Pointer 에 관하여

심상돈

shimsd@hanmail.net

2002 년 8 월 23 일

- 2002년 10월 7일 **함수 포인터** 추가
- 2002년 10월 24일 **포인터의 깊은 곳** 추가
- 2003년 8월 11일 **부록: 분할 컴파일** 추가

이 글은 pointer 를 자세하게 알고 싶어하는 사람들은 위한 안내서입니다.

pointer 는 누구나 한번쯤 고민하게 만들고 힘들게 만드는 개구장이입니다. 이런 천덕꾸러기를 자신의 것으로 만들기 위한 도움이 될 것입니다.

여기서는 pointer 에 대해서 기본적인 개념에서부터 활용 및 고급 기술까지 설명과 더불어 많은 예제가 있습니다.

차례

---

1 장. 포인터를 시작하기 전에...

안녕하십니까? C 를 공부하면서 가장 힘들었고 실제적으로 사용하는데 있어서 가장 문제가 많았던 포인터에 대해서 강좌를 시작해보려 하는 심상돈 입니다.

포인터 강좌를 시작하기 전에 염두해 두어야할 몇가지를 적어보겠습니다.

포인터를 공부하기 전에 충분한 C 의 문법을 익혀야 합니다. 이 글을 읽는 분들은 숙제 때문에 하는 공부가 아닐것이라는 가정하에 어느정도는 스스로 코딩을 해보고 프로그램을 짜봤지만, 포인터에 대해서 정확한 개념이 없어서 고생을 하실것이라 생각합니다.

포인터를 공부하기 전에, 얼마만큼 C 를 알려고 노력했는지 스스로 생각해 보시기 바랍니다. C 가 만능이라는 말을 하고 싶은건 아닙니다. C 가 다른 언어를 배우기 위한 기본적인 것이다라고 하고싶은건 더더욱 아닙니다.

왜 C 를 선택했는지 대부분의 배우시는 분들은 잘 모를것입니다.

하지만, 일단 C 언어라는 것을 시작했으면 걱정마시고 C 라는 넘을 자신의 부하로 삼으세여. 여러분이 시키는 일을 전혀 군말 없이 해 줄 수 있는 그런 충성스런 부하로 말이져.

그런 부하로 만들기 위해서 C 가 가지고 있는 능력중 가장 뛰어난 포인터를 알아야 합니다. 자.... 서두는 그냥 이정도로만 하고 포인터 강좌로 들어가겠습니다.

---

2 장. 변수에 관하여

차례

2.1. 일반 변수

변수란 무엇일까여? 어떤 값을 저장하는 것 정도로 알고 계실 겁니다. 훌륭한 대답이며 정답입니다.

그럼 변수를 하나만 만들어 보고 그 변수의 특성과 성질을 알아봅시다.

```
int number = 1000;
```

자 number 라는 이름의 int 형 변수가 만들어 지고 그 안에 1000 을 넣었습니다.

여기서 잠깐!! int 형에 대해서 자칫 지나칠 수 있을지 모르는 것에 대해서 언급을 하겠습니다. int 형은 2 바이트 또는 4 바이트 어떤 것일 수도 있습니다. 그것은 운영체제나 컴파일러에 따라서 다릅니다.

운영체제가 16 비트 체제이면 int 형은 16 비트(2 바이트)입니다. 그런 운영체제로는 DOS 가 있습니다. Windows 95, 98 을 사용한다 하더라도, DOS 창에서 Borland 계열의 컴파일러(Turbo-C, Borland-C)를 사용하면 int 는 2 바이트 입니다.

운영체제가 32 비트 체제이면 int 형은 32 비트(4 바이트)입니다. 이런 운영체제로는 Windows NT, 2000, Unix, Linux 등등이 있습니다. 이 운영체제 에서는 int 가 4 바이트 입니다.

여기서 포인터 역시 마찬가지 입니다. 포인터의 크기(sizeof 연산자로 확인 해보세여) 역시 int 의 크기와 동일합니다. 물론 이것은 16, 32 비트 체제에서만 그렇고 64 비트 체제의 운영체제에서는 아직 정확히 표준화 되지 않았으므로 그냥 넘어갑니다.

자... 그러면 위에서 number 라는 변수가 어디에 생성이 되고 어떻게 이루어져 있는지 알아봅시다.

```
0x2000
  +---+---+---+---+
  |   |   | 10| 00 |  number
  +---+---+---+---+
```

네모칸 하나가 한 바이트라 가정하고, 32 비트 운영체제라고 가정하에 설명을 하겠습니다. 그래서 int 가 4 바이트로 되어 있져?

number 라는 변수를 선언했을때 우리는 위와 같은 모습을 상상하면 됩니다.

number 라는 변수의 메모리상에서 번지값은 0x2000 이라고 가정하겠습니다. :) 그리고 number 는 그 번지를 참조할 수 있는 이름이며 그 안에는 1000 이라는 값이 들어가 있습니다.

자... 그러면 우리는 이 변수를 사용해야 할 것입니다. 변수를 만들었으면 그 변수를 사용할 수 있어야 합니다.

number 변수를 사용할 수 있는 방법은 2 가지가 있습니다.

```
 number ->    1000
&number -> 0x2000
```

그냥 number 라고 사용을 하면 그 메모리 안에 들어있는 값을 말합니다. 그래서 1000 이 되는 것이고,

앞에 & 이 붙어있는 것은 number 라는 변수의 메모리 번지를 말하는 것입니다. 즉 0x2000 이라는 주소값을 나타내는 것이져..

그럼 실제로 다음과 같은 코드를 작성해서 테스트를 해보자구여..

```
#include <stdio.h>

int main(void)
{
	int number = 1000;

	printf(" number = %d₩n", number );
	printf("&number = %d₩n", (int)&number );

	/* makes compiler happy :) */
	return 0;
}
```

컴파일 해서 실행을 해보면

```
 number = 1000
&number = 37813784
```

이렇게 나오네여 제 컴에서. 물론 &number 의 값은 컴파일 하는 환경마다 다 다를 수 있습니다. 어쨌든 number 라는 변수를 이용할 때 이 2 가지의 값을 이용할 수가 있다는 것을 말하려고 하는 거니까여.

2.2. 단일 포인터

자... 그러면 이제는 포인터 변수를 하나 만들어 봅시다.

```
int *p = &number;
```

위의 number 변수를 이용해서 p 라는 포인터 변수에 number 변수의 번지를 저장합니다.
그럼 p 라는 포인터 변수도 number 처럼 그림으로 나타내 봅시다.

```
0x3000
  +---+---+---+---+
  |       0x2000      |  *p
  +---+---+---+---+
```

위와 같이 p 라는 포인터 변수가 0x3000 번지에 만들어 지고 그 안에 number 의 주소값인 0x2000 이란 값이 저장되어 있습니다.

이번엔 p 라는 변수를 사용하는 방법이 몇개가 있을까여? 일반 변수는 2 가지였지만 이 포인터 변수는 3 가지 입니다.

```
 p -> 0x2000
&p -> 0x3000
*p -> 1000
```

그냥 p 만 사용하면 p 라는 변수 안에 있는 값 자체를 뜻하므로 number 의 번지값인 0x2000 이 됩니다.

&p 는 p 의 번지값을 말하는 것이져? 그래서 0x3000 이 되는 거져...

마지막으로 *p 는 무엇을 뜻할까여? *(p) 이렇게 생각을 하시고, *(0x2000) 이렇게 됩니다.
여기서 *은 "거기로 가라" 라는 뜻입니다. 즉 0x2000 번지로 가서 그 값을 읽어라...
이렇게 되는 거져.. 0x2000 번지로 가면 무엇이 있져? 네 number 의 값인 1000 이 있습니다.

어라.....이해가 힘드나여? 그럼 number 와 p 를 한번에 그림으로 나타내보져...

```
0x2000                              0x3000
  +---+---+---+---+                     +---+---+---+---+
  |         1000      | number          |        0x2000     | *p
  +---+---+---+---+                     +---+---+---+---+

  number -> 1000                          p -> 0x2000
 &number -> 0x2000                        &p -> 0x3000
                                          *p -> 1000
```

이 그림을 머리 속에다가 확실히 입력해 두시기 바랍니다. 이 그림만 확실히 외우고 있으면 어떤 포인터가 나오더라도 이 그림에서 벗어나질 않습니다.

그럼 지금까지 배운 것을 코드로 짜보고 확인도 해볼까여?

```
#include <stdio.h>

int main(void)
{
	int number = 1000;
	int *p = &number;

	printf(" number = %d₩n", number );
	printf("&number = %d₩n", (int)&number );

	printf(" p      = %d₩n", (int)p );
	printf("&p      = %d₩n", (int)&p );
	printf("*p      = %d₩n", *p );

	return 0;
}
```

위의 소스를 컴파일 해서 실행 시켜보면.. 다음과 같은 결과가 나옵니다.

```
 number = 1000
&number = 37813784
 p      = 37813784
&p      = 37813780
*p      = 1000
```

물론 주소값들은 컴퓨터 마다 다를 수 있습니다. 위의 코드를 그대로 복사하지 마시고 그대로 보고 치세여.. 직접 쳐서 실행 해보는게 중요합니다. :)

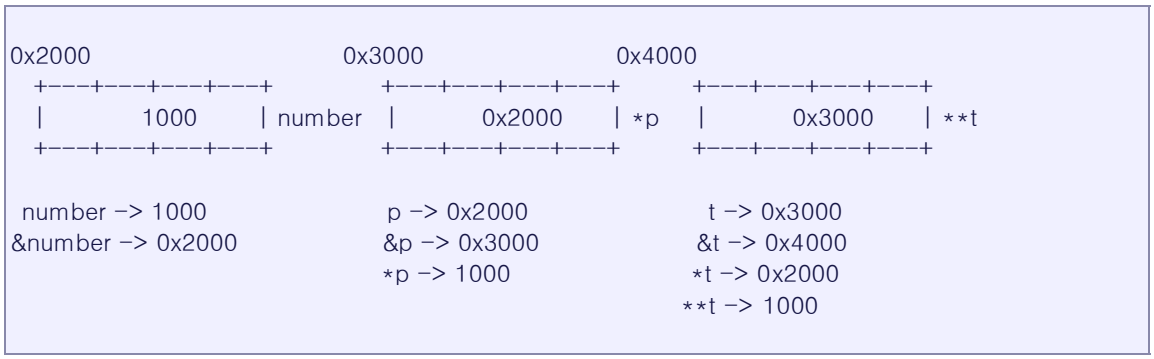
포인터에 대한 많은 강좌들을 보면 너무 어려운 표현들에 대해서 설명을 하려고 노력을 하는 모습을 많이 봐왔는데 그러한 어려운 표현들도 결국은 위의 그림을 토대로 하나씩 따라가다 보면 전혀 어려울게 없다는 것을 꼭 명심하기 바랍니다.

2.3. 이중 포인터

이번엔 이중 포인터 변수를 하나 선언해 보겠습니다.

```
int **t = &p;
```

이렇게 t 라는 더블 포인터 변수를 선언하고 p 라는 포인터의 번지값을 저장했습니다. 그럼 이 t 라는 더블 포인터 변수를 그림으로 다시 나타내 보겠습니다.



여기에서 *t 의 값은 *(0x3000)이 되겠져? 0x3000 번지로 가서 그 값을 읽으라는 것이니까 0x3000 번지에 가면 0x2000 이라는 값이 있습니다. 그래서 *t 의 값은 0x2000 이 되는 겁니다.

이번엔 **t 의 값을 따져 보져.. *(*(t)) 이렇게 됩니다. 따라서 *(0x2000)이 되고 역시 0x2000 번지로 가서 그 값을 읽어 보면 1000 이 있져? 따라서 **t 의 값은 1000 이 되는 것입니다.

이렇게 *(별표)는 거기로 가라...라는 뜻입니다. 그래서 가리키다라는 뜻의 pointer 라는 단어를 사용 하는 거져..

나중에 구조체에서 -> 가 나오게 되는데 이것 역시 "거기로 가라"는 뜻입니다. 이것에 관해서는 나중에 다시 자세하게 설명하겠습니다.

이렇게 해서 몇중 포인터 이든지 위의 그림에서 설명 했듯이 하나하나 따라가 보면 어렵지 않게 포인터가 가리키는 값을 알아낼 수 있습니다.

그렇다면 과연 이 포인터는 왜 사용을 하는 것일까여? 그냥 변수의 이름만으로도 충분히 변수를 사용할 수 있는데 왜 머리아프게 포인터라는 것을 사용해서 어지럽게 변수를 사용하는 걸까여? 그래서 다음장은 포인터의 쓰임에 관해서 설명하겠습니다.

---

3 장. 포인터의 쓰임

차례

3.1. 문자열

포인터가 가장 많이 사용되는 곳은 아마도 문자열일 것입니다. 문자열은 그 자체가 포인터이며 배열과도 많이 혼용해서 사용됩니다. 하지만 문자열은 알고 있는 것보다도 더 심오한 내용이 숨어있습니다.

그래서 첫번째로 문자열에 대해서 알아보도록 하겠습니다.

```
printf("Hello, world!₩n");
```

위의 코드는 아주 많이 봐왔을 것입니다. Hello, world! 라는 문자열을 화면에 출력하는(stdout 에 출력하는) 것입니다.

하지만 이 문장을 정확하게 해석을 할 줄 알아야 합니다.

"Hello, world!₩n" 는 문자열입니다. " "(큰 따옴표;double quote)로 둘러싸인 문자들을 문자열이라고 하지여..

여기서 " " 이 하는 일을 알아봅시다. " " 이 하는 일은 그 안에 있는 문자들을 RAM 의 CODE 영역에 Read only 속성을 가지고 저장을 합니다. 그리고 저장된 영역의 첫번째 주소값을 리턴해줍니다.

그래서 printf("Hello, world!₩n"); 이 문장은, 다음의 그림과 같이 되는 것입니다.

```
0x1234 (RAM 의 CODE 영역)
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |'H'  'e'   'l'   'l'   'o'   ','   ' '  'w'  'o'   'r'   'l'  'd'   '!'  ₩n  ₩0 |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

 printf( 0x1234 );
```

이렇게 되는 거지여...

그러면 printf() 함수의 인자가 숫자값이 되네여? -_-; 그렇다면 printf() 함수의 원형(prototype)을 알아봅시다.

```
int printf( const char *format, ... );
```

이렇게 됩니다. 즉, printf()함수의 첫번째 인자가 바로 포인터인 것입니다. 그래서 printf()함수가 내부적으로 하는 일을 간략하게 적으면,

첫번째 인자로 들어온 주소값으로 이동해서 그 안에 있는 값을 아스키코드로 출력을 하고, 다음 주소로 이동을 한 다음 또 찍고... 이것을 반복하다가 이동한 주소에 0(NULL)이 오면 문자를 출력하는 것을 중단하고나서 지금까지 출력한 문자의 갯수를 리턴해주는 것이 바로 printf() 함수입니다.

위의 그림대로 하자면,

0x1234 번지로 이동을 해서 그 안에 'H' 이 있져? 그래서 H 를 출력하고 다음 번지인 0x1235 로 이동하겠져.. 그 안에 있는 'e' 를 출력하고... 이것을 반복하다가 마지막에 있는 '₩0'(NULL)을 만나게 되어서 출력하는 것을 중단하게 되는 겁니다.

그러면 이해를 돕기 위해서 간단한 코드를 작성해 봅시다.

```
#include <stdio.h>

int main(void)
{
	printf("address = %d₩n", (int)"Hello, world" );
	return 0;
}
```

위의 코드를 컴파일 하여 실행을 해보니,

```
address = 4198464
```

이렇게 나오네여.. 출력된 값은 컴퓨터 마다 다를 수 있습니다. 중요한건 문자열 자체가 어떤 값이라는 거져.. 위에서 말했져? RAM 의 CODE 영역에 문자열을 저장하고 그 첫 번지 값을 리턴한다.. 그리고 그 문자열이 저장된 영역은 Read only 속성을 갖는다. 그래서 임의로 값을 바꾸려 하면 segmentation fault 를 일으킵니다.

문자열을 사용할때 printf("Hello"); 처럼 함수의 인자에 곧바로 문자열을 사용하기도 하지만 포인터 변수를 사용할 때도 많이 있져? 전에는 이것때문에 많이 헷갈렸을지도 모르지만 이제는 그렇지 않아도 됩니다. " "가 하는 일에 대해서 위에서 공부를 했자나여..

다음의 예제를 봅시다.

```
#include <stdio.h>

int main(void)
{
	char *s = "Hello, world!₩n";

	printf( s );
	return 0;
}
```

위의 예제를 컴파일 해서 실행시켜 보지 않아도 결과가 보이져? 차근차근 분석해 봅시다.

char *s = "Hello, world!₩n";

s 라는 포인터 변수를 선언하고 "Hello, world!₩n" 이라는 문자열을 RAM 의 CODE 영역에 저장한 다음에 그 첫번째 주소값을 리턴하여 s 에 그 주소값을 대입하라. 이거져?

그리고 printf(s); 이렇게 하면 printf() 함수는 s 가 갖고 있는 그 주소로 이동을 하고 화면에 출력을 하게 되는 겁니다.

조금 변형을 해서 다음과 같이 해도 됩니다.

```
#include <stdio.h>

int main(void)
{
	char *s = NULL;
	s = "Hello, world!₩n";

	printf( s );
	return 0;
}
```

전의 것과는 다른게 없습니다. 다만 문자열이 주소값을 리턴한다는 사실을 정확히 알고 있으면 포인터 변수에 문자열을 대입하는 것 같은 모양의 코드를 사용한다 하더라도 실제적으로 그 포인터에 값이 들어가는게 아니라 RAM 의 CODE 영역에 문자열이 저장이 되고 그 주소값만 포인터에 대입한다는 것을 알수 있으니까여..

문자열을 사용하기 위해서 선언한 포인터에 대해서 자세히 알아봅시다.

char *s = "Hello, world!₩n";

char *s; 에서 *s 앞에 붙은 char 라는 것은 무엇을 의미 할까여? 첫번째 강좌에서는 int *p; 를 사용했는데 *p 앞에 붙은 int 는 무슨 의미??

포인터 변수 앞에 붙은 type 은 포인터가 가리키고 있는 곳에 저장된 값의 type 을 뜻하는 것입니다.

int *p; 에서는 p 가 가리키고 있는 곳에 저장되어 있는 값이 int 형이라는 뜻이고, char *s; 에서는 s 가 가리키고 있는 곳에 저장되어 있는 값이 char 형이라는 뜻이져.

문자열은 문자들의 집합이라는 것은 다들 아실테고, 문자를 char 형으로 사용하는 이유도 아실테지만 노파심에 다시 한번 설명을 하져.

char 형은 어떤 운영체제이든지 상관 없이 1 바이트 입니다. 1 바이트는 8 비트. 따라서 char 형의 범위는 -128 ~ 127 입니다. unsigned char 형으로 하면 0 ~ 255 가 되져..

각 컴퓨터마다 아주 조금씩은 다를 수 있지만 0 ~ 127 번까지의 아스키 코드는 공통으로 사용됩니다. 따라서 char 형의 양의 정수 범위 만큼은 어떤 컴퓨터라도 공통적인 아스키 코드를 사용한다는 말이져. 그래서 0 ~ 127 번까지의 아스키 코드 테이블에는 영문자와 숫자 특수 문자들이 선언되어 있고 char 형의 변수를 이용해서 문자들을 다룰 수 있습니다.

물론 int, long 형으로도 다룰 수 있지만 쓸데없는 메모리 낭비가 되겠져? 가장 작은 단위인 char 형으로 문자를 훌륭하게 다룰 수 있습니다.

다시 포인터로 넘어가서, 문자열을 사용하는데 있어서 자칫 실수 하기 쉬운 것에 대해서 알아보져.

```
#include <stdio.h>

int main(void)
{
	char *s = "Hello, world!₩n";

	*s = 'T';

	printf( s );
	return 0;
}
```

위의 코드를 컴파일 하고 실행하기 전에 분석부터 해봅시다.

char *s = "Hello, world!₩n";

이제 지겹져? -_-; 같은게 계속 나와서...

그럼 그 다음 줄..

*s = 'T';

이것은 무슨 뜻일까여? s 의 주소로 가서(*이 있기 때문에 그 주소로 가야합니다.) 'T' 값을 저장해라. 이런 뜻이져? 설마 'T' 가 무슨 뜻인지 모르지는 않겠지만.. 'T' 는 T 의 아스키 코드값을 뜻합니다.

즉, s 가 가리키는 첫 주소에 있는 값은 무엇이져? 네.. 'H' 져? 문자열의 첫번째 주소를 저장하고 있을테니까여..

그런데 그 'H' 라는 값을 'T'로 바꾸고자 위의 코드를 쓴겁니다.

그리고 printf(s); 로 출력..

문법에는 전혀 문제가 없습니다. 하지만 컴파일하고 실행을 해보면 문제가 발생합니다.

" "로 둘러싸인 문자열이 저장되는 곳이 어디라고 했었져? RAM 의 CODE 영역이라고 했었져? 근데 중요한건 " "로 둘러싸인 문자열을 저장한 그곳은 Read only 영역입니다. 따라서 쓰기를 하게 되면 에러가 발생합니다.

'H'를 'T'로 바꾸기 위해서 *s = 'T'; 라는 코드를 썼었져? 읽기 속성인 곳에 'T'를 쓰려고 했기 때문에 실행시에 에러가 발생하는 겁니다.

이것이 " "로 둘러싸인 문자열의 특성입니다.

자.. 그렇다면 바꿀 수 있는 문자열을 선언하는 방법은 없을까여? 바로 배열을 이용하면 가능합니다. 다음의 코드를 봅시다.

```
#include <stdio.h>

int main(void)
{
	char s[ ] = "Hello, world!₩n";

	*s = 'T';

	printf( s );
	return 0;
}
```

지금의 코드와 바로 전의 코드와 다른점은 char *s -> char s[] 이렇게 바뀐 겁니다.
이렇게 배열을 사용하게 되었을때 어떻게 되는 건지 자세하게 알아봅시다.

char s[ ] = "Hello, world!₩n";

여기에서 s 는 크기가 정해져 있지 않은 배열입니다. " "으로 둘러싸인 문자열은 위에서 설명한 것 처럼 일을 하고나서, s 라는 배열을 동적으로 문자열의 크기만큼 잡은다음에 read only 속성을 갖고 있는 문자열을 s 배열에 복사를 합니다. 즉, 문자열이 2 번 저장되게 되는 거져. read only 속성을 갖는 문자열과 read/write 다 가능한 s 배열에 있는 문자열..

s 배열의 속성은 read/write 이고, " " 문자열은 read only 입니다.

s 배열에 대해서 조금 더 알아보져.. 위의 자료들을 그림으로 자세하게 나타내 보겠습니다.

```
0x1234 (RAM 의 CODE 영역 read only 속성)
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |'H'  'e'   'l'   'l'   'o'   ','   ' '   'w'  'o'   'r'   'l'  'd'   '!'   ₩n ₩0 | " " 문자열
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

0x2000 (RAM 의 DATA 영역 - stack)
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  | 'H'  'e'   'l'   'l'   'o'  ','  ' '   'w'  'o'   'r'   'l'   'd'  '!'   ₩n ₩0 | s[ ]
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

위의 문자열은 " " 로 둘러싸인 문자열이고 아래의 문자열은 s 배열에 복사된 문자열 입니다.

여기서 s 라는 배열의 이름을 그대로 사용을 하면 s 는 포인터가 됩니다. 즉,

s -> 0x2000

이렇게 된다는 겁니다.

이 뜻은 배열 s 를 포인터로도 사용할 수 있다는 거져. 사실 포인터와 배열은 뗄 수 없는 관계에 있지만 더 자세한 배열과 포인터와의 관계는 나중에 다시 설명하도록 하겠습니다.

자.. 그렇다면 *s 의 값은 얼마일까여?

*(0x2000) 이 되져? 0x2000 번지에는 'H' 값이 있져? 그러므로 *s 의 값은 'H' 가 됩니다.

다시 소스 코드로 돌아가서

*s = 'T';

이렇게 하면 s 배열의 첫번지에 'T'값을 넣으라는 것이 되겠져? s 배열은 분명히 read/write 가 다 가능하므로 이 코드를 컴파일하고 실행하면 H 가 T 로 바뀌어서 출력되는 모습을 볼 수 있습니다.

위에서는 배열의 크기가 정해져 있지 않은 동적인 배열을 이용했지만, 실제적으로는 배열의 크기를 정해주는 경우가 많습니다.

하지만 포인터와 배열의 약간 다른 이해가 필요하다는 것을 보여주기 위해서 다음과 같은 코드를 예로 들어보겠습니다.

```
char *s = NULL;
s = "Hello, world!₩n";    ----- 1

char s[15];
s = "Hello, world!₩n";    ----- 2
```

1 번의 경우는 포인터 변수에 문자열이 저장된 곳의 첫번지 값을 저장하는 코드이고, 2 번의 경우는 배열을 잡아 놓고 그 배열의 이름 s 가 포인터라 했으니 그 포인터에 문자열이 저장된 곳의 첫번지 값을 저장하는 코드라 생각하고 한건데...

1 번은 전혀 문제가 없는 코드입니다. 하지만 2 번은 문제가 있는 코드입니다. 왜냐면 배열의 이름을 가지고 나타내는 포인터는 읽기만 가능합니다. 쓰기는 안되는 겁니다.

s 라는 배열은 예를 들어서 0x3000 번지에 할당이 되었다고 가정하면,

```
 s -> 0x3000
&s -> 0x3000
```

이렇게 됩니다. s 라는 이름을 사용한 포인터나 &s 가 나타내는 값이나 같게 됩니다. 결국은 &s[0]; 이것과도 같은 의미가 됩니다.

이런 것들은 변하는 것들이 아닙니다. 따라서 포인터 변수와 배열의 이름을 사용해서 나타내는 포인터와는 꼭 구분을 할 줄 알아야 합니다.

문자열에서 배열을 사용하지 않고 왜 포인터를 사용할까여? 배열을 사용하면 읽기와 쓰기도 다 가능한데 왜 포인터를 사용하는 걸까여?

그렇다면 그 이유를 간단하게 보여줄 수 있는 코드를 하나 작성해 보져..

```
#include <stdio.h>

int main(void)
{
	char buffer[256];

	printf("Input string: ");
	fgets( buffer, 256, stdin );

	printf( buffer );
	return 0;
}
```

위의 코드는 256 개의 문자를 저장할 수 있는 buffer 라는 배열을 잡아서, 사용자로부터 문자열을 입력받아서 buffer 에 저장한 다음에 printf() 함수로 출력을 하는 겁니다.

여기서 배열의 크기를 256 으로 잡았는데, 사용자가 얼만큼의 문자열을 넣을지 모르므로 가능한 크게 잡아놓은 거지여.

위의 예제에서 buffer 를 포인터로 선언하고 문자열을 저장하기 위해서 동적 메모리 할당을 이용하여 저장을 하는 것은 오히려 메모리 낭비가 되지만.. 저장해야 할 문자열이 많다면 상황은 달라집니다.

예를 들어서,

char buffer[100][256];

이렇게 배열을 선언해서 사용자에게 100 개의 문자열을 입력 받는다고 생각하면, 사용자가 256 개의 문자를 다 사용하지는 않을테져.. 한글자를 넣을 수도 있고, 10 개를 넣을 수도 있고.. 그렇다면 256 개의 배열을 잡아놓았던 것이 낭비가 되는 겁니다.

그런것이 쌓이고 쌓이면 엄청난 메모리가 낭비가 됩니다. 그렇다고 배열을 작게만 잡을 수도 없져.

char buffer[100][20];

이렇게 잡아놨더니만 사용자가 30 개의 문자열을 입력하게 되면 어떻게 합니까? 그래서 배열로 문자열을 다루기는 참으로 애매한 경우가 많습니다. 따라서 문자열을 입력받아 저장할 때는 동적 메모리 할당을 많이 사용합니다.

메모리를 동적으로 할당 받는 예제를 하나 들어보져.

```
#include <stdio.h>
#include <stdlib.h>   /* malloc() */
#include <string.h>   /* strcpy() */

#define MAX_LEN       256

int main(void)
{
     char *p[10];
     char buffer[MAX_LEN];
     int i;

     for( i = 0 ; i < 10 ; i++ )
     {
          printf("Input %2d string: ", i );
          fgets( buffer, MAX_LEN, stdin );

          /* fgets() 함수로 입력을 받으면 마지막에 엔터키 까지 저장이 된다.
           * 이 엔터키를 없애기 위해서 문자열 크기 - 1 인 곳에 NULL 을 넣는다
           */
          buffer[ strlen(buffer)-1 ] = '₩0';

          /* buffer 크기 + 1 만큼 메모리를 할당 받고,
           * 할당 받은 메모리의 첫 번지 값을 p[i] 에 저장한다.
           * 메모리를 할당 받을때 문자열의 크기보다 1 개가 더 큰 이유는
           * NULL 값을 저장하기 위함이다.
           */
          p[i] = (char *)malloc( strlen(buffer) + 1 );
          if( p[i] == NULL )
          {
               printf("main(): memory allocation error!₩n");
               exit(-1);
          }

          /* buffer 의 문자열을 p[i]가 가리키고 있는 방금 할당 받은 메모리로
           * 복사 한다.
           * p[i]가 가리키고 있는 영역은 메모리의 Heap 영역이다
           */
          strcpy( p[i], buffer );
     }

     for( i = 0 ; i < 10 ; i++ )
          printf("%2d: %s₩n", i, p[i] );

     return 0;
}
```

나중에 더 자세하게 설명을 하겠지만 이렇게 동적 할당을 이용하면 배열을 사용하는 것보다 메모리를 낭비하지 않을 수가 있습니다.

문자열에서 항상 중요한 것은 문자열의 맨 마지막에는 꼭 NULL 값이 있어야 한다는 것입니다. 이 NULL 값은 숫자로는 0 과 같습니다.

문자열이 끝나는 것을 바로 이 NULL 값으로 판단을 하는 겁니다. 많은 문자열을 다루는 C library 들이 문자열의 끝을 판단할 때 NULL 로 하게 됩니다.

3.2. 문자열 함수 예제

문자열을 다루는 함수중에서 가장 많이 쓰이는 함수중에 strlen(), strcpy(), strcat() 이 있습니다.

이 세가지 함수의 하는 일과 이 함수들을 실제로 구현해 보면서 문자열에 대해서 이해를 할 수 있으면 좋겠군여.

3.2.1. strlen() 함수

첫번째로, strlen() 함수에 대해서 알아볼까여?

이 함수는 인자로 받은 문자열의 개수를 리턴하는 함수입니다. 우선 함수의 원형을 알아보져..

size_t strlen(const char *s);

size_t 는 unsigned long int 로 typedef 되어 있습니다.

문자열 포인터를 인자로 받아서 문자열 개수를 리턴하는 함수입니다.

그럼 간단하게 이 함수의 사용예를 들어보겠습니다.

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

int main(void)
{
	char *s = "Hello, world!";

	printf("string length = %d₩n", strlen(s) );
	return 0;
}
```

위 코드의 결과는

```
string length = 13
```

이렇게 됩니다.

strlen() 함수의 하는 일을 알았으니 이번엔 이 함수를 구현해 봐야겠져? 문자열의 문자들의 개수를 세어서 그 숫자를 리턴하는 함수를 만들면 되는 거져.

문자열의 개수를 셀 때 처음은 알지만 언제 끝나는지를 알아야 수를 셀텐데..

그렇습니다. 문자열의 마지막에는 NULL 값이 있다고 했져? 바로 이 NULL 값이 나올때까지 루프를 돌리면 되는 거져.

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

int my_strlen( const char *s )
{
	int i = 0;
	for( i = 0 ; *s ; i++, s++ );

	return i;
}

int main(void)
{
	char *s = "Hello, world!";

	printf("string length = %d", my_strlen(s) );
	return 0;
}
```

분석을 해볼까여? for 문에서 조건식을 봅시다. 조건식이 딸랑 *s 이렇게 되어 있네여..

왜 이렇게 했을까여? 이것은 s 포인터가 가리키는 곳의 값을 나타내는 것인데, 문자가 있으면 0 이 아닌 값을 갖겠져? 그러므로 조건은 참이 되는 것이지여. 그래서 증감식에서 i 의 값을 더해주고, s 포인터의 주소값도 하나 늘려주게 됩니다. 그리고 또 다시 *s 조건식을 만나서 0 인지를 판단..

이렇게 하면 결국 주소값이 계속 증가되면서 NULL(0)을 만나게 되고 이때까지 증가한 i 의 값이 바로 문자열의 길이가 되는 겁니다.

여기서 for()문은 i 의 값과 s 의 주소값만 증가 시킬뿐 다른 일은 하지 않습니다.

이렇게 포인터를 사용한 문자열을 다루는 것은 생각보다 아주 간단합니다. 마지막에 0 이 온다는 사실만 확실하게 주지하고 있으면 문자열을 다루는데 많은 아이디어를 가지고 다룰 수가 있습니다.

3.2.2. strcpy() 함수

strcpy() 함수는 이름에서도 알 수 있듯이 문자열을 복사하는 함수 입니다. 이 함수의 원형을 알아보면,

```
char *strcpy( char *dest, const char *src );
```

src 의 문자열을 dest 에 복사를 하는 겁니다. 그럼 strcpy() 함수를 사용하는 예제를 들어보져.

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

int main(void)
{
	char buffer[100];
	char *s = "Hello, world!";

	strcpy( buffer, s );

	printf("buffer = %s₩n", buffer );

	return 0;
}
```

실행 결과는

```
buffer = Hello, world!
```

이렇게 됩니다.

s 가 가리키고 있는 문자열을 buffer 배열에 복사를 하는 것이지여.

