고 a[i][j] 라고 쓸 수 밖에 없는 이유를.....



[그림 3-2] 이차원 배열의 해석

– 그렇게 말씀하시니까 알겠습니다.

엄밀하게 말하면 z(여기에서는 x[2])는 int형의 배열 이름입니다. 제2장 포인터와 배열에서도 설명했듯이 배열의 이름은 포인터이고 값으로써 배열의 시작 주소를 가집니다. 배열의 시작으로부터 i개 뒤의 요소는 *(z+i) 이고 이것은 z[i]라고 쓸 수도 있습니다.

– 여기에서, *(z+i)의 z를 본래의 x[2]로 되돌리면 *(x[2]+i)가 되고, 결국 x[2][i]가 되는 것이죠.

그렇습니다. 일차원 배열에서 배열명 c만을 사용하면 int의 포인터가 됩니다. 그럼 x는 무엇이 될까요.

– int형의 크기 3의 배열의 포인터입니다.

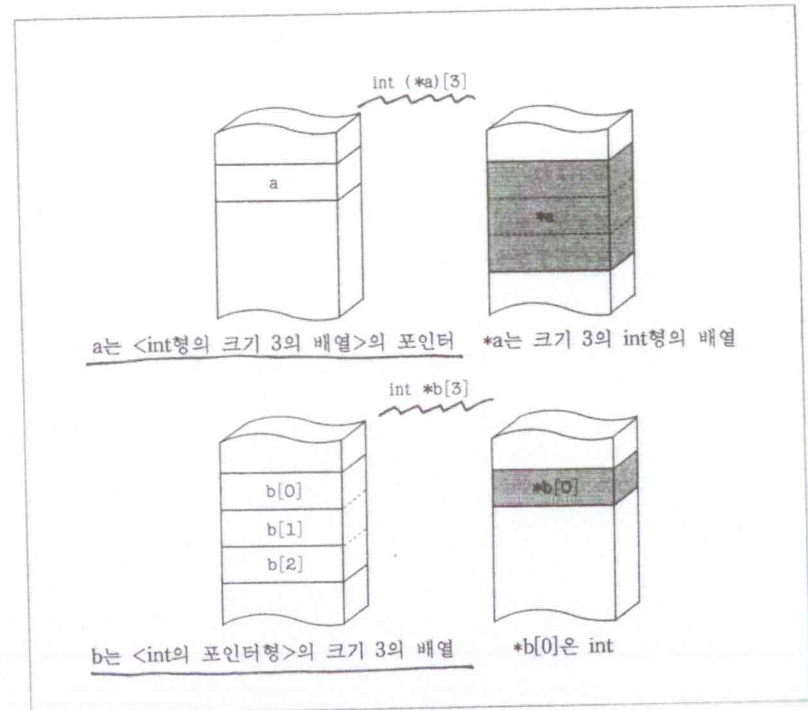
그렇습니다. int (*a)[3]과 int *b[3]을 그림으로 나타내면 그림 3-3과 같이 되고 서로 전혀 다르다는 것을 알 수 있습니다.

▶ 문제 3-1

본 문의 문답에 나왔던 다음의 두 가지

(1) int형의 크기 3이 배열의 포인터 int (*a)[3]
(2) int의 포인터형의 크기 3의 배열 int *b[3]

을 sizeof 연산자를 사용해 그 차이를 확인하는 프로그램을 작성하십시오.



[그림 3-3] 배열의 포인터와 포인터 변수의 배열

▷····· 3-1-2 함수간의 다차원 배열의 주고 받음

처음에 나왔던 함수 인자에 관한 예들을 이제는 좀 이해할 수 있지 않습니까? 함수를 부르는 쪽에서는

```
int x[4][3];
  .....
func(x);
```

라고 합니다. 불리워지는 쪽의 함수의 선언을 한 번 더 나타내 보겠습니다.

void func(int a[4][3])
{
 	
}
(1) 배열(크기를 정함)
	void func(int a[][3])
	{
	
	}
	(2) 배열(크기 없음)
		void func(int (*a)[3])
		{
		
		}
		(3) 포인터

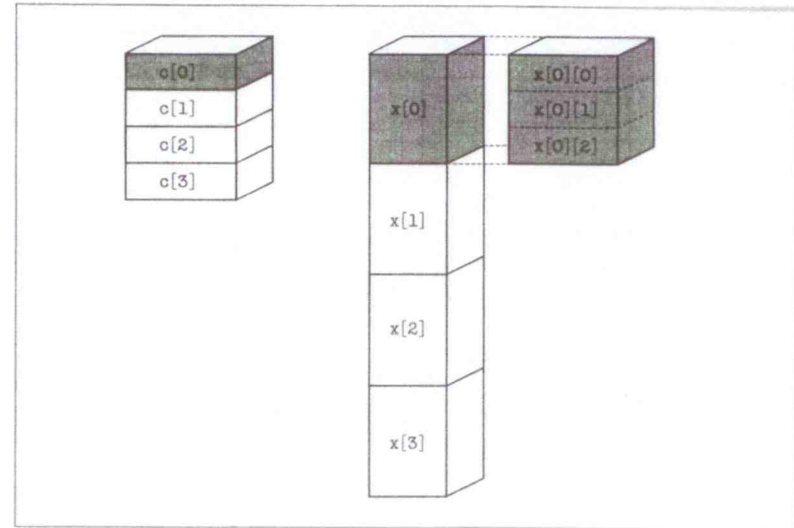
우선 (3)을 생각해 봅시다. int(*a)[3]이라는 표현에서, *a의 둘레에 ()가 필요하다는 것을 주의하십시오. 배열의 포인터이기 때문입니다.

– 어째서 그렇게 까다로운 방법을 써야 하는 것입니까? 잘 이해가 되지 않습니다.

일차원 배열의 경우를 생각해 보십시오. C 언어에서는 직접 <배열>을 전달할 수가 없기 때문에 그 시작 요소의 포인터라는 형으로 실행했습니다.

예를 들면, 일차원 배열 c를 넘겨 주고 싶을 때는 c[0]의 어드레스–즉, c(&c[0]와 같음)–를 넘겨 주었습니다.

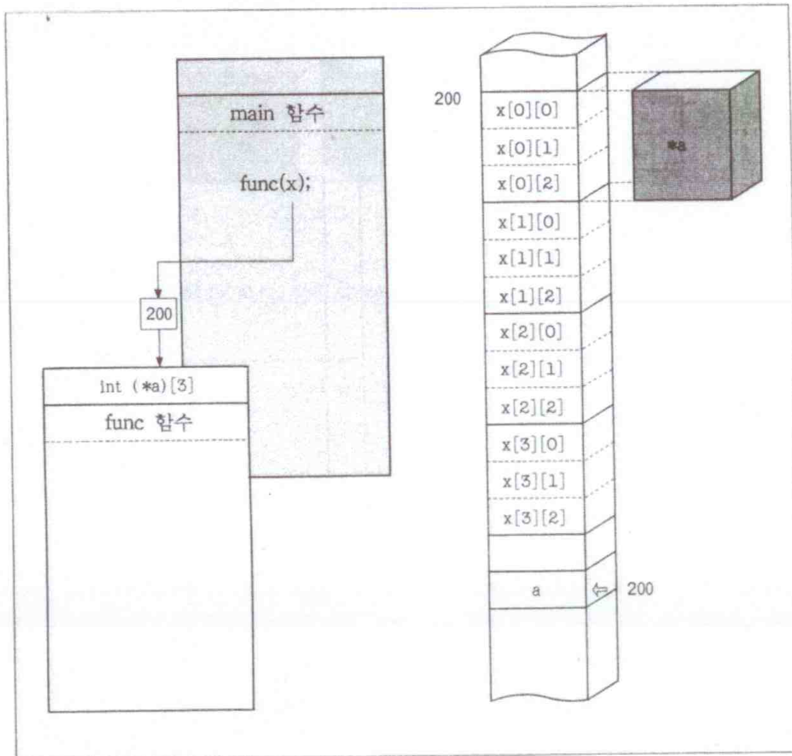
이제 좀 일반적으로 설명하면, 호출하는 쪽은 수형 type의 배열의 시작 요소(c[0]에 해당)의 어드레스를 넘겨 주고, 받는 쪽은 그 값을 수형 type의 포인터로써 받는 것입니다. 이차원 배열의 경우에도 같습니다. 여기에서는 이런 형태의 type이 단지 크기가 3인 int형의 배열이며 수형 type의 포인터를 받을 뿐입니다.



[그림 3-4] 일차원 배열과 이차원 배열

[리스트 3-1] 이차원 배열을 주고 받는 프로그램

```
/*
    List 3-1  이차원 배열을 주고 받는 프로그램
*/

void func(int (*a)[3])
{
    a[2][2] = 3;
}

int main(void)
{
    int x[4][3];

    func(x);                /* func(&x[0])와 같음 */
    return(0);
}
```



[그림 3-5] 값에 의한 2차원 배열의 교환

— 알았습니다. 일차원 배열도 이차원 배열도 모두 「일차원 배열」로 간주해 그 시작 요소, 그림 3-4을 보면 검게 칠 한 부분의 어드레스를 넘겨 준다는 의미군요

그림 3-5를 보면서 한 번 더 복습해 봅시다.

main 함수는

```
func(x);
```

에서 배열의 시작 어드레스 200을 넘겨 줍니다. 여기에서 배열이라는 것은 이차원 배열이 아닌, x[0]부터 x[3]까지의 일차원 배열로 생각합니다. 규칙에 의해 &x[0]은 x라고 쓸 수 있기 때문에 func(x)가 됩니다. &x[0][0]이 아닌 것에 주의해 주십시오. int의 포인터가 아닌 int형의 크기 3인 배열의 포인터를 넘겨 주는 것입니다.


```
void func(int (*a)[3])
```

의 (*a)[3]을 살펴 보기로 하겠습니다. a는 int형의 크기 3의 배열의 포인터가 되고, 받은 값—즉, x[0]의 어드레스—이 복사됩니다. 따라서 a는 x[0]을 가리키는 것이지, x[0][0]을 가리키는 것이 아님에 주의해 주십시오.

— *a는 x[0]의 앨리어스가 됩니까.

그렇습니다. *a는 a[0]이라고도 쓸 수 있습니다. a[0]은 int형의 크기 3인 배열의 포인터이기 때문에 x[0][0] - x[0][2] 의 영역은 *(a[0]+0)~*(a[0]+2)—즉, a[0][0]~a[0][2]—와 같이 액세스 할 수 있게 됩니다.

— 이제, (3)번에 대해서는 알겠습니다. (1)이나 (2)번은 어떻게 된 것입니까?

여기에서도, 일차원 배열의 경우를 생각하십시오. func(int a[])라고 기술한다면, a는 int의 포인터라고 해석됩니다. 보다 일반적으로 말하면 type a[]은, type *a라고 해석됩니다. 따라서 (2)번의

```
void func(int a[][3])
```

에서 type은 int[3]이기 때문에 (3)번의

```
void func(int (*a)[3])
```

과 같은 의미인 것을 알 수 있습니다. 따라서 [3]을 []라고 쓸 수 없습니다. 즉, 3을 생략할 수 없다는 것을 알 수 있습니다.

(1)번에 관해서는 설명할 필요가 없겠지요.

— 잘 알겠습니다.

한 가지 중요한 결론을 말씀드리면 앞에서도 말했듯이 C 언어에는 엄밀한 의미에서 다차원 배열이라는 것은 존재하지 않습니다. 다차원 배열은 <배열의 배열....의 배열>로써 실현됩니다.

그리고 이제까지 배워 왔듯이 다차원 배열에 대한 특별한 사항은 아무 것도 존재하지 않습니다. 프로그램에서 다차원 배열은 단지 배열의 요소가 배열이 되고 있을 뿐이라는 것을 항상 유의하면 그것으로 충분합니다.

☆ 중 요 ☆

C 언어에서는 일차원 배열만이 존재하고 다차원 배열은 배열을 요소로 하는 배열로써 구현되고 있다.

□ 컬럼 3-2 □

다차원 배열의 첨자

위의 내용에서 2차원 배열을 x[i][j]와 같이 나타내지 않으면 안되는 이유를 조금 부자연스럽게(?) 설명하였다.
이것을 반대로 생각해 보자.

```
int x[4][3];
```

선언한 x에 대해 생각해 보면 x[i][j]는 (x[i])[j]이다([] 연산자는 왼쪽으로부터 오른쪽으로 결합하기 때문에). 여기에서 (x[i])는 int형의 크기 3인 배열의 포인터이다. 〈포인터[정수]〉라는 표현은 〈*(포인터)+(정수)〉가 된다. 따라서, x[i][j]는 *(x[i]+j)라고 나타낼 수도 있다. 여기에서 x[i]는 int의 포인터이며 *(x+i)라고 나타낼 수 있다.
결국 x[i][j]는 *(*(x+i)+j)와 같은 의미이다. 이것으로부터도 x는 int의 포인터가 아닌 것과 x가 &x[0][0]이 아닌 &x[0]인 것을 알 수 있다.

▶ 문제 3-2

아래와 같이 정의된 삼차원 배열을 인자로 주고 받는 프로그램을 작성해 보십시오.

```
int a[3][4][5];
```


▷ 3-2 포인터의 배열과 다차원 배열

▷ ····· 3-2-1 포인터의 배열

다음 프로그램은 프로그래밍 언어의 이름으로 된 문자열을 되돌려 주는 함수입니다.

[리스트 3-2] 포인터의 배열

```
/*
    List 3-2 포인터의 배열
*/

char *language_str(int n)
{
    static char *name[3] = {
        "C",
        "Pascal",
        "???",
    };

    return((n < 0 || n > 1) ? name[2]: name[n]);
}
```

□ 컬럼 3-3 □

초기값 나열시의 컴마

리스트 3-2의 프로그램을 잘 살펴보자.

```
static char *name[3] = {
        "C",
        "Pascal",
        "???",
};
```

"???"의 뒤쪽에 ,(컴마)가 붙어있는 것을 알 수 있다.
이 선언을 가로로 나열해 보면 다음과 같다.

```
static char *name[3] = { "C", "Pascal", "???", };
```

이 컴마는 불필요한 느낌이 든다.
처음의 예에서와 같이 세로로 나열했던 경우는 컴마가 있는 편이 균형이 잡혀 자연스럽게 느껴진다. 이와 같은 기술은 K&R에서도 사용되고 있으며 ANSI 규격에서는 정식으로 채택되었다. 즉, 요소의 나열의 경우 마지막 요소의 뒤에 컴마를 붙여도 관계 없다는 것이 명문화 되어 있다.

— n이 0이나 1일 때는 "C", "Pascal"을 되돌려주고, 그 이외일 때는 "???"를 되돌려 주는 것입니까?

그렇습니다. name은 크기 3인 char형 포인터의 배열입니다. 각각의 값은 무엇이 되겠습니까?

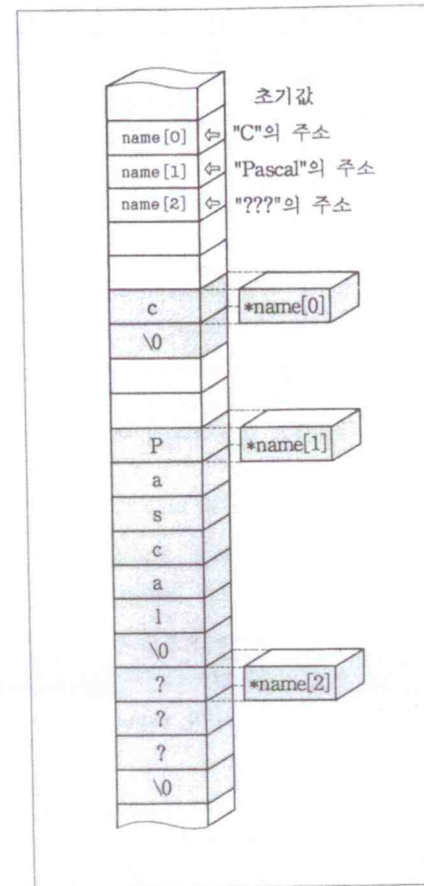
— 문자열 상수는 그 선두 문자의 어드레스를 갖기 때문에, 각각 어딘가에 저장되어 있는 문자열의 선두 어드레스가 되겠지요.

맞습니다. 그림 3-6에 나타냈듯이, name이라고 하는 포인터의 배열을 정의해 name[0]~name[2]의 각각에 "C", "Pascal", "???"이 저장되어 있는 어드레스를 초기값으로 대입한다는 의미입니다.

— "C"와 "Pascal"의 사이가 비어 있는데요.

여기에서 "C", "Pascal", "???" 등은 문자열 상수입니다. 이것들은 메모리상의 어딘가에 저장되어 있지만 연속적으로 저장되어 있다고 하는 보증은 어디에도 없습니다. 간혹 연속하는 경우도 있다고 생각하는 편이 좋습니다.

— 그림 중에 *name[0]이라고 하는 것은 무엇입니까?



[그림 3-6] 포인터의 배열

기본적인 내용을 금방 잊어 버렸군요.
포인터 변수 ptr이 x를 가리킬 때 *ptr은 x의 앨리어스가 된다는 이야기를 했지요.
지금 name[0]은 "C"의 주소를 가지고 있기 때문에 'C'를 가리키고 있습니다.

— 알겠습니다. ptr이 가리키는 실체가 *ptr이므로 여기에서는 ptr 부분이 name[0]이 되는 것이군요?

name[0]~name[2]는 각각 "C", "Pascal", "???"을 가리키고 있기 때문에 *name[0]~*name[2]는 각각 문자열의 선두 문자의 엘리어스가 됩니다.

여기에서 배열과 포인터의 규칙에 관해서 생각해 보십시오. 일반적으로 *ptr은 ptr[0]이라고 쓸 수 있습니다.

```
*ptr  ->  ptr[0]
```

그러므로

```
*name[0]  ->  name[0][0]
```

이 되고, 문자열 "C"의 선두 문자 'C'는 name[0][0]과 같은 방법으로 액세스할 수 있다는 의미입니다.

'C'의 다음 문자 '\0'은 어떻게 되겠습니까?

— 이것은 포인터 변수 ptr이 가리키고 있는 요소의 다음 요소이기 때문에 일반형은,

```
*(ptr + 1)
```

입니다. 그리고 ptr이 nama[0]이기 때문에

```
*(name[0] + 1)
```

이 됩니다.

그렇습니다. 또, *(ptr + i)는 ptr[i]나 i[ptr]으로도 기술될 수 있다고 말했습니다.

따라서 name[0][1]이라고 쓸 수도 있습니다.

마찬가지로,"Pascal"의 각 문자들은 name[1][0], name[1][1],.....라고 하는 식으로 액세스 할 수 있습니다.

— 이차원 배열과 같은 것입니까?

그것과는 다릅니다. 포인터의 배열과 이차원 배열은 전혀 다른 것입니다. 다른 각 요소에 대한 액세스가 많이 닮은 것 뿐입니다.

▶ 문제 3-3

요일을 나타내는 문자열의 포인터를 되돌려 주는 함수

```
char *strweek(int n);
```

를 작성하시오. 단, 일요일을 0, 월요일을 1,.....토요일을 6이라고 한다.

▷ ····· 3-2-2 포인터의 배열과 다차원 배열

그럼 이차원 배열을 사용해서 작성한 다음 프로그램 (리스트 3-3)을 봐 주십시오.

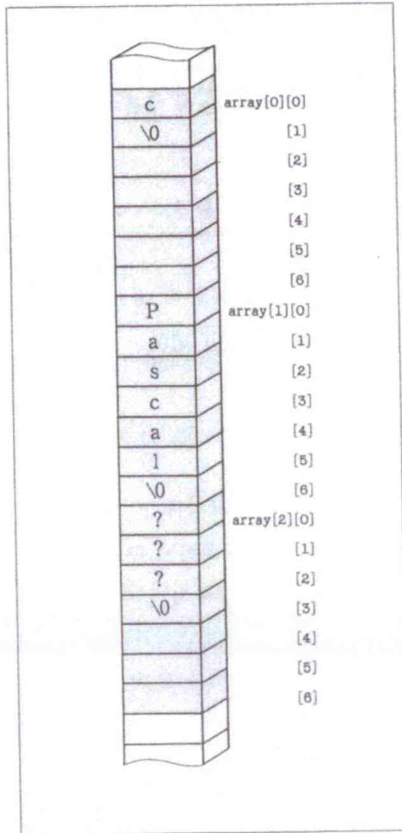
— 이 프로그램은 리스트 3-2와 거의 같군요.

그렇지만, 실은 이것들의 차이를 명확하게 구분해야 합니다. 우선 array는 이차원 배열이기 때문에, 3*7의 고정 영역이 확보됩니다. 즉, 그림 3-7과 같이 됩니다.

[리스트 3-3] 이차원 배열

```
/*
    List 3-3  이차원 배열
*/

char *language_str(int n)
{
    static char array[3][7] = {
        "C",
        "Pascal",
        "???",
    };

    return((n < 0 || n > 1) ? array[2]: array[n]);
}
```

— 당연한 것이지만, 포인터의 경우와는 다르게 이것들은 연속적인 영역을 갖게 되는군요.



[그림 3-7] 이차원 배열

그렇습니다. array는 그 영역의 선두 어드레스를 갖고 이것은 바꿔 쓸 수 없는 상수입니다. 이점도 포인터와는 다릅니다. 또, 포인터의 경우 name[0]~name[2]라는 포인터의 배열이 문자열이 저장되 있는 영역과는 별도로 존재했지만 배열의 경우는 존재하지 않습니다.

