C 컴파일러 개발 (문법편)

HandyPost는 한 도영(HDNua)이 작성하는 포스트 문서입니다.

1. 개요

우리는 이 문서를 통해, C 프로그래밍 언어가 실제로 어떻게 설계되었는지를 보면서 C 프로그래밍 언어의 문법을 이해할 것이다. 여기서 다루는 C의 문법은 비교적 고급이다. C의 문법에 대해 설명하는 절이지만, 이 내용은 입문용으로는 적절하지 않다.

추가로, 이 문서의 내용 중 많은 부분은 The C Programming Language 2판 번역본에 나와 있는 내용을 많은 부분 참조한 것이다. 이 문서의 내용 중에는 TCPL에서 나온 설명을 그대로 가져다쓰거나, 말을 약간 바꾸어 설명한 것들이 제법 많다. TCPL의 집필자가 이 글을 볼 일은 없을 테지만, 이 점에서는 양해를 바란다. 만약 이 문서가 문제가 된다면, 그건 그 때 가서 생각해야겠다. 이 문서는 이 프로젝트에서 가장 중요해서, 이 내용이 설명되지 않으면 더 이상 프로젝트를 진행할 수 없다.

2. 외부 정의(external declaration)[1)]

컴파일러는 모든 코드를 한 번에 번역하지 않는다. 컴파일러는 번역을 단위 별로 실행하는데, 여기서 컴파일러가 문장을 번역하는 단위를 **번역 단위(translation-unit)**라고 한다. C 컴파일러의 번역 단위는 외부 정의(external declaration)라는 것의 모임으로 이루어져있다. 그럼 외부 정의가 무엇인지부터 알아보자.

외부 정의(external declaration)는 함수 정의(function-definition)와 선언(declaration)을 포함하는 개념이다. The C Programming Language에서는 이를 다음과 같이 표현한다.

```
external-declaration:
        function-definition
        declaration
```

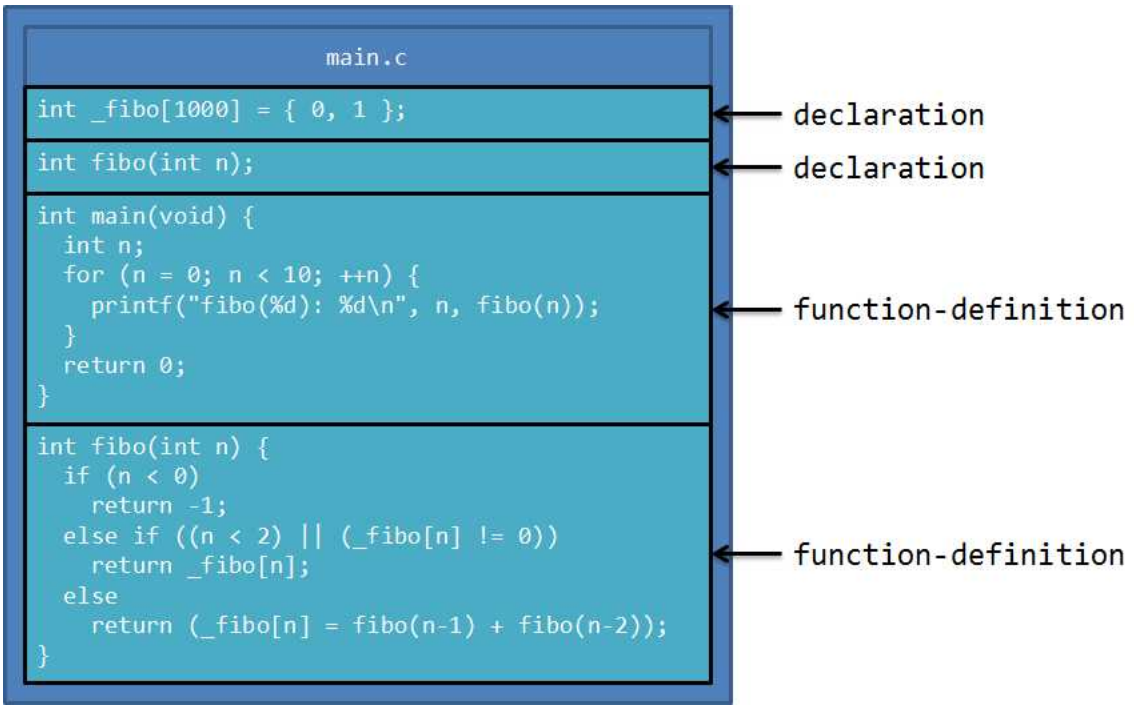
이해를 돕기 위해 예를 들어보겠다. 다음과 같이 피보나치 수를 구하는 C 코드가 있다고 하자.

```
int _fibo[1000] = { 0, 1 };
int fibo(int n);

int main(void) {
    int n;
    for (n = 0; n < 10; ++n) {
        printf("fibo(%d): %d\n", n, fibo(n));
    }
    return 0;
}

int fibo(int n) {
    if (n < 0)
        return -1;
    else if ((n < 2) || (_fibo[n] != 0))
        return _fibo[n];
    else
```

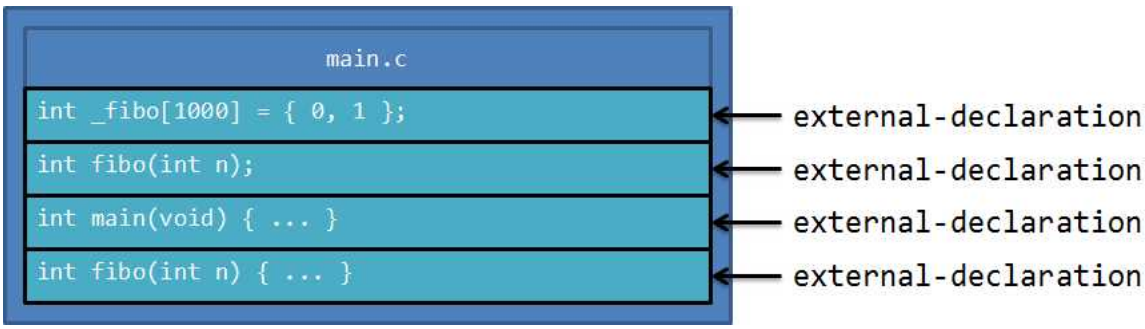
1) declaration은 선언이라는 뜻이지만, The C Programming Language의 번역본을 따랐다.

```
        return (_fibo[n] = fibo(n-1) + fibo(n-2));
}
```

그러면 이 코드는 다음과 같이 분석 가능하다.



결국 C언어로 작성된 코드는 선언과 함수 정의로 나뉘어있다. 그리고 '외부 정의'의 정의에 의해, 선언과 함수는 모두 외부 선언으로 볼 수 있다.



그런데 당장 이것만으로는, 이것이 어디에 사용되는지 감이 덜 잡힌다. 이를 위해 번역 단위를 같은 방식으로 표현해보겠다.

```
translation-unit:
        external-declaration
        translation-unit external-declaration
```

외부 정의의 경우는 그것이 함수 정의와 선언으로 이루어져있다는 사실을 이해할 수 있었다. 그런데 이건 이상하게도, 번역 단위의 정의에 번역 단위가 다시 들어간다! 두 번째 문장은, "번역 단위는 번역 단위와 외부 선언으로 이루어져있다"라고 말하는 것인데, 이것이 어떻게 가능할까?

목표는 번역 단위의 정의를 이용하여, 번역 단위가 외부 정의의 모임이라는 것을 보이는 것이다. 생각해보자. 만약 다음과 같이 A가 정의되어있다면,

```
A:
        B C
D:
        E A F
```

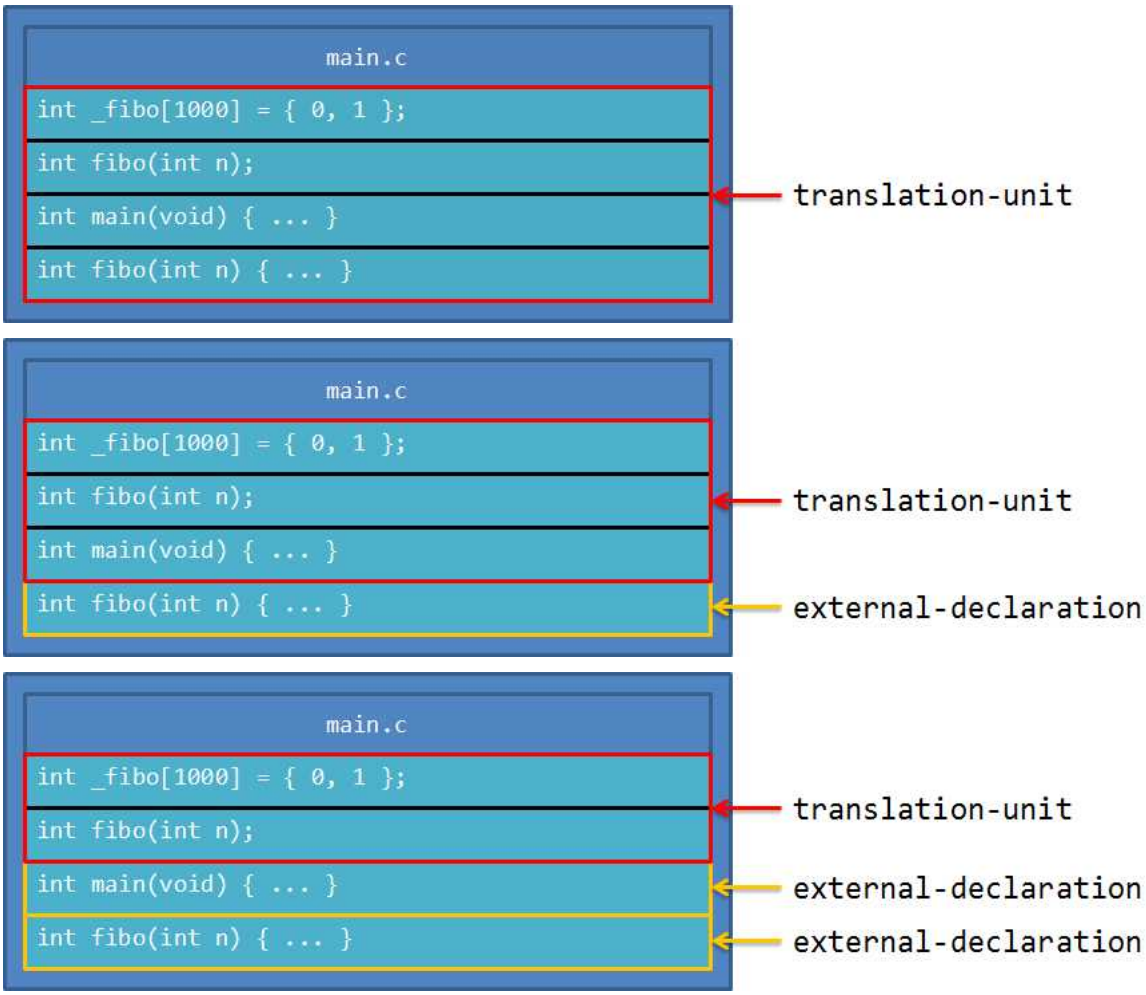
A의 정의에 의해 D를 다음과 같이 대치하는 것이 가능하다.

