SABER-C APPENDIX INFORMATION

Version 1.2

APPENDIX A: Warnings and Errors

1. RUN-TIME WARNINGS

1.1. Undefined arithmetic operations

0 Not used.
1 Undefined right shift.
2 Undefined left shift.
3 Over/underflow from unary minus expression.
4 Over/underflow from minus expression.
5 Over/underflow from plus expression.
6 Over/underflow from multiplication expression.
7 Over/underflow from division expression.
8 Over/underflow from modulo expression.

1.2. Undefined/illegal pointer operations

9 Cannot compare pointers which point to different items.
10 Subtracting pointers which point to different items.
11 Dereferencing a pointer that is out of bounds.
12 Questionable index into array.
13 Initializing a bad pointer.
14 Storing a bad pointer.
15 Copying a bad pointer.
16 Assigning a bad pointer in a structure/union.
17 Using a bad pointer in a comparsion.
18 Using a pointer which points into freed memory.
19 Pointer holding non-aligned address should not be dereferenced.

1.3. Enumerator warnings

20 Value is not a member of enumerated type.
21 Retrieving illegal enumeration value.
22 Assigning illegal enumeration value.

1.4. Losing information during converions/assignments

23 Losing information during conversion to a bitfield.
24 Losing information during assignment.
25 Losing information during float assignment.
26 Losing information due to sign extension.
27 Losing information during bitfield assignment.
28 Losing information during automatic conversion.
29 Losing information during call to function.

1.5. Function warnings

30 Calling function with too many arguments.
31 Calling function with too few arguments.
32 Signed vs. unsigned type mismatch during call to function.
33 Benign type mismatch during call to function.
34 Serious type mismatch during call to function.
35 Passing illegal enumeration value to function.
36 Function call is passing a bad pointer.
37 Function call is passing a bad pointer in a structure/union.

38 Calling function which contains unresolved references.
39 Calling compiled function which contains unresolved references.
40 Signed vs. unsigned type mismatch during return from a function.
41 Benign type mismatch during a return from a function.
42 Serious type mismatch during a return from a function.
43 Returning garbage from a function.
44 Returning a pointer to an auto or formal variable.
45 Returning a pointer that is out of bounds.

1.6. Memory allocation warnings

46 Cannot free memory address (not within data space).
47 Cannot free memory address (not currently allocated).
48 Cannot free memory address (memory in use, but not via alloc).
49 Cannot free memory address (not at start of allocated block).
50 Cannot reallocate memory address (not within data space).
51 Memory has been allocated since something was freed.
52 Cannot reallocate memory address (not currently allocated).
53 Cannot reallocate memory address (not at start of allocated block).

1.7. Storage warnings

54 Attempting to retrieve an object as a different type
55 Using an object which has not been set.
56 Cannot assign to a read-only address.
57 Using a value that has been corrupted.

1.8. Variable warnings:

58 Using a variable which contains unresolved references.
59 Formal variable should not be used.

2. RUN-TIME ERRORS

2.1. Function/Variable errors

60 Undefined variable.
61 Calling undefined function.
62 Cannot use formal variable as lvalue.
63 There is no function to execute at specified address.
64 Recursion error. Too many function calls.

2.2. Floating point errors

65 Modulo by 0 error.
66 Division by 0 error.

2.3. Addressing errors

67 Dereferencing a pointer that is out of bounds.
68 Illegal index into array.
69 Cannot select a field which is past end of a structure/union.
70 Bad address.
71 Not enough room to copy required number of bytes to specified address.
72 Not enough room to assign to structure/union.

2.4. Miscellaneous errors

73 Out of memory.
74 Cannot long jump back to a function that has already returned.

75 Long jump error.

3. LOAD-TIME WARNINGS AND ERRORS

3.1. I/O errors:

200 File *’name’* cannot be opened for input.
201 The last line of text is incomplete and is being ignored.

3.2. Illegal characters:

202 Null character was seen.
203 Illegal character: decimal value is *’D’*; prints as *’C’*’
204 The ’#’ is misplaced. It should be the first token on the line.
205 The ’\’ is misplaced. It should be the last character on the line.