– 알겠습니다.

그밖에 또 생각나는 차이점이 없습니까?

– "C"는 '\0'을 포함해 두 문자 밖에 없기 때문에, 뒤에 공백이 생겨서 비효율적입니다.

그렇습니다. 그러면 공통점은 무엇입니까?

– 간단합니다. 각 문자를 array[i][j]라는 같은 표현으로 액세스할 수 있습니다.

2장에서 배열과 포인터는 액세스 방법이 많이 닮았을 뿐 전혀 다른 것이라는 이야기를 했는데 이번의 경우에도 같은 이야기를 할 수 있습니다.

▶ 문제 3-4

문제 3-3과 같은 기능을 갖는 함수

char *strweek2(int n);

를 작성 하시오. 단, 이차원 배열을 사용할 것.

▷ …… 3-2-3 포인터의 배열과 다차원 배열의 중요한 차이점

그러면 이차원 배열

```
static char array[3][7] = {
        "C",
        "Pascal",
        "???",
};
```

을 main 함수에서 정의하고 이것을 함수 print를 써서 출력하도록 변경하겠습니다.

[리스트 3-4] 함수간의 이차원 배열의 주고 받음

```
/*
    List 3-4  함수간의 이차원 배열의 주고 받음
*/

#include <stdio.h>

void print(char (*a)[7], int i)
{
     printf("%s\n", a[i]);
}

int main(void)
{
    static char array[3][7] = {
        "C",
        "Pascal",
        "???",
    };

    print(array, 0);
    print(array, 1);
    return(0);
}
```

■ 실행 결과 ■

```
C
Pascal
```

이차원 배열을 주고 받는 것은 「3-1절 다차원 배열」에서 설명했기 때문에 아래와 같이 되는 것은 알 수 있을 것입니다.

```
void print(char (*a)[7],int)
```

이것을 다음과 같이 기술할 수도 있습니다.

```
void print(char a[][7],int i)
void print(char a[3][7],int i)
```

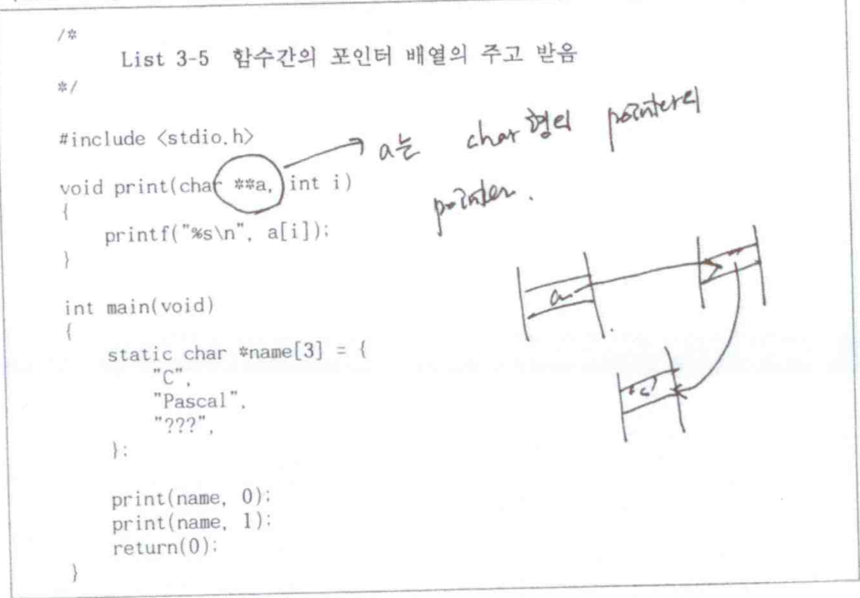
– 알겠습니다. 이차원 배열을 주고 받는 경우는 일차원 배열로 간주해 그 선두 요소의 포인터라는 형으로 다루기 때문이군요.

그러면 포인터의 배열

```
static char *name[3] = {
        "C",
        "Pascal",
        "???",
};
```

을 넘겨주는 프로그램을 작성해 보십시오.

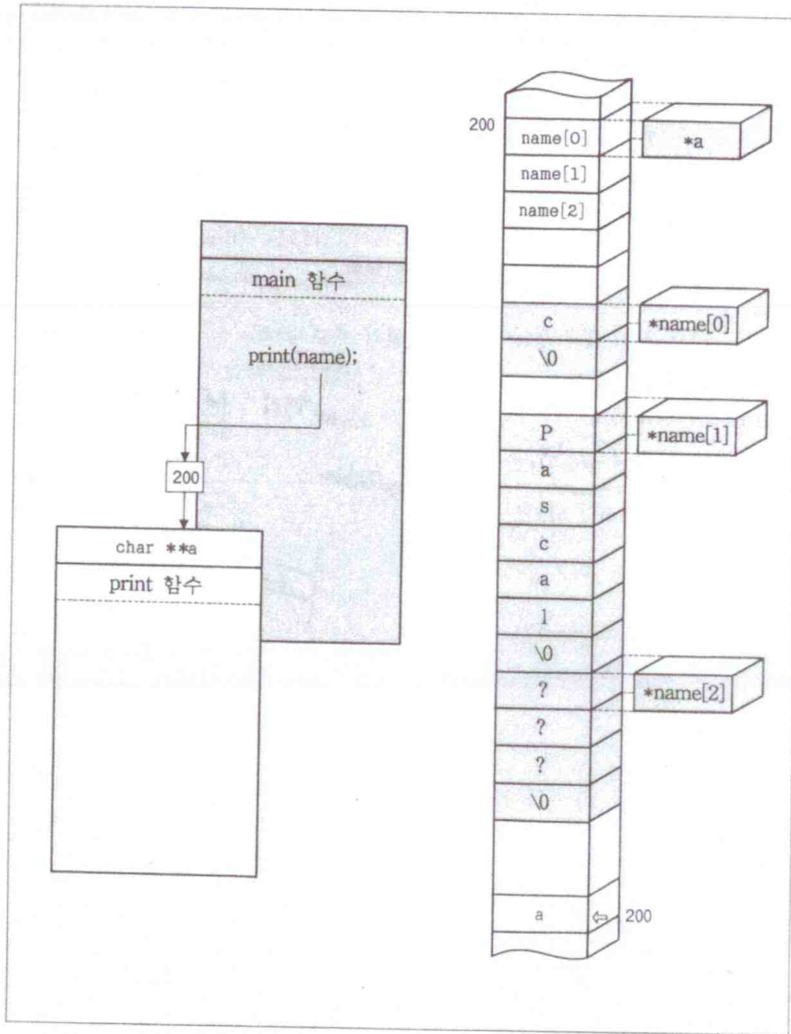
[리스트 3-5] 함수간의 포인터 배열의 주고 받음



```
/*
    List 3-5  함수간의 포인터 배열의 주고 받음
*/

#include <stdio.h>

void print(char **a, int i)
{
    printf("%s\n", a[i]);
}

int main(void)
{
    static char *name[3] = {
        "C",
        "Pascal",
        "???",
    };

    print(name, 0);
    print(name, 1);
    return(0);
}
```

– 저.... 잘 모르겠습니다.

그러면 제가 작성해 보겠습니다. 리스트 3-5와 같이 됩니다.



[그림 3-8] 함수간의 포인터 배열의 주고 받음

— void print(char **a, int i)의 char **a는 낯익은 형태가 아닌데요.

char **a는 a가 char의 포인터의 포인터형 변수라는 선언입니다.

— 포인터의 포인터라고요?

그림 3-8을 보면서 설명하겠습니다.

main 함수는

```
print(name,0);
```

과 같이 실인자로 name을 넘겨줍니다. name은 배열이기 때문에 &name[0]입니다.

이 그림에서는 배열 name[0]~name[2]가 200번지에 저장되어 있기 때문에 200이라는 값이 넘겨지게 됩니다.

— 배열을 주고 받을 때에 선두 요소의 포인터라는 형으로 주고 받는다는 것은 알겠지만 왜 char **가 되는지 이해가 가지 않습니다.

실은 이 규칙에 대해서는 이미 설명했습니다. 「포인터의 배열」이라는 익숙하지 않은 형이기 때문에 잘 모를 뿐입니다.

배열의 선두 요소의 어드레스를 넘겨줄 때 포인터라는 형으로 주고받은 것입니다.

예를들면 배열의 경우

```
int x[10];
```

을 넘겨줄 때, x 즉, &x[0]을 넘겨주며 받는 쪽은 int *a 등으로 선언한다는 뜻입니다.

— 그것은 잘 알지만 .....

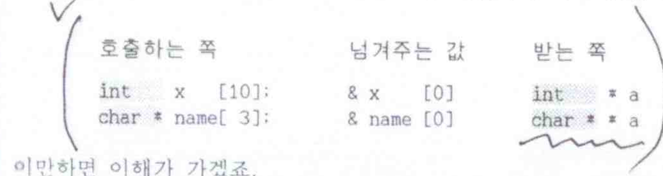
여기에서는

```
char *name[3];
```

이라는 포인터의 배열을 전해주고 싶다는 뜻입니다. 이것은 각 요소가 포인터일 뿐 보통의 배열과 같습니다.

— 단순한 일차원 배열이라는 것이군요.

그렇습니다. x와 name을 주고 받는 것을 나열하여 써 보겠습니다.



호출하는 쪽	넘겨주는 값	받는 쪽
int x [10];	& x [0]	int * a
char * name[3];	& name [0]	char * * a

이만하면 이해가 가겠죠.

— 단지 배열의 선두 요소의 포인터를 교환하고 있을 뿐이며, 그 선두 요소의 형이 int형이 아닌 char의 포인터형으로 되어 있군요. 그리고 char의 포인터의 포인터를 받는다는 의미이기 때문에 char **가 되겠네요.

int형의 배열을 받는 쪽의 선언에는, 다음의 3종류

void func(int a[10])
{
 	
}
(1)배열(크기를 정함)
	void func(int a[])
	{
	
	}
	(2)배열(크기 없음)
		void func(int *a)
		{
		
		}
		(3)포인터

가 있으며 이것들은 모두 같은 의미라는 것은 설명했습니다. 그러면 포인터의 배열 name을 받는 쪽은 어떻게 될까요.

— 이제 할 수 있습니다. 다음과 같이 됩니다.

void func(char *a[3])
{
 	
}
(1)배열(크기를 정함)
	void func(char *a[])
	{
	
	}
	(2)배열(크기 없음)
		void func(char **a)
		{
		
		}
		(3)포인터

간결하고 요령있는 설명이군요. 그러면 받은 쪽인 print 함수로 이야기를 돌리겠습니다. name 즉, &name[0]이 a에 copy됩니다. 따라서 그림 3-9와 같이 *a는 name[0]의 앨리어스가 됩니다. 또, 이것은 a[0]이라고 표현할 수 있습니다. 마찬가지로 a[1]은 name[1]의, a[2]는 name[2]의 앨리어스가 됩니다.

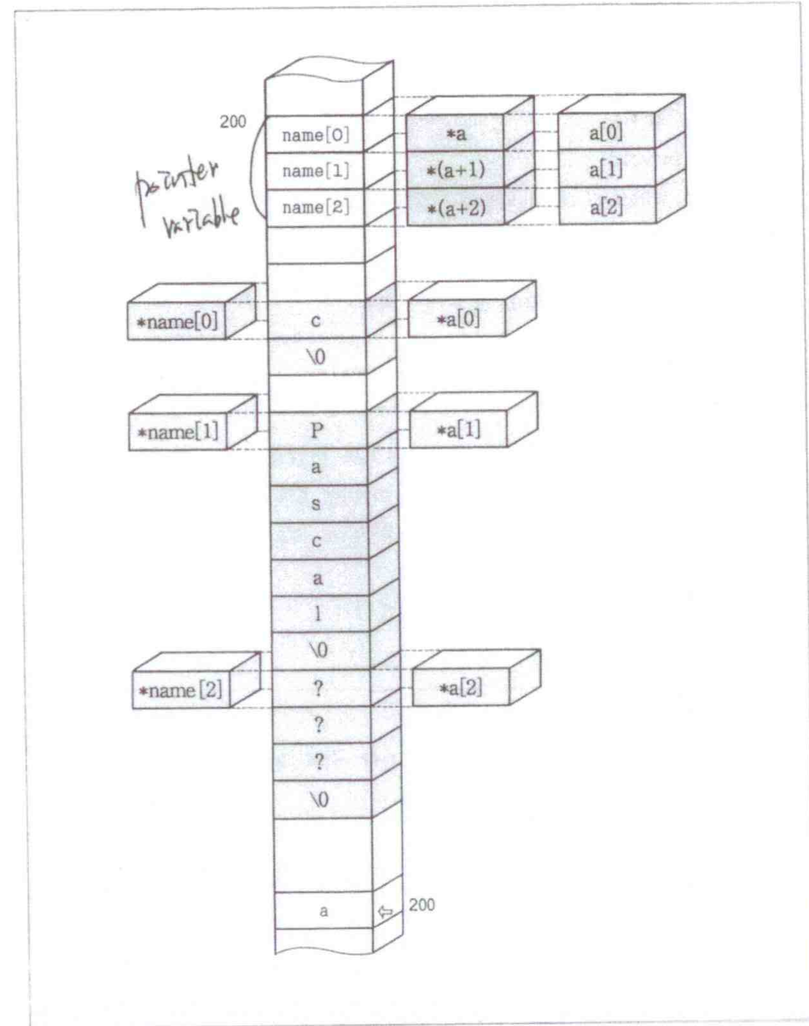
— 넘겨 주는 쪽의 배열과, 받는 쪽의 포인터는 []을 적용함에 따라 똑같이 액세스할 수 있기 때문에 일차원 배열일 때와 같군요.

그래서 제가 일차원 배열과 같다고 말했었죠.

— ......

이제부터 뒤는 설명하지 않아도 알 수 있을 것입니다.

— *a[0]은 *name[0]의 앨리어스가 된다는 의미이기 때문에, name과 a는 동일하게 사용할 수 있군요!



[그림 3-9] 함수간의 포인터 배열의 주고 받음

▷ 3-3 커맨드 라인 인자

▷ …… 3-3-1 argc와 argv

이제까지 main 함수는

```
int main(void)
```

로 기록하였지만 실제로는 main 함수도 인자를 받을 수 있습니다.

리스트 3-6의 프로그램을 봐 주십시오

[리스트 3-6] 커맨드 라인 인자의 표시(첫번째)

```
/*
    List 3-6  커맨드 라인 인자의 표시(첫번째)
*/
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for (i = 0; i < argc; i++)
        printf("%d : %s\n", i, argv[i]);
    return(0);
}
```

프로그램 수행시, 커맨드 라인으로부터 인자를 주고 실행시켜 보십시오. 예를들어 MS-DOS의 경우 다음과 같이 됩니다.

```
A> LIST0306 HELLO ABC
0 : A:\LIST0306.EXE
1 : HELLO
2 : ABC
```

int argc, char *argv[]는 무엇입니까

우선, 명칭의 유래부터 설명하면, argc는 argument count, argv는 argument vector의 약자입니다. 즉, 인자의 갯수와 인자의 배열을 의미합니다.

— argc가, 인자의 갯수라고요?

네, 그렇습니다. 실행 결과로도 알 수 있듯이 프로그램 자신의 이름—여기에서는 "A:\LIST0306.EXE"입니다—도 인자에 포함됩니다. 그리고 argc는 인자의 갯수를 나타내고 있습니다.

— 인자 선언의 배열 []은 포인터와 같은 의미이기 때문에 이것은

```
int main(int argc, char **argv)
```

와 같은 것이 되는군요.

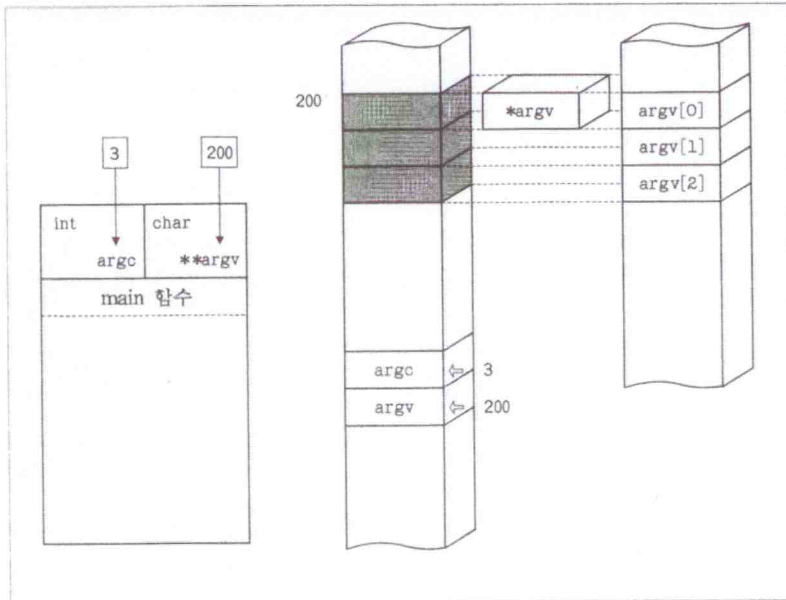
그렇습니다. 포인터의 포인터입니다.

— 네? 또 뭐가 뭔지 잘 모르겠습니다.

argc는 고사하고 argv의 정확한 의미도 알지 못한 채 아무 생각없이 사용하고 있는 프로그래머도 많은 것 같습니다. 여기에서 정확히 설명해 보기로 합시다.

먼저 질문을 하나 하겠습니다. argv[0]과 argv[1]는 무엇일까요?

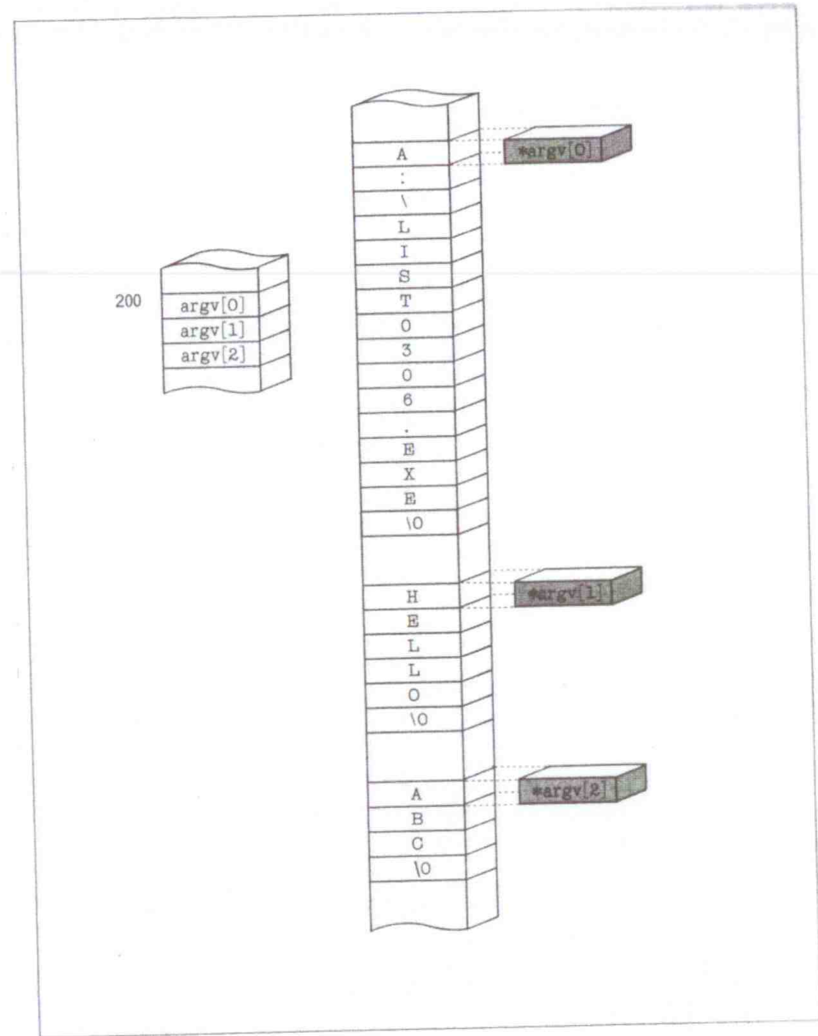
— 잘 모르겠습니다.



[그림 3-10] 커맨드 라인 인자의 전달

포인터의 포인터에 관해서 앞에서 설명했는데 잘 모른다고 하면 안되지요.

이 경우, 인자는 3개입니다. 따라서 그림 3-10의 검게 칠해진 부분과 같이 크기 3의 char 포인터 배열이 자동으로 만들어집니다. 예를들어 이것이 200번지에 저장되어 있다고 하면 형식 인자인 argc에는 3, argv에는 200이라는 초기값이 주어지게 됩니다.



[그림 3-11] 커맨드 라인 인자를 가리키는 포인터

― 아, 알아서 배열이 만들어진다는 뜻이군요.

맞습니다. *argv는 200번지의 포인터 변수의 앨리어스가 되는 것은 알겠지요.

— 포인터 변수 argv가 가리키고 있는 실체는 *argv로써 나타낼 수 있겠네요.

그렇습니다. 따라서 포인터 변수의 배열은 각각 *(argv), *(argv+1), *(argv+2)로 나타낼 수 있으며, 이것은 [] 연산자를 사용하여 argv[0], argv[1], argv[2]로 나타낼 수도 있습니다.

