The C Programming Language (2nd edition)
Reference Manual (부록 A) 번역본

최종 수정일: 2004년 1월 2일

번역자: 전웅 (woong@icu.ac.kr)
원본 출처: http://www.woong.org

▷ 번역문을 보기 전에 알아야 할 사항

언젠가도 이야기한 적이 있지만, 영어라는 언어는 참 논리적입니다. (물론 한국어도 잘 쓰면 충분히 논리적입니다) 제가 번역을 할 때도 가능하면 그 분명한 논리를 오해없이 전달하려고 노력했습니다. (비록 결과는 어떨지 모르지만 ^^)

또한 영어로 된 C언어 용어를 옮길 때는 제 나름대로 익숙한 한국어 용어를 선택하고, 오해가 없도록 영어를 괄호 안에 병서하였습니다. 같은 영어용어라고 해도 사람마다 한국어로는 다르게 알고 있을 수 있기에, 될 수 있으면 많은 단어를 영어로 함께 표기하려 했지만, 너무 복잡해 질 수도 있기 때문에 레퍼런스 메뉴얼의 처음에서 끝으로 갈수록 자주 나오는 단어는 점차 병서표기를 하지 않았습니다.

원서에는, 일반 영어단어, C 언어에 사용되는 단어, C 프로그램에 사용되는 단어들을 구분하기 위해, 글자체를 다르게 하여 사용하고 있습니다. 실제로 이러한 방법은 글을 읽는데 많은 도움이 됩니다. 하지만 제 정성이 부족해서 (저도 바쁠 때가 있습니다 ^^;) 그 정도의 노력까지는 들이지 못했습니다. 제 짧은 생각으로는, 우리말이 영어가 아니기에 그러한 구분 없이도 그리 큰 혼란을 주지는 않는다고 믿습니다. 단, A13 절의 문법요약에서는 원서에 있는 글자체 표기를 그대로 따랐습니다.

C 언어와 관련된 내용이기에, 영어는 이해가 되지만 그 내용이 C 언어와 결부되지 못하는 부분이 있으면 (→ 해석은 되도 이해가 되지 않는 부분), 제 주관적인 기준으로 가차없이(?) 주(註)를 달았습니다.

그리고, 한국어로 쓰면 뜻이 비슷해지지만, 영어로 쓰면 다소 다른 뜻을 갖는 단어들에 대해 미리 알리고자 합니다. (참고로 이곳에 쓰여진 용어의 정의는 C99 표준안과, C FAQs 를 바탕으로 한 것입니다)

implementation	번역에서는 마땅한 단어를 찾기 힘들어, 그냥 '컴파일러' 라고 했습니다. 하지만, 표준에서의 정확한 의미는 "프로그램의 번역을 수행하고, 함수의 실행을 지원하는, 특별한 제어 옵션 아래에 있는 특별한 번역환경에서 실행되는, 소프트웨어의 특별한 집합, 특별한 실행환경" 입니다. 다른 곳에서는 흔히 '구현' 이라고 번역하기도 합니다.
undefined	이식성이 없거나 잘못된 프로그램 구조, 혹은 에러가 있는 데이터나 값이 불확실한 대상체의 사용에 의존하는 행동으로, 표준에서는 어떠한 요구도 하지 않는 것입니다. 프로그래머는 이러한 사항에 의존해서는 절대 안됩니다.
unspecified	표준에서는 2개 이상의 가능성을 제시하고, 그것의 선택에 어떠한 강요도 하지 않는 것입니다. 결정한 사항을 프로그래머에게 알릴 의무는 없으며, 이러한 사항에 의존하는 프로그래밍 역시 좋지 않습니다.
implementation-defined	표준에서 제시한 가능성 중에서, 각 implementation 이 어떠한 결정을 내렸는지 정확한 사항을 프로그래머에게 알려줄 의무가 있는 것입니다. implementation-dependent 라고도 합니다.
int	말 그대로, 아무것도 붙지않은 int 형을 말합니다.
integer	int 형만을 가리키는 것이 아니라, 어떤 크기의 정수형을 말합니다. (대개 short 나 long)

integral	정수를 표현할 수 있는 모든 데이터형을 말합니다. C 에서는, char, 세 종류의 int (short, int, long), 앞에 것들의 signed 와 unsigned, 열거형이 있습니다.

맨 위에서 다룬 세 단어는 한국어로 번역하면 그 의미가 "표준에서 정의하지 않았다" 정도에 그치기 때문에, 매번 뒤에 영어단어를 명시해 두었습니다. 물론 그 밑에 있는 세 단어도 의미가 모두 유사하게 "정수형" 이지만 분명 그 의미가 다르기에 명시해 두었습니다. (C 언어에는 정수진급 integral promotion 이 있기 때문에, 대부분 integral 이 나옵니다)

▷ 번역에 도움주신 분들

이 번역을 마치기까지 도움주신 분들이 많습니다. 일단, 하이텔 소프트 동호회 C/C++ 분과에서 제 고리타분한 질문에 답변을 주셨던 박태윤 (thilbong), 황용호 (조각달) 님께 감사드립니다. 또한 이 글을 볼리 없겠지만 (^^;), 뉴스그룹을 통한 제 질문에 성실히 답변해 준 Jerry Coffin, Martijn Lievaart, Mike Wahler, Jack Klein, Jim Gewin, Chris Dollin, Joona I Palaste, Alf Salte, Kaz Kylheku, Pansy, Morris M. Keesan, Richard Stamp (무순) 에게도 감사드립니다. 이렇게 도움 주신 분들이 없었다면 이 번역도 완성되지 못했을 겁니다.

▷ 잘못된 점 및 의문사항

마지막으로, 번역된 내용이나 그 외의 모든 부분에서 잘못된 부분이 보이거나, 의문점이 생기면 메일이나 게시판을 통해 제게 연락주시기 바랍니다. 부족한 실력이지만 최선을 다해서 대답해 드리겠습니다.

▷ A1. 소개 (Introduction)

이 메뉴얼은 1988년 10월 31일, "American Standard for Information Systems - C 프로그래밍 언어, 문서번호 X3.159-1989" 라는 제목으로 ANSI 에 제출된 초안을 기본으로 C언어에 대해서 서술한 것입니다. 이 메뉴얼은 제안된 표준안에 대한 해설서이지 표준 그 자체를 실어 놓은 것은 아닙니다. 하지만 C언어에 대한 확실한 길잡이가 될 수 있도록 최대한의 주의를 기울였습니다.

거의 대부분의 내용에서 초판에서 다룬바 있는 표준안의 개요를 따르고 있지만, 세부적인 사항에서는 다를 수도 있습니다. 하지만 문법에서 이름 몇 개를 바꾼 것 (renaming of a few production) 과, 토큰이나 전처리기의 문법 정의를 형식화 (formalizing) 하지 않은 것을 제외하면, 여기에 제시된 문법은 현재 사용하고 있는 표준과 동일합니다.

주석은 지금 보시는 것처럼 작은 글씨로 들여쓰기 해서 적을 것입니다. 주석을 통해서는 새로 정해진 ANSI-C 표준안이, 이 책의 초판 그리고 실제 컴파일러 지원 내용과 어떻게 다른지 보여줄 것입니다.

▷ A2. 어휘적 규정 (Lexical Convention)

프로그램은 파일 안에 저장된 하나 혹은 그 이상의 번역단위 (translation unit) 로 구성되며, A12 절에서 소개하는 여러 단계를 거쳐 기계가 알아들을 수 있도록 번역됩니다. **가장 첫번째 단계에서는, # 로 시작하는 지시자 (directive) 에 의한 저수준 어휘변환 (low-level lexical transformation) 이 일어나며, 매크로 정의와 전개 (macro definition and expansion) 가 수행**됩니다. A12 절에서 소개할 **전처리 기능이 완료되면, 프로그램은 '토큰'이라는 조각만으로 구성**되게 됩니다.

▷ A2.1 토큰 (Token)

토큰에는 **명칭 (identifier), 키워드 (keyword), 상수 (constant), 문자열 (string literal), 연산자 (operator), 분리자 (separator)** 의 여섯 가지가 있습니다. **공백 (blank), 수직 수평탭 (horizontal and vertical tab), 개행문자 (newline), FF (formfeed), 주석 (comment) 은 묶어서 공백문자 (white space)** 라고 하며, 이들은 토큰의 분리역할을 할 때를 제외하면 컴파일러에 의해 무시되어 버립니다. 붙어있는 명칭, 키워드, 상수의 구분을 위해서는 반드시 이 공백문자를 사용해야 합니다.

공백문자에 의해 토큰으로 분리된 문자열은, **컴파일러가 취할 수 있는 가장 긴 문자까지를 하나의 토큰으로** 취하게 됩니다.

> **▶ 역자 주**
> 컴파일러가 취할 수 있는 가장 긴 문자까지를 하나의 토큰으로 취하는 것을 "최대한 잘라먹기 규칙 (maximal munch rule)" 이라고 부르기도 합니다.

▷ A2.2 주석 (Comment)

/* 로 시작해서 */ 로 끝나는 문자들은 주석입니다. **주석은 중첩될 수 없으며, 문자상수나 문자열 내에서는 나올 수 없습니다**. 즉, 문자열 내의 주석은 문자열의 일부분입니다.

▷ A2.3 명칭 (Identifier)

명칭은 문자와 숫자로 구성됩니다. 첫번째 문자는 숫자일 수 없으며, 밑줄문자 _ 는 문자로 취급되고, 대소문자는 구분됩니다. **명칭의 길이는 제한이 없지만 내부명칭 (internal identifier) 은 최소 31 자만이 유효**합니다. 어떤 컴파일러에서는 더 많은 문자가 유효하기도 합니다. **내부명칭에는 외부연결 (external linkage) 을 갖지 않는 모든 명칭과**

전처리기용 매크로 명칭이 해당됩니다. (A11.2 참고) 외부연결을 가지고 있는 명칭은 좀 더 많은 제약이 있습니다. **어떤 컴파일러는 외부 명칭으로 최초의 6 자만을 유효하다고 지정하며, 대소문자 구분을 하지 않기도 합니다**.

▷ A2.4 키워드 (Keyword)

아래에 소개되는 명칭들은 키워드로 이미 예약되어 있어서 다른 용도로는 사용할 수 없습니다:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

어떤 컴파일러는 fortran 과 asm 도 키워드로 사용합니다.

> 키워드 const, signed, volatile 은 ANSI 표준에서 새로 등장한 것입니다. enum 과 void 도 초판에서는 없었던 것이며, 키워드로 예약되어 있으나 사용하지 않던 entry 는 사라졌습니다.

▷ A2.5 상수 (Constant)

C 에는 각각의 데이터형을 가지고 있는 여러 종류의 상수들이 있습니다. 기본 데이터형에 대해서는 A4.2 절을 참고하시기 바랍니다.

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

▷ A2.5.1 정수상수 (Integer Constant)

숫자들로 이루어진 정수상수는 숫자 0 으로 시작하는 경우에는 8 진수 숫자, 그외에는 10 진수 숫자로 취급됩니다. 8 진수 정수상수에는 숫자 8, 9 는 나올 수 없습니다. 0x 나 0X 뒤에 나오는 숫자는 16 진수 정수상수로 취급되며, 16 진수에서 숫자 10 부터 15 를 나타내기 위해서 문자 A (혹은 a) 부터 F (혹은 f) 를 사용합니다.

u 나 U 가 접미사로 붙는 정수상수는 unsigned 형임을 나타내며, l 이나 L 이 접미사로 붙는 정수상수는 long 형임을 나타냅니다. **정수상수의 데이터형은 그것의 형태(form), 값, 접미사에 따라 달라질 수 있습니다**. (데이터형에 대해서는 A4 절 참고) 정수상수가 **접미사 없이 10 진수 형태이면, int, long int, unsigned long int 중 첫번째로 가능한 데이터형으로 취급**됩니다. 만약 **접미사 없이 8 진수, 16 진수 형태이면 int, unsigned, long int, unsigned long int 중에서 첫번째로 가능한 데이터형으로 취급**됩니다. u 나 U 가 붙으면 unsigned int, unsigned long int 중 첫째로 가능한 데이터형으로, l 이나 L 이 붙으면 long int, unsigned long int 중 첫째로 가능한 데이터형으로 취급됩니다. 만약 UL (ul, Ul, uL) 접미사가 붙으면, unsigned long 형으로 취급됩니다.

큰 정수상수를 long 으로만 취급하던 초판에 비하면 이러한 데이터형 결정방식은 상당히 발전된 것입니다. 접미사 u 는 새로 등장한 것입니다.

▷ A2.5.2 문자상수 (Character Constant)

'x' 처럼 작은 따옴표 ' 에 둘러싸인 하나 혹은 그 이상의 문자들은 문자상수 입니다. 하나의 문자로 이루어진 문자상수의 값은 프로그램이 컴파일되는 컴퓨터 기종에 따른 문자코드에서의 숫자값입니다. **2 문자 이상의 문자상수 (multi-character constant) 값은 컴파일러마다 다르게 정의**합니다 (implementation-defined).

문자상수 내에는 작은 따옴표 ' 나 개행문자를 직접 담을 수 없습니다. 따라서 아래에 나열된 확장열 (escape sequence) 을 사용합니다.

개행문자	NL (LF)	\n	백슬러쉬	\	\\
수평탭	HT	\t	물음표	?	\?
수직탭	VT	\v	작은 따옴표	'	\'
백스페이스	BS	\b	큰 따옴표	"	\"
Carriage Return	CR	\r	8진수 숫자	ooo	\ooo
FormFeed	FF	\f	16진수 숫자	hh	\hh
벨소리	BEL	\a			

원하는 임의의 문자를 다루기 위해서 백슬러쉬 뒤에 1-3 자리의 8 진수 숫자를 쓰는 \ooo 확장열을 사용합니다. \ooo 확장열의 일반적인 예로는 널문자 (character NUL) 를 나타내는 \0 이 있습니다. 또한 백슬러쉬 뒤에 16 진수 숫자를 적어주는 \xhh 확장열도 있습니다. **적어주는 숫자의 자릿수에는 제한이 없지만, 만약 그 값이 문자코드의 최대값을 초과하는 경우에는 컴파일러의 행동이 따로 규정되어 있지 않습니다 (undefined). 8 진수나 16 진수 확장열의 경우 컴파일러가 char 형을 signed char 형으로 취급한다면, 캐스트 연산자에 의해 char 형으로 변환되듯이 부호확장 (sign-extend) 에 의해 변환**됩니다. 확장열을 나타낼 때 사용하는 **백슬러쉬 \ 뒤에 나오는 문자가 위에서 열거한 확장열이 아닐 경우, 그 행동은 정의되어 있지 않습니다 (undefined).**

어떤 컴파일러는 char 형으로 나타낼 수 없는 **확장된 문자코드를 지원**합니다. **확장된 문자코드의 상수는 그 상수앞에 L 을 붙여 L'x' 처럼 표현**하며, 이를 'wide character constant' 라고 부릅니다. 이러한 상수는 **표준 헤더파일인 <stddef.h> 에 정의된 정수형 (integral) 인 wchar_t 형을 갖습니다. wchar_t 형 문자상수에는 똑같은 확장열을 사용할 수 있으며, 확장열의 값이 wchar_t 형의 범위를 넘어가는 경우의 영향은 정의되어 있지 않습니다 (undefined).**

확장열 표현방법중 일부, 특히 16 진수 표현방법은 새로운 것입니다. 또한 확장된 문자코드 또한 새로운 것입니다. 미국이나 서유럽에서는 문자를 표현하는데 일반적으로 char 형을 사용하면 족하며, wchar_t 를 추가한 의도는 아시아의 언어를 지원하기 위한 것입니다.

▷ A2.5.3 부동상수 (Floating Constant)

부동상수는 정수부, 소수점, 소수부, e 혹은 E, 부호가 붙을 수 있는 정수 지수부와 생략 가능한 데이터형 접미사 (f, F, l, L 중 하나) 로 구성됩니다. 정수부와 소수부는 모두 숫자로 표시합니다. **정수부나 소수부중 하나는 생략할 수 있으며 (둘 다 생략하는 것은 불가능), 소수점이나 e 와 지수부 중 하나도 생략할 수 있습니다 (둘 다 생략하는 것은 불가능). 부동상수의 데이터형은 접미사에 의해 결정**됩니다; F (혹은 f) 는 float 형, L (혹은 l) 은 long double 형, 그 외의 경우에는 double 형으로 취급됩니다.

부동상수에 접미사를 붙이는 방법은 새로운 것입니다.

▷ A2.5.4 열거상수 (Enumeration Constant)

열거자 (enumerator) 로 선언된 명칭 (열거상수) 들은 int 형의 상수입니다.

▷ A2.6 문자열 (String Literal)

'문자열 상수' 라고도 불리는 문자열은 "..." 와 같이 큰 따옴표에 쌓인 문자들 입니다. 문자열의 데이터형은 '문자배열', 기억부류는 정적 (static) 이며, 주어진 문자들에 의해 초기화 됩니다. **동일한 문자열이 구별되어 사용되는지는 컴파일러마다 다르며 (implementation-defined), 문자열을 바꾸려는 프로그램의 행동에 대해서는 표준에서 정의한 바가 없습니다** (undefined).

인접해 있는 문자열은 하나의 문자열로 연결됩니다. **어떠한 방식으로든 문자열이 연결되면 문자열 뒤에는 널문자 '\0' 이 첨가**되어 프로그램이 문자열의 끝을 탐지할 수 있도록 해줍니다. 문자열 역시 개행문자나 큰 따옴표를 내용에 담을 수 없으며, 그러한 문자를 표현하기 위해서는 문자상수에서 알아본 확장열을 사용하면 됩니다.

문자상수에서처럼, **확장된 문자열은 문자상수 앞에 L 을 붙여 L"..." 과 같이 표현**합니다. **확장된 문자열은 'wide-character string' 이라고 부르며 데이터형은 wchar_t 의 배열형** 입니다. **보통의 문자열과 확장된 문자열의 결합에 대해서는 표준에서 정의해주지 않았습니다** (undefined).

동일한 문자열이 구별되어 사용될 필요가 없으며, 수정할 수 없다는 것, 인접한 문자열의 연결은 ANSI-C 표준에서 새로 정한 사실입니다. 또한 확장된 문자열인 wide-character string 도 새로운 것입니다.

▷ A3. 구문표기 (Syntax Notation)

이 메뉴얼에서 구문을 표기하는 방법은 다음과 같습니다. 구문의 종류 (syntactic categorie) 는 이탤릭체로, 단어나 문자 (literal word and character) 들은 타이프체로 표기됩니다. 선택할 수 있는 구문의 종류는 분리된 줄에 나열됩니다; 몇몇의 경우 길이가 짧은 구문 종류들이 다수 열거되는 경우에는, 한 줄에 표시한 후 "one of" (-중 하나) 라는 말을 붙입니다. 생략할 수 있는, 선택적인 종료자 (terminal) 나 종료자가 아닌 (non terminal) 기호에는 "opt" (optional, 선택적인) 라는 첨자를 붙였습니다.

▶ 역자 주

종료자 (terminator) 란, 문법 표기에서 왼쪽에 나올 수 없는 것입니다. 즉, 아래와 같은 문법이 있을 때,

primary-expression:
    identifier
    constant
    string
    (expression)

primary-expression 은 이미 왼쪽에 나왔으므로 종료자가 아닙니다. identifier, constant, string, (expression) 중 문법을 살펴보아서 primary-expression 처럼 왼쪽에 나오지 않는 것이 있다면 종료자가 되는 것입니다.

예를 들어,

{ expression opt }

는 중괄호에 쌓인 선택적인 수식을 의미합니다. 문법에 대해서는 A13 절에 요약해 두었습니다.

이 책의 초판과는 달리 여기에서 제시한 문법에는 수식 연산자의 우선순위와 결합순서를 명확히 밝혀 두었습니다.

▶ 역자 주

역자가 매우 골치가 아픈 관계로 (^^;) 위에서 이야기하는 이탤릭체나 타이프체의 구분을 하지 않았습니다. 그래도 보시는데 큰 어려움은 없을 겁니다. 글씨체의 구분이 필요하다면 A13 절의 요약된 문법을 보시기 바랍니다.

또한 정확한 기준없이 번역해서 오히려 혼란이 오지 않을까 하는 우려에서 문법은 번역하지 않고 원문에 나와있는 그대로를 실었습니다.

▷ A4. 명칭의 의미 (Meaning of Identifier)

명칭 혹은 이름 (name) 이라고 하는 것은 함수 (function), 구조체 (structure) · 공용체 (union) · 열거형 (enumeration) 의 택 (tag), 구조체 · 공용체의 멤버, 열거상수, 사용자정의 typedef 이름, 대상체 (object), 라벨 (label) 들을 의미합니다. 때때로 변수 (variable) 라고도 불리는 대상체는 기억장소의 어떠한 위치입니다. **대상체에는 크게 2 가지의 속성이 있습니다. 하나는 기억부류 (storage class) 이며, 다른 하나는 데이터형 (type) 입니다**. 기억부류는 대상체의 존속시간 (life time of storage) 을 결정해 주며, 데이터형은 그 대상체 안에 저장된 값의 의미를 결정합니다. **명칭 (identifier or name) 은 그것이 알려지는 프로그램의 범위를 나타내는 통용범위 (scope) 와, 다른 통용범위 내에 있는 같은 명칭이 동일한 대상체나 함수를 의미하는지를 결정하는 연결 (linkage) 이라는 속성을 가지고 있습니다**. 통용범위와 연결에 대해서는 A11 절에서 다룰 것입니다.

▷ A4.1 기억부류 (Storage Class)

기억부류에는 자동 (automatic) 과 정적 (static) 2가지가 있습니다. 대상체의 선언과 몇 개의 키워드가 기억부류를 결정해 줍니다. 자동 대상체 (automatic object) 는 블럭 (block, A9.3 참고) 에 귀속되며, 그 블럭을 빠져나오면 소멸되어 버립니다. **블록 내에서 선언된 대상체는 그것의 기억부류를 따로 지정해 주지 않아도 '자동 대상체' 가 됩니다. 물론 auto 라는 지정자로 자동 대상체 임을 명시해 줄 수도 있습니다. 가능하다면 CPU 의 빠른 레지스터 영역에 값을 저장하라는 의미의 register 지정자도 기억부류는 '자동' 입니다**.

정적 대상체 (static object) 는 하나의 블럭뿐 아니라 전체 블럭 (프로그램 전체) 에도 귀속될 수 있으며, 프로그램 제어가 함수나 블럭을 아무리 드나들어도 그 값을 유지합니다. **함수나 블록 내에서 정적 대상체는 static 이라는 키워드로 선언됩니다**. 함수 정의 (function definition) 와 같은 수준의, 모든 블럭 밖에서 선언된 대상체는 모두

정적 기억부류를 갖습니다. **static 키워드를 이용하면 대상체를 특별한 블록 내에서만 유효하도록 만들 수 있는데 이를 내부연결 (internal linkage) 이라고 합니다.** 또한 따로 기억부류를 지정해주지 않거나, extern 키워드를 이용해 **대상체가 프로그램 전체에서 유효하도록 만들 수도 있는데 이를 외부연결 (external linkage) 이라고 합니다.**

▷ A4.2 기본형 (Basic Type)

C 에는 여러 가지 기본형이 있습니다. 부록 B 에서 설명할 표준 헤더파일 <limits.h> 에는 각 컴파일러에서 지원하는 각 데이터형의 최대, 최소값이 정의되어 있습니다. 부록 B 에 주어진 값들은 최소의 허용치를 보여주는 것입니다.