3.3. Illegal constant formats:

206 Arithmetic constants must terminate with a puctuator or white space.
207 Decimal points are not allowed in hexadecimal or octal constants.
208 Hexadecimal digits are required after the ’0x’ prefix.
209 Digits ’8’ and ’9’ are not allowed in octal constants.
210 The floating point constant has no exponent.
211 Float constants must contain a decimal point or an exponent.
212 The character constant is empty. (For the value of ’ use ’\’’)
213 Character constants with multiple characters are non-standard.
214 The character constant is too long. (Maximum length is 4 chars.)
215 Newlines are not allowed in string literals and character constants.

3.4. Illegal escape sequences:

216 Escape sequence is unknown.
217 Hexadecimal digits should follow the ’\x’ escape.
218 Escape sequence in string literal has null value.

3.5. Lexical constant overflow:

219 Constant causes overflow.
220 Integer constant (*N*) exceeds maximum unsigned long (*M*).
221 Decimal constant (*N*) exceeds maximum signed long (*M*).
222 Escape sequence (*N*) exceeds maximun value of an unsigned char (*M*).

3.6. Improper comments:

223 ’/*’ occurred inside a comment.
224 The comment was not terminated.

3.7. Preprocessing errors:

225 Illegal directive syntax near *’token’*.
226 Directive *’name’* is undefined.
227 An infinite loop of ’#include’ directives was detected.
228 Macros in ’#include’ must be either ’"file"’ or ’<file>’.
229 Illegal ’#include’ file name.
230 ’#include’ file *’name’* is not a source file.
231 Ignoring text from *’token’* in directive.
232 ’#endif’ has no matching ’#if’, ’#ifdef’, or ’#ifndef’.
233 An unbalanced number ’#if’, ’#ifdef’, or ’#ifndef’ occurred.
234 Floating point constants are not allowed in ’#if’ or ’#elif’ directives.
235 String literals are not allowed in ’#if’ or ’#elif’ directives.

3.8. Macro expansion errors:

236 Macro parameter *'name'* was repeated.
237 Only 127 macro parameters are allowed.
238 The '(' is missing after macro *'name'*.
239 Recursion was detected during the expansion of macro *'name'*.
240 Macro *'name'* requires only *N* parameters; extras are being ignored.
241 Macro *'name'* requires *N* parameters, but only *M* are given.
242 EOF occurred while reading arguments to macro *'name'*.

3.9. Syntax errors:

243 Illegal declaration syntax near *'token'*.
244 Illegal statement syntax near *'token'*.
245 Illegal expression syntax near *'token'*.
246 Illegal #if expression syntax near *'token'*.

3.10. Illegal statements:

247 'break' is not in a loop or switch.
248 'continue' is not in a loop.
249 'return' is not in a function.
250 No 'case' occurred in the body of the 'switch'.
251 'case' is in not in the body of a 'switch'.
252 'case' value *N* was repeated.
253 'default' is in not in the body of a 'switch'.
254 'default' was repeated in 'switch'.
255 The result of operator *'op'* was not used.
256 The statement cannot be reached.

3.11. Illegal expressions:

257 A constant expression is required.
258 A constant expression or static address is required.
259 The *'op'* operator is not allowed in *expr* expressions.
260 Illegal argument type.
261 Questionable argument type.
262 The function both returns with a value and returns without a value.
263 Function both returns with a value and exits through the bottom.
264 The function was declared as returning a value, but never did.
265 Calling function *'name'* with illegal parameter. Argument *#N* has type *(typename)*.
266 'case' requires a constant expression.
267 Objects of type *(typename)* cannot be assigned.
268 An lvalue (addressable object) is required.
269 Const objects cannot not be assigned.
270 The address of register variables is undefined.
271 The address of a bitfield is undefined.
272 The size of type *(typename)* is undefined.
273 String literals are not allowed in constant expressions.
274 Casts to type *(typename)* are not allowed in constant expressions.
275 The conditional expression for *'op'* is always true/false.
276 The expression for *'switch'* is a constant.

3.12. Undefined identifiers:

277 *'name'* is undefined.
278 *'name'* is currently not in scope.
279 *'name'* is currently multiply defined.
280 *'name'* is not a member of type *(typename)*.
281 Tag *'name'* is currently undefined.
282 Label *'name'* was not defined.