알겠읍니까?

— 네.

argv[0]~argv[2] 각각은 포인터 변수입니다.

이것은 각각 문자열 "A:\LIST0306.EXE", "HELLO", "ABC"를 가리키도록 초기값이 주어져 있습니다. 이것을 그림으로 나타내면 그림 3-11과 같이 됩니다.

따라서, *argv[0]~*argv[2]는 각 문자열의 선두 문자에 대한 앨리어스가 됩니다.

— 앞에서 포인터의 배열과 같다고 한 의미를 이제야 겨우 알겠습니다.

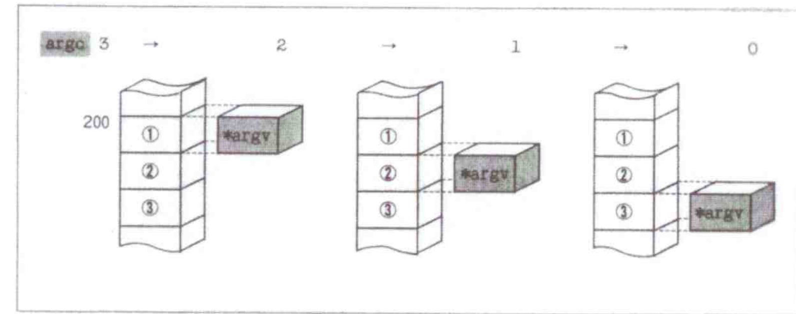
argv는 포인터의 포인터입니다. 배열을 주고 받을 때 「포인터를 받는 경우 한 개의 객체를 가리키는 것으로 취급하든, 배열처럼 취급하든 어느 쪽이든 괜찮다」라는 이야기를 했습니다. 리스트 3-6의 프로그램은 배열로 사용한 경우입니다. 문자열의 복사에서 했던 것처럼 argv를 포인터 답게 사용하여 다시 작성한 것이 리스트 3-7입니다.

[리스트 3-7] 커맨드 라인 인자의 표시(두번째)

```
/*
    List 3-7  커맨드 라인 인자의 표시(두번째)
*/
#include <stdio.h>

int main(int argc, char **argv)
{
    int    i = 0;

    while (argc--)
        printf("%d : %s\n", i++, *argv++);
    return(0);
}
```

이 프로그램에서는 루프를 반복할 때마다 argc를 감소시킵니다.

그림 3-12에서와 같이 argv는 하나씩 증가됩니다.



[그림 3-12] 커맨드 라인 인자를 가리키는 포인터

아래의 화면에 출력하는 부분을 보면

```
printf("%d : %s\n", i++, *argv++);
```

*argv가 출력됩니다. while문의 각 루프를 단계별로 살펴보면 표 3-1과 같습니다.

각루프에서의 argv와 *argv

	argv	*argv
1회째	1을 가리키는 포인터	1의 앨리어스 즉, "A:\LIST0306.EXE"를 가리키는 포인터
2회째	2를 가리키는 포인터	2의 앨리어스 즉, "HELLO"를 가리키는 포인터
3회째	3을 가리키는 포인터	3의 앨리어스 즉, "ABC"를 가리키는 포인터

□ 컬럼 3-4 □

커맨드 라인 인자와 실행 환경

MS-DOS Ver3.0 이전의 버전에서는 프로그램이 자기 자신의 이름을 아는 기능을 갖고 있지 않았다.
따라서 리스트 3-6의 프로그램을 실행한 경우 실행 결과는 다음과 같이 된다.

```
A> LIST0306 HELLO ABC
0 :
1 : HELLO
2 : ABC
```

ANSI 에서는 위에서 처럼 실행 환경으로부터 프로그램 이름이 주어지지 않는 경우 argv[0]은 널 문자를 가리키게 된다. 즉, argv[0][0]이 '\0'이 된다.

▶ 문제 3-5

리스트 3-7 프로그램은 프로그램 자신의 이름도 표시한다. 프로그램 자신의 이름은 표시하지 않고, 인자만을 표시하도록 프로그램을 작성하시오.

리스트 3-7의 프로그램에서는 printf 함수를 사용하기 때문에 문자열의 선두 문자의 포인터인 *argv를 넘겨주어 출력을 했습니다.

다음에는 포인터에 대한 이해를 보다 깊게 하기 위해 putchar 함수를 사용해 봅시다.

– 문자열을 한 번에 표시하지 않고 한 문자씩 출력한다는 의미입니까?

그렇습니다 프로그램은 리스트 3-8과 같이 됩니다.

[리스트 3-8] 커맨드 라인의 인자의 표시(세번째)

```
/*
    List 3-8  커맨드 라인의 인자 표시(세번째)
*/
#include <stdio.h>

int main(int argc, char **argv)
{
    int  i = 0;
    char c;

    while (argc--) {
        printf("%d : ", i++);
        while (c = *(*argv)++)
            putchar(c);
        argv++;
        putchar('\n');
    }
    return(0);
}
```

– 지쳤어요. 또 자신이 없어졌습니다.

바깥쪽의 while 루프에 관해서는 리스트 3-7의 프로그램과 같기 때문에 설명할 필요는 없겠지요. printf 함수의 호출 부분도 마찬가지입니다. 문제는 이부분이군요.

```
while (c = *(*argv)++)
    putchar(c);
argv++;
```

– 그렇습니다. 마지막부분의

```
argv++;
```

는 앞의 설명으로 알 수 있습니다.

그러면 안쪽의 while 루프 부분을 쪼개어 알기 쉽게 써 봅시다.

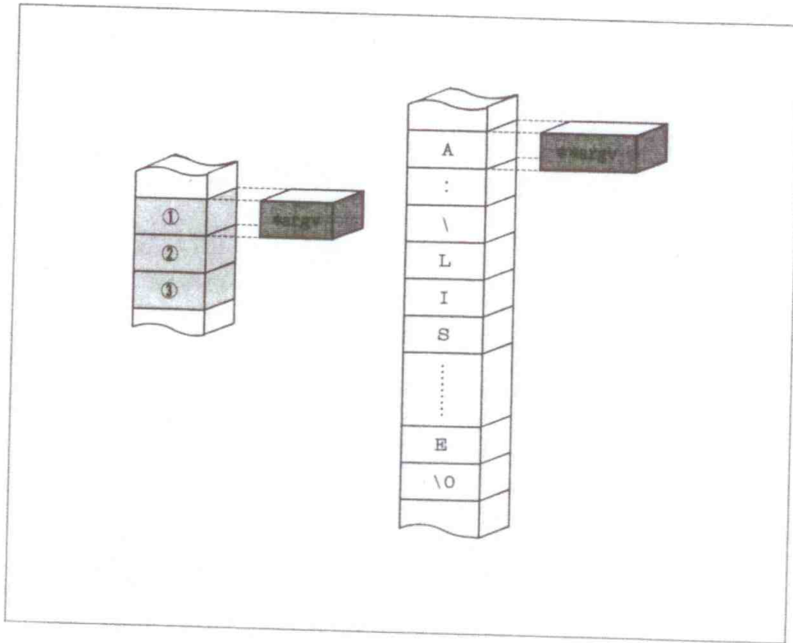
```
while ( (c = **argv) !="\0") {
   putchar(c);
   ++*argv;
}
```

이 프로그램을 가지고 생각해 보도록 하겠습니다. 이 프로그램의 초기 상태가 그림

3-13과 같다는 것은 알겠지요. 좀더 확실하게 정리해 보면

argv : 문자열 "A:\LIST0308.EXE 의 최초의 문자 'A'의 포인터의 포인터
*argv : 문자열 "A:\LIST0308.EXE"의 최초의 문자 'A'의 포인터
**argv : 문자열 "A:\LIST0308.EXE"의 최초의 문자 'A'

가 됩니다.



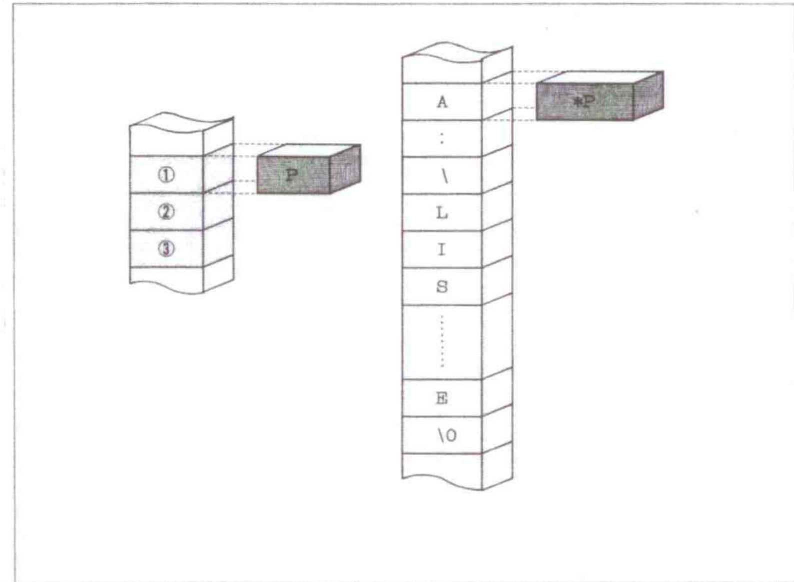
[그림 3-13] argv의 초기 상태

– 그렇게 되는 것은 알 것 같습니다만 왠지 좀.....

그러면 아래와 같이 해 봅시다. argv는 포인터의 포인터이기 때문에 혼동하기 쉽습니다. 그러나 *argv는 단순히 문자의 포인터 입니다. 결국 char *형인 것입니다.
따라서, *argv를 P라고 하면 앞의 프로그램 부분은 다음과 같이 됩니다.

```
while ( (c = *P) != '\0'  {
    putchar(c);
    ++P;
}
```


또, 그림 3-13을 P를 사용해서 다시 그려보면 그림 3-14와 같이 됩니다.

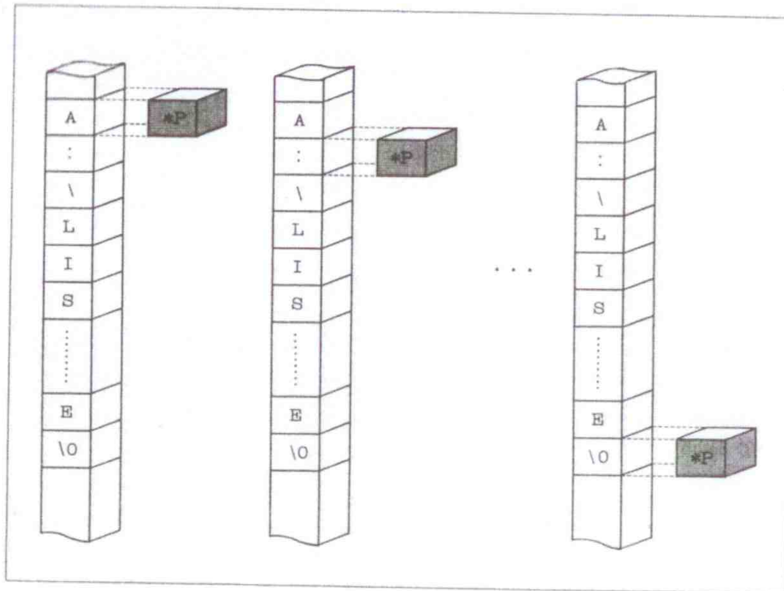


[그림 3-14] *argv를 P로 바꾼 경우 argv의 초기 상태

– 그림 3-14는 단순한 문자열의 설명도와 같은 것인가요?

그렇습니다. 프로그램도 문자열의 길이를 구하는 프로그램과 거의 똑같이 됩니다(리스트 2-14 참조).

– P가 가리키고 있는 문자가 '\0'이 되기 전까지 문자를 출력하고 포인터를 증가시키는군요. 그림 3-15와 같이 되겠네요.



[그림 3-15] P의 증가

그렇습니다. 더이상의 설명은 필요없다고 생각됩니다. 언뜻 보아 이해하기 어려운 프로그램은 그 내용을 적당히 바꿔 놓음에 따라 간단하게 이해할 수 있다는 것을 알 수 있습니다. 단, 이런 경우 연산자의 우선 순위 등에 주의를 기울이지 않으면 안됩니다.

☆ 교 훈 ☆

포인터의 포인터 등, 언뜻 보아 어려운(복잡한) 프로그램을 해독할 때에는 적당히 바꿔 놓으면 좋다. 단, 연산자의 우선 순위 등에는 주의를 기울일 것.

▶ 문제 3-6

지정된 화일의 탭(\t)을 다음의 탭 스톱까지 적당한 수의 공백 문자(' ')로 바꾸어 화면에 출력하는 프로그램(detab)을 작성하시오.

커맨드 라인으로부터의 인자로,

(1) -n ····· 행번호를 표시
(2) -t* ····· 탭 스톱을 *문자마다 위치하게 한다(*은 상수)
(3) 화일명을 나타내는 문자열

등을 정할 수 있도록 하시오. 단 (1),(2)는 생략이 가능하다. (1)을 생략한 경우는 행번호를 표시하지 말고 (2)를 생략한 경우는 탭 스톱을 8문자마다 설정하시오.

인자 (1)(2)(3)의 순서는 관계 없도록 한다.

프로그램은 포인터에 대해서 []연산자를 사용해 포인터를 배열처럼 사용하는 방법을 사용하지 않고 작성할 것.

<예>

(1) detab -t2 ABC.DAT
(2) detab -n ABC.DAT -t4

ABC.DAT
```
ABC\tDEF
1234\t56
\tabz
```

(1)의 실행 결과
```
ABC DEF
1234  56
  abz
```

(2)의 실행 결과
```
1: ABC DEF
2: 1234    56
3:     abz
```

제 4 장

메모리의 동적 할당

▷ 4-1 메모리 할당의 기본
▷ 4-2 배열의 할당
▷ 4-3 리스트 처리의 응용

C 언어에서는 필요할 때에만 변수나 배열 등의 영역을 할당 받고 불필요하다면 해제할 수 있습니다. 이 기능을 사용함에 따라, 미리 여분으로 배열을 할당해야 하는 효율적이지 못한 프로그램을 작성하지 않아도 됩니다.

메모리의 동적 할당과 해제 기능은 라이브러리로 구현되어 있고 C 언어의 일부는 아닙니다. 이 기능들을 이용하는 경우에는 반드시 포인터를 사용합니다.

이제까지 배운 포인터에 관한 사항을 완전히 이해한다면 이장을 공부하는 것은 간단하다고 생각됩니다.

정확히 이해해서 효율적인 프로그램을 작성할 수 있도록 합시다.

▷ 4-1 메모리 할당의 기본

▷ ····· 4-1-1 calloc : 메모리의 할당

C 언어에서는 필요한 경우에 한해 변수를 위한 기억 장소를 할당하고 필요하지 않게 되면 해제할 수 있습니다.

— 굉장하군요.

중요한 것은 이 기능들이 C 언어 자체에서 제공되는 것이 아니라 라이브러리에서 제공되고 있다는 것입니다. calloc, malloc 등의 함수를 사용하여 동적으로 기억 장소를 할당할 수 있습니다.

먼저 calloc 함수를 공부합시다. calloc 함수의 개요는 다음과 같습니다.

calloc

형식	:	void *calloc(size_t n , size_t size);
프로토타입	:	<stdlib.h>
기능	:	calloc는 크기가 size인 n개의 객체(기억 장소)의 영역을 할당한다. 그 영역은 모두 0으로 초기화 된다.
리턴값	:	할당이 성공했을 경우에는 할당된 영역의 선두 포인터를 되돌려 준다. 할당에 실패한 경우는 NULL을 되돌려 준다.

— 어렵습니다.

실은 그렇게 어렵지는 않습니다. 지금까지 배운 포인터를 이해하고 있다면 의외로 간단하게 알 수 있습니다. size_t형은 알고 있겠지요.

— 문자열의 길이를 구할 때에 배웠습니다.

정확히 알고 있는 듯 하군요. 그렇다면 정수를 1개 할당하는 프로그램을 작성하겠습니다. 리스트 4-1이 그것입니다.

□ 컬럼 4-1 □

동적 메모리 할당의 헤더 화일

ANSI 규격에서, calloc, malloc 등의 함수의 프로토타입 선언은 <alloc.h> 에서 제공하도록 요구하고 있다. 규격이 결정되기 전에는 각 컴파일러에서 헤더 화일을 준비하고 있었기 때문에, <malloc.h>나 <alloc.h> 등에서 선언 되었다.

현재는 ANSI와 규격에 맞추기 위해 <stdlib.h>를 선언하고, 이전의 컴파일러와 호환성을 유지하기 위해 <malloc.h>나 <alloc.h> 등에서, 양쪽으로 선언하는 경우가 대부분이다.

만약, ANSI 규격에 따른 호환성이 높은 프로그램을 기술한다면 <stdlib.h>를 사용해야 하며 <malloc.h>나 <alloc.h> 등을 사용해서는 안된다.

[리스트 4-1] 정수를 한 개 할당하는 프로그램

```
/*
    List 4-1 정수를 한 개 할당하는 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p;

    p = (int *)calloc(1 , sizeof(int));
    if (p == NULL)
            printf("not allocated");
    return(0);
}
```

— sizeof(int)란 무엇입니까.

sizeof(수형명)라고 쓰면 그 형의 크기를 돌려 줍니다. sizeof(char)는 반드시 1이 됩니다.

— 제가 사용하고 있는 컴파일러는 int형의 크기가 2바이트이기 때문에 sizeof(int) 대신에 2라고 써도 좋습니까.

물론 틀린 것은 아니지만 2라고 써버리면 int형이 4바이트인 컴파일러에서는 올바르게 동작하지 않게 됩니다. 프로그램의 호환성을 높이려면 반드시 sizeof 연산자를 사용해야 합니다.

□ 컬럼 4-2 □

두 개의 sizeof

제2장에서 설명했던 sizeof는

sizeof 식

의 형식이었다. 여기에서의 sizeof는

sizeof(수형명)

이다. 후자는 반드시 ()가 필요하다. 전자에서는 ()가 필요하지 않지만 제2장에서도 설명했듯이 ()를 붙이는 습관을 들이도록 하자.

— 알겠습니다. 리스트 4-1 프로그램의 의미를 설명해 주십시오.

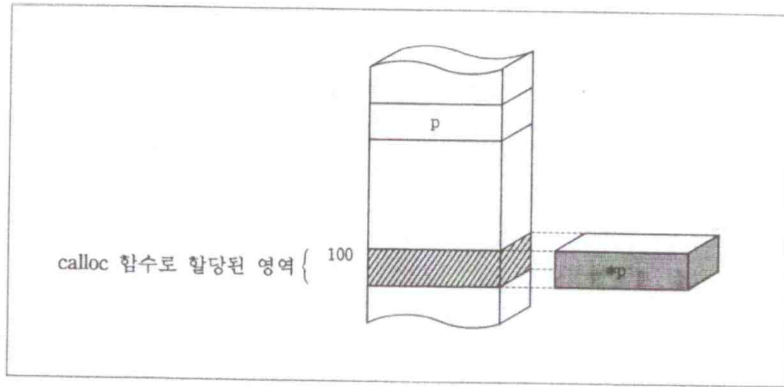
메모리의 동적 할당을 행하는 부분은

```
p = (int *)calloc(1, sizeof(int));
```

입니다. 여기에서는 int형의 크기 1개분의 기억 장소를 할당하고 그 어드레스를 돌려줍니다. 예를들면 int형이 2바이트면 2바이트의 기억 장소가 할당됩니다. 그림 4-1을 봐 주십시오. 여기에서는 100~101번지에 할당되었다고 하겠습니다. 이경우 calloc 함수는 100을 돌려주고 그것이 p에 대입되기 때문에 p는 할당된 영역—그림의 사선 부분—을 가리킵니다.

— p가 그 장소를 가리키고 있기 때문에 *p가 그 앨리어스가 되는 것입니까?

그렇습니다. 결국 100~101번지는 *p로 액세스 할 수 있게 됩니다.



[그림4-1] 메모리의 동적 할당

— 잘 생각하면 할당한 기억 장소의 선두 어드레스를 포인터 변수에 대입하고 있을 뿐이군요.

그렇습니다. calloc 함수는 지정된 크기의 기억 장소를 기계적으로 할당하고 그 선두 어드레스를 되돌려 줍니다.

프로그램 중에는 마치 *p라고 하는 변수가 있는 것처럼 사용할 수 있습니다. 예를 들면

```
*p = 1;
printf("%d", *p);
```

등과 같은 사용이 가능하게 됩니다.

— calloc 함수의 되돌려주는 형은 void *라고 되어 있는데 어떤 의미입니까?

임의의 수형 T의 포인터는 정보를 잃지 않은 채로 void *형으로 변환될 수 있으며 그 반대도 가능합니다. calloc 함수는 할당한 영역의 선두 어드레스를 가리키는 포인터를 되돌려 주지만 이것이 calloc의 포인터인지 int의 포인터인지는 미리 알 수 없습니다. 따라서 void *형은 [만능] 포인터로 사용합니다.

— void *를 int의 포인터형에 대입하기 때문에 (int*)로 캐스트하고 있다는 의미군요

그렇습니다. 그러나, void *은 임의의 수형 T에 정보가 손상되지 않은 채 대입되기 때문에 (int *)에 의한 캐스트가 반드시 필요한 것은 아닙니다.

