ANSI C 2

Introduction

As part of the move towards standardization with System V Release 4, Solaris 2.x supports a C compiler that complies with the ANSI C standard X3.159-1989 (ANSI C), rather than the common Kernighan and Ritchie (K&R) C compiler previously supplied.

The ANSI C standard or *ANSI C* refers to the standard C as defined by the American National Standards Institute (ANSI), an organization that reviews and approves product standards in the United States. "Sun ANSI C" is Sun's ANSI C implementation with extensions that comply with the ANSI C standard. "Common C" refers to Sun's implementation of common C (non ANSI C).

To ease the migration to Solaris 2.x, the ANSI C compiler has been backported to run under SunOS 4.1.x as part of the unbundled Sun C 2.0.1 compiler package. This package consists of two compilers: `cc`, a C compiler, and `acc`, an ANSI C compiler with compatibility flags that allow it to accept certain common C constructs. Additionally, SPARCworks Professional C 2.0, the ANSI C compiler and tools for Solaris 2.x, also supports compatibility flags that allow it to accept both common C constructs and ANSI C.

This chapter outlines the migration path from common C in SunOS 4.1.x to Sun ANSI C in the Sun C 2.0.1 package and in Solaris 2.x. Because SunOS 4.1.x is not a fully SVID3-compliant platform, software that is compiled with the Sun C 2.0.1 ANSI C compiler must also be compliant with SVID3 to run successfully under Solaris 2.x. See Chapter 8, "SVID3 Compliance," for details.

For additional reference material, refer to the bibliography at the end of this chapter.

This chapter discusses the following topics:

- compatibility flags
- preprocessor differences
- function prototypes
- unsigned versus value preserving
- error messages
- other considerations, including new keywords, header files, grouping and expression evaluations, incomplete types, and internationalization support.

This chapter also contains the following transition topics:

- basic modes
- mixing old and new style functions
- functions with varying arguments
- promotions: unsigned vs. value
- tokenization and preprocessing
- using `const` and `volatile`
- multibyte characters and wide characters
- standard headers and reserved names
- internationalization
- grouping and evaluation in expressions
- incomplete types
- compatible and composite types.

Appendix A, "Sun C and Sun ANSI C Differences," contains much of this information in tabular form. Appendix B, "Comparison of `cc` Options," includes a comparison between `cc` options supported in K&R C (SunOS 4.1), ANSI C (SunOS 4.1) and ANSI C (SunOS 5.x).

Compatibility Flags

Basic Modes

The ANSI C compiler allows both old-style and new-style C code. The following `-X` (note case) options provide varying degrees of compliance to the ANSI C standard:

- `-Xa`
  (`a` = *ANSI*) ANSI C plus K&R C compatibility extensions, with semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the ANSI C interpretation.

- `-Xc`
  (`c` = *conformance*) Maximally conformant ANSI C, without K&R C compatibility extensions. The compiler will reject programs that use non-ANSI C constructs.

- `-Xs`
  (`s` = *senescent*) The compiled language includes all features compatible with pre-ANSI K&R C. The compiler warns about all language constructs whose differs behavior between ANSI C and the old K&R C.

- `-Xt`
  (`t` = *transition*) ANSI C plus K&R C compatibility extensions, without semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the K&R C interpretation.

Differences Between Sun C and ANSI C

Introduction

This section discusses the differences in compiler behavior when you use the `-Xs` option. The `-Xs` option tries to emulate `/bin/cc`, Sun C 1.0, Sun C 2.0.1 (K&R style), but in some cases the emulation fails.

Table 2-1 `-Xs` Behavior

Data Type	Sun C (K&R)	Sun ANSI C

aggregate initialization
 	
`struct {`
`    int a[3];`
`    int b;`
`} w[] = { {1} , 2};`
	`sizeof (w) = 16`
	`w[0].a = 1, 0, 0`
	`w[0].b =2`
		`sizeof (w) = 32`
		`w[0].a = 1, 0, 0`
		`w[0].b =2`

incomplete `struct, union, enum` declaration	`struct fq {`
	`    int i;`
	`    struct unknown;`
	`};`
		Does not allow incomplete `struct, union, enum` declaration.

switch expression integral type	Allows non-integral type.	Does not allow non-integral type.
order of precedence	Allows `if (rcount > count += index)`	Does not allow `if (rcount > count += index)`

unsigned, short, and long `typedef` declarations	Allows
	
	`typedef short small`
	`unsigned small;`
		Does not allow (all modes).

Table 2-1 −Xs Behavior (Continued)

Data Type	Sun C (K&R)	Sun ANSI C

`struct` or `union` tag mismatch in nested `struct` or `union` declarations	Allows tag mismatch
	
	`struct x {`
	`int i;`
	`} s1;`
	
	`/* K&R treats as a struct */`
	`{`
	`union x s2;`
	`}`
		Does not allow tag mismatch in nested `struct` or `union` declaration.

incomplete `struct` or `union` type	Ignores an incomplete type declaration.	`struct x {`
		`int i;`
		`} s1;`
		
		`main()`
		`{`
		`struct x;`
		`struct y {`
		`struct x f1;`
		`/* in K&R, f1 refers`
		`to outer struct */`
		`} s2;`
		`struct x {`
		`int i;`
		`};`
		`}`

casts as `lvalues`	Allows `(char *) ip = &foo;`	Does not allow casts as `lvalues` (all modes).

Preprocessor Differences

Probably the least specified part of previous versions of C concerned the operations that transformed each source file from a bunch of characters into a sequence of (ready to parse) tokens. These operations included recognition of white space (including comments), bundling consecutive characters into tokens, handling preprocessing directive lines, and macro replacement. However, their respective ordering was never guaranteed.

Tokenization

ANSI C Translation Phases

The order of these translation phases is specified by ANSI C:

1. Every *trigraph sequence* in the source file is replaced. ANSI C has exactly nine trigraph sequences that were invented solely as a concession to deficient character sets (as far as C is concerned). These are three-character sequences that name a character not in the ISO 646-1983 character set. Table 2-2 contains a list of trigraph sequences.

Table 2-2 Trigraph Sequences

Trigraph Sequence	Converts to	Trigraph Sequence	Converts to
`??=`	`#`	`??<`	`{`
`??-`	`~`	`??>`	`}`
`??(`	`[`	`??/`	`\`
`??)`	`]`	`??'`	`^`
`??!`	|		

These sequences must be understood by ANSI C compilers, but we would not recommend their use except (possibly) to obscure code. The ANSI C compiler warns you whenever it replaces a trigraph while in transition (`-Xt`) mode, even in comments. For example, consider the following:

```
/* comment *??/
/* still comment? */
```

The `??/` becomes a backslash. This character and the following newline are removed. The resulting characters are:

```
/* comment */* still comment? */
```

The first `/` from the second line is the end of the comment. The next token is the `*`.

2. Every backslash/newline character pair is deleted.

3. The source file is converted into preprocessing tokens and sequences of white space. Each comment is effectively replaced by a space character. (Preprocessing tokens are discussed in more detail later in this chapter.)

4. Every preprocessing directive is handled and all macro invocations are replaced. Each `#include`d source file is run through the earlier phases before its contents replace the directive line. (Preprocessing directives are discussed in more detail later in this chapter.)

5. Every escape sequence (in character constants and string literals) is interpreted.
6. Adjacent string literals are concatenated.
7. Every preprocessing token is converted into a (regular) token; the compiler properly parses these and generates code.
8. All external object and function references are resolved, resulting in the final program.

Old C Translation Phases

Previous C compilers did not follow such a simple sequence of phases, nor were there any guarantees for when the above steps were applied. A separate preprocessor recognized tokens and white space at essentially the same time as it replaced macros and handled directive lines. The output was then completely retokenized by the compiler proper, which then parsed the language and generated code.

Because the tokenization process within the preprocessor was a moment-by-moment thing and macro replacement was done as a character-based operation (and not token-based), the tokens and white space could have a great deal of variation during preprocessing.

There are a number of differences that arise from these two approaches. The rest of this section will discuss how code behavior may change due to line splicing, macro replacement, *stringizing*, and token *pasting*, which occur during macro replacement.