283 Variable *'name'* has undefined type *(typename)*.
284 Static variable was never defined.

3.13. Unused variables:

285 Formal parameter *'name'* was not used.
286 Automatic variable *'name'* was not used.
287 Static variable *'name'* was not used.
288 Label *'name'* was not used.
289 *'name'* was set but not used.
290 Automatic variable *'name'* may be used before set.

3.14. Improper type specifiers:

291 Pointers to bitfields are not allowed.
292 Pointers to arrays of unspecified dimension are of questionable use.
293 Arrays of bitfields are not allowed.
294 Arrays of functions are not allowed.
295 Arrays of void are not allowed.
296 Only the innermost array dimension may be ommitted.
297 Functions returning arrays are not allowed.
298 Functions returning bitfields are not allowed.
299 Functions returning functions are not allowed.
300 The ‘const’ specifier was repeated.
301 The ‘volatile’ specifier was repeated.

3.15. Illegal bitfield declarations:

302 Bit field *'name'* was declared with a length of zero.
303 Bit fields of type *(typename)* are not allowed. Type (int) is being substituted.
304 Bitfield sizes of type *(typename)* are not allowed.
305 Bitfield size *N* is too large. The maximum size (*M*) is being used.
306 Bitfield size *N* is too large for type *(typename)*. Type (int) is being substituted.

3.16. Declaration errors:

307 Storage class *'stor'* is not allowed in formal parameter declarations.
308 Storage class *'stor'* is not allowed in external variable declarations.
309 Type specifier *'spec'* cannot be combined with type *(typename)*.
310 Variables cannot have the void type.
311 Variables cannot have a bitfield type.
312 Automatic variables with storage class *'stor'* cannot have type *(typename)*.
313 The body of the type definition is empty.
314 All array dimensions must be specified.
315 Array dimensions of type *(typename)* are not allowed.
316 Array has zero dimension.
317 The declaration is not at the top of the block.
318 The declaration has no effect.
319 *N* fields were declared. Only 255 are allowed.
320 Type was declared with size *N*. The maximum allowed is 2147483647.
321 Cannot allocate space for *'name'*.

3.17. Illegal parameter declarations:

322 Formal parameters with storage class *'stor'* cannot have type *(typename)*.
323 The formal parameter names are being ignored.
324 The function parameter list has an illegal format.
325 *'name'* is not in the formal parameter list.
326 The prototype for function *'name'* has no parameter names.

3.18. Initialization errors:

327 Variable *'name'* has already been initialized.
328 Function *'name'* has already been defined.
329 Object cannot be initialized.
330 Use '=' to initialize data variables.
331 Use '{ ... }' for static variables with aggregate types.
332 The '{ ... }' construction cannot be use with automatic aggregates.
333 The '{' does not occur on an aggregate boundary.
334 The initializer list is empty.
335 Too many initializers are given. Only *N* are allowed.
336 The string literal is too long. Only the first *N* chars are being used.
337 The initializer is not a static address or constant.
338 The static initialzer involves an illegal combination of addresses.

3.19. Redefinition errors:

339 Incorrect redefinition of *'name'*.
340 Function *'name'* was redeclared.
341 Variable *'name'* is being redefined with type *(new_type)*. It was previously defined with type *(old_type)*.
342 Formal parameter *'name'* was repeated.
343 Enumeration constant *'name'* was repeated.
344 Label *'name'* has already been defined.
345 Member*'name'* has already been declared.
346 Tag *'name'* is already defined.
347 Function *'name'* has already been used without a prototype in scope.

3.20. Linking errors:

348 Variable *'name'* is being redefined with type *(new_type)* and size *new_size*. It was previously defined with type *(old_type)* and size *old_size*.
349 Variable *'name'* is being redefined as text. It was previously defined with data type *(old_type)*.
350 *'name'* has already been defined as unblocked.
351 The current extern declaration for static *'name'* is being ignored.
352 The previous extern declaration for static *'name'* is being ignored.
353 *'name'* is the name of a built-in saber command. To use the *'name'*, first rename the saber command.
354 Cannot allocate *N* bytes of memory for variable *'name'*.
355 Function *'name'* is too large.

3.21. Intentional bugs (suppressed by default):

356 Non-ANSI syntax: Missing ';' after last field declaration.
357 The result of a conditional expression (?:) was used an an lvalue.
358 The result of a cast expression was used an an lvalue.