□ 컬럼 4-3 □

void *

K&R의 시절에는 void *형을 만들지 않았기 때문에(void *은 ANSI에서 만들어져 C++에 도입되었음) char *형이 이와같은 만능 포인터로써 사용되었다.
따라서, calloc 함수가 되돌려 주는 값은 char *형이며, 다음과 같이

```
p = (int *)calloc(1, sizeof(int));
```

명시적으로 캐스트 할 필요가 있었다. 그러나 본문에서도 설명하고 있듯이 ANSI라면 캐스트는 필요하지 않다.

▷ ····· 4-1-2 free : 메모리의 해제

그런데, 자신이 할당한 기억 장소는 책임지고 해제시키지 않으면 안됩니다.

— 그냥 두면 좋지 않기 때문에 되돌려 주는 것이군요. 기억 장소의 해제는 어떻게 하면 됩니까?

해제를 하려면 함수를 사용합니다. free 함수의 개요는 다음과 같습니다.

free		
형 식	:	void free(void *p);
프로토타입	:	<stdlib.h>
기 능	:	free는 p가 가리키는 기억 장소를 해제한다. 단 p가 NULL일 때는 아무것도 하지 않는다. p는 calloc, malloc 혹은 realloc에 의하여 이전에 할당되었던 기억 장소의 포인터이어야 한다.

리스트 4-1의 프로그램에서 해제 하는 기능을 첨가해 보십시오.

— 했습니다. 리스트 4-2와 같이 됩니다.

[리스트 4-2] 정수를 1개 할당, 해제하는 프로그램

```
/*
    List 4-2  정수를 1개 할당, 해제하는 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int  *p;

    p = (int *)calloc(1, sizeof(int));
    if (p== NULL)
        printf("not allocated");
    else {
        /* 하고싶은 처리 */
        free(p);
    }
    return(0);
}
```

맞습니다. 리스트 4-3의 프로그램을 봐 주십시오.
이 프로그램의 잘못된 점을 알겠습니까?

[리스트 4-3] 정수를 할당하는 프로그램

```
/*
    List 4-3 정수를 할당하는 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

void func(void)
{
    int *p;

    p = (int *)calloc(1, sizeof(int));
    if (p == NULL)
       printf("not allocated");
}
```



```
int main(void)
{

    int i;

    for (i = 0; i < 100 ; i++)
        func();
    return(0);
}
```

― func 함수는 호출되는 때에 int형의 영역을 할당하지만 해제를 하지 않았기 때문에 메모리가 점점 줄어드는 점이 잘못된 것이 아닐까요?

그렇습니다. 이와 관련되어 중요한 것이 하나 더 있습니다. free 함수에는 calloc 함수 등에서 할당한 어드레스를 넘겨줘야 합니다. 이 어드레스는 func 함수 내의 p라고 하는 포인터 변수에 저장됩니다. 그러나 이 변수의 값은 func 함수를 벗어나면 자동으로 소멸해 버립니다. 따라서 영역을 해제하는 free를 행할 수가 없습니다.

리스트 4-4와 같이 free 함수를 반드시 써야 합니다.

[리스트 4-4] 정수를 할당, 해제하는 프로그램

```
/*
    List 4-4  정수를 할당, 해제하는 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

void func(void)
{
    int *p;

    p = (int *)calloc(1,sizeof(int));
    if (p == NULL)
        printf("not allocated");
    free(p);
}

int main(void)
{
    int i;

    for (i = 0; i < 100; i++)
```

```
        func();
    return(0);
}
```

√교 훈 ☆

동적으로 할당한 기억 장소 영역은 책임지고 해제하지 않으면 안된다.
영역을 해제할 때까지는 반드시 포인터를 기억해 두어야 한다.

그런데 calloc 등의 함수를 호출하면 자신이 알지 못하는 곳에 기억 장소를 할당할 수 있는데 일반적으로 이 공간을 히프 영역이라고 합니다.

덧붙여서 말하면 히프라고 하는 말은 일반적으로 사용되고는 있지만 ANSI 규격에서는 사용하지 않으므로 주의하십시오.

calloc 함수 이외의 메모리 할당의 함수로 malloc 함수가 있습니다. malloc 함수의 개요는 다음과 같습니다.

malloc		
형 식	:	void *malloc(size_t size);
프로토타입	:	<stdlib.h>
기 능	:	malloc은 크기가 size인 객체(기억 장소)를 할당한다. calloc 함수와는 다르며 할당하는 기억 장소를 초기화하지 않는다.
리 턴 값	:	영역 할당이 성공한 때는 확보된 영역의 선두 포인터를 되돌려 준다. 영역 할당에 실패한 경우는 NULL을 되돌려 준다.

— calloc 함수와는 다르게 단순히 크기만을 지정하는 것입니까?

그렇습니다. 좀더 큰 차이는 calloc 함수는 할당한 영역의 모든 비트를 0으로 채우지만 malloc 함수는 그렇지 않습니다.

— 각각 어떤 식으로 사용하면 좋습니까?

제 경우에는 malloc 함수는 순수하게 [메모리 할당]을 하고 싶을 때에만 사용합니다. 구체적으로 비교적 낮은 레벨의 프로그램을 작성할 때나 문자열을 위한 영역을 할당하는 때에 한합니다.

그외에는 모두 calloc 함수를 사용합니다. 정렬(alignment) 기능 때문에 배열이나 구조체의 요소들간에 생기는 「구멍」 부분의 모든 비트가 0으로 세트되기 때문입니다.

여기에서는 이것들에 관해서 자세하게 설명하지는 않겠습니다.

▶ 문제 4-1

char형, int형, long형, double형의 변수를 동적으로 할당, 해제하는 프로그램을 작성하시오.

▷ 4-2 배열의 할당

▷ ····· 4-2-1 일차원 배열의 할당

이번에는 int형의 8개의 영역을 할당해 보겠습니다. 리스트 4-5의 프로그램은 리스트 4-2 프로그램의 calloc 함수의 첫번째 인자를 8로 바꾸었을 뿐입니다.

[리스트 4-5] 정수를 8개 할당 해제하는 프로그램

```
/*
    List 4-5  정수를 8개 할당, 해제하는 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p;

    p = (int *)calloc(8, sizeof(int));
    if (p == NULL)
        printf("not allocated");
    else {
        /* 하고싶은 처리 */
        free(p);
    }
    return(0);
}
```

― 이것만 변경하면 되는 것인가요?

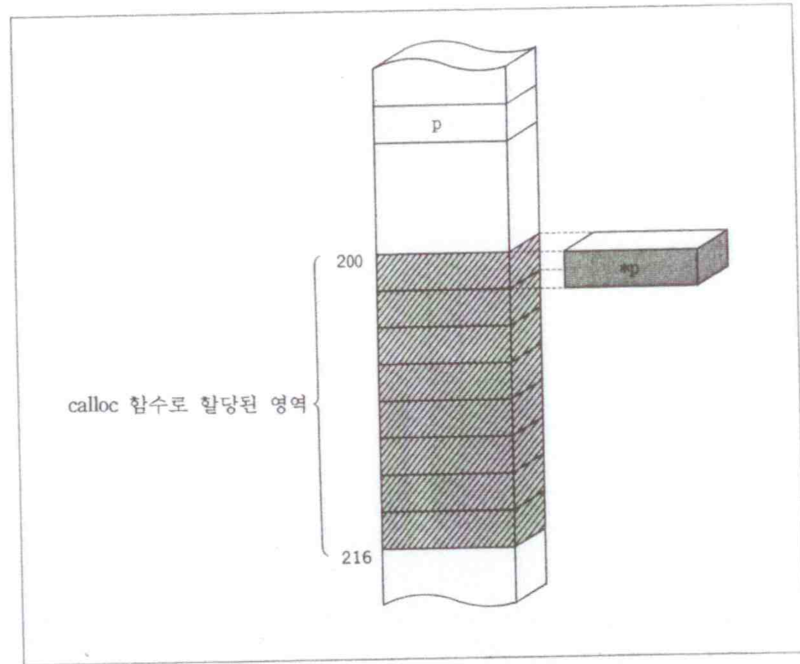
할당을 행하는 다음 문장에서

```
p = (int *)calloc(8 , sizeof(int));
```

calloc은 8개의 int를 위한 기억 장소를 할당해 그 선두 어드레스를 되돌려 줍니다. 앞에서와 마찬가지로 int형이 2바이트이므로 16바이트의 기억 장소가 할당됩니다. 가령 200~215번지의 기억 장소가 할당되었다고 하면 calloc은 200을 되돌려 주며 그것이 p에 대입됩니다.

― 할당된 영역의 선두 2바이트를 *p로써 액세스할 수 있다는 것은 알겠습니다만 그 뒷부분은 어떻게 합니까?

군은 항상 기본적인 것을 곧잘 잊어버리는군요. 포인터 변수 p에 관해서 p가 가리키는 요소의 i개 뒤의 요소는 *(p+i)라는 표현으로 액세스할 수 있고 이것은 p[i]라고도 표현할 수 있다는 이야기를 했었지요.



[그림4-2] 메모리의 동적 할당

― 맞아요. 포인터 변수가 1개 있으면 그 포인터가 가리키고 있는 영역을 배열처럼 액세스할 수 있었죠. 그러면 그림 4-2는 그림 4-3과 같이 됩니까?

그렇습니다.

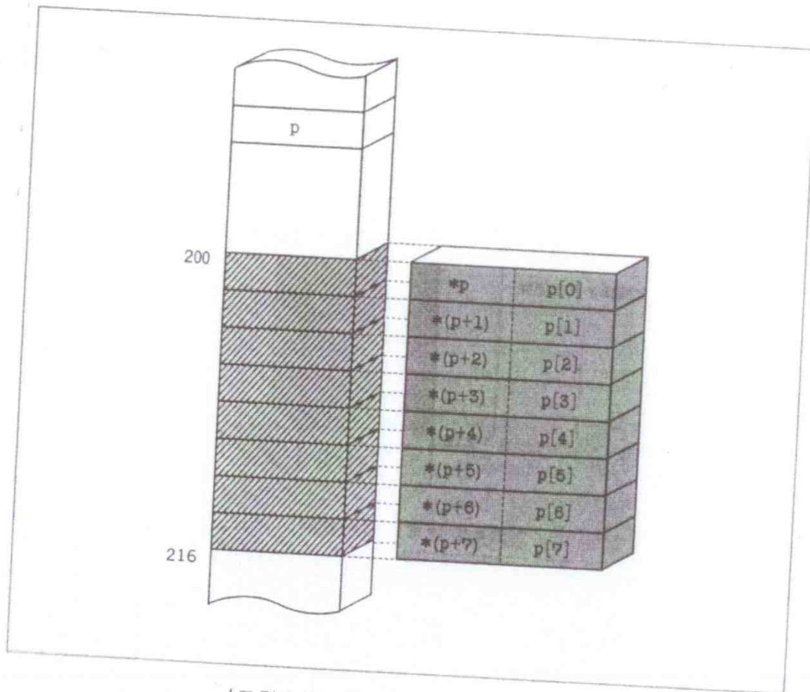
― 프로그램 중에
/* 하고 싶은 처리 */

라고 쓰여있는 곳에서는 마치 p[0]~p[7]이라는 배열이 있는 것처럼 사용할 수 있군요.

그렇습니다 단, calloc 함수는 단순히 지정된 크기의 기억 장소를 물리적으로 할당할 뿐입니다. 따라서 [1개의 변수]를 할당한다든지 [배열]을 할당한다는 의미는 아닙니다. 사용자 입장에서는 할당된 기억 장소의 포인터를 사용함에 따라－게다가 []연산자를 사용함에 따라－ 마치 배열처럼 보일 뿐입니다.

－ 알았습니다. 포인터의 기본만 알면 간단하게 알 수 있는 내용이군요.

C 언어에서는 고정 크기의 배열만을 정의할 수 있으나 calloc 함수를 능숙하게 사용할 수 있다면 가변 크기의 배열(닮은 것)을 작성할 수 있습니다. 잘 생각하면 알 수 있듯이 p자체는 포인터이며, 엄밀히 말해서 배열이 아님을 알 수 있습니다.



[그림4-3] 할당된 메모리의 해석

컬럼 4-4

배열과 수명

객체에는 기억 수명이 있어 그 객체의 생존 기간을 결정한다.
기억 수명에는 정적(static)과 자동(automatic)의 두 가지가 있다.

□ 정적 기억 수명
정적 기억 수명을 갖는 객체는, 프로그램 전체의 실행중에 계속 존재한다.

□ 자동 기억 수명을 갖는 객체는 선언된 블럭 내에서만 유효하다.
예를 들면

```
void function(void)
{
    int        a[10];
    static int b[10]
    ...
}

int main(void)
{
    ...
    function();
    ...
}
```

에서 배열 a는 함수 function이 호출된 동안 존재하지만 배열 b는 프로그램 실행이 시작되었을 때부터 끝날 때까지 존재하며 실행중에 항상 값이 유지된다.

a와 b는 기억 클래스가 다르지만 함수 function 내에서는 다음과 같이

```
a[2] = 0;
b[3] = 5;
```

똑같은 방법으로 취급된다.

calloc 함수에서 동적으로 할당된 객체는 동적 기억 수명을 갖는다고 말한다.

```
void functione(void)
{
    int           a[10];
    static int    b[10];
    int           *c;

    c = (int *)calloc(10, sizeof(int));

    ...

    free(c);

}
```

따라서 배열 a, b, c는 각각 자동, 정적, 동적인 기억 수명을 갖는다(단, 엄밀히 말하여 c는 포인터이지 배열이 아니다). 이들 요소는 모두 a[i], b[i], c[i] 등의 방법으로 액세스할 수 있으며 더우기, a, b, c는 배열의 선두 어드레스를 갖는다고 볼 수 있으므로 수명에 관계없이 프로그램을 기술할 수 있게 된다. 또, 다른 기억 수명으로의 변경도 간단하다는 것을 알 수 있다.

▷ ····· 4-2-2 할당 영역 크기의 변경 - 범하기 쉬운 실수

할당한 기억 장소의 크기를 변경시킬 때 realloc 함수를 사용합니다. realloc 함수의 개요는 다음과 같습니다.

realloc

형 식 : void *realloc(void *ptr, size_t size);
프로토타입 : <stdlib.h>
기 능 : realloc은 ptr이 가리키는 객체(기억 장소)의 크기를 size 바이트로 변경한다. 변경 전후의 객체의 내용은 조금도 변하지 않는다. 영역 할당에 실패한 경우에도 객체의 내용은 변하지 않는다. ptr이 NULL인 경우에는 malloc(size)과 같은 동작을 한다. ptr이 NULL이 아니며, 이전에 calloc, malloc, realloc 함수가 돌려준 포인터와 일치하지 않을 경우의 동작은 정의 되어있지 않다. 또, 영역이 모두 free, realloc 함수로 해제되고 있는 경우의 동작도 정의 되어있지 않다. size가 ptr이 NULL이 아닌 경우는, free(ptr)와 같은 동작을 한다. size가 0이고 ptr이 NULL이 아닌 경우의 동작은 컴파일러에 따라 다르다.
리 턴 값 : 영역 할당이 성공한 때는 할당된 영역의 선두 포인터를 되돌려준다. 영역 할당에 실패한 경우는, NULL을 되돌려 준다.

— 첫번째 인자인 포인터는 변경하고 싶은 영역을 가리키고 두 번째 인자를 사용해서 새로운 크기를 지정하는 것입니까?

그렇습니다. 그리고 변경된 영역의 포인터를 되돌려 받게 됩니다. 리스트 4-5 프로그램에서는 int형 8개분 크기의 영역을 할당하였습니다. 이 프로그램에서 할당된 영역을 int형 10개분의 크기로 변경하는 프로그램을 작성해 보십시오.

— realloc을 덧붙이면 되겠지요..... 리스트 4-6입니다.

이 프로그램은 틀렸습니다.

— 왜 그렇습니까? 경고도 에러도 없이 컴파일했는데요.

틀린 곳은

```
p = (int *)realloc(p, 10 * sizeof(int));
```

부분입니다. 만약 영역의 크기 변경에 실패한다면 어떻게 되겠습니까.

— p에 NULL이 대입됩니다.

[리스트 4-6] 영역의 크기를 변경하는(틀린) 프로그램

```
/*
    List 4-6  영역의 크기를 변경하는(틀린) 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p;
```

```
        p = (int *)calloc( 8, sizeof(int));
        if ( p == NULL)
            printf("not allocated");
        else {
            p = (int *)realloc(p,10 * sizeof(int));
            /* 하고싶은 처리 */
            free(p);
        }
        return(0);
    }
```

만약 실패한 경우

```
free(p);
```

에서 0을 넘겨주기 때문에 원래 할당하고 있던 8개분의 영역이 해제되지 않게 됩니다.

— 그렇습니까?

올바른 프로그램을 리스트 4-7에 보였습니다.
영역의 크기를 변경하는 부분이 다음과 같이

```
temp = (int *)realloc(p, 10 * sizeof(int));
```

temp라는 변수에 realloc 함수의 리턴값이 대입됩니다.
만약 이것이 NULL이라면 영역 할당에 실패한 것이지요.

— 성공한 경우에 else의 다음 행에서

```
p = temp;
```

라고 되어 있는데 이것은 무엇 때문입니까?

p에 temp를 대입함으로써 영역의 크기를 변경한 후의 어드레스가 p에 대입되게 됩니다. 이처럼 영역의 크기를 변경하는데, 실패한 경우 p는 원래 할당하고 있던 8개분의 영역을 가리키며, 성공한 경우는 변경 후의 10개분의 영역을 가리키게 됩니다.

[리스트 4-7] 영역의 크기를 변경하는 프로그램

```
/*
    List 4-7  영역의 크기를 변경하는 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p;

    p = (int *)calloc( 8,sizeof(int));
    if (p == NULL)
        printf("not allocated");
    else {
        int *temp;

        temp = (int *)realloc(p, 10 * sizeof(int));
        if (temp == NULL)
            printf("fail to resize area\n");
        else {
            p = temp;
            /* 하고 싶은 처리 */
        }
        free(p);
    }
    return(0);
}
```

어쨌든 p는 8개분 또는 10개분의 영역을 가리키고 있기 때문에

```
free(p);
```

를 이용하여 영역의 해제를 바르게 할 수 있게 됩니다.

▷ …… 4-2-3 이차원 배열의 할당

그러면 이번에는 이차원 배열을 동적으로 할당하는 프로그램을 작성해 보십시오.

― 잘 모르겠는데요.

예를들어 5×3 크기의 int형의 이차원 배열을 할당할 경우 리스트 4-8과 같은 프로그램이 됩니다.

― 이번에는 포인터형이 복잡하군요.

int (*p)[3];이라는 형을 본 기억이 있지요?

[리스트 4-8] 정수 5×3의 이차원 배열을 할당, 해제하는 프로그램

```
/*
   List 4-8  5x3의 이차원 배열을 할당, 해제하는 프로그램
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int (*p)[3];

    p = (int (*)[3])calloc( 5*3 , sizeof(int));
    if ( p == NULL)
        printf("not allocated");
    else {
        /*  하고 싶은 처리 */
        free(p);
    }
    return(0);
}
```

― 이차원 배열을 건네 받은 함수의 인자 선언 때에 나왔었습니다.

이차원 배열은 배열의 배열 즉, 각 요소가 배열인 배열이었습니다.

따라서 동적으로 할당한 영역을 5×3의 이차원 배열로써 사용하기 위해서는 p[0]이 그 영역의 선두에 <int형의 크기 3의 배열>로 존재하며, 그 뒤에 p[1]이 <int형 크기 3의 배열>로 존재하도록 배치해야 합니다. 따라서 p는 <int형의 크기 3의 배열>의 포인터이어야 합니다.

― 그림 4-4를 보면 p는 일차원 배열이라는 것을 알겠습니다. 각 요소가 배열인 것이지요.

네, 그렇습니다. 3장에서도 설명했듯이 C 언어에서는 다차원 배열은 존재하지 않습니다. 각 요소가 배열인 배열로 실현됩니다.

p[0]은 int의 포인터형을 갖기 때문에 그림 4-4의 p[0]의 첫번째, 두 번째, 세 번째 부분은 각기 int형을 가지며, *p[0], *(p[0]+1), *(p[0]+2)라고 표시할 수 있습니다.

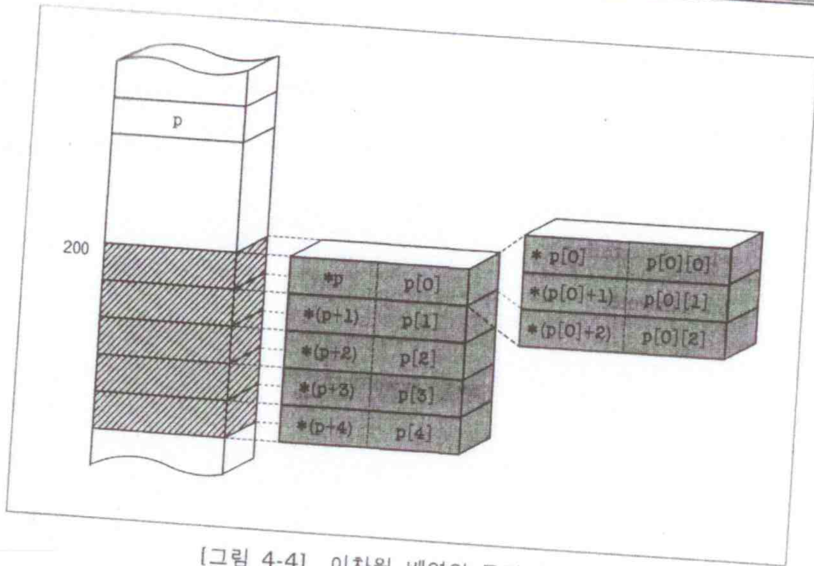
― *p[0]～*(p[0]+2)는 p[0][0]～p[0][2]이라고도 표현할 수 있다는 의미입니까?