Logical Source Lines

In K&R C, backslash/newline pairs were allowed only as a means to continue a directive, a string literal, or a character constant to the next line. ANSI C extended the notion so that a backslash/newline pair can continue anything to the next line. (The result is a *logical source line*.) Therefore, any code that relied on the separate recognition of tokens on either side of a backslash/newline pair will not behave as expected.

Using Strings

Note – In ANSI C, the examples marked with a ‡ produce a warning about use of old features. Only in the transition mode (`–Xt` and `–Xs`) will the result be the same as in previous versions of C.

In K&R C, the following produces the string literal `"x y!"`:

```
#define str(a) "a!"       ‡
str(x y)
```

Thus, the preprocessor searched inside string literals (and character constants) for characters that looked like macro parameters. ANSI C recognized the importance of this feature, but could not condone operations on parts of tokens. (In ANSI C, all invocations of the above macro produce the string literal "a!".) To achieve the old effect in ANSI C, we make use of the # macro substitution operator and the concatenation of string literals.

```
#define str(a) #a "!"
str(x y)
```

This example produces the two string literals "x y" and "!" which, after concatenation, produces the identical "x y!".

Unfortunately, there is no direct replacement for the analogous operation for character constants. The major use of this feature is similar to the following:

```
#define CNTL(ch) (037 & 'ch')      ‡
CNTL(L)
```

This produces:

```
(037 & 'L')
```

which evaluates to the ASCII control-L character. The best solution we know of is to change all uses of this macro (at least this can be done automatically) to:

```
#define CNTL(ch) (037 & (ch))
CNTL('L')
```

which is arguably more readable and more useful, as it can also be applied to expressions.

Token Pasting

In K&R C, there were at least two ways to combine two tokens. Both invocations in the following produced a single identifier x1 out of the two tokens x and 1.

```
#define self(a) a
#define glue(a,b) a/**/b ‡
self(x)1
glue(x,1)
```

Note – # and ## should be used as macro substitution operators only when __STDC__ is defined.

Again, ANSI C could not sanction either approach to what K&R believed to be an important capability. In ANSI C, both the preceding invocations would produce the two separate tokens `x` and `1`. The second of these two methods can be rewritten for ANSI C by using the `##` macro substitution operator, as shown in the following example:

```
#define glue(a,b) a ## b
glue(x, 1)
```

Because `##` is an actual operator, the invocation can be much freer with respect to white space in both the definition and invocation.

There is no direct approach to effect the first of the two old-style pasting schemes, but since it put the burden of the pasting at the invocation, it was used less frequently than the other form.

Preprocessing

Preprocessing Tokens

A *token* is the basic lexical unit of the language. All source input must be formed into valid tokens by translation phase seven. Preprocessing tokens (pp-tokens) are a superset of regular tokens. Preprocessing tokens allow the source file to contain non-token character sequences that constitute valid preprocessing tokens during translation. There are four categories of preprocessing tokens:

- Header filenames, meant to be taken as a single token.
- Preprocessing numbers (discussed in “Preprocessing Numbers” below).
- All other single characters that are not otherwise (regular) tokens; see the example under “Preprocessing Numbers” below.
- Identifiers, numeric constants, character constants, string literals, operators, and punctuators.

Preprocessing Numbers

- A preprocessing number is made up of a digit, optionally preceded by a period, and may be followed by letters, underscores, digits, periods, and any one of `e+ e- E+ E-`.
- Preprocessing numbers include all valid number tokens, plus some that are not valid number tokens. For example, in the macro definition:

  ```
  #define R 2e ## 3
  ```

the preprocessing number `2e` is not a valid number. However, the preprocessing operator `##` will *paste* it together with the preprocessing number `3` when `R` is replaced, resulting in the preprocessing number `2e3`, which is a valid number. See “Preprocessing Operators” later in this chapter for a discussion of the `##` operator.

Preprocessor Rescanning

The common C preprocessor will recursively substitute. However, in ANSI C a macro will not be replaced if it is found in the replacement list during the rescan.

Preprocessing Directives

The preprocessing operators are evaluated left to right, without any defined precedence.

#

A macro parameter preceded by the `#` preprocessing operator has its corresponding unexpanded argument tokens converted into a string literal. (Any double quotes and backslashes contained in character constants or part of string literals are escaped by a backslash). The `#` character is sometimes referred to as the *stringizing* operator. This rule applies only within function-like macros.

##

If a replacement token sequence (see “Macro Definition and Expansion” below) contains a `##` operator, the `##` and any surrounding white space are deleted and adjacent tokens are concatenated, creating a new token. This occurs only when the macro is expanded.

Macro Definition, Expansion, and Replacement

Macro Replacement

The macro replacement process has never been described in any significant detail prior to ANSI C. This vagueness spawned a great many divergent implementations and any code that relied on anything fancier than manifest constant replacement and simple `?:` – like macros was probably not truly portable. This chapter cannot begin to uncover all the subtle and not so subtle differences between the old C macro replacement implementation and the ANSI C version. Fortunately, nearly all uses of macro replacement, with the exception of token pasting and stringizing, will produce exactly the same series of tokens as before. Furthermore, the ANSI C macro replacement algorithm can do things not possible in the old C version. For example,

```
#define name (*name)
```

causes any use of `name` to be replaced with an indirect reference through `name`. (The old C preprocessor would produce a huge number of parentheses and stars and eventually complain about macro recursion.)

The major change in the macro replacement approach taken by ANSI C is to require macro arguments (other than those that are operands of the macro substitution operators `#` and `##`) to be expanded recursively prior to their substitution in the replacement token list. However, this change seldom produces an actual difference in the resulting tokens.

Macro Definition and Expansion

- An object-like macro is defined with a line of the form:

```
#define identifier token-sequence
```

  where `identifier` is replaced with `token-sequence` wherever `identifier` appears in regular text.

- A function-like macro is defined with a line of the form:

```
#define identifier ( identifier-list )  token-sequence
```

  where the macro parameters are contained in the comma-separated `identifier-list`. The `token-sequence` following the identifier list determines the behavior of the macro, and is referred to as the *replacement list*. There can be no space between the identifier and the `(` character. For example:

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

  The replacement-list `a+b` determines that the two parameters `a` and `b` are added.

- A function-like macro is invoked in normal text by using its identifier, followed by a `(` token, a list of token sequences separated by commas, and a `)` token, as in:

```
FLM(3,2)
```

  The arguments in the invocation (comma-separated token sequences) may be expanded, and they then replace the corresponding parameters in the replacement token sequence of the macro definition. Macro arguments in the invocation are *not* expanded if they are operands of `#` or `##` operators in the replacement string. Otherwise, expansion does take place.

  For example, if `M1` is defined as `3`:

```
#define M1 3
```

when the function-like macro `FLM` is used, use of the `#` or `##` operators affects expansion (and the result), as follows:

Definition	Invocation	Result	Expansion?
`a+b`	`FLM(M1,2)`	`3 + 2`	Yes, Yes
`#a`	`FLM(M1)`	`"M1"`	No
`a##b`	`FLM(M1,2)`	`M12`	No, No
`a+#a`	`FLM(M1,2)`	`3 + ”2”`	Yes, No

In the last example line, the first `a` in `a+#a` is expanded, but the second `a` is not expanded because it is an operand of the `#` operator.

- The number of arguments in the invocation must match the number of parameters in the definition.
- A macro’s definition, if any, can be eliminated with a line of the form:

```
#undef identifier
```

There is no effect if the definition does not exist.

File Inclusion

A line of the form:

```
#include "filename"
```

causes the entire line to be replaced with the contents of `filename`. The following directories are searched, in order:

1. The current directory (of the file containing the `#include` line).
2. Any directories named in `-I` options to the compiler, in order.
3. A list of standard places, typically, but not necessarily, `/usr/include`.

A line of the form:

```
#include <filename>
```

causes the entire line to be replaced with contents of `filename`. The angle brackets surrounding `filename` indicate that `filename` is not searched for in the current directory.

