10

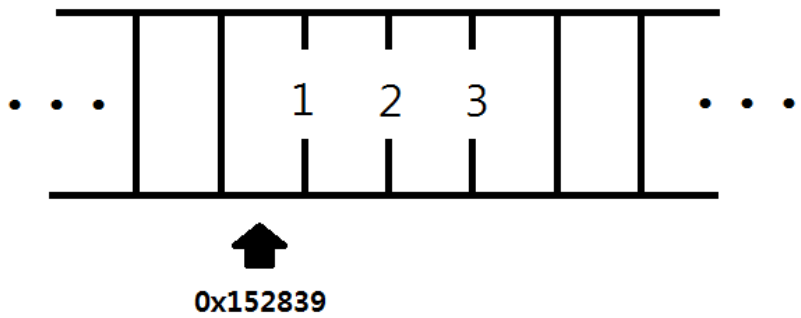
포인터

우왕~ 안녕하세요 여러분. 아마 C 언어를 배웠거나 배우고 있는 사람들은 포인터에 대해 익히 들어보셨을 것 입니다. 이해하기 힘들기로 악명 높은 그 포인터를 말이죠. 하지만, 저와 함께 한다면 큰 무리 없이 배우 실 수 있을 것이라 생각됩니다.

포인터를 이해하기 앞서

앞서 3 강에서 이야기 하였지만 모든 데이터들은 메모리 상에 특정한 공간에 저장 되어 있습니다. 우리는 앞으로 편의를 위해, 메모리의 특정한 공간을 '방' 이라고 하겠습니다. 즉, 각 방에는 데이터들이 들어가게 되는 것 입니다. 보통 사람들은 한 방의 크기를 1 바이트 라고 생각합니다. 우리가 만약 `int` 형 변수를 정의한다면 4 바이트 이므로 메모리 상의 4 칸을 차지하게 됩니다. 그런데 말이죠. 프로그램 작동 시 컴퓨터는 각 방에 있는 데이터를 필요로 하게 됩니다. 따라서, 서로 구분하기 위해 각 방에 고유의 **주소(address)** 를 붙여 주었습니다. 우리가 아파트에서 각 집들을 호수로 구분하는 것 처럼 말입니다. 예를 들어 우리가 아래와 같은 `int` 변수 `a` 를 정의하였다면 특정한 방에 아래 그림 처럼 변수 `a` 가 정의됩니다.

```
int a = 123;  // 메모리 4 칸을 차지하게 한다.
```



이 때, `0x152839` 는 제가 아무렇게나 정한 이 방의 시작 주소 입니다. 참고로, `0x` 가 뭐냐고 물어보는 사람들이 있을 텐데, 이전 강좌에서도 이야기 하였지만 16 진수라고 표시한 것 입니다. 즉, 16 진수로 `152839` (10 진수로 1386553) 라는 위치에서 부터 4 바이트의 공간을 차지하며 123 이라는 값이 저장되어 있게 하라는 뜻이지요.

그렇다면 아래와 같은 문장은 어떻게 수행 될까요?

```
a = 10;
```

사실 컴파일러는 위 문장을 아래와 같이 바꿔주게 됩니다.

메모리 `0x152839` 위치에서 부터 4 바이트의 공간에 있는 데이터를 10 으로 바꾸어라!

결과적으로, 컴퓨터 내부에서는 올바르게 수행되겠지요.

참고적으로 말하는 이야기 이지만 현재 (아마 이 블로그에 접속하는 사람들 중 99% 이상이) 많은 사람들은 32 비트 운영체제를 사용하고 있습니다.[1] 이 32 비트에서 작동되는 컴퓨터들은 모두 주소값의 크기가 32 비트 (즉, 4 바이트.. 까먹었다면 2 - 3 강 참조) 로 나타내집니다. 즉 주소값이 `0x00000000` ~ `0xFFFFFFFF` 까지의 값을 가진다는 것이지요.

어랏! 조금 똑똑하신 분들이라면 32 비트로 사용할 수 있는 주소값의 가지수는 2 의 32 승 바이트, 즉 RAM 은 최대 4 GB 까지 밖에 사용할 수 없다는 사실을 알 수 있습니다. 맞습니다. 이 때문에 32 비트 운영체제에서는 RAM 의 최대 크기가 4 GB 로 제한되지요(즉, 돈을 많이 들여서 RAM 을 10GB 로 만들어도 컴퓨터는 4 GB 까지 밖에 인식하지 못합니다. 어찌 이렇게 슬플수가..)

여기까지는 상당히 직관적이고 단순해서 이해하기 쉬웠을 것 입니다. 그런데 C 를 만든 사람은 아주 유용하면서도 골때리는 것을 하나 새롭게 만들었습니다. 바로 '포인터(pointer)' 입니다. 영어를 잘하는 분들은 이미 아시겠지만 '포인터' 라는 단어의 뜻이 '가리키는 것(가르켜지는 대상체를 말하는 것이 아닙니다)' 이란 의미를 가지고 있습니다.

사실, 포인터는 우리가 앞에서 보았던 `int` 나 `char` 변수들과 다른 것이 전혀 아닙니다. 포인터도 '변수' 입니다. `int` 형 변수가 정수 데이터, `float` 형 변수가 실수 데이터를 보관했던 것 처럼, 포인터도 특정한 데이터를 보관하는 '변수' 입니다. 그렇다면 포인터는 무엇을 보관하고 있을 까요?

바로, 특정한 데이터가 저장된 주소값을 보관하는 변수 입니다. 여기서 강조할 부분은 '주소값' 이라는 것 이지요. 여기서 그냥 머리에 박아 넣어 버립시다. 이전에 다른 책들에서 배운 내용을 싹 다 잊어 버리고 그냥 망치로 때려 넣듯이 박아버려요. 포인터에는 특정한 데이터가 저장된 주소값을 보관하는 변수 라고 말이지요. 크게 외치세요. '주소값!!!!!'

암튼, 뇌가 완전히 세뇌되었다고 생각하면 다음 단계로 넘어가도록 하겠습니다. 아직도 이상한 잡념이 머리에 남아 있다면 크게 숨을 호흡하시고 주소값이라고 10 번만 외쳐 보세요..

자. 되었습니다. 이제 포인터의 세계로 출발해 봅시다. 뿡

1) 참고로 강좌가 처음 제작된 년도가 2009년 입니다. 지금 (2018년 이후) 이 글을 보시는 분들은 아마 대부분 64 비트 운영체제를 사용하고 있을 것입니다.

포인터

다시 한 번 정리하자면

> 포인터 : 메모리 상에 위치한 특정한 데이터의 (시작)주소값을 보관하는 변수

우리가 변수를 정의할 때 `int` 나 `char` 처럼 여러가지 **형(type)** 들이 있었습니다. 그런데 놀랍게도 포인터에서도 형이 있습니다.

이 말은 포인터가 메모리 상의 `int` 형 데이타의 주소값을 저장하는 포인터와, `char` 형 데이터의 주소값을 저장하는 포인터가 서로 다르다는 말입니다. 응?? 여러분의 머리속에는 아래와 같은 생각이 번개 처럼 스쳐 지나갈 것입니다.

> 아까 포인터는 주소값을 저장하는 거래며. 근데 우리가 쓰는 컴퓨터에선 주소값이 무조건 32 비트, 즉 4 바이트래며! 그러면 포인터의 크기는 다 똑같은것 아냐? 근데 왜 포인터가 형(type)을 가지는 건데?!

휴우우. 진정좀 하시고. 여러분 말이 백번 맞습니다 - 단, 현재 까지 배운 내용을 가지고 생각하자면 말이지요. 포인터를 아주 조금만 배우면 왜 포인터에 형(type) 이 필요한지 알게 될 것입니다.