그렇습니다. 일차원 배열 때에도 설명했듯이 calloc 함수는 단순하게 지정된 크기의 기억 장소를 물리적으로 할당합니다. 따라서 「1개의 변수」를 할당한다든가 「배열」을 할당한다는 의미가 아닙니다. 여기에서는 int형 5×3 크기의 이차원 배열과 같은 크기의 영역, 즉, int형 15개분의 영역을 할당하고 그것을 int형의 크기 3의 배열의 포인터에 대입하고 있을 뿐입니다. p자체가 포인터라는 것을 잊지 말아 주십시오.

p[i][j]와 같이 표현함으로써 이차원 배열처럼 보이게 한 것입니다.

☆ 중 요 ☆

calloc 함수는 지정한 크기의 기억 장소를 기계적으로 할당할 뿐이다.
한개의 변수 또는 배열(다차원 배열)로써의 사용 여부는 그것을 가리키는 포인터의 형에 의해 자유로이 결정할 수 있다. 즉, 「프로그램에 맡겨라」는 뜻이다.



[그림 4-4] 이차원 배열의 동적 할당

— 그러면 이차원 배열을 동적으로 할당할 경우에는 열의 크기(예를 들어 5*3이면 3)을 미리 알고 있어야 합니까?

그렇습니다. 하지만 흔히 사용하는 테크닉을 이용하면 열의 크기를 미리 알지 못해도 이차원 배열처럼 작성할 수 있습니다.

— 어떻게 하는 것입니까.

리스트 4-9를 봐 주십시오.

— int *x[3];은 int의 포인터 3개로 된 배열을 정의하고 있습니다.

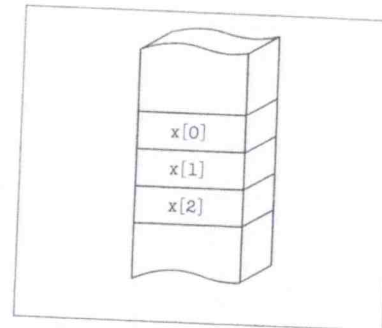
그렇습니다. 이것을 표현한 것이 그림 4-5입니다. 그렇지요?

— 네.

for문을 반복함으로써 size 즉, int형 4개분 만큼의 영역을 할당합니다. 할당 후의 메모리 상태는 그림 4-6과 같이 됩니다.

[리스트 4-9] 이차원 배열(?)을 동적으로 할당하는 프로그램

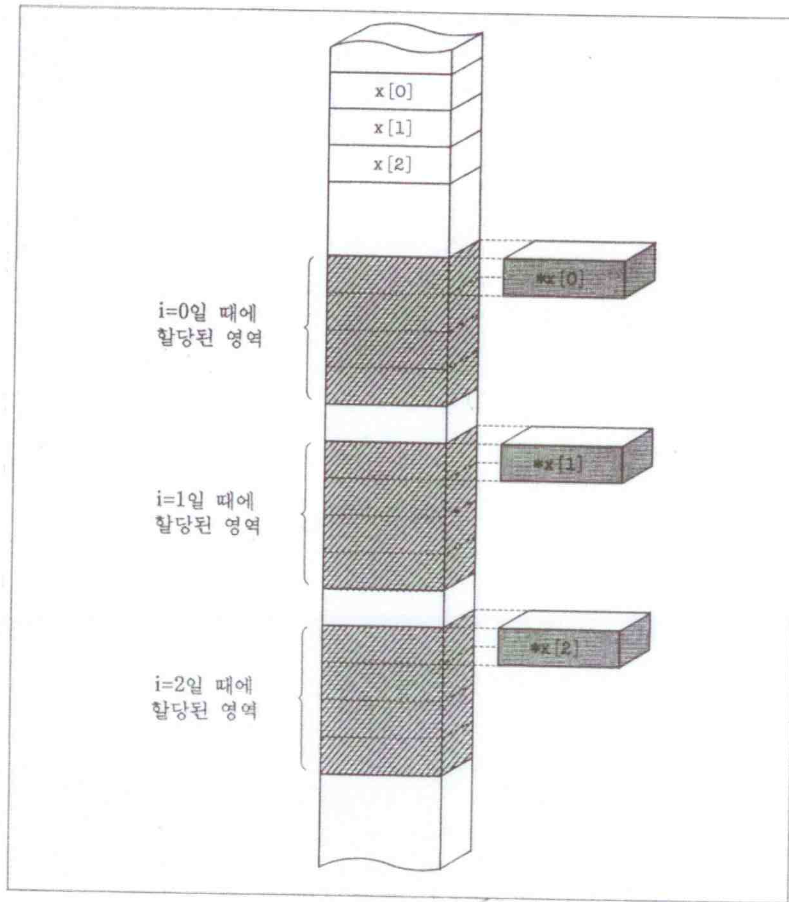
```
/*
    List 4-9  이차원 배열(?)을 동적으로 할당하는 프로그램
*/
#include <stdlib.h>

int main(void)
{

    int i, size = 4;
    int *x[3];

    for (i = 0; i < 3; i++)
        x[i] = (int *)calloc(size, sizeof(int));
    /* .... */
    for (i = 0;i < 3; i++)
        free(x[i]);
    return(0);
}
```



[그림 4-5] 포인터의 배열

— 알겠습니다.

그리고 할당된 영역의 선두 요소의 어드레스가 x[i]에 대입되기 때문에 그림 4-6과



[그림 4-6] 포인터의 배열과 할당된 영역

같이 *x[i]는 각각 할당된 영역의 선두 요소의 앨리어스가 됩니다.

— 왜 그렇습니까?

일반적으로 포인터 변수 ptr이 가리키는 요소의 실체는 *prt로 표현할 수 있었습니다. 이경우 x[i]가 ptr에 해당됩니다. 결국

ptr 이 가리키는 요소의 실체는 * ptr이라고 표현할 수 있다.
x[i] 가 기리키는 요소의 실체는 * x[i]라고 표현할 수 있다.

— 알겠습니다. 이것은 아주 기본적인 것이군요. x[i]라는 배열의 형이 된 것 뿐인데 응용을 하지 못했습니다.

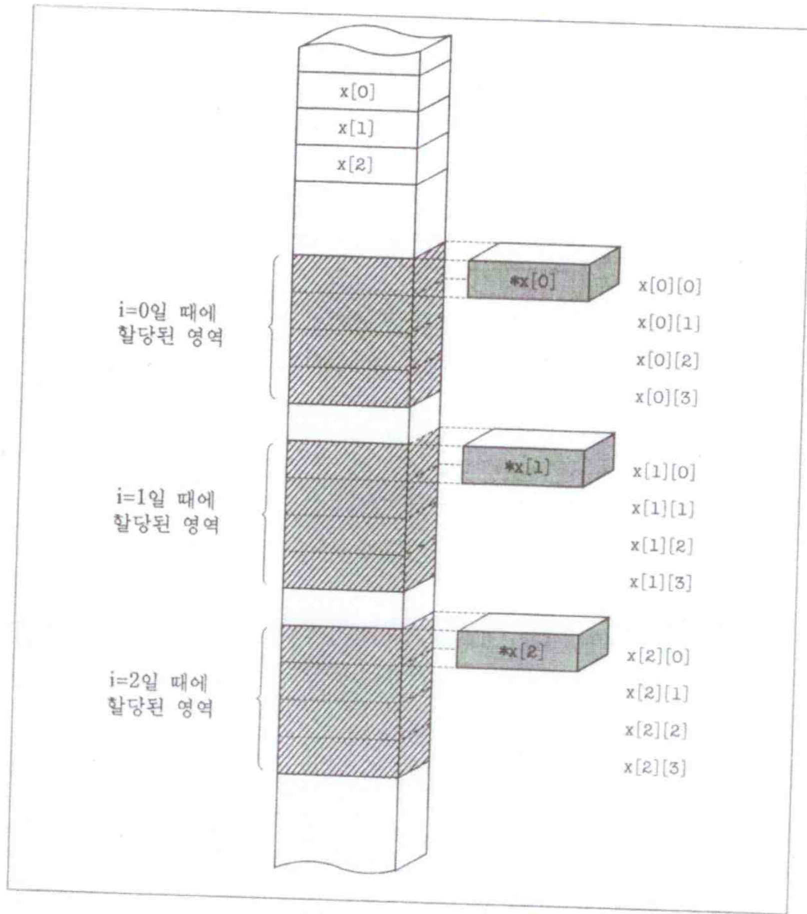
그러면 제일 처음 할당된 영역에 주목해 봅시다.

— int형 4개 크기의 영역입니다.

그림 4-6에서도 알 수 있듯이 선두의 요소는 *x[0]으로 표현할 수 있지만 다른 표현 방법은 없습니까?

— x[0]은 포인터 변수입니다. 포인터 변수 ptr에 대해 *ptr은 ptr[0]이라고 쓸 수 있었습니다. 따라서, 지금은 x[0]이 ptr에 해당되므로 *x[0]은 x[0][0]이라고 표현할 수 있습니다.

맞습니다. 그러면 두 번째 요소는 이해할 수 있겠지요. 이것은 x[0]이 가리키는 다음 요소의 실제 내용이기 때문에 *(x[0] +1)이지만 x[0][1]이라고도 쓸 수 있습니다. 최종적으로 그림 4-7과 같이 표현할 수 있습니다.



[그림 4-7] 포인터의 배열과 할당된 영역

— 만일 size의 값을 변경한다면 열의 크기가 다른 이차원 배열을 할당할 수 있겠군요. 이렇게 해서 완벽한 이차원 배열을 얻게 되는군요.

유감입니다. 실은 여기에서 할당된 것은 이차원 배열이 아닙니다. 이차원 배열과 닮았을 뿐 결정적인 차이가 있습니다.

— 네?

다차원 배열과 포인터 배열의 차이를 생각해 보십시오. 함수의 인자로

```
func(x);
```

다차원 배열을 넘겨줄 때와 포인터의 배열을 넘겨주는 것은 전혀 의미가 달랐습니다. 이밖에도 차이가 있습니다.

— 그 외에도 또 있습니까?

여기에서 할당한 것은 3×4의 배열과 비슷한 것입니다.

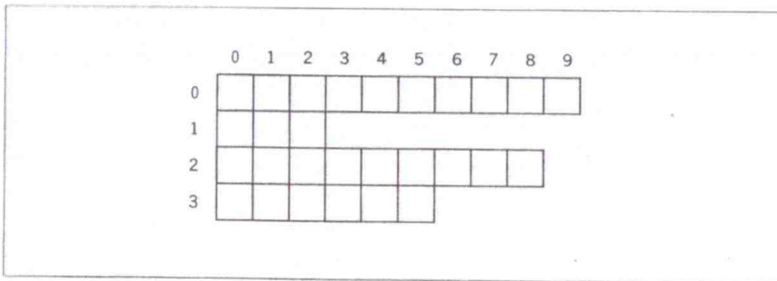
```
int a[3][4];
```

와 같이 정의된 본래의 이차원 배열은 모든 영역이 연속해서 메모리상에 존재합니다. 예를 들면 a[0][3] 직후에 a[1][0]이 있으나 여기에서 할당된 이차원 배열은 그림 4-7로부터 알 수 있듯이 연속된 영역을 갖는다고는 할 수 없습니다.

4-2-4 행에 대한 열의 크기가 다른 배열의 할당

그러면 다음에는 행에 대한 열의 크기가 다른 배열을 할당하는 프로그램을 작성해 보겠습니다.

— 그림 4-8과 같이 들쑥 날쑥한 배열을 작성하는 것입니까?



[그림 4-8] 요철 배열

그렇습니다. 리스트 4-10이 그 프로그램 입니다. 이제 설명하지 않아도 알 수 있겠지요.

– 기본적으로는 리스트 4-9와 같으며 각 요소는 x[i][j]이라고 하는 표현으로 액세스 할 수 있습니다.

이 프로그램의 잘못된 점을 지적할 수 있겠습니까?

– 글쎄요?

이 프로그램은 영역을 할당하고 해제할 뿐 제대로 할당되었나를 검사하고 있지 않습니다. 만약 영역이 제대로 할당되지 않았는데, 예를 들어

```
x[0][5]=3;
```

을 실행하면 뜻하지 않은 결과가 발생합니다.

☆ 교 훈 ☆

기억 장소 영역 할당이 제대로 되었는지 반드시 체크한다.

[리스트 4-10] 이차원 배열을 할당하는 프로그램

```
/*
    List 4-10  이차원 배열을 할당하는 프로그램
*/
#include <stdlib.h>

int main(void)
{
    int    *x[4];

    x[0] = (int*)calloc(10, sizeof(int));
    x[1] = (int*)calloc( 3, sizeof(int));
    x[2] = (int*)calloc( 9, sizeof(int));
    x[3] = (int*)calloc( 6, sizeof(int));

     /* 하고 싶은 처리 */

    free(x[0]);
    free(x[1]);
    free(x[2]);
    free(x[3]);
    return(0);
}
```

– 이렇게 간단하게 배열을 사용할 수 있을 줄은 생각도 못했었습니다.

하하, 당신은 이미 이러한 배열을 사용하고 있습니다.

– 네? 기억이 나지 않는데요.

무엇보다도, 당신이 이러한 배열을 사용한 때는 이와 같이 동적으로 할당했던 것은 아니었습니다만 실은 문자(열)의 포인터 배열에서 사용했습니다. 다음과 같은 변수를 기억합니까?

```
static char *name[3] = {
    "C",
    "Pascal",
    "???",
};
```

– 아! 그렇군요. 각 요소는 name[i][j]로 액세스할 수 있으며 name[0], name[1], name[2]가 가리키는 문자열의 길이가 각각 다릅니다.

이제 생각났군요. 이번에는 동적으로 할당했다 뿐이지, 기본적으로 배열 name과 같은 것임을 알 것입니다.

문자열의 경우 이와 같이 간단하게 정의할 수 있지만 그것은 요철 배열의 경우도 마찬가지입니다.

☆ 중 요 ☆

요철 배열

```
static char *name[3] = {
        "C",
        "Pascal",
        "???",
};
```

은 문자열에서는 간단하게 정의할 수 있지만, 그 이외의 경우는 동적으로 할당하여 간단하게 구현할 수 있다.

▶ 문제 4-2

행의 수와 열의 수가 모두 가변인 이차원 배열을 동적으로 할당하는 프로그램을 작성하시오.

▷ 4-3 리스트 처리의 응용

〉…… 4-3-1 리스트 구조

— 메모리의 동적 할당 기능을 사용하면 리스트 처리를 간단하게 할 수 있다고 들은 적이 있습니다.

간단한 것이 어느 정도인지는 차차 하더라도 주기억 장치 상에서 리스트를 처리하는 경우 배열을 사용하기 보다도 동적 할당 기능을 사용하는 경우가 많은 것은 확실합니다.

— 그러면 리스트 처리란 무엇입니까?

우선 클럽의 전화 연락부를 생각해 보겠습니다. 각각의 회원은 다른 한 사람의 회원에게 전화를 합니다. 우선 회원은 최대 5명으로, 데이터로는 이름만을 생각하기로 합시다. 다음과 같은 구조의 배열을 생각할 수 있습니다.

```
struct net {
  char name[20];      /* 이름 */
  int  next;          /* 다음 사람의 인덱스 */
} data[5];
```

표 4-1과 같은 초기값이 주어진다고 합시다.

[표 4-1] 전화 번호부의 데이터

첨 자	name	next
0	Lee Young Ran	3
1	Jang Soo Cheol	2
2	Seo Mi Ri	-1
3	Kim Eun Soo	1
4		-

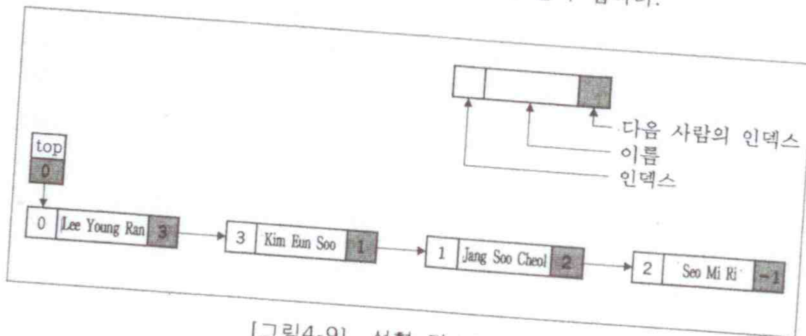
― 구성원 next에는 그 사람이 전화를 걸어야 하는 인덱스가 들어있는 것입니까?

네, 맞습니다. 단, -1은 전화를 걸 필요가 없는 것을 나타냅니다. 즉, 연락표의 마지막 회원이라는 것을 나타내고 있습니다.

― data[4]는 미등록을 나타냅니까?

그렇습니다. 여기에서 top이라고 하는 변수를 도입합니다. 이 top에는 제일 처음에 전화를 걸 사람의 인덱스를 저장합니다.

표 4-1을 알기 쉽게 그림으로 나타내면 그림 4-9와 같이 됩니다.



[그림4-9] 선형 리스트의 예

― 처음 사람으로부터 next를 찾아가면 전화를 거는 순서를 알 수 있군요.

단, next가 -1이 되면 그 사람이 마지막이므로 더이상 찾아갈 필요가 없습니다.

이것을 프로그램으로 작성한 것이 리스트 4-11입니다.

[리스트 4-11] 전화 연락부의 표시

```
/*
    List 4-11  전화 연락부의 표시
*/
#include  <stdio.h>

struct net {
    char name[20];        /* 이름 */
    int  next;            /* 전화할 상대방의 인덱스 */
};

int top = 0;                    /* 제일 처음 전화할 사람의 인덱스 */
```



```
struct net data[5] = {          /* 전화 연락부의 데이터 */
    "Lee Young Ran",  3,
    "Jang Soo Cheol", 2,
    "Seo Mi Ri",     -1,
    "Kim Eun Soo",    1,
    "",               0,
};

int main(void)
{
    int     now;            /* 현재 전화할 회원(의 인덱스) */

    now = top;
    while (now != -1) {
        printf("%s\n", data[now].name);
        now = data[now].next;
    }
    return(0);
}
```

그림 4-9로부터 알 수 있듯이 한 개의 블럭은 최후의 것을 제외하고 다음 블럭을 가리키는 포인터를 한 개 포함하고 있으며 사슬 모양으로 연결되어 있습니다.

― 포인터 변수는 없군요.

여기에서의 포인터는 이른바 C 언어의 『포인터』가 아닌, 일반적으로 <어떤 것을 가리킨다>라는 의미로써의 포인터입니다.

― 알겠습니다. 설명을 계속해 주십시오.

이와 같이 연결된 블럭 전체의 모임을 선두 블럭을 가리키는 포인터―여기에서는 top입니다―를 포함하여 **선형 리스트(linear list)**, **연결 리스트(linked list)**, **일방향 리스트** 등으로 부릅니다. 선형 리스트를 구성하고 있는 각 블럭을 리스트의 **요소** 또는 **노드**라고 부릅니다.

그것들은 이 예에서와 같이 C 언어에서는 구조체로 표현할 수 있습니다.

― 이 경우, 한 개의 구조체가 노드가 되며 노드 속에는 이름의 데이터와 다음 사람을 가리키는 포인터를 갖고 있는 것입니까?

그렇습니다. 선형 리스트의 각 노드는 최후의 것을 제외하고는 모두 그 후속 노드(successor node)를 가리키는 포인터를 포함하며 최초의 것을 제외하고 모두 그 선행 노드에 의해 가리켜지게 됩니다.

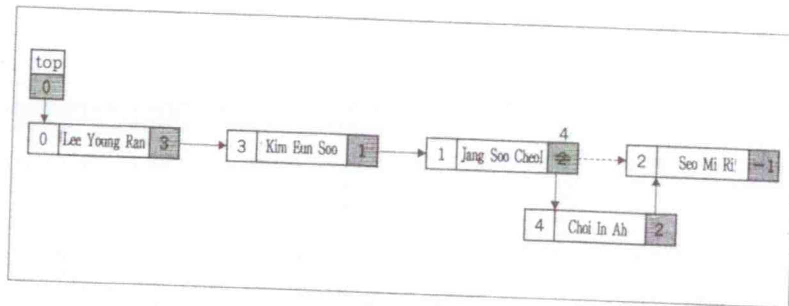
처음과 끝의 노드는 각각 선두 노드(head node), 말미 노드(tail node)라고 불립니다.

– 어렵습니다.

전혀 어렵지 않습니다. 단어가 많이 나왔기 때문에 어려운 느낌이 드는 것입니다.

그런데 이 리스트에 새로운 노드를 덧붙이는 것을 생각해 보겠습니다. 최인아 양이 신입회원으로 들어오게 되었습니다. 최인아 양에게는 선배인 장수철군이 전화를 하는 것이 적당합니다.

