IBM Advanced Interactive Executive for the RT, PS/2, and System/370 C Language Reference Version 1.2.1

Document Number SC23-2058-02

IBM Advanced Interactive Executive
for the RT, PS/2, and System/370

C Language Reference

Version 1.2.1

Document Number SC23-2058-02

Edition Notice

Third Edition (March 1991)

This edition applies to Version 1.2.1 of the IBM Advanced Interactive Executive for the System/370 (AIX/370), Program Number 5713-AFL, to Version 2.2.1 of the IBM Advanced Interactive Executive for RT (AIX RT), Program Number 5601-061, and for Version 1.2.1 of the IBM Advanced Interactive Executive for the Personal System/2, Program Number 5713-AEQ, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation Department 52QA MS 911
Neighborhood Road
Kingston, NY, 12401
U.S.A.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Subtopics

Trademarks and Acknowledgments

Trademarks and Acknowledgments

The following trademarks apply to this book:

Portions of the code and documentation were developed at th Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California under the auspices of the Regents of the University of California.

AIX, Personal System/2, PS/2, RT, RT PC, and RT Personal Computer ar registered trademarks of International Business Machines Corporation.

IBM is a registered trademark of International Business Machine Corporation.

System/370 is a trademark of International Business Machine Corporation.

About This Book

This book describes the C programming language as implemented on the Advanced Interactive Executive (AIX) Operating System. It includes reference information on the lexical and syntactic elements that make up the C programming language and the structure and format of C language programs.

Subtopics

Who Should Read This Book
What You Should Know
How to Use This Book
Related Publications

Who Should Read This Book

This book is written for programmers who want to write application programs in C language that run on the AIX Operating System.

What You Should Know

To get the most out of this book, you should have an intermediate to advanced understanding of the C programming language. You should also have a general understanding of programming concepts and terminology and some experience in writing programs.

How to Use This Book

This book is intended as a companion reference to the *C Language User's Guide*. It is organized according to the general classes of elements that are used to construct programs in the C language. To locate specific topics, use the table of contents or the index.

Subtopics
Highlighting
Syntax Diagrams

Highlighting

This book uses different type styles to distinguish among certain kinds of information. General information is printed in the standard type style (for example, this sentence).

The following type styles indicate other types of information:

Commands, keywords, types, objects, expressions, declarations statements, functions, and parameters appear in **bold** type.

Examples, words, and characters that must be entered literally appea in **monospace** type.

Variables appear in *italics*.

New terms appear in ***bold italic*** type.

Blue type indicates an AIX Family extension to the C language

Syntax Diagrams

The following typographic conventions are used in the syntax diagrams. If you need information on how to read the syntax diagrams, refer to the *AIX Operating System Commands Reference*.

Syntactic categories appear between angle brackets (< >)

Alternative syntactic categories appear on separate lines

Ellipses indicate that a preceding parameter can be repeated, fo example:

```
<object>...
```

Variables that should be replaced by data objects in actual progra statements appear in *italics*.

An optional terminal symbol or non-terminal symbol is indicated by th notation:

```
<object>...
           opt
```

A syntactic definition is indicated by the name of the object bein defined, followed by a colon, followed by the symbols that make up the object. Here is an example of the syntactic definition for a compound-statement:

```
<compound-statement>:
     { <declaration>...    <statement>...   }
                      opt               opt
```

This specification states that a compound-statement is made up of a left brace, followed by one or more optional declarations and one or more optional statements followed by a right brace. Note that this definition provides for an empty compound statement.

Brackets [ ] indicate optional items and subscripts of an array

Braces { } enclose optional elements that can be repeated more tha once.

Related Publications

For additional information, you may want to refer to the following publications:

AIX C Language User's Guide, SC23-2057, describes how to develop, link, and execute C language programs. This book also describes the operating dependencies of C language and shows how to use C language-related software utilities and other program development tools.

AIX Commands Reference, SC23-2292 (Vol. 1) and SC23-2184 (Vol. 2), lists and describes the AIX/370 and AIX PS/2 Operating System commands.

AIX Programming Tools and Interfaces, SC23-2304, describes the programming environment of the AIX Operating System and includes information about operating system tools that are used to develop, compile, and debug programs.

SAA Common Programming Interface C Reference, SC26-4353, describes each component of the common programming interface.

Table of Contents

Figures

Tables

1.0 Chapter 1. Introduction

Subtopics
1.1 CONTENTS
1.2 About This Chapter
1.3 Overview

1.1 CONTENTS

1.2 About This Chapter

This chapter includes the main features of the AIX C language and a summary of terms and concepts in the book.

The topics you will find covered in this reference include:

- Lexical Element
- Declaration
- Expression
- Statement
- Program Structur
- Preprocessor Statements

1.3 Overview

IBM AIX C compilers are high-performance optimizing compilers that produce object code for execution under the AIX Operating System. AIX PS/2 supports two C compilers. The C Language compiler (VSC) can be invoked with the **vs** command and can be used to compile C language source code. For information on this command, see *AIX Operating System Commands Reference*. The other compiler is the Extended C Language compiler (MCC).

C language contains many building blocks that you can use to construct programs. Most of these building blocks fit into one of a few categories.

Lexical Elements

There are six basic classes of lexical elements in C language:

- Identifiers
- Keywords
- Constants
- String constants
- Operators
- Other separators.

Declarations

Declarations specify the way in which the C compilers interpret each identifier. When an identifier is declared, the declaration does not necessarily reserve any storage in memory for that identifier. Some declarations simply define a template, for instance, in **struct** and **union** declarations.

Expressions

An expression is a construct that defines the rules of computation for creating a value by performing operations (specified by operators) on operands (specified by variables, constants, and function references). These newly created values can then be used in assignment statements or can be used (in conditional expressions) to control subsequent program actions.

Statements

The C programming language contains expression statements and control flow statements. The expression statements are used to compute and assign new values to objects at runtime. The control-flow statements determine the order in which the computations are performed.

Functions

A complete C program consists of a collection of external objects. These objects are either functions or variables. A function is the fundamental C method of grouping blocks into manageable units.

Preprocessor Statements

The ***preprocessor*** is a program that prepares C language programs for compilation. The preprocessor, rather than the compiler, interprets preprocessor statements. The **cc** command automatically sends programs through the preprocessor, then sends the output of the preprocessor through the compiler.

Preprocessor statements enable you to:

- Replace identifiers or strings in the current file with specified code
- Embed files within the current file
- Conditionally compile sections of the current file
- Change the line number of the next line of code and change the file name of the current file.

2.0 Chapter 2. Lexical Elements

Subtopics

2.1 CONTENTS

2.2 About This Chapter

This chapter describes the lexical and syntactic elements that make up the C programming language.

2.3 Lexical Elements

There are six basic classes of lexical elements in C language:

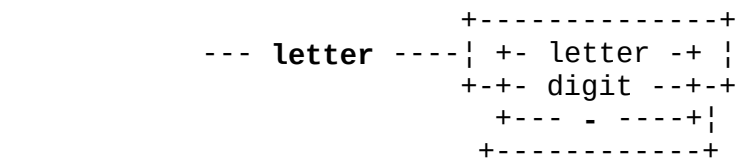
- Identifier
- Keyword
- Constant
- String constant
- Operator
- Other separators

2.4 Identifiers

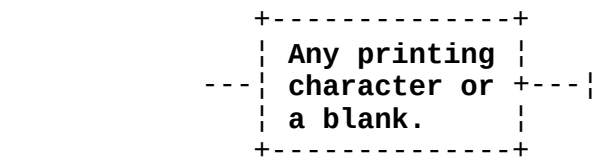
Identifiers, which are also called names, are used to identify variables, functions, and macros. An identifier in C is a sequence of letters and digits. The first character of an identifier *must* be a letter. The underscore character (_) acts as a letter in the context of an identifier.

Note: External identifiers beginning with an underscore are reserved as are all other identifiers beginning with two underscores or an underscore followed by an uppercase character.

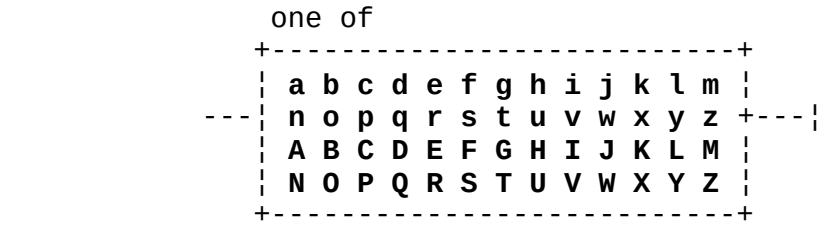
identifier



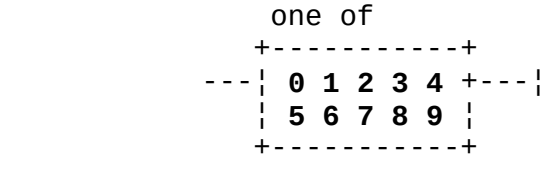
character



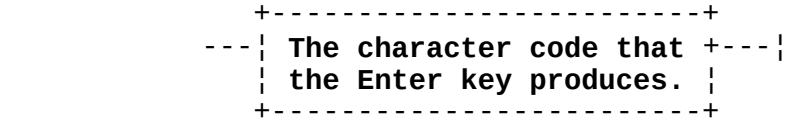
letter



digit



new-line



Note: In the VSC and RT compilers, only the first 64 characters of an identifier are recognized, but the user may write identifiers of

any length, as long as they are unique in the first 64 characters. This is no limit for the identifiers in the AIX/370 and PS/2 MCC compilers.

Uppercase and lowercase letters are considered different in C language identifiers. It is common practice to use uppercase names for macros and constants and lowercase names for variables.

Examples:

```
Albert      Ada_Augusta    Boole_and_Babbage

Tau_Ceti    Z_Transform

UP_to_low   up_to_LOW      up_TO_low
```

Note that the C Compiler considers the last row of names to be quite different, since the placing of the uppercase and lowercase letters is not the same from one to the other.

Invalid Examples:

```
1st_char _is_digit   odd_#$"[_char
```

Note: The RT, AIX/370, and PS/2 MCC C compilers prepend the underscore character (_) to external identifiers.

Subtopics
2.4.1 Keywords and Basic Symbols

2.4.1 Keywords and Basic Symbols

C has a set of basic symbols that the compiler uses for specific purposes in the language. These symbols include selected (reserved) identifiers and special symbols composed of one or more characters. These basic symbols are used as keywords, operators, delimiters, and separators.

Following are two lists of basic symbols. One is a list of C reserved identifiers (keywords) and the other is a list of the special symbols that C uses.

Subtopics
2.4.1.1 Keywords
2.4.1.2 C Special Symbols

2.4.1.1 Keywords

No C keyword may be employed as a user-defined identifier.

The C keywords are *always* lowercase. The C Compiler does not recognize keywords that are typed with uppercase letters in them.

Table 2-1. C Reserved Identifiers (Keywords)

asm (3)	**double**	**if**	**struct**
auto	**else**	**int**	**switch**
break	**entry** (1), (2)	**long**	**typedef**
case	**enum**	**register**	**union**
char	**extern**	**return**	**unsigned**
const (1)	**float** (3)	**short**	**void**
continue	**for**	**signed** (1)	**volatile**
default	**fortran** (2), (3)	**sizeof** (1)	**while**
do	**goto**	**static**	

(1) **const**, **entry**, and **signed** are not supported on the RT.

(2) **entry** and **fortran** are used as reserved words but PS/2 does not associate any meaning with them. System/370 does not support these words.

(3) These are reserved words for the PS/2 VSC compiler and are not supported by the PS/2 MCC compiler.

2.4.1.2 C Special Symbols

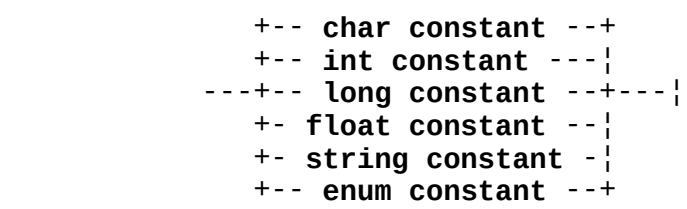
Table 2-2. C Special Symbols	
+	Addition operator; unary plus operator
++	Increment operator
-	Subtraction operator; unary minus operator
--	Decrement operator
*	Multiplication operator; indirection operator
/	Division operator
%	Remainder operator
=	Assignment operator
.	Separates integer from fraction in a **float** number; references a member of a **struct** or **union**
,	Separates items in lists; comma operator
;	Statement and declaration separator
:	Used after **case** and statement labels and bit field declarations
'	Character delimiter
"	String delimiter
==	Relational operator for equality
!=	Relational operator for inequality
<	Relational operator for "less than"
<=	Relational operator for "less than or equal to"
>=	Relational operator for "greater than or equal to"
>	Relational operator for "greater than"
( )	Enclose lists of elements; enclose parts of expressions that are to be considered indivisible factors; casting operator; function call operator
[ ]	Enclose array subscripts
/* */	Comment delimiters
>>	Right-shift operator
<<	Left-shift operator
? :	Ternary (conditional expression) operator

~	Logical ones complement operator
^	Bitwise exclusive OR operator
|	Bitwise inclusive OR operator
||	Logical connective OR
&	Bitwise AND; address operator
&&	Logical connective AND
!	Logical NOT operator
{ }	Enclose block list of initializers; encloses compound statements
->	Pointer to a member of a **struct** or **union**
...	Use to indicate variable argument parameter list in a function declaration
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Remainder and assign
^=	Exclusive OR and assign
|=	Inclusive OR and assign
<<=	Left shift and assign
>>=	Right shift and assign
&=	Bitwise AND and assign

2.5 Constants

The C programming language contains several types of constants. These constants are described in this section.

constant

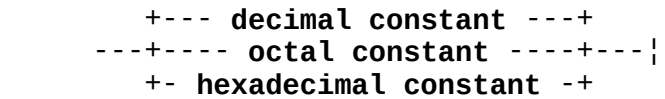


Subtopics
2.5.1 Integer Constants
2.5.2 Floating Constants
2.5.3 Character Constants
2.5.4 String Constants

2.5.1 Integer Constants

An ***integer constant*** is a sequence of digits. Integers are assumed to be in the decimal number base, unless specifically designated as octal or hexadecimal numbers. Integer constants have type **int**. A plus (+) or a minus (-) sign preceding the constant is a unary operator and is not part of the constant. The following diagram lists three types of integer constants:

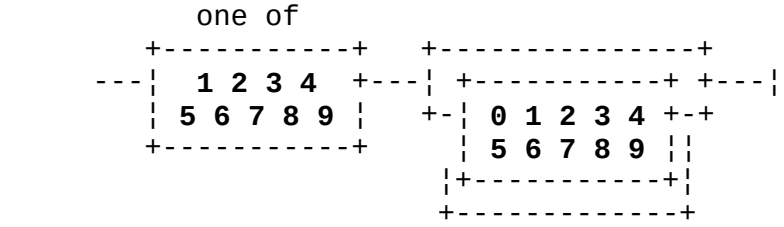
int constant



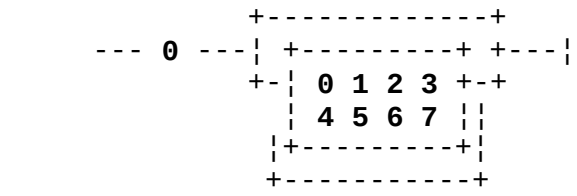
If an integer constant starts with the digit zero (0), it is assumed to be an octal number. The octal digits range from 0 to 7.

If an integer constant starts with the characters 0X or 0x (digit 0 followed by an uppercase or lowercase X), it is taken as a hexadecimal number. The hexadecimal digits include the characters (a-f or A-F), which have the decimal values 10 through 15, respectively.

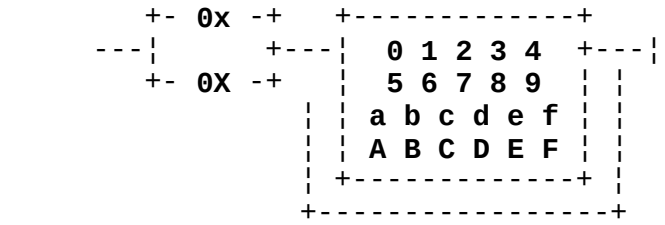
decimal constant



octal constant



hexadecimal constant



A decimal, octal, or hexadecimal number, as defined above, can be denoted as a **long** constant by following it immediately with an uppercase or lowercase L.

Examples:

666 is a decimal number.

+99 -457L are decimal numbers.

0377 is an octal number.

0x3e8 is a hexadecimal number.

Note: The RT C Compiler does not accept the unary plus operator (+).

Invalid Examples:

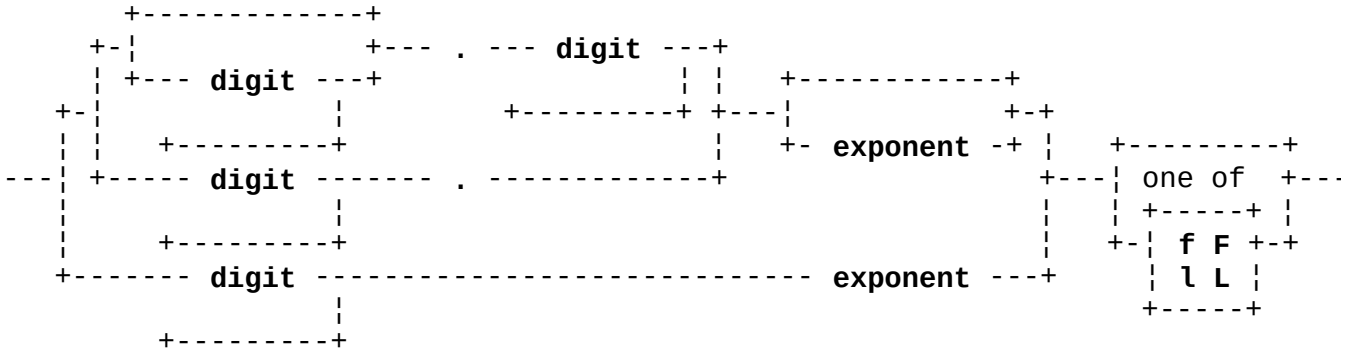
2FC9 is an invalid decimal number.

F034 is an identifier, not a hex number.

2.5.2 Floating Constants

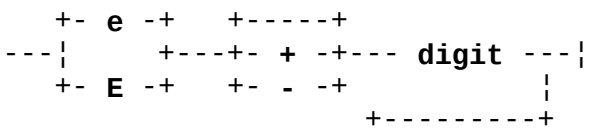
A ***floating constant*** is used to represent real or floating point numbers. Such a constant has an integer part, a decimal point, a fraction part, and an optional exponent. A plus or a minus sign preceding the constant is a unary operator and is not part of the constant. A floating constant has the form:

float constant



The integer and the fraction parts consist of a sequence of decimal digits. Either the integer part or the fraction part (but not both) may be omitted. An exponent has the form:

exponent



The exponent part consists of an uppercase or lowercase E followed by an optional sign and a sequence of decimal digits. Either the exponent part or the decimal point (but not both) may be omitted.

Unsuffixed floating-point constants in a C program are taken as having **double** type. However, a floating point constant can be denoted as having **float** type, by following it immediately with an uppercase or lowercase F. It can be denoted as a **long double** type by following it immediately with an uppercase or lowercase L.

Examples:

```
  0.0         3.14159          5.          1.02e3L
  1.5E10       .618F           3.784e-8    2e0
```

Notes:

1. The L suffix for floating constants is not supported on the RT.

2. The decimal floating-point number that you store as a floating constant may lose some accuracy when stored by the computer because of the nature of decimal-to-binary conversions. Calculations made with the floating constant will reflect any inaccuracy. A floating-point number may also lose accuracy when converted from the internal binary

representation to decimal. Any inaccuracy resulting from the conversion are reflected when you print the number in decimal format.

2.5.3 Character Constants

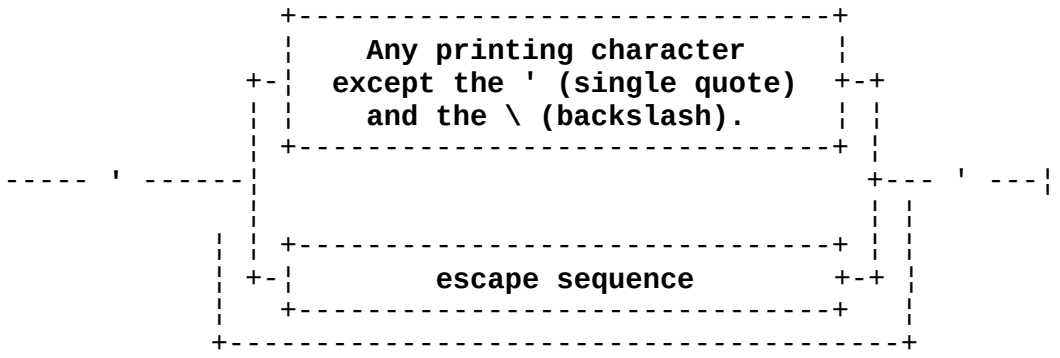
A ***character constant*** is one or more characters or the representation of one or more characters, enclosed in apostrophes ('). The representation of characters (and characters in strings) is based on the ASCII character set. Character constants have type **int**.

Multiple-character character constants are supported. Up to four characters may appear inside apostrophes. The bit pattern of the characters is stored in 4 bytes. The entire character constant is stored in a 32-bit integer formed by shifting the characters one at a time onto the low order bits of the integer. Therefore, if more than four characters are enclosed in apostrophes, only the last four characters are stored. Assignment of such multiple-character character constants to shorter types causes truncation.

```
int  i;
char  c;
short  s;
i = 'abcd';      /* i has the value (bit pattern) of all 4 chars */
c = 'abcd';      /* c will only have the value of 'd'            */
s = 'abcd';      /* s will only have the value of 'cd'           */
```

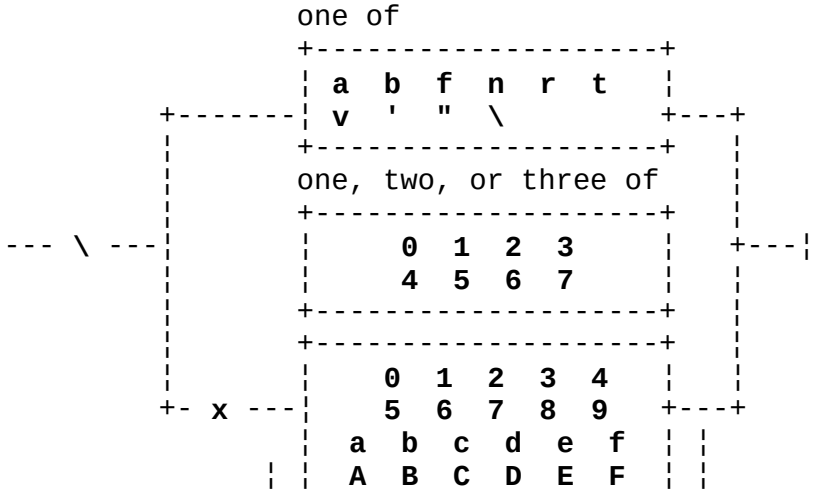
A character constant has the form:

char constant



An escape sequence has the form:

escape sequence



The value of each escape sequence character is always an 8-bit quantity even though several characters may be required to specify it.

Non-printable characters, the apostrophe sign, and certain other characters must be represented by multiple-character escape sequences in character constants according to the following table:

Table 2-3. Character Constants

C Escape Sequence	Meaning
\a	alert (audible) (not on RT)
\b	backspace
\f	form feed
\n	new-line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\'	single quote
\"	double quote
\octal digits	octal character constant
x hexadecimal digits	hex character constant
\?	question mark

A backslash (\ ) followed by one, two, or three octal digits can be used to construct a single character. The numerical value of this octal integer is used as the value of the character. A backslash (\ ) followed by a lowercase **x** that is followed by one or more hexadecimal digits can also be used to construct a single character. The numerical value of this hexadecimal integer is used as the value of the character. A hexadecimal escape sequence is terminated by a non-hexadecimal digit. A backslash (\ ) followed by any other character not defined in the previous table is treated as that character.