C 언어에서 포인터는 다음과 같이 정의할 수 있습니다 C 언어에서 포인터는 다음과 같이 정의할 수 있습니다.

> (포인터에 주소값이 저장되는 데이터의 형) *(포인터의 이름);

혹은 아래와 같이 정의할 수 도 있습니다.

> (포인터에 주소값이 저장되는 데이터의 형)* (포인터의 이름);

예를 들어 `p` 라는 포인터가 `int` 데이터를 가리키고 싶다고 하면

```
int* p;  // 라고 하거나
int* p;  // 로 하면 된다
```

라 하면 올바르게 됩니다. 즉 위 포인터 `p` 는 `int` 형 데이터의 주소값을 저장하는 변수가 되는 것입니다. 와우!

& 연산자

그런데 말입니다. 아직도 2% 부족합니다. 포인터를 정의하였으면 값을 집어 넣어야 하는데, 도대체 우리가 데이터의 주소값을 어떻게 아냐는 말입니까? 걱정 마십시요. 바로 & 연산자를 사용하면 됩니다.

그런데, 아마 복습을 철저하게 잘하신 분들은 당황할 수 도 있습니다. 왜냐하면 & 가 AND 연산자이기 때문입니다. (4 강 참조) 그런데, & 연산자를 사용하기 위해서는 두 개의 피연산자를 사용해야 합니다. 즉,

```
a& b;  // 괜찮음
a&     // 오류
```

와 같이 언제나 2 개가 필요 하다는 것이지요. 그런데, 여기에서 소개할 & 연산자는 오직 피연산자가 1 개인 연산자 입니다. (이러한 연산자를 **단항(unary)** 연산자라 합니다) 따라서 위의 AND 연산자와 완전히 다르게 해석됩니다.

단항 & 연산자는 피연산자의 주소값을 불러 옵니다. 사용하는 방법은 그냥

```
&/* 주소값을 계산할 데이터 */
```

예를 들어서 어떤 변수 a 의 주소값을 알고 싶다면

```
&a
```

로 쓰면 됩니다!

백설(說)이 불여일행(行). 한 번 프로그램을 짜 봅시다.

```
/* & 연산자 */
#include <stdio.h>
int main() {
  int a;
  a = 2;

  printf("%p \n", &a);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
0x7fff80505b64
```

와 같이 나옵니다. 참고로, 여러분의 컴퓨터에 따라 결과가 다르게 나올 수 도 있습니다. 사실, 저와 정말 인연 이상의 무언가가 있지 않는 이상 전혀 다르게 나올 것 입니다. 더 놀라운 것은 실행할 때 마다 결과가 달라질 것입니다.

2 번째 실행한 것

실행 결과

```
0x7ffe37d03104
```

위와 같이 나오는 이유는 나중에 설명하겠지만 주목할 것은 어떠한 **값**이 출력되었다는 것 입니다.

```
printf("%x \n", &a);
```

위 문장에서 &a 의 값을 16 진수 형태 (%p) 로 출력하라고 명령하였습니다. 근데요. 눈치가 있는 사람이라면 금방 알겠지만 위에서 출력된 결과는 8 바이트 (16 진수로 16 자리)가 아닙니다! (여러분의 컴퓨터는 다를 수 있습니다.) 제가 지금 64 비트 운영체제를 사용하고 있는데도 말이지요!

그렇다면 뭐가 문제인가요? 사실, 문제는 없습니다. 단순히 앞의 0 이 잘린 것 이지요. 주소값은 언제나 8 바이트 크기, 즉 16 진수로 16 자리 인데 앞에 0 이 잘려서 출력이 안된 것일 뿐입니다. 따라서 변수 a 의 주소는 아마도 `0x000007ffe37d03104` 가 될 것입니다.[2)]

아무튼 위 결과를 보면, 적어도 제 컴퓨터 상에선 `int` 변수 a 는 메모리 상에서 `0x7ffe37d03104` 를 시작으로 4 바이트의 공간을 차지하고 있었다는 사실을 알 수 있습니다.

자, 이제 & 연산자를 사용하여 특정한 데이터의 메모리 상의 주소값을 알 수 있다는 사실을 알았으니 배고픈 포인터에게 값을 넣어 봅시다.

```
/* 포인터의 시작 */
#include <stdio.h>
int main() {
  int *p;
  int a;

  p = &a;

  printf("포인터 p 에 들어 있는 값 : %p \n", p);
```

2) 아마 지금(2018년 이후) 강좌를 보고 계시는 분들은 앞에서도 말했듯이 64 비트 시스템을 사용하고 계실 가능성이 높기 때문에, 4 바이트가 아니라 8 바이트 숫자가 출력될 것입니다.

```
  printf("int 변수 a 가 저장된 주소 : %p \n", &a);

  return 0;
}
```

실행해 보면 많은 이들이 예상했던 것 처럼....

실행 결과

```
포인터 p 에 들어 있는 값 : 0x7fff894c8b3c
int 변수 a 가 저장된 주소 : 0x7fff894c8b3c
```

똑같이 나옵니다. 어찌 보면 당연한 일입니다.

```
p = &a;
```

에서 포인터 p 에 a 의 주소를 대입하였기 때문이죠. 참고로, 한 번 정의된 변수의 주소값은 바뀌지 않습니다. 따라서 아래 printf 에서 포인터 p 에 저장된 값과 변수 a 의 주소값이 동일하게 나오게 됩니다. 어때요. 쉽죠?

* 연산자

현재 까지 우리가 배운 바로는 **포인터는 특정한 데이터의 주소값을 보관한다. 이 때 포인터는 주소값을 보관하는 데이터의 형에 * 를 붙임으로써 정의되고, & 연산자로 특정한 데이터의 메모리 상의 주소값을 알아올 수 있다** 였습니다.

& 연산자가 어떠한 데이터의 주소값을 얻어내는 연산자라면 거꾸로 주소값에서 해당 주소값에 대응되는 데이터를 가져오는 연산자가 필요하겠지요? 이 역할은 바로 * 연산자가 수행합니다!

잠깐만. * 연산자는 이미 곱셈 연산자로 사용되고 있지 않나요? 맞습니다. 다만, * 연산자가 피연산자 두 개에 작용할 때만 곱셈 연산자로 해석됩니다. 즉,

```
a * b; // a 와 b 를 곱한다.
a *; // 오류!
*a; // 단항 * 연산자
```

와 같이 해석됩니다.

* 연산자의 역할을 쉽게 풀이하자면

> "나(포인터)를 나에게 저장된 주소값에 위치한 데이터로 생각해줘!"

의 역할을 수행합니다. 한 번 아래 예제를 봅시다.

```c
/* * 연산자의 이용 */
#include <stdio.h>
int main() {
  int *p;
  int a;

  p = &a;
  a = 2;

  printf("a 의 값 : %d \n", a);
  printf("*p 의 값 : %d \n", *p);

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
a 의 값 : 2
*p 의 값 : 2
```

가 됩니다.

```c
int *p;
int a;
```

일단 int 데이터를 가리키는 포인터 p 와 int 변수 a 를 각각 정의하였습니다. 평범한 문장 이지요.

```c
p = &a;
a = 2;
```

그리고 포인터 p 에 a 의 주소를 집어 넣었습니다. 그리고 a 에 2 를 대입하였습니다.

```c
printf("a 의 값 : %d \n", a);
printf("*p 의 값 : %d \n", *p);
```

일단 위의 문장은 단순 합니다. a 의 값을 출력하란 말이지요. 당연하게도 2 가 출력됩니다. 그런데, 아래에서 *p 의 값을 출력하라고 했습니다. * 의 의미는 앞서, 나에 저장된 주소값에 해당하는 데이터로 생각하시오! 로 하게 하는 연산자라고 하였습니다.

