

# **TopSpeed® C**

**For IBM® Personal Computers and Compatibles**

## **Language Reference**

**TopSpeed Corporation**

Copyright© 1990-1991, by TopSpeed Corporation. All rights reserved.

TopSpeed® is a registered trademark of TopSpeed Corporation.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Printed in the United Kingdom.

10 9 8 7 6 5 4 3 2 1

# Contents

<b>CHAPTER 1 .....</b>	<b>9</b>
Introduction .....	9
Notation .....	10
<b>CHAPTER 2 .....</b>	<b>11</b>
Program structure.....	11
Introduction .....	11
Program Startup .....	11
Names .....	12
Scopes of Identifiers .....	13
Linkages of Identifiers .....	14
Name Spaces for Identifiers .....	15
Storage Durations of Objects .....	16
<b>CHAPTER 3 .....</b>	<b>17</b>
C Language Elements .....	17
Character Sets .....	17
Source and Target character sets .....	18
Trigraph Sequences .....	18
Escape Sequences .....	19
Numerical Representation of Characters .....	20
Lexical Elements .....	21
Keywords .....	21
Identifiers .....	22
Operators .....	23
Punctuators .....	23
Comments .....	23
<b>CHAPTER 4 .....</b>	<b>25</b>
Types and Constants .....	25
Types .....	25
Declarations .....	25

Object Types .....	26
Function Types .....	30
Incomplete Types .....	30
Qualified Types .....	30
Composite Type .....	31
Constants .....	31
Floating Constants .....	32
Integer Constants .....	33
Enumeration Constants .....	35
Character Constants .....	35
String Literals .....	37
<b>CHAPTER 5 .....</b>	<b>39</b>
Conversions .....	39
Introduction .....	39
Arithmetic Operands .....	39
Characters and Integers .....	39
Integral Promotions .....	40
Effects of conversions .....	41
Signed and Unsigned Integers .....	41
Effects of conversions .....	43
Floating and Integral Values .....	43
Floating Point Values .....	44
Usual Arithmetic Conversions .....	45
Conversions Involving Other Operands .....	46
Lvalues and Function Designators .....	46
void Values .....	47
Pointers .....	47
<b>CHAPTER 6 .....</b>	<b>48</b>
Declarations .....	48
Introduction .....	48
Storage-Class Specifiers .....	49
Type Specifiers .....	50

struct and union Specifiers .....	52
Bit Fields .....	53
Structure and Union Tags .....	55
Enumeration Specifiers .....	56
Type Qualifiers .....	58
Declarators .....	59
Pointer Declarators .....	60
Relative Pointers .....	61
Array Declarators .....	61
Function Declarators .....	63
Declarators with Special Keywords .....	66
Variable Declarations .....	66
Pointer Declarations .....	67
Function Declarations .....	68
The inline Keyword .....	68
Type Names .....	69
Type Definitions and Type Equivalence .....	70
Function Definitions .....	72
Initialization .....	74
Initializing Arrays, Structures and Unions .....	75
Initializing Multidimensional Arrays .....	77
Initializing Strings .....	79
Initializing Functions .....	79
External Definitions .....	80
External Object Definitions .....	81
<b>CHAPTER 7 .....</b>	<b>83</b>
Expressions .....	83
Introduction .....	83
Precedence of Operators .....	85
Primary Expressions .....	86
Postfix Operators .....	87
Array Subscripting .....	87
Function Calls .....	88

Default Argument Promotions .....	90
Structure and Union Members .....	90
Postfix Increment and Decrement Operators .....	93
Unary Operators .....	93
Prefix Increment and Decrement Operators .....	94
Address and Indirection Operators .....	95
Unary Arithmetic Operators .....	96
The sizeof Operator .....	97
Cast Operators .....	98
Multiplicative Operators .....	99
Additive Operators .....	100
Bitwise Shift Operators .....	102
Relational Operators .....	103
Equality Operators .....	104
Bitwise AND Operator .....	105
Bitwise Exclusive OR Operator .....	106
Bitwise Inclusive OR Operator .....	106
Logical AND Operator .....	106
Logical OR Operator .....	107
Conditional Operator .....	107
Assignment Operators .....	109
Simple Assignment .....	109
Compound Assignment .....	110
Comma Operator .....	111
Constant Expressions .....	111
<b>CHAPTER 8 .....</b>	<b>114</b>
Statements .....	114
Labeled Statements .....	114
Compound Statement, or Block .....	114
Expression and Null Statements .....	115
Jump Statements .....	116
The goto Statement .....	116
The break Statement .....	117

The return Statement .....	118
Selection Statements .....	118
The if Statement .....	119
The switch Statement .....	120
Iteration Statements .....	121
The while Statement .....	121
The do Statement .....	122
The for Statement .....	122
<b>CHAPTER 9 .....</b>	<b>124</b>
The Preprocessor .....	124
Introduction .....	124
Preprocessing Directives .....	125
Defining a Macro .....	126
Object-Like Macros .....	126
Function-Like Macros .....	126
Redefining a Macro Name .....	127
Scope of Macro Definitions .....	128
Macro Replacement .....	128
Argument Substitution .....	129
Rescanning and Further Replacement .....	129
The # Operator .....	131
The ## Operator .....	131
Conditional Inclusion .....	132
Evaluating Constant Expressions .....	134
Source File Inclusion .....	134
Line Control .....	135
Error Directive .....	136
Pragma Directive .....	136
Null Directive .....	136
Predefined Macro Names .....	136
Preprocessor Syntax Summary .....	137
<b>APPENDIX A .....</b>	<b>140</b>
References .....	140

<b>APPENDIX B .....</b>	<b>141</b>
Implementation Defined Behavior .....	141
1. Translation .....	141
2. Environment .....	141
3. Identifiers .....	142
4. Characters .....	142
5. Integers .....	143
6. Floating point.....	143
7. Arrays and pointers .....	144
8. Registers .....	144
9. Structures, Unions, Enumerations and Bit-fields .....	144
10. Qualifiers .....	145
11. Declarators .....	145
12. Statements .....	145
13. Preprocessing directives .....	145
Additional .....	146
<b>APPENDIX C .....</b>	<b>147</b>
Undefined Behavior .....	147
<b>APPENDIX D .....</b>	<b>149</b>
TopSpeed C Extensions. ....	149

# CHAPTER 1

## *Introduction*

This manual documents the C language as implemented in TopSpeed C. The implementation adheres to the February 1990 ANSI C standard and was the first C compiler for PC to be certified as such. The manual is intended as a guide for knowledgeable programmers; it is not an introduction or tutorial on the language. The TopSpeed C Language Tutorial, by K.N. King, provides a more elementary and extensive summary of the language. This book is included with the TopSpeed C package.

The C language is a general purpose programming language. It has been designed to allow efficient, perhaps machine specific, and optionally portable programs to be written.

In creating a standard for C, a major goal of the ANSI committee was preserving the traditional spirit of C. The rationale to the standard gives the following guidelines to this spirit:

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

This manual discusses the following:

- The syntax of the C language.
- The constraints on text written in the C language.
- The semantic rules for interpreting C programs.
- The restrictions and limits imposed by the ANSI language standard.
- The restrictions and limits imposed by the TopSpeed C implementation.

## Notation

---

The following notational conventions are used throughout this manual:

**Boldface** is used when a special term is discussed or defined.

*Italic* is used to emphasize a particular concept or point.

*Courier oblique* is used for C syntax listings.

*Courier* is used for C program examples and for C keywords and program fragments within text.

Thing<sub>opt</sub> is used to indicate that ‘Thing’ is optional.

# CHAPTER 2

## *Program structure*

### Introduction

---

A C program consists of one or more source files. These source files will contain C code and may contain material (e.g., macro definitions) for processing by the preprocessor. Each source file can be independently processed by the compiler.

A translation unit consists of a source file together with all the headers and source files included via the preprocessing directive `#include` however, the translation unit does not include any source lines skipped as a result of conditional inclusion preprocessing directives.

One or more translation units may be compiled separately and joined together with a linker to produce an executable program.

The process of turning C source text into an executable program occurs (conceptually) in a series of stages. These are enumerated in detail in Chapter 9 (section 9.1).

Note: Cross-references will use section numbers to identify the reference.

### Program Startup

---

The TopSpeed C compiler and linker create executable C programs from source files. Executable programs generated by TopSpeed C run in a hosted environment, i.e., they run under the control of an operation system.

An executable program must contain a function called `main` within a translation unit used to create the executable program. This function is called when the program starts executing. When a program starts up, all objects in static storage are initialized.

The main function must have been defined as:

```
int main(void) {...}
```

or

```
int main(int argc, char *argv[]){...}
```

As an extension, TopSpeed C also allows the following form for the main function:

<pre>int main(int argc, char *argv[],char *env[]){...}</pre>	
argc	(for argument count) represents the number of command line arguments when the program is called. This value is non-negative.
argv	(for argument value) represents the individual arguments.  If argc is greater than 0, then argv[0] through argv[argc-1] will contain pointers to strings; argv[argc] is a null pointer. argv[0] points to the program name.  If argc is greater than 1, subsequent array members (i.e., argv[1] through argv[argc-1]) represent additional command line arguments.
env	is a TopSpeed C extension, and represents the DOS (or OS/2) environment strings that have been defined.

## Names

---

In C, names, or identifiers, may be given to any of the following types of entities:

- objects (storage locations)
- types (a specification for how data in the storage location are to be interpreted)
- functions (collections of declarations and executable statements)
- macros (mechanisms for giving a meaningful name to a sequence of tokens)
- tags (a sort of second class type concept)
- typedefs (synonyms for existing types)
- labels (names for certain individual statements)

In C, there are rules for deciding each of the following:

- Which, if any, of these names are usable at a point in the translation unit (visibility).
- Whether it is possible to temporarily reuse a name (scope and name space).
- How identical names may relate to each other (linkage).
- The period of time (lifetime) during which storage is reserved for names of particular objects.

Details of these rules are discussed in the relevant sections.

## Scopes of Identifiers

There are constraints on identifiers, with respect to where in the program each identifier can be used (i.e., is visible). In particular, an identifier is visible only within a region of program text called the identifier's scope.

The scope of all identifiers is determined by the location of the identifier's declaration. There are four kinds of scope:

Block	If a declaration or type name appears inside a block (such as a function body), or if it is one of the parameters in a function definition, the identifier has block scope. This scope terminates at the <code>}</code> that closes the associated block.
File	If a declaration or type name appears outside any block or list of parameters, the declaration has file scope. This scope terminates at the end of the translation unit.
Function	The only identifier that has function scope is a label name. A label name is declared implicitly by its syntactic appearance, which must be within a function. The label name can be used in a <code>goto</code> statement anywhere in the function in which it appears.
Function prototype	A function prototype is a declaration of a function that also declares the types of the function's parameters. If a declaration or type name appears within the list of parameter declarations in a function prototype (i.e., not as part of a function definition), the identifier has function prototype scope. This scope terminates at the end of the function declarator.

Two identifiers have the same scope if, and only if, their scopes terminate at the same point.

- The scope of `struct`, `union` and `enum` tags begins just after the appearance of the tag in the type specifier.
- The scope of every enumeration constant begins just after its appearance in an enumerator list.
- The scope of any other identifier begins just after the completion of its declarator.

E.g.,

```
int i;           /* global i, file scope */ int f(int i);
/* function prototype scope */ void fs(i) /* introduces a new i, block
scope */ long i; /* ... */ { float i; /* introduces another i, block
scope */ /* ... */ } i;; /* label name, function scope */
```

## Linkages of Identifiers

---

Historically, implementations of C have offered a variety of solutions to the problem of linking, referencing, and defining occurrences of identifiers. There have also been several solutions to the linking of identifiers across translation units. The C standard has been influenced in this area by existing code. Linkage is the name given to the process whereby two or more lexically identical identifiers are made to refer to the same object or function.

There are three kinds of linkage:

External	<p>An identifier for an object or a function has external linkage if either of the following is true:</p> <ul style="list-style-type: none"><li>• If the declaration of an identifier contains the storage-class specifier <code>extern</code>, and there is no in-scope declaration, with file scope, of the same identifier. If there is such a declaration with file scope, the identifier with the <code>extern</code> specifier has the same linkage as the declaration with file scope.</li><li>• If there is no storage-class specifier associated with the (lexically) first declaration, with file scope, of the identifier in the translation unit.</li></ul> <p>In the collection of translation units that makes up a complete program, each instance of a particular identifier with external linkage denotes the same object or function.</p>
Internal	<p>An identifier (declared as an object or function) has internal linkage if the (lexically) first declaration, with file scope, of that identifier in the translation unit contains the storage-class specifier <code>static</code>.</p> <p>Within a translation unit, every instance of an identifier with internal linkage denotes the same object or function.</p>
None	<p>Identifiers with no linkage denote unique entities. The following identifiers have no linkage:</p> <ul style="list-style-type: none"><li>• An identifier declared to be anything other than an object or a function.</li><li>• An identifier declared to be a function parameter.</li><li>• An identifier declared to be an object inside a block without the storage-class specifier <code>extern</code>.</li></ul>

If an identifier declared with external linkage is used in an expression, then somewhere in the entire program there must be exactly one external definition for the identifier. This definition must be a declaration that has external linkage and must be one for which storage is allocated. An exception is when the identifier is part of the operand of a sizeof operator, which is evaluated at compile time.

If the same identifier appears with both internal and external linkage within a single translation unit, TopSpeed C gives the identifier internal linkage.

## Name Spaces for Identifiers

---

In C, the same identifier may be associated with up to four program entities at the same time. This overloading of an identifier is resolved by examining the syntactic context in which the identifier occurs. Thus, there are separate name spaces for various categories of identifiers.

- Label names. These are identifiable by the syntax of the label declaration and the goto keyword.
- Tags of structures, unions, and enumerations. These are identifiable by the struct, union or enum keyword preceding the identifier. There is only one name space for all tags.
- Members of structures or unions. Each structure or union has a separate name space for each member. This is distinguished by the type of the expression used to access the member via the . or -> operator.
- All other identifiers. These are called ordinary identifiers, and include variable and function names, enumeration constants, typedef names and function parameters.

It is illegal to have more than one identically named identifier in the same name space, in any one scope. The one exception to this rule is the forward declaration of tags (Chapter 6).

E.g.,

```
struct dup;      /* incomplete definition */ struct dup {    /* tag */
int dup;        /* member */ dup;                          /* ordinary identifier */ if (dup.dup
== 1) goto dup;   /* label */ {                             /* open a new scope */ int dup
= 1; struct dup { float dup; } x; dup :                      /* definition of label */
```

## Storage Durations of Objects

---

An object has a storage duration that determines the object's lifetime. There are two classes of storage duration:

- |           |  |
|-----------|--|
| static    | <p>An object has static storage duration if it is declared with either the storage class specifier <code>static</code> or with external or internal linkage.</p> <p>Such an object is created and initialized only once, prior to program startup. The object exists and retains its most recently-stored value throughout the execution of the entire program.</p>  |
| automatic | <p>An object has automatic storage duration if it is declared with no linkage and without the <code>static</code> storage class specifier.</p> <p>If an object is declared with automatic storage duration, then a new instance of that object is created on each normal entry into the block in which the object is declared. On entry to the function or block, TopSpeed C allocates the storage required for such an object. If an initialization for the object is specified, it is performed on each normal entry, but not if the block is entered by a jump to a label. Storage for the object is discarded when execution of the block ends in any way.</p> |

# CHAPTER 3

## *C Language Elements*

Various types of elements are distinguished in C programs. When processing each translation unit, a C compiler identifies elements that have significance in a C program, and interprets these elements.

### Character Sets

---

The source character set consists of the characters with which source files are created; the execution, or target, character set contains the characters that are interpreted in the execution environment. The following characters are defined in both the source and execution character sets:

The 52 upper-case and lower-case letters of the English alphabet:

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

The 10 decimal digits:

0 1 2 3 4 5 6 7 8 9

The 29 graphic characters:

! exclamation point	; semi-colon	" double quote
< less than	# hash, or number sign	> greater than
% percent	= equal	& ampersand
? question mark	' single quote	[ left bracket
( left parenthesis	] right bracket	) right parenthesis
/ slash	* asterisk	^ circumflex
+ plus	_ underscore	- minus
{ left brace	, comma	} right brace
. period	vertical bar	\ backslash
~ tilde	: colon	

In addition, the space character and control characters representing the following are defined in the source and execution character sets:

- horizontal tab (HT)
- vertical tab (VT)
- form feed (FF)
- the end-of-line indicator (newline character), which has the value ASCII 13.

Any other characters encountered in a source file generate illegal tokens - unless encountered in a preprocessing token, a character constant, a string literal, or a comment.

The space character, horizontal tab (HT), vertical tab (VT), and form feed (FF), are collectively known as white space characters.

## Source and Target character sets

---

C allows a distinction to be made between the character set used to write the program (source character set) and the character set used in the execution environment that runs the program (target character set). It is possible for the internal encoding of the character sets be different in the two cases. This could occur if a cross compiler were used to compile programs for another processor. It is assumed that users of TopSpeed C execute programs in the same hosted environment as they were compiled (source and target character sets are identical).

## Trigraph Sequences

---

Trigraph sequences allow the input of characters that may not appear on some keyboards. Use of trigraph sequences makes it possible to input characters not defined in the “ISO 646-1983” Invariant Code Set, which is a subset of the seven-bit ASCII code set.

The only valid trigraph sequences are represented by the three-character sequences in the following table. Such sequences are replaced with the corresponding single character A ? is altered only if it begins one of the trigraphs listed.

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

E.g, after replacement of the trigraph sequence ??' and ??/, the following source line

```
printf("??'???/??/");
```

becomes

```
printf("^?\\");
```

## Escape Sequences

---

```
escape-sequence:
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence

simple-escape-sequence: one of
    \'  \"  \?  \\
    \a  \b  \f  \n  \r  \t  \v

octal-escape-sequence:
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
    \x hexadecimal-digit
    hexadecimal-escape-sequence
    hexadecimal-digit
```

Escape sequences, or characters, are used in character and string constants to represent characters that might otherwise be impossible to enter directly into the source text.

The following alphabetic escape sequences produce the described effects on terminals, printers, etc. Most of the escape sequences affect the active position, which is generally the current location of the cursor on the screen. These sequences represent non-graphic characters in the target character set.

<code>\a</code> alert	Produces an audible or visible alert. The active position is not changed. (ASCII value: 07.)
<code>\b</code> backspace	Moves the active position one character to the left. (ASCII value: 010.)
<code>\f</code> form feed	Moves the active position to the initial position on the next logical page. (ASCII value: 014.)
<code>\n</code> new line	Moves the active position to the initial position of the next line. (ASCII value: 012.)
<code>\r</code> carriage return	Moves the active position to the initial position of the current line. (ASCII value: 015.)
<code>\t</code> horizontal tab	Moves the active position to the next horizontal tab position on the current line. (ASCII value: 011.)
<code>\v</code> vertical tab	Moves the active position to the initial position of the next vertical tab position. (ASCII value: 013.)

<code>\'</code>	Outputs a single quote (') (ASCII value: 047.)
<code>\"</code>	Outputs a double quote (") (ASCII value: 042.)
<code>\?</code>	Outputs a question mark (?) (ASCII value: 077.)
<code>\\</code>	Outputs a backslash (\) (ASCII value: 0134.)

## Numerical Representation of Characters

---

It is also possible to represent characters by specifying their bit pattern, either in octal or hexadecimal form. Such a specification consists of a backslash followed by a sequence of octal or hexadecimal digits. E.g.,

```
\015      /* newline character */\X61      /* 'a'          */
```

The construction `\0` is commonly used to represent the null character.

The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character. The numerical value of the octal integer specifies the value of the desired character.

The hexadecimal digits that follow the backslash and the letter `x`, or `X`, in a hexadecimal escape sequence are taken to be part of the construction of a single character. The numerical value of the hexadecimal integer specifies the value of the desired character.

In creating an octal or hexadecimal escape sequence the compiler will form the longest possible sequence allowed by the syntax.

- In the case of an octal sequence, the longest sequence will be three octal digits.
- In the case of a hexadecimal sequence, the sequence continues until a non-hexadecimal character is encountered.

If any other escape sequence is encountered a warning is given and the sequence of characters is replaced by a single backslash character.

**Note:** By default, `char` is signed `char` so the character constant `\xFF` has the value `-1`. If the pragma option `(uns_char=>on)` is specified, then `char` is treated as an unsigned `char`. In that case, the character constant `\xFF` has the value `255`.

Even though eight bits are used for values that have type `char`, the token `\x123` specifies a character constant containing only one character. To specify a character constant containing the two characters whose values are `0x12` and `3`, the sequence `\0223` must be used. The octal form must be used because a hexadecimal escape sequence is terminated only by a non-hexadecimal character.

A more complete discussion of character constants is given in Chapter 4.

## Lexical Elements

---

A token is the minimal lexical element of the language. Different tokens are accepted during the preprocessing and the compilation phases. Chapter 9 provides more details about the preprocessor.

The following tokens are recognized during compilation:

keyword	identifier	constant
string-literal	operator	punctuator

During compilation, white space and comments are ignored except when they serve as token separators. White space may appear within a token only as part of a character constant, string literal or header name. In such cases, the white space is significant.

If the input stream has been parsed into tokens up to a given character, the next token is the longest sequence of characters (not separated by white-space) that could constitute a token. Thus, the program fragment `x+++y` is parsed as `x ++ + y`, because `x` and `+` are both valid tokens. In contrast, `x+++y` is parsed as `x + ++ y`.

The program fragment `0xG` is parsed as an invalid token, even though a valid parse might be obtained if the identifier `x`, or `G`, previously defined as a macro name, were replaced by its macro definition (e.g. if `x` were defined as `+`). Similarly, the program fragment `0xF` is parsed as a (valid) hexadecimal constant token, regardless of whether either `x` or `F` is a macro name.

## Keywords

---

The following tokens are reserved in C. These are called keywords, and they may not be used as identifiers or otherwise redeclared in the program.

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

TopSpeed C has the following additional keywords, which are disabled by the ANSI keywords only `pragma` option (`ansi=>on`):

`cdecl` `pascal` `near` `farhuge` `interrupt` `inline`

## Identifiers

---

```
identifier:
    non-digit
    identifier non-digit
    identifier digit

non-digit: one of
    a b c d e f g h i j k l m
    n o p q r s t u v w x y z
    A B C D E F G H I J K L M
    N O P Q R S T U V W X Y Z

digit: one of
    0 1 2 3 4 5 6 7 8 9
```

In C, an identifier is a sequence of characters, which can denote any of the following:

- an object
- a function
- A tag or a member of a structure, union, or enumeration.
- A typedef name.
- A label name.
- A macro name.
- A macro parameter.

Macro names and parameters are identifiers only during the preprocessing phase.

Macro names are not considered further here, because prior to phase 5 of source text translation any occurrences of macro names in the source file will have been replaced by the token sequences that constitute their definitions.

Identifiers can contain

- lowercase and uppercase letters
- digits
- the underscore character

The first character in an identifier must be a letter or underscore.

The case of letters is significant in determining whether two identifiers are the same. E.g., `abc`, `ABC`, `AbC` are treated as different identifiers.

Names may be internal or external. There is no limit on the length of either internal or external names.