A third form allows an arbitrary number of preprocessing tokens to follow the `#include`, as in:

```
#include preprocessing-tokens
```

The preprocessing tokens are processed the same way as when they are used in normal text. Any defined macro name is replaced with its replacement list of preprocessing tokens. The preprocessing tokens must expand to match one of the first two forms (`"..."` or `< ... >`).

- A filename beginning with a slash `/` indicates the absolute pathname of a file to include, no matter which form of `#include` is used.
- Any `#include` statements found in an included file cause recursive processing of the named files.

Conditional Compilation

Different segments of a program may be compiled conditionally. Conditional compilation statements must observe the following sequence:

1. One of either `#if`, `#ifdef`, or `#ifndef`.
2. Any number of optional `#elif` lines.
3. One optional `#else` line.
4. One `#endif` line.

For example:

```
#if integral-constant-expression
```

is true if `integral-constant-expression` evaluates to nonzero. If true, tokens following the `if` line are included. The `integral-constant-expression` following the `if` is evaluated by following this sequence:

1. Any preprocessing tokens in the expression are expanded. Any use of the `defined` operator evaluates to `1` or `0` depending on whether its operand is defined.
2. If any identifiers remain, they evaluate to `0`.
3. The remaining integral constant expression is evaluated. The constant expression must be made up of components that evaluate to an integral constant. In the context of a `#if`, the integral constant expression may not contain the `sizeof` operator, casts, or floating point constants.

The following table shows how various types of constant expressions following a `#if` would be evaluated. Assume that *name* is not defined.

Table 2-3 Constant Expression Evaluation

Constant Expression	Step 1	Step 2	Step 3
`__STDC__`	1	1	1
`!defined(__STDC__)`	!1	!1	0
3||*name*	3||*name*	3||0	1
2 + *name*	2 + *name*	2 + 0	2

- `#ifdef`
- *identifier*
  Is true if *identifier* is currently defined by `#define` or by the `-D` option to the compiler command line.
- `#ifndef` *identifier*
  Is true if *identifier* is not currently defined by `#define` (or has been undefined).
- `#elif` *constant-expression*
  Indicates alternate if-condition when all preceding if-conditions are false.
- `#else`
  Indicates alternate action when no preceding `if` or `elif` conditions are true. A comment may follow the `else`, but a token may not.
- `#endif`
  Terminates the current conditional. A comment may follow the `endif`, but a token may not.

Line Control

Line control is useful for programs that generate C programs. A line of the form:

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

causes the compiler to believe, for the purposes of error diagnostics and debugging, that the line number of the next source line is equal to `constant` (which must be a decimal integer) and the current input file is `filename` (enclosed in double quotes). The quoted filename is optional. `constant` which must be a decimal integer in the range `1` to `MAXINT`. `MAXINT`, is defined in `limits.h`.

Assertions

A line of the form:

```
#assert predicate (token-sequence)
```

associates the `token-sequence` with the predicate in the assertion name space (separate from the space used for macro definitions). The predicate must be an identifier token.

```
#assert predicate
```

asserts that `predicate` exists, but does not associate any token sequence with it.

The compiler provides the following predefined predicates by default:

```
#assert machine ( SPARC )
#assert system ( unix )
#assert cpu ( SPARC )
```

Any assertion may be removed by using `#unassert`, which uses the same syntax as `assert`. Using `#unassert` with no argument deletes all assertions on the predicate; specifying an assertion deletes only that assertion.

An assertion may be tested in a `#if` statement with the following syntax:

```
#if #predicate(non-empty token-list)
```

For example, the predefined predicate `system` can be tested with the following line:

```
#if #system(unix)
```

This evaluates true.

Version Control

The `#ident` directive is used to help administer version control information. Typing:

```
#ident "version"
```

puts an arbitrary string in the `.comment` section of the object file. The `.comment` section is not loaded into memory when the program is executed.

Pragmas

Preprocessing lines of the form:

```
#pragma pp-tokens
```

specify implementation-defined actions.

The following `#pragmas` are recognized by the compilation system:

- `#pragma fini identifier`
  (SunOS 5.x only). Marking `identifier` as a "finalization function." Such functions are expected to be of type `void` and to accept no arguments, and are called either when a program terminates under program control or when the containing shared object is removed from memory. As with "initialization functions," finalization functions are executed in the order processed by the link editor(s).

- `#pragma init identifier`
  (SunOS 5.x only). Marking `identifier` as an "initialization function." Such functions are expected to be of type `void` and to accept no arguments, and are called while constructing the memory image of the program at the start of execution. In the case of initializers in a shared object, they will be executed during the operation that brings the shared object into memory, either program start-up or some dynamic loading operation such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they were processed by the link editor(s), both static and dynamic.

- `#pragma ident string`
  Place `string` in the `.comment` section of the executable.

- `#pragma int_to_unsigned function name`
  For a function that returns a type of `unsigned`, in `-Xt` or `-Xs` mode, change the function return to be of type `int`.

- `#pragma unknown_control_flow (name, [,name] )`
  Specifies a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`. Since such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

- `#pragma weak function name = _function name`
  (SunOS 5.x only). If a defined global symbol `function name` exists, the appearance of a weak symbol `_function name` with the same name will not cause an error.

- `#pragma weak function name`
  (SunOS 5.x only). The linker will not complain if it does not find a definition for `function name`.

The compiler ignores unrecognized pragmas.

Error Generation

A preprocessing line consisting of:

```
#error token-sequence
```

causes the compiler to produce a diagnostic message containing the `token-sequence` and then stop.

Predefined Names

The identifiers listed in Table 2-4 are predefined as object-like macros.

Table 2-4 Predefined Identifiers

Identifier	Description
`__LINE__`	The current line number as a decimal constant.
`__FILE__`	A string literal representing the name of the file being compiled.
`__DATE__`	The date of compilation as a string literal in the form "*Mmm dd yyyy.*"
`__TIME__`	The time of compilation as a string literal in the form "*hh:mm:ss.*"
`__STDC__`	The constant `1` under compilation mode `-Xc`, otherwise `0`.

With the exception of `__STDC__`, these predefined names may not be undefined or redefined. Under compilation mode `-Xt`, `__STDC__` may be undefined (`#undef __STDC__`) to cause a source file to think it is being compiled by a previous version of the compiler.

The identifier `__STDC__` is not defined in `-Xs` mode.

ANSI C Examples

Note – acc refers to the ANSI C compiler under SunOS 4.x.

Token Pasting and the ## Preprocessor Operator

ANSI C defines ## as the preprocessor operator that performs token pasting. Common C does not recognize the ## operator.

Code Example 2-1 Token Pasting and the ##Preprocessor Operator

```
#include <stdio.h>
#define INC “++”
#define TAB “internal_table”
#define INCTAB “table_of_increments”
#define CONC(x,y) x/**/y
main()
{
    printf(“%s\n”, CONC(INC,TAB));
}
```

The following example results from compiling common C and ANSI C with the -Xs switch:

Code Example 2-2 Common C and ANSI C -Xs

```
% cc pp_token.c
% a.out
table_of_increments
```

The following example results from compiling ANSI C with the -Xt switch:

Code Example 2-3 ANSI C with the -Xt switch

```
% acc -Xt pp_token.c
“pp_token.c”, line 7: warning: comment is replaced by “##”
% a.out
table_of_increments
```

The following example results from compiling ANSI C with the -Xa and -Xc switches:

Code Example 2-4 ANSI C with the -Xa and -Xc switches

```
% acc -Xc pp_token.c
“pp_token.c”, line 7: warning: comment does not concatenate tokens
% a.out
++internal_table
```

Comments Spanning Include Files

Common C allow comments which start in an include file to be terminated by the file that includes the first file. ANSI C does not support this behavior.