따라서 *p 를 통해 p 에 저장된 주소(변수 a 의 주소)에 해당하는 데이터, 즉 변수 a 그 자체를 의미할 수 있게 됩니다.

다시 말해 *p 와 변수 a 는 정확히 동일합니다. 즉, 위 두 문장은 아래 두 문장과 백프로 일치합니다.

```
printf("a 의 값 : %d \n", a);
printf("*p 의 값 : %d \n", a);
```

마지막으로 * 와 관련된 예제 하나를 더 살펴 봅시다.

```
/* * 연산자 */
#include <stdio.h>
int main() {
  int *p;
  int a;

  p = &a;
  *p = 3;

  printf("a 의 값 : %d \n", a);
  printf("*p 의 값 : %d \n", *p);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

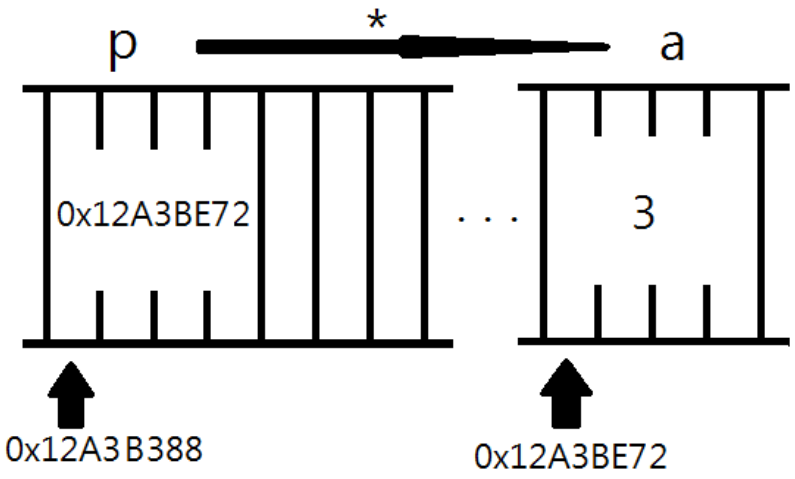
```
a 의 값 : 3
*p 의 값 : 3
```

아마 많은 여러분들이 예상했던 결과 이길 바랍니다!

```
p = &a;
*p = 3;
```

위에서도 마찬가지로 p 에 변수 a 의 주소를 집어 넣었습니다. 그리고 *p 를 통해 "나에 저장된 주소(변수 a 의 주소)에 해당하는 데이터(변수 a) 로 생각하시오" 를 의미하여 *p = 3 은 a = 3 과 동일한 의미를 지니게 되었습니다. 어때요. 간단하지요? 이로써 여러분은 포인터의 50% 이상을 이해하신 것 입니다~! 짝짝짝짝

자. 그럼 **포인터** 라는 말 자체의 의미를 생각해 봅시다. int 변수 a 와 포인터 p 의 메모리 상의 모습을 그리면 아래와 같습니다.



참고로 주소값은 제가 임의로 정한 것 입니다.

포인터 p 에 어떤 변수 a 의 주소값이 저장되어 있다면 **포인터 p 는 변수 a 를 가리킨다** 라고 말합니다. 포인터 또한 엄연한 변수 이기 때문에 특정한 메모리 공간을 차지합니다. 따라서 위 그림과 같이 포인터도 자기 자신만의 주소를 가지고 있습니다.

포인터에는 왜 타입이 있을까

여기 까지 왔다면 아마 다음과 같은 의문을 가질 수 있을 것입니다.

> 포인터가 주소값만 보관하는데 왜 굳이 타입이 필요할까? 어차피 주소값은 32 비트 시스템에서 항상 4 바이트이고, 64 비트 시스템에서는 8 바이트 인데 그냥 pointer 라는 타입을 만들어버리면 안될까?

아주 좋은 질문 입니다. pointer 라는 타입이 있다고 생각해고 아래의 코드를 살펴봅시다.

```
int a;
pointer *p;
p = &a;
*p = 4;
```

컴퓨터 입장에서 위 코드를 어떤 식으로 해석할 지 생각해볼까요.

```
int a;
pointer *p;
p = &a;
```

위 세 문장 까지는 아주 좋습니다. 메모리에 a 를 위해서 4 바이트 짜리 공간을 마련해줬고, 마찬가지로 p 를 위해 메모리 상에 8 바이트 짜리 공간을 마련하였습니다. 그리고 p 에 a 의 주소값을 잘 전달하였지요.

문제는 아래 문장 입니다.

```
*p = 4;
```

포인터 p 에는 명백히 변수 a 의 주소값이 들어 있습니다. 여기서 문제는 a 가 메모리에서 차지하는 모든 주소들의 위치가 들어 있는 것이 아니라 **시작 주소** 만 들어가 있다는 점입니다.

따라서, *p 라고 했을 때 컴퓨터는 메모리에서 얼마만큼을 읽어들어야 할지 알 길이 없습니다.

한편

```
int a;
int *p;
p = &a;
*p = 4;
```

라고 한다면 어떨 까요? 컴퓨터는 포인터 p 가 `int *` 라는 사실을 보고 **이 포인터는 `int` 데이터를 가리키는 구나!** 라고 알게 되어 시작 주소로 부터 정확히 4 바이트를 읽어 들어 값을 바꾸게 됩니다.

포인터도 변수다

```
/* 포인터도 변수이다 */
#include <stdio.h>
int main() {
  int a;
  int b;
  int *p;

  p = &a;
  *p = 2;
  p = &b;
  *p = 4;

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

성공적으로 컴파일 하였다면

실행 결과

```
a : 2
b : 4
```

```
p = &a;
*p = 2;
p = &b;
*p = 4;
```

사실, 이런 예제까지 굳이 보여주어야 하나 하는 생각이 들었지만 그래도 혹시나 하는 마음에 했습니다. 앞에서도 말했듯이 포인터는 **변수** 입니다.

즉, 포인터에 들어간 주소값이 바뀔 수 있다는 것이지요. 위와 같이 처음에 a 를 가리켰다가, (즉 p 에 변수 a 의 주소값이 들어갔다가) 나중에 b 를 가리킬 수 (즉 p 에 변수 b 의 주소값이 들어감) 있다는 것 이지요. 뭐 특별히 중요한 예제는 아니였습니다만. 나중에 상수 포인터, 포인터 상수에 대해 이야기 하면서 다시 다루어 보도록 하겠습니다.

마지막으로, 강의를 마치며 여러분에게 포인터에 대해 완벽히 뇌리에 꽂힐 만한 동화를 들려드리겠습니다.