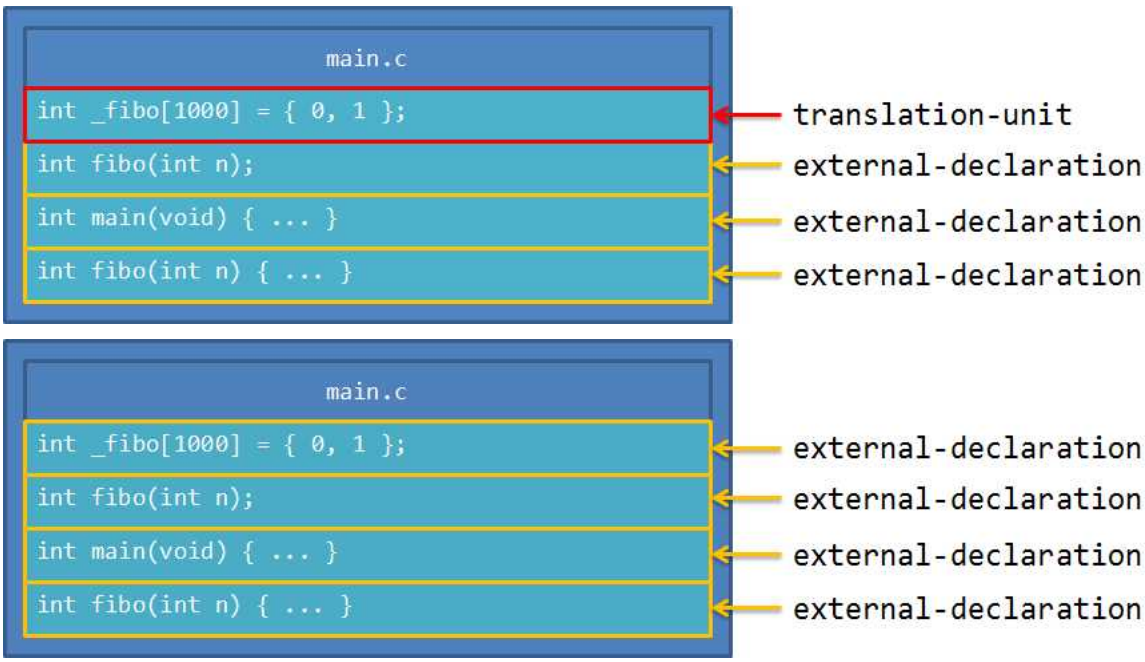
```
D:
        E B C F
```

D가 "E A F"이고 A가 "B C"이니 D는 "E B C F"와 같다는 이야기다. 더 이상 단순할 수가 없다. 그리고 이 방식은 번역 단위에도 똑같이 적용이 된다. 즉 다음은 모두 번역 단위로 볼 수 있다.

```
translation-unit:
        definition 1) external-declaration
        definition 2) translation-unit external-declaration

translation-unit:
        d2+d1) external-declaration external-declaration
        d2+d2+d1) external-declaration external-declaration external-declaration
        d2+d2+d2+d1) ...
```

결국 어떻게 하더라도 번역 단위는 외부 정의의 모임이 된다. 그림으로 표현하면 다음과 같다.





이를 이용하면 우리는 C언어로 작성된 모든 코드를, 다음 한 줄로 분석하는 것이 가능해진다.

```
...
// 번역 단위를 획득하고 어셈블리 언어로의 변환을 수행합니다.
translateUnit(buffer);
...
```

이제 이러한 문법 정의를 납득할 수 있을 것이다.

```
translation-unit:
        external-declaration
        translation-unit external-declaration

external-declaration:
        function-definition
        declaration
```

이 절에서 외부 정의를 먼저 설명하긴 했지만, 실제로는 번역 단위의 정의를 먼저 기술한 다음에 외부 정의의 정의를 기술한다. 이처럼, The C Programming Language에서는 C의 문법을 하향 설계 방식으로 설명하고 있다. 앞으로의 내용도 되도록 하향식으로 설명을 할 것이다.

3. 선언(declaration)

함수 정의보다 상대적으로 쉬운 선언에 대해 먼저 다루자. 사실 우리는 2장에서 C의 선언을 분석하는 dcl 예제를 작성한 경험이 있다. 여기서는 그때보다 본질적으로 C의 선언을 배운다.

선언(declaration)이란, 어떤 대상(object)에 대해 그 형태 및 메모리에 기억하는 방법을 정하기 위해 사용하는 문법을 말한다. 선언은 그 대상에 이름을 부여하지만, 그 이름에 관한 메모리를 확보하지는 않는다. 이는 매우 중요한 내용이다. 많은 사람들은 선언(declaration)과 정의(definition)의 차이를 잘 모르고 있다. 선언은 메모리(**기억 장소(storage)**라고도 한다)를 확보하지 않는 반면, 정의는 기억 장소를 확보한다. 우리가 C언어로 "struct data { int a; };" 라는 문장을 작성했다면, "구조체를 정의했다"가 아니라 "구조체를 선언했다"고 표현해야 옳다.

선언은 다음과 같이 정의된다.

```
declaration:
        declaration-specifiers init-declarator-list_opt ;
```

선언은 선언 지정자(declaration-specifiers)와 초기 선언자 리스트(init-declarator-list), 그리고 마지막의 세미콜론(;)으로 이루어져있다. 여기서 일단 눈여겨볼 것은 초기 선언자 리스트 오른쪽에 조그맣게 쓰여 있는 opt라는 단어다. 이는 해당 요소가 생략 가능함을 나타낸다.

선언의 예를 몇 가지 들어보자.

```
int num; // { decl-spec:"int", init-decl-list:"num" }
const int MAX = 10; // { decl-spec:"const int", init-decl-list:"MAX = 10" }
static unsigned int a, b; // { decl-spec:"static unsigned int", init-decl-list:"a, b" }
typedef int *int_ptr = 0; // { decl-spec:"typedef int", init-decl-list:"*int_ptr = 0" }
```

C에서 가능한 각종 선언을 선언 지정자와 초기 선언자 리스트를 구분하여 보였다. 일단 이 정도면 서로를 구분하는 기준은 감을 잡을 수 있을 것이다. 선언 지정자에 대해서는 잠시 후에 다루기로 하고, 일단 int만 존재하는 것으로 가정하자.

```
declaration-specifiers:
        int
```

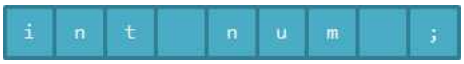
int에는 이탤릭체가 적용되지 않았음에 주목하라. 이는 int는 다른 것으로 대치되지 않음을 의미한다.

3.1) 초기 선언자 리스트(init-declarator-list)

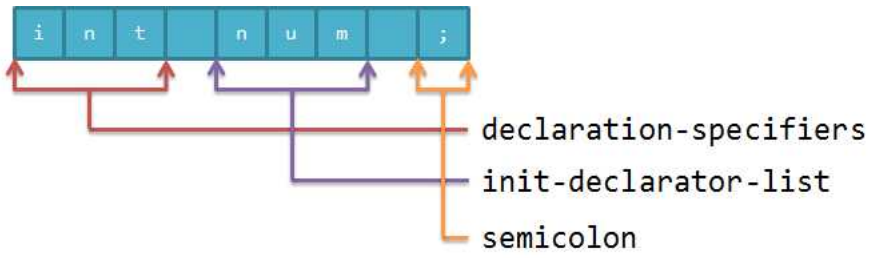
초기 선언자 리스트(init-declarator-list)는 말 그대로, 초기 선언자가 나열되어있는 것을 말한다. 초기 선언자 리스트와 초기 선언자는 다음과 같이 정의된다.

```
init-declarator-list:
        init-declarator
        init-declarator-list, init-declarator

init-declarator:
        declarator
        declarator = initializer
```

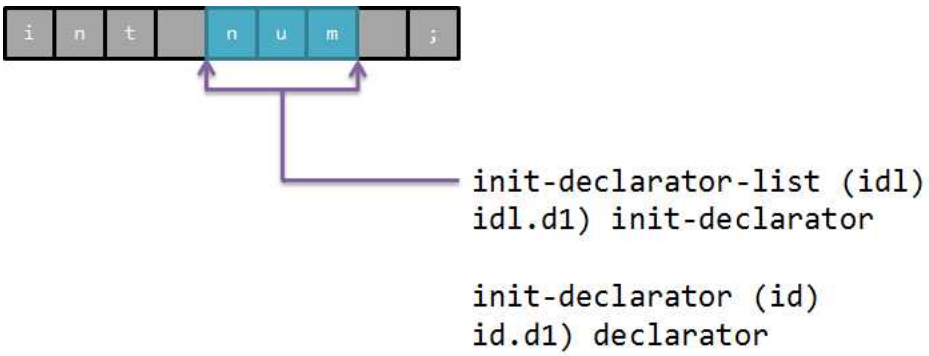
초기 선언자 리스트는 변환 단위와 외부 정의의 관계를 생각하면 어렵지 않게 받아들일 수 있다. 그림을 보면서 이해하자. 다음과 같이 C언어로 작성된 선언이 있다고 하자.



그러면 선언은 다음과 같이 분리할 수 있다.

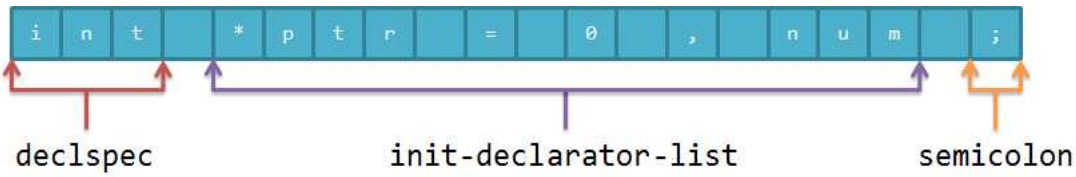


num은 init-declarator-list의 정의에 의해 다음과 같이 분리가 된다.

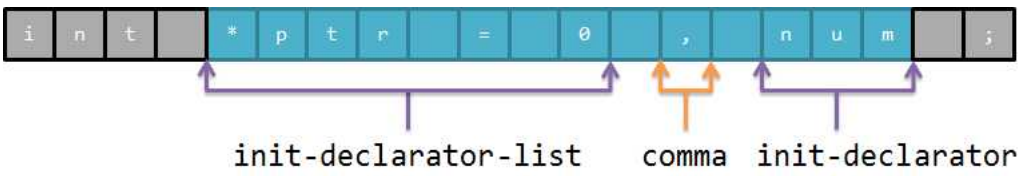


num의 경우는 그 자체로 선언자이기 때문에, 정의에 의해 초기 선언자이기도 하고, 또한 초기 선언자 리스트이기도 하다. 아무래도 이 경우만 보면 헷갈리기 때문에, 다른 선언의 경우도 예를 들어보고자 한다. 다음은 포인터 변수를 0으로 초기화하여 선언하고, 변수 하나를 선언하는 문장이다.

먼저 이것을 선언의 정의대로 구분하면 다음과 같이 된다.



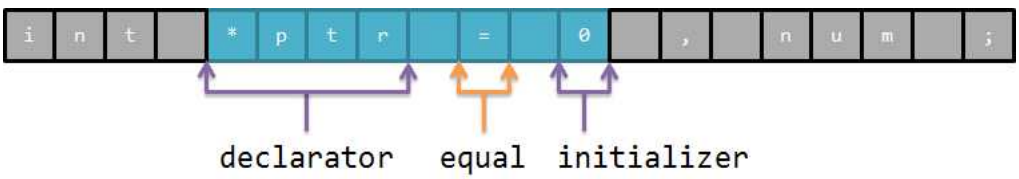
이제 초기 선언자 리스트의 정의대로 분리를 시도하면 다음과 같이 된다.