– 지금 data[4]가 비어있기 때문에 거기에 최인아 양의 데이터를 넣는다는 의미입니까?



[그림4-10] 선형 리스트의 추가 예

그렇습니다. 그림 4-10을 봐 주십시오.

– 장수철씨의 포인터 next만을 바꾸면 되는군요.

다른 것은 전혀 변경할 필요가 없는 것을 알 수 있습니다.

만약 next에 의해 다음 노드(회원)를 가리키는 포인터가 없다–즉, 선형 리스트 구조가 아니다–고 할 때 단순 배열의 데이터 구조로 처리할 경우 배열을 항상 전화거는 순서로 나열하지 않으면 안되며 노드의 추가나 삭제시 많은 배열을 이동해야 하는 단점이 있습니다.

선형 리스트의 장점은 삽입이나 삭제가 물리적인 이동을 동반하지 않고 실행할 수 있다는 것입니다.

▶ 문제 4-3

리스트 4-11의 프로그램에 위의 삽입 절차(신입 회원 최인아 양이 장수철씨와 서미리씨 사이에 들어감)을 함수로 작성해 추가하십시오.

▶ 문제 4-4

리스트 4-11의 프로그램에 장수철씨가 탈퇴하면 수행해야 되는 삭제 절차를 함수로 추가 작성하십시오.

4-3-2 자기 참조 구조체

여기까지 설명한 배열에 의한 선형 리스트의 실현은 최선의 방법이라고는 말할 수 없습니다. 왜냐하면 미리 데이터의 갯수를 알 수 있는 경우에 한하여 적용할 수 있기 때문입니다.

데이터의 갯수를 미리 알 수 없을 때는 데이터의 추가가 있을 때에 노드를 동적으로 할당하면 됩니다.

– 다시 동적 메모리 할당 이야기가 되는군요.

노드를 동적으로 할당하기 때문에 다음 노드를 가리키는 포인터로써 본래(!)의 포인터 변수를 사용하면 좋을 것 같습니다. 포인터를 사용한 선형 리스트의 설명에 들어가기 전에 자기 참조 구조체(self-referential structure)에 관해 간단하게 설명하겠습니다.

— 자기 참조 구조체라구요? 어려울 것 같습니다.

무엇이든 우선 어렵다고 생각하는군요. 통상의 구조체를 이해할 수 있으면 자기 참조 구조체를 이해하는 것은 간단합니다.

— 그렇습니까?

전화 연락부의 구조체는 다음과 같이 정의되어 있습니다.

```
struct net {
    char name[20];          /* 이름 */
    int  next;              /* 다음 사람의 인덱스 */
} data[5];
```

구조체에서는 자기 자신의 수형을 가리키는 포인터도 멤버가 될 수 있습니다. struct net형의 next를 자기 자신의 형을 가리키는 포인터로 변경하겠습니다.

```
struct net {
    char   name[20];          /*  이  름  */
    struct net *next;         /*  다음 사람을 가리키는 포인터 */
};
```

이와 같이 자기 자신의 구조체형을 가리키는 포인터를 구성원으로 갖는 구조체를 자기 참조 구조체라고 부릅니다.

— 앞에서 간단하다는 이야기를 했습니다만, 역시 저에게는 어렵습니다.

하하, 명칭 때문에 혼란스러운 뿐입니다. 그러면 이런 구조체라면 이해할 수 있겠지요?

```
struct net {
    char  name[20];          /* 이름 */
    char  *ptr;              /* 문자(열)을 가리키는 포인터 */
}
```

— 이것이라면 알 수 있습니다. 구조체의 구성원 하나가 char의 포인터가 되고 있을 뿐이지 않습니까.

그렇습니다. 앞에서의 자기 참조 구조체도 같은 것입니다. 『자기 참조』라고 하는 말에 당혹해 하고 있는 것은 아닙니까? 구성원

```
struct net *next;
```

의 next는 자기 자신을 가리켜야만 되는 것이 아닐까하고 착각할 필요는 없습니다. 자신과 같은 형을 가리키는 포인터를 구성원으로 갖고 있다고 생각하면 됩니다.

— 아하! 그런 것이 있군요. 선생님이 말씀하신 대로, 자기 참조이기 때문에 자기 자신을 가리키지 않으면 안되는 것은 아닐까? 하고 착각했습니다.

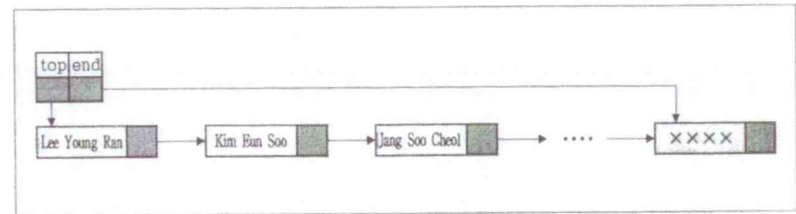
4-3-3 선형 리스트의 구현

그러면 선형 리스트를 자기 참조 구조체를 사용해 표현해 봅시다.

— 겨우 주제로 돌아왔군요.

그림 4-11에 나타낸 것과 같이 선두 노드를 가리키는 포인터 이외에, 말미 노드를 가리키는 포인터를 도입합니다.

여기에서 말미 노드는 더미 노드 (dummy node) 즉, 데이터를 저장하지 않는 여분으로 도입한 노드입니다.



[그림4-11] 더미 노드를 도입한 선형 리스트의 예

— 그림 중의 xxxx 부분이군요. 왜 그런 노드가 필요합니까?

리스트의 마지막을 나타내는 표시가 되며 여러가지 처리를 할 때에 편리하기 때문입니다. 이와 같은 테크닉은 여러가지 알고리즘에서 사용됩니다. 여기에서 도입한 더미 노드와 같이 여분으로 도입한 표시가 되는 것을 일반적으로 파수(sentinel)라고 부릅니다. 전화 연락표의 구조체는 아래와 같이 정의됩니다.

```
struct net {
    char   name[20];          /* 이름 */
    struct net *next;         /* 다음 사람을 가리키는 포인터 */
};
```

— 네.

여기서는 typedef를 사용해 다음과 같이 두 개의 형을 정의하겠습니다.

```
typedef struct _net {
    char    name[20];          /* 이 름 */
    struct _net *next;         /* 다음 사람을 가리키는 포인터 */
} net;

typedef struct  {
    net   *top;                /* 선두 노드를 가리키는 포인터 */
    net   *end;                /* 말미 노드를 가리키는 포인터 */
} cont_net;
```

— 최초의 typedef는 노드를 표현하는 구조체의 정의군요. _net와 net의 차이점은 무엇입니까?

typedef를 사용해 <name과 next를 멤버로 갖는 구조체에 net형이라는 형 명칭을 준다>라는 선언을 하고 있습니다. 그러나 두 번째의 멤버 next의 선언에 있어서는

```
net *next;      /* 다음 사람을 가리키는 포인터 */
```

라고 기술할 수 없습니다.

— 왜 그렇습니까?

왜냐하면, next를 선언하는 시점에서 net형의 typedef가 아직 끝나지 않았기 때문입니다. 다음과 같이 쓰면 알 수 있습니다.

```
typedef struct _net {
    char    name[20];
    struct _net  *next;        /* 아직 net의 선언은 끝나지 않음 */
} net;                         /* 여기에서 net의 선언이 완료 */
```

따라서 컴파일러는 net라는 이름이 있다고 하는 것조차 인식할 수 없습니다.

— 그래서, 구조체 태그인 _net를 사용하는군요. 선언 한 개라도 깊은 의미가 있군요. 두 번째 구조체의 typedef는 선두 노드와 말미 노드를 가리키는 포인터를 구성원으로 하는 구조체형을 선언하고 있는 것입니까?

그렇습니다. 이 두 개의 포인터를 합한 것은 제어 블럭(control block)이라고 부르겠습니다. 단, 제어 블럭은 일반적인 용어가 아니므로 주의해 주십시오.

그런데 데이터의 추가가 있을 때에는 노드를 메모리 상에 동적으로 할당해야 합니다. 크기가 요소 한 개의 크기인 영역을 할당해, 그 포인터를 되돌려 주는 함수는 다음과 같습니다.


```
net *AllocNet(void)
{
    return((net *)calloc(1,sizeof(net)));
}
```

— 이것은 알겠습니다. calloc 함수로써 메모리를 할당할 뿐입니다. 포인터를 되돌려 주는 함수라는 것도, 2장에서 배웠습니다.

그런데, 선형 리스트는 맨처음이 비어 있습니다.

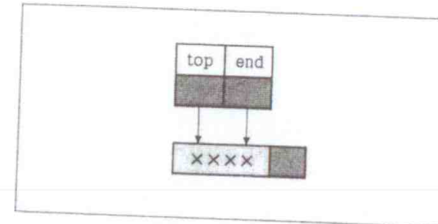
— 비어있다고요?

선형 리스트는 처음에는 데이터가 한 개도 없기 때문입니다. 그런 의미로 비어 있는 것입니다. 단, 앞에서도 설명했듯이 말미 노드로서 더미 노드가 도입되는 것입니다. 따라서 요소는 한 개만 존재합니다.

리스트 4-12가 빈 선형 리스트를 작성하는 프로그램입니다.

— InitNet 함수는, 더미 노드를 할당하며, 제어 블럭의 top과 end가 그것을 가리키도록 하는군요.

그렇습니다. 그림 4-12와 같은 상태가 됩니다.



[그림4-12] 빈 선형 리스트

[리스트 4-12] 빈 선형 리스트를 작성한다

```
/*
    List 4-12 빈 선형 리스트를 작성한다
*/
#include <stdlib.h>

typedef struct _net {
    char          name[20];       /* 이름 */
    struct _net *next;            /* 다음 사람을 가리키는 포인터 */

} net;

typedef struct {
    net    *top;                  /* 선두 노드를 가리키는 포인터 */
    net    *end;                  /* 말미 노드를 가라키는 포인터 */
} cont_net;

/* 블럭 한 개를 할당 */
net *AllocNet(void)
{
    return((net *)calloc(1,sizeof(net)));
}

/* 제어 블럭의 초기화 */
void InitNet(cont_net *control)
{
    control->top = control->end = AllocNet();
}

/*  메인 함수 */
int main(void)
{
    cont_net    cont;         /* 제어 블럭용의 변수 */

    InitNet(&cont);           /* 제어 블럭 */
    return(0);
}
```

리스트 4-12의 프로그램은 영역의 할당에 실패한 경우의 처리를 하고 있지 않기 때문에 좋은 예라고 할 수 없습니다. 그러나 검사를 하면 처리가 복잡하게 되므로 알고리즘을 설명하는 데 번잡하게 되기 때문에 앞으로는 모든 예에서 이와 같은 영역 할당이 제대로 되었는지의 검사는 하지 않겠습니다. 그러면 선형 리스트의 선두에 노드를 삽입하는 절차를 생각해 봅시다. 그림 4-13과 같이 됩니다.

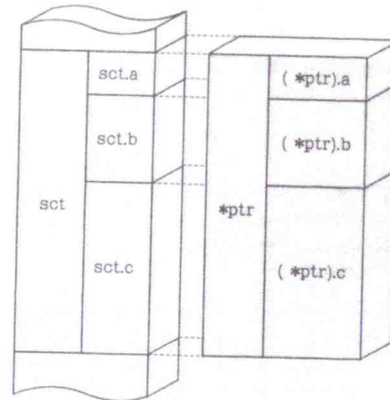
컬럼 4-5

화살표 연산자

다음과 같은 구조체를 생각해 보자.

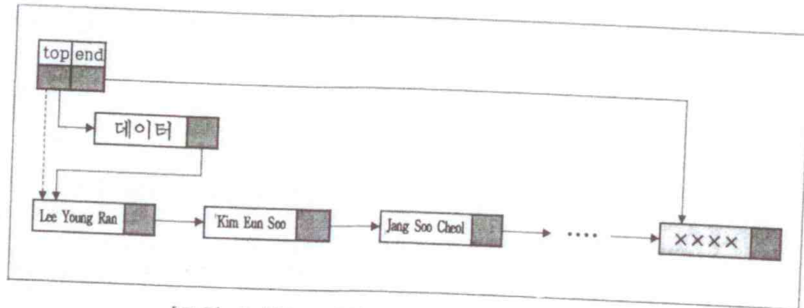
```
struct abc {
    int     a;
    long    b;
    double c;
} sct , *ptr;
```

구조체의 구성원을 액세스하고 싶을 때는, sct.a나 sct.b로 기술한다.
이것은 복수 요소의 집합인 배열에 []로 첨자를 붙여 임의의 요소를 나타내듯이 구조체에서는 도트(.) 연산자를 이용해 임의의 요소를 나타낸다.
그럼 ptr이 가리키고 있는 구조체의 임의의 요소를 나타내려면 어떻게 하면 좋을까? 지금, ptr이 sct를 가리키고 있다고 하자. *ptr은 sct의 앨리어스가 된다. 따라서 아래의 그림으로 알 수 있듯이 sct.a라고 하는 것과 마찬가지로 (*ptr).a라고 쓸 수 있다. 그러나 일일이 (*ptr).a라고 쓰는 것은 성가시기 때문에 이것을 ptr->a라고 생략해서 쓸 수 있다.
ptr->a라고 하는 표현이 이해하기 어려운 경우 아래와 같이 그림으로 나타내 (*ptr).a를 생각해 보면 쉽게 이해된다. 연산자는 화살표와 비슷하기 때문에 화살표 연산자라고 불리워진다.



— top이 추가하는 데이터를 가리키는 것과 같이 변경해야 하는군요.

또, 추가한 데이터의 next가 그때까지 선두였던 사람을 향하도록 수정해야 합니다.



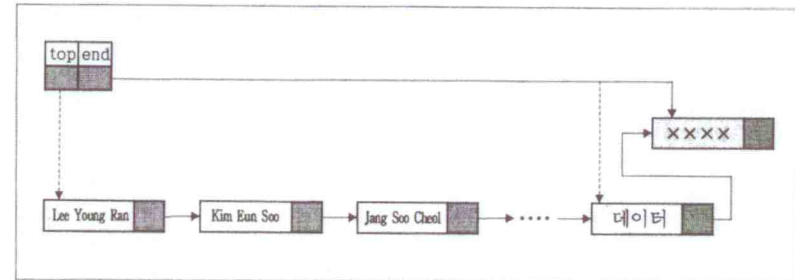
[그림 4-13] 선형 리스트의 처음 노드에 삽입

선두에 노드를 추가하는 함수는 다음과 같습니다.

[리스트 4-13] 선두 노드에의 삽입

```
/*
    List 4-13  선두 노드에의 삽입
*/

/* 선두 노드의 추가 */
void InsertNet(cont_net *control, const char *name)
{
    net    *ptr;

    ptr = control->top;
    control->top = AllocNet();
    strcpy(control->top->name, name);
    control->top->next = ptr;
}
```

바꾸어서, 선형 리스트의 끝에 어떤 데이터를 갖는 노드를 삽입해 보겠습니다. 그림 4-14와 같이 새롭게 더미 노드를 만들고, 이제까지 더미 노드였던 노드에 이름 데이터를 복사하여 구현할 수 있습니다. 그러면 이러한 일을 수행하는 함수를 작성해 보십시오.



[그림 4-14] 선형 리스트의 끝 노드에 삽입

— 네, 했습니다. 리스트 4-14입니다.

[리스트 4-14] 끝 노드의 삽입

```
/*
    List 4-14  끝 노드의 삽입
*/

/* 끝 요소의 추가 */
void AppendNet(cont_net *control, const char *name)
{
    net     *ptr;

    ptr = control->end;
    control->end = AllocNet();
    strcpy(ptr->name, name);
    ptr->next = control->end;
}
```

맞았습니다. 그러면 조금 핵심적인 문제를 내겠습니다. 리스트 4-11에서 배열을 사용해, 모든 회원을 표시하는 프로그램을 나타냈습니다.

이 프로그램을 토대로 지금 설명한 메모리의 동적 할당, 자기 참조 구조체를 사용하여 프로그램을 작성해 주십시오.

— 선두로부터 next를 찾아가면 됩니다. 네 했습니다.

[리스트 4-15] 선형 리스트상의 데이터를 모두 표시

```
/*
    List 4-15 선형 리스트상의 데이터를 모두 표시
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _net {
    char        name[20];         /* 이름 */
    struct _net *next;            /* 다음 사람을 가리키는 포인터 */
} net;

typedef struct {
    net     *top;                 /* 선두 노드를 가리키는 포인터 */
    net     *end;                 /* 말미 노드를 가리키는 포인터 */
} cont_net;

/* -------------------------------------------------------- */
/*  아래의 함수는 리스트 4-12 ~ 리스트 4-14와 같음           */
/*                                                          */
/*      net *AllocNet(void)                                 */
/*      void InitNet(cont_net *control)                     */
/*      void InsertNet(cont_net *control, const char *name) */
/*      viod AppendNet(cont_net *control, const char *name) */
/*                                                          */
/* -------------------------------------------------------- */

/* 모든 요소를 출력한다 */
void PrintNet(cont_net *control)
{
    net     *ptr;

    ptr = control->top;
    while (ptr != control->end){
            printf("%s\n", ptr->name);
            ptr = ptr->next;
    }
}

/* 메인 함수 */
int main(void)
{
    cont_net    cont;             /* 제어 블럭용 변수 */
    InitNet(&cont);               /* 빈 선형 리스트를 작성 */

    InsertNet(&cont, "Lee Young Ran");       /* 선두에 추가 */
    AppendNet(&cont, "Kim Eun Soo");         /* 말미에 추가 */
    AppendNet(&cont, "Jang Soo Cheol");      /* 말미에 추가 */
    AppendNet(&cont, "Seo Mi Ri");           /* 말미에 추가 */

    PrintNet(&cont);              /* 전화 연락부의 모든 이름 출력 */

    return(0);
}
```


◎ 실행 결과 ◎

```
Lee Young Ran
Kim Eun Soo
Jang Soo Cheol
Seo Mi Ri
```

지금까지는 노드를 추가하는 절차를 설명했습니다. 그러면 다음에는 노드를 삭제하는 절차를 설명하겠습니다. 우선, 선두 노드를 삭제하는 방법을 생각해 보겠습니다.

– top이 가리키고 있는 노드를 삭제하는 것이군요. 그림 4-15와 같이 top이 두 번째 데이터를 가리키도록 변경하면 되며, 프로그램은 리스트 4-16과 같이 됩니다.

[그림4-15] 처음 노드의 삭제

[리스트 4-16] 선두 노드의 삭제(첫번째)

```
/*
    List 4-16  선두 노드의 삭제(첫번째)
*/

/* 선두 요소의 삭제 */
void DeleteNet(cont_net *control)
{
    net     *ptr;

    ptr = control->top->next;
    free(control->top);
    control->top = ptr;
}
```

유감이지만 이 프로그램은 경계 조건을 만족하지 못합니다. 왜냐하면 리스트가 비었을 경우—즉, 더미 노드 뿐일 때—를 생각해야 합니다.

군이 작성한 프로그램은 더미 노드를 삭제하게 되며, 따라서 정의되지 않은 값이 ptr에 대입되어 버립니다. 그러므로, 리스트가 비었을 때는 삭제하지 않도록 해야 합니다. 어떻게 하면 좋을까 생각해 보십시오.

– 리스트가 비어 있을 때는, top과 end가 같이 더미 노드를 가리키고 있기 때문에 그때는 아무 것도 실행하지 않도록 하면 된다고 생각합니다. 프로그램은 리스트 4-17과 같이 됩니다.

[리스트 4-17] 선두 노드의 삭제(두번째)

```
/*
    List 4-17  선두 노드의 삭제(두번째)
*/

/* 선두 요소의 삭제 */
void DeleteNet(cont_net *control)
{
    if (control->top != control->end) {
        net *ptr = control->top->next;
        free(control->top);
        control->top = ptr;
    }
}
```


그러면 모든 노드를 삭제해 빈 리스트를 만드는 프로그램을 작성해 보십시오.

[리스트 4-18] 모든 노드를 삭제한다.

```
/*
    List 4-18 모든 노드를 삭제한다.
*/

/* 모든 요소의 삭제 */
void ClearNet(cont_net *control)
{
    net   *ptr;

    ptr  = control->top;
    while (ptr !=control->end){
            net *ptr2 = ptr->next;
            free(ptr);
            ptr = ptr2;
    }
    control->top = control->end;
}
```

이제 선형 리스트의 조작에 관한 가장 기본적인 함수를 작성했습니다. 그밖에도 탐색, 처음과 끝 이외의 임의의 위치에 추가, 임의의 노드 삭제 등의 기능을 수행하는 함수를 작성해야 되지만 이것은 과제로 남기겠습니다(문제 4-5 및 연습 문제, 해답 참조).

– 질문이 하나 있습니다.

무엇이지요?

– 이번 <클럽 주소록>을 나타내는 선형 리스트를 구현한 모든 함수에게 인자로 제어 블럭을 넘겨주고 있지만 왜 이렇게 하는지 잘 모르겠습니다.

왜 그런 생각을 하게 되었습니까?

– 이 프로그램의 어디에서라도 액세스할 수 있도록 광역 변수로 해버리면 일부러 인자로 넘겨주지 않아도 좋지 않습니까?

