Chapter 5 Language specifications

This chapter describes the C language specifications. Since it is premised that this chapter is intended for users who are familiar with K&R, novice users must read appropriate guides first, and then read this chapter.
The language specifications, which depend on the processor, are described in the next chapter.

5.1 Form of program statement

The same conventions as used in the C reference manual of K&R are applied throughout this chapter. Additionally, if “opt” is put, that statement can be omitted.
For example;

Expression:
    Name
    Name (Expression list$_{opt}$)

Name or Name (Expression list) is put at a position where the end symbol of “Expression” appears. Additionally, “Expression list” can be omitted.

5.2 C language tokens

This section describes tokens, basic elements that make up the C language. Tokens are classified into six groups, reserved word, name, constant, character string, operator, and delimiter. A space, tab code, line feed, and comment can also be put in the C language source, but they do not function as a delimiter and are not read.

5.2.1 Reserved words

Table 5.1 shows words that the compiler reserves for special purpose. These words cannot be used for other purpose.

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

Table 5.1: Reserved words

Four reserved words, far, near, nonrec, and recursive, are newly added to this compiler. Words, far and near, of these reserved words are used only for LSI C-86.

5.2.2 Names

A name begins with an alphabetic character (alphabet and under score) and then is followed by alphanumeric characters.

Upper and lower case compiler internal names (name not transferred to the linker) are distinct. At least up to 31 characters are recognized as an effective name.
For external names, distinction between upper and lower case characters, and effective length may vary depending on the linker. lld, standard linker for LSI C-86, distinguishes between upper and lower case characters, and recognizes up to 255 characters.

5.2.3 Constants

There are four types of constants, integer constant, floating-point constant, string constant, and character constant.

5.2.3.1 Integer constant

Integer constants have the following formats.
Integer constant:

Decimal integer suffix_{opt}
Octal integer suffix_{opt}
Hexadecimal integer suffix_{opt}

Decimal number:
Decimal number starting with that other than 0
Octal number:
Octal number starting with 0
Hexadecimal number:
Hexadecimal number starting with 0x or 0X
Integer suffix:
u, U, l, L

Integer suffixes specify types of constants. Two types of suffixes are available, u (or U) specifies an unsigned constant and l (or L) specifies a long constant. The type of the integer constant is the first type that can store values in relevant lists.

No suffix	int, unsigned int, long int, unsigned long int
u or U is put.	unsigned int, unsigned long int
l or L is put.	long int, unsigned long int
u or U and l or L are put.	unsigned long int

In the ANSI standard, it is specified that a decimal constant exceeding the range expressed by int becomes long. However, it becomes unsigned int in LSI C.

5.2.3.2 Floating-point constant
Floating-point constants have the following formats.
Floating-point constant:
Mantissa part Exponent part$_{opt}$ Floating point suffix$_{opt}$
Decimal number Exponent part$_{opt}$ Floating-point suffix$_{opt}$

Mantissa part:
Decimal number$_{opt}$. Decimal number
Decimal number

Exponent part:
e sign$_{opt}$ Decimal number
E sign$_{opt}$ Decimal number

Sign:
- or +

Floating-point suffix:
f, F, l, or L

Floating-point number and integer are distinct based on whether the decimal point or exponent part is provided.
A number that begins with the decimal point is also accepted. At this time, however, the decimal part cannot be omitted. A floating point number is normally a type of double. Adding f or F to its end may become a floating type while adding l or L may become a long double type.

However, the current LSI C handles all floating-point constants as long double data type. Therefore, if the following program is run, the message, “NG”, may appear.

```
void f()
{
        double x = 1.1;
        if (x == 1.1)
                printf (“OK!\n”);
        else
                printf (“NG!\n”);
}
```

In the conditional judgement of the if statement, double type variable x is compared with a long double type constant of 1.1. To extend x to a long double type, 0 is added to the lower bits. On the contrary, 1.1 becomes a loop decimal number in the internal expression, and therefore == is not established.

At this time, x is declared as long double type or 1.1 is put in a double variable and then compared with x. Always pay special attention to this point.

5.2.3.3 Enumeration constant

Names declared as enumeration type are enumeration constants. For details, see section 5.3, Declarations and data types

5.2.3.4 Character constant

Character constants have the following formats.

Character constant:
‘String of character expression’
String of character expression:
Character expression

String of character expression Character expression
Character expression:
A desired character other than ‘, \, and line feed character.
Escape sequence

Escape sequence:

\’	‘
\”	“
\?	?
\\	\
\ddd	Octal number
\xddd	Hexadecimal number
\a	Bell
\b	Back space
\f	Page feed
\n	Line feed
\r	Return
\t	Horizontal tab
\v	Vertical tab

A character enclosed in ‘ ‘ is an integer expressing relevant character code. Most characters except for ‘, \, and line feed character can be put directly. To put such characters and control codes in a character constant, an escape sequence starting with \ is provided. Use of the escape sequence makes it possible to put various control characters, such as ‘ and \.
In LSI C, it is accepted that a character constant contains two characters. At this time, first character is put in the upper byte and second character is put in the lower byte. Therefore, a value of ‘a’ becomes 0x6162. Using this feature, it is possible to express a 2-byte character code as a character constant.
However, note that LSI C does not support multi-byte character, like L’…’.

5.2.4 Character string

Character strings have the following formats.

Character string:
‘String of character string expression’;
String of character string expression:

Blank
String of character string expression Character string expression

Character string expression:
A desired character other than “, \, and line feed character
Escape sequence

A character string is a row of more than 0 character enclosed in “ “. ‘ can be directly put in a character string. However, to put “, it is necessary to describe \”. Additionally, the same escape sequences as for the character constant are also used.
A character string is a char-type array and ‘\0’ is automatically put at the end. If two character strings appear successively (blank, line feed, and comment can be put between them), the compiler connects them and handles as one character string. Use of this feature makes it possible to write a long character string on several lines or replace a part of the character string with a macro.
For example,

```
printf (“hello, “ “word” “\n”) ;
```

is equivalent to

```
printf (“hello, word \n”);
```

However, note that LSI C does not support multi-byte character string, like L’…’.