문자형 (char) 으로 선언된 대상체는 문자코드의 어떤 문자도 저장할 수 있는 충분한 공간을 갖습니다. 만약 문자코드 중에 순수한 문자만을 char 형에 저장한다면, 그 값은 문자코드의 정수 숫자값과 동일하며 양수임이 보장됩니다. 문자형 변수에는 다른 임의의 값도 저장할 수 있지만 저장 가능한 값의 범위나 그 값의 부호는 컴파일러가 정하기 나름입니다 (implementation-defined).

unsigned char 에 의해 선언되는 부호 없는 문자형은 보통의 문자형과 동일한 크기를 가지며 다만 음수값을 표현할 수 없습니다. signed char 는 부호 있는 문자형임을 명백히 지정해 줄때 사용하는데, 이 역시 보통 문자형과 동일한 크기를 갖습니다.

> unsigned char 형은 이책의 초판에서는 나오지 않았지만 일반적으로 사용되는 것입니다. signed char 라는 지정법도 새로운 것입니다.

문자형 외에도 short int, int, long int 로 선언되는 3 가지 크기의 정수형 (integer) 이 있습니다. 보통의 int 형 대상체는 컴파일러에서 추천하는 가장 자연스러운 크기를 갖습니다. 다른 정수형은 특별한 경우에 사용하기 위한 것들입니다. 크기가 큰 정수형 (longer integer) 은 최소한 크기가 작은 정수형 (shorter integer) 이상의 크기를 제공하지만, **대부분의 컴파일러는 보통의 int 형이 short int 나 long int 와 같은 크기를 갖도록 제공**합니다. **특별한 지정자가 붙지 않는 이상 int 형은 부호 있는 값을 나타낼 수 있습니다.**

unsigned 로 선언되는 부호 없는 정수형은 2^n (n 은 표현에 사용되는 비트수) 으로 나머지 연산되기 때문에 부호 없는 값의 연산에서 오버플로우 (overflow) 가 발생할 수 없습니다. 부호 있는 대상체에 저장될 수 있는 양수값은, 대응하는 부호 없는 형에 저장할 수 있는 값의 부분집합이며, 겹치는 값의 내부표현은 동일합니다.

단정도 부동형 (float), 배정도 부동형 (double), 특수정도 부동형 (extra precision floating point, long double) 은 동일한 데이터형으로 취급될 수도 있지만, 후에 언급한 것일수록 더 정밀합니다.

> long double 은 새로운 것입니다. 초판에서는 long float 가 double 과 동일하다고 했었으나, **이제 long float 이라는 표현은 사용하지 않습니다.**

열거형은 정수값 (integral) 을 멤버로 갖는 특이한 데이터형이며, 각 열거와 관련이 있는 그 정수값들은 이름이 붙여진 상수들의 집합입니다. **열거형은 정수형 (integer) 처럼 행동하지만, 특별한 열거형 대상체에 그 상수와는 다른 것이 대입되거나 상수의 수식형태가 대입될 경우, 경고가 발생하는 것이 일반적입니다.**

위에서 알아본 데이터형은 숫자와 관련된 것들이기 때문에 산술형 (arithmetic type) 이라고 합니다. 부호에 관계없이 모든 크기의 char 형과 int 형, 열거형은 합해서 정수형 (integer type) 이라고 합니다. 또한 float, double, long double 은 부동형 (floating type) 이라고 합니다.

void 형은 값이 없는 것을 의미합니다. 이 데이터형은 리턴값 (반환값) 이 없는 함수의 반환 데이터형으로 사용됩니다.

▷ A4.3 유도형 (Derived Type)

기본형 외에도, C 에는 아래와 같은 방법으로 기본형으로부터 유도되는, 개념적으로 무한한 유도형이 있습니다.

주어진 데이터형의 배열
주어진 데이터형을 반환하는 함수
주어진 데이터형을 가리키는 포인터
다양한 데이터형의 대상체를 저장하는 구조체
다양한 데이터형 대상체중 어떤 하나를 저장할 수 있는 공용체

일반적으로 위와 같은 유도형 대상체의 구성은 재귀적으로 (recursively) 적용될 수 있습니다.

▷ A4.4 형한정어 (Type Qualifier)

대상체의 데이터형에 추가적으로 한정어를 적용할 수 있습니다. 대상체가 const 로 선언되면 이는 그 대상체의 값이 변할 수 없음을 의미합니다. **대상체가 volatile 로 선언되면 이는 그 대상체가 최적화에 적합하도록 특별한 특성을 가지고 있음을 말합니다. 어떠한 한정어라 할지라도 대상체의 산술계산적 특성이나 값의 범위에 영향을 주지 않습니다.** 한정어에 대해서는 A8.2 절에서 설명하였습니다.

▷ A5. 대상체와 좌변값 (Object and Lvalue)

대상체란 기억영역에 이름이 붙여진 (named) 부분을 일컫는 말입니다. 좌변값은 그러한 대상체를 참조하는 (referring) 수식입니다. **좌변값 수식의 대표적인 예는 적절한 데이터형과 기억부류를 가지고 있는 명칭 (identifier) 입니다.** 또한 **연산자 중에는 좌변값을 만들어주는 연산자도 있습니다.** 예를 들어 E 가 포인터형 수식이라고 한다면 *E 는 E 가 가르키는 대상체를 참조하는 좌변값 수식입니다. '좌변값 (Lvalue)' 이라는 이름은 E1 = E2 라는 대입식이 있을 때, 좌측 피연산자인 E1 이 반드시 좌변값 수식이어야 한다는 데서 유래한 것입니다. 각각의 연산자가 좌변값을 요구하는지 아니면 좌변값을 만들어주는지에 대해서는 각 연산자를 소개할 때 다루었습니다.

▷ A6. 변환 (Conversion)

피연산자에 의존적인 연산자는 어떠한 데이터형에서 다른 데이터형으로 피연산자의 값을 변환하기도 합니다. 이 절에서는 변환이 발생한 후의 결과에 대해서 설명하겠습니다. A6.5 절에서는 대부분의 연산자에서 수행되는 형변환에 대해 요약해 두었습니다. 그 요약된 내용에서 부족한 부분은 각 연산자를 설명할 때 보충될 것입니다.

▷ A6.1 정수진급 (Integral Promotion)

부호에 관계없이 문자형, short int 형, 정수 비트필드 (integer bit-field) 와 열거형 대상체는 정수형 (integer) 이 사용될 수 있는 수식이라면 사용될 수 있습니다. **int 형이 원래 데이터형의 모든 값을 표현할 수 있다면 그 값이 int형으로 변환됩니다. 만약 그렇지 않다면 unsigned int 형으로 변환**됩니다. **이 과정을 정수진급 이라고 합니다.**

▷ A6.2 정수변환 (Integral Conversion)

어떠한 정수형 (integer) 이라도 그 정수와 일치하는 가장 작은 양수값을 찾음으로써 (부호 없는 데이터형에서 표현할 수 있는 최대값보다 하나 큰 수와 나머지 연산 (modulo)), 주어진 부호 없는 형으로 변환할 수 있습니다. 2 의 보수 표현에서 이는 부호 없는 형의 비트구조가 더 작은 경우에는 왼쪽 절단 (left-truncation) 이, 더 넓은 경우에는 0 을 채운 부호 없는 값, 부호 확장된 (sign-extending) 부호 있는 값과 동일합니다.

> **▶ 역자 주**
> 위의 나머지 연산 운운하는 부분을 수식으로 설명하면 (unsigned 의 최대값 + 1 + 변환할 값) % (unsigned 의 최대값 + 1) 입니다.

정수형 (integer) 이 부호 있는 데이터형으로 변환되는 경우, 새로운 데이터형이 그 값을 표현할 수 있다면 그 값은 변하지 않습니다. 표현이 불가능한 경우에는 결과가 컴파일러마다 다릅니다 (implementation-defined).

▷ A6.3 정수와 부동소수 (Integer and Floating)

부동형의 값이 정수값 (integral) 으로 변환될 때, 소수부분은 잘려 나갑니다. **만약 변환된 값을 정수값 (integral) 으로 나타낼 수 없다면 그때의 행동은 알 수 없습니다 (undefined). 특히 음수 부동소수가 부호 없는 정수값 (integral) 으로 변환될 경우에 대해서는 명시된 바가 없습니다 (unspecified).**

정수값 (integral) 이 부동형으로 변환될 때, **그 값이 부동형으로 표현 가능하지만, 정확히 표현할 수 없을 경우에는 하나 크거나 작은 값이 결과가 됩니다. 만약 결과가 표현범위 밖으로 벗어난다면 결과를 알 수 없습니다 (undefined).**

▷ A6.4 부동형 (Floating Type)

작은 정밀도의 부동형이 같거나 더 큰 정밀도를 갖는 부동형으로 변환된다면 값은 변하지 않습니다. 반대로 **큰 정밀도의 부동형이 작은 정밀도의 부동형으로 변환될 때는, 결과가 표현 가능한 범위 내에 있다면 결과값은 (정밀도가 더 작으므로) 더 크거나 작은 값이 됩니다. 만약 표현 가능한 범위를 벗어나 버리면, 그때의 행동은 정의되어 있지 않습니다 (undefined).**

▷ A6.5 산술변환 (Arithmetic Conversion)

많은 연산자들이 변환을 행하고 비슷한 방법으로 연산 결과를 생성합니다. 그 효과는 피연산자를 공통된 데이터형으로 만들어 주는 것이며, 결과 또한 같은 데이터형이 됩니다. 이러한 것을 **일반적 산술변환 (usual arithmetic conversion)** 이라고 합니다.

> 우선, 피연산자에 long double 형이 있으면 나머지도 long double 로 바꿉니다.
> 그렇지 않으면, 피연산자에 double 형이 있으면 나머지도 double 로 바꿉니다.
> 그렇지 않으면, 피연산자에 float 형이 있으면 나머지도 float 로 바꿉니다.
> 그렇지 않으면, 정수진급이 각 피연산자에 수행된 후에, 피연산자에 unsigned long int 가 있으면 나머지도 unsigned long int 로 바꿉니다.
> 그렇지 않으면, 한 피연산자가 long int 이고 다른 피연산자는 unsigned int 라면, long int 가 unsigned int 를 모두 표현할 수 있는지가 중요합니다. 가능하다면 unsigned int 가 long int 로 변환되고, 불가능하다면 모두 unsigned long int 로 변환됩니다.

그렇지 않으면, 피연산자에 long int 가 있으면 나머지도 long int 로 바꿉니다.
그렇지 않으면, 피연산자에 unsigned int 가 있으면 나머지도 unsigned int 로 바꿉니다.
그렇지 않으면, 두 피연산자 모두 int 형입니다.

이곳에는 두가지 변화가 생겼습니다. 우선 **float 형의 연산이 double 형으로 변환되지 않고 단정도내에서 해결**된다는 점입니다. 둘째는 **작은 크기의 부호없는 형과 큰 크기의 부호있는 형이 만났을 때, 부호없는 형의 특성이 결과의 데이터형으로 전파되지 않는다는 점**입니다. **초판에서는 부호없는 형이 항상 결과를 지배했었습니다**. 새로운 규칙은 다소 복잡하기는 하지만, 부호없는 형이 부호있는 형과 만났을 때 발생하는 예상 밖의 결과를 줄였습니다. **하지만 같은 크기의 부호없는 형과 부호있는 형이 만난다면 결과는 여전히 예상 불가능 합니다**.

▷ A6.6 포인터와 정수 (Pointer and Integer)

정수수식 (integral) 은 포인터에 더하거나 뺄 수 있습니다. 이러한 경우 정수수식 (integral) 은 덧셈 연산자에서 이야기할 것처럼 (A7.7) 변환됩니다.

같은 배열 내에서, 같은 데이터형을 가리키는 두 포인터는 서로 뺄 수 있습니다. 결과는 뺄셈 연산자에서 설명할 것처럼 (A7.7) 정수형 (integer) 으로 변환됩니다.

0 값을 갖는 정수 상수수식 (integral) 이나 void * 로 변환된 그러한 수식은 캐스트, 대입, 비교에 의해 어떤 형의 포인터로도 변환될 수 있습니다. 이것은 같은 데이터형의 null 포인터를 생성합니다. 하지만 함수나 대상체를 가리키는 포인터와는 같을 수 없습니다.

그 외의 명확한 포인터 변환은 허가는 되어 있지만 컴파일러에 따라 약간씩 다른 면모를 보입니다. 포인터의 변환은 캐스트 연산자나 확실한 형변환 연산자 (type-conversion operator) 에 의해 서술되어야 합니다. (A7.5 와 A8.8 참고)

포인터는 그것을 담을 수 있는 충분한 크기의 정수형 (integral) 으로 변환될 수 있습니다. **이 때 요구되는 정수형 (integral) 의 크기는 각각 다릅니다** (implementation-dependent). **매핑 함수 (mapping function) 에 대한 사항은 컴파일러에 따라 다릅니다** (implementation-dependent).

▶ 역자 주
매핑 함수 (mapping function) 란, 포인터가 가리키는 메모리상의 주소를 정수수치 (integral) 로 바꾸는 방법을 의미합니다. 같은 주소라고 해도 컴파일러마다 그 표현방법이 다릅니다.

정수형 대상체 (integral) 는 명백히 포인터로 변환될 수 있습니다. **정수형 (integral) 의 크기가 포인터를 충분히 담을 수 있다면, 포인터로부터 변환된 정수값 (integral) 이 다시 같은 포인터로 변환되지만, 그렇지 않다면 결과는 컴파일러마다 다릅니다** (implementation-dependent).

▶ 역자 주
C90 표준안에서는, 어떠한 경우라도 정수와 포인터간의 변환은 컴파일러가 정의한다고 (implementation-defined) 명시하고 있습니다.

어떠한 데이터형의 포인터는 다른 형의 포인터로 변환될 수 있습니다. **만약 본래의 포인터가 적당히 정렬되지 않은 기억장소의 대상체를 참조하고 있었다면, 변환된 결과로 나오는 포인터의 주소에 예외적 상황 (addressing exception) 이 발생할 수도 있습니다. 포인터는 변화 없이 기억장소의 정렬을 덜 요구하는 데이터형의 포인터로 변환될 수**

있다는 것이 보장됩니다. **'정렬 (alignment)' 이라는 용어 자체가 명확히 규정될 수는 없지만 (implementation-dependent), char 형의 대상체는 기억장소 정렬에 대해 가장 엄격하지 않습니다**. A6.8 절에서 설명하겠지만, 변화 없이 어떤 포인터이든 void * 로 변환되었다가 되돌려질 수 있습니다.

▶ **역자 주**

기억장소 정렬의 한가지 예를 들어 보겠습니다. 바이트 기반 (byte-based) 의 메모리를 갖고 있는 어떤 기종에서는, 2 바이트 크기의 short int 형은 짝수 번지에, 4 바이트 크기의 long int 형은 4의 배수 번지에 저장했을 때 가장 효과적으로 다루어집니다. 경우에 따라서는 정렬이 되지 않은 값은 아예 다룰 수 없는 경우도 있습니다.

포인터는, 가리키는 대상체에 적용되는 한정어의 첨가나 제거 (addition or removal) 를 제외하면 같은 데이터형의 포인터로 변환될 수 있습니다. 만약 **한정어가 첨가되면, 새 한정어에 의해 적용되는 제한을 제외하면 예전의 포인터와 동일**합니다. 만약 **한정어가 제거되면, 기초가 되는 대상체에서의 작업은 그것의 실제 선언에 있는 한정어에 귀속**됩니다.

▶ **역자 주**

우선, 한정어가 첨가되는 경우의 예를 들어보겠습니다.

```
char char_var;
char *pointer_to_char;
const char *pointer_to_const;

pointer_to_char = &char_var;
*pointer_to_char = 'a';          /* 가능 */
pointer_to_const = pointer_to_char;
*pointer_to_const = 'a';         /* 컴파일시 에러 */
```

위의 예에서, pointer_to_char 가 pointer_to_const 로 대입되면서, 형변환에 의해 한정어가 첨가됩니다. 분명 pointer_to_char 와 pointer_to_const 는 const 가 붙지 않은 일반 char 형 변수를 가르키고 있지만, 한정어의 영향을 받아 pointer_to_const 에는 값을 대입할 수 없습니다.

다음은, 한정어가 제거되는 경우의 예입니다.

```
const char const_char;
char *pointer_to_char;
const char *pointer_to_const;

pointer_to_const = &const_char;
pointer_to_char = (char *) pointer_to_const;
*pointer_to_char = 'a';          /* 실행시 에러 */
```

pointer_to_const 를 캐스트 없이 pointer_to_char 로 변환할 수는 없습니다. 캐스트를 사용하면 pointer_to_const 의 형한정어가 제거되어 pointer_to_char 로 변환은 되지만, 그렇다고 const_char

의 const 특성이 없어지는 것은 아닙니다. 따라서 일반적인 경우라면 pointer_to_char 의 값을 변경하는 것이 가능하겠지만, 여기서는 불가능합니다. 특히 주의해야 할 점은, 대부분의 컴파일러가 위처럼 형한정어가 없는 pointer_to_char 에 대입이 이루어지는 것을 적법한 것으로 판단한다는 점입니다. 하지만, 대부분 const 로 한정이 되는 대상체는 롬 (ROM) 에 존재할 가능성이 크기 때문에, 실행시 에러 (run-time error) 가 발생하게 됩니다.

마지막으로 함수를 가리키는 포인터는 다른 형의 함수 포인터로 변환될 수 있습니다. **변환된 함수 포인터에 의한 함수 호출에 대해서는 컴파일러마다 다르지만 (implementation-dependent), 변환되었던 포인터가 그것의 원래 데이터형으로 돌아간다면 결과는 변환전의 포인터와 똑같습니다.**

▶ 역자 주

C90 표준안에 따르면, 변환된 함수 포인터를 통해 함수가 호출될 때, 그 함수의 데이터형이 실제로 호출되는 함수의 데이터형과 호환되지 않는 경우, 그 행동은 정의되어 있지 않습니다 (undefined).

▷ A6.7 Void

void 대상체의 (존재하지 않는) 값은 어떠한 방법으로도 사용될 수 없으며, 은연중이든 아니든 void 형이 아닌 다른 값으로 변환될 수 없습니다. void 수식은 값이 없다는 것을 나타내기 때문에 오직 값이 필요 없는 곳에만 사용할 수 있습니다. 예를 들면, 수식문 (expression statement) 이나 쉼표연산자의 좌측 피연산자가 있습니다.

수식은 cast 에 의해 void 형으로 변환될 수 있습니다. 예를 들면, 수식문으로 사용되는 함수 호출의 결과값을 무시해 버리는 경우입니다.

'void' 는 이 책의 초판에서는 나오지 않았지만, 그 이후 일반적으로 사용되는 것입니다.

▷ A6.8 void 형 포인터 (Pointer to Void)

어떠한 대상체를 가리키는 포인터이든 가지고 있는 정보의 손실 없이 void * 형으로 변환될 수 있습니다. void 형 포인터로 변환되었다가 다시 원래 포인터로 돌아온다면 본래의 포인터와 아무런 차이가 없습니다. A6.6 절에서 설명한, 확실한 cast 를 필요로 하는 포인터와 포인터간의 변환과는 달리, **void 형 포인터는 다른 포인터에 대입되는 것과 비교하는 것도 가능**합니다.

void 형 포인터에 대한 이러한 설명은 새로운 것입니다; 전에는 char 형 포인터가 일반적인 포인터 역할을 수행했습니다. ANSI 표준안은, 다른 포인터의 혼용에는 명백한 캐스트를 요구하는 것과는 달리, 다른 대상체 포인터와 void 형 포인터와의 대입, 비교 (relational) 에는 많은 아량을 베풀었습니다.

▷ A7. 수식 (Expression)

연산자의 우선순위 (precedence) 는 앞으로 연산자를 열거하는 순서와 동일합니다. (연산순위가 높은 것부터 설명합니다) 따라서 + 연산자 (A7.7) 에서 피연산자로 언급되는 수식들은 A7.1 - A7.6 사이에서 정의된 수식들입니다. 같은 절에서 소개하는 연산자들은 동일한 연산순위를 가지고 있습니다. 연산자의 결합순서 (associativity) 는 각 연산자를 설명하면서 함께 적어 두었습니다. A13 절에 나오는 문법에는 우선순위와 결합순서를 함께 표시하였습니다.

연산자의 우선순위와 결합순서는 명백히 정해져 있지만, 수식의 평가순서 (the order of evaluation of expression) 는 몇몇 정해진 예외를 제외하면, 그 수식이 부작용 (side effect) 을 낳는다고 해도 구체적으로 정의된 바가 없습니다 (undefined). 이는 연산자의 정의가 피연산자를 특별한 순서로 평가한다고 보장하지 않는 이상, 컴파일러 맘대로 피연산자들을 평가하고 심지어는 순서사이에 끼워넣을 수도 있다는 의미입니다. **하지만 각 연산자는 피연산자에 의해 생성된 값들을 수식의 구문분석 (parsing) 에 따라 묶습니다**.

▶ 역자 주

구문분석 (parsing) 이란, 컴파일러 혹은 인터프리터가 프로그램을 이해해서 기계어로 번역하는 과정중의 한단계로, 각 문장의 문법적 구성·구문을 분석하는 과정을 말합니다. 더 자세한 내용은 A13 절을 참고하시기 바랍니다.

이러한 규칙은, 수학적으로는 교환법칙 (commutative) 과 결합법칙 (associative) 에 아무 문제가 없지만 컴퓨터 계산상으로는 실패할 수 있는, 수식의 정리에 제한이 가해진다는 것을 의미합니다. 하지만 이러한 변화는 정밀도 한계 근처의 부동소수와 오버플로우가 발생할 수 있는 상황에서만 영향을 주게 됩니다.

▶ 역자 주

수학적으로는 (a + b) / c 와 (a / c) + (b / c) 가 완전히 동일합니다. 하지만 컴퓨터로 계산할 때는 각 계산과정에서 자료형의 표현 가능한 유효숫자가 제한되어 있기 때문에 그렇지 못합니다. 하지만 이러한 영향은 극히 미미하고 오버플로우가 일어날만한 상황에서야 보인다는 뜻입니다. 이러한 이유로 예전에는 이 영향을 최소화하기 위해 프로그래머가 (a / c) + (b / c) 로 써주어도 컴파일러는 (a + b) / c 로 계산하도록 하기도 했었지만, ANSI 에서는 이를 금지하고 나와있는 수식 그대로를 문법분석 (parsing) 에 따라 계산하도록 했다는 말입니다.

오버플로우 (overflow), 나눗셈 검사 (divide check), 수식평가에서 다른 예외적인 상황들을 다루는 것에 대해서는 언어에서 정의한 바가 없습니다 (undefined). 대부분의 C 컴파일러들은 부호 있는 정수수식 (integral) 과 대입문 평가에서 오버플로우를 무시합니다. 하지만 이러한 행동이 보장되지는 않습니다. 0 으로 나누는 경우 (treatment of division by 0) 와 모든 부동 소수점의 예외적 상황 (all floating-point exception) 은 컴파일러마다 다릅니다; 때때로 비표준 라이브러리 함수에 의해 이러한 것들이 조정되기도 합니다.