If identifier occurring in translation phases 7 and 8 has the same sequence of characters as a keyword, it will be processed as a keyword. It is possible to define a macro name

The following are all valid identifiers:

```
L001    _bufptr  WHILE
array   integer  Number_Of_Elements
```

## Operators

---

```
operator: one of
[ ] ( ) . -> + - ~ ! / % < > ^ |
? : = , # sizeof
++ Æ & * << >> <= >= == != && ||
*= /= %= += -= <<= >>= &= ^= |= ##
```

An operator specifies an operation to be performed on one or more operands. This operation yields a value. See Chapter 7 for more information about operators.

## Punctuators

---

```
punctuator: one of
[ ] ( ) { } * , : = ; ... #
```

A punctuator is a symbol that has independent syntactic and semantic significance in a particular context. Unlike an operator, however, a punctuator does not specify a value-producing operation to be performed. Depending on context, the same symbol may represent a punctuator, an operator, or part of an operator

## Comments

---

In C, a comment begins with `*` and ends with `*/`. All characters between these two delimiters are treated as part of the comment. Comments are recognized anywhere in a program except in the following places:

- within a character constant,
- within a string literal,
- within a comment

The contents of a comment are examined only to find any multibyte characters and the characters `*/` that terminate the comment. Because of this, comments ordinarily cannot be nested.

As an extension, TopSpeed C provides the nested comments option and the pragma option (`nest_cmt=>on`) to get around the restriction on nested comments.

Constants are the only type of tokens not discussed in this chapter. They are described in section 4.2.

# CHAPTER 4

## *Types and Constants*

### **Types**

---

Each object in C has an associated type, which is specified when the object is declared. A type defines the set of values that an object may take. Every type has a set of operations that may be performed on its values. There are three categories of types:

1. Types that designate data objects (object types).
2. Types that designate functions (function types).
3. Types that designate objects but lack the information needed to determine the storage allocated for the objects (incomplete types).

### **Declarations**

In C, a declaration associates an identifier with some entity (i.e., with a type, a variable or a function). Among other things, a declaration may specify

- the amount of storage required for an object.
- how an identifier should be interpreted in an expression, which specifies a type.
- where an identifier can be referenced  $\text{\AA}$  i.e., whether only within a module or throughout the program.
- whether an object continues to exist after its function finishes executing or whether the object must be created each time its function executes.
- whether the declaration is valid just for the function or for the entire module within which the declaration occurs.
- an initial value for an object, by including this value as part of the declaration.

A definition is a declaration that also causes storage to be reserved for the object or function named by an identifier.

## **Object Types**

There are several object types in C, as seen in the following sections. The first three types on the list `char`, integer, and floating point `double` are sometimes known as the simple, or basic, types.

### **Character Types**

The storage allocated for an object of type `char` is one byte. All values of the source character set (see Chapter 3) are positive.

A `char` may be signed or unsigned.

An unsigned `char` occupies the same amount of storage as a plain `char`, but can represent only nonnegative values.

A signed `char` occupies the same amount of storage as a plain `char`, but can represent both positive and negative character values. In TopSpeed C, a plain `char` is interpreted as a signed `char`.

Together, the `char`, signed `char`, and unsigned `char` types make up the character types.

### **Integer Types**

C's integer types are used to represent whole numbers. The range of valid values depends on which integer type is being used. Integer types may be signed or unsigned.

A plain `int` has a storage size of two bytes. The valid range for this type falls within the bounds `INT_MIN` and `INT_MAX`, whose values are given in the header file `<limits.h>`.

There are four kinds of signed integer types:

- signed `char` (one byte)
- short `int` (two bytes)
- `int` (two bytes)
- long `int` (four bytes)

The range of values for each signed type in the list depends on the type's size. The range for each signed type is a subrange of the values for the types below it in the list. E.g., values for signed `char` are a subrange of values for any other integral type, values for short `int` are a subrange of values for `int` and long `int`.

There is an unsigned type corresponding to every signed integer type. This unsigned type utilizes the same amount of storage (including the sign flag).

All nonnegative values that can be represented by a signed integer type can be represented by its corresponding unsigned type, and the representation of the same value is the same in each type.

Computations involving unsigned operands will not cause an overflow error. A result that cannot be represented by the resulting unsigned type is reduced as follows:

$$\text{reduced} = \text{large} \bmod N$$

where  $N$  is the number one greater than the largest value that can be represented by the unsigned type.

## Floating Point Types

Real numbers are represented using floating point types. Representations of real values will be approximations to the actual value, due to limitations imposed by representation in a finite storage area.

There are three floating point types:

- float (four bytes)
- double (eight bytes)
- long double (ten bytes)

The set of values of a float is a subset of the set of values of a double, which is a subset of the set of values of a long double.

See <float.h> for definitions of the size limits for floating point types in TopSpeed C.

## Enumeration Types

An enumeration consists of a collection of named integer constant values. Integer constants are associated with each name, beginning within 0 for the first name in an enumeration, and incrementing by 1. It is possible to specify other values to be associated with names.

Each distinct enumeration represents a different enumerated type.

E.g.,

```
enum card_suit    { club, diamond, heart, spade};  
enum rainbow      { red, orange, yellow = 10, green,  
                  blue = 0, indigo, violet = 10};
```

specifies two enumerations. In the first example, the integer value 0 is associated with the enumerated value club, 1 is associated with diamond, etc. In the second example, 0 is associated with red and with blue, because of the initialization for the latter value. Similarly, both orange and indigo are associated with 1. green is associated with 11, because of the initial value assigned to yellow.

## The void type

The void type is an incomplete type, and specifies an empty set of values.

## Derived Types

Derived types can be created using object, function, and incomplete types. Some derived types consist of aggregates of component types. An aggregate is constructed from the basic, enumerated, and incomplete types. The resulting type is a derived type. C's derived types are shown in the following list:

array type	<p>This is used to specify a contiguously allocated set of members of any one type of object, called the base, or element, type.</p> <p>An array type is derived from its base type, and is characterized by the base type and the number of elements. E.g.,</p> <p><code>double dbl_array[10];</code> declares an array type named <code>dbl_array</code>, with element type. This array has 10 elements, which are indexed from 0 through 9.</p>
structure type	<p>This is used to specify a sequentially allocated set of named elements, called members. Members may be of the same or different types. E.g.,</p> <pre>struct {     double real, imaginary; };</pre> <p>specifies a structure with two members, each of type double.</p> <p>A structure type will often have a tag associated with it. The tag provides a name for a particular structure type. E.g., the following example specifies a similar structure, but also includes a tag, <code>complex_nr</code>:</p> <pre>struct complex_nr {     double real, imaginary; };</pre>
union type	<p>This is used to specify an overlapping set of named elements, called members. Members may be of the same or different types. E.g.,</p>

```
union {
    char   ch_val;
    int    int_val;
    double dbl_val;
}
```

specifies a union which can have values for any one of three different members Æ `ch_val`, `int_val`, or `dbl_val` Æ at a given point in the program. A union type will often have a tag associated with it. The tag provides a name for a particular union type. E.g., the following example specifies a similar union, but also includes a tag,

```
nr_name:
    union nr_name {
        char   ch_val;
        int    int_val;
        double dbl_val;
    }
```

At a given point in the program, at most one member of a union has a value.

#### function type

This is used to specify a function. A function is characterized by its parameters and return type.

A function type is said to be derived from its return type. E.g.,

```
double sqrt ( double val);
```

represents a prototype declaration for a function named `sqrt`, which has a return type `double` and which has one parameter, also of type `double`. `sqrt` is said to be a “function returning double.”

#### pointer type

Objects having pointer type may point to functions, to objects of any type, and to incomplete types. A pointer type is said to “point to,” or “reference,” another entity, called the target for the pointer. If the target entity is of type `T`, the pointer type is said to be a “pointer to `T`.”

A pointer type represents an object whose value provides a means of accessing an entity of the pointer’s target type. E.g.,

```
struct complex_nr *dbl_ptr;
```

specifies a pointer named `dbl_ptr`. This pointer references (i.e., points to) a structure type with the tag `struct complex_nr`.

The methods for constructing derived types can be applied recursively, although there are restrictions. Such restrictions are mentioned in Chapter 6.

Various combinations of types have collective names that make it easier to refer to them. The following list summarizes these collective names:

<b>Collective Name</b>	<b>Included types</b>
character types	The types char, signed char and unsigned char.
integral types	Types char, int (both signed and unsigned), long int and short int (both signed and unsigned), and enumerations.
floating types	Types float, double and long double.
basic types	Character types, integral types, and floating types. Two basic types may both have the same size and representation; however, they are still distinct types.
arithmetic types	Integral and floating types.
scalar types	Arithmetic types and pointers.
aggregate types	Arrays and structures (but not unions). (Unions are not considered aggregate types because an object with this type can contain only one member at a time.)

### **Function Types**

This type represents a function with a specified return type. A function type is characterized by its parameters and return type.

A function definition creates an identifier of function type. When used in an expression, this identifier may cause a sequence of statements to be executed or the address of the code for those statements to be assigned or compared.

### **Incomplete Types**

An array of unknown size is an incomplete type. The type is completed, for an identifier of that type, by specifying a size in a later declaration, with internal or external linkage.

A structure or union of unknown content is an incomplete type. The type is completed, for all declarations of that type, by declaring the same structure, or union tag with its defining content later in the same scope.

### **Qualified Types**

The types described above are all unqualified types. Every unqualified type has three corresponding qualified types:

1. A version qualified with the keyword `const`.
2. A version qualified with the keyword `volatile`.
3. A version having both qualifiers.

The qualified and unqualified versions of a type are distinct types, but belong to the same type category. These versions have the same representation and storage requirements.

A type derived from a qualified type does not inherit any qualifiers. For example, an array derived from a qualified type is not a qualified type.

E.g.,

```
typedef const char CC;
CC start_char[10];      /* start_char is not const */
struct x{
    int    f1;
    const int f2;
} v;                    /* struct inherits the const */
                        /* from the contained field */
volatile const float vc[3];
```

## **Composite Type**

Two types are compatible if their types are the same, or if any of several conditions holds, as described in the discussion of compatible types (see Chapter 6 for more information).

Two compatible types can be used to construct a composite type. This type will be compatible with the original two types and will satisfy the following conditions:

- If one type is an array of known size, the composite type is an array of that size.
- If only one type is a function type with a parameter type list, the composite type is a function prototype with the parameter type list.
- If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

These rules are applied recursively to the types from which the two types are derived.

If an identifier with internal or external linkage has more than one declaration in the same scope, the type of the identifier is that of the composite type.

## **Constants**

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

Constants are used to represent specific values that will not change during program execution. The type of a constant is determined by its form and value.

## **Floating Constants**

```
floating-constant:
    fractional-constant exponent-partopt
floating-suffixopt
    digit-sequence exponent-part
floating-suffixopt

fractional-constant:
    digit-sequenceopt . digit-sequence
    digit-sequence

.exponent-part:
    e signopt digit-sequence
    E signopt digit-sequence

sign:
    +
    -

digit-sequence:
    digit
    digit-sequence digit

floating-suffix:
    f
    l
    F
    L
```

A floating constant consists of a value part, possibly followed by an exponent or a suffix or both. The value, or significant, part may include the following:

- Whole number part: a sequence of digits.
- Fractional part: a sequence of digits following a period.
- Period: if present, follows the whole number part and precedes the fractional part, if that is present.

Either the whole number part or the fractional part must be present.

The exponent part consists of e or E, followed by a digit sequence, which may be signed.

Either the period or the exponent part must be present.

Digit sequences are interpreted as decimal integers. The exponent indicates the power of 10 by which the value part is to be scaled.

A floating constant has type double, unless it is suffixed. A constant with f or F as a suffix has type float; a constant with l or L as a suffix it has type long double.

The following are all valid floating constants. The second example has type long double, and the fifth example has type float; the others have type double.

```
1.0e36    12e6L    .618    01e-9
34159f    42.      012e4
```

## **Integer Constants**

```
integer-constant:
    decimal-constant integer-suffixopt
    octal-constant integer-suffixopt
    hexadecimal-constant integer-suffixopt

decimal-constant:
    non-zero-digit
    decimal-constant digit

octal-constant:
    0
    octal-constant octal-digit

hexadecimal-constant:
    0x hexadecimal-digit
    0X hexadecimal-digit
    hexadecimal-constant hexadecimal-digit

non-zero-digit: one of
    1 2 3 4 5 6 7 8 9

octal-digit: one of
    0 1 2 3 4 5 6 7

hexadecimal-digit: one of
    0 1 2 3 4 5 6 7 8 9
    a b c d e f A B C D E F

integer-suffix:
    unsigned-suffix long-suffixopt
    long-suffix unsigned-suffixopt

unsigned-suffix:
    u
    U

long-suffix:
    l
    L
```

An integer constant is a sequence of one or more digits. It may not contain a period or exponent part.

The constant may have a suffix that specifies its type. If the suffix is u or U, the constant is unsigned; if the suffix is l or L, the constant is a long int.

The constant may also have a prefix that specifies the constant's base as octal, decimal, or hexadecimal. Octal or hexadecimal constants begin with zero (0); otherwise the constant is considered a decimal constant.

**Octal constant** Starts with the prefix 0 optionally followed by a sequence of the digits, 0 through 7 only. Such a value is computed base 8.

**Decimal constant** Starts with a nonzero digit optionally followed by a sequence of decimal digits. Such a value is computed base 10.

**Hexadecimal constant** Starts with the prefix 0x or 0X followed by a sequence of one or more decimal digits and the letters a (or A) through f (or F) representing values 10 through 15, respectively. Such a value is computed base 16.

The type of an integer constant is the first type from a candidate list that can accurately represent the constant. The candidate list depends on the constant's base and suffixes. The following table shows the candidate lists for the possible base and suffix combinations.

<b>Combination</b>	<b>Candidate list</b> (searched left to right)
Unsuffix decimal	int, long int, unsigned long int
Unsuffix octal	int, unsigned int, long int, unsigned long int
Unsuffix hexadecimal	int, unsigned int, long int, unsigned long int
U or u suffix only	unsigned int, unsigned long int
L or l suffix only	long int, unsigned long int
U or u and L or l suffixes	unsigned long int.

The following list shows some example values. An int is represented in 16 bits and a long is represented in 32 bits.

<b>Value</b>	<b>Type</b>
80000	long int
300000000	unsigned long int
040000	unsigned int
0100000	long int
0x0	int
0xfeed	unsigned int
2u	unsigned int
0xfacedU	unsigned long int

3000000000l	unsigned long int
034l	long int
0lu	unsigned long int
0x3ful	unsigned long int

**Note:** Negative values such as -38 actually consist of the unary minus applied to a positive constant. The calculation is done at compile time. However, the type of the constant is decided before the unary minus is applied. Thus -32768 is actually a long int because 32768 is a long int (see unsuffixed decimal constants above). However, -0x8000 is an unsigned int because 0x8000 is unsigned int.

## **Enumeration Constants**

```
enumeration-constant:
    identifier
```

An enumeration constant consists of a valid identifier. An identifier declared as an enumeration constant has type int. Numeric values are assigned to the enum constants starting at 0 and incrementing by 1. As described in Chapter 6, this sequence may be interrupted and the numeric values need not be unique. For example, given the following enumeration:

```
enum colour = {
    black, magenta, cyan, pink=0,
    yellow, orange, green
};
```

the identifiers have the following values:

```
black    = 0
magenta  = 1
cyan     = 2
pink     = 0
yellow   = 1
orange   = 2
green    = 3
```

## **Character Constants**

```
character-constant:
    'c-char-sequence'
    L 'c-char-sequence'
```

```
c-char-sequence:
    c-char
    c-char-sequence c-charc-
```

```
char:
    any character in the source character set
except
    the single-quote ', backslash \, or newline
    character escape-sequence
```

A character constant is generally a single character within single quotes, but may consist of multiple characters, also within single quotes. The character may be any character in the source character set (except a single quote, a backslash, or a newline), or it may be a character specified by an escape sequence.

A wide character is the same except that it is prefixed by the letter L. Wide characters have type `wchar_t`, which is defined as unsigned char in TopSpeed C.

Certain characters require special handling:

- A single quote must be represented by an escape sequence in which the quote is preceded by a backslash (`'\''`)
- A backslash must be prefixed with a backslash (`'\\'`).
- A double quote may be represented by the escape sequence (`'\"'`) or by using the character itself (i.e., `""`).
- or a question mark may be represented by the escape sequence (`'\?'`) or by using the character itself (i.e., `'?'`).

```
E.g.,
't' /* letter t */
'Q' /* letter Q */
'\'' /* single quote */
'\\' /* backslash */
'"' /* double quote */
'\'' /* double quote */
```

Escape sequences can also consist of octal or hexadecimal values prefixed by a backslash (octal) or a backslash and an x (hexadecimal). For example,

```
'\0' /* character 0 */
'\222' /* character 146 */
'\xFE' /* character 254 */
```

Finally, the following escape sequences can be used to represent nongraphic character constants:

### Escape Sequence    Effect

<code>\a</code> (ASCII 07)	(Alert) Produces an audible alert (a bell).
<code>\b</code> (ASCII 010)	(Backspace) Backs up one character.
<code>\f</code> (ASCII 014)	(Form feed) Moves to start of next logical page.
<code>\n</code> (ASCII 012)	(Newline) Moves to start of next line.
<code>\r</code> (ASCII 015)	(Carriage return) Moves to start of current line.
<code>\t</code> (ASCII 011)	(Horizontal tab) Moves to next horizontal tab.
<code>\v</code> (ASCII 013)	(Vertical tab) Moves to next vertical tab.

A character constant has type `int`. The value of a single-character constant is the numerical value of the character's representation, interpreted as an integer.

TopSpeed C also allows multiple-character constants. The number of characters allowed is equal to the number of bytes used to represent a long `int`. Thus, up to four characters are allowed in character constants for TopSpeed C.

The values of individual characters in a multiple-character constant are placed in bytes from left to right in the object being used to store the value. For example, the constant `\ttwd{ 'abcd' }` would be stored in four bytes as follows:

```
01100001 01100010 01100011 01100100
  97      98      99      100
  'a'     'b'     'c'     'd'
```

By default the type `char` is treated as a signed `char`. I.e., the high-order bit position of a single-character constant is treated as a sign bit. If the pragma option (`uns_char=>on/off`) is enabled then a `char` is treated as an unsigned `char`.

## String Literals

```
string-literal:
    "s-char-sequenceopt"
    L "s-char-sequenceopt"

s-char-sequence:
    s-char
    s-char-sequence s-char

s-char:
    any character in the source character set
except
    the double-quote " , backslash \ , or newline
    escape-sequence
```

A string literal is a sequence of zero or more characters within double quotes. E.g.,

```
"39*23""
a sample string literal"
""
```

are all string literals. The third one contains no characters.

A wide string literal is the same, except that it is prefixed by `L`. Each element in a wide string literal has type `wchar_t`, which is unsigned `char` in TopSpeed C.

The rules that apply to each character in a string literal are almost the same as those for a character constant. In a string literal, a single quote can be

represented by itself (') or by an escape sequence (\'). The double quote, on the other hand, must be represented by an escape sequence (\"). A string literal has static storage duration and is of type array of char. The string literal is initialized with the given characters. String literals that are adjacent tokens are concatenated into a single string literal during preprocessing. For example,

```
"One" "|" "Two"
```

is concatenated to the following string literal:

```
"One|Two"
```

A null character is appended onto the end of all string literals during compilation. Thus, the string literal,

```
"TopSpeed"
```

actually has nine characters.

A string literal containing a single character is not the same as a single-character constant. The value of the character constant is an int and is the value of the character; the value of the string literal is the letter itself plus the null character at the end. The string literal is of type array of char, which may be converted to a pointer to char. Thus, the value of the string literal is actually the location of the first character in the string.

All string literals are stored separately, even if they are identical. Thus, altering one literal does not affect any other, even if they are identical character by character.

Escape sequences are converted into single characters in the execution character set just prior to adjacent string literal concatenation.

The following are examples of string literals:

```
"Hello\n"  
"The bell\\a tolled"  
"\\0\\1\\2"  
"\\"
```

# CHAPTER 5

## Conversions

### Introduction

---

Many expressions involve different types of operands. In such cases, conversions will need to be made, to bring operands into conformity with each other. For many operators, such conversions will be done automatically.

For example, when you want to add a short int and a long int, the former will automatically be converted to a long int before the addition is carried out. This process is known as implicit conversion, since it occurs without any explicit instructions. Nevertheless, implicit conversion does follow explicit rules, which are summarized in the following pages.

It is also possible to produce an explicit conversion by using a cast operator, as described in Chapter 7 (see Chapter 7).

Regardless of whether a conversion is implicit or explicit, the conversion rules will always try to leave the operand's value unchanged. If it is necessary to change a value, high bits of quantities will be discarded when converting to smaller types.

The compiler will sign extend when converting from a smaller signed type to a larger one. That is, the sign bit from the smaller type will become the sign bit in the larger value.

### Arithmetic Operands

---

Conversions are very common when arithmetic operands are involved because such operands will often involve different types of values. For example, several types can be involved in expressions containing integer operators. Similarly, all these types as well as floating types can be involved in expressions using floating point operators.

#### **Characters and Integers**

In C, characters are treated as an integral type. As a result, the following types can all be used in any context in which an int or unsigned int is allowed:

- a char

- a short int
- an int bit-field
- the signed or unsigned forms of the preceding types
- an object of enumeration type

If an int can represent all values of the original type, the value is converted to an int. Otherwise, a value is converted to an unsigned int. This conversion is called integral promotion. The integral promotions preserve value, including sign. Whether a plain char is treated as signed depends on the .

### **Integral Promotions**

<b>Original type</b>	<b>Promoted type</b>
signed char	int
unsigned char	int
short	int
unsigned short	unsigned
int : N	int
unsigned:N	unsigned

Depending on the conversion, the representation of a value may change.

## Effects of conversions

---

From	To	Effect
signed char	short	sign extend
signed char	long	sign extend
signed char	unsigned char	no representation change: -128..-1 maps to 127..255
signed char	unsigned short	sign extend to short change to unsigned
signed char	unsigned long	sign extend to long
short	signed char	preserve low order byte (possible truncation)
short	unsigned char	preserve low order byte (possible truncation)
long	signed char	preserve low order byte (possible truncation)
long	unsigned char	preserve low order byte (possible truncation)
unsigned char	signed char	no representation change: 128..255 maps to -128..-1
unsigned char	short	zero extend
unsigned char	long	zero extend
unsigned char	unsigned short	zero extend
unsigned char	unsigned long	zero extend
unsigned short	signed char	preserve low order byte (possible truncation)
unsigned short	unsigned char	preserve low order byte (possible truncation)
unsigned long	signed char	preserve low order byte (possible truncation)
unsigned long	unsigned char	preserve low order byte (possible truncation)

### Signed and Unsigned Integers

The value of an unsigned integer is unchanged when it is converted to another integral type  $\mathbb{E}$  provided the value can be represented by the new type.

The value of a nonnegative signed integer is unchanged when it is converted to an unsigned integer of equal or greater size. Otherwise, the following two steps are taken:

1. If the unsigned integer is bigger, the signed integer is first promoted to a signed integer of the same size as the unsigned integer.
2. The value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.

In a two's-complement representation (which is used in TopSpeed C), the only change in the bit pattern is that the high-order bits are filled with copies of the sign bit if the unsigned integer is longer.

When an integer is demoted to a shorter unsigned integer, the result is the nonnegative remainder on division by  $1 + \text{maxrep}$ , where  $\text{maxrep}$  is the largest unsigned number that can be represented in the shorter type.