APPENDIX B: Hints on Using Saber-C

1.1. Saving Keystrokes

All **saber** commands are invoked through C functions calls. This requires the user to type numerous parentheses, quotes, and semi-colons. Since the environment includes a full C preprocessor, the user can define macros for the commands that are used most frequently. The example below defines a single-letter macro that can be used to call the next command.

```
1 -> #define N next() ;
```

Now the user can enter N (followed by a <Return>) to execute the command.

Many **saber** commands take a string as an argument. The new '#' preprocessor operator (part of the ANSI draft standard) can be used to avoid typing double quotes around these arguments, When a macro parameter is preceded by a '#', its text becomes quoted. (See the appendix on the ANSI draft standard for more information.)

```
1 -> #define ld(file) load(# file) ;
2 -> ld(source.c)
Loading: source.c
(int) 1
3 ->
```

In the example above, the argument source.c is quoted, so that the load command is called with the argument "source.c".

Frequently entered commands can be placed in a file and executed by including the file with the #include directive. When a file is included, its contents are read just as if the user had typed them from the keyboard. Commands that should be executed at startup should be placed in the startup file .saberinit.

1.2. Loading Files

Saber does not yet recognize any comments in source code that are meant to suppress **lint** warnings (e.g. /*VARARGS1*/). However, it is possible to suppress warnings for any source code that has been #ifdef'd **for lint by adding** "-Dlint" **to the** load_options **option.**

```
1 -> setopt("load_options","-Dlint") ;
```

To load up an entire project in **saber**, it often helps to use an existing makefile. One method of using a makefile is to add a dependency to create a script file that can be included into **saber**. Here is a simple example of an addition to a makefile:

```
saber:
    rm -rf saber.ld
    echo "setopt(\"load_options\",\"$(CFLAGS)\");" > saber.ld
    echo "load( \"$(SRCS)\"); " >> saber.ld
    echo "load( \"-l$(LIB)\"); " >> saber.ld
    echo "Script 'saber.ld' for Saber-C is finished."
```

After the command 'make saber' has been executed, the script can be executed by including it into the **saber** environment with the #include **directive.**

```
    1 -> #include "saber.ld"
```

1.3. Customizing for emacs in a windowing environment

If the shell environment variable **EDITOR** is set to something other than **vi**, **saber** assumes the editor is of the **emacs** variety. Unfortunately **saber** cannot distinguish between the many different dialects of **emacs**, so it configures itself for **gnuemacs**.

The implication of this assumption is that the option that describes the editor command sequence to move between one file and another is set to the gnuemacs-specific sequence of

```
    "C-xC-fC-u0C-k%s\nM-<C-u%dC-nC-p"
```

Roughly translated, this reads: visit-file, erase-default-name, %s, goto-top, %d, go-down, go-up-one. **Saber** provides the filename and line number in place of the "%s" and "%d" wherever they occur in this sequence.

Other versions of emacs have a slightly different sequence to switch files and jump to a specific line. In order to have **saber** understand your dialect of emacs, you must place the proper sequence in this option. This should be done in the system-wide `.saberinit` file or in the user's `.saberinit` file.

To provide command line switches for when emacs is invoked, set the option `edit_startup`. If a "+%d %s" is present, **saber** will substitute the starting line number and filename. Note: if your version of emacs opens its own window, you must tell it not to, usually with the -nx or -nw switch.

APPENDIX C: Portability

This document gives a brief summary of the major differences between **saber** and **cc**. **Saber** was designed according to the provisions of the ANSI Draft Proposed C Standard (X3J11/85-044), which extends and slightly modifies the language described by Kernighan and Ritchie in *The C Programming Language* , (Prentice-Hall, 1978). This means that **saber** recognizes everything that **cc** does with the few exceptions listed below. **Saber** also supports several new additions to the language which are described in the ansi reference manual page.

1.1. Changes to C

New Keywords:

Three keywords have been added to the language. They are const, signed, and volatile. Their meaning is explained in the ansi manual page. As with all C keywords, these identifiers are reserved; they may not be used to name variables, tags, labels, typedefs, enumeration constants, or members of structures or unions.