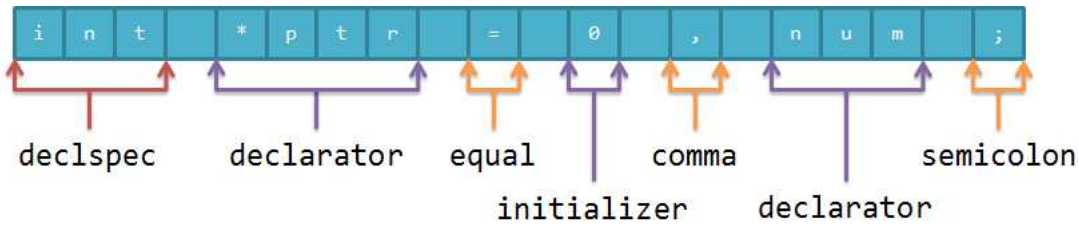
그러면 초기 선언자의 정의에 의해 num은 declarator가 된다.

이제 남은 부분에 대해 다시 초기 선언자 리스트의 정의를 적용하자.

이는 초기 선언자의 정의에 의해 다음과 같이 분리가 된다.



결국 이 선언은 다음과 같다.



이와 같이 초기 선언자 리스트를 이해할 수 있다.

3.2) 선언 지정자(declaration specifier)

선언 지정자(declaration specifier)는 어떤 대상에 대해 그 형태를 지정할 때 사용하는 요소를 말한다. 이는 다음과 같이 정의된다.

```
declaration-specifier:
        storage-class-specifier declaration-specifiersopt
        type-specifier declaration-specifiersopt
        type-qualifier declaration-specifiersopt
```

선언 지정자를 구성하는 요소는 세 가지가 있다. 기억 형태 지정자(storage class specifier), 형식 지정자(type specifier), 형식 한정자(type qualifier)가 그것이다. 이들에 대해 알아보자.

기억 형태(storage class)란 어떤 대상이 기억 장소에 저장되는 형태를 말한다. 따라서 **기억 형태 지정자(storage class specifier)**는 어떤 대상에 대해 기억 형태를 지정하기 위해 사용한다. 기억 형태는 자동 기억 형태(automatic)와 정적 기억 형태(static)의 두 가지로 나뉜다. 자동 기억 형태로 저장되도록 지정된 대상을 **자동 대상(automatic object)**이라고 하고, 정적 기억 형태로 저장되도록 지정된 대상을 **정적 대상(static object)**이라고 한다. 자동 대상은 어떤 한 블록에만 지역적으로 유효하고, 해당 블록을 벗어나면 자동으로 없어지는 반면, 정적 대상은 정의된 이후에는 블록을 벗어나도 그 값을 유지한다.

기억 형태 지정자는 다음과 같이 정의된다.

```
storage-class-specifier:
        auto
        register
        static
        extern
        typedef
```

여기서 중요한 내용이 있다. 선언 지정자의 문법만을 생각한다면, 다음과 같은 구문도 정의에 의해 타당한 선언 지정자로 볼 수 있다.

```
storage-class-specifier storage-class-specifier
```

하지만 다음과 같은 코드는 타당하지 않음이 명백하다.

```
static extern int global;
```

이렇듯 어떤 문장이 문법적으로 올바르지 않다면, 그 원인은 설계 시에 정의한 문법에 어긋나서 발생하는 경우도 있지만, 논리적으로 맞지 않기 때문에 개발자가 특별히 막아놓은 경우도 있다는 것이다. 그래서 필자는 이 두 상황을 다음과 같은 용어로 정의하고자 한다.

- 일반 문법 위반(general syntax error): 설계 시에 정의한 문법을 위반하는 경우
- 특수 문법 위반(special syntax error): 개발자가 특별히 문법 위반으로 간주하는 경우

위에서 타당하지 않다고 예시로 든 코드는 특수 문법 위반에 해당한다. 기억 형태 지정자의 경우, 하나의 선언에 두 가지 이상의 기억 형태 지정자를 넣을 수 없기 때문에, 이러한 경우는 모두 특수 문법 위반으로 처리한다.

기억 형태 지정자가 주어지지 않은 경우는 함수 내에서 선언된 대상은 auto로 간주하고, 함수 내에서 선언된 함수는 extern으로 간주한다.

형식 지정자(type specifier)는 대상의 본질적인 형식을 나타낼 때 사용한다. 종류는 다음과 같다.

```
type-specifier:
        void
        char
        short
        int
        long
        float
        double
        signed
        unsigned
        struct-or-union-specifier
        enum-specifier
        typedef-name
```

기본 형식과 unsigned, signed 등의 조합에 관해서는 이미 알고 있을 것이다. 여기서 다시 한 번 설명하겠다. long이나 short는 int 앞에 붙을 수 있는데, 이 경우 int라는 말은 사용하지 않아도 된다. long의 경우 double과 함께 쓰일 수도 있으며, long long과 같은 표현도 가능하다. signed와 unsigned는 부호를 결정할 때 사용하며, int 또는 short, long이 붙은 int, char와 함께 쓰일 수 있다. 이 둘은 홀로 사용될 수 있는데, 이 경우 int라는 말이 붙어있는 것으로 간주한다. 이외의 경우는 형식 지정자는 하나만 사용 가능하며, 형식 지정자가 없는 경우 int로 간주한다.

마지막으로 **형식 한정자(type qualifier)**는 선언하는 대상의 특별한 성질을 나타낼 때 사용한다. 종류는 다음의 두 가지다.

```
type-qualifier:
        const
        volatile
```

const는 컴파일러에게 해당 대상의 값이 변경되지 않음을 알릴 때, volatile은 컴파일러가 최적화하지 말아야 하는 대상임을 컴파일러에게 알릴 때 사용한다.

3.3) 사용자 정의 자료형

형식 지정자의 마지막 줄에는 구조체, 공용체, 열거형 및 typedef로 정의된 형식과 같은 사용자 정의 자료형에 대해 설명하고 있다. 구조체와 공용체의 문법을 먼저 살펴보자. 이들의 정의는 다음과 같다.

```
struct-or-union-specifier:
        struct-or-union identifier_opt { struct-declaration-list }
        struct-or-union identifier

struct-or-union:
        struct
        union
```

일단 이들 정의에 적응하도록 예시를 들어보자.

```
// type-specifier(struct...}) ;
// struct-or-union identifier { struct-declaration-list }
struct data1 { int a; };
```

```
// storage-class-spec(typedef) type-spec(struct...}) init-decl-list(Data2) ;
// struct-or-union { struct-declaration-list }
typedef struct { int a, b; } Data2;
```

```
// storage-class-spec(typedef) type-spec(union...) init-decl-list(Data3) ;
// struct-or-union identifier
typedef union data3 Data3;
```

이 정도만 이해하면 여기서 소개한 정의가 타당하다는 것을 이해할 수 있을 것이다. 중괄호로 둘러싸인 **구조체 선언 리스트(structure declaration list)**는 이미 알고 있듯, 구조체나 공용체의 멤버를 선언하는 부분을 말한다.

중요한 사실이 있다. 구조체나 공용체를 선언하기 위해 사용하는 이름은 **태그(tag)**라고 부른다. 이것이 왜 중요하냐면, 식별자와 태그는 서로 다르기 때문이다. 예를 들어보자. 동일한 블록에서 식별자는 중복될 수 없다. 즉 한 블록에서 int num; int num;과 같은 문장이 연속해서 나올 수는 없다. 그럼 다음의 선언은 유효한가?

```
typedef struct node { int value; struct node *next; } node;
```

이 선언은 유효하다. 왜냐하면 struct node에서 사용된 node는 식별자가 아니라 태그이기 때문에, 식별자 중복 문제가 발생하지 않기 때문이다. 이 사실은 반드시 이해하고 있어야 한다.

구조체 선언 리스트 및 이를 이루는 요소들에 대한 정의는 다음과 같다.

struct-declaration-list:
        struct-declaration
        struct-declaration-list struct-declaration

struct-declaration:
        specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
        type-specifier specifier-qualifier-list$_{opt}$
        type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
        struct-declarator
        struct-declarator-list, struct-declarator

구조체 선언 리스트의 정의는 이는 번역 단위와 외부 정의의 관계를 생각해보면 명백하다. 구조체 선언은 기본적으로 초기 값이 없는 선언과 그 형태가 거의 같기 때문에, 선언을 이해하고 있다면 이것을 이해하는 것이 크게 부담스럽지 않다. 지정자-한정자 리스트는 선언 지정자의 경우와, 구조체 선언자 리스트는 초기 선언자 리스트의 경우와 아주 비슷하다. 따라서 이들에 대해 별도로 설명하지는 않겠다.

구조체 선언자(structure declarator)는 구조체나 유니온의 멤버에 대한 선언자를 말한다. 구조체의 멤버를 선언할 때, 지정된 수의 비트만 사용하도록 할 수 있는데, 이러한 멤버를 **비트 필드(bit-field)**라고 한다. 구조체 선언자는 일반 선언자와 비트 필드를 포함하여 정의된다.

```
struct-declarator:
        declarator
        declarator_opt : constant-expression
```

비트 필드를 지정할 때 콜론의 오른쪽에는 상수 수식이 와야 한다. **상수 수식(constant expression)**이란 말 그대로 상수인 수식을 말한다. 이는 C언어에서 const 키워드를 통해 값을 정하는 것과는 다른데, 왜냐하면 C의 const는 어떤 식이 상수라는 의미보다는, 값이 변경되지 않음을 나타내는 읽기 전용 값(readonly value)의 성격이 강하기 때문이다. 후에 상수 식과 읽기 전용 값의 차이를 설명하겠다. 일단 구조체와 공용체의 문법은 이 정도로 마무리가 된다.

다음은 열거(enumeration)에 관한 내용이다. 열거는 열거 지정자(enumeration specifier)를 통해 이루어진다. 그 정의는 다음과 같다.

```
enum-specifier:
        enum identifier_opt { enumerator-list }
        enum identifier

enumerator-list:
        enumerator
        enumerator-list, enumerator

enumerator:
        identifier
        identifier = constant-expression
```

이에 대해서도 다시 한 번 설명을 하자. 열거형으로 정의된 이름은 온전히 상수 수식이며, 상수가 사용될 수 있는 곳이면 어느 곳이든 사용될 수 있다. **열거자 리스트(enumerator-list)**는 말 그대로 열거자(enumerator)의 리스트를 말하는데, 열거자에 등호가 있으면 해당 이름의 값은 등호 오른쪽의 상수 식이 되고, 등호가 없는 경우 열거자의 값은 가장 이전에 정의한 열거자의 값에 1을 더한 값이 된다. 가장 이전에 정의한 열거자가 없는 경우 시작 값은 0이 된다. 열거 지정자의 이름은 태그로 간주된다.

마지막으로 typedef-name에 대한 내용이다. typedef-name은 typedef 기억 형태 지정자가 적용된 선언의 결과로 나온 식별자를 말하는데, 이는 다음 한 문장으로 정리가 된다.

```
typedef-name:
        identifier
```

그도 그럴 것이, typedef는 식별자를 형으로 지정하는 역할을 하기 때문이다. typedef에 관해서는, 이 키워드가 적용된 선언은 기억 장소가 메모리가 아니라 컴파일러라는 점만 기억하면 된다.

3.4) 선언자(declarator)

우리는 이미 2장에서 C의 선언 분석 방식에 대해 공부한 적이 있다. 복습도 하고, 이전에 작성했던 코드를 강화하기 위한 준비도 할 겸 선언자에 대해 문법적으로 다시 다뤄보자.

선언자(declarator)란 선언 지정자를 통해 결정한 특성을 바탕으로, 실제 선언을 수행할 때 추가적으로 필요한 정보를 갖고 있는 요소를 말한다. 예를 들면 변수를 선언하려면 변수의 이름은 반드시 필요한데, 바로 변수의 이름이 선언자에 포함된다.