그럼 이제 하는 일을 알았으니까 구현을 해봐야져?

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

char *my_strcpy( char *dest, const char *src )
{
	char *s = dest;
	for( ; *src ; dest++, src++ )
		*dest = *src;

	*dest = '₩0';
	return s;
}

int main(void)
{
```

```
    char buffer[100];
    char *s = "Hello, world!";

    my_strcpy( buffer, s );

    printf("buffer = %s₩n", buffer );

    return 0;
}
```

코드를 분석 해 보면, s 포인터는 복사를 한 곳의 주소값을 리턴하기 위해서 dest 를 대입한것이고,

for() 문을 보면, 초기식은 필요가 없으므로 초기식은 없고, 조건식을 보면 위에서 설명한 my_strlen()와 비슷하져? 원본 문자열인 src 가 0 이 나올때 까지 루프를 돌리기 위한 조건식이 되겠져. 그리고 증감식을 보면 원본 포인터인 src 와 대상 포인터인 dest 의 주소를 하나씩 각각 늘려주면서 *dest = *src; 이렇게 한거져.. 이것은 src 가 가리키고 있는 곳의 값을 dest 가 가리키고 있는 곳에 저장해라.. 이런뜻이져?

마지막으로 문자열이 끝났다는 것을 저장하기 위해서 dest 의 맨 마지막 부분에 0 값을 저장했습니다.

차근차근 하나씩 따져보면서 코드를 분석하면 그리 어렵지 않다는 것을 알게 될겁니다.

3.2.3. strcat() 함수

그럼 마지막으로 strcat() 함수에 대해서 알아봐야겠네여..

strcat() 함수는 문자열을 붙이는 함수입니다. 함수의 원형을 보면,

```
char *strcat( char *src, const char *tail );
```

tail 의 내용을 src 의 뒤에 붙이는 것입니다. 그럼 strcat() 함수를 사용하는 예제를 들어보져.

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

int main(void)
{
    char buffer[30] = "1234567890";
    strcat( buffer, "abcd" );

    printf("buffer = %s₩n", buffer );
```

```
	return 0;
}
```

결과는

```
buffer = 1234567890abcd
```

이렇게 됩니다.

tail 이 가리키고 있는 곳의 문자열을 src 의 뒤에 연결해서 저장을 하는 겁니다.

하는 일을 알았으니 구현을 해보져.

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

char *my_strcat( char *src, const char *tail )
{
	char *s = src;

	for( ; *src ; src++ );
	for( ; *tail ; src++, tail++ )
		*src = *tail;

	*src = '₩0';

	return s;
}

int main(void)
{
	char buffer[30] = "1234567890";
	my_strcat( buffer, "abcd" );

	printf("buffer = %s₩n", buffer );

	return 0;
}
```

코드를 분석해 보면.. my_strcat() 함수내의 첫번째 for()문이 하는 역할은 src 의 포인터를 맨 마지막으로 이동하는 것입니다. 문자열의 뒤에 내용을 붙여야 하므로 src 가 가리키고 있는 포인터를 0 값이 있는 곳까지 이동을 하는 것입니다.

그리고 두번째 for()문에서 붙여야 할 문자열의 주소값과 src 의 주소값을 을 하나씩 증가하면서 src 가 가리키고 있는 곳에 tail 이 가리키고 있는 값을 저장합니다.

그리고 마지막으로 src 의 맨 뒤에 0 값을 써서 문자열이 끝났음을 저장하는 거지여.

지금까지 한 내용이 이해가 가지 않는다면 다시 한번 코드를 찬찬히 훑어 보시고 하나하나 계산을 해보세여. 그러면 포인터의 움직임을 조금은 알 수가 있을겁니다.

이처럼 문자열을 다루는 함수는 생각보다 간단하게 포인터를 이용해서 작성할 수가 있습니다. 포인터를 사용하는 많은 이유중에 문자열을 다루기가 쉽기 때문이라는 이유를 들어서 문자열을 다루어 보았습니다.

---

4 장. 다른 지역의 변수 바꾸기

이번 장은 포인터를 이용해서 다른 지역에 있는 값을 바꾸어 보도록 하겠습니다. 다른 지역에 있는 값을 바꾼다는 말에 대해서 먼저 설명하도록 하겠습니다.

이 강좌에서 다른 지역에 있는 변수라는 말의 뜻은 다른 함수에 존재하는 지역변수 라는 뜻입니다.

다른 함수에 존재하는 지역변수의 값을 바꾸려면 일반적인 방법으로는 불가능합니다. 일반적으로 가장 많이 설명되는 swap 기능을 하는 함수로 예를 들어보겠습니다.

```
 1  #include <stdio.h>
 2
 3  void swap( int x, int y );
 4
 5  void swap( int x, int y )
 6  {
 7        int temp;
 8
 9        temp = x;
10        x = y;
11        y = temp;
12  }
13
14  int main(void)
15  {
16        int x = 5, y = 10;
17
18        printf("before swap: x = %d, y = %d₩n", x, y );
19
20        swap( x, y );
21
22        printf("after swap: x = %d, y = %d₩n", x, y );
23
24        return 0;
25  }
```

위의 코드는 main() 함수내에 있는 x, y 변수의 값을 바꾸기 위해서 swap() 함수를 만들고 이 함수를 호출함으로써 x, y 의 값이 바뀌기를 기대하고 있는 건데, 실제로 컴파일을 하고 실행을 해보면 출력결과가 기대한것 처럼 나오지 않습니다.

그럼 코드를 분석해 볼까여?

3 라인은 swap() 함수의 원형(prototype)을 선언한 것입니다.

7 라인의 temp 변수는 x 와 y 의 값을 바꾸어 주기위한 임시 변수입니다.

9 라인에서 temp 의 값은 x(여기에서 5)로 되고,

10 라인에서 x 의 값은 y(10)이 되고,

11 라인에서 y 의 값은 temp(5)가 됩니다.

실제로 swap() 함수는 x 와 y 의 값을 뒤바꾸었습니다. 그런데 main()함수에서 출력하는 결과는 둘다 x = 5, y = 10 입니다.

왜그럴까여? 그 이유는 아주 단순한데에 있습니다.

main() 함수 내에 있는 x, y 와 swap() 함수에 있는 x, y 는 다른 지역내에 있는 변수로서 이름만 같을뿐 메모리상에 위치하는 주소는 다른 변수입니다.

따라서 swap() 함수에서 뒤바뀐 x, y 는 main() 함수에 있는 x, y 와는 전혀 무관한 변수인 것입니다.

그렇다고 swap() 함수가 return 을 이용해서 main() 함수의 x, y 를 바꿀 수 있는것도 아닙니다. 아.. 물론 구조체를 이용해서 return 을 사용한다면 또 가능할 수도 있겠지만 그건 여기서 원하는 것이 아니지여.

자.. 방금 설명에서 중요한 것이 있습니다. 그것은 바로 이름만 같을 뿐 메모리상에 위치하는 주소는 다른 변수라는 말입니다.

다른 지역에 있는 변수를 바꾸기 위해서 이름을 사용해서는 불가능하다면, 주소를 이용하면 되는 것입니다.

그럼 포인터를 이용한 주소로 다른 지역에 있는 변수의 값을 변경해보도록 하져.

```
1   #include <stdio.h>
2
3   void swap( int *x, int *y );
4
5   void swap( int *x, int *y )
6   {
7         int temp;
8
```

```
 9        temp = *x;
10        *x = *y;
11        *y = temp;
12   }
13
14   int main(void)
15   {
16        int x = 5, y = 10;
17
18        printf("before swap: x = %d, y = %d₩n", x, y );
19
20        swap( &x, &y );
21
22        printf("after swap: x = %d, y = %d₩n", x, y );
23
24        return 0;
25   }
```

코드를 분석해 보도록 하겠습니다.

swap() 함수의 인자가 포인터로 바뀌었습니다. 이것은 인자로 변수의 값을 받는게 아니라 그 변수의 번지를 인자로 받기 위함입니다.

```
 7 라인의 temp 변수는 임시 변수이고,

 9 라인에서 temp 의 값은 *x(main 에 있는 x 변수의 주소값으로 가서 그 값을 읽는다)
   으로 변경. 즉 temp 는 5

10 라인에서 *x = *y; 이것은 main 에 있는 x 변수의 번지로 가서,
   main 의 y 번지값에 있는 값을 저장해라. 즉, main() 함수의 x 변수의 값이
   10 으로 변경.

11 라인에서 main 에 있는 y 변수 주소값으로 가서 temp(5)를 저장해라.
   즉, main 의 y 변수가 5 로 변경.

20 라인에서 x 와 y 변수의 번지를 swap()함수에 넘겨주며 호출을 했습니다.

22 라인에서 변경된 x, y 의 값을 출력.
```

자... 이렇게 되는 것입니다. 포인터라는 것의 위력인 것이지여.

지금의 이 예제는 C 의 문법책이라면 거의 언급하는 예제 이므로 꼭 이해를 해 두어야 합니다.

이 예제를 설명하기 이전에 저번 강좌에서 사실은 다른 지역내에 있는 변수를 바꾼바가 있습니다. 기억이 잘 안나신다구여? 그럼 **3.2.3. strcat() 함수**을 다시 보세여. main() 함수내에 있는 문자배열의 내용을 함수를 호출해서 바꾸었을 겁니다. :)

이번 강좌는 포인터가 가지고 있는 능력중 하나를 설명드렸습니다. 하지만 아직도 왜 포인터를 꼭 사용해야 하는지에 대해서는 다들 의아해 하실 수 있을 것같지만, 천천히 계속되는 이 강좌를 읽으면서 나오는 코드를 직접 쳐보고 실행해보면서 이해를 하시면 왜 포인터를 사용하는지에 대해서 좀더 명확하게 알 수가 있을것입니다.

---

5 장. 구조체와 포인터

차례

이번 장은 구조체와 포인터의 연계해서 포인터가 어떻게 사용되고 또 그것이 어떤 의미를 갖는 것인지에 대해서 알아보겠습니다.

5.1. 구조체의 선언

구조체를 사용하는 가장 큰 이유는 공통적으로 사용하는 변수들을 묶어서 사용하기 위함일 것입니다. 공통적으로 사용하는 변수를 묶는 것을 예제로 보여드리지여.

```
 1  #include <stdio.h>
 2
 3  #define MAX_NUM 5
 4
 5  typedef struct _student Student;
 6
 7  struct _student
 8  {
 9        int kor;
10        int eng;
11        int math;
12  };
13
14  int main(void)
15  {
16        Student student[MAX_NUM];
17        int i = 0;
18        int sum = 0;
19        float avg = 0.0;
20
21        for( i = 0 ; i < MAX_NUM ; i++, printf("₩n") )
22        {
23              printf("Input student[%d]'s korean score: ", i);
24              scanf( "%d", &student[i].kor );
25              printf("Input student[%d]'s english score: ", i);
26              scanf( "%d", &student[i].eng );
27              printf("Input student[%d]'s math score: ", i);
```

```
28              scanf( "%d", &student[i].math );
29          }
```

```
30
31          printf("INDEX₩tKOREAN₩tENGLISH₩tMATH₩tTOTAL₩tAVERAGE₩n");
32          printf("-----₩t------₩t-------₩t----₩t-----₩t-------₩n");
33
34          for( i = 0 ; i < MAX_NUM ; i++ )
35          {
36              sum = student[i].kor + student[i].eng + student[i].math;
37              avg = (float)sum / 3;
38
39              printf("%2d₩t%3d₩t%3d₩t%3d₩t%d₩t%3.2f₩n",
40                      i, student[i].kor, student[i].eng, student[i].math,
41                      sum, avg );
42          }
43
44          return 0;
45      }
```

코드를 분석해 보겠습니다.

```
 5 라인에서 뒤에 나올 struct _student 에 대해서 Student 로 typedef 을 해놓은
   것입니다. 이것은 보통 헤더파일 등에서 어떤 구조체 타입들이 사용될지를
   미리 한눈에 알아볼 수 있도록 파일의 처음부분에 선언을 해 놓기 위해서
   사용되는 방법입니다.
   struct _student 에 대해서는 아래 7 라인부터 12 라인까지 선언되어 있습니다.

 7 라인부터 학생의 성적을 저장하기 위한 구조체를 선언합니다.
   여기서 공통적으로 사용되는 kor, eng, math 세가지 변수를 한데 묶어서
   struct _student 로 만들었으며 5 라인에서 이 구조체를 Student 로 typedef 해서
   사용할 것입니다.

16 라인에서 Student 형 배열을 MAX_NUM(5)만큼 선언합니다.

21 라인의 for 문을 통해서 사용자에게 5 명의 학생 성적을 입력받습니다.

36 라인에서 각각의 학생의 총점을 구하고,

37 라인에서 평균을 구합니다.

39 라인에서 각 학생들의 점수를 출력합니다.
```

내용은 그렇게 어렵지 않을 것입니다. 구조체를 사용해서 5 명의 학생 데이터를 student 라는 이름의 배열 하나로 전부 처리를 할 수가 있었습니다.

구조체의 선언 및 사용하는 코드를 보았으니, 이제 포인터와 연계되는 것을 공부 해봐야 겠네여.

5.2. 구조체와 포인터의 연계

앞장의 예를 포인터로 바꾸기 전에 왜 구조체를 사용할 때 포인터가 사용되어지는지 간단하게 알아볼 수 있는 예제를 들어보겠습니다.

```
 1  #include <stdio.h>
 2
 3  typedef struct _data      Data;
 4
 5  struct _data
 6  {
 7        int ID;
 8        int number;
 9        int color;
10  };
11
12  void printData( Data data );
13
14  int main(void)
15  {
16        Data data;
17
18        data.ID = 0;
19        data.number = 10;
20        data.color = 15;
21
22        printData( data );
23
24        return 0;
25  }
26
27  void printData( Data data )
28  {
29        printf("ID = %d, number = %d, color = %d₩n",
30                  data.ID, data.number, data.color );
31  }
```

위의 예제는 Data 구조체 변수를 한개 만들어서 그 구조체 멤버에 값을 넣고 printData() 함수를 호출하여 그 값들을 출력하는 예제입니다.

눈여겨 봐야할 것은 printData(Data data); 여기서 인자가 Data data 이렇게 되어 있다는 것입니다.

이렇게 함수를 작성하여 실제로 22 라인처럼 호출을 하게 되면 main() 에 있는 data 구조체 변수가 복사가 되어 printData() 함수에도 생성이 됩니다. 이렇게 되면 printData() 함수가 호출되어서 끝나기 전에는 똑같은 내용이 메모리상에 두개가 있는 것입니다.

문제는 메모리의 낭비뿐 아니라 실제로 printData() 함수에 인자를 넘겨줄 때 그 구조체를 생성하느라 시간이 걸린다는 것입니다. Data 구조체 만큼의 메모리를 필요로 하며, 각각의 멤버를 복사하는 시간이 걸리는 것입니다.

따라서 구조체를 사용할 때 위의 예제처럼 구조체 변수를 인자로 갖는 함수를 작성하면 그만큼 느린 프로그램이 됩니다.

그럼 위의 예제를 구조체 포인터 변수를 사용해서 바꾸어 보겠습니다.

```
 1  #include <stdio.h>
 2
 3  typedef struct _data      Data;
 4
 5  struct _data
 6  {
 7        int ID;
 8        int number;
 9        int color;
10  };
11
12  void printData( Data *data );
13
14  int main(void)
15  {
16        Data data;
17
18        data.ID = 0;
19        data.number = 10;
20        data.color = 15;
21
22        printData( &data );
23
24        return 0;
25  }
26
27  void printData( Data *data )
28  {
29        printf("ID = %d, number = %d, color = %d₩n",
30                  data->ID, data->number, data->color );
31  }
```

12 라인에서 인자가 포인터로 바뀌었으며 22 라인에서 data 의 주소값을 넘겨주었고, 30 라인에 data.ID 가 data->ID 로 바뀌었습니다.

포인터를 사용하면 22 라인에서 data 의 주소를 넘겨주면 printData() 함수는 단지 4 바이트의 주소값만 받게 되는 것입니다. 따라서 똑같은 구조체가 생성되는 것이 아니라 main() 함수에 있는 data 변수를 직접 사용하게 되는 것입니다.

따라서 구조체를 복사하느라 드는 시간을 없앨 수 있게 되는 것입니다.

다음은 전 장에서 들었던 예제를 포인터로 바꾸는 예를 들어보겠습니다.

```
1  #include <stdio.h>
2
```

```
 3  #define MAX_NUM 5
 4
 5  typedef struct _student Student;
 6
 7  struct _student
 8  {
 9      int kor;
10      int eng;
11      int math;
12  };
13
14  void printScore( Student student[MAX_NUM] );
15
16  int main(void)
17  {
18      Student student[MAX_NUM];
19      int i = 0;
20
21      for( i = 0 ; i < MAX_NUM ; i++, printf("₩n") )
22      {
23          printf("Input student[%d]'s korean score: ", i);
24          scanf( "%d", &student[i].kor );
25          printf("Input student[%d]'s english score: ", i);
26          scanf( "%d", &student[i].eng );
27          printf("Input student[%d]'s math score: ", i);
28          scanf( "%d", &student[i].math );
29      }
30
31      printScore( student );
32
33      return 0;
34  }
35
36  void printScore( Student student[MAX_NUM] )
37  {
38      int i;
39      int sum = 0;
40      float avg = 0.0;
41
42      printf("INDEX₩tKOREAN₩tENGLISH₩tMATH₩tTOTAL₩tAVERAGE₩n");
43      printf("-----₩t------₩t-------₩t----₩t-----₩t-------₩n");
44
45      for( i = 0 ; i < MAX_NUM ; i++ )
46      {
47          sum = student[i].kor + student[i].eng + student[i].math;
48          avg = (float)sum / 3;
49
50          printf("%2d₩t%3d₩t%3d₩t%3d₩t%3d₩t%3.2f₩n",
51                  i, student[i].kor, student[i].eng, student[i].math,
52                  sum, avg );
53      }
```

```
54  }
```

출력하는 함수의 인자가 Student student[MAX_NUM] 이렇게 되었지여? 이것은 33 라인에서 printScore() 함수를 호출할 때 main() 함수에 있는 student 배열의 주소값이 printScore() 함수의 student 포인터 상수로 넘겨집니다.

즉, printScore() 함수에 main()에 있는 student 배열의 주소값을 갖는 student 포인터 상수가 생성됩니다.

그렇게 해서 printScore() 함수에서는 student 포인터 상수를 이용해서 main()에 선언된 student 배열의 주소를 참조해서 문제 없이 출력을 할 수가 있는 것입니다.

포인터 상수를 사용하거나 포인터 변수를 사용하거나 실제 프로그램이 실행되는데 큰 차이는 없습니다. 이번엔 printScore() 함수의 인자를 포인터 상수를 선언하는 것이 아닌 포인터 변수를 사용해 보도록 하겠습니다.

```
 1  #include <stdio.h>
 2
 3  #define MAX_NUM 5
 4
 5  typedef struct _student Student;
 6
 7  struct _student
 8  {
 9      int kor;
10      int eng;
11      int math;
12  };
13
14  void getScore( Student *student );
15  void printScore( Student *student );
16
17  int main(void)
18  {
19      Student student[MAX_NUM];
20
21      getScore( student );
22      printScore( student );
23
24      return 0;
25  }
26
27  void getScore( Student *student )
28  {
29      int i = 0;
30
31      for( i = 0 ; i < MAX_NUM ; i++, printf("₩n") )
32      {
```

```
33              printf("Input student[%d]'s korean score: ", i);
34              scanf( "%d", &student[i].kor );
35              printf("Input student[%d]'s english score: ", i);
36              scanf( "%d", &student[i].eng );
37              printf("Input student[%d]'s math score: ", i);
38              scanf( "%d", &student[i].math );
39          }
40  }
41
42  void printScore( Student *student )
43  {
44          int i;
45          int sum = 0;
46          float avg = 0.0;
47
48          printf("INDEX₩tKOREAN₩tENGLISH₩tMATH₩tTOTAL₩tAVERAGE₩n");
49          printf("-----₩t------₩t-------₩t----₩t-----₩t-------₩n");
50
51          for( i = 0 ; i < MAX_NUM ; i++ )
52          {
53              sum = student[i].kor + student[i].eng + student[i].math;
54              avg = (float)sum / 3;
55
56              printf("%2d₩t%3d₩t%3d₩t%3d₩t%3d₩t%3.2f₩n",
57                      i, student[i].kor, student[i].eng, student[i].math,
58                      sum, avg );
59          }
60  }
```

코드를 보면 printScore() 함수의 인자를 포인터로 바꾸었을 뿐 코드 내용은 변한게 없습니다.

여기서 조금 이상하다고 생각되는 부분이 있을 것입니다. 포인터를 인자로 받아서 어떻게 배열로 받은 것과 똑같이 사용을 하게 된건지...

그래서 포인터와 배열의 관계에 대해서 잠시 설명을 드리겠습니다.

```
char s[10] = "abcdefghi";
```

위와 같이 10 글자 짜리 배열 s 를 선언했다고 했을때, 처음의 a 라는 글자를 사용하려면 어떻게 해야 할까여?

네, s[0] 이 되겠져?

그럼 d 라는 글자를 사용하려면? 네, s[3] 입니다.

그런데 분명 배열의 이름인 s 는 포인터라고 말씀을 드렸었져? 이 포인터를 사용해서 d 를 나타낼 수도 있습니다.

*s 이것은 s 배열의 첫번째 주소로 가서 그 값을 나타냅니다. 그렇져? 따라서 s 배열이 있는 곳으로 가서 그 값을 읽으면 바로 a 가 됩니다.

그럼 4 번째 존재하는 d 라는 문자를 읽으려면 어떻게 할까여?

*(s + 3) 이렇게 하면 s 의 주소값에 3 을 더한 다음에 그 주소에 있는 값을 읽어라. 이렇게 되는 겁니다.

그림으로 나타내 보겠습니다.

```
0x1000
   +---+---+---+---+---+---+---+---+---+---+---+
   | 'a'  'b'   'c'  'd'  'e'   'f'  'g'   'h'   'i'   'j'  ₩0 | s
   +---+---+---+---+---+---+---+---+---+---+---+
       |   |   |
       |   |   |
       |   |   +---> 0x1003 번지
       |   |
       |   +---> 0x1002 번지
       |
       +---> 0x1001 번지
```

그림은 위와 같습니다.

처음 번지가 0x1000 라고 가정을 했을때 char 의 크기가 1 바이트 이므로 주소값은 1 씩 증가해서 저장이 됩니다. 따라서 그림에서 보듯이 a 라는 문자열은 0x1000 번지에, b 라는 문자열은 0x1001 번지에, c 라는 문자열은 0x1002 번지에, d 라는 문자열은 0x1003 번지에 각각 저장이 됩니다.

위에서 *(s + 3) 의 표현을 계산 해보도록 하겠습니다.

s 는 배열의 첫번지 이므로 0x1000 입니다. 여기에 3 을 더했으므로 0x1003 이 되져? 그렇다면 *(0x1003) 이 된것입니다. 이 표현은 0x1003 번지에 가서 그 값을 읽어라. 이뜻인데 0x1003 번지에는 무엇이 있지여? 네 바로 d 라는 문자가 있습니다. 배열로 표시를 하면 s[3] 이렇게 되져?

여기에서 다음과 같은 표현이 성립된다는 것을 알 수가 있습니다.

```
s[0] == *( s + 0 ) == *s
s[1] == *( s + 1 )
s[2] == *( s + 2 )
s[3] == *( s + 3 )
```

이 관계에 따라서 student 포인터를 가지고 배열처럼 사용을 할 수가 있었던 것이지여..

그래서 student[i].kor 을 배열로 표시하지 않고 포인터로 표현을 하면, (student+ i)->kor 처럼 되는 것입니다.

여기서 -> 에 대해서도 알아볼까여?

-> 은 포인터를 사용할때 구조체의 멤버를 접근할 수 있게 하는 것입니다. 예를 들어서 위의 Student 구조체를 사용한다고 할때,

```
Student s, *p;
p = &s;

s.kor = 10;
p->eng = 20;
```

이렇게 됩니다. s 는 포인터가 아닌 변수이며 이 s 를 이용해서 멤버에 접근할때는 .(Dot)를 사용합니다. p 는 s 의 주소값을 저장하고 있는 포인터이며 이 p 를 이용해서 s 의 멤버에 접근할 때는 ->를 사용하는 것입니다.

-> 도 * 처럼 "거기로 이동하라" 라고 인식을 하면 이해하기가 쉽습니다. p->eng 는 s 의 주소로 이동을 해서 eng 라는 멤버를 접근해라.. 이렇게 이해를 하면 되는 것입니다.

따라서 (student+ i)->kor 이것은 student 가 저장하고 있는 주소값에 i 를 더한 후, kor 멤버에 접근을 해라... 이런 뜻이 됩니다.

여기서 잠깐.. (student+ i)->kor 에서 student 가 저장하고 있는 주소값이 만약 0x1000 이고, i 의 값은 2 라고 가정을 하면, (student+ i) 이 값은 얼마가 될까여?

0x1002 일까여?

그렇지 않습니다. 여기에서 알아야할 것이 또 한가지가 있습니다. 포인터는 증감 및 곱하기 나누기등의 연산을 하면 안됩니다.

아니, 그럼 지금까지 포인터에 더하기를 한것은 무엇이란 말입니까?

그것은 일반적인 덧셈과는 조금 다른 의미를 갖고 있는 것입니다. 예를 들어보져..

```
short int s = 10;      /* 0x1000 */
short int *sp = &s;

long int l = 10;       /* 0x2000 */
long int *lp = &l;

float f = 10.0;        /* 0x3000 */
float *fp = &f;

double d = 10.0;       /* 0x4000 */
double *dp = &d;
```

위와 같이 변수들과 포인터들이 선언되어 있습니다. 변수 옆에 주석으로 처리되어 있는 숫자는 그 변수의 주소값을 임의로 정한겁니다.

우선 short int 형인 s 변수의 주소를 저장하고 있는 sp 포인터에 대해서 알아보져. sp 는 0x1000 을 갖고 있는데 (sp + 1)의 값이 과연 얼마일까여? 0x1001 일까여?

그렇지 않습니다. 결과적으로 답은 0x1002 입니다. 그 이유는 sp 포인터 앞에 타입이 무엇이져? short int 이져? short int 는 2 바이트 크기의 자료형입니다.

즉 sp 포인터가 가리키고 있는 곳의 자료형이 short int 형이라는 것이져. 그래서 (sp + 1)을 한다는 것은 단순히 1 을 더하는 것이 아니라 자신이 가리키고 있는 곳의 자료형의 크기만큼 곱해서 더하는 것이지여.

그럼 (sp + 3)의 값은? 네 0x1006 이 되는 것입니다.

마찬가지로 가리키는 곳이 long int 형인 lp 포인터에 1 을 더하면 어떻게 되져? long int 형이 4 바이트 자료형이므로 0x2004 가 되는 것입니다.

다른 것들도 마찬가지 입니다.

자, 그렇다면 Student 로 돌아가서, (student+ i) 의 값을 알아보도록 하져.

student 의 자료형은 Student 구조체져? 이 구조체는 int 자료형 3 개가 묶인 것이져. 그렇다면 이 구조체의 크기는? 네 int 가 4 바이트이므로 12 바이트가 되겠져. 왜 int 가 4 바이트라고 했져? 32 비트 운영체제라고 가정을 하고 설명을 한다고 처음에 말씀을 드렸었져?

student 가 저장하고 있는 주소값이 만약 0x1000 이라고 하면, (student+ i)의 값은 0x1000+ (i*12) 이렇게 되는 것입니다.

그래서 student 가 저장하고 있는 주소값이 0x1000 이고, i 의 값이 2 라고 가정하면, (student+ i)의 값은 0x1000 + 2*12 = 0x1018 이렇게 되는 것입니다.

포인터 변수 앞에 붙는 * 과 뒤에 붙는 ->는 그 의미가 아주 비슷합니다. 포인터 변수가 저장하고 있는 주소값으로 이동하라.. 라고 해석을 하면 되기 때문입니다.

이 내용의 더 심도있는 얘기는 좀더 뒤에서 다루기로 하겠습니다. 컴파일러의 내부에서 동작하는 포인터에 대해서 설명을 해야하는데, 지금 이 얘기를 하면 이 장이 너무 길어지게 되고 어려워 질것 같군여.

---

6 장. 함수 포인터

차례

함수 포인터라는 것은 일반 C 문법책에서도 그리 자세하게는 설명되지 않은 분야라서 알고는 있지만 제대로 사용을 못하는 경우가 많습니다.

함수 포인터라는 것 역시 32 비트 체제하에서 4 바이트의 메모리를 갖는 포인터 변수 입니다. 일반 포인터 변수와 다른 점은 일반 포인터 변수가 변수들의 주소값을 저장하는 반면에 함수 포인터는 함수의 주소값을 갖는다는 것입니다.

아니, 함수도 주소값이 있나? 라고 생각하는 분들도 있겠지만 함수는 code 부분입니다. 즉 프로그래머가 짠 코드가 컴파일 되어서 기계 코드로 변화된 것, 그것이 바로 code 입니다. 프로그램이 실행되기 위해서는 이 code 가 메모리에 올라가 있어야 하는 것입니다. 여기서 어떤 함수의 호출은 이 code 중에서 그 함수 부분으로 jump(이동) 하는 것이지여. 바로 이 함수 부분이라는 것이 그 함수의 주소값이 되는 것이고 이 함수의 주소값을 저장하는 포인터가 함수 포인터입니다.

함수 포인터에 대한 개념은 이미 C 의 문법책을 통해서 공부를 했으리라 가정을 하고 실제 함수 포인터가 어떤때에 사용이 되는지에 대해서 알아보겠습니다.

6.1. 함수 포인터의 기초 예제

일반 C 문법책에도 설명되어 있는 가장 간단한 함수 포인터의 예제를 들어보겠습니다.

```
#include <stdio.h>

void myPrint(void);

int main(void)
{
    /* 함수 포인터를 선언 */
    void (*func)(void);

    /* 함수 포인터에 myPrint 함수의 주소값을 대입 */
    func = myPrint;

    /* myPrint 함수의 주소값이 저장된 func 함수 포인터로
     * 호출을 하면 myPrint() 함수를 호출한 것과 동일한
     * 효과를 갖는다.
     */
    func();

    return 0;
}
```

```
void myPrint(void)
```

```
{
    printf("This is test printing..₩n");
}
```

실행 결과는

```
This is test printing..
```

위의 예제는 함수 포인터가 하는 일을 가장 명확히 보여주는 예제입니다. func 라는 함수 포인터에 myPrint() 함수의 주소값을 대입해서 func 를 호출하면 myPrint() 함수가 호출이 되는 것입니다. 마치 포인터 변수가 자신이 갖고 있는 다른 변수의 주소를 이용해서 다른 변수를 사용하는 것과 같습니다.

위에서 func(); 과 같이 호출을 했는데 실제의 의미는 (*func)();이 더 확실한 의미를 갖습니다. 다만 컴파일러가 두가지를 동일하게 해석을 하기 때문에 func();와 같이 사람이 보기에 편한 모양을 사용하는 것입니다.

6.2. 함수 포인터의 확장

앞의 기본적인 예제를 이해했다면 함수 포인터를 조금 확장해 봅시다. 도대체 함수 포인터를 왜 사용하는지 그 이유를 모르는 분들의 궁금증을 조금 해소시켜 드리기 위해서 일반적으로 잘 사용하지는 않지만 함수 포인터를 사용하면 아주 알기 쉬운 코드가 되는 예제를 들겠습니다.

```
 1  #include <stdio.h>
 2
 3  void plus( int x, int y );
 4  void minus( int x, int y );
 5  void multiple( int x, int y );
 6  void divide( int x, int y );
 7
 8  int main(void)
 9  {
10      /* 함수 포인터 배열을 선언해서 네개의 함수의 주소값을
11       * 차례로 저장한다.
12       */
13      void (*calc[4])( int x, int y ) = { plus, minus, multiple, divide };
14      int i = 0;
15
16      /* 루프를 4 번 돌면서 함수 포인터 배열에 들어 있는 함수의 주소값을
17       * 한번씩 실행을 한다.
18       */
19      for( i = 0 ; i < 4 ; i++ )
20          calc[i]( 10, 2 );
```

```
21
```

```
22      return 0;
23  }
24
25  void plus( int x, int y )
```

```
26  {
27      printf("PLUS       : %d + %d = %d₩n", x, y, x + y );
28  }
29
30  void minus( int x, int y )
31  {
32      printf("MINUS      : %d - %d = %d₩n", x, y, x - y );
33  }
34
35  void multiple( int x, int y )
36  {
37      printf("MULTIPLE : %d * %d = %d₩n", x, y, x * y );
38  }
39
40  void divide( int x, int y )
41  {
42      printf("DIVIDE     : %d / %d = %d₩n", x, y, x / y );
43  }
```

실행 결과는 다음과 같습니다.