Static Variables:

The ANSI Draft Standard requires that when a static variable is forward referenced, the storage class static must appear in all the declarations for that variable. The following example is accepted by **cc** but produces a warning in **saber**.

```
extern int i ; /**forward reference **/
int *p = &i ;
static int i = 1 ; /**definition **/
```

However, the proper declaration for **saber** is rejected by **cc**.

```
static int i ; /**forward reference **/
int *p = &i ;
static int i = 1 ; /**definition **/
```

String Literals:

Saber treats string literals as read-only data. If a modifiable array of characters is required, a variable should be used. For example:

```
/**Replaces the trailing 'X' with a version number **/
file_name = "tmpX";
file_name[3] = version ;
```

must be rewritten as:

```
static char template[] = "tmpX" ;
file_name = template ;
file_name[3] = version ;
```

Saber concatenates adjacent string literals into one string. This is useful for creating long strings without using backslashes. When a backslash is used, the remaining text must begin at the beginning of the next line.

```
printf("This is the first sentence in a long string.\n\
Using a backslash to continue a string is awkward.\n") ;
```

By using string literal concatenation, the program is made more readable.

```
printf("This is the first sentence in a long string.\n"
     "This string will be appended  to the first.\n");
```

This is new feature which is not supported by **cc**.

Labels:

Saber places labels and variables into separate *name spaces*. This means that a label and a variable with the same name may be visible at the same time, as in the following example.

```
{
   int error = 0 ;
   ...
   if(error) goto error ;
   ...
error: /** Here on an error. **/
   exit(1) ;
}
```

With **cc** the example above will cause a redeclaration error, since both labels and variables share one set of names.

Long int:

The Draft Standard specifies that `int` and `long int` are distinct types. However **cc** treats `int` and `long int` as being identical. In deference to the compiler, **saber** does the same, even though this contradicts the Draft Standard.

1.2. Intentional Bugs

There are several bugs in **cc** which over the years have crept into a great deal of code and have become *de facto* features. Most of these bugs have intentionally been reproduced in **saber** to allow greater compatibility with existing code.

The first two bugs involve lvalues (objects that may be assigned a value). Many compiler will consider the result of a cast to be an lvalue, if both the type of the cast and the type of object are integers or pointers of the same size.

```
int *p ;
++(int)p ; /* Adds 1 (not 4) to 'p' */
```

According to the Draft Standard the proper form of the previous example should be:

```
int *p ;
++*(int *)&p ; /* Adds 1 (not 4) to 'p' */
```

The compiler will also allow the result of a conditional expression to be used as an lvalue, if the expression being tested is a constant.

```
int i, j ;
((1 > 0) ? i : j) = 3 ; /* Assigns to 'i' (not 'j') */
```

Finally, the compiler allows the semi-colon after the last field declaration in a tag definition to be omitted.

```
structs
{
   int i ;
   double d /* Missing last ';' */
} ;
```

Saber reproduces the compiler's behavior described above. Warnings are generated when any of the bugs is detected, but the warnings are suppressed by default. To have **Saber** complain about these three bugs, unsuppress the warnings. (Refer to Appendix C for a listing of warning and error messages. Refer to chapter 3 of the or the for a description of the suppress and unsuppress commands.)

1.3. Porting

To assist in the few areas where **cc** and **saber** differ, **saber** predefines two macros called __SABER__ and SABER. The macro SABER1_2 is also defined under **saber** version 1.2.

APPENDIX D: ANSI Draft Proposed C Standard

This document gives a brief summary of additions made to the C programming language by the Draft Proposed C Standard (X3J11/85-044), which was released by the American National Standards Institute (ANSI) for public review on April 30, 1985. The Draft Standard extends and slightly modifies the language described by Kernighan and Ritchie in *The C Programming Language*, (Prentice-Hall, 1978). The proposed changes to the language may break some existing code. The portability manual page summarizes those changes which will most likely affect programmers who are currently using the C compiler **cc**. The additions to C, however, overcome some of the deficiencies which currently plague the language. Except where noted below, all of these new features are supported by **saber**.

