C++ Korea 4th Seminar

C++ 프로젝트 ~처음 만난 세계~

외계인만 아는

C++ 타입 파생 규칙

SK Telecom

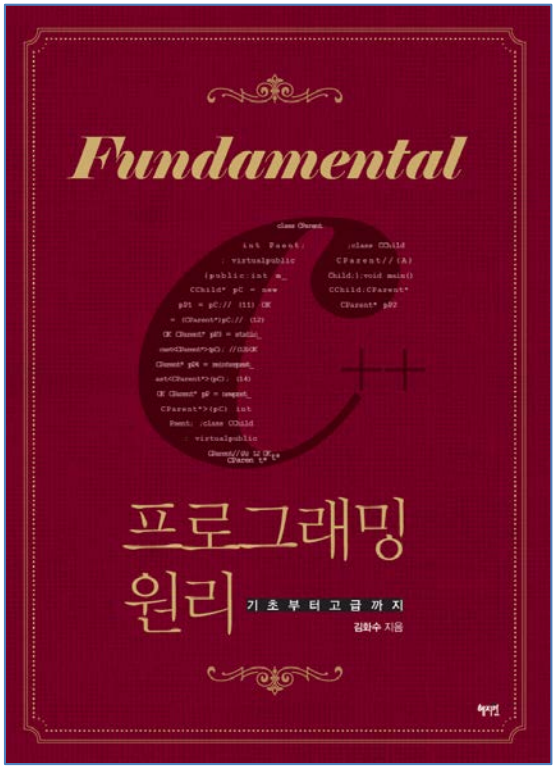
김화수(金花秀)

김화수(金花秀)