Code Example 2-5 comment.c

```
#define number 111
/* This is the first part of comment. Comment will be terminated in main()
#include <stdio.h>
#include "comment.h"

/* This is the second part of comment */
main ()
{
    printf(" number=%d\n", number);
}
```

A compile and run of `comment.c` under Common C and ANSI C using the `-Xs` switch produces the following result:

```
% cc comment.c
% a.out
number=111
```

A compile and run of `comment.c` under ANSI C using the `-Xt`, `-Xa`, and `-Xc` switches produces the following result:

```
% acc -Xc comment.c
"./comment.h", line 3: warning: EOF in comment
"comment.c", line 5: syntax error, probably missing ",", ";" or "="
"comment.c", line 5: syntax error before or at: is
"comment.c", line 5: syntax error, probably missing ",", ";" or "="
"comment.c", line 5: syntax error, probably missing ",", ";" or "="
"comment.c", line 5: syntax error, probably missing ",", ";" or "="
"comment.c", line 5: syntax error, probably missing ",", ";" or "="
"comment.c", line 5: syntax error, probably missing ",", ";" or "="
Compilation failed
```

Formal Parameter Substitution within a Character Constant

The common C preprocessor substitutes character constant when it matches the replacement list macros. In ANSI C, the character is not replaced.

Code Example 2-6 param.c

```
#include <stdio.h>
#define string(c) "c"
main ()
{
    printf("char = %s\n", string(testing));
}
```

A compile and run of `param.c` under Common C and ANSI C using the `-Xs` switch produces the following result:

```
% cc param.c
% a.out
char = testing
```

A compile and run of `param.c` under ANSI C using the `-Xt` switch produces the following result:

```
% acc -Xt param.c
"param.c", line 3: warning: macro replacement within a string literal
% a.out
char = testing
```

A compile and run of `param.c` under ANSI C using the `-Xa` and `-Xc` switches produces the following result:

```
% acc -Xt param.c
% "param.c", line 3: warning: no macro replacement within a string literal
% a.out
% char = c
```

Formal Parameter Substitution within a String Constant

The common C preprocessor will substitute a formal parameter when enclosed with a string constant. This behavior is not supported in ANSI C. For string substitution in ANSI C, the `#` preprocessor operator should be used.

Code Example 2-7 str_sub.c

```
#include <stdio.h>
#define string(cc) "cc"
main()
{
    printf("string = %s\n", string(test));
}
```

A compile and run of `str_sub.c` under Common C and ANSI C using the `-Xs` switch produces the following result:

```
% cc str_sub.c
% a.out
string = test
```

A compile and run of `str_sub.c` under ANSI C using the `-Xt` switch produces the following result:

```
% acc -Xt str_sub.c
"str_sub.c", line 3: warning: macro replacement within a string literal
% a.out
string = test
```

A compile and run of `str_sub.c` under ANSI C using the `-Xa` and `-Xc` switches produces the following result:

```
% acc -Xc str_sub.c
"str_sub.c", line 3: warning: no macro replacement within a string literal
% a.out
string = cc
###
```

Function Prototypes

Supplying Function Prototypes

ANSI C's most sweeping change to the language is the function prototype borrowed from the C++ language. A *function prototype* is a function declaration or definition that follows the ANSI C prototype syntax or a function written in that syntax. Like a traditional function declaration, a function prototype declares the data type of the value returned. Unlike a traditional declaration, a function prototype also declares the number and type of the function's formal parameters. By specifying the number and types of its parameters, for each function not only does every regular compile get the benefits of argument and parameter checks (similar to those of `lint`) for each function call, but arguments are automatically converted (just as with an assignment) to the type expected by the function. Function prototypes are typically specified in header files or at the beginning of a source file.

Prototype Classes

There are three classes of prototypes. The class of a prototype is determined by whether the function has zero, a fixed number, or a variable number of parameters.

- A function that has no parameters must have a parameter type list consisting of the single type specifier `void`, such as:

```
extern int random_generation(void);
static void do_nothing(void) {return;}
```

- A function that has a fixed number of parameters indicates the type of those parameters in the parameter type list. If the prototype appears in a function declaration, parameter names may be included or not, as desired, which helps in documenting the function. Parameter names must appear in the definition, such as:

```
double square (double x) { return x*x; }
extern char *strncpy(char *, char *, int);
```

- A function that has a variable number of parameters or parameters of varying types indicates the types of any fixed parameters as above and follows them by the notion "`, ...`" (which is composed of four tokens: a comma followed by three periods):

```
extern in fprintf (FILE *file, char *format,...)
```

  There must at least one fixed parameter, or the parameter list cannot be accessed using the standard library facilities from `stdarg.h`

In general, there is only one prototype that matches a non-prototype function definition. This prototype is sometimes referred to as the function's *Miranda* prototype, since it is "appointed" to a function definition that otherwise would not have one. Table 2-5 lists some compatible and incompatible prototype declarations for the following code:

```
int f(x, y)
    float x;
    int y;
{ ...}
```

Table 2-5 Prototype Compatibility with Function Definition

Prototype	Compatible with `int f()`	Reason
`extern double f(double x, int y);`	no	the parameter list is OK; the return types are not compatible
`extern int f(float, int);`	no	the first parameter must have prototype `double`
`extern int f(double a, int b);`	yes	this is the only compatible prototype; the parameter names do not matter

Functions with Varying Arguments

In previous implementations, you could not specify the parameter types that a function expected, but ANSI C encourages you to use prototypes to do just that. In order to support functions such as `printf()`, the syntax for prototypes includes a special ellipsis `(...)` terminator. Because an implementation might be required to do unusual things to handle a varying number of arguments, ANSI C requires that all declarations and the definition of such a function include the ellipsis terminator.

Since there are no names for the "`...`" part of the parameters, a special set of macros, contained in `stdarg.h` gives the function access to these arguments. Earlier versions of such functions had to use similar macros contained in `varargs.h`.

For example, let us assume that the function we wish to write is an error handler called `errmsg()` that returns `void` and whose only fixed parameter is an `int` that specifies details about the error message. This parameter may be followed by a filename, or a line number, or both, and these are followed by `printf()` — like format and arguments that specify the text of the error message.

In order to allow our example to compile with earlier compilers, we will make extensive use of the macro `__STDC__` which is defined only for ANSI C compilation systems. Thus the function's declaration (in the appropriate header file) would be:

```
#ifdef __STDC__
    void errmsg(int code, ...);
#else
    void errmsg();
#endif
```

The file that contains the definition of `errmsg()` is where the old and new styles can get complex. The header you include depends on the compilation system:

```
#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif
#include <stdio.h>
```

In this example, the header file `stdio.h` is included because the program calls `fprintf()` and `vfprintf()` later.

Next comes the function's definition. The identifiers `va_alist` and `va_dcl` are part of the old-style `varargs.h` interface.

```
void
#ifdef __STDC__
errmsg(int code, ...)
#else
errmsg(va_alist) va_dcl /* note: no semicolon!  */
#endif
{
    /* more detail below */
}
```

Since the old-style variable argument mechanism did not allow we to specify any fixed parameters (at least not officially), we had to arrange for them to be accessed before the varying portion. Also, due to the lack of a name for the "`...`" part of the parameters, the new `va_start()` macro has a second argument—the name of the parameter that comes just before the "`...`" terminator.

As an extension, Sun ANSI C, allows functions to be declared and defined with no fixed parameters, as in:

```
int f(...);
```

For such functions, `va_start()` should be invoked with an empty second argument, as in:

```
va_start(ap,)
```

The following is the body of the function:

```
{
    va_list ap;
    char *fmt;
#ifdef __STDC__
    va_start(ap, code);
#else
    int code;

    va_start(ap);
    /* extract the fixed argument */
    code = va_arg(ap, int);
#endif
    if (code & FILENAME)
        (void)fprintf(stderr, "\"%s\": ", va_arg(ap, char *));
    if (code & LINENUMBER)
        (void)fprintf(stderr, "%d: ", va_arg(ap, int));
    if (code & WARNING)
        (void)fputs("warning: ", stderr);
    fmt = va_arg(ap, char *);
    (void)vfprintf(stderr, fmt, ap);
    va_end(ap);
}
```

Both the `va_arg()` and `va_end()` macros work the same for the old-style and ANSI C versions. Because `va_arg()` changes the value of `ap`, the call to `vfprintf()` cannot be:

```
(void)vfprintf(stderr, va_arg(ap, char *), ap);
```

The definitions for the macros `FILENAME`, `LINENUMBER`, and `WARNING` are presumably contained in the same header as the declaration of `errmsg()`.