Examples:

'w' is the lowercase w.

'\\' is the backslash character itself.

'\002' introduces an STX character in the text.

'\n' is the new-line character.

'\147' is the ASCII code (in octal) for the letter g.

Subtopics
2.5.3.1 Wide Character Constants

2.5.3.1 Wide Character Constants

A wide character constant is the same as a character constant except that it is prefixed by the letter *L*. The elements of the sequence are any members of the source character set. Their mapping to the members of the execution character set is implementation defined.

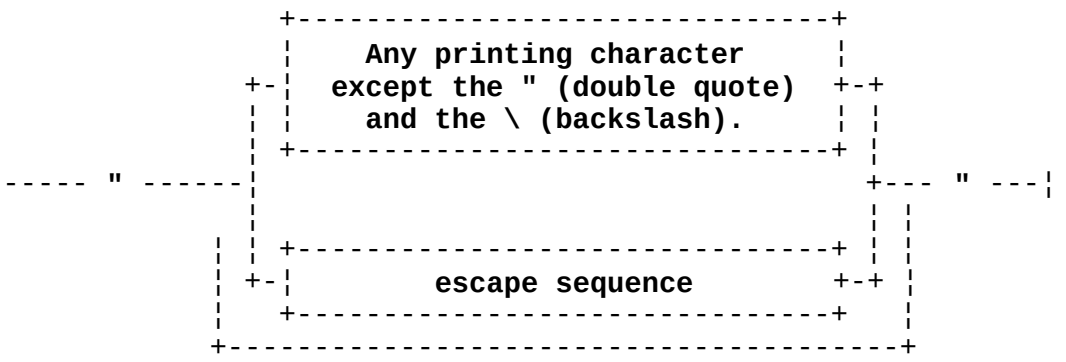
A wide character constant has type **wchar_t**, an integral type defined in the **<stddef.h>** header. The value of a wide character constant containing a single multibyte character that maps into a member of the extended character set is the wide character code corresponding to the multibyte character as defined by the **mbtowc** function with implementation-defined current conversion locale. In the following example, the implementation-defined value that results from the combination of the values 0123 and 4 is specified:

```
wchar_t c=L'\1234';
```

2.5.4 String Constants

A sequence of characters enclosed in double quotation marks (") is called a ***string constant***. A string constant in the C language has the data type of **array of char**. Chapter 3, "Declarations" contains the discussions on storage classes and type definitions. A string constant has the form:

string constant



The initial value of a string constant is the characters inside the double quotation marks. In addition, the compiler places a null byte (value \0) at the end of a string so that a program which scans it can determine its length.

Every string constant is stored at a distinct location, even when two string constants are written identically. In other words, the compiler does not share storage for string constants. A string constant occupies its storage the entire time its containing program is resident.

To enter a double quotation mark character into a string constant, use the \" notation. The same escape sequences can be used as those described in Table 2-3 in topic 2.5.3 for character constants. Escape sequences are replaced before string concatenation as shown in the examples following.

A string constant can be continued over more than one line by placing a (\) *immediately* before the end of the line (the end-of-line following the (\) is then ignored). Another way to continue a string constant is to have two or more consecutive string constants. Adjacent string constants are concatenated to produce a single string constant.

Note: Adjacent string constant concatenation is not supported on the RT.

Examples:

```
"This is a string constant"

"An imbedded \"quote sign"

"A control-D \004 in a string"

"\43"     /* string contains "#" */

"\0043"   /* string contains Control-D, then the character 3 */

"To get a backslash, just type \\"
```

```
  "Here is a string continued \
   over two lines"

  "This is a first string"
  "Which is concatenated with a second string"

  "Bach's \"Jesu, Joy of Man's Desiring\" "

  "A string has the form:  "

  "The following tests proved positive:\n"

  ""

  "Last Name       First Name     MI        Street Address  \
   City            State          Zipcode "
```

For information about operators and a table listing operators, see "Operators in Expressions" in topic 4.5.

Subtopics
2.5.4.1 Wide String Constants

2.5.4.1 Wide String Constants

A wide string constant is the same as a string constant except that it is prefixed by the letter *L*. For wide string literals, array elements have type **wchar_t**.

The multibyte character sequences specified by any sequence of adjacent character string literal tokens, or adjacent wide string tokens are concatenated into a single multibyte sequence. If a character string literal token is adjacent to a wide sring literal token, the behavior is undefined.

Wide string literals are initialized with the sequence of wide characters corresponding to the multibyte character sequence. For example:

```
wchar_t *wstring=L"abc";
```

This is initialized to implementation-defined values that result from the values of a, b, and c.

2.6 Other Separators

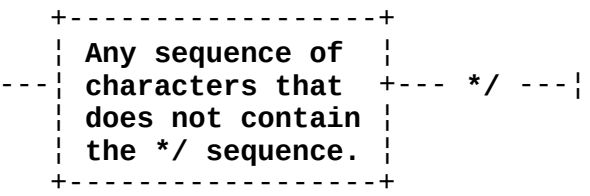
Spaces (also called blanks), tab characters, new-line (end-of-line) characters, and comments are collectively called ***whitespace***, and are ignored except as separations between tokens.

Subtopics
2.6.1 Comments

2.6.1 Comments

Comments are delimited by the characters /* and */. Comments do not nest, but they can span multiple lines. Comments have the form:

comment



Example:

```
/* A comment extended
   over two lines */
```

3.0 Chapter 3. Declarations

Subtopics

3.1 CONTENTS

3.2 About This Chapter

This chapter describes how variables and C objects are declared.

3.3 Declarations

Declarations specify the way in which the C compilers interprets each identifier. When an identifier is declared, the declaration does not necessarily reserve any storage in memory for that identifier. Some declarations simply define a template, for instance, in **struct** and **union** declarations.

3.4 Objects and Lvalues

An ***object*** is a region of storage that can be manipulated in some way. The common examples of objects are simple variables and structured variables. Functions are not considered objects.

An ***lvalue*** is an expression that refers to an object. The basic and most obvious **lvalue** expression is an identifier. There are specific operators that generate **lvalues**. For example, if the expression:

```
E
```

is an expression whose type is pointer to object, then the construct:

```
*E
```

is an **lvalue** expression that refers to the object to which the expression points.

The term **lvalue** derives from the assignment expression:

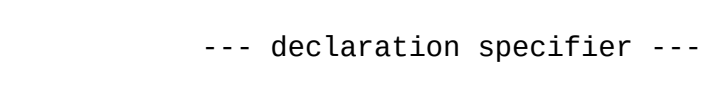
```
E1 = E2
```

in that the operand to the left of the assignment operator must be an **lvalue** expression. In Chapter 4, "Expressions," the discussion of each operator states whether the operator expects its operands to be **lvalues**, and whether the operator generates an **lvalue**.

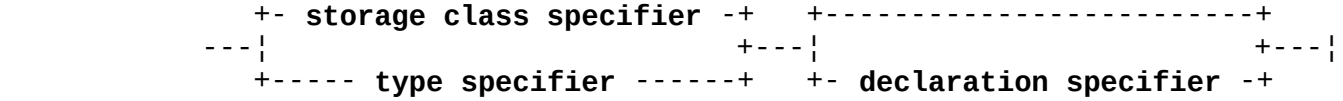
3.5 Declarations

The basic form of a declaration in C is:

declaration



declaration specifier

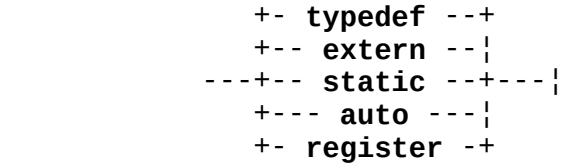


Subtopics
3.5.1 Storage Class Specifiers
3.5.2 Type Specifiers
3.5.3 Type Qualifiers
3.5.4 Declarators
3.5.5 Meaning of Declarators
3.5.6 Arrays
3.5.7 Pointers
3.5.8 Enum
3.5.9 Void
3.5.10 Complex Declarators
3.5.11 typedef -- Declaring Type Name Synonyms

3.5.1 Storage Class Specifiers

The ***storage class*** of an identifier defines how the compiler reserves storage for objects of that type.

storage class specifier



The storage classes defined are:

auto Variables are "automatic" variables and are considered local to each invocation of a block. The variable is accessible to the current block and any nested blocks provided that the inner blocks do not declare a variable having the same identifier. Space is allocated for the variable on entry to the block and is discarded on exit from that block. The variable must be defined within a block or declared as a parameter to a function. Initialization occurs when the system allocates storage for the variable. Using an *auto* variable saves space, as its storage is freed upon exit from the block in which it is defined.

register Variables are essentially synonymous with *auto* variables. They obey all the same rules. The *register* designation can be considered as "advice" to the compiler that this variable will be used heavily and that the compiler should attempt to allocate this variable in a machine register for faster access. Code with *register* variables is usually smaller as well as faster. Note that you cannot take the address of a *register* variable.

static Variables are local to a compilation file. Variables local to a block retain their values across different (in time) executions of the block. If the program reenters a block containing a static variable, that variable has retained the value it had when the block was last exited.

As is described in Chapter 6, "Functions," the static storage class alters the visibility rules for external objects in a compilation unit. The static variables declared inside a block can only be referenced by that block or any nested blocks. A static variable declaration is also a definition of the variable, and allocates storage for it. A static function definition specifies that the function may only be accessed inside its compilation unit. You can use an external static variable or function any place following the definition in the source file that contains the definition. Also, a static function may be referenced before it is defined, provided an **extern** or **static** declaration precedes the reference. Initialization of static variables occurs at the start of program execution.

extern Variables are global to an entire program and retain their values throughout the execution of a program. The *extern* declarations are thus a way to reference variables and functions in different compilation units. Variables and functions declared with the *extern* storage class refer to actual definitions which are later

in the same compilation file, or in a different compilation file.

typedef *typedef* is not a storage class, but obeys the syntax of a storage class keyword. A declaration whose storage class is *typedef* does not reserve any storage. A *typedef* defines an identifier that can later be used as if it were a type keyword.

Only one storage class specifier may be given in a declaration. If the storage class specifier is omitted from a declaration, the following default rules apply:

Inside a function, a missing storage class specifier is assumed to b *auto*.

Outside a function (that is, at the global level of the compilatio unit), a missing storage class specifier for a function declaration or definition is assumed to be *extern*.

The following information applies to the use of the *extern* storage class specifier.

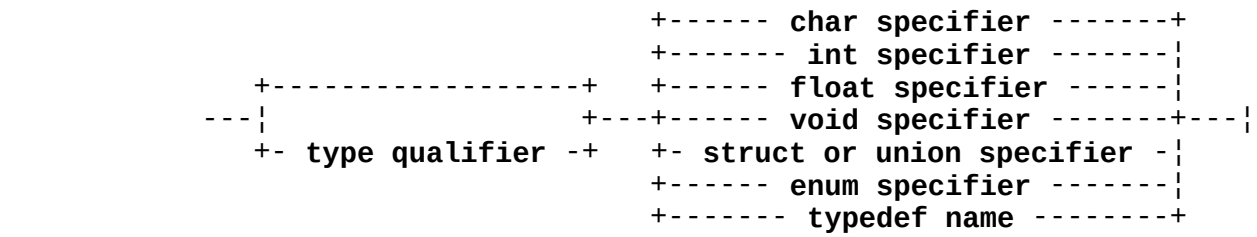
External variable definitions are indicated by a declaration without a storage class specifier. An external definition can appear only outside a function. External function definitions are indicated by a function declaration followed by a compound statement, with no storage class specifier or with the storage class specifier *extern*. An external variable definition allocates storage for the specified variable. An external variable or function definition or *extern* declaration also makes the described variable or function usable by the succeeding part of the current source file. If you want to use an external variable or function prior to its definition or in a file other than the file in which it is defined, you must explicitly declare the variable or function. This declaration does not replace the definition. The declaration just helps to describe the variable that is externally defined.

An *extern* declaration can be distinguished from an external definition by the presence of the keyword *extern*. If the keyword *extern* is present, it is a declaration. Otherwise, it is a definition and a declaration. Only one external definition for an identifier may be present in a C program.

3.5.2 Type Specifiers

The ***type specifier*** is what assigns a specific data type to an identifier.

type specifier

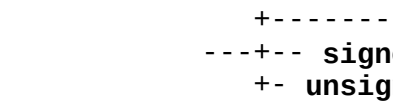


The different type specifiers are:

char Declares the object to be of type **char** (8-bit). Such an object holds a single character from the ASCII character set. This data type defines the set of 256 values of the ASCII character set. Their numeric values are 0..255. The unadorned type **char** is treated as an unsigned quantity.

wchar_t Declares the object to be of type wchar_t (32 bits). Such an object holds a single multibyte character. This character can represent an ASCII character as well as MBCS characters such as Kanji. This datatype defines a set of thousands of characters of the multibyte character set.

char specifier



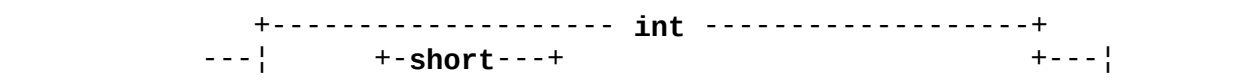
signed Declares that the object is signed and therefore the normal rules for signed arithmetic (sign change, sign extension, and propagation) do apply. The **signed** type may be used as a modifier to **char**, **short**, **int**, and **long** with the resulting ranges being -128..127, -32768..32767, -2147483648..2147483647, and -2147483648..2147483647, respectively. The type **signed** by itself stands for **signed int**.

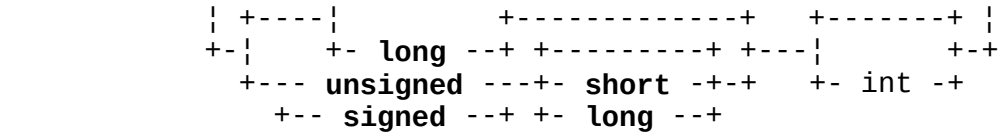
Note: The **signed** type is not supported on the RT.

unsigned Declares that the object is unsigned and therefore the normal rules for signed arithmetic (sign change, sign extension and propagation) do not apply. The **unsigned** type may be used as a modifier to **char**, **short**, **int**, and **long** with the resulting ranges being 0 . . 255, 0 . . 65535, 0 . . 4294967295, and 0 . . 4294967295, respectively.

Note: The type **unsigned** by itself stands for **unsigned int.**

int specifier





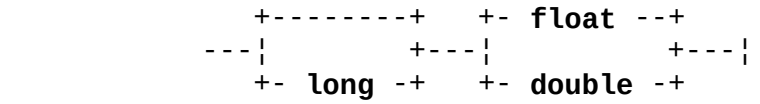
long Declares the object as a **long** (32-bit) integer object. In the AIX Operating System, the length of a **long** specifier is the same as that of an **int**.

short Declares the object as a **short** (16-bit) integer object, which represents an implementation-defined subset of the integers. It is equivalent to a range of integers between the values -32768 and 32767. Thus **short** variables are also normally signed.

int Declares the object as a standard size (32-bit) integer object. It is equivalent to a range of integers between the values -2147483648 and 2147483647.

float Declares the object as a floating-point object. Objects of type **float** are 32-bit quantities, having an 8-bit biased exponent and a 24-bit signed mantissa. The range of **float** numbers is approximately -3.4E38..+3.4E38, with a precision of approximately seven decimal places. Refer to *AIX C Language User's Guide* for more details.

float specifier



double Declares the object as a double-precision floating-point object. The size of the double-precision object is 64 bits, having an 11-bit exponent and a 53-bit signed mantissa. The range of **double** numbers is approximately -1.00E308 .. +1.00E308, with a precision of approximately 16 decimal places. Refer to *C Language User's Guide* for more details.

long double Declares the object as a double-precision floating-point object. **long double** is treated the same as **double**.

struct or **union** Declares the object as a structure or a **union**. The details of structures and **unions** are discussed in "Structures and Unions" in topic 3.5.7.1.

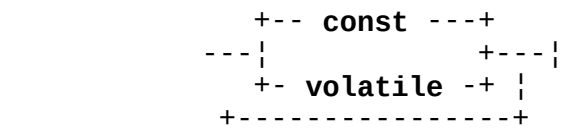
enum Declares the object to be one of an enumerated type. The details of **enum**s are discussed under "Enum" in topic 3.5.8.

void Declares the object as one having no value, such as a function called only as a procedure, that is, having no return value.

typedef-name	Declares the object as one of whatever type is synonymous with the **typedef-name** previously defined in a **typedef** specification.

3.5.3 Type Qualifiers

type qualifier



volatile The **volatile** attribute declares an object as modifiable in ways unknown to the implementation or having other unknown side effects. Any reads or writes to the object will not be removed by the optimizing feature of the compiler. Function return types may not have the **volatile** attribute. Examples of the **volatile** attribute are:

```
volatile int clock;     /* Declares clock as a volatile int. */
int * volatile psoup;   /* Declares psoup as a volatile pointer */
                        /* to an object of type int. */
volatile double * pnut; /* Declares pnut as a pointer to a double */
                        /* having the volatile attribute. */
```

const The **const** attribute declares an object as being unmodifiable. The object may only be assigned a value through initialization when it is defined. Function return types may not have the **const** attribute. Examples of the **const** attribute are:

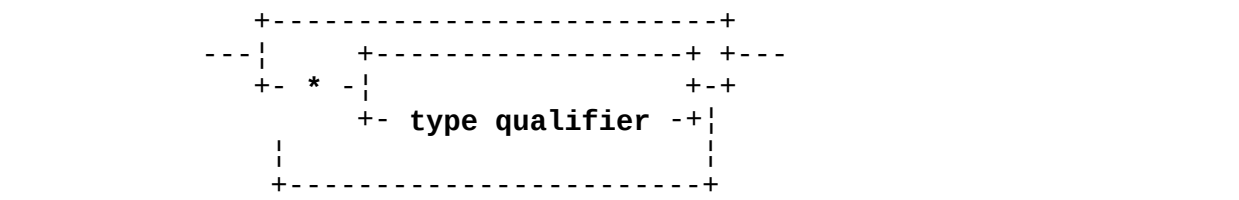
```
extern const volatile int clock; /* Declares clock as being volatile */
                                         /* and unmodifiable. */
     const int x = 4;                  /* Declares x as a const int. */
     int * const ptr;                  /* Declares ptr as a const pointer to
/* an int */
                  .
                  .
                  .
     x = 5;          /* ERROR: x is a const int */
     ptr++;          /* ERROR: ptr is a const pointer */
     (*ptr)++;       /* OK: increments the integer at which ptr points */
```

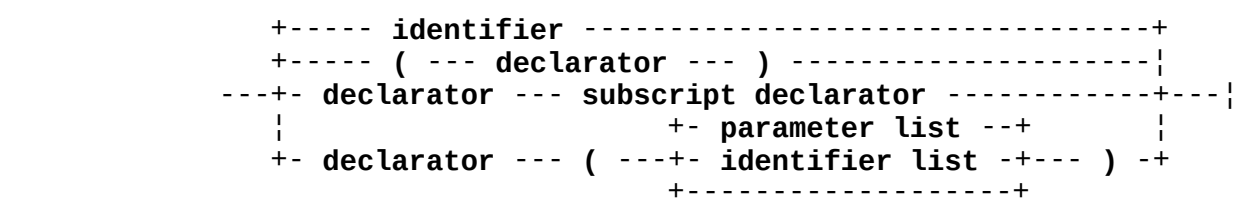
Note: The **const** attribute is not supported on the RT.

3.5.4 Declarators
In the definition of a declaration list, a sequence of declarators, separated by commas and with an optional initializer, is specified.

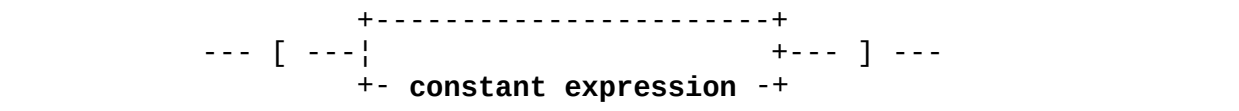
Variable declarations consist of a type specification followed by a list of identifiers that represent variables of that type. Declarators have the form:

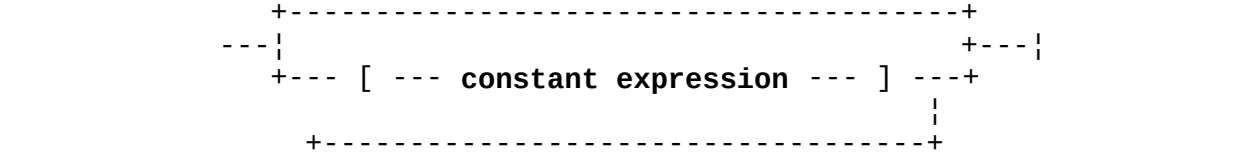
declarator



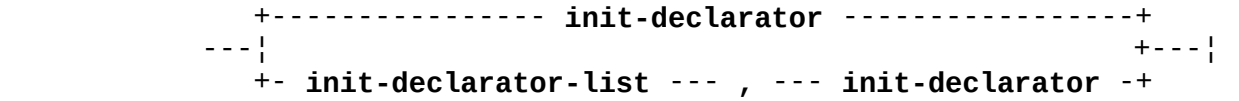


subscript declarator

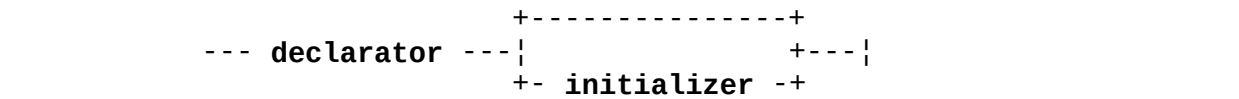




init-declarator-list



init-declarator



3.5.5 Meaning of Declarators

Each declarator is taken as a statement to the compiler. When a declarator appears in an expression, it yields an object of the storage class and type indicated by the declarator. Each declarator contains exactly one identifier, and it is this identifier that is actually declared.

A plain identifier (unqualified in any way) appearing in a declarator has a type indicated by the type specifier that starts the declaration.

A declarator can appear in parentheses ( ). Such a declarator is identical to the plain declarator as mentioned above. However, the binding of more complex declarators can be altered by parentheses.

If **Type** is a type specifier, and **Object** is a declarator, the declaration:

```
Type  Object
```

indicates that **Object** is declarator of type **Type**.

For example:

```
int  count;
```

declares **count** as an object of type **int**.

The declaration:

```
Type  *Object
```

declares **Object** as a pointer to an object of type **Type**.

For example:

```
int  *count;
```

declares that **count** is a pointer to an object of type **int**.

The declaration:

```
Type  Object()
```

declares **Object** as a function returning a value of type **Type**.

For example:

```
float  fact();
```

declares that **fact** is a function that returns a value of type **float**.

3.5.6 Arrays
Either of the declarations:

```
Type Object [constant-expression]
```

Type Object [] declares that **Object** is an array of type **Type**. In the first instance, the constant expression is an expression whose value the compiler can determine at compile time and whose type is **int**. A multi-dimensional array is specified when more than one "array of" specifications appear:

```
Type Object [constant-expression] [constant-expression] ...
```

The constant expression must be of integral type. For a simple array, the size may remain unspecified. For a multi-dimensional array, only the size of the first dimension may remain unspecified. This is used in cases where the array is an external or formal parameter array, and the actual storage for the array is allocated somewhere else in the program.

The other case where the first constant expression may be omitted is when the array declaration is followed by a list of initializers. When this happens, the compiler calculates the size of the array from the number of initial elements that are actually supplied with the declaration.

The index of the first element is zero. An array can be constructed from any of the basic types, from pointers, from structures, from unions, or from another array, in which a multi-dimensional array is constructed.