옛날 옛날에 대략 2 년 전에 (뭐.. 전 여러분과 옛날의 정의가 다릅니다ㅋ) 변철수, 변수철, 포영희라는 세 명의 사람이 OO 아파트에 살고 있었습니다.

```
int chul, sue;
int *young;
```

그런데 말이죠. 포영희는 변철수를 너무나 좋아한 나머지 자기 집 대문 앞에 큰 글씨로 "우리집에 오는 것들은 모두 철수네 주세요" 라고 써 놓고 철수네 주소를 적어 놓았습니다

```
young = &chul;
```

어느날 택배 아저씨가 영희네 집에 물건을 배달하러 왔다가 영희의 메세지를 보고 철수네에 가져다 주게 됩니다.

```
*young = 3; // 사실 chul = 3 과 동일하다!
```

영희에 짝사랑이 계속 되다가 어느날 영희는 철수 보다 더 미남인 수철이를 보게 됩니다. 결국 영희는 마음이 변심하고 수철이를 좋아하기로 했죠. 영희는 자기 대문 앞에 있던 메세지를 떼 버리고 "우리집에 오는 것은 모두 수철이네 주세요." 라 쓰고 수철이네 주소를 적었습니다.

```
young = &sue;
```

며칠이 지나 택배 아저씨는 물건을 배달하러 영희네에 왔다가 메세지를 보고 이번엔 수철이네에 가져다 줍니다.

```
*young = 4; // 사실 sue = 4 와 동일하다
```

이렇게 순수한 사랑이 OO 아파트에서 모락 모락 피어났습니다..... 끝

```
return 0; // 종료를 나타내는 것인데, 아직 몰라도 되요. (정확히 말하면 리턴...)
```

생각해 볼 문제

문제 1

`*` 와 `&` 연산자의 역할이 무엇인지 말해보세요 (난이도 : 下)

문제 2

`int **a;` 와 같은 이중 포인터(double-pointer) 에 대해 생각해 보세요 (난이도 : 中上)

상수 포인터, 포인터의 덧셈 뺄셈, 배열과 포인터

안녕하세요 여러분! 지난 시간에 포인터의 기본 중의 기본이라 할 수 있는 것 들에 배워보았습니다. 다시 정리해 보자면 포인터는 특정한 데이터의 메모리 상의 (시작) 주소값을 보관하는 변수 입니다.

제가 C 언어를 배우면서 포인터를 배울 때 가장 많이 든 생각은

> 근데 말야. 이거왜 배워?

이였습니다. 맞아요. 여러분들도 위와 같은 생각이 머리속에 끊임없이 맴돌 것 입니다. `int a;` 와 `int *p;` 가 있을 때 `p` 가 `a` 를 가리킨다고 하면 `a = 3;` 이라 하지 `*p = 3;` 과 같이 귀찮게 할 필요가 없잖아요. 하지만 나중에 가면 알겠지만 포인터는 C 언어에서 정말로 중요한 역할을 담당하게 될 것입니다. 포인터의 중요한 역할에 대해 지금 이야기 하는 것은 무리라고 생각합니다. 일단, 포인터가 뭔지만 알아 놓고 이걸 도대체 왜 배우는지에 대해선 나중에 이야기 하도록 합시다.

상수 포인터

이전에 11 - 1 강에서 상수에 대해 잠깐 언급한 것이 기억이 나시나요? 그 때 저는 어떠한 데이터를 상수로 만들기 위해 그 앞에 `const` 키워드를 붙여주면 된다고 했습니다. 예를 들어서

```
const int a = 3;
```

과 같이 값이 3 인 `int` 변수 `a` 를 상수로 정의할 수 있습니다. `const` 는 단순히 말해서 '이 데이터의 내용은 절대로 바뀔 수 없다' 라는 의미의 키워드 입니다. 따라서, 위 문장의 의미는 '이 `int` 변수 `a` 의 값은 절대로 바뀌면 안된다!!!' 가 됩니다. 위와 같이 정의한 상수 `a` 를 아래 문장에

```
a = 4;
```

와 같이 하려고 해도 컴파일 시에 오류가 발생하게 됩니다. 왜냐하면 `a` 는 상수로 선언이 되어 있으므로 값이 절대로 변경될 수 없기 때문이죠. 심지어 '값이 변경될 가능성이 있는 문장' 조차 허용되지 않습니다. 예를 들어

```
a = 3;
```

이라고 한다면, a 의 값은 이미 3 이므로 a 의 값은 바뀌지 않습니다. 그런데 왠일? 컴파일 해보면 오류가 출력됩니다. 왜냐하면 위 문장은 a 의 값이 바뀔 '가능성' 이 있기 때문이죠. 즉, 컴파일러는 a 에 무슨 값이 들어가 있는지 신경 쓰지 않습니다. 그냥 무조건 가능성이 있다면 오류를 출력합니다.

여러분은 도대체 왜 상수를 사용하는지 의문을 가질 것 입니다. 하지만 상수는 프로그래밍 상에서 프로그래머들의 실수를 줄여주고, 실수를 했다고 해도 실수를 잡아내는데 중요한 역할을 하고 있습니다. 예를 들어 아래와 같은 문장을 봅시다.

```
const double PI = 3.141592;
```

즉 double 형 변수 PI 를 3.141592 라는 값을 가지게 선언하였습니다. 왜 이렇게 해도 되냐면 실제로 PI 값은 절대로 바뀌지 않는 상수 이기 때문이죠. 따라서, 프로그래머가 밤에 졸면서 코딩을 하다가 아래와 같이

```
PI = 10;
```

PI 의 값을 문장을 집어 넣었다고 해도 컴파일 시 오류가 발생하여 프로그래머는 이를 고칠 수 있게 됩니다. 반면에 PI 를 그냥 double 형 변수로 선언했다고 해봅시다.

```
double PI = 3.141592;
```

그렇다면 프로그래머가 아래와 같은 코드를 잠결에 집어 넣었다면

```
PI = 10;
```

컴파일러는 이를 오류로 처리하지 않습니다. 이는 엄청나게 큰일이 아닐 수 없죠. 만일 고객들에게 원의 넓이를 계산하는 프로그램을 만들어 주었는데 잘못해서 이상한 값이 나오면 어떻겠습니까? 물론 위와 같이 간단한 오류는 잡아내기 쉽지만 프로그램이 커지만 커질 수록 위와 같은 오류를 잡아내는 것은 여간 힘든 일이 아닙니다. 따라서, 우리는 '절대로 바뀌지 않을 것 같은 값에는 무조건 const 키워드를 붙여주는 습관' 을 기르는 것이 중요합니다.

아무튼. 이번에는 포인터에도 const 를 붙일 수 있는지 생각해 봅시다.

```
/* 상수 포인터? */
#include <stdio.h>
int main() {
  int a;
  int b;
  const int* pa = &a;

  *pa = 3;  // 올바르지 않은 문장
```

```
  pa = &b;  // 올바른 문장
  return 0;
}
```

컴파일 해보면 오류가 발생합니다.

컴파일 오류

```
error C2166: l-value가 const 개체를 지정합니다.
```

일단, 위 오류가 왜 발생하였는지에 대해 이야기 하기 앞서서 아래 문장이 무슨 의미를 가지는지 살펴 봅시다.

```
const int* pa =
  &a;  // int* pa 와 같이 정의해도 int *pa 와 같다는 사실은 다 알고 있죠?
```

여러분은 위 문장을 보면 다음과 같은 생각이 떠오를 것입니다. "저 포인터는 const int 형을 가리키는 포인터인데, 어떻게 int 형 변수 a 의 주소값이 대입 될 수 있지? 그러면 안되는 거 아니야?". 하지만, 제가 앞에서 강조해 왔듯이 const 라는 키워드는 **이 데이터의 값은 절대로 바뀌면 안된다** 라고 일러주는 키워드라고 하였습니다.

다시 말해, const int a 라는 변수는 그냥 int 형 변수 a 인데 값이 절대로 바뀌면 안되는 변수일 뿐입니다. 따라서, const int a 변수도 그냥 int 형이라 말할 수 있습니다. (다만 '변'수가 아닐 뿐)

따라서 const int* 의 의미는 const int 형 변수를 가리킨다는 것이 아닙니다. int 형 변수를 가리키는데 **그 값을 절대로 바꾸지 말라** 라는 의미이죠. 즉, pa 는 어떠한 int 형 변수를 가리키고 있습니다. 그런데 const 가 붙었으므로 pa 가가리키는 변수의 값은 절대로 바뀌면 안되게 됩니다.