A sample call to `errmsg()` is:

```
errmsg(FILENAME, "<command line>", "cannot open: %s\n", argv[optind]);
```

Mixing Old-Style and New-Style Functions

For function prototype declarations to work with old-style function definitions, both must specify functionally identical interfaces or, using ANSI C's terminology, have *compatible types*.

For functions with varying arguments, there can be no mixing of ANSI C's ellipsis notation and the old-style `varargs()` function definition. For functions with a fixed number of parameters, the situation is fairly straightforward: just specify the types of the parameters as they were passed in previous implementations.

In K&R C, each argument was converted just before it was passed to the called function according to the default argument promotions. These specified that all integral types narrower than `int` were promoted to `int` size, and any `float` argument was promoted to `double`. This simplified both the compiler and libraries. Function prototypes are more expressive; the specified parameter type is what is passed to the function. Thus, if a function prototype is written for an existing (old-style) function definition, there should be no parameters in the function prototype with any of the following types:

`char`	`signed char`	`unsigned char`	`float`
`short`	`signed short`	`unsigned short`	

There still remain two complications with writing prototypes: `typedef` names and the promotion rules for narrow unsigned types.

If parameters in old-style functions were declared using `typedef` names such as `off_t` and `ino_t`, it is important to know whether or not the `typedef` name designates a type that is affected by the default argument promotions.

For these two, `off_t` is a `long`, so it is appropriate to use in a function prototype, but `ino_t` used to be an `unsigned short`, so if it were used in a prototype, the compiler would issue a diagnostic (possibly fatal) because the old-style definition and the prototype specify different and incompatible interfaces.

What should be used instead of an `unsigned short` leads into the final complication. One of the biggest incompatibilities between K&R C and the ANSI C compiler is the promotion rule for the widening of `unsigned char` and `unsigned short` to an `int` value. (For more information, see "Promotions: Unsigned Versus Value Preserving" below.) Unfortunately, the parameter type that matches such an old-style parameter depends on the compilation mode used when you compile. For `-Xs` and `-Xt`, `unsigned int` should be used; `-Xa` and `-Xc` should use `int`. The best approach is to change the old-style definition to specify either `int` or `unsigned int` and use the matching type in the function prototype. You can always assign it to a local variable with the narrower type, if necessary, after you enter the function.

Writing New Code

When you write an entirely new program, use new-style function declarations (prototypes) in headers and new-style function declarations and definitions in other C source files. However, if there is a possibility that someone will port the code to a machine with a pre-ANSI C compiler, use the macro `__STDC__` (which is defined only for ANSI C compilation systems) in both header and source files.

An ANSI C conforming compiler must issue a diagnostic whenever two incompatible declarations for the same object or function are in the same scope. Thus, if all functions are declared and defined with prototypes and the appropriate headers are included by the correct source files, all calls should agree with the definition of the functions. This eliminates one of the most common C programming mistakes.

Updating Existing Code

If you have an existing application and want the benefits of function prototypes, you can update your code in any of the following ways, depending on how much of the code you care to change:

1. **Recompile without making any changes.**

   Even with no coding changes, the compiler will warn you about mismatches in parameter type and number when invoked with the `-v` option.

2. **Add function prototypes just to the headers.**

   All calls to global functions are covered.

3. **Add function prototypes to the headers and start each source file with function prototypes for its local (static) functions.**

   All calls to functions are covered, but this requires typing the interface for each local function twice in the source file.

4. **Change all function declarations and definitions to use function prototypes.**

   For most programmers, choices 2 and 3 are probably the best cost/benefit compromise. Unfortunately, these options are also the ones that require detailed knowledge of the rules for mixing of old and new styles.

Examples

Watch out for the use of id's in prototypes that may be affected by preprocessing. Consider the following example:

```
#define status 23
    void my_exit(int status); /* Normally, scope begins */
                              /* and ends with prototype */
```

Do not mix function prototypes with old-style function declarations that contain narrow types.

```
void foo(unsigned char, unsigned short);
void foo(i, j) unsigned char i; unsigned short j; {...}
```

typedef Names in Formal Parameter Lists

Common C lets you use `typedef` names as formal parameter names in a function declaration, effectively hiding the `typedef` declaration.

ANSI C disallows the use of an identifier declaration as a `typedef` name as a formal parameter.

Code Example 2-8 Code Using typedef

```
#include <stdio.h>
typedef int yy;
main ()
{
    yy num=10;
    test (num);
    test (100);
}
test(yy)
int yy;
{
    printf("yy = %d\n", yy);
}
```

In Common C, the code compiles and runs as follows:

```
% cc typedef_param.c
% a.out
yy = 10
yy = 100
```

In ANSI C, with `-Xs, -Xt, -Xa, -Xc` switches, a run of the same code looks like this:

```
% acc -Xc typedef_param.c
"typedef_param.c", line 14: parameter not in identifier list: yy
"typedef_param.c", line 15: formal parameter lacks name: param #1
```

Appropriate use of `__STDC__` produces a header file that can be used for both old and new compilers:

```
header.h:

    struct s { /* .  .  .  */ };

    #ifdef __STDC__
       void errmsg(int, ...);
       struct s *f(const char *);
       int g(void);
    #else
       void errmsg();
       struct s *f();
       int g();
    #endif
```

The following function uses prototypes and can still be compiled on an older system:

```
struct s *
#ifdef __STDC__
f(const char *p)
#else
f(p) char *p;
#endif
{
    /* .  .  .  */
}
```

Here is an updated source file using function prototypes to the headers and starting the source file with function prototypes for its local (static) functions. The local function still uses an old-style definition, but a prototype is included for newer compilers:

Code Example 2-9 source.c

```
#include <header.h>
typedef /* .   .   .  */ MyType;
#ifdef __STDC__
    static void del(MyType *);
    /* .   .   .  */
#endif
static void
del(p)
    MyType *p;
{
    /* .   .   .  */
}
```

Unsigned Versus Value Preserving

Promotions: Unsigned Versus Value Preserving

The following information appears in the Rationale that accompanies the draft C Standard:

QUIET CHANGE

> A program that depends on unsigned preserving arithmetic conversions will behave differently, probably without complaint. This is considered to be the most serious change made by the Committee to a widespread current practice.

This section explores how this change affects your code.

Background

According to K & R C, `unsigned` specified exactly one type; there were no `unsigned char`, `unsigned short`, or `unsigned long` variables, but most C compilers added these very soon thereafter. (Some compilers did not implement `unsigned long` but included the other two.) Naturally, implementations chose different rules for type promotions when these new types mixed with others in expressions.

In most C compilers, the simpler rule—*unsigned preserving*—is used: when an `unsigned` type needs to be widened, it is widened to an `unsigned` type; when an `unsigned` type mixes with a `signed` type, the result is an `unsigned` type.

The other rule, specified by ANSI C, is called *value preserving*, where the result type depends on the relative sizes of the operand types. When an `unsigned char` or `unsigned short` is *widened*, the result type is `int`, if an `int` is large enough to represent all the values of the smaller type; otherwise, the result type is `unsigned int`. The *value preserving* rule produces the *least surprise* arithmetic result for most expressions.

Compilation Behavior

In the transition or preANSI modes (`-Xt` and `-Xs`), the ANSI C compiler uses the unsigned preserving promotions; in the other two modes, conforming (`-Xc`) and ANSI (`-Xa`), the value preserving promotion rules are used. Whatever the current mode may be, the compiler warns you about each expression with a behavior that might depend on the promotion rules used.

This warning is not optional because this is a serious change in behavior. Fortunately, these situations do not often occur, and it is always possible to suppress the warning by making the intended behavior explicit, as is shown in the next set of examples.

First Example: Using a Cast

In the following code, assume that an `unsigned char` is smaller than an `int`.

```
int f(void)
{
    int i = -2;
    unsigned char uc = 1;

    return (i + uc) < 17;
}
```

The preceding code causes the compiler to issue the following message:

```
line 6: warning: semantics of "<" change in ANSI C; use explicit cast
```

The result of the addition has type `int` (value preserving) or `unsigned int` (unsigned preserving), but the bit pattern does not change between these two. On a two's-complement machine we have:

```
    i: 111...110 (-2)
+   uc: 000...001 ( 1)
===================
       111...111 (-1 or UINT_MAX)
```

This bit representation corresponds to -1 for `int` and `UINT_MAX` for `unsigned int` . Thus, if the result has type `int`, a signed comparison is used and the less-than test is true; if the result has type `unsigned int`, an unsigned comparison is used and the less-than test is false.