▷ A7.1 포인터 생성 (Pointer Generation)

수식이나 부분수식 (subexpression) 의 데이터형이 "어떠한 데이터형 T 의 배열형" 이면, 그 값은 그 배열의 첫번째 대상체를 가리키는 포인터이며 수식의 데이터형은 T형 포인터로 대체됩니다. 만약 수식이 **단항 번지연산자 & 나 ++, --, sizeof 의 피연산자이거나, 대입연산자, . 연산자의 좌측 피연산자라면 이러한 형변환은 일어나지 않습니다. 유사하게 "T형을 반환하는 함수"형의 수식은 & 연산자의 피연산자로 사용되는 경우를 제외하면 "T형을 반환하는 함수를 가리키는 포인터"로 변환**됩니다.

▷ A7.2 기본수식 (Primary Expression)

기본 수식은 명칭, 상수, 문자열, 괄호에 쌓인 수식입니다.

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

명칭은 아래에서 설명하는 것처럼 적당히 선언되면 기본수식이 됩니다. 명칭의 데이터형은 선언에 의해 규정됩니다. **명칭이 대상체를 참조하고, 명칭의 데이터형이 산술형, 구조체, 공용체, 포인터라면 좌변값 (lvalue) 이 됩니다**.

상수 역시 기본수식이며, 그것의 데이터형은 A2.5 절에서 설명한 것처럼 형태에 의존합니다.

문자열 역시 기본수식입니다. **데이터형은 본래 문자배열 이지만, A7.1 절에서 주어진 규칙을 따라 일반적으로 문자형 포인터로 변환되고 결과는 문자열의 첫번째 문자를 가리키는 포인터**입니다. **이러한 변환은 확실한 초기치 (initializer) 에서는 일어나지 않습니다**; A8.7 참고

괄호에 쌓인 수식은 괄호가 없는 수식의 데이터형, 값과 동일한 속성을 갖는 기본수식 입니다. **괄호의 존재는 수식의 좌변값 여부에는 영향을 주지 않습니다**.

▷ A7.3 접미수식 (Postfix Expression)

접미수식에서 연산자는 왼쪽에서 오른쪽으로 묶입니다.

```
postfix-expression:
        primary-expression
        postfix-expression [ expression ]
        postfix-expression ( argument-expression-list opt )
        postfix-expression . identifier
        postfix-expression -> identifier
        postfix-expression ++
        postfix-expression --

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

▷ A7.3.1 배열참조 (Array Reference)

수식 뒤에, 대괄호 [] 에 둘러싸인 수식이 붙는 접미수식은 첨자 (subscript) 에 의한 배열참조 수식입니다. **두 수식 중 하나는 반드시 T형 포인터여야 하며, 다른 하나는 정수 (integral)** 여야 합니다. **이 배열첨자 수식의 데이터형 (결과) 은 결국 T 형이 됩니다. 수식 E1[E2] 는 정의에 의해 *((E1)+(E2)) 와 동일합니다**. 자세한 사항은 A8.6.2 절을 참고하시기 바랍니다.

▷ A7.3.2 함수호출 (Function Call)

함수호출은 함수 지정자 (function designator) 라고 불리는 접미수식으로, 뒤에 괄호에 쌓여 쉼표에 의해 분리되는 생략 가능한 대입수식 리스트가 따라오며, 이 수식은 함수의 매개변수 (parameter) 로 대치됩니다. **만약 현재의 통용범위에서 선언이 존재하지 않는 명칭이 들어간 수식이 있다면 함수호출을 담고 있는 가장 안쪽 블럭에 아래와 같은 선언이 주어진다고 간주됩니다**.

extern int identifier();

함수호출 접미수식은 (가능한 암시적인 선언과 A7.1 절에서 설명한 포인터 생성 후에) **T 형을 반환하는 함수 포인터여야 하며, 함수 호출의 값은 T 형이 됩니다.**

> 초판에서는 "함수"라는 데이터형이 매우 제한되었고, 함수 포인터를 통해 호출할 때 명확한 * 연산자가 필요했었습니다. ANSI 표준안은, 함수와 함수 포인터를 통한 함수호출에 같은 문법을 적용하는 현존하는 컴파일러의 관례를 따랐습니다. 예전의 문법은 아직 사용할 수 있습니다.

인자 (argument) 는 함수호출에 의해 전달되는 수식에 사용되는 용어입니다; 매개변수 (parameter) 는 함수 정의에 의해 받는 입장에서, 혹은 함수선언에 기술되어 있는 입력 대상체 (혹은 그것의 명칭) 에 대해 사용되는 용어입니다. "실매개변수 (actual parameter)" 와 "형식매개변수 (formal parameter)" 는 때로는 같은 뜻으로 쓰이기도 합니다.

▶ 역자 주

말은 복잡하고 많지만, 결국 '인자 (실매개변수)' 와 '매개변수 (형식매개변수)' 의 용어 설명을 하고 있는 것에 불과합니다.

함수호출을 준비하는 과정에서 각 인자의 복사본이 만들어집니다; 모든 인자 전달은 값에 의해 (by value) 이루어집니다. 함수는 인자수식의 복사본인 매개변수 대상체의 값을 바꿀 수 있습니다. 하지만 값의 변화는 인자 (argument) 의 입장에서는 아무런 변화를 주지 못합니다. 그러나 인자를 포인터 형태로 전달해 함수가 인자의 값을 간접적으로 변경할 수는 있습니다.

함수가 선언되는 방법에는 2 가지가 있습니다. 현대적인 방식 (new style) 에서 매개변수의 데이터형은 명백히 드러나고 함수 데이터형의 일부분입니다; 이러한 선언을 함수원형 (function prototype) 이라고 부릅니다. 고전적인 방식 (old style) 에서 매개변수는 명시되지 않습니다. 함수선언은 A8.6.3 과 A10.1 절에서 더 자세히 다루겠습니다.

만약 호출되는 통용범위의 함수선언이 고전적인 방식이면, 기본적인 인자진급 (argument promotion) 이 아래와 같이 적용됩니다: 정수진급 (integer promotion) 이 각각의 정수형 인자 (integral) 에 수행되고, float 형 인자들은 double 형으로 변환됩니다. 인자의 개수가 함수정의에 있는 매개변수의 개수와 맞지 않거나, 진급후의 인자의 데이터형이 대응하는 매개변수의 데이터형과 맞지 않는 경우, 함수호출의 영향은 정의되어 있지 않습니다 (undefined). 데이터형 일치 (agreement) 에 대한 사항은 함수의 정의가 현대적인지 고전적인지에 따라 달라집니다. 만약에 함수의 정의가 고전적이라면, 인자의 진급된 데이터형과 매개변수의 진급된 데이터형 사이에 비교가 이루어집니다. 만약 함수정의가 현대적인 방식이라면, 인자의 진급된 데이터형과 진급이 이루어지지 않은 매개변수 자체의 데이터형이 같아야 합니다.

함수선언이 현대적 방식이면, 인자가 함수원형의 대응하는 매개변수의 데이터형으로 대입되듯이 변환됩니다. 인자의 개수는 함수선언의 매개변수가 생략부호 (, ...) 로 끝나지 않는 이상, 명확히 기술된 매개변수의 개수와 같아야 합니다. 매개변수가 생략부호로 끝나는 경우, 인자의 개수는 매개변수와 같거나 혹은 더 많아야만 합니다; 명확히 기술된 매개변수의 개수를 넘어서는 인자들에는 앞에서 이야기한 기본적인 인자진급 (default argument promotion) 이 적용됩니다. 만약 함수의 정의가 고전적 방식이라면, 호출 시에 보이는 함수원형의 매개변수 데이터형은 인자진급이 일어난 후의 함수정의에서의 대응하는 매개변수와 같아야 합니다.

이러한 규칙들은 현대적 방식과 고전적 방식을 모두 수용하기 때문에 복잡합니다. 두 방식을 혼합해 사용하는 것은 가능하면 피해는 것이 좋습니다.

인자의 평가순서 (the order of evaluation of argument) 는 따로 정해져 있지 않습니다 (unspecified); 컴파일러마다 다르다는 의미입니다. 하지만 **인자와 함수지정자 (function designator) 는 그 부작용 (side effect) 을 포함해, 함수에 진입하기 전에 모두 평가됩니다. 어떠한 함수라도 재귀호출 (recursive call) 이 가능**합니다.

▷ A7.3.3 구조체 참조 (Structure Reference)

점 뒤에 명칭이 따라오는 수식은 접미수식 입니다. **첫번째 피연산자 수식은 반드시 구조체나 공용체여야 하고, 명칭은 구조체나 공용체의 멤버 이름이어야 합니다. 수식의 값은 구조체나 공용체의 멤버이고, 데이터형은 그 멤버의 데이터형** 입니다. **첫번째 수식이 좌변값이고, 두번째 수식의 데이터형이 배열형이 아니라면 수식은 좌변값이 됩니다.**

화살표 (->) 와 명칭이 뒤에 붙는 수식은 접미수식 입니다. **첫번째 수식은 구조체나 공용체를 가리키는 포인터여야 하고, 명칭은 구조체나 공용체의 멤버여야 합니다.** 결과는 포인터 수식이 가리키는 구조체나 공용체의 멤버를 참조합니다; **결과는 데이터형이 배열형이 아니라면 좌변값입니다.**

수식 E1->MOS 는 (*E1).MOS 와 같습니다. 구조체와 공용체는 A8.3 절에서 논의합니다.

초판에서 이러한 수식에 있는 멤버 이름이 접미 수식에서 언급된 구조체나 공용체에 속해야만 한다는 것이 이미 규칙으로 되어 있었습니다. 그러나 이 규칙이 제대로 지켜지지 않아서 현재의 컴파일러나 ANSI 에서는 이 규칙을 강력히 지키도록 하고 있습니다.

▷ A7.3.4 접미증가 (Postfix Incrementation)

++, -- 연산자가 뒤따라오는 수식은 접미수식 입니다. **수식의 값은 피연산자의 값**입니다. **값이 정해진 후에, 피연산자는 1 만큼 증가(++)되거나, 감소(--)됩니다. 피연산자는 좌변값이어야 합니다**; 피연산자와 연산의 세세한 것들에 대한 제약은 덧셈연산자 (A7.7) 와 대입연산자 (A7.17) 를 참고하시기 바랍니다. **결과는 좌변값이 아닙니다.**

▷ A7.4 단항연산자 (Unary Operator)

단항연산자가 있는 수식은 오른쪽에서 왼쪽으로 묶입니다.

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

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

▷ A7.4.1 접두 증가연산자 (Prefix Incrementation Operator)

++ 나 -- 연산자가 앞에 붙는 수식은 단항수식 (unary expression) 입니다. 피연산자는 1 만큼 증가(++)되거나 감소(--)됩니다. **수식의 값은 증감 처리된 값입니다. 피연산자는 좌변값이어야 합니다**; 피연산자와 연산자의 세세한 것들에 대한 제약은

덧셈연산자 (A7.7) 와 대입연산자 (A7.17) 를 참고하시기 바랍니다. **결과는 좌변값이 아닙니다.**

▷ A7.4.2 번지연산자 (Address Operator)

단항 & 연산자는 피연산자의 주소(번지)를 취합니다. **피연산자는 비트필드 (bit-field) 나 register 로 선언된 대상체를 참조하지 않는 좌변값이거나 혹은 함수형이어야 합니다. 결과는 좌변값에 의해 참조되는 대상체나 함수의, 포인터**입니다. **피연산자의 데이터형이 T 라면, 결과는 T 형 포인터**입니다.

▷ A7.4.3 간접지정 연산자 (Indirection Operator)

단항 * 연산자는 간접지정을 의미하고, 피연산자가 가리키는 대상체나 함수를 반환합니다. **만약 피연산자가 산술형, 구조체, 공용체, 포인터를 가리키는 포인터라면 결과는 좌변값**입니다. 수식의 데이터형이 "T 형 포인터" 라면 결과의 데이터형은 T 형입니다.

▷ A7.4.4 단항 플러스 연산자 (Unary Plus Operator)

단항 + 연산자의 **피연산자는 산술형이어야 하고** 결과는 피연산자의 값 그대로 입니다. **정수형 피연산자 (integral) 는 정수진급 (integral promotion) 을 거칩니다. 결과의 데이터형은 진급된 (promoted) 피연산자의 데이터형**입니다.

> 단항 + 연산자는 ANSI 표준안에서 새로운 것입니다. 단항 - 연산자와의 대칭성을 위해 새롭게 추가되었습니다.

▷ A7.4.5 단항 마이너스 연산자 (Unary Minus Operator)

단항 - 연산자의 **피연산자는 산술형이어야 하고** 결과는 피연산자의 음수값 입니다. **정수형 피연산자 (integral) 는 정수진급 (integral promotion) 을 거칩니다. 부호 없는 값의 음수값 (the negative of an unsigned quantity) 은 진급된 데이터형의 최대값에서 진급된 값을 빼고 1 을 더해서 구합니다; 하지만 0 의 음수값은 0 입니다. 결과의 데이터형은 진급된 피연산자의 데이터형**입니다.

▷ A7.4.6 1 의 보수 연산자 (One's Complement Operator)

단항 ~ 연산자의 **피연산자는 정수형 (integral) 이어야 하고**, 결과는 피연산자의 1 의 보수입니다. **피연산자에는 정수진급이 일어납니다.** 만약 **피연산자가 부호 없는 형이면, 결과는 진급된 데이터형의 최대값에서 그 값을 빼서 구합니다.** 만약 **부호 있는 피연산자라면 대응하는 부호 없는 형으로 변환해서 ~ 연산자를 적용한 후에 다시 부호 있는 형으로 되돌려 구합니다. 결과의 데이터형은 진급된 피연산자의 데이터형**입니다.

▷ A7.4.7 논리 부정 연산자 (Logical Negation Operator)

! 연산자의 **피연산자는 산술형이거나 포인터여야 하고**, 결과는 피연산자가 0 이면 1 이고, 아니라면 0 입니다. **결과는 int 형입니다.**

▷ A7.4.8 sizeof 연산자 (Sizeof Operator)

sizeof 연산자는 피연산자 데이터형의 대상체를 저장하는데 필요한 바이트수를 계산해 줍니다. **피연산자는 계산되지 않은 (not evaluated) 수식이거나 괄호에 쌓인 데이터형 이름**입니다. **sizeof 연산자가 char 에 적용된다면 결과는 1 입니다; 배열에 적용했을 때는 배열의 총 바이트수가 됩니다. 구조체나 공용체에 적용했을 때의 결과는, 대상체를 배열의 요소로 만들기 위해 필요한 빈공간 (padding) 을 포함한 바이트수 입니다**: n 개의 요소 (element) 를 갖는 배열의 크기는 한 요소의 크기에 n 을 곱한 것입니다. **sizeof 연산자는 함수나 불완전한 데이터형 (incomplete type), 비트필드에는**

적용될 수 없습니다. 연산의 결과는 부호 없는 정수상수 (integral) 이지만, 정확한 데이터형은 컴파일러마다 다릅니다 (implementation-defined). 표준 헤더파일 <stddef.h> 에는 이 데이터형이 size_t 라는 데이터형으로 정의되어 있습니다. (부록 B 참고)

▷ A7.5 캐스트 (Cast)

데이터형의 이름이 괄호에 쌓여 선행되는 단항 수식은, **수식의 값**을 그 데이터형으로 변환해 줍니다.

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

이러한 구조를 캐스트 (cast) 라고 합니다. 데이터형명 (type name) 은 A8.8 절에서 설명합니다. 변환의 영향은 A6 절에서 설명했습니다. **캐스트된 수식은 좌변값이 아닙니다**.

▷ A7.6 곱셈연산자 (Multiplicative Operator)

*, /, % 같은 곱셈연산자는 왼쪽에서 오른쪽으로 묶입니다.

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

*** 와 / 의 피연산자는 산술형이어야 합니다; % 의 피연산자는 정수형 (integral)** 이어야 합니다. **각 피연산자에는 일반적 산술변환 (usual arithmetic conversion) 이 수행**되며, 그로 인해 결과의 데이터형이 예측됩니다.

이항연산자 * 는 곱셈을 의미합니다.

이항연산자 / 는 첫번째 피연산자가 두번째 피연산자로 나누어졌을 때의 몫을, % 연산자는 나머지를 구합니다; **두번째 피연산자가 0 인 경우 (0 으로 나누는 경우), 결과는 정의되어 있지 않습니다** (undefined). **두번째 피연산자가 0 이 아니라면 (a / b) * b + a % b 가 a 임이 보장**됩니다. **피연산자가 둘 다 음수가 아니라면, 나머지도 음수가 아니며 제수보다 작습니다; 피연산자 중 하나라도 음수가 있다면, 나머지의 절대값이 제수의 절대값보다 작다는 사실만이 보장**됩니다.

▷ A7.7 덧셈연산자 (Additive Operator)

덧셈연산자 +, - 는 왼쪽에서 오른쪽으로 묶입니다. 만약 **피연산자가 산술형이라면 일반적 산술변환이 수행**됩니다. 산술형 외에도 각 연산자마다 부가적인 데이터형 가능성이 있습니다.

▶ 역자 주

'부가적인 데이터형 가능성' 이란 덧셈연산자와 뺄셈연산자에 산술형 외에 포인터형을 피연산자로 둘 수 있다는 뜻입니다.

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

+ 연산자의 결과는 피연산자의 합입니다. **배열의 어떠한 대상체를 가리키는 포인터와 정수값 (integral) 은 더해질 수 있습니다. 정수값 (integral) 은 포인터가 가리키는 대상체의 크기를 곱해서 메모리상의 주소 (address offset) 로 변환**됩니다. **연산의 결과인 합은 원래의 대상체로부터 적당한 상대번지 (offset) 만큼 떨어져있는, 같은 배열내의 다른 대상체를 가리키는 포인터**입니다. 따라서 포인터 p 가 어떤 배열내의 대상체를 가리키고 있다면, p+1 은 그 다음 요소를 가리키는 것입니다. **배열의 한쪽 끝 바로 다음 위치 (first location beyond the high end) 를 제외하고, 배열의 한계를 넘어서서 가리킬 경우의 결과는 정의되어 있지 않습니다 (undefined).**

배열의 끝을 넘어서는 포인터에 대한 금지조항은 새로운 것입니다. 이는 배열의 소들에서만 루프를 도는 일반적인 관례를 정통으로 받아들인 것입니다.

▶ 역자 주

배열의 한계를 넘어서는 포인터에 대해서는 어떠한 사항도 정의해 주지 않지만, 배열의 바로 다음 위치에 대해서는 포인터가 배열을 벗어난다고 해도 인정을 해줍니다. 더 자세한 사항은 뺄셈연산자와 관계연산자를 참고하시기 바랍니다.

포인터로 배열을 다룰 때, 전에는 금지조항이 없어도 배열을 벗어나는 포인터 연산은 필요가 없어 행해지지 않았습니다. 하지만 이제는 ANSI 에서 그러한 관례를 받아들여 이를 명확히 금지하게 되었습니다.

- 연산자의 결과는 피연산자의 차입니다. **포인터에서 어떤 정수값 (integral) 도 뺄 수 있으며 덧셈연산자처럼 동일한 변환이 적용**됩니다.

같은 데이터형 포인터는 서로 뺄 수 있으며, 결과는 두 대상체 사이의 거리 (변위, displacement) 를 나타내는 부호 있는 정수값 (integral) 입니다; 인접해 있는 대상체를 가리키는 포인터의 차는 1 입니다. **결과의 구체적인 데이터형은 따로 정의되어 있지 않지만 (implementation-dependent), 표준 헤더파일 <stddef.h> 에 ptrdiff_t 로 정의되어 있습니다. 같은 배열내의 대상체를 가리키는 포인터의 차가 아니라면 그 결과값은 정의되어 있지 않습니다 (undefined); 하지만 배열의 마지막 요소를 p 라는 포인터가 가리키고 있다면, (p + 1) - 1 의 결과는 1 임이 보장**됩니다.

▷ A7.8 쉬프트 연산자 (Shift Operator)

쉬프트 연산자 << 와 >> 는 왼쪽에서 오른쪽으로 묶입니다. **피연산자는 정수형 (integral) 이어야 하고 정수진급이 수행**됩니다. **결과의 데이터형은 진급된 왼쪽 피연산자의 데이터형**입니다. 만약 **오른쪽 피연산자가 음수이거나 왼쪽 피연산자 비트수 이상의 크기일 경우에 결과는 정의되어 있지 않습니다 (undefined).**

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

E1 << E2 의 값은 E2 비트수만큼 좌측 쉬프트된 (비트패턴으로 해석된) E1 입니다; **오버플로우를 무시한다면, 이 값은 2^{E2} 를 곱한 것과 동일**합니다. E1 >> E2 의 값은 E2 비트수만큼 우측 쉬프트된 E1 입니다. **E1 이 부호 없는 형이거나 양수값을 갖고 있다면 결과는 2^{E2} 로 E1 을 나눈 것과 동일**합니다; **E1 이 음수인 경우에는 컴파일러에 따라 결과가 달라집니다** (implementation-defined).

▷ A7.9 관계연산자 (Relational Operator)

관계연산자는 왼쪽에서 오른쪽으로 묶입니다. 하지만 이러한 사실은 별로 유용하지 않습니다; a < b < c 는 (a < b) < c 로 묶이고, (a < b) 는 0 아니면 1 로 평가되어 버립니다.

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

연산자 < (미만), > (초과), <= (이하), >= (이상) 는 서술된 관계식이 거짓이면 0 을, 참이면 1 을 생성합니다. **결과의 데이터형은 int 형입니다. 일반적 산술변환이 산술형 피연산자에 수행**됩니다. **(형한정어를 무시하고) 동일한 데이터형을 가리키는 포인터는 비교될 수 있습니다; 결과는 포인터가 가리키는 대상체의 메모리상의 상대적인 위치에 달려있습니다. 포인터 비교는 오직 동일한 대상체의 일부분에 대해서만 정의되어 있습니다**: 만약 두 포인터가 동일한 대상체를 가리킨다면 결과는 같다고 (equal) 나옵니다; 만약 **포인터가 동일한 구조체의 멤버들을 가리킨다면, 구조체에서 후에 선언되는 대상체의 포인터가 더 크다고 (higher) 비교**됩니다; 만약 **포인터가 동일한 배열을 가리키고 있다면 포인터 비교의 결과는 곧 첨자 비교와 같습니다**. 만약 **포인터 p 가 배열의 마지막 요소를 가리키고 있다면 p+1 이 비록 배열의 밖이라 해도 p 보다 크게 평가**됩니다. **그 외의 경우는 포인터 비교에 대해서 정의되어 있지 않습니다** (undefined).

> 이러한 규칙들은 구조체나 공용체의 서로 다른 멤버들에 대한 포인터 비교를 허가함으로써, 초판에서의 제약을 다소 완화하였습니다. ANSI 는 또한 배열의 끝을 벗어나는 비교를 공인하였습니다.

▷ A7.10 상등연산자 (Equality Operator)

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

== (같음) 과 != (같지 않음) 연산자는 낮은 연산순위를 제외하면 관계연산자와 유사합니다. (따라서 a < b 와 c < d 가 같은 진리값을 갖을 때에만 a < b == c < d 는 1 이 됩니다)

상등연산자는 관계연산자와 동일한 규칙을 따르지만, 추가적인 가능성도 허가되어 있습니다: **포인터는 void 형 포인터 혹은 0 값을 갖는 상수수식과 비교될 수 있습니다**. (A6.6 참고)

▷ A7.11 비트 AND 연산자 (Bitwise AND Operator)

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

피연산자에 일반적 산술변환이 수행됩니다; 결과는 피연산자의 비트 AND 값입니다. **정수 피연산자 (integral) 에만 적용할 수 있습니다**.

▷ A7.12 비트 XOR 연산자 (Bitwise Exclusive OR Operator)

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

피연산자에 일반적 산술변환이 수행됩니다; 결과는 피연산자의 비트 XOR 값입니다. **정수 피연산자 (integral) 에만 적용할 수 있습니다**.

▷ A7.13 비트 OR 연산자 (Bitwise Inclusive OR Operator)

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

피연산자에 일반적 산술변환이 수행됩니다; 결과는 피연산자의 비트 OR 값입니다. **정수 피연산자 (integral) 에만 적용할 수 있습니다**.

▷ A7.14 논리 AND 연산자 (Logical AND Operator)

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

&& 연산자는 왼쪽에서 오른쪽으로 묶입니다. 두 피연산자가 모두 0 과 다르면 1 을, 그렇지 않으면 0 을 반환합니다. & 와는 달리, **&& 는 왼쪽 피연산자가 먼저 평가된다는 것이 보장**됩니다: **모든 부작용 (side effect) 을 포함해 왼쪽 피연산자가 먼저 평가**됩니다; 그 값이 0 이라면, 수식의 값은 0 이 됩니다. 그렇지 않으면 오른쪽 피연산자가 평가되고, 그 값이 0 이라면 수식의 값은 0 이, 0 이 아니라면 수식의 값은 1 이 됩니다.

피연산자는 같은 데이터형을 갖을 필요는 없지만, 산술형이나 포인터여야 합니다. **연산결과는 int 형**입니다.

▷ A7.15 논리 OR 연산자 (Logical OR Operator)

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

|| 연산자는 왼쪽에서 오른쪽으로 묶입니다. 피연산자 중 하나가 0 과 다르면 1 을, 그렇지 않으면 0 을 반환합니다. | 와는 달리, **|| 는 왼쪽 피연산자가 먼저 평가된다는 것이 보장**됩니다: **모든 부작용 (side effect) 을 포함해 왼쪽 피연산자가 먼저 평가**됩니다; 그 값이 0 과 다르면, 수식의 값은 1 이 됩니다. 그렇지 않으면 오른쪽 피연산자가 평가되고, 그 값이 0 과 다르면 수식의 값은 1 이, 0 이면 수식의 값은 0 이 됩니다.

피연산자는 같은 데이터형을 갖을 필요는 없지만, 산술형이나 포인터여야 합니다. 연산 결과는 int 형입니다.

▷ A7.16 조건연산자 (Conditional Operator)

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

모든 부작용 (side effect) 을 포함해 첫번째 피연산자가 평가됩니다; 그 값이 0 과 다르면 결과는 두번째 수식의 값이 됩니다. **두번째와 세번째 피연산자 중 하나만이 평가**됩니다. 만약 **두번째와 세번째 피연산자가 산술형이면, 두 피연산자를 공통된 데이터형으로 만들기 위해 일반적 산술변환이 적용**됩니다. 그리고 **그것이 곧 결과의 데이터형**이 됩니다. 만약 **두 피연산자가 void 형이거나, 같은 구조체나 공용체이거나, 같은 형의 포인터라면 결과는 두 피연산자의 공통된 데이터형**이 됩니다. 만약 **한 피연산자는 포인터, 다른 피연산자는 상수 0 이라면, 0 은 포인터로 변환되고 결과형 역시 포인터**입니다. **하나는 void 형 포인터고, 다른 하나는 일반 포인터라면 일반 포인터는 void 형 포인터로 변환되고 결과형 역시 void 형 포인터**입니다.

포인터의 데이터형 비교에서, 포인터가 가리키는 데이터형의 어떠한 형한정어 (type qualifier, A8.2) 도 무시되어 버립니다. 하지만 **결과의 데이터형은 조건연산자의 두 피연산자로부터 한정어를 상속** 받습니다.

> **▶ 역자 주**
> 두 포인터의 데이터형을 비교할 때 형한정어는 데이터형 비교에 아무런 영향을 주지 못합니다. 다만 데이터형 비교에서는 영향이 없으나, 결과의 데이터형이 결정될 때는 두 피연산자의 한정어를 물려받게 된다는 의미입니다.

▷ A7.17 대입수식 (Assignment Expression)

C 에는 여러 종류의 대입연산자 (assignment operator) 가 있습니다; 모든 대입연산자는 오른쪽에서 왼쪽으로 묶입니다.

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

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

모든 대입연산자는 왼쪽 피연산자로 좌변값을 필요로 하며, 그 좌변값은 반드시 수정이 가능해야 합니다: **배열이어서는 안되며, 불완전한 형 (incomplete type) 이나 함수여도 안됩니다.** 또한 **좌변값의 데이터형이 const 로 한정되어 있어도 안됩니다; 좌변값이 구조체나 공용체인 경우, 구조체나 공용체중 어떠한 멤버도, 재귀적으로 되어 있는 경우, 어떠한 부분멤버 (submember) 도 const 로 한정되어서는 안됩니다. 대입수식의**

데이터형은 왼쪽 피연산자의 데이터형이며, 값은 대입이 일어난 후에 왼쪽 피연산자에 대입된 값입니다.

= 와 같은 간단한 대입에서, 수식의 값은 좌변값에 의해 참조되는 대상체의 값으로 치환됩니다. 다음 사실 중 하나에 반드시 해당되어야 합니다: **두 피연산자가 모두 산술형이라면, 오른쪽 피연산자가 대입에 의해 왼쪽 피연산자의 데이터형으로 변환**됩니다; 혹은 **두 피연산자가 모두 같은 형의 구조체나 공용체**입니다; 혹은 **한 피연산자는 포인터이고 다른 피연산자는 void 형 포인터**입니다; 혹은 **왼쪽 피연산자는 포인터이고 오른쪽 피연산자는 0 값을 갖는 상수수식**입니다; 혹은 **오른쪽 피연산자에 있는 const 나 volatile 를 제외하면 같은 데이터형의 대상체나 함수를 가리키는 포인터여야 합니다**.

E1 op= E2 형태의 수식은 **E1 이 한번만 평가**된다는 것을 제외하면 E1 = E1 op E2 와 동일합니다.

▷ A7.18 쉼표연산자 (Comma Operator)

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

쉼표에 의해 분리된 한 쌍의 수식은 **왼쪽에서 오른쪽으로 평가**되고, **수식의 데이터형과 값은 오른쪽 피연산자와 같습니다. 왼쪽 피연산자를 평가하면서 발생하는 모든 부작용 (side effect) 은 오른쪽 피연산자의 평가를 시작하기 전에 모두 완료**됩니다. **함수 인자 리스트나 초기치 (initializer) 와 같이 쉼표가 특별한 의미로 주어지는 문맥에서, 필요한 문법적 단위 (syntactic unit) 는 대입수식이고, 따라서 쉼표연산자는 괄호 안에서만 나옵니다**; 예를 들어,