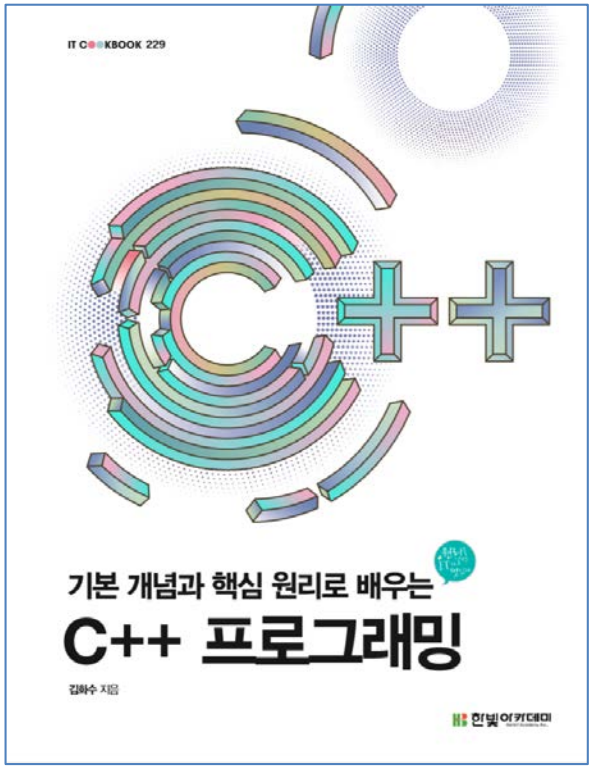
전) Naver NDrive 탐색기 개발

현) SK Telecom CLOUDBERRY 탐색기 개발

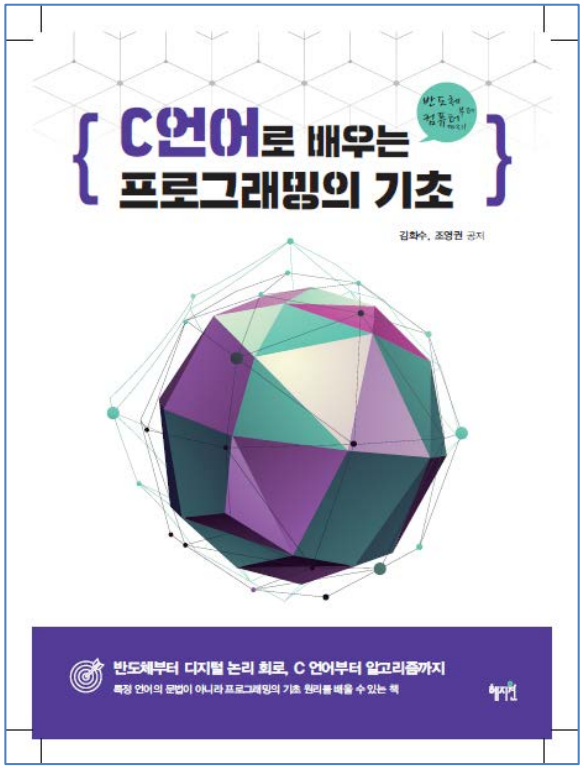
<저서>



(2015, 혜지원)



(2017, 한빛아카데미)



(2018, 혜지원)

간단한 Quiz

```
char ( *Name( int (&)[3] ) )[3];
```

What is the type of Name ?

1) Array
2) Function
3) Pointer
4) Reference

일반적인 반응들...

- 이런 변태 문법은 몰라도 된다.
- 고문받고 있는 외계인이 만든 문법
- 나는 앞으로 C++는 절대 안할건데...
- 이래서 C++가 안되는거야!
- C++ Code에서 이런 변태 문법을 만날 가능성은 1%도 채 안된다.

하지만...

- 늘 1%가 발목을 잡을 때가 있다.
- 먹고 살려면 외계인의 문법도 익혀야 한다.
- **실제 이런 문법이 사용될까?**

실제 사용 예제

배열 요소의 개수를 구해봅시다.

```
void main()
{
    int arr[3];
    cout << ARRAYSIZE(arr) << endl;
}
```

<출력> 3

ARRAYSIZE는 매크로입니다. 어떻게 만들까요?

ARRAYSIZE - 사람 Version

```
#define ARRAYSIZE(A) (sizeof(A) / sizeof((A)[0]))
```

- 하지만 약간의 문제가 있습니다.
- 개발자들이 많이 하는 실수

```
void main()
{
    int* arr = new int[3];
    cout << ARRAYSIZE(arr) << endl;
}
```

<출력>
x86 : 1
x64 : 2

ARRAYSIZE - 외계인 Version

```
template <typename T, size_t N>
char (*RtlpNumberOf(T (&)[N]))[N];

#define ARRAYSIZE(A) (sizeof(*RtlpNumberOf(A)))

void main()
{
    int* arr = new int[3];
    cout << ARRAYSIZE(arr) << endl;
}
```

```
<출력>
error C2672:'RtlpNumberOf': 일치하는 오버로드된 함수가 없습니다.
error C2784:'char (*RtlpNumberOf(__unaligned T (&)[N]))[N]':
            '__unaligned T (&)[N]'의 템플릿 인수를 'int *'에서 추론할 수 없습니다.
```

ARRAYSIZE - 외계인 Version

```
template <typename T, size_t N>
char (*RtlpNumberOf(T (&)[N]))[N];

#define ARRAYSIZE(A) (sizeof(*RtlpNumberOf(A)))
```

```
      T          →  int
      N          →  3
RtlpNumberOf     →  Name

char (*RtlpNumberOf(T (&)[N]))[N];
→ char (*Name(int (&)[3]))[3];
```

왜 우리는 이런 문법을 어려워할까?