5.2.5 Operators

C provides the following operators. For details, see section 5.5, Expressions.

```
[ ] ( ) . ->
++ -- & * + - = ! sizeof
/ % << >> < > <= >= == != ^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=
,
```

5.2.6 Delimiters

C provides the following delimiters.

```
[ ] ( ) { } * , : = ; ...
```

5.2.7 Comments

A comment begins with “/*” and ends with “*/”. Comments cannot be nested. If you do not wish to compile a part of the program including comments, you enclose that part by #if 0 and #endif. In LSI C, however, if you specify -cn option in lcc86, you can write nested comments.

“//” is a comment provided by C++. Character strings between // and line feed become a comment. However, this cannot be used in standard C. If you accept lowering of the portability, you may specify -B option for lcc86, making it possible to use // comments.

5.3 Declarations and data types

C can handle various data types. The section describes types used in C and their declaration rules. The entity of data used in C has two attributes, memory class and type. The memory class determines where the entity is located in the memory, and when the entity is generated and cleared. The type determines which calculation is applied to this entity.

The type is classified into three groups, basic type, derivative type, and void type. The basic type includes character, integer, and floating-point types. The derivative type includes an array, structure, union, enumeration type, pointer, and function. The void type shows “no value” and used as function return type.

The declaration connects names and data entities. In C, it is absolutely necessary to declare all names except for functions and labels before using them. The general declaration format is shown as follows.

Declaration:
  Declaration specifier Initialization declarator list$_{opt}$

Declaration specifier:
  Memory class Declaration specifier$_{opt}$
  Type specifier Declaration specifier$_{opt}$
  Type qualifier Declaration specifier$_{opt}$

5.3.1 Memory class

The following shows the syntax notation for the memory class.

Memory class:
  auto
  static

extern
register
typedef

Among them, typedef is not a memory class, but a keyword to define another name of the type. However, this is included in the memory class from a viewpoint of syntax.
An auto variable is made every time a function or block is called, and then deleted when the execution is completed. A static variable is referred to only in the declared function or block. However, its entity always exists while the program is running. An extern variable always exists while the program is running and can be referred from any place. A register variable is the same as the auto variable. However, since the register variable is assigned to the CPU register (if possible), this may improve the program running speed.
If the memory class is omitted, a variable declared inside the function is a type of auto and that declared outside the function is a type of extern.
The extern variable is further classified into 2 groups. As described above, a variable outside the function become a type of extern if the memory class is omitted. At this time, the data entity is taken and output to the object file. On the other hand, if extern is clarified and the variable is not initialized, it is considered that data exists in other places. Therefore, the variable is not output to the object file.
As a result, if the same variable is referred by two programs which are compiled individually, extern is omitted or initialized in one file and extern is written in another file. If not, the error message, “Symbol XXX is not defined.” or “Symbol XXX is defined more than once.”, may appear when programs are linked.

5.3.2 Type

The following shows the syntax notation for the data type.

Type specifier:
char
short
int
long
signed
unsigned
float
double

void
structure
enumeration
typedef name

It is also possible to express a type by combining several keywords. The following shows allowable combinations. (Keywords in ( ) can be omitted.)

Char type:
signed char
(unsigned) char
Integer type
(signed) short (int)
unsigned short (int)
(signed) int
unsigned (int)
(signed) long (int)
unsigned long (int)
Floating-point type
float
double
long double

Above types between signed char and unsigned char express integer data. Particularly, (unsigned) char and signed char among them are most suitable for expression of character data. Therefore, they are called char type, but actually they belong to the integer type. The integer type is classified into two groups, signed and unsigned types. Whether only char or signed/unsigned char is used may vary depending on the machine. In LSI C-86, it is possible to forcibly change this type to that with sign (-cs option) or without sign (-cu option), in order to keep the compatibility.
If no option is specified, the unsigned type is applied.
The signed integer type includes four data types, signed char, short int, int, and long int. The unsigned integer type also includes four data types, (unsigned) char, unsigned short int, unsigned int, and unsigned long int.
The floating-point type includes three data types, float, double, and long double.
The generic name for a combination of character types and integer types described above is called integer type. When adding the floating-point type to this integer type, it is called arithmetic type. The size and internal expression of each type are described in the following chapter.

5.3.3 void

void type shows that no data type exists. Therefore, normal variables cannot be declared as void. Types and parameters that are returned by the function, and types specified by the pointer are only declared as void.

If the type returned from the function is a type of void, a return statement with a format of “return statement;” cannot be written in this function. Additionally, a void type value returned from the function cannot be referred to in the expression.

If function parameters are declared as void, this means that this function uses no parameters.

It is allowed that the pointer specifies the void type, but such pointer variable cannot get or write the contents in the memory pointed by *.

Additionally, the value cannot be subtracted or added. Only put and comparison operators are used. However, the pointer to void data type can freely perform the substitution and comparison with other desired pointers. (In general, it is prohibited to perform the substitution and comparison between pointers that point different data types.)

Therefore, it is convenient to declare that memory allocation functions return void pointers.

5.3.4 Structure and union

The following shows the syntax notation for the structure.