여기서 *pa* 가 라는 부분을 강조한 이유는 a 자체는 변수 이므로 값이 자유롭게 변경될 수 있기 때문입니다. 하지만 pa 를 통해서 a 를 간접적으로 가리킬 때 에는 컴퓨터가 **아, 내가 const 인 변수를 가리키고 있구나** 로 생각하기 때문에(const int* 로 포인터를 정의하였으므로) 값을 바꿀 수 없게 됩니다.

결과적으로 아래의 문장은 오류를 출력합니다.

```
*pa = 3;  // 올바르지 않은 문장
```

물론 a = 3; 과 같은 문장은 오류를 출력하지 않습니다. 앞에서도 말했듯이 변수 a 자체는 const 가 아니기 때문이죠.

```
pa = &b;  // 올바른 문장
```

그렇다면 위 문장은 옳은 문장 입니다. 왜 일까요? (아마 당연하다고 생각하면 여러분은 훌륭한 학생들 입니다) 이는 아래 예제와 함께 설명하도록 하겠습니다.

```c
/* 상수 포인터? */
#include <stdio.h>
int main() {
  int a;
  int b;
  int* const pa = &a;

  *pa = 3;  // 올바른 문장
  pa = &b;  // 올바르지 않은 문장

  return 0;
}
```

역시 컴파일 해보면

컴파일 오류

```
error C2166: l-value가 const 개체를 지정합니다.
```

앞서 보았던 오류와 동일한 오류가 뜹니다. 그런데 위치가 다릅니다. 앞에서는 위 문장에서 오류가 발생했는데 이번엔 아래에서 발생합니다. 일단, 포인터의 정의 부분 부터 이야기 해봅시다.

```c
int* const pa = &a;
```

차근차근 봐 보면, 우리는 `int*` 를 가리키는 `pa` 라는 포인터를 정의하였습니다. 그런데 이번에는 `const` 키워드가 `int*` 앞에 있는 것이 아니라 `int*` 와 `pa` 사이에 놓이고 있습니다. 뭐지? 하지만 이 것은 `const` 키워드의 의미를 그대로 생각해 보면 간단합니다. `pa` 의 값이 바뀐 안된다는 것이지요.

그런데 제일 처음에 포인터를 배울 때 강조했듯이, 포인터에는 가리키는 데이터의 주소값, 즉 위 경우 `a` 의 주소값이 `pa` 저장되는 것이지요. 따라서, 이 `pa` 가 `const` 라는 의미는 `pa` 의 값이 절대로 바뀔 수 없다는 것인데, `pa` 는 포인터가 가리키는 변수의 주소값이 들어 있으므로 결과적으로 `pa` 가 처음에 가리키는 것 (a) 말고 다른 것은 절대로 건드릴 수 없다는 것 입니다.

```
pa = &b; // 올바르지 않은 문장
```

결론적으로 위 문장은 오류를 뿜게 됩니다. 왜냐하면 `pa` 가 다른 변수를 가리키기 때문이죠 (즉 `pa` 에 저장된 주소값을 바꾸므로) 반면에 위의 예제에서 오류가 났던 문장은 올바르게 돌아갑니다.

```
*pa = 3;  // 올바른 문장
```

왜냐하면 pa 가 가리키는 값을 바꾸면 안된다는 말은 안했기 때문이죠. (그냥 int*)

한 번 위에 나와있던 것을 모두 합쳐 보면

```
/* 상수 포인터? */
#include <stdio.h>
int main() {
  int a;
  int b;
  const int* const pa = &a;

  *pa = 3;  // 올바르지 않은 문장
  pa = &b;  // 올바르지 않은 문장

  return 0;
}
```

와 같이 되겠지요. 어때요? 쉽죠?

포인터의 덧셈

이번에는 포인터의 덧셈과 뺄셈에 대해서 다루어 보도록 하겠습니다. 앞에서도 강조하였지만 지금 하는 작업들이 무의미해 보이고 쓸모 없어 보이지만 나중에 정말로 중요하게 다루어 집니다. 조금만 힘내세요 (아마도 C 언어에서 가장 재미 없는 부분일듯.)

```
/* 포인터의 덧셈 */
#include <stdio.h>
int main() {
  int a;
  int* pa;
  pa = &a;

  printf("pa 의 값 : %p \n", pa);
  printf("(pa + 1) 의 값 : %p \n", pa + 1);

  return 0;
}
```

성공적으로 컴파일 해보면

실행 결과

```
pa 의 값 : 0x7ffd6a32fc4c
(pa + 1) 의 값 : 0x7ffd6a32fc50
```

여러분의 출력 결과는 위에 나온 결과와 다를 수 있습니다. 다만, 두 수의 차이는 4 일 것입니다. (16 진수임에 유의하세요; 50 에서 4c 를 빼면 4!)

아마 여러분은 출력된 결과를 보면서 깜짝 놀랐을 것입니다. 우리는 분명히

```
printf("(pa + 1) 의 값 : %p \n", pa + 1);
```

에서 `pa + 1` 의 값을 출력하라고 명시하였습니다. 제가 앞에서도 이야기 하였듯이 `pa` 에는 자신이 가리키는 변수의 주소값이 들어갑니다. 따라서, `pa + 1` 을 하면 0x7ffd6a32fc4c 에 1 이 더해진 0x7ffd6a32fc4d 가 아니라, 4 가 더해진 0x7ffd6a32fc50 이 출력되었습니다. 이게 도대체 무슨 일입니까? `0x7ffd6a32fc4c + 1 = 0x7ffd6a32fc50` 이라고요?

위 해괴한 계산 결과를 해결하기 앞서, 우리는 포인터의 형이 `int*` 라는 것을 알 수 있었습니다. 그런데 `int` 가 4 바이트 이니까...설마?

일단, 위 추측을 확인해보기 위해 `int` 포인터 말고도 크기가 다른 `char` 이다 `double` 등에 대해서도 해봅시다.

```c
/* 과연? */
#include <stdio.h>
int main() {
  int a;
  char b;
  double c;
  int* pa = &a;
  char* pb = &b;
  double* pc = &c;

  printf("pa 의 값 : %p \n", pa);
  printf("(pa + 1) 의 값 : %p \n", pa + 1);
  printf("pb 의 값 : %p \n", pb);
  printf("(pb + 1) 의 값 : %p \n", pb + 1);
  printf("pc 의 값 : %p \n", pc);
  printf("(pc + 1) 의 값 : %p \n", pc + 1);

  return 0;
}
```

성공적으로 컴파일 후 실행해 보면

실행 결과

```
pa 의 값 : 0x7ffcf64a2e04
(pa + 1) 의 값 : 0x7ffcf64a2e08
pb 의 값 : 0x7ffcf64a2e03
(pb + 1) 의 값 : 0x7ffcf64a2e04
pc 의 값 : 0x7ffcf64a2e08
(pc + 1) 의 값 : 0x7ffcf64a2e10
```

여러분의 출력 결과는 위에 나온 결과와 다를 수 있습니다.

우왕. 우리의 예상과 정확하게 맞아 떨어졌습니다. pb 의 경우 1 이 더해졌고, pc 의 경우 8 이 더해졌습니다. 그런데, char 은 1 바이트, double 은 8 바이트 이므로 모두 우리가 예상한 결과와 일치합니다. 놀랍군요. 하지만 머리 한 켠에는 또다른 의문이 남습니다. 왜 하라는 대로 안하고 포인터가 가리키는 형의 크기 만큼 더할까요. 사실 이에 대한 해답은 뒤에 나옵니다.