선언자에 대한 문법은 다음과 같다.

```
declarator:
        pointer_opt direct-declarator
```

선언자는 포인터와 직접 선언자로 구성되어있음을 알 수 있다. **직접 선언자(direct declarator)**란 기억 장소에 직접 접근하는 선언자를 이야기한다. 예를 들어 변수 A가 int num;과 같이 선언되었다고 하자. 이 경우 num이 곧 A이기 때문에, num이라는 식별자를 사용하면 변수 A에 직접 접근할 수 있다. 하지만 변수 A를 포인터 변수가 가리키는 상황을 생각하자. int *ptr = #인 경우를 생각하는 것이다. 이 경우 ptr을 통해 A에 접근하려면 갖고 있는 A의 주소 값을 통해 간접적으로 접근해야만 한다. 직접 선언자와 선언자의 차이는 여기서 나타난다. 선언자는 직접 선언자를 포함하는 개념이다. 직접 선언자는 대상에 직접 접근하지만, 선언자라고 하면 대상에 접근하는 방법이 직접적인지, 간접적인지는 상황에 따라 다르다.

다음은 직접 선언자의 정의를 나타낸 것이다.

```
direct-declarator:
        identifier
        ( declarator )
        direct-declarator [ constant-expression_opt ]
        direct-declarator ( parameter-type-list )
        direct-declarator ( identifer-list_opt )
```

직접 선언자의 정의로부터, 직접 선언자가 될 수 있는 조건은 다음과 같음을 알 수 있다.

1) 식별자
2) 괄호로 둘러싸인 선언자
3) 배열 선언자
4) 인자 형식 리스트로 표현되는 함수 선언자
5) 식별자 리스트로 표현되는 함수 선언자

여기서 가장 중요한 것은 2번 정의다. 선언자의 정의엔 직접 선언자가 포함된다. 그런데 직접 선언자의 정의에도 선언자가 포함된다. 결국 서로가 서로를 정의의 일부분으로 사용하고 있는 것이다. 이에 대해 예시를 들어 설명하기 위해, 선언자의 정의에 포함된 포인터에 대해 먼저 이야기해보자.

3.4.1) 포인터

포인터의 정의 및 포인터의 요소에 대한 정의는 다음과 같다.

```
pointer:
        *type-qualifier-list_opt
        *type-qualifier-list_opt pointer

type-qualifier-list:
        type-qualifier
        type-qualifier-list type-qualifier
```

포인터의 요소로 형식 한정자 리스트(type qualifier list)가 존재한다. 형식 한정자는 이전에 학습한 것처럼 const 또는 volatile의 두 종류 밖에 없다. 이 정의를 이용하여, 문법적으로 올바른 포인터를 나열해보면 다음과 같다.

pointer:
	definition 1) **type-qualifier-list*$_{opt}$
	definition 2) **type-qualifier-list*$_{opt}$ *pointer*
type-qualifier-list:
	definition 3) *type-qualifier*
	definition 4) *type-qualifier-list type-qualifier*

pointer:
	d1) *
	d1+d3) * const
	d1+d4) * const volatile
	d2+d1) * *
	d2+d3+d1) * const *
	d2+d2+d1) * * *
	...

이렇게 획득한 포인터들을 선언자의 정의에 대입해보면 이렇게 될 것이다.

declarator:
	pointer$_{opt}$ *direct-declarator*

declarator:
	d1) * *direct-declarator*
	d1+d3) * const *direct-declarator*
	d1+d4) * const volatile *direct-declarator*
	d2+d1) * * *direct-declarator*
	d2+d3+d1) * const * *direct-declarator*
	d2+d2+d1) * * * *direct-declarator*
	...

이들은 모두 선언자이므로, 직접 선언자의 2번 정의에 의해 다음은 모두 직접 선언자가 된다.

direct-declarator:
	d1) (* *direct-declarator*)
	d1+d3) (* const *direct-declarator*)
	d1+d4) (* const volatile *direct-declarator*)
	d2+d1) (* * *direct-declarator*)
	d2+d3+d1) (* const * *direct-declarator*)
	d2+d2+d1) (* * * *direct-declarator*)
	...

그리고 이들은 직접 선언자이기 때문에, 선언자의 정의에 의해 다음은 모두 선언자가 된다.

declarator:
	d1) *pointer*$_{opt}$ (* *direct-declarator*)
	d1+d3) *pointer*$_{opt}$ (* const *direct-declarator*)

```
d1+d4) pointer_opt ( * const volatile direct-declarator )
d2+d1) pointer_opt ( * * direct-declarator )
d2+d3+d1) pointer_opt ( * const * direct-declarator )
d2+d2+d1) pointer_opt ( * * * direct-declarator )
...
```

그리고 이것은 또한 선언자이기 때문에, 직접 선언자의 2번 정의에 의해 이들 모두에 소괄호를 씌운 것 또한 역시 직접 선언자가 된다. 이 과정을 선언자의 분석이 끝날 때까지 진행하면 된다.

3.4.2) 배열 선언자와 함수 선언자

직접 선언자의 정의에 의해, 배열 선언자와 함수 선언자는 직접 선언자다. 정의를 다시 한 번 보자.

```
direct-declarator:
        definition 1) identifier
        definition 2) ( declarator )
        definition 3) direct-declarator [ const-expr_opt ]
        definition 4) direct-declarator ( param-type-list )
        definition 5) direct-declarator ( identifier-list_opt )
```

먼저 배열 선언자의 경우를 보자. 배열 선언자의 정의 앞에 직접 선언자가 포함되어있는 것을 볼 수 있다. 즉 우리는 다음과 같은 식으로 직접 선언자를 확장할 수 있다.

```
direct-declarator:
        d3+d1) identifier [ const-expr_opt ]
        d3+d2) ( declarator ) [ const-expr_opt ]
        d3+d3+d1) identifier [ const-expr_opt ] [ const-expr_opt ]
        d3+d2+d4+d1) ( * identifier ( param-type-list ) ) [ const-expr_opt ]
        ...
```

여기서 특수 문법 위반이 발생할 수 있다. 우리는 배열을 "int arr[][];"과 같이 선언할 수 없다는 사실을 알고 있다. 또한 상수 식이 0보다 큰 정수여야 한다는 사실도 알고 있다. 그래서 이 상황에서도 특수 문법 위반에 대해 처리해주어야 한다.

배열 선언자의 경우는 단순하여 더 볼 것이 없다. 다음은 함수 선언자다. 함수 선언자는 4번, 5번의 정의를 따르는데, 여기에 사용된 매개변수 형식 리스트(parameter type list)와 식별자 리스트(identifier list)의 정의를 알아보자.

먼저, 다음은 매개변수 형식 리스트의 정의를 표현한 것이다.

```
parameter-type-list:
        parameter-list
        parameter-list, ...

parameter-list:
        parameter-declaration
        parameter-list, parameter-declaration

parameter-declaration:
```

declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

매개변수 형식 리스트(parameter type list)는 매개변수 리스트로 구성이 된다. 두 번째 정의인 "..."이 신경쓰이지만, 일단 매개변수 리스트부터 이야기를 진행하자. 매개변수 리스트(parameter list)는 매개변수 선언의 리스트로 구성이 되어있다. 마지막으로 매개변수 선언(parameter declaration)은 우리가 자주 보던 선언을 포함하여, 선언 지정자와 추상 선언자(abstract declarator)라는 이상한 녀석이 붙은 문장으로 정의가 되어있다.

이들은 사실 개념으로 이해하는 것보다 예제로 이해하는 것이 빠르다. 다음은 모두 함수의 선언이다. 반환형은 선언자가 아닌 선언 지정자에 속하기 때문에 이 예제에선 제외되었다. 읽기 정 불편하다면 반환형이 int인 것으로 가정하고 보면 된다.

```
func(void);
> param-type-list: "param-list"
> param-list: "param-decl"
> param-decl: "decl-spec abstract-decl_opt" (void)

sum(int num1, int num2);
> param-type-list: "param-list"
> param-list: "param-decl, param-decl"
> param-decl: "decl-spec decl" (int num1)
> param-decl: "decl-spec decl" (int num2)

qsort(void *, int, int, int (*)(int, int));
> param-type-list: "param-list"
> param-list: "param-decl, param-decl, param-decl, param-decl"
> param-decl: "decl-spec abstract-decl_opt" (void *)
> param-decl: "decl-spec abstract-decl_opt" (int)
> param-decl: "decl-spec abstract-decl_opt" (int)
> param-decl: "decl-spec abstract-decl_opt" (int (*)(int, int))

printf(char *fmt, ...);
> param-type-list: "param-list, ..."
> param-list: "param-decl"
> param-decl: "decl-spec decl" (char *fmt)
```

2.2.1절에서 설명한 분석 요령과 같기 때문에, 적응하면 이해하기 어렵지 않을 것이다. 그리고 여기서도 특수 문법 위반이 발생한다. 매개변수 선언에 선언 지정자(declaration specifier)가 존재하지만, 실제 매개변수에는 기억 형태 지정자(storage class specifier)를 정의할 수 없다. 따라서 함수의 매개변수 선언에서 기억 형태 지정자가 나타난 경우도 우리는 특수 문법 위반으로 처리해야 한다. 추상 선언자는 후에 설명할 것인데, 일단은 불완전한 선언자로 생각하면 된다.

다음은 직접 선언자의 5번 정의에 나타난, 식별자 리스트의 정의를 나타낸 것이다.

identifier-list:
identifier
identifier-list, *identifier*

구형 C에서는 함수 선언을 다음과 같이 했다. 아직도 호환성을 위해 이를 지원하는 컴파일러가 있다.

```
func(a, b, c); // 함수 func는 매개변수로 a, b, c를 받습니다.
```

사실 이는 요즘에 쓰이는 함수 선언 형태가 아니다. 매개변수의 형식을 지정할 수 없어서 쓰기 불편하기 때문이다. 하지만 모든 경우에 선언자를 분석해야 하는 4번 정의와 달리, 이 경우는 반점을 기준으로 식별자만 획득하면 되기 때문에 컴파일러를 구현하는 입장에서는 아주 편리하다. 따라서 우리는 이 둘을 모두 구현할 것이다. 참고로 이런 선언 방식은 숏코딩(short coding)이라고 하는, 코드 줄이기 놀이를 할 때 아주 유용하니 관심이 있다면 찾아보기 바란다.

3.5) 형의 이름(type name)

복잡하게 선언된 대상에 형 변환을 수행할 경우, 또는 함수 선언 시에 매개변수의 형을 선언하는 경우에 대해 알아보자. 예를 들어 다음은 적법한 코드다.

```
int (*cmp)(int, int) = (int (*)(int, int))ascending; // int (*)(int, int)로 형 변환
void qsort(void *, int, int, int (*)(int, int)); // int (*)(int, int) 매개변수 선언
```

그리고 여기에 사용된 형식에 대한 문자열을 형의 이름(type name)이라고 한다. 형의 이름은 다음과 같이 정의된다.

```
type-name:
        specifier-qualifier-list abstract-declarator_opt
```

그리고 이 정의에서 추상 선언자가 다시 등장한다. 이전에 추상 선언자를 불완전한 선언자라고 설명한 바 있다. 보다 정확히 말하자면, **추상 선언자(abstract declarator)**란 식별자가 존재하지 않아서 불완전한 선언자를 말한다. 추상 선언자의 정의는 다음과 같다.

```
abstract-declarator:
        pointer
        pointer_opt direct-abstract-declarator

direct-abstract-declarator:
        ( abstract-declarator )
        direct-abstract-declarator_opt [ constant-expression_opt ]
        direct-abstract-declarator_opt ( parameter-type-list_opt )
```

추상 선언자라고 어려운 게 있는 게 아니라, 그냥 선언자의 정의에서 식별자만 빠져나갔다. 나머지 요소는 선언자 내용을 보면서 충분히 이해할 수 있다. 결국 형의 이름이란 기억 형태 지정자와 식별자가 없는 선언이란 것으로 정리할 수 있다. 이 정도로 형의 이름과 추상 선언자의 설명은 끝이 난다.