Structure type:
    struct or union $Name_{opt}$ {Structure declaration list}
    struct or union name
struct or union:
    struct
    union
Structure declaration list:
    Structure declaration
    Structure declaration list Structure declaration
Structure declaration:
    Type list Structure declarator list
Structure declarator list
    Structure declarator
    Structure declarator list, Structure declarator
Structure declarator:
    Declarator
    $Declarator_{opt}$: Constant expression

Structure is a data type that gathers several elements. Union stacks multiple different data types on the same address.
Any data type, such as basic, pointer, and array types, and structure and union (except for function and void) can be used for elements of the structure and union. They can be declared in the same manner as normal variable declaration.
In the structure, : and constant expression can be added next to the decelerator to specify a bit width sharing that data. This is called a bit field. If bit fields appear continuously, they are allocated to a byte or word. This may require a long access time, but reduces the memory capacity used.
The type of the bit field must be any of signed int, int, and unsigned int. If the signed int or int type is specified, the bit field is handled as signed data. If the unsigned int type is specified, it is handled as unsigned data.
It is also accepted that the bit field does not have a decelerator, but have only : and bit width. At this time, an area only for these is allocated.
Additionally, if the bit width is specified as 0, subsequent elements are allocated to the new byte or word boundary.

```
struct x {
    char *a;
    int b;
    int c: 10;
    int : 1;
    int d: 3;
    int: 0;
    int e: 2;
};
```

In the above example, the structure named x is defined and contains pointer a to char data and int elements b, c, d, and e. Elements c and d are allocated to the same word and a gap of one bit exists between them.

5.3.5 Enumeration type

The following shows the syntax notation for the enumeration type.

Enumeration type:
enum Name$_{opt}$ {Enumeration list}
enum Name

Enumeration list:
Enumeration
Enumeration list, Enumeration

Enumeration:
Enumeration constant
Enumeration constant = Constant expression

The enumeration type is newly introduced to put a meaning on a group of constants and make reading of the program more easily.
When the enumeration type is defined, values are assigned to enumeration constants written in the enumeration list. Normally, a value that starts with 0 and is incremented is assigned to the enumeration constants. However, if "= constant expression" is put at the last, that value is assigned. After that, a value that is incremented is assigned to the next enumeration constant.

```
enum news {north, east, west, south}
enum color {black, blue, red, green = 4, skyblue,
            violet = 3, yellow = 6, white};
```

In the first example, the enumeration type data news containing four values, north, east, west, and south, is defined. These values are, north = 0, east = 1, west = 2, south = 3. In the second example, the enumeration type data color containing eight values is defined. These constant values are shown below.

```
black = 0
blue = 1
red = 2
green = 4
skyblue = 5
violet = 3
yellow = 6
white = 7
```

Enumeration constants and variables can be written at any locations where an integer expression can be put. However, only the same type data is substituted for the enumeration type variable.

5.3.6 Type qualifier

The following shows the syntax notation for the type qualifier.

Type qualifier:
const
volatile

const shows that a variable cannot be changed as if it is a constant.
volatile shows that a variable may be changed regardless of the process system (such as interrupt process). volatile is also used to operate registers of a peripheral unit allocated to the memory mapped I/O addresses.

```
extern const volatile int real_time_clock;
```

The above example declares the read-only variable named real_time_clock which may be changed by the hardware.

```
const int *pci;
```

In the above example, pci is a pointer variable. Since const is not put on pci itself, a value can be substituted for pci. However, *pci (data pointed by pci) is specified as const, no values can be substituted for *pci.

5.3.7 Declarator

The following shows the syntax notation for the declarator.

Initialization declarator list:
- Initialization declarator
- Initialization declarator list, Initialization declarator

Initialization declarator:
- Declarator
- Declarator = Initialization

Declarator:
- Pointer declarator
- Direct declarator

Direct declarator:
- Type modifier list$_{opt}$ Name
- (Declarator)
- Direct declarator [Constant expression$_{opt}$]
- Direct declarator (Parameter name list$_{opt}$)
- Direct declarator (Parameter type list$_{opt}$)

Pointer:
- Type modifier list$_{opt}$ * Type qualifier list$_{opt}$

Type qualifier list:
    Type qualifier
    Type qualifier list Type qualifier
Type modifier list:
    Type modifier
    Type modifier list Type modifier
Type modifier:
    far
    near
Parameter name list:
    Name list
Name list:
    Name
    Name list, Name
Parameter type list:
    Parameter list
    Parameter list, ….
Parameter list:
    Parameter declaration
    Parameter list, Parameter declaration
Parameter declaration
    Declaration specifier Declarator
    Declaration specifier Abstract declarator$_{opt}$

A declarator is used to show a data type by combining it with the type specifier. The type specifier directly shows a basic type, structure, union, enumeration type, and void. To show an array, pointer, and function, it is absolutely necessary to use a declarator.

Generally, the declaration has the format shown below.

    T D1

Where, T is a type specifier and D1 is a declarator. If D1 is a name, this means that this name is a type of T. For example,

    int x;

means that variable x is declared as int data type. Declaration having a complicated form of D1 is explained on the following pages.

5.3.8 Pointer declaration

If D1 has the following format,

```
* D
```

this means that D is a pointer to T. For example,

```
int *p;
```

declares that variable p is a pointer pointing int type data.
A combination of the arithmetic type and pointer type is called scalar type.

5.3.9 far and near

The standard C does not have the far and near features. These features are specially designed for C for i8086.
The address area of i8086 is classified into two gropes, narrow area where access can be made efficiently and wide area where access is difficult.
Keywords far and near are used to select either area.

If D1 has the following format,

```
far D
```

this means that D is a type of T which is located in a wide address area. Accordingly,

```
near D
```

means that D is a type of T which is located in a narrow address area

```
int far x;
char far *p;
```

In the above example, x is a type of int and located in a wide address area. p is a pointer to char type data. p itself is located in a narrow address area, but it points a wide address area.

5.3.10 Array declaration

If D1 has the following format,

```
D [N]                (N is a constant expression.)
```

this means that D is a type of array where T is put N times. In the constant expression N at the left most position, [ ] can be omitted. At this time, however, one of the following conditions must be full filled.

- Declared as extern explicitly.
- Already initialized.
- Is a function parameter.

```
char buf [100];
char *lines [100];
extern int x [];
```

The above examples show the following declarations.

"buf is an array where 100 of char type data are put", "lines is an array where 100 of pointers to char data are put", and "x is an int array, but the number of array elements is unknown".
A combination of the array type and structure type is called an aggregate type.