```
f(a, (t=3, t+2), c)
```

는 3 개의 인자를 가지고 있으며, 두번째 인자의 값은 5 가 됩니다.

▷ A7.19 상수수식 (Constant Expression)

문법적으로, **상수수식은 연산자의 부분집합에 제한된 수식**입니다.

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

상수수식은 **case 문의 뒤, 배열의 크기 (bound) 설정, 비트필드의 길이, 열거상수의 값, 초기치, 명확한 전처리기 수식**과 같은 여러 문맥에서 필요합니다.

상수수식은 sizeof 의 피연산자로 쓰이는 경우를 제외하고는 대입문, 간접지정 (indirection), 증감연산자, 함수호출, 쉼표연산자를 포함할 수 없습니다. 만약 **상수수식이 정수값 (integral) 일 필요가 있다면, 그것의 피연산자는 반드시 정수 (integer), 열거, 문자, 부동상수로 구성되어야만 하며, 정수형 (integral) 으로의 캐스트가 명시되어서 부동상수가 정수 (integer) 로 변환되어야만** 합니다. **상수수식에는 필수적으로 배열, 간접지정, 주소연산, 그리고 구조체 멤버 연산은 배제**됩니다. (다만, **sizeof 의 피연산자로 쓰인다면 모두 가능**합니다)

초기치 (initalizer) 로 쓰이는 상수수식에는 더 많은 것들이 허용됩니다. **피연산자는 어떠한 형의 상수여도 가능하고, 단항 & 연산자가 외부 (external) 혹은 정적 (static) 대상체에 사용되어도 좋으며, 첨자가 있는 (subscripted) 외부 혹은 정적 배열이 들어있어도 괜찮습니다. 또한 단항 & 연산자는 첨자가 없는 배열이나 함수가 나옴으로써**

암시적으로 적용될 수도 있습니다. **초기치는 반드시 이미 선언된 외부, 정적 대상체의 주소에 상수가 가감된 값이나 상수값으로 평가되어야만** 합니다.
#if 뒤에 나오는 정수 상수수식 (integral) 에는 더 많은 제약이 따릅니다; **sizeof 수식, 열거상수, 캐스트가 허가되어 있지 않습니다**. A12.5 참고.

▶ 역자 주
sizeof 수식은 컴파일 전에 이미 상수수식으로 평가되므로,
상수수식에는 사용할 수 없는 피연산자라 해도 sizeof 를 거치면
충분히 사용할 수 있는 것입니다.

▷ A8. 선언 (Declaration)

선언은 각 명칭에 주어지는 여러 가지 사항들을 알려주는 것일 뿐, 필수적으로 기억장소를 할당하지는 않습니다. 기억장소를 할당하는 선언은 정의 (definition) 라고 부릅니다. 선언은 아래와 같은 형태를 갖습니다.

```
declaration:
        declaration-specifiers    init-declarator-list opt ;
```

초기-선언자-리스트 (init-declarator-list) 에 있는 선언자 (declarator) 에는 선언되는 명칭 (identifier) 이 포함됩니다; **선언-지정자 (declaration-specifier) 는 형지정자와 기억부류 지정자 (type and storage class sprcifier) 로 구성**됩니다.

```
declaration-specifiers:
        storage-class-specifier    declaraion-specifiers opt
        type-specifier    declaraion-specifiers opt
        type-qualifier    declaraion-specifiers opt

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

init-declarator:
        declarator
        declarator = initializer
```

선언은 후에 (A8.5) 보다 자세히 설명할 것입니다; 선언에는 선언될 명칭들이 포함되어 있습니다. **선언은 최소한 한개의 선언자 (declarator) 를 포함하거나, 아니면 형지정자 (type specifier) 가 구조체 택이나 공용체 택, 열거형 멤버를 선언해야만 합니다**; **비어있는 (empty) 선언은 허용되지 않습니다**.

▷ A8.1 기억부류 지정자 (Storage Class Specifier)

기억부류 지정자는 다음과 같습니다:

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

기억부류 지정자의 의미는 A4 절에서 알아보았습니다.

auto 와 register 는 선언된 대상체를 자동 기억부류 (automatic storage class) 로 만들어 주며, 함수 안에서만 사용할 수 있습니다. **이러한 선언은 '정의'처럼 기억장소를 할당해 줍니다**. register 선언은 auto 선언과 동일하지만, 컴파일러에게 선언된 대상체가 빈번히 사용됨을 알려줍니다. **실제로 CPU 레지스터에 저장될 수 있는 대상체의 개수와 데이터형에는 제한이 있으며, 자세한 제한사항은 컴파일러마다 다릅니다 (implementation-dependent)**. 하지만 **일단 대상체가 register 로 선언된다면, 명시적이든 아니든 단항 & 연산자가 적용될 수 없습니다**.

> 실제로는 대상체가 일반적인 auto 로 취급된다고 해도, register 로 선언되었다면 그 대상체의 주소를 얻을 수 없다는 것은 새로운 규칙입니다.

static 지정자는 선언된 대상체를 정적 기억부류 (static storage class) 로 만들어 주며, 함수 안팎에서 모두 사용할 수 있습니다. **함수 안에서, 이 지정자는 '정의'처럼 기억장소를 할당해 줍니다**; 함수 밖에서의 영향은 A11.2 를 참고하시기 바랍니다.

함수 안에서 사용되는 extern 선언은, 선언된 대상체를 위한 기억장소가 다른 곳에서 정의되었음을 알려줍니다. 함수 밖에서의 영향은 A11.2 를 참고하시기 바랍니다.

사용자 정의 typedef 지정자는 기억장소를 예약하지 않으며, 문법적 편의를 위해 기억부류 지정자로 분류합니다.

선언에는 최대 한 개의 기억부류 지정자가 주어져야 합니다. 만약 주어지지 않는다면 다음과 같은 규칙이 적용됩니다: **함수 안에서 선언된 대상체는 auto 로 취급**됩니다; **함수 안에서 선언된 함수는 extern 으로 취급**됩니다; **함수 밖에서 선언된 대상체와 함수는 외부연결 (external linkage) 과 정적 기억부류를 갖습니다**. (A10 - A11 참고)

▷ A8.2 형지정자 (Type Qualifier)

형지정자는 다음과 같습니다.

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

long 과 short 중 하나만이 int 와 함께 사용될 수 있습니다. long, short 가 있으면 int 를 써주지 않아도 의미는 int 형입니다. long 은 double 과 함께 쓰일 수 있습니다. signed 와 unsigned 중 하나가 int, short int, long int, char 와 함께 쓰일 수 있습니다. signed 나 unsigned 가 혼자 나타나면 int 형으로 인식됩니다. signed 지정자는 char 형 대상체가 부호 있는 값을 갖도록 하는데 유용합니다; 정수형에도 signed 를 붙일 수는 있지만 중복된 선언입니다.

위의 경우가 아니라면, 최대 1 개의 형지정자만이 선언에 나올 수 있습니다. 선언에 형지정자가 생략되면, int 형으로 취급됩니다.

데이터형은 선언된 대상체에 특별한 특성을 나타내기 위해 한정되기도 (qualified) 합니다.

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

형한정어는 형지정자와 함께 나타날 수 있습니다. **const 대상체는 초기화는 되지만 값이 대입될 수는 없습니다**. **volatile 로 한정된 대상체에 대한 사항은 컴파일러마다 다르게 정의**합니다 **(implementation-dependent)**.

> const 와 volatile 의 특성은 ANSI 에서 새로이 정한 것입니다. const 의 목적은 대상체가 ROM 에 위치할 수도 있음을 알려, 최적화의 기회를 증가시키는 것입니다. volatile 의 목적은 컴파일러에게 최적화를 억제하도록 알리는 것입니다. 예를 들어, 메모리 매핑 입출력 (memory-mapping input/output) 을 하는 기종에서, 장치 레지스터 (device register) 를 가리키는 포인터를 통해 분명히 중복된 참조를 할 때, 컴파일러가 이를 없애는 것을 막기 위해서 그 포인터를 volatile 로 선언할 수 있습니다. **const 대상체의 값을 변경하려는 명백한 시도를 검사하는 것을 제외하면, 컴파일러는 이러한 한정어를 무시할 수 있습니다**.

▷ A8.3 구조체와 공용체 선언 (Structure and Union Declarations)

구조체는 다양한 데이터형의 여러 멤버를 포함하는 대상체입니다. 공용체는 다른 시간에 (동시에는 불가능하다는 뜻) 다양한 데이터형의 여러 멤버 중 하나를 담을 수 있는 대상체입니다. **구조체 지정자와 공용체 지정자는 같은 형태를 갖습니다**.

```
struct-or-union-specifiers:
        struct-or-union    identifier opt    { struct-declaration-list }
        struct-or-union    identifier

struct-or-union:
        struct
        union
```

구조체-선언-리스트 (struct-declaration-list) 는 구조체나 공용체 멤버들이 나열된 것입니다:

```
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
```

보통, **구조체-선언자 (struct-declarator) 는 구조체나 공용체의 멤버들을 위한 선언자**입니다. 구조체 멤버는 또한 비트수를 지정한 값을 포함하기도 합니다. 이러한 멤버는 비트필드 (bit-field), 혹은 드물게 그냥 필드 (field) 라고 부릅니다; 비트필드의 길이는 선언자로부터 콜론(:)으로 분리되어 지정됩니다.

```
struct-declarator:
        declarator
        declarator opt : constant-expression
```

아래와 같은 형태의 형지정자는 명칭을, 리스트에 의해 서술된 구조체나 공용체의 택 (tag) 으로 선언합니다.

```
struct-or-union identifier { struct-declaration-list }
```

위와 같이 선언이 되면, **같거나 더 안쪽의 통용범위 (scope) 에서 리스트 없이 택만을 사용해서 같은 데이터형을 참조할 수 있습니다**:

```
struct-or-union identifier
```

만약, **리스트 없이 택만 지정되고 또 이 택이 전에 선언된 적이 없다면, 불완전한 형 (incomplete type) 이 됩니다**. **불완전한 구조체나 공용체형의 대상체는 선언이나 (정의는 안됨), 포인터 지정, typedef 생성과 같이 구체적인 크기를 필요로 하지 않는 곳에서만 쓰일 수 있습니다**. **불완전한 형은 선언 리스트를 갖는 동일한 택의 선언이 나타나면 완전한 형이 될 수 있습니다**. 비록 **리스트를 갖고 있는 지정자라고 해도, 구조체나 공용체는 리스트 안에서는 불완전한 형이며, 지정자를 종료시켜주는 } 가 나와야만 완전한 형**이 됩니다.

구조체는 불완전한 형의 멤버를 포함할 수 없습니다. 따라서 **자기 자신을 포함하는, 구조체나 공용체를 선언하는 것은 불가능**합니다. 하지만 **불완전한 형을 가리키는 포인터는 선언할 수 있고, 또 구조체나 공용체는 자기 자신을 참조하는 포인터를 포함할 수 있기 때문에, 결국 자기참조 구조체 (self-referential structure) 를 정의하는 것이 가능합니다**.

선언 리스트 (declaration list) 와 선언자 (declarator) 없이 구조체와 공용체를 선언하는 아래와 같은 선언에는 매우 특별한 규칙이 적용됩니다.

```
struct-or-union identifier;
```

명칭이 이미 바깥쪽 통용범위 (outer scope) 에서 선언된 구조체나 공용체의 택이라고 해도, 위와 같은 선언은 **현재의 통용범위 내에서 그 명칭을 불완전한 형을 갖는 구조체나 공용체의 택으로 만들어 줍니다**.

이러한 난해한 규칙은 새로운 것입니다. 이 규칙의 목적은 바깥 통용범위에서 이미 선언된 택을 갖는, 안쪽 범위에서 선언된 상호-재귀적인 (mutually-recursive) 구조체를 다루기 위한 것입니다.

▶ 역자 주

상호 재귀적인 구조체란 다음과 같은 것을 말합니다.

```
struct x { int a; struct y *yp };
struct y { int b; struct x *xp };
```

이제 위에서 이야기한 이른바 '난해한 규칙'이 필요한 경우를 예로 들어보겠습니다.

```
struct alpha { char c; };
struct beta { float f; };

int main()
{
    struct alpha { int x; struct beta *b; } aa;
    struct beta { int y; sttuct alpha *a; } bb;
    aa.b = &bb;
    bb.a = &aa;
    return 0;
}
```

위의 예에서 대입문 aa.b = &bb 는 허가되지 않습니다. 그 이유는 구조체 aa 안에 있는 'struct beta' 가 main() 밖에 있는 beta 를 참조하기 때문입니다. (원래 의도는 main() 안의 alpha 밑에 있는 beta 를 참조하는 것임) 이와 같은 경우 main() 안에 있는 alpha 와 beta 의 선언 순서를 바꿔도 문제는 해결되지 않습니다. 이럴 때, main() 함수의 본체 안에서 alpha, beta 를 선언하기 전에 다음과 같은 선언을 넣어주면, 위에서 이야기한 '난해한 규칙' 에 의해 블록 밖과는 다른 새로운 구조체 alpha, beta 가 선언됩니다.

```
struct alpha;
struct beta;
```

물론 위의 예는 블록 밖의 alpha, beta 가 명확히 눈에 보이기 때문에 어느 한쪽을 고치면 상관없지만, 블록 밖의 구조체 선언이 다른 사람이 작성한 헤더파일에 파묻혀 있는 경우라면 문제가 될 수도 있습니다.

택 없이 리스트만 있는 구조체 · 공용체 지정자는 하나의 제대로 된 데이터형을 만들지 못합니다; 따라서 이러한 구조체나 공용체는 **오직 선언되는 부분에서만 직접적으로 참조될 수 있습니다**.

멤버명과 택명은 서로간에 혹은 일반 변수와 충돌하지 않습니다. 동일한 구조체나

공용체에 같은 멤버명이 두 번 나올 수는 없지만, **다른 구조체에는 동일한 멤버명이 사용될 수 있습니다.**

> 하지만, 이러한 택과 멤버명의 관계는 ANSI 표준안 이전부터 여러 컴파일러에게는 보편적인 것이었습니다.

구조체나 공용체의 비트필드가 아닌 멤버는 어떠한 데이터형을 가져도 좋습니다. (선언자가 필요 없고, 따라서 이름이 붙지 않아도 되는) **비트필드 멤버는 int, unsigned int 혹은 signed int 형을 갖으며, 주어진 비트만큼의 길이를 갖는 정수 대상체 (integral) 로 해석**됩니다; **int 형 비트필드가 부호 있는 데이터형으로 다루어지는지는 컴파일러마다 다릅니다 (implementation-dependent). 구조체의 인접해 있는 비트필드 멤버들은 컴파일러가 정한 방향으로, 컴파일러마다 다른 기억단위 (storage unit) 로 묶입니다. 비트필드가 부분적으로 채워진 기억단위에 맞지 않을 경우, 두 기억단위에 걸쳐 분리되어 저장될 수도 있고, 부분적으로 채워진 기억단위를 무의미한 값으로 채워서 (padding) 버릴 수도 있습니다.**

> **▶ 역자 주**
> padding 이란, 쓸모없는 기억장소에 무의미한 값을 채워서 필요없는 부분을 버리는 작업을 의미합니다.

길이 0 을 갖는 이름없는 비트필드는 이렇게 필요 없는 부분을 채우도록 해서, 다음 번 비트필드가 새로운 기억단위부터 시작하도록 해줍니다.

> ANSI 는 비트필드와 관련된 내용을 이전보다 더욱 컴파일러에 맡겨두었습니다 (implementation-dependent). 이는 곧 **비트필드 저장과 관련된 언어의 규칙은 무조건 컴파일러에 달려있다**는 뜻입니다. 비트필드가 있는 구조체는 (필드를 다루기 위한 내부 명령어가 더 복잡해지고, 필드에 접근하는 시간이 길어진다고 해도) **비트필드를 위한 기억장소를 줄이는 이식성 있는 방법 (portable way) 으로 다루기도 하고, 비트 수준에서 알아보기 쉬운 이식성 없는 방법 (non-portable way) 으로 다루기도** 합니다. 두번째 경우라고 해도, 각 컴파일러의 규칙을 따로 알아야 합니다.

구조체의 멤버는 **선언된 순서대로** 메모리상의 주소가 증가되어 저장됩니다. **비트필드가 아닌 구조체 멤버는 그 멤버의 데이터형에 따른 주소를 경계로 정렬**됩니다; 따라서 **구조체 내에는 명칭이 붙지않은 빈공간 (hole) 이 생길 수 있습니다.**

> **▶ 역자 주**
> A6.6 절에서 설명했듯이, 많은 수의 기종 (machine) 들은 기억장소가 적당히 정렬이 되어야 기억장소에 효율적으로 접근할 수 있으며, 심지어는 정렬이 되지 않은 기억장소에는 접근을 못하는 경우도 있습니다. 바이트 기반의 기종에서, 2 바이트의 short int 는 짝수번지에, 4 바이트의 long int 는 4의 배수번지에 정렬되는 경우가 있습니다. 그럴 때, 다음과 같은 구조체가 있다고 가정하면,

```
struct {
        char c;
        int i;
};
```

컴파일러는 int 멤버가 적당히 정렬되도록, char 와 int 멤버
사이에 이름도 없고, 사용되지 않는 공간 (hole) 을 남겨둡니다.

구조체를 가리키던 포인터가 첫번째 멤버를 가리키는 포인터형으로 캐스트 된다면, 결과는 첫번째 멤버를 참조합니다.

공용체는 멤버가 모두 상대번지 (offset) 0 에서 시작하고, 멤버 중 어떠한 것도 저장할 수 있는 충분한 크기를 갖는 구조체라고 생각하면 됩니다. 공용체에는 멤버중 하나만을 저장할 수 있습니다. 만약 **공용체를 가리키던 포인터가 어떠한 멤버를 가리키는 포인터형으로 캐스트 된다면, 결과는 그 멤버를 참조**합니다.

구조체 선언의 간단한 예는 다음과 같습니다.