The first subscript of each dimension is always zero. The following example defines a two-dimensional array that contains six elements of the type **int**:

```
int roster [3][2];
```

In multi-dimensional arrays, when referencing elements in order of increasing storage location, the last subscript varies the fastest. Thus, the array roster contains the elements:

```
roster [0][0]
roster [0][1]
roster [1][0]
roster [1][1]
roster [2][0]
roster [2][1]
```

In single-dimensional arrays, there are two different but equivalent ways of accessing elements of an array. The first is simply to place the array index or indices in brackets after the array name. The second is to use the array name as a pointer and perform pointer arithmetic on it. These two ways are equivalent because an array reference is a pointer to the first element in the array, so that the subscript operation **[ ]** is interpreted such that **A[I]** is equivalent to ***(A+I)**. See "Addition Operators" in topic 4.6.3.3 for a further explanation of what A+I means.

Examples of array variable references:

```
#include <stdio.h>
#include <math.h>

main ()
```

```
{

    /* Declare some array variables */
    int    egress[10] ;
    float  lightly[5]  [4] ;
    char   coal[70] ;
    int    idx, idy;

    /* Now reference those variables */
    for(idx = 0; idx < 10; idx++)
             egress [idx] = 10; /* Set it to a constant */

    for(idx = 0; idx < 5; idx++)
       for(idy = 0; idy < 4; idy++) {
             lightly[idx][idy]= .6;
             printf ("%f", sin (lightly[idx] [idy]));
       }

    for(idx = 0; idx < 70; idx++) {
              coal[idx]= 'b';   /* Write to standard output */
              putchar(coal[idx] );
    }
    putchar('\n');
}
```

In the case of a multi-dimensional array, if **array** is the name of an array of *n* dimensions, an expression containing a reference to **array** is converted to a pointer to an array of *n*-1 dimensions. Thus, in the above context the type of **lightly** is pointer to array[4] of **float**, the type of **lightly[I]** is "pointer to **float**", and **lightly[I][J]** is simply **float**.

3.5.7 Pointers

A **pointer** type holds the address of a data object or function, except that a pointer can never refer to an object having **register** storage class or to a bit-field object. Some common uses for pointers are:

To pass the address of a variable to a function. By referencing th address of a variable, a function can change the contents of that variable.

To access dynamic data structures, such as linked lists, trees, an queues.

To access elements of an array or members of a structure

You can use any type specifier in a pointer declaration or definition. An asterisk (*) precedes the identifier. The following example declares **pcoat** as a pointer to an object having type **double**:

```
double *pcoat;
```

The following example declares **argv** as an array of pointers to characters:

```
extern char *argv[];
```

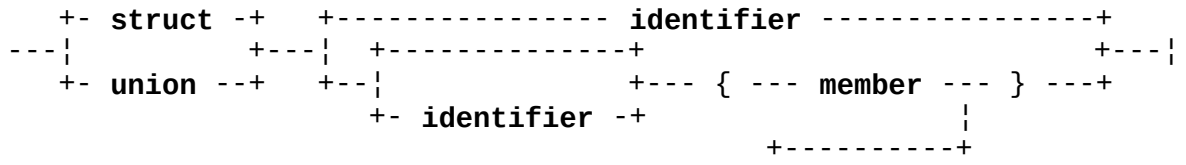
Subtopics

3.5.7.1 Structures and Unions

3.5.7.1 Structures and Unions

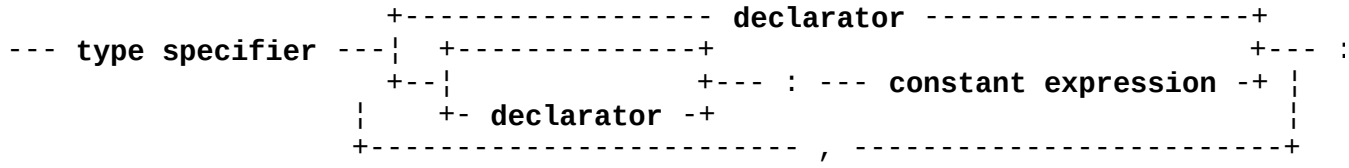
The form of a **struct** or **union** specifier is:

struct or union specifier



A **member** has the form:

member



A ***structure*** is an object that contains a collection of components called ***members***. Each member can be of any type, including another structure, but not, recursively, of the parent **struct** type itself. The names of the members are defined at the time that the structure is defined.

A structure defines a sequence of members, each with a unique name, that are all present simultaneously. They are stored in sequential memory locations.

A ***union*** is similar in concept to a structure, but a union can, at any given time, contain any one of several different members. A union defines several ways of looking at the same area in memory. Other than that, the forms of declaring and referencing structures and unions are the same.

For example, a basic structure declaration looks like this:

```
  struct   office { list of members };
```

This declaration states that the identifier **office** refers to a structure specifier containing the **list of members**.

The definition of the **office** structure above can now be filled in:

```
       struct  office
       {
          int   room_num;
          char  rooms [9];
          int   phone_ext;
       };
```

This form of the **struct** or **union** specifier declares the identifier as a structure or union tag. This means that the declaration of the structure or union does not actually allocate any storage at this time, but instead declares a "template" that may be used later, in subsequent declarations using the second form of the declaration:

```
struct   office off1, *off_ptr;
```

Note that the bracketed list of members is no longer given once the **struct** or **union** specifier has been declared.

The third form of the **struct** or **union** specifier does not specify a structure or **union** tag. For this form, all declarations of objects must follow the **struct** or **union** specifier, as the **struct** or **union** specifier has no name and may not be referenced later:

```
struct   {
          int num;
          char * name;
         } data [20];
```

A **struct** or **union** specifier contains declarators for the members of a structure or a union. A member of a structure can also consist of a number of bits. Such a member is called a ***bit-field***. The bit length of a bit-field is specified by following the member name by a colon (:) and the number of bits.

The members declared within a structure have addresses that increase as their declarations are read from left to right. Each member that is not a bit-field always starts on an addressing boundary that is appropriate for its type. Because of this, there may be anonymous holes in a structure in order to get things lined up on correct addressing boundaries.

The following example defines the structure type **switches** and the structure **kitchen**, which has the type **switches**:

```
struct switches  {
                  unsigned light : 1;
                  unsigned toaster : 1;
                  int count;
                  unsigned ac : 4;
                  unsigned : 4;
                  unsigned clock : 1;
                  unsigned : 0;
                  unsigned flag : 1;
                 }  kitchen;
```

The structure **kitchen** contains six members. The following describes the storage that each member occupies:

Member Name	**Storage Occupied**
light	1 bit
toaster	1 bit
count	the size of **int**
ac	4 bits
	4 bits (unnamed field)
clock	1 bit
	undefined number of bits (unnamed field)
flag	1 bit

The fields **light** and **toaster** each require 1 bit of storage. These members are assigned storage next to each other in the same word. **count** is stored in the next word. **ac** requires 4 bits of storage and is aligned on the next word boundary. The next bit-field has no name. This unnamed field uses 4 bits to separate **ac** and **clock**. **clock** is stored in the following

bit. The unnamed field with a length of 0 (zero) forces **flag** to be on the next word boundary.

All references to structure fields must be fully qualified. Therefore, you cannot reference the first field by **light**. You must reference this field by **kitchen.light**. The following expression sets the **light** field to 1:

```
kitchen.light = 1
```

The following expression sets the **toaster** field to 0 because only the first low-order bit is assigned to the **toaster** field:

```
kitchen.toaster = 2
```

Bit-fields are assigned starting with the low-order bits of the 4-byte field on the PS/2, and starting with the high-order of the 4-byte field on the RT. Bit-fields do not cross from one 4-byte unit to the next, and thus are limited to a maximum size of 32 bits. If a bit-field is too large to fit into the current **long** word, it is placed starting in the next **long** word.

Bit-fields may be declared as having type **int**, **signed int**, or **unsigned int**. However, the compiler always takes the type of the bit-field to be **unsigned int**.

A colon having only a constant expression after it but no preceding declarator specifies an unnamed bit-field that is used to make the alignment of the other members in the structure appear in specific places. The special case of a bit-field width of 0 is used to align the next field on the next word boundary.

A union can be considered as a structure whose members all start at offset zero, and whose size is big enough to contain the largest one of its members. At any time, only one of the members can be stored in the union.

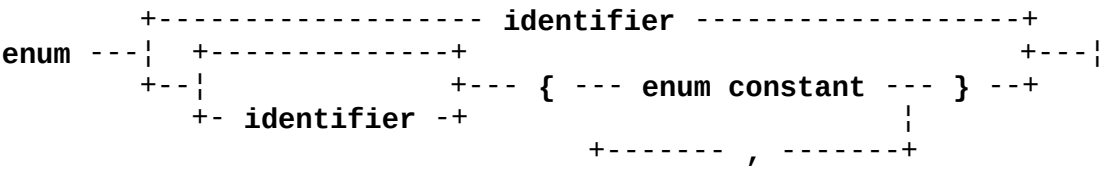
A structure or a union cannot contain an instance of itself as a member. However, it can contain a pointer to itself as a member. In this way, structures or unions that refer to themselves (such as a linked list) are possible.

Within a **struct** or **union** declaration, the names of members must be unique. The same member name may appear in more than one **struct** or **union** declaration without restriction.

3.5.8 Enum

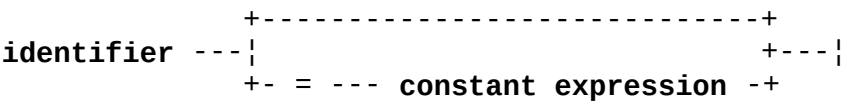
An **enum** is an object much like an object declared to be an **int**, except that an **enum** contains one of an enumerated set of values. These values are constants which are associated with identifiers in the **enum** declaration.

enum specifier



An **enum** constant has the form:

enum constant



The **enum** type specifier declares the **enum** identifier for possible later use as a type specifier. It also declares as constants the identifiers in the list enclosed in
braces { }. These constants take on numerically ascending values, starting with zero, unless an explicit *constant expression* sets the identifier to a specific value. In this case, the ascending sequence of values for succeeding unspecified identifiers is based on the prior constant expression as a base. Consider the following example:

```
enum colors  { red, green, blue = 12, black };
```

Objects declared to be of type **colors** would be able to take on the values 0, 1, 12, and 13, corresponding to the four constants **red, green, blue,** and **black** which are also declared in the **enum** declaration above.

Objects declared as type **enum** can be operated on as if they were type **int**.

It is possible to have duplicated values among the constant identifiers declared in a given **enum** type specifier. Moreover, objects declared using an **enum** type specifier are not restricted by the compiler to the values listed in the **enum** declaration, although program behavior may be unpredictable if an object declared as an **enum** is set to an unexpected value.

3.5.9 Void

The **void** type specifier indicates that the associated identifier has no value. Thus, the nonexistent value of a **void** expression returns no value (that is, procedures) and allows the compiler to detect any use of that function that expects a return value. Variables may not be declared with **void** type, but they may be declared as pointers to **void**.

void specifier

--- **void** ---¦

The following example declares **buf** as a pointer to **void**:

```
  void *buf;
```

This is the proper way of declaring a generic pointer.

Note: This pointer to **void** is not a valid type specifier under the RT C compiler. A **char *b** pointer is needed to access any data pointed to by the generic pointer.

Expressions may be cast to **void** type. A cast expression consists of a left parenthesis (, followed by a type name, followed by a right parenthesis), and an operand expression. The cast causes the operand value to be converted to the type named within the parentheses. Any permissible conversion may be invoked by a cast expression.

This is normally done to ignore the return value of a function. An example of this is:

```
  (void) printf ("hello\n");
```

3.5.10 Complex Declarators

Combining various declarations is also possible. When used in combination, the declaration can be thought of with the right most operation happening first, unless another order is enforced by additional parentheses. Thus the declaration:

```
Type  *Object()
```

declares **Object** as a function that returns a pointer to an object of type **Type**. For example:

```
char  *malloc();
```

declares a function called **malloc** that returns a pointer to an object of type **char**. If parentheses are used to bind the pointer operator directly to the name, a different type is constructed. Thus the declaration:

```
Type  (*Object) ()
```

declares **Object** as a pointer to a function that returns an object of type **Type**. For example:

```
int  (*ItemProc) ();
```

declares that the **ItemProc** is a pointer to a function that returns an object of type **int**.

In general, it is helpful to read such declarations "from the inside out", obeying the grouping suggested by parentheses first, then using a right-to-left ordering for operators at the same level. For example:

```
char  *(*twisted) ();
```

is read as "twisted is a pointer (the innermost *) to a function (indicated by the *( )* signs) returning a pointer (the leftmost *) to a character (**char**)." There are some restrictions on the possibilities indicated by these rules. In particular:

Functions cannot return arrays. Functions can, however, retur pointers to arrays.

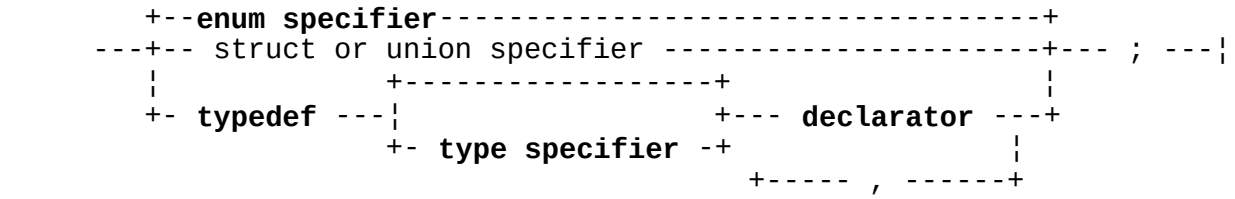
There is no such thing as an array of functions. However, there ca be an array of pointers to functions.

A structure or a union cannot contain a function, but can contain pointer to a function.

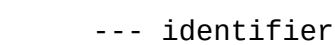
3.5.11 typedef -- Declaring Type Name Synonyms

The **typedef** keyword is used in the context of a storage class, but it has nothing to do with storage classes. A declaration whose storage class is **typedef** actually declares an identifier that can be used later as if it were a type name. A type definition has the form:

type definition



typedef name



Example:

```
typedef unsigned *MICA;

move (horiz, vert)
      MICA horiz, vert;
{
      MICA dist;
}
```

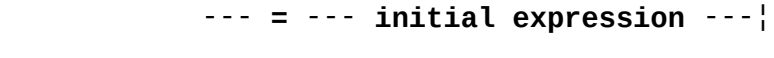
This example shows a **typedef** defining the word **MICA** as a synonym for a pointer to a value of the unsigned data type. The word **MICA** can then be used later, as shown in the declarations in the **move** function, to declare objects of this type.

3.6 Initializing Variables

C Language provides for the initialization of most variables in a convenient and flexible manner. Even most *auto* and *register* variables may have an initial value specified. Arrays can be specified in such a way that the compiler computes their size from the number of initial values supplied.

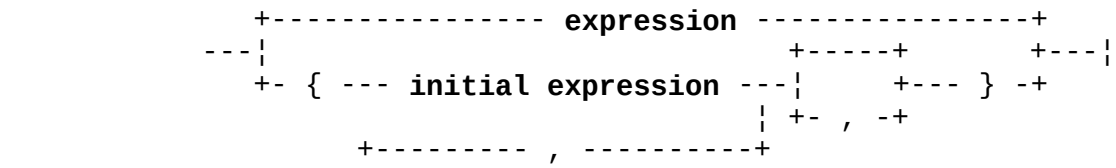
The initial value for a variable is supplied with the declarator for that variable. The initial values consist of an expression, or a list of values nested within braces { }, all preceded by an equal sign (=). An initializer has the form:

initializer



An initial expression has the form:

initial expression



All the expressions in an initializer for a **static** or external variable must be constant expressions or expressions that reduce to the address of a previously declared variable or function, possibly offset by a constant expression. The *auto* and *register* variables can be initialized by arbitrary expressions containing constants and previously declared variables and functions. All *auto* and *register* variables except arrays may be initialized.

Note: Automatic aggregates cannot be initialized on the RT.

An example of initialization of a **struct** is:

```
struct   {
      int  a, b;
      double  d;
       char  C1,C2; }
x = { 1, 2, 3.4, '5' };
```

The members of the **struct** are initialized in order with the values shown. Since there is no value present for **C2** it is initialized with the value zero, which is '\0'.

Uninitialized **static** or *extern* variables have an initial value of zero (0). Uninitialized *auto* and *register* variables are guaranteed to start off with undefined values.

When a scalar type is initialized to a pointer or arithmetic type, the initializer consists of a single expression that may or may not appear within braces { }. The initial value of the object is taken from the expression, and the same conversions are performed as for assignment.

When the variable is a structure, union or an array, the initializer consists of a list of initializers, separated by commas (,) and enclosed within braces { }. The initial values are written in ascending order of subscript or member. If the structure or array contains other structures or arrays, each member of the aggregate is also initialized according to the rule just stated. If there are fewer initializers in the list than there are members of the structure or array, the remaining members are filled with zeros. If there are too many, it is normally an error. When a variable is a union, only the first member can be initialized.

Note: Unions cannot be initialized on the RT.

It is an error to attempt to initialize any array whose storage class is **auto**. It is possible to leave out the internal braces from an initializer. If the internal braces are omitted, the meaning of the initializer list changes. If an initializer starts with a left brace {, the list of initializers that follows the brace represent initial values for the members of the structure or array. Note that it is an error if there are more initial values supplied than there are members of the structure or array.
If the list of initial values does not start with a left brace, only enough elements of the list are used to initialize the members of the structure or array. Any elements left over in the list are then used to initialize the next member of the structure or array of which the current one is a part.

Examples:

```
int temp = 10;
```

This is a simple initialization of the variable **temp**. An array can also be initialized:

```
static float logs[4] = { 2.5, 3.8, 4.9, 10.76 };
```

This is a completely declared array of four elements with its initial values. However, the C Compiler could compute the size on behalf of the programmer if the declaration is stated like this:

```
static float logs[] = { 2.5, 3.8, 4.9, 10.76 };
```

In this case, the declaration omits the size of the array, and the compiler determines the size from the number of initial values supplied. Now a two-dimensional array is declared and initialized:

```
static float  stuff [3] [3]  = {
                        { 1.0, 2.0, 3.0 },
                        { 4.0, 5.0, 6.0 },
                        { 7.0, 8.0, 9.0 } };
```

This is a completely bracketed initialization of the **stuff** array. The first three elements in the list of initial values initialize the first row of the array **stuff** [0]. The next two lines of initial values initialize the rows **stuff** [1] and **stuff** [2]. But, according to the rules stated previously for omitting the braces { }, it is possible to state this initialization more simply:

```
static float stuff [3] [3]  = {
                        1.0, 2.0, 3.0,
                        4.0, 5.0, 6.0, 7.0,
```

```
                8.0, 9.0 };
```

Now the compiler takes the first three elements from the list and assigns them to **stuff [0] [0]** through **stuff [0] [2]**, the next three elements are assigned to the second row of the array, and the last three elements to the third row of the array.

Example:

```
       union data {
                     char charctr;
                     int whole;

                  } input = {'h'};
```

This initializes the first member, **charctr**, of **input** to character **h**.

Subtopics
3.6.1 Initializing Strings

3.6.1 Initializing Strings

Initialization of strings is made convenient by a shorthand notation that simply places successive characters of the array adjacent (no commas in the list) and enclosed in double quotation marks ("). Therefore, the declaration:

```
static char  greet[] = "Hello";
```

is a compact way of stating the more clumsy version of the same declaration:

```
static char  greet[] = { 'H','e','l','l', 'o','\0' };
```

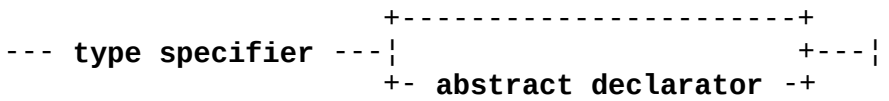
Also note that the declaration:

```
static char  hi[5] = "Hello";
```

can be used to initialize the five character array **hi**. There will be no final **\0** stored. If an array bound is given, and exactly that many characters are supplied, only the supplied characters are stored.

3.7 Type Names

type name



A ***type name*** is the data type name of an object, divorced from the actual identifier that names the object itself.

The type name construct is used in three contexts in the C language:

In type conversions where a cast is require

As an argument to the **sizeof** operator

In function prototype parameter declarations

3.8 Lifetimes of Variables

In an executable C program, each variable has a predetermined lifetime depending on its storage class and location of declaration. These lifetimes are discussed here.

Subtopics

3.8.1 Automatic and Register Variables

The lifetime of an **auto** or **register** variable is that of the function or compound statement in which it is declared. Allocation occurs on each entry to that function or compound statement, and de-allocation occurs on each exit from that function or compound statement.

3.8.2 Static and External Variables

Static or ***external*** variables are those variables declared either outside any function or compound statement (that is, at the level of the compilation unit), or declared inside a function or compound statement, but given the **static** or **extern** storage class. The lifetime of a static or external variable is the lifetime of the program.

3.8.3 Formal Arguments

The lifetime of a formal argument is the lifetime of the function in which that formal argument is declared. The formal argument becomes established upon each entry to the function, and becomes undefined upon exit from the function.

3.9 Implicit Declarations

When declaring identifiers, it is not always necessary to declare the storage class or the type of the identifier. In many cases, the C compiler can infer the storage class from the context in which the identifier is declared.

For example, any function declared at the external level is automatically assumed to have the **extern** storage class. See the notes pertaining to the use of **extern** on page 3-3 for more details.

Inside a function, an identifier is automatically assumed to have the **auto** storage class. Note that this rule does not apply to the function declarations themselves, since functions can never have the **auto** storage class; they are automatically given the **extern** storage class.

Formal arguments of functions are also given the **auto** storage class by default. Similarly, an identifier without a specified type is given the type **int**.

Within a function, a previously undeclared identifier that is followed by a left parenthesis (and appears in the context of an expression is assumed to be an **extern** function returning a value of type **int**.

3.10 Name Spaces

In any C program, identifiers refer to many items. You use identifiers for functions, variables, parameters, union members, and other items. C lets you use the same identifier for more than one class of identifier, as long as you follow the rules outlined in this section.

Name spaces are categories used to group similar types of identifiers.

The C compiler sets up name spaces to distinguish among classes of identifiers. You must assign unique names within each name space to avoid conflict. The same identifier can be used to declare different objects as long as each identifier is unique within its name space. The context of an identifier within a program lets the compiler resolve its class without ambiguity.
Identifiers in the same name space must be distinct from one another.

Within each of the following four name spaces, the identifiers must be unique:

Group 1 - These identifiers must be unique within a single scope

- Function names
- Variable names
- Names of parameters of a function
- Enumeration constants
- *typedef* names.

Group 2 - These identifiers must be unique within a single scope

- Enumeration tags
- Structure tags
- Union tags.

Group 3 - These identifiers must be unique within a single aggregate

- Structure member
- Union member.

Group 4 - These identifiers have function scope and must be uniqu within a function.

- Statement Labels

Structure tags, structure members, and variable names are in three different naming classes; no conflict occurs between the three items named **student** in the following example:

```
struct student                /* structure tag */
{
  char student[20];           /* structure member */
  int class;
  int id;
}student;                     /* structure variable */
```

The compiler interprets each occurrence of **student** by its context in the program. For example, when **student** appears after struct, it is a structure tag. When **student** appears after either of the member selection operators (**.**) or (**->**), the name refers to the structure member. In other contexts, the identifier **student** refers to the structure variable.

3.11 Scope

An object is visible in a block or source file if the data type and the declared name of the object are known within the block or source file.

The region where an object is visible is referred to as the object's ***scope***. The four kinds of scope are: function, file, block, and function *prototype*. The scope of an identifier is determined by the location of the identifier's declaration. An identifier has ***block scope*** if its declaration is located inside a block. An identifier with block scope is visible from the point where it is declared to the closing brace (}) that terminates the block.

The only type of identifier with ***function scope*** is a label name. A label is implicitly declared by its appearance in the program text. A **goto** statement is used to transfer control to the label specified on the **goto** statement. The label is visible to any **goto** statement that appears in the same function as the label.