- 외계인 문법을 배운 적이 없기 때문에…
- 우리가 책에서 배운 것들은 대략 이런 것들뿐…

```
TYPE var;                  // Variable
TYPE arr[N];               // Array
TYPE* p;                   // Pointer
TYPE& ref;                 // Reference
TYPE func(ARGLIST);        // Function
```

<Right-Left Rule> - 외계인의 가르침

1. 식별자를 찾아라! 'Identifier is'
2. 식별자 오른쪽으로 진행하며 심볼을 찾아본다.
2-1. (ARGLIST)를 만난다면 'function returning'
2-2. [N]을 만난다면 'array of N'
3. 더 이상 심볼을 찾지 못하거나 오른쪽 괄호 ')'를 만나면 왼쪽으로 방향 전환
3-1. *를 만난다면 'pointer to'
3-2. &를 만난다면 'reference to'
3-3. TYPE을 만난다면 'TYPE'
4. 왼쪽 괄호 '('를 만나면 오른쪽으로 방향 전환
5. 왼쪽에서 더 이상 심볼을 찾지 못하면 끝

```
char (*Name(int (&)[3]))[3];
```

Name is function returning pointer to array of 3 char.

그래서

```
char (*Name(int (&)[3]))[3];
```

Name is function returning pointer to array of 3 char

```
template <typename T, size_t N>
char (*RtlpNumberOf(T (&)[N]))[N];
```

⇒ RtlpNumberOf(~)의 반환 타입은

char 요소 N개인 배열을 가리키는 포인터, 따라서

*RtlpNumberOf(~)는 char 요소 N개인 배열을 나타냄.

```
#define ARRAYSIZE(A) (sizeof(*RtlpNumberOf(A)))
```

⇒ char 요소 N개인 배열에 sizeof를 적용하면 N이 도출

⇒ 여기서 A의 타입은 T (&)[N]과 같은 배열 형식만 가능

하지만

<Right-Left Rule>은 주어진 타입 표기를 읽는 것이 주 목적

→ 복잡한 타입은 외계인이 만들테니 인간들은 읽기만 해라!

지금 당장 char 요소 3개인 배열을 가리키는 포인터를 반환하는 함수를 선언하려면 어떻게 해야할까?

→ 숙달된 사람은 <Right-Left Rule>을 역으로 적용하여 만들 수 있지만 결코 쉽지 않다.

그래서 지금부터 타입 파생 규칙의 기본을 배워보겠습니다.

이름 자리[NP]

1. 각 TYPE은 식별자가 놓이는 이름 자리[NP]를 가지고 있다.
2. 기본 타입(char, int, ...)의 이름 자리[NP]는 TYPE 바로 오른쪽이다.
3. 이름 자리[NP]에 식별자를 넣어서 객체를 정의할 수 있다.

```
char [NP]
int [NP]
long [NP]
float [NP]
double [NP]
...
```

1. 배열 타입 파생 1

1. 배열 요소의 타입에서 시작한다.
2. 배열 요소의 수가 n일 때, [n]을 [NP] 바로 뒤(오른쪽)에 붙인다.
3. [NP]를 생략하면 최종 타입을 얻을 수 있다.

Ex) int 요소 3개인 배열

1. int [NP]
2. int [NP][3]
3. int [3]

< 중요 >

이름 자리[NP]는 변하지 않는다.

즉, int [3]의 [NP]는 int와 [3] 사이에 있다.

1. 배열 타입 파생 2

1. 배열 요소의 타입에서 시작한다.
2. 배열 요소의 수가 n일 때, [n]을 [NP] 바로 뒤(오른쪽)에 붙인다.
3. [NP]를 생략하면 최종 타입을 얻을 수 있다.

Ex) int 요소 3개인 배열이 요소로서 2개인 배열(2차원 배열)

1. 1. int [NP]
   2. int [NP][3]
2. int [NP][2][3]
3. int [2][3]

* 왜 int [3][2]로 쓰지 않는지 알 수 있음

2. 포인터 타입 파생 1

1. 포인터의 대상(가리키는) 타입에서 시작한다.
2. 간접 연산자(*)를 [NP] 바로 앞(왼쪽)에 붙인다.
3. 필요한 경우 [NP]와 결합 우선 순위를 높이기 위하여
   간접 연산자(*)와 [NP]를 괄호( )로 묶는다.
   * [NP]의 결합 우선 순위 : 오른쪽 심볼 > 왼쪽 심볼
4. [NP]를 생략하면 최종 타입을 얻을 수 있다.