The addition of a cast serves to specify which of the two behaviors is desired:

value preserving:
```
(i + (int)uc) < 17
```
unsigned preserving:
```
(i + (unsigned int)uc) < 17
```

This expression can be viewed as ambiguous because differing compilers chose different meanings for the same code. Thus, the addition of a cast is as much to help the reader as it is to eliminate the warning message.

Bit-Fields

The same situation applies to the promotion of bit-field values. In ANSI C, if the number of bits in an `int` or `unsigned int` bit-field is less than the number of bits in an `int`, the promoted type is `int`; otherwise the promoted type is `unsigned int`. In most older C compilers, the promoted type is `unsigned int` for explicitly unsigned bit-fields, and `int` otherwise.

Similar use of casts can eliminate situations that are ambiguous.

Second Example: Result Is the Same

In the following code, assume that both `unsigned short` and `unsigned char` are narrower than `int`.

```
int f(void)
{
    unsigned short us;
    unsigned char uc;

    return uc < us;
}
```

In this example, both automatics are either promoted to `int` or to `unsigned int`, so the comparison is sometimes `unsigned` and sometimes `signed`. However, the C compiler will not warn you because the result is the same for both choices.

Integral Constants

As with expressions, the rules for the types of certain integral constants have changed. In K&R C, an unsuffixed decimal constant had type `int` only if its value fit in an `int`, and an unsuffixed octal or hexadecimal constant had type `int` only if its value fit in an `unsigned int`. Otherwise, an integral constant had type `long`. (At times the value did not fit in the resulting type.) In ANSI C, the constant type is the first type encountered in the following list that corresponds to the value:

unsuffixed decimal:
`int, long, unsigned long`

unsuffixed octal or hexadecimal:
`int, unsigned int, long, unsigned long`

`U` *suffixed:*
`unsigned int, unsigned long`

`L` *suffixed:*
`long, unsigned long`

`UL` *suffixed:*
`unsigned long`

The ANSI C compiler warns you about any expression with a behavior that might change according to the typing rules of the constants involved. The old integral constant typing rules are used only in the transition mode; the ANSI and conforming modes use the new rules.

Third Example: Integral Constants

In the following code, assume `ints` are 16 bits.

```
int f(void)
{
    int i = 0;

    return i > 0xffff;
}
```

Because the hexadecimal constant's type is either `int` (with a value of –1 on a two's-complement machine) or an `unsigned int` (with a value of 65535), the comparison will be true in `-Xs` and `-Xt` modes and false in `-Xa` and `-Xc` modes.

Again, an appropriate cast clarifies the code and silences the ANSI C compiler:

-Xt, -Xs modes:
```
    i > (int)0xffff
```
-Xa, -Xc modes:
```
    i > (unsigned int)0xffff
```
or
```
    i > 0xffffU
```

The `U` suffix character is a new feature of ANSI C and may produce an error message with older compilers.

Implicit Conversions

Single- and Double-Precision Calculations

Common C promotes the operands of floating point expressions to `double`. ANSI C allows operations on floats to be performed in single-precision calculations.

Also note that `-Xs` and `-Xt` modes promote single precision expressions to double precision, but `-Xa` and `-Xc` do not.

Characters and Integers

Any of the following may be used in an expression where an `int` or `unsigned int` may be used.

- `char`
- `short int`
- A `char`, `short`, or `int` bit-field
- The signed or unsigned varieties of any of these types
- An object or bit-field that has enumeration type

If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. This process is called *integral promotion*.

Note – The promotion rules for ANSI C are different from previous versions of C. The compiler warns about expressions where this may lead to different behavior.

Compilation Mode Dependencies

Under compilation modes `-Xs` and `-Xt`, `unsigned char` and `unsigned short` are promoted to `unsigned int` (unsigned preserving).

Under compilation modes `-Xa` and `-Xc`, `unsigned char` and `unsigned short` are promoted to `int`.

Signed and Unsigned Integers

When an integer is converted to another integral type, the value is unchanged if the value can be represented by the new type.

If a negative signed integer is converted to an unsigned integer with greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer.

Integral and Floating

When a floating type is converted to any integral type, any fractional part is discarded.

Float and Double

A `float` is promoted to `double` or `long double`, or a `double` is promoted to `long double` without a change in value.

The actual rounding behavior used when a floating point value is converted to a smaller floating point value depends on the rounding mode in effect at the time of execution. The default rounding mode is “round to nearest.”

Usual Arithmetic Conversions

Some binary operators convert the types of their operands in order to yield a common type, which is also the type of the result. These are called the *usual arithmetic conversions*.

- If either operand is type `long double`, the other operand is converted to `long double`.
- Otherwise, if either operand has type `double`, the other operand is converted to `double`.
- Otherwise, if either operand has type `float`, the other operand is converted to `float`.
- Otherwise, if the integral promotions are performed on both operands. Then, the following rules are applied:
  - If either operand has type `unsigned long long int`, then the other operator is converted to `unsigned long long int`. (Type `long long int` is not available in `-Xc` mode.)
  - If either operand has type `long long int`, then the other operator is converted to `long long int`.

- If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
- Otherwise, if one operand has type `long int` and the other has type `unsigned int`, both operands are converted to `unsigned long int.`
- Otherwise, if either operand has type `long int,` the other operand is converted to `long int.`
- Otherwise, if either operand has type `unsigned int,` the other operand is converted to `unsigned int.`
- Otherwise, both operands have type `int`.

Error Messages

This section contains the explanation for all the warning and error messages produced by the ANSI C compiler.

The compiler will display many of the messages shown in "C Error Messages," Chapter 6, from the *SPARCompiler C 2.0 Programmer's Guide* only when used with the `cc -v` option or the `acc -vc` option. With this option, the compiler performs stricter semantics checking and therefore displays more diagnostic messages.

The `lint` program checks your C code for errors. In many cases, `lint` warns you about incorrect, dangerous, or nonstandard code that the compiler will not necessarily flag. It is good coding practice to run `lint` on your source code before compiling.

Many warning and error messages are common to both `lint` and the compiler. The manual *Profiling Tools* contains a chapter on warning messages produced *only* by `lint` and not those error messages produced by the compiler.

Message Types and Applicable Options

Each message description includes a `Type` and an `Options` field.

`Type` indicates whether the message is a warning, an error, a fatal error, or a combination of error types.

`Options` indicates which `compiler` command options must be set for the message to appear. The `all` option implies that the message is independent of `compiler` options.

Warning messages, in which the word `warning:` appears before the filename and line number, provide useful information without interrupting compilation. They may diagnose a programming error, or a violation of C syntax or semantics for which the compiler will nevertheless generate valid object code.

Error messages, which lack the `warning:` prefix, cause the `compiler` command to fail. Errors occur when the compiler has diagnosed a serious problem that makes it unable to understand the program or to continue to generate correct object code. It will attempt to examine the rest of your program for other errors, however. The `compiler` will not link your program if the compiler diagnoses errors.

Fatal errors cause the compiler to stop immediately and return an error indication to the `compiler` command. A fatal error message is prefixed with the word `fatal:`. Such messages typically apply to start-up conditions, such as being unable to find a source file.

For more detailed information on compiler error messages, please refer to "C Error Messages", Chapter 6, from the *SPARCompilers C 2.0 Programmer's Guide*.

Other Considerations

New Keywords (`const` and `volatile`)

The keyword `const` was one of the C++ features that found its way into ANSI C. When an analogous keyword, `volatile`, was invented by the ANSI C committee, in effect, the `type qualifier` category was created. This still remains one of the more nebulous parts of ANSI C.