```
PLUS       : 10 + 2 = 12
MINUS      : 10 - 2 = 8
MULTIPLE : 10 * 2 = 20
DIVIDE     : 10 / 2 = 5
```

위의 코드를 조금 분석해 보도록 하겠습니다.

13 라인에서 인자를 int 형으로 두개를 받고 int 형을 되돌리는 함수의 주소를 저장할 수 있는 포인터 4 개를 선언하고 plus, minus, multiple, divide 네개의 함수 주소를 대입했습니다. 여기에서 함수 포인터 배열을 선언한 형태에 대해서 잘 기억해 두시기 바랍니다.

20 라인에서 calc 배열의 각각의 요소에 저장되어 있는 함수의 주소값을 이용해서 4 개의 함수를 호출한 코드입니다.
실제로 calc[0]은 plus 함수, calc[1]은 minus 함수, calc[2]는 multiple 함수, calc[3]은 divide 함수의 주소를 갖고 있습니다.

위에서 보듯이 함수 포인터는 함수의 주소값을 갖고 있으며 그 함수의 주소값을 이용해서 자신이 갖고 있는 주소에 있는 함수를 호출할 수 있습니다.

이번엔 좀 더 쓸모있는 예제를 예를 들어보겠습니다.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
```

```
 4  #include <ctype.h>
 5
 6  typedef struct _command       Command;
 7  typedef enum _tokenType       TokenType;
 8  typedef void (*Callback)(void);
 9
10  /* 토큰의 type 을 정의 */
11  enum _tokenType
12  {
13      TOKEN_INIT,
14
15      TOKEN_INT,              /* "int" */
16      TOKEN_LONG,             /* "long" */
17
18      TOKEN_EQ,               /* "=" */
19      TOKEN_SEMICOLON,        /* ";" */
20
21      TOKEN_NAME,             /* 일반 문자열 */
22      TOKEN_NUMBER            /* 숫자 */
23  };
24
25  /* 명령어들의 속성을 정의하는 구조체 */
26  struct _command
27  {
28      TokenType     type;
29      char *        command;
30      Callback      call;
31  };
32
33  void intFunction(void);
34  void longFunction(void);
35
36  /* 토큰 분리를 위한 정의 */
37  #define MAX_TOKEN_LENGTH      128
38
39  char tokenBuffer[MAX_TOKEN_LENGTH];
40  char *token = tokenBuffer;
41  char *_MP = NULL;
42
43  void gettoken(void);
44  int is_white( char ch );
45  int is_delim( char ch );
46  /* 토큰 분리를 위한 정의 끝 */
47
48  Command command[ ] =
49  {
50      { TOKEN_INT,      "int",        intFunction },
51      { TOKEN_LONG,     "long",       longFunction }
52  };
53
54  TokenType tokenType = TOKEN_INIT;
55  int commandNumber = sizeof(command)/sizeof(command[0]);
```

```
 56
 57
 58  int main(void)
 59  {
 60      char *src = "int a = 10; long b = 1000000;";
 61      int i = 0;
 62
 63      /* gettoken() 함수에 의해서 토큰들이 분리가 될 수 있도록
 64       * src 의 주소를 _MP 에 할당한다.
 65       * gettoken()함수는 _MP 를 이용해서 토큰을 분리하기 때문이다.
 66       */
 67      _MP = src;
 68
 69      do
 70      {
 71          /* 토큰을 받아들여서 그 토큰에 해당하는 함수를 호출한다.
 72           * 바로 여기에서 함수 포인터의 위력이 나타나는 것이다.
 73           */
 74          gettoken();
 75          for( i = 0 ; i < commandNumber ; i++ )
 76              if( command[i].type == tokenType )
 77                  command[i].call();
 78      } while( tokenType != TOKEN_INIT );
 79
 80      return 0;
 81  }
 82
 83  /* intFunction
 84   *
 85   * "int" 명령이 들어왔을때 실행이 되는 함수
 86   */
 87  void intFunction(void)
 88  {
 89      char name[MAX_TOKEN_LENGTH];
 90      int   var = 0;
 91
 92      /* 토큰을 받아들인다. 변수 이름이 와야 한다. */
 93      gettoken();
 94
 95      /* 받아들인 토큰을 name 배열에 저장한다. 즉, 변수의 이름 */
 96      strcpy( name, token );
 97
 98      gettoken();
 99      if( tokenType == TOKEN_EQ )
100      {
101          gettoken();
102          var = atoi(token);
103
104          gettoken();
105          if( tokenType != TOKEN_SEMICOLON )
106          {
107              fprintf( stderr, "`;' is missing in `int' operation.₩n" );
```

```
108                 exit(1);
109             }
110         }
111         else if( tokenType != TOKEN_SEMICOLON )
112         {
113             fprintf( stderr, "`;' is missing in `int' operation.₩n" );
114             exit(1);
115         }
116
117         fprintf( stdout, "%s(%d;int) is created!₩n", name, var );
118     }
119
120     /* longFunction
121      *
122      * "long" 명령이 들어왔을때 실행이 되는 함수
123      */
124     void longFunction(void)
125     {
126         char name[MAX_TOKEN_LENGTH];
127         long   var = 0;
128
129         gettoken();
130
131         strcpy( name, token );
132
133         gettoken();
134         if( tokenType == TOKEN_EQ )
135         {
136             gettoken();
137             var = atol(token);
138
139             gettoken();
140             if( tokenType != TOKEN_SEMICOLON )
141             {
142                 fprintf( stderr, "`;' is missing in `long' operation.₩n" );
143                 exit(1);
144             }
145         }
146         else if( tokenType != TOKEN_SEMICOLON )
147         {
148             fprintf( stderr, "`;' is missing in `long' operation.₩n" );
149             exit(1);
150         }
151
152         fprintf( stdout, "%s(%ld;long) is created!₩n", name, var );
153     }
154
155     /* gettoken
156      *
157      * 토큰을 받아들이는 함수
158      */
159     void gettoken(void)
```

```
160  {
161       register int i = 0;
162       char *temp = tokenBuffer;
163
164       while( is_white( *_MP ) ) _MP++;
165
166       if( *_MP == '₩0' )
167       {
168            *temp = '₩0';
169            tokenType = TOKEN_INIT;
170            return;
171       }
172
173       *temp++ = *_MP++;
174
175       switch( *(_MP - 1) )
176       {
177            case '=' :
178                 *temp = '₩0';
179                 tokenType = TOKEN_EQ;
180                 return;
181            case ';' :
182                 *temp = '₩0';
183                 tokenType = TOKEN_SEMICOLON;
184                 return;
185            default :
186                 if( isdigit( *(_MP-1) ) )
187                 {
188                      while( !is_delim(*_MP) ) *temp++ = *_MP++;
189                      *temp = '₩0';
190                      tokenType = TOKEN_NUMBER;
191                 }
192                 else
193                 {
194                      while( !is_delim(*_MP) ) *temp++ = *_MP++;
195                      *temp = '₩0';
196
197                      for( i = 0 ; i < commandNumber ; i++ )
198                      {
199                           if( strcmp( command[i].command, token ) == 0 )
200                           {
201                                tokenType = command[i].type;
202                                return;
203                           }
204                      }
205                      tokenType = TOKEN_NAME;
206                      return;
207                 }
208       }
209  }
210
211  /* is_white
```

```
212    *
213    * ch 가 공백문자이면 1, 그렇지 않으면 0 을 리턴
214    */
215   int is_white( char ch )
216   {
217         if(ch == ' ' || ch == '₩t' || ch == '₩r' || ch == '₩n' ) return(1);
218         else return(0);
219   }
220
221   /* is_delim
222    *
223    * ch 가 분리자이면 1 을 그렇지 않으면 0 을 리턴
224    */
225   int is_delim( char ch )
226   {
227         if(strchr("=; ₩t₩r₩n", ch) || ch == 0 )
228               return(1);
229         return(0);
230   }
```

위 코드의 실행 화면은 다음과 같습니다.

```
a(10;int) is created!
b(1000000;long) is created!
```

위의 예제는 간단한 인터프리터를 예제로 든 것입니다. 컴파일러를 작성하는데 필요한 하나의 기법인 셈입니다. 위의 소스는 실제로 제가 자주 사용하는 모양입니다. 보통 설정파일을 인식하는 프로그램을 작성한다든지, 간단한 스크립트 언어를 만들어야 한다든지 하는 작업이 있을 때, 위와 같은 구성으로 프로그램을 만듭니다.

코드의 분석은 강좌를 읽으시는 분들이 하나하나 분석을 해보시기 바랍니다. 물론 이상한 부분이나 모르는 부분에 대해서는 제 홈페이지의 질답 게시판에 올려주시면 됩니다.

---

7 장. 포인터가 사용되는 분야

차례

이 장에서는 실제 실무에서 포인터가 사용되는 분야에 대해서 설명을 하도록 하겠습니다. 실무를 하다 보면 엄청난 크기의 구조체와 공용체, 그리고 포인터의 쓰임에 황당해 하는

프로그래머들을 많이 보아왔습니다. 그리고 그 이유는 포인터에 대한 아주 기초적인 부분에서 부터 흔들리기 때문이라는 것을 알아차리게 된것입니다.

따라서 이 장이 조금 어렵게 느껴질 지는 모르지만 최소한 이 장에 나오는 내용을 자유자재로 쓸 수 있을 정도가 되면 실무에 나가서도 C 를 이용해서 프로그래밍을 할 때 고생하지 않을 것입니다.

7.1. 객체 지향 프로그래밍 1

이번 장에서는 객체 지향 프로그래밍(Objective-Oriented Programming)에 관해서 공부를 해 보겠습니다.

사실 객체 지향 프로그래밍을 하려면 객체 지향적인 문법을 갖고 있는 컴파일러로 공부를 해야 정확합니다. 예를 들어서 C++, Java 등등..

하지만 구조적인 언어인 C 언어로도 객체 지향의 개념을 도입해서 얼마든지 프로그래밍을 할 수가 있습니다. 물론 객체 지향의 문법이 지원되지 않기 때문에 구조체와 포인터의 테크닉을 이용해서 흉내를 내는 것이지여.

그럼 일단 다음의 예제를 보자구여.

```
 1  #include <stdio.h>
 2  #include <stdlib.h> /* malloc(), free() */
 3  #include <string.h> /* strlen(), strcpy() */
 4
 5  /* 구조체를 타입에 맞게 캐스팅 해주는 매크로 함수 */
 6  #define OBJECT(object)   ((Object *)(object))
 7  #define MANKIND(object) ((Mankind *)(object))
 8
 9  typedef struct _object   Object;
10  typedef struct _mankind Mankind;
11
12  typedef enum _type        Type;
13
14  enum _type
15  {
16      OBJECT_TYPE,          /* 객체(Object) 타입   */
17      MANKIND_TYPE          /* 사람(Mankind) 타입 */
18  };
19
20  struct _object
21  {
22      Type              type;    /* 객체의 종류        */
23      unsigned long    ID;      /* 객체의 일련 번호 */
24  };
25
26  struct _mankind
```

```
27  {
28      Object   object;       /* 객체(Object) 속성 상속 */
29      char *   name;          /* 이름                   */
30  };
31
32  /* 객체의 일련번호 부여를 위한 전역 변수 */
33  unsigned long _objectID = 0;
34
35  /*
36   * 함수의 원형(prototype) 선언
37   */
38  void                setObjectID( Object *object );
39  unsigned long       getObjectID( Object *object );
40  void                setObjectType( Object *object, Type type );
41  Type                getObjectType( Object *object );
42  void                freeObject( Object *object );
43
44  Object *      createMankind(void);
45  void          setMankindName( Mankind *mankind, char *name );
46  char *        getMankindName( Mankind *mankind );
47  void          freeMankind( Mankind *mankind );
48
49  /* setObjectID
50   *
51   * 객체의 고유 일련번호를 설정한다.
52   * 이 함수는 사용자가 마음대로 사용할 목적이 아니라,
53   * 객체가 생성되었을 때 한번 호출해서 객체에게 일련번호를
54   * 부여하는 것이다.
55   *
56   * 전역변수인 _objectID 은 객체에게 일련번호를 부여하고 난 후,
57   * 1 씩 증가한다.
58   */
59  void setObjectID( Object *object )
60  {
61      object->ID = _objectID++;
62  }
63
64  /* getObjectID
65   *
66   * 객체의 고유 일련번호(ID)를 구한다.
67   */
68  unsigned long getObjectID( Object *object )
69  {
70      return object->ID;
71  }
72
73  /* setObjectType
74   *
75   * 객체의 타입을 설정한다.
76   */
77  void setObjectType( Object *object, Type type )
78  {
```

```
79      object->type = type;
80  }
81
82  /* getObjectType
83   *
84   * 객체의 type 을 구한다.
85   * 객체의 type 은 객체의 특성을 말한다.
86   *
87   * 객체 생성함수에 따라서 그 객체의 type 은 설정된다.
88   */
89  Type getObjectType( Object *object )
90  {
91      return object->type;
92  }
93
94  /* freeObject
95   *
96   * 생성된 객체의 할당받은 메모리를 해제한다.
97   * Object 의 type 을 조사하여 각 type 에 맞는 해제 함수를 호출한다.
98   */
99  void freeObject( Object *object )
100 {
101     /* 해제시킬 객체의 종류를 알아낸다. */
102     Type type = getObjectType(object);
103
104     /* 종류에 맞는 파괴 함수를 호출 */
105     switch( type )
106     {
107         case OBJECT_TYPE :
108             break;
109         case MANKIND_TYPE :
110             freeMankind( MANKIND(object) ); break;
111     }
112
113     /* 마지막으로 각 type 의 크기만큼 할당된 메모리를 해제한다. */
114     free( object );
115 }
116
117 /* createMankind
118  *
119  * 사람 객체를 생성하는 함수이다.
120  *
121  * 모든 객체는 create*() 함수로 생성되어지며, 그 객체의 데이터 형은
122  * Object * 이다.
123  * 모든 객체의 구조체 타입이 다 다르지만 Object 구조체로 모든 객체를
124  * 다룰 수 있도록 하기 위해서 메모리는 sizeof(Mankind) 만큼 할당되지만
125  * 그 할당 받은 메모리를 (Object *)로 캐스팅한다.
126  *
127  * 그렇다고 해서 할당받은 메모리의 크기가 변한다던지 하는게 아니라
128  * 단지 그 할당받은 메모리를 참조하기 위한 index 로 Object 구조체를
129  * 사용하겠다는 것만 알려주는 것일 뿐이다.
130  *
```

```
131   * 실제로 Mankind 구조체에 있는 속성을 사용하기 위해서,
132   * (object *)로 캐스팅된 메모리는 다시 (Mankind *)로 캐스팅되야 한다.
133   * MANKIND(object) 매크로 함수를 사용해도 된다.
134   *
135   * 이렇게 캐스팅을 통해서,
136   * 많은 종류의 객체들을 대표적인 객체 제어 함수로 제어를 할 수 있게 된다.
137   * 예를 들면 어떤 타입의 객체라도 freeObject() 함수로 파괴를 할 수가
138   * 있게 되는 것이다.
139   */
140  Object *createMankind(void)
141  {
142      /* Mankind 구조체 크기 만큼의 메모리를 할당 받은 후
143       * (Object *)로 캐스팅 한다.
144       */
145      Object *mankind = (Object *)malloc( sizeof(Mankind) );
146      if( mankind == NULL ) return NULL;
147
148      /* 할당받은 메모리를 0 으로 초기화 한다. */
149      memset( mankind, 0, sizeof(Mankind) );
150
151      /* 생성된 객체의 type 을 MANKIND_TYPE 으로 설정한다.
152       * 즉, 이 객체는 사람 객체임을 뜻한다.
153       */
154      setObjectType( mankind, MANKIND_TYPE );
155
156      /* 생성된 객체의 고유 일련번호를 부여한다. */
157      setObjectID( mankind );
158
159      return mankind;
160  }
161
162  /* setMankindName
163   *
164   * 사람 객체의 이름 속성을 설정한다.
165   */
166  void setMankindName( Mankind *mankind, char *name )
167  {
168      /* name 인자의 문자열 길이 + 1 만큼 메모리를 할당한다.
169       * 1 을 더하는 이유는 문자열 마지막에 NULL 을 저장하기 위함이다.
170       */
171      mankind->name = (char *)malloc( strlen(name) + 1 );
172      if( mankind->name == NULL ) return;
173
174      /* 사람 객체의 이름 속성에 name 을 복사한다. */
175      strcpy( mankind->name, name );
176  }
177
178  /* getMankindName
179   *
180   * 사람 객체의 이름 속성을 구한다.
181   */
182  char *getMankindName( Mankind *mankind )
```

```
183  {
184       return mankind->name;
185  }
186
187  /* freeMankind
188   *
189   * 사람 객체의 속성을 해제하는 파괴함수.
190   */
191  void freeMankind( Mankind *mankind )
192  {
193       /* 사람 이름 속성 해제 */
194       if( mankind->name != NULL )
195            free( mankind->name );
196  }
197
198  int main(void)
199  {
200       Object *man = NULL;
201
202       man = createMankind();
203       setMankindName( MANKIND(man), "Shim sang don" );
204
205       printf("Man ID = %ld, name = %s₩n",
206                 getObjectID(man), getMankindName( MANKIND(man) ) );
207
208       freeObject(man);
209
210       return 0;
211  }
```

지금까지의 소소 코드보다 꽤 길어졌져? 복사해서 컴파일 하지 마세여.. 직접 보고 쳐보세여. 별거 아닌 코드라도 다른 사람은 어떤 식으로 코딩을 하는지 따라해 보는 것도 좋은 공부가 될 수 있으니까여.

코드의 분석은 main() 함수에서 시작을 하도록 하겠습니다.

```
198 라인의 main() 함수에서 시작을 하도록 하져..

200 라인에서 Object 포인터 변수를 하나 선언합니다.
202 라인에서 createMankind() 라는 사람 객체 생성 함수를 호출해서 사람 객체를
    생성합니다. 그리고 이 함수는 사람 객체 구조체형인 Mankind * 형을 되돌리지
    않고 Object * 를 되돌립니다.
    바로 여기에서 포인터의 오묘함이 숨어 있습니다.
    그럼 여기서 createMankind() 함수로 가보기로 하겠습니다.

117 라인부터 createMankind() 함수의 코드가 있습니다.
    주석에 적혀 있는 것을 잘 읽어 본 다음에 다음 설명을 보세여.
```

위의 예제 코드는 C 로 객체 지향 프로그래밍을 하기 위한 구조체를 만든것입니다. C 가 객체 지향 문법을 지원하지는 않지만 구조체를 잘 이용해서 객체지향적인 프로그래밍을 얼마든지 할 수가 있습니다.

아래의 그림은 객체의 구조도를 나타내는 것입니다.

```
+----------+
|  Object  |                  이 그림은 Object 구조체가 가장 기본적인
+----+-----+                  구조체 이며, Mankind 구조체는 Object 의
     |                       속성들을 상속한 것을 보여줍니다.
     |      +------------+      그림이 너무 간단해서 좀 그렇지만,
     +------|  Mankind   |      다음 강좌에서 더욱 심화된 것을 보면 그런
            +------------+      생각이 없어질거예여 :)
```

C 에서 구조체를 사용해서 상속을 받는 것은 부모 구조체 타입의 변수를 자식 구조체 에서 선언을 하는 것입니다. 여기서 부모 구조체는 Object 구조체이고 자식 구조체는 Mankind 구조체 입니다.

C++ 로 설명을 하면 Object 는 Mankind 의 super class 이고, Mankind 는 Object 의 child class 가 되는 겁니다.

26 라인에서 Mankind 구조체(struct _mankind)가 선언되어 있는데, 이 구조체의 첫번째 멤버 변수가 바로 Object 타입입니다. 즉, Object 구조체의 속성들을 사용하고자 함이지여.

그리고 29 라인의 name 멤버는 Object 의 속성에는 없는 사람객체에 부가적으로 필요한 속성인 것입니다.

사람도 역시 객체 이므로 객체를 나타내는 구조체인 Object 를 상속받고, 객체의 특성 외에 사람만이 갖는 또다른 속성을 부가적으로 더해주는 것입니다.

이렇게 함으로써 사람 객체(Mankind)는 객체(Object)의 속성을 전부 상속받게 된것입니다.

그리고 예를 들어서 학생을 나타내는 Student 객체를 표현하기 위한 구조체를 만든다고 가정을 해보면,학생 객체 역시 사람의 특성을 다 갖고 있어야 겠져? 따라서 학생 구조체에서 Mankind 타입의 멤버 변수를 선언하면 학생 구조체는 사람 객체의 모든 속성을 상속받게 되는 것입니다.

```
struct _student
{
    Mankind      mankind;
    char *       schoolName;
    int          score;
};
```

이렇게 하면 학생 구조체가 만들어지는 것입니다. 사람 객체 구조체인 Mankind 타입의 변수를 선언함으로써 사람 객체의 속성을 상속 받고 나머지 멤버 변수들은 사람 객체에는 없는 학생만의 속성을 부가한 것입니다.

이런 식으로 객체를 나타내는 구조체를 만들어 갈 수가 있는 것입니다.

하지만 꼭 염두해 두어야 할 것이 있습니다. 상속받는 구조체 타입의 멤버 변수는 꼭 제일 처음에 선언을 해야 합니다. 그 이유는 뒤에서 설명을 하겠지만 구조체의 캐스팅을 할 경우에 부모 객체 혹은 자식 객체를 다루는 함수를 사용하기 위함입니다.

다른 객체 형식으로 캐스팅을 하게 되면 객체가 저장되어 있는 메모리는 변함이 없지만 사용하는 방법이 바뀌게 됩니다. 즉, 그 메모리를 참조하기 위한 index 를 바꾸어 사용하는 것입니다.

글로만 설명을 하면 이해가 어려울 수 있으니 그림으로 그려보겠습니다.

이 그림은 Mankind 형의 객체를 생성했을 때 메모리에 변수들이 할당된 모습입니다. 이 변수들이 할당되어 있는 곳의 주소값을 저장하고 있는 포인터 변수 이름을 man 이라 하고, (Object *)man 이렇게 캐스팅을 하면 man 이 가리키고 있는 곳의 메모리를 접근하는 index 로 Object 구조체를 사용하겠다는 말입니다.

Object 구조체에 선언되어 있는 변수는 type 과 ID 져? 즉, man 이 가리키고 있는 곳에서 type 과 ID 를 사용하겠다는 뜻이 됩니다.

이번엔 (Mankind *)man 이렇게 캐스팅을 하면 man 이 가리키고 있는 곳의 메모리를 접근하는 index 로 Mankind 구조체를 사용하겠다는 말이 됩니다. Mankind 구조체는 Object 구조체형 멤버 변수가 처음에 선언되어 있기 때문에 type 과 ID 를 사용할 수도 있고 name 도 사용을 할 수가 있게 되는 것입니다.

이런식으로 객체는 부모, 자식 관계에 있는 객체 타입으로 바뀌어서 사용을 할 수가 있게 되는 것입니다.

이런 구조를 설명하는 이유는 객체 지향 프로그래밍에 관해서 공부를 하면 자세히 알수가 있습니다. 다만 이 강좌에서는 구조체와 포인터를 사용해서 객체 지향 프로그래밍을 할 수 있다는 것을 보여줌과 동시에 포인터의 위력을 보여주기 위함 이므로 객체 지향 프로그래밍의 장점에 대해서는 간단하게 설명하겠습니다.

객체 지향 프로그래밍을 하게 되면, 부모 객체의 내용을 업그레이드 하면 그 객체를 상속받은 자식 객체의 내용까지 저절로 업그레이드가 되는 것입니다. 그래서 유지 보수가 편리해지고 대단위 프로젝트를 수행할때 시간 및 인력의 절약과 같은 이점이 있습니다.

예제 코드에서 Object 객체를 업그레이드 하면 Mankind 객체도 저절로 업그레이드 되는 것입니다.

그리고 이런 코딩 방식의 이점으로는 모든 객체를 하나의 구조체 타입으로 다룰수 가 있습니다. 즉, Object 객체이건 Mankind 객체이건, Mankind 를 상속받은 Student 객체이건

전부 Object 포인터로 다룰수가 있게 되는 것입니다. 객체를 생성하는 함수는 다를 수 있지만 그 객체를 다루는 함수를 Object * 형식 에 맞추어서 작성하게 되면 라이브러리를 작성할때는 좀 더 복잡하긴 하겠지만 그 라이브러리를 사용해서 프로그래밍 할때는 아주 편리하게 되는 것입니다.

위의 예제 코드에서는 freeObject() 함수가 바로 그런 맥락의 함수 입니다. freeObject() 함수로 모든 형식의 객체를 파괴할 수가 있는 것입니다. 물론 freeObject() 함수는 각 형식의 객체를 파괴하는 함수를 switch 를 통해서 호출하고 있습니다.

하지만 이 라이브러리를 사용해서 프로그래밍하는 입장에서는 객체를 파괴할때는 freeObject() 만 알고 있으면 되는 것입니다. Mankind 객체를 파괴할때 freeObject() 함수가 내부적으로 freeMankind() 함수를 호출하는 것을 모르더라도 freeObject() 함수만 사용하면 모든 객체를 파괴할 수가 있는 것입니다.

자.. side 설명이 조금 길어졌군여..

다시 코드로 돌아가서 117 라인의 createMankind() 함수로 가겠습니다.

```
145 라인에서 Object 포인터 변수인 mankind 를 선언하고 malloc() 함수를 호출해서
    Mankind 구조체 크기만큼의 메모리를 힙(Heap)영역에 할당하고 그 첫번지를
    리턴 받습니다. 할당된 메모리는 Mankind 크기이지만 (Object *)로 캐스팅을
          하여 Object 객체로 사용을 하게 되는 겁니다.

149 라인은 할당 받은 메모리를 0 으로 초기화 하는 것입니다.
    malloc() 함수가 메모리를 할당할때 사용 가능한 메모리 영역중 요구하는 크기로
    메모리를 할당만 할 뿐 그 할당받은 메모리 내에는 할당 받기 이전에 사용이
    되던, 이제는 쓸모 없는 쓰레기 값이 담겨져 있으므로 깨끗하게 0 으로 세팅을
    해 주는 것입니다.

154 라인은 mankind 객체의 종류를 MANKIND_TYPE 으로 설정합니다.
    이것은 매우 중요합니다. 그 객체의 종류를 설정함으로써 내부적으로 캐스팅
    되었을 때도 자신의 주체성(?)을 잃지 않을 수가 있는 것입니다.
    mankind 객체가 사람 객체지만 Object 객체로 캐스팅이 되었다 하더라도
          자신의 근본은 사람이라는 것을 저장하고 있어야 하지 않겠어여?
    setObjectType() 함수는 73 라인에 선언되어 있습니다.

157 라인은 생성된 객체에게 고유한 일련번호를 부여하는 것입니다.
    setObjectID() 함수는 49 라인에 선언되어 있습니다.
```

자 다시 main() 함수로 돌아가서,

```
203 라인은 man 객체(사람 객체로 생성되었고 현재는 Object 형식으로 캐스팅 되어
    있음)의 이름을 설정하는 함수를 호출하는 것입니다.
    이 함수를 호출 할 때, (Object *)로 캐스팅 되어 있는 man 객체를
    Mankind 객체 형식으로 캐스팅 해서 인자로 전달합니다.
    MANKIND() 매크로 함수가 바로 그것입니다. 7 라인에 보면 MANKIND() 매크로
```

```
    함수가 선언되어 있습니다.
```

setMankindName() 함수가 선언되어 있는 곳으로 가서,

```
171 라인은 사람 객체의 이름을 저장하기 위해서 name 멤버 변수에 메모리를 할당
    해주는 것입니다.

175 라인에서 두번째 인자로 들어온 문자열을 사람 객체의 name 속성에 복사를
    합니다.
```

자, 다시 main() 함수로 돌아와서,

```
205 라인에서 man 객체의 속성들을 출력을 해 줍니다.
    각 사용된 함수의 코드를 보면 그렇게 어렵지 않게 이해를 할 수가 있을 것
    입니다.

208 라인에서 사람 객체인 man 객체를 파괴합니다.
    위에서 설명 했듯이 사람 객체이든 Object 객체이든 상관 없이 freeObject()
    함수를 호출해서 생성된 객체를 파괴합니다.
```

그럼 freeObject() 함수를 분석해 보기로 하져. 94 라인부터 선언되어 있습니다.

```
102 라인에서 그 객체의 본질을 알아냅니다. Object 객체인지, 사람 객체인지..

105 라인의 switch() 문에 의해서 그 객체의 본질에 맞는 파괴함수를 호출합니다.
    그렇게 함으로써 그 객체의 속성 중에서 메모리를 할당 받은 것을 빠짐 없이
    해제할 수가 있습니다.

114 라인에서 마지막으로 create*() 함수군에 의해서 생성된 객체의 메모리를
    해제합니다.
```

소스 코드에 자세하게 주석이 적혀 있으므로 코드 분석과 주석을 같이 보면서 코드를 살펴보면 내용을 이해할 수가 있을 겁니다.

위의 내용이 잘 이해가 가지 않으면 몇번이고 설명을 다시 읽어보시고, 포인터의 가장 기본적인 개념이었던 1, 2 번 강좌의 그림을 그려보시면서 이해를 해보세여. 시간이 걸리더라도 직접 종이에 그림을 그려가며 이해를 해야합니다.

7.2. 객체 지향 프로그래밍 2

저번 장에서 객체 지향에 대해서 알아보았습니다. 이번 장에서는 조금 더 복잡한 객체의 구조를 만들어 보겠습니다.

그럼 우선 다음 소스코드를 보세여.