5.3.11 Function declaration

If D1 has the following format,

```
   D (parameter name list)
or
   D (parameter type list)
```

this means that D is a type of function which returns T.
The first format declares function parameter names and can only be used to define functions. The second format is newly added to the ANSI standard and allowed to use for both the function declaration and function definition.
In the function declaration, parameter names can be omitted.
This format is called a prototype of the function and plays a role in transmission of parameter types and the number of parameters to the compiler. If a prototype is detected before using the function, the compiler checks whether the parameter types or the number of parameters for the function are matched with those for the prototype. If any data is unmatched, the error is informed.
However, if the function declaration f() has a blank in ( ), this does not mean that f() has no parameters, but means that there is no information related to this function. To express a function having no parameters, it is necessary to write a prototype like f (void). This is rather strange, but this rule must be used to keep the compatibility with conventional programs.
The parameter type list may finish like “…”. This means that there is no information on the parameter next to “…”. This type of parameter type list is used for a function that receives different type data and number of data every time the function is called, such as printf() (such function is called a variable parameter function.)

At this time, parameters immediately before "…" are checked. After that, however, no error occurs even though any parameter type and any number of parameters are written.

```
int f(void), g();
char *fcp (char *);
void (*pf) (int x, long y);
int pintf (char *, …);
```

The above declarations have the following meanings.

- f is a function that returns int having no parameters.
- g is a function that returns int having unknown parameters.
- fcp is a function, first parameter of which is a pointer to char data and that returns a pointer to char data.
- pf returns no values and is a pointer to a function having int and long parameters.
- printf is a function, first parameter of which is a pointer to char data and that returns int data, from which data is unknown.

Under these declarations, the function call

```
f(1);
```

may cause an error, but

```
g(1);
```

causes no errors.

5.3.12 Type name

The type name is used for the type conversion or the operator sizeof, and expresses the type. The type name has the same format as declaration without name.
The following shows the syntax notation for the type name.

Type name:
  Type specifier 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$_{opt}$)

int
int *
int *[3]
int (*)[3]
int *()
int (*)(void)
int (*const [])(unsigned int, …)

The above examples have the following meanings.

- int
- Pointer to int data
- Array containing three pointers to int data
- Pointer to array containing three int data
- Function that returns a pointer to int data
- Pointer to a function having no parameters that returns int data
- Constant array in which the numbers of unsigned int parameters that return int data and pointers to functions that receive other parameters are not specified.

5.4 Initialization

The following shows the syntax notation for the initialization.

Initialization:
Expression

{Initialization list}
{Initialization list, }
Initialization list:
Initialization
Initialization list, Initialization

A variable declared is set to an initial value by the initialization. The initialization of variables with a memory class of extern or static (static variables) must be done by a constant expression. The initialization of scalar variables with a memory class of auto or reregister must be done only by a single expression. Additionally, this expression can also be enclosed in braces { }. In the expression, desired variables and functions, which are already declared, can be referred in addition to constants. Every time the program step enters a block containing declarations, the calculation is made and substituted.
In the initialization of aggregate type variables with a memory class of auto or register, the initial value is a single expression with the same type or must meet the following conditions. At this time, the initial value must be a constant expression.
If a declared static variable is a char or signed char type array, it is initialized by a character string. Normally, '\0' is put at the last. If a whole size of this array is not allocated, '\0' is not put. (See examples at the last of this section.)
The initialization of union is made for the first element. The initial value must be enclosed by braces { }. Constant expressions are sequentially set to elements until the initialization of the structure and array encounters a sentence starting with { and ending with }.
At this time, structure and union members without name are ignored. If the number of constant expressions is insufficient, remaining elements are set to 0.
If elements are structure or array, the rule described above is applied when the relevant initialization starts with {. If the initialization starts without {, constant expressions, the number of which is the same as that of elements, are set sequentially.

```
int v[10] = {1, 2. 3};
```

The above example has the following meanings.
1, 2, and 3 are set to v[0], v[1], and v[2], respectively. 0 is set to v[3] through v[9].

```
int v[] = {1, 2, 3};
```

The above example has the following meanings.
The size of v is declared as 3. 1, 2, and 3 are set sequentially.

```
int m[2][3] = {
{1, 2, 3},
{4, 5, 6}
};
```

In the above example, values are set like, m[0][0] = 1, m[0][1] = 2, m[0][2] = 3, m[1][0] = 4, and so on. { } symbols which are used inside can be omitted from the above example. So, you can also express as follow:

```
int m[2][3] = {
1, 2, 3, 4, 5, 6
};
```

Additionally,

```
int m[][3] = {
{1},
{2, 3},
{4, 5, 6}
};
```

0 is set to m[0][1], m[0][2], and m[1][2] in the above example.

```
char hello[] = “hello”;
char hallo[5] = “hello”;
```

In the above example, the size of array hello is 6 and ‘\0’ is put at the last. The size of array hallo is 5 and ‘\0’ is not put at the last.

5.5 Expression

This section describes the expression. The following shows the syntax notation for the expression.

Primary expression:
- Name
- Constant
- Character string
- (Expression)

Post-fixed expression:
- Primary expression
- Post-fixed expression [Expression]
- Post-fixed expression (Expression list)
- Post-fixed expression . Name
- Post-fixed expression -> Name

Post-fixed expression ++
Post-fixed expression --

Parameter expression list:
Expression
Parameter expression list, Expression

Unary expression:
Post-fixed expression
++ unary expression
-- unary expression
Unary operator Cast expression
sizeof unary expression
sizeof (type name)

Unary operator:
& * + - ~ !