훌륭한 학생이라면 여러가지 모험을 해볼 것 입니다. 예를 들어 포인터의 뺄셈은 허용되는지, 포인터끼리 더해도 되는지 등등.. 말이죠. 우리도 한 번 궁금증을 해결해 봅시다.

일단 직관적으로 포인터의 뺄셈은 허용될 것 같습니다. 왜냐하면 뺄셈은 본질적으로 덧셈과 다를 바 없기 때문이죠. (1 - 1 = 1 + (-1)) 아무튼 해 보면 덧셈과 유사한 결과가 나타납니다.

```c
/* 포인터 뺄셈 */
#include <stdio.h>
int main() {
  int a;
  int* pa = &a;

  printf("pa 의 값 : %p \n", pa);
  printf("(pa - 1) 의 값 : %p \n", pa - 1);

  return 0;
}
```

성공적으로 컴파일 후 실행 해보면

실행 결과

```
pa 의 값 : 0x7ffe4f4fa47c
(pa - 1) 의 값 : 0x7ffe4f4fa478
```

여러분의 출력 결과는 위에 나온 결과와 다를 수 있습니다. 역시 우리의 예상대로 4 가 빼졌습니다.

```c
/* 포인터끼리의 덧셈 */
#include <stdio.h>
int main() {
  int a;
  int *pa = &a;
  int b;
  int *pb = &b;
  int *pc = pa + pb;

  return 0;
}
```

아마 컴파일 해보면 아래와 같은 오류를 만날 수 있습니다.

컴파일 오류

```
error C2110: '+' : 두 포인터를 더할 수 없습니다.
```

왜 C 에서는 두 포인터끼리의 덧셈을 허용하지 않는 것일까요? 사실, 포인터끼리의 덧셈은 아무런 의미가 없을 뿐더러 필요 하지도 않습니다. 두 변수의 메모리 주소를 더해서 나오는 값은 이전에 포인터들이 가리키던 두 개의 변수와 아무런 관련이 없는 메모리 속의 임의의 지점 입니다. 아무런 의미가 없는 프로그램 상에 상관없는 지점을 말이죠. 무언가, 설명이 불충분한 느낌이 들지만 아무튼 포인터 끼리의 덧셈은 아무런 의미가 없기 때문에 C 언어에선 수행할 수 없습니다. 그렇다면, 포인터에 정수를 더하는 것은 왜 되는 것일까요. 아까도 말했듯이 이에 대해선 아래에서 설명해드리겠습니다.

그런데 한 가지 놀라운 점은 포인터끼리의 뺄셈은 가능하다는 것입니다. 왜 그런지에 대한 설명은 나중에 합시다.

```c
/* 포인터의 대입 */
#include <stdio.h>
int main() {
  int a;
  int* pa = &a;
  int* pb;

  *pa = 3;
  pb = pa;

  printf("pa 가 가리키고 있는 것 : %d \n", *pa);
  printf("pb 가 가리키고 있는 것 : %d \n", *pb);

  return 0;
}
```

성공적으로 컴파일 해보면

실행 결과

```
pa 가 가리키고 있는 것 : 3
pb 가 가리키고 있는 것 : 3
```

와 같이 나옵니다. 뭐 당연한 일이지요.

```
pb = pa;
```

부분에서 pa 에 저장되어 있는 값 (즉, pa 가 가리키고 있는 변수의 주소값) 을 pb 에 대입하였습니다. 따라서 pb 도 pa 가 가리키던 것의 주소값을 가지게 되는 것이지요. 결과적으로 pb 와 pa 모두 a 를 가리키게 됩니다. 주의해야 될 점은 pa 와 pb 가 형이 같아야 한다는 점 입니다. 다시 말해 pa 가 `int*` 면 pb 도 `int*` 여야 합니다. 만일 형이 다르다면 형변환을 해주어야 하는데 이에 대한 이야기는 나중에 합시다.

배열과 포인터

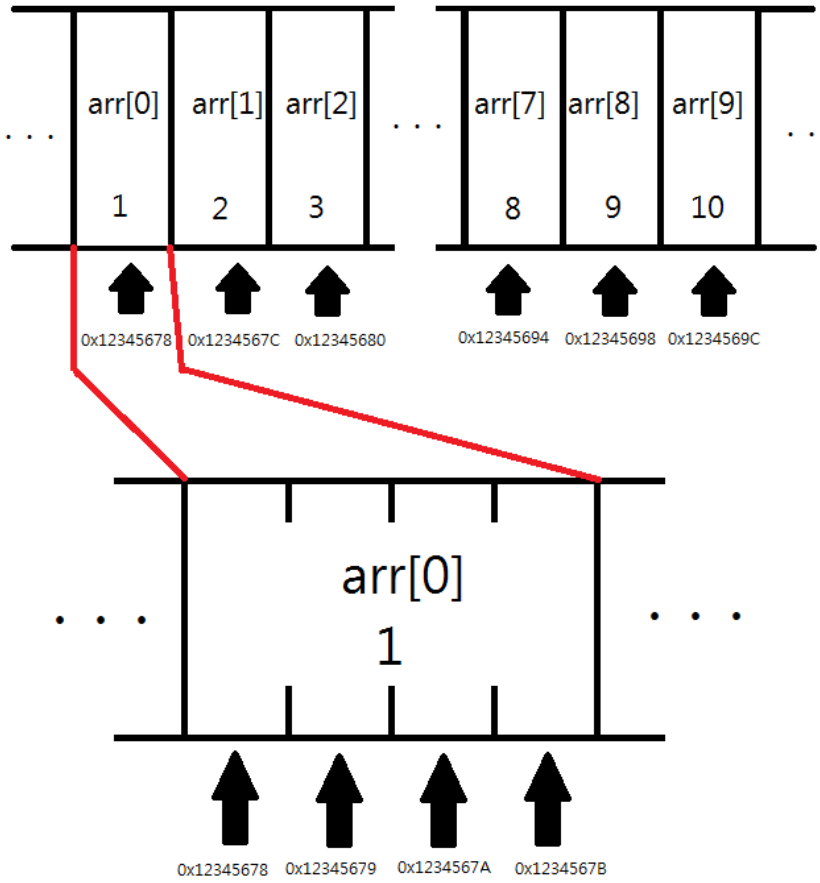
아마 이 단원을 읽다 보면 쇼크를 받을 지도 모르므로 심장이 약하신 분들은 의사와 함께 하십시오.(참고로 저의 경우 많이 놀라서 잠을 잘 못잤습니다)

제가 C 언어를 배우면서 가장 감탄하고도 쇼킹했던 부분이 바로 여기였습니다. 물론, 모든 사람들이 그다지 놀라워 하는 것은 아니지만 저한테는 신선한 충격이였습니다. 아마 이 단원을 배운다면 앞서 '포인터의 연산은 왜 이따구로 하는 거야!' 에 대한 답안을 찾을 수 있을 것 입니다.

이전 강좌에서 (11 강) 저는 여러분에게 배열에 대해 이야기 했었습니다. 기억을 상기해보자면, 배열은 **변수가 여러개 모인 것으로 생각할 수 있다** 라고 이야기 했었지요. 그런데 말이죠. 또다른 놀라운 특징이 있습니다. 바로 배열들의 각 원소는 메모리 상에 연속되게 놓인 다는 점입니다. 뭐, 놀랍지 않다면 말고요. 어쨋든,

```
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

이라는 배열을 정의한다면 메모리 상에서 다음과 같이 나타납니다.



즉, 위와 같이 메모리 상에 연속된 형태로 나타난다는 점이지요. 한 개의 원소는 `int` 형 변수이기 때문에 4 바이트를 차지하게 됩니다. 물론, 위 사실을 믿지 못하시는 분들은 아래와 같이 컴퓨터를 통해 직접 확인해 볼 수 있습니다.