코드가 꽤 깁니다. 약 1000 라인 정도 되는 군여.. 주석문때문에 그래여 ;)

```
 1  #include <stdio.h>
 2  #include <stdlib.h> /* malloc(), free() */
 3  #include <string.h> /* strlen(), strcpy() */
 4
 5  /*
 6   * 매크로 상수 선언
 7   */
 8  #define BUS_FARE               600      /* 버스의 요금 */
 9  #define TAXI_FARE             1500      /* 택시의 기본 요금 */
10  #define BUS_MAX_CAPACITY        50        /* 버스의 최대인원 */
11
12  /*
13   * 매크로 함수 선언
14   */
15  #define OBJECT(object)         ((Object *)(object))
16  #define MANKIND(object)        ((Mankind *)(object))
17  #define STUDENT(object)        ((Student *)(object))
18  #define TEACHER(object)        ((Teacher *)(object))
19  #define MERCHANT(object)       ((Merchant *)(object))
20  #define VEHICLE(object)        ((Vehicle *)(object))
21  #define BUS(object)            ((Bus *)(object))
22  #define TAXI(object)           ((Taxi *)(object))
23
24  /*
25   * 구조체 및 열거형 상수 선언
26   */
27  typedef struct _object         Object;
28
29  typedef struct _mankind        Mankind;
30  typedef struct _student        Student;
31  typedef struct _teacher        Teacher;
32  typedef struct _merchant       Merchant;
33
34  typedef struct _vehicle        Vehicle;
35  typedef struct _bus            Bus;
36  typedef struct _taxi           Taxi;
37
38  typedef enum _type             Type;
39  typedef enum _major            Major;
40  typedef enum _taxiCallType     TaxiCallType;
41
42
43  /* 객체들의 종류를 열거형으로 선언 */
44  enum _type
45  {
46      OBJECT_TYPE,      /* 객체       타입 */
47      MANKIND_TYPE,     /* 사람       타입 */
48      STUDENT_TYPE,     /* 학생       타입 */
```

```
49      TEACHER_TYPE,    /* 선생님    타입 */
50      MERCHANT_TYPE,   /* 상인      타입 */
51      VEHICLE_TYPE,    /* 운송수단 타입 */
52      BUS_TYPE,        /* 버스      타입 */
53      TAXI_TYPE        /* 택시      타입 */
54  };
55
56  /* 선생님의 전공과목을 열거형으로 선언 */
57  enum _major
58  {
59      MAJOR_KOREAN,    /* 국어 전공 */
60      MAJOR_ENGLISH,   /* 영어 전공 */
61      MAJOR_MATH,      /* 수학 전공 */
62      MAJOR_HISTORY,   /* 역사 전공 */
63      MAJOR_MUSIC      /* 음악 전공 */
64  };
65
66  /* 택시를 타게된 경위를 열거형으로 선언 */
67  enum _taxiCallType
68  {
69      TAXI_NORMAL_TYPE,    /* 보통 택시를 잡는 경우 */
70      TAXI_CALL_TYPE       /* 택시를 호출했을 경우  */
71  };
72
73  /* 객체의 가장 기본이 되는 구조체
74   * C++의 용어로 하자면 수퍼클래스가 되네여.
75   */
76  struct _object
77  {
78      Type              type;    /* 객체의 종류       */
79      unsigned long     ID;      /* 객체의 일련 번호 */
80  };
81
82  /* 사람 구조체.
83   * Object 의 속성을 상속받는다.
84   */
85  struct _mankind
86  {
87      Object   object;           /* Object 속성 상속 */
88      char *   name;             /* 이름              */
89      int      height;           /* 키                */
90      int      weight;           /* 몸무게            */
91  };
92
93  /* 학생 구조체.
94   * 사람의 속성을 상속받는다.
95   */
96  struct _student
97  {
98      Mankind           mankind;     /* Mankind 속성을 상속 */
99      char *            schoolName; /* 학교 이름           */
100     unsigned long     sNumber;     /* 학생 번호           */
```

```
101     int             score;      /* 성적                 */
102 };
103
104 /* 선생님 구조체.
105  * 사람의 속성을 상속받는다.
106  */
107 struct _teacher
108 {
109     Mankind         mankind;    /* Mankind 속성을 상속 */
110     char *          schoolName; /* 학교 이름            */
111     unsigned long   tNumber;    /* 교원 번호            */
112     Major           major;      /* 전공 과목            */
113 };
114
115 /* 상인 구조체.
116  * 사람의 속성을 상속받는다.
117  */
118 struct _merchant
119 {
120     Mankind         mankind;    /* Mankind 속성을 상속 */
121     char *          storeName;  /* 점포 이름            */
122     unsigned long   mNumber;    /* 점포 번호            */
123     int             profits;    /* 년수익 (만원 단위)  */
124 };
125
126 /* 운송 매체 구조체.
127  * 객체의 속성을 상속받는다.
128  */
129 struct _vehicle
130 {
131     Object      object;         /* Object 속성을 상속          */
132     int         wheelNumber;    /* 바퀴 개수                   */
133     int         size;           /* 운송 매체의 크기 (cm 단위) */
134     int         weight;         /* 운송 매체의 무게 (kg 단위) */
135 };
136
137 /* 버스 구조체.
138  * 운송 매체의 속성을 상속 받는다.
139  */
140 struct _bus
141 {
142     Vehicle     vehicle;        /* Vehicle 속성을 상속     */
143     int         maxCapacity;    /* 실을 수 있는 최대 인원 */
144     int         fare;           /* 요금 (원 단위)          */
145 };
146
147 /* 택시 구조체.
148  * 운송 매체의 속성을 상속 받는다.
149  */
150 struct _taxi
151 {
152     Vehicle         vehicle;    /* Vehicle 속성을 상속 */
```

```
153     int             fare;       /* 요금 (원 단위)       */
154     int             extra;      /* 할증 요금 (원 단위) */
155     TaxiCallType    state;      /* 호출을 했는지, 아니면 그냥 세웠는지 */
156 };
157
158 /*
159  * 전역 변수 선언
160  */
161 unsigned long _objectID = 0;
162
163
164 /*
165  * 함수의 원형(prototype) 선언
166  */
167
168 /* 객체(Object) 관련 함수 */
169 void        setObjectType( Object *object, Type type );
170 void        setObjectID( Object *object );
171 Type        getObjectType( Object *object );
172 unsigned long getObjectID( Object *object );
173 void        freeObject( Object *object );
174
175 /* 사람(Mankind) 관련 함수 */
176 void        setMankindName( Mankind *mankind, char *name );
177 void        setMankindHeight( Mankind *mankind, int height );
178 void        setMankindWeight( Mankind *mankind, int weight );
179 char *      getMankindName( Mankind *mankind );
180 int         getMankindHeight( Mankind *mankind );
181 int         getMankindWeight( Mankind *mankind );
182 void        freeMankind( Mankind *mankind );
183
184 /* 학생(Student) 관련 함수 */
185 Object *    createStudent(void);
186 void        setStudentSchoolName( Student *student, char *schoolName );
187 void        setStudentNumber( Student *student, unsigned long number );
188 void        setStudentScore( Student *student, int score );
189 char *      getStudentSchoolName( Student *student );
190 unsigned long getStudentNumber( Student *student );
191 int         getStudentScore( Student *student );
192 void        freeStudent( Student *student );
193
194 /* 선생님(Teacher) 관련 함수 */
195 Object *    createTeacher(void);
196 void        setTeacherSchoolName( Teacher *teacher, char *schoolName );
197 void        setTeacherNumber( Teacher *teacher, unsigned long number );
198 void        setTeacherMajor( Teacher *teacher, Major major );
199 char *      getTeacherSchoolName( Teacher *teacher );
200 unsigned long getTeacherNumber( Teacher *teacher );
201 Major       getTeacherMajor( Teacher *teacher );
202 void        freeTeacher( Teacher *teacher );
203
204 /* 상인(Merchant) 관련 함수 */
```

```
205 Object *      createMerchant(void);
206 void          setMerchantStoreName( Merchant *merchant, char *storeName );
207 void          setMerchantNumber( Merchant *merchant, unsigned long number );
208 void          setMerchantProfits( Merchant *merchant, int profits );
209 char *        getMerchantStoreName( Merchant *merchant );
210 unsigned long getMerchantNumber( Merchant *merchant );
211 int           getMerchantProfits( Merchant *merchant );
212 void          freeMerchant( Merchant *merchant );
213
214 /* 운송수단(Vehicle) 관련 함수 */
215 void          setVehicleWheelNumber( Vehicle *vehicle, int wheelNumber );
216 void          setVehicleSize( Vehicle *vehicle, int size );
217 void          setVehicleWeight( Vehicle *vehicle, int weight );
218 int           getVehicleWheelNumber( Vehicle *vehicle );
219 int           getVehicleSize( Vehicle *vehicle );
220 int           getVehicleWeight( Vehicle *vehicle );
221 void          freeVehicle( Vehicle *vehicle );
222
223 /* 버스(Bus) 관련 함수 */
224 Object *      createBus(void);
225 void          setBusMaxCapacity( Bus *bus, int number );
226 void          setBusFare( Bus *bus, int fare );
227 int           getBusMaxCapacity( Bus *bus );
228 int           getBusFare( Bus *bus );
229 void          freeBus( Bus *bus );
230
231 /* 택시(Taxi) 관련 함수 */
232 Object *      createTaxi(void);
233 void          setTaxiFare( Taxi *taxi, int fare );
234 void          setTaxiExtraFare( Taxi *taxi, int fare );
235 void          setTaxiState( Taxi *taxi, TaxiCallType type );
236 int           getTaxiFare( Taxi *taxi );
237 int           getTaxiExtraFare( Taxi *taxi );
238 TaxiCallType getTaxiState( Taxi *taxi );
239 void          freeTaxi( Taxi *taxi );
240
241
242 /* freeObject
243  *
244  * 생성된 객체의 할당받은 메모리를 해제한다.
245  * Object 의 type 을 조사하여 각 type 에 맞는 해제 함수를 호출한다.
246  */
247 void freeObject( Object *object )
248 {
249     /* 객체(object)의 type 을 구한다. */
250     Type type = getObjectType( object );
251
252     switch( type )
253     {
254         case OBJECT_TYPE :
255             break;
256         case MANKIND_TYPE :
```

```
257                 freeMankind( MANKIND(object) ); break;
258             case STUDENT_TYPE :
259                 freeStudent( STUDENT(object) ); break;
260             case TEACHER_TYPE :
261                 freeTeacher( TEACHER(object) ); break;
262             case MERCHANT_TYPE :
263                 freeMerchant( MERCHANT(object) ); break;
264             case VEHICLE_TYPE :
265                 freeVehicle( VEHICLE(object) ); break;
266             case BUS_TYPE :
267                 freeBus( BUS(object) ); break;
268             case TAXI_TYPE :
269                 freeTaxi( TAXI(object) ); break;
270         }
271
272         /* 마지막으로 각 type 의 크기만큼 할당된 메모리를 해제한다. */
273         free( object );
274     }
275
276     /* setObjectType
277      *
278      * 객체의 타입을 설정한다.
279      */
280     void setObjectType( Object *object, Type type )
281     {
282         object->type = type;
283     }
284
285     /* setObjectID
286      *
287      * 객체의 고유 일련번호를 설정한다.
288      * 이 함수는 사용자가 마음대로 사용할 목적이 아니라,
289      * 객체가 생성되었을 때 한번 호출해서 객체에게 일련번호를
290      * 부여하는 것이다.
291      *
292      * 전역변수인 _objectID 은 객체에게 일련번호를 부여하고 난 후,
293      * 1 씩 증가한다.
294      */
295     void setObjectID( Object *object )
296     {
297         object->ID = _objectID++;
298     }
299
300     /* getObjectType
301      *
302      * 객체의 type 을 구한다.
303      * 객체의 type 은 객체의 특성을 말한다.
304      *
305      * 객체 생성함수에 따라서 그 객체의 type 은 설정된다.
306      */
307     Type getObjectType( Object *object )
308     {
309         return object->type;
```

```
310  }
311
312  /* getObjectID
313   *
314   * 객체의 고유 일련번호(ID)를 구한다.
315   */
316  unsigned long getObjectID( Object *object )
317  {
318       return object->ID;
319  }
320
321  /* setMankindName
322   *
323   * 사람 객체의 이름속성을 설정한다.
324   */
325  void setMankindName( Mankind *mankind, char *name )
326  {
327       mankind->name = (char *)malloc( strlen(name) + 1 );
328       if( mankind->name == NULL ) return;
329       strcpy( mankind->name, name );
330  }
331
332  /* getMankindName
333   *
334   * 사람 객체의 이름 속성을 구한다.
335   */
336  char *getMankindName( Mankind *mankind )
337  {
338       return mankind->name;
339  }
340
341  /* setMankindHeight
342   *
343   * 사람 객체의 키 속성을 설정한다.
344   */
345  void setMankindHeight( Mankind *mankind, int height )
346  {
347       mankind->height = height;
348  }
349
350  /* getMankindHeight
351   *
352   * 사람 객체의 키 속성을 구한다.
353   */
354  int getMankindHeight( Mankind *mankind )
355  {
356       return mankind->height;
357  }
358
359  /* setMankindWeight
360   *
361   * 사람 객체의 몸무게 속성을 설정한다.
362   */
```

```
363  void setMankindWeight( Mankind *mankind, int weight )
364  {
365       mankind->weight = weight;
366  }
367
368  /* getMankindWeight
369   *
370   * 사람 객체의 몸무게 속성을 구한다.
371   */
372  int getMankindWeight( Mankind *mankind )
373  {
374       return mankind->weight;
375  }
376
377  /* freeMankind
378   *
379   * 사람 객체의 속성을 해제한다.
380   *
381   * 객체의 속성을 해제하는 free*() 함수들은
382   * 각 객체의 속성중에서 malloc()에 의해서 메모리를 할당 받은 속성에 대한
383   * 메모리 해제가 이루어진다.
384   *
385   * 각 객체의 파괴함수는 자기의 부모 객체에서 물려받은 속성들의 메모리를
386   * 해제하기 위하여 부모 객체의 파괴함수를 호출한다.
387   * 여기서 Object 에 대한 파괴함수는 대표 파괴함수 이므로
388   * 가장 상단에 존재하고 있는 Object 객체의 파괴함수는 없다.
389   *
390   * freeObject() 함수는 어떤 type 을 가진 객체라도 파괴할 수 있도록
391   * 각 객체의 type 을 조사하여 그 객체의 파괴함수를 호출한다.
392   */
393  void freeMankind( Mankind *mankind )
394  {
395       /* 사람 객체의 이름 속성이 NULL 이 아니면 메모리를 해제한다. */
396       if( mankind->name != NULL )
397            free( mankind->name );
398  }
399
400  /* createStudent
401   *
402   * 학생 객체를 생성하는 함수이다.
403   *
404   * 모든 객체는 create*() 함수로 생성되어지며, 그 객체의 데이터 형은
405   * Object * 이다.
406   * 모든 객체의 구조체 타입이 다 다르지만 Object 구조체로 모든 객체를
407   * 다룰 수 있도록 하기 위해서 메모리는 sizeof(Student) 만큼 할당되지만
408   * 그 할당 받은 메모리를 (Object *)로 캐스팅한다.
409   *
410   * 그렇다고 해서 할당받은 메모리의 크기가 변한다던지 하는게 아니라
411   * 단지 그 할당받은 메모리를 참조하기 위한 index 로 Object 구조체를
412   * 사용하겠다는 것만 알려주는 것일 뿐이다.
413   *
414   * 실제로 Student 구조체에 있는 속성을 사용하기 위해서,
```

```
415   * (object *)로 캐스팅된 메모리는 다시 (Student *)로 캐스팅되야 한다.
416   * STUDENT(object) 매크로 함수를 사용해도 된다.
417   *
418   * 이렇게 캐스팅을 통해서,
419   * 많은 종류의 객체들을 대표적인 객체 제어 함수로 제어를 할 수 있게 된다.
420   * 예를 들면 어떤 타입의 객체라도 freeObject() 함수로 파괴를 할 수가
421   * 있게 되는 것이다.
422   */
423  Object *createStudent(void)
424  {
425      /* Student 구조체 크기 만큼의 메모리를 할당 받은 후
426       * (Object *)로 캐스팅 한다.
427       */
428      Object *student = (Object *)malloc( sizeof(Student) );
429      if( student == NULL ) return NULL;
430
431      /* 할당 받은 메모리를 0 으로 초기화 한다. */
432      memset( student, 0, sizeof(Student) );
433
434      /* 생성된 객체의 type 을 STUDENT_TYPE 으로 설정한다.
435       * 즉, 이 객체는 학생 객체임을 뜻한다.
436       */
437      setObjectType( student, STUDENT_TYPE );
438
439      /* 생성된 객체의 고유 일련번호를 부여한다. */
440      setObjectID( student );
441
442      return student;
443  }
444
445  /* setStudentSchoolName
446   *
447   * 학생 객체의 학교 이름 속성을 설정한다.
448   */
449  void setStudentSchoolName( Student *student, char *schoolName )
450  {
451      /* schoolName 인자의 문자열 길이 + 1 만큼 메모리를 할당한다.
452       * 1 을 더하는 이유는 문자열 마지막에 NULL 을 저장하기 위함이다.
453       */
454      student->schoolName = (char *)malloc( strlen(schoolName) + 1 );
455      if( student->schoolName == NULL ) return;
456
457      /* 학생 객체의 학교이름 속성에 schoolName 을 복사한다. */
458      strcpy( student->schoolName, schoolName );
459  }
460
461  /* getStudentSchoolName
462   *
463   * 학생 객체의 학교 이름 속성을 구한다.
464   */
465  char *getStudentSchoolName( Student *student )
466  {
```

```
467         return student->schoolName;
468     }
469
470     /* setStudentNumber
471      *
472      * 학생 객체의 학생 번호 속성을 설정한다.
473      */
474     void setStudentNumber( Student *student, unsigned long number )
475     {
476         student->sNumber = number;
477     }
478
479     /* getStudentNumber
480      *
481      * 학생 객체의 학생 번호 속성을 구한다.
482      */
483     unsigned long getStudentNumber( Student *student )
484     {
485         return student->sNumber;
486     }
487
488     /* setStudentScore
489      *
490      * 학생 객체의 성적 속성을 설정한다.
491      */
492     void setStudentScore( Student *student, int score )
493     {
494         student->score = score;
495     }
496
497     /* getStudentScore
498      *
499      * 학생 객체의 성적 속성을 구한다.
500      */
501     int getStudentScore( Student *student )
502     {
503         return student->score;
504     }
505
506     /* freeStudent
507      *
508      * 학생 객체의 속성을 해제하는 파괴함수.
509      *
510      * 학생의 부모 객체는 사람 객체이므로 학생에 대한 속성을 해제한 후,
511      * 사람 객체 파괴함수를 호출한다.
512      * 이렇게 함으로써 자칫 메모리 누수가 되는 것을 꼼꼼이 방지할 수 있다.
513      */
514     void freeStudent( Student *student )
515     {
516         /*
517          * 학생 객체의 속성을 파괴
518          */
```

```
519
520         /* 학교 이름 속성 해제 */
521         if( student->schoolName != NULL )
522             free( student->schoolName );
523
524         /*
525          * 사람 객체 파괴 함수 호출
526          */
527         freeMankind( MANKIND(student) );
528     }
529
530     /* createTeacher
531      *
532      * 선생님 객체를 생성하는 함수이다.
533      */
534     Object *createTeacher(void)
535     {
536         /* Teacher 구조체 크기 만큼의 메모리를 할당 받은 후
537          * (Object *)로 캐스팅 한다.
538          */
539         Object *teacher = (Object *)malloc( sizeof(Teacher) );
540         if( teacher == NULL ) return NULL;
541
542         /* 할당 받은 메모리를 0 으로 초기화 한다. */
543         memset( teacher, 0, sizeof(Teacher) );
544
545         /* 생성된 객체의 type 을 TEACHER_TYPE 으로 설정한다.
546          * 즉, 이 객체는 선생님 객체임을 뜻한다.
547          */
548         setObjectType( teacher, TEACHER_TYPE );
549
550         /* 생성된 객체의 고유 일련번호를 부여한다. */
551         setObjectID( teacher );
552
553         return teacher;
554     }
555
556     /* setTeacherSchoolName
557      *
558      * 선생님 객체의 학교 이름 속성을 설정한다.
559      */
560     void setTeacherSchoolName( Teacher *teacher, char *schoolName )
561     {
562         /* 학교 이름 속성을 저장하기 위해 메모리를 할당한다. */
563         teacher->schoolName = (char *)malloc( strlen(schoolName) + 1 );
564         if( teacher->schoolName == NULL ) return;
565
566         /* 속성에 학교 이름을 복사한다. */
567         strcpy( teacher->schoolName, schoolName );
568     }
569
570     /* getTeacherSchoolName
```

```
571   *
572   * 선생님 객체의 학교 이름 속성을 구한다.
573   */
574  char *getTeacherSchoolName( Teacher *teacher )
575  {
576       return teacher->schoolName;
577  }
578
579  /* setTeacherNumber
580   *
581   * 선생님 객체의 교원 번호 속성을 설정한다.
582   */
583  void setTeacherNumber( Teacher *teacher, unsigned long number )
584  {
585       teacher->tNumber = number;
586  }
587
588  /* getTeacherNumber
589   *
590   * 선생님 객체의 교원 번호 속성을 구한다.
591   */
592  unsigned long getTeacherNumber( Teacher *teacher )
593  {
594       return teacher->tNumber;
595  }
596
597  /* setTeacherMajor
598   *
599   * 선생님 객체의 전공 과목 속성을 설정한다.
600   */
601  void setTeacherMajor( Teacher *teacher, Major major )
602  {
603       teacher->major = major;
604  }
605
606  /* getTeacherMajor
607   *
608   * 선생님 객체의 전공 과목 속성을 구한다.
609   */
610  Major getTeacherMajor( Teacher *teacher )
611  {
612       return teacher->major;
613  }
614
615  /* freeTeacher
616   *
617   * 선생님 객체의 속성을 해제하는 파괴함수.
618   * 선생님의 부모 객체는 사람 객체이므로 선생님에 대한 속성을 해제한 후,
619   * 사람 객체 파괴함수를 호출한다.
620   */
621  void freeTeacher( Teacher *teacher )
622  {
```

```
623         /*
624          * 선생님 객체의 속성을 파괴
625          */
626
627         /* 학교 이름 속성을 해제 */
628         if( teacher->schoolName != NULL )
629             free( teacher->schoolName );
630
631         /*
632          * 사람 객체 파괴 함수 호출
633          */
634         freeMankind( MANKIND(teacher) );
635     }
636
637     /* createMerchant
638      *
639      * 상인 객체 생성 함수이다.
640      */
641     Object *createMerchant(void)
642     {
643         /* Merchant 구조체 크기 만큼의 메모리를 할당 받은 후
644          * (Object *)로 캐스팅 한다.
645          */
646         Object *merchant = (Object *)malloc( sizeof(Merchant) );
647         if( merchant == NULL ) return NULL;
648
649         /* 할당 받은 메모리를 0 으로 초기화 한다. */
650         memset( merchant, 0, sizeof(Merchant) );
651
652         /* 생성된 객체의 type 을 MERCHANT_TYPE 으로 설정한다.
653          * 즉, 이 객체는 상인 객체임을 뜻한다.
654          */
655         setObjectType( merchant, MERCHANT_TYPE );
656
657         /* 생성된 객체의 고유 일련번호를 부여한다. */
658         setObjectID( merchant );
659
660         return merchant;
661     }
662
663     /* setMerchantStoreName
664      *
665      * 상인 객체의 점포 이름 속성을 설정한다.
666      */
667     void setMerchantStoreName( Merchant *merchant, char *storeName )
668     {
669         /* 점포 이름 속성을 저장하기 위해 메모리를 할당한다. */
670         merchant->storeName = (char *)malloc( strlen(storeName) + 1 );
671         if( merchant->storeName == NULL ) return;
672
673         /* 속성에 점포 이름을 복사한다. */
674         strcpy( merchant->storeName, storeName );
```

```
675  }
676
```

```
677  /* getMerchantStoreName
678   *
679   * 상인 객체의 점포 이름 속성을 구한다.
680   */
681  char *getMerchantStoreName( Merchant *merchant )
682  {
683       return merchant->storeName;
684  }
685
686  /* setMerchantNumber
687   *
688   * 상인 객체의 점포 번호 속성을 설정한다.
689   */
690  void setMerchantNumber( Merchant *merchant, unsigned long number )
691  {
692       merchant->mNumber = number;
693  }
694
695  /* getMerchantNumber
696   *
697   * 상인 객체의 점포 번호 속성을 구한다.
698   */
699  unsigned long getMerchantNumber( Merchant *merchant )
700  {
701       return merchant->mNumber;
702  }
703
704  /* setMerchantProfits
705   *
706   * 상인 객체의 연수익 속성을 설정한다.
707   */
708  void setMerchantProfits( Merchant *merchant, int profits )
709  {
710       merchant->profits = profits;
711  }
712
713  /* getMerchantProfits
714   *
715   * 상인 객체의 연수익 속성을 구한다.
716   */
717  int getMerchantProfits( Merchant *merchant )
718  {
719       return merchant->profits;
720  }
721
722  /* freeMerchant
723   *
724   * 상인 객체의 속성을 해제하는 파괴함수.
725   * 상인의 부모 객체는 사람 객체이므로 상인에 대한 속성을 해제한 후,
726   * 사람 객체 파괴함수를 호출한다.
```

```
727   */
728  void freeMerchant( Merchant *merchant )
729  {
730       /*
731        * 상인 객체의 속성을 파괴
732        */
733
734       /* 점포 이름 속성을 해제 */
735       if( merchant->storeName != NULL )
736            free( merchant->storeName );
737
738       /*
739        * 사람 객체의 파괴 함수 호출
740        */
741       freeMankind( MANKIND(merchant) );
742  }
743
744  /* setVehicleWheelNumber
745   *
746   * 운송 매체 객체의 바퀴 개수 속성을 설정한다.
747   */
748  void setVehicleWheelNumber( Vehicle *vehicle, int wheelNumber )
749  {
750       vehicle->wheelNumber = wheelNumber;
751  }
752
753  /* getVehicleWheelNumber
754   *
755   * 운송 매체 객체의 바퀴 개수 속성을 구한다.
756   */
757  int getVehicleWheelNumber( Vehicle *vehicle )
758  {
759       return vehicle->wheelNumber;
760  }
761
762  /* setVehicleSize
763   *
764   * 운송 매체 객체의 크기 속성을 설정한다.
765   */
766  void setVehicleSize( Vehicle *vehicle, int size )
767  {
768       vehicle->size = size;
769  }
770
771  /* getVehicleSize
772   *
773   * 운송 매체 객체의 크기 속성을 구한다.
774   */
775  int getVehicleSize( Vehicle *vehicle )
776  {
777       return vehicle->size;
778  }
```

```
779
780  /* setVehicleWeight
781   *
782   * 운송 매체 객체의 무게 속성을 설정한다.
783   */
784  void setVehicleWeight( Vehicle *vehicle, int weight )
785  {
786       vehicle->weight = weight;
787  }
788
789  /* getVehicleWeight
790   *
791   * 운송 매체 객체의 무게 속성을 구한다.
792   */
793  int getVehicleWeight( Vehicle *vehicle )
794  {
795       return vehicle->weight;
796  }
797
798  /* freeVehicle
799   *
800   * 운송 매체 객체의 속성을 해제하는 파괴함수.
801   */
802  void freeVehicle( Vehicle *vehicle )
803  {
804       /*
805        * 운송 매체 객체의 속성을 파괴
806        */
807  }
808
809  /* createBus
810   *
811   * 버스 객체를 생성하는 함수이다.
812   */
813  Object *createBus(void)
814  {
815       /* Bus 구조체 크기 만큼의 메모리를 할당 받은 후
816        * (Object *)로 캐스팅 한다.
817        */
818       Object *bus = (Object *)malloc( sizeof(Bus) );
819       if( bus == NULL ) return NULL;
820
821       /* 할당받은 메모리를 0 으로 초기화 한다. */
822       memset( bus, 0, sizeof(Bus) );
823
824       /* 생성된 객체의 type 을 BUS_TYPE 으로 설정한다.
825        * 즉, 이 객체는 버스 객체임을 뜻한다.
826        */
827       setObjectType( bus, BUS_TYPE );
828
829       /* 생성된 객체의 고유 일련번호를 부여한다. */
830       setObjectID( bus );
```

```
831
832         /*
833          * 버스의 부모 객체인 Vehicle 속성을 설정한다.
834          * 속성의 일반적인 값을 초기값으로 설정을 해주는 것이다.
835          */
836         setVehicleWheelNumber( VEHICLE(bus), 4 );
837         setVehicleSize( VEHICLE(bus), 5000 );
838         setVehicleWeight( VEHICLE(bus), 2500 );
839
840         /*
841          * 버스 객체의 속성을 설정한다.
842          * 이것 역시 일반 적인 값을 초기값으로 설정을 하는 것이다.
843          */
844         setBusMaxCapacity( BUS(bus), BUS_MAX_CAPACITY );
845         setBusFare( BUS(bus), BUS_FARE );
846
847         return bus;
848     }
849
850     /* setBusMaxCapacity
851      *
852      * 버스 객체의 최대 인원 속성을 설정한다.
853      */
854     void setBusMaxCapacity( Bus *bus, int number )
855     {
856         bus->maxCapacity = number;
857     }
858
859     /* getBusMaxCapacity
860      *
861      * 버스 객체의 최대 인원 속성을 구한다.
862      */
863     int getBusMaxCapacity( Bus *bus )
864     {
865         return bus->maxCapacity;
866     }
867
868     /* setBusFare
869      *
870      * 버스 객체의 요금 속성을 설정한다.
871      */
872     void setBusFare( Bus *bus, int fare )
873     {
874         bus->fare = fare;
875     }
876
877     /* getBusFare
878      *
879      * 버스 객체의 요금 속성을 구한다.
880      */
881     int getBusFare( Bus *bus )
882     {
```

```
883         return bus->fare;
884     }
```

```
885
886     /* freeBus
887      *
888      * 버스 객체의 속성을 해제하는 파괴함수.
889      */
890     void freeBus( Bus *bus )
891     {
892         /*
893          * 버스 객체의 속성을 파괴
894          */
895
896         /*
897          * 운송 매체 객체의 파괴 함수 호출
898          */
899         freeVehicle( VEHICLE(bus) );
900     }
901
902     /* createTaxi
903      *
904      * 택시 객체를 생성하는 함수
905      */
906     Object *createTaxi(void)
907     {
908         /* Taxi 구조체 크기 만큼의 메모리를 할당 받은 후
909          * (Object *)로 캐스팅 한다.
910          */
911         Object *taxi = (Object *)malloc( sizeof(Taxi) );
912         if( taxi == NULL ) return NULL;
913
914         /* 할당 받은 메모리를 0 으로 초기화 한다. */
915         memset( taxi, 0, sizeof(Taxi) );
916
917         /* 생성된 객체의 type 을 TAXI_TYPE 으로 설정한다.
918          * 즉, 이 객체는 택시 객체임을 뜻한다.
919          */
920         setObjectType( taxi, TAXI_TYPE );
921
922         /* 생성된 객체의 고유 일련번호를 부여한다. */
923         setObjectID( taxi );
924
925         /*
926          * 택시의 부모 객체인 Vehicle 속성을 설정한다.
927          * 속성의 일반적인 값을 초기값으로 설정을 해주는 것이다.
928          */
929         setVehicleWheelNumber( VEHICLE(taxi), 4 );
930         setVehicleSize( VEHICLE(taxi), 3000 );
931         setVehicleWeight( VEHICLE(taxi), 700 );
932
933         /*
934          * 택시 객체의 속성을 설정한다.
```

```
935         * 이것 역시 일반 적인 값을 초기값으로 설정을 하는 것이다.
936         */

937         setTaxiFare( TAXI(taxi), TAXI_FARE );
938         setTaxiFare( TAXI(taxi), 0 );
939         setTaxiFare( TAXI(taxi), TAXI_NORMAL_TYPE );
940
941         return taxi;
942     }
943
944     /* setTaxiFare
945      *
946      * 택시 객체의 요금 속성을 설정한다.
947      */
948     void setTaxiFare( Taxi *taxi, int fare )
949     {
950         taxi->fare = fare;
951     }
952
953     /* getTaxiFare
954      *
955      * 택시 객체의 요금 속성을 구한다.
956      */
957     int getTaxiFare( Taxi *taxi )
958     {
959         return taxi->fare;
960     }
961
962     /* setTaxiExtraFare
963      *
964      * 택시 객체의 할증 요금 속성을 설정한다.
965      */
966     void setTaxiExtraFare( Taxi *taxi, int fare )
967     {
968         taxi->extra = fare;
969     }
970
971     /* getTaxiExtraFare
972      *
973      * 택시 객체의 할증 요금 속성을 구한다.
974      */
975     int getTaxiExtraFare( Taxi *taxi )
976     {
977         return taxi->extra;
978     }
979
980     /* setTaxiState
981      *
982      * 택시 객체의 호출 상태 속성을 설정한다.
983      */
984     void setTaxiState( Taxi *taxi, TaxiCallType type )
985     {
986         taxi->state = type;
```

```
987  }
988
```

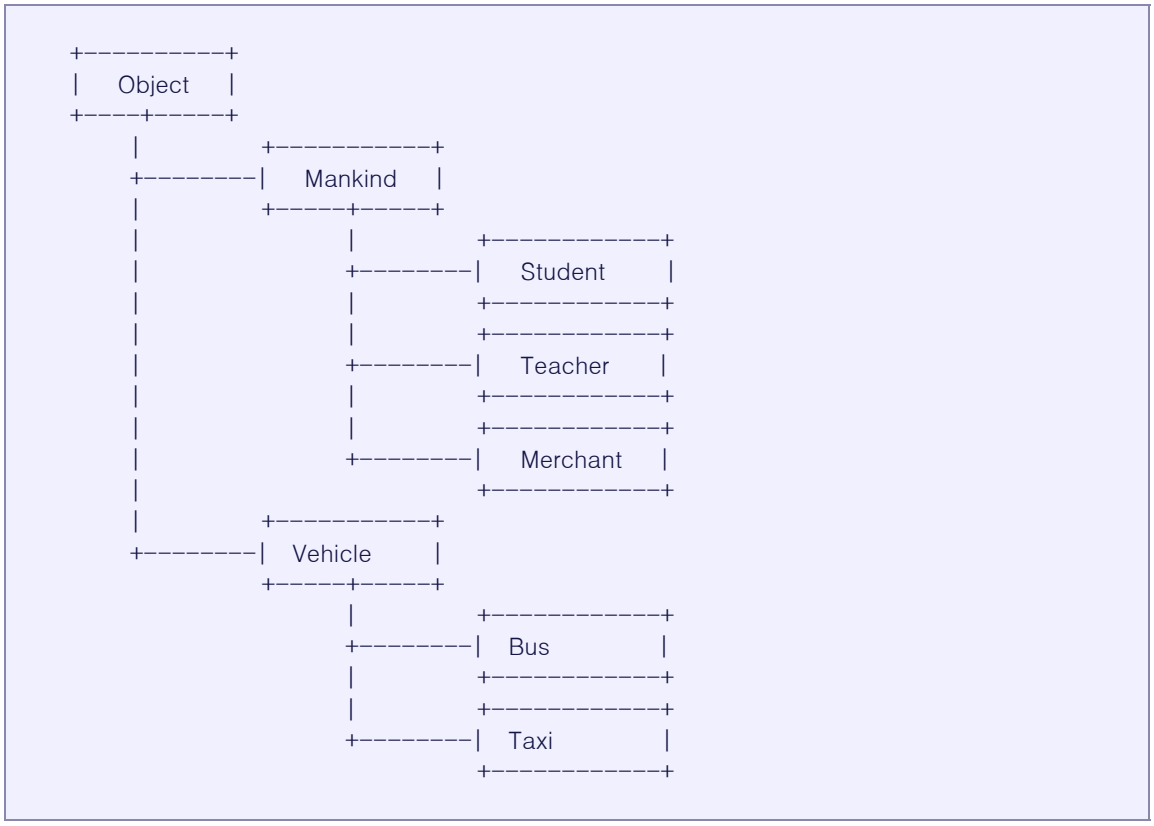
```
989  /* getTaxiState
990   *
991   * 택시 객체의 호출 상태 속성을 구한다.
992   */
993  TaxiCallType getTaxiState( Taxi *taxi )
994  {
995      return taxi->state;
996  }
997
998  /* freeTaxi
999   *
1000  * 택시 객체의 파괴 함수.
1001  */
1002 void freeTaxi( Taxi *taxi )
1003 {
1004     /*
1005      * 택시 객체의 속성을 파괴
1006      */
1007
1008     /*
1009      * 운송 매체 객체의 파괴 함수 호출
1010      */
1011     freeVehicle( VEHICLE(taxi) );
1012 }
1013
1014 /* main
1015  *
1016  * 원래 main() 함수의 원형은,
1017  *
1018  * int main( int argc, char **argv ); 입니다.
1019  *
1020  * argc 는 자신을 포함한 인자의 개수를 뜻하고,
1021  * argv 는 자신의 이름과 함께 인자들의 문자열을 갖고 있습니다.
1022  *
1023  * 예를 들어서 프로그램의 실행파일 이름이 exam 이라고 할때,
1024  *
1025  * $ exam a b c
1026  *
1027  * 이렇게 실행을 하게 되면 argc 는 4 가 되고,
1028  * argv[0] == "exam"
1029  * argv[1] == "a"
1030  * argv[2] == "b"
1031  * argv[3] == "c"
1032  *
1033  * 이렇게 되는 것이지여.
1034  *
1035  * 여기서는 main() 함수의 인자를 void 형으로 했습니다.
1036  * 그 이유는 인자를 받을 필요가 없기 때문이지여.
1037  *
1038  * main() 함수의 return 형이 int 라는 것을 주지하시기 바랍니다.
```

```
1039    * main() 함수도 OS(운영체제)의 입장에서 본다면 하나의 Sub function 에
1040    * 해당합니다. 그리고 main() 함수가 일을 다 마치고 난다음 프로그램을 종료할때
1041    * OS 에게 종료 값을 리턴합니다. OS 가 그 종료코드를 사용할 수 있도록 하는
1042    * 것이지여. 따라서 void main(); 과 같은 코드를 사용하지 마시고,
1043    * int main(void); 혹은 int main( int argc, char **argv ); 와 같은 코드를
1044    * 사용하도록 습관을 들이는 것이 중요합니다.
1045    */
1046   int main(void)
1047   {
1048       Object *student = NULL;
1049       Object *teacher = NULL;
1050       Object *merchant = NULL;
1051       Object *bus = NULL;
1052       Object *taxi = NULL;
1053
1054       student = createStudent();        /* 학생 객체 생성    */
1055       teacher = createTeacher();        /* 선생님 객체 생성 */
1056       merchant = createMerchant();      /* 상인 객체 생성    */
1057       bus = createBus();                /* 버스 객체 생성    */
1058       taxi = createTaxi();              /* 택시 객체 생성    */
1059
1060       printf("student ID = %ld₩n", getObjectID(student) );
1061       printf("teacher ID = %ld₩n", getObjectID(teacher) );
1062       printf("merchant ID = %ld₩n", getObjectID(merchant) );
1063       printf("bus ID = %ld, size = %d cm, weight = %d kg₩n",
1064               getObjectID(bus), getVehicleSize(VEHICLE(bus)),
1065               getVehicleWeight(VEHICLE(bus)) );
1066       printf("taxi ID = %ld, size = %d cm, weight = %d kg₩n",
1067               getObjectID(taxi), getVehicleSize(VEHICLE(taxi)),
1068               getVehicleWeight(VEHICLE(taxi)) );
1069
1070       freeObject(student);      /* 학생 객체 파괴    */
1071       freeObject(teacher);      /* 선생님 객체 파괴 */
1072       freeObject(merchant);     /* 상인 객체 파괴    */
1073       freeObject(bus);          /* 버스 객체 파괴    */
1074       freeObject(taxi);         /* 택시 객체 파괴    */
1075
1076       /* makes compiler happy :) */
1077       return 0;
1078   }
```

위의 코드는 설명을 하기 위한 코드라서 하나의 파일로 코딩했습니다. 실제로 코딩 을 할 때에는 각각의 객체마다 헤더 파일과 소스 코드 파일을 따로 만들어서 관리를 합니다. 그래야 나중에 확장을 시키거나 디버깅을 할때 편리합니다.