Cast expression:
Unary expression
(Type name) Cast expression

Multiplication expression:
Cast expression
Multiplication expression * Cast expression
Multiplication expression / Cast expression
Multiplication expression % Cast expression

Addition expression:
Multiplication expression
Addition expression + Multiplication expression
Addition expression - Multiplication expression

Shift expression:
Addition expression
Shift expression << Addition expression
Shift expression >> Addition expression

Relational expression:
Shift expression
Relational expression < Shift expression

Relational expression > Shift expression
Relational expression <= Shift expression
Relational expression >= Shift expression

Equivalent expression:
Relational expression
Equivalent expression == Relational expression
Equivalent expression != Relational expression

AND expression:
Equivalent expression
AND expression & Equivalent expression

Exclusive OR expression:
AND expression
Exclusive OR expression ^ AND expression

Inclusive OR expression:
Exclusive OR expression
Inclusive OR expression | Exclusive OR expression

Logical AND expression:
Inclusive OR expression
Logical AND expression && Inclusive OR expression

Logical OR expression:
Logical AND expression
Logical AND expression || Logical AND expression

Conditional expression:
Logical OR expression
Logical OR expression ? Expression : Conditional expression

Substitution expression:
Conditional expression
Unary expression Substitution operator Substitution expression

Substitution operator:
= *= /= %= += -= <<= >>= &= ^= |=

Expression:
Substitution expression
Expression, Substitution expression

Table 5.2 shows the operator precedence.

Associativity	Operators
Left to right	() [] -> .
Right to left	! ~ ++ -- + - (type) * & sizeof
Left to right	* / %
Left to right	+ -
Left to right	<< >>
Left to right	< > <= >=
Left to right	== !=
Left to right	&
Left to right	^
Left to right	|
Left to right	&&
Left to right	| |
Right to left	?:
Right to left	= += -= *= /= %= <<= >>= &= |= ^=
Left to right	,

Table 5.2 Operator precedence

In the above table, upper operator has higher precedence. Additionally, the associativity means which operator is associated faster when operators having the same rank appear continuously.

```
a + b * c
a & b == c
*a++
a + b - c
a += b *= c
a?b : c?d : e
```

When using ( ), the above examples are re-written as follows.

```
a + (b * c)
a & (b == c)
```

```
*(a++)
(a + b) - c
a += (b *= c)
a?b : (c?d : e)
```

The following describes operators.

Name	Name itself is used as an expression. Its value is a name itself. If the name is declared as an array, this value is converted into a pointer to the first element of the array automatically. Additionally, if the name is a function name, it is converted into a pointer to that function automatically.
Constant	Constant itself becomes an expression. Its value is a constant itself.
Character string	Character string is used as an expression. Its value is a start address of the character string.
(x)	( ) gather expressions and are used to change the associativity of the calculation. The value of the expression (x) is equivalent to the value x.
x[y]	[ ] are used together with arrays and pointers. The value of expression x[y] is y-th value of the array x. x[y] is equivalent to *(x + y).
x.y	. is an operator that takes an element from the structure. The value of expression x.y is that of the element y of the structure x.
x->y	x->y is equivalent to (*x).y.
x (p1, p2, …)	The function x is called with parameters p1, p2, … put and the returned values are put in the expression. The expression x must be a type of function.
*x	*x shows the value of the memory pointed by x. x must be a type of pointer. If the type of x is a pointer to T, the type of *x becomes T.
&x	The value of expression &x is an address of x. If the type of x is T, the type of &x becomes a pointer to T.
-x	-x inverses the sign of x. x must be a type of arithmetic.
!x	When the value of x is 0, this expression is 1, otherwise it is 0.

~x	The value of this expression becomes one's complement of x. ~x and !x may be confused, but return different values.
++x	++x is equivalent to x += 1. That is, x is incremented and new value of x is put in this expression.
--x	--x is equivalent to x -= 1. That is, x is decremented and new value of x is put in this expression.
x++	x++ is equivalent to (x += 1) -1. That is, x is incremented and previous value of x is put in this expression.
x--	x-- is equivalent to (x -= 1) + 1. That is, x is decremented and previous value of x is put in this expression.
(T)x	T is a type name. The value of this expression is that x is converted into the T type. This operator is called a cast operator.
sizeof x	The value of this expression becomes the size (number of bytes) of x. For example, if int x [10]; is declared, the value of sizeof x is (size of int) x 10.
sizeof (T)	T is a type name. The value of this expression becomes the size (number of bytes) of type T. In 16-bit processors, for example, sizeof (int) becomes 2 and sizeof (int [10]) is 20.
x*y	x is multiplied by y. X and y are types of arithmetic.
x/y	x is divided by y. X and y are types of arithmetic. When both x and y are types of integer, the calculation result is also a type of integer. If either x or y is a type of floating-point, the calculation result is a type of floating point. In division of integer type data, if both x and y are positive numbers, the result is cut. If either x or y is a negative value, a different result is obtained depending on the computer.
x%y	As a result of division of x by y, a reminder is calculated. Both x and y are types of integer. x-(x/y)*y is always equivalent to x%y.
x+y	x is added to y. Both x and y are types of arithmetic, or one data is a type of pointer and other data is a type of integer. If x is a pointer to type T and y is a type of integer, the value of x + y becomes that the size of the type T is multiplied by y, and added to x.
x-y	y is subtracted from x. Both x and y are types of arithmetic, or x is a type of pointer and y is a type of integer. If x is a pointer to type T and y is a type of integer, the value of x - y becomes that the size of the type T is multiplied by y, and subtracted from x. Additionally, if both x and y are types of

	pointer to type T, the value of this expression becomes that (x-y) is divided by the size of type T.