```
struct tnode {
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

이 구조체는 문자 20 개를 저장할 수 있는 배열, 정수형 (integer), 자기 자신을 참조하는 포인터 2 개를 담고 있습니다. 우선 위와 같은 선언이 주어지면, 다음과 같은 선언은, s 를 주어진 종류의 구조체로, sp 를 주어진 종류의 구조체 포인터로 선언해 줍니다.

```
struct tnode s, *sp;
```

위와 같은 선언이 주어지면, 수식

```
sp -> count
```

는 sp 가 가리키는 구조체의 count 멤버를 참조합니다;

```
sp.left
```

위의 수식은 구조체 s 의 좌측 부분트리 (subtree) 를 참조합니다;

```
s.right -> tword[0]
```

그리고 위 수식은 s 의 우측 부분트리의 tword 의 첫번째 문자를 가리킵니다.

일반적으로, **값이 대입되지 않은 공용체의 멤버는 조사할 수가 없습니다**. 하지만 한가지 사실이 확실히 보장되어 있어서, 공용체의 사용을 편하게 해줍니다: **만약 공용체가 같은 초기멤버 (initial sequence) 를 공유하는 여러 개의 구조체를 포함하고 있고, 현재 사용중인 공용체가 이 구조체중 하나를 포함한다면, 포함된 구조체중 어떠한 (공통된) 초기멤버라도 참조하는 것이 허가**되어 있습니다. 예를 들어, 아래와 같은 경우는 적법한 것입니다:

```
union {
        struct {
                int type;
        } n;
        struct {
                int type;
                int intnode;
        } ni;
        struct {
                int type;
                float floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

▷ A8.4 열거 (Enumeration)

열거는 열거자 (enumerator) 라고 불리는, 이름이 붙어있는 상수의 집합 내에서 값이 변하는 독특한 데이터형 입니다. 열거 지정자 (enumeration specifier) 의 형태는 구조체 · 공용체 지정자의 형태와 비슷합니다.

```
enum-specifiers:
        enum    identifier opt    { enumerator-list }
        enum    identifier

enumerator-list:
        enumerator
        enumerator-list , enumerator

enumerator:
        identifier
        identifier = constant-expression
```

열거자 리스트 (enumerator list) 의 명칭은 int 형의 상수로 선언되고, 상수가 요구되는 곳에 사용할 수 있습니다. 만약 열거자에 = 가 없으면, 대응하는 상수의 값은, 0 에서 시작하고 왼쪽에서 오른쪽으로 가면서 1 씩 증가됩니다. = 가 주어진 열거자는 관련된 명칭에 명시된 값을 대입해 줍니다; = 가 없이 이어지는 명칭은 이전에 대입된 값부터 계속해서 증가합니다.

같은 통용범위 (scope) 내에 있는 열거자 명칭은, 서로간에 혹은 보통의 변수들과 구분되어야 합니다. 하지만 **값은 구분될 필요가 없습니다**.

열거-지정자 (enum-specifier) 에 주어진 명칭 (identifier) 의 역할은 구조체-지정자 (struct-specifier) 의 구조체 택과 유사합니다; 즉 특별한 열거를 명명하게 됩니다. 택과 리스트가 있든 없든 **열거-지정자 (enum-spicifier) 의 규칙은 구조체 · 공용체 지정자와 같습니다. 단 불완전한 열거형 (incomplete enumeration type) 은 존재하지 않습니다**; 따라서 열거자 리스트가 없는 열거-지정자의 택은, 통용범위 안의 리스트가 있는 지정자를 참조해야만 합니다.

열거형은 이 책의 초판 이후 새로운 것이지만, 이미 여러해동안 C 언어의 한 부분이었습니다.

▷ A8.5 선언자 (Declarator)

```
declarator:
        pointer opt     direct-declarator

direct-declarator:
        identifier
        ( declarator )
        direct-declarator [ constant-expression opt ]
        direct-declarator ( parameter-type-list )
        direct-declarator ( identifier-list opt )

pointer:
        * type-qualifier-list opt
        * type-qualifier-list opt     pointer

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

선언자의 구조는 간접지정 (indirection), 함수 (function), 배열수식 (array expression) 의 구조와 비슷합니다; **결합방식 (grouping) 은 동일**합니다.

▷ A8.6 선언자의 의미 (Meaning of Declarator)

선언자의 리스트는 형지정자와 기억부류 지정자 뒤에 나타납니다. 각 선언자는 직접-선언자 (direct-declarator) 의 처음에 제시된 한 개의 주요한 명칭 (identifier) 을 선언합니다. **기억부류 지정자는 이 명칭에 직접 적용되지만, 데이터형은 선언자의 형태에 달려 있습니다. 선언자는, 선언자의 명칭이 그 선언자와 동일한 형태의 수식에 나타났을 때, 주어진 데이터형의 대상체를 생성한다는 의미로** 이해됩니다.

▶ 역자 주

위 이야기는, 문법적인 표현에서 보았을 때 C언어의 선언은 수식에서 사용될 때와 동일한 형태를 갖는다는 의미입니다. 함수와 배열을 예로 들면 아래와 같습니다.

```
int f(double);    /* 선언자 f 는 함수이며, 이는 함수를
                     사용할 때와 동일한 문법적 구성 (괄호와
                     하나의 값) 을 갖습니다 */
f(3.14);          /* 선언과 동일한 형태를 갖는 함수 사용.
                     int 형이 생성됩니다 */

double a[10];     /* 선언자 a 는 배열입니다. 대괄호안에
                     하나의 값이 들어가 있는 형태입니다 */
A[3];             /* 선언과 동일한 형태를 갖는 배열 사용.
                     double 형이 생성됩니다 */
```

선언 지정자 (A8.2) 의 데이터형 부분과 특별한 선언자에 대해서만 생각해보면, 선언은 "T D" 의 형태를 갖으며, T 는 데이터형, D 는 선언자를 나타냅니다. 다양한 형태의 선언자에서 명칭에 속성을 주는 데이터형은 이러한 표기법을 이용해 귀납적으로 (inductively) 서술됩니다.

선언 T D 에서, D 가 아무것도 붙지않은 명칭이라면, 그 명칭의 데이터형은 T 입니다.

선언 T D 에서, D 가

(D1)

과 같은 형태를 갖고 있다면, D1 에 있는 명칭의 데이터형은 D 와 동일합니다. **괄호는 데이터형을 바꾸지 못하지만, 복잡한 선언자를 묶는 방식은 바꿀 수 있습니다.**

▷ A8.6.1 포인터 선언자 (Pointer Declarator)

선언 T D 에서,

* type-qualifier-list opt D1

D 가 위와 같은 형태를 갖고, 선언 T D1 에 있는 명칭의 데이터형이 "type-modifier T (T 형변경자)" 라면, **D 에 있는 명칭의 데이터형은 "type-modifier type-qualifier-list pointer to T (형한정어 T 형 포인터 형변경자)"** 입니다. *** 뒤에 주어진 형한정어는 포인터가 가리키는 대상체가 아닌, 포인터 자체에 적용**됩니다.

> **▶ 역자 주**
> 데이터형을 적는 부분은 원문을 그대로 옮기고, 괄호 안에 번역을 적었습니다. 영어의 특성상 맨 앞에 걸리는 형변경자가 한국어로 번역할 때는 맨 뒤로 가기 때문에 그렇게 적어주는 등의 노력은 했지만, 가능하면 원문 그대로를 참고하시기 바랍니다.
> 이 부분의 내용은 복잡해 보이지만, 선언을 분석해 데이터형을 결정하는 부분입니다. 아래에 나오는 예를 보면 이해가 좀 더 쉬울 겁니다. 이해가 쉽도록 원문에는 없는 설명을 나름대로 추가하였습니다.

예를 들어, 다음과 같은 선언을 생각해보면,

```
int *ap[];
```

여기서 ap[] 가 D1 에 해당됩니다; 선언 "int ap[]" (아래 '배열 선언자' 참고) 는 ap 를 "array of int (int 형 배열)" 로 만들어 주고, 따라서 * 뒤에 나오는 형한정어 리스트 (type-qualifier list) 는 없고, 형변경자 (type-modifier) 는 "array of (~의 배열, ap[] 가 D1 이므로)" 가 되는 것입니다. 결국 명칭 ap 에 실제로 선언되는 데이터형은 "array of pointers to int (int 형 포인터의 배열)" 이 되는 것입니다.

다른 예를 살펴보면,

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

위에서는 정수 i 와 정수형 포인터 pi 를 선언합니다. 상수 포인터 cpi 의 값은 변할 수 없습니다; 즉 **cpi 는 항상 같은 위치를 가리켜야만 하며, cpi 가 가리키는 곳에 저장되는 값은 변할 수 있습니다.** 정수형 ci 는 상수이고 (위에서처럼 초기화는 되지만) 값이 변할 수는 없습니다. pci 의 데이터형은 "pointer to const int (const int 형 포인터)" 이며, **pci**

자체는 다른 장소를 가리키도록 변할 수 있지만, pci 를 통해서는 pci 가 가리키는 곳의 값을 바꿀 수 없습니다.

▷ A8.6.2 배열 선언자 (Array Declarator)

선언 T D 에서,

D1 [constant-expression opt]

D 가 위와 같은 형태를 갖고, 선언 T D1 에 있는 명칭의 데이터형이 "type-modifier T (T 형변경자)" 라면, **D 에 있는 명칭의 데이터형은 "type-modifier array of T (T 형 배열 형변경자)"** 입니다. 만약 **상수수식이 있다면, 그 수식은 정수형이어야 하며, 0 보다 커야만 합니다. 배열의 크기 (bound) 를 나타내는 상수수식이 생략된다면, 배열은 불완전한 형 (incomplete type) 이 됩니다.**

배열은 산술형, 포인터, 구조체, 공용체, 혹은 (다차원 배열의 경우) 다른 배열을 요소로 갖을 수 있습니다. 배열의 요소는 완전한 형 (complete type) 이어야만 합니다; 즉 불완전한 형을 갖는 배열이나 구조체로는 배열을 구성할 수 없습니다. 이는 곧 **다차원 배열에서 오직 첫번째 첨자만을 생략할 수 있음을 의미**합니다. **불완전한 형의 배열은 완전한 다른 선언이 나오든가 (A10.2) 아니면 초기화 됨으로써 (A8.7) 완전한 형 (complete type) 이 됩니다.** 배열의 예를 들면,

```
float fa[17], *afp[17];
```

위와 같은 선언은 float 형 숫자들의 배열과, float 형 포인터의 배열 (array of pointer to float) 을 의미합니다. 또한,

```
static int x3d[3][5][7];
```

과 같은 선언은 3x5x7 행렬 형태의 정적 3 차원 배열을 의미합니다. 자세하게 설명하면, x3d 는 3 개의 요소를 가지고 있습니다; 각각의 요소는 각각 5 개의 배열을 갖고 있는 배열입니다; 또 5 개의 배열은 각각 7 개의 정수를 갖는 배열입니다. x3d, x3d[i], x3d[i][j], x3d[i][j][k] 중 어떠한 표현도 수식에 나올 수 있습니다. **처음 3 개의 데이터형은 "배열"이고, 마지막 하나만 int 형**입니다. 더 상세히 설명하면, x3d[i][j] 는 7 개의 정수를 갖는 배열이고, x3d[i] 는 7 개의 정수를 갖는 5 개의 배열의 배열입니다.

배열의 첨자연산 (subscripting operation) E1[E2] 는 *(E1+E2) 와 동일하다고 정의되어 있습니다. 따라서 모양이 이상하지만, **첨자연산은 교환법칙이 성립하는 연산**입니다.

> **▶ 역자 주**
> 따라서 a 가 배열이고, i 가 정수값이라고 할 때, a[i] 뿐만 아니라 i[a] 와 같은 형태의 배열 사용도 가능합니다. a[i] 와 i[a] 는 각각 *(a+i) 와 *(i+a) 로 결국 동일한 결과를 가져옵니다.

+ 연산자와 배열에 적용되는 변환규칙 (A6.6, A7.1, A7.7) 때문에, E1 이 배열이고 E2 가 정수형이라면, E1[E2] 는 E1 의 E2 번째 요소를 참조합니다.

예를 들면, x3d[i][j][k] 는 *(x3d[i][j] + k) 와 동일합니다. 첫번째 부수식인 x3d[i][j] 는 A7.1 에서 설명한 규칙에 의해 "정수형 배열을 가리키는 포인터"로 변환됩니다; A7.7 절에서 설명했듯이, 덧셈(+)은 배열을 구성하는 정수형의 크기에 곱한 값을 더하게 됩니다. **배열은 행 (row) 단위로 저장되며 (따라서 마지막 첨자가 가장 빨리**

변합니다), 선언에서의 첫번째 첨자는 그 배열에 의해 소비되는 기억장소의 양을 결정하는데 도움을 줄 뿐, 첨자계산에 참여하지는 않습니다.

▶ 역자 주

배열의 첫번째 첨자는 수식의 값을 결정하는데 필요하지 않습니다. 이는 배열연산의 정의가 포인터 연산이기 때문입니다. 예를 들면, 1 차원 배열에서, 어떠한 요소에 저장된 값을 알기 위해서 배열의 크기가 얼마인지 (→ 첫번째 첨자의 의미) 알 필요가 없습니다. 이는 다차원 배열에서도 마찬가지입니다. 다차원 배열에서도, 어떠한 요소의 값을 아는 데는 첫번째 첨자를 제외한 나머지 첨자만 있으면 됩니다.

▷ A8.6.3 함수 선언자 (Function Declarator)

현대적 방식 (new-style) 의 함수선언 T D 에서,

D1 (parameter-type-list)

D 가 위와 같은 형태를 갖고, 선언 T D1 에 있는 명칭의 데이터형이 "type-modifier T (T 형변경자)" 라면, **D 에 있는 명칭의 데이터형은 "type-modifier function with arguments parameter-type-list returning T (매개변수-데이터형-리스트를 인자로 갖고 T 형을 반환하는 함수 형변경자)"** 입니다.

매개변수 (parameter) 의 문법은 아래와 같습니다.

```
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
```

현대적 방식의 선언에서, **매개변수 리스트 (parameter list) 는 매개변수의 데이터형을 알려줍니다. 현대적 방식의 함수 선언자가 매개변수를 가지고 있지 않다면, 매개변수 데이터형 리스트 (pararmeter type list) 에는 void 라는 키워드만** 나옵니다. 만약 매개변수 데이터형 리스트가 생략부호 ", ..." 로 끝난다면, 그 함수는 주어진 매개변수의 개수보다 더 많은 인자를 받을 수 있습니다. (A7.3.2 참고)

매개변수의 데이터형이 함수나 배열이라면, A10.1 절에서 설명할 매개변수 변환 (parameter conversion) 규칙에 의해 포인터로 변환됩니다. **매개변수의 선언 지정자 (declaration specifier) 에 허락되는 기억부류 지정자는 register 뿐이며, 이러한 지정자는 선언자가 함수정의 (function definition) 부분에 나오지 않는 이상은 무시**됩니다.

▶ 역자 주

함수의 정의부분에서 매개변수에 register 지정자를 주지않는 이상, 아무리 함수 선언에 register 지정자를 준다고 해도 그 지정자는 무시된다는 의미입니다.

유사하게, **매개변수 선언에 있는 선언자가 명칭을 포함하고 있고, 함수 선언자가 함수정의 부분에 있는 것이 아니라면, 그 명칭은 즉시 통용범위 (scope) 를 빠져나가게 됩니다.**

▶ 역자 주

즉, 함수정의가 아닌 함수선언에 나오는 명칭의 통용범위는 매개변수를 둘러싸고 있는 괄호에서 끝난다는 의미입니다. 아래와 같이, 함수 선언에서 a 라는 명칭을 사용했다고 해도, 이 명칭이 함수정의에서 사용된 것이 아니므로 매개변수 리스트를 닫는 괄호에서 그 통용범위가 끝나는 것입니다.

```
int func(int a);
double a;
```

함수 func 의 선언에서 사용된 a 라는 명칭의 통용범위는 닫는 괄호) 에서 끝나기 때문에, 아래에 있는 double 형 a 에게 영향을 주지 않습니다.

```
int func(int a, int a); /* 불가능 */
```

단 위에서 보는 것처럼, 선언에서의 중복된 명칭 사용은 분명 a 의 통용범위가 끝나지 않은 상태에서 나온 것이므로 불가능합니다.

반면에, 함수정의에서 사용된 매개변수의 명칭은 함수 본체 (body) 의 가장 윗부분에서 선언된 것으로 보기 때문에 함수 전체에 걸쳐 통용범위가 적용됩니다.

함수선언과 정의에 사용된 명칭에 대한 보다 자세한 사항은 나중에 또 다루게 됩니다.

명칭이 나오지 않는 추상적 (abstract) 선언자는 A8.8 절에서 설명합니다.

고전적인 방식 (old-style) 의 함수선언 T D 에서,

```
D1 (identifier-list opt)
```

D 가 위와 같은 형태를 갖고, 선언 T D1 에 있는 명칭의 데이터형이 "type-modifier T (T 형변경자)" 라면, **D 에 있는 명칭의 데이터형은 "type-modifier function of unspecified arguments returning T (인자의 정보가 주어지지 않고, T 형을 반환하는 함수 형변경자)"** 입니다. 매개변수가 있다면 다음과 같은 형식을 갖습니다.

```
identifier-list:
        identifier-list
        identifier-list , identifier
```

고전적 방식의 선언자에서, 그 선언자가 함수정의 (A10.1) 부분에 있는 것이 아니라면 명칭 리스트 (identifier list) 는 없어야만 합니다. 매개변수의 데이터형에 대한 어떠한 정보도 선언에 의해 주어지지 않습니다.

예를 들어,

```
int f(), *fpi(), (*pfi)();
```

위와 같은 선언은 정수를 반환하는 함수 f, 정수형 포인터를 반환하는 함수 fpi, 정수를 반환하는 함수를 가리키는 포인터 pfi 를 의미합니다. 위의 선언에는 매개변수의 데이터형에 대한 사항이 전혀 없습니다; 즉 위의 선언은 고전적 방식입니다.
다음과 같은 현대적 방식의 선언에서는,

```
int strcpy(char *dest, const char *source), rand(void);
```

strcpy 는 문자형 포인터와 문자 상수형 포인터 (pointer to constant character) 를 인자로 갖고, int 형을 반환하는 함수입니다. **함수선언에서 매개변수의 이름은 실제로는 주석에 불과**합니다. 두번째 함수 rand 는 인자가 없으며, int 형을 반환합니다.

▶ 역자 주

선언에서 매개변수의 명칭이 주석에 불과하다는 말은, 컴파일러가 함수정의 부분의 명칭과 선언의 명칭이 동일한가 검사하지 않는다는 뜻입니다. (데이터형만을 검사합니다) 즉 위에서 이야기한 것처럼 중복되는 명칭만 사용하지 않으면 굳이 함수정의 부분의 명칭을 그대로 써줄 필요도 없다는 의미입니다. 따라서 다음과 같은 것이 가능합니다.

```
int func(int a, float b)        /* 함수정의 */
{
    ... 함수 본체 ...
}

/* 아래의 선언은 모두 적법 */
int func(int a, float b);
int func(int x, float y);
int func(int b, float a);
int func(int y, float m);
int func(int, float);
...
```

매개변수의 원형 (prototype) 을 갖는 함수 선언자는 ANSI 표준에 의한 변화 중 가장 중요한 것입니다. 물론 두 방식을 모두 수용해야만 하는 필요성과 소개하는데 따르는 어려움이 있기는 하지만, 현대적 방식의 선언은 초판에서 설명했던 고전적 방식과 비교해, 함수호출에 전달하는 인자의 강제변환과 에러검사 같은 이점을 제공합니다. 두 방식의 호환성을 위해, 매개변수가 없는 현대적 방식의 함수선언에는 void 라는 명백한 표시가 필요하다는 보기 싫은 문법이 필요해 집니다.

가변인자 함수 (variadic function) 를 위한 생략부호 ", ..." 는 표준헤더 <stdarg.h> 에 있는 매크로와 함께 새로운 것이며, 그 동안 공식적으로는 금지되어 비공식적으로 사용되던 가변인자 기술을 정식으로 받아들이는 것입니다.

이러한 표기법들은 C++ 로부터 받은 것입니다.

▷ A8.7 초기화 (Initialization)

대상체가 선언될 때, 대상체의 초기-선언자 (init-declarator) 는 선언되는 명칭의 초기값을 지정해 줄 수 있습니다. **= 뒤에 오는 초기치 (initializer) 는 하나의 수식이나, 중괄호에 쌓인 초기값의 리스트**입니다. 초기값의 리스트는 미관상의 목적으로 쉼표로 끝날 수 있습니다.

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

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

▶ 역자 주

쉼표로 끝나는 초기값 리스트에서 쉼표의 역할은 어디까지나 외관상의 표현일 뿐입니다. 즉 마지막 쉼표는 초기값의 리스트에 어떠한 영향도 주지 못합니다.

정적 대상체나 배열의 초기치로 주어지는 모든 수식은 7.19 절에서 알아본 상수수식 (constant expression) 이어야 합니다. **초기치가 중괄호에 쌓인 형태라면, auto 나 register 대상체나 배열의 초기치도 상수여야 합니다**. 하지만 **자동 기억부류 대상체의 초기치가 하나의 수식이라면, 초기치는 상수수식일 필요가 없으며, 다만 그 대상체에 대입되기 위한 적당한 데이터형이면 됩니다**.

> 초판에서는 자동 기억부류의 구조체, 공용체, 배열의 초기화를 지원하지 않았습니다. ANSI 는 이를 허락하였지만, 초기치가 중괄호에 쌓인 형태라면 상수수식이어야 한다고 제한하고 있습니다.