이와 같이 C의 선언에 대해 문법적으로 정리할 수 있었다.

4. 함수 정의(function definition)

이제 외부 정의의 나머지 요소인 함수 정의를 살펴보자. 함수 정의는 다음과 같은 형태를 갖는다.

```
function-definition:
        declaration-specifiers_opt declarator declaration-list_opt compound-statement
```

선언 지정자 중에서 허용되는 기억 형태 지정자는 extern 또는 static 뿐이다. 선언자의 경우, 함수 정

의이기 때문에 반드시 다음 구조를 가지게 된다.

```
direct-declarator ( parameter-type-list )
direct-declarator ( identifier-list_opt )
```

그 다음에 있는 선언 리스트(declaration list)는 구형 함수 정의를 위한 것이다. 예를 들어보자. 구형 함수 정의를 이용하여 다음과 같은 식으로 함수를 정의할 수 있다.

```
void swap(vec, idx1, idx2)
int *vec;
int idx1, idx2;
{
    int tmp = vec[idx1];
    vec[idx1] = vec[idx2];
    vec[idx2] = tmp;
}
```

이전에는 이런 식으로, 함수 정의에 식별자 이름을 먼저 쓴 다음 이들의 선언을 후에 추가하여 형식을 맞추었다. 이는 함수 선언에 형식을 넣기 힘들다는 점 때문에 요즘에 사용되는 방법은 아니다. 위는 단지 예시로 든 것뿐이다.

선언 리스트는 다음과 같이 직관적으로 정의된다.

```
declaration-list:
        declaration
        declaration-list declaration
```

선언 리스트를 이용하는 함수 정의는 구형에만 적용되는 것이다. 새 함수 정의에서는 이미 보인 바와 같이 매개변수 형식 리스트를 사용하기 때문에, 이 둘을 섞어서 사용하면 오류가 발생하게 된다.

함수 정의의 마지막에는 복합문(compound statement)이 온다. **복합문(compound statement)**이란 하나의 문장이 쓰일 곳에 여러 개의 문장을 사용할 수 있도록 하기 위해 만든 것이며, 블록(block)이라고도 한다. 복합문의 정의는 다음과 같다.

```
compound-statement:
        { declaration-list_opt statement-list_opt }
```

선언 리스트는 선언의 리스트를 말하는데 이에 대해서는 이미 설명했다. 뒤따라오는 것은 문장 리스트이다. 여기서의 문장이란 if, for와 같은 문장들을 말한다.

우리가 아직 문장이 문법적으로 어떻게 정의되는지는 모르지만, 여기서 C의 중요한 사실이 나타난다. 복합문은 중괄호로 둘러싸여있고, 선언 리스트가 문장 리스트보다 앞서서 나온다. 이것이 중요하다. 선언 리스트가 문장 리스트보다 앞서있기 때문에, 우리는 선언을 블록의 앞부분에 해야 한다. 다시 말해, C에서는 변수 선언을 중간에 할 수 없다. 중간에 변수 선언을 하려면 복합문을 추가한 후 그 복합문의 선언 리스트에서 작업을 진행해야 한다.

복합문의 정의를, 선언 리스트와 문장 리스트를 포함하여 다시 정리하자.

```
compound-statement:
        { declaration-list_opt statement-list_opt }
```

```
declaration-list:
        declaration
        declaration-list declaration

statement-list:
        statement
        statement-list statement
```

결국 추가로 공부해야 하는 건 문장(statement)뿐이다. 나머지는 모두 알고 있는 것들이다.

5. 문장(statement)

문장(statement)이란 우리가 함수 내에서 사용하는 선언 이외의 구문을 말한다. 문장은 다음과 같이 정의된다.

```
statement:
        labeled-statement
        expression-statement
        compound-statement
        selection-statement
        iteration-statement
        jump-statement
```

정의에 의해, 문장은 레이블이 있는 문장(labeled statement), 수식문(expression statement), 복합문(compound statement), 선택문(selection statement), 반복문(iteration statement), 점프문(jump statement)의 6가지로 나눌 수 있다. 이 절에서는 이들에 대해 학습한다.

5.1) 레이블이 있는 문장(labeled statement)

문장의 앞에는 레이블이 붙을 수 있다. C에서 **레이블(label)**은 goto 문에서 점프할 실제 위치를 나타내는 역할을 한다. switch-case에서 case, default에 사용되는 것 또한 레이블이라고 부른다. 즉 레이블의 정의는 다음과 같다.

```
labeled-statement:
        identifier : statement
        case constant-expression : statement
        default : statement
```

어셈블리의 경우와 마찬가지로, 레이블은 그 자체로는 프로그램에 아무 영향을 미치지 않는다.

5.2) 수식문(expression statement)

수식문(expression statement)은 말 그대로 식을 계산하는 문장을 말한다. 정의는 다음과 같다.

```
expression-statement:
        expression_opt ;
```

수식문은 다음 절에서 더 자세하게 다룬다. 여기서는 수식(expression)이라는 생략 가능한 요소가 있다는 사실만 이해하면 된다. 수식이 생략 가능하므로 정의에 의해 다음 문장 또한 수식문이 된다.

```
expression-statement:
        ;
```

이렇게 아무 일도 하지 않는 문장을 **널 문장(null statement)**이라고 한다.

5.3) 선택문(selection statement)

선택문(selection statement)은 여러 갈래의 제어 흐름에서 한 가지를 고르는 기능을 한다. 선택문의 정의는 다음과 같다.

```
selection-statement:
        if ( expression ) statement
        if ( expression ) statement else statement
        switch ( expression ) statement
```

잘 알고 있듯, if문의 경우 expression의 값이 0인지, 0이 아닌지에 따라 분기한다. 이 세 가지 구문에서 expression은 모두 정수형 값이어야 한다. 예를 들면 반환형이 void인 함수는 선택문에 들어가는 expression이 될 수 없다. switch 문의 경우 case 수식은 모두 정수형이어야 하며, 같은 값을 같은 둘 이상의 case문이 존재하면 안 된다. 나머지는 알고 있는 대로다.

5.4) 반복문(iteration statement)

반복문(iteration statement)은 문장을 반복하는 기능을 한다. 정의는 다음과 같다.

```
iteration-statement:
        while ( expression ) statement
        do statement while ( expression ) ;
        for ( expression_opt ; expression_opt ; expression_opt ) statement
```

for, while, do-while의 사용법은 이미 잘 알고 있으리라 생각한다. for 문의 초기식, 조건식, 증감식은 모두 생략 가능하다. 여기서 몇 가지만 더 설명하겠다. 초기식과 증감식의 경우는 식에 제한이 없지만, 조건식은 반드시 산술 형식 또는 포인터 형식으로 구성되어야 한다. 반환형이 void인 함수는 조건식으로 적합하지 않다. 지금까지 일일이 설명하지 않았지만 이렇게 특수 문법 위반이 일어나는 경우는 아주 많고, 우리는 이들을 모두 처리해야 한다.

5.5) 점프문(jump statement)

점프문(jump statement)은 프로그램의 흐름을 무조건적으로 바꾸는 기능을 한다. 정의는 이렇다.

```
jump-statement:
        goto identifier ;
        continue ;
        break ;
        return expression_opt ;
```

goto문의 식별자는 레이블이어야 한다. break문은 루프나 switch문을 탈출하는 용도로, continue문은 반복문을 다시 실행하는 용도로 사용한다. return은 자신을 호출한 함수로 복귀하는 키워드다. 함수의 끝까지 이것이 등장하지 않으면 함수는 임의의 값을 리턴하게 된다.

5.6) 복합문(compound statement)

이전에 설명한 복합문의 정의를 다시 가져와보자.

```
compound-statement:
        { declaration-list_opt statement-list_opt }

statement:
        labeled-statement
        expression-statement
        compound-statement
        selection-statement
        iteration-statement
        jump-statement
```

여기서 가장 재미있는 건 복합문 또한 문장으로 간주된다는 것이다. 이는 결국 다음과 같이 재귀적으로 복합문을 정의할 수 있다는 것을 나타내기 때문이다.

```
compound-statement:
        definition 1) { declaration-list_opt statement-list_opt }

compound-statement:
        d1) { declaration-list_opt statement }
        d1) { declaration-list_opt compound-statement }
        d1+d1) { declaration-list_opt { declaration-list_opt statement-list_opt } }
        d1+d1) { declaration-list_opt { declaration-list_opt statement } }
        d1+d1) { declaration-list_opt { declaration-list_opt compound-statement } }
        d1+d1+d1) ...
```

복합문 그 자체를 문장으로 본다는 것은, 문장을 획득하는 메서드와 복합문을 획득하는 메서드를 별도로 작성할 필요가 없다는 것을 시사한다. 이는 컴파일러를 구현하기 매우 쉽게 만들어준다. 예를 들어 다음과 같은 두 가지 반복문을 해석해야 한다고 해도,

```
while (condition() == true)
  func();

while (condition() == true)
{
  func();
}
```

while 반복문의 정의가 다음과 같기 때문에,

```
iteration-statement:
        while ( expression ) statement
```

우리는 다음과 같이 코드를 작성하는 것만으로 두 코드를 분석해낼 수 있다.

```
if (next_token() == 'while') { // 다음 토큰이 while이라면
  ... // 조건문 부분을 획득합니다.
  translate_statement(); // 문장을 번역합니다.
}
```

아직 이해가 안 될 테지만 약간만 알려주겠다. 첫 번째 반복문에서 “func();” 문장은 명백하게 수식문이다. 수식문은 문장이므로, 문장을 번역하는 함수를 한 번 호출하는 것으로 처리가 끝난다. 그렇지? 그럼 그 다음에 나타나는 “{ func(); }" 문장은 어떨까? 이는 중괄호로 둘러싸여있으므로 복합문이다. 그런데 문장의 정의에 의해 복합문 또한 문장이다! 따라서 이 역시 문장을 번역하는 함수를 호출하는 것으로 번역에 성공해야 한다. 아주 영리한 방법이다. 이는 아주 중요한 내용이며, 후에 실제로 코드를 작성할 때도 다시 언급할 것이다.

일단 이 정도로 문장의 종류에 대한 설명은 끝이 난다. 다음은 프로그래밍 언어에 빠질 수 없는 수식 계산에 대한 내용을 다룬다.

6. 수식(expression)

6.1) 개요

TCPL에서 수식을 설명하는 절은, 수식을 우선순위 순으로 정렬하여 우선순위가 높은 수식부터 낮은 수식의 순서로 설명하는데, 여기서는 최대한 설명을 하향식으로 진행한다는 원칙에 따라, 정의가 나타나는 순서대로 설명을 진행할 것이다.