```c
/* 배열의 존재 상태? */
#include <stdio.h>
int main() {
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int i;

  for (i = 0; i < 10; i++) {
    printf("arr[%d] 의 주소값 : %p \n", i, &arr[i]);
  }
  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
arr[0] 의 주소값 : 0x7ffeb5683890
arr[1] 의 주소값 : 0x7ffeb5683894
arr[2] 의 주소값 : 0x7ffeb5683898
arr[3] 의 주소값 : 0x7ffeb568389c
arr[4] 의 주소값 : 0x7ffeb56838a0
arr[5] 의 주소값 : 0x7ffeb56838a4
arr[6] 의 주소값 : 0x7ffeb56838a8
arr[7] 의 주소값 : 0x7ffeb56838ac
arr[8] 의 주소값 : 0x7ffeb56838b0
arr[9] 의 주소값 : 0x7ffeb56838b4
```

와 같이 나타납니다. 여러분의 결과와 주소값은 약간 다를 수 있지만, 어쨋든 4 씩 증가하면 된 것입니다.

아마 여기쯤 왔다면 여러분의 머리를 스쳐지나가는 생각이 들 것입니다! 아! 포인터로도 배열의 원소에 쉽게 접근이 가능하겠구나! (이 생각이 떠오르지 않는 사람은 아마 이 글을 다시 처음부터 읽으셔야 합니다.) 배열의 시작 부분을 가리키는 포인터를 정의한 뒤에 포인터에 1 을 더하면 그 다음 원소를 가리키겠군! 그리고 2 를 더한 그 다음 다음 원소를 가리킨다!!

위와 같은 일이 가능한 이유는 포인터는 자신이 가리키는 데이타의 '형' 의 크기를 곱한 만큼 덧셈을 수행하기 때문이죠. 즉 `p` 라는 포인터가 `int a;` 를 가리킨다면 `p + 1` 을 할 때 `p` 의 주소값에 사실은 `1*4` 가 더해지고, `p + 3` 을 하면 `p` 의 주소값에 `3 * 4` 인 12 가 더해진다는 것입니다.

한 번 이 아이디어를 적용시켜서 배열의 원소를 가리키는 포인터를 만들어봅시다.

```c
/* 과연? */
#include <stdio.h>
int main() {
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int* parr;
  int i;
  parr = &arr[0];

  for (i = 0; i < 10; i++) {
    printf("arr[%d] 의 주소값 : %p ", i, &arr[i]);
    printf("(parr + %d) 의 값 : %p ", i, (parr + i));

    if (&arr[i] == (parr + i)) {
      /* 만일 (parr + i) 가 성공적으로 arr[i] 를 가리킨다면 */
      printf(" --> 일치 \n");
    } else {
      printf("--> 불일치\n");
    }
  }
```

```
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
arr[0] 의 주소값 : 0x7ffedbe31530 (parr + 0) 의 값 : 0x7ffedbe31530
 ↪  --> 일치
arr[1] 의 주소값 : 0x7ffedbe31534 (parr + 1) 의 값 : 0x7ffedbe31534
 ↪  --> 일치
arr[2] 의 주소값 : 0x7ffedbe31538 (parr + 2) 의 값 : 0x7ffedbe31538
 ↪  --> 일치
arr[3] 의 주소값 : 0x7ffedbe3153c (parr + 3) 의 값 : 0x7ffedbe3153c
 ↪  --> 일치
arr[4] 의 주소값 : 0x7ffedbe31540 (parr + 4) 의 값 : 0x7ffedbe31540
 ↪  --> 일치
arr[5] 의 주소값 : 0x7ffedbe31544 (parr + 5) 의 값 : 0x7ffedbe31544
 ↪  --> 일치
arr[6] 의 주소값 : 0x7ffedbe31548 (parr + 6) 의 값 : 0x7ffedbe31548
 ↪  --> 일치
arr[7] 의 주소값 : 0x7ffedbe3154c (parr + 7) 의 값 : 0x7ffedbe3154c
 ↪  --> 일치
arr[8] 의 주소값 : 0x7ffedbe31550 (parr + 8) 의 값 : 0x7ffedbe31550
 ↪  --> 일치
arr[9] 의 주소값 : 0x7ffedbe31554 (parr + 9) 의 값 : 0x7ffedbe31554
 ↪  --> 일치
```

정확히 모두 일치가 나옵니다. 위 소스코드가 이해가 안되는 분들이 있을 까봐 살짝 설명을 드리기는 하겠습니다.

```
parr = &arr[0];
```

`parr` 이라는 `int` 형을 가리키는 포인터는 `arr[0]` 이라는 `int` 형 변수를 가리킵니다. (배열의 각 원소는 하나의 변수로 생각할 수 있다는 사실은 까먹지 않았죠?)

```
printf("arr[%d] 의 주소값 : %p ", i, &arr[i]);
printf("(parr + %d) 의 값 : %p ", i, (parr + i));
```

이제, arr[i] 의 주소값과 (parr + i) 의 값을 출력해봅니다. 만일 parr + i 의 값이 arr[i] 의 주소값과 같다면 하단의 if-else 에서 일치가 출력되고 다르다면 불일치가 출력되게 됩니다. 그런데, 이미 예상하고 있던 바이지만 parr 이 int 형이므로 + i 를 하면 주소값에는 사실상 4*i 가 더해지게 되는 것이지요. 이 때 arr[i] 의 주소값도 i 가 하나씩 커질 때 마다 4 씩 증가하므로 (int 형 배열이므로) 결과적으로 모든 결과가 일치하게 되는 것 입니다.

이렇게 포인터에 정수를 더하는 것 만으로도 배열의 각 원소를 가리킬 수 있습니다. 그렇다면 * 를 이용하여 원소들과 똑같은 역할을 할 수 있게 되겠군요. 마치 아래 예제 처럼 말이지요.

```c
/* 우왕 */
#include <stdio.h>
int main() {
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int* parr;

  parr = &arr[0];

  printf("arr[3] = %d , *(parr + 3) = %d \n", arr[3], *(parr + 3));
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
arr[3] = 4 , *(parr + 3) = 4
```

와 같이 동일하게 접근할 수 있게 됩니다.

즉 parr + 3 을 수행하면, arr[3] 의 주소값이 되고, 거기에 * 를 붙여주면 * 의 연산자의 역할이 '그 주소값에 해당하는 데이터를 의미해라!' 라는 뜻이므로 *(parr + 3) 은 arr[3] 과 동일하게 된다는 것입니다. 어때요? 놀랍지요. 포인터의 덧셈이 왜 그렇게 수행되는지 속 시원하게 해결되는 것 같나요?

배열의 이름의 비밀

아마 여러분들 중 대다수는 배열을 처음 배울 때 다음과 같은 실수를 하신 경험이 있을 것 입니다. (나만 그런가?)

```c
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};
```

```
  printf("%d", arr);
}
```

그러곤 1 도, 2 도, 3 도, 아닌 이상한 값이 나오는 것을 보고 당황하셨겠죠. 그런데, 놀랍게도 그 때 출력되는 값은 아래와 같습니다.