When an integer is demoted to a shorter signed integer, or when an unsigned integer is converted to a signed integer of equal length, the result depends on whether the original value can be represented within the range of the target type. If so, the value remains unchanged. If not, the least significant bits of the value are used to produce the result:

E.g., since `int` is a 16-bit type, when `0x3ffffff` (a signed long) is converted to a signed `int`, the result is `0xffff` ( $= -1$ ); when this value is converted to an unsigned `int`, the result is also `0xffff` ( $= 65535$ ).

**Effects of conversions**

<b>From</b>	<b>To</b>	<b>Effect</b>
short	long	sign extend
short	unsigned short	no representation change: -32768..-1 maps to 32768..65535
short	unsigned long	sign extend to long, change to unsigned long
long	short	preserve low order word (possible truncation)
long	unsigned short	preserve low order word (possible truncation)
long	unsigned long	no representation change: -2147483648..-1 maps to 2147483648..4294967295
unsigned short	short	no representation change: 32768..65535 maps to 32768..-1
-		
unsigned short	long	zero extend
unsigned short	unsigned long	zero extend
unsigned long	short	preserve low order word (possible truncation)
unsigned long	long	no representation change: 2147483648..4294967295 maps to -2147483648..-1
unsigned long	unsigned short	preserve low order word (possible truncation)

**Floating and Integral Values**

The value of the fractional part is discarded when an object of floating type is converted to integral type. If the value of the resulting integral part cannot be represented in the storage space assigned to the integral type the result is undefined.

When a value of integral type is converted to floating type, the value being converted will be within the range for the floating type, but will not necessarily be representable by an exact value. In that case, the result will be the nearest higher or lower value.

### Effects of conversions

From	To	Effect
any integral type	any floating point type	represent as the floating point value nearest to the original value
any floating type	any integral type	discard fractional part and change to point appropriate integral representation (as if converted to unsigned long first)

### Floating Point Values

The value of a float is unchanged when it is promoted to double or long double. Similarly, the value of a double is unchanged when it is promoted to long double.

When a double is demoted to float, or a long double to double or float, the result depends on whether the original value is within the range of representable values for the smaller type.

If the value being converted is outside this range, the result is undefined. If the value is within in the range, but cannot be represented exactly, the result is either the nearest higher or nearest lower value, chosen according to the IEEE Standard.

### Effects of conversions

From	To	Effect
float	double	no loss of precision
float	long double	no loss of precision
double	float	possible loss of precision and/or range
double	long double	no loss of precision
long double	float	possible loss of precision and/or range
long double	double	possible loss of precision and/or range

In TopSpeed C, double and long double are the same width. Therefore, no change in representation is involved in converting from double to long double or from long double to double.

## Usual Arithmetic Conversions

The term usual arithmetic conversions refers to the process of converting the operands of an operator to the same, defined type. The purpose of these conversions is to reduce the possible combinations of operands on which an operator might actually have to operate. The type produced after performing the usual arithmetic conversions is also the type of the resulting value for most operators. (See the discussions about these operators in Chapter 7 for exceptions to this rule.)

In general, conversions are to the higher operands, using a hierarchy with long double at the top, and int at the bottom.

IF	either operand has type long double, the other operand is converted to long double.
ELSE IF	either operand has type double, the other operand is converted to double.
ELSE IF	either operand has type float, the other operand is converted to float.
ELSE	the integral promotions are performed on both operands, that is
IF	either operand has type unsigned long int, the other operand is converted to unsigned long int.
ELSE IF	either operand has type long int, the other operand is converted to long int.
ELSE IF	either operand has type unsigned int, then the other operand is converted to unsigned int.
ELSE	both operands have type int.

```

/*EG*/          /* given the following definitions: */
long double   ld1 = 1.0;
double        d1 = 1.0;
float         f1 = 1.0;
unsigned long ul1 = 1;
long          l1 = 1;
unsigned      u1 = 1;
int           i1 = 1;

/* the following conversions will be made          */
/* when the specified expressions are evaluated: */
/* f1 is converted to long double;                  */
/* l1 is converted to double;                        */
/* quotient is converted to long double.            */
ld1 = f1 * ld1 + d1 / l1; /* i1 is converted to unsigned; */
/* product is converted to long;                    */
/* sum is converted to unsigned long                 */
ul1 = u1 * i1 + l1;

```

## Conversions Involving Other Operands

---

There are also conversion rules for operators that work with non-arithmetic types.

### **Lvalues and Function Designators**

An lvalue is an expression. It has an object type or an incomplete type (other than void) that designates an object.

The name lvalue comes originally from the assignment expression

A modifiable lvalue is an lvalue that:

- Does not have array type.
- Is not an incomplete type.
- Does not have a type declared with the const qualified type.
- Does not have any struct or union member declared with the const qualified type. This restriction includes, recursively, any member of all contained structures or unions.

An lvalue is converted to the value stored in the object, and is no longer an lvalue. The exceptions to this conversion are:

- When the lvalue is the operand for any of the following operators:
  - `sizeof`
  - unary `&`
  - `++`
  - `—`
- When the lvalue is the left operand of the `.` (dot) operator or of an assignment operator.

Array types are converted to pointer types. Thus, an lvalue that has type array of type is converted to an expression that has type pointer to type. This converted type points to the initial member of the array object and is not an lvalue.

This conversion does not occur when the lvalue is the operand of the `sizeof` operator, the `&` operator, or is a string literal used to initialize an array of characters.

A function designator is an expression that has function type. Such a function designator (with type function returning type) is converted to an expression that has type

pointer to function returning type. The exceptions to this are when the function designator is the operand of the sizeof operator or of the unary & operator.

### **void Values**

The value of a void expression may not be used in any way. This value is nonexistent, and implicit or explicit conversions may not be applied to such an expression. Conversions to void are allowed, however. The value of an expression of any other type is discarded if this value occurs in a context where a void expression is required.

### **Pointers**

A pointer to void may be converted to a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to void and back again. The result after these conversions will compare equal to the original pointer.

A null pointer constant is an integral constant expression with the value 0, or an expression evaluating to zero, cast to type void \*. If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. The null pointer constant is guaranteed to compare unequal to a pointer to any object or function.

A pointer to an unqualified type may be converted to a qualified form of that type. Both pointers will compare as equal.

# CHAPTER 6

## Declarations

### Introduction

---

In C, a declaration associates an identifier with some object (i.e., with a type, a variable, a function, or a tag).

```

declaration:
    declaration-specifiers
    init-declarator-listopt;

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

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

init-declarator:
    declarator
    declarator = initializer
  
```

A declaration specifies several things about the entity being declared.

type	An object's type determines the amount of storage required for an object or for a function's return value. By specifying the type, a declaration also specifies how to interpret an identifier in an expression.
linkage	An object's linkage determines where an identifier can be referenced Æ i.e., whether only within a translation unit or throughout the program. The linkage can have values extern or static.
lifetime	An object's lifetime, or storage duration, specifies whether an object continues to exist after its containing function finishes executing or whether the object must be created each time its function executes. The lifetime can have values auto and static.
scope	The location of a declaration determines whether the declaration is valid just for the function or for the entire module within which the declaration occurs. An object's scope can be file, block, function, or function prototype.
initial value	It is possible to specify an initial value for an object, by including this value as part of the object's declaration.

A definition is a declaration that also causes storage to be reserved for the object or function named by an identifier. A declaration with initialization is always a definition. An init-declarator-list is a sequence of declarators, separated by commas. Each of these declarators may have associated type information or an initializer, or both.

Linkage and storage duration make up the storage class for the declaration. Either a linkage or a type must be provided for an identifier. Thus, a declaration must specify either the storage class (i.e., the linkage) or the type  $\mathcal{A}$  or both  $\mathcal{A}$  for an identifier.

If an identifier has no linkage, there may be no more than one declaration of that identifier within a given scope and name space  $\mathcal{A}$  unless the second declaration is for a tag.

All declarations that refer to the same object or function within the same scope must specify compatible types.

A declaration must define an identifier (by means of a declarator) or it must define a struct or union tag or an enumeration type. Thus, the following two declarations are valid:

```
struct x;
    /* x is a tag */
enum    {e1, e2};
```

On the other hand, the following declaring is invalid, because no identifier is defined:

```
static int;
```

## Storage-Class Specifiers

---

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

The storage-class specifier determines the lifetime of a declared object. At most one storage-class specifier may be given in the declaration specifiers for a declaration.

auto	The object has local lifetime $\mathcal{A}$ i.e., within a function. This specifier is not allowed outside function definitions.
extern	The object has static storage duration. This specifier may appear on an object in a declaration at any point in a program. See Chapter 6 for a discussion of defining versus reference occurrences of this specifier.
register	This specifier follows the same rules as auto. Its purpose

is to give a hint from the programmer to the code generator that the object being declared is frequently used, and that an attempt should be made to keep the object's value in a register for as long as possible. It is not possible to obtain the address of an object with register storage class.

TopSpeed C does its own register allocation.

**static** The object has static storage duration. When applied to a function, this specifier also means that the function name is not visible outside of the translation unit. Objects declared with the static storage class are defining occurrences.

The following rules apply to a declaration without a storage-class specifier:

- For a function, the meaning is the same as if the storage-class specifier were `extern`.
- For an object declared inside a function or as one of its parameters, the declaration specifies automatic storage duration.
- For an object declared outside a function, the declaration is an external object definition or tentative definition. A tentative definition is a declaration for an object which has file scope, and which has no initializer and no storage-class specifier other than `static`.

typedefs are described separately, in chapter 6 of this guide.

## Type Specifiers

---

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

C provides a set of basic types and also makes it possible to create aggregates of one or more of these types.

An object's type is indicated by a type specifier. Objects may be declared as being of basic types (e.g., `int i`, which specifies an integer), or of aggregate types built from the basic types (e.g., `char s[3]`, which specifies an array of characters).

A type specifier may include qualifiers that determine the amount of storage allocated for the type or the range of values for the type. The following list describes the possible qualifiers.

long or short	At most one of these may be included in a type specifier. The qualifier can be present by itself (in which case int is implied), with the unsigned specifier, or with the int specifier. The long qualifier may also be used in conjunction with double.
signed or unsigned	Only one of these may be included in a type specifier (in conjunction with char, int, long, or short). These may also be specified alone, in which case int is implied.

If a declaration has no type specifier, the type is assumed to be int by default.

### **Basic types**

### **Abbreviations**

#### **integer**

#### **types**

signed char	char
signed short int	short
signed int	int
signed long int	long
unsigned char	char (or unsigned char)
unsigned short int	unsigned short
unsigned int	unsigned
unsigned long int	unsigned long

#### **floating types**

float	float
double	double
long double	long double

#### **other**

#### **types**

void	void
char is signed by default	

char is unsigned if the Pragma option (uns\_char=>on); is given.

In TopSpeed C the storage used by each type and its possible range of values is:

Type	Storage	Range of values
signed char	1 byte	-128..127
signed short	2 bytes	-32768..32767
signed int	2 bytes	-32768..32767
signed long	4 bytes	-2147483648..2147483647
unsigned char	1 byte	0..255
unsigned short int	2 bytes	0..65535
unsigned int	2 bytes	0..65535
unsigned long int	4 bytes	0..4294967295
float	4 bytes	0,± 1.17549435e-38..3.40282347e+38
double	8 bytes	0,± 2.225073858507201e-308.. 1.797693134862316e+308
long double	10 bytes	same as double

## struct and union Specifiers

---

```

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

struct-or-union:
    struct
    union

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

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

specifier-qualifier-list:
    type-specifier specifier-qualifier-listopt
    type-qualifier specifier-qualifier-listopt

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

struct-declarator:
    declarator
    declaratoropt : constant-expression

```

Both structs and unions are types that consist of a sequence of named members. These two types differ in the way the members are laid out.

A struct consists of an ordered sequence of named members. E.g., the following structure has three different members.

```
struct {
    int    first;
    char    second;
    float * third[5];
}
```

A union consists of an overlapping sequence of named members. A union is a struct all of whose members begin at the same address. The value of only one of the members can be stored in a union object at any time. E.g., the following union has one member at any given time.

```
union {
    int    ival;
    char    cval;
    float * apf[5];
} x;

x.ival = 0;

/* ... */ x.ival /* ... */

x.cval = 'A';
/* ... */ x.cval /* ... */

*x.apf[3] = 1.2;

/* .. */ *x.apf[3] /* ... */
```

The presence of a struct-declaration-list declares a new type when it appears in a struct-or-union-specifier. The type is incomplete until after the } that terminates the list.

A member of a struct or union may have any object type. However, a struct or union may not contain a member with function or incomplete type (such as the struct or union being defined).

## Bit Fields

The declaration of a member may be augmented with a field width specifying the number of bits (including an optional sign bit) to be allocated to that object. Such a member is called a bit-field. Bit-fields provide a means of storing more than one object in a word of storage.

The constant expression that specifies the width of a bit-field must have integral type, must be non-negative, and the value must not exceed the number of bits in an int. If the width is zero, the declaration may not have a declarator.

In the ANSI Standard, the value of a bit-field is interpreted as an integral type, and may have type int, unsigned int, or signed int. The high-order bit position of a plain int bit-field is treated as a sign bit. As an extension, TopSpeed C also allows char and unsigned char as bit-field types.

The unary & (address-of) operator may not be applied to a bit-field object. Consequently, there are no pointers to bit-fields.

TopSpeed C will allocate sufficient storage to hold a bit-field. If enough space remains, a bit-field that follows another bit-field will be packed into adjacent bits of the word. If insufficient space remains, space is allocated from the next word. Bit-fields can straddle byte boundaries, but not word boundaries.

A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field. Its purpose is to provide anonymous padding.

E.g.,

```
struct {  
    int rdy_flag : 2;  
    int          : 4;  
    int rcv_flag : 1;  
    int snd_flag : 1;}
```

As a special case, a bit-field with a width of 0 indicates that no further bit-fields are to be packed into the unit in which the previous bit-field, if any, was placed.

The first bit field encountered is the least significant in the byte (or word) in which the bit field is allocated. E.g.,

```
struct {  
    int aa : 2;  
    int bbb : 3;  
    int : 7;  
    int c : 1;  
    int ddd : 3;  
}
```

will pack as dddxxxxxxxxbbbaa in a 16-bit int, where the rightmost a is the least significant bit and x represents an anonymous bit.

Within a struct object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a struct object points to the object's initial member or, if it is a bit-field, to the unit in which it resides. Pointers to subsequent fields compare greater than earlier fields.

## **Structure and Union Tags**

A struct or union type can have an identifier associated with it. This identifier is known as a tag, and can be used elsewhere in the program to refer to the structure or union defined.

For structures and unions, a tag is included as an identifier in a specifier of the form

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

Such a specification declares the identifier to be the tag of the struct or union specified by the list. This is a complete type. Subsequent declarations in the same scope may then use the tag. The declaration list within curly brackets must be omitted in subsequent declarations involving the tag.

E.g.,

```
struct tree_node {
    double value;
    struct tree_node *left, *right;
};
```

declares a struct that contains a floating point value and two pointers to objects of the same type. Once this declaration has been given, the declaration:

```
struct tree_node t, *t_ptr;
```

defines `t` as an object of the specified type (i.e., `struct tree_node`) and `t_ptr` as a pointer to an object of the specified type. With these declarations, the expression `t_ptr->left` refers to the left `struct tree_node` pointer of `t_ptr`'s target object. The expression `t.right->value` refers to the value member of the target `struct tree_node` for `t`'s right pointer.

A specifier of the form:

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

is an incomplete type  $\text{\AE}$  if the specifier occurs before the declaration in which the structure or union's members are listed. This specifier declares a tag that may be used only when the size of an object of the specified type is not needed. Such a construct is useful when defining mutually referencing structures. E.g., two structures that point to each other can be defined by using a tag as part of an incomplete type specification.

```
struct s2;
/* specify tag in this scope, but*/
/* leave definition incomplete */
struct s1 { struct s2 *p1; /*...*/ };
struct s2 { struct s1 *p2; /*...*/ };
/* complete the type */
```

The size of pointers to structs is not dependent on the size of the target object. Hence `p1` can be declared to point to an incomplete type. The

```
struct s2;
```

statement declares a new tag, even if s2 was declared as a struct in an enclosing scope. Therefore, s2 in s1 will refer to the correct s2 below.

Without the first declaration of the tag s2 there is the danger that another s2 might already have been declared in an outer scope, thus causing p1 to refer to this tag and not the one following it. The third declaration completes the declaration of the new type.

To complete the type, another declaration of the tag in the same scope (but not in an enclosed block) must define the structure's members. If the definition were in an enclosed block, this would declare a new type known only within that block.

In the specifier

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

the declaration supersedes any prior declaration of the struct or union tag in an enclosing scope. It specifies a new type that is distinct from any other type with the same tag.

In contrast to the use of a tag, a struct or union specifier of the form:

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

specifies a distinct struct or union. This type can be referred to only by the declaration of which the type is a part. (Thus, the type is anonymous).

An alternative formulation of types uses the typedef mechanism to provide a synonym for an existing type. This synonym can then be used in other declarations. E.g.,

```
typedef struct tree_node TNODE;
struct tree_node {
    double value;
    TNODE *left, *right;
};
TNODE t, *t_ptr;
```

See Chapter 6 for more information about typedef.

## Enumeration Specifiers

---

```
enum-specifier:
    enum identifieropt { enumeration-list }
    enum identifier
enumeration-list:
    enumeration
    enumeration-list , enumeration

enumeration:
    enumeration-constant
    enumeration-constant = constant-expression
```

An enumeration consists of a list of named integer constant values. The identifiers in an enumeration list are declared as constants that have type `int` and may appear wherever an `int` is allowed. An `enum` specifier is a method of associating a list of identifiers with values of an object or type.

An `enum` specifier such as

```
enum { enumerator-list }
```

specifies an anonymous enumeration type. Such a type can be referenced only in the declaration in which the type is found.

A tag can be included in the declaration, to provide an identifier by which to refer to the enumeration. Such a declaration takes the following form:

```
enum identifier { enumerator-list }
```

Each enumeration constant is associated with a particular integer value. In the default case the `enum` constants are assigned values starting from zero in increments of 1.

It is possible to reset this sequence by explicitly assigning a particular integer value to one or more of the `enum` constants. This is done by specifying `=` and the integer value after the enumeration identifier. An expression used to give a value to an enumeration identifier in an `enum` list must be an integral constant expression of type `int`.

E.g.,

```
enum numbers {  
    zero, one, two, three  
};  
enum roman_enum {  
    I=1, V=5, X=V+V, L=V*X, C=L+L, D=C*V, M=X*C  
} date;
```

The values of the enumeration constants declared above are as follows: zero is 0, one is 1, etc. In the second declaration, X is 10, L is 50, etc.

The values of the constants need not be ordered or unique within the specifier. However, the identifiers of enumeration constants in the same scope must all be distinct from each other and from other identifiers declared in ordinary declarators.

The role of the identifier in the `enum`-specifier is to name a particular enumeration. This is analogous to the role of the tag in a `struct`-or-`union`-specifier.

In the examples, `numbers` and `roman_enum` name their respective enumerated types. As with structures and unions, these names can be used elsewhere in the program to refer to the two enumerated types.

## Type Qualifiers

---

In C, type qualifiers can be used to specify additional information to the compiler, about the objects being declared. C includes the `const` and `volatile` qualifiers. These identify their associated objects as unchanging and as subject to change, respectively.

The idea behind the `const` type qualifier is to provide the programmer with a method of telling the compiler that objects may be regarded as “read-only”. Objects declared with the `const` qualifier may only be given a value via an initializer. If the object is a parameter variable, the object may be initialized by the value being passed as an argument to the function of which the object is a parameter.

According to the ANSI standard, the behavior is undefined if an explicit cast is used to convert a pointer to a `const` object to a pointer to a type without the `const` attribute, and an attempt is made to modify the (constant) object by means of the pointer to non-`const`. This will work in TopSpeed C, but is not portable and, therefore, not recommended.

The `volatile` type qualifier is a way of telling the compiler that the values of certain objects may be modified through external means, e.g., memory mapped I/O.

An object whose type includes the `volatile` type qualifier may be modified in ways unknown to TopSpeed C. Expressions with a `volatile` element are not optimized; `volatile` variables are kept in storage.

According to the ANSI standard, the behavior is undefined if an explicit cast is used to convert a pointer to a `volatile` object to a pointer to a type without the `volatile` attribute, and an attempt is made to reference the object by means of the pointer to non-`volatile`. In TopSpeed C, the object is no longer considered to be `volatile`. A declaration may include `const` or `volatile` modifying the type of a declarator that has aggregate type or union type. If so, the type of each member of the aggregate or union inherits the modifying type qualifier. E.g., `a` is a `const` object in the following example, but `b` is not.

```
const struct s { int f1; } a;  
struct s b;
```

Both `a` and `b` are `const` objects in the following example.

```
typedef const struct s { int f1; } cs;  
cs a;  
cs b;
```

A type specifier list or type qualifier list may contain at most one instance of a particular type qualifier, either directly or via one or more typedefs.

An object declared as

```
extern volatile const data_monitor;
```

cannot be modified by the program, although it may be modified by hardware.

## Declarators

---

```

declarator:
    pointer
    direct-declaratoropt

direct-declarator:
    identifier
    ( declarator )
    direct-declarator
    [ constant-expressionopt ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifier-listopt )

pointer:
    * type-qualifier-listopt
    * type-qualifier-listopt pointer

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

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

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

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers
    abstract-declaratoropt

identifier-list:
    identifier
    identifier-list , identifier

```

A declarator is an identifier used to declare an array, pointer, or function returning type, or just a plain identifier. E.g., in the following example, `dval`, `ch`, and `cval` are all declarators.

```

double dval;
char ch;
struct complex_nr cval;

```

A declarator is not a complete declaration. A type specifier is also required to provide the base type. In the preceding listing, `double`, `char`, and `struct complex_nr` are the type specifiers needed to make three declarations

An identifier may be augmented by asterisks ( `*` ), brackets ( `[]` ) and parentheses ( `()` ) to indicate that the identifier represents a pointer, array,

or function, respectively. E.g, dptr, ch, and cfn are the declarators, and \*, [], and () are the augmentations in the following listing.

```
double          *dptr;
char            ch[];
struct complex_nr cfn( double, double);
```

Each declarator declares one identifier. When a construction of the same form as the declarator appears in an expression, the construction constitutes a function or object of the scope, storage duration, and type indicated by the declaration specifiers.

TopSpeed C provides several extensions in the way in which declarators work for various types and in the flexibility declarators have. These extensions are identified as such in the discussion.

When creating C programs, the non-ANSI keywords can be disabled by using the pragma option (ansi=>on); other language extensions can be disabled by using the pragma option (lang\_ext=>off).

## Pointer Declarators

---

A pointer is derived from another type, which represents the type of the variable to which the pointer is pointing Æ that is, the target type. A pointer is declared to point to a declared type. E.g., in the following declaration, P is the pointer object. The type of the identifier P is pointer to T

```
T * P
```

The \* indicates that P is a pointer, and not simply a variable of type T. E.g.,

```
double *dval/* pointer to double */
double *nresult (int);
                /* function returning pointer */
                /* to double */
```

This declaration may be augmented with one or more type-qualifiers, as in the following declaration form:

```
T * type-qualifier-listopt P
```

In this case, the type of the identifier P is type-qualifier pointer to T. E.g., if the type specifier list includes const, the identifier is a constant pointer. If the type specifier list includes volatile, the identifier is a volatile pointer.

E.g.,

```
double *volatile d_p;    /* volatile pointer */
int *const c_p;          /* const pointer */
const int *p_2_c;        /* pointer to const int */
```

The first declaration specifies d\_p as a volatile pointer. The target object for this is a double. The pointer's (but not the target's) value may be changed by things other than the program Æ because the volatile qualifier affects the pointer itself, not the target.