1.1. Additions to C

Prototypes:

The most useful addition to the language in the Draft Standard is the prototype. A prototype is a function declaration that specifies the type of the parameters as well as the result type. Here are some examples of prototypes:

```
/* 'strlen' is a function which returns an int */
/* and takes one parameter of type pointer to */
/* char. */
int strlen(char *);

/* 'main' is a function which returns nothing */
/* and takes an int and the address of an array */
/* of pointers to a char. */
void main(int argc, char *argv[]);

/* 'handler' is a function returning a pointer */
/* to a function which takes an int and returns*/
/* nothing. 'handle' takes two parameters. */
/* The first is an int. The second is a pointer */
/* to a function which takes an int and returns*/
/* nothing. */
void (*handler(int sig, void (*func)(int)))(int);
```

The formal parameter names given in a prototype declaration are optional. A function may be defined using a prototype, in which case, of course, the parameter names must be specified. The storage class register is allowed in the parameter declarations. Using the prototype format, strlen might be defined as:

```
int strlen(registerchar *str)
{
    register int i = 0;
    while(*str++) i++;
    returni;
}
```

If a function takes a variable number of arguments, then an ellipsis is used after the last specified parameter. For example, the prototype for printf reads as follows:

```
/* 'printf' is a function which returnsan */
/* int and takes one parameter of type */
/* pointer to char and zero or more other */
/* parameters of unspecified type. */
int printf(char *format, ...);
```

The advantage of the function prototype is that it permits load-time checking of the types of the actual parameters being passed to a function. (Note that **saber** also performs run-time checking.)

```
1 -> extern int printf(char *, ...) ;
2 -> printf(1) ; /* Draws a warning. */
```

In addition, the ANSI Draft Standard specifies that when a prototype for a function is in scope the standard automatic conversions are performed on the arguments passed to that function. In the following example, the integer value 100 is converted to double before being passed to the function sqrt.

```
1 -> extern double sqrt(double) ;
2 -> sqrt(100) ; /* 100 is converted to 100.0*/
```

Prototypes are very useful since they facilitate type checking, but the automatic conversion performed on the arguments can be annoying since in is not compatible with **cc**. For this reason, the ansi option is provided. Automatic conversion of arguments passed to a function declared with a prototype only occurs when the ansi option is set. If this option is not set, then no conversion will occur and a warning will be generated issued when the call to the function is seen.

```
1 -> /* Turn automatic conversion on */
2 -> setopt("ansi") ;
3 -> double inc(double d) { returnd + 1.0;}
4 -> inc(3) ; /* 3 is converted to 3.0 */
(double) 4.000000e+00
5 -> /* Turn automatic conversion off */
6 -> unsetopt("ansi") ;
7 -> inc(3) ; /* 3 is not converted. */
/* Parameter type mismatch warning. */
```

To ensure backwards compatibility with **cc**, the ansi option is unset by default.

Signed Types:

The keyword signed has been added to the language. Signed is the opposite of unsigned, and it may be used just like unsigned to modify any of the integral types char, short, int, and long. If the type is short, int, or long, and neither signed or unsigned is specified, then the type defaults to being signed. If, however, the type is char, and neither signed or unsigned is specified, then whether the type is signed or unsigned is implementation defined. The main consequence of all of this is that since some machines automatically sign extended chars and others do not, the only way to guarantee one behavior or the other is to explicitly use signed char where sign extension must be performed and unsigned char where sign extension must not be performed. Otherwise, the implementation will do whatever is most efficient for the host machine.

A third floating point type, long double, has been created. Just as a double is something at least as long as a float, a long double is something at least as long as a double. The use of long float as a synonym for double has been discarded. In summary, there are now 11 predefined types in C:

```
signed char     unsigned char     float
signed short    unsigned short    double
signed int      unsigned int      long double
signed long     unsigned long
```

Constants:

For each of the int and long types and for all of the floating point types there is now a corresponding constant format. Also, for character constants and string literals, hexadecimal values may be in escape sequences. Here are some examples of the different types of arithmetic constants and escape sequences now allowed:

```
/* Constants with type signed int. */
255      0xff      0XFF      0377

/* Constants with type unsigned int. */
255U     0xFFu     0377U     0x00ffu

/* Constants with type signed long. */
255L     255l      0xffl     000377L

/* Constants with type unsigned long. */
255UL    255lu     0xffLu    0377Ul

/* Constants with type double */
255.0    0.0255E4  .255e+3   25500E-2

/* Constants with type float. */
255.0f   0.0255E4F .255e+3f  25500E-2F

/* Constants with type long double. */
255.0L   0.0255E4l .255E+3L  25500e-2l

/* Octal and hexadecimal escape sequences in */
/* string literals and character constants. */
"\377"   '\377'    "\xff"    '\xFF'
```