6.2) 수식(expression)

TCPL이 제시하는 수식의 정의는 다음과 같다.

```
expression:
        assignment-expression
        expression , assignment-expression
```

수식은 할당식(assignment expression)의 리스트로 이루어져있다. 그럼 할당식의 정의는 무엇인지 살펴보자.

```
assignment-expression:
        conditional-expression
        unary-expression assignment-operator assignment-expression
```

할당식(assignment expression)은 조건식(conditional expression)과, 단항식(unary expression), 할당 연산자(assignment operator)와 할당식으로 이루어져있다. 할당식을 이해하기 위해 정의를 봐야 하는 요소가 3개나 되니 슬슬 부담스러워지기 시작하지만, 차근차근 정의를 따라가다 보면 이를 이해할 수 있게 된다. 먼저 가장 간단한 할당 연산자의 정의를 보자.

```
assignment-operator: one of
        = *= /= %= += -= <<= >>= &= ^= |=
```

할당 연산자(assignment operator)는 할당을 위해 사용하는 연산자를 말한다. one of라는 표현은 아래 제시된 토큰 중 하나임을 말한다. 할당 연산자는 모두 오른쪽에서 왼쪽으로 결합한다. 다시 말해, 오른쪽 피연산자가 먼저 계산된 다음 왼쪽 피연산자가 계산된다. 또한 할당 연산자의 왼쪽 피연산자는 반드시 변경 가능한 좌변값(l-value)이어야 한다는 특성이 있다. 좌변값이 무엇인지를 알아보자.

6.3) 좌변값(l-value)

좌변값(l-value)이란, 할당식 "E1 = E2"에서 왼쪽의 피연산자 E1으로 사용할 수 있는 값을 말한다. 좌변값은 배열, 함수나 그 외의 불완전한 형태는 될 수 없다. 예를 들어 "int arr[10];"과 같이 선언된 배열 arr에 대해 "arr = arr2;"와 같은 식은 성립하지 않는다. 또한 const로 선언되지 않아야 하고, 구조체나 공용체인 경우 const를 포함하는 멤버를 가져서는 안 된다.

TCPL은 할당식의 좌변과 우변에 있는 피연산자가 다음 조건을 만족해야 한다고 설명하고 있다.

- 두 피연산자 모두 산술 형태이고, 오른쪽 피연산자는 왼쪽 피연산자의 형으로 변환된다.
- 두 피연산자 모두 같은 형태의 구조체나 공용체다.
- 한 피연산자는 포인터이고, 다른 하나는 void에 대한 포인터이다.
- 왼쪽 피연산자는 포인터이고, 오른쪽 피연산자는 0 값을 갖는 상수 수식이다.

좌변값의 예를 들어보자. 다음은 수식과, 그 수식에 나타난 좌변값을 표현한 것이다.

```
num = 1 + 2; // lvalue: num
*ptr = 20; // lvalue: *ptr
arr[3] = 30; // lvalue: arr[3]
**dp = 40; // lvalue: **dp
*(arr+3) = 50; // lvalue: *(arr+3)
*(int *)str = 60; // lvalue: *(int *)str
```

이와 같이 좌변값이 될 수 있는 식은 종류가 여러 가지 있다. 할당식의 정의에도 있지만, 할당 연산자의 왼쪽 피연산자는 단항식(unary expression)인데, 바로 이것이 좌변값이 된다. 그럼 이제 단항식의 정의를 살펴보자.

6.4) 단항식(unary expression)

단항식의 정의는 다음과 같다.

```
unary-expression:
        postfix-expression
        ++ unary-expression
        -- unary-expression
        unary-operator cast-expression
        sizeof unary-expression
        sizeof ( type-name )
```

단항식은 접미 수식(postfix expression), 증감 연산자를 포함하는 단항식, 단항 연산자와 캐스트 수식(cast expression), sizeof 수식으로 정의되어있다. 하지만 "sizeof(int) = 20;"과 같은 수식이 불가능하듯, 이들 모두가 좌변값이 될 수 있는 것은 아니다. 이러한 처리는 모두 특수 문법 위반에 들어간다.

우리가 모르는 표현은 접미 수식과 단항 연산자, 캐스트 수식의 세 가지다. 단항 연산자부터 정의를 알아보자. 사실 우리는 이미 이 내용을 잘 알고 있다.

```
unary-operator: one of
        & * + - ~ !
```

덧셈, 뺄셈, 별표, AND 연산자는 각각 피연산자 두 개를 갖는 덧셈, 뺄셈, 곱셈, 논리곱 연산에 사용되지만, 단항 연산자로서도 사용할 수 있다. 이를테면 다음과 같은 경우 각 연산자는 모두 단항 연산자로 쓰였다.

```
sum = +a + -b; // 단항 연산자 두 개(+, -)와 이항 연산자 둘(*, =)
*ptr = *p1 * *p2; // 단항 연산자 세 개(*)와 이항 연산자 둘(*, =)
ptr = &num; // 단항 연산자 하나(&)와 이항 연산자 하나(=)
```

이는 같은 기호를 갖는 연산자라도, 그것이 결합된 식이 어떠한지에 따라 연산자의 역할이 바뀐다는 것을 의미한다. 예를 들면 두 번째 예시는 별표 연산자가 참조 연산으로써도 사용되었고, 곱셈 연산으로써도 사용되었다. 이는 식을 분석하는 것을 한층 어렵게 만드는데, 이에 대해서는 후에 다루겠다.

두 번째로 다룰 것은 캐스트 수식이다. **캐스트 수식(cast expression)**이란 말 그대로 캐스트를 포함하는 수식을 말한다. 정의는 다음과 같다.

```
cast-expression:
        unary-expression
        ( type-name ) cast-expression
```

우리는 이러한 정의가 무엇을 의미하는지를 계속 공부해왔다. 따라서 추가로 설명하지 않더라도 캐스트 수식은 어렵지 않게 이해할 수 있다.

단항식에서 다루지 않은 마지막 요소는 접미 수식인데, 이에 대해서는 설명이 더 필요하다.

6.5) 접미 수식(postfix expression)

접미 수식이란 자신의 뒤에 배열 대괄호, 함수 괄호 등이 붙어있는 수식을 말한다. 접미 수식의 정의는 다음과 같다. 이들은 모두 왼쪽에서 오른쪽으로 결합한다.

```
postfix-expression:
        primary-expression
        postfix-expression [ expression ]
        postifx-expression ( argument-expression-list_opt )
        postfix-expression . identifier
        postfix-expression -> identifier
        postfix-expression ++
        postifx-expression --
```

그리고 이 정의에서 나타난 요소 중 기본 수식(primary expression)의 정의를 먼저 보고 가자.

```
primary-expression:
        identifier
        constant
        string
        ( expression )
```

기본 수식(primary expression)이란 말 그대로 수식의 기본이 되는 요소를 말한다. 위의 정의에 따르면, 다음은 모두 기본 수식이 된다.

```
num                 // D1) identifier
10                  // D2) constant
"Hello, world!"     // D3) string
(1 + 2)             // D4) ( expression )
```

즉, 다음의 식은 모두 접미 수식이며, 따라서 모두 단항식으로 본다.

```
arr[idx]      // d2) postfix-expression [ expression ]
sum()         // d3) postfix-expression ( )
data.value    // d4) postfix-expression . identifier
pdata->value  // d5) postfix-expression -> identifier
num++         // d6) postfix-expression ++
```

```
10[arr]              // D2+d2) constant [ expression ]
"Hello, world!"[2]   // D3+d2) string [ expression ]
(ptr+1)[10]          // D4+d2) ( expression ) [ expression ]
```

그리고 접미 수식의 정의가 재귀적이기 때문에 다음의 식 또한 유효한 단항식이 된다.

```
arr3d[0][1]          // D1+d2+d2) identifier [ expression ] [ expression ]
person.wallet.money  // D1+d4+d4) identifier . identifier . identifier
```

남은 건 인자 수식 리스트(argument expression list)인데, 이에 대해서는 설명할 것이 없다.

> *argument-expression-list*:
> *assignment-expression*
> *argument-expression-list* , *assignment-expression*

결국 **인자 수식 리스트(argument expression list)**란 반점으로 구분된 할당식의 리스트일 뿐이다. 우리는 함수를 호출할 때 이러한 구문을 작성한다.

```
printf("%d == %d: %s, %p \n", 10, input, (input == 10 ? "true" : "false"), &input);
```

이것으로 더 이상의 설명은 필요하지 않을 것이다.

6.6) 조건식(conditional expression)

할당식의 두 번째 정의에 존재하는 요소는 모두 다루었다. 이제 남아있는 조건식에 대해 다루자. **조건식(conditional expression)**은 말 그대로 조건을 위한 식이다. 정의는 다음과 같다.

> *conditional-expression*:
> *logical-OR-expression*
> *logical-OR-expression* ? *expression* : *conditional-expression*

이러한 정의는 삼항 연산자를 위한 것이다. 첫 번째 수식인 논리합 수식(logical OR expression)을 계산하고, 그 값이 0이 아니라면 두 번째 수식을, 0이라면 세 번째 수식을 반환한다. 또한 두 식 중 조건에 맞는 하나의 식만 계산된다는 점도 주의해야 한다. 삼항 연산자를 사용할 때는 반환해야 하는 값의 형식이 서로 다른 경우가 있는데, 이에 대한 처리를 나열해보면 다음과 같다.

- 두 피연산자가 산술 식이라면, 같은 형이 되도록 형 변환이 일어난다.
- 두 피연산자의 형이 모두 void, 같은 구조체 또는 공용체, 같은 형의 대상에 대한 포인터라면, 같은 형이 되도록 형 변환이 일어난다.
- 한 피연산자는 포인터이고 다른 하나는 상수 0이라면, 0이 포인터 형태로 형 변환한다.
- 한 피연산자가 void에 대한 포인터이고 다른 하나는 또 다른 형의 포인터라면, 다른 형의 포인터를 void에 대한 포인터로 형 변환한다.
- 포인터의 형 비교 시에 형식 한정자(type qualifier)는 중요하지 않지만, 계산이 끝나면 두 피연산자의 형식 한정자를 물려받는다.

몇 가지 이해하기 힘든 것이 있는데, 이에 대해 당장 모든 것을 이해할 필요는 없다. 자세한 처리는 실제로 컴파일러를 구현하면서 다룰 것이다.

조건식에는 논리합 수식이 포함되어있으므로, 이에 대한 정의를 보지 않을 수 없다. 논리합 수식의 정의는 다음과 같다.

> *logical-OR-expression*:
> *logical-AND-expression*
> *logical-OR-expression* || *logical-AND-expression*

여기에는 논리곱 수식(logical AND expression)이 포함되어있다. 그런데 논리합 수식에 논리합 연산

자 기호(||)가 들어간 걸로 보아, 논리곱 수식이 어떤 형태를 하고 있을지는 어느 정도 짐작이 가지 않는가? 실제로 나머지도 거의 같은데, 그럼 이들의 정의를 차례로 훑어보자.

논리곱 수식의 정의는 다음과 같다.

```
logical-AND-expression:
        inclusive-OR-expression
        logical-AND-expression && inclusive-OR-expression
```

논리곱 수식은 자신과 비트합 수식(Bitwise inclusive OR expression)으로 이루어져있다. 그런데 여기서 짚고 넘어갈 것이 있다. 비트합 수식이 무엇이든 간에, 논리곱 수식은 && 연산자를 사용하는 식이고, 논리합 수식은 || 연산자를 사용하는 식이다. 그러면 다음의 식은 정당한 논리합 수식으로 볼 수 있다.

```
if ( cond1 && cond2 || cond3 && cond4 ) { ... }
```

여기서 중요한 것은, 논리합 연산자인 || 연산자를 해석하기 전에 논리곱 연산자인 && 연산자가 먼저 해석된다는 점이다. 이는 곧 연산자의 우선순위는 || 연산자보다 && 연산자가 높다는 사실을 나타낸다. 실제로 C 프로그래밍 언어의 연산자 우선순위는 다음과 같다.