Similarly, `c_p` is a const pointer with an integer target. In this case, the target's value may change, but the pointer cannot be made to point to any other location.

On the other hand, the contents of the target for `p_2_c` may not be modified, because the target itself is a const int. In this case, the pointer (`p_2_c`) itself may be changed to point to another const int.

Two pointers are compatible if they are identically qualified and the types pointed to (that is, the target types) are compatible.

## **Relative Pointers**

Generally, a pointer consists of a segment value and an offset. For near pointers the segment value is implicit (DS for data pointers and CS for code pointers). As an extension to this, TopSpeed C has a special pointer type, called a relative pointer. A relative pointer is a near (16-bit) pointer; however, instead of having DS or CS as the segment value, you can specify the segment yourself in the pointer declarator. The syntax is:

```
pointer : '<' expression '>' '*'
        type-qualifier-list
```

The expression must be an lvalue.

E.g., the following listing declares `basep` to be a near pointer whose segment value is `seg`.

```
unsigned seg = 0xB800;
char <seg> * basep;
```

When `basep` is dereferenced, `seg` will be used as the segment value.

This has the advantage that, if `basep` points to a pool of data, this pool can be moved in memory. You just have to update `seg` to indicate the new location of the pool. This works because the value of `seg` is calculated every time `basep` is dereferenced.

## **Array Declarators**

An array is a construct for grouping together one or more objects of the same type. Arrays generally occur in declarations as a method of obtaining a defined amount of storage. An array declaration specifies a type for the array's elements, and a declarator to specify a name for the array. The declarator will be augmented with `[]`, possibly containing an integral value that specifies the array's size. E.g.,

```
double dvals [35];
/* 35-element array of double */
struct cplex cvals [5];
/* 5-element array of struct cplex */
float fvals [];
```

```
/* array of unspecified # of float */
```

The constant expression specifying the size of an array must have integral type and must be a constant value greater than zero.

In the following contexts an array bound may be omitted:

- When an array is being declared as a parameter of a function.
- When the array declaration has storage-class specifier `extern` and the definition that actually allocates storage is given elsewhere.
- In a multi-dimensional array declaration, only the first size may be omitted.
- When the declarator is followed by initialization. In such a case, the size is determined by the number of initializers supplied.
- In an incomplete type, such as `int a[]` In such a case, the size must be given later in a declaration.

If the size is not present, the array type is an incomplete type. If `A` has the form:

```
T A[constant-expressionopt ]
```

the identifier `A` has type array of `T`. E.g.,

```
float fa[6];
```

declares a 6-element array of float values.

In the following declarations,

```
extern int *a;extern int b[];
```

The first declares `a` as a pointer to `int`, the second declares `b` as an incomplete array of `int`.

When several array of specifications are adjacent, a multi-dimensional array is declared.

```
double two_d[2][3];
```

declares a two-dimensional array of double. The declaration

```
double *ptr_array[5];
```

specifies a 5-element array of pointers to double values.

For two arrays to be compatible their element types must be compatible.

If both array sizes are given they must be equal.

## Function Declarators

A function declaration declares an identifier as a function. The declaration also specifies the type of the function's parameters and the function's return type. The return type can be anything except a function type or an array type.

In addition to function calls, assignment and comparison operations may be performed on function expressions.

The identifier, *F*, has the type function returning *T*, if *F* has either of the following two forms:

```
T F(parameter-type-list) T F(identifier-listopt )
```

The following statement

```
double frexp(double value, int *exp);
```

declares `frexp` as a function that returns a double. The function takes two arguments, a double and a pointer to int. The parameter names `value` and `exp` serve no purpose other than documentation. (These names would be required in a function definition, however.)

The following statements illustrate additional declarations:

```
/* function with double parameter, returning int */
int ifn(double);
/* function with int parameter, returning double */
double dfn(int);
/* function with unspecified parameters, */
/* returning pointer to int */
int *pfm();
/* function with no parameters, returning pointer */
/* to structure */
struct cplex *spfn( void);
/* function with int and other */
/* parameters, returning a double */
double ddfn(int, ...);
```

A parameter type list specifies the types of the function's parameters. The list also may declare identifiers for the parameters. If the parameter list terminates with an ellipsis (`...`), this implies that zero or more arguments of unknown type may follow.

**Note:** when using ellipses, there must be at least one argument defined before the ellipsis. E.g., the first of the following declarations is not allowed, but the second is:

```
int f1 (...);
/* illegal, no first argument */
int f2 (char *, ...); /* legal */
```

The specifier `void` is used to specify a function having no parameters. E.g.,

```
int f(void)
```

declares a function that has no parameters and that returns an int.

Note: The declaration,

```
int f1();
```

does not declare a function having no parameters. For compatibility with pre-standard C usage, this declaration specifies a function returning int, with no information known about the parameters.

The only storage-class specifier allowed in a parameter declaration is register. However, in a parameter declaration that is not part of a function definition (e.g., in a function prototype declarator), this storage-class specifier is ignored.

A parameter declaration may include type qualifiers. E.g., in the following,

```
FILE *freopen(const char *, const char *, FILE *);
```

freopen is declared as a function returning a pointer to FILE. This function takes three arguments: two constant strings (const char \*) and a pointer to FILE. (FILE is a typedef name declared in the library. See <stdio.h>. .)

The declaration,

```
int (*p_to_fn) (void);
```

declares p\_to\_fn as a pointer to a function. This function takes no parameters and returns an int. Similarly, the statement

```
void (*signal (int, void (*)(int) ) ) (int);
```

declares signal as a function that returns a pointer to function. This target function for signal returns void, and has one int parameter.

signal itself has two parameters:

- an int
- a pointer to function that returns void.

signal's second parameter points to a function that has a single int parameter.

The following is an equivalent declaration:

```
typedef void (*ptr_to_fn_void)(int);  
ptr_to_fn_void signal(int, ptr_to_fn_void);
```

Two function types in the same scope must declare compatible return types in order for the functions to be compatible. Each parameter type list, if present, must agree in the number of parameters and in the use of the ellipsis terminator. Corresponding parameters must have compatible types.

Also, if the declarator in a function definition contains an identifier list, the type of a parameter identifier must also agree with its corresponding

prototype parameter. Any default argument promotions (see Chapter 7) are made before enforcing this agreement. Parameters declared with function or array type are converted to pointer type.

The following will work in ANSI code because the char parameter is promoted to int.

```
void absf1(int);
void absf1(a)char a;
{
}
```

The following will work in TopSpeed C, not because of direct type equivalence, but because int and char parameters occupy the same amount of stack space:

```
void absf1(char);

void absf1(a)
char a;
/* this argument is promoted to int;the result is
   not compatible with the previous declaration.*/
{
}
```

TopSpeed C will give a warning when this code is encountered, because the code is not ANSI.

Certain cases containing both float and double declarations will cause undefined results since the objects use different amounts of space on the stack when passed as parameters. E.g., the following code will result in an error, because float and double occupy different amounts of storage:

```
double fn1(float);

double fn1(f)
float f;
/* this argument is promoted to double */
{
}
```

On the other hand, the following example is valid, because f is promoted to double.

```
double fn1(double);

double fn1(f)
float f;
/* this argument is promoted to */
/* double, compatible with the */
/* original declaration.*/
{
}
```

An identifier list in a function declarator that is not part of a function definition must be empty.

If a function type has a parameter type list and a matching function declarator (not a definition) has an empty identifier list, the parameter type list may not contain an ellipsis terminator. Also the type of each parameter must be compatible with the type resulting from the default argument promotions being applied.

## Declarators with Special Keywords

---

TopSpeed C provides several extensions in the form of a variety of non-ANSI C keywords. These may be used to modify declarations of variables, pointers and functions. (Such keywords can be disabled by the ANSI keywords only option `/a` i.e., `/a.`)

`cdecl, far, huge, interrupt, near, pascal, inline`

Here we only focus on the syntax of how to specify the modifiers in declarations. For a complete discussion of the effect of the various modifiers see the discussions of mixed model programming (`near`, `far`, `huge`), mixed language programming (`cdecl`, `pascal`) and interrupt functions in the TopSpeed C User's Manual.

Generally, a modifier affects the entity immediately to the right. More than one modifier can modify the same entity.

- The `near`, `far` and `pascal` keywords can modify variable declarations.
- The `near`, `far`, `huge`, `interrupt`, `cdecl` and `pascal` keywords can modify pointer declarations.
- Function declarations can be modified with the `near`, `far`, `interrupt`, `cdecl` and `pascal` modifiers.
- The `inline` modifier specifies that the code is to be expanded inline. This keyword can be used only in function declarations. It does not make sense to use this modifier with any of the others (i.e., `interrupt`, `near`, etc.).
- The effect of all the keywords (except `huge`) can also be achieved using pragmas. See the TopSpeed C Developer's Guide for more information about pragmas.

### Variable Declarations

The `near` and `far` keywords can be used to override the memory model. The `near` modifier forces a variable to be allocated in the default data segment regardless of the memory model used. The `far` keyword forces a variable to be allocated in a segment of its own.

E.g., if you are using the small model, you can do the following to make sure a variable is not allocated in the default data segment:

```
int far store[10000]; /* far modifies store */
```

If you are using one of the large data models, the `near` keyword can force a variable to be allocated in the default data segment, even if the size of the variable is larger than the data threshold. E.g.,

```
int near table[18000]; /* near modifies table */
```

A modifier affects only the variable immediately to the modifier's right. E.g., in the following declaration only `x` is declared to be `far`.

```
long far x, y;
```

The `pascal` modifier can also be used in variable declarations. E.g.,

```
extern unsigned pascal x;
```

The effect is that `x` has (traditional) Pascal linkage. I.e., the external name is in upper case (since Pascal is not case sensitive), no `_` is put in front of `x`.

**Note:** the `near` and `far` modifiers cannot be used for declarations of function parameters and local variables (unless declared as `static` or `extern`) because these are always allocated on the run-time stack.

## Pointer Declarations

There are two kinds of pointers, `near` (16-bit) pointers and `far` (32-bit) pointers. The size of the pointers depends on which memory model is being used.

The default pointer size can be modified by the `near` and `far` modifiers:

```
char near *npc;
```

This declares `npc` to be a `near` pointer Æ i.e., it points to a character in the default data segment. Note that `near` modifies the `*`.

```
int far *fpi;
```

Here `fpi` is a `far` pointer to an `int`

```
long * near * nppl;
```

The `near` keyword modifies the `*` to the right, so `nppl` is a `near` pointer to a pointer to a `long`.

```
int far * near fpi;
```

The pointer variable `fpi` itself is allocated in the default data segment because of the `near` modifier; it is a `far` pointer to an `int`.

```
int far *fp1, far *fp2, near *np;
```

Several declarators with modifiers can be given in the same declaration, `fp1` and `fp2` are `far` pointers, `np` is a `near` pointer.

```
long huge *hp;
```

This declares hp to be a pointer to a huge object Æ i.e., an object that can have a size large than 64K. Such an object must be allocated using the malloc function.

Declarations of pointers to functions can have the following additional modifiers: cdecl, pascal and interrupt. There are no huge pointers to functions.

```
int (near *npf)(char);
```

This declares npf to be a near pointer to a function. E.g., the call

```
(*npf)(a)
```

is done with a near function call.

```
void (interrupt *pif)();
```

This declares pif to be a pointer to an interrupt function

```
int (far pascal *pf)();
```

Here the \* has two modifiers far and pascal. The order is not significant. pf is a far pointer to a function using the Pascal calling convention.

## **Function Declarations**

Function declarations can be modified by the near, far, cdecl, pascal, inline and interrupt keywords.

```
unsigned near func();
```

This declares func to be a near function Æ i.e., it is called with a near call.

```
double pascal far func2(int);
```

func2 is a far function; it has Pascal calling convention and linkage. The order of the pascal and far modifiers are insignificant.

```
int cdecl func3();
```

This declares func as using the traditional C calling convention,.

```
void interrupt far MyIntr();
```

This declares MyIntr to be an interrupt function,.

## **The inline Keyword**

The following listing shows how to use the inline modifier.

```
static int inline max(a,c,d)
{
    if (a > c)
        if (a > d)
            return a;
        else
            return d;
```

```

    else if (c > d)
        return c;
    else return d;
}

```

## Type Names

---

```

type-name:
    type-specifier-list abstract-declaratoropt

```

```

abstract-declarator:
    pointer
    pointeropt
    direct-abstract-declarator

```

```

direct-abstract-declarator:
    ( abstract-declarator )
    direct-abstract-declaratoropt
    constant-expressionopt ]
    direct-abstract-declaratoropt
    ( parameter-type-listopt )

```

In C, a type name is required in two situations:

- In cast expressions.
- When applying sizeof to a type.

A type name is not a typedef name (see below). Rather, a type name is basically a declaration that omits the identifier being declared.

Empty parentheses in a type name are interpreted as a function with no parameter specification (not as redundant parentheses).

Several cases are shown in the following list of constructions and what they name illustrate.

```
char
```

a char.

```
unsigned *
```

pointer to unsigned int.

```
double [10]
```

an array of 10 doubles.

```
int *[5]
```

an array of five pointers to int.

```
float *()
```

a function (with no parameter specification) returning a pointer to type float.

```
long (*)(*)[20]
```

a pointer to an array of 20 longs.

```
void (*)(int)
```

a pointer to function with int parameter and no return value.

```
char (*const [])(double, ...)
```

an array with an unspecified number of constant pointers to functions, each with one parameter that has type double and an unspecified number of other parameters. These functions each return a char.

## Type Definitions and Type Equivalence

---

```
typedef-name:  
    identifier
```

A typedef declaration does not introduce a new type. Rather, such a declaration provides a synonym for an existing type. E.g.,

```
typedef enum {club, diamond, heart, spade} suit;  
typedef struct linknode {  
    double info;  
    struct linknode *next;  
} *NODE;  
typedef int distance;  
  
suit    hand [];  
NODE    tempnode;  
        /* tempnode is a pointer to linkmode */  
distance boston_ny;
```

The first typedef statement specifies suit as a synonym for the enumerated type specified by the list of values. Similarly, the second statement specifies a NODE as a pointer to struct linknode. Finally, the third statement specifies distance as a synonym for int.

After the typedefs have been specified, it is possible to declare objects of these “types,” as shown in the remaining statements in the listing.

A typedef name shares the same name space as other identifiers declared in ordinary declarators.

An identifier declared as a typedef may be redeclared in an inner block. A member of a struct or union may also have the same name as a typedef declared in the same or an enclosing scope. The type specifier may not be omitted in either of these cases.

Two type specifier lists are the same if they contain the same set of type specifiers (including tags). For this purpose, two structs, union, or enumerations are considered to have different types if either or both do not have a tag.

Two types have compatible type if their types are the same. Two types are the same if any of the following holds:

They have the same ordered set of type specifier lists and abstract declarators (including the types of function parameters), either directly or via typedefs.

Both are arrays that have the same type, and one or the other array declarator has no size specification.

They are two functions that return the same type. In addition, if one function declarator has no parameter specification and the other has a parameter type list, then it must specify a fixed number of parameters, none of which is affected by the default argument promotions.

E.g., after

```
typedef int range, domain(char);
typedef struct { double re, im; } complex;
```

the declarations:

```
range step;
extern domain *set;
complex z, *zp, za[3];
```

are all valid declarations.

The type of `step` is `int`, because of the first typedef. The object `step` is considered to have exactly the same type as any other `int` object.

Similarly, `domain` is defined in the same typedef statement as a function that has a `char` argument and that returns an `int`. As a result, the type of `set` is pointer to such a function.

`z` is a struct having two members, because of the second typedef. Similarly, `zp` is a pointer to such a struct, and `za` is an array with three elements of the complex structure type.

After the declarations:

```
typedef struct s1 { int f; } t1, *tp1;
typedef struct s2 { int f; } t2, *tp2;
```

type `t1` and the target type for `tp1` are compatible with each other and to the type `struct s1`. These types are distinct from the types `struct s2` and `t2` and the target type for `tp2`. Because they are structures, these types are also different from an `int`.

If an identifier is declared more than once in the same scope (e.g., by declarations of an object with internal or external linkage or of a function), all the declarations must specify the same type. This is also the case if the same identifier is declared with external linkage in separate translation units or in disjoint scopes within the same translation unit.

## Function Definitions

---

```
function-definition:  
decl-specifiersopt declarator decl-listopt  
compound-statement
```

The actual instructions carried out by a function are specified in a function definition. In addition to a function declaration (which contains a return type and an optional definition of arguments), a function definition contains a block of zero or more declarations and statements. The return type of a function must be void or an object type other than array. The return type also cannot be a function.

The declaration of a function must be a function declarator. That is, the function name must have a function type associated with it. This is specified by the declarator portion of the function definition. There are two methods of specifying the names and types of the parameters. The two styles of declaration may not be mixed syntactically.

1. In the older (pre-ANSI) style, a list of identifiers appears in the function parameter list. The types of these parameters then follow outside the list. If the type of a parameter is not given int is assumed. E.g.,

```
f(a,b)int a; char b;
```

The declaration specifies a function that returns an int. The return type is implicit. The function has two parameters, a and b. These are identified after the function declarator as being of type int and char, respectively.

2. In the newer style, both the names and types of the parameters are given in the parameter list. E.g.,

```
f(int a, char b)
```

Specifies a function with the same (implicit) return type and parameter types as in the preceding style.

There is also a special case that is used to denote a function definition with no parameters:

```
f(void)
```

The difference between the two definition styles is that, in the old style, the default argument promotions occur. These are described in section 5.3. Another difference is that the second definition style serves as a prototype declaration that forces conversion of arguments for subsequent calls to the function.

Programmers should use the second form for several reasons. In addition to providing greater protection for the programmer, this prototype form is intended to replace the older form. The ANSI committee intends for the first form to become obsolete.

An identifier declared as a typedef name may not be redeclared as a parameter in the identifier list (i.e., older) syntax. Only the register storage class specifier can be used with parameters in a function definition. Parameter declarations may not contain an initializer.

On entry to a function, the value of the argument expression is converted to the type of the parameter, as if this value were assigned to the parameter. Arguments that are array expressions and function designators are converted to pointers before the call. Thus, a declaration of a parameter as array of type will be adjusted to pointer to type, and a declaration of a parameter as function will be adjusted to pointer to function, as in lvalues and function designators (see Chapter 5).

Each parameter for a function is treated as having automatic storage duration.

The parameter's identifier is an lvalue. In effect, a parameter is declared at the head of the compound statement that constitutes the function body. As a result, the parameter may not be redeclared in the function body (except in an enclosed block).

To pass one function to another, use a format such as the following:

```
int a(void); /* function declaration */
/*...*/
b(a);       /* function call in which function */
            /* a is passed to function b */
```

**Note** that a must be declared explicitly as its appearance in the expression b(a) was not followed by a left parenthesis (.

The definition of b might be:

```
b(int (*funcp)(void))
{
    /*...*/ (*funcp)() /* or funcp() ... */
}
```

or, equivalently:

```
b(int func(void))
{
    /*...*/ func() /* or (*func)() ... */
}
```

## Initialization

An object can be given an initial value when the object is declared. When this is done, the value is assigned immediately after storage is allocated to the object. This means that objects of static storage class are initialized once at program startup. Objects of non-static storage class are initialized every time they start a new lifetime.

The initializer for a scalar must be a single expression, optionally enclosed in braces. The value of the expression becomes the initial value of the object. The same type constraints and conversions apply as for simple assignment.

A brace-enclosed initializer for a union object initializes the member that appears first in the declaration list of the union type.

There are several constraints on initialization:

- There must be no more initializers in an initializer list than there are objects to be initialized.
- The type of the entity to be initialized must be an object type or an array of unknown size.
- All the expressions in an initializer for an object that has static storage duration, or in an initializer list for an object that has aggregate or union type must be constant expressions.
- If the declaration of an identifier has block scope, and the identifier has internal or external linkage, the declaration may not include an initializer.

static objects are assigned their value prior to executing the first statement in main. The initialization of auto objects makes it appear as if assignments were made as the declarations were processed upon entering the block containing them.

If an object with static storage duration is not initialized explicitly, it is initialized implicitly. In this implicit initialization, every member that has arithmetic type is assigned 0 and every member that has pointer type is assigned the null pointer constant.

The value of an auto object declared without initialization is undefined until it is explicitly assigned a value.

Initialization will not occur if a goto statement jumps to a label in a compound statement containing declarations with initializers.

E.g.,

```
#include <stdio.h>

static i = 0;
```

```

float f = 1.23;

void vf()
{
    long   lv = i + 1;
           /* static i will be used; e.g., lv = 1 */
    int     s_f = sizeof(f);
    double  dval = f * 2.0;
           /* The following declaration would be invalid,
              because the initial value is not a constant
              expression. static unsigned ls = 14 + i;
              */
    int     i;
    for (i = 0; i < 4; i++) {
        char af = 'Z' + i;
           /* valid because not static */
           /* some code here */
    }
    /* some code here */
}

void wf()
{
    int     i = 5;
    long   lv = i + 1;
           /* local i will be used; i.e., lv = 6 */
    for (i = 0; i < 4; i++) {
        char bf = 'a' - i;
           /* some code here */
    }
    /* some code here */
}

main ()
{
    for ( i = 0; i < 3; i++) {
        vf ();
        wf ();
        f *= 2.0;
    }
}

```

In function `vf`, the initial value of `lv` depends on the static `i` declared at the start of the listing. In function `wf` the initial value of `lv` depends on the `i` declared in function `wf`.

Each time through the loop in the main program, the `lv` in `vf` gets a different value, as does the `dval`. In contrast, `lv` in `wf` gets the same initial value each time through the main loop.

## Initializing Arrays, Structures and Unions

```

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

```

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

It is possible to initialize entire arrays, structures and unions when declaring them. This is accomplished by specifying a list of values between braces. The values must be constant expressions. E.g.,

```
int ivals[3] = {10, 20, 30};  
  
struct{  
    double re, im;  
} svals = {2.5, 18.7};
```

If there are fewer initializers in a list than there are members of an aggregate, the remainder of the aggregate will be initialized implicitly. The values used will be the same as those assigned to objects of static duration, defined without any initial value. E.g., the last three elements in the following array are initialized to 0.0.

```
double dvals [5] = {2.3, 4.9};
```

An array of characters may be initialized by a string literal, optionally enclosed in braces. Successive characters of the string literal initialize corresponding elements of the array. The null character terminating the string literal is also assigned to the array if there is room or if the array is of unknown size. E.g., the following initialization makes the `_word` a 6-element array, including the terminating null character.

```
char the_word [] = "hello";
```

If the aggregate contains members that are aggregates, or if the first member of a union is an aggregate, the rules apply recursively to the subaggregates or the contained unions.

If the initializer of a subaggregate or contained union begins with a left brace, the succeeding initializers initialize the members of the subaggregate or the first member of the contained union. Otherwise Æ that is, if the subaggregate is not being initialized by using braces Æ only enough initializers from the list are taken to process the members of the first subaggregate or the first member of the contained union. Any remaining initializers are left to initialize the next member of the aggregate of which the current subaggregate or contained union is a part.

If an incomplete array is initialized, the array's size is determined by the number of initializers provided. Therefore, at the end of its initializer list, the array is no longer an incomplete type.