Const and Volatile:

Two new type attributes, const and volatile, have been added to the language. Volatile specifies that the value of an expression may change in unpredictable ways. The most important use of volatile is to tell the compiler not to perform certain types of optimizations. This is especially useful for data that may be modified asynchronously, such as occurs with signals. Consider the following functions which perform a loop until an interrupt occurs.

```
int interrupt_flag = FALSE;

void do_signal()
{
    interrupt_flag = TRUE;
}

do_loop()
{
    /* Loop exits when an interrupt occurs. */
    signal(SIGINT, do_signal);
    for( ; ; )
    {
        /* Some actions ... */
        if(interrupt_flag) return;
        /* Some more actions ... */
    }
}
```

Since the value of interrupt_flag does not appear to change inside the for-loop, an optimizing compiler may rearrange the second function as if it read:

```
do_loop()
{
    signal(SIGINT, do_signal);
    if(interrupt_flag) return;
    for( ; ; )
    {
        /* Some actions ... */
    }
}
```

To prevent this from happening the declaration of interrupt_flag should be changed to:

```
volatile int interrupt_flag = FALSE;
```

An expression with a const type cannot be an lvalue, that is, it cannot be modified. For example, a string literal, according to the Draft Standard, is an array of const char. The const attribute may be used to guarantee that data is not inadvertently changed. The prototype for strlen might more correctly be given as:

```
int strlen(const char *);
```

This guarantees that strlen examines, but does not change, the string addressed by its parameter. Static objects which are const but not volatile, such as string literals, may be placed in read-only memory.

Unary plus:

A unary plus operator has been added to C. Even though the unary plus operator simply returns the value of its sole operand, expressions involving unary plus must be evaluated as if the operation had a real effect. The main purpose of the unary plus operator to prevent an optimizing compiler from rearranging sub-expressions, since, especially for floating pointer operations, this may affect the precession of the result.

Preprocessor:

Several additions have been made to the C preprocessor. There is now a #elif directive, which is a combination of the #else and #if directives. Consider the following sequence of preprocessor directives:

```
#if FOO == FUM
   /* Here if FOO == FUM */
#else
#if FOO == FEE
   /* Here if FOO != FUM and FOO == FEE*/
#else
   /* Here if FOO != FUM and FOO != FEE*/
#endif
#endif
```

Using #elif this example can be written more concisely as:

```
#if FOO == FUM
   /* Here if FOO == FUM */
#elif FOO == FEE
   /* Here if FOO != FUM and FOO == FEE*/
#else
   /* Here if FOO != FUM and FOO != FEE*/
#endif
```

In #if directives, expressions of the form

```
defined MACRO_NAME
```

or

```
defined ( MACRO_NAME )
```

may appear. The expressions evaluates to 1 if macro *MACRO_NAME* is currently defined, or 0 if it is not. The #defined *operator effectively makes the #ifdef* and #ifndef *directives obsolete, because*

```
#ifdef MACRO_NAME
```

and

```
#ifndef MACRO_NAME
```

may now be written as:

```
#if defined(MACRO_NAME)
```

and

```
#if !defined(MACRO_NAME)
```

The parentheses are optional.

Some changes have been made in the way macro expansion is performed. Direct macro recursion is now permitted. In the body of a macro, the name of that macro is not expanded. For example

```
#define func(arg)  func((arg) + 1)
func(count)
```

expands to

```
func((count) + 1)
```

On the other hand the mutual recursion or macros is still not allowed. The following definitions will cause an error if either macro is ever used:

```
#define foo(x)  fum(x)
#define fum(y)  foo(y)
```

Two new preprocessing operators, # and ## have been created. The occurrence of # followed by the name of a macro formal parameter is replaced by the actual parameters surrounded by double-quotes. This means that

```
#define ld(file) load(#file)
ld(foo.c)
```

expands to

```
load("foo.c")
```

The occurrence of ## in the body of a macro surrounded by any two tokens expands to the concatenation of those two tokens. For example

```
#define debug(num)  printf("%d\n",x ## num)
debug(1)
```

expands to

```
printf("%d\n",x1)
```

White space surrounding the ## is ignored.

Table of Contents