An identifier has ***file scope*** if its definition appears outside of any block. An identifier with file scope is visible from the point where it is declared to the end of the source file. If there are source files included by means of preprocessing directives, the identifier will be visible to all included files that appear after the definition of the identifier. An identifier has ***function prototype scope*** if its declaration appears within the list of parameters in a function prototype. An identifier with function prototype scope is visible from the point where it is declared to the terminating semicolon (;) of the prototype declaration. Identical identifiers declared in different source files without the storage class **static** can refer to the same object or function. This is called ***external linkage***. In Figure 3-1 the variable **b** is declared in both Source File 1 and Source File 2 as **extern** and refers to the same data object. By default, **c** is also an **extern** variable.

If the first declaration of an identifier contains the keyword **static**, it has ***internal linkage***. Within the source file, each variable with internal linkage refers to the same object or function. In Figure 3-1 all references to the variable **a** in Source File 1 refer to the same data object. The variable **a** in Source File 2 refers to a different data object than **a** in Source File 1.

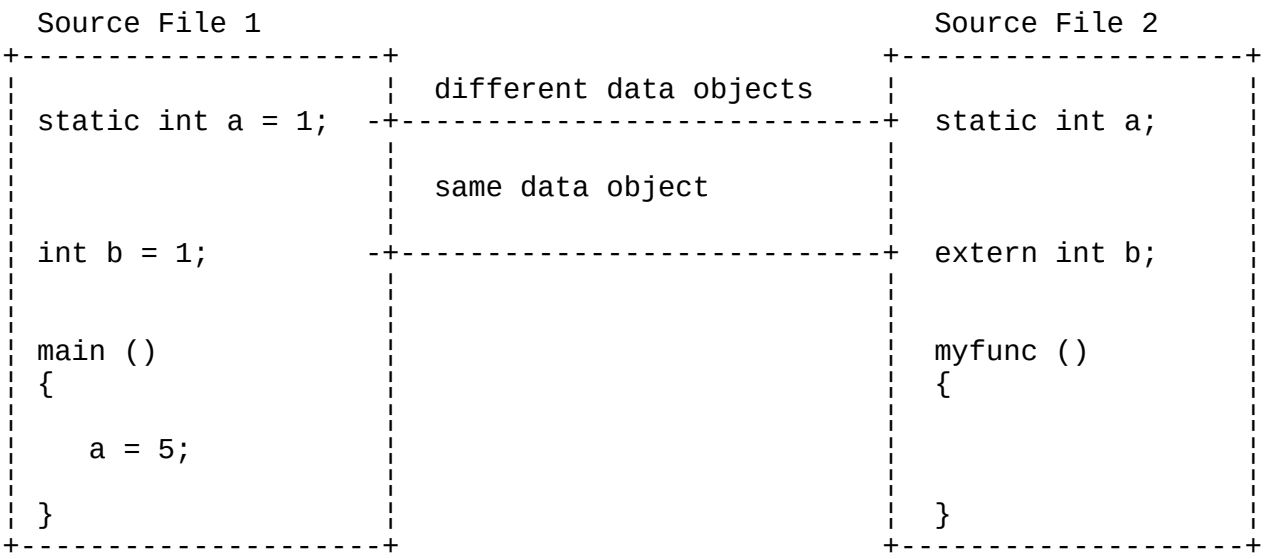


Figure 3-1. Example of External and Internal Linkage

Variables declared or defined at the external level are visible from the point at which you declare or define them to the end of the source file. Variables with a **static** storage class at the external level are visible only within the source file in which you define them. In general, variables declared or defined at the internal level are visible from the point at which you first declare them to the end of that block. These variables are ***local variables***. If a variable declared inside a block has the same name as a variable declared at the external level, the block definition replaces the external-level definition to the end of the block. The compiler restores the visibility of the external-level variable when the current point of execution leaves the block.

You can nest ***block visibility***. This means that a block nested inside a block can contain declarations that redefine variables declared in the outer block. The new definition of the variable applies to the inner block. C restores the original definition when the current instruction returns to the outer block. A variable from the outer block is visible inside inner blocks that do not redefine the variable. Functions with **static** storage class are visible only in the source file in which you define them. All other functions are ***globally visible***. The following program illustrates blocks, nesting, and visibility of variables.

In this example, there are four levels of visibility: the external level and three block levels. Assuming that you have defined the function **printf** elsewhere, the **main** function prints the values 1,2,3,0,3,2,1.

Example:

```
int i = 1;                              /* i defined at external level */

main ()
{

   printf("%d\n", i);                   /* Prints 1 */

   {
      int i = 2, j = 3;                 /* i and j defined at
                                           internal level */
      printf("%d\n%d\n", i, j);         /* Prints 2, 3 */

      {
         int i = 0;                     /* i is redefined */

         printf("%d\n%d\n", i, j);      /* Prints 0, 3 */
      }

      printf("%d\n", i);                /* Prints 2 */

   }

   printf("%d\n", i);                   /* Prints 1 */

}
```

4.0 Chapter 4. Expressions

Subtopics
4.1 CONTENTS
4.2 About This Chapter
4.3 Expressions
4.4 Conversions
4.5 Operators in Expressions
4.6 Summary of Operators

4.1 CONTENTS

4.2 About This Chapter

This chapter describes C expressions which are used to derive new data values.

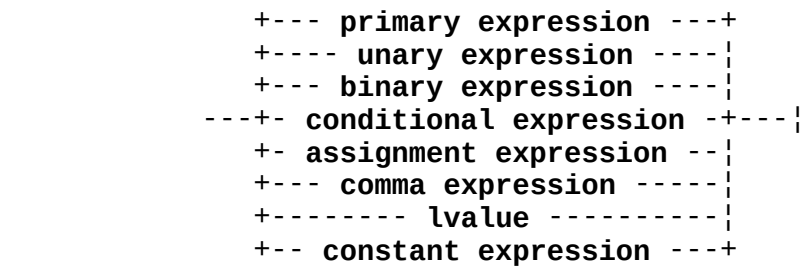
4.3 Expressions

C is an ***expression language***. This means that operations, such as **assignment**, can be part of expressions.

Expressions consist of variables, constants, operators, and functions operating on specified objects to produce new values. New values are obtained by evaluating expressions. These newly-created values can then be used in **assignment** statements or can be used (in conditional expressions) to control subsequent program actions.

An ***expression*** is a construct that defines the rules of computation for creating a value by performing operations (specified by operators) on operands (specified by variables, constants, and function references). Operands of expressions are either declared in the program or are standard C entities. C contains a fixed set of operators that define a mapping from given operand types into result types.

expression



4.4 Conversions

When an expression is being evaluated, it is sometimes necessary for an operand value to be converted to a different type. This section describes the conversion rules in effect during expression evaluation.

Subtopics

4.4.1 Integers, Shorts and Characters

The types **int**, **short** and **char** form a group of compatible integer types of varying precision. By default, the types **int** and **short** are treated as signed, and **char** is treated as unsigned. Each type may also be either signed or unsigned. For the most part, a value of one type in this group may be used anywhere a value of another type may be used. When the precision required differs from that of the expression present, the value of the expression is modified to meet the required precision.

Several simple rules govern this modification process:

When converting from a longer form to a shorter form, the excess mos significant bytes are simply discarded, and the least significant bytes are used as the resulting value. If the original value cannot be represented by the target type, the result is pre-defined, and may not be what the user expected. For example, assigning the **int** value 256 to a **char** variable gives it the value of 0, and assigning the **int** value -1 to an **unsigned short** gives it the value 65535.

Values of the same size are used unaltered. Again unsigned and signe quantities may yield surprising results when the original value cannot be represented in the target type.

When values of the smaller size are changed to values of a large size, sign-extension is performed when the *original* type is signed and zero fill is done when it is unsigned, independent of the target type. The only potentially surprising results from such treatment is that the conversion of negative signed values to unsigned values results in a very large number instead of a negative one. This is unavoidable since there is no representation of negative values in unsigned types.

4.4.2 Float and Double

Floating-point arithmetic is carried out in "single precision" (**float**) unless one of the operands of the operation is double precision.

When a value of type double is converted to type **float,** it is rounded to **float** with associated normal loss in precision.

4.4.3 Floating and Integral

The **int** values are converted to **double** with no loss of precision. The **int** values are converted to **float** with no loss of precision unless the **int** contained more digits than are accurately representable in **float**.

When a **double** (**float**) value is converted to an **int**, the **double (float)** is truncated at the decimal point. It is possible that the resulting **int** will contain an incorrect value if the **double** value was outside the range of integers. The result is always the minimum value representable by **int** when the value is out of range, namely -2147483648. If such an out-of-range floating-point value is assigned to a **short int**, the result will be zero, the value of the lowest 16 bits of -2147483648.

Note: On the RT, if the **double** is greater than 2147483647, the result will be 2147483647.

Conversion of **unsigned int** to **double** (**float**) is the same as first converting the unsigned **int** to an **int** and then converting the resulting **int** value to a **double**. This means that if the most significant bit of the unsigned **int** was a 1, the resulting **double** will be negative.

Note: On the RT, **unsigned int** is converted directly to **double** or **float**.

4.4.4 Pointers and Integers

A value of type **int** or of type **long** may be added to or subtracted from a pointer, and two pointers to objects of the same type may be subtracted. See "Addition Operators" in topic 4.6.3.3 for the rules that apply in these cases.

4.4.5 The Usual Arithmetic Conversions

This section describes what is called the ***usual arithmetic conversions***. Such a term means that many operators convert their operands according to similar rules. The term usual arithmetic conversions will appear in many subsequent discussions in this manual.

If either operand is of type **double**, the other operand is converted to type **double** and the result of the operation is also of type **double**.

Otherwise, if either operand is of type **float**, the other operand is converted to type **float**.

Otherwise, the integral promotions are performed:

Operands of type signed **char** or **short** are converted to type **int**.

Operands of type **char** or unsigned **short** are converted to type **int**.

After these conversions, the following rules apply:

If either operand is of type **unsigned int**, the other operand is converted to type **unsigned int** and the result of the operation is also of type **unsigned int**.

Otherwise, both operands must be of type **int**, and the result of the operation is also of type **int**.

4.5 Operators in Expressions

Operators perform operations on a value or a pair of values to produce a new value. This section describes the different operators that can be applied in expressions. The ordering of the subsections is in the same order as the precedence of the operators discussed in this section. Operators of the highest precedence are described first.

With the exception of the **&** (take the address of) operator, an operation on a variable or field that has an undefined value produces an undefined result. Normally there is no indication when this happens. The following table lists the operators. They are described in detail later in this chapter.

4.6 Summary of Operators

Operator	**Meaning**	**Associativity**
()	Application of function	left to right
[]	Indexing an array	left to right
->	Member of **struct** or **union**	left to right
.	Member of **struct** or **union**	left to right
!	Negation of expression	right to left
~	Bitwise ones complement	right to left
++	Increment	right to left
--	Decrement	right to left
-	Unary minus	right to left
+	Unary plus	right to left
(type)	Type casting	right to left
*	Dereference	right to left
&	Address of	right to left
sizeof	Obtain size of object	right to left
*	Multiply	left to right
/	Divide	left to right
%	Remainder	left to right

+	Add	left to right
-	Subtract	left to right
<<	Left shift	left to right
>>	Right shift	left to right
<	Less than	left to right
<=	Less than or equal to	left to right
>	Greater than	left to right
>=	Greater than or equal to	left to right
==	Equal to	left to right
!=	Not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical connective AND	left to right
||	Logical connective OR	left to right
? :	Ternary conditional	right to left
=	Assignment	right to left
+=	Add and assign	right to left
-=	Subtract and assign	right to left
*=	Multiply and assign	right to

		left
/=	Divide and assign	right to left
%=	Remainder and assign	right to left
<<=	Shift left and assign	right to left
>>=	Shift right and assign	right to left
&=	AND and assign	right to left
^=	Exclusive OR and assign	right to left
|=	Inclusive OR and assign	right to left
,	Expression separator	left to right

Note: Some operators (*, +, -, &) have a higher unary precedence than binary.

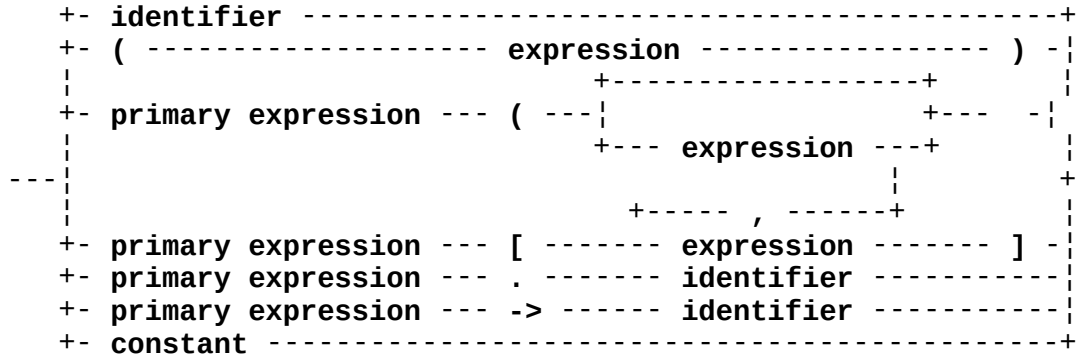
Subtopics
4.6.1 Primary Expressions
4.6.2 Constants
4.6.3 Unary Operators

4.6.1 Primary Expressions

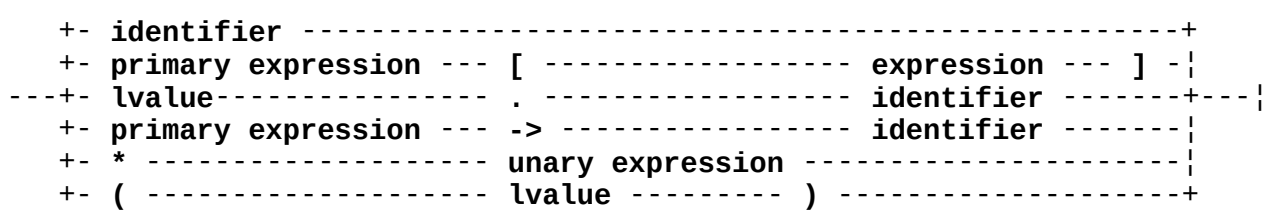
Primary expressions involving the following operators group left to right:

.	the member of operator
->	the member of operator
[ ]	subscripting
( )	function references

primary expression



lvalue



The paragraphs following contain descriptions of the properties of the different objects that may appear in expressions and include examples of the way these elements are referenced, where appropriate.

Subtopics
4.6.1.1 Identifiers
4.6.1.2 Constant Expressions

4.6.1.1 Identifiers

An identifier is a primary expression. The type of an identifier is as specified by its declaration.

A variable of simple scalar type is accessed by its identifier. Since such a simple variable has no structure, its identifier alone is enough to reference it.

Examples of simple variable references:

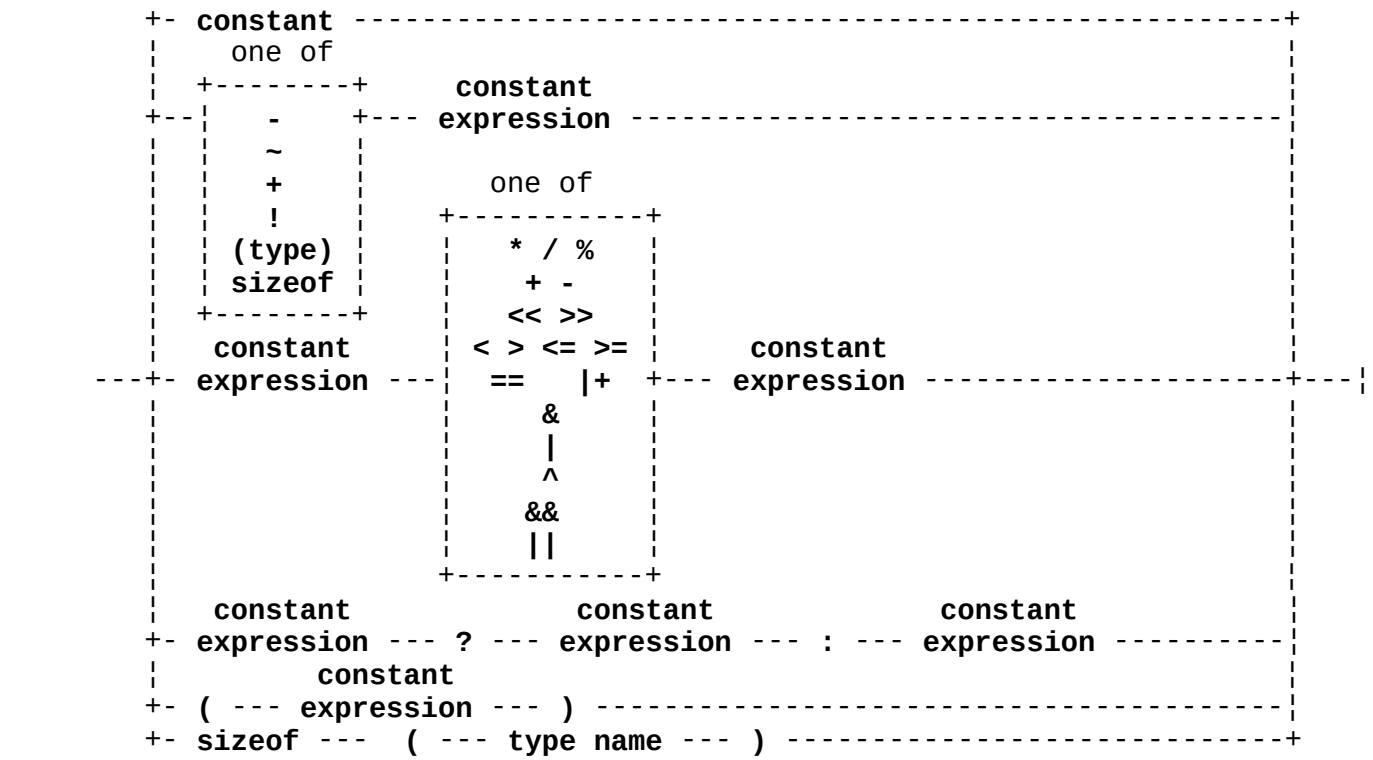
```
#include <stdio.h>
#include <math.h>

main()
{
        /* Declare some simple variables */
        int egress;
        float lightly=1.3;
        char coal='A';

        /* Now reference those variables */
        egress = 10;   /* Set it to a constant  */

        /* Pass it as an argument  */
        printf("%f", sin(lightly));

        /* Write it to the standard output */
        putchar(coal);
        putchar('\n');
}
```

If the type of the identifier is an array, then the value of the expression is a pointer to the first element in the array and the type of the expression is a pointer to the type of object in the array. If the identifier is a function not followed by a left parenthesis, (, then the value of the identifier, and the type of the expression, is the address of the function. The type of the expression is also a pointer to the function returning the type specified by the identifier function.

4.6.1.2 Constant Expressions

A ***constant expression*** in C consists of constant operands together with operators listed below. Parentheses ( ) may be used to alter the precedence of operators. The **sizeof** construct is considered a constant of type **int**. A constant expression has the form:

constant expression



The operators allowed are:

Binary Operators

+ - * / % & | ^ << >> == != < > <= >= || &&

Unary Operators

. -> [] & ~ + ! sizeof (type) sizeof <expression> (type)

(*Conditional*) *Ternary Operator*

?:

There are places in a C program where the compiler requires that expressions evaluate to a constant. These places are:

After a **case** in a **switch** statement

At the bounds of an array when the array is declare

In initializers for certain variable

Bit-field width

Enumeration constants

In the **case** and array bounds situations, the expressions can only use integer constants, character constants, enumeration constants, and **sizeof** expressions.

In the case of initializers, the rules are more relaxed. The constant expressions as defined above can be used, and in addition, the unary (**&**) operator can be applied to **extern** or **static** objects, to members of **extern** or **static structures** or unions and to **extern** or **static** arrays subscripted with a constant expression. The unary (**&**) can also be applied implicitly by the appearance of unsubscripted arrays or functions. In all cases, an initializer must eventually evaluate to either:

A constan

The address of a previously declared **extern** or **static** object plus or minus a constant.

4.6.2 Constants

A ***constant*** is a primary expression. The type of the constant may be **int**, **long**, **float**, or **double**, depending on its form. Integer constants have type **int**. Long constants are type **long int**. Floating-point constants are either of type **double** or **float**. Character constants are considered to be of type **int**.

Subtopics
4.6.2.1 Strings
4.6.2.2 Parenthesized Expressions
4.6.2.3 Member References
4.6.2.4 Function References

4.6.2.1 Strings

A ***string*** is a primary expression. Since the type of a string is assumed to be of type **array of char**, the result of the expression is actually a **pointer to char** and the value is a pointer to the first character in the string.

4.6.2.2 Parenthesized Expressions

A ***parenthesized expression*** is a primary expression whose type and value are the same as that of a plain expression without the parentheses. The presence of the parentheses does not affect whether the expression is an **lvalue**. However, brackets do affect the order in which operations are done. For example, ***A[I]** would mean use the **Ith** element of **A** as a pointer to the value desired, but **(*A) [I]** would mean use **A** as a pointer to an array, and use the value of the **Ith** element of that array.

4.6.2.3 Member References

A ***primary expression*** followed by a dot (.) followed by an identifier is an expression. The first expression must have **struct** or **union** type. The identifier normally must name a member of that **struct** or **union.** The result of the expression refers to the named member of the **struct** or **union**, and its type is the type of that member. The expression is an **lvalue** only if the **struct** or **union** expression preceding the dot (.) is an **lvalue**.

A primary expression followed by a (**->**) sign followed by an identifier is also an expression. The first expression must be a pointer to a **struct** or **union.** The identifier normally must be the name of a member of that **struct** or **union.** The result of the expression is an **lvalue** that refers to the named member of the **struct** or **union** and its type is the type of that member. The operator (**->**) is just a shorthand for a combination of the unary (*****) operator with the (**.**) operator; therefore, **a->b** is equivalent to **(*a).b**.

There are five operations that may be done on a structure or a union:

Referencing a member of the structure or union by means of the (**.**) or (**->**) operators.

Taking the address of the entire structure or union by using the (**&**) operator.

Finding the size of a structure or union by using the **sizeof** operator.

Passing (by value) the entire structure or union as an actua parameter to a subroutine.

Assigning the entire structure or union to another similarly type variable with a normal assignment statement.

The (.) operator is used in contexts where the structure or union identifier is available directly to the expression. The (**->**) operator is used when the identifier for the structure or union is a pointer to the object.

Examples of accessing members of structures:

```
#include <stdio.h>

struct record
{
     int number;
     struct record *next_num;
};

main()
{
   struct  record name1, name2, name3;
   struct  record *recd_pointer = &name1;
   int sum = 0;

   name1.number = 144;
   name2.number = 203;
   name3.number = 488;
```

```
    name1.next_num = &name2;
    name2.next_num = &name3;
    name3.next_num = NULL;

    while (recd_pointer != NULL)
    {
      sum += recd_pointer->number;
      recd_pointer = recd_pointer->next_num;
    }

    printf("sum = %d\n", sum);
}
```

4.6.2.4 Function References

A ***function call*** is a primary expression. The function call must be followed by parentheses ( ) containing a possible empty list of actual arguments to the function. The primary expression must be of the type:

```
function returning widget
```

and the result of that function reference is of type:

```
widget
```

A previously undeclared identifier followed immediately by a left parenthesis (is declared to be an **extern** function returning **int**.

When a function call is made, the compiler performs some automatic conversions:

Any actual arguments of type **float** are converted to **double** before the function call is made.

Actual arguments of type **char** or **short** are converted to type **int** before the call is made.