x << y	x is shifted left by y bits. Both x and y are types of integer.
x >> y	x is shifted right by y bits. Both x and y are types of integer. If x is signed data, the arithmetic shift is made depending on the type of computer (a copy of sign is put in upper bit). If x is unsigned data, logical shift (0 is always put in upper bit) is made in all computers.
x <= y	If x is smaller than y or equivalent to y, the value of the expression is 1, otherwise it becomes 0. Both x and y are types of arithmetic or pointer to the same type.
x >= y	If x is larger than y or equivalent to y, the value of the expression is 1, otherwise it becomes 0. Both x and y are types of arithmetic or pointer to the same type.
x < y	If x is smaller than y, the value of the expression is 1, otherwise it becomes 0. Both x and y are types of arithmetic or pointer to the same type.
x > y	If x is larger than y, the value of the expression is 1, otherwise it becomes 0. Both x and y are types of arithmetic or pointer to the same type.
x == y	If x is equivalent to y, the value of the expression is 1, otherwise it becomes 0. Both x and y are types of arithmetic or pointer to the same type.
x != y	If x is not equivalent to y, the value of the expression is 1, otherwise it becomes 0. Both x and y are types of arithmetic or pointer to the same type.
x & y	Logical AND of each bit of both x and y is calculated. Both x and y are types of integer.
x ^ y	Exclusive OR of each bit of both x and y is calculated. Both x and y are types of integer.
x | y	Logical OR of each bit of both x and y is calculated. Both x and y are types of integer.
x && y	If both x and y are not 0, the value of the expression is 1, otherwise it becomes 0. However, if the value of x is 0, y is not calculated.
x | | y	If either x or y is not 0, the value of the expression is 1, otherwise it becomes 0. However, if the value of x is 0, y is not calculated.

x ? y : z	If the value of x is not 0, the value of the expression is y. If the value of x is 0, the value of the expression is z. However, if the value of x is not 0, z is not calculated. If the value of x is 0, y is not calculated.
x = y	y is substituted for x. New value of x is substituted for this expression.
x += y etc.	x + y is substituted for x. New value of x is substituted for this expression. Ten operators, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, and |=, can also be defined in the same manner.
x, y	x is calculated, and then y is calculated. The value of y is used as value of this expression.

5.6 Type conversion rules

This section describes rules which data type is applied to the result when calculating a combination of data with several different types.

Operation may vary depending on the type of operator when calculating a combination of different type data. Generally, however, weak operand (operator to be calculated) is converted into a strong type, and the calculation is made. The type of stronger data is applied to the type of result. The following table shows the weakness and strength of the data type.

Strong	long double
	double
	float
	unsigned long int
	signed long int
	unsigned short int
	signed short int
↓	unsigned char
Weak	signed char

int has the same strength as long int or short int. The precedence may vary depending on the type of computer.
In LSI C-86, int has the same strength as short.
The following shows the type conversion rules for different operators.

Unary operators, +, -, ~	Making no type conversion
*, /, %, +, -, &, |, ^	Both operands are compared. The weak type is converted into the strong type. The result is the strong type.

<<, >>	Regardless of the right operand, the calculation is made with the type of the left operand, and the result is also the type of the left operand.
==, !=, < >, >=, <=	Both operands are compared. The weak type is converted into the strong type. The result is a type of int.
&&, ||, !	No type conversion is made. The result is a type of int.
=, +=, -=, etc.	The type of the right operand is converted into that of the left operand, and the substitution is made. The result is the type of the left operand.
?:	Among 2nd and 3rd operands, the weak type is converted into the strong type.
,	No type conversion is made. The result is the type of the right operand.
Function parameters	If the prototype of a function is declared, the type is converted into that declared. If the prototype is not declared, char and short are converted into int, and float is converted into double. However, if cast operators are put in the parameters even though the prototype is not declared, the type is not converted as shown below.

```
int foo ():
bar ()
{
        char c;
        foo (1);        /* transfers as int. */
        foo (c);        /* converts into int, and then transfers it. */
        foo ((char)c);          /* leaves as char, and then transfers it. */
}
```

In the above rules, the conversion means that the result of this expression is not only converted, but also minimal elements of the expression (normally variables) are converted.

```
int a;
char b, c;
```

```
a = b + c;
```

The above program example does not mean that b is added to c as the char type is kept, this result is converted into int, and then substituted for a. This means that b is converted into int, c is converted into int, b is added to c, and the result is substituted for a.

If addition of b and c is made using the char type, the program is changed to

```
a = (char) (b + c).
```

The type conversion rules described above do not meet the definitions of K&R and ANSI in details. The type conversion rules do not affect almost all programs, but some programs compiled by other compilers may not run correctly on this compiler.

5.7 Constant expression

A constant expression is an expression, a value of which is fixed during compiling. A variable and/or function call, a value of which is not clear unless it is executed, cannot be written in the constant expression.

The following constants and operators can be written in the constant expression.

Integer constant, character constant, floating-point constant, enumeration constant	
Unary operator	-, !, ~, (type name), sizeof
Infix operator	+, -, *, /, %, <<, >>, ==, !=, <, >, <=, >=, &, |, ^, &&, ||, ?:
()	

The constant expression appears in the size of array for array declaration, in the initialization of static or extern variable, case label, and #if statement. Depending on the location, usable operators may vary.
In the #if statement, the floating-point constant, enumeration constant, sizeof operator, and cast operator cannot be used.
In the size of array and case label, the floating-point constant cannot be used.
In the initialization, the following can be written in addition to the above.

Variable name, function name	
Unary operator	&, ., ->, [ ]

However, the variable name must have & operator (array name and function name do not need it). Additionally, variables and functions referred to in the initialization must be declared before use.

5.8 Statements

This section describes statements. A statement is put in the main body of the function to control the program. The following shows the syntax notation for the statement.

Statement:

- Compound statement
- Expression ;
- ;
- if (Expression) Statement
- if (Expression) Statement else Statement
- while (Expression) Statement
- do Statement while (Statement)
- for (Expression_{opt}; Expression_{opt}; Expression_{opt}) Statement
- switch (Expression) Statement
- case Constant expression : Statement
- default : Statement
- break ;
- continue ;
- return ;
- return Statement ;
- goto Name ;
- Name : Statement

Compound statement

- {Declaration list Statement list}

Declaration list:

- Blank
- Declaration list Declaration

Statement list:

- Blank
- Statement list Statement

5.8.1 Compound statement

A compound statement gathers several statements into one statement. This means like ( ) that gather several complicated expressions into one expression. Only one statement can be written in if statement and while statement. Therefore, if more than one statement needs to write, { } are used to make a compound statement.
At the start of the compound statement (it is also called a block), variables can be declared. However, after the statement has been written once, the declaration is not allowed.