Ex) int 요소 3개인 배열을 가르키는 포인터

1. 1. int [NP]
   2. int [NP][3]
2. int (*[NP])[3] // int *[NP][3]은 int* 요소 3개인 배열
3. int (*)[3]

2. 포인터 타입 파생 2

```
void main()
{
    int arr[3];
    int* p1 = arr;          // OK
    int* p2 = &arr;         // Error
    int (*p3)[3] = &arr; // OK
}
```

주1) 배열명은 첫 번째 배열 요소를 가리키는 상수 포인터
주2) 배열명에 주소(&) 연산자를 적용하면 배열 자체를
가리키는 상수 포인터

3. 함수 타입 파생 1

1. 함수의 반환 타입에서 시작한다.
2. 인자열(ARGLIST)을 [NP] 바로 뒤(오른쪽)에 붙인다.
3. [NP]를 생략하면 최종 타입을 얻을 수 있다.

Ex) int 요소 3개인 배열을 가리키는 포인터를
반환하는 함수(단, 함수 인자열은 (char))

1. 1. 1. int [NP]
   2. int [NP][3]
2. int (*[NP])[3]

2. int (*[NP](char))[3]

3. int (*(char))[3]

3. 함수 타입 파생 2

```
int arr[3];  // 전역 배열

int (*Func(char arg))[3]
{
    for(int i = 0; i < 3; i++)
    {
        arr[i] = arg;
    }
    return &arr;
}

void main()
{
    int (*p)[3] = Func(7);
    cout << (*p)[0] << (*p)[1] << (*p)[2] << endl;
}
```

<출력> 777

4. 클래스 멤버를 가리키는 포인터 타입 파생 0

일반적으로 포인터란 메모리의 특정 영역을 가리키는 것이지만...

클래스 멤버를 가리키는 포인터는 비정적 멤버 그 자체를 가리킨다.

```
class CTest
{
public:
    int m_Member;          // 비정적 멤버
    static int s_Member; // 정적 멤버
};
int CTest::s_Member;

void main()
{
    CTest t;
    &t.m_Member;          // 메모리 영역
    &CTest::s_Member;   // 메모리 영역
    &CTest::m_Member;   // m_Member 그 자체
}
```

4. 클래스 멤버를 가리키는 포인터 타입 파생 1

```
class CTest
{
public:
    int m_Member1;        // 비정적 멤버 - 오프셋 0
    int m_Member2;        // 비정적 멤버 - 오프셋 4
    static int s_Member; // 정적 멤버
};
int CTest::s_Member;

void main()
{
    CTest t;
    printf("&t.m_Member1: %p\r\n", &t.m_Member1);
    printf("&t.m_Member2: %p\r\n", &t.m_Member2);
    printf("&CTest::s_Member: %p\r\n", &CTest::s_Member);
    printf("&CTest::m_Member1: %p\r\n", &CTest::m_Member1);
    printf("&CTest::m_Member2: %p\r\n", &CTest::m_Member2);
}

&t.m_Member1: 000000CD356FD438
&t.m_Member2 : 000000CD356FD43C
&CTest::s_Member : 00007FF785E0BB00
&CTest::m_Member1 : 0000000000000000
&CTest::m_Member2 : 0000000000000004
```

4. 클래스 멤버를 가리키는 포인터 타입 파생 2

1. 클래스 멤버의 타입에서 시작한다.
2. 클래스 범위 연산자(CLASS::)를 [NP] 바로 앞(왼쪽)에 붙인다.
3. 간접 연산자(*)를 [NP] 바로 앞(왼쪽)에 붙인다.
4. 필요한 경우 [NP]와 결합 우선 순위를 높이기 위하여
   CLASS::*와 [NP]를 괄호( )로 묶는다.
5. [NP]를 생략하면 최종 타입을 얻을 수 있다.