Array names are converted to pointers

Function names are converted to pointers

Types get implicitly converted if a prototype exists before the function call. If function prototyping is used and the types of the arguments do not match the types of the formal parameters indicated by the prototype, the arguments being passed are converted to the types of the parameters indicated by the prototype. If any argument cannot be converted, an error occurs. Also, if a function prototype exists, the number of passed arguments to the function must equal the number of parameters specified in the function prototype, or an error occurs. See Chapter 6, "Functions," for a full discussion of function prototypes.

Example:

```
double atan2(double y, double x);

main ()
{
printf ("Arc tangent of y/x=%f\n", atan2 (1,1));
}
```

In this example, both arguments to **atan2** are converted to type **double** before they are passed.

Note: Function prototypes are not supported on the RT.

The discussion on function declarations is covered in detail in Chapter 6, "Functions."

There are only two operations that may be performed on a function:

Invoking the function as part of an expressio

Taking the address of the function

If a function name is followed by a left parenthesis (, it is assumed to be a function reference. If a function name appears and is not followed by a left parenthesis, a pointer to the function is generated.

Example of function reference:

```
int acker(m, n)
     int m, n;
{
     if (m ==0)
        return(n + 1);
     else if (n == 0)
        return(acker(m - 1, 1));
     else
        return(acker(m - 1, acker(m, n - 1)));
 }
```

Example of passing functions:

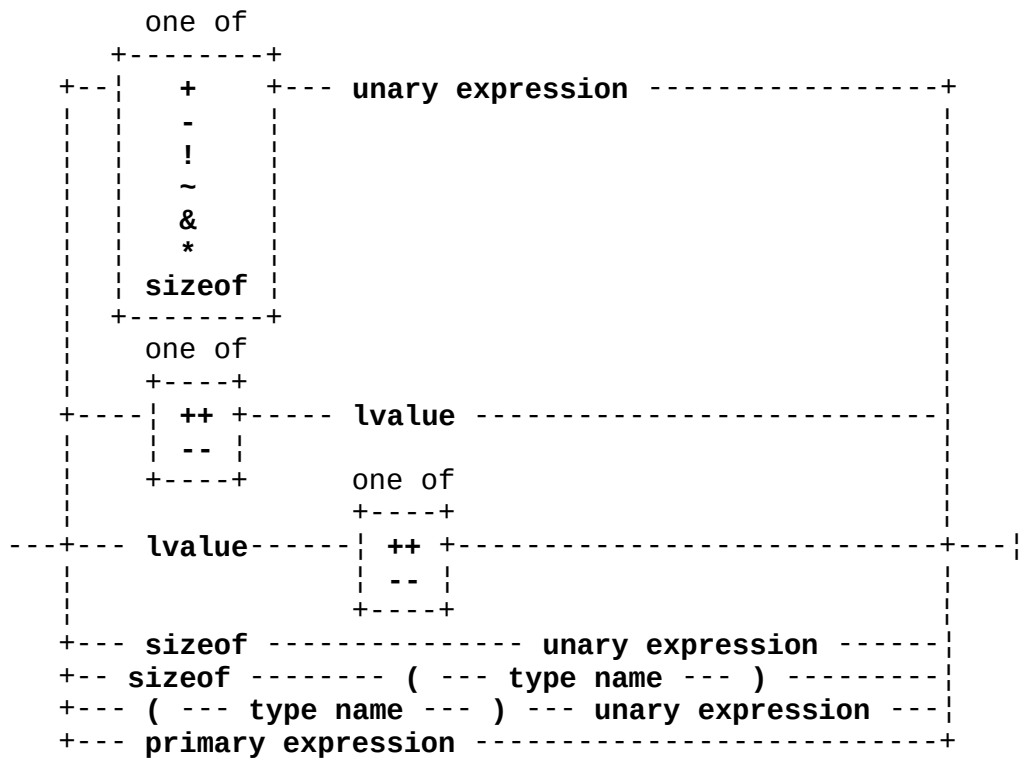
```
CloseGen(Type)
    char  Type;
{
    int  CloseBlock(), CloseList();

    if (Type == 'b')
        CloseEnv(CloseBlock);
    else if  (Type == 'l')
        CloseEnv(CloseList);
    else
        printf("Disaster\n");
}
```

This example shows the declaration of some functions that are passed as arguments to another generic function, depending on the value of some variable.

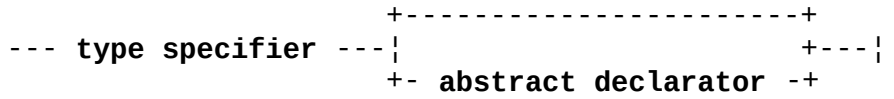
4.6.3 Unary Operators

Unary operators appear to the right or left of an expression and are considered to have a single operand. A unary expression has the form:

unary expression



type name



Indirection

The (*****) operator means indirection. The expression to the right of the (*****) operator must be a pointer. The result of the operator is an **lvalue** that refers to the object to which the pointer points.

If the type of the expression is:

pointer to widget

then the type of the result is:

widget

Examples of pointer references:

```
#include <stdio.h>
#include <math.h>

main()
```

```
{
        /*Declare some pointer variables */
        int *egress, egr;
        float *lightly, lgt =1.3;
        char  *coal = "A";

        /* Initialize the pointer variables */
        egress = &egr;
        lightly = &lgt;

        /* Now reference those variables */
        *egress = 10;   /* Set it to a constant */

        /* Pass it as an argument */
        printf("%f", sin(*lightly));

        /* Write it to the standard output */
        putchar(*coal);
        putchar('\n');
}
```

As mentioned previously in the discussion on array referencing, an array name can be used with the (*****) operator to access elements of an array by pointer arithmetic.

Example of array referencing with the ******* *operator*:

```
int egress[100];      /* Declare an array variable */
int idx;
                  /* Initialize the array to zeros */
    for (idx = 0;  idx < 100;  idx++)
        *(egress+idx) = 0;
```

In this example, note that the **lvalue** expression is enclosed in parentheses ( ). This is because the (*****) operator is evaluated before the (**+**) operator unless they are present.

```
*(egress+idx)
```

Address Evaluation

The (**&**) operator generates the address of the object referred to by the **lvalue** to the right of the operator. If the type of the **lvalue** is:

```
widget
```

the type of the result by applying the (**&**) operator is:

```
pointer to widget
```

The (**&**) operator can be applied to any **lvalue** except objects with the **register** storage class and bit field members of structs. This includes non-bit field members of structs, members of unions, arrays and array elements, functions, and any variable.

Examples:

&Gizmo generates the address of a variable named **Gizmo**

&Book_Case[shelf] generates the address of the **shelf** element of the array **Book_Case**.

&based->value generates the address of the member **value** of the **struct** or **union** that the identifier **based** points to.

Unary Plus

The unary plus operator (**+**) maintains the value of the operand. The expression must be numeric. The usual arithmetic conversions apply.

Note: The unary plus operator is not supported on the RT.

Unary Negation

The unary negation operator (**-**) generates the negative of the expression to its right. The expression must be numeric. The usual arithmetic conversions apply.

Logical Negation

The logical negation operator (**!**) generates a zero value if the value of the expression to its right is nonzero, and generates a value of 1 if the value of the expression to its right is zero. The type of the result of the operator is **int**.

The (**!**) operator can be applied to any numeric type and to pointers.

Logical Ones Complement

The logical ones complement operator (**~**) generates the ones complement of the expression to its right. The usual arithmetic conversions are performed on the operand. The type of the operand must be integral.

Increment and Decrement

The increment and decrement operators, (**++**) and (**--**), may be applied to their operands either as prefix or as postfix operators. The types accepted include all numeric types (including **float**, **double**, **long double**) and pointer. The operands must be **lvalues**. The result of the operation is not an **lvalue**.

When either the (**++**) or the (**--**) operators are applied as prefix operators, the object that the **lvalue** refers to is incremented or decremented. The value of the resulting expression is the new incremented or decremented value.

When either the (**++**) or the (**--**) operators are applied as postfix operators, the result is the original value of the object referred to by the **lvalue**. After the result of the expression has been noted, the object is incremented or decremented. The type of the result is the same as the type of the **lvalue** expression.

When applied to a numeric object, that object is incremented or decremented by one. When applied to any object of type pointer, that object is incremented or decremented by the size of the type of the object that it points to. Thus if p is type pointer to **double**, the p++ increments p by the size of a **double**, which is 8.

Examples:

```
int a,b,c,d,e;
a = 6;       /*  Set up a */
b = ++a;     /*  b becomes 7;  a becomes 7 */
c = --a;     /*  c becomes 6;  a becomes 6 */
d = a++;     /*  d becomes 6;  a becomes 7 */
e = a--;     /*  e becomes 7;  a becomes 6 */
```

These examples illustrate the use of the increment and decrement operators, showing the results generated by the prefix and postfix forms.

Cast

A type **cast** is used to specify an explicit conversion of an expression to a specific type. The unary expression being converted as well as the parenthesized **type name**, must have numeric or pointer type.

A **cast** may or may not actually change the bit pattern of the expression. Conversion of an **int** to a **float** does, but conversion of a pointer to **char** into an **int** does not. A pointer may be converted to an integral type, and integral types may be converted to pointers.

Example:

```
#include <stdio.h>

main()
{

  int value;

  value = 10;
  printf("integer = %f\n", (double) value);
}
```

Sizeof

When the **sizeof** operator is applied to an array or structure, the result is the total number of bytes in that array or structure. When the **sizeof** operator is applied to a **union**, the result is the size of the largest member of the **union**. In general, the **sizeof** operator results in the number of storage bytes the following object occupies, including any rounding up to accommodate alignment. For example, when the **sizeof** operator is applied to the following structure, the result is 16 since the structure is given 16 bytes when stored.

Example:

```
struct s1{
  int i;
  char char_array[11];
      };
```

However, when the **sizeof** operator is applied to the following structure, the result is 15.

Example:

```
struct s2{
  char a[4];
  char char_array[11];
      };
```

This example shows that the compiler aligns elements within a structure based on the alignment of the most restrictive element. See the "Data Representation" sections in *C Language User's Guide* for information on storage and alignment.

The **sizeof** operator is semantically equivalent to an integer constant, and it may be used anywhere an integer constant can be used.

The **sizeof** operator may also be applied to a type name enclosed in parentheses ( ). In this case, the value is the size (in bytes) of any object which has that type.

The construction **sizeof** (type) is taken as a single indivisible unit.

Examples:

```
int  fred [10];
int a , b;

   a = sizeof fred;              /*  a becomes 40  */

   b = sizeof (double);          /*  b becomes 8   */
```

In this example, the result of the first **sizeof** operator is 40 since the size of the **fred** array is 10 times 4 bytes. The second **sizeof** operator is taking the size of a **double** object, which is 8 bytes in this implementation. It is illegal to apply the **sizeof** operator to a bit field.

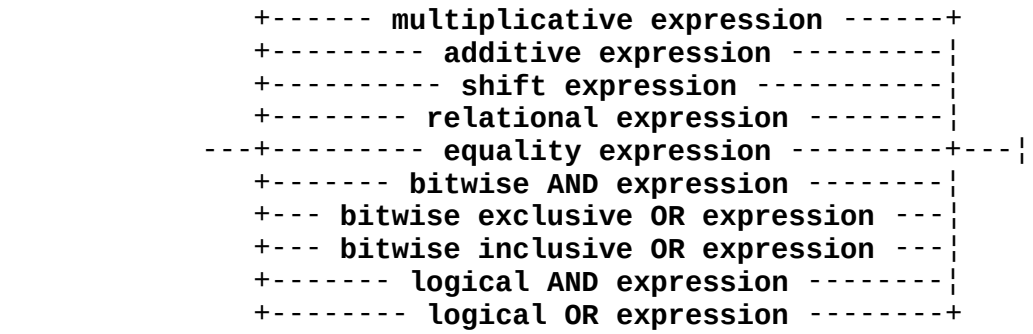
Note: The RT C Compiler allows the **sizeof** operator on a bit field. It always returns 4.

Subtopics

4.6.3.1 Binary Operators

Binary operators have two operands. A binary expression has the form:

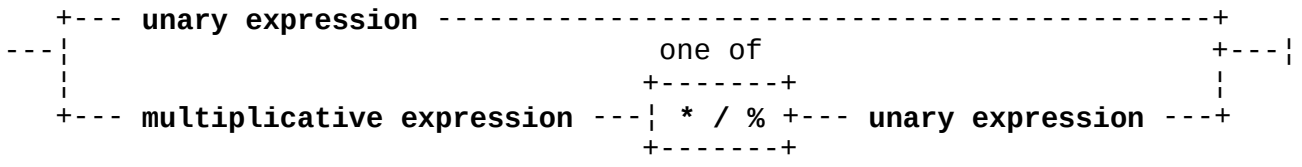
binary expression



4.6.3.2 Multiplication Operators

The multiplication operators (*****, **/**, and **%**) group from left to right.

multiplicative expression



The (*****) operator means multiplication. Both operands must be numeric. The usual arithmetic conversions are performed.

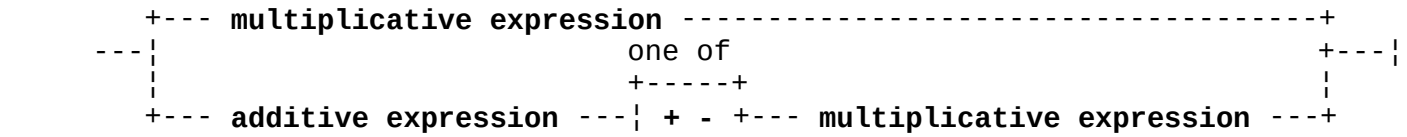
The (**/**) operator means division. Both operands must be numeric. When integers are divided, the result is truncated towards zero. If the right operand of the binary
(**/**) operator is zero, an illegal instruction error is generated for integral operands, and INF (Infinity) for floating-point operands. The usual arithmetic conversions are performed. If either operand is **unsigned int** or **unsigned long**, then unsigned arithmetic is done; otherwise, signed arithmetic is performed.

The (**%**) operator defines the remainder operation between its operands. Both operands must be integral value; **float** and **double** are not allowed. An illegal instruction error is generated if the right operand of (**%**) is zero. If either operand is **unsigned int** or **unsigned long**, then unsigned arithmetic is done, otherwise signed arithmetic is done. The interpretation of (**%**) is:

```
a % b = a - (a / b) * b
```

4.6.3.3 Addition Operators

additive expression



The addition operators (**+** and **-**) group left to right. The usual arithmetic conversions are performed, as described at the start of this chapter. Both operands may be numeric or, in some cases, pointers.

The result of the (**+**) operator is the sum of its operands. The usual arithmetic conversion rules determine if the addition is to be done in floating-point arithmetic or integral. Two pointers cannot be added. A pointer and an integral type may be added, in either order, but the result is not simply adding the two values. If the pointer points to an object of size *n* bytes, then the integer is multiplied by *n* before the addition is performed. Thus, the value of the expression is the address of the *nth* element relative to what *p* points to, as if it were an array. Hence, if *a* is an array, then the value of the expression ***(a+n)** is the value of the *nth* element in *a*, exactly the same as indexing *a* by *n*, *a[n*].

The result of the (**-**) operator is the difference of its operands. The usual arithmetic conversions are performed. A value of integral type may be subtracted from a pointer, and the same discussion applies to subtraction as to addition.

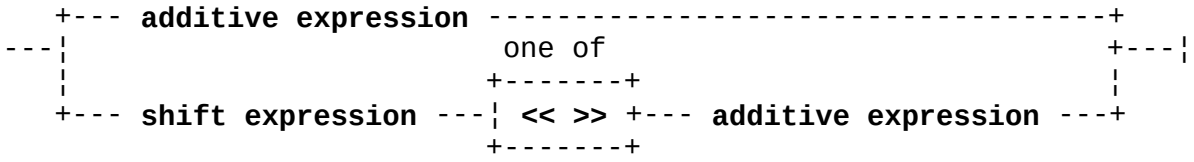
Two pointers to objects of the same type may be subtracted. The result is converted (by dividing the result by the size in bytes of an object of the type pointed to by the pointer) to an **int** whose value is the number of objects separating the objects pointed to. Note, however, that if the pointers do not point to elements in the same array, the distance between them may not divide evenly and the result will probably be meaningless.

4.6.3.4 Shift Operators

The shift operators are left shift (**<<**) and right shift (**>>**). The shift operators group left to right. Both operands of the shift operators must be integral. The usual arithmetic conversions are performed on the operands.

The right operand of a shift operator is converted to an **int**. The type of the result is the same as the type of the left operand.

shift expression



If the right operand is negative, or if the right operand is greater than the number of bits in the left operand, the results are undefined.

The (**<<**) operator is a left shift. The result of the expression:

```
E1 << E2
```

is E1 (interpreted as a bit pattern) shifted left the number of bits given by the value of E2. The vacated bits are filled with zeros.

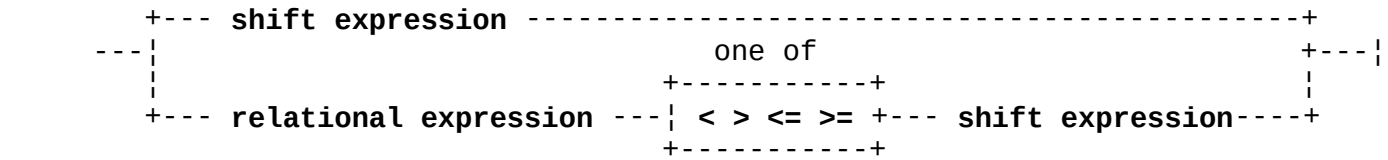
The (**>>**) operator is a right shift. The result of the expression:

```
E1 >> E2
```

is E1 (interpreted as a bit pattern) shifted right the number of bits given by the value of E2. If the expression E1 is **unsigned int** or **unsigned long**, the right shift is a logical right shift; that is, the vacated bits are filled with zeros. If the expression E1 is signed, the right shift is an arithmetic shift; that is, the sign bit is propagated.

4.6.3.5 Relational Operators

relational expression



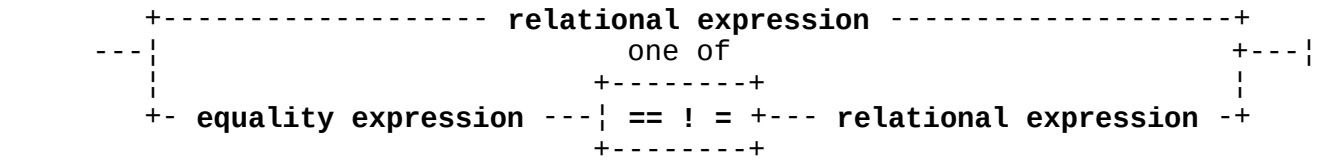
The relational operators group left to right. Relational operators apply to all numeric types as well as to pointers and the usual arithmetic conversions apply.

All these operators generate a value of 1 if the specified relation is true, and 0 if the specified relation is false. The type of the result is of type **int**. The usual arithmetic conversions are performed on the operands. If either operand is unsigned **int**, an unsigned comparison is done; otherwise, a signed comparison is done.

Pointers to the same type may be compared. The result of comparing pointers depends on the relative locations of the pointed to objects in the address space of the machine. An unsigned comparison is done when comparing pointers.

4.6.3.6 Equality Operators

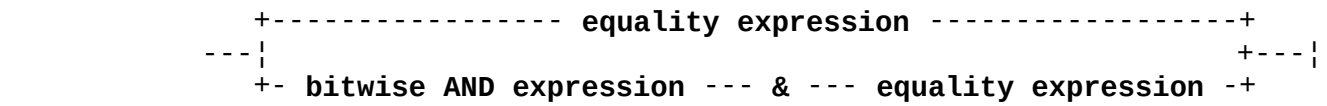
equality expression



The equality operator (**==**) and inequality operator (**!=**) have a lower precedence than the relational operators. The result is either a value of 1 for true or 0 for false. The type of the result is **int**. Both operands must be arithmetic types, must be pointers to objects of the same type, or one must be a pointer to an object and the other a pointer to **void**, or one must be a pointer and the other must be 0.

4.6.3.7 Bitwise AND Operator

bitwise AND expression



The bitwise AND operator (**&**) performs a bit-by-bit AND function on the bits of its operands. The usual arithmetic conversions are performed on the operands. Both operands must be integral.

The following table is the truth table for the bitwise AND operator (**&**) :

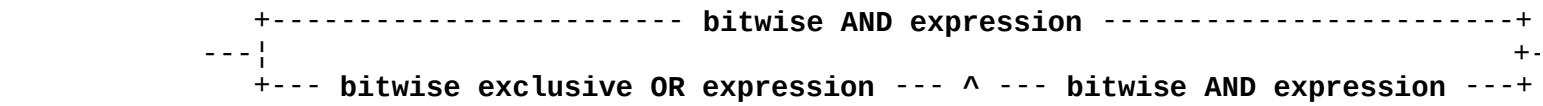
E1	**E2**	**E1 & E2**
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
#define SIGNCLR 0x7fffffff
demo(thing)
int  thing;
{

    return(thing & SIGNCLR); /* Clears the sign bit */
}
```

4.6.3.8 Bitwise Exclusive OR Operator

bitwise exclusive OR expression



The bitwise exclusive OR operator (^) performs a bit-by-bit exclusive OR function on the bits of its operands. The usual arithmetic conversions are performed on the operands. Both operands must be integral.

The following table is the truth table for the bitwise exclusive OR operator (^) :

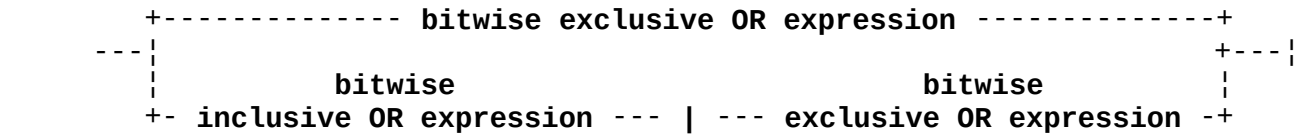
E1	E2	E1 ^ E2
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```
#define  TOGGLE  0x80000000
int demo(thing)
int  thing;
{

    return(thing ^ TOGGLE); /* Toggles the sign bit */
}
```

4.6.3.9 Bitwise Inclusive OR Operator

bitwise inclusive OR expression



The bitwise inclusive OR operator (**|**) performs a bit-by-bit inclusive OR function on the bits of its operands. The usual arithmetic conversions are performed on the operands. Both operands must be integral.

The following table is the truth table for the bitwise inclusive OR operator (**|**) :

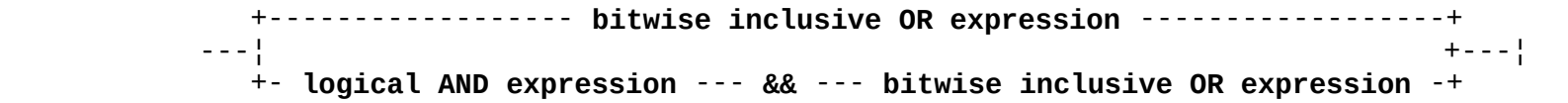
E1	**E2**	**E1 | E2**
0	0	0
0	1	1
1	0	1
1	1	1

Example:

```
#define  SETSIGN 0x80000000
int demo(thing)
int  thing;
{
   return(thing | SETSIGN); /* Sets the sign bit */
}
```

4.6.3.10 Logical AND Operator

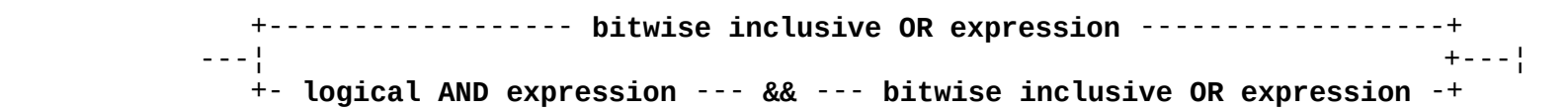
logical AND expression



The logical AND operator (**&&**) returns the value 1 if both of its operands are true (nonzero) and 0 if either is false (zero). Each operand must have numeric or pointer type. The type of the result is **int**. Left-to-right short circuit evaluation is guaranteed; that is, the left operand is checked first. If it is false, then the result of the expression must also be false, so the value of the right operand is never computed. This fact can be useful as illustrated in the following example.