5.8.2 Expression statement

The following shows the syntax notation for the expression statement.

Expression ;

This statement calculates the value of the expression. However, this value is not stored and is disposed of. Therefore, a substitution or function call is normally put in this expression.

5.8.3 Blank statement

The following shows the syntax notation for the blank statement.

;

This statement does nothing. This statement is used if the main loop of for or while statement does nothing or if a label is put immediately before }.

5.8.4 if statement

if statement has two syntax notations shown in the following.

if (Expression) Statement
if (Expression) $Statement_1$ else $Statement_2$

In the first example, the expression is calculated. If the result of the expression is not 0, the statement is executed. If the result is 0, nothing is done.
In the second example, the expression is calculated. If the result of the expression is not 0, the $statement_1$ is executed. If the result is 0, the $statement_2$ is executed.

When if statements are nested, else corresponds to most inner statement without else. This is called dangling else.

5.8.5 while statement

The following shows the syntax notation for the while statement.

while (Expression) Statement

The while statement executes the statement repeatedly. The expression is calculated first. If the result is not 0, the statement is executed. The statement is executed repeatedly until the value of the expression is 0.

5.8.6 do statement

The following shows the syntax notation for the do statement.

do Statement while (Expression);

The do statement is similar to the while statement. In the do statement, the end of the loop is judged at the end of the repetition. The statement is executed and the expression is calculated. This operation is repeated until the value of the expression is 0.

5.8.7 for statement

The following shows the syntax notation for the for statement.

for ($Expression_{1opt}$; $Expression_{2opt}$; $Expression_{3opt}$) Statement

The for statement is also executed repeatedly. The for statement is expressed as shown in the following using the while statement.

$Expression_1$:
while ($Expression_2$) {
Statement
$Expression_3$;
}

Any of $Expression_1$, $Expression_2$, and $Expression_3$ can be omitted. If $Expression_2$ is omitted, it is determined as 1. However, if continue statement exists in the statement, the program does not jump to {, but jumps to a position before $Expression_3$.

5.8.8 switch statement

The following shows the syntax notation for the switch statement.

switch (Expression) Statement

The switch statement is used to select a statement from multiple statements and execute it. Normally, compound statement is used, and label

```
case Constant expression:
```

put in it. If the case label includes a constant that is the same as the value of the expression, the subsequent statements are executed. If the case label does not include a constant that is the same as the value of the expression, the default label is found and the subsequent statements are executed. In either case, the switch statement is exited when the break statement is encountered.

5.8.9 break statement

The following shows the syntax notation for the break statement.

```
break;
```

The break statement exits the most inner while, do, for, or switch statement.

5.8.10 continue statement

The following shows the syntax notation for the continue statement.

```
continue;
```

The continue statement jumps to the end of the most inner while, do, or for loop. That is, the program jumps to the calculation of the expression in the while or do statement, and to the re-initialization part in the for statement.

5.8.11 return statement

The following shows the syntax notation for the return statement.

```
return;
return Expression;
```

The return statement exits the execution of the function. The first example exists only the function and returns no values. The second example returns the value of the expression as value of the function.

5.8.12 goto statement and label

The following shows the syntax notation for the goto statement.

```
goto Name;
```

The goto statement moves the control to the statement with a label of Name. When a name and : are put in front of the statement, it becomes a label.

5.9 Program structure

This section describes the overview of the C program. A program unit that is processed by the C compiler at once is called a compile unit.

The following shows the syntax notation for the compile unit.

Compile unit:
  External definition
  Compile unit External definition
External definition:
  Function definition
  Declaration
Function definition:
  Function mode$_{opt}$ Declaration specifier$_{opt}$ Declarator Function main body
Function mode:
  recursive
  nonrec
Function main body:
  Declaration list Compound statement

The compile unit includes declarations or function definitions. The declaration has already been described. In the function definition, a function main body is placed after the function declaration.
In LSI C, it is possible to put a function mode in the function definition. This informs to the compiler whether or not that function does the recursive call. A function that uses the recursive call is declared as recursive. A function that does not use the recursive call is declared as nonrec. The compiler for processors that are difficult to execute the recursive program (such as that for i8086) uses this information to generate more efficient codes. If this declaration is omitted, it is considered as recursive.

However, when using the following preprocessor statement, nonrec is declared when omitted.

```
#pragma nonrec
```

In the function definition, declaration specifier can be omitted, different from normal declaration.

For example,

```
main ()
{
}
```

is allowed.

However,

```
x;
f();
```

the above examples do not declare x and f() as int type. An error occurs.

In the function main body, if the function is declared as a prototype, the declaration list is not needed.

- In case of prototype declaration:

```
int f(int a, int b)
{
        return ((a + b)/2);
}
```

- In case of conventional declaration:

```
int f(a, b)
int a;
int b;
{
        return ((a + b)/2);
}
```

5.10 Preprocessor commands

This section describes preprocessor commands. The preprocessor is independent from the C language, and performs including of other files, conditional compilation, and macro replacement.

The preprocessor command can be placed at any position in the program source. If the first non-space character in a line is #, that line is considered as a preprocessor. Spaces and tabs can be put between # symbol and next keyword.

5.10.1 Macro

The preprocessor allocates a character string to the name. Every time the name appears, it is replaced with the defined character string. Such name is called a macro. The following shows the syntax notation for the macro.

```
#define        Name   Row of words
```

A name that has been defined once can be referred to at any position after #define statement.