초기치가 주어지지 않은 정적 대상체는 상수 0 이 대입된 것처럼 초기화됩니다. 자동 대상체에 초기치가 주어지지 않으면, 어떤 값으로 초기화되는지 알 수 없습니다 (undefined).

포인터나 산술형 대상체의 초기치는 (중괄호가 있을 수 있는) 단일 수식입니다. 주어진 수식은 대상체에 대입됩니다.

구조체의 초기치는 동일한 데이터형의 수식이거나, 중괄호 안에 멤버 순서대로 나열된 초기치의 리스트입니다. **이름이 붙지않은 비트필드는 무시되며, 초기화 되지 않습니다**. 만약 **구조체의 멤버보다 적은 초기치가 주어지면, 나머지 부분은 0 으로 초기화**됩니다. **구조체 멤버보다 많은 초기치가 주어질 수는 없습니다**.

배열의 초기치는 각 요소에 해당하는 중괄호 안의 초기치 리스트입니다. 만약 배열의 크기가 주어지지 않았다면, 초기치의 개수가 배열의 크기를 결정해 주며, 배열은 완전한 형 (complete type) 이 됩니다. **만약 배열의 크기가 명시되어 고정되었다면, 초기치의 개수는 배열 요소의 개수를 넘어서는 안됩니다; 만약 초기치가 더 적은 경우에는, 나머지 부분은 0 으로 초기화**됩니다.

특별한 경우, **문자배열은 문자열로 초기화될 수 있습니다**; 문자열의 연속된 각 문자는 각각의 배열 요소를 초기화합니다. 유사하게 wide character string (확장 문자열, A2.6) 은 wchar_t 형의 배열을 초기화할 수 있습니다. 만약 **배열의 크기가 주어지지 않은 경우에는, 문자열의 끝을 알려주는 널문자를 포함한 문자열의 문자개수가 배열의 크기를 결정**합니다; 만약 **배열의 크기가 고정되어 있다면, 널문자를 제외한 문자열의 문자개수가 배열의 크기보다 커서는 안됩니다**.

공용체의 초기치는 공용체의 첫번째 멤버를 초기화해주는, 동일한 데이터형의 단일 수식이나 중괄호에 쌓인 초기값입니다.

초판에서는 공용체의 초기화를 허락하지 않았습니다. "첫번째 멤버 (만 초기화 할 수 있다는)" 규칙이 별로 세련된 규칙은 아니지만, 새로운 문법을 만들지 않고는 이 규칙을 일반화하기 어렵습니다. 덜 떨어진(^^) 방법으로라도 공용체를 명백히 초기화할 수 있도록 허락하는 것외에도, ANSI 는 명백히 초기화되지 않은 정적 (static) 공용체의 의미를 확실히 해주고 있습니다.

▶ 역자 주

ANSI-C 에서는 초기치가 없는 정적 기억부류의 대상체는 0 으로 초기화된다고 명시하고 있으므로, 초기치를 주지 않은 정적 공용체의 첫번째 멤버는 확실히 상수 0 이 대입된 것처럼 초기화됩니다.

'**집합체 (aggregate)' 는 구조체나 배열을 의미**하는 것입니다. 만약 **집합체가 다른 집합체를 멤버로 포함한다면, 초기화 규칙은 재귀적으로 적용**됩니다. 초기화에서의 괄호는 다음과 같이 생략될 수 있습니다: 만약 어떠한 집합체의 멤버로서 중첩된, 부분집합체 (subaggregate) 의 초기치가 왼쪽 중괄호 { 로 시작한다면, 괄호 뒤에 따라오는 초기치들은 그 부분집합체의 멤버를 초기화합니다; **이 경우에는, 멤버보다 많은 초기치가 주어지는 것은 잘못된 것입니다**. 하지만 만약 부분집합체의 초기치에 중괄호가 없다면, 초기치 리스트에서 필요한 만큼을 부분집합체의 멤버로 취합니다; **초기치로 정해지고 남아있는 초기치들은 다음 번 부분집합체를 초기화하는데 사용**됩니다.
예를 들면,

```
int x[ ] = { 1, 3, 5 };
```

위와 같은 선언은, 배열의 크기가 지정되지 않고 3 개의 초기치가 주어져 있으므로, x 를 3 개의 요소를 갖는 1 차원 배열로 만들어 줍니다.

```
float y[4][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

위의 예는 완벽하게 괄호가 쓰인 초기화 입니다: 1, 3, 5 는 배열 y[0] 의 첫번째 줄을 초기화합니다. 즉, y[0][0], y[0][1], y[0][2] 가 각각 1, 3, 5 로 초기화됩니다. 비슷하게 나머지 두 줄 (2, 4, 6 과 3, 5, 7) 은 y[1] 과 y[2] 를 초기화합니다. 초기치 리스트가 먼저 끝났으므로 y[3] 의 요소들은 0 으로 초기화됩니다. 위 예는 아래의 초기화와 정확하게 동일한 효과를 가져옵니다.

```
float y[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

전체 배열 y 의 초기치는 중괄호로 시작하지만, y[0] 의 초기치는 그렇지 않습니다 (먼저 예에서는 y[0], y[1], y[2] 의 초기치들이 중괄호로 각각 구분되어 있었음); 따라서 y[0] 을 초기화하는데 3 개의 초기치가 사용됩니다. 유사하게 다음 3 개가 y[1] 를 위해, 또 다음 3 개가 y[2] 를 위해 취해집니다. 또 다른 예를 보면,

```
float y[4][3] = {
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

위와 같은 경우에는 (2 차원 배열임을 생각해서) y 의 첫번째 칸 (column) 이 (즉, y[0][0], y[1][0], y[2][0], y[3][0]) 각각 1, 2, 3, 4 로 초기화되며, 나머지는 0 으로 남습니다.

마지막으로,

```
char msg[ ] = "Syntax error in line %s \n";
```

는 문자열로 초기화되는 문자배열을 보여줍니다; 배열의 크기는 널문자까지 포함한 크기가 됩니다.

▷ A8.8 데이터형명 (Type Name)

(캐스트로 명백한 형변환을 지정하거나, 함수 선언자에서 매개변수의 데이터형을 선언하거나, sizeof 연산자의 인자를 지정하는 것 같은) 여러 문맥에서 데이터형의 이름이 필요합니다. **이 때는 대상체의 선언에서 그 대상체의 명칭만을 제외한, 데이터형명 (type name) 을 사용해주면 됩니다**.

```
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 )
```

추상적-선언자 (abstract-declarator) 에서, 그 선언자가 실제 선언에서 사용된다면 있어야만 하는 명칭의 위치를, 한 곳으로 결정하는 것이 가능합니다. **추상적 선언자에 의한 데이터형은 선언자에 가상의 명칭이 있을 때의 데이터형과 동일**합니다. 예를 들면,

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

위의 예는 각각, "정수 (integer)", "정수형 포인터 (pointer to integer)", "3 개의 정수형 포인터를 요소로 갖는 배열 (array of 3 pointers to integers)", "크기를 모르는 정수배열을 가리키는 포인터 (pointer to an array of an unspecified number of integers)", "알 수 없는 매개변수를 갖고, 정수형 포인터를 반환하는 함수 (function of unspecified parameters returning pointer to integer)", "매개변수가 없고 정수형을

반환하는 함수의 포인터를 요소로 하는 크기를 모르는 배열 (array of, unspecified size, of pointer to functions with no parameter returning integer)" 을 나타냅니다.

▷ A8.9 Typedef

기억부류 지정자 typedef 를 포함하는 선언은 대상체를 선언하는 것이 아니라, 주어진 명칭을 데이터형명으로 정의해 줍니다. 이 명칭을 typedef 명이라고 부릅니다.

```
typedef-name:
        identifier
```

typedef 선언은 통상적인 방법 (A8.6 참고) 으로 선언자에 주어진 명칭을, 주어진 데이터형으로 만들어 줍니다. 따라서 **각각의 typedef 명은 그 데이터형을 나타내는 본래의 형지정자와 문법적으로 동일**합니다.

예를 들면,

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

와 같은 선언이 있은 후에,

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

위와 같은 선언은 적법한 것입니다. 즉, b 의 데이터형은 long, bp 의 데이터형은 "long 형 포인터" 이며, z 은 구조체로, zp 는 그 구조체의 포인터로 선언됩니다.

typedef 는 기존에 없는 새로운 데이터형을 만드는 것이 아니라, 지정된 데이터형의 다른 동의어를 만들어 주는 것뿐입니다. 위의 예에서, b 는 다른 long 형 대상체와 동일한 데이터형을 갖습니다.

typedef 명은 안쪽 통용범위 (inner scope) 에서 재선언될 수 있지만, 생략되지 않은 형지정자를 주어야만 합니다. 예를 들어,

```
extern Blockno;
```

는 Blockno 를 재선언해주지 않지만,

```
extern int Blockno;
```

는 재선언해줍니다.

▷ A8.10 데이터형 일치 (Type Equivalence)

두 형지정자 리스트는 생략된 일부 지정자까지 고려해서 (예를 들어, long 은 long int 를 의미함), **동일한 형지정자를 가지고 있다면 동일하다고 말합니다. 다른 택을 갖고 있는 구조체, 공용체, 열거는 구별**되며, **택이 없는 구조체, 공용체, 열거도 각각의 독립적인 데이터형을 지정**하는 것입니다.

함수 매개변수의 명칭을 모두 무시하고, typedef 명을 확장시킨 후에, 추상적 선언자 (abstract declarator, A8.8) **가 형지정자까지 동일하다면, 두 데이터형은 동일한 것**입니다. **배열 크기와 함수 매개변수의 데이터형은 데이터형 비교에서 유효**합니다.

▷ A9. 문장 (Statement)

특별한 경우를 제외하면, 문장은 순서대로 실행됩니다. **문장은 실행효과를 위해 실행되는 것이며, 값을 갖지는 않습니다**. 문장에는 다음과 같은 것들이 있습니다.

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

▷ A9.1 라벨문 (Labeled Statement)

문장에는 라벨이 붙을 수 있습니다.

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

라벨문 중, **명칭 (identifier) 을 포함하는 라벨은 그 명칭을 라벨명으로 선언**합니다. **명칭이 붙은 라벨의 유일한 사용처는 goto 문의 목적지 (target) 로 쓰이는 것**입니다. **라벨명의 통용범위 (scope) 는 그 라벨이 사용된 함수 전체**입니다. **라벨명은 고유의 명칭종류 (name space, A11.1) 를 가지고 있기 때문에, 다른 종류의 명칭과는 충돌하지 않으며, 재선언될 수 없습니다**.

case 라벨과 default 라벨은 switch 문 (A9.4) 에 사용됩니다. **case 라벨의 상수수식은 정수형 (integral) 이어야 합니다**.

라벨문 만으로는 프로그램 제어의 흐름 (flow of control) 을 바꿀 수 없습니다.

▷ A9.2 수식문 (Expression Statement)

대부분의 문장은 수식문이며, 수식문의 형태는 아래와 같습니다.

```
expression-statement:
        expression opt ;
```

대부분의 수식문은 대입문이나 함수호출 입니다. **수식의 모든 부작용 (side effect) 은 다음 문장이 실행되기 전에 완료**됩니다. 수식문에서 **수식이 생략된 형태는 널문장 (null statement) 이라고 하며, 순환문이나 라벨에 빈 본체 (empty body) 를 제공하는데 종종 사용**됩니다.

▶ 역자 주

위에서 라벨문의 문법을 보면 알 수 있듯이, 라벨 뒤에는 반드시 문장이 나와야만 합니다. 하지만 라벨이 블럭 (함수본체 포함) 의 끝부분에 놓이는 등의 이유로 라벨 뒤에 놓일 문장이 없을 때 사용하는 것이 널문장 입니다.

▷ A9.3 복문 (Compound Statement)

하나의 문장이 있는 곳에는, 복문 ('블럭' 이라고도 함) 을 이용해서 여러 문장이 나올 수 있습니다. 함수정의의 본체 부분 (body) 도 복문입니다.

```
compound-statement:
        { declaration-list opt    statement-list opt }

declaration-list:
        declaration
        declaration-list    declaration

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

만약 **선언-리스트 (declaration-list) 에 있는 명칭이 블러 밖에 이미 존재한다면, 바깥쪽 선언의 영향은 블록 안에서는 보류 (suspend) 되며 (A11.1), 블럭을 빠져나오면 다시 영향력을 갖게 됩니다. 이러한 규칙은 같은 종류의 명칭 (same name space) 들에** 적용됩니다 (A11); **다른 종류의 명칭들은 구분되어 다루어집니다.**

▶ 역자 주

블럭 내외로 같은 명칭이 선언된다면, 블록 내에서는 블록 밖의 선언이 영향력을 갖지 못합니다. 예를 들면 아래와 같습니다.

```
int a;
{
        float a=3.141592;
        ...
        a = funcf();       /* 부동형 a */
        ...
}
a = funci();          /* 정수형 a */
...
```

위의 예에서, 바깥쪽의 정수형 변수 a 가 블록 안에서 부동형으로 새로 선언되었지만, 블럭 밖의 a 와 블록 안의 a 는 전혀 별개의 변수입니다. 블럭이 끝나면 부동형 a 는 소멸되고, 정수형 a 가 다시 영향력을 갖게 됩니다. 만약 a 가 블록 안에서 블록 밖의 데이터형과 동일하게 정수형으로 선언되었다고 해도 블록 안팎의 a 는 별개의 변수로 취급됩니다.

또한 같은 블록 내에 있다고 해도 명칭종류 (name space) 가 다르면 동일한 명칭이 있을 수 있습니다. 예를 들면,

```
int whatisthematrix;
...
goto whatisthematrix;
...
whatisthematrix:
...
```

위의 예에서, 변수명과 라벨명이 모두 whatisthematrix 로 같지만, 변수명과 라벨명은 명칭종류 (name space) 가 다르기 때문에 서로 충돌하지 않고 사용할 수 있습니다.

자동 기억부류의 대상체는 블럭에 진입할 때마다 초기화되며, 선언자 (declarator) 가 쓰인 순서대로 초기화됩니다. **만약 점프문으로 인해 블록 안으로 진입하게 되면, 초기화는 일어나지 않습니다. 정적 대상체는 프로그램이 실행되기 전에 딱 한번만 초기화**됩니다.

▷ A9.4 선택문 (Selection Statement)

선택문은 주어진 여러 가지 제어흐름 중에 하나를 선택할 때 사용됩니다.

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

위에 제시된 두 가지 형태의 if 문에서, **수식 (산술형 arithmetic type 이나 포인터형 pointer type 이어야 함) 이 모든 부작용 (side effect) 을 포함해 평가**된 후, 수식값이 0 과 다르면 ('참'이면) 첫번째 문장이 실행됩니다. 두번째 형태의 if 문에서는 수식의 값이 0 이면 ('거짓'이면) 두번째 문장이 실행됩니다. 복잡한 if-else 문에서, **else 에 생기는 모호함 (어떤 if 문과 연결된 else 인가?) 은 같은 중첩레벨 (nest level) 의 블럭에 있는, else 문이 없는 마지막 if 문과 연결시킴으로써 해결**됩니다.

switch 문은 **정수형 수식 (integral)** 의 값에 따라 여러 문장 중 하나로 프로그램 제어를 옮깁니다. **switch 문에 주어지는 문장은 전형적으로 복문**입니다. **복문 안의 어떠한 문장에도 case 라벨 (A9.1) 이 붙을 수 있습니다**. switch 문의 흐름제어 수식에는 **정수진급 (integral promotion, A6.1) 이 수행되며, case 상수들은 그 진급된 데이터형으로 변환**됩니다. **하나의 switch 문안에는, (변환된 후의 데이터형을 갖는) 동일한 case 상수가 중복되어 존재할 수 없습니다**. 또한 복문 안에는 **최대 1 개의 default 라벨이 있을 수 있습니다**. switch 문은 **중첩될 수 있습니다**; case 와 default **라벨은 그 라벨을 포함하는 가장 안쪽의 switch 문과만 관련**이 있습니다.

▶ 역자 주

즉, 같은 switch 문에는 중복된 case 상수가 있을 수 없지만, 중첩된 switch 문의 안팎으로는 중복된 case 상수가 있어도 구분이 되기 때문에 가능하다는 의미입니다.

switch 문이 실행되면, **모든 부작용 (side effect) 을 포함해 수식이 평가**되고, 각 case 상수와 비교됩니다. 수식의 값이 case 상수 중 하나와 일치한다면, 프로그램 제어는 그 case 라벨로 옮겨집니다. 일치하는 case 라벨이 없고, default 라벨이 있다면, default 라벨로 제어가 옮겨집니다. 만약 default 라벨마저 없다면 switch 문의 문장들은 실행되지 않습니다.

초판에서, switch 문의 흐름제어 수식과 case 수식은 int 형이어야만 했습니다.

▷ A9.5 순환문 (Iteration Statement)

순환문은 루프 (loop) 를 실행해 줍니다.

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

while 과 do 문에서, 귀속된 문장 (substatement) 은 수식의 값이 0 이 아닌 동안 ('참'인 동안) 반복해서 실행됩니다; **수식은 산술형 (arithmetic type) 이나 포인터형 (pointer type) 이어야 합니다**. while 문에서 조건검사는 매 루프가 시작되기 전에 **수식의 부작용 (side effect) 까지 포함해** 이루어집니다; do 문에서는 매 루프가 실행된 후에 이루어집니다.

for 문에서, 첫번째 수식은 한번만 평가되고, 따라서 루프를 초기화하는데 적당합니다. **첫번째 수식의 데이터형에는 제약이 없습니다**. **두번째 수식은 산술형 (arithmetic type) 이나 포인터형 (pointer type) 이어야 합니다**; 두번째 수식은 매 루프를 돌기 전에 평가되며, 수식의 값이 0 이 되면 for 문은 종료됩니다. 세번째 수식은 매 루프를 돈 후에 평가되고, 따라서 루프제어변수의 재초기화 (re-initialization) 에 사용하면 됩니다. **세번째 수식의 데이터형에는 제약이 없습니다**. **각 수식의 부작용 (side-effect) 은 그 수식이 평가된 후에 바로 완료**됩니다. for 문의 문장이 continue 문을 포함하지 않는다면, 다음의 문장은

```
for ( expression1 ; expression2 ; expression3 ) statement
```

아래와 동일합니다.

```
expression1 ;
while ( expression2 ) {
        statement
        expression3 ;
}
```

세 수식 중 어떠한 것도 생략될 수 있습니다. **두번째 수식이 생략된다면 조건검사는, 0 이 아닌 상수값으로 (항상 '참'으로) 평가**됩니다.

▷ A9.6 점프문 (Jump Statement)

점프문은 프로그램의 제어를 무조건 옮깁니다.

```
jump-statement:
        goto identifier ;
        continue ;
        break ;
        return expression opt ;
```

goto 문에서 **명칭 (identifier) 은 현재 함수 안의 라벨 (A9.1) 이어야** 합니다. 프로그램 제어가 그 라벨로 옮겨집니다.

continue 문은 순환문에만 나올 수 있습니다. continue 문은, continue 문이 있는 가장 안쪽 루프의 남아있는 문장들을 건너뛰게 해줍니다. 더 자세하게, 다음의 각 문장에서,

```
while (...) {          do {                   for (...) {
        ...                    ...                    ...
contin: ;              contin: ;              contin: ;
}                      } while (...);         }
```

continue 문이 중첩된 안쪽 순환문에 있지 않는 한, continue 문의 행동은 goto contin 을 사용한 것과 동일합니다.

break 문은 오직 순환문과 switch 문에서만 나올 수 있으며, break 문이 있는 가장 안쪽 루프를 벗어나게 해줍니다; 프로그램의 제어는 벗어난 루프의 다음 문장으로 옮겨집니다.

함수는 return 문에 의해 그 함수를 호출한 곳으로 돌아갑니다. return 문 뒤에 수식이 따라오면, 그 수식의 값은 그 함수의 호출자 (caller) 로 반환됩니다. **수식은 마치 대입된 것처럼 함수의 반환형으로 변환**됩니다.

프로그램의 제어가 함수의 끝부분까지 도달해 함수가 종료된다면, 이는 수식이 없는 return 문을 사용한 것과 동일합니다. **두 경우 모두, 반환되는 값은 정의되어 있지 않습니다 (undefined)**.

▷ A10. 외부선언 (External Declaration)

C 컴파일러로 제공되는 소스의 입력단위 (unit of input) 를 번역단위 (translation unit) 라고 합니다. 번역단위는 선언 (declaration) 이나 함수정의 (function definition) 를 의미하는 외부선언 (external declaration) 의 나열로 구성됩니다.

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

external-declaration:
        function-definition
        declaration
```

▶ 역자 주

번역단위 (translation unit) 란, 컴파일러에 의해 인식되어 하나의 단위로 번역되는 소스파일의 집합을 의미합니다. 보통 #include 지시자에 의해 포함되는 모든 헤더파일을 더한, 하나의 .c 파일 (소스파일) 입니다.

외부선언의 통용범위는, 블록 내에서 선언의 효과가 블럭의 끝까지 지속되는 것처럼, **그것들이 선언되는 번역단위의 끝까지 지속**됩니다. **외부선언의 문법은, 함수의 정의가 이 단계 (level) 에서만 주어질 수 있다는 것을 제외하면 다른 모든 선언과 동일**합니다.

▷ A10.1 함수정의 (Function Definition)

함수정의는 다음과 같은 형태를 갖습니다.

function-definition:
declaration-specifiers opt declarator
declaration-list opt compound-statement

▶ 역자 주
위에 제시된 함수정의의 문법은 원문에는 한 줄로 제시된 것입니다. 문법이 너무 길어 한 줄에 다 넣을 수 없어 부득이하게 두 줄로 나눠서 씁니다.

선언 지정자 (declaration specifier) 에 허락되는 유일한 기억부류 지정자는 extern 이나 static 입니다; 두 지정자의 차이점은 A11.2 절에서 다룹니다.
함수는, 함수나 배열을 제외한 산술형, 구조체, 공용체, 포인터, void 형을 반환할 수 있습니다. 함수선언의 선언자 (declarator) 는 선언되는 명칭이 함수형임을 명백히 지정해줘야 합니다; 따라서 다음 형태 중 하나를 포함해야만 합니다. (A8.6.3 참고)

direct-declarator (parameter-type-list)
direct-declarator (identifier-list opt)

위에서 직접-선언자 (direct-declarator) 는 명칭이거나 괄호에 쌓인 명칭이어야 합니다. 특별히, **typedef 의 방법으로 함수를 정의할 수는 없습니다.**
첫번째 형태는, 현대적 방식 (new-style) 의 함수정의이며, 매개변수는 데이터형과 함께 매개변수 데이터형 리스트 (parameter type list) 에서 선언됩니다; **함수의 선언자 (declarator) 뒤에 오는 선언-리스트 (declaration-list) 는 없어야만 합니다**. 만약 **매개변수 데이터형 리스트에 매개변수가 없다는 것을 의미하는 void 가 혼자 쓰인 경우가 아니라면, 매개변수 데이터형 리스트의 각 선언자는 반드시 명칭을 포함해야 합니다.** 매개변수 데이터형 리스트가 ", ..." 로 끝난다면, 그 함수는 매개변수 (parameter) 보다 더 많은 개수의 인자 (argument) 를 받을 수 있습니다; 표준 헤더파일 <stdarg.h> 에서 정의된 va_arg 매크로 기법을 사용해야만 그 가변인자를 참조할 수 있습니다. **가변인자 함수 (variadic function) 에는 적어도 1 개의 이름이 붙은 매개변수가 있어야 합니다.**
두번째 형태는, 고전적 방식 (old-style) 의 함수정의 입니다: 명칭 리스트 (identifier list) 는 매개변수의 이름들이며, 선언 리스트 (declaration list) 에서 그 매개변수의 데이터형을 선언해 줍니다. 만약 어떤 매개변수에 데이터형이 주어지지 않으면, int 형으로 취급됩니다. **선언 리스트는 명칭 리스트에 있는 매개변수만을 선언해야 하며, 초기화는 허락되지 않고, 사용할 수 있는 기억부류 지정자는 register 뿐입니다.**
두 가지 형태의 함수정의에서, **매개변수들은 함수본체 (body) 를 구성하는 복문 (compound statement) 이 시작하자마자 선언된다고 이해**됩니다. 따라서 **동일한 명칭이 함수본체에서 재선언될 수 없습니다. (단, 다른 명칭들과 동일하게, 안쪽 블럭에서는 재선언될 수 있습니다)** 만약 **매개변수가 "배열형 (array of type)" 이라면 선언이 "포인터형 (pointer to type)" 으로 조정되며, 유사하게 "함수형 (function returning type)" 이라면 "함수 포인터형 (pointer to function returning type)" 으로 조정**됩니다. 함수를 호출하는 동안, 인자 (argument) 는 필요에 따라 변환되고, 매개변수로 대입됩니다. (A7.3.2 참고)

현대적 방식의 함수정의는 ANSI 표준안에서 새로운 것입니다. 여기에는 진급 (promotion) 에 대한 또 하나의 작은 변화가 있습니다: 초판에서는 float 형의 매개변수가 double 형으로 조정되었습니다. 이 변화는 함수 안에서 매개변수를 가리키는 포인터를 생성하는 경우에야 겨우 눈에 띌만한 것입니다.

현대적 방식의 함수정의에 대한 완벽한 예가 아래에 있습니다.