Example:

logical AND expression

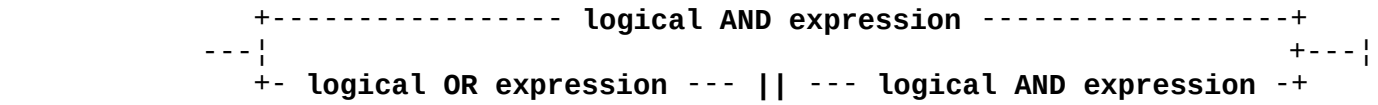



```
while ( p!=NULL && p->size > 10 ) {
        . . .
        p = p->next;
        }
```

If the full expression were evaluated in the case in which p has the value NULL, then an erroneous memory reference may occur when attempting to evaluate **p->** size.

4.6.3.11 Logical OR Operator

logical OR expression



The logical OR operator (**||**) evaluates the boolean OR of its operands. Each operand must have numeric or pointer type. The type of the result is **int**. The value 1 is returned if either operand is true (nonzero); otherwise, the value 0 is returned. As with the logical operator, **short** circuit left-to-right evaluation is guaranteed.

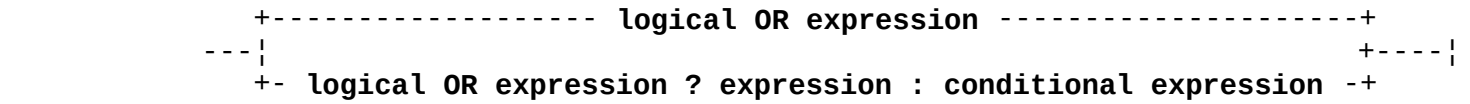
Example:

```
if (p == NULL || p-> size < 10)
     printf ("error\n");
```

In the example, it is important that the second operand of the **||** operator *not* be evaluated if the first is true. The logical OR operator ensures that this is the case.

4.6.3.12 Conditional Expression

conditional expression



The conditional operator (**?:**), sometimes known as the ternary operator, selects between the value of two expressions based on a boolean expression. The value of the first operand is computed. If it is true (nonzero), then the value of the second operand is computed and returned as the value of the conditional expression. In this case, the third operand is never evaluated. If the value of the first operand is false (zero) then the second operand is skipped and the value of the third operand is used. The result is not an **lvalue**.

If both the second and third operands have arithmetic type, then the type of the result is determined by applying the usual arithmetic rules to both operands. If both the second and third operands have **void** type, then the result has **void** type. If both the second and third operands have the same **struct** or **union** type, the result has that type. If both the second and third operands are pointers to the same type, or one is a pointer and the other is an integer constant, then the result has that pointer type. If either the second or third operand is a pointer to an object and the other is a pointer to a **void**, then the result has type pointer to **void**.

Example:

```
#define MAXSECTORS  15

demo(blocks)
   int blocks;
{

   int sec_count;
   sec_count = (blocks > MAXSECTORS) ? MAXSECTORS : blocks;
}
```

The example demonstrates the shorthand notation that the conditional operator provides.

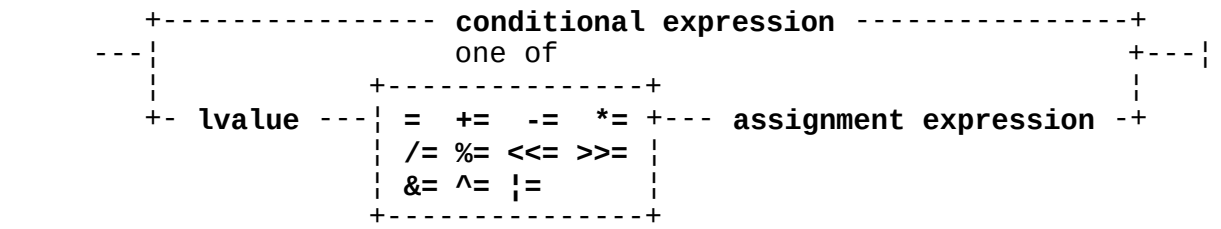
The equivalent program using other conditional forms might be:

```
#define MAXSECTORS  15

demo(blocks)
    int blocks;
{
    int sec_count;

    if (blocks > MAXSECTORS)
       sec_count = MAXSECTORS;
    else
       sec_count = blocks;
}
```

4.6.3.13 Assignment Operators

Assignment operators assign run-time values to objects. An assignment expression has the form:

assignment expression



It takes two operands, an **lvalue** on the left and an expression on the right. The meaning is to assign the value of the right expression to the object described by the **lvalue**. Both operands must have arithmetic type, the same structure type, or the same **union** type. Otherwise, both operands must be pointers to the same type, or the left operand must be a pointer and the right operand must be the constant 0 (zero). If both operands have arithmetic type, the value of the right operand is converted to the type of the **lvalue** prior to the assignment. The result of an assignment operator is the value assigned into the object described by the **lvalue**, and the type is the type of that object. Assignment associates to the right. Thus the expression **i=j=0** assigns the value **0** to first **j**, and then assigns **j** to **i**. The result of an assignment operator is not an **lvalue**.

C contains a series of other compound assignment operators that provide a shorthand for operating on an object. In addition, the compiler can sometimes generate more efficient code when the compact operators are used. Both operands of a compound assignment operator must have arithmetic type consistent with those allowed for the corresponding binary operator, except for (**+=**) and (**-=**) operands must have numeric type, or the left shall have pointer type and the right shall have integral type.

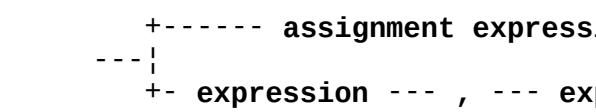
The meaning of **a op= b** is similar to **a = a op b**, except that it is guaranteed that the **lvalue a** is only evaluated once. Thus if the **lvalue** has any side effects, the result is well defined, such as **A[I++] += 5**.

Type conversions are performed as follows. The usual arithmetic conversions are performed, depending upon the operator, and the result is computed. Then, if needed, it is coerced into the type of the **lvalue**, as described previously for the assignment operator , (**=**) and the result is stored into the object described by the **lvalue**.

4.6.3.14 Comma Operator

The comma operator (,) is used to combine a pair of expressions into a single expression. A comma operator has the form:

comma expression



When the comma operator appears between a pair of expressions, the left expression is evaluated and its value is then discarded. Then the right expression is evaluated, and the type and value of the resulting expression is the type and value of the right expression. The expressions are evaluated left to right.

The comma operator is most useful when the syntax of C expects a single expression but it is necessary to evaluate more than one expression. An example would be initializing two variables at the start of a for loop, as illustrated here:

```
for (i = 0,j = 10; i<20; i++) { ... }
```

If used in a function call, the comma expression must be enclosed in parentheses ( ).

5.0 Chapter 5. Statements

Subtopics
5.1 CONTENTS
5.2 About This Chapter
5.3 Statements

5.1 CONTENTS

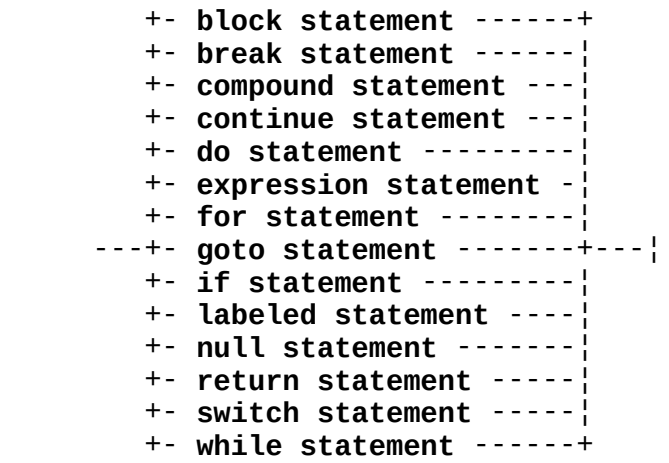
5.2 About This Chapter

This chapter describes C statements and how they are used in programs.

5.3 Statements

The C programming language contains expression statements and control flow statements. The expression statements are used to compute and assign new values to objects at runtime. The control-flow statements determine the order in which the computations are performed. Below is a summary of the statements available in C:

statement



Structured statements specify sequential, selective, or repetitive execution of their component statements. Sequential execution is specified by the compound statement; conditional and selective execution by the if statement and the **switch** statement; and repetitive execution either by the while and **do-while** statements or by the for statement. The **break** statement provides the means to exit a loop prematurely or to end a **case** of a switch statement; the continue statement starts the next iteration of its enclosing loop.

Subtopics
5.3.1 Expression Statement
5.3.2 Compound Statement
5.3.3 Conditional Statement
5.3.4 Switch Statements
5.3.5 While Statements
5.3.6 Do Statement
5.3.7 For Statement
5.3.8 Break Statement
5.3.9 Continue Statement
5.3.10 Return Statement
5.3.11 Goto Statement and Labels
5.3.12 asm Statement
5.3.13 Null Statement

5.3.1 Expression Statement

An ***expression statement*** contains an expression. An expression statement has the form:

expression statement

```
--- expression --- ; ---¦
```

An expression (as described in Chapter 4, "Expressions") becomes a statement when it is followed by a semicolon (;). The semicolon is the statement terminator. For example, the following constructs:

```
temp = 25
++count
printf("hello\n")
```

are all expressions, and may generate values that can be used in the context of larger expressions. If they are followed by a semicolon, they become statements:

```
temp = 25;
++count;
printf("hello\n");
```

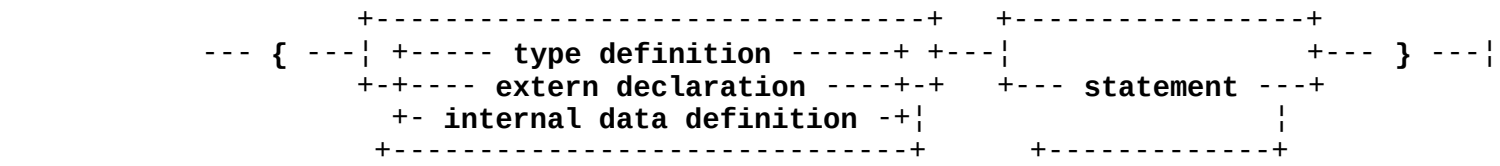
An expression statement is normally only useful when it has some kind of side effect, such as assigning a new value to a variable or calling a function that does something. It is permissible, however, to have a statement that is just a value, such as **I+3**; or even **25**; but this is of minimum advantage.

Note: The semicolon is part of a C statement. It is not a statement separator as in other languages.

5.3.2 Compound Statement

A ***compound*** or ***block statement***, is a sequence of statements grouped together so that they appear to be a single syntactic statement. This is done by surrounding the sequence of statements with braces, **{** and **}.** The compound statement has the form:

block statement



Note that new variables may be declared at the beginning of any compound statements. Any new identifiers declared specifically within a compound statement have a scope and lifetime that are bound by that statement; that is, they are both unknown outside of that statement and, in the case of **auto** and **register** variables, become undefined upon exit from that statement.

The following example shows how the values of data objects change in nested blocks:

```
1   #include <stdio.h>
2
3   main()
4   {
5      int x = 1;                  /* Initialize x to 1 */
6      int y = 3;
7
8      if (y > 0)
9      {
10        int x = 2;               /* Initialize x to 2 */
11        printf("second x = %4d\n", x);
12     }
13     printf("first x = %4d\n", x);
14  }
```

The preceding example produces the following output:

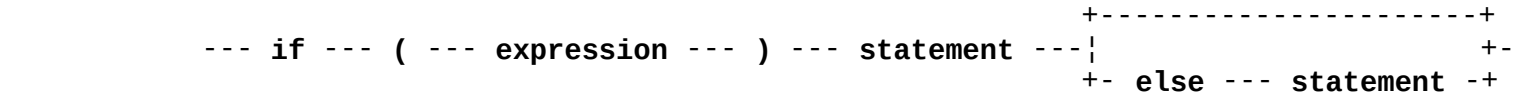
```
second x =   2
first  x =   1
```

Two variables named **x** are defined in **main**. The definition of **x** on line **5** retains storage throughout the execution of **main**. However, since the definition of **x** on line **10** occurs within a nested block, line **11** recognizes **x** as the variable defined on line **10**. Line **13** is not part of the nested block. Thus, line **13** recognizes **x** as the variable defined on line **5**.

5.3.3 Conditional Statement

The basic conditional statements in C are the ***if statement*** and the ***if-else statement***. The **if** and **if-else** statement have the form:

if statement



The type of the expression must be either numeric or pointer. If the type is a pointer, then the statement is equivalent to testing the pointer for the value NULL. A non-NULL value causes the first statement to be executed.

In both cases, the expression is evaluated. If the value of the expression is nonzero (true), the first statement is executed.

If the value of the expression is zero (false), the actions defined by the second statement are performed, assuming that there is an else part in the statement. If there is no else part, the next statement in order after the **if** statement is executed.

Because statements are open forms, it is possible to construct a chain of **else-if** clauses to select one out of many different conditions.

In common with similar languages, C resolves the so-called *dangling else* problem by having the else clause match the most recent **un-elsed** preceding if statement. The following example clarifies this point. If an alternate grouping of else statements is required, the nested if statement that does not contain an else clause must be enclosed in braces { }, making it a compound statement containing a single statement.

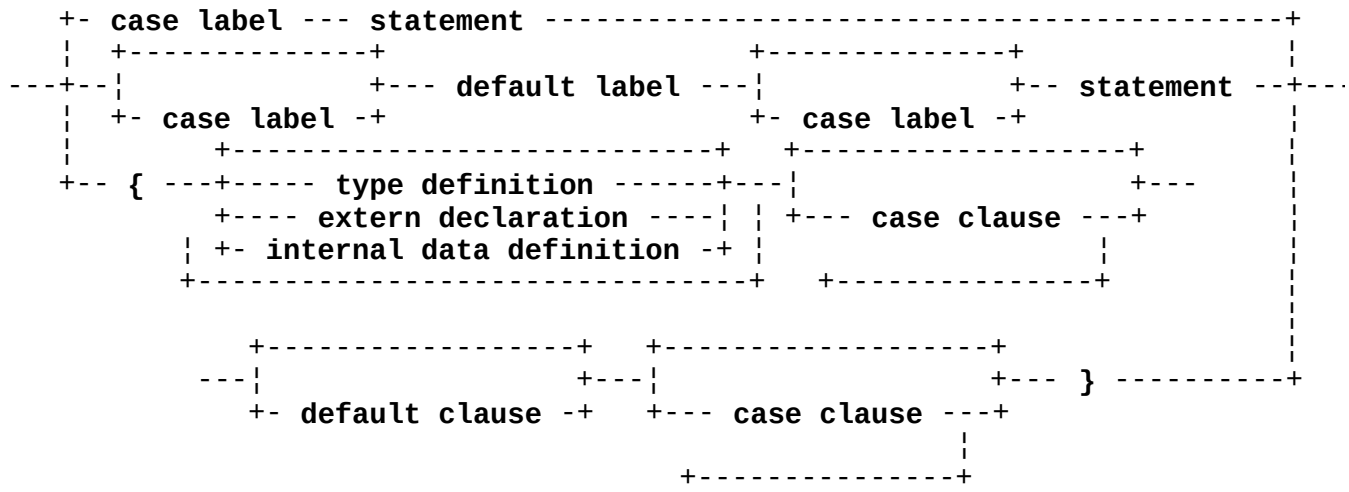
Examples:

```
main()
{
     int paygrade = 6;
     int level = 3;
     float salary = 10.30;

     if (paygrade == 7)
                 if (level > 0 && level < = 8)
                  salary *= 1.05;
                 else
                  salary *= 1.04;
     else
                  salary *= 1.06;
}
```

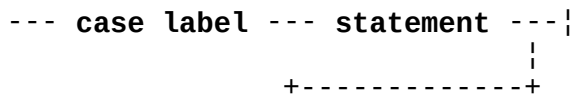
5.3.4 Switch Statements

switch statement

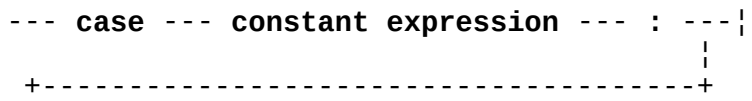
switch body



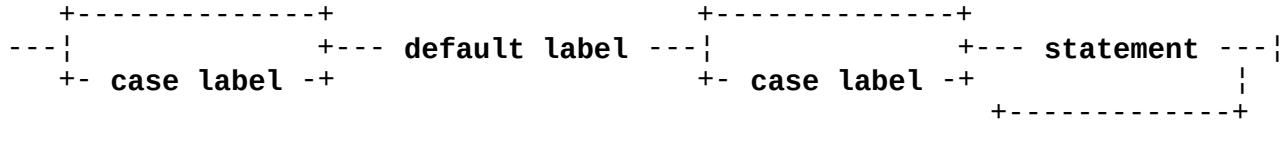
case clause



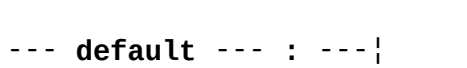
case label



default clause



default label



A ***switch statement*** selects one of its component statements depending on the value of the expression. The expression is called the switch selector. It must be an integral type. Each of the component statements is tagged with one or more simple scalar constants. The tags are called selection specifications.

If the value of the selector matches that of one of the statement tags, control is transferred to that statement. Control continues from the selected statement onwards until altered by another change of control. If the selector value matches none of the statement selection specifications, the statement tagged by a **default** symbol is executed, if present. If no **default** statement exists, then the body of a **switch** statement does nothing.

Example:

```
main()
{
 char command;

 command = getchar();
 switch (command)
 {

 case 'H':
 case 'h':
     leftcursor();
     break;

 case 'L':
 case 'l':
     rightcursor();
     break;

 case 'J':
 case 'j':
     downcursor();
     break;

 case 'K':
 case 'k':
     upcursor();
     break;

 default:
     nomove();
  }
         /* End of switch statement */
}
```

As the example shows, the **switch** statement is normally used in conjunction with the **break** statement, which is described in "Break Statement" in topic 5.3.8. Without the **break** statement, execution of any selection would also execute all subsequent selections.

5.3.5 While Statements

A ***while statement*** controls repetitive execution of another statement until evaluation of an expression yields a zero value. The while statement has the form:

while statement

--- **while** --- **(** --- **expression** --- **)** --- **statement** ---¦

The value of an expression is computed. If it is nonzero (true), then the statement is executed. The value of the expression is then tested again and the statement is executed repeatedly while the value of expression remains nonzero (true). The type of the expression must be numeric or a pointer. When the expression evaluates to zero (false), control passes to the statement after the while statement. If the value of expression is zero at the time that the while statement is encountered for the first time, the subordinate statement is never executed. Contrast this behavior with the do statement described in "Do Statement" in topic 5.3.6.

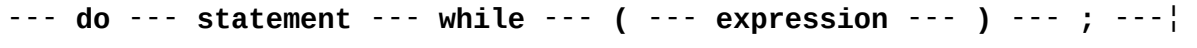
Example:

```
#define MAX_INDEX (sizeof(item) / sizeof(item[0]))

#include <stdio.h>

main()
{
     static int item[] = { 12, 55, 62, 85, 102 };
     int index = MAX_INDEX;

     while (--index >= 0)
     {
         item[index] *=3;
         printf("item[%d] = %d\n", index,
            item[index]);
     }
}
```

5.3.6 Do Statement

The ***do statement*** controls the repetitive execution of a list of statements. The statements are executed until the expression at the end of the statement evaluates to zero. The **do** statement has the form:

do statement



The statement between the **do** and **while** symbols is executed repeatedly and the expression evaluated until the expression is zero. The type of the expression must be either numeric or a pointer. Note that the body of a **do** statement is always executed at least once, since the termination test is at the end. Contrast this behavior with the **while** statement described in "While Statements" in topic 5.3.5.

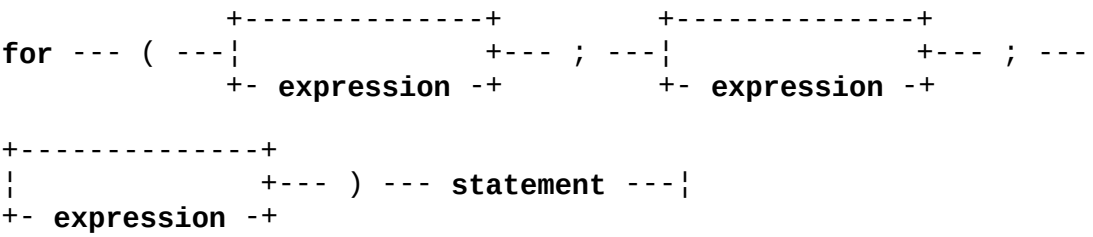
Example:

```
main()
{
    int reply1;

    do
    {
       printf("Enter a 1.\n");
       scanf("%d", &reply1);
    }  while (reply1 != 1);
}
```

5.3.7 For Statement

The ***for statement*** in C is a convenient and special form of the while statement. The **for** statement is more compact than the while, and is better suited to loops in which the control statements are single and logically related. The **for** statement has the form:

for statement



The three components of the **for** statement are expressions. Any or all of the expressions may be left out, but the semicolons must be there. The type of the expression 2 must be numeric or a pointer. If either expression 1 or expression 3 is left out, it is simply dropped from the expansion. If the expression 2 is omitted, the condition is always considered nonzero. This means that a *loop forever* construct can be made from a **for** statement that looks like this:

```
for (;;)  {

       . . . statements . . .

}
```

In such a loop, it is assumed that some other means of ending the loop (such as a **break** statement) is being used.

Examples:

```
          /* initialize an array to zero */
for (index = 0;  index < 100;  index ++)
     row[index] = 0;

       /* scan from the end of an array */
for (where = 200;  where > 0;)
    if (what[--where]== thing)
       foundit = 1;
```

5.3.8 Break Statement

The ***break statement*** provides a mechanism for breaking out of a loop structure prematurely. The **break** statement has the form:

break statement

The **break** statement supplies an exit mechanism from **while**, **for**, and **do** statements. A **break** must also be used to exit from the specific cases of a **switch** statement; otherwise, a **case** falls through to the next one. An error will occur if the **break** statement is used outside these statements.

A **break** statement causes an immediate exit from the innermost enclosing loop structure or **switch** statement. There is no provision for breaking out of loop constructs other than the innermost.

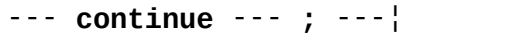
Example:

```
#include <stdio.h>
demo()
{

    int   ch;

    while ((ch = getchar())  != EOF)
      if  (ch == '\003')
         break; /*  exit when ETX read */
}
```

5.3.9 Continue Statement

The ***continue statement*** is the logical analog to the **break** statement. A **continue** statement causes the next iteration of the innermost enclosing loop structure to begin. The **continue** statement has the form:

continue statement



In a **while** or a **do** loop, the **continue** statement executes the test part of the structure. In a **for** loop, expression 3 is executed, followed by expression 2, the test condition. The **continue** statement does not apply to the **switch** statement. An error occurs if the **continue** statement is used outside a **for**, **while**, or **do** statement.

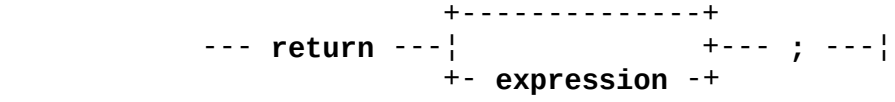
Example:

```
/*  weed out zero elements */
demo(array,n)
    int array[];
    int n;
{
    int from;
    int to = 0;

    for (from = 0; from < n;   from++)  {
        if (array[from] == 0)
           continue;
      array[to++] = array[from];
    }
    return(to);
}
```

5.3.10 Return Statement

A function returns to its caller by means of a ***return statement***. A **return** statement can return a value. The **return** statement has the form:

return statement



In the first form of the **return** statement, the value of the function from which the return is made is undefined. In the second form of the **return** statement, the value of the expression is the value of the function. If necessary, the expression is converted to the type of the function in which it appears. This form may not be used in a **function** defined with a return type of **void**.