이번 강좌의 예제는 바로 전 강좌인 7 번째 강좌에 나왔던 예제를 좀 더 심화시킨 것 입니다. 저번 강좌의 예제를 이해 하셨다면 이번 예제도 그리 어려울게 없습니다.

그럼 우선 위의 객체 구조도를 그려보겠습니다.



Object 가 가장 상위 객체이며 Object 객체를 Mankind 와 Vehicle 객체가 상속받습니다. Mankind(사람) 객체를 Student(학생), Teacher(선생님), Merchant(상인) 객체가 다시 상속을 받고, Vehicle(운송수단) 객체를 Bus(버스), Taxi(택시) 객체가 다시 상속을 받습니다.

1046 라인의 main() 함수에서 하는 일은 student, teacher, merchant, bus, taxi 객체를 하나씩 만들고 그 객체들의 ID 및 기본 정보를 출력하고 나서 각 객체를 파괴하는 것입니다.

모든 객체들의 생성 함수는 Object 포인터 타입을 리턴함으로써 모든 객체를 Object 포인터로 다룰 수 있게 하고 있습니다. 7 번째 강좌에서도 설명 했듯이 많은 종류의 객체들을 전부 다른 형식으로 다루어야 한다면 아주 많이 불편할 것입니다. 라이브러리를 만들어야 하는 프로그래머라면 이렇게 객체 지향으로 프로그래밍하는 것이 훨씬 복잡하고 코드의 길이도 길어지지만 그렇게 작성된 라이브러리를 사용해서 프로그래밍하는 프로그래머는 프로그램을 작성 하는데 훨씬 간편하고 쉽게 작성을 할 수가 있는 것입니다.

247 라인의 freeObject() 함수를 보면, 내부에서 각각의 객체 속성에 맞게 파괴 함수를 호출해 주는 방식으로 생성된 모든 객체는 freeObject() 함수만 호출함으로써 파괴할 수가 있는 것입니다.

이러한 이점 외에도 객체 지향 프로그래밍의 장점은 프로그래밍의 방법이 바뀔 수가 있다는 점입니다. 기존의 프로그래밍에서는 데이터와 코드 및 action(계산 등등)등이 분리가 되지 않고 서로 복잡하게 얽혀 있었습니다. 하지만 이렇게 객체 지향적인 프로그래밍을 하게

되면 데이터와 동작의 분리를 할 수가 있고 그렇게 함으로써 부가적으로 디버깅 시간의 단축 및 인력을 줄일 수가 있는 것입니다.

물론 프로그래밍 언어에서 객체 지향 문법을 지원해야만 더 효율적으로 객체 관리가 되겠지만 C++이 아닌 C로 프로그래밍을 하더라도 객체 지향적인 개념을 도입해서 프로그래밍을 한다면 충분히 이점을 끌어 낼 수가 있습니다.

각 객체가 어떻게 생성이 되고 어떻게 사용이 되어지며 어떻게 파괴가 되는지에 대해서 다시 한번 7번째 강좌와 이번 강좌를 읽어보시고 그 객체들을 나타내기 위해서 사용된 포인터에 대해서 꼼꼼히 체크를 해보시기 바랍니다.

저번 강좌에서 객체를 나타내기 위해서 사용된 포인터에 대해서 설명을 했으므로 이번 강좌에서는 어떻게 확장을 해 나갈 수 있는지에 대해서 설명을 하겠습니다.

Student 객체를 만들기 위해서 Student 구조체를 만들었는데 이 구조체를 한 번 분석하도록 하겠습니다.

96 라인에 Student 구조체가 있습니다. 우선 Student(학생) 이라는 객체는 사람에서 파생이 되어지는 것입니다. 그러므로 Student 구조체 내에 Mankind 변수를 하나 만들어서 Mankind(사람) 객체의 속성을 상속받아야 합니다. 그렇게 되면 Mankind가 상속받은 Object 객체의 속성들도 자연히 따라서 상속되겠져? 그리고 Mankind 객체가 갖고 있지 않는 학생만의 고유한 속성을 부가적으로 선언해 주면 되는 것입니다.

새로운 객체를 만들 때 새로 만들 객체는 어떤 객체에게 속성을 상속 받아야 하는지를 잘 생각해서 상속을 받으면 객체를 새로 만드는 것이 그렇게 어렵지 않습니다.

예를 들어서 초등학생, 중학생, 고등학생, 대학생 과 같은 객체를 타나내기 위해서 구조체를 만든다고 생각해보면 당연히 새로 만들 객체들은 Student 객체를 상속받으면 편리하다는 것을 알수 있겠져?

그리고 부모 객체에 대한 함수를 사용할 때에는 생성된 객체의 포인터 타입을 부모의 객체 타입으로 바꾼 다음에 부모 객체를 위한 함수를 호출 하면 되는 것입니다. 예를 들면, Student 객체를 하나 생성했을 때, 몸무게를 설정해 보자구여.

Student 객체의 속성에는 몸무게에 대한 속성이 없습니다. 몸무게에 대한 속성은 부모 객체인 Mankind 객체의 속성입니다. Student 객체는 Mankind 객체를 상속 받았으므로 몸무게에 대한 속성을 사용할 수가 있습니다. 다음과 같이여..

```
setMankindWeight( MANKIND(student), 70 );
```

student 라는 학생 객체를 Mankind 객체로 캐스팅해서 setMankindWeight() 함수의 인자로 보내면 되는 거져.

C를 사용한 객체지향은 아무래도 언어 자체가 객체지향 문법을 지원하지 않기 때문에 불편한 점이 많습니다. 상속받은 속성들이 어떤 객체의 속성인지를 알고 있어야 그에 맞는 함수를 호출 할 수가 있기 때문입니다. 위에서 몸무게에 대한 속성이 어떤 객체의

속성인지를 알아야 setMankindWeight()과 같은 함수를 호출 할 수가 있게 되는 것이지여. 그래서 함수는 많아지지만 각 객체의 함수로 또 다시 작성을 해 주는 경우가 많습니다.

예를 들어서 setMankindWeight() 함수는 Mankind 객체의 함수이므로 Student 객체의 몸무게를 설정하기 위해서 setStudentWeight()과 같은 함수를 만들어 주게 되는 것입니다.

```
void setStudentWeight( Student *student, int weight )
{
    setMankindWeight( MANKIND(student), weight );
}
```

이런식으로 함수를 작성해 놓으면 학생 객체의 몸무게를 설정할 때 다음과 같이 함수를 호출하면 되는 것이지여.

```
setStudentWeight( STUDENT(student), 70 );
```

물론 이렇게 되면 객체가 상속을 많이 하면 할 수록 함수는 점점 더 많아지겠져? 그렇기 때문에 정확한 객체지향을 하기 위해서는 객체 지향을 지원하는 문법을 갖고 있는 언어로 작성을 하는 것이 좋습니다. 하지만 각 객체와 속성들을 어떻게 설계하느냐에 따라서 C 로도 훌륭하게 객체 지향 프로그래밍을 할 수가 있습니다.

객체지향에 대해서는 이정도로만 마무리를 짓겠습니다. 물론 저 뒤에 나올 강좌에서 또다시 언급이 될 것입니다. 포인터라는 것을 공부하는데는 아주 많은 분야의 내용들을 참조해야 할 것입니다. 포인터의 개념은 **2.1. 일반 변수**, **2.2. 단일 포인터**, **2.3. 이중 포인터**에서 이미 설명이 끝났지만 그 포인터의 활용을 보면서 포인터의 본질에 더더욱 깊게 다가설 수 있으면 좋겠습니다.

7.3. 단일 링크드 리스트

프로그래밍을 할 때 많이 사용되는 자료 구조로는 스택(stack), 큐(queue), 링크드 리스트(linked list), 트리(tree) 등이 있습니다. 그 외에도 탐색이나 정렬 등을 하 기 위한 기법들도 있지만, 자료구조를 위한 강좌가 아니므로 자료구조를 표현하는데 있어서 사용되는 포인터에 초점을 두도록 하겠습니다.

이번 장은 기본적인 자료구조인 링크드 리스트에 대해서 알아보겠습니다.

데이터를 저장하기 위해서 변수를 사용하고, 더 많은 공통된 자료들을 묶기 위해서 구조체를 사용하지여. 그리고 그 구조체 변수 배열을 선언 해서 많은 자료를 저장하고 사용합니다. 그런데 그 배열의 크기를 알고 있지 못할 때 최대로 큰 크기의 배열을 선언해서 작업을 하게 되면 자칫 많은 메모리 낭비를 초래할 수 있다는 얘기를 앞에 있는 강좌에서 언급을 했습니다.

그렇다면 동적으로 메모리를 할당하도록 하면 되지 않을까.. 라고 생각 하셨다면 그래도 50 점은 맞았습니다. 동적 메모리로 할당을 하면 메모리의 낭비는 막을 수 있습니다. 그렇다면 다음과 같은 구조체가 있다고 가정을 해보겠습니다.

```
struct _data
{
    int       number;
    int       kind;
    char      name[20];
    int       kor;
    int       eng;
    int       math;
    int       science;

    char      buffer[200];
};
```

그리고 위의 구조체를 사용해서 만개의 데이터를 표현하기 위해서 배열을 선언 하게되면 얼마의 메모리가 필요할까여?

위의 구조체의 크기는 int 형 데이터 6 개(4*6 = 24) + char 형 배열 220 개. 따라서 총 244 바이트가 됩니다. 만개의 배열을 잡으면 244 만 바이트가 필요하져. 2 메가바이트가 넘네여.. 배열로 하지 않고 데이터가 필요한 만큼 메모리 동적 할당을 한다고 하더라도 위와 같이 만개의 데이터가 필요하다고 하면 2 메가가 넘는 메모리가 한번에 할당이 되어야 하는 것입니다.

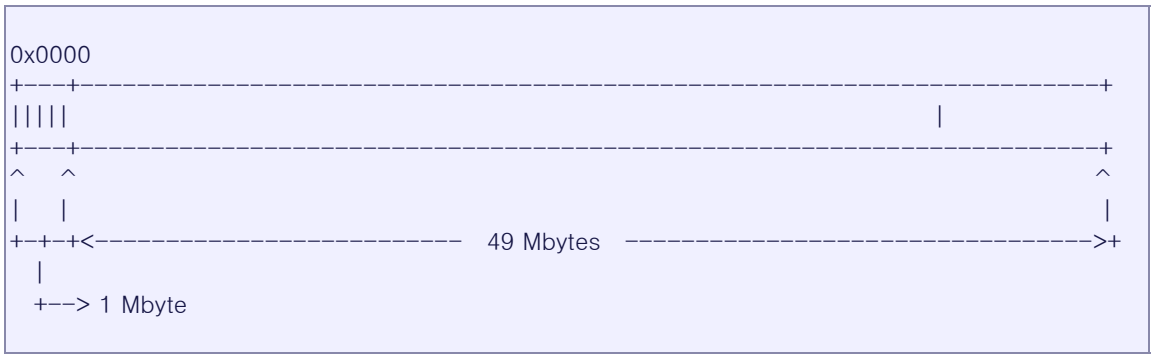
현재 사용 가능한 메모리가 50 메가 라고 해서 50 메가를 한번에 동적 메모리 할당을 할 수 있는 것은 아닙니다. 그 이유는 메모리의 중간 중간을 다른 프로그램이나 프로세스에서 할당 하고 해제하고 하는 일을 해서 메모리가 조각이 나있기 때문입니다.

그림으로 살펴보져.

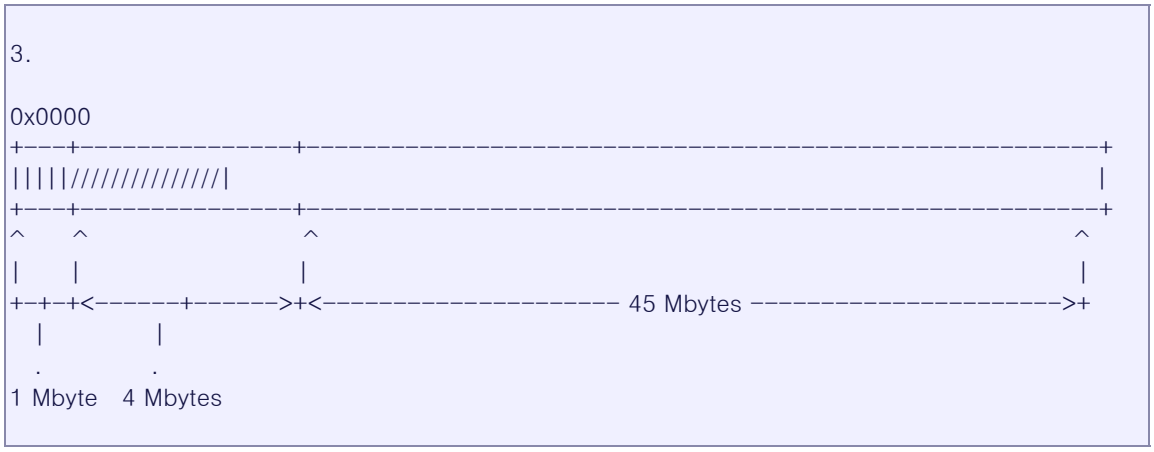
```
1.

0x0000(이라고 가정)
+-------------------------------------------------------------------------------+
|                                                                               |
+-------------------------------------------------------------------------------+
^                                                                               ^
|                                                                               |
+------------------------------  50 Mbytes  ------------------------------------+
```

처음에 위와 같이 50 메가 바이트의 메모리가 있다고 가정하고 그 메모리는 아직 다 른 프로세스에서 사용되어 지지 않은 얼마든지 할당이 가능한 상태입니다.

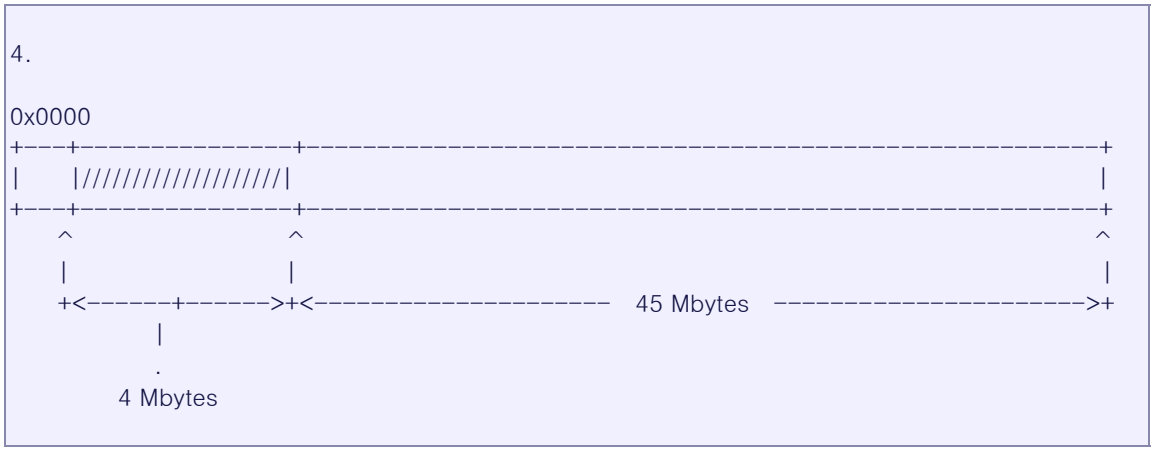
```
2.
```



1 메가 바이트만큼 어떤 프로세스에서 사용을 하고 있습니다. 즉 그 프로세스에서 1 메가 바이트 만큼 메모리를 할당을 한 것입니다.



다시 4 메가 바이트만큼 다른 프로세스에서 사용하기 위해 메모리를 할당했습니다.



처음에 할당 했던 1 메가 바이트를 해제해서 사용 가능한 메모리로 돌려 놓았습니다.
그래서 남은 할당 가능한 메모리는 46 메가 바이트입니다. 하지만 여기에서 문제는 46 메가 바이트가 연속적으로 존재하지 않고 따로 떨어져 있다는 것입니다. 그렇기때문에 할당 가능한 가장 큰 크기는 45 메가 바이트가 됩니다. 한번에 할당을 받을 수 있으려면 메모리가 연속적으로 연결되어 있어야 하기 때문입니다.

위의 그림으로 남아 있는 메모리의 크기와 한번에 할당을 할 수 있는 메모리의 크기는 다른 것이라는 것을 알았습니다. 위의 그림은 단순화 해서 그린 그림이지만 실제로 많은 프로세스가 메모리를 할당하고 해제하는 일을 함으로써 메모리의 중간 중간에 구멍이 생기게 됩니다. 사용 가능한 메모리는 많은데 한번에 할당 할 수 있는 메모리의 크기는 그렇게 크지 않게 되는 그러한 현상을 메모리 단편화라고 합니다.

이러한 메모리 단편화는 사실 메모리가 할당 되고 해제되면서 운영체제가 단편화된 메모리의 조각들을 모아주는 일을 해줍니다. 메모리의 조각을 모아준다는 것은 비어있는 메모리 부분으로 현재 사용하고 있는 메모리를 옮겨주는 것을 말합니다. 디스크 조각 모음과 비슷한 개념으로 생각하면 됩니다. 그렇다 하더라도 사용 가능한 메모리의 크기와 한번에 할당할 수 있는 최대 메모리의 크기는 차이가 나게 됩니다.

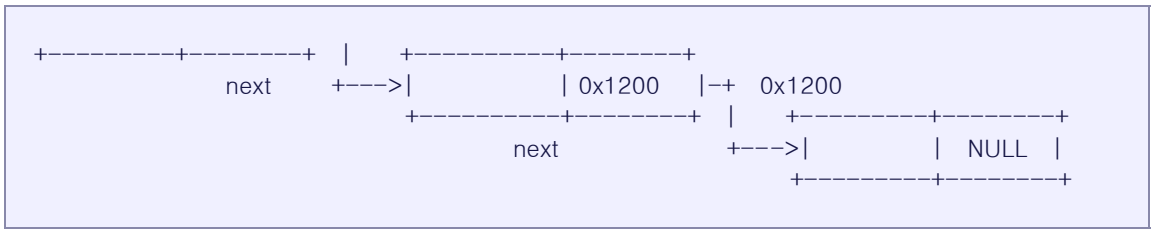
따라서 많은 데이터를 사용할 때, 한번에 큰 크기의 메모리를 할당하는 것이 아니라 작은 데이터 단위의 크기로 메모리를 할당하고 그 할당된 많은 데이터들을 연결시켜서 사용을 하게됩니다. 바로 여기에서 링크드 리스트의 개념이 나오게 되는 것입니다.

저 위의 구조체의 크기가 244 바이트라고 했었지여? 244 바이트 만큼의 메모리 할당을 만개를 하고 그 만개의 데이터를 서로 연결시켜주면 실제로 2 메가 바이트를 한번에 할당하지는 않았지만 그와 같은 효과를 갖게 되는 것입니다. 물론 작은 단위로 많은 할당을 했으므로 메모리는 더 작은 크기로 단편화가 됩니다. 그렇지만 위에서 언급했듯이 운영체제에서 알아서 필요할 때 메모리 조각 모음을 해주므로 일단은 크게 염두하지 않고 많은 메모리를 사용할 수가 있게 됩니다.

작은 데이터 단위의 메모리 할당은 그렇다 치는데 그럼 어떻게 그것들을 연결시킬까여? 위의 구조체를 아주 조금 변경시켜 보겠습니다.

```
struct _data
{
    int       number;
    int       kind;
    char      name[20];
    int       kor;
    int       eng;
    int       math;
    int       science;

    char      buffer[200];

    struct _data *next;
};
```

구조체의 멤버 중에서 next 라는 자기 자신 참조 포인터가 선언되어 있습니다. 바로이 next 변수가 하는 일이 다른 작은 데이터 단위의 주소를 저장함으로써 연결을 하기 위한 변수입니다. 그림을 그려보겠습니다.



위의 그림은 작은 데이터 단위가 3 개 할당 된 모습을 그린 그림입니다. 그림에서 작은 데이터 단위가 2 개로 나누어져 있는 것을 볼 수 있는데 앞의 것은 구조체에서 앞에 선언되어 있는 일반 변수들을 나타내며, 뒤의 것은 next 포인터를 나타내는 것입니다. 각각 0x1000, 0x1100, 0x1200 이라는 주소값에 메모리가 할당이 되어 있다고 가정을 하면, 작은 데이터 블럭들은 연속되어 존재할 수도 있고 메모리의 상태에 따라서 연속되지 않게 존재할 수도 있습니다. 하지만 next 라는 포인터가 다음 데이터 블럭의 주소값을 저장해서 다음에 존재하는 데이터 블럭과 연결을 하고 있게 됩니다.

위의 그림이 나오도록 코드를 작성해 볼까여?

```
 1  #include <stdio.h>
 2  #include <stdlib.h> /* malloc(), free() */
 3
 4  typedef struct _data Data;
 5
 6  struct _data
 7  {
 8      int     number;
 9      int     kind;
10      char    name[20];
11      int     kor;
12      int     eng;
13      int     math;
14      int     science;
15
16      char    buffer[200];
17
18      struct _data *next;  /* 다음 데이터의 주소값을 저장할 포인터 */
19  };
20
21  int main(void)
22  {
23      Data *data1 = NULL;
24      Data *data2 = NULL;
25      Data *data3 = NULL;
26      Data *temp = NULL;  /* for() 문을 돌리기 위한 임시 포인터 변수 */
27
28      /* 데이터 블럭 3 개를 할당함 */
29      data1 = (Data *)malloc( sizeof(Data) );
30      data2 = (Data *)malloc( sizeof(Data) );
31      data3 = (Data *)malloc( sizeof(Data) );
32
33      /* 각각의 데이터 블럭을 연결 시킴 */
```

```
34        data1->next = data2;
35        data2->next = data3;
36        data3->next = NULL;
37
38        /* data1 변수 하나 가지고 연결된 모든 데이터 블럭을 접근 */
39        for( temp = data1 ; temp != NULL ; temp = temp->next )
40        {
41             printf("data block..₩n");
42        }
43
44        /* 할당받은 메모리를 해제시킴 */
45        free( data1 );
46        free( data2 );
47        free( data3 );
48
49        /* makes compiler happy :) */
50        return 0;
51   }
```

23 라인에서 할당 받을 데이터 메모리 블럭의 주소값을 저장할 변수 3 개를 선언하고

29 라인에서 각각의 포인터 변수에 메모리를 할당하여 주소값을 저장합니다.

34 라인에서 data1 의 데이터 블록 내에 있는 next 포인터 변수에 data2 의 주소값을 저장합니다. 마찬가지로 data2 의 next 포인터 변수에 data3 의 주소값을 저장하고 마지막으로 data3 의 next 포인터 변수에는 더이상 연결시킬 데이터가 없다는 뜻으로 NULL 을 저장합니다.

39 라인의 for()문에서는 temp 포인터 변수에 data1 의 주소값을 대입하고 루프를 돌리는데 for()문의 마지막 증감식에서 temp = temp->next 라는 코드로써 temp 변수가 다음 데이터 블럭의 주소값을 갖고 있도록 하는 것입니다. 결과적으로 이 for()문은 데이터 블럭의 개수만큼 3 번 루프가 돌게 됩니다. 이때 for()문을 실행하기 위해서 사용된 데이터 블럭은 data1 뿐입니다. data1 의 주소값 하나만 알고 있으면 data1->next 에 의해서 data2 를 알 수가 있고, data2->next 에 의해서 data3 를 알 수가 있게됩니다.

즉, 뒤 이어서 아무리 많은 작은 데이터 블럭이 할당 되어 연결이 된다 하더라도 제일 처음 데이터 블럭의 주소값만 알고 있으면 모든 데이터 블럭을 접근 할 수가 있게 됩니다. 이것을 링크드 리스트(Linked list)라고 합니다.

이런 링크드 리스트의 목적은 처음에 설명 했듯이 큰 데이터 블럭을 할당 하려고 하는 것보다 작은 데이터 블럭을 서로 연결시켜 놓음으로써 큰 데이터 블럭을 할당한 것과 같은 효과를 내기 위함입니다. 그리고 이 링크드 리스트는 부가적인 이점도 있습니다. 작은 블럭 단위로 서로 연결을 해 놓았기 때문에 중간에 있는 데이터를 삭제 하거나 삽입을 하기가 아주 편리합니다. 서로 연결된 고리 정보만 바꾸어 주면 되기 때문입니다. 만약 한번에 메모리가 할당되었을 때 어느 하나의 데이터를 삭제하거나 삽입을 할 경우가 발생하면 곤란한 일이 발생할 것입니다.

위에 제시한 예제는 그림을 설명하기 위한 예제일 뿐이고, 실제로 위와 같이 코딩을 하지는 않습니다. 보통 데이터 블럭을 생성하는 함수, 제거하는 함수로 구분을 해서 작성을 하게 됩니다. 부가적으로 삽입이나 정렬을 위한 함수를 작성해서 사용하기도 합니다. 다음 예제에서는 보다 실전적인 링크드 리스트 함수들을 구현해 보겠습니다.

```
 1  #include <stdio.h>
 2  #include <string.h> /* strlen(), strcpy() */
 3  #include <stdlib.h> /* malloc(), free() */
 4
 5  /* struct _object 를 Object 로 typedef */
 6  typedef struct _object   Object;
 7
 8  /* 구조체 선언 */
 9  struct _object
10  {
11      unsigned long    ID;
12      char *           name;
13      int              age;
14
15      Object *         next;
16  };
17
18  /* 전역 변수 선언 */
19  unsigned long _objectID;
20  Object *_objectHead = NULL; /* 링크드 리스트의 가장 처음 */
21  Object *_objectTail = NULL; /* 링크드 리스트의 가장 마지막 */
22
23  /* 함수 원형 선언 */
24  Object * createObject(void);
25  Object * createObjectWithData( char *name, int age );
26
27  int         deleteObject( Object *object );
28  int         deleteObjectWithName( char *name );
29  int         deleteObjectWithAge( int age );
30  void        deleteAllObject(void);
31
32  void        printObject( Object *object );
33  void        printAllObject(void);
34
35  /* createObject()
36   *
37   * Object 를 하나 메모리에 할당 받고 링크드 리스트에 연결을 시킨다.
38   */
39  Object *createObject(void)
40  {
41      Object *object = NULL;
42
43      /* 메모리 할당 */

44      object = (Object *)malloc( sizeof(Object) );

45      if( object == NULL )
46      {
47          fprintf( stderr,
48                  "createObject(): memory allocational error!₩n" );
49          return NULL;
50      }
51      memset( object, 0, sizeof(Object) );
52
```

```
53      /* Object 의 ID 를 저장 */
54      object->ID = _objectID++;
55
56      if( _objectHead == NULL ) /* 링크드 리스트가 비어있으면 */
57      {
58          _objectHead   = object; /* 링크드 리스트의 처음이 object */
59          _objectTail   = object; /* 링크드 리스트의 마지막이 object */
60          object->next = NULL;    /* 방금 생성된 obejct 의 다음은 NULL */
61      }
62      else /* 링크드 리스트가 비어있지 않으면 */
63      {
64          _objectTail->next = object; /* 현재 마지막 리스트의 다음이
65                                         object */
66          _objectTail         = object; /* 링크드 리스트의 마지막이
67                                         object */
68          object->next        = NULL;    /* object 의 다음은 NULL */
69      }
70
71      return object;
72  }
73
74  /* createObjectWithData()
75   *
76   * 내부적으로 createObject()를 호출하고 인자로 넘어온 데이터를
77   * 저장한다.
78   */
79  Object *createObjectWithData( char *name, int age )
80  {
81      Object *object = createObject();
82
83      object->name = (char *)malloc( strlen(name) + 1 );
84      strcpy( object->name, name );
85      object->age = age;
86
87      return object;
88  }
89
90  /* deleteObject()
91   *
92   * 인자로 들어온 object 를 리스트에서 삭제한다.
93   */
94  int deleteObject( Object *object )
95  {
96      Object *temp = _objectHead;
97      Object *prev = NULL; /* 이전 데이터를 저장할 포인터 */
98
99      if( object == NULL ) return 0;
100
101     /* 리스트에서 인자로 들어온 object 와 일치하는 것을 찾을 때
102      * 까지 루프를 돌린다.
103      */
104     while( temp )
```

```
105         {
106             if( object == temp ) break;
107             prev = temp;
108             temp = temp->next;
109         }
110         /* 리스트에서 찾는 것이 없을때 0 을 리턴 */
111         if( temp == NULL ) return 0;
112
113         /* 찾은 데이터가 _objectHead 일때 - 가장 처음에 존재하는 데이터
114          * 일때
115          */
116         if( temp == _objectHead )
117             _objectHead = temp->next;
118
119         /* 찾은 데이터가 가장 나중에 존재하는 데이터 일때 */
120         else if( temp == _objectTail )
121         {
122             _objectTail = prev;
123             _objectTail->next = NULL;
124         }
125         else prev->next = temp->next;
126
127         /* 데이터의 name 속성에 메모리가 할당되어 있으면 해제한다. */
128         if( temp->name != NULL ) free( temp->name );
129
130         /* 데이터 메모리 블럭을 해제한다. */
131         free( temp );
132
133         /* 오류없이 종료했음을 나타내는 1 을 리턴한다. */
134         return 1;
135     }
136
137     /* deleteObjectWithName()
138      *
139      * 리스트에서 인자로 넘어온 name 이 일치하는 데이터를 삭제한다.
140      */
141     int deleteObjectWithName( char *name )
142     {
143         Object *temp = _objectHead;
144
145         while( temp )
146         {
147             if( strcmp( temp->name, name ) == 0 ) break;
148             temp = temp->next;
149         }
150
151         return deleteObject( temp );
152     }
153
154     /* deleteObjectWithAge()
155      *
156      * 리스트에서 인자로 넘어온 age 가 일치하는 데이터를 삭제한다.
```

```
157   */
158  int deleteObjectWithAge( int age )
159  {
160       Object *temp = _objectHead;
161
162       while( temp )
163       {
164            if( temp->age == age ) break;
165            temp = temp->next;
166       }
167
168       return deleteObject( temp );
169  }
170
171  /* deleteAllObject()
172   *
173   * 리스트에 있는 모든 데이터를 삭제한다.
174   */
175  void deleteAllObject(void)
176  {
177       Object *temp = _objectHead;
178       Object *next = NULL; /* 다음 데이터를 저장하는 포인터 */
179
180       while( temp )
181       {
182            /* 블럭을 해제하고 나면 temp->next; 를 사용할 수 없으므로
183             * 블럭을 해제하기 전에 다음 블럭의 주소를 저장해 놓은 다음
184             * 에 블럭을 해제한다.
185             */
186            next = temp->next;
187            if( temp->name != NULL ) free( temp->name );
188            free( temp );
189
190            temp = next;
191       }
192
193       /* 리스트의 처음, 마지막을 나타내는 전역 변수를 초기화 한다. */
194       _objectHead = _objectTail = NULL;
195  }
196
197  /* printObject()
198   *
199   * 인자로 넘어온 데이터를 출력한다.
200   */
201  void printObject( Object *object )
202  {
203       printf("%2d: %s, %d₩n", object->ID, object->name, object->age );
204  }
205
206  /* printAllObject()
207   *
208   * 리스트의 모든 데이터를 출력한다.
```

```
209    */
210   void printAllObject(void)
211   {
212         Object *temp = _objectHead;
213
214         while( temp )
215         {
216               printObject( temp );
217               temp = temp->next;
218         }
219   }
220
221   int main(void)
222   {
223         /* 4 개의 데이터를 생성한다.
224          * 여기서는 createObjectWithData() 함수가 리턴하는 주소값은
225          * 무시하고 있다.
226          * 하지만 생성된 데이터들이 연결되어 있는 리스트의 첫번째 값은
227          * _objectHead 전역 변수에 의해서 알 수가 있다.
228          */
229         createObjectWithData( "Shim sang don", 28 );
230         createObjectWithData( "David", 25 );
231         createObjectWithData( "Robert", 34 );
232         createObjectWithData( "Josephin", 26 );
233
234         printAllObject();
235
236         printf("--------------------------₩n");
237
238         /* "david"라는 이름을 갖는 데이터와 나이가 26 인 데이터 두개를
239          * 삭제한다.
240          */
241         deleteObjectWithName( "David" );
242         deleteObjectWithAge( 26 );
243
244         printAllObject();
245
246         /* 리스트의 모든 데이터를 삭제한다. */
247         deleteAllObject();
248
249         return 0;
250   }
```

39 라인의 createObject() 함수에 의해서 데이터 블럭이 할당되며, 링크드 리스트에 삽입이 됩니다. 이 함수는 createObjectWithData() 함수에 의해서 내부적으로 호출되며 데이터 블럭을 생성하고, 리스트의 마지막에 삽입을 하는 일을 하게 됩니다. 여기에서 데이터가 생성되고 리스트에 삽입될 때, 위의 예제 처럼 리스트의 마지막에 삽입을 하는 경우가 있고 정렬을 위해서 어떤 조건에 의해 리스트를 검색하면서 삽입을 하는 경우가 있습니다.
보통 데이터가 많은 경우에는 리스트에 있는 데이터들을 정렬하는 함수를 호출해야하는 부가적인 일을 하지 않도록 삽입할 때 미리 정렬된 위치에 삽입하는 경우가 많지만 데이터가 많지 않거나 데이터를 삽입하고 삭제하는 일을 아주 많이 해야하는 일을 할 경우에는 일단 마지막에 삽입을 하도록 해 놓은 다음에 리스트를

정렬하는 함수를 호출 해서 한번에 정렬을 합니다.