연산자의 우선순위 (위로 갈수록 우선순위가 높다)
() [] -> . ! ~ ++ -- + - * & (type) sizeof
* / % + - << >> < <= > >= == != & ^ | && || ?: = += -= *= /= %= &= ^= |= <<= >>= ,

즉 우리는 이 내용을 통해, 앞으로 어떻게 각 요소들이 정의될지를 책의 도움 없이 추론할 수 있다. 우리는 다음과 같이 정의되는 수식(expression)의 반점 연산자(,)에서 시작하여,

```
expression:
        assignment-expression
        expression , assignment-expression
```

할당식과 할당 연산자의 정의를 보았고,

```
assignment-expression:
        conditional-expression
        unary-expression assignment-operator assignment-expression
```

```
assignment-operator: one of
        = *= /= %= += -= <<= >>= &= ^= |=
```

삼항 연산자를 포함하는 조건식의 정의를 지난 다음,

```
conditional-expression:
        logical-OR-expression
        logical-OR-expression ? expression : conditional-expression
```

비로소 논리합 수식까지 오게 된 것이다. 다시 말해 우리는 연산자의 우선순위가 낮은 쪽부터 차례대로 연산자 우선순위가 높은 연산자까지 요소 탐색을 진행하고 있다.

계속 진행해보자. 비트합 수식의 정의는 다음과 같다.

```
inclusive-OR-expression:
        exclusive-OR-expression
        inclusive-OR-expression | exclusive-OR-expression
```

비트합 수식은 자신과 배타 비트합 수식(Bitwise exclusive OR expression)으로 이루어져있다.
배타 비트합 수식의 정의는 다음과 같다.

```
exclusive-OR-expression:
        AND-expression
        exclusive-OR-expression ^ AND-expression
```

배타 비트합 수식은 자신과 비트곱 수식(Bitwise AND expression)으로 이루어져있다.
비트곱 수식의 정의는 다음과 같다.

```
AND-expression:
        equality-expression
        AND-expression & equality-expression
```

비트곱 수식은 자신과 등가 수식(equality expression)으로 이루어져있다.
등가 수식의 정의는 다음과 같다.

```
equality-expression:
        relational-expression
        equality-expression == relational-expression
        equality-expression != relational-expression
```

등가 수식은 자신과 관계 수식(relational expression)으로 이루어져있다.
관계 수식의 정의는 다음과 같다.

```
relational-expression:
        shift-expression
        relational-expression < shift-expression
        relational-expression > shift-expression
        relational-expression <= shift-expression
```

```
relational-expression >= shift-expression
```

관계 수식은 자신과 쉬프트 수식(shift expression)으로 이루어져있다.
쉬프트 수식의 정의는 다음과 같다.

```
shift-expression:
        additive-expression
        shift-expression << additive-expression
        shift-expression >> additive-expression
```

쉬프트 수식은 자신과 덧셈식(additive expression)으로 이루어져있다.
덧셈식의 정의는 다음과 같다.

```
additive-expression:
        multiplicative-expression
        additive-expression + multiplicative-expression
        additive-expression - multiplicative-expression
```

덧셈식은 자신과 곱셈식(multiplicative expression)으로 이루어져있다.
곱셈식의 정의는 다음과 같다.

```
multiplicative-expression:
        cast-expression
        multiplicative-expression * cast-expression
        multiplicative-expression / cast-expression
        multiplicative-expression % cast-expression
```

결국 곱셈식은 우리가 단항식을 정의할 때 본, 캐스트 수식으로 이루어져있음을 알게 된다. 이는 다시 단항식이기 때문에, 결국 수식은 모두 단항식의 결합으로 볼 수 있게 된다. 각 연산자들의 역할에 대해서는 우리가 이미 알고 있으니 별도로 설명하지 않아도 좋을 것 같다.

6.7) 상수(constant)

상수는 다음과 같이 정의된다.

```
constant:
        integer-constant
        character-constant
        floating-constant
        enumeration-constant
```

정의에서 알 수 있듯, 상수는 여러 종류가 있음을 알 수 있다. 이들에 대해 알아보자.

6.7.1) 정수 상수(integer constant)

정수 상수(integer constant)란 연속된 숫자로 구성된 토큰을 말한다. 0으로 시작하면 8진법의 수를, 0x로 시작하면 16진법의 수를 의미하고, 앞에 붙은 것이 없으면 10진법의 수를 의미한다.

unsigned임을 나타내기 위해 토큰의 뒤에 u 또는 U 문자가 붙을 수 있다. long 형태임을 나타내려면 토큰의 뒤에 l 또는 L 문자를 붙인다.

6.7.2) 문자 상수(character constant)

문자 상수(character constant)는 'x'와 같이, 하나 이상의 문자들이 작은따옴표 사이에 있는 것이다. 한 문자로 된 문자 상수의 값은, 프로그램을 실행하는 기종의 문자 셋에서 해당 문자의 수치 값이 된다. 말이 이해하기 어려운데 간단히 설명하자면, ASCII 코드 표를 쓰는 컴퓨터에서 'A'라고 쓰면 ASCII 코드 표에서 문자 A의 정수 값인 65가 나온다고 이해하면 된다.

작은따옴표 또는 개행 문자를 표현하고자 할 때는 이스케이프 시퀀스(escape sequence)를 사용한다. 이스케이프 시퀀스의 목록은 다음과 같다.

newline	NL (LF)	\n	backslash	\	\\
horizontal tab	HT	\t	question mark	?	\?
vertical tab	VT	\v	single quote	'	\'
backspace	BS	\b	double quote	"	\"
carriage return	CR	\r	octal number	ooo	\ooo
formfeed	FF	\f	hex number	hh	\xhh
audible alert	BEL	\a			

이 정도로 문자 상수는 이해할 수 있다.

6.7.3) 부동소수점 상수(floating constant)

부동소수점 상수(floating constant)는 실수를 표현하는 토큰이다. 정수 상수의 경우와 마찬가지로 접미어가 붙을 수 있는데, 접미어로 가능한 것은 f, F, l, L 중의 하나이다. 접미어가 f 또는 F이면 float, L이나 l이면 long double이고 그 외의 경우 모두 double로 간주한다. 접미어는 아니지만 e 표기법이라고 하여 실수를 표현하는 방법이 별도로 존재하는데 이는 그렇게 어렵지 않다. 보통 그 꼴은 다음과 같이 나타난다.

```
1.2e8 // same with 1.2 * pow(10, 8)
3.45e-23 // same with 3.45 * pow(10, -23)
```

이 정도면 e를 사용하는 방법을 이해할 수 있을 것이다.

6.7.4) 열거 상수(enumeration constant)

열거 상수(enumeration constant)는 enum 키워드를 이용해 정의한 명칭을 말한다. 모두 int 값이다.

6.7.5) 문자열(string literal)

문자열(string)은 큰따옴표 사이에 들어가는 연속적인 문자들을 말하며, 문자열 상수라고도 한다. 문자들의 배열 형태를 가지고, 기억 형태 지정자가 static인 것으로 간주된다. C에서는 문자열의 끝에 널 바이트를 두어서 문자열을 서로 구분한다. 문자열 사이에 별도의 토큰이 없으면 두 문자열을 하나로 간주한다. 예를 들어 다음과 같이 코드를 작성하면,

```
char *s = "Hello, " "world!";
puts(s);
```

출력 결과는 “Hello, world!”가 된다.

6.7.6) 상수 수식(constant expression)

상수 수식은 다음과 같이 정의된다.

```
constant-expression:
        conditional-expression
```

상수 수식(constant expression)이란 상수에 사용할 수 있는 연산자만 사용된 수식을 말한다. 상수 수식의 정의는 조건식으로 되어있지만, 여기에는 변수가 포함될 수 없다. 상수 수식은 case 문, 배열의 크기, 비트 필드의 길이, 열거 상수의 값, 전처리기의 수식 등 여러 가지 경우에 필요하다. 상수 수식은 sizeof의 피연산자인 경우를 제외하고는 지정, 증감 연산, 함수 호출이나 반점 연산자를 포함할 수 없다. 참고로 말하자면 C에서 const 형식 한정자를 이용해 선언된 변수는 읽기 전용 메모리일 뿐 상수로 취급되지는 않는다.[2)]

6.8) 초기 값 지정자(initializer)

우리는 초기 선언자의 정의가 다음과 같음을 배웠다.

```
init-declarator:
        declarator
        declarator = initializer
```

초기 값 지정자(initializer)는 선언에 대하여 초기 값을 지정하는 역할을 한다. 정의는 다음과 같다.

```
initializer:
        assignment-expression
        { initializer-list }
        { initializer-list , }

initializer-list:
        initializer
        initializer-list , initializer
```

이러한 정의는 어렵지 않게 받아들일 수 있다. 다만 이를 응용하는 것은 설명이 더 필요하므로 여기서 이야기하자.

정적 변수나 배열에 대한 초기 값 지정자 내의 모든 수식은, 반드시 상수 수식이거나 이전에 선언된 변수의 주소 값으로 이루어진 수식이어야 한다. 초기 값이 주어지지 않은 정적 변수와 외부 변수는 초기 값이 0으로 정의된다.

집합체 내에서 원소의 수보다 초기 값 지정자의 수가 적으면, 초기화되지 않은 나머지 원소는 모두 0으로 채워진다. 예를 들어 다음과 같은 배열의 선언이 있으면,

```
int x[10] = { 1, 2, 3, }; // [1, 2, 3, 0, 0, 0, 0, 0, 0, 0]
```

3번째 이후의 원소는 모두 0으로 채워지게 된다. 다른 예시를 들어보자. 다음과 같이 이차원 배열이 선언되어있는 경우,

```
float y[4][3] = { // 4행 3열의 float 형식 2차원 배열
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

실제 y는 다음과 같은 형식으로 기록된다.

```
[ [1,0,0], [2,0,0], [3,0,0], [4,0,0] ]
```

마지막으로 문자 배열의 초기 값으로 문자열을 줄 수도 있다.

```
char *msg = "Hello, world! \n";
```

---

2) 이는 C++가 C와 다른 점이기도 하다. 이에 관한 문서(https://kldp.org/node/68231)를 참조하라.

이와 같이 초기 값 지정자의 사용법에 대해 이해할 수 있었다.

7. 전처리기(Preprocessor)

전처리기(Preprocessor)는 컴파일 이전의 소스 코드에 대한 추가적인 처리를 담당한다. 여기에는 매크로 치환, 조건부 컴파일이나 파일 포함과 같은 것이 관련되어있다. 전처리기에 처리를 요청하려면 반올림 기호(#)를 이용한다. #으로 시작되는 문장은 C언어의 문법과는 관계가 없다. 이들은 코드 어디에나 위치할 수 있고, 블록과 같은 범위(scope)에 관계없이 프로그램 전체에 걸쳐서 영향을 미친다.

전처리기도 전처리기 나름의 문법이 있다. 길지 않으므로 이 절에서 모두 다루자.

7.1) 제어 라인(control line)

전처리기에 전달되는 행 텍스트를 제어 라인(control line)이라고 한다. 정의는 다음과 같다.

```
control-line:
        # define identifier token-sequence
        # define identifier ( identifier-list ) token-sequence
        # undef identifier
        # include <filename>
        # include "filename"
        # include token-sequence
        # line constant "filename"
        # line constant
        # error token-sequence_opt
        # pragma token-sequence_opt
        #
        preprocessor-conditional
```

얼핏 보면 많아 보이지만, 실제로는 전처리기가 담당하는 건 이것이 전부다. 1~3번 정의는 매크로에 관한 것이고, 4~6번 정의는 파일 포함에 관한 것이다. 7~9번 정의는 디버깅 용도로 사용하는 것인데 우리는 사용하지 않는다. 10번 정의는 전처리기 개발자 스스로가 원하는 기능을 임의로 넣는 pragma 전처리를 의미한다. 11번 정의는 널 문장이라고 하여 아무 일도 하지 않는 문장이고, 전처리기 조건식(preprocessor conditional)은 #if, #elif, #endif를 사용하는 문장을 말한다. 이들은 모두 어렵지 않게 이해할 수 있다.

7.2) 매크로 정의

매크로를 정의하려면 다음의 형식을 따른다.