그러면 여기에서 작성한 모든 함수를 사용한 간단한 샘플 프로그램을 보도록 하죠. 리스트 4-19에 보였습니다. 여기에서 TermNet 함수는 더미 노드를 포함한 선형 리스트 상의 모든 데이터가 점유하는 메모리를 해제하여 리스트 처리를 종료하기 위한 함수 입니다. 이 프로그램을 잘 생각해 보면 군의 질문에 분명한 답이 될 수 있다고 생각됩니다.

[리스트 4-19] 두 개 클럽의 전화 번호부

```
/*
    List 4-19 두 개 클럽의 전화 번호부
*/
#include   <stdio.h>
#include   <stdlib.h>
#include   <string.h>

typedef struct _net {
    char        name[20];       /* 이름 */
    struct _net *next;          /* 다음 사람을 가리키는 포인터 */
} net;

typedef struct {
    net     *top;               /* 처음 노드를 가리키는 포인터 */
    net     *end;               /* 끝 노드를 가리키는 포인터 */
} cont_net;

/* 한 개의 블럭을 할당한다 */
net *AllocNet(void)
{
    return((net *)calloc(1, sizeof(net)));
}

/* 제어 블럭의 초기화 */
void InitNet(cont_net *control)
{
    control->top = control->end = AllocNet();
}

/* 처음 요소의 추가 */
void InsertNet(cont_net *control, const char *name)
{
    net     *ptr;

    ptr = control->top;
    control->top = AllocNet();
    strcpy(control->top->name, name);
```



```
    control->top->next = ptr;
}

/* 끝 요소의 추가 */
void AppendNet(cont_net *control, const char *name)
{
    net     *ptr;

    ptr = control->end;
    control->end = AllocNet();
    strcpy(ptr->name, name);
    ptr->next = control->end;
}

/* 처음 요소의 삭제 */
void DeleteNet(cont_net *control)
{
    if (control->top != control->end) {
        net *ptr = control->top->next;
        free(control->top);
        control->top = ptr;
    }
}

/* 모든 요소의 삭제 */
void ClearNet(cont_net *control)
{
    net     *ptr;

    ptr = control->top;
    while (ptr != control->end){
        net *ptr2 = ptr->next;
        free(ptr);
        ptr = ptr2;
    }
    control->top = control->end;
}

/* 리스트 처리의 종료 */
void TermNet(cont_net *control)
{
    ClearNet(control);
    free(control->top);
    control->top = control->end = NULL;
}
```

```
/* 모든 요소를 출력 */
void PrintNet(cont_net *control)
{
    net  *ptr;

    ptr = control->top;
    while (ptr != control->end){
          printf("%s\n", ptr->name);
          ptr = ptr->next;
    }
}

/* 메인 함수 */
int main(void)
{
   cont_net trckfld;              /* 제어 블럭용 변수(육상부) */
   cont_net photo;                /* 제어 블럭용 변수(사진부) */

   InitNet(&trckfld);             /* 빈 선형 리스트를 작성(육상부) */
   InitNet(&photo);               /* 빈 선형 리스트를 작성(사진부) */

   InsertNet(&trckfld, "Choi In Ah");      /* 육상부 선두에 추가 */
   InsertNet(&photo, "Lee Young Ran");     /* 사진부 선두에 추가 */
   AppendNet(&trckfld, "Oh Dal Ja");       /* 육상부 말미에 추가 */
   InsertNet(&trckfld, "Kim Young Sim");   /* 육상부 말미에 추가 */
   AppendNet(&photo, "Lee Mong Ryong");    /* 사진부 말미에 추가 */
   AppendNet(&photo, "Doggo Tak");         /* 사진부 말미에 추가 */

   printf("Track field\n");
   PrintNet(&trckfld);            /* 육상부의 모든 이름을 출력 */
   putchar('\n');

   printf("Photo\n");
   PrintNet(&photo);              /* 사진부의 모든 이름을 출력 */
   putchar('\n');

   TermNet(&trckfld);             /* 육상부의 선형 리스트 사용 종료 */
   TermNet(&photo);               /* 사진부의 선형 리스트 사용 종료 */

   return(0);
}
```


◎ 실 행 결 과 ◎

```
Track field
Kim Young Sim
Choi In Ah
Oh Dal Ja

Photo
Lee Young Ran
Lee Mong Ryong
Doggo Tak
```

▶ 문제 4-5

본문에서 설명한 선형 리스트에 대해, 아래의 함수를 추가하십시오.

(1) net *SearchNet(cont_net *control, char *name);

name이 가리키는 문자열과 같은 이름을 갖는 요소를 탐색해, 성공한 경우는 그 요소의 포인터를, 실패한 경우는 NULL을 돌려준다.

(2) void PutNet(cont_net *control, const char *name, net *ptr);

ptr이 가리키는 요소 직전에 name이 가리키는 문자열을 이름으로 갖는 요소를 삽입한다.

(3) void PushNet(cont_net *control, const char *name, net *ptr);

ptr이 가리키는 요소 직후에 name이 가리키는 문자열을 이름으로 갖는 요소를 삽입한다.

(4) void PopNet(cont_net *control);

끝 요소를 삭제한다.

(5) void GetNet(cont_net *control, net *ptr);

ptr이 가리키는 요소를 삭제한다.

제 5 장

포인터와 화일 처리

▷ 5-1 화일 처리의 기본
▷ 5-2 FILE형의 정체

표준 입출력을 사용하여 화일을 액세스할 경우에는 반드시

FILE *fp;

를 만납니다. 이 정해진 문구와 같은 FILE *XX;의 의미를 정확히 이해하고 있습니까? 포인터에 대해서는 이제 이해가 됐다고 생각합니다만 이것의 정체는 무엇일까요? 이 장에서는 화일 처리와 화일 처리에 관련된 포인터에 대해 설명하겠습니다.

▷ 5-1 화일 처리의 기본

▷ ····· 5-1-1 화일 오픈 : 범하기 쉬운 실수

화일을 처리하는 방법을 알고 있습니까?

— 우선 FILE형 변수를 선언하는 방법이 있는데 다음과 같습니다.

FILE *fp;

어떤 의미인지 알고 사용하고 있습니까?

— 글쎄요.

FILE 객체의 내용을 모르더라도 프로그램을 작성할 수 있기 때문에 이 이야기는 뒤로 돌리겠습니다.

— 리스트 5-1의 프로그램을 작성하다가 생각이 났는데 이 프로그램은 함수 open_files가 input, output이라는 전역 변수를 사용하고 있기 때문에 함수의 범용성과 독립성이 낮다고 느꼈습니다.

[리스트 5-1] 화일을 오픈하는 프로그램

```
/*
    List 5-1  화일을 오픈하는 프로그램
*/

#include <stdio.h>

FILE *input, *output;

int open_files(void)
{
    if ((input = fopen("INPUT.DAT","r")) == NULL) return(1);
    if ((output = fopen("OUTPUT.DAT", "w")) == NULL) return(1);
    return(0);
}

int main(void)
{
```

```
    int flag;

    flag = open_files();
    /* 하고 싶은 처리 */
    return(0);
}
```

좋은 지적입니다.

— 그래서 리스트 5-2와 같이 프로그램을 수정하였더니 전혀 동작하지 않게 되어 버렸습니다.

[리스트 5-2] 화일을 오픈하는(틀린) 프로그램

```
/*
   List 5-2  화일을 오픈하는(틀린) 프로그램
/*

#include <stdio.h>

int open_files(FILE *fpi, FILE *fpo)
{
    if ((fpi = fopen("INPUT.DAT", "r")) == NULL) return(1);
    if ((fpo = fopen("OUTPUT.DAT", "w")) == NULL) return(1);
    return(0);
}

int main(void)
{
    int flag;
    FILE *input, *output;

    flag = open_files(input, output);
    /* 하고 싶은 처리 */
    return(0);
}
```

이 프로그램은 문법적으로는 맞는 프로그램이지만 버그가 있는 것이 틀림없습니다.

— 부끄럽지만 어디가 잘못되었는지 전혀 짐작이 안가는데요.

우선 open_files 함수에 주목해 봅시다. fpi, fpo는 FILE형의 포인터 입니다. fopen 함수가 돌려주는 형은 FILE형의 포인터이기 때문에 표면상 문법적인 문제는 없지만 여기에 큰 실수가 있습니다.

fopen 함수의 개요는 다음과 같습니다.

fopen		
형 식	:	FILE *fopen(const char *filename, const char *mode);
프로토타입	:	<stdio.h>
기 능	:	fopen은 filename이 가리키는 문자열을 이름으로 갖는 화일을 mode 문자열이 나타내는 모드로 오픈한다.
리 턴 값	:	화일을 성공적으로 오픈한 경우에는 화일과 결부된 스트림을 제어하는 객체의 포인터를 되돌려 준다. 오픈에 실패한 경우는 NULL을 되돌려 준다.

여기서는 fpi, fpo에 어드레스를 대입하여, 즉 fpi, fpo의 값만이 바뀌었을 뿐입니다. 이제 틀린 부분을 알겠습니까?

— 아직도 잘 모르겠습니다.

포인터의 기본을 생각해 주십시오. 인자로써 받은 변수 x에 새로운 값을 대입하고 그 값을 바꾸어도 호출하는 쪽의 실인자 값은 변화하지 않습니다. 따라서, 만약 인자를 변경한 후 호출하는 쪽에 그 결과를 되돌려 주고 싶을 경우에는 인자의 주소를 포인터로 받지 않으면 안됩니다.

— input과 output은 포인터인데 그러면 된 것이 아닙니까?

참 태평이군요. 변수 x의 값을 변경해 함수로 되돌려주고 싶을 때는 x의 어드레스를 받지 않으면 안됩니다. 지금 변경하는 것은 FILE형의 변수 fpi, fpo이 아니고 FILE형의 포인터 변수 fpi, fpo입니다. 따라서 그 어드레스를 받아야 합니다.

— 포인터 변수의 어드레스입니까?

그렇습니다. 알기 쉽게 설명하면

```
int  x -> int *  x
```

이렇게 하지 않으면 안되는 것처럼

```
FILE   *fpi -> FILE * *fpi
```

라고 해야 하지 않을까요?

— 포인터의 포인터가 되는 것이군요. 이것에 관해서는 포인터의 포인터에서 배웠습니다.

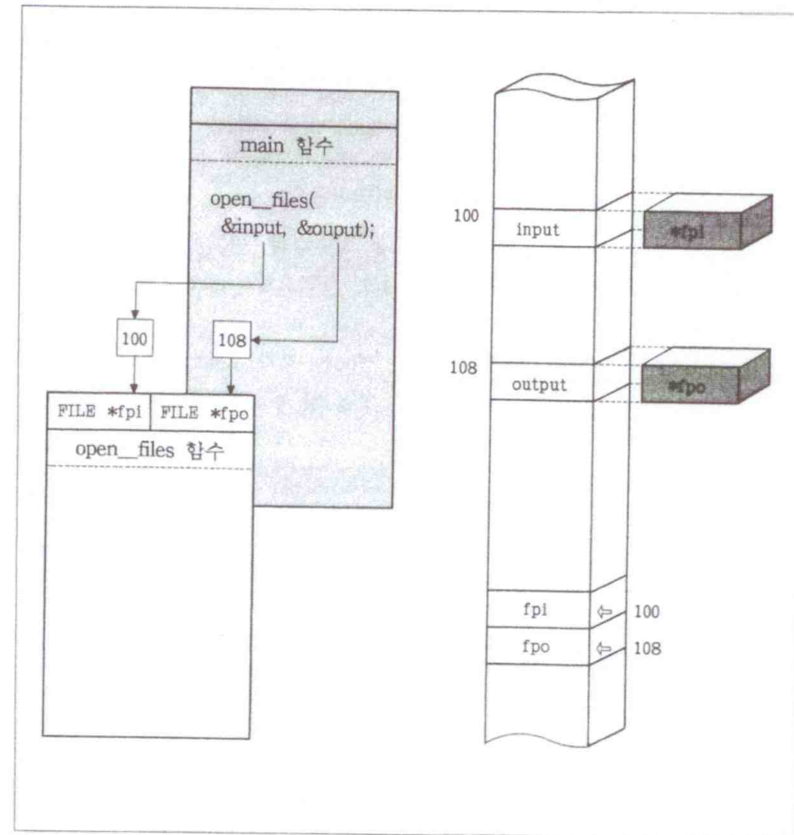
그렇습니다. 어렵게 생각할 필요가 없습니다. 『어떤 변수의 변경을 호출하는 쪽에 되돌려 주고 싶을 때에는 그것의 포인터를 받는다』라는 규칙의 <어떤 변수>가 포인터가 될 뿐입니다.

— 그렇군요. 그렇게 생각하니까 이해하기 쉽습니다.

함수 open_files를 올바르게 작성한 프로그램이 리스트 5-3입니다. 메인 함수에서 포인터 변수의 어드레스를 건네주면 됩니다.

[리스트 5-3] 화일을 오픈하는 방법

```
/*
    List 5-3 화일을 오픈하는 방법
*/

#include <stdio.h>

int open_files(FILE **fpi, FILE **fpo)
{

    if (( *fpi = fopen("INPUT.DAT", "r")) == NULL) return(1);
    if (( *fpo = fopen("OUTPUT.DAT", "w")) == NULL) return(1);
          return(0);
}

int main(void)
{
    int flag;
    FILE *input, *output;

    flag = open_files(&input, &output);
    /* 하고 싶은 처리 */
    return(0);
```

[그림 5-1] 포인터 변수의 주고받음

— 포인터 변수에 &를 붙여도 괜찮습니까?

그 질문은 전에도 했었습니다. 포인터 변수라고 해서 특별하지 않으며 변수인 이상 메모리상의 어딘가에 저장된다는 이야기를 했습니다(제1장 참조).

— 아! 생각났습니다.

한번 더 그림 5-1을 보면서 설명하겠습니다.

main 함수에서 open_files 함수를 호출할 때 input, output이라는 변수의 내용이 변경되기를 바라므로 이 변수들에 &연산자를 붙여 어드레스를 넘겨줍니다. input, output는 포인터 변수이지만 특별하게 취급할 필요는 없습니다. 그 변수가 어떤 변수이건간에 상관없이 그 변수의 어드레스를 넘겨주면 되기 때문입니다.

— 여기에서는 &input, &output는 100, 108이군요.

open_files 함수는 fpi, fpo에 그 어드레스를 받습니다. 여기에서 주의해야 할 것은 『FILE형 변수』의 어드레스를 받는 것이 아니고, 『FILE형의 포인터 변수』의 어드레스를 받는다는 것입니다.

fpi, fpo는 포인터 변수이기 때문에, *fpi, *fpo는 각각 input, output의 앨리어스가 됩니다. 따라서 프로그램이 제대로 실행되는 것을 알 수 있을 것입니다.

▶ 문제 5-1

리스트 5-3의 프로그램에 화일을 닫는 함수 close_file()를 추가하십시오.

▷ 5-2 FILE형의 정체

그러면 FILE형에 대해서 알아보겠습니다.

Turbo-C(Ver2.0)의 <stdio.h>를 살펴보면 FILE형은 리스트 5-4와 같이 정의 되어 있습니다.

[리스트 5-4] FILE형의 정의(Turbo-C Ver2.0 <stdio.h>에서)

```
/*
   List 5-4 FILE형의 정의(Turbo-C Ver2.0 <stdio.h>에서)
*/
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 */
    short           bsize;    /* Buffer size */
    unsigned char   *buffer;  /* Data transfer buffer */
    unsigned char   *curp;    /* Current active pointer */
    unsigned        istemp;   /* Temporary file indicator */
}   FILE;                     /* This is the FILE object */
```

— 구조체로써 구현되어 있는 것을 알 수 있습니다.

그렇습니다. 이것은 Turbo-C의 경우이며 다른 컴파일러에서도 같다고는 할 수 없습니다. 어쩌면 long형일지 모릅니다.

— long형 이라구요?

그렇습니다. 사실 그러한 컴파일러도 존재합니다.

— fopen 함수는 FILE형의 포인터를 되돌려 줍니다. 그리고 fclose 함수를 이용하여 처리의 끝을 선언하는군요.

그렇습니다.

— 위의 것이 앞에서 배웠던 리스트 구조로 구현되어 있는 것은 아닙니까?

왜 그렇게 생각했습니까?

— FILE형의 변수는 필요한 때만 존재하면 되므로 필요한 때에 동적으로 할당해 필요하지 않게 되었을 때—즉, 화일을 닫을 때에 그 메모리를 해제하면 되기 때문입니다.

적당한 생각입니다. 실제로 선형 리스트를 사용해 구현한 컴파일러도 있습니다. 그러나, 유감입니다만 Turbo-C의 경우, 배열을 사용하고 있습니다.

— 배열이라구요? 잠시만 기다려 주십시오. 배열은 정의할 때 크기를 미리 알고 있지 않으면 안되는데요.

그렇습니다.

— 그럼 Turbo-C에서는 오픈할 수 있는 화일의 수가 고정되어 있다는 뜻입니까.

네, 맞습니다. Turbo-C의 <stdio.h>에서 다음과 같은 선언을 발견할 수 있습니다.

```
extern FILE _Cdecl _streams[];
```

이것으로부터, FILE형의 변수가 미리 배열로 확보되어 있는 것을 알 수 있습니다.

— 어느 정도 크기의 배열입니까?

20개 입니다. 다음과 같이 정의됩니다.

```
#define OPEN_MAX 20                /* Total of 20 open files */
#define SYS_OPEN 20
```

OPEN_MAX와 SYS_OPEN을 이용하여 동시에 오픈할 수 있는 최대 화일 수를 정의 합니다.

— 그와같은 규정이 있군요.

그렇지만 이것은 C 언어의 규격이 아닙니다. ANSI 규격에서는 FOPEN_MAX라는 매크로가 <stdio.h>의 속에 정의되어 있을 것과 그것이 동시에 오픈할 수 있는 화일 갯수를 나타낼 것을 요구합니다.

프로그램은 FOPEN_MAX의 값을 참조하여 동시에 오픈할 수 있는 화일의 수를 알 수 있습니다.

OPEN_MAX나 SYS_OPEN은 관례적으로 사용되온 것이나 규격에는 어긋납니다.

— 위의 말씀은, Turbo-C는 ANSI 표준이 아니라는 뜻입니까.

그렇지만 Turbo-C++(Ver1.0)의 <stdio.h>에서는, 다음과 같이

```
#if __STDC__
#define FOPEN_MAX 18
        /* Able to have 18 files (20 - stdaux & stdprn) */
#else
#define FOPEN_MAX 20              /* Able to have 20 files */
#define SYS_OPEN 20
#endif
```

정의되어 ANSI 규격을 충족시키고 있는 것을 알 수 있습니다.

▶ 문제 5-2

FOPEN_MAX를 출력하는 프로그램을 작성하십시오. 단, FOPEN_MAX가 정의되어 있지 않은 경우에는 "FOPEN_MAX는 정의되어 있지 않습니다"라고 출력하도록 하십시오(이것을 해 봄으로써, 자신이 사용하고 있는 컴파일러가 ANSI를 따르고 있는지 판단할 수 있습니다).

— Turbo-C의 경우, 다음과 같은 배열로 정의되어 있습니다.

```
extern FILE _Cdecl _streams [];
```

fopen 함수는, 배열 _streams 중에서 사용하지 않은 요소—예를 들면 _streams[6]—가 사용되지 않았다면 _streams[6]의 어드레스, 즉 &_streams[6]을 되돌려 주는 것은 아닙니까.

그렇습니다. 실제로 표준 입력, 표준 출력 등을 나타내는 스트림은 리스트 5-5와 같이 미리 정의되어 있습니다.

[리스트 5-5] stdin의 정의(Turbo-C Ver2.0<stdio.h>)에서

```
/*
   List 5-5  stdin의 정의
*/

#define stdin     (&_streams[0])
#define stdout    (&_streams[1])
#define stderr    (&_streams[2])
#define stdaux    (&_streams[3])
#define stdprn    (&_streams[4])
```

제 6 장

함수와 포인터

▷ 6-1 함수의 포인터
▷ 6-2 함수의 포인터 배열

5장까지에서의 포인터는 모두 객체를 가리키는 것이었습니다. C 언어에서는 함수의 포인터라는 것이 있는데 함수의 포인터를 사용함으로써 범용성, 확장성이 높은 프로그램을 작성할 수 있습니다.

▷ 6-1 함수의 포인터

▷ ····· 6-1-1 프로그램의 범용성

그러면 좀 색다른 문제를 내겠습니다. 사다리꼴 공식을 사용해 적분을 하는 프로그램을 작성해 주십시오.

— 사다리꼴 공식은 알고 있습니다. 네 했습니다.