94 라인의 deleteObject() 함수를 보면 삭제해야할 데이터를 찾은 다음에 연결 정보를 갱신하는 코드를 볼 수가 있습니다(113 ~ 125 라인). 이렇게 해줌으로써 리스트에 연결되어 있는 데이터들의 연결 고리가 끊어지지 않고 계속해서 유지될 수가 있는 것입니다.

위의 예제에서 조금은 비효율적인 부분도 있습니다. 예를 들어서 141 라인에 선언되어 있는 deleteObjectWithName() 함수를 보면, 내부적으로 인자로 들어온 name 과 일치하는 데이터를 찾기 위해서 while 루프를 돌아서 데이터를 찾아내는데 그 찾은 데이터를 삭제할 때 deleteObject() 함수를 호출하게 됩니다. 그런데 deleteObject() 함수의 내부를 보면 리스트에서 그 데이터 주소값과 일치하는 것을 찾기 위해서 또 while 루프를 돌고 있는 것을 볼 수가 있습니다.

현재의 코드를 잘 분석해보고 비효율적인 부분을 찾아내고 효율적으로 개선을 해 보는 공부를 스스로 해보시기 바랍니다.

이번 장은 단일 링크드 리스트에 대해서 알아보았습니다. 다음 장에서 이중 링크드 리스트에 대해서 공부를 해보도록 하겠습니다.

7.4. 이중 링크드 리스트

저번 강좌에서 링크드 리스트에 대해서 알아보았습니다. 하지만 저번 강좌에서 설명한 링크드 리스트는 단일 링크드 리스트라고 불리우는 next 포인터만 존재하는 리스트였고, 이번 강좌에서 이중 링크드 리스트에 대해서 공부를 해보도록 하겠습니다.

우선 저번 강좌의 단일 링크드 리스트에 대해서 복습을 해 보면, 구조체의 멤버 변수 중에 자기 구조체 참조 포인터 변수가 있어서, 그 구조체로 생성된 다른 데이터 블럭의 주소값을 저장하고 있기 때문에 모든 데이터 블럭을 서로 연결시킬 수가 있었지여? 그 자기 구조체 참조 포인터 변수를 통상 next 라는 이름으로 선언합니다.

자, 그렇다면 그 next 포인터는 자기 자신 데이터 블럭의 뒤에 연결될 데이터 블럭의 주소값을 저장하기 위한 포인터였는데, 그렇다면 자기 앞의 데이터 블럭의 주소값을 저장하기 위한 포인터를 선언해서 사용해도 되지 않을까여?

흠... 마치 이중 링크드 리스트가 나오도록 유도심문 한것처럼 느껴지네여. :)

단일 링크드 리스트 강좌의 예제 소스중에서 분명히 비 효율적인 부분이 있습니다. 그것은 두번의 while()문을 통해서 원하는 것을 삭제한 부분도 있겠지만, 그것은 내부적으로 함수를 호출을 했기 때문에 그러한 결과가 나온것입니다. 즉, name 이라는 인자를 받아서 데이터 블럭을 삭제할 때 deleteObject() 함수를 호출하지 않고 바로 삭제하는 코드를 작성하면 while() 문이 두번 실행되지는 않겠지여.

그것 외에도 삭제하기 위한 데이터 블럭의 바로 이전 데이터 블럭을 알고 있어야 그 다음에 있는 데이터 블럭과 이전 데이터 블럭과 연결을 시켜줄 수 있었는데, 그러기 위해서 임시 포인터 변수인 prev 를 선언했던 것 기억나시져?

그리고 단일 링크드 리스트를 사용하면 위에서 아래로의 탐색은 가능하지만, 아래에서 위로의 탐색은 불가능 합니다. (불가능 하지는 않지만 엄청 비 효율적이 됩니다.) 그 이유는 next 만 갖고 있기 때문입니다. 그래서 prev 라는 자기 구조체 참조 포인터 변수를 한개 더 두는 겁니다. 그렇게 하고 그 prev 변수에는 이전 데이터 블럭의 주소를 저장하도록 하게 되면 각각의 데이터 블럭들은 자기 앞의 데이터 블럭과 뒤의 데이터 블럭의 위치를 알고 있게 됩니다. 마치 우리가 한줄로 설 때 앞사람과 뒤사람을 알고 있으면 흐트러져 있더라도 다시 그 상태로 모일 수 있는 것과 같지여.

저번 강좌의 단일 링크드 리스트 예제를 조금 수정해서 이중 링크드 리스트 예제로 바꾸어 보겠습니다.

```
 1  #include <stdio.h>
 2  #include <string.h> /* strlen(), strcpy() */
 3  #include <stdlib.h> /* malloc(), free() */
 4
 5  /* struct _object 를 Object 로 typedef */
 6  typedef struct _object   Object;
 7
 8  /* 구조체 선언 */
 9  struct _object
10  {
11        unsigned long     ID;
12        char *              name;
13        int                 age;
14
15        Object *           prev;    /* 이전 데이터 블럭을 저장하기 위함 */
16        Object *           next;
17  };
18
19  /* 전역 변수 선언 */
20  unsigned long _objectID;
21  Object *_objectHead = NULL; /* 링크드 리스트의 가장 처음 */
22  Object *_objectTail = NULL; /* 링크드 리스트의 가장 마지막 */
23
24  /* 함수 원형 선언 */
25  Object * createObject(void);
26  Object * createObjectWithData( char *name, int age );
27
28  int         deleteObject( Object *object );
29  int         deleteObjectWithName( char *name );
30  int         deleteObjectWithAge( int age );
31  void        deleteAllObject(void);
```

```
32
```

```
33  void        printObject( Object *object );
34  void        printAllObject(void);
35
36  /* createObject()
37   *
38   * Object 를 하나 메모리에 할당 받고 링크드 리스트에 연결을 시킨다.
39   */
40  Object *createObject(void)
```

```
41  {
42      Object *object = NULL;
43
44      /* 메모리 할당 */
45      object = (Object *)malloc( sizeof(Object) );
46      if( object == NULL )
47      {
48          fprintf( stderr,
49                  "createObject(): memory allocational error!₩n" );
50          return NULL;
51      }
52      memset( object, 0, sizeof(Object) );
53
54      /* Object 의 ID 를 저장 */
55      object->ID = _objectID++;
56
57      if( _objectHead == NULL ) /* 링크드 리스트가 비어있으면 */
58      {
59          _objectHead   = object; /* 링크드 리스트의 처음이 object */
60          _objectTail   = object; /* 링크드 리스트의 마지막이 object */
61          object->prev = NULL;    /* 방금 생성된 object 의 이전은 NULL */
62          object->next = NULL;    /* 방금 생성된 obejct 의 다음은 NULL */
63      }
64      else /* 링크드 리스트가 비어있지 않으면 */
65      {
66          _objectTail->next = object; /* 현재 마지막 리스트의 다음이
67                                         object */
68          object->prev = _objectTail; /* object 의 이전은 현재의 마지막 데이터 */
69          _objectTail         = object; /* 링크드 리스트의 마지막이
70                                         object */
71          object->next        = NULL;    /* object 의 다음은 NULL */
72      }
73
74      return object;
75  }
76
77  /* createObjectWithData()
78   *
79   * 내부적으로 createObject()를 호출하고 인자로 넘어온 데이터를
80   * 저장한다.
81   */
82  Object *createObjectWithData( char *name, int age )
83  {
84      Object *object = createObject();
85
86      object->name = (char *)malloc( strlen(name) + 1 );
87      strcpy( object->name, name );
88      object->age = age;
89
90      return object;
91  }
92
```

```
 93  /* deleteObject()
 94   *
 95   * 인자로 들어온 object 를 리스트에서 삭제한다.
 96   */
 97  int deleteObject( Object *object )
 98  {
 99      if( object == NULL ) return 0;
100
101      /* 삭제할 데이터가 _objectHead 일때 - 가장 처음에 존재하는 데이터
102       * 일때
103       */
104      if( object == _objectHead )
105      {
106          object->next->prev = NULL;
107          _objectHead = object->next;
108      }
109      /* 삭제할 데이터가 가장 나중에 존재하는 데이터 일때 */
110      else if( object == _objectTail )
111      {
112          object->prev->next = NULL;
113          _objectTail = object->prev;
114      }
115      else
116      {
117          object->prev->next = object->next;
118          object->next->prev = object->prev;
119      }
120
121      /* 데이터의 name 속성에 메모리가 할당되어 있으면 해제한다. */
122      if( object->name != NULL ) free( object->name );
123
124      /* 데이터 메모리 블럭을 해제한다. */
125      free( object );
126
127      /* 오류없이 종료했음을 나타내는 1 을 리턴한다. */
128      return 1;
129  }
130
131  /* deleteObjectWithName()
132   *
133   * 리스트에서 인자로 넘어온 name 이 일치하는 데이터를 삭제한다.
134   */
135  int deleteObjectWithName( char *name )
```

```
136  {
```

```
137      Object *temp = _objectHead;
138
139      while( temp )
140      {
141          if( strcmp( temp->name, name ) == 0 ) break;
142          temp = temp->next;
143      }
144
```

```
145         return deleteObject( temp );
146     }
147
148     /* deleteObjectWithAge()
149      *
150      * 리스트에서 인자로 넘어온 age 가 일치하는 데이터를 삭제한다.
151      */
152     int deleteObjectWithAge( int age )
153     {
154         Object *temp = _objectHead;
155
156         while( temp )
157         {
158             if( temp->age == age ) break;
159             temp = temp->next;
160         }
161
162         return deleteObject( temp );
163     }
164
165     /* deleteAllObject()
166      *
167      * 리스트에 있는 모든 데이터를 삭제한다.
168      */
169     void deleteAllObject(void)
170     {
171         Object *temp = _objectHead;
172         Object *next = NULL; /* 다음 데이터를 저장하는 포인터 */
173
174         while( temp )
175         {
176             /* 블럭을 해제하고 나면 temp->next; 를 사용할 수 없으므로
177              * 블럭을 해제하기 전에 다음 블럭의 주소를 저장해 놓은 다음
178              * 에 블럭을 해제한다.
179              */
180             next = temp->next;
181             if( temp->name != NULL ) free( temp->name );
182             free( temp );
183
184             temp = next;
185         }
186
187         /* 리스트의 처음, 마지막을 나타내는 전역 변수를 초기화 한다. */
188         _objectHead = _objectTail = NULL;
189     }
190
191     /* printObject()
192      *
193      * 인자로 넘어온 데이터를 출력한다.
194      */
195     void printObject( Object *object )
196     {
```

```
197         printf("%2d: %s, %d₩n", object->ID, object->name, object->age );
198     }
199
200     /* printAllObject()
201      *
202      * 리스트의 모든 데이터를 출력한다.
203      */
204     void printAllObject(void)
205     {
206         Object *temp = _objectHead;
207
208         while( temp )
209         {
210             printObject( temp );
211             temp = temp->next;
212         }
213     }
214
215     int main(void)
216     {
217         /* 4 개의 데이터를 생성한다.
218          * 여기서는 createObjectWithData() 함수가 리턴하는 주소값은
219          * 무시하고 있다.
220          * 하지만 생성된 데이터들이 연결되어 있는 리스트의 첫번째 값은
221          * _objectHead 전역 변수에 의해서 알 수가 있다.
222          */
223         createObjectWithData( "Shim sang don", 28 );
224         createObjectWithData( "David", 25 );
225         createObjectWithData( "Robert", 34 );
226         createObjectWithData( "Josephin", 26 );
227
228         printAllObject();
229
230         printf("--------------------------₩n");
231
232         /* "david"라는 이름을 갖는 데이터와 나이가 26 인 데이터 두개를
233          * 삭제한다.
234          */
235         deleteObjectWithName( "David" );
236         deleteObjectWithAge( 26 );
237
238         printAllObject();
239
240         /* 리스트의 모든 데이터를 삭제한다. */
241         deleteAllObject();
242
243         return 0;
244     }
```

저번 강좌의 예제와 비교를 해보면서 보는것도 좋을것 같네여.

단일 링크드 리스트와 코드가 바뀐곳에 대해서 설명을 하겠습니다.

15 라인에 prev 포인터가 추가되었습니다. 이 prev 포인터가 하나 늘어남으로써 구조체의 크기는 4 바이트 증가되었지만, 4 바이트 늘어난만큼 편리함을 제공해 줄 것입니다.

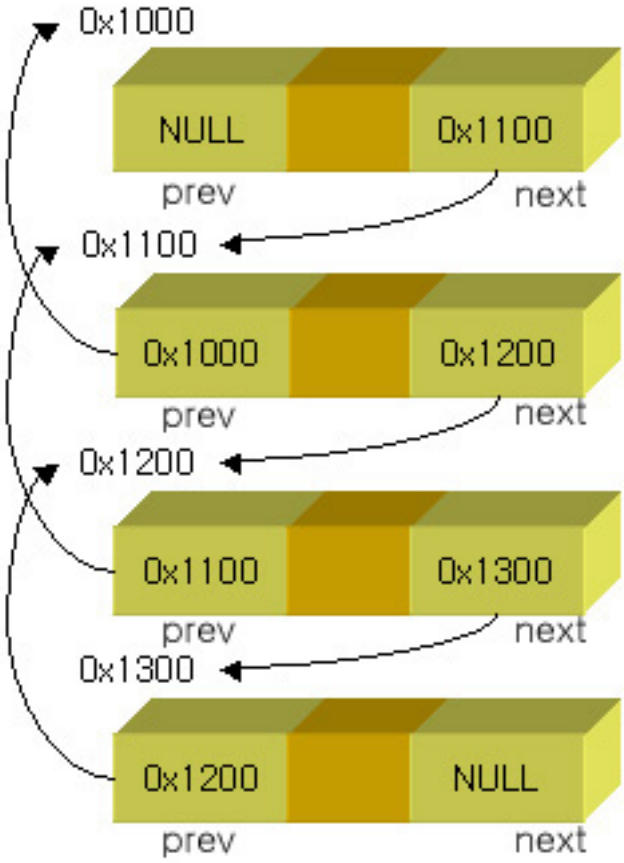
61 라인에 생성된 데이터 블럭의 이전 블럭에 대해서 저장을 합니다. 여기에서는 링크드 리스트가 비어있는 상태이므로 가장 처음에 생성된 데이터 블럭이 되는 것입니다. 따라서 방금 생성된 데이터 블럭의 앞에는 아무것도 없으므로 prev 는 NULL 이 되는 것입니다.

68 라인에서 생성된 데이터 블럭의 prev 는 링크드 리스트의 가장 마지막 데이터라는 것을 나타내는 것입니다.

97 라인의 deleteObject() 함수가 가장 크게 개선된 함수입니다. 단일 링크드 리스트 예제에서의 이 함수는 삭제하고자 하는 데이터 블럭 앞에 있는 것이 무엇인지 찾기 위해서 while() 루프를 사용했었는데, 이중 링크드 리스트에서는 while 루프를 실행하지 않아도 prev 포인터에 의해서 바로 알 수가 있으므로 prev 를 찾기 위해서 불필요한 while 루프를 돌지 않아도 되므로, 수행 속도가 빨라지게 됩니다.
삭제하기 전에 삭제될 데이터 블럭의 앞, 뒤 데이터들 끼리 연결 고리를 이어주고 자신은 삭제되는 것입니다.

그 외에 바뀐것은 없습니다. prev 변수가 하나 늘어나고 코드가 조금 바뀐것 말고는 눈에 띄게 바뀐것을 체험하기가 어려울 수도 있지만, 다음 강좌에 나올 탐색과 정렬을 구현해 보면서 prev 가 있어서 더 많은 일을 해 줄 수가 있다는 것을 알 수 있을 것입니다.

이중 링크드 리스트에 대한 그림이 빠지면 구색이 맞지 않으므로 그림을 그려보도록 하겠습니다.



위에서 아래로 데이터가 생성 이 되었다고 할 때, 첫번째 데이터의 prev 에는 NULL 이 들어 갑니다. 그 이유는 앞에 존재하는 데이터가 없기 때문이져. 그리고 next 에는 다음 데이터 블럭의 주소값인 0x1100 이 저장 됩니다.

두번째 데이터의 prev 는 앞에 있는 0x1000 번지에 있는 데이터 블럭을 가리키고 있게 되고 next 는 다음 데이터 블럭의 주소값인 0x1200 을 저장하고 있습니다.

이런 식으로 모든 데이터 블럭들은 앞, 뒤가 연결되어 있게됩니다.

텍스트로 그림을 그리려니 조잡해 지더군여. 그래서 그냥 그림파일로 올렸습니다.
개인적으로 이런 그림 보다는 아스키로 그린 그림을 더 좋아하는 편이라서 아스키로 그리려고 시간을 조금 보냈는데 알아보기가 쉽지 않게 되더군여. :(

7.5. 링크드 리스트에서의 검색

저번 장까지 링크드 리스트에 대한 구조체 선언과 next, prev 포인터가 어떻게 사용이 되는지 알아보았습니다. 이번 장에서는 링크드 리스트의 사용에 대해서 알아보도록 하겠습니다. 앞에서 잠깐 언급된 탐색에 대해서 잠깐 언급하고 넘어가겠습니다.

탐색이란 리스트 내에 있는 데이터 중에서 원하는 데이터를 찾아내는 것을 말합니다.
검색이라는 단어를 사용하는 편이 더 옳겠네여. 탐색이라는 것은 일치하는 데이터를 찾는 것을 나타내는 것 외에 가장 빠르게 도달할 수 있는 경로를 찾는 최단 거리 탐색 이라든가 또는 트리 구조등의 깊이를 알아내는 것을 지칭하는 단어로도 사용이되므로 여기서는 탐색이라는 단어보다는 조금 의미가 좁은 검색이라는 단어를 사용하겠습니다.

검색을 한다는 것은 리스트에 있는 데이터 블럭에서 원하는 검색조건과 일치하는 데이터 블럭을 찾아내는 일을 하는 것을 말합니다. 보통 가장 쉬운 검색 방법은 리스트의 처음부터 끝까지 차례로 모든 데이터 블럭을 통과하면서 검색 조건과 일치하는 데이터 블럭을 찾아내는 것입니다. 쉬운 방법이지만 가장 많이 사용되는 방법입니다. 효율을 증대시키기 위해서 리스트 자체를 변형하는 방법을 사용하기 때문에, 일단 리스트를 변경하는 것에 대해서는 나중에 언급을 하기로 하고 지금은 단순히 리스트의 처음부터 끝까지 데이터 블럭을 조사하는 방법에 대해서 알아보겠습니다.

앞장의 예제 소스와 연계해서, 리스트 내에 이름을 조건으로 해서 일치하는 데이터를 찾아서 그 데이터 블럭의 주소값을 리턴하는 함수를 작성해 보겠습니다.

```
 1  /* findObjectWithName()
 2   *
 3   * 리스트의 처음 데이터 블럭에서 마지막 데이터 블럭까지의
 4   * 데이터 블럭(여기에서는 object 로 대표됨)의 name 멤버와
 5   * name 인자로 들어온 문자열과 비교를 해서 일치하는 경우에
 6   * 그 일치하는 데이터 블럭의 주소값을 리턴하는 함수
 7   */
 8  Object *findObjectWithName( char *name )
 9  {
10      /* Object 형 포인터를 선언해서 리스트의 가장 처음에 존재하는
11       * 데이터 블럭의 주소값을 대입한다.
12       * 여기서 _objectHead 가 가장 처음에 존재하는 데이터 블럭의
13       * 주소값을 저장하고 있는 전역변수.
14       * 10 번째 강좌의 소스코드를 참조하면 됩니다.
15       */
16      Object *object = _objectHead;
17
18      /* 인자로 들어온 name 에 아무 문자도 없으면 NULL 을 리턴. */
19      if( name == NULL ) return NULL;
20
21      /* object = object->next; 라는 코드로 인해서 while 문으로
22       * 모든 리스트들을 경유할 수가 있습니다.
23       */
24      while( object )
25      {
26          if( strcmp( object->name, name ) == 0 )
27              return object;
28          object = object->next;
29      }
30
31      /* while 문 내에서 return 이 일어나지 않고 여기까지 왔다는 것은
32       * 일치하는 데이터 블럭을 찾지 못했다는 뜻이므로 NULL 를 리턴
33       */
```

```
34        return NULL;
35   }
```

16 라인에서 리스트의 가장 처음에 있는 데이터의 주소를 object 라는 포인터를 선언하고 저장합니다.

24 라인의 while()문에서 리스트의 처음부터 끝까지 name 인자와 일치하는 데이터가 있는지 검색을 합니다. 일치하는 데이터가 있으면 일치하는 데이터의 주소값을 리턴하고 그렇지 않으면 계속 해서 다음 데이터와 비교를 합니다. 리스트의 끝까지 검색을 했는데 일치하는 데이터가 검색되지 않으면 NULL 을 리턴합니다.

검색을 하는 코드는 이렇게 간단히 처리될 수 있습니다. 그 코드가 효율적이든 그렇지 않든 리스트가 존재하는 구조를 이용하여 검색을 하는 방법에 대해서 알아보았습니다.

위의 예제 코드에서 name 을 가지고 찾는 함수를 작성했지만 독자분들이 스스로 age 를 검색하는 함수를 작성해보세여. 관련 코드는 앞장의 예제 코드를 보시면됩니다.

이번에는 리스트에 대한 정렬을 한번 해보기로 하겠습니다. 정렬 알고리즘은 그 수가 아주 많습니다. 오늘 설명할 정렬 알고리즘은 가장 이해하기 편하고 쉽게 응용할 수 있는 버블정렬에 대해서 설명을 하고 리스트에 적용을 해보기로 하겠습니다.

버블 정렬은 그 원리가 아주 간단합니다. 처음 데이터와 나머지 데이터를 비교해서 정렬조건이 작은 것을 앞으로 하는 경우에, 작은 것과 큰것의 위치를 뒤바꾸어 주는 것입니다. 역시 말로 설명을 하려니 이해시키기가 무척 어렵네여. 예를 들어볼께여.

```
2 3 1 6 5 4
```

위와 같이 데이터가 있다고 가정을 하겠습니다. 가장 처음에 정렬을 하기 위한 대상 데이터는 2 와 3 입니다. 2 와 3 을 비교해서 작은것은 앞으로 큰것은 뒤로 하도록 서로 위치를 뒤 바꾸는 것입니다. 현재는 2 가 작으므로 위치를 바꾸는 일은 하지 않겠지여? 그래서 데이터의 위치는 처음과 같습니다.

그 다음은 2 와 1 이 대상입니다. 1 이 작으므로 2 와 자리를 바꿉니다.

```
1 3 2 6 5 4
```

그 다음에 정렬을 하기 위한 대상은 2 와 6 이 아니라 1 과 6 입니다. 왜냐면 첫번째에 있는 데이터와 네번째에 있는 데이터를 정렬하는 것이기 때문입니다. 1 과 6 이 그 위치에 있는 데이터이므로 둘을 정렬합니다. 1 이 작으므로 위치는 변함없습니다.

이런식으로 6 번째 데이터까지 비교를 하면 결과는 1 3 2 6 5 4 이렇게 됩니다. 지금까지의 결과는 어떤 의미를 갖고 있을까여? 그것은 가장 작은 수가 가장 처음에 와 있다는 것을 뜻합니다. 1 이 가장 처음에 위치하게 된거져..

자 다음에는 두번째 위치의 데이터와 세번째, 네번째..여섯번째 위치의 데이터와 비교를 해야합니다.

3 과 2 를 비교해서 위치를 뒤바꾸겠져?

```
1 2 3 6 5 4
```

이렇게 됩니다. 그리고 다시 2 와 6, 2 와 5, 2 와 4 를 비교합니다.

```
1 2 3 6 5 4
```

결국 두번째 데이터까지 정렬이 되었습니다.

그다음은 세번째 데이터와 네번째, 다섯번째, 여섯번째 위치의 데이터를 비교해야겠져? 3 이 제일 작으므로 위치 변동은 없네여. 결국 세번째 데이터까지 정렬이 되었습니다.

다음 네번째와 다섯번째, 여섯번째 데이터를 비교하겠습니다. 6 과 5 를 뒤바꾸면,

```
1 2 3 5 6 4
```

이렇게 되져?

다음에 네번째인 5 와 여섯번째인 4 를 비교해서 뒤바꾸면,

```
1 2 3 4 6 5
```

이렇게 됩니다. 4 번째 데이터까지 정렬되었습니다.

마지막으로, 다섯번째와 여섯번째 데이터를 비교해서 뒤바꾸면 6 과 5 가 서로 바뀌겠 져? 따라서

```
1 2 3 4 5 6
```

이렇게 정렬이 완료가 됩니다.

버블 정렬의 개념과 동작 원리를 알아봤으므로, 버블 정렬을 하는 간단한 예제 코드를 작성해보도록 하겠습니다.

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

```
 2
 3  void swap( int *x, int *y );
 4  void printArray( int *array );
 5
 6  /* swap()
 7   *
 8   * 두개의 인자를 받아서 그 인자의 값을 서로 뒤바꾸어 주는 함수
 9   */
10  void swap( int *x, int *y )
11  {
12        int temp;
13
14        temp = *x;
15        *x = *y;
16        *y = temp;
17  }
18
19  /* printArray()
20   *
21   * 인자로 들어온 배열을 화면에 출력해주는 함수
22   */
23  void printArray( int *array )
24  {
25        int i;
26
27        for( i = 0 ; i < 6 ; i++ )
28              printf( "%2d", array[i] );
29        printf("₩n");
30  }
31
32  int main(void)
33  {
34        int array[6] = { 2, 3, 1, 6, 5, 4 };
35        int i, j;
36
37        for( i = 0 ; i < 6 ; i++ )
38              for( j = i + 1 ; j < 6 ; j++ )
39                    if( array[i] > array[j] )    /* 비교 대상 데이터중 뒤의 것이 작
40                                                      다면 서로 바꾸어 준다 */
41                    {
42                          swap( &array[i], &array[j] );
43                          printArray( array );
44                    }
45
46        return 0;
47  }
```

위에서 개념 설명했던 그대로의 데이터를 가지고 코드를 작성해 보았습니다. 코드를 작성해서 컴파일 하고 실행해 보시면 위에서 설명한 대로 그 과정이 출력될 것입니다.

위의 알고리즘으로 정렬을 하는 것을 버블정렬이라고 부릅니다. 버블정렬은 그 개념이 쉬워서 많이 사용되기는 하지만 단점도 있습니다. 아주 많은 데이터를 정렬할 때 그 효율이 아주 많이 떨어진다는 것입니다. 최악의 배열로 데이터가 존재할 때, 비교하고 자리를

뒤바꾸는 일을 (n*n)/2 번 수행하게 됩니다. 데이터의 갯수가 10 개일때 최고 50 번의 swap 이 일어나야 한다는 뜻입니다. 데이터의 갯수가 많아지면 많아질수록 그 수행 회수는 기하 급수로 늘어나게 되겠져. 따라서 이 버블 정렬은 데이터의 갯수가 적을 경우에 아주 간단하게 적용을 할 수 있는 정렬 알고리즘입니다.

자, 그럼 이제 리스트에 버블 정렬을 적용하도록 함수를 작성해 보도록 하겠습니다.

```
#include <stdio.h>
#include <string.h> /* strlen(), strcpy() */
#include <stdlib.h> /* malloc(), free() */

/* struct _object 를 Object 로 typedef */
typedef struct _object   Object;

/* 구조체 선언 */
struct _object
{
      unsigned long    ID;
      char *           name;
      int              age;

      Object *         prev;    /* 이전 데이터 블럭을 저장하기 위함 */
      Object *         next;
};

/* 전역 변수 선언 */
unsigned long _objectID;
Object *_objectHead = NULL; /* 링크드 리스트의 가장 처음 */
Object *_objectTail = NULL; /* 링크드 리스트의 가장 마지막 */

/* 함수 원형 선언 */
Object * createObject(void);
Object * createObjectWithData( char *name, int age );

int        deleteObject( Object *object );
int        deleteObjectWithName( char *name );
int        deleteObjectWithAge( int age );
void       deleteAllObject(void);

void       printObject( Object *object );
void       printAllObject(void);

void       sortObjectWithAge(void);
void       swapObjectValue( Object *i, Object *j );
void       swap( int *x, int *y );

/* createObject()
 *
 * Object 를 하나 메모리에 할당 받고 링크드 리스트에 연결을 시킨다.
 */
Object *createObject(void)
```

```
45  {
46      Object *object = NULL;
47
48      /* 메모리 할당 */
49      object = (Object *)malloc( sizeof(Object) );
50      if( object == NULL )
51      {
52          fprintf( stderr,
53                  "createObject(): memory allocational error!₩n" );
54          return NULL;
55      }
56      memset( object, 0, sizeof(Object) );
57
58      /* Object 의 ID 를 저장 */
59      object->ID = _objectID++;
60
61      if( _objectHead == NULL ) /* 링크드 리스트가 비어있으면 */
62      {
63          _objectHead   = object; /* 링크드 리스트의 처음이 object */
64          _objectTail   = object; /* 링크드 리스트의 마지막이 object */
65          object->prev = NULL;    /* 방금 생성된 object 의 이전은 NULL */
66          object->next = NULL;    /* 방금 생성된 obejct 의 다음은 NULL */
67      }
68      else /* 링크드 리스트가 비어있지 않으면 */
69      {
70          _objectTail->next = object; /* 현재 마지막 리스트의 다음이
71                                         object */
72          object->prev = _objectTail; /* object 의 이전은 현재의 마지막 데이터 */
73          _objectTail          = object; /* 링크드 리스트의 마지막이
74                                         object */
75          object->next        = NULL;    /* object 의 다음은 NULL */
76      }
77
78      return object;
79  }
80
81  /* createObjectWithData()
82   *
83   * 내부적으로 createObject()를 호출하고 인자로 넘어온 데이터를
84   * 저장한다.
85   */
86  Object *createObjectWithData( char *name, int age )
87  {
88      Object *object = createObject();
89
90      object->name = (char *)malloc( strlen(name) + 1 );
91      strcpy( object->name, name );
92      object->age = age;
93
94      return object;
95  }
96
```

```
 97  /* deleteObject()
 98   *
 99   * 인자로 들어온 object 를 리스트에서 삭제한다.
100   */
101  int deleteObject( Object *object )
102  {
103       if( object == NULL ) return 0;
104
105       /* 삭제할 데이터가 _objectHead 일때 - 가장 처음에 존재하는 데이터
106        * 일때
107        */
108       if( object == _objectHead )
109       {
110            object->next->prev = NULL;
111            _objectHead = object->next;
112       }
113       /* 삭제할 데이터가 가장 나중에 존재하는 데이터 일때 */
114       else if( object == _objectTail )
115       {
116            object->prev->next = NULL;
117            _objectTail = object->prev;
118       }
119       else
120       {
121            object->prev->next = object->next;
122            object->next->prev = object->prev;
123       }
124
125       /* 데이터의 name 속성에 메모리가 할당되어 있으면 해제한다. */
126       if( object->name != NULL ) free( object->name );
127
128       /* 데이터 메모리 블럭을 해제한다. */
129       free( object );
130
131       /* 오류없이 종료했음을 나타내는 1 을 리턴한다. */
132       return 1;
133  }
134
135  /* deleteObjectWithName()
136   *
137   * 리스트에서 인자로 넘어온 name 이 일치하는 데이터를 삭제한다.
138   */
139  int deleteObjectWithName( char *name )
140  {
141       Object *temp = _objectHead;
142
143       while( temp )
144       {
145            if( strcmp( temp->name, name ) == 0 ) break;

146            temp = temp->next;

147       }
148
```

```
149         return deleteObject( temp );
150     }
151
152     /* deleteObjectWithAge()
153      *
154      * 리스트에서 인자로 넘어온 age 가 일치하는 데이터를 삭제한다.
155      */
156     int deleteObjectWithAge( int age )
157     {
158         Object *temp = _objectHead;
159
160         while( temp )
161         {
162             if( temp->age == age ) break;
163             temp = temp->next;
164         }
165
166         return deleteObject( temp );
167     }
168
169     /* deleteAllObject()
170      *
171      * 리스트에 있는 모든 데이터를 삭제한다.
172      */
173     void deleteAllObject(void)
174     {
175         Object *temp = _objectHead;
176         Object *next = NULL; /* 다음 데이터를 저장하는 포인터 */
177
178         while( temp )
179         {
180             /* 블럭을 해제하고 나면 temp->next; 를 사용할 수 없으므로
181              * 블럭을 해제하기 전에 다음 블럭의 주소를 저장해 놓은 다음
182              * 에 블럭을 해제한다.
183              */
184             next = temp->next;
185             if( temp->name != NULL ) free( temp->name );
186             free( temp );
187
188             temp = next;
189         }
190
191         /* 리스트의 처음, 마지막을 나타내는 전역 변수를 초기화 한다. */
192         _objectHead = _objectTail = NULL;
193     }
194
195     /* printObject()
196      *
197      * 인자로 넘어온 데이터를 출력한다.
198      */
199     void printObject( Object *object )
200     {
```

```
201         printf("%2d: %s, %d₩n", object->ID, object->name, object->age );
202     }
203
204     /* printAllObject()
205      *
206      * 리스트의 모든 데이터를 출력한다.
207      */
208     void printAllObject(void)
209     {
210         Object *temp = _objectHead;
211
212         while( temp )
213         {
214             printObject( temp );
215             temp = temp->next;
216         }
217     }
218
219     /* sortObjectWithAge()
220      *
221      * 리스트에 있는 데이터들을 나이 순서대로 정렬을 하는 함수
222      */
223     void sortObjectWithAge(void)
224     {
225         Object *i = NULL, *j = NULL;
226
227         for( i = _objectHead ; i != NULL ; i = i->next )
228             for( j = i->next ; j != NULL ; j = j->next )
229                 if( i->age > j->age )
230                     swapObjectValue( i, j );
231     }
232
233     /* swapObjectValue()
234      *
235      * 두개의 데이터 인자들의 내용을 바꾸어 주는 함수
236      * 실제로 리스트에 있는 데이터를 바꾸어 주기 위해서는 두가지 방법이 있다.
237      *
238      * 실제로 객체가 갖고 있는 멤버들의 값을 전부 바꾸어 주는 방법과,
239      * 객체들의 연결 고리인 next, prev 를 바꾸어 주는 방법.
240      *
241      * 이 함수는 첫번째 방법으로 작성된 합수이다.
242      */
243     void swapObjectValue( Object *i, Object *j )
244     {
245         swap( &i->age, &j->age );    /* 나이 멤버값을 바꾼다 */
246
247         /* name 멤버가 가지고 있는 이름이 저장되어 있는 곳의 주소값을
248          * 서로 바꾸어 준다. 포인터의 크기도 4 바이트이고 int 의 크기도
249          * 4 바이트 이므로 포인터가 갖고 있는 주소값을 (int *)형으로
250          * 캐스팅 한 다음에 swap()을 호출한다.
251          *
252          * NOTE:
```

```
253         *     별로 추천하고 싶은 방법은 아니지만 원하는 결과는 훌륭히
254         *     해냅니다.
255         */
256        swap( (int *)&i->name, (int *)&j->name );
257    }
258
259    /* swap()
260     *
261     * 두개의 인자를 받아서 그 인자의 값을 서로 뒤바꾸어 주는 함수
262     */
263    void swap( int *x, int *y )
264    {
265        int temp;
266
267        temp = *x;
268        *x = *y;
269        *y = temp;
270    }
271
272    int main(void)
273    {
274        /* 4 개의 데이터를 생성한다.
275         * 여기서는 createObjectWithData() 함수가 리턴하는 주소값은
276         * 무시하고 있다.
277         * 하지만 생성된 데이터들이 연결되어 있는 리스트의 첫번째 값은
278         * _objectHead 전역 변수에 의해서 알 수가 있다.
279         */
280        createObjectWithData( "Shim sang don", 28 );
281        createObjectWithData( "David", 25 );
282        createObjectWithData( "Robert", 34 );
283        createObjectWithData( "Josephin", 26 );
284
285        printAllObject();
286
287        printf("-------------------------₩n");
288
289        sortObjectWithAge();
290        printAllObject();
291
292        /* 리스트의 모든 데이터를 삭제한다. */
293        deleteAllObject();
294
295        return 0;
296    }
```

앞장의 예제 코드에 정렬을 하는 함수를 추가했습니다. 어떤 부분이 추가되었는지 비교해서 살펴보세여..... 라고는 했지만 단지 함수 3 개 늘어나고 main()에서 아주 조금 바뀌었을 뿐입니다.