```c
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};

  printf("arr 의 정체 : %p \n", arr);
  printf("arr[0] 의 주소값 : %p \n", &arr[0]);

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
arr 의 정체 : 0x7fff1e868b1c
arr[0] 의 주소값 : 0x7fff1e868b1c
```

와 같이 나옵니다. 와우! 놀랍게도 `arr` 과 `arr[0]` 의 주소값이 동일합니다.

따라서 배열에서 배열의 이름은 배열의 첫 번째 원소의 주소값을 나타내고 있다는 사실을 알 수 있습니다. 그렇다면 배열의 이름이 배열의 첫 번째 원소를 가리키는 포인터라고 할 수 있을까요? **아닙니다!**

주의 사항

이 부분은 (저를 포함한) 많은 사람들이 헷갈렷던 부분들 중 하나 입니다. 포인터를 갓 배운 상태에서 읽어보면 이해가 잘 가지 않을 수 도 있으니, 나중에 포인터와 조금 친숙해진다면 꼭 다시 읽어보는 것을 추천합니다.

배열은 배열이고 포인터는 포인터이다.

예를 들어서 다음과 같이 `sizeof` 를 사용하는 코드를 살펴봅시다. 기억을 상기해보자면 `sizeof` 는 크기를 알려주는 연산자 입니다.

```c
#include <stdio.h>
int main() {
```

```
  int arr[6] = {1, 2, 3, 4, 5, 6};
  int* parr = arr;

  printf("Sizeof(arr) : %d \n", sizeof(arr));
  printf("Sizeof(parr) : %d \n", sizeof(parr));
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Sizeof(arr) : 24
Sizeof(parr) : 8
```

와 같이 나옵니다. 재미 있게도

```
  printf("Sizeof(arr) : %d \n", sizeof(arr));
```

sizeof 를 arr 자체에 그대로 썼을 경우 배열의 **실제 크기** 가 나옵니다. 우리의 arr 배열에는 int 원소 6 개가 있으므로 크기가 24 가 되겠지요. 반면에 parr 에 sizeof 연산자를 사용하였을 경우

```
  printf("Sizeof(parr) : %d \n", sizeof(parr));
```

배열의 자체의 크기가 아니라 그냥 포인터의 크기를 알려줍니다 (64 비트 컴퓨터 이므로 출력된 것 처럼 8 바이트 겠지요).

따라서 배열의 이름과, 첫 번째 원소의 주소값은 엄밀히 **다른 것** 인 것입니다. 그렇다면 도대체 왜 두 값을 출력 했을 때 같은 값이 나왔을까요?

그 이유는 C 언어 상에서 배열의 이름이 sizeof 연산자나 주소값 연산자(&)와 사용될 때 (예를 들어 &arr) 경우를 빼면, **배열의 이름을 사용시 암묵적으로 첫 번째 원소를 가리키는 포인터로 타입 변환되기 때문입니다.**

그렇다면 이제 왜 아래 코드에서 배열의 시작 원소의 주소값이 나왔는지 이해가 가시나요?

```
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};

  printf("arr 의 정체 : %p \n", arr);
  printf("arr[0] 의 주소값 : %p \n", &arr[0]);
```

```
  return 0;
}
```

arr 이 sizeof 랑도, 주소값 연산자랑도 사용되지 않았기에, arr 은 첫 번째 원소를 가리키는 포인터로 타입 변환되었기에, &arr[0] 와 일치하게 됩니다.

[] 연산자의 역할

여러분들 중에서 많은 분들은 [] 가 연산자였다는 사실을 보고 깜짝 놀랐을 것 입니다. 그런데, 4강에서 연산 순위에 대해 이야기 하였을 때 눈썰미가 좋으신 분들은 [] 가 연산자로 나와있음을 보셨을 것입니다.

1	( ) [ ] -> .	왼쪽 우선
2	! ~ ++ -- + -(부호) *(포인터) & sizeof 캐스트	오른쪽 우선
3	*(곱셈) / %	왼쪽 우선
4	+ -(덧셈, 뺄셈)	왼쪽 우선
5	<< >>	왼쪽 우선
6	< <= > >=	왼쪽 우선
7	== !=	왼쪽 우선
8	&	왼쪽 우선
9	^	왼쪽 우선
10	|	왼쪽 우선
11	&&	왼쪽 우선
12	||	왼쪽 우선
13	? :	오른쪽 우선
14	= 복합대입	오른쪽 우선
15	,	왼쪽 우선

www.winapi.com 에서 가져온 자료 입니다.

그런데, 우리는 앞서 포인터 연산이 어떻게 돌아가는지 배웠기 때문에 [] 연산자의 역할을 대충 짐작할 수 있습니다.

```
/* [] 연산자 */
#include <stdio.h>
int main() {
  int arr[5] = {1, 2, 3, 4, 5};

  printf("a[3] : %d \n", arr[3]);
  printf("*(a+3) : %d \n", *(arr + 3));
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
a[3] : 4
*(a+3) : 4
```

음... 이미 앞에서 다룬 내용을 모두 이해했더라면 위 정도쯤은 쉽게 이해할 수 있을 것입니다. 사실 컴퓨터는 C 에서 [] 라는 연산자가 쓰이면 자동적으로 위 처럼 형태로 바꾸어서 처리하게 됩니다. 즉, 우리가 arr[3] 이라 사용한 것은 사실 *(arr + 3) 으로 바뀌어서 처리가 된다는 뜻이지요.

그리고 arr 은 + 연산자와 사용되기 때문에 앞서 말했듯이 **첫 번째 원소를 가리키는 포인터** 로 변환되어서 arr + 3 이 포인터 덧셈을 수행하게 됩니다. 그리고 이는 배열의 4 번째 원소를 가리키게 되겠지요.

따라서 다음과 같이 신기한 연산도 가능합니다.

```c
/* 신기한 [] 사용 */
#include <stdio.h>
int main() {
  int arr[5] = {1, 2, 3, 4, 5};

  printf("3[arr] : %d \n", 3 [arr]);
  printf("*(3+a) : %d \n", *(arr + 3));
  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
3[arr] : 4
*(3+a) : 4
```

3[arr] 은 무언가 조금 이상한 표현 입니다. 사실 이렇게 사용한다면 가독성도 떨어지고 한 번에 이해도 되지 않기에 대부분의 프로그래머들은 arr[3] 으로 사용할 것입니다. 하지만, 앞에서도 [] 는 연산자로 3[arr] 을 *(3+arr) 로 바꿔주기 때문에 arr[3] 과 동일한 결과를 출력할 수 있게 되지요.

포인터의 정의

앞에서 말하기를 int 를 가리키는 포인터를 정의하기 위해 다음의 두 문장을 모두 사용할 수 있다고 했습니다.

```
int* p;
int *p;
```

그런데 말이죠. 제 강좌 말도 다른 곳에서 C 언어를 공부했던 사람들이라면 아래와 같은 형식을 훨씬 많이 쓴다는 사실을 알 수 있었을 것입니다.

```
int *p;
```

왜 일까요? 우리가 int 형 변수를 여러개 한 번에 선언하려 했을 때 int a,b,c,d; 라 하잖아요. 포인터 변수를 여러개 선언 하려면 아래와 같이 해야 합니다.

```
int *p, *q, *r;
```

물론

```
int *p, *q, *r;
```

게 해도 됩니다. 다만,

```
int* p;
```

꼴로 한다면 다음과 같이 실수 할 확률이 매우 커지게 됩니다. 왜냐하면 아래와 같이 한다면

```
int *p, q, r;
```

p 만 int 를 가리키는 포인터 이고, q, r 은 평범한 int 형 변수가 됩니다. 따라서, 앞으로 저는 제 강좌에서 모든 포인터들은

```
int *p;
```

꼴로 선언 하도록 하겠습니다.

생각해 볼 문제

문제 1

`int arr[3][3];` 과 같은 배열은 내부적으로 어떻게 처리되는지 생각해보세요 (난이도 : 中)

문제 2

`int* arr[3];` 과 같은 배열이 가지는 의미는 무엇일까요? (난이도 : 中)