Types, Only for lvalues

The `const` and `volatile` keywords are part of an identifier's type, not its storage class. However, they are peculiar in that they are often removed from the topmost part of the type. This occurs when an object's value is fetched in the evaluation of an expression — exactly at the point when an `lvalue` becomes an `rvalue`. (These terms arise from the prototypical assignment `L=R` in which the left side must still refer directly to an object (an `lvalue`) and the right side need only be a value (an `rvalue`).) Thus, only expressions that are `lvalues` can be qualified by `const` or `volatile` or both.

Type Qualifiers in Derived Types

The type qualifiers are unique in that they may modify type names and derived types. Derived types are those parts of C's declarations that can be applied over and over to build more and more complex types: pointers, arrays, functions, structures, and unions. Except for functions, one or both type qualifiers can be used to change the behavior of a derived type.

For example:

```
const int five = 5;
```

declares and initializes an object with type `const int` whose value will not be changed by a correct program. The order of the keywords is not significant to C. For example, the declarations:

```
int const five = 5;
```

and

```
const five = 5;
```

are effectively identical to the preceding declaration.

The declaration:

```
const int *pci = &five;
```

declares an object with type pointer to `const int`, which initially points to the previously declared object. Note that the pointer itself does not have a qualified type; it points to a qualified type, and, as such, the pointer can be changed to point to essentially any `int` during the program's execution. However, `pci` cannot be used to modify the object to which it points unless a cast is used, as in the following:

```
*(int *)pci = 17;
```

(If `pci` actually points to a `const` object, the behavior of this code is undefined.)

The declaration:

```
extern int *const cpi;
```

says that somewhere in the program there exists a definition of a global object with type `const` pointer to `int`. In this case, the value of `cpi` is not changed by a correct program, but can be used to modify the object to which it points. Note that `const` comes after the `*` in the preceding declaration. The following pair of declarations produces the same effect:

```
typedef int *INT_PTR;
extern const INT_PTR cpi;
```

These can be combined as in the following declaration in which an object is declared to have type `const` pointer to `const int`.

```
const int *const cpci;
```

`const` Means Read Only

In hindsight, `readonly` would have been a better choice for a keyword than `const`. If your program reads `const` in this manner, declarations such as:

```
char *strcpy(char *, const char *);
```

are easily understood to mean the second parameter is only used to read character values, while the first parameter undoubtedly overwrites the characters to which it points. Furthermore, despite the fact that in the previous example, the type `cpi` is a pointer to a `const int`, you can still change the value of the object to which it points through some other means unless the type `cpi` actually points to an object declared with `const int` type.

Examples of `const` Usage

The two main uses for `const` are to declare large compile-time initialized tables of information as unchanging, and to specify that pointer parameters will not modify the objects to which they point.

The first use potentially allows portions of the data for a program to be shared by other concurrent invocations of the same program, just as the code for the program can be. This may cause attempts to modify this presumably invariant data to be detected immediately by means of some sort of memory protection fault because the data resides in a read-only portion of memory.

The second use most likely helps locate potential errors before generating a memory fault during that critical demo. For example, functions that temporarily place a null character into the middle of a string will be detected at compile time, if passed a pointer to a string that cannot be so modified.

`volatile` Means Exact Semantics

So far, the examples have all used `const` because it is conceptually simpler. The `volatile` keyword has multiple meanings to a programmer. To a compiler writer it has one meaning: take no code generation shortcuts when accessing such an object. Moreover in ANSI C, it is the programmer's responsibility to declare every object that has the appropriate special properties with a `volatile` qualified type.

The usual four examples of `volatile` objects are

1. An object that is a memory-mapped I/O port.
2. An object that is shared between multiple concurrent processes.
3. An object that is modified by an asynchronous signal handler.

4. An automatic storage duration object declared in a function that calls `setjmp` and whose value is changed between the call to `setjmp` and a corresponding call to `longjmp`.

The first three examples are all instances of an object with a particular behavior; its value can be modified at any point during the execution of the program. Thus, the seemingly infinite loop

```
flag = 1;
while (flag)
    ;
```

is completely reasonable as long as `flag` has a `volatile` qualified type. Presumably, some asynchronous event will set `flag` to zero in the future. Otherwise, the compilation system is free to change the above loop (because the value of `flag` is unchanged within the body of the loop) into a truly infinite loop that completely ignores the value of `flag`.

The fourth example, involving variables local to functions that call `setjmp`, is more involved. If you read the fine print about the behavior of `setjmp` and `longjmp`, you will find there are no guarantees about the values for objects matching the fourth case. It is necessary for `longjmp` to examine every stack frame between the function calling `setjmp` and the function calling `longjmp` for saved register values in order to get the most desirable behavior. The possibility of asynchronously created stack frames makes this expensive job even harder. Therefore, most implementations have documented the undesirable side effect and used an inexpensive implementation.

When an automatic object is declared with a `volatile` qualified type, the compilation system knows that it has to produce code that exactly matches what the programmer wrote. Therefore, the most recent value for such an automatic object will always be in memory (not just in a register) and, as such, will be guaranteed to be up-to-date when `longjmp` is called.

Common C treats the words `const`, `volatile`, and `signed` as ordinary identifiers. In ANSI C, these are keywords.

Code Example 2-10 key.c

```
#include <stdio.h>
main()
{
    int const=1, volatile=2, signed=3;
    printf("const=%d, volatile=%d, signed=%d\n", const, volatile,
        signed);
}
```

A compilation and execution of `key.c` in Common C looks like this:

```
% cc key.c
% a.out
const=1, volatile=2, signed=3
```

A compilation and execution of `key.c` in ANSI C with the `-Xs, -Xt, -Xa, and -Xc` switches looks like this:

```
% acc -Xc key.c
"key.c", line 5: syntax error before or at: =
"key.c", line 7: syntax error before or at: const
Compilation failed
```

Header Files

This section discusses the differences between ANSI C and Common C header files.

Standard Headers and Reserved Names

Very early in the standardization process, the ANSI Standards Committee chose to include library functions, macros, and header files as part of ANSI C. While this decision clearly was necessary for writing truly portable C programs, a side-effect is the basis of some of the most negative comments about ANSI C from the public: a large set of reserved names.

This section presents the various categories of reserved names and some rationale for their reservations. At the end is a set of rules to follow that can steer your programs clear of any reserved names.

Balancing Process

To match existing implementations, the ANSI C committee had to choose names like `printf` and `NULL`; to have done otherwise would have disqualified virtually all existing C programs from conformance, and would obviously run counter to their charter to standardize existing practice. However, each such name reduced the set of names available for free use in C programs.

On the other hand, before standardization implementors felt free to add both new keywords to their compilers and names to headers. This meant programs could not be guaranteed to compile from one release to another, let alone port from one vendor's implementation to another.

Thus, the ANSI C committee made a hard decision: to restrict all conforming implementations from including any extra names, except those with certain forms. It is this decision, more than any other, that causes most C compilation systems to be almost conforming. Nevertheless, the Standard contains 32 keywords and almost 250 names in its headers, none of which necessarily follow any particular naming pattern.

Standard Headers

The following are standard C headers:

```
assert.h          locale.h          stddef.h
ctype.h           math.h            stdio.h
errno.h           setjmp.h          stdlib.h
float.h           signal.h          string.h
limits.h          stdarg.h          time.h
```

Most implementations will provide more headers, but a strictly conforming ANSI C program can only use these.

Other standards disagree slightly regarding the contents of some of these headers. For example, POSIX (IEEE 1003.1) specifies that `fdopen` is declared in `stdio.h`. To allow these two standards to coexist, POSIX requires that you `#define` the macro `_POSIX_SOURCE` prior to the inclusion of any header to guarantee these additional names exist. (In actuality, the POSIX committee believes that almost the opposite will occur: that `_POSIX_SOURCE` will be used to limit headers to only those names POSIX describes, and that, by default, headers will contain even more names than POSIX specifies.) The *X/Open Portability Guide* also uses this macro scheme for its extensions. X/Open's macro is `_XOPEN_SOURCE`. The following section describes why this scheme is sufficient.

ANSI C requires the standard headers to be both self-sufficient and idempotent. In other words, no standard header needs any other header to be `#include`d before or after it, and each standard header can be `#include`d more than once without causing problems. The Standard also requires that its headers be `#include`d only in safe contexts so that the names used in the headers are guaranteed to remain unchanged.