223 라인의 sortObjectWithAge() 함수 내용을 보시면, 위에서 int 배열을 정렬하는 함수와 모양도 아주 흡사한 것을 볼 수가 있습니다. 루프를 돌리기 위해서 사용된 변수가 Object * 형이라는 점이 조금 다르지여. 이 함수에서 호출하고 있는

swapObjectValue() 함수의 내용이 중요합니다.

233 라인에 적혀있는 주석문을 읽어보시면 알겠지만, 리스트에 있는 데이터를 바꾸어 주기 위해서 보통 2 가지 방법을 사용합니다. 리스트에 있는 데이터(객체)의 멤버들을 전부 바꾸어 주는 방법과, 객체가 가리키고 있는 next, prev 포인터를 바꾸어 주는 방법이 있습니다.

첫번째 방법은 가장 알기도 쉽고 코딩하기도 어렵지는 않지만, 객체의 구조체가 수정되면 swap()해주는 함수의 내용도 바꾸어 주어야 하는 불편함도 있고, 가장 중요한 것은 시간이 많이 걸린다는 것입니다. 멤버의 값들을 일일이 다 바꾸어 주는 식의 코딩을 하면 멤버가 20~30 개 이상의 많은 수로 이루어져 있다면 그만큼 많은 대입하는 프로세스가 필요하게 되는 것입니다.

두번째 방법은 링크드 리스트의 next, prev 포인터 연결 고리를 재 설정 해줌으로써 데이터 값을 바꾸는 것이 아니라 객체의 리스트 내에서의 위치를 바꾸어주는 것입니다. 다만 링크드 리스트의 next, prev 포인터를 문제없이 바꾸게 하려면 여러가지 생각하여야 할 것이 있기 때문에 자칫 실수 하기가 쉽습니다.

위의 예제에서는 첫번째 방식으로 코딩을 했습니다. 그 이유는 예제 구조체의 구조가 매우 간단하며, 한번에 많은 생각을 하게 하면, 이 강좌를 읽으시는 분들의 흰머리가 하나라도 더 늘것을 염려해서 입니다.

이번 장은 여기서 마칠까 합니다. 마치기 전에 위의 sortObjectWithAge() 함수의 기능처럼 name 멤버에 대한 정렬 함수를 작성해 보시기 바랍니다. 그리고 리스트의 prev, next 포인터를 이용해서 정렬하는 함수도 작성해 보시기 바랍니다. 힌트는 위 예제 코드의 101 라인에 있는 deleteObject() 함수의 내용을 참조하시기 바랍니다.

---

8 장. 포인터의 깊은 곳

차례

포인터의 기본 개념인 "다른 변수의 주소값을 저장하고 있으며 그 주소값을 이용하여 다른 변수를 참조할 수 있다" 를 처음에 나왔던 그림과 함께 정확하게 이해하고 있으면 지금까지의 포인터 예제들에 대해서 큰 어려움은 없을것입니다.

하지만 포인터는 메모리와 아주 밀접하게 연관이 되어 있으므로 메모리에 대한 이해가 뒷받침이 되어주지 않으면 자유자재로 사용하는데 제약이 따르게 됩니다. 이러한 제약은 좀 더 강력한 프로그램을 작성할 때 자주 장애로 나타나며 어쩔 수 없이 메모리를 더 사용한다던가 혹은 코드를 더 길게 작성하는 일들이 일어나게 됩니다.

포인터의 깊은 곳이라는 제목의 이번 장은 좀 더 포인터의 본질에 대해서 접근을 해 볼수 있는 장이 될 것입니다.

8.1. 자료형의 크기와 포인터

자료형이란 보통 char, int, long, float 등등의 키워드를 지칭하며 이러한 각각의 자료형들은 일정한 크기의 메모리를 할당하게 됩니다.

예를 들어 char 형은 몇 bit 운영체제인지에 관계 없이 1 byte(8 bit)의 메모리를 할당하게 됩니다. int 형은 운영체제의 bit 수에 따라가게 되어 있는데, 16 bit 운영체제(예를 들어 DOS)에서는 int 가 2 byte(16 bit)가 할당 되며, 32 bit 운영체제(Windows NT, 2000, UNIX, Linux 등등)에서는 int 가 4 byte(32 bit)가 할당이 됩니다.

이렇게 자료형들은 운영체제에 따라서 그 크기가 유동적입니다. 마찬가지로 포인터 역시 운영체제에 따라서 크기가 달라집니다.

16 bit 운영체제에서는 포인터 변수의 크기는 2 byte(16 bit) 이며, 32 bit 운영체제에서는 4 byte(32 bit) 입니다.

노파심에 포인터 변수의 크기에 대해서 오해를 하는 독자들이 있을 지 몰라서 다시 한번 포인터 변수의 크기를 예를 들어보겠습니다.

```
/* 32 bit 운영체제라고 가정 */

#include <stdio.h>

int main(void)
{
	char *cp = NULL;
	int *ip = NULL;
	long *lp = NULL;
	double *dp = NULL;

	printf( "cp = %d bytes₩n", sizeof(cp) );
	printf( "ip = %d bytes₩n", sizeof(ip) );
	printf( "lp = %d bytes₩n", sizeof(lp) );

	printf( "dp = %d bytes₩n", sizeof(dp) );
```

```
    return 0;
}
```

```
---- 출력 결과 ----

cp = 4 bytes
ip = 4 bytes
lp = 4 bytes
dp = 4 bytes
```

위에서 보듯이 포인터 변수의 앞에 있는 자료형에 관계 없이 모든 포인터 변수는 4 바이트를 차지하고 있습니다. 그럼 포인터 앞에 붙은 자료형은 무엇인가요? 라는 질문을 아주 많이 들어 보았습니다. 그 질문에 대한 대답은 다음과 같습니다.

```
포인터 변수 앞에 붙은 자료형의 뜻은 그 포인터 변수가 저장하고 있는 주소에
들어가 있는 값의 자료형이라는 뜻입니다.

예를 들어서

int *ip = &number;

라는 코드가 있을 때 ip 의 크기는 4 바이트이며 ip 가 저장하고 있는 주소에 있는
값의 자료형은 int 형이라는 뜻입니다.
```

모르고 있었던 독자들은 꼭 기억을 해주시고 알고 있었던 독자들은 돌다리도 두드려 보고 건넌다는 생각으로 다시 한번 상기시켜주길 바랍니다.

별것 아닌 것을 왜 커다란 장(章)까지 만들면서 설명을 하려하는지 의문을 가질 수 있지만 다음에 나올 각종 예제들을 보면 이해가 갈 것입니다.

8.1.1. 크기가 다른 변수

그동안 항상 int 형으로 선언한 변수에는 int 값만 대입을 해 왔을 것이고, 가끔 작은 크기의 값을 대입한 적도 있었을 것입니다. 그렇다면 다음의 예제를 보시기 바랍니다.

```
#include <stdio.h>

int main(void)
{
    int a = 0x01020304;
    int i = 0;

    printf("&a = %p₩n₩n", &a );
```

```
    for( i = 0 ; i < 32 ; i++, ( i && i % 8 == 0 ) ? printf(" ") : 0 )
        printf("%d", a & ( 1 << (31-i) ) ? 1 : 0 );

    printf("₩n₩n");

    for( i = 0 ; i < 4 ; i++ )
        printf("((char *)&a)[%d] = %d, (%p)₩n",
                        i, ((char *)&a)[i], &((char *)&a)[i] );

    return 0;
}
```

```
---- 출력 결과 ----

&a = 0x22feb0

00000001 00000010 00000011 00000100

((char *)&a)[0] = 4, (0x22feb0)
((char *)&a)[1] = 3, (0x22feb1)
((char *)&a)[2] = 2, (0x22feb2)
((char *)&a)[3] = 1, (0x22feb3)
```

위의 코드는 int 타입의 a 변수에 0x01020304 의 값을 넣고 메모리에 값이 존재하는 모습을 출력하고 a 변수가 존재하는 메모리상의 주소를 char 로 인식을 하도록 하여 값들을 출력한 모습입니다. 출력 결과만 봐서 잘 이해가 가지 않는 독자를 위하여 그림으로 그려보겠습니다.

```
 0x22feb0       0x22feb1       0x22feb2       0x22feb3
   +------------+------------+------------+------------+
   |  00000100  |  00000011  |  00000010  |  00000001  |  a
   +------------+------------+------------+------------+
```

가만히 보면 뭔가 뒤바뀌어 있는 것을 알 수가 있습니다. 그 이유는 테스팅을 한 환경이 IBM-PC 의 Intel CPU 를 사용하는 Linux 입니다. Intel 의 PC 용 CPU 는 역워드 방식을 사용하기 때문에 실제 메모리상에 값이 들어갈 때 그 값이 뒤집어져서 들어가게 됩니다.

따라서 int 형인 a 를 사용하게 되면 메모리에서 읽은 값을 다시 뒤집어서 사용하므로 0x01020304 값이 되지만, 실제로 메모리상에 들어있는 모습을 출력하기 위해서 ((char *)&a)[0] 의 값을 출력하면 1 이 아닌 4 가 나오게 되는 것입니다.

바로 여기에서 포인터의 위력이 나오게 되는 것입니다. int 형의 a 변수 주소값은 0x22feb0 인데 이 주소값을 접근할 때 어떻게 접근을 하겠는지를 판단하는 것이 바로 포인터 변수의 앞에 적혀있는 자료형입니다.

다시 위의 예제로 돌아가서, ((char *)&a)[0] 의 의미를 생각해 봅시다.

&a 는 a 변수의 주소 즉, 0x22feb0 입니다. 이 주소에 들어있는 값은 원래 int 형인 0x01020304 값이져? 그런데 강제로 이 주소값에 들어 있는 값이 char 형이라고 캐스팅을 해주었습니다. (char *)&a 이렇게 말이져.

char 형이라고 해주는데 왜 (char) 가 아니고 (char *)로 캐스팅을 하느냐고 생각하는 독자가 있을겁니다. 그 이유는 &a 가 나타내는 것은 주소값입니다. 따라서 주소값을 저장하는 것은 포인터져? 따라서 (char)로 캐스팅 하는 것이 아니라 (char *)로 캐스팅을 하여 &a, 즉 0x22feb0 안에 char 형 값이 들어있다라고 하는 것입니다.

그 다음에 ((char *)&a)[0] 과 같이 표현을 했는데 이것도 의아해 하실 독자가 많을거라고 생각되네여. 일단 ((char *)&a)[0]을 설명하기 전에 배열에 관해서 좀더 깊은 의미를 설명하겠습니다.

다음의 예제를 보겠습니다.

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

char *string(void);

int main(void)
{
	int i = 0;
	int len = 0;

	len = strlen( string() );

	printf("string = %s₩n", string() );

	for( i = 0 ; i < len ; i++ )
		printf("%d: %c₩n", i, string()[i] );

	return 0;
}

char *string(void)
{
	return "ABCDEFG";
}
```

```
---- 출력 결과 ----

string = ABCDEFG
0: A
1: B
2: C
3: D
4: E
5: F
```

6: G

위의 예제에서 생소한 모습을 볼 수가 있습니다. string()[i] 와 같은 표현이 이상해 보일 수가 있어도 전혀 그렇지 않다는 것을 설명하겠습니다.

```
int a[5] = { 0, 1, 2, 3, 4 };

a[0] == *( a + 0 ) == 0
a[1] == *( a + 1 ) == 1
a[2] == *( a + 2 ) == 2
a[3] == *( a + 3 ) == 3
a[4] == *( a + 4 ) == 4
```

위의 코드는 이해가 갈 것입니다. a[3] 은 *(a + 3) 과 같은 의미라는 것을 이미 전에 얘기를 했으므로 기억을 할 것입니다. 그렇다면 a[3]의 의미를 다시 한번 짚어 보겠습니다.

실제로 a[3]의 의미는 a 배열의 주소로 부터 3 번째 값을 뜻하는 것입니다. 좀 더 자세히 말하면 a 배열의 자료형은 int 입니다. 그러므로 a 배열의 주소로부터 sizeof(int) * 3 만큼 더한 주소값에 있는 값이 바로 a[3]인 것입니다.

따라서 a[3]의 코드는 실제로 3[a] 와 같이 표현을 해도 전혀 이상이 없는 코드가 됩니다. 언뜻 보기에는 말도 안되는 코드라고 생각되겠지만 위에서 설명한 것처럼 배열을 포인터로 고쳐보면 이해가 가게됩니다.

```
a[3] == *( a + 3 )
3[a] == *( 3 + a )

a + 3 과 3 + a 에서 + 는 교환법칙이 성립되므로 둘은 같은 식입니다.
따라서 a[3]과 3[a] 는 완전히 같은 식입니다.

실제로 컴파일러도 전혀 경고라든지 에러를 출력하지 않고 실행이 잘 됩니다.
```

int a[5]; 와 같은 선언을 했을 때 a 는 무엇을 나타내져? a 배열의 첫번째 주소값을 나타내는 포인터 상수입니다.

a == &a[0] 이것은 기억을 할 것입니다. 여기에서 a 의 값이 0x2000 이라고 가정을 하면, a[3]은 (0x2000)[3] 과 같은 의미입니다. 이것은 *(0x2000 + 3) 이렇게 되고 이 의미는 실제적으로는 *(0x2000 + sizeof(int)*3) 의 의미입니다. 따라서 *(0x200C) 가 되는 것입니다.

자, 그럼 저 위로 가서 string()[2] 의 의미를 다시 알아봅시다. string() 함수의 역할이 무엇이져? "ABCDEFG" 라는 문자열을 리턴해주는 역할이져? 실제적으로는 "ABCDEFG"라는 문자열을 CODE 영역의 constant 영역(read only)에 저장을 하고 그 첫번째 주소값을 리턴하는 역할을 string() 함수가 하는 것입니다.

그럼 string()[2]의 의미를 그림과 같이 설명을 하겠습니다.

```
0x1000 (CODE 영역중 constant 영역)
  +---+---+---+---+---+---+---+---+
  | A | B | C | D | E | F | G | 0 |
  +---+---+---+---+---+---+---+---+

string() 함수가 리턴하는 값은 문자열 "ABCDEFG"가 저장되어 있는 첫번째 주소값
0x1000 이 됩니다. 따라서 string()[2] 의 뜻은 (0x1000)[2] 의 의미가 되는 것이
고, 이것을 포인터로 바꾸어 보면 *( 0x1000 + 2 )가 됩니다.

sizeof(char)는 1 이므로 0x1000 + 2 는 0x1002 가 됩니다.
그러므로 결국 string()[2]는 *(0x1002)의 의미이며 이 값은 'C' 문자가 됩니다.
```

이렇게 배열은 num[4] 와 같은 모습이 아닌 string()[5] 과 같은 모습이 될 수가 있는 것입니다. 왜냐하면 C 언어에서 배열은 포인터로 바뀌어서 계산이 되기 때문에 그렇습니다.

자, 이렇게 해서 배열에 대한 설명은 줄이기로 하고, 다시 ((char *)&a)[0] 의 의미를 알아보겠습니다.

&a 의 값은 0x22feb0 이며 (char *)에 의해서 0x22feb0 에 저장되어 있는 값은 char 형이라고 강제 캐스팅 되었습니다. 그러면 ((char *)&a)[0]의 의미는 어떻게 될까여? (0x22feb0)[0]이 되며 다시 이것은 *(0x22feb0 + 0) 이 됩니다. 결국 *(0x22feb0) 가 되며 0x22feb0 번지에 가서 1 byte 를 읽어온 값(왜냐면 char * 형으로 강제 캐스팅을 했으므로 1 byte 를 읽어오게 됩니다.)을 나타내어 4 가 되는 것입니다.

8.1.2. 문자열 출력

앞에서 설명한 포인터의 강제 캐스팅에 의해서 int 값을 1 바이트씩 끊어서 읽을 수 있는 방법을 알아보았습니다.

이번엔 long 형의 배열에 값을 넣어서 문자열로 출력하는 것을 해보겠습니다.

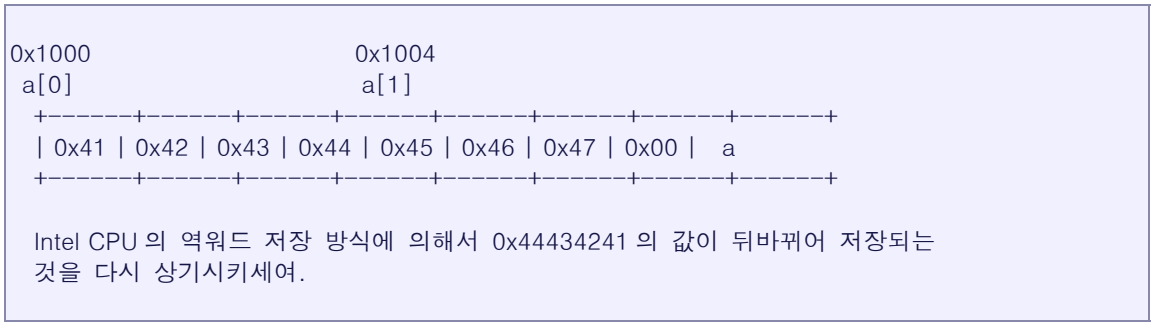
```
#include <stdio.h>

int main(void)
{
    long a[2] = { 0x44434241, 0x00474645 };

    printf("a = %s₩n", (char *)a );
    return 0;
}
```

```
---- 출력 결과 ----
```

```
a = ABCDEFG
```

코드는 매우 짧지만 아주 많은 것을 내포하고 있습니다. long 형 배열 a 에 0x44434241, 0x00474645 값을 대입하고 a 를 (char *)로 캐스팅 하여 문자열로 출력을 하고 있습니다.

이 코드를 눈으로 보려하지 말고 다음과 같이 그림으로 그려봅시다.



메모리에 위와 같이 값들이 들어가 있게 됩니다. 그리고 (char *)a 를 문자열로 출력을 하도록 했습니다. 즉 0x1000 번지에 가서 한 바이트씩 읽어서 0 을 만나기 전까지 문자로 출력을 하는 것입니다.

0x41 은 A 의 아스키 코드 값입니다. 따라서 ABCDEFG 가 출력이 되는 것입니다.

위의 예제는 역워드 저장 방식과 포인터의 캐스팅에 관해서 정확한 지식을 필요로 하는 하드웨어 제어 및 임베디드 시스템 개발과 같은 분야를 하기 위한 가장 기초적이면서도 중요한 예제입니다.

8.2. 동적 메모리 할당

이 장은 동적인 메모리 할당(Dynamic memory allocation)에 관한 내용을 다룹니다. 프로그램을 코딩하다 보면 단순한 배열만으로는 충분하지 않을 때가 많습니다. 그 이유는 사용자들이 입력하는 데이터의 갯수가 일정하지 않기 때문입니다.

배열을 사용하게 되면 일반적으로 정해져 있는 메모리의 크기에 데이터를 저장하여 사용을 하게 되는데, 사용자들의 입력이 큰지 작은지에 따라서 배열의 크기도 결정이 됩니다. 이 배열의 크기는 런타임(Run-time;프로그램이 실행되는 때)시에 변경이 되지 않으므로 한번 크기가 결정되면 그 배열의 크기를 임의로 변경시킬 수가 없습니다.

이러한 경우에 사용이 되는 동적 메모리 할당에 대해서 자세하게 다루어보도록 하겠습니다.

8.2.1. 메모리의 영역

동적 할당을 공부하기 전에 메모리의 영역에 대해서 알아보겠습니다. 지역 변수, 전역 변수, 코드 등등이 각자가 다른 메모리 영역을 사용하며 그 쓰임새도 다릅니다.

8.2.1.1. 상수 영역(Constant area)

상수 영역(Constant area)는 변하지 않는 값들이 저장되는 영역으로 큰 따옴표로 묶여진 문자열등이 저장되는 메모리 영역입니다. 이미 문자열에 관해서 설명을 할 때에 잠시 언급이 되었던 것으로 다시 상기시키는 목적으로 설명을 하겠습니다.

Constant area 는 이 곳에 저장된 데이터들은 변하지 않는 특성을 갖고 있으므로 임의로 바꿀 수가 없습니다. 즉, 읽기는 가능하지만 쓰기는 불가능 하다는 말입니다.

```
char *s = "Hello";
```

위의 코드에서 "Hello" 라는 문자열은 Constant area 에 저장이 되며 s 를 통해서 문자열을 읽을 수는 있지만 s[0] = 'T'; 와 같은 코드를 이용해서 문자를 바꿀수가 없습니다.

이 Constant area 는 CODE 영역 안에 존재하는 영역입니다.

8.2.1.2. 코드 영역(Code area)

코드 영역(Code area)은 프로그램 코드가 탑재되는 영역으로 이 영역 역시 읽기는 가능하나 쓰기는 불가능한 영역입니다. 코드가 탑재된다는 말은 실행파일이 메모리에 탑재가 될 때 여러가지 제어(if, for, while 등등)을 통한 일을 수행하는 코드부분이 기계 코드로 컴파일 되어 있는데 바로 이 기계코드가 탑재된다는 말입니다. 뿐만 아니라 ".."로 둘러싸인 문자열도 바로 이 코드 영역에 저장이 됩니다.

이 코드 영역은 실제 프로그램을 움직이는 코드가 들어있는 영역이므로 그 영역이 바뀌어지거나 덮어써져 버리면 프로그램이 실행되는데 치명적인 영향을 주게 되므로 함부로 바꿀 수 없도록 되어있습니다.

8.2.1.3. 스택 영역(Stack area)

스택 영역(Stack area)은 프로그램이 동작하면서 필요한 변수들이나, 내부적으로 함수를 호출하면서 함수의 인자들을 pass 하기 위한 임시 데이터들과, 다른 함수로 jump 하면서 현재의 코드 영역 위치를 임시로 저장하기 위한 영역입니다.

이 영역에 저장되는 대표적인 것으로는 지역 변수가 있습니다. 지역 변수는 메모리의 스택 영역에 저장이 되며 그 지역 범위(local scope)를 벗어나게 되면 폐기해 버리기 때문에 또 다른 지역 변수에 의해서 이미 폐기된 지역 변수가 존재했던 메모리 주소를 다른 지역 변수가 사용할 수 있습니다.

이 스택 영역은 무한정 존재하지 않기 때문에 너무 큰 지역변수를 사용하게 되면 스택 오버플로우(Stack overflow)가 일어나서 프로그램이 종료되어 버릴 수도 있습니다.

특히나 재귀 호출(recursive call)을 하게 되면 호출되는 깊이 만큼 지역 변수가 스택 영역에 생성이 되고, 코드 영역의 함수 호출 부분 주소도 스택 영역에 임시로 저장하고 있게 되므로 스택 영역을 많이 사용하게 됩니다.

이 스택 영역은 읽기와 쓰기가 다 가능합니다.

8.2.1.4. 전역 영역(Global area)

전역 영역(Global area)은 전역 변수들이 저장되는 영역으로 초기화 된 전역 변수 영역과 초기화 되지 않은 전역 변수 영역으로 나누어집니다. 하지만 둘의 특성의 크게 차이가 없으므로 설명은 생략합니다.

이 전역 영역 역시 그 크기가 내부적으로 정해져 있으므로 전역 변수도 너무 많이 사용하지 않는 것이 좋습니다.

이 전역 영역도 읽기와 쓰기가 다 가능합니다.

8.2.1.5. 힙 영역(Heap area)

다른 영역들이 코드가 컴파일 되면서 이미 그 크기들이 정해지는 반면에 이 힙 영역(Heap area)은 실행이 되면서 그 크기가 늘어나고 줄어듭니다. 즉, 프로그래머가 자유롭게 할당을 하여 사용을 할 수 있는 영역입니다.

자유 영역이라고도 불리며 이 영역의 메모리를 할당하는데 사용되는 함수는 malloc(), realloc() 등이 있으며 사용한 힙 영역의 메모리를 해제하는 함수는 free()가 있습니다.

이 영역의 메모리는 요즘같은 32 비트 운영체제에서 물리적인 메모리가 가능한한 전부 사용을 할 수가 있습니다. 만약 물리적인 메모리가 부족하다면 디스크 swap 을 통해서 가상 메모리를 사용할 수도 있게 해줍니다. 따라서 힙 영역의 사용가능한 크기는 남아있는 물리적인 메모리(RAM)의 크기에 남아있는 하드디스크의 크기라고 생각하면 됩니다.

물론 이 영역도 읽기와 쓰기 모두 가능합니다.

8.2.2. 배열의 크기

배열의 크기는 컴파일타임(Compile-time;컴파일을 할 때)시에 결정이 됩니다. 보통 배열을 선언할 때 배열의 크기는 다음과 같이 배열을 선언할 때 결정을 합니다.

```
#define MAX_NUM      500

int      array[100];
double board[MAX_NUM];
```

array 라는 이름의 int 형 100 개로 이루어진 배열을 선언하고, board 라는 이름의 double 형 MAX_NUM(500)개로 이루어진 배열을 선언했습니다. 이렇게 프로그래머가 배열의 크기를 직접 정해주는 것입니다.

배열의 크기가 컴파일타임이 아닌 런타임시에 결정이 되는지 테스트를 하기 위해서 다음과 같은 코드를 작성해서 실행을 해보도록 하겠습니다.

```
#include <stdio.h>

void makeArray( int num );

int main(void)
{
     int num = 0;

     printf("Input array number: ");
     scanf("%d", &num );

     makeArray( num );

     return 0;
}

void makeArray( int num )
{
     int i = 0;
     int array[num];  /* 바로 이부분이 런타임시에 배열의 크기를 결정하기
                         위해서 넣은 코드입니다. 언뜻 보기에는 될 것 처럼
                         보이지만 실제로는 그렇지 않습니다. */

     for( i = 0 ; i < num ; i++ )
          array[i] = i;

     for( i = 0 ; i < num ; i++ )
          printf("array[%d] = %d₩n", i, array[i] );
}
```

위의 코드를 Turbo-C 2.0 컴파일러에서 컴파일을 한 결과 출력되는 에러 메세지는 다음과 같습니다.

```
Error exam.c 20: Constant expression required in function makeArray
Error exam.c 20: Size of structure or array not known in function makeArray
```

첫번째 에러의 의미는 상수 수식만을 사용해야 되는 곳에 변수나 수식등이 들어있을때 나타나는 에러이며, 두번째 에러의 의미는 구조체나 배열의 크기가 정해져 있지 않았을 때 나타나는 에러입니다.

결과를 봐서 알 수 있듯이 배열의 크기는 변수를 통해서 동적으로 그 크기가 변화될 수 있는 것이 아닙니다.

참고:

gcc(GNU cc) 에서는 동적으로 배열의 크기를 결정할 수 있도록 컴파일러 차원에서

지원해주고 있습니다. 따라서 위의 코드를 gcc 로 컴파일 하면 전혀 문제 없이 실행이 될 것입니다. 하지만 모든 컴파일러가 동적인 배열을 지원하는 것은 아니므로 위와 같은 코드는 gcc 로만 컴파일 할 것이면 상관 없지만 기타 다른 컴파일러로도 컴파일을 해야한다면 피해야 하는 코드입니다.

8.2.3. malloc() 사용하기

앞의 예제를 동적인 메모리 할당으로 수정을 해보겠습니다.