The declaration

```
int p[] = { 2, 16, 256 };
```

defines and initializes `p` as a one-dimensional array object that has three members. The declarations

```
struct cnum {
```

```

        double re, im;
        int info [3];
    } cvals = { 3.9, 6.7, {1, 5, 9}};
              /* compare initialization of info */
              /* for cvals and more_cvals      */
    struct cnum more_cvals = {6.3, 5.8, 2, 12, 36};

```

initialize two basic type members of a structure and also an aggregate type that is a member of the structure.

## **Initializing Multidimensional Arrays**

Arrays whose elements are themselves aggregates (e.g., structures or arrays) can be initialized cell-by-cell or by initializing an entire subaggregate at a time. Braces may be used to delimit subaggregates. For example, if the initializer for cell [0][0] of a two-dimensional array begins with a left brace, then subsequent values are used to initialize the first subaggregate. The declaration,

```

double matrix[4][3] = {
{ 1,  2.0,  3 },
{ 1.0,  4,  9 },
{ 1,      8, 81.0 },
};

```

defines and initializes a two-dimensional array. The declaration contains three groups of bracketed values. Each of these initializes one row of matrix. Thus, the elements of matrix[0] are assigned the values:

```

matrix[0][0] 1
matrix[0][1] 2.0
matrix[0][2] 3

```

Similarly, matrix[2] is assigned the values contained in the third group so that matrix[2][2] would have the value 81.0 after initialization.

Because the initializer ends early, the elements of matrix[3] are initialized with zeroes. Precisely the same effect could have been achieved by

```

double matrix[4][3] = {
1.0, 2, 3, 1, 4, 9.0, 1, 8.0, 81
};

```

In this case, the initializer for matrix[0] does not begin with a left brace, so as many items from the list are used as are needed to initialize the entire first row (in this case, 3). Subsequent groups of three are taken, successively, for matrix[1] and matrix[2].

Braces can also be used to limit the number of values used for initializing a particular subaggregate. E.g., the declaration

```

double matrix[4][3] = {
{ 3 }, { 5 }, { 7.0 }, { 11 }
};

```

initializes the first element of each row of matrix (that is, the first column of the matrix) as specified. The remaining elements in each row are (implicitly) initialized with zeros.

The `fi` array in the following declaration

```
struct {
    float f;
    int i;
} fi[] = { 1.2, 3, { 7 }, 0.4, 14 };
```

has 3 elements. The initializers for this array are inconsistently bracketed. As a result field `fi[1].i` is implicitly assigned the value 0, since the initialization for this middle element is incomplete.

The following three declarations all accomplish the same thing Æ initializing a three-dimensional array of integers.

```
int cube[4][3][2] = {
    { 1 },
    { 2, 3, 4, 5, 6 },
    { 7, 8, 9 }
};
int cube[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 4, 5, 6, 0,
    7, 8, 9
};
int cube[4][3][2] = {
    {
        { 1 },
    },
    {
        { 2, 3 }, { 4, 5 }, { 6 }
    },
    {
        { 7, 8 },
        { 9 },
    }
};
```

Each of these declarations accomplishes the following assignments:

```
cube[0][0][0] is 1
cube[1][0][0] and cube[1][0][1] are 2 and 3, respectively
cube[1][1][0] and cube[1][1][1] are 4 and 5, respectively
cube[1][2][0] is 6, and cube[1][2][1] is (implicitly) 0
cube[2][0][0] and cube[2][0][1] are 7 and 8, respectively
cube[2][1][0] is 9, and cube[2][1][1] is (implicitly) 0
```

the remaining elements are implicitly initialized to 0.

In the first declaration, the groupings are as described because the initializer for `cube[0][0][0]` does not begin with a left brace. As a result, up to six items from the current list may be used Æ to initialize the entire matrix `cube[0]`. There is only one value, however, so the values for the remaining five members are initialized with zero.

Similarly, neither the initializers for `cube[1][0][0]` nor `cube[2][0][0]` begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates.

## Initializing Strings

In C, a string is an array of characters. Such arrays can be initialized using string literals. The following declaration

```
char with_null[] = "hello",
    too_short_for_null[5] = "hello";
```

defines and initializes two such objects. `with_null` is defined as an array of `char`, but no size is specified. The string's size is determined implicitly through the initialization. The terminating null string is automatically added to such an array, so that the resulting length of `with_null` is 6.

The array `too_short_for_null` will not contain a terminating null, since there is not enough room to fit both the initial string value and the terminating null character.

The preceding declaration is equivalent to

```
char with_null[] = { 'h', 'e', 'l', 'l', 'o', '\0' },
    too_short_for_null[] = { 'h', 'e', 'l', 'l', 'o' };
```

The contents of such arrays are modifiable. The declaration

```
char *v = "abc";
```

defines a character pointer, `v`. This pointer is initialized to point to a character array object. The members of this character array are initialized with a string literal (four elements, including a trailing `'\0'`).

## Initializing Functions

---

Normally you cannot initialize functions in C. However, TopSpeed C provides an extension that allows this for the purpose of defining in-line machine code.

### Example 1

The in-line machine code facility is useful if you wish to write code that cannot be expressed in C Æ e.g., the 8086 out instruction. Pragmas can be used to control the generation an expansion of in-line code.

```
#pragma save, call(reg_param => (dx,ax),
    inline => on)
/* the port has to be in dx, the byte */
/* to output must be in ax */

static void __outportb__(
    int port,
```

```

    unsigned char byte) =
{
    0xEE,
    /* out dx,al */
};
#pragma restore

```

## Example 2

Another reason to use in-line machine code is for writing optimal code. In the example below, calls to the `strlen` function will be expanded in-line instead of actually calling the function. This is because of the `inline` pragma. Such a strategy improves the runtime speed.

```

#pragma save, call(reg_param=>(di),
reg_saved=>(bx,si,ds),inline=>on)
/* this pragma sets up the correct calling */
/* sequence for the function below, small model */

static unsigned strlen(const char *s) =
{
    0x1E,      /* push ds */
    0x07,      /* pop  es */
    0xB9, 0xFF, 0xFF, /* mov  cx,-1
    */ 0x2A, 0xC0, /* sub  al,al
    */ 0xF2, 0xAE, /* repnz; scasb*/
    0xF7, 0xD1, /* not  cx */
    0x8B, 0xC1, /* mov  ax,cx*/
    0x48,      /* dec  ax */
};
#pragma restore

```

## External Definitions

---

```

translation-unit:
    external-definition
    translation-unit external-definition

external-definition:
    function-definition
    declaration

```

A translation unit consists of a sequence of external definitions of

- functions
- objects
- other declarations, such as type declarations (which are described as external when they appear outside of any function).

An external definition implicitly declares its identifier to have file scope and static storage duration. If there is no storage-class specifier, the identifier has external linkage.

As with other declarations, if there are no type specifiers, the type is taken to be `int`.

The storage-class specifiers `auto` and `register` may not appear in an external definition.

There cannot be more than one external definition for each identifier declared with internal linkage in a translation unit.

## **External Object Definitions**

In C, it is valid to declare identifiers in one translation unit, and to allocate storage for these elsewhere. When looking at a declaration, the following rules are used to decide whether it is a definition (storage is actually allocated), or whether it is a reference:

- An identifier defines an object if the identifier is declared outside of any function (file scope) and is initialized at the same time.
- An identifier constitutes a tentative definition for an object if the identifier is declared outside any function, but without an initializer and either without a storage-class specifier or with the static storage-class specifier.

E.g., the following declarations are all definitions, because initial values are assigned.

```
int      g1 = 7;
        /* definition, external linkage */
static int g2 = 3;
        /* definition, internal linkage */
extern int g3 = 94;
        /* definition, external linkage */
```

The following declarations are tentative definitions. `g4` has external linkage, since it is declared outside a function. `g5` has internal linkage because of the static storage class specifier.

```
int      g4;
        /* tentative definition, external linkage */
static int g5;
        /* tentative definition, internal linkage */
```

It is possible to declare an identifier multiple times in the same translation unit, provided the identifier has external linkage. Only one of these declarations can be a definition. Consequently, only one declaration for such an identifier can have an initializer.

In light of the declarations for `g1` through `g5` in the preceding listings, the following declarations are all valid.

```
int g1;
    /* valid tentative definition, refers to previous */
int g3;
```

```
/* valid tentative definition, refers to previous */  
int g4;  
/* valid tentative definition, refers to previous */
```

In contrast, neither of the following declarations are valid, because identifiers with internal linkage can only be declared once.

```
int g2;  
/* Error; linkage disagreement.*/  
int g5;  
/* Error; linkage disagreement */
```

A declaration (tentative or otherwise) for the same identifier with the same linkage may be encountered elsewhere in the translation unit. All such tentative definitions are taken to be declarations of the same object. The type for this object is the composite types of the declarations (subject to the linkage rules given in chapter 2). At the end of the translation unit, the compiler acts as if a definition of an object, with the corresponding composite type, occurred with an initializer of 0.

An identifier may be redeclared even if it has internal linkage  $\mathcal{A}$  if all declarations but one are prefixed with the storage class specifier `extern`. Thus, all the following are valid, given the earlier definitions and declarations.

```
extern int g1; /* external linkage */  
extern int g2; /* internal linkage */  
extern int g3; /* external linkage */  
extern int g4; /* external linkage */  
extern int g5; /* internal linkage */
```

An identifier declared as a tentative definition of an object and as having internal linkage may not be an incomplete type.

# CHAPTER 7

## Expressions

### Introduction

---

An expression is a sequence of operators and operands. This sequence can specify

- an object or function
- how to compute a value (e.g., multiplication, addition)
- how to generate side effects (e.g., assignment)
- some combination of these.

An expression can contain subexpressions among its elements. The order of evaluation of subexpressions, and the order in which side effects take place are both unspecified. Factors that affect the order of evaluation include:

- The order of precedence for operators in an expression. This order is specified by the syntax and it applies in the absence of parentheses. Operations with higher order of precedence are carried out first Æ unless parentheses are present to override this precedence.
- The use of parentheses to regroup subexpressions containing operators of different precedence.
- The ordering of elements in the expression. The C standard allows regrouping of any expression that involves more than one occurrence of the same commutative and associative binary operator or of operators that have the same precedence. Parentheses are honored in such a regrouping. The commutative and associative binary operators include ( `*`, `+`, `&`, `^`, `|` ).
- The presence of unary operators (such as the unary `+` operator), which restrict regrouping.

If the evaluation of an expression causes an object to be modified more than once the value of the expression is undefined.

Certain operators have operands that are of integral type. The return values for these operators depend on the internal representations of integers. In TopSpeed C, a two's complement representation is used.

An lvalue is an expression that designates an object. Certain lvalues are said to be modifiable. In particular, an lvalue is modifiable if it is not any of the following:

- an array type
- an incomplete type
- a type that has been qualified with the `const` qualifier
- a structure or union that contains a member (either directly or in a nested member) that is qualified with the `const` qualifier

In general, an lvalue is converted to the value stored at the object that the lvalue specifies. The exceptions are if the lvalue is any of the following:

- an array
- an operand for any of the following operators:
  - `sizeof`
  - unary `&`
  - `++`
  - `—`
- the left operand of any of the following operators
  - the dot operator (`.`)
  - an assignment operator

In general, an object that has type array of `<type>` is converted to an expression that has type pointer to `<type>`. The exceptions are if the lvalue is one of the following:

- the operand of the `sizeof` operator
- the operand of the unary `&` operator
- a string literal being used to initialize an array of `char`
- a wide string literal being used to initialize an array of a type that is compatible with `wchar_t`

A full expression is an expression that is not part of another expression. Each of the following is a full expression:

- An initializer.
- The expression in an expression statement.
- The controlling expression of a selection statement (if or switch).
- The controlling expression of an iteration statement (while, do, or for).
- The expression in a return statement.

The end of a full expression is a sequence point. At a sequence point, all side effects (e.g., assignments) of previous evaluations have been completed and no side effects of later evaluations have yet occurred.

## Precedence of Operators

---

The following table shows the order of precedence for TopSpeed C's operators. The values in the left column represent the precedence level for the specified types. All elements with the same value have the same precedence.

### primary expressions

16	literals names	simple tokens
16	a[i]	subscripting
16	f()	function call
16	.	direct selection
16	->	indirect selection

### unary operators

15	++ - -	postfix increment/decrement
14	++ - -	prefix increment/decrement
14	sizeof	size
14	(type-name)	& cast
14	~	bitwise not
14	!	logical not
14	-	arithmetic negation
14	+	unary '+'
14	&	address of
14	*	contents of

**binary operators**

13L	* / %	multiplicative
12L	+ -	additive
11L	<< >>	shift
10L	< > <= >=	inequality
9L	== !=	equality
8L	&	bitwise xor
6L		bitwise or
5L	&&	logical and
4L		logical or
3R	?:	conditional
2R	= + = - = * = / = % = << = >> = = ^ =   =	assignment
1L	,	comma

*L* indicates left associative operators;

*R* indicates right associative operators

## Primary Expressions

---

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

A primary expression is any of the following:

- An identifier that has been declared as naming an object or function. An identifier is an lvalue if it specifies an object and a function designator if it specifies a function. The value of an identifier depends on the type given when the identifier was declared.
- A constant. The constant's type depends upon its form, as described in chapter 4.
- A string literal, which is an lvalue of type array of char.
- A parenthesized expression. The type, value and "lvalueness" of a parenthesized expression are identical to those of the unparenthesized expression. Such an expression is a function designator, or void expression if the unparenthesized expression is, respectively, a function designator, or void expression.

The following list shows several examples of primary expressions.

```
Count          2.6
"fred"         (a+b*c)
301u           (0)sqrt(7.9)
```

In some circumstances identifiers of particular types have implicit conversions performed on them, see Chapter 5 for details.

## Postfix Operators

---

```
postfix-exp:
    primary
    postfix-exp [ expression ]
    postfix-exp ( argument-expression-listopt )
    postfix-exp . identifier postfix-exp -> identifier
    postfix-exp ++ postfix-exp -

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

Postfix operators are used to

- access elements in aggregate objects (such as arrays or structures)
- call functions
- increase or decrease (++ or —) the value of an object after use

### Array Subscripting

Array subscripting is used to access individual cells of an array. This is accomplished by using the [] operator. E.g., the array object specified by

```
int iarray[5]
```

is a 5-element array of int. The cells of such an array have indexes 0 through 4. The [] operator can be used to specify the third element of such an array as:

```
iarray[2]
```

One of the expressions in an array specification must have type pointer to T. The other expression (i.e., the element number) must have integral type. The result of applying the array subscription operator has type T.

Applying the [] operator is equivalent to applying the unary indirection operator (\*). This is because, in C, the following two expressions are equivalent:

```
iarray[2] and (*(iarray+(2))).
```

An array may have more than one subscript operator associated with it, in which case the array has multiple dimensions. Thus, successive subscript operators designate a member of a multidimensional array object.

If  $a$  is an  $n$ -dimensional array with dimensions  $i_1 \wedge j_1 \wedge \dots \wedge k$ , then  $a$  (when not used as an lvalue) is converted to a pointer to an  $(n-1)$ -dimensional array with dimensions  $j_1 \wedge \dots \wedge k$ . This is consistent with the conversion of an array type to a pointer (see Chapter 7)

Applying the unary  $*$  operator to this pointer results in the target  $(n-1)$ -dimensional array. This target array is, itself, converted into a pointer (when not used as an lvalue). The unary  $*$  operator can be called implicitly (through the subscript operator), with the same effect.

E.g., in the array object given by the declaration:

```
int b[7][3];
```

$b$  is a  $7 \wedge 3$  array of ints. Actually,  $b$  is an array of seven member objects, each of which is an array of three ints.

In the expression

```
b[i]
```

$b$  is first converted to a pointer to the initial 3-element array of ints. Then  $i$  is multiplied by the size of the pointer's target object (i.e.,  $\text{sizeof}(\text{int}) * 3$ ). The results are added and indirection is applied to yield the specified 3-element array. When used in the expression  $b[i][j]$ , that array, in turn, is converted to a pointer to the first of the ints, so  $b[i][j]$  yields an int.

Thus, arrays are stored in row-major order, in which the last subscript varies fastest.

**Note:**  $b[i, j]$  is not a two dimensional reference. It is equivalent to  $((b)+(i, j))$  i.e., the subscript is a comma expression

## Function Calls

---

A function call is a postfix expression followed by parentheses,  $()$ . The postfix expression denotes the function called. The parentheses will contain zero or more expressions, separated by commas. The expressions within the parentheses represent the arguments to the function.

E.g., the following expressions contain four function calls. These calls are to functions `getchar`, `putchar`, `printf`, and `abs`.

```
i = getchar();
putchar('\E n');
printf("%d items\E n", abs( Count))
```

There is a sequence point just before a function call.

The expression that denotes the function called must have type pointer to function returning either void or an object type other than array. A function name is converted to a pointer to the function.

An identifier is implicitly declared if the identifier represents a function  $\mathcal{A}E$  i.e., precedes an argument list  $\mathcal{A}E$  and the function is not within scope. This declaration occurs as if, the following declaration appeared

```
extern int identifier();
```

in the innermost block containing the function call.

An argument may be an expression of any object type. Prior to the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.

Parameters are passed by value, except arrays and functions whose address is implicitly taken and passed. The address-of operator, `&`, makes it possible to pass the address of an object, thereby passing that object by reference. In the case of array parameters, an implicit address-of occurs, so that arrays are always passed by reference.

**Note:** **when handling complex function declarations, it is useful to remember that a function is called in the same form as it was declared:**

```
long (*fp)(char, int); /* declaration */
(*fp)('a', 73);      /* call      */
```

If no function prototype declarator is in scope at the function call, the default argument promotions (see Chapter 7) are performed on each parameter.

The order of evaluation of the function designator and arguments in TopSpeed C is such that arguments are evaluated from left to right, followed by the function designator.

The behavior is undefined if the number of arguments does not agree with the number of parameters or if the types of the formal and actual arguments differ.

If a function prototype declarator is in scope at the function call, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters. The ellipsis notation `( , ... )` in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

If a parameter is declared with a type that is affected by the default argument promotions, and no semantically equivalent function prototype is in scope

where the function is defined, and a call is executed, the behavior is undefined.

The arguments in the call may not differ in number or type from the prototype. The types of the arguments in the call and the formal parameters in the prototype must be compatible as in an assignment.

**Note:** The arguments in a function call are only compared against the parameters of a function prototype declarator, if one exists.

Recursive function calls are permitted. The maximum depth of recursion is limited only by the size of the available stack.

The following listing illustrates some function calls:

```
void max_and_min(int [], int, int *, int *);
char contains(int [], int, int);
short process(int [], int, short (int));
short print_item(int);

int  list[LIST_SIZE],
     max_val,
     min_val,
     search_val;
short err;

void call_fns ( void)
{
    if (contains(list, LIST_SIZE, search_val)) {
        /* some processing */
    }
    max_and_min(list, LIST_SIZE, &max_val,
                &min_val);
    err = process(list, LIST_SIZE, print_item);
}
```

## Default Argument Promotions

The default argument promotions consist of the integer promotions (see Chapter 7) and the conversion of float to double. These default conversions are performed on expressions passed as arguments to functions where:

- No prototype declarator is in scope at the point of call.
- A prototype declarator with the (...) notation is in scope and the argument corresponds to one of the unspecified parameters.

## Structure and Union Members

The dot (.) and arrow (->) operators are used to select a member of a structure or union

A postfix expression followed by a dot `.` and an identifier selects a member of the structure or union object preceding the dot. The first operand of the `.` operator must have a struct or union type, which may be qualified. The second operand must be a member of the specified structure or union.

E.g.,

```
struct complex
  re, im : double;
} cval;

/* ... */
cval.re = 2.9;
cval.im = -7.6;
```

The value of the expression is the value of the named member. This is an lvalue if the original expression was an lvalue. If the left operand is a qualified type, the result inherits the same qualification

A postfix expression followed by an arrow `->` and an identifier also selects a member. The first operand of the `->` operator must have type pointer to struct or pointer to union. (The target aggregate may optionally be qualified.) The second operand must be a member of the target type. In this case, the member selected by the arrow operator is in the pointer's target structure or union.

The value of the expression is the value of the target object's member. This is always an lvalue.

If `&S` is a valid pointer expression, then the following two expressions are equivalent:

```
(&S->field and S . field)
```

The behavior is undefined if a member of a union object is assigned a value and is then accessed as a different member. In one special case the results of such an access are defined. A union may contain several structures that share a common initial sequence, and the union object must currently contain such a structure. If these criteria are met, it is possible to inspect the common initial part of any of the structures to obtain the expected results.

Two structs share a common initial sequence if the following conditions hold for a sequence of members, starting from the first.

- Corresponding members have compatible types.
- Any bit fields have the same width.

E.g.,

```
enum lisp_type { LIST, INTEGER, STRING };
union lisp {
  struct {
```

```

        enum lisp_type cell_is;
        union lisp *head, *tail;
    } list;
    struct {
        enum lisp_type cell_is;
        int value;
    } integer;
    struct {
        enum lisp_type cell_is;
        char *string;
    } string;
};

void print_cell(union lisp *x)
{
    switch (x->list.cell_type) {
        case LIST : {
            do {
                print_cell(x->list.head);
                x = x->list.tail;
            } while (x); /* x != NULL */
            break;
        }
        case INTEGER : {
            printf("%d", x->integer.value);
            break;
        }
        case STRING :
            printf("%s", x->string.string); }
    }
}

```

If *f* is a function returning a struct or union, and *x* is a member of that struct or union, then *f().x* is a valid postfix expression but is not an lvalue (because *f()* is not an lvalue).

The following is a valid fragment:

```

struct cell {
    int value;
    struct cell *next;
};

struct cell find(struct cell *, int),
    *new(void),
    *head,
    *tail,
    temp;
(head = new())->next = new();
head->value = 3;
head->next->value = 5;

temp = find(head, 3);
tail = find(head, 6).next;

```

## Postfix Increment and Decrement Operators

---

The postfix increment and decrement operators change the values of objects. Such changes are made only after the value is obtained. The operand for these operators

- must have scalar type (which may be qualified),
- must be a modifiable lvalue

The postfix increment operator (++) returns the value of the operand as the operator's result. After the result is returned, the value of the operand is incremented by 1. The side effect of updating the stored value of the operand occurs between the previous and the next sequence point.

For pointers, the value of each increment is the sizeof the target object. For other scalars, the increment is the value 1 of the appropriate type.

The postfix decrement operator (--) is analogous to the postfix ++ operator, except that the value of the operand is decremented. That is, the value 1 (of the appropriate type) is subtracted from the operand.

E.g.,

```
i = 0;
while (*s)    /* find end of string and length */
    i++, s++;
s++;
while (i) {
    i--;      /* reverse string s into string p */
    *p++ = *s;
}
*p = 0;
/* 0 terminates p */
```

## Unary Operators

---

```
unary-exp:
    postfix-exp
    ++ unary-exp
    - unary-exp
    unary-operator cast-exp
    sizeof unary-exp
    sizeof ( type-name )
    unary-operator: one of
    & * + - ~ !
```

C has several unary operators, which are used to accomplish such things as:

- changing the value of an object before using the object (++ and —)
- converting an expression's value from its declared type to another type
- accessing particular storage locations
- determining the storage allocated for an object
- doing arithmetic on an expression (e.g., +, -, ~)

All these operators have high precedence as seen in Chapter 7.

### **Prefix Increment and Decrement Operators**

The prefix increment and decrement operators change the value of an object. The difference between these operators and the postfix versions is in the timing of the changes. The operand of the prefix increment or decrement operator

- must have scalar type (which may be qualified),
- must be a modifiable lvalue

The value of the operand of the prefix increment operator (++) is incremented before the value is used in an expression. The result is the new value of the operand (i.e., after being incremented). The expression ++val is equivalent to changing a value by adding 1. For pointers, the value of the increment is the sizeof the target object.

The prefix decrement operator (—) is analogous to the prefix ++ operator, except that the value of the operand is decremented. The expression —val is equivalent to (val-=1).

E.g,