```
# define identifier token-sequence
# define identifier ( identifier-list ) token-sequence
```

토큰 시퀀스(token sequence)는 토큰이 나열된 것을 말한다. 우리는 #define 전처리 메시지를 다음과 같이 사용하는 일이 흔하다.

```
#define MAX_ARRAY_LEN 30
#define ABS(a, b) ((a) > (b) ? (a)-(b) : (b)-(a))
```

위의 정의는 이를 일반적으로 표현한 것이다. 토큰 시퀀스 양 옆의 공백은 무시된다. 또한 #define의 경우 토큰 시퀀스 자체가 생략될 수도 있다. 이는 우리가 헤더 파일을 중복으로 포함하지 않기 위해 작성하는 코드에서 나타난다.

```
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__
...
#endif
```

이 코드의 두 번째 행에서 __MY_HEADER_H__라는 식별자를 정의하고 있음을 알 수 있다. 이 경우 식별자 __MY_HEADER_H__는 공백으로 정의된 것으로 간주한다.

```
# undef identifier
```

이는 식별자의 정의를 해제하는 것인데, 정의되지 않은 식별자를 해제해도 에러가 발생하지 않는다.

매크로 정의에서 샵 기호(#)는 아주 특별하게 사용된다. 다음과 같은 코드를 생각하자.

```
#define tempfile(dir) #dir "/%s"

main() {
  char *tmpname = 0;
  tmpname = tempfile(/usr/tmp);
}
```

이 코드에서 매크로 tempfile의 결과는 다음이 된다.

```
"/usr/tmp" "/%s"
```

전처리기 연산자는 토큰열 내의 매개변수 앞에 붙은 경우, 해당 매개변수의 양 옆에 큰따옴표를 붙인다. 매개변수 안에 있는 문자 상수나 문자열 내에 큰따옴표나 백슬래쉬 기호가 이미 있는 경우엔 그 앞에 자동으로 백슬래쉬 기호를 붙인다.

두 번째 경우를 생각해보자. 다음과 같은 코드가 있다.

```
#define cat(x, y) x ## y

main() {
  int foo = 10, bar = 20;
  int foobar = 30;
  cat(foo, bar) = 40;
}
```

이 코드에서 매크로 cat의 결과는 다음이 된다.

```
foobar = 40;
```

전처리기 연산자는 ## 양쪽의 공백을 없애고, 앞뒤의 문자열을 붙여서 하나로 만든다.

7.3) 파일 포함

파일을 포함하는 방법은 두 가지다. 하나는 부등호('<', '>')를 이용하는 것이고, 다른 하나는 큰따옴표를 이용하는 것이다. 정의는 다음과 같다.

```
# include <filename>
# include "filename"
# include token-sequence
```

부등호를 사용하는 경우는 filename을 표준 파일에서만 탐색한다. 반면, 큰따옴표를 사용하는 경우는 우선 원래의 소스 파일 디렉터리에서 찾아본 다음, 거기에 없으면 표준 파일에서 탐색한다. 세 번째 정의는 토큰 문자열을 사용 가능하다고 되어있는데, 이는 #define 전처리 메시지로 정의한 식별자를 이야기한다. 예를 들어 (#define TEST "stdlib.h")라고 정의했다면 (#include TEST)라고 쓸 수 있다.

7.4) 전처리기 조건식(preprocessor conditional)

전처리기 조건식(preprocessor conditional)은 조건부 컴파일을 위해 필요한 문법이다. 정의는 다음과 같다.

```
preprocessor-conditional:
        if-line text elif-parts else-part_opt #endif

if-line:
        # if constant-expression
        # ifdef identifier
        # ifndef identifier

elif-parts:
        elif-line text
        elif-parts_opt

elif-line:
        # elif constant-expression

elif-parts:
        else-line text

else-line:
        # else
```

이미 이러한 구조를 몇 번이나 이해한 우리에게 이 정도는 전혀 어렵지 않다.

8. C 프로그래밍 언어의 문법

결국 C 프로그래밍 언어의 문법은 다음과 같이 정리할 수 있다.

```
translation-unit:
        external-declaration
        translation-unit external-declaration

external-declaration:
        function-definition
        declaration

declaration:
        declaration-specifiers init-declarator-list_opt ;
```

```
init-declarator-list:
        init-declarator
        init-declarator-list, init-declarator

init-declarator:
        declarator
        declarator = initializer

declaration-specifier:
        storage-class-specifier declaration-specifiers_opt
        type-specifier declaration-specifiers_opt
        type-qualifier declaration-specifiers_opt

storage-class-specifier:
        auto
        register
        static
        extern
        typedef

type-specifier:
        void
        char
        short
        int
        long
        float
        double
        signed
        unsigned
        struct-or-union-specifier
        enum-specifier
        typedef-name

type-qualifier:
        const
        volatile

struct-or-union-specifier:
        struct-or-union identifier_opt { struct-declaration-list }
        struct-or-union identifier

struct-or-union:
        struct
```

```
        union

struct-declaration-list:
        struct-declaration
        struct-declaration-list struct-declaration

struct-declaration:
        specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
        type-specifier specifier-qualifier-list_opt
        type-qualifier specifier-qualifier-list_opt

struct-declarator-list:
        struct-declarator
        struct-declarator-list, struct-declarator

struct-declarator:
        declarator
        declarator_opt : constant-expression

enum-specifier:
        enum identifier_opt { enumerator-list }
        enum identifier

enumerator-list:
        enumerator
        enumerator-list, enumerator

enumerator:
        identifier
        identifier = constant-expression

typedef-name:
        identifier

declarator:
        pointer_opt direct-declarator

direct-declarator:
        identifier
        ( declarator )
        direct-declarator [ constant-expression_opt ]
        direct-declarator ( parameter-type-list )
```

```
        direct-declarator ( identifer-list_opt )

pointer:
        *type-qualifier-list_opt
        *type-qualifier-list_opt pointer

type-qualifier-list:
        type-qualifier
        type-qualifier-list type-qualifier

parameter-type-list:
        parameter-list
        parameter-list, ...

parameter-list:
        parameter-declaration
        parameter-list, parameter-declaration

parameter-declaration:
        declaration-specifiers declarator
        declaration-specifiers abstract-declarator_opt

identifier-list:
        identifier
        identifier-list, identifier

type-name:
        specifier-qualifier-list abstract-declarator_opt

abstract-declarator:
        pointer
        pointer_opt direct-abstract-declarator

direct-abstract-declarator:
        ( abstract-declarator )
        direct-abstract-declarator_opt [ constant-expression_opt ]
        direct-abstract-declarator_opt ( parameter-type-list_opt )

function-definition:
        declaration-specifiers_opt declarator declaration-list_opt compound-statement

declaration-list:
        declaration
        declaration-list declaration
```

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

labeled-statement:
identifier : *statement*
case *constant-expression* : *statement*
default : *statement*

expression-statement:
expression$_{opt}$;

selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* else *statement*
switch (*expression*) *statement*

iteration-statement:
while (*expression*) *statement*
do *statement* while (*expression*) ;
for (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

jump-statement:
goto *identifier* ;
continue ;
break ;
return *expression*$_{opt}$;

expression:
assignment-expression
expression , *assignment-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

```
assignment-operator: one of
       = *= /= %= += -= <<= >>= &= ^= |=

unary-expression:
       postfix-expression
       ++ unary-expression
       -- unary-expression
       unary-operator cast-expression
       sizeof unary-expression
       sizeof ( type-name )

unary-operator: one of
       & * + - ~ !

cast-expression:
       unary-expression
       ( type-name ) cast-expression

postfix-expression:
       primary-expression
       postfix-expression [ expression ]
       postifx-expression ( argument-expression-list_opt )
       postfix-expression . identifier
       postfix-expression -> identifier
       postfix-expression ++
       postifx-expression --

primary-expression:
       identifier
       constant
       string
       ( expression )

argument-expression-list:
       assignment-expression
       argument-expression-list , assignment-expression

conditional-expression:
       logical-OR-expression
       logical-OR-expression ? expression : conditional-expression

logical-OR-expression:
       logical-AND-expression
       logical-OR-expression || logical-AND-expression
```

```
logical-AND-expression:
        inclusive-OR-expression
        logical-AND-expression && inclusive-OR-expression

inclusive-OR-expression:
        exclusive-OR-expression
        inclusive-OR-expression | exclusive-OR-expression

exclusive-OR-expression:
        AND-expression
        exclusive-OR-expression ^ AND-expression

AND-expression:
        equality-expression
        AND-expression & equality-expression

equality-expression:
        relational-expression
        equality-expression == relational-expression
        equality-expression != relational-expression

relational-expression:
        shift-expression
        relational-expression < shift-expression
        relational-expression > shift-expression
        relational-expression <= shift-expression
        relational-expression >= shift-expression

shift-expression:
        additive-expression
        shift-expression << additive-expression
        shift-expression >> additive-expression

additive-expression:
        multiplicative-expression
        additive-expression + multiplicative-expression
        additive-expression - multiplicative-expression

multiplicative-expression:
        cast-expression
        multiplicative-expression * cast-expression
        multiplicative-expression / cast-expression
        multiplicative-expression % cast-expression
```

```
constant:
        integer-constant
        character-constant
        floating-constant
        enumeration-constant

constant-expression:
        conditional-expression

initializer:
        assignment-expression
        { initializer-list }
        { initializer-list , }

initializer-list:
        initializer
        initializer-list , initializer

control-line:
        # define identifier token-sequence
        # define identifier ( identifier-list ) token-sequence
        # undef identifier
        # include <filename>
        # include "filename"
        # include token-sequence
        # line constant "filename"
        # line constant
        # error token-sequence_opt
        # pragma token-sequence_opt
        #
        preprocessor-conditional

preprocessor-conditional:
        if-line text elif-parts else-part_opt #endif

if-line:
        # if constant-expression
        # ifdef identifier
        # ifndef identifier

elif-parts:
        elif-line text
        elif-parts_opt
```

```
elif-line:
        # elif constant-expression

elif-parts:
        else-line text

else-line:
        # else
```

이로써 여러분은 C 프로그래밍 언어의 대부분의 문법을 익힐 수 있었다.

9. 단원 마무리

C 프로그래밍 언어는 배우기 쉽고, 사용하기도 쉬운 언어다. 하지만 이를 설계의 관점에서 바라보는 것은 제법 난이도가 높다. 이 문서를 모두 이해한 사람이라면 적어도 C의 문법에 대해서는 많이 알고 있다고 자부해도 좋다.[3] 이 문서를 통해 얻을 수 있는 가장 큰 장점은, 프로그래밍 언어의 설계도를 직접 보고, 그 설계도를 읽을 수 있는 능력이 생겼다는 점이다. 다음 문서를 통해 자신의 재귀적 사고력이 크게 늘었음을 실감할 수 있을 것이며, 또한 설계가 있는 개발이 얼마나 편리한 것인지를 알게 될 것이다.

다음 문서는 드디어 컴파일러의 최종장이다. 모든 것을 구현하는 방법을 가르치지는 않지만, 필요한 것을 어떻게 구현해야 하는지는 가르친다. 이는 필자가 1년간 연구해온, JSCC라는 거대한 프로젝트의 종착역이다. 다음 문서를 끝으로 이 시리즈는 연재를 마친다.

3) 사실 이 문서에서 소개한 문법은 최근의 표준인 C99가 반영되지 않았기 때문에, C의 문법을 모두 설명했다고 하기 힘들다.