However, this name cannot be referred to in a comment, character constant, and character string.
Using the macro, a name is put on the constant as shown in the following example.

```
#define MAXLINE 80
char line[MAXLINE];

    if (i >= MAXLINE) {
    putline (line);
    i = 0;
    }
```

In the following syntax,

#define Name (Name list$_{opt}$) Row of words

a macro having parameters can be defined. A temporary parameter in the name list is put in the main body. This is replaced with an actual parameter while developing the macro.

Using the above syntax, it is possible to make a part of the program as a macro, which is not so long to produce a function, but is used frequently.

```
#define        min (x, y)        ((x) <= (y) ? (x) : (y))

   c = min (a, b);
```

If temporary parameters exist in a character string, replacement is not made.

```
#define debug (x)      printf (“value of x is: %d\n”, x)

    debug (alpha);
```

Therefore, the above program is developed as shown in the following.

```
printf (“value of x is: %d\n”, alpha)
```

x in the character string is not changed to alpha, and retained as x. As shown in the above example, to embed a parameter in the character string, * is put in front of that parameter. Then, the preprocessor encloses the parameter by “ when developing the macro.
When changing the above example to the following,

```
#define     debug (x)     printf (“value of “#x” is: %d\n”, x)
```

it is developed as shown below.

```
printf (“value of “”alpha”” is: %d\n”, alpha);
```

Consecutive character strings are jointed as one string and the following result is obtained.

```
printf (“value of alpha is: %d\n”, alpha);
```

Additionally, if symbols ## exist in the macro main body, words before and after the symbols are jointed as one string. This feature is used to compose variable names from the parameters.

```
#define     debug (n)     printf (“value of x” #n” is: %d\n”, x ##n)
    debug (1);
    debug (2);
```

The above example is developed into the following.

```
    printf (“value of x1 is: %d\n”, x1)
    printf (“value of x2 is: %d\n”, x2)
```

To make the macro, which has been defined, invalid, #undef command shown below is used.

```
#undef      Name
```

5.10.2 include

#include statement includes another file into the program source, and compiles it. The following three formats are provided.

```
#include  <File name>
#include  “File name”
#include  Name
```

The first and second examples search for a file with File name and includes it into the program source. A difference between both examples is which directory is searched for.

The first #include statement checks only the directory specified by the I-option in the compiler.
The second #include statement first checks the current directory. If a file is not found, the directory specified by the I-option in the compiler is checked.

Therefore, commonly used files, such as library headers are put in a specific directory and referred to using a format of <File name>. Additionally, files used by only the created program are placed in the same directory as that containing the program, and referred to using a format of “File name”.
In the third example, Name must be a macro name that has been defined. The preprocessor first develops this macro, and returns it to either first or second example for processing.

5.10.3 Conditional compilation

The preprocessor can generate different sources depending on the conditions. The following syntax is used.

```
#ifdef   Name
   or
#ifndef Name
   or
#if      Constant expression
      :
      :
##elif  Constant expression (More than one expression can be put or
expressions are omitted.)
      :
      :
#else   (This can be omitted.)
      :
      :
#endif
```

#ifdef checks whether or not a macro having the name written after #ifdef is defined. If defined, subsequent lines until (first) #endif, #else, or #endif are compiled to generate the source codes.

On the contrary, #ifndef compiles subsequent lines if the macro is not defined.
#if calculates the subsequent constant expression. If this value is not 0, subsequent lines are compiled to generate source codes. A floating-point constant, enumeration constant, sizeof operator, and cast operator cannot be written in this constant expression. For details, see section, 5.7, Constant expression.

In #if, defined (Name) can be allowed. If Name is defined as a macro, the value of this statement is 1, otherwise 0.
Therefore, #ifdef X is equivalent to #if defined (X), and #ifndef X is equivalent to !defined (X).

Additionally, in #if line, the macro replacement is made. However, an error does not occur even though names not defined exist. If this occurs, it is calculated as 0.

If conditions described above are not satisfied, reading of sources is skipped until #elif is encountered. When #elif is encountered, the subsequent constant expression is calculated. If the value is not 0, subsequent lines are compiled to generate the source codes.
If these conditions are not also satisfied, the same process continues again. If #else is encountered, lines to #endif are compiled unconditionally to generate source codes. If #else does not exist, reading of sources is skipped until #endif. In either case, when the conditions are satisfied, and #elif and #else are encountered after generating the source code, reading of sources is skipped until #endif.
These conditional compiling commands can be placed.

5.10.4 Line number control

#line statement informs a line number and file name, source of which is being read currently, to the compiler. The compiler uses this information to display an error message and generate the debug information.

```
#line      number
#line      number character string
```

The first example changes only the line number. The second example changes both the line number and file name. The line number is a number assigned to the next line of this command.

5.10.5 Preprocessor blank statement

A line, in which nothing is placed after #, is a preprocessor blank statement and has the following format.

```
#
```

This statement does nothing.

5.10.6 Special macro

Some macros have been defined by the preprocessor.
__FILE__ is developed to a file name (character string), source of which is currently being read. Additionally, __Line__ is developed to a line number (decimal constant) of the source currently being read.

```
#define my_assert (cond)  if (cond); else \
        printf ("%s %d: assertion failed\n", __FILE, __LINE__)
my_assert (p == NULL);
```

In the above example, if calling my_assert is located in 3142 lines of the file foo.c, it is developed to the following.

if (p == NULL); else printf (“%s %d: assertion failed\n”, “foo.c”, 3142)

Macro __eval__ has only one parameter and a constant expression is put to this parameter. Then, this macro calculates the value of the expression and develops the result in the decimal notation. This macro is normally used for debugging of the preprocessor, and also used in the in-line function.

```
#define     bar (x, v)      printf (“the value of “#x” is“ #v ”\n”)
#define     foo (x)         bar (x, __eval__ (x))
#define     V               5
    foo (V+3);
```

The above example is developed to the following.

```
    printf (“the value of 5+3 is 8\n”)
```