If execution of a function reaches the end of the function (falling off the end), it is equivalent to a **return** statement without an expression. If a **return** statement without an expression is in a function where a return value is expected, the result is unpredictable.

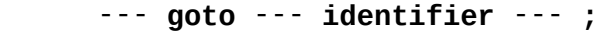
Examples:

```
return;            /* Returns no value */
return result;     /* Returns the value of result */
return 1;          /* Returns the value 1 */
return (x * x);    /* Returns the value of x * x */
```

5.3.11 Goto Statement and Labels

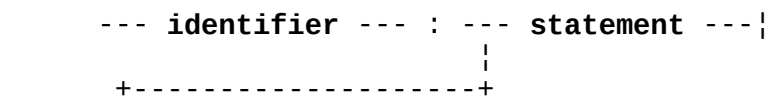
The ***goto statement*** passes control to a statement that has a label attached to it. The **goto** statement has the form:

goto statement



A ***label*** in the C Language is simply an identifier followed by a colon (:). If a label is to be associated with a statement, it must precede that statement. Transfer to a label via a **goto** statement continues execution with the statement preceded by the appropriate label. The label statement has the form:

labeled statement



The scope of a label is the entire function in which that label is defined. Nested scopes defined by compound statements have no effect of limiting the range of a label. It is not possible (nor valid) to jump into a function from outside that function. It is possible to jump into the middle of a structured statement, although this is not a safe practice. Using **goto** can violate structured programming constructions; the use of the **goto** statement should be evaluated carefully. It is sometimes difficult for the user to be aware of the results of **goto** when the label used is internal to a statement.

A label becomes defined when it is encountered as a label on a statement in the body of the function.

Example:

```
if  (status == error)
       /*  exit to end of function  */
    goto wipeout;
...statements...
wipeout: ...
```

The following example shows a **goto** statement that is used to jump out of a nested loop. This function could be written without using a **goto** statement.

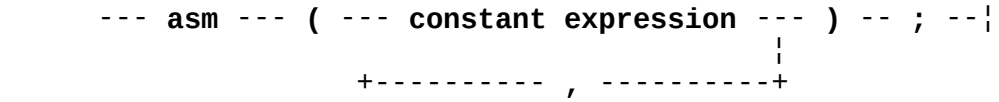
```
      void display(matrix)
      int matrix[3] [3];
      {
       int i, j;

       for (i = 0; i < 3; i++)
          for (j = 0; j < 3; j++)
          {
          if ( (matrix[i] [j] < 1) | | (matrix
                 [i] [j] > 6) )
```

```
            goto out_of_bounds;
          printf("matrix[%d] [%d] = %d\n", i, j,
                  matrix[i] [j]);
           }
 return;
 out_of_bounds: printf("number must be 1 through 6\n");
}
```

5.3.12 asm Statement

The ***asm statement*** is provided for the very limited insertion of machine instructions in-line in a program written in C. The **asm** statement on the PS/2 VSC C compiler has the form:

asm statement



The comma separated list of constant expressions is interpreted as a sequence of values of type **char**. The values are placed into the generated object code *text space* (instructions) literally at the point of the **asm** statement.

The **asm** statement on the AIX/370 and PS/2 MCC compilers has the form:

```
_ASM ("    assembler instruction    \n"):
```

The assembler code is actual 370 or PS/2 assembler mnemonics.

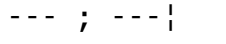
Note: Embedding assembler code can disturb the contents of registers that the compiler uses. Therefore, take care when using the **asm** statement.

Also, portability of C programs containing **asm** statements is likely to be restricted. A more portable usage would be to call an assembly routine, instead of coding in-line. See *AIX C Language User's Guide* program examples for an example of the **asm** statement.

5.3.13 Null Statement

A ***null statement*** can appear in a C program simply by your entering a semicolon (;). The **null** statement has the form:

null statement



A **null** statement is useful to carry a label before the closing bracket of a compound statement, or to supply a null body to a **loop** statement where all the computation is done in the loop control expression.

Examples:

```
if (snarks == boojums)
{
   . . .
   statements
     . . .
   if (time_to_get_out)
        goto out;
  . . .
      more statements
     . . .
out: ;
}

for (i = 0;  i <= 100 && what[i] == 0; i++)
    ;
```

Beware of extra semicolons. They can easily lead to loops with null bodies, producing unexpected results, as this example shows:

```
while ((c = getchar()) !=EOF) ;
  nextok();
```

In the example, the semicolon inadvertently placed after the closing parenthesis of the **while** makes the loop consume the entire input file, and the **nextok** function only gets called after the end of file is reached.

6.0 Chapter 6. Functions

Subtopics

6.1 CONTENTS

6.2 About This Chapter

This chapter describes the structure and usage of functions.

6.3 Functions

Functions are the fundamental C method of grouping blocks into manageable units. Declaring a function requires an identifier and, usually, a type description.

C differs from some languages in that it contains only functions. The effect of a procedure or subroutine is achieved by declaring a **void** function, one that does not return a result. C functions have similarities to mathematical usage -- a C function is applied to some arguments and generates a result. C functions differ from the rigorous mathematical variety in that they can have side effects, such as altering a global variable or writing to a file. The type of the returned value may be specified as part of the function declaration.

A function can itself contain declarations of new objects and compound statements, but a function may not contain other functions. These newly defined objects can be referenced only within that function and are thus said to be ***local*** to the function. The program text that comprises a function body is called the scope of any identifiers declared local to that function.

Objects declared outside of any function in the compilation unit without the storage class specifier **static**, are said to be ***global***, in that their scope is that of the entire program. Objects declared outside of any function with the storage class specifier **static** have scope limited to the point where the object is defined to the end of the compilation unit only.

A function can have a number of arguments that are determined at the time the function is defined. Each argument is denoted by an identifier called the formal parameter. When a function is called, each of the formal parameters has the value of a run-time expression at the calling location associated with that parameter. This value, which is accessed by naming the formal parameter identifier, is called an actual parameter.

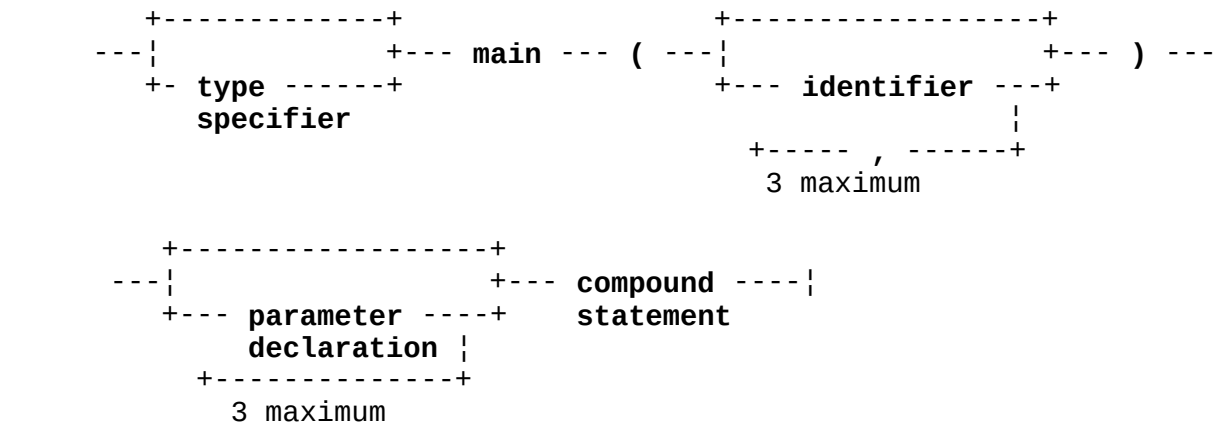
Arguments in C, except for arrays, are passed by value. That means that the called function receives a copy of the actual argument and cannot directly alter the value of a variable whose value is passed. For a function to modify the caller's copy of the argument, it must have a pointer to the argument and it must use pointer reference notation.

C functions can be ***recursive***. This means that a function may call itself again before the current activation has been completed. On each activation, a fresh set of all the automatic variables is created. Recursive invocation can be direct (the reference is contained within the function itself) or indirect (the reference is from another function that in turn is referenced from the current function).

6.4 The Main Function

An executable C program must have one function whose name is **main**. This **main** function is considered the place where program execution starts. Under normal circumstances, execution terminates upon exit from **main**. A **main** function has the form:

main function



The function **main** can declare optional parameters. The first parameter, **argc**, has type **int** and indicates how many arguments were entered on the command line. The second parameter, **argv**, has type array of pointers to **char** objects. The value of **argc** indicates the number of pointers in the array **argv**. The first element in **argv** always points to a character array that contains the name (as invoked) of the program that is executing.

A third parameter, **envp**, has type array of pointers to **char** objects. The array **envp** contains pointers to the environment of the program. The system determines the value of this parameter during program initialization (before calling **main**).

These parameters are always passed to **main**, and **argc** must always be declared first, followed by **argv**, then by **envp**.

Note: Some operating systems do not generate the **envp** parameter. You can access the value of the **envp** pointer using the function **getenv**. The AIX PS/2 Operating System does generate **envp** parameters.

Example:

```
main()
{
    . . .
}
```

Example:

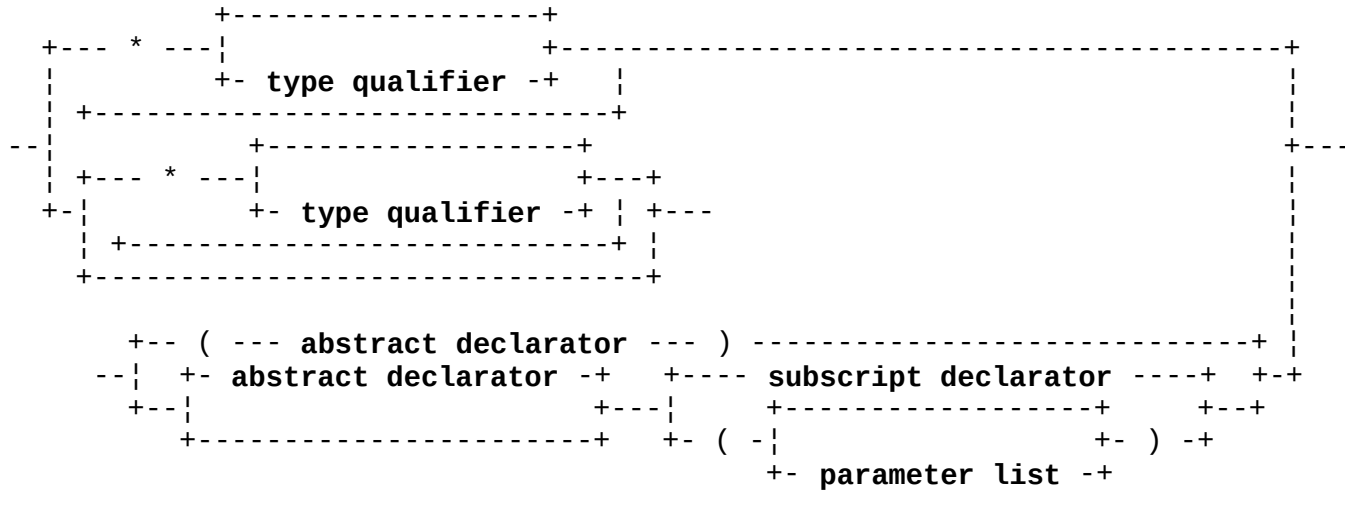
```
main (argc,argv)
        int argc;
        char    *argv[];
{
        int i;
```

```
        printf("argc = %d\n",argc) ;
        for (i = 0; i < argc; i++)
                printf("argv[%d] = %s\n", i,argv[i]);
}
```

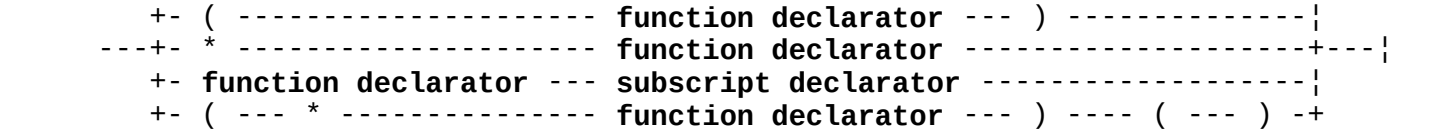
6.5 Defining Functions

A function definition usually contains the code for a single programming task.

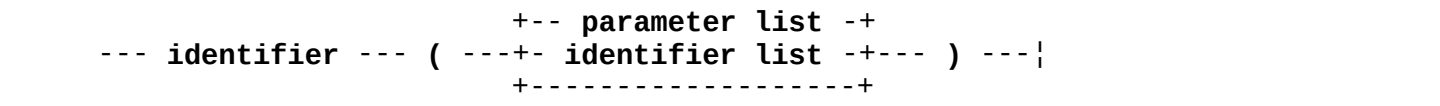
abstract declarator



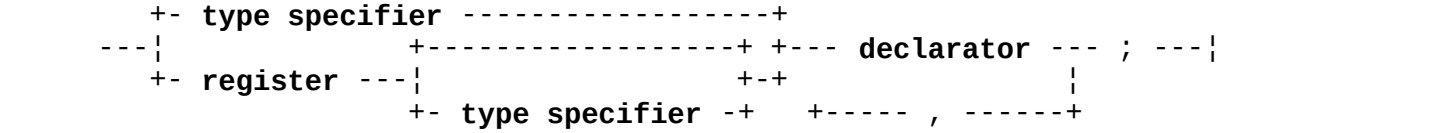
function declarator



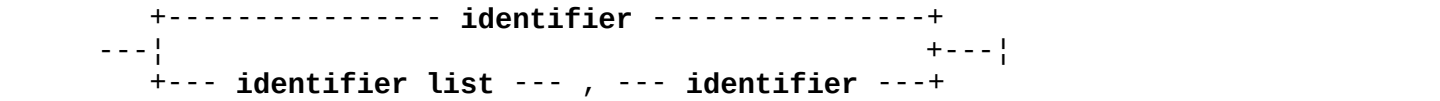
function header



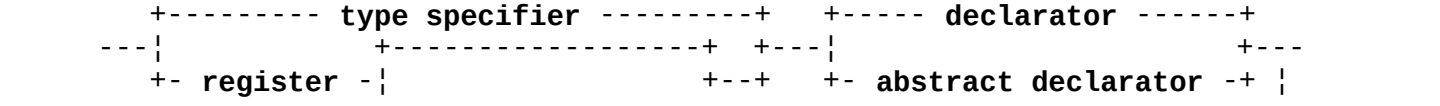
parameter declaration

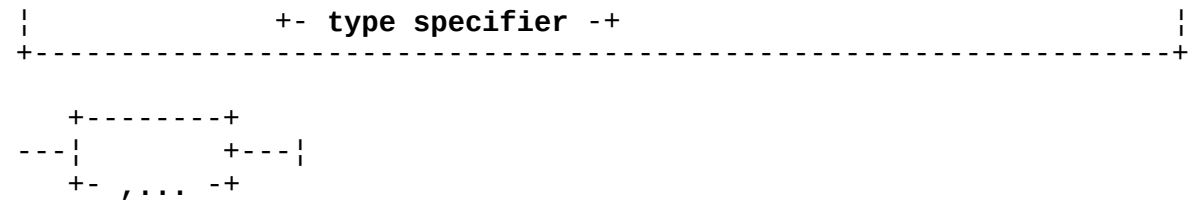


identifier list



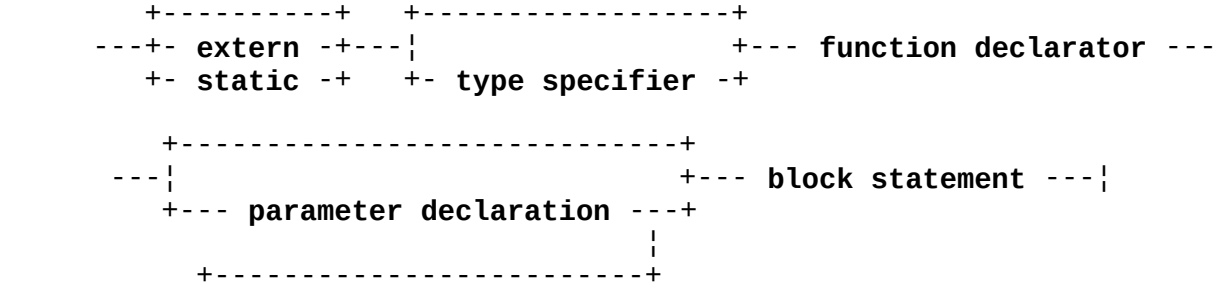
parameter list





A function is defined when a declarator is in the form:

function definition



Thus the simplest function, a dummy ***do nothing*** function, is:

```
useless()
{
}
```

A function is declared when a declarator of the above form is not followed by a compound statement. For example:

```
double sin(), cos(), tan() ;
```

The type specified in the function declaration specifies the type of the function. The type specification must be included in the function.

The traditional function definition requires the list of names of the formal arguments to the function in the function header. The declarations of the formal arguments must immediately follow the function header, before the braces { } that begin the body of the function. Any storage class for a formal argument, if given, must be **register**. Formal arguments that are not specifically declared are taken as type **int** with the **auto** storage class.

Example:

```
float balsa(p1, p2, p3) /*function header (not using      prototype for
  float p1;  /*declare formal arguments */
  int p2;
  char p3;
{            /*opening brace for the block */
    . . . .
      body of the balsa function
    . . . .
}            /*end of the function*/
```

The function prototype form of a function definition requires the names and types of the formal arguments to the function in the function header.

The brace which begins the function must immediately follow the function header. Any storage classes, if given, must be **register**. If the list terminates with an ellipsis (,...), no information about the number or types of the parameters after the comma is supplied. If **void** is used as the only item in the list, this specifies that the function has no parameters.

Note: The use of function prototypes is not supported on the RT.

Example:

```
       float balsa(float p1, int p2, char p3)
                       /* function header (using prototype form) */
       {
                       /*  opening brace for the block */
           . . . .
              body of the balsa function
           . . . .
       }
                       /*    end of the function       */
```

Example:

```
   int printf(char * format_string,...); /* declaration
            of printf using the prototype ellipsis form */
            /* defines printf's formal parameters to be at least
            a pointer to a char and also allows any number of
            parameters of any type after it */
```

Example:

```
  char *function(void);
                /* declares function to have no parameters */
```

Note that if prototyping is used in a function declaration, then argument names may or may not be supplied. These names have scope limited to that function declaration, and only exist for clarity of the parameter declarations.

Subtopics
6.5.1 Arguments to Functions
6.5.2 External Objects with the Static Attribute

6.5.1 Arguments to Functions

Parameters to C functions are always passed by value. The type of all integral expressions are first converted to **int** (that is, 4 bytes) and all floating-point expressions are converted to **double**. When an array or function is given as a parameter, a pointer to that array or function is passed since the type of an array name or function name (without a following left parenthesis) is *pointer to...*. A structure or union can be passed as an argument (by value) or the address of either can be passed with the use of the address operator (**&**).

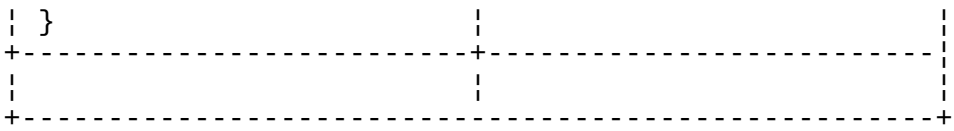
6.5.2 External Objects with the Static Attribute

When an external function or variable is declared in a compilation unit, it is possible to place the **static** storage class in the declaration of that object.

The meaning of the **static** storage class in the declaration of an external object is that the specified object is private to the compilation unit where it resides. Thus, the object is not visible outside the enclosing compilation unit, and is only directly accessible from functions in the same compilation unit. Of course, there is nothing to prevent the other functions in the compilation unit from passing the address of the **static** object to functions outside that compilation unit.

Example:

```
+-----------------------------------------------------+
| COMPILATION UNIT A       | COMPILATION UNIT B       |
+--------------------------+--------------------------+
| static  func_a()         | func_p()                 |
+--------------------------+--------------------------+
| {                        | {                        |
+--------------------------+--------------------------+
|    statements            |      int hold;           |
+--------------------------+--------------------------+
| }                        |                          |
+--------------------------+--------------------------+
|                          |     statements           |
+--------------------------+--------------------------+
| static func_b()          |     hold = func_a();     |
+--------------------------+--------------------------+
| {                        | }                        |
+--------------------------+--------------------------+
|                          |                          |
+--------------------------+--------------------------+
|    int grab;             |                          |
+--------------------------+--------------------------+
|                          | func_ind(who)            |
+--------------------------+--------------------------+
|    statements            |      int (*who)();       |
+--------------------------+--------------------------+
|    grab = func_a();      | {                        |
+--------------------------+--------------------------+
| }                        |                          |
+--------------------------+--------------------------+
|                          |     statements           |
+--------------------------+--------------------------+
| func_c()                 |     return(who());       |
+--------------------------+--------------------------+
| {                        | }                        |
+--------------------------+--------------------------+
|    int grip;             |                          |
+--------------------------+--------------------------+
|                          |                          |
+--------------------------+--------------------------+
|    statements            |                          |
+--------------------------+--------------------------+
|    grip =                |                          |
| func_ind(func_b);        |                          |
+--------------------------+--------------------------+
```

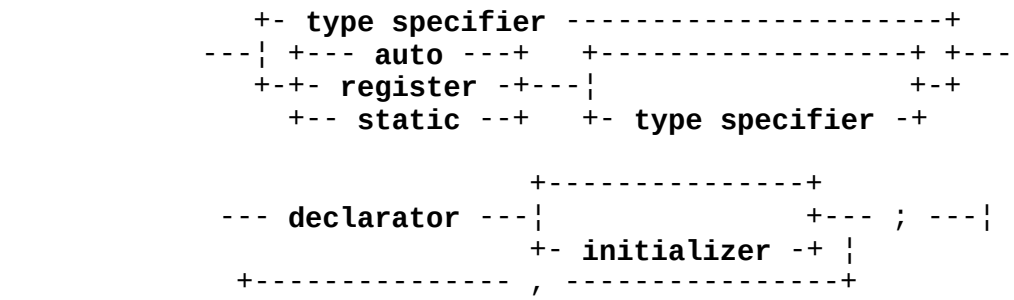


This example shows two compilation units side by side. In compilation unit A, the function **func_b** can reference **func_a** because it is in the same compilation unit. The function **func_p** in compilation unit B, however, will eventually generate a linker error, because **func_a** has been declared with the **static** storage attribute and so will never be visible.

The function **func_c** in compilation unit A, on the other hand, passes the address of **func_b** as an argument to the function **func_ind** in compilation unit B, and this is perfectly correct.

6.6 Block Structure

internal data definition



Although the C Language has only a single level of nesting for functions, compound statements can place a block structure within functions. A compound statement enclosed in braces { } can have declarations following the left brace that starts the compound statement. This process can nest indefinitely. If variables are declared within a block, the scope of such variables is the duration of that block. Variables declared outside the block may be referenced inside the block, with the restriction that any variables of the same name that might already have been declared in outer blocks are hidden from view of expressions in the inner block. This also applies to external variables and to formal arguments to the enclosing function.

Example:

```
demo() /* here is a demonstration function */
{
int  count;
int  c;

   ...some statements in the demo function...

while ((c = getchar())  != EOF)  {
         /*  redeclare 'count'  */
  int count;
    ...statements in the while loop...

  for (count = 0; count < 100; count ++) {
         /*  redeclare 'count' again  */
     int  count;
         ...statements in the for loop...
     }     /* end of the for loop */

     ...more statements in the while loop...
  }        /* end of the while loop */

  ...more statements in the demo function
}
```

In this example, the variable **count** is declared in the demo function. All statements in the body of the **demo** function that are outside of the **while** loop are using the **count** variable declared at the head of the function.