```
int max(int a, int b, int c)
{
        int m;
        m = (a > b) ? a : b;
        return (m > c) ? m : c;
}
```

여기서 첫번째 int 는 선언 지정자 (declaration specifier) 입니다; max(int a, int b, int c) 는 함수의 선언자이며, { ... } 부분은 함수본체를 위한 블럭입니다. 위의 예에 대응하는 고전적 방식의 함수정의는 아래와 같습니다.

```
int max(a, b, c)
int a, b, c;
{
        /* ... */
}
```

위에서 int max(a, b, c) 는 선언자 (declarator) 이며, int a, b, c; 는 매개변수의 선언 리스트 (declaration list) 입니다.

▷ A10.2 외부선언 (External Declaration)

외부선언은 대상체, 함수, 혹은 다른 명칭들의 특성을 명시해 줍니다. **여기서 사용하는 "외부 (external)" 라는 용어는 함수의 밖이라는 위치를 의미하는 것일 뿐, extern 키워드와는 직접적으로 관계가 없습니다; 외부에서 선언된 대상체의 기억부류 지정자는 없어도 (empty) 좋으며, extern 이나 static 을 지정해줘도 됩니다.**

만약 그 명칭에 대한 정의가 한 개만 존재한다면, 데이터형과 연결 (linkage) 이 일치하는 한, 같은 명칭에 대한 여러 개의 외부선언이 동일한 번역단위 (translation unit) 내에 존재할 수 있습니다.

두 선언은 A8.10 절에서 설명했던 규칙에 의해, 데이터형이 동일한가 판단됩니다. **만약, 한 선언의 데이터형은 불완전한 (incomplete) 구조체 · 공용체 혹은 열거형 (A8.3) 이고, 다른 선언의 데이터형은 동일한 택을 갖고 있는 대응하는 완전한 형 (complete type) 이라면, 두 데이터형은 일치한다고 취급**됩니다. 또, **한 데이터형은 불완전한 배열형 (A8.6.2) 이고, 다른 선언은 완전한 배열형이며, 그 외 다른 부분이 동일하다면, 그 두 데이터형도 일치한다고 취급**됩니다. 마지막으로 **한 데이터형은 고전적 방식 (old-style) 의 함수이고, 다른 데이터형은 다른점에서 동일하고 매개변수 선언 (parameter declaration) 이 있는 현대적 방식 (new-style) 의 함수라면, 그 두 데이터형도 일치한다고 취급**됩니다.

만약 **함수나 대상체의 첫번째 외부선언이 static 지정자를 포함한다면, 그 명칭은 내부연결 (internal linkage) 을 갖습니다; 그렇지 않으면 외부연결 (external linkage) 을 갖습니다.** 연결 (linkage) 에 대해서는 A11.2 절에서 설명합니다.

대상체의 외부선언이 초기치를 가지고 있다면 그것은 정의 (definition) 가 됩니다. **대상체의 외부선언이 초기치를 가지고 있지 않으며, extern 지정자를 포함하지 않는다면, 그 선언은 임시정의 (tentative definition)** 가 됩니다. 만약 **대상체의 정의가 번역단위 (translation unit) 내에 나타난다면, 모든 임시정의는 다만 중복된 선언으로** 다루어집니다. **대상체의 정의가 번역단위 내에 나타나지 않으면, 그 대상체의 모든 임시정의는 초기치 0 을 갖는 하나의 정의**가 됩니다.

각 대상체는 한번만 정의될 수 있습니다. 내부연결 (internal linkage) 을 갖는 대상체는

각 번역단위에서 유일하기 때문에, 이러한 규칙이 각 번역단위마다 분리되어 적용됩니다. **외부연결 (external linkage) 을 갖는 대상체는 이 규칙이 프로그램 전체에 적용**됩니다.

> 이 책의 초판에서는 정의는 한번만 있어야 한다는 규칙이 다소 다르게 설명되었지만, 실제로는 여기에서 서술한 것과 동일한 내용입니다. 일부 컴파일러에서는 임시정의 (tentative definition) 의 개념을 일반화해서 그 규칙을 다소 완화하였습니다. UNIX 시스템에서는 이미 일반적이며, 표준에 의해 일반적인 확장 (common extension by the Standard) 으로 인정된, 대체된 규칙에서는, 외부연결 (external linkage) 을 갖는 대상체의 모든 임시정의는, 분리된 각 번역단위가 아닌, 프로그램의 모든 번역단위 전체를 통틀어서 판단됩니다. 만약 프로그램 어딘가에 정의가 나타난다면, 임시정의는 선언으로 바뀌며, 정의가 나타나지 않는다면, 모든 임시정의가 초기치 0 을 갖는 하나의 실제 정의가 됩니다.

▷ A11. 통용범위와 연결 (Scope and Linkage)

프로그램은 한번에 모두 컴파일될 필요는 없습니다; 프로그램의 소스는 각 번역단위 (translation unit) 를 담는 여러 개의 파일로 나누어질 수 있으며, 라이브러리 (library) 로부터 미리 컴파일된 프로그램 루틴 (routine) 이 로드 (load) 될 수 있습니다. 프로그램에서 함수간의 의사소통은 함수호출과 외부 데이터를 통해서 이루어집니다.

따라서, 두 가지 종류의 통용범위를 생각해 볼 수 있습니다. 첫째로, **프로그램 소스에서 어떠한 명칭의 특성이 알려지는 범위를 의미하는, 어휘적 통용범위 (lexical scope)** 가 있습니다; 둘째로는 **분리되어 컴파일되는 번역단위에서 명칭들 사이의 관계 (connection) 를 결정하는 외부연결 (external linkage) 을 갖는, 대상체와 함수에 관련된 통용범위**입니다.

▷ A11.1 어휘적 통용범위 (Lexical Scope)

명칭들은 서로간에 충돌하지 않는 여러 종류로 분류됩니다. 만약 **명칭의 종류가 다르다면 (different name space), 같은 명칭이 같은 통용범위 (scope) 에서 다른 목적으로 사용될 수 있습니다**. 명칭의 종류 (name space) 는 다음과 같습니다: **대상체, 함수, typedef명, enum 상수; 라벨; 구조체 · 공용체 · 열거의 택; 독자적으로 취급되는 각 구조체와 공용체의 멤버**입니다.

> 이 규칙은 초판에서 서술한 것과는 여러 가지로 다릅니다. 라벨은 고유의 명칭종류 (name space) 를 갖지 못했었습니다; 또한 구조체, 공용체의 택이 각각 독립된 명칭종류를 가졌으며, 일부 컴파일러에서는 열거택도 독립된 명칭종류를 갖을 수 있었습니다; 다른 종류의 택을 같은 종류의 명칭으로 묶은 것은 새로운 제약입니다. 초판과 비교해 가장 중요한 변화는 각 구조체와 공용체가 자신의 멤버들에 대해 독립된 명칭종류를 생성한다는 것입니다. 이로 인해, 여러 다른 구조체에 동일한 명칭의 멤버가 나올 수 있습니다. 이러한 규칙은 이미 여러해동안 컴파일러에서 일반적으로 사용되고 있었습니다.

외부선언 (external declaration) 에 있는 대상체명이나 함수명의 어휘적 통용범위는 그 선언자 (declarator) 의 끝에서 시작해 그 번역단위 (translation unit) 의 끝까지 지속됩니다. **함수정의에 있는 매개변수의 통용범위는** 함수를 정의하는 본체 (body) 에서 시작해서 그 함수 안에서 계속 지속됩니다; **함수선언에 있는 매개변수의 통용범위는** 선언자 (declarator) 의 끝에서 함께 끝납니다. **블럭의 상단부에서 선언된 명칭의 통용범위는** 그 선언자의 끝에서 시작해서, 그 블럭의 끝까지 지속됩니다. **라벨의 통용범위는** 그 라벨이 있는 함수 전체입니다. **구조체 택, 공용체 택과 열거 상수의 통용범위는** 형지정자 (type specifier) 에 그것들이 나타나면서 시작해서, (함수 외부에서 선언될 경우에는) 번역단위의 끝까지, (함수 안에서 선언될 때에는) 블럭의 끝까지 지속됩니다.

함수 본체를 구성하는 블럭을 포함해, 블럭의 상단부에서 명칭이 명백히 선언된다면, 블럭 밖에 있는 동일한 명칭의 선언은 블럭이 끝날 때까지 보류 (suspend) 됩니다.

▷ A11.2 연결 (Linkage)

한 번역단위 (translation unit) 내에서, **내부연결 (internal linkage) 을 갖는 대상체명이나 함수명의 모든 선언은 같은 것을 참조하고, 그 명칭이나 함수는 그 번역단위 안에서 유일 (unique) 해야** 합니다. **외부연결 (external linkage) 을 갖는 같은 대상체명이나 함수명의 모든 선언은 같은 것을 참조하고, 그 대상체나 함수는 프로그램 전체에 걸쳐 공유**됩니다.

A10.2 절에서 이야기했듯이, **static 지정자가 어떠한 명칭의 첫번째 외부선언 (external declaration) 에 사용된다면 그 명칭은 내부연결 (internal linkage) 을 갖으며, 그렇지 않으면 외부연결 (external linkage) 을 갖습니다. 블록 안에 있는 명칭의 선언에 extern 지정자가 없으면, 그 명칭은 연결을 갖지 않으며, 함수 안에서만 유일 (unique)** 합니다. 만약 **블록 안에서 extern 지정자를 포함하고, 그 명칭의 외부선언이 블럭을 둘러싸고 있는 통용범위 내에서 유효 (active) 하다면, 그 명칭은 외부선언과 동일한 연결을 갖으며, 같은 대상체나 함수를 참조**합니다; **외부선언이 보이지 않는다면, 그 명칭의 연결은 외부연결 (external linkage) 로 결정**됩니다.

▷ A12. 전처리기 (Preprocessor)

전처리기 (preprocessing) 는, 매크로 치환 (macro substitution), 조건부 컴파일 (conditional compilation), 파일 첨가 (inclusion of named file) 를 수행합니다. **# 로 시작하고, 공백문자 (white space) 가 붙기도 하는 라인 (line) 은 전처리기와 관련이 있는 것**입니다. **이러한 라인에 적용되는 문법은 C 언어의 문법과는 관련이 없습니다**; **전처리기는 프로그램의 어디에라도 나올 수 있으며, (통용범위와 상관없이) 그 영향이 번역단위 (translation unit) 의 끝까지 지속**됩니다. **라인의 경계는 유효**합니다; 즉, **각 라인은 개별적으로 분석**됩니다. (단, A12.2 절에서 라인을 연결할 수 있는 방법을 제공합니다) **전처리기의 토큰 (token) 은 C언어의 모든 토큰, 혹은 #include 지시자 (A12.4) 에 파일명 (file name) 으로 제공되는 문자열**입니다; 게다가 **다르게 정의되지 않은 모든 문자는 토큰으로 취해집니다**. 하지만, **전처리기 라인 안에서, 공백 (space), 수평탭 (horizontal tab) 이 아닌, 다른 공백문자 (white space) 의 영향은 정의되지 않았습니다 (undefined).**

전처리기 기능 자체는 (**특정 컴파일러에서는 몇몇 순서가 생략되기도 하는**) 여러 개의 논리적 과정 가운데서 일어납니다. 그 과정 (phase) 이란 아래와 같습니다.

1. 우선, A12.1 절에서 설명하는 **삼중자 (trigraph sequence)** 가 동일한 의미의 문자로 치환됩니다. 또한 **운영체제가 요구하는 경우, 소스파일의 각 라인 사이에 개행문자 (newline character) 가 삽입**됩니다.
2. 개행문자 (newline) 앞에 있는 백슬러쉬 문자 \ 가 제거되고, **두 라인이 연결**됩니다. (A12.2)
3. 프로그램 소스가 **공백문자 (white space) 로 분리된 토큰 (token) 으로 분석**됩니다; 이 때 **주석은 하나의 공백 (space) 으로 치환**됩니다. 그런 후에, **전처리기 지시자 (preprocessing directive) 에 따라 행동**하고, **매크로 (A12.3 ~ A12.10) 가 확장**됩니다.
4. 문자상수와 문자열 (A2.5.2, A2.6) 내의 **확장열 (escape sequence) 이 동일한 의미의 문자로 치환**되고, **인접해 있는 문자열이 연결**됩니다.
5. 필요한 프로그램 데이터를 수집하고, 외부 함수와 대상체 참조 (object

reference) 를 정의 (definition) 에 연결함으로써, **결과가 컴파일러에 의해 번역**되고, **다른 프로그램, 라이브러리 (library) 와 함께 링크 (link)** 됩니다.

▷ A12.1 삼중자 (Trigraph Sequence)

C 프로그램 소스의 문자세트 (character set) 는 7비트 ASCII 코드 내에 포함되지만, ISO 646-1983 Invariant Code Set 보다 큰 문자세트입니다. (즉, 집합으로 표현하면, **7비트 ASCII 코드 ⊃ C 소스 문자세트 ⊃ ISO 646-1983 Invariant Code Set** 입니다)

> **▶ 역자 주**
> ISO 646 은 1983년 ISO 에 의해 표준화된 82자의 기본 문자세트 입니다. C 언어의 소스문자세트와 실행문자세트 (source and execution character set) 는 ISO 646 안의 문자들을 모두 포함해야 합니다. 하지만 ISO 646 문자세트는 C 언어에서 사용되는 모든 문자를 담고 있지 않으며, 일부 외국어의 문자세트 역시 마찬가지인 경우가 있습니다.

프로그램을 더 적은 문자세트로 표현할 수 있도록, 다음에 나오는 모든 삼중자 (trigraph sequence) 는 대응하는 하나의 문자로 대체됩니다. 이러한 **삼중자의 대체작업은 다른 어떤 전처리 기능 보다 우선**합니다.

??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!	|	??-	~

위에서 나열된 삼중자 외에 대체작업은 일어나지 않습니다.

삼중자는 ANSI 표준에서 새로운 것입니다.

▷ A12.2 라인 연결 (Line Splicing)

백슬러쉬 \ 로 끝나는 라인은, 백슬러쉬와 그 뒤에 따라오는 개행문자 (newline) 을 제거해서 다음 라인과 연결됩니다. 이러한 **라인 연결은 소스가 토큰으로 분석되기 전에 일어납니다.**

▷ A12.3 매크로 정의와 확장 (Macro Definition and Expansion)

define identifier token-sequence

위와 같은 제어라인 (control line) 은 프로그램 소스에 나타나는 명칭 (identifier) 을 주어진 토큰열 (token-sequence) 로 바꿔줍니다; **토큰열의 앞뒤에 있는 공백문자 (white space) 는 제거**됩니다. **토큰 안에 있는 공백문자까지 고려해서, 주어진 토큰열이 동일하지 않으면, 동일한 명칭의 중복된 매크로가 주어지는 것은 잘못된 것**입니다.

define identifier(identifier-list $_{opt}$) token-sequence

첫번째 명칭 (identifier) 과 여는 괄호 (사이에 공백이 없어야 하는, 위와 같은 형태의 라인은 명칭 리스트 (identifier-list) 를 매개변수 (parameter) 로 갖는, 매크로

정의입니다. 첫번째 형태에서처럼, **토큰열 앞뒤의 공백문자 (white space) 는 제거**되며, **매개변수의 개수·철자와 토큰열이 동일해야만 매크로가 중복 정의될 수 있습니다**.

```
# undef identifier
```

위와 같은 제어라인 (control line) 은 명칭의 전처리기 정의를 없던 것으로 만들어 줍니다. **정의되지 않은 명칭에 #undef 를 적용하는 것은 잘못된 것이 아닙니다**.

매크로가 두번째 형태로 정의되면, **프로그램 소스에 나오는 매크로 명칭 (매크로 명칭 뒤에는, 생략 가능한 공백문자, 여는 괄호, 쉼표로 분리된 토큰열들, 닫는 괄호가 따라옵니다) 은 그 매크로를 호출**합니다. 호출시의 인자 (argument) 는 쉼표로 분리된 토큰열입니다; **따옴표에 쌓여 있거나, 중첩된 괄호로 보호되는 (protected) 쉼표는 인자를 분리하지 않습니다**. 매크로 명칭을 검사 (scan) 할 때, **매크로 인자 (argument) 안에 있는 매크로는 확장되지 않습니다**.

▶ 역자 주

아래와 같은 경우, 인자에 있는 매크로 FRAG 는 (FOO 를 확장하는 과정에서 지워져 버리므로) 아예 확장되지도 않습니다.

```
#define FRAG 1
#define FOO(x,y) (x)

FOO(x,FRAG)
```

매크로 인자 안에 있는 매크로는 미리 확장되지 않기 때문에, FRAG 가 1 로 확장되기 전에 FOO 가 확장되면서 FRAG 를 없애버리게 됩니다.

호출시 인자 (argument) 의 개수는 매크로 정의에 있는 매개변수 (parameter) 의 개수와 일치해야만 합니다. 확장 작업에서, **각 인자의 앞뒤에 있는 공백문자 (white space) 는 제거**됩니다. 그런 후에, 인자로 주어지는 토큰열들은, 매크로의 치환 토큰열 (replacement token sequence) 에서 대응하는 매개변수 (parameter) 가 **따옴표 없이** 나올 때마다 치환됩니다.

▶ 역자 주

따옴표 안에 있는 매개변수는 문자열이나 문자상수의 일부로 취급되기 때문에 확장되지 않습니다. 만약 주어지는 인자를 문자열로 만들고자 할 때는 다음에 설명하는 # 연산자를 사용하면 됩니다.

치환 토큰열에 나오는 매개변수에 # 나 ## 가 붙어있지 않는 한, 인자로 주어지는 토큰은 매크로 호출을 위해 검사되고, 삽입 (insertion) 되기 전에 필요에 따라 확장됩니다.

▶ 역자 주

전처리기는 프로그램 소스상에 보이는 토큰의 매크로 정의가 있는지 찾아보고, 있다면 그 매크로를 확장해 줍니다. 단 # 연산자의 뒤나, ## 연산자 앞뒤에 있는 것은 다른 매크로로 정의되어 있다고 해도 확장되지 않습니다. (새로운 토큰을 만들기 위해서 입니다)

```
#define X Y
#define A Z ## X
```

위와 같은 상태에서 매크로 A 를 호출하면, ZY 가 아닌 ZX 로 치환됩니다.

두개의 특별한 연산자가 치환 작업 (replacement process) 에 영향을 줍니다. 첫째, 치환 토큰열 (replacement token sequence) 에 있는 매개변수 (parameter) 앞에 # 가 붙어 있다면, **대응하는 매개변수 주위에 문자열 따옴표(")가 들어가고, # 와 매개변수 명칭이 인자 (argument) 로 치환**됩니다. (**인자에 포함된, 문자열이나 문자상수의, 내부 혹은 주변에 있는 " 나 \ 문자 앞에는 \ 가 추가됨**니다)

▶ 역자 주

" 나 \ 문자 앞에 \ 를 추가해 주는 것은, 주어진 문자를 확장열로 만들어 주기 위해서 입니다.

둘째, **두 종류의** 매크로 정의에서, 치환 토큰열 (replacement token sequence) 이 **## 연산자를 포함한다면, 매개변수 치환이 일어난 후에 곧바로, ## 와 양쪽의 공백문자 (white space) 가 지워져서, 인접한 토큰이 연결되어 하나의 토큰이 형성**됩니다.

▶ 역자 주

당연한 이야기겠지만, 다음과 같은 경우를 생각해 볼 수 있습니다.

```
#define FOO 1
#define BAR 2
#define FOOBAR 3
#define conexp(A,B) A ## B

conexp(FOO,BAR)
```

위에서 conexp(FOO,BAR) 의 호출 결과는, 12 (FOO 의 1, BAR 의 2) 가 아니라 3 (FOOBAR 의 3) 이 됩니다. 위에서 이야기했듯이, 매개변수 치환이 일어난 후에 곧바로 두개의 토큰 (FOO 와 BAR) 이 연결되어, 하나의 새로운 토큰 (FOOBAR) 을 형성하기 때문입니다.

만약, **유효하지 않은 토큰이 생성되거나, 결과가 ## 연산자의 작업순서에 따라 달라지는 경우라면, 그 영향은 알 수 없습니다 (undefined)**.

▶ 역자 주

C90 표준안에서, ## 연산자의 평가순서는 명시되지 않는다고 (unspecified) 되어 있습니다. 만약 ## 연산자의 작업순서에 의존적인 경우, 가능한 순서 중 하나라도 부적절한 전처리기 토큰을 생성한다면, 그 행동을 알 수 없게 됩니다 (undefined). 이와 관련된 예를 이해하기 위해서는 C 언어의 전처리기 토큰 중 하나인 전처리기 숫자 (pp-number) 의 개념을 필요로 합니다. 여기서는 전처리기 숫자에 대한 개념 설명은 생략하고 예를 설명하겠습니다.

```
314159 ## e ## - ## 5
```

우선 하나의 토큰을 형성하기 위해 3개의 ## 연산자가 모두 필요합니다. 이때 ## 연산자가 좌측에서 우측으로 차례대로 평가되지 않을 경우, 최종적으로 생성되는 토큰 (314159e-10) 은 올바른 토큰일지라도 중간에 생성되는 토큰 (e-, -5, e-5) 은 모두 부적절한 토큰이 되어 결과를 알 수 없게 됩니다.

참고로, 이 문서에서 예전에 사용했던 아래의 예제와 관련된 설명은 잘못된 것입니다.