Ex) Class의 멤버인 int 요소 2개인 배열을 가리키는 포인터

```
1. 1. int [NP]
   2. int [NP][2]
2. int Class::[NP][2]      // 범위 연산자 우선 순위 최고
3. int (Class::*[NP])[2]   (X) int Class::(*[NP])[2]
4. int (Class::*)[2]
```

4. 클래스 멤버를 가리키는 포인터 타입 파생 3

```
class CTest
{
public:
    int m_Array[2] = {1, 2}; // 비정적 멤버
    static int (*sp)[2];     // 정적 멤버
};
int (*CTest::sp)[2];         // CTest의 멤버인 포인터 sp

void main()
{
    CTest t;
//  int (CTest::*mp)[2] = &t.m_Array; // Error
    int (CTest::*mp)[2] = &CTest::m_Array;
    cout << (t.*mp)[0] << (t.*mp)[1] << endl;

//  CTest::sp = &CTest::m_Array;  // Error
    CTest::sp = &t.m_Array;
    cout << (*CTest::sp)[0] << (*CTest::sp)[1] << endl;
}
```

5. 종합 실습

Ex) int 요소 3개인 배열이 요소로서 2개인 배열을 가리키는
포인터를 반환하는 Class의 멤버 함수를 가리키는 포인터
단, 함수의 인자열은 (char)

1. 1. 1. 1. 1. int [NP]
2. int [NP][3]
2. int [NP][2][3]
2. int (*[NP])[2][3]
2. int (*[NP](char))[2][3]
2. int (*Class::[NP](char))[2][3]
3. int (*(Class::*[NP])(char))[2][3]
4. int (*(Class::*)(char))[2][3]

객체(변수, 배열, 포인터, 함수 등) 정의

1. 원하는 타입을 파생시킨다.
2. 이름 자리[NP]를 객체 이름으로 변경한다.
3. 문장의 끝일 경우 세미콜론(;)을 붙여준다.

Ex) int 요소 3개인 배열이 요소로서 2개인 배열을 가리키는 포인터를 반환하는 Class의 멤버 함수를 가리키는 포인터 mpf를 정의하자! 단, 함수의 인자열은 (char)

1. `int (*(Class::*[NP])(char))[2][3]`
2. `int (*(Class::*mpf)(char))[2][3]`
3. `int (*(Class::*mpf)(char))[2][3];`

typedef를 이용한 타입 정의 1

1. 원하는 타입을 파생시킨다.
2. 이름 자리[NP]를 새로운 타입 이름으로 변경한다.
3. 앞에 typedef를 붙이고 문장의 끝에 세미콜론(;)을 붙여준다.

Ex) int 요소 3개인 배열이 요소로서 2개인 배열을 가리키는
포인터를 반환하는 Class의 멤버 함수를 가리키는 포인터
타입을 새로운 타입 이름 TMPF로 정의하자!
단, 함수의 인자열은 (char)

1. `int (*(Class::*[NP])(char))[2][3]`
2. `int (*(Class::*TMPF)(char))[2][3]`
3. `typedef int (*(Class::*TMPF)(char))[2][3];`

typedef를 이용한 타입 정의 2

typedef를 이용하여 새롭게 정의된 타입(TYPE)의 이름 자리[NP]는 기본 타입(char, int, ...)처럼 TYPE 바로 오른쪽이다.

```
typedef int (*PARRAY23)[2][3];

void main()
{
    int Array[2][3] = {{1, 2, 3}, {4, 5, 6}};

    int (*pArr1)[2][3] = &Array;
    cout << (*pArr1)[0][0] << (*pArr1)[1][2] << endl;

    PARRAY23 pArr2 = &Array;
    cout << (*pArr2)[0][0] << (*pArr2)[1][2] << endl;
}
```

<출력>
16
16

Q & A

감사합니다.