```
int i;
char *s, *p;

i = 0;
while (*s) /* find end of string and length */
    ++i, ++s;
while (i) {
    Æi; /* reverse string s into string p */
    *p++ = *Æs;
    /* post-increment p! */
}
*p = 0;
```

## **Address and Indirection Operators**

The unary address-of operator (&) produces a pointer to the object or function designated by the operand. The result has type pointer to T, where T is the operand's type. E.g.,

```
int *i_ptr, ival;
i_ptr = &ival; /* ival becomes target object */
               /* for i_ptr */
```

The operand of the unary & operator must be one of the following:

- A function designator.
- An lvalue that designates an object that is neither a bit-field nor declared with the register storage-class specifier

The unary indirection operator (\*) produces the value of the operand's target object. The operand must have a pointer type. E.g.,

```
ival = *i_ptr;
*i_ptr = ival;
```

If the operand has type pointer to T, the result has type T. If the operand points to a function, the result is a function designator. If it points to an object, the result is an lvalue designating the object.

If \*P is an lvalue and T is the name of an object pointer type, the cast expression \*(T)P is an lvalue that has the same type as that to which T points.

Under certain conditions, the result of applying the indirection operator will be invalid. In such cases, the result is undefined. Invalid values for dereferencing a pointer by the unary \* operator result when any of the following is accessed:

- A null pointer constant.
- The address of an object that has automatic storage duration (e.g., a parameter or a local object) when execution of the block in which the object is declared has terminated.

E.g.,

```
void check_bounds(int *val_addr, int lo, int hi)
/* force object into given range lo..hi */
{
    if (*val_addr < lo)
        *val_addr = lo;
    else if (*val_addr > hi)
        *val_addr = hi;
```

```
}

main()
{
    int value,
        *val_ptr,
        table[TAB_SIZE];

    /* get value */

    check_bounds(&value, 0, 100);

    /* fill table with data */

    for (val_ptr = table;
        val_ptr < &table[TAB_SIZE];
        val_ptr++)
        check_bounds(val_ptr, 0, 100);
}
```

## **Unary Arithmetic Operators**

The unary arithmetic operators return or change the numerical values of their operands

The result of the unary `+` operator is the value of its operand. The expression `+E` is equivalent to `(0+E)`

The result of the unary `-` operator is the negation of its operand. The expression `-E` is equivalent to `(0-E)`

Operands for the unary `+` and `-` operators must have arithmetic type.

The result of the `~` operator is the bitwise complement of its operand. Each bit in the operand is processed. If the bit is on, the `~` operator turns it off; if the bit is off, the operator turns it on. The expression `~E` is equivalent to either of the following values (which are defined in `<limits>`  $\mathcal{A}$  depending on the type of the operand).

- `(ULONG_MAX-E)` if `E` has type unsigned long
- `(UINT_MAX-E)` if `E` has any other unsigned type.

For these three operators, the result has the promoted type after integral promotions have been performed on the operand.

The operand for the bitwise complement operator must have integral type

Logical Negation Operator

The result of the logical negation operator `!` is

- 0 if the value of its operand is nonzero.

- 1 if the value of its operand is zero.

The result has type `int`. The expression `!E` is equivalent to `(0==E)`.

The operand for the logical negation operator must have scalar type.

E.g.,

```
#define ERR_FLAG 0x0400
short FLAGS;
/* ... */

FLAGS &= ~ERR_FLAG; /* clear error flag */
if (!FLAGS) {
    /* FLAGS is zero */
    FLAGS = -1;
}
```

## **The sizeof Operator**

The result of the `sizeof` operator is an integer constant that represents the size (in bytes) of its operand. The size is determined from the type of the operand, which is not itself evaluated. The result has type `size_t` defined in the `<stddef>` header. In TopSpeed C, a `size_t` is an unsigned `int`.

The operand may be an expression or the parenthesized name of a type. The size of an operand is not affected if the operand has an associated type qualifier.

The result of applying the `sizeof` operator depends on the operand's type.

- For an operand that has character type (i.e., `char`, `unsigned char` or `signed char`), the result is 1.
- For an operand that has type `short int` or `unsigned short int`, the result is 2.
- For an operand that has type `int` or `unsigned int`, the result is 2.
- For an operand that has type `long int` or `unsigned long int`, the result is 4.
- For an operand that has type `float`, the result is 4.
- For an operand that has type `double`, the result is 8. For long double, 10.
- For an operand that has array type, the result is the total number of bytes in the array  $\text{Æ}$  i.e., the number of elements times `sizeof (<element-type>)`.
- For a parameter declared to have array or function type, the `sizeof` operator yields the size of the pointer obtained by conversion.

- For an operand that has structure or union type, the result is the total number of bytes in such an object, including internal padding. Internal padding may occur when the structure includes bit fields.

The sizeof operator may not be applied to any of the following:

- An expression that has function type.
- An incomplete type
- The parenthesized type name of a function type or an incomplete type
- A bit-field object

E.g.,

```
int table[TABSIZE],
    table_length,
    no_of_elements;

table_length = sizeof table;
/* == TABSIZE*sizeof(int) */

no_of_elements = table_length / sizeof table[0];
```

The sizeof function is very useful for calls to malloc.

## Cast Operators

---

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

The cast operator converts the value of an expression to a type specified within parentheses. If the type name specifies the void type, the expression may have any type; otherwise, the type name and the operand (i.e., the expression) must have scalar type.

In ANSI C, a cast does not yield an lvalue. TopSpeed C has a pragma option (`lang_ext=>on`) in which a cast yields an lvalue if the expression is an lvalue.

A cast changes the type of its operand. A change of representation may occur or the conversion may be quiet Æ i.e., no change of representation is involved. However, the new type may cause the code generator to act differently.

A pointer may be converted to an integral type. In near pointer models, an int can hold a pointer; in far pointer models, a long int can hold a pointer. In TopSpeed C, long is the only integral type guaranteed to hold a complete pointer. Zero (0) may be converted to the null pointer constant.

A pointer to an object or incomplete type may be converted to a pointer to a different object type or to a different incomplete type.

A pointer to a function of one type may be converted to a pointer to a function of another type and back again. The resulting pointer will compare equal to the original pointer. The behavior is undefined if a converted pointer is actually used to call a function that is not compatible with the original type.

E.g,

```
void *malloc();
struct pair {
    int value;
    char name[NAMELEN];
} *ptr;
ptr = (struct pair *) malloc(sizeof *ptr);

*(int *) ptr = 0;
strncpy((char *)
    (1 + (int *) ptr), "ZERO", NAMELEN);
```

## Multiplicative Operators

---

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

Interpreted as a binary operator, the `*` symbol represents the multiplication operator, which yields the product of its operands. These operands must have arithmetic type. The multiplication operator is commutative and associative.

The binary division operator (`/`) yields the quotient from dividing the first operand by the second. Both operands must have arithmetic type. The second operand must be nonzero.

The binary remainder operator (`%`) requires two integral operands. This operator yields the whole number remainder after dividing the first operand by the second. The second operand must be nonzero.

When integers are divided and leave a non-zero remainder, the results depend on the signs of the arguments. The magnitude of `a/b` is the integer given by `abs(a) / abs(b)`.

- If both operands are positive,
- The result of the `/` operator is the largest integer less than the algebraic quotient.
- The result of the `%` operator is positive.

- If either operand is negative,
- The result of the / operator is the smallest integer greater than the algebraic quotient.
- If the two operands have the same sign, then the result of applying the / operator is positive; otherwise it is negative.
- If one operand is negative, the sign of the result of the % operator is negative.

If the quotient  $a/b$  can be represented, the following expression is true:

$$(a/b)*b + a\%b == a$$

The usual arithmetic conversions are performed on the operands

## Additive Operators

---

```
additive-exp:  
    multiplicative-exp  
    additive-exp + multiplicative-exp  
    additive-exp - multiplicative-exp
```

Interpreted as a binary operator, the + symbol represents the addition operator, which yields the sum of its operands. The addition operator is commutative and associative.

One of the following type constraints must hold for the addition operator:

- Both operands have arithmetic type.
- One operand is a pointer to an object and the other operand has integral type

The binary subtraction operator (-) yields the difference resulting when the second operand is subtracted from the first.

One of the following type constraints must hold for the subtraction operator:

- Both operands have arithmetic type.
- Both operands are pointers to compatible object types, which may be qualified.
- The left operand is a pointer to an object and the right operand has integral type.

Additive operators can be used with pointers. When an expression that has integral type is added to or subtracted from a pointer:

1. The integral value is first multiplied by the size of the target object.
2. The result is a pointer of the same type as the original pointer.

If the original pointer points to a member of an array object and the array is large enough, the result points to another member of the same array object. This new target object is offset an appropriate amount from the original member. Thus if *P* points to a member of an array object, the expression *P*+1 points to the next member of the array object.

Unless both the pointer operand and the result point to a member of the same array object, the behavior of the result may be undefined. In particular, this result is undefined if used as the operand of a unary *\** operator. Thus, the result of applying an additive operator is undefined if the result points to a location outside the array containing the original pointer operand.

When two pointers to members of the same array object are subtracted, the difference is divided by the size of a member. The result represents the difference of the subscripts of the two array members. The result has type `ptrdiff_t`, which is defined in the `<stddef>` header as an int. (When subtracting huge pointers, the result is actually a long, and not a `ptrdiff_t`)

If two pointers that do not point to members of the same array object are subtracted, the result is undefined. However, if *P* points to the last member of an array object, the expression (*P*+1) - *P* has the value 1, even though *P*+1 does not point to a member of the array object.

If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

E.g.,

```
int table[TABSIZE], pos, ix;
struct {
    int *start, end;
} index[INDEXSIZE];

#define NEXT      0
#define LENGTH    1
#define DATA      2

/*
table represents a linked list of
variable length data blocks
held as an array of pseudo structures:
    struct {
        int offset;                /* of next entry */
        int length;                /* of this data block */
        int data[];
        /* incomplete array!! */
    }
BASELOC  is the first entry to be indexed
```

```

        table[NEXT] retrieves the offset field
        table[LENGTH] retrieves the length field
        table[DATA] is the zeroth element of
            the data field
    */
    pos = BASELOC;
    for (ix = 0; pos > 0; ix++) {
        index[ix].start = &table[pos + DATA];
        /* zeroth data element */
        index[ix].end = index[ix] + table[pos + LENGTH];
        /* all data before that address */
        pos = table[pos + NEXT];
        /* advance to next logical block */
    }

```

## Bitwise Shift Operators

---

```

shift-exp:
    additive-exp
    shift-exp << additive-exp
    shift-exp >> additive-exp

```

The left shift operator (<<) takes two integral operands, namely `left_op` and `right_op`. The result of

```
left_op << right_op
```

is `left_op` shifted left `right_op` bit positions. Vacated bits are filled with zeros.

If `left_op` has an unsigned type, the value of the result is `left_op` multiplied by (2 raised to the power `right_op`)

The right shift operator (>>) takes two integral operands, namely `left_op` and `right_op`. The result of

```
left_op >> right_op
```

is `left_op` shifted right `right_op` bit positions.

The value resulting from such a shift depends on the left operand's type and value. The value represents the integral part of the quotient of `left_op` divided by 2 raised to the power `right_op`, if either of the following holds:

- `left_op` has an unsigned type
- `left_op` has a signed type and a non-negative value

If `left_op` has a signed type, the vacated bits are filled with the same bit value as the original leftmost bit.

The integral promotions are performed on each of the operands for bitwise shift operators. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width (in bits) of the promoted left operand, the result will be 0 or -1

(depending on whether the original value being shifted was positive or negative).

E.g.,

```
unsigned long date_time_stamp,
/* year (base 1980), month, day, */
/* hours, minutes, seconds: */
/* YYYYYYMMDDDDDDhhhhmmmmsssss */

    US_date;

/* month, day, year (base 1900): */
/* MDDDDDDDDYYYYYY */

date_time_stamp =
    (hours << 12) | (minutes << 6) | seconds |
    (
        ((year-1980 << 9) |
         (month << 5) | day) << 17
    );

US_date = (date_time_stamp >> 26) + 80 |
/* year... */
    (
        (date_time_stamp >> 10) & 0x0F80
    ) |

/* day... */
    (
        (((date_time_stamp >> 17) >> 5) & 0x0F) << 12
        /* month... */
    );
```

## Relational Operators

---

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

The relational operators are used to compare two values. Depending on the result of such a comparison, the result from a relational operator will be 0 or 1.

When applying relational operators, one of the following must hold:

- Both operands have arithmetic type.
- Both operands are pointers to objects that have compatible types. These types may be qualified.
- Both operands are pointers to compatible incomplete types. These types may be qualified.

ANSI C does not allow a mixture of pointer and integer operands for the relational operators. TopSpeed C allows this as an extension, so that an expression such as

`p > 0`

(where `p` is a pointer) is valid.

When two pointers are compared, the result depends on the relative locations of the target objects. If the target objects are members of the same aggregate object, the following rules apply:

- Pointers to structure members declared later compare higher than pointers to members declared earlier in the structure.
- Pointers to array elements with larger subscript values compare higher than pointers to elements of the same array with lower subscript values.
- All pointers to members of the same union object compare equal.

If the target objects are not members of the same aggregate or union object, the result is undefined unless `P` points to the last member of an array object. In that case, the pointer expression `P+1` compares higher than `P`, even though `P+1` does not point to a member of the array object.

Each of the following operators yields 1 if the specified relation is true and 0 if it is false. The result has type `int`

- less than operator (`<`)
- greater than operator (`>`)
- less than or equal to operator (`<=`)
- greater than or equal to operator (`>=`)

If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

## Equality Operators

---

```
equality-exp:  
    relational-exp  
    equality-exp == relational-exp  
    equality-exp != relational-exp
```

The equality operators are used to compare two values. Depending on the result of such a comparison, the result from an equality operator will be 0 or 1. The `=` (equal to) and the `!=` (not equal to) operators behave in a similar way to the relational operators except for their lower precedence.

When applying equality operators, one of the following must hold:

- Both operands have arithmetic type.
- Both operands are pointers to the same type. This type may be qualified.
- One operand is a pointer to an object or incomplete type and the other is a pointer to void.
- One operand is a pointer and the other is a null pointer constant.

The following results apply when using the equality operators:

- If two pointers to objects or pointers to incomplete types compare equal, they point to the same object, they are both null pointers, or they both point to one past the last element of the same array.
- If two pointers to functions compare equal, they point to the same function.
- If one of the operands is a pointer to an object (or pointer to an incomplete type) and the other has type pointer to void (optionally qualified), the pointer to an object (or pointer to an incomplete type) is converted to type pointer to void.

E.g.,

```
void *malloc(),
    *mem_base;
struct list *head;
    /* struct with fields value, next */

mem_base = malloc(sizeof *head);
head = (struct list *) mem_base;

/* .. */

while (head != mem_base &&
    head != 0 && head->value != search_val)
    head = head->next;
```

## Bitwise AND Operator

---

```
AND-exp:
    equality-exp
    AND-exp & equality-exp
```

Interpreted as a binary operator, the & symbol represents the bitwise AND operator. This operator takes two integral operands, and yields the bitwise AND of the operands as an integral result. Corresponding bits in each operand are examined, and the resulting bit is set to 1 only if corresponding bits in both operands are 1; otherwise the resulting bit is set to 0

The binary & operator is commutative and associative. However, order of evaluation for the operands is not guaranteed to be from left to right.

The usual arithmetic conversions are performed on the operands.

## Bitwise Exclusive OR Operator

---

```
exclusive-OR-exp:  
    AND-exp  
    exclusive-OR-exp ^ AND-exp
```

The bitwise exclusive OR operator (^) is one of two bitwise OR operators in C. This operand takes two integral operands and yields an integral result. Corresponding bits in each operand are examined, and the resulting bit is set to 1 only if exactly one of the corresponding bits is 1; otherwise the resulting bit is set to 0.

The ^ operator is commutative and associative. However, order of evaluation for the operands is not guaranteed to be from left to right.

The usual arithmetic conversions are performed on the operands.

## Bitwise Inclusive OR Operator

---

```
inclusive-OR-exp:  
    exclusive-OR-exp  
    inclusive-OR-exp | exclusive-OR-exp
```

The bitwise inclusive OR operator (|) takes two integral operands, and yields an integral value.

The result of the | operator is the bitwise inclusive OR of the operands. Corresponding bits in each operand are examined, and the resulting bit is set to 1 if either or both of the corresponding bits is 1; otherwise the resulting bit is set to 0.

The | operator is commutative and associative. However, order of evaluation for the operands is not guaranteed to be from left to right.

The usual arithmetic conversions are performed on the operands

## Logical AND Operator

---

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

The logical AND operator (&&) takes two scalar operands, and yields 1 if both of its operands are nonzero; otherwise the && operator yields 0. The result has type int.

Unlike the bitwise binary & operator, the operands for the && operator are always evaluated from left to right. There is a sequence point after the first operand is evaluated. If the first operand compares equal to 0, the second operand is not evaluated.

E.g.,

```
int *pif (p!=NULL) && (p->val != key) ..
```

The order in which the subexpressions for && are evaluated is significant, and assures that a NULL pointer is not dereferenced. (In Pascal this piece of code would have to be written as two if statements.)

## Logical OR Operator

---

```
logical-OR-exp:
    logical-AND-exp
    logical-OR-exp || logical-AND-exp
```

The logical or operator (||) takes two operands of scalar type, and yields 1 if either of its operands is nonzero; otherwise it yields 0. The result has type int.

Unlike the bitwise binary | operator, the operands for the || operator are always evaluated from left to right. There is a sequence point after the first operand is evaluated. If the first operand is nonzero, the second operand is not evaluated.

E.g.,

```
i = (m == n) || (o == p);
```

is equivalent to:

```
if (m == n)      i = 1;else      if (o == p)
i = 1;           else           i = 0;
```

## Conditional Operator

---

```
conditional-exp:
    logical-OR-exp
    logical-OR-exp ? exp : conditional-exp
```

The conditional operator (?:) is a ternary operator that selects one of two possible operands, depending on the value of the first operand. The first operand must have scalar type. There is a sequence point after the first operand is evaluated. If the first operand is nonzero, the second operand is evaluated and the result of applying the conditional operator is the second operand's value; otherwise the third operand is evaluated and its value is the result.

A conditional expression does not yield an lvalue.

One of the following must hold for the second and third operands:

- Both operands have arithmetic type.
- Both operands are pointers to compatible types, which may be qualified.
- Both operands have compatible structure or union type.
- Both operands are void expressions.
- One operand is a pointer to an object or incomplete type and the other is a pointer to void (possibly qualified).
- One operand is a pointer and the other is a null pointer constant.

The following considerations regarding the result apply when evaluating the second and third arguments:

- If both of these operands have arithmetic type, the usual arithmetic conversions are performed to bring the operands to the same type. The result has that type.
- If both the operands have structure, union, or pointer type, the result has that type.
- If both the operands are void expressions, the result is a void expression.
- If one of the operands is a pointer to an object or incomplete type and the other is a pointer to void, the pointer to an object or incomplete type is converted to type pointer to void, and the result has that type.
- If one operand is a pointer and the other operand is a null pointer constant, the result has the type of the pointer with all the qualifiers of the types pointed to by both operands.
- If both operands are pointers (optionally qualified), the result has the composite type.

E.g.,

```
max = a > b ? a : b;  
select = (y ? s1 : s2).f;  
ptr = count > 0 ? src_ptr : NULL;
```

## Assignment Operators

```

assignment-exp:
    conditional-exp
    unary-exp
    assignment-operator assignment-exp

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

```

C's assignment operators are used to modify the values of objects in the program. C provides both simple and compound assignment operators.

An assignment operator stores a value in the object specified by the left operand. An assignment operator's main task (i.e., updating an object's value) actually occurs as a side effect. The left operand for an assignment operator must be a modifiable lvalue.

After applying an assignment operator, an assignment expression has the value and the unqualified type of the left operand. This, however, is not an lvalue.

The side effect of updating the stored value of the left operand occurs between the sequence points preceding and following the assignment expression

### Simple Assignment

The simple assignment operator (=) assigns the value of the right operand to the left operand. The right operand's value replaces the value stored in the object designated by the left operand. Prior to this, the value of the right operand is converted to the type of the left operand.

One of the following conditions regarding possible operand types must hold:

- Both operands have arithmetic type. (The left operand may be qualified.)
- Both operands have compatible structure or union type. (The left operand may be qualified.)
- Both operands are pointers to compatible types. They may be qualified types and, if so, the left operand must have all the qualifiers belonging to the right operand.
- One operand is a pointer to an object or incomplete type and the other is a pointer to void (possibly qualified).
- The left operand is a pointer and the right is a null pointer constant.

E.g.,

```
i = 0;
a[i] = ++i - 1; /* a[0] = 0, i = 1 */
```

If the left and right operand objects overlap in any way the behavior is undefined.

In the program fragment:

```
int i;
char c;
/* ... */
((c = i) == -1)
```

the int value i may be truncated when stored in the char. The truncated value will then be converted back to an int prior to the comparison. The resulting value will not necessarily be equal to the original value.

## Compound Assignment

A compound assignment takes the following form:

```
left_op <op>= right_op;
```

<op> represents a slot for an operator. E.g.,

```
left_op += right_op;
left_op &= right_op;
left_op /= right_op;
left_op %= right_op;
left_op <<= right_op;
```

are all compound assignments.

A compound assignment is equivalent to the following:

```
left_op = left_op <op> right_op;
```

except that left\_op is evaluated only once

For the operators += and -= only, either of the following special rules may apply:

- Both operands must have arithmetic type. (The left operand may be optionally qualified.)
- The left operand may be a pointer to an object type and the right must have integral type.

For the other compound assignment operators, each operand must have an arithmetic type consistent with those allowed for the corresponding binary operator

**Note:** The unary increment operator (++) and decrement (--) operators are also assignment operators. Thus, `++i` is equivalent to `(i+=1)`.

E.g.,

```
int *p;
long count; short FLAG;
while (*p == ENTRY)
    p += ENTRY_LEN;
count /= 4;
FLAG |= err_bits;
```

## Comma Operator

---

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

Both operands for a comma operator are evaluated, but only one value is returned as a result. The left operand of a comma operator is evaluated as a void expression  $\mathcal{E}$  that is, no value is returned. There is a sequence point after this operand is evaluated.

After this sequence point, the right operand is evaluated. The result of applying the comma operator has the right operand's type and value.

A comma operator does not yield an lvalue.

In contexts where a comma is used as a punctuator (such as in argument lists for functions and initializer lists), the comma operator as described here may appear only within matching nested parentheses. E.g., in the function call:

```
f(a, (b=3, b+2, b+7), (c=25, c-10), d)
```

the function has four arguments, the second of which has the value 12 and the third of which has the value 15.

E.g., after the following statement is processed, `val` will have the value corresponding to `table[index] + table[index+1]`.