[리스트 6-1] 사다리꼴 공식에 의한 적분 프로그램(첫번째)

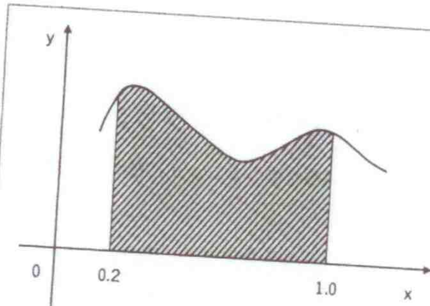
```
/*
   List 6-1 사다리꼴 공식에 의한 적분 프로그램(첫번째)
*/
#include <stdio.h>

/*  함 수 f(x)  */
double f(double x)
{
    return(x * x);
}
/* 사다리꼴 공식을 사용해 함수 f(x)를 x1부터 x2까지 n 분할로 적분하는
   함수 */
double sadari(double x1, double x2, int n)
{
    register int i;
    double   s = 0.0;
    double   step = (x2 - x1) / n;

    for (i = 0; i< n; i++)
        s += (f(x1 + step*i) + f(x1 + step*(i+1))) * step /2.0;
    return(s);
}

int main(void)
{
    int n;
    double x1, x2;

    printf("\nStart : ");
```

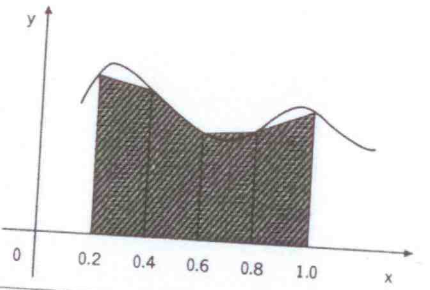
```
        scanf("%lf", &x1);
        printf("\nEnd : ");
        scanf("%lf", &x2);
        printf("\nPartition Number : ");
        scanf("%d", &n);
        printf("Integral Value = %lf", sadari(x1, x2, n));
        return(0);
}
```

■ 컬럼 6-1 ■

사다리꼴 공식에 의한 적분

오른쪽 그림의 함수를 0.2에서부터 1.0까지 적분하는 방법을 생각해 보자. 그림의 빗금친 부분의 면적을 구해 보자 사다리꼴 공식은 근사치로 적분값을 구하는 간단한 계산법으로 아래 그림과 같이 몇개의 구간으로 분할한다. 그리고 빗금친 부분을 사다리꼴의 집합이라고 간주해 그 면적을 구하는 방법이다. 프로그램은 각 사다리꼴의 면적을 더하기만 하므로 매우 간단하다.





군은 함수 $f(x) = x^2$의 적분을 하는 사다리꼴 함수를 작성했습니다. x^2이 아닌, 예를 들면 x^3+3x^2의 적분을 하고 싶은 때는 어떻게 하겠습니까.

— f 함수를 리스트 6-2와 같이 바꿔 쓰겠습니다. 그렇게 하면 사다리꼴 함수를 전혀 변경할 필요가 없습니다.

[리스트 6-2] 함수 f(x)

```
/*
  List 6-2  함수 f(x)
*/
double f(double x)
{
    return(x*x*x + 3*x*x);
}
```

그럼, 1개의 프로그램으로 함수 $f(x)= x^2$와 함수 $g(x)= x^3+3x^2$의 적분을 하고 싶을 때는 어떻게 하겠습니까.

— 사다리꼴 함수는 f 함수에 밖에 적용할 수 없기 때문에 g 함수를 적용하는 함수를 또 작성해야 합니다. 리스트 6-3과 같이 됩니다.

[리스트 6-3] 사다리꼴 공식에 의한 적분 프로그램(두 번째)

```
/*
  List 6-3 사다리꼴 공식에 의한 적분 프로그램(두 번째)
*/
#include  <stdio.h>

/* 함수 f(x) */
double f(double x)
{
    return(x * x);
}

/* 함수 g(x) */
double g(double x)
{
    return(x*x*x + 3*x*x);
}

/* 사다리꼴 공식을 사용해 함수 f(x)를 x1부터 x2까지 n 분할로 적분하는
   함수 */
double sadari(double x1, double x2, int n)
{
    register int i;
    double       s = 0.0;
    double       step = (x2 - x1) / n;
```

```
        for (i = 0; i < n; i++)
            s += (f(xl + step*i) + f(xl + step*(i+1))) * step /2.0;
        return(s);
    }

/* 사다리꼴 공식을 사용해 함수 g(x)를 x1부터 x2까지 n 분할로 적분하는
   함수 */
double sadari2(double x1, double x2, int n)
{
    register int i;
    double      s = 0.0;
    double      step = (x2 - xl) / n;

    for (i = 0; i < n; i++)
        s += (g(xl + step*i) + g(xl + step*(i+1))) * step /2.0;
    return(s);
}

int main(void)
{

    /* f(x)를 0.0부터 1.0까지 10 분할로 적분 */
    /* integral f(x) by 10 partition from 0.0 to 1.0 */
    printf("%lf\n", sadari(0.0, 1.0, 10));

    /* g(x)를 1.0부터 3.0까지 15 분할로 적분 */
    /* integral g(x) by 15 partition from 1.0 to 3.0 */
    printf("%lf\n", sadari2(1.0, 3.0, 15));
    return(0);
}
```

그러면 같은 프로그램에 함수 h(x) = sin(x)의 적분도 실행하고 싶을 때는 어떻게 하겠습니까?

— 선생님도 참 심술궂으시군요. 손들었습니다.

군의 프로그램의 sadari 함수는 전혀 범용성이 없습니다. f 함수의 적분에만 이용할 수 있을 뿐 그 외의 다른 함수의 적분은 할 수 없기 때문입니다. 다른 이름의 함수, 예를들어 g 함수나 h 함수의 적분은 할 수 없습니다.

어떻게 하면 범용성이 있는 함수를 작성할 수 있습니까?

6-1-2 함수의 포인터

수의 포인터라는 것을 사용합니다. 우선 프로그램을 보면 리스트 6-4와 같습니다. 사다리꼴 함수의 정의에 주목해 주십시오.

```
doubl  sada i(do  l  x1, dou  e x2, int n, double ( fun )(d uble  ))
```

— 제일 마지막 인자인

```
double (*func)(double x)
```

은 처음 보는데요.

이것은 인자 func가 double형의 인자를 한 개 갖는 함수의 포인터라는 것을 나타냅니다. 여기에서 ()속의 x는 프로그램을 쉽게 읽기 위하여 썼지만 생략해도 괜찮습니다.

— 좀더 자세하게 설명해 주십시오.

보통의 int형 선언과 int의 포인터형의 선언은 다음과 같이 되는데

```
int  x;
int  *p;
```

[리스트 6-4] 사다리 꼴 공식에 의한 적분 프로그램(세 번째)

```
/*
   List 6-4 사다리 꼴 공식에 의한 적분 프로그램(세 번째)
*/
#include <stdio.h>

/* 함수 f(x)  */
double f(double x)
{
    return(x * x);
}

/* 함수 g(x) */
double g(double x)
{
    return(x*x*x +
}
```

```
/* 사다리꼴 공식을 사용해 함수 func(x)를 x1부터 x2까지 n 분할로
   적분하는 함수 */
double sadari(double x1, double x2, int n, double (*func)(double x))
{
    register int i;
    double       s = 0.0;
    double       step = (x2 - x1) / n;

    for (i = 0; i < n; i++)
        s+=((*func)(x1+step*i)+(*func)(x1+step*(i+1)))*step/2.0;
    return(s);
}

int  main(void)
{
    /* f(x)를 0.0부터 1.0까지 10 분할로 적분 */
    printf("%lf\n", sadari(0.0, 1.0, 10, f));
    /* g(x)를 1.0부터 3.0까지 15 분할로 적분 */
    printf("%lf\n", sadari(1.0, 3.0, 15, g));
    return(0);
}
```

포인터의 경우는 *를 붙였습니다.

함수의 포인터의 경우도 마찬가지입니다. double형의 인자를 한 개 받아, double형을 되돌려 주는 함수 func의 선언은 double func(double)라고 합니다. 따라서 이와 같은 함수의 포인터는 식별자의 이름 앞에 *를 붙입니다.

— 그럼, double *func(double)이 되는 것이 아닙니까?

그렇게 선언하면 func는 double형의 인자를 한 개 받아, double의 포인터형을 되돌려 주는 함수가 되어 버립니다. 포인터를 되돌려 주는 함수에 관해서는 제2장에서 배웠습니다. 함수의 포인터 라는 것을 나타내려면 반드시 다음과 같이 해야 합니다.

```
double (*func)(double)
```

— 함수의 포인터를 선언하는 방법을 알겠습니다. 함수명 앞에 *을 붙여 ()로 묶으면 되는군요.

그렇습니다.

— 호출하는 쪽의 main 함수에서는

```
sadari(0.0,1.0,10,f);
sadari(1.0,3.0,15,g);
```

과 같이 함수명 f나 g를 사용하고 있는데요?

배열에서 이름만 쓰게 되면 예를 들어 a라는 배열에서 a라고만 쓰면 그 선두 요소의 포인터가 됩니다. 마찬가지로 이 경우와 같이 f나 g라고 함수명 만을 쓰면 그 함수의 포인터가 되어, 그 함수의 어드레스를 값으로 갖게 됩니다.

— 결국, 함수의 포인터를 넘겨주고 있다는 뜻이군요.

네, 그렇습니다.

```
sadari(0.0, 1.0, 10, f);
```

는 f 함수의 포인터를 넘겨주고 0.0부터 1.0까지 10 분할로 적분해 주십시오 라고 요청하는 것입니다.

호출된 사다리꼴 함수에는, 그 포인터를 func로 받습니다. 그림 6-1을 봐 주십시오.

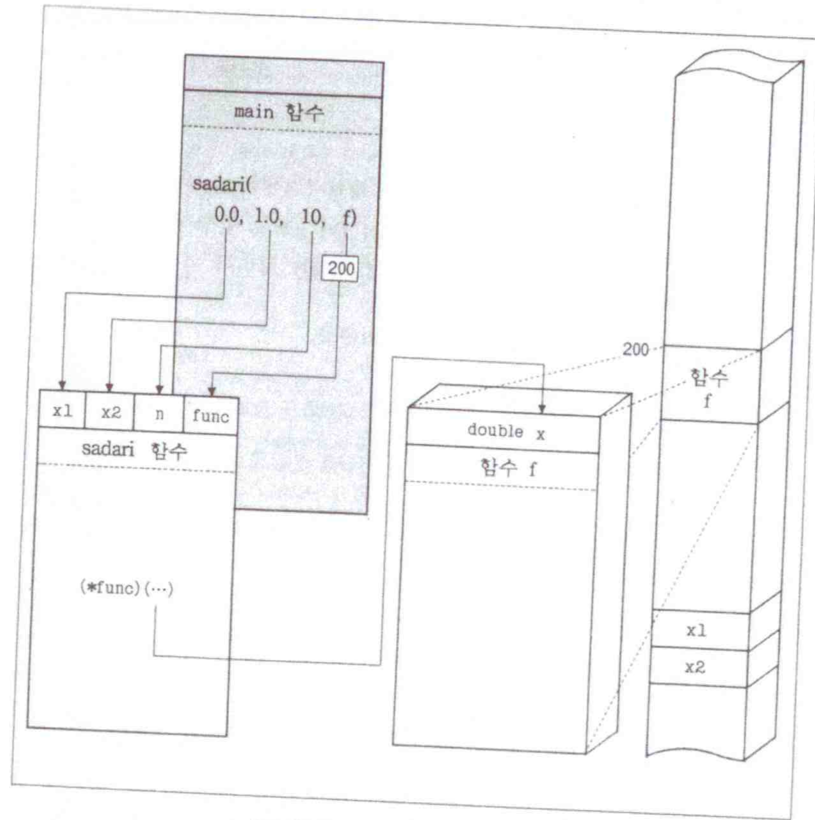
— 만약 f 함수가 200번지 부터 저장되었다고 하면, 호출하는 쪽은 200이라는 값을 넘겨주기 때문에, func는 200번지부터 저장되어 있는 함수를 가리키는 것이 됩니까?

맞습니다.

변수(객체)를 가리키는 포인터 변수에 * 연산자를 적용하면, 그 객체의 실제 내용을 나타낼 수 있듯이 함수의 포인터에 * 연산자를 붙이면, 그 포인터가 가리키는 함수를 호출할 수 있습니다. 따라서, 사다리꼴 함수의 다음 문장은

```
s += ((*func)(x1+step*i)+(*func)(x1+step*(i+1))) * step / 2.0;
```

func 포인터가 가리키는 함수를 호출하고 있습니다.



[그림 6-1] 함수의 포인터

– 잘 알았습니다. 이 경우는 f 함수를 호출하며, 만약 sadari 함수에 g 함수의 포인터를 넘겨주면, 사다리꼴 함수 중의(func*)(...)라는 식은 g 함수를 호출한다는 의미군요.

함수의 포인터를 사용함에 따라, 범용성이 높은 프로그램을 작성할 수 있습니다.

☆ 중 요 ☆

여러개의 함수를 적절하게 다룰 수 있는 범용성이 높은 프로그램을 작성할 때에는 함수의 포인터를 사용한다.

ANSI에서는 (*func)(...)은 단순히 func(...)라고 기술할 수 있도록 되어 있습니다. 따라서 리스트 6-4 프로그램은 리스트 6-5와 같이 될 수 있습니다.

– 이편이 더 알기 쉬운데요. 여기에서는 포인터라는 어려운 것을 잊어버리고, 간단하게–보다 직감적으로–사용할 수 있을 것 같은 기분이 듭니다.

[리스트 6-5] 사다리꼴 공식에 의한 적분 프로그램(네 번째)

```
/*
    List 6-5 사다리꼴 공식에 의한 적분 프로그램(네 번째)
*/
#include <stdio.h>

/*  함수 f(x)  */
double f(double x)
{
    return(x * x);
}

/* 함수 g(x) */
double g(double x)
{
    return(x*x*x + 3*x*x);
}
/* 사다리꼴 공식을 사용해 함수 func(x)를 x1부터 x2까지 n 분할로
   적분하는 함수 */
double sadari(double x1, double x2, int n, double func(double x))
{
    register int i;
    double      s = 0.0;
    double      step = (x2 - x1) / n;

    for (i = 0; i < n; i++)
        s += (func(x1+step*i) + func(x1+step*(i+1))) * step /
        2.0;
    return(s);
}

int main(void)
```

```
{
    /* f(x)를 0.0부터 1.0까지 10 분할로 적분 */
    printf("%lf\n", sadari(0.0, 1.0, 10, f));

    /* g(x)를 1.0부터 3.0까지 15 분할로 적분 */
    printf("%lf\n", sadari(1.0, 3.0, 15, g));
    return(0);
}
```

▶ 문제 6-1

본문에서 주어진 sadari 함수를 사용해 sin 함수나 cos 함수의 적분을 해보십시오.

▷ 6-2 함수의 포인터 배열

▷ ····· 6-2-1 메뉴 방식 프로그램

어떤 경우에는 메뉴를 선택해, 이에 따라 다양한 처리를 하고 싶을 때가 자주 있습니다.

— 그렇습니다. 일반 응용 프로그램에서는 반드시 필요합니다.

이와같은 프로그램을 메뉴 방식 프로그램이라고 합니다. 리스트 6-6이 그 예입니다. display 함수로 메뉴를 표시합니다.

[리스트 6-6] 메뉴-선택 프로그램(첫번째)

```
/*
    List 6-6 메뉴-선택 프로그램(첫번째)
*/
#include <stdio.h>

void menue1(void)                    /* 메뉴-1의 처리 */
{
    printf("Menu 1 is selected\n");
}

void menue2(void)                    /* 메뉴-2의 처리 */
{
    printf("Menu 2 is selected\n");
}

void menue3(void)                    /* 메뉴-3의 처리 */
{
    printf("Menu 3 is selected\n");
}

void display(void)                   /* 메뉴 선택 화면 출력 */
{
    printf("1 ...  Menu 1\n");
    printf("2 ...  Menu 2\n");
    printf("3 ...  Menu 3\n");
```

```
        printf("0 ...  End\n");
        printf("input number.");
    }

    int main(void)
    {
        int  selected;                      /* 선택된 번호 */

        do {                                /* 메뉴 선택 화면 출력 */
           display();
           scanf("%d", &selected);          /* 번호 입력 */
           switch (selected) {
              case 1:                       /* 메뉴-1 */
                     menue1();
                     break;
              case 2:                       /* 메뉴-2 */
                     menue2();
                     break;
              case 3:                       /* 메뉴-3 */
                     menue3();
                     break;
              }
        } while (selected);
        return(0);
    }
```

― scanf 함수로 메뉴의 번호를 입력하는군요.

그렇습니다. 그리고

```
switch (selected) {
    /* . . . */
}
```

로 선택된 번호에 따라, 이에 따른 메뉴를 실행하는 함수를 호출합니다. 여기에서 잘 생각해 주십시오.

만약, 메뉴가 100개 있다고 하면 어떻게 하겠습니까.

― case를 100개 나열할 수 밖에 없지요. 배열처럼 함수를 호출할 수 있다면 편리할 텐데요.

실제로 그렇게 할 수 있습니다. 함수의 포인터를 사용하면 되지요.

▷ …… 6-2-2 함수의 포인터 배열

― 함수를 배열처럼 호출하는 방법

함수를 배열처럼 호출하도록 변경한 프로그램이 리스트 6-7입니다.

다음은

```
typedef void (*mfunc)(void);
```

「인자와 리턴값이 모두 void인 함수의 포인터」라는 수형을 mfunc라고 정의하고 있습니다.

― 이것은 수형에 이름을 붙인 것이지요?

그렇습니다. 다음은

```
mfunc menu [] = { menue1, menue2, menue3 };
```

[리스트 6-7] 메뉴-선택 프로그램(두 번째)

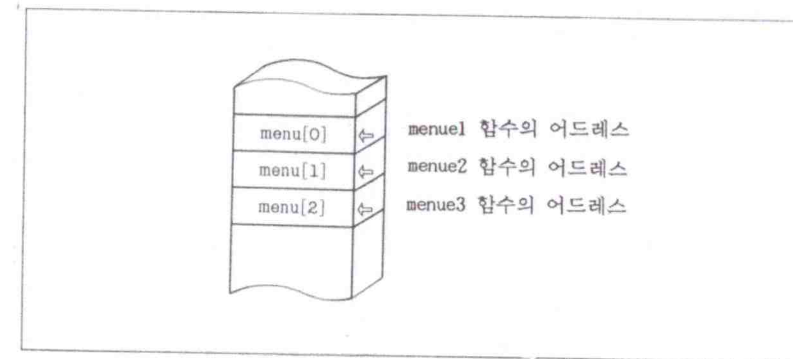
```
/*
    List 6-7 메뉴-선택 프로그램(두 번째)
*/
#include <stdio.h>

/* 인자와 리턴값이 void인 함수의 포인터, mfunc형을 작성한다 */
typedef void (*mfunc)(void);

void  menue1(void)                          /* 메뉴-1의 처리 */
{
    printf("Menu 1 is selected\n");
}

void  menue2(void)                          /* 메뉴-2의 처리 */
{
    printf("Menu 2 is selected\n");
}

void  menue3(void)                          /* 메뉴-3의 처리 */
{
    printf("Menu 3 is selectted\n");
}

void display(void)                          /* 메뉴 선택 화면 출력 */
```

```
{
    printf("1 ... Menu - 1\n");
    printf("2 ... Menu - 2\n");
    printf("3 ... Menu - 3\n");
    printf("0 ... End \n");
    printf("Input Number");
}

/* mfunc형의 배열 */
mfunc menu[] = { menue1, menue2, menue3 };

int main(void)
{
    int     selected;                       /* 선택된 번호 */

    do {                                    /* 메뉴 선택 화면 출력 */
        display();
        scanf("%d", &selected);             /* 번호 입력 */
        if (selected >=1 && selected <= 3
              menu[selected-1]();           /* 처리 실행 */
    } while (selected);
    return(0);
}
```

mfunc형의 크기 3인 배열을 선언하고 있습니다.

배열 menu의 각 요소에는, 초기값으로 함수의 포인터가 대입됩니다. 구체적으로 설명하면 menu[0]에는 menue1 함수의, menu[1]에는 menue2 함수의 menu[2]에는 menue3 함수의 어드레스가 들어갑니다.

— 그렇군요.



[그림 6-2] 함수 포인터의 배열

한 번 더 정리 하겠습니다. 예를들면 menu[0]에 주목해 보십시오. 이 변수의 수형은 무엇입니까?

— 인자, 리턴값 모두 void인 함수의 포인터입니다.

그러면 그 값은 무엇입니까?

— menue1 함수의 어드레스입니다.

아래의 x는 인자를 갖지 않은 함수의 포인터입니다.

```
(*x)();
```

무엇을 실행합니까?

— 함수의 포인터에 (*)를 붙이면 그 포인터가 가리키는 함수를 호출하게 됩니다. 따라서, 이 경우는 함수 x를 호출합니다.

이것은 간단하게

```
x();
```

라고도 쓸 수 있습니다. x를 menu[0]이라고 바꿔놓으면 어떻게 될까요?

— 그러면

```
(*menu [0])();
```

또는

```
menu[0]();
```

라고 써서, menu[0]이 가리키고 있는 함수를 호출할 수 있습니다.

맞습니다. 이제 함수의 포인터를 이해하게 됐군요. 그것이 배열이 되었기 때문에 잘 몰랐던 것 뿐입니다.

— menu[i]()라 하면, menu[i]가 가리키는 함수를 호출하게 되는군요. 따라서 함수를 배열처럼 호출할 수 있게 되는군요.

이와 같이, 함수의 포인터는 메뉴 방식 프로그램을 작성할 때 그 위력을 발휘하게 됩니다. 게다가, 처음에 switch 문을 사용한 것과 비교해 보면, 매우 간단하게 기술할 수 있습니다.

제 7 장

8086 CPU 특유의 포인터

▷ 7-1 세그먼테이션의 개념
▷ 7-2 near, far, huge 포인터
▷ 7-3 메모리 액세스 라이브러리의 작성
▷ 7-4 far 히프 영역의 사용