```
AA ## 11_ ## 22
```

11_ ## 22 가 먼저 평가되면 올바른 숫자도 명칭도 아닌 부적절한 토큰 (11_22) 이 생성된다고 설명했으나, 정수 상수와 부동 상수의 어휘적 형태를 모두 포함하는 전처리기 숫자 토큰에 의해 11_22 역시 유효한 전처리기 토큰이 됩니다.

또한 **## 는 치환 토큰열의 맨 앞이나 맨 뒤에 나타날 수 없습니다.**

두 종류의 매크로 정의에서, 치환 토큰열 (replacement token sequence) **은 다른 매크로 정의 명칭으로 반복해서 검사** (scan) 됩니다. 하지만, **치환 토큰열 안에 그 토큰열을 확장해주는 매크로 명이 들어가 있다면 (중첩된 매크로 사이에 들어 있다고 해도), 한번만 확장됩니다. 즉, 재검사** (rescan) **할 때 그 매크로가 다시 발견된다고 해도 확장되지 않습니다.**

▶ **역자 주**

다음과 같은 경우를 말하는 것입니다.

```
#define char unsigned char
```

위와 같이 매크로가 정의되어 있다면, 프로그램 소스상의 char 는 unsigned char 로 치환됩니다. 만약 위에서 설명한 규칙이 존재하지 않는다면, unsigned char 의 일부분인 char 가 또 매크로로 인식되는 일이 반복되므로 언젠가는 시스템 에러로 전처리 기능이 종료됩니다. 하지만 위에서 이야기한 규칙이 이를 방지해 주기 때문에 한번만 확장되어 unsigned char 까지만 확장해 줍니다.

```
#define ONE IS TWO
#define TWO ARE THREE TWO
#define THREE WERE ONE
```

또한 여러 번 중첩된 매크로 사이에서도 위의 규칙은 제대로 적용됩니다. 따라서 위와 같이 매크로를 정의해 놓은 상태에서, ONE 을 호출하면 정확하기 IS ARE WERE ONE TWO 로만 확장됩니다. (더 복잡하게 만들면 더 재미있습니다 ^^;)

만약 **매크로 확장의 최종 결과가 # 로 시작한다고 해도, 이는 전처리기 지시자로 취급되지 않습니다**.

> 매크로 확장 작업의 상세한 내용은 초판보다 ANSI 에서 더 세밀하게 정의했습니다. 가장 중요한 변화는, 문자열을 만들어 주고, 토큰을 연결해 주는 # 와 ## 연산자를 추가한 것입니다. 새 규칙 중 일부, 특히 토큰 연결과 관련된 부분은 조금 특이합니다. (아래 예를 보시기 바랍니다)

예를 들어, 아래와 같은 매크로는 "manifest constant" 를 사용할 수 있도록 해줍니다.

```
#define TABSIZE 100
int table[TABSIZE];
```

아래와 같은 경우는, 두 인자 사이의 차를 절대값으로 반환하는 매크로를 정의해 줍니다.

```
#define ABSDIFF(a, b)   ((a)>(b) ? (a)-(b) : (b)-(a))
```

동일한 일을 담당하는 함수와는 달리, **매크로로 작성된 경우에는 인자와 반환값이 어떠한 산술형 (arithmetic type) 이나 포인터 (pointer) 여도 상관없습니다**. 또한 **주어진 인자는, 검사하는데 한번, 값을 생성하는데 한번, 총 두 번 평가되므로 부작용 (side effect) 이 발생할 수도** 있습니다.

다음과 같은 정의가 주어지면,

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

매크로 호출 tempfile(/usr/tmp) 는 아래와 같은 결과를 생성합니다.

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

또 위의 결과는 하나의 문자열로 연결됩니다. 다음과 같은 매크로가 정의되면,

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

cat(var,123) 와 같은 호출은 var123 이라는 토큰을 생성합니다. 하지만, **cat(cat(1,2),3) 와 같은 호출은 정의되어 있지 않습니다 (undefined). ## 의 존재는 바깥쪽 호출의 인자가 확장되지 않도록 해줍니다**. 따라서 다음과 같은 토큰이 생성됩니다.

```
cat ( 1 , 2 )3
```

여기서 첫번째 인자의 마지막 토큰과 두번째 인자의 첫번째 토큰이 연결된)3 은 적법하지 않은 토큰이 됩니다. 만약 **다음과 같이 두 단계에 걸쳐 매크로가 정의되면,**

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

좀 더 유연하게 작동합니다; **xcat 의 확장 자체는 ## 연산자와 관련이 없기 때문에, xcat(xcat(1, 2), 3) 는 올바르게 123 을 생성**합니다.

▶ **역자 주**

xcat(xcat(1,2),3) 의 확장과정은 다음과 같습니다.

단계	변환 결과
1. (원래상태)	xcat(xcat(1,2),3)
2. (인자확장)	xcat(1,2) cat(1,2) 12
3.	cat(12,3)
4. (완료)	123

위에서, 가장 바깥쪽 호출 xcat() 의 두 인자는 xcat(1,2) 와 3 이며, 첫번째 인자인 xcat(1,2) 가 매크로 호출이므로 치환 리스트에 삽입되기 전에 완전히 확장됩니다.

xcat(cat(1,2),3) 역시 위와 동일한 과정을 거쳐, 123 으로 완전히 확장됩니다. 하지만, cat(xcat(1,2),3) 의 경우에는, 첫번째 인자인 xcat(1,2) 가 ## 의 영향을 직접 받아서 확장되지 않기 때문에, cat(cat(1,2),3) 와 마찬가지로 유효하지 않은 토큰을 생성하게 됩니다 (undefined behavior).

유사하게, ABSDIFF(ABSDIFF(a,b),c) **는 예상되는대로 완전하게 확장된 결과를 생성**합니다.

▷ A12.4 파일첨가 (File Inclusion)

```
# include <filename>
```

위와 같은 형태의 제어라인 (control line) 은 파일명 (filename) 이 가리키는 파일의 전체내용으로 치환됩니다. **파일명에는 > 문자나 개행문자 (newline) 가 들어갈 수 없으며, ", ', \, /* 문자가 파일명에 포함되어 있는 경우의 영향은 정의되어 있지 않습니다 (undefined). 파일을 찾는 위치는 컴파일러가 따로 정합니다** (implementation-dependent).

```
# include "filename"
```

유사하게, 위와 같은 제어라인 (control line) 은, 주어진 파일을 우선 소스파일이 있는 곳에서 찾고 (**정확한 사항은 컴파일러에 따라 달라집니다** – implementation-dependent), 그곳에서 찾지 못하면 첫번째 형태에서처럼 컴파일러가 정한 위치에서 찾습니다. **파일명에 ', \, /* 를 사용할 때의 영향은 정의되어 있지 않지만 (undefined), > 는 허락**됩니다.

```
# include token-sequence
```

마지막으로, 앞에서 살펴본 두 형태와는 다른, 위와 같은 지시자는 **주어진 토큰열 (token-sequence) 을 일반적인 텍스트 (normal text) 로 확장**합니다; **결과는 <...> 나 "..." 중 하나여야 하며, 앞에서 살펴본 두 형태의 #include 문과 동일하게 다루어집니다.**

#include 파일은 중첩될 수 있습니다.

▷ A12.5 조건부 컴파일 (Conditional Compilation)

프로그램 소스 중 일부분은 다음과 같은 문법에 따라 선택적으로 컴파일될 수 있습니다.

```
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

else-part:
        else-line  text

else-line:
        # else
```

각 지시자 (if-line, elif-line, else-line, #endif) 는 한 줄에 하나만 나타납니다. #if 문과 그에 뒤따르는 #elif 문에 있는, 상수수식 (constant expression) 은 0 값이 아닌 수식이 발견될 때까지 차례대로 평가됩니다; 0 값을 갖는 라인 뒤에 따라오는 텍스트 (text) 는 버려집니다. 0 이 아닌 값을 갖는 라인 뒤에 따라오는 텍스트는 일반적인 텍스트로 다루어집니다. 여기서 "텍스트 (text)" 란 전처리기 라인을 포함한 (단, 조건부 컴파일 구조의 일부분은 안됩니다) 어떠한 것이라도 상관없습니다; 심지어는 비어있어도 (empty) 괜찮습니다. 일단 값의 평가가 성공한 #if 나 #elif 라인이 발견되면 그에 따르는 텍스트가 수행되고, 따라오는 (성공하지 못한) #elif 와 #else 라인은 그것들의 텍스트와 함께 버려집니다. 모든 수식이 0 이고 (모든 라인이 성공하지 못하고), #else 가 있으면, #else 뒤의 텍스트가 수행됩니다. 조건부 컴파일에서 활동하지 않는 (inactive) 부분의 텍스트는, 조건부 컴파일의 중첩여부를 검사할 때를 제외하면 무시됩니다.

#if 와 #elif 에 있는 상수수식은 일반적인 매크로 치환의 영향을 받습니다. 또한,

```
defined  identifier
```

혹은

```
defined  ( identifier )
```

와 같은 형태를 갖는 수식은, 매크로 검색 (scan) 을 하기 전에, 전처리기에 의해 그 명칭 (identifier) 이 정의되어 있으면 1L 로, 정의되어 있지 않으면 0L 로 치환됩니다. 매크로 확장 뒤에 남아있는 모든 명칭은 0L 로 치환됩니다.

▶ 역자 주

정의되지 않은 명칭을 포함하는 #if 문을 사용한다면, 그 명칭은 #if 문이 평가되기 전에 0L 로 치환됩니다. 이는 #if 문이 문법적으로 옳도록 만들어 줍니다. 예를 들어, 다음과 같은 경우가 있다고 해보겠습니다.

```
#if HPUX >= 10
    ... HP-UX 10.0 이나 그 상위 버전일 때 ...
#else
    ... 그 외의 경우 ...
#endif
```

만약 위 프로그램을 HPUX 가 아닌 다른 곳에서 컴파일한다면, HPUX 라는 명칭은 정의되어 있지 않습니다. 만약 매크로 확장 후에 남아있는 명칭을 그냥 지워 버린다면 위의 #if 문은 다음과 같이 되고,

```
#if >= 10
```

이는 문법적으로 틀린 것이 되어 컴파일이 중단됩니다. 하지만, 위에서 이야기한 것처럼 매크로 확장 후에 남아있는 명칭이 0L 로 치환된다면,

```
#if 0 >= 10
```

로 되어, 문법적으로 옳으며, (항상 그런 것은 아니지만) 논리적으로도 옳은 결과를 가져옵니다. - 예를 들어, Linux 에서 컴파일한다면, 위의 #if 문은 Linux 가 HP-UX 10.0 이상의 버전이 아니라고 올바르게 판단 내립니다.

마지막으로, **전처리기에 사용된 정수상수 (integer constant) 는 접미사 L 이 붙어있다고 간주되며, 따라서 모든 산술형이 long 이나 unsigned long 으로 취급됩니다.**

매크로에 나오는 상수수식 (constant expression, A7.19) 에는 제약이 있습니다: **정수 (integral) 여야 하고, sizeof, 캐스트 (cast), 열거상수는 포함할 수 없습니다.**

다음의 제어라인 (control line) 은

```
#ifdef   identifier
#ifndef  identifier
```

각각 아래와 동일합니다.

```
# if   defined   identifier
# if   !   defined   identifier
```

#elif 가 일부 전처리기에 이미 존재했었지만, 초판과 비교하면 새로운 것입니다. defined 전처리기 연산자 역시 새로운 것입니다.

▷ A12.6 라인제어 (Line Control)

C 프로그램을 생성하는 다른 전처리기를 위해, 다음과 같은 라인은

```
# line   constant   "filename"
# line   constant
```

에러진단을 목적으로, 컴파일러로 하여금 다음 소스라인의 라인번호 (line number) 를 주어진 10 진수 정수상수로, 현재 컴파일중인 파일명을 주어진 명칭으로 알도록 만들어 줍니다. 만약 파일명이 생략되면, 현재 컴파일중인 파일명이 그대로 유지됩니다. **#line 에 매크로가 있다면, 매크로가 확장된 후에 해석**됩니다.

▶ 역자 주

다음과 같은 경우,

```
#define   LINE_NUMBER   123
#line   LINE_NUMBER
```

#line 에 있는 LINE_NUMBER 매크로가 123 으로 확장된 후,
컴파일러에게 #line 다음에 나오는 소스라인의 번호가 123 이라고 알려줍니다.

▷ A12.7 에러생성 (Error Generation)

아래와 같은 전처리기 라인은

```
# error   token-sequence opt
```

전처리기로 하여금 주어진 토큰열 (token-sequence) 을 포함한 에러 메시지를 발생하도록 합니다.

▷ A12.8 Pragma

아래와 같은 제어라인 (control line) 은

```
# pragma   token-sequence opt
```

전처리기로 하여금 컴파일러가 정한 행동 (implementation-dependent action) 을 하도록 합니다. **인식되지 않는 pragma 문은 무시**됩니다.

▷ A12.9 널지시자 (Null Directive)

아래와 같은 전처리기 라인은

```
#
```

아무런 효과도 없습니다.

▷ A12.10 기정의 매크로 (Predefined Name)

C 언어에는 미리 정의되어 있는 다양한 명칭들이 특별한 정보를 생성하도록 확장됩니다. **기정의 매크로와 전처리기 수식 연산자 defined 는 재정의 (redefined) 되거나, 정의가 취소 (undefined) 될 수 없습니다.**

__LINE__	현재 소스 라인번호 (line number) 를 담고 있는 10진수 상수
__FILE__	현재 컴파일중인 파일명을 담고 있는 문자열
__DATE__	컴파일될 때의 날짜를 "Mmm dd yyyy" 형태로 담고 있는 문자열 (Mmm 는 숫자가 아닌 영문 3자리로 표시된 월입니다)
__TIME__	컴파일될 때의 시간을 "hh:mm:ss" 형태로 담고 있는 문자열
__STDC__	상수 1. 이 명칭은 ANSI-C 표준을 따르는 컴파일러에서만 정의되어 있습니다.

#error 와 #pragma 는 ANSI 에서 새로운 것입니다. 기정의 매크로 역시 새로운 것이지만, 일부 매크로는 이미 몇몇 컴파일러에서 쓰이고 있었습니다.

▷ A13. 문법 (Grammar)

아래는 이 부록의 앞부분에 나왔던 문법을 요약한 것입니다. 내용은 앞에서 다룬 것과 동일하지만, 순서는 바뀌었습니다.

이 문법에서는, 종료자 기호 (terminal symbol) 인 *integer-constant (정수상수)*, *character-constant (문자상수)*, *floating-constant (부동상수)*, *identifier (명칭)*, *string (문자열)*, *enumeration-constant (열거상수)* 를 정의하지 않았습니다; 타이프 체 (typewriter style) 단어와 기호는 문법적으로 주어진 종료자 (terminal) 입니다. 이 문법은 자동 파서-생성기 (automatic parser-generator) 가 받아들일 수 있는 입력형태로 변환이 가능합니다.

▶ 역자 주

파서-생성기 (parser-generator) 란, 문법구조를 받아들여 그 언어의 파서 (parser) 프로그램을 생성해 주는 것입니다. 프로그래밍 언어를 정교하게 이론화하면, 이러한 작업을 자동화할 수 있습니다. 대표적인 것으로는 YACC 가 있습니다.

파서 (parser) 란, 컴파일러나 인터프리터에서 원시 프로그램을 읽어 들여, 그 문장의 구조를 알아내는 구문분석 (parsing) 을 행하는 프로그램을 말합니다.

구문분석 (parsing) 이란, 컴파일러나 인터프리터가 프로그램을 이해해 기계어로 번역하는 과정중의 한단계로, 각 문장의 문법적 구성·구문을 분석하는 과정입니다. 정확하게 이야기하면, 원시 프로그램의 토큰열 (token sequence) 을 받아들여 문법에 맞게 파스 트리 (parse tree) 로 구성하는 것을 말합니다. 구문분석의 종류에는 다음과 같은 것들이 있습니다:

하향식 파싱 (top-down parsing)
상향식 파싱 (bottom-up parsing)
재귀 하강식 파싱 (recursive descendent parsing)
예측형 파싱 (predictive parsing)
픽쳐 파싱 (picture parsing)
연산자 우선순위 파싱 (operator precedence parsing)
LR 파싱 (LR parsing)

문법으로 제시된 것에 대체 가능함 (alternative) 을 의미하는 문법적 표시 (syntactic marking) 를 사용해주고, "one of" 구조를 확장하고, (파서-생성기의 규칙에 따라) *opt* 기호가 붙은 것을 한번은 기호와 함께, 한번은 기호 없이 복사해 써줄 필요가 있습니다.

▶ 역자 주

예를 들어 설명하면, 다음과 같이 바꿔줘야 한다는 것입니다.
(아래에서 기호 | 는 'or' 을 의미합니다)

(원래) *struct-or-union:* one of
struct union

(수정후) *struct-or-union:*
struct | union

(원래) *declarator:*
pointer opt *direct-declarator*

(수정후) *declarator:*
pointer direct-declarator
direct-declarator

또한, *typedef-name: identifier* 로 제시된 것을 없애고, *typedef-name* 을 종료자 기호 (terminal symbol) 로 만들면, 이 문법은 YACC 파서-생성기 (YACC parser-generator) 가 받아들일 수 있게 됩니다.

▶ **역자 주**

YACC 는, "Yet Another Compiler-Compiler" 의 약자로, 유닉스 운영체제에서 돌아가는 파서-생성기의 이름입니다. 원하는 구문 (syntax) 을 백커스 정규형 (BNF) 으로 입력하면, 그에 해당하는 파서 (parser) 를 C 언어로 만들어 줍니다.

백커스 정규형 (Backus Normal Form 혹은 Backus-Nour Form) 은, 백커스 (P.Backus) 와 나우어 (P.Nour) 가 중심이 된 위원회에서 ALGOL 60 언어의 문법구조를 정리하기 위해 만든 형식입니다. 1963 년 출판된 ALGOL 60 보고서에 처음으로 사용되었으며, 비종료자 기호로부터 종료자 기호와 비종료자 기호로 이루어진 기호열 (symbol sequence) 을 도출하는 생성규칙 (production) 으로 언어의 문법을 정리하도록 되어 있습니다. 이는 언어의 문법 구조만을 설명해 주는 것이며, 각각의 의미를 알려주지는 않습니다.

이 문법에는, if-else 모호함에 의해 발생하는 충돌 (conflict) 이 하나 존재합니다.

▶ **역자 주**

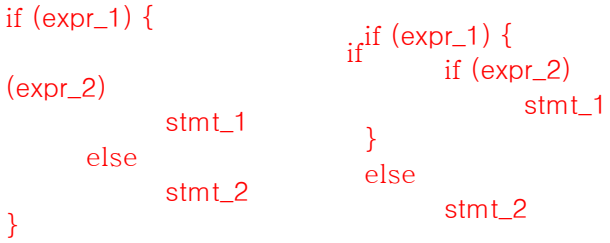
여기서 말하는 충돌 (conflict) 이란, 선택 가능성이 (한 개만 존재해야 하는데) 여러개 존재하는 것을 의미합니다. 즉, 이곳에서 제시된 if 문의 문법을 살펴보면 다음과 같습니다. (원래 있는 switch 문은 생략했습니다)

Selection-statment:
if (*expression*) *statement*
if (*expression*) else *statement*

위와 같이 문법이 주어진 상태에서, 다음과 같은 문장은

if (expr_1) if (expr_2) stmt_1 else stmt_2

각각 아래와 같이 두 가지 의미로 해석될 수 있습니다.



이러한 충돌을 해결하려면, 두개의 비종료자 기호 (non-terminal symbol) 를 추가해야 합니다.

아래는 문법 요약입니다. 한 줄에 나와야 하는 것이 부득이한 사정으로 두 줄로 나누어진 경우, 라인 연결의 의미를 갖는 \ 문자를 사용해서 표시했습니다.

translation-unit:
external-declaration
translation-unit *external-declaration*

external-declaration:
function-definition
declaration

function-definition:
declaration-specifiers$_{opt}$ *declarator* \
declaration-list$_{opt}$ *compound-statement*

declaration:
declaration-specifiers *init-declarator-list*$_{opt}$;

declaration-list:
declaration
declaration-list *declaration*

declaration-specifiers:
storage-class-specifier *declaration-specifiers*$_{opt}$
type-specifier *declaration-specifiers*$_{opt}$
type-qualifier *declaration-specifiers*$_{opt}$

storage-class-specifier: one of
auto register static extern typedef

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

type-qualifier: one of
const volatile

struct-or-union-specifier:

struct-or-union *identifier*$_{opt}$ { *struct-declaration-list* }
struct-or-union *identifier*

struct-or-union: one of
struct union

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

init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*

init-declarator:
declarator
declarator = *initializer*

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*

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

direct-declarator:
identifier

(*declarator*)
direct-declarator [*constant-expression*$_{opt}$]
direct-declarator (*parameter-type-list*)
direct-declarator (*identifier-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-specifier *declarator*
declaration-specifier *abstract-declarator*$_{opt}$

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

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

initializer-list:
initializer
initializer-list , *initializer*

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*$_{opt}$)

typedef-name:
identifier

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:
$expression_{opt}$;

compound-statement:
{ $declaration\text{-}list_{opt}$ $statement\text{-}list_{opt}$ }

statement-list:
statement
statement-list *statement*

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

iteration-statement:
while (*expression*) *statement*
do *statement* while (*expression*) ;
for ($expression_{opt}$; $expression_{opt}$; $exrepssion_{opt}$) *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
= *= /= %= += -= <<= >>= &= ^= |=

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

constant-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*

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

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

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

postfix-expression:
primary-expression
postfix-expression [*expression*]
postfix-expression ($argument\text{-}expression\text{-}list_{opt}$)
postfix-expression . *identifier*
postfix-expression -> *identifier*
postfix-expression ++
postfix-expression --

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

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

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

아래에 주어진 전처리기 (preprocessor) 에 대한 문법은 제어라인 (control line) 의 구조를 요약한 것이며, 구문분석 (parsing) 에 사용하기에는 적합하지 않습니다. 여기에는, *텍스트 (text)* 라는 기호 (symbol) 가 포함되어 있으며, 이 *텍스트 (text)* 는 일반 프로그램 텍스트 (ordinary program text), 비조건부 컴파일 제어라인 (non-conditional preprocessor control line), 완전한 조건부 컴파일 구조 (complete preprocessor conditional construction) 를 의미합니다. 여기서도 역시 부득이한 사정으로 두 줄로 나눠진 경우에는 \ 를 사용해 표시하였습니다.

control-line:
define *identifier* *token-sequence*
define *identifier*($identifier_{opt}$, ... \
, $identifier_{opt}$) *token-sequence*
undef *identifier*
include <*filename*>
include "*identifier*"
include *token-sequence*
line *constant* "*filename*"
line *constant*
error $token\text{-}sequence_{opt}$
pragma $token\text{-}sequence_{opt}$
#
preprocessor-conditional

preprocessor-conditional:
if-line *text* *elif-parts* $else\text{-}part_{opt}$ # endif

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

elif-parts:
elif-line *text*
$elif\text{-}parts_{opt}$

elif-line:
elif *constant-expression*

else-part:
else-line *text*

else-line:
else