```
val = (index += offset,
       table[index] + table[index+1]);
```

## Constant Expressions

---

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

A constant expression is any expression that can be evaluated to a single value during translation. A constant expression may appear anywhere a value of the same type as the constant expression may appear. As a result, constant expressions are often used to initialize objects. A constant expression cannot cause side-effects.

In certain situations, a constant expression is required. E.g., the following cases require an integral constant expression:

- specifying the size of an array
- specifying a value for an enumeration
- specifying a case value for a switch statement
- specifying the size of a bit-field

The operands in an integral constant expression must have one of the following types.

- integer constant
- character constant
- enumeration
- sizeof expression
- cast of floating constant to integral type

Any operators for which these operands are valid can be used in an integral constant expression. E.g, +, %, -, etc., can all be used.

On the other hand, operators that require lvalues cannot be used. These operators include ++, Æ, & ->, . (dot operator). Such operators can be used within the operands for the sizeof operator.

When used as an initializer, a constant expression must evaluate to one of the following:

- An arithmetic constant expression. (This means that floating point constants can be used when initializing.)
- A null pointer constant.
- An address constant.
- An address constant for an object type plus or minus an integral constant expression.

An arithmetic constant expression is similar to an integral constant expression, but with arithmetic type substituted for integral type.

Any operators that can be used with arithmetic operands can be used in an arithmetic constant expression Æ provided the operator is valid for the actual operands that will be passed to the operator.

An address constant is a pointer to an lvalue. This lvalue must designate one of the following:

- An object having static storage duration
- a function designator

An address constant may not be used to access the target value, as part of a subexpression of an address constant.

**Note:** Because the operands to `sizeof` are not evaluated it is possible to use non-constant expressions in this context

E.g.,

```
static intdbl_size = sizeof(double);
static inta[4] = { 0, 1+1, (char)1.7, 'a'};
struct {int f,g;}s = { 1+(1*2), 97%6};
extern int*p = &a[2],
        *q = &(s.f);
```

Further constraints that apply to the integral constant expressions used in conditional inclusion preprocessing directives are discussed in chapter 9.

# CHAPTER 8

## Statements

A statement specifies an action to be performed by the program. A statement can be simple or compound. A compound statement groups zero or more simple and compound statements together. This group is treated as a single statement.

Statements are executed in the order in which they are encountered in the program, unless this sequence is modified by statement types described below.

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

### Labeled Statements

---

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

A labeled statement serves as a target statement for an unconditional jump (i.e., goto) instruction. A statement may be preceded by an identifier that serves as a label name for the statement. The label is followed by a colon and then by the statement.

E.g.,

```
the_label: val *= 7.5;
```

Two labels, case and default are predefined in C. These two labels may only appear within the body of a switch statement (see Chapter 8).

### Compound Statement, or Block

---

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

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

A compound statement (also known as a block) consists of a collection of statements treated as one syntactic unit. A compound statement begins with a left curly brace ( { ) and ends with the corresponding right brace ( } ).

A compound statement causes a new scope to be opened. It is possible to declare new variables, typedefs and tags in this new scope.

TopSpeed C calculates the maximum amount of storage required by all blocks (nested blocks adding to the outer enclosing block). This storage is allocated when the function is entered.

Initializers for objects with automatic storage duration in a block are evaluated prior to the execution of the first statement in that block. The objects are initialized in the order in which they appear in the block.

E.g.,

```
for (i = 0; i < NROWS; i++) {
    int row_total = 0;
    for (j = 0; j < NCOLS; j++)
        row_total += matrix[i][j];
    printf("row %d total is %d\n", i, row_total);
}
```

## Expression and Null Statements

---

```
expression-statement:
    expressionopt;
```

The expression in an expression statement is evaluated as a void expression for its side effects. Some typical examples of this are assignments, functions calls, and the use of the ++ and — operators.

A null statement does nothing. (Such a statement consists of just a semicolon, or matching braces not containing any statements.)

A function call may be evaluated for its side effects only Æ that is, the function's return value is not needed. In such a case, it is possible to discard the returned value explicitly. To accomplish this, it is necessary to use a cast to convert the expression to a void expression. (Such a conversion is not mandatory, however.)

E.g.,

```
int p(int);    /* function declaration */
/*...*/
(void) p(0);  /* function call          */
```

In the program fragments:

```
char *s;
/*...*/
while (*s++ != '\\')
```

```

;
and
char *s;
/*...*/
while (*s++ != '\\')
{ }

```

null statements are used to supply empty loop bodies to the iteration statements.

## Jump Statements

---

```

jump-statement:
goto identifier;
continue;
break;
return expressionopt;

```

A jump statement causes an unconditional transfer of control to another place within the function.

### The goto Statement

A goto statement causes an unconditional jump to the named label in the current function. The label must be a prefix for a statement in the same function.

E.g.,

```

/* ... */
val++;
if ( val > CUTOFF)
    goto alldone;
else
    /* ... */
/* ... */
alldone: printf ( "all done\n");

```

### The continue Statement

A continue statement can appear only within an iteration statement. A continue statement causes a jump to the end of the loop body of the enclosing iteration statement. The loop will continue executing from this point. I.e., the loop condition will be tested again, etc.

E.g.,

```

for (i=1; i<10; i++)
{
    if (a[i+1]==1)
        continue;
    a[i+1]=0;
    /* continue arrives here */
}

```

```

while (i)
{
    switch (i)
    {
        case 1: if (a[i+2]==2)
                continue;
                a[i+2]=0;

        case 2: if (a[i+3]==3)
                continue;
                a[i+3]=0;
        default :
    }
    a[i+4]=0;
    /* continue arrives here */
}
do {
    continue;
    ...
}
/* continue arrives here */
while (i);

```

## **The break Statement**

A break statement can appear only within a switch or iteration statement. When encountered during execution, a break statement stops execution of the smallest enclosing switch or iteration statement.

E.g.,

```

switch (k)
{
    case 0: for (i=1; i++; i<10)
        {
            for (j=1; j++; j<10)
                if (j==5)
                    break; /* terminate j loop */
            else
                a[i+j]=0;
            if (a[i+5]==6)
                break; /* terminate i loop */
        }
    /* fall through */
    case 1: switch (l)
        {
            case 0: a[1]=0; /* fall through */
            case 1: a[2]=0
                break; /* terminate l switch */
            default :
        }
    break; /* terminate k switch */
}

```

## **The return Statement**

A return statement terminates execution of the function in which the statement is contained. After the return statement is executed, control is returned to the function's caller.

A return statement may include an expression. When a return statement with an expression is executed, the expression is evaluated, and its value is returned to the caller. If the expression's value has a different type from the enclosing function's return type, the value is converted, as if the value were assigned to an object of that type.

Reaching the end of a function is equivalent to executing a return statement without an expression. If such an empty return statement is executed, the returned value is undefined.

A function may contain any number of return statements, with and without expressions. A function with return type void may not contain a return statement with an expression.

In TopSpeed C the storage for the return value is allocated in the calling function.

E.g.,

```
int f1()
{
    int i;
    /* code */
    return i;
}

struct s f2()
{
    struct s v1;
    /* code */
    if (a[6]==9)
        return v1;
    /* change v1 */
    return v1;}

```

## **Selection Statements**

---

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

```

A selection statement chooses among a set of statements. The selection is dependent on the value of a controlling expression.

C provides two types of selection statements: the if and switch statements.

## **The if Statement**

The if statement has two forms:

- An if statement with no alternative action specified.
- An if statement with an alternative action specified in an else clause.

In each form, the if statement has a controlling expression and a (simple or compound) substatement associated with it. The controlling expression of an if statement must have scalar type, and must be enclosed in parentheses when specified in a statement.

In both forms, the controlling expression is evaluated. If this result is nonzero, the first substatement is executed.

If the controlling expression evaluates to zero, the action taken depends on which form of the if statement is being used.

- If no alternative action is specified, the program does nothing in the if statement. Execution continues with the statement after the if statement.
- If an alternative action is specified, the substatement associated with the else clause will be executed.

If the first substatement is reached via a label (i.e., a goto is performed into the first branch of the if statement), the second substatement is not executed.

An else is associated with the nearest preceding if that satisfies the following conditions:

- The if must be in the same block (but not in an enclosed block)
- The if must not yet be matched with an else.

E.g., compare the following two if statements:

```
if (index < TABSIZE)
    if (table[index])
        zero++;
    else
        puts("Error: index out of bounds");
if (index < TABSIZE) {
    if (table[index])
        zero++;
}
else
    puts("Error: index out of bounds");
```

The first one is probably not what you intended, since it displays an “out of bounds” message if the value of table[index] is zero, rather than if index is too large.

## **The switch Statement**

The switch statement is used to select from among several possible actions, depending on the value of a controlling expression. This value will determine the point in the switch body to which control will jump during execution. The jump will be to one of the following:

- To the statement following a case label whose expression has the same value as the controlling expression.
- To the statement following the default label in the switch body, if no case label matches the value of the controlling expression and if the switch body contains a default label.
- To the statement following the end of the switch body, if no case label matches the value of the controlling expression and if the switch body does not include a default label. In this case, none of the statements in the switch body is executed.

The controlling expression for a switch statement must have integral type, and the case labels must be integral constant expressions. The integral promotions are performed on the controlling expression. The constant expression in each case label is converted to the same type as the promoted controlling expression. All case constants in the same switch statement must have different values after conversion. There may be at most one default label in a switch statement.

A case or default label is accessible only within the closest enclosing switch statement.

In TopSpeed C, the number of case labels in a switch statement is limited only by the amount of memory available.

E.g.,

```
switch (error_level) {
    case 0 :
        case 1 :
            warnings++;
            /* fall through */
        case 2 :
            errors++;
            printf("Error (level %d) : %s\n",
                error_level, error_msg);
            break;
    default :
        puts("FATAL error");
        exit();
}
switch (value) /* untraditional use of switch
statement */
    default :
        if (value > 5)
```

```

        case 0 :
        case 1 :
            value = 0;
    else
        case 9 :
            value = 1;

/* value is set to 1 if it was originally
   2, 3, 4, 5 or 9 (or negative)
   or set to 0 if it was originally
   0, 1, 6, 7, 8 or greater than 9*/

```

## Iteration Statements

---

```

iteration-statement:
    while ( expression ) statement
    do statement while ( expression );
    for ( expropt; expropt; expropt ) statement

```

Iteration statements are used to repeat something, possibly with variations. An iteration statement includes a controlling expression and a loop body.

The controlling expression must have scalar type. This expression is evaluated to determine whether to execute another iteration of the loop body. That is, an iteration statement causes the loop body to be executed repeatedly until the controlling expression evaluates to zero. The loop body refers to the statements executed as long as the controlling expression is nonzero. The loop body may be a simple or compound statement.

Iteration statements take one of three forms:

- the while statement
- the do, or do-while statement
- the for statement

### **The while Statement**

In a while statement, the loop body executes as long as the controlling expression is nonzero. The evaluation of this expression takes place before each execution of the loop body. Thus, the while statement executes zero or more times.

E.g.,

```

while (index < TABLE_SIZE)
    if (table[index] = key)
        break;
    else
        index++;

```

## **The do Statement**

In a do statement, the evaluation of the controlling expression takes place after each execution of the loop body. Thus, the do statement executes at least once.

E.g.,

```
do {
    ch = getchar();
    switch (ch) {
        /* ... */
    }
} while (ch != 'Q');
```

## **The for Statement**

The for statement makes it possible to specify a while statement more succinctly. Except for the behavior of a continue statement in the loop body, the statement

```
for ( ex-1; ex-2; ex-3 ) statement
```

and the sequence of statements:

```
ex-1;
while (ex-2) {
    statement/* continue in for-loop would arrive here... */
    ex-3;
/* ...but continue in while-loop arrives here */
}
```

are equivalent.

Thus ex-1 specifies the initialization for the loop. ex-2 is the controlling expression; it specifies an evaluation made before each iteration. Execution of the loop continues until this expression evaluates to zero. ex-3 specifies an operation that is performed after each iteration.

Both ex-1 and ex-3 may be omitted, or may have any type, including void. If ex-2 is omitted, it is treated as if a non-zero constant had been written  $\text{Æ}$  i.e., the loop condition is always true.

E.g.,

```
int count = 10;

/* will execute forever, since count */
/* never changes */

for (; count < 100; )
    printf ( "%d hello\n", count);

/* assumes count has been initialized;*/
/* will eventually end. */

for (; count < 100; )
```

```
    printf ( "%d hello\n", count++);

/* executes nine times, for count = 1 through 9 */

for ( count = 1; count < 10; count++)
    printf ( "%d hello\n", count);

/* executes forever */

for (;;)
    printf ( "Forever");
```

# CHAPTER 9

## *The Preprocessor*

### Introduction

---

The C preprocessor processes tokens from the source text file before passing these tokens on for syntax checking. This makes it possible to do certain things before compiling the program. In particular, the preprocessor makes it possible to:

- Include other files. These may contain definitions and declarations that can be used in the program.
- Selectively compile certain parts of the code, depending on specific conditions (such as the hardware on which the program is to run).
- Replace macros with other tokens.

Tokens are built, processed and then analyzed for syntax in the following conceptual translation phases, as specified by ANSI C:

1. Physical source file characters are mapped to the source character set. During this phase,
  - A newline character is substituted for each end-of-line characters sequence.
  - Trigraphs are replaced by their corresponding single-character representations.
2. Physical source lines are spliced to form logical source lines, by deleting each backslash-newline pair. Thus, lines that extend over two lines are combined into a single line. (A non-empty source file must end in a newline character, but cannot end with a newline immediately preceded by a backslash character `\` that is, the file cannot end in the middle of a continued line.)
3. The source file is decomposed into preprocessing tokens and white space character sequences. Newline characters are retained. Each comment is replaced by a single white space character, and multiple white space characters are replaced by a single white space character. White space characters in strings and character constants are not affected.

4. Preprocessing directives are executed and macro invocations are expanded. Any header or source files specified with an `#include` preprocessing directive are processed recursively, from phase 1 through phase 4.
5. Escape sequences in character constants and string literals are converted to single characters in the execution character set.
6. Adjacent string literals are concatenated.
7. Preprocessing tokens are converted into (normal) tokens. White space characters separating individual tokens are no longer needed, and are discarded. The tokens remaining are analyzed and translated, according to syntactic and semantic rules of the language. If a token cannot be translated, a syntax error results.
8. All external object and function references are resolved. I.e., the object code is linked to produce an executable program. This linking phase generally occurs outside the compilation process.

E.g., the following sequence of characters:

```
01< <h3/1.2>=x+++b#include <2/1.3x>#define struct.field •
```

forms the following sequence of preprocessing tokens. In the list, individual preprocessing tokens are delimited by a { on the left and a } on the right.

```
{01} {<} {<} {h3} {/} {1.2} {>=} {x} {++} {+} {b}{#} {include} {<2/  
1.3x>}{#} {define} {struct} {·} {field} {·}
```

## Preprocessing Directives

---

A preprocessing directive is a command to the preprocessor. A directive begins with a `#`. This must be the first character on the source line except for white space characters. The directive is terminated by a newline character.

The `#` may be followed by one of the following preprocessor directives:

`define``undef``if``endif``def``ifndef``elif``else``endif``include``line``error``pragma`

Space and horizontal tab characters may appear between the `#` and the directive. Preprocessor directives do not end with a semicolon.

If the `#` is the only token on a line, it has no effect, and is discarded.

After preprocessing, all preprocessing tokens must have the form of a lexical token.

## Defining a Macro

---

The `#define` directive is used to associate a meaningful name with a token sequence, usually a number, identifier, or expression. It may also be used to aid portability and enhance program readability without sacrificing performance.

There are two types of macros. Those without parameters (object-like macros) and those with parameters (function-like macros).

### Object-Like Macros

```
# define identifier replacement-list newline
```

An object-like macro directive defines a macro whose name corresponds to the identifier after the `#define`. This identifier is called the macro name. Each subsequent instance of the macro name is replaced by the replacement list of tokens that forms the remainder of the directive. After this substitution, the replacement list is rescanned for more macro names to replace.

White space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for a macro.

E.g.,

```
#define BUFLen 512
#define TRUE 1
#define forever while (TRUE)
```

In the first example, all occurrences of `BUFLen` (after the one in the definition) are replaced by `512`. In the third example, all occurrences of `forever` are replaced by `while (TRUE)`.

### Function-Like Macros

```
# define identifier lparen identifier-listopt rparen
      replacement-list newline
```

A directive with a macro name followed by a left parenthesis, an optional identifier list, and a right parenthesis defines a function-like macro with arguments.

Such a macro invocation is syntactically similar to a function call. The parameters for such a macro are specified in the optional list of identifiers. The scope of these parameters extends until the newline character that terminates the `#define` preprocessing directive.

The left parenthesis in a function-like macro definition must come immediately after the macro name. That is, there can be no intervening white space or tokens between the macro name and the left parenthesis

Each subsequent instance of the function-like macro name  $\text{\#}$  followed by a ( preprocessing token and terminated by a ) preprocessing token  $\text{\#}$  is replaced by the replacement list given in the macro definition.

The actual arguments specified in the macro occurrence are substituted for the appropriate identifiers in the replacement list.

E.g., given the following two macros:

```
#define square(x)      ((x) * (x))
#define setbit(x, y)  ((x) |= 1 << (y))
```

If a fragment such as

```
square(7.5)
```

is encountered in the source, the fragment is replaced by the following:

```
((7.5) * (7.5))
```

Similarly, if a fragment such as

```
setbit(123, 3)
```

is encountered in the source, the fragment is replaced by the following:

```
((123) |= 1 << (3))
```

### **Redefining a Macro Name**

It is an error to redefine a name already defined as a macro name unless the replacement lists are identical. In the case of a function-like macro the parameters must also be identical in spelling and number.

Two replacement lists are identical if they differ only in the white space in the source. All tokens in one list must be in the second list  $\text{\#}$  with the same spelling and in the same sequence. The amount of white space in the lists is not taken into account when comparing the lists.

E.g., the following redefinitions are valid:

```
#define BUFLen 512
#define max(a, b) ((a)>(b) ? (a) : (b))
#define BUFLen 512
#define max(a,b) ((a)>(b) ? (a) : (b))
```

but the following ones are invalid:

```
#define BUFLen 256
#define BUFLen 100
#define max(x,y) ((x)>(y) ? (x) : (y))
#define max(a,b) ((a)>(b)?(a):(b))
```

## **Scope of Macro Definitions**

A macro definition remains in scope until an `#undef` directive for the macro name is encountered. If no such `#undef` is encountered then the scope extends until the end of the translation unit.

A preprocessing directive of the form:

```
# undef identifier newline
```

removes the specified identifier from the list of identifiers recognized as a macro name. If the specified identifier is not currently defined as a macro name, the `#undef` directive is ignored.

## **Macro Replacement**

During translation phase 4, each identifier is checked to see if it is identical to a defined name. If the token matches a macro name, it is replaced by the body of that macro name as follows:

```
\setlongest{Function-like macros}
```

**Object-like macros** The token is replaced by the tokens making up the replacement list for the macro name

**Function-like macros**

In this case, the macro name must be followed by a ( token. (A function-like macro name token that is not followed by a left parenthesis is not replaced).

The tokens between the matching opening and closing parenthesis constitute the actual parameters of the macro invocation. Individual arguments between the outermost parenthesis are separated by comma tokens. (Comma preprocessing tokens bounded by nested parenthesis do not separate arguments). Within the sequence of preprocessing tokens making up an invocation of a function-like macro, a newline is considered a white space character.

TopSpeed C gives an error if the number of actual arguments in the macro invocation does not match the number of formal parameters in the macro definition.

Given the following legal macro definitions,

```
#define TRUE 1
#define BUFLen 512
#define forever while (TRUE)
#define square(x) ((x) * (x))
#define setbit(x, y) ((x) |= 1 << (y))
```

the following fragments:

```
char buffer[BUFLen];

forever {
    receive(buffer);
    send(buffer);
}
num = square(num);
setbit(flag, 4);
```

expand as follows:

```
char buffer[512];

while (1) {
    receive(buffer);
    send(buffer);
}
num = ((num) * (num));
((flag) |= 1 << (4));
```

## **Argument Substitution**

argument substitution is used to replace the arguments identified when a function-like macro is invoked. With certain exceptions (listed below), a parameter in the replacement list is replaced by the corresponding argument after all contained macros have been expanded. All macros in the argument's preprocessing tokens are completely replaced before such tokens are substituted for the parameters.

A parameter in the replacement list is replaced with the corresponding argument. There is no expansion if that parameter is preceded by a # or ## preprocessing token or is followed by a ## preprocessing token (see Chapter 9).

## **Rescanning and Further Replacement**

After the required parameter substitutions have been made in the replacement list, the resulting preprocessing token sequence is scanned again, for more macro names to replace.

Not all macro names are replaced in subsequent scannings. No substitution is made if the name of the macro being replaced is found during this rescan of the replacement list. This name is also left if encountered by any nested replacements. Once left, such unreplaced macro name tokens are not replaced, even if they are later processed in contexts in which they would otherwise be replaced.

The sequence of tokens resulting from a macro replacement is not available for preprocessing as a preprocessor directive.

E.g.,

```
#define A    A B C
```

```

#define B   B C A
#define C   C A B

A
/* expands to... */
/A B C
/* rescan: A is not expanded... */
A { B C A } { C A B }
/* rescan of B: A and B not expanded... */
A { B { C A B } A } { C A B }
/* no further expansion possible in B */
/* rescan of C: A and C not expanded... */
A B C A B A { C A B C A }
/* no further expansion possible in C */
A B C A B A C A B C A
/* ...which is the result */

```

The following defines a function-like macro whose value is the maximum of its arguments. The advantage over a function is that the code is generated in-line. The possible disadvantage is that one of the arguments is evaluated twice.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound correctly

To illustrate the rules for substitution and reexamination, consider the following sequence:

```

#define pair(x,y) (x->next = y)
#define last(q)   (q->next ? last(q->next) : q)
pair(a,b);
pair(a, pair(b, c));
a = last(c);
a = last(last(b));
pair(a, last(c));
b = last(pair(a, c));

```

Macro replacement produces the following:

```

(a->next = b);
(a->next = (b->next = c));
a = (c->next ? last(c->next) : c);
a = ((b->next ? last(b->next) : b)->next ?
    last((b->next ? last(b->next) : b)->next) : (b-
    >next ? last(b->next) : b));
(a->next = (c->next ? last(c->next) : c));
b = ((a->next = c)->next ?
    last((a->next = c)->next) :
    (a->next = c));

```

The name following the # preprocessing token at the start of a preprocessing directive is not subject to macro replacement, even if the name has been defined as a macro name.

E.g.,

```
#define include error
```

```
#include "inc.h" /* still includes inc.h */
```

The second line still includes the specified file, because the macro replacement specified in the first line is not applied. The preprocessing token sequence that results from macro replacement is not processed as a preprocessing directive even if it resembles one

### **The # Operator**

It's possible to replace a preprocessing token parameter with a string literal token. If the stringize operator (#) is found in the replacement list, it must be followed by a parameter token. The parameter is replaced by a single string literal, which contains the spelling of the preprocessing token sequence for the argument.

**Note:** The tokens for the argument are not rescanned for macro substitutions, as is normally the case.

Every sequence of white space between the argument's preprocessing tokens is reduced to a single white space character in the string literal. Any white space is removed before the argument's first or after its last preprocessing token. Beyond that, the original spelling of each preprocessing token in the argument is retained in the string literal.

Certain character constants require special handling, however, a \ character is inserted before each " and \ character (This is also done for the delimiting . characters)

## **The ## Operator**

---

Before the replacement list for a macro invocation is re-examined for more macro names to replace, each instance of a glue operator preprocessing token (##) in the replacement list is deleted and the token before the glue is concatenated with the token after the glue. (Parameters are handled differently, as described in the next paragraph.) The result of this concatenation must be a valid token. The resulting token is available for further macro replacement. Glue is handled in this way in both object-like and function-like macros

If, in the replacement list, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence (which is not rescanned).