```
#include <stdio.h>
#include <stdlib.h> /* malloc(), free() 함수를 사용하기 위해서는
                       stdlib.h 를 포함해야 합니다. */

void makeArray( int num );

int main(void)
{
    int num = 0;

    printf("Input array number: ");
    scanf("%d", &num );

    makeArray( num );

    return 0;
}

void makeArray( int num )
{
    int i = 0;
    int *array = NULL;

    /* array 포인터에 malloc()으로 메모리를 할당 받아서 그 할당 받은 메모리의
     * 주소값을 저장합니다. 만약 malloc()이 메모리 할당에 실패를 하면 NULL 을
     * 리턴하므로 메모리를 할당한 다음에는 꼭 에러체크를 해야 합니다.
     */
    array = (int *)malloc( sizeof(int) * num );
    if( array == NULL )
    {
        fprintf( stderr, "%s:%d: memory allocational error!₩n",
            __FILE__, __LINE__ );
        exit(1);
    }

    for( i = 0 ; i < num ; i++ )
        array[i] = i;

    for( i = 0 ; i < num ; i++ )
        printf("array[%d] = %d₩n", i, array[i] );
```

```
    /* 할당 받은 메모리의 주소를 갖고 있는 array 를 이용해서 그 메모리를
     * 해제해주는 코드를 잊지 않아야 합니다.
     */
    free( array );
}
```

malloc() 함수를 이용해서 메모리를 동적으로 할당 받아서 원하는 크기의 배열을 만들어서 사용하는 예제를 들어보았습니다. 여기에서 malloc()으로 할당 받은 메모리를 배열처럼 사용한 것에 대해서는 이미 앞에서 배열은 포인터로 바꾸어서 컴파일러가 이해를 한다는 언급을 했으므로 그것에 관한 설명은 생략하겠습니다.

malloc()을 사용해서 sizeof(int) * num 만큼의 크기만큼 Heap area 에 메모리를 할당해서 사용을 하는 모습을 보았습니다. 이렇게 프로그래머가 필요할 때 수시로 할당하고 해제하는 일을 자유롭게 해주는 Heap area 를 사용하게 되면 프로그램은 굉장히 유연해 지며 다룰 수 있는 데이터의 양도 많아질 수가 있게 됩니다.

보통의 간단한 프로그램이라면 아주 적은 양의 메모리로도 전부 가능합니다. 즉, Stack area 에 저장될 수 있는 지역 변수들만 가지고도 충분히 프로그램이 실행 될 수가 있지만 다루어야 할 데이터가 많은 프로그램인 경우에는 Stack area 만 가지고는 불가능 하므로 Heap area 를 자주 사용하게 됩니다.

예를 들면 백만명 이상의 사용자에 대해서 검색을 하거나 삽입하고 삭제하는 일들을 하는 커다란 프로그램을 상상해 보면 알 수가 있을 것입니다. 그 많은 데이터들을 Stack area 에 배열을 잡아서 사용하기에는 엄청난 무리가 있으므로 Heap area 를 사용하게 되는 것입니다. 물론 실제로는 디스크에 저장되어 있는 데이터 베이스 파일로 부터 일정한 크기 만큼 읽어들이면서 검색을 하도록 되어있습니다. 백만명 이상의 사용자 정보를 메모리에 다 올려놓을 수는 없으니까여. :)

malloc()으로 Heap area 에 메모리를 할당하여 사용을 하고 난 후에는 꼭 해제를 해주어야 합니다. 물론 프로그램이 종료가 될 때 그 프로그램이 사용했던 모든 메모리를 OS 가 알아서 전부 해제해 주지만 프로그램이 실행되고 있을 때에는 최대한 해제해 줄 수 있는 메모리는 해제해 주는 것이 좋습니다. 컴퓨터에서 내가 짠 프로그램만 실행되는 것이 아니라 다른 프로그램도 동시에 실행되고 있을 수 있기 때문에 사용을 하고 난 후에 더 이상 사용하지 않는 할당된 메모리가 있으면 꼭 그때그때 해제를 해 주어야 합니다.

메모리를 해제할 때에는 free() 함수를 사용합니다. 이 함수는 인자로 Heap area 의 주소값을 받아서 그 주소에 할당되어 있는 메모리를 더이상 사용하지 않는 다고 OS 에게 알려주는 일을 합니다. 그래서 OS 에서는 다른 프로그램이 malloc() 을 호출해서 Heap area 의 메모리를 요구할 때 사용하지 않는 부분을 할당해 줄 수 있도록 하는 것입니다.

이렇게 free() 함수를 통해서 더 이상 사용하지 않는 메모리를 해제해 주지 않으면 메모리 누수(memory leak)가 발생해서 다른 프로그램이 사용할 수 있는 Heap area 의 영역을 쓸데없이 잡아먹게 됩니다. 그러므로 malloc()을 한 메모리는 꼭 free() 함수로 해제를 하는 습관을 들여야 합니다.

8.2.4. realloc() 사용하기

malloc()으로 할당한 메모리를 사용하다 보면 그 크기를 늘려야 하거나 줄여야 할 필요가 발생합니다. 그럴 경우에는 다음과 같은 코드를 사용합니다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> /* memcpy()를 사용하기 위해서 포함 */

int main(void)
{
    char *s = NULL;
    /* 여기에서 11 bytes 만큼 메모리를 할당 합니다. */
    s = (char *)malloc( 11 );
    if( s == NULL )
    {
        fprintf( stderr, "memory allocational error!₩n" );
        exit(1);
    }
    strcpy( s, "0123456789" );

    /* 만약 s 가 가리키는 Heap area 에 할당되어 있는 메모리를 늘려주기 위해서는
     * 다음과 같이 코딩을 합니다.
     */
    {
        /* temp 라는 임시 포인터 변수를 만들어서 100 bytes 를 할당하고
         * s 가 가리키고 있는 주소에 있는 데이터를 복사한 다음에
         * s 를 해제하고 s 에 temp 가 가리키고 있는 주소를 대입합니다.
         */
        char *temp = NULL;
        temp = (char *)malloc( 100 );
        if( temp == NULL )
        {
            fprintf( stderr, "memory allocational error!₩n" );
            exit(1);
        }
        memcpy( temp, s, 20 );
        free(s);
        s = temp;
    }

    strcat( s, "abcdefghijklmn" );

    printf("s = %s₩n", s );

    free(s);

    return 0;
}
```

위의 코드 처럼 temp 포인터 변수에 원하는 크기의 메모리를 할당 한 다음에 s 의 주소에 있는 데이터를 복사한 다음에 원래의 s 는 해제를 하고 temp 가 가리키고 있는 주소를 s 가 가리키게 함으로써 메모리의 크기를 늘려주었습니다.

약간 불편하기는 하지만 전혀 문제 없는 코드로 실제 많이 사용되기도 합니다. 하지만 realloc() 이라는 함수를 사용하게 되면 아주 간단히 해결이 됩니다.

위의 코드를 realloc()을 사용해서 고쳐보도록 하겠습니다.

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

int main(void)
{
    char *s = NULL;
    /* 여기에서 11 bytes 만큼 메모리를 할당 합니다. */
    s = (char *)malloc( 11 );
    if( s == NULL )
    {
        fprintf( stderr, "memory allocational error!₩n" );
        exit(1);
    }
    strcpy( s, "0123456789" );

    /* realloc()으로 직접 s에 있는 메모리의 크기를 늘려줍니다. */
    s = (char *)realloc( s, 100 );
    if( s == NULL )
    {
        fprintf( stderr, "memory allocational error!₩n" );
        exit(1);
    }
    strcat( s, "abcdefghijklmn" );

    printf("s = %s₩n", s );

    free(s);

    return 0;
}
```

아주 간단하게 해결이 되져? 이때 처음에 malloc()으로 할당된 Heap area 의 주소와 realloc()으로 다시 바뀐 주소값은 같을 수도 있고 다를 수도 있습니다.

원래 할당되어 있는 메모리의 뒤에 사용하고 있지 않아서 연속해서 메모리를 늘릴 수가 있으면 s 의 주소는 변하지 않지만 만약 원래 할당되어 있는 메모리의 뒤에 다른 데이터를 사용하고 있다면 늘리고자 하는 만큼의 메모리가 확보되는 부분에 메모리를 할당하고 원래의 메모리에 있던 데이터를 복사해줍니다. 즉, 첫번째 예제에서 temp 를 사용한 것과 동일한 일을 내부적으로 해 주는 것이져.

참고 :

realloc()을 사용해서 코딩을 하다가 이상한 문제가 발생하는 경우가 가끔 있습니다. 제대로 메모리를 늘려주거나 줄여주지 못하는 경우가 발생하기도 하기 때문입니다. 그럴 때에는 첫번째 예제 처럼 직접 temp 를 사용하여 메모리를 할당하고 복사해주는 코딩을 하게 되면 문제가 없어지는 경우도 있습니다.

8.2.5. 동적 메모리할당 diagram

지금까지의 동적 메모리 할당에 대해서 그림으로 diagram 을 그려보겠습니다.

```
int *s = NULL;
s = (int *)malloc( sizeof(int) * 3 );


Stack area 에 4 bytes 로 잡혀있는 s 포인터 변수
0x1000
  +---+---+---+---+
  |     0x200010      |  s
  +---+---+---+---+


Heap area 에 할당된 메모리
0x200010
  +---+---+---+---+---+---+---+---+---+---+---+---+
  |    |    |   |    |    |   |    |   |    |    |   |    |
  +---+---+---+---+---+---+---+---+---+---+---+---+


s = (int *)realloc( s, sizeof(int) * 4 );

1. 0x200010 에 할당된 메모리의 뒤에 사용이 가능한 경우

Stack area 에 4 bytes 로 잡혀있는 s 포인터 변수
0x1000
  +---+---+---+---+
  |     0x200010      |  s
  +---+---+---+---+


Heap area 에 할당된 메모리
0x200010
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |    |   |    |   |    |    |    |   |    |   |    |   |    |    |   |   |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

원래의 위치인 0x200010 에 할당되어 있는 메모리의 뒤에 4 bytes 가 덧붙여진다.


2. 0x200010 에 할당된 메모리의 뒤를 다른 곳에서 사용하고 있을 경우

Stack area 에 4 bytes 로 잡혀있는 s 포인터 변수
```

```
0x1000
  +---+---+---+---+
```

```
  |    0x201000     | s
  +---+---+---+---+


Heap area 에 할당된 메모리
0x201000
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

원래의 위치인 0x200010 이 아닌 다른 곳에 sizeof(int) * 4 만큼의 메모리가
할당된다. 여기에서는 0x201000 번지라고 가정.
```

malloc()과 realloc()으로 Heap area 에 메모리를 할당하고 free()를 이용하여 할당 받은 메모리를 해제하는 것을 알아봤습니다.

malloc()으로 메모리를 할당하고 free()로 해제하는 과정을 반복하면서 실제적으로는 Heap area 가 조각조각 나뉘어지게 됩니다. 이렇게 되는 현상을 **메모리 단편화**라고 합니다. 앞의 단일 링크드 리스트부분에서 잠깐 언급했었는데 메모리를 할당하고 해제하는 과정으로 Heap area 에 연속된 메모리가 크게 존재하지 못하고 작은 구멍들이 생기게 됩니다. 그래서 이러한 단편화된 메모리를 조각 모음을 해주지 않으면 실제로 남아있는 메모리는 100Mbytes 라고 하더라도 실제로 1Mbytes 의 메모리를 할당할 수가 없게 되는 현상이 발생합니다. 그래서 OS 에서 스스로 이 단편화된 메모리를 조각모음을 해 주도록 되어있습니다.

OS 에서 메모리를 조각모음 해주는 것에 대해서는 이 강좌의 범위를 벗어나므로 자세한 내용은 다른 서적을 참고하시기 바랍니다.

8.3. 바이트 오더(Byte order)

바이트 오더(Byte order)라는 말을 처음 들어본 독자도 있을 것입니다. 이 바이트 오더라는 말의 뜻은 제조회사가 다른 CPU 의 특성으로 인해서 메모리에 데이터가 저장되는 순서가 다르다는 것을 뜻합니다.

앞에서 Intel PC 용 CPU 는 역워드 저장방식을 사용한다는 말을 언급한 적이 있습니다. 이 말이 사실은 바이트 오더에 관한 내용이었던 것입니다.

그럼 도대체 어떻게 저장방식이 다른 것인지 예를 들어보겠습니다.

```
int a = 0x01020304;

1. Intel 기종
```

```
0x1000
```

```
 +------------+------------+------------+------------+
 |  00000100  |  00000011  |  00000010  |  00000001  |  a
 +------------+------------+------------+------------+
      0x04         0x03         0x02         0x01

 2. VAX, SPARC 기종

0x1000
 +------------+------------+------------+------------+
 |  00000001  |  00000010  |  00000011  |  00000100  |  a
 +------------+------------+------------+------------+
      0x01         0x02         0x03         0x04
```

위에서 보듯이 0x01020304 값을 저장하는데 순서가 완전히 거꾸로 메모리에 저장되는 것을 볼 수가 있습니다. 이렇게 바이트의 순서가 다른 것을 바이트 오더 문제라고 합니다.

그럼 이 바이트 오더 문제가 어떨때 정말 문제로 나타나게 될까여?

다른 기종간에 데이터를 통신할 때 Intel 기종인 PC 에서 VAX 시스템에 0x01020304 의 int 값을 보냈다고 했을 때 바이트 오더가 서로 다르기 때문에 VAX 시스템에서는 0x04030201 값을 받아들이게 되는 것입니다.

그럼 어떻게 서로 바이트 오더가 같은지 알아낼 수가 있을까여? 그렇다고 CPU 의 종류를 알아낼 수도 없는 일이지여. 방법은 포인터를 사용하면 의외로 쉽게 알아낼 수가 있습니다.

다음의 코드를 보세여.

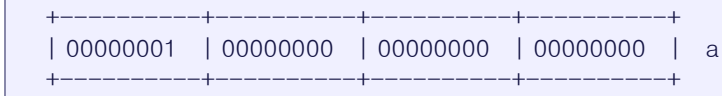
```
#include <stdio.h>

int main(void)
{
    int a = 1;

    if( *(char *)&a == 1 )
        printf("Little endian.₩n");
    else printf("Big endian.₩n");

    return 0;
}
```

위의 코드를 컴파일 해서 실행을 해보면 Intel 기종에서는 **Little endian** 이라는 결과가 나오며 VAX, SPARC 등의 워크스테이션급 시스템에서는 **Big endian** 이라는 결과가 나오게 됩니다.

그럼 왜 그런지 설명을 하겠습니다.

1. Intel 시스템의 경우

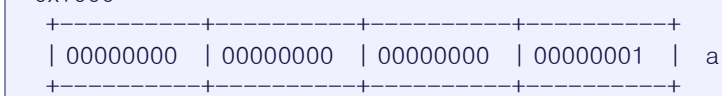
0x1000



*(char *)&a 의 값은 0x1000 번지의 1byte 를 읽어들이게 되어서 1 의 값이 됩니다.

2. VAX, SPARC 시스템의 경우

0x1000



*(char *)&a 의 값은 0x1000 번지의 1byte 를 읽어들이게 되어서 0 의 값이 됩니다.

위와 같이 *(char *)&a 라는 코드를 이용해서 바이트 오더를 알아낼 수가 있는 것입니다. 이렇게 알아낸 바이트 오더를 이용해서 다른 기종간의 통신을 할 때에 바이트 오더의 순서를 바꾸어서 보내주게 되는 것입니다.

포인터에 관한 내용은 아니지만 바이트 오더의 순서를 바꾸어 주는 함수들이 있습니다.

- htons(), htonl()

  htons 는 Host to network short 의 줄임말입니다. Host 시스템에서 Network 로 short 형 데이터를 보낼 때 바이트 오더를 바꾸어주는 함수입니다.

  htonl 은 Host to network long 의 줄임말로 long 형 데이터의 바이트 오더를 바꾸어주는 함수입니다.

- ntohs(), ntohl()

  ntohs 는 Network to host short 의 줄임말로 Network 에서 Host 로 short 형 데이터의 바이트 오더를 바꾸어주는 함수입니다.

  ntohl 은 Network to host long 의 줄임말로 long 형 데이터의 바이트 오더를 바꾸어주는 함수입니다.

위의 함수들은 소켓을 통해 다른 기종간에 데이터를 전송하거나 또는 받아들여서 자신의 바이트 오더에 맞게 변환해 줄 때 사용하는 함수입니다.

Little endian 인지 Big endian 인지를 알아내서 서로의 바이트 오더 정보를 교환한 다음에 그 바이트 오더에 맞추어서 데이터를 통신하게 되면 정확한 값을 송수신 할 수가 있게 되는 것입니다.

8.4. 배열의 음수 첨자

배열에는 항상 0 과 양수의 첨자로써 사용을 해 왔습니다. 하지만 때에 따라서 음수를 사용할 수가 있습니다. 그러한 때는 배열의 기준점을 다른 곳으로 옮겼을 때 발생합니다. 다음의 코드를 보겠습니다.

```
#include <stdio.h>

int main(void)
{
    int    a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int * p = NULL;

    p = &a[5];

    printf("p[ 0] = %d₩n", p[ 0] ); /* p[ 0] == a[5] */
    printf("p[ 1] = %d₩n", p[ 1] ); /* p[ 1] == a[6] */
    printf("p[-1] = %d₩n", p[-1] ); /* p[-1] == a[4] */
    printf("p[-2] = %d₩n", p[-2] ); /* p[-2] == a[3] */
    printf("p[-3] = %d₩n", p[-3] ); /* p[-3] == a[2] */

    return 0;
}
```

위의 코드에서 배열 a 의 기준점을 a[5]로 하여 그 주소값을 p 에 저장을 했습니다. 그리고 p 를 사용하여 값을 프린트 해보았습니다. 결과는 다음과 같습니다.

```
p[ 0] = 5
p[ 1] = 6
p[-1] = 4
p[-2] = 3
p[-3] = 2
```

이제 곧바로 그 이유를 알 수가 있겠지만 혹시나 앞의 강좌를 읽지 않았거나 이해를 못하는 독자를 위해서 간단히 설명을 하겠습니다.

p[-1] 을 예를 들면 p[-1] == *(p + (-1)) 이렇게 됩니다. 여기에서 p 의 값은 &a[5] 이고 여기에서 sizeof(int)만큼 주소를 빼주면 &a[4]가 되겠지여? 따라서 p[-1] 은 a[4]와 같게 되는 것입니다. 사용하면 안되는 코드지만 (-1)[p] 도 같은 의미입니다. 여기서 괄호를 빼면 -a[4]의 뜻입니다. 연산자 우선순위에 의해서 배열이 번저 계산되기 때문입니다.

이렇게 음수가 괄호안에 가능하기 때문에 구태여 괄호안에 양수를 넣기 위해서 다음과 같이 코딩을 하지 않아도 됩니다.

```
for( i = -5 ; i < 5 ; i++ )
```

```
    a[5 + i] = NUM * i;
```

위와 같이 코딩을 한다면 사실 a[5 + i]에서 5 + i가 루프 돌때마다 계속 계산을 해야하므로 수행 속도는 떨어지게 됩니다. 그래서 다음과 같이 바꿔보겠습니다.

```
int *p = &a[5];

for( i = -5 ; i < 5 ; i++ )
    p[i] = NUM * i;
```

이렇게 되면 루프가 돌때마다 계산되던 5 + i가 사라져 버리므로 수행 속도는 빨라지게 되는 것입니다. 배열의 괄호안에 들어가는 값이 음수의 범위로 될 때에 일부러 위의 예제처럼 양수로 하려고 하지 않더라도 포인터를 사용하여 음수를 그대로 사용할 수가 있게 됩니다.

저렇게 해서 수행속도가 얼마나 빨라지겠냐는 의문을 가질 수 있지만, 티끌모아 태산이라고 했습니다. 저런 것들이 코드의 중요한 부분마다 최적화를 이루고 있으면 프로그램의 성능이 좋아지는 것입니다.

---

9 장. 포인터 예제

차례

9.1. 기초 예제

```
#include <stdio.h>

int main(void)
{
    int number = 1000;
    int *p = NULL;

    p = &number;

    printf(" number = %d₩n", number );
    printf("&number = %p₩n", &number );
    printf(" p      = %p₩n", p );
    printf("&p      = %p₩n", &p );
    printf("*p      = %d₩n", *p );

    /* makes compiler happy :) */
```

```
    return 0;
}
```

결과

```
  number = 1000
 &number = 0xbffff988
  p      = 0xbffff988
 &p      = 0xbffff984
 *p      = 1000
```

---

10 장. 부록

차례

10.1. 분할 컴파일

작은 예제를 작성할 때에는 코드 파일이 main.c 혹은 exam.c 처럼 하나의 파일로 이루어지는 경우가 아주 많습니다. 하지만 조금만 프로젝트가 커져도 하나의 코드 파일로 개발할 수 없음을 알 수 있게 됩니다. 따라서 코드 파일을 분할하게 됩니다. 이렇게 코드 파일을 하나가 아닌 여러개로 분할해서 작성하고 컴파일 하는 것을 **분할 컴파일**이라고 합니다.

분할 컴파일을 하는 방법은 크게 두가지로 나뉩니다.

첫째는 하나의 코드 파일로 부터 코드를 분할해내는 방법과, 둘째는 처음부터 프로그램을 설계할 때에 코드를 어떻게 분할해서 작성할지 미리 결정하는 방법입니다.

하나의 코드 파일로 부터 코드를 분할하는 방식은 처음 분할 컴파일에 대한 개념을 잡기위해서 연습을 할때 주로 사용하는 방식입니다. 실제로 실무에서도 전체 설계를 하기 위한 이전 단계로 간단한 테스트 코딩을 하는 경우가 많은데 이렇게 생성된 테스트 코드를 기반으로 분할을 하는 경우도 있습니다.

하지만 실무에서는 설계를 할 때에 프로그램의 성격과 필요한 라이브러리, 클래스 및 구조체에 대한 정의를 하면서 그렇게 정의된 클래스나 구조체들이 각각 다른 파일로 코딩되는 경우가 많습니다. 즉, 설계할 때에 코드의 분할이 이루어지는 것이지요.

그럼 아주 기본적인 코드에서 분할하는 방법에 대해서 알아보겠습니다.

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

```
int plus( int x, int y );

int main(void)
{
	int a = 10;
	int b = 20;
	int sum = 0;

	sum = plus( a, b );

	printf("%d + %d = %d₩n", a, b, sum );

	return 0;
}

int plus( int x, int y )
{
	return x + y;
}
```

위의 코드는 하나의 파일에 main() 함수와 사용자 함수인 plus() 함수를 기술한 것입니다.

이 코드에서 plus() 함수는 재 사용이 가능한 사용자 라이브러리가 될 수 있는 함수입니다. 이 함수만 따로 파일로 만들어서 라이브러리를 만들어 보겠습니다.

```
/* calc.h */

int plus( int x, int y );
/* calc.c */

#include "calc.h"

int plus( int x, int y )
{
	return x + y;
}
/* main.c */

#include "calc.h"

int main(void)
{
	int a = 10;
	int b = 20;
	int sum = 0;

	sum = plus( a, b );
```

```
    printf("%d + %d = %d₩n", a, b, sum );

    return 0;
}
```

위와 같이 세개의 파일로 분리가 됩니다. 이렇게 분리된 코드들을 컴파일 할 때는 다음과 같이 합니다.

```
/* gcc 로 컴파일 하는 예 */

$ gcc -c -o calc.o calc.c          --> (1)
$ gcc -c -o main.o main.c          --> (2)
$ gcc -o exam main.o calc.o        --> (3)
```

- (1)은 plus() 함수가 있는 calc.c 파일을 컴파일 하여 calc.o 파일을 생성.
- (2)는 main() 함수가 있는 main.c 파일을 컴파일 하여 main.o 파일을 생성.
- (3)은 main.o calc.o 두개의 파일을 링크하여 exam 이라는 실행파일을 생성.

gcc 에서 -c 옵션은 compile only 라는 뜻으로 실행파일을 만들기 위한 링킹작업을 하지 않고 컴파일만 하라는 뜻입니다.

이렇게 사용자 함수만 따로 분리하여 calc.c 파일을 만드는 작업을 라이브러리화 라고 합니다.

위의 예제에서 calc.h 파일을 범용적인 모양으로 다듬어 보겠습니다.

```
/* calc.h */

#ifndef __CALC_H__              ---> (1)
#define __CALC_H__

#ifdef __cplusplus              ---> (2)
extern "C" {
#endif

int plus( int x, int y );

#ifdef __cplusplus              ---> (3)
}
#endif

#endif /* __CALC_H__ */       ---> (4)
```

(1)은 calc.h 파일이 다른 소스파일에서 include 될 때에 중복되어 포함되지 않게 하기 위한 테크닉입니다.

지금의 예제와 같이 함수의 원형만 존재할 때에는 크게 효용가치가 없지만 구조체나 매크로등이 선언되었을 경우에 헤더 파일이 중복되어 포함 되면 같은 구조체나 매크로가 선언되었다는 에러를 발생하게 됩니다.

이럴때 (1)처럼 헤더 파일 전체를 감싸 안아주면 제일 처음에 포함될 때에는 __CALC_H__ 라는 매크로가 정의되어 있지 않으므로 __CALC_H__를 정의한 후 아래 이어지는 것들이 선언이 되지만, 한번 읽혀진 후에 다시 이 헤더파일이 포함될 때에는 #ifndef __CALC_H__ 라는 문장 때문에 다음을 진행하지 않고 제일 마지막 줄인 (4)로 건너 뛰게 되므로 중복되어서 선언될 염려가 없습니다.

(2)는 C 로 작성된 라이브러리일 경우 라이브러리 소스를 컴파일 하여 생성된 object 파일을 C++에서 사용할 수 있기 위해서는 호출 규약을 맞추어 주어야 합니다. 호출 규약에 대해서는 이 강좌의 범위를 벗어나므로 다른 책에서 공부해보기 바랍니다.

C 에서 작성된 라이브러리를 C++에서 호출하여 사용하기 위해서는 헤더파일이 extern "C" { .... } 처럼 싸여 있어야 합니다. 따라서 __cplusplus 매크로가 선언되어 있다면(C++로 컴파일 한다면) extern "C" { ... } 가 헤더파일에 포함 되도록 하기 위한 테크닉 입니다.

위와 같이 하나의 파일에서 분리해 낼 수 있는 부분을 떼내어 라이브러리로 작성을 하는 것을 아주 간단히 살펴보았습니다.

하지만 위의 예제에는 상당히 많은 부분이 빠져있고 단지 라이브러리 함수 하나만 딸랑 정의가 되어있었습니다.

이번엔 구조체와 매크로가 존재할 때의 예제를 들어보겠습니다.

```
/* object.h */

#ifndef __OBJECT_H__
#define __OBJECT_H__

#ifdef __cplusplus
extern "C" {
#endif

/* Object 로 캐스팅을 위한 매크로 */
#define OBJECT(object)   ((Object *)(object))

typedef struct _object Object;

struct _object
{
    unsigned long    id;
    char *           name;
};

Object *          createObject(void);
Object *          createObjectWithName( char *name );

unsigned long          getObjectId( Object *object );
void              setObjectName( Object *object, char *name );
```

```
char *              getObjectName( Object *object );

void                destroyObject( Object *object );

#ifdef __cplusplus
}
#endif

#endif /* __OBJECT_H__ */
/* object.c */

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

#include "object.h"

/* __object_id
 *
 * 생성되는 object 들의 id 를 설정하기 위해서 사용되는 변수.
 * createObject()에 의해서 object 가 생성되며 id 가 세팅된다.
 * id 를 세팅해주고 난 후에는 1 증가한다.
 */
unsigned long __object_id = 0L;


/* createObject
 *
 * Object 를 생성
 */
Object *createObject(void)
{
    Object *object = NULL;

    object = (Object *)malloc( sizeof(Object) );
    if( object == NULL )
    {
        fprintf( stderr, "object != NULL failed!₩n" );
        return NULL;
    }

    /* 할당받은 메모리를 0 으로 초기화 한다. */
    memset( object, 0, sizeof(Object) );

    /* object 의 id 를 설정한다.
     * 이 id 는 전역변수인 __object_id 값이 세팅되고 __object_id 의 값은
     * 하나씩 증가한다.
     */
    object->id = __object_id++;

    return object;
}
```

```
/* createObjectWithName
 *
 * Object 를 생성하며 Object 의 name 을 인자로 주어 name 도 세팅한다.
 */
Object *createObjectWithName( char *name )
{
    Object *object = NULL;

    object = createObject();
    if( object == NULL ) return NULL;

    /* strdup()함수를 이용해서 이름을 할당한다.
     * 이 함수는 내부적으로 메모리를 할당하므로 destroy 할 때에
     * 메모리를 해제시켜야 한다.
     */
    object->name = strdup( name );

    return object;
}

/* getObjectId
 *
 * Object 의 id 값을 구한다.
 */
unsigned long getObjectId( Object *object )
{
    return object->id;
}

/* setObjectName
 *
 * Object 의 name 을 설정한다.
 */
void setObjectName( Object *object, char *name )
{
    if( object == NULL )
    {
        fprintf( stderr, "object != NULL failed!₩n" );
        return;
    }

    /* 이미 object 의 name 이 설정되어 있었다면 그 이름에 대한
     * 메모리를 먼저 해제한다.
     */
    if( object->name != NULL )
        free( object->name );

    object->name = strdup( name );
}

/* getObjectName
```

```
 *
 * Object 의  name 을  얻어낸다.
 */
char *getObjectName( Object *object )
{
	return object->name == NULL ? "NoName" : object->name;
}

/* destroyObject
 *
 * Object 를  제거한다.
 */
void destroyObject( Object *object )
{
	if( object == NULL )
	{
		fprintf( stderr, "object != NULL failed!₩n" );
		return;
	}

	if( object->name != NULL )
		free( object->name );

	free( object );
}
#include <stdio.h>
#include "object.h"

int main(void)
{
	Object *object1 = NULL;
	Object *object2 = NULL;
	Object *object3 = NULL;

	object1 = createObjectWithName( "object1" );
	object2 = createObjectWithName( "object2" );
	object3 = createObjectWithName( "object3" );

	printf("Id = %d Name = `%s'₩n",
		getObjectId(object1), getObjectName(object1) );
	printf("Id = %d Name = `%s'₩n",
		getObjectId(object2), getObjectName(object2) );
	printf("Id = %d Name = `%s'₩n",
		getObjectId(object3), getObjectName(object3) );

	setObjectName( object1, "Test" );

	printf("Name = `%s'₩n", getObjectName(object1) );

	destroyObject( object1 );
	destroyObject( object2 );
	destroyObject( object3 );
```

```
    return 0;
}
```

위의 소스를 컴파일 하려면 다음과 같이 합니다.

```
/* gcc 로 컴파일 하는 예 */

$ gcc -c -o object.o object.c
$ gcc -c -o main.o main.c
$ gcc -o exam main.o object.o

혹은

$ gcc -o exam main.c object.c
```

생성된 exam 실행파일을 실행하면 다음과 같은 결과가 나옵니다.

```
$ ./exam
Id = 0 Name = `object1'
Id = 1 Name = `object2'
Id = 2 Name = `object3'
Name = `Test'
```

예상한 대로 결과는 잘 나왔습니다. 하지만 컴파일 할 때에 object.c 파일을 같이 계속 컴파일을 해주어야 하는 것이 자꾸 마음에 걸립니다.

object.c 는 라이브러리로 만들수 있는 부분이므로 라이브러리를 만들어보겠습니다.

```
/* gcc 로 컴파일 하는 예 */

$ gcc -c -o object.o object.c    ---> (1)
$ ar r libmyobject.a object.o    ---> (2)
```

- (1은 object.c 파일을 compile only로 컴파일 하여 object.o를 생성.
- (2)는 ar 유틸을 이용해서 object.o 파일을 libmyobject.a라는 라이브러리 생성.

위와 같은 작업을 하면 libmyobject.a 파일이 생성되며 이 파일은 다른 프로그램을 컴파일 할 때에 같이 링크될 수 있는 형태가 되었습니다.

그렇다면 이 libmyobject.a 파일을 같이 링크하여 컴파일 하는 모습을 보겠습니다.

```
/* gcc 로 컴파일 하는 예 */
```

```
$ gcc -c -o main.o main.c
$ gcc -o exam main.o libmyobject.a

혹은

$ gcc -o main.c libmyobject.a
```

이렇게 실제 컴파일은 main.c 만 하고 생성된 main.o 파일과 libmyobject.a 를 같이 링크하여 exam 을 생성하는 것입니다.

생성된 libmyobject.a 이 있으면 object.c 파일이 없다 하더라도 그 내용을 사용할 수 있습니다.

요즘은 Open source 가 많이 대중화 되어서 그런일은 거의 없습니다만 예전 DOS 시절만 해도 라이브러리 소스코드를 공개하는 일이 극히 드물어서 라이브러리를 사용하고 싶을 때에는 이미 컴파일 되어서 제공되는 *.lib 형태의 object 라이브러리와 헤더파일을 사용했었습니다. 즉, 라이브러리화 하면 소스코드 없이 그 함수들을 사용할 수 있게 되는 것이지요.

지금까지 아주 간단하게 분할 컴파일에 대해서 아주 기초적인 것에 대해서 알아봤습니다. 좀 더 시간이 된다면 추후에 좀 더 상세한 분할 컴파일에 대해서 다시 다루기로 하고 기초적인 강좌는 여기서 마치겠습니다.

---

11 장. 포인터 강좌 후기

지금까지 포인터에 관한 내용을 전반적으로 알아보았습니다. 이 강좌는 계속해서 내용이 추가될 예정이며 추가된 내용은 강좌의 제일 첫 페이지에 적힐 것입니다.

포인터의 기본적인 개념에서부터 활용, 그리고 조금 깊은 내용까지 기존의 배열에 대한 상식을 깨어버리고 포인터로 배열을 바라보는 시각을 가질 수 있도록 작성을 해 보았습니다.

포인터는 배열뿐 아니라 C 언어에서 사용되는 모든 자료형과 구조체, 공용체 변수들을 참조할 수 있고 함수조차 참조를 할 수가 있습니다. 따라서 자칫 잘못 사용하게 되면 원하지 않는 결과를 만들기도 하지만 잘 사용을 하게되면 성능이 좋아질 뿐 아니라 힘들게 여겨졌던 일들도 아주 간단히 해결할 수가 있게 됩니다.

포인터는 C 에서 가장 중요한 위치를 차지하는 부분입니다. 이해하기 힘들거나, 실제 사용하기 너무 까다롭다거나 하는 이유로 포인터를 등한시 하게 되면 역시 C power user 가 되기 힘듭니다. 포인터를 자유자재로 사용할 수 있는 날이 올때까지 포인터를 사용한 예제를 자꾸 반복 타이핑 해보고 스스로 종이에 하나하나 그려가며 분석을 하는 습관을 들여야 합니다.

그러한 습관이 자연스레 몸에 배게되면 아무리 복잡한 포인터 연산이라 하더라도 쉽게 주소값을 머리속으로 찾아갈 수 있게되며 그렇게 됨으로써 프로그래밍을 하는 재미도 부가가 될 것입니다.

저 역시 포인터를 공부하면서 힘들었던 생각을 하면서 최대한 초보에서 중급으로 옮겨 갈 수 있도록 강좌를 썼지만 아직도 부족한 점이 많습니다. 시간이 허락하는 한 계속해서 포인터 강좌는 업데이트 될 것이므로 계속해서 관심을 가지고 강좌를 읽어주시기 바랍니다.

부디 포인터라는 길들여지지 않은 야생마를 자신의 명마로 꼭 길들여서 멋진 C 프로그래밍 세상을 마음껏 뛰어다니기를 바라마지 않습니다.

새벽 4 시에 졸린눈 비비며 후기를 적고 있는 돈데크만 심상돈이었습니다.