In the body of the while loop, the variable **count** is declared again. All the statements in the body of the while loop that are outside the for loop

are using the **count** variable declared in the **while** loop, and the **count** variable declared in the function body is hidden from those statements.

In the body of the **for** loop, the variable **count** is declared once more. All the statements in the body of the for loop are now using the **count** variable declared in the for loop, and both of the other **count** variables declared in the function body and in the body of the **while** loop are hidden from those statements in the body of the **for** loop.

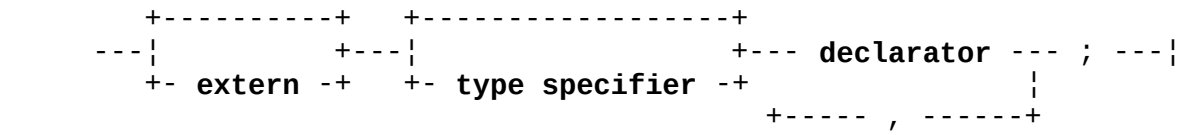
Example:

```
unsigned  pie;
char  broil;

demo(pie)
   double  pie;
{

    int  broil;

    . . . statements in the demo function
}
```

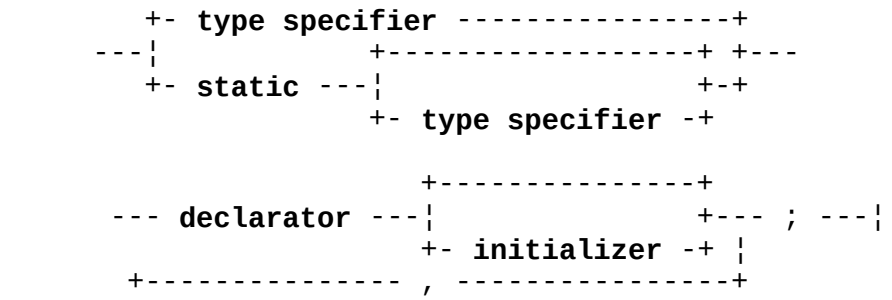
This example shows the way that external variables can be redeclared within the body of a function. The external variables **pie** and **broil** are declared one way outside of the **demo** function. But within the function **demo( )** the second declarations take precedence, and the original types of those variables are hidden from statements inside the **demo** function.

6.7 External and Static Variables

extern declaration



external data definition



By definition, variables declared outside of the scope of a function body are external variables. Variables declared within the body of functions, or within blocks within functions, normally have the **auto** storage class by default. This means that the lifetime of that variable is only the dynamic lifetime of the enclosing block.

It is however possible for the programmer to assign the **static** storage class to a local variable. When this is done, the variable remains defined across successive invo-
cations of the function or block but it remains private to that function or block.

Example:

```
demo()
{
  static int  init = 0;

  if (init == 0) { /* First time flag */
     ... do initialization code...
     init = 1;
  }
  ...normal sequence of events...
}
```

This example demonstrates one of the common uses of **static** variables, namely as a first-time flag to determine whether the **demo** function should perform some initialization work before it goes on to its main line. See "Storage Class Specifiers" in topic 3.5.1 for more details.

7.0 Chapter 7. Preprocessor Statements

Subtopics

7.1 CONTENTS

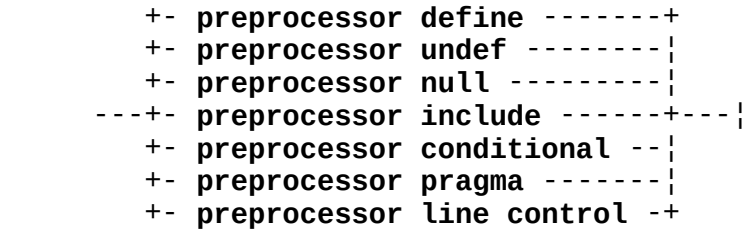
7.2 About This Chapter

This chapter describes the C preprocessor statements.

7.3 Preprocessor Statements

The preprocessor, rather than the compiler, interprets preprocessor statements. The ***preprocessor*** is a program that prepares C language programs for compilation. The **cc** command automatically sends programs through the preprocessor, then sends the output of the preprocessor through the compiler. The preprocessor recognizes the following types of statements:

preprocessor statement



Preprocessor statements enable you to:

Replace identifiers or strings in the current file with specified cod

Embed files within the current fil

Conditionally compile sections of the current fil

Change the line number of the next line of code and change the fil name of the current file.

C preprocessor directives may be included in C source code. They are interpreted by the C preprocessor command **cpp**. Directives in a source file apply to that source file and its included files only. Each directive applies only to the portion of the file following the directive. If a set of directives applies throughout a source program, all the source files must include the set.

The preprocessor handles such things as symbolic constant definition and macro expansion via the **#define** directive. The **#include** directive provides for the inclusion of other source text into the source text of the current compilation unit. The **#if** and **#ifdef** directives provide for conditional compilation.

7.4 Preprocessor Statement Format

Preprocessor statements begin with a number sign (#) character followed by a preprocessor keyword. A number sign (#) need not be the first character on the line, as long as it is preceded only by white space. Only space and tab characters can separate the number sign (#) and the preprocessor keyword. The remainder of the line can be filled with arguments to the preprocessor, C Language comments, and white space.

White space is a general term for blanks, tabs, new lines, formfeeds, and comments. Comments begin with the characters (/*) and end with the characters (*/).

Notes:

1. On the RT, the number sign (#) must be the first character on the line. White space is not allowed before the number sign.

2. For readability and portability, it is recommended that the number sign (#) be placed in column 1 for all preprocessor statements.

When a back slash (\ ) character appears as the last character in the preprocessor line, the preprocessor interprets the back slash (\ ) (and the following new-line character) and interprets the following line as a continuation of the current preprocessor line.

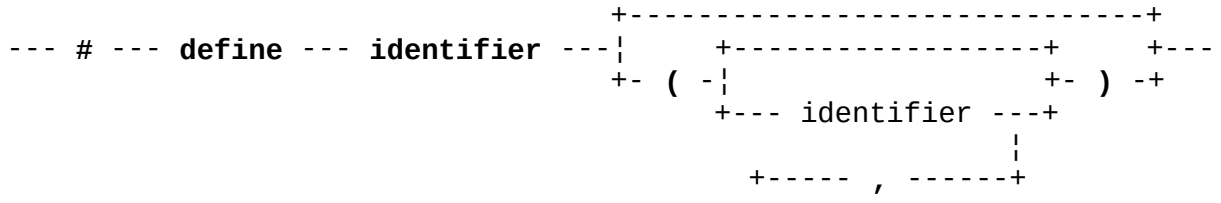
Preprocessor statements can appear any place in a program. They cannot, however, appear on the same line as C Language code that is not part of a preprocessor statement.

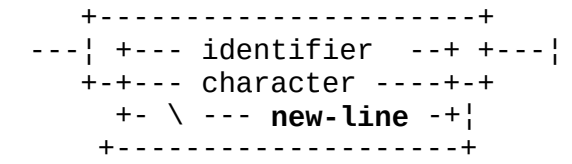
The effect of a preprocessor statement lasts until the end of the source file in which the statement appears.

7.5 #define

A ***define statement*** causes the preprocessor to replace an identifier or macro with specified code. A **define** statement has the form:

preprocessor define





The **define** statement can contain a simple macro definition or a complex macro definition.

Subtopics
7.5.1 Simple Macro Definition
7.5.2 Complex Macro Definition

7.5.1 Simple Macro Definition

A ***simple macro definition*** replaces a single identifier with another identifier or with a string of characters and identifiers. The following simple definition causes the preprocessor to replace all subsequent instances of the identifier **COUNT** with the constant **1000**:

```
#define COUNT 1000
```

This definition would cause the preprocessor to change the following statement (if the statement appeared after the previous definition and in the same file as the definition):

```
int arry[COUNT];
```

In the output of the preprocessor, the preceding statement would appear as:

```
int arry[1000];
```

The following definition references the previously defined identifier **COUNT**:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of **MAX_COUNT** with **COUNT + 100**, which the preprocessor then replaces with **1000 + 100**.

7.5.2 Complex Macro Definition

A ***complex macro definition*** receives parameters from a macro call, embeds these parameters in some replacement code, and substitutes the replacement code for the macro call. A complex definition is an identifier followed by a parenthesized parameter list and the replacement code. White space cannot separate the identifier (which is the name of the macro) and the parameter list. A comma (,) must separate each parameter.

A ***macro call***, like a function call, is an identifier followed by a list of arguments enclosed in parentheses ( ). Unlike a function call, white space cannot separate the identifier and the argument list. A comma must be used to separate the arguments.

The following line defines the macro **SUM** as having two parameters **a** and **b** and the replacement code **(a + b)**:

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appeared after the previous definition and in the same file as the definition):

```
c = SUM(x,y);
c = d * SUM(x,y);
```

In the output of the preprocessor, the preceding statement would appear as:

```
c = (x + y);
c = d * (x + y);
```

A macro call must have the same number of arguments as the corresponding macro definition has parameters.

In the macro call argument list, commas that appear as character constants, in string constants, or surrounded by parentheses do not separate arguments.

A definition is not required to specify replacement code. The following definition removes all instances of the word **static** from subsequent lines in the current file:

```
#define static
```

You can change the definition of a defined identifier or macro with a second preprocessor **define** statement or with a preprocessor **undef** statement.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro calls.

Examples:

The following program contains two macro definitions and a macro call that references both of the defined macros:

```
#define SQR(s) ( (s) * (s) )
#define PRNT(a,b) printf("value 1 = %d\n", a); \
printf("value 2 = %d\n", b)
```

```
main()
{
  int x = 2;
  int y = 3;

  PRNT(SQR(x),y);
}
```

After being interpreted by the preprocessor, the preceding program appears as follows:

```
# 1 "macro.c"



main()
{
  int x = 2;
  int y = 3;

  printf("value 1 = %d\n", ( (x) * (x) )); printf("value 2 = %d\n", y);
}
```

In the preceding example, the preprocessor inserted the line:

```
# 1 "macro.c"
```

The preprocessor inserted this line, which indicates the line number **1** and the name of the file in which the program was stored (**macro.c**), so that any line number references to the preprocessed code would match the line numbers in the original source code.

Execution of this program produces the following output:

```
value 1 = 4
value 2 = 3
```

7.6 #undef

An ***undef statement*** causes the preprocessor to end the scope of a preprocessor definition. An **undef** statement has the form:

preprocessor undef

--- # --- **undef** --- **identifier** ---¦

Examples:

The following statements define **BUFFER** and **SQR**:

```
#define BUFFER 512
#define SQR(x) (x) * (x)
```

The following statements nullify the preceding definitions:

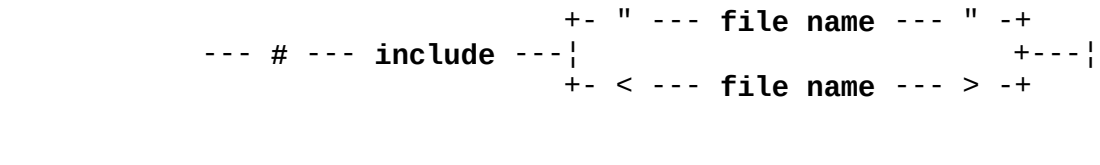
```
#undef BUFFER
#undef SQR
```

Occurrences of the identifiers **BUFFER** and **SQR** that appear following these **undef** statements are not substituted for the previously defined code.

7.7 #include

An ***include statement*** causes the preprocessor to replace the statement with the contents of the specified file. An **include** statement has the form:

preprocessor include



If the file name is enclosed in double quotation marks ("), the preprocessor searches the directory that contains the source file, then a standard or specified sequence of directories until it finds the specified file. For example:

```
#include "lib/payroll.h"
```

If the file name is enclosed in angle brackets (**<** and **>**), the preprocessor searches only the standard or specified directories for the specified file. For example:

```
#include <stdio.h>
```

If you have a number of definitions that several files use, you can place all these definitions in one file and include that file in each file that must know the definitions. For example, the following file **defs.h** contains several definitions and an inclusion of an additional file of definitions:

```
/* defs.h  */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "/u/david/defs.h"
```

You can embed the definitions that appear in **defs.h** with the following statement:

```
#include "defs.h"
```

The preprocessor would look for the file **defs.h** first in the directory that contains the source file. If not found there, the preprocessor would search a sequence of specified or standard places.

If the file name begins with the slash (**/**) character, the preprocessor searches only the specified directory for the file. For example:

```
#include "/u/david/defs.h"
```

The C Language does not define how you can specify a sequence of directories for the preprocessor to search. The command **cc**, however, recognizes the flag **-I***directory*, which enables you to specify a directory for the preprocessor to search before searching the standard directories.

Assume the file **pgm.c** contains the following statement:

```
#include "in_file"
```

If **pgm.c** were compiled using the following command:

```
cc -I melanie/include pgm.c
```

The preprocessor would search for the file **in_file** in the following directories:

The directory that contains the file **pgm.c**

The directory **melanie/include**

The standard sequence of directories

If instead, the file **pgm.c** contained the statement:

```
#include <in_file>
```

The preprocessor would search for the file **in_file** in the following directories:

The directory of **melanie/include**

The standard sequence of directories

7.8 Conditional Compilation

A ***preprocessor conditional compilation statement*** causes the preprocessor to insert specified code in the file depending on how a specified condition evaluates. A preprocessor conditional compilation statement spans several lines:

The condition specification lin

Lines containing code that the preprocessor inserts in the program i the condition evaluates to a nonzero value (optional)

A preprocessor **elif** statement (optional)

Lines containing code that the preprocessor inserts in the program i the condition in the **elif** line evaluates to 1 (one), or true (optional)

Note: The previous 2 steps may be repeated any number of times.

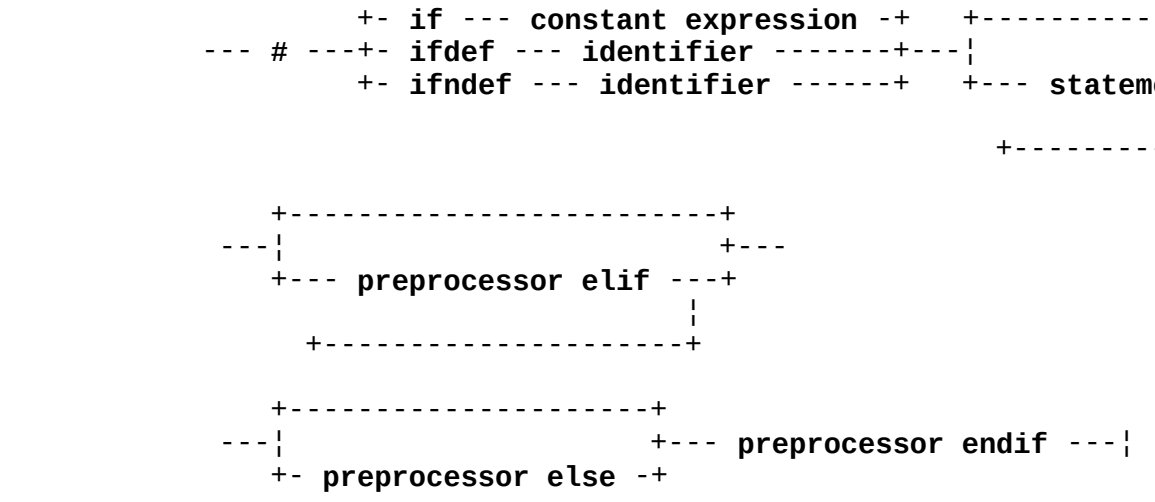
The **else** line (optional)

Lines containing code that the preprocessor inserts in the program i all previous conditions evaluate to 0 (zero) (optional)

The preprocessor **endif** statement.

A preprocessor conditional compilation statement has the form:

preprocessor conditional



A preprocessor conditional compilation statement can have one of three types of conditions: **if**, **ifdef**, and **ifndef**.

The following describes the usage of each:

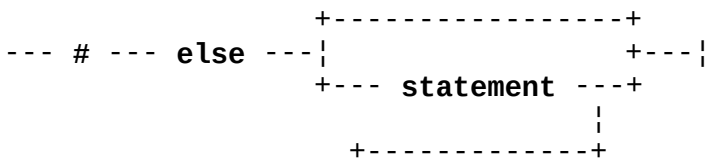
if Inserts the code that immediately follows the condition if the condition evaluates to a nonzero value.

ifdef Inserts the code that immediately follows the condition if the identifier specified in the condition is defined.

ifndef Inserts the code that immediately follows the condition if the identifier specified in the condition is not defined.

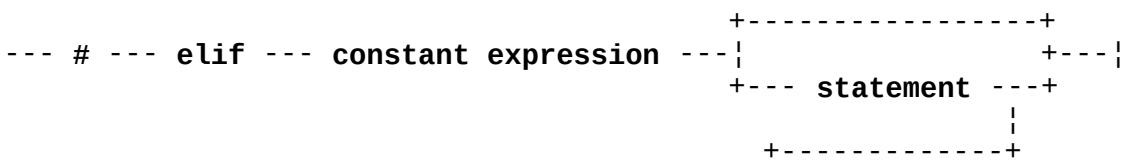
If the condition evaluates to 0 (zero), or false, and the conditional compilation statement contains a preprocessor **else** statement, the preprocessor inserts the lines that appear between the preprocessor **else** statement and the preprocessor **endif** statement. Otherwise, the preprocessor deletes these lines. The preprocessor **else** statement has the form:

preprocessor else



If the condition evaluates to 0 (zero), or false, and the conditional compilation statement contains a preprocessor **elif** statement, the constant expression following the **elif** is evaluated. If the condition evaluates to 1 (one), or true, the preprocess or inserts the lines that appear between the preprocessor **elif** statement and the next **elif**, **else**, or **endif** statement. Otherwise, the preprocessor deletes these lines. Each **elif** statement is evaluated in turn, until the constant expression of the **elif** evaluates to 1 (one), or true. Only one group of lines in a conditional compilation unit will be inserted in the program. If a preprocessor **else** statement is present, the lines which appear between it and the **endif** statement will only be inserted if the condition evaluates to 0 (zero), or false, and all **elif** statements, if any, evaluate to 0 (zero) or false. The preprocessor **elif** statement has the form:

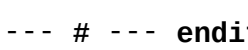
preprocessor elif



Note: The preprocessor **elif** is not supported on the RT.

The preprocessor **endif** statement ends the conditional compilation statement. The preprocessor **endif** statement has the form:

preprocessor endif



You can nest preprocessor conditional statements.

Subtopics
7.8.1 #if
7.8.2 #ifdef
7.8.3 #ifndef

7.8.1 #if

The if keyword must be followed by a constant expression. The constant expression cannot contain a **sizeof** expression, an enumeration constant, or a cast operator. For example:

```
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#endif
```

The constant expression can contain the keyword **defined**. This keyword can be used only with the preprocessor keyword if. The expression:

defined *identifier*

OR

defined(*identifier*)

evaluates to 1 if the *identifier* is defined in the preprocessor, otherwise to 0 (zero). For example:

```
#if defined (TEST1)|| defined(TEST2)
#   define   PHASE   1
#elif defined (TEST3)
#   define   PHASE   2
#else
#   define   PHASE   3
#endif
```

7.8.2 #ifdef

An identifier must follow the **ifdef** keyword. The following example defines **SIZEOF_INT** to be **32** if **I80386** is defined for the preprocessor. Otherwise, **SIZEOF_INT** is defined to be **16**.

```
#ifdef I80386
#   define SIZEOF_INT 32
#else
#   define SIZEOF_INT 16
#endif
```

7.8.3 #ifndef

An identifier must follow the **ifndef** keyword. The following example defines **SIZEOF_INT** to be **16** if **I80386** is not defined for the preprocessor. Otherwise, **SIZEOF_INT** is defined to be **32**.

```
#ifndef I80386
#   define SIZEOF_INT 16
#else
#   define SIZEOF_INT 32
#endif
```

The command **cc** recognizes the flag **-D***identifier*, which enables you to specify at compile time an identifier for the preprocessor to define. For example, the following command defines the identifier **I80386** in the file **pgm.c**:

```
cc -DI80386 pgm.c
```

Examples:

The following example shows how you can nest preprocessor conditional compilation statements:

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#      define MAX_PHASE 2
#   else
#      define MAX_PHASE 8
#   endif
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#endif
```

The following program contains preprocessor conditional compilation statements:

```
main()
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#endif

   }
}
```

7.9 #line

A ***line control statement*** causes the compiler to view the line number of the next source line as the specified number. A **line** statement has the form:

preprocessor line control

A file name specification enclosed in quotes can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the file specified by the preceding **line** control statement. If a **line** control statement does not precede the current statement, the compiler views the line as part of the current source file. The compiler recognizes the identifiers _ _**LINE**_ _ and _ _**FILE**_ _. _ _**LINE**_ _ evaluates to the current line number. The identifier _ _**FILE**_ _ evaluates to the current file name. Thus, the following statement prints an error message that contains the current line number and file name:

```
printf("Error on line %d in file %s.\n", _ _LINE_ _, _ _FILE_ _);
```

The preprocessor and other programs may produce **line** control statements (other than those specified in the file). For example, if the first line of a file is an **include** statement, the preprocessor inserts the specified file and a **line** control statement that sets the number of the line that follows the included code to *2*.

You can use **line** control statements to make the compiler provide more meaningful error messages. The following program uses **line** control statements to give each function an easily recognizable line number:

```
#include <stdio.h>
main()
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d\n",      _ _LINE_ _);
}

#line 200
func_2()
{
   printf("Func_2 - the current line number is %d\n",      _ _LINE_ _);
}
```

The preceding program produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

7.10 # (Null Statement)

The ***null statement*** performs no action. The **null** statement consists of a single number sign (#) on a line of its own.

preprocessor null

--- # ---¦

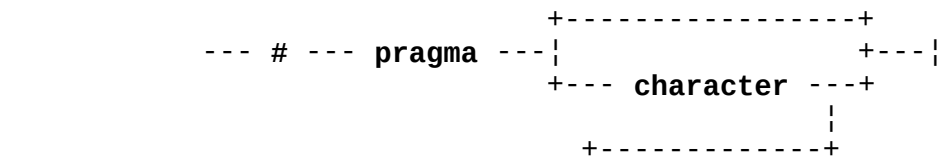
In the following example, if **MINVAL** is a defined macro name, no action is performed. If **MINVAL** is not a defined identifier, it is defined as the value 1.

```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

7.11 #pragma

A ***pragma*** is an implementation-defined instruction to the compiler. It has the general form given below, where ***character-sequence*** is a series of characters giving a specific compiler instruction and arguments, if any.

preprocessor pragma



The character-sequence on a pragma is not subject to macro substitutions. White-space characters (for example, blanks, tabs, and new lines) can appear between the number sign and the word **pragma**.

Note: The preprocessor **pragma** is not supported on the RT.

There are no pragmas currently defined for AIX C language.

7.12 Preprocessor Flags

AIX runs on several hardware platforms and offers several C compilers. By combining preprocessor flags and **#ifdef** statements, you can write a single C program which will be compiled differently according to the platform it is intended for or the compiler which is to be used.

Those blocks of code which make the program suitable for each hardware platform or compiler are preceded by a line of the following form:

```
#ifdef TAG
```

where **TAG** is one of the predefined symbols recognized by the preprocessor, such as i386. A program meant for several platforms or compilers contains a series of such blocks. Each is preceded with a different **#ifdef TAG** sequence.

The program is then compiled one or more times; each compilation specifies one of these **TAG**s on the command line. The **TAG** is used as a preprocessor flag, and is preceded by **-D**:

```
cc sourcefile -DTAG
```

This causes **TAG** to be defined within the program. The block of code preceded by **#ifdef TAG** is then compiled.

The **cc** command reads the file **/etc/cc.cfg** to determine which **cpp** flags to recognize. See **cc.cfg** in *AIX Technical Reference* for further information.

Special Characters

A

B

C