The order of evaluation is left unspecified by the ANSI standard; in TopSpeed C the order is left to right.

To illustrate the rules for creating string literals and concatenating tokens, consider the following sequence:

```

#define str(x) #x
#define xstr(x) str(x)

#define VERSION 2
#define header(v)
    xstr(incv ## v)
#define incfile(v) header(v.h)

str(hello) str(!)
    str(str(?))
    xstr(str(?))
#include incfile(VERSION)

```

This results in:

```

"hello" "!"
    "str(?)"
    "\ " " "
#include "incv2.h"

```

(See Chapter 9 for a full description of the lexical rules governing the characters in a filename in a `#include` directive)

Space around the `#` and `##` tokens in the macro definition is optional

A `##` preprocessing token may not occur at the beginning or at the end of a replacement list for either form of macro definition.

## Conditional Inclusion

---

The preprocessor can be used to selectively include/exclude lines of input source from further processing by the compiler.

```

# if const-expr
group-of-lines1
#else
group-of-lines2
#endif

```

If the `const-expr` is zero, the text `group-of-lines1` is skipped. Instead, the text `group-of-lines2` is processed and passed on to the translator. If the `const-expr` has a nonzero value, the text `group-of-lines1` is processed while `group-of-lines2` is skipped. The `#else` part is optional.

It is possible to nest `#if` directives. The preprocessor matches `#elses`, `#elifs` and `#endifs`.

For nested `#if`, `#else` and `#endif` directives, the `#elif` directive may be used:

```

# if const-exprnewline
    groupopt
# else
# if const-expr newline
    groupopt

```

```
.i.# if;
```

can be replaced by:

```
# if const-expr newline
    groupopt #
elif const-expr newline
    groupopt
```

Preprocessing directives of the forms:

```
# ifdef identifier newline
    groupopt
# ifndef identifier newline
    groupopt
```

check whether or not the identifier is currently defined as a macro name. Their conditions are equivalent to `#defined identifier` and `#if !defined identifier`, respectively.

Each directive's condition is checked in order (from left to right). If the condition evaluates to zero (false), the group that the condition controls is skipped. When skipping, directives are processed only as far as the name that determines the directive, in order to keep track of the level of nested conditionals. The rest of the directives' preprocessing tokens are ignored. Only the first group whose control condition evaluates to nonzero (true) is processed.

If none of the conditions evaluates to true, and there is an `#else` directive, the group controlled by the `#else` is processed. If there is no `#else` directive, all the groups until the `#endif` are skipped.

The expression must be an integral constant expression, and may not contain a `sizeof` operator or a cast. However, the expression may contain unary expressions of the form:

```
defined identifier
```

or

```
defined ( identifier )
```

These evaluate to 1 if the identifier is currently defined as a macro name and to 0 if it is not. An identifier is defined as a macro name if the identifier has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive.

E.g.,

```
#ifdef CHEMISTRY
    #define AVOGADRO 6.024E23
#else
    #define PI 3.1415926
#endif
```

## **Evaluating Constant Expressions**

Prior to evaluation, identifiers currently defined as macro names are replaced in the list of preprocessing tokens just as in normal text. (The exceptions are identifiers modified by `defined`). Any remaining identifiers are replaced with `0L` and each integer constant not already suffixed with `l` or `L` is considered to be additionally suffixed with `L`.

During the evaluation of the expression the usual arithmetic conversions apply, except that `int` and `unsigned int` act as if they had the same representation as `long` and `unsigned long`, respectively. This includes interpreting character constants, which may involve converting escape sequences into characters.

In TopSpeed C the numeric value for these character constants

E.g, the following will compare equal on a processor that uses the ASCII character set:

```
#define LOWER_A  'a'
#define UPPER_A  'A'
#if (UPPER_A - LOWER_A) == 32
  blah_blah
#endif

if ((UPPER_A - LOWER_A) == 32)
  blah_blah;
```

## **Source File Inclusion**

---

A preprocessing directive of the form:

```
# include <h-char-sequence> newline
```

is replaced with the entire contents of the source file identified by the specified character sequence between the `<` and `>` delimiters and by a set of pathnames. (In TopSpeed C, this set is determined by the contents of the TS.RED file.) E.g.,

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

is replaced with the entire contents of the file `stdio.h`. This file is sought in whatever directories are specified for `.H` files in the TS.RED file. The search continues until the file has been found or until all the directories have been checked.

TopSpeed C allows `#include` files to be nested to a depth limited by the maximum number of simultaneously open files that is allowed by the operating system

A preprocessing directive of the form:

```
# include "q-char-sequence" newline
```

causes that directive to be replaced with the entire contents of the source file identified by the specified character sequence between the “ delimiters. In TopSpeed C, this case is treated in the same manner as the previous one: the TS.RED is used to determine which directories to search

Consider a preprocessing directive of the form:

```
# include pp-tokens newline
```

In this directive, pp-tokens is not delimited by a “ or < and >. As a result, the preprocessing tokens found after include in this directive are processed just as in normal text. That is, each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens. The directive resulting after all replacements have been made must match one of the two previous forms

The sequence of preprocessing tokens between a < and a > preprocessing token pair is combined into a single header name preprocessing token.

Tokens are not implicitly concatenated Therefore, the glue operator ## must be used to form a single token between the < and > delimiters,

E.g.,

```
#if __STDC__
    #define header stdio ## . ## h
#else
    #define header extio ## . ## h
#endif
#define incname < header >
#include incname
```

For more complex examples of macro-replaced #includes, see the sections on # and ## operators above.

## Line Control

---

The source file is processed beginning at line 1. The line number of the current source line is one greater than the number of newline characters read or introduced up to that point in the first translation phase.

A preprocessing directive of the form:

```
# line digit-sequence newline
```

causes TopSpeed C to behave as if the line number of the next and subsequent source lines is based from the digit-sequence (interpreted as a decimal integer). The range of possible values is 1..32767.

A preprocessing directive of the form:

```
# line digit-sequence string-literal newline
```

sets the line number as above, and also changes the presumed name of the source file. The new name is specified by the characters contained within the string-literal, which must be a character string literal.

A preprocessing directive of the form:

```
# line pp-tokens newline
```

allows the digit-sequence and string-literal to be generated via macro names. The preprocessing tokens after line on the directive are processed just as in normal text. The directive resulting after all replacements must match one of the two previous forms.

## Error Directive

---

A preprocessing directive of the form:

```
# error pp-tokensopt newline
```

generates an error message of the form:

```
# error directive: pp-tokens
```

This error is treated like any other compilation error.

## Pragma Directive

---

A preprocessing directive of the form:

```
# pragma pp-tokensopt newline
```

provides a mechanism for passing information to the compiler. TopSpeed C has numerous pragmas, which are described in the TopSpeed C User's Manual.

## Null Directive

---

A preprocessing directive of the form:

```
# newline
```

has no effect.

## Predefined Macro Names

---

The following ANSI macros are predefined:

`_LINE_`

This represents the line number of the current source line, a decimal constant.

<code>_FILE_</code>	This is the name of the source file, a string literal.
<code>_DATE_</code>	This represents the date of compilation of the source file. The replacement is a string literal of the form “Mmm\dd\ yyyy”, where the names of the months are the same as those generated by the <code>asctime</code> function, and the first character of <code>dd</code> is a space character if the value is less than 10.
<code>_TIME_</code>	This represents the time of compilation of the source file. The replacement is a string literal of the form “hh:mm:ss”, as in the time generated by the <code>asctime</code> function.
<code>_STDC_</code>	This is defined as the decimal constant 1 if the <code>/A</code> (ANSI) option is specified; otherwise the macro is 0. In addition, TopSpeed C has other predefined macros. These are:
<code>__TSC__</code>	The current version number.
<code>M_I86xM</code>	Memory model, <code>x</code> being <b>S</b> , <b>M</b> , <b>C</b> , <b>L</b> , <b>X</b> , <b>MT</b> or user-defined.
<code>__MSDOS__</code>	Defined if TopSpeed C is running under DOS.
<code>__OS2__</code>	Defined if TopSpeed C is running under OS/2.
<code>__CHAR_UNSIGNED__</code>	Defined if chars are unsigned.
<code>_INLINE</code>	Defined if optimizing for time.
<code>__OS2_MT__</code>	Defined if the multi-thread option is <b>ON</b> .

None of these macro names, nor the identifier defined, may be the subject of a `#define` or a `#undef` preprocessing directive.

## Preprocessor Syntax Summary

---

The following listing summarizes the syntax for preprocessor directives.

```

preprocessing-file:
    groupopt

group:
    group-part
    group group-part

group-part:
    pp-tokensopt newline
    if-section
    control-line

```

```

if-section:
    if-group elif-groupsopt else-groupopt endif-line

if-group:
    # if  const-expr newline groupopt
    # ifdef  identifier newline groupopt
    # ifndef identifier newline groupopt

elif-groups:
    elif-group
    elif-groups elif-group

elif-group:
    # elif  const-expr newline groupopt

else-group:
    # else  newline groupopt

endif-line:
    # endif newline

control-line:
    # include pp-tokens newline
    # define identifier replacement-list newline
    # define identifier lparen ident-listopt )
        replace-list newline
    # undef identifier newline
    # line  pp-tokens newline
    # error pp-tokensopt  newline
    # pragma pp-tokensopt  newline
    #  newline

paren:
    the left-parenthesis character without preceding
    white space

replacement-list:
    pp-tokensopt

pp-tokens:
    preprocessing-token
    pp-tokens preprocessing-token

preprocessing-token:
    header-name (only within a #include directive)
    identifier (no keyword distinction)
    pp-number
    character-constant
    string-literal
    operator
    punctuator
    each non-white space character that cannot be
    one of the above

header-name:
    <h-char-sequence>
    "q-char-sequence"

```

"h-char-sequence:

h-char

h-char-sequence h-char

h-char: any character in the source character set except  
the newline character and >

q-char-sequence:

q-char

q-char-sequence q-char

q-char:

any character in the source character set except  
the newline character and "

newline:

the newline character

pp-number:

digit

. digit

pp-number digit

pp-number nondigit

pp-number e sign

pp-number E sign

pp-number .

# APPENDIX A

## References

ANSI standard: “ANSI x3.159 - 1989 Programming Language C” Validated summary report ref CLPVS/0007/UK

“Draft Proposed American National Standard for Information Systems Æ Programming Language C,” 7 December 1988.

“Rationale for Draft Proposed American National Standard for Information Systems Æ Programming Language C,” 14 November 1988.

“The C Reference Manual” by Dennis M. Ritchie, a version of which was published in The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., (1978), 2nd edition, Prentice-Hall, Inc. (1988).

“1984 /usr/group Standard,” by the /usr/group Standards Committee, Santa Clara, California, USA (November, 1984).

“American National Dictionary for Information Processing Systems,” Information Processing Systems Technical Report ANSI X3/TR-1-82 (1982).

“ISO 646-1983 Invariant Code Set.”

“IEEE Standard for Binary Floating-Point Arithmetic,” (ANSI/IEEE Std 754-1985).

# APPENDIX B

## Implementation Defined Behavior

The ANSI C standard defines implementation-defined behavior as behavior which depends on the characteristics of the implementation. It is assumed that the program construction and data are correct.

The implementation-defined features for TopSpeed C are described here. The features are numbered in accordance with Appendix F3 (Implementation defined behavior) in the ANSI C standard.

### 1. Translation

- Diagnostics are reported either as warnings or errors as follows:

```
(File.c line,col) Error:message
(File.c line,col) Warning:message
```

The messages are documented in Appendix F of this manual.

### 2. Environment

- If the main function is declared as

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

the array elements `argv[0]` to `argv[argc - 1]` contain pointers to strings. These strings contain the command line arguments for the program. The case of the arguments is preserved. The command line arguments are separated by one or more space characters, except for text enclosed in double quotes (which will count as a single argument)

`argv[0]` is a pointer to the program name, except under version 2 of DOS where this element is not available. `argv[0]` is a NULL pointer in that version.

The program invocation

```
myprog First "Second plus more"
```

results in the following values for `argv`:

```
argv[0] points to "myprog"
argv[1] points to "First"
argv[2] points to "second plus more"
```

`env`, an optional third argument to `main`, is an array of pointers to strings containing the operating system environment variables. The last pointer in `env` is a NULL pointer.

- The operating system determines what is to be considered as an interactive device. Thus under DOS and OS2 the console, printer

and serial port (if present) are treated as interactive devices, but disc files are not.

### **3. Identifiers**

- In TopSpeed C, all characters in an identifier are significant. This is true for all identifiers, both during compilation and linking.

Note     **The maximum token length (and thus identifier length) is 1024 characters.**

- Case is significant for external linkage.

### **4. Characters**

- The translation and execution character sets consist of the ASCII character set (character values 0-127) plus the IBM character set (character values 128-255)
- TopSpeed C treats multibyte characters in a simple way: a multibyte sequence consists of only one character (ie there is only an initial shift state) Thus the “extended” character set has the range 0 to 255.
- There are 8 bits in a character in the execution character set
- Source characters are mapped directly without change onto execution characters. Thus any string literal will be the same in the source character set as in the execution character set.
- Any integer character constant that contains a character or escape sequence not represented in the basic execution character set, or the extended character set for a wide character constant, will be represented as an int if possible, otherwise the value has overflowed. A warning is issued if an escape sequence cannot be represented as a char.
- An integer character is a sequence of 1 to 4 characters. If the sequence has one character, it is mapped to its corresponding ASCII value. If there are more characters, they are mapped to an integer type that can represent the value. Each character is mapped to a byte in this integer type.

For example the constant ‘abcd’ is represented as a long with the hexadecimal value 0x61626364. If a character constant contains an escape sequence whose value is not representable as a char, a warning is issued and the value of the escape sequence is truncated to a char value. The same applies to wide character constants.

- The current locale used to convert multibyte characters into corresponding wide character (codes) for a wide character constant is the C locale.

- A plain char is signed by default. This can be changed to an unsigned char using the pragma option (`uns_char=>on`).

The char type is signed. All characters in the source character set have positive values when stored in a char variable; all other values (that is, those in the range 128 to 255) will have negative values.

## **5. Integers**

- The sets of values represented by various integer types are described in the `limits.h` file. See also page 44 for the representation of two's complement numbers.
- If a value with integral type is converted to a signed integer with smaller size, the high-order part of the converted value is discarded. There is no change in the actual bit pattern. If a value of unsigned type is converted to a signed integer of the same size, the bit pattern is unchanged.
- Bitwise operations for signed integers are performed as for unsigned integers except for '`<<`': see below
- The remainder has a negative sign when only one of the operands is negative. The remainder is positive when neither or both operands are negative.
- The right shift on a negative-valued signed integral type preserves the sign: that is, the sign bit is propagated to the right.

## **6. Floating point**

- The representation of floating point numbers conforms to the IEEE standard. The set of values is defined in `float.h` include file.
- When a value of integral type is converted to floating type, and the value being converted cannot be represented exactly, the result is the nearest value smaller than the original value.
- When a double (or long double) is converted to a float, and the value cannot be represented exactly, the result is the nearest value smaller than the original value.

## **7. Arrays and pointers**

- The `sizeof` operator yields a constant value that has type `size_t`. This type is defined in the header file `<stddef.h>`. In TopSpeed C, `size_t` has type unsigned int.
- In TopSpeed C pointers can have two sizes. A pointer can be a far (4 byte) or a near (2 byte) pointer. The size depends on the memory model used, as described in greater detail in the User's Guide. In a conversion between an integer and a pointer, the pointer is seen as having an unsigned integer type corresponding

to its size

- A near pointer can be converted to an unsigned int or to an int without loss of information. No change takes place in the bit pattern. A far pointer can be converted to an unsigned long or to a long without loss of information. If a far pointer is converted to an int or to an unsigned int, the segment part is discarded.
- An integer can be converted to a pointer. An int or unsigned int can be converted to a near pointer without change in the bit pattern. If the integer is converted to a far pointer, the segment part will contain the default data segment (or default code segment if a code pointer)

A long or unsigned long can be converted to a far pointer without change in value. If a long is converted to a near pointer, the high-order word is discarded.

- The result of subtracting two pointers has the type int, which is also defined as the typedef ptrdiff\_t in the header file <stddef>

Note     **The result of subtracting two huge pointers has type long int.**

## **8. Registers**

- TopSpeed C is an optimizing compiler that does its own optimal register allocation for variables. Therefore the register storage class does not have any effect except that the address of operator (&) cannot be applied to such an object.

## **9. Structures, Unions, Enumerations and Bit-fields**

- If a member of a union (m1) is accessed after a value has been assigned to a different member (m2), then the assigned value will be interpreted according to the type of m1. Note that no conversion takes place; rather, the bit pattern is simply interpreted according to the type of m1.
- Except for bit-fields, all members of a struct are byte aligned. That is, the members are packed as tightly as possible.
- The high-order bit in an int bit-field is treated as a sign bit.
- Bit-fields are allocated within a word from the least significant bit
- Several bit-fields can be packed into a word. If insufficient space remains for a bit-field, it is put in the next word (i.e., a bit-field will never overlap two words, unless the bit-field is larger than 16 bits). int and unsigned int bit-fields are word aligned, while char and unsigned char bit-fields are byte aligned
- All enumeration types are implemented as the type int.

## **10. Qualifiers**

- An object of volatile qualified type is considered to be accessed when its value is read or a new values is written into it.

## **11. Declarators**

- There is no explicit limit to the maximum number of declarators that can modify an arithmetic, structure or union type

## **12. Statements**

- TopSpeed C has no explicit limit for the number of case values in a switch statement. Keep in mind, however, that the compilers can run out of member for the internal representation of a very large number of cases.

## **13. Preprocessing directives**

- Character constants occurring in the conditional expression for a #if command are treated in a manner similar to the way in which character constants are handled outside the preprocessor.
- Such character constants may have negative values
- The file name specified in a #include directive is searched for via the redirection file (ts.red). (See the User's Guide for a description of the redirection file.)
- TopSpeed C treats both the double quote form and the angle bracket form of file name specification in the same way: they both use the redirection file.
- The file name given in both the double quote form and the angle bracket form must be a valid DOS or OS/2 file name (full name, partial name or tail name). I.e., no "translation" of the specified name takes place.
- For pragma directives see the TopSpeed Developer's Guide.
- The date and time of translation are always available

## **Additional**

The following additional implementation-dependent features should also be noted:

- In the preprocessing phase, sequences of whitespace characters (except for newline) are replaced by a single space character.
- The characters ' and " cannot for preprocessing tokens on their own. A compilation error is issued if this occurs.
- #include directives can be nested up to a limit set by the operating system. The operating system (DOS or OS/2) has an

upper limit as to how many files can be open at the same time.

- An error message is issued if any character other than the characters specified in Chapter 3: ‘C language elements’ appears in the source file, unless such a character occurs in a character constant, string constant or in source that is skipped because of conditional compilation.

# APPENDIX C

## Undefined Behavior

The standard defines undefined behavior as behavior for which no requirements are imposed. The behavior in this case is assumed to involve one or more of the following:

- use of a nonportable or erroneous program construct
- use of erroneous data
- use of indeterminately-valued objects

The behavior in the TopSpeed C environment when such cases arise is summarized in this appendix.

1. If the main function returns without specifying a value, the termination status given to DOS or OS2 is undefined.
2. If the source contains a character not contained in the source character set, an error message will be given  $\text{\AE}$  unless the source is skipped by conditional compilation.
3. If an identifier has both external and internal linkage within the same translation unit, the identifier has internal linkage.
4. The compiler checks as far as possible that all declarations referring to the same object or function have compatible types. Incompatible declarations not found by the compiler (e.g. if the declarations occur in different translation units) will be found by TopSpeed linker which has type-safe linking (i.e., the object files contain type information that is checked by the linker).
5. If the  $\backslash$  in an escape sequence is followed by a character that is not defined as an escape sequence in the ANSI standard, the  $\backslash$  character is ignored. E.g. the escape sequence  $\backslash J$  yields the character J.
6. Adjacent string literals are concatenated. This is also true if one is a character string literal and the other is a wide character string literal.
7. Identical string literals are distinct. A string literal can be modified; however, this is not recommended since future versions could put string literals in read only memory.
8. Function calls. By default, TopSpeed C uses an optimizing calling convention: parameters are passed in registers, and the called function will pop off any parameters are passed on the stack. This requires the number of arguments in a call to match the number of formal parameters in the function definition. This constraint is checked by the compiler if possible; otherwise the

type-safe linking will spot any inconsistencies and will issue a warning

When interfacing to functions in externally precompiled (or assembled) object files that use traditional C calling conventions, you must make sure to declare these functions as having the correct calling convention. This can be done either by using the `cdecl` keyword or the `c_conv` pragma. E.g.,

```
int cdecl search(int *, int);  
/* interface to externally */  
/* compiled function */
```

If a function takes a variable number of parameters, it should be properly prototyped with the ellipsis terminator. E.g.,

```
int f(int n, ...);
```

9. If a pointer has an invalid value assigned to it, and the indirection operator `*` is applied to it:
  - Under OS2 a General Protection error will result.
  - Under DOS some random memory location (corresponding to the pointer value) will be referenced. If a value is assigned to the specified location, the result is unpredictable

10. If the second operand of a division or remainder operator is of integral type and has the value zero, an error is reported.

11. If the right hand operand of a shift left operator is negative or has a value greater than the word width (16), the result will be 0.

If the right hand operand of a shift right operator is negative or has a value greater than the word width, the result will be 0 if the left operand is unsigned, or else signed with sign bit 0; if the left operand is signed and the sign bit is 1, the result is -1

12. If an actual argument for a macro call consists of no tokens, then “nothing” is substituted for the formal argument in the replacement list

# APPENDIX D

## TopSpeed C Extensions.

This appendix summarizes the extensions to the ANSI C language definitions that are provided by the TopSpeed C implementation

1. TopSpeed C allows `env` as an optional third parameter for `main`. `env` is an array of pointers to strings. The strings contain environment variables. See Chapter 2.
2. TopSpeed C provides a nested comments option and the pragma option (`nest_cmt=>on`) to get around the restriction on nested comments. See Chapter 3.
3. TopSpeed C allows `char` and `unsigned char` as bit-field types. See Chapter 6.
4. TopSpeed C provides several extensions in the way in which declarators work for various types and in the flexibility declarators have. Such language extensions can be disabled by using the pragma option (`lang_ext=>off`). See Chapter 6.
5. TopSpeed C has a special pointer type, called a relative pointer. A relative pointer is a near (16-bit) pointer; however, instead of having `DS` or `CS` as the segment value, you can specify the segment yourself in the pointer declarator. See Chapter 6.
6. TopSpeed C provides a variety of non-ANSI C keywords. These may be used to modify declarations of variables, pointers and functions. These keywords can be disabled by the ANSI keywords only pragma option (`ansi=>on`). See Chapter 6.

`cdecl` `pascal` `near` `far` `huge` `interrupt` `inline`

7. TopSpeed C allows you to initialize functions, for the purpose of defining in-line machine code. See Chapter 6.
8. TopSpeed C has a `#pragma` option (`ansi=>on`) extension in which a cast yields an lvalue if the expression is an lvalue. See Chapter 7.
9. TopSpeed C allows a mixture of pointer and integer operands for the relational operators. As a result, an expression such as

`p > 0`