Names Reserved for Implementation Use

The Standard places further restrictions on implementations regarding their libraries. While in the past, most programmers learned not to use names like `read` and `write` for their own functions on UNIX systems (usually after encountering interesting program behavior), ANSI C requires that only names reserved by the Standard be introduced by references within the implementation.

Thus the Standard reserves a subset of all possible names for implementations to use as they so choose. This class of names consists of identifiers that begin with an underscore and continue with either another underscore or a capital letter. In other words, the class of names contains all names matching the following regular expression:

```
_[_A-Z][0-9_a-zA-Z]*
```

Strictly speaking, if your program uses such an identifier, its behavior is undefined. Thus, programs using `_POSIX_SOURCE` (or `_XOPEN_SOURCE`) have undefined behavior.

However, undefined behavior comes in different degrees. If in a POSIX-conforming implementation you use `_POSIX_SOURCE`, you know that your program's *undefined behavior* consists of certain additional names in certain headers, and your program still conforms to an accepted standard. This deliberate loophole in the ANSI C standard allows implementations to conform to seemingly incompatible specifications. On the other hand, an implementation that does not conform to the POSIX standard is free to behave in any manner when encountering a name such as `_POSIX_SOURCE`.

The Standard also reserves all other names that begin with an underscore for use in header files as regular file scope identifiers and as tags for structures and unions, but not in local scopes. This means that the common existing practice of having functions named `_filbuf` and `_doprnt` to implement hidden parts of the library is sanctioned.

Names Reserved for Expansion

In addition to all the names explicitly reserved, the Standard also reserves (for implementations and future standards) names matching the patterns shown in Table 2-6.

Table 2-6 Reserved Names

File	Reserved Name Pattern
`errno.h`	`E[0-9A-Z].*`
`ctype.h`	`(to|is)[a-z].*`
`locale.h`	`LC_[A-Z].*`
`math.h`	current function names `[fl]`
`signal.h`	`(SIG|SIG_)[A-Z].*`
`stdlib.h`	`str[a-z].*`
`string.h`	`(str|mem|wcs)[a-z].*`

In the lists shown in Table 2-6, names that begin with a capital letter are macros and are thus reserved only when the associated header is included. The rest of the names designate functions and therefore cannot be used to name any global objects or functions.

Names Safe To Use

As you can tell by now, the rules regarding when certain names are reserved are complicated. There are, however, four fairly simple rules you can follow to keep from colliding with any ANSI C reserved names:

1. `#include` all system headers at the top of your source files (except possibly after a `#define` of `_POSIX_SOURCE`, `_XOPEN_SOURCE`, or both).
2. Do not define or declare any names that begin with an underscore.
3. Use an underscore or a capital letter somewhere within the first few characters of all file scope tags and regular names, but beware of the "`va_`" prefix found in `stdarg.h` and `varargs.h`.
4. Use a digit or a non-capital letter somewhere within the first few characters of all macro names, but note that almost all names beginning with an `E` are reserved if you `#include errno.h`.

As noted earlier, most implementations will continue to add names to the standard headers by default. These rules are therefore just guidelines.

Grouping and Expression Evaluations

Grouping and Evaluation in Expressions

One of the choices made by Dennis Ritchie in the design of C was to give compilers license to rearrange expressions involving adjacent operators that are mathematically commutative and associative, even in the presence of parentheses. This was explicitly noted as being so in the "Reference Manual" appendix of the *The C Programming Language* by Kernighan and Ritchie. However, ANSI C does not grant compilers this same freedom. Please see the end of this section for an example.

This section discusses the differences between these two definitions of C and clarifies the distinctions between an expression's side effects, grouping, and evaluation by considering the expression statement from the following code fragment.

```
int i, *p, f(void), g(void);
/*...*/
i = *++p + f() + g();
```

Definitions

The side effects of an expression are its modifications to memory and its accesses to `volatile` qualified objects. The side effects in the above expression are the updating of `i` and `p` and any side effects contained within the functions `f()` and `g()`.

An expression's *grouping* is the way values are combined with other values and operators. The above expression's grouping is, primarily, the order in which the additions are performed.

An expression's *evaluation* includes everything necessary to produce its resulting value. To evaluate an expression, all specified side effects must occur anywhere between the previous and next sequence point, and the specified operations must be performed with a particular grouping. For the preceding expression, `i` and `p` must be updated after the previous statement and by the `;` of this expression statement. Calls to the functions can occur in either order any time after the previous statement, but must occur before their return values are used. In particular, note that the operators that cause memory to be updated are not required to assign the new value before the value of the operation is used.

The K&R C Rearrangement License

The K&R C rearrangement license applies to the preceding expression because addition is mathematically commutative and associative. To distinguish between regular parentheses and the actual grouping of an expression, the left and right curly braces will designate grouping. The three possible groupings for the expression are:

```
i = { {*++p + f()} + g() };
i = { *++p + {f() + g()} };
i = { {*++p + g()} + f() };
```

all of which are valid given K&R C rules. Moreover, all of these groupings are valid even if the expression were written instead, for example, in either of these ways:

```
i = *++p + (f() + g());
i = (g() + *++p) + f();
```

If this expression is evaluated on an architecture for which either overflows cause an exception or addition and subtraction are not inverses across an overflow, the first three groupings behave differently if one of the addition operations overflows.

For such expressions on these architectures, the only recourse available in K&R C was to split the expression to force a particular grouping. The following are possible rewrites that respectively enforce the above three groupings.

```
i = *++p; i += f(); i += g();
i = f(); i += g(); i += *++p;
i = *++p; i += g(); i += f();
```

The ANSI C Rules

ANSI C does not allow operations to be rearranged that are mathematically commutative and associative, but that are not actually so on the target architecture. Thus the precedence and associativity of the ANSI C grammar completely describes the grouping for all expressions; all expressions must be grouped as they are parsed. The expression under consideration is grouped in this manner:

```
i = { {*++p + f()} + g() };
```

This still does not mean that `f()` must be called before `g()`, or that `p` must be incremented before `g()` is called.

In ANSI C, expressions need not be split to guard against unintended overflows.

Using Parentheses in ANSI C

ANSI C is often erroneously described as honoring parentheses or evaluating according to parentheses due to an incomplete understanding or an inaccurate presentation.

Since ANSI C expressions have the grouping specified by their parsing, parentheses still only serve as a way of controlling how an expression is parsed; the natural precedence and associativity of expressions carry exactly the same weight as parentheses.

The above expression could have been written as:

```
i = (((*(++p)) + f()) + g());
```

with no different effect on its grouping, and, thus, on its evaluation.

The As If Rule

There were good reasons for the K&R C rearrangement rules:

- The rearrangements provide many more opportunities for optimizations such as compile-time constant folding.

- The rearrangements do not change the result of integral-typed expressions on most machines.
- Some of the operations are both mathematically and computationally commutative and associative on all machines.

The ANSI C committee eventually became convinced that the rearrangement rules were intended to be an instance of the *as if* rule when applied to the described target architectures. ANSI C's *as if* rule is a general license that permits an implementation to deviate arbitrarily from the abstract machine description as long as the deviations do not change the behavior of a valid C program.

Thus, all the binary bitwise operators other than bit-shifting are allowed to be rearranged on any machine because there is no way to notice such regroupings. On typical two's-complement machines in which overflow silently wraps around, integer expressions involving multiplication or addition can be rearranged for the same reason. Therefore, this change in C does not have a significant impact on most C programmers.

The following example illustrates grouping and expression evaluation rules.

Note – Regarding assignment operators, Common C treats the following operator pairs as two tokens, and as a consequence, permits whitespace between them. ANSI C treats them as a single tokens, and therefore disallows white space between them.

Code Example 2-11 operator.c

```
#include <stdio.h>
main()
{
    int i=2;
    i * = 10;
    printf("i=%d\n", i);
}
```

A compilation and execution of `operator.c` in Common C looks like this:

```
% cc operator.c
% a.out
i=20
```

A compilation and execution of `key.c` in ANSI C with the `-Xs, -Xt, -Xa, and -Xc` switches looks like this:

```
% acc -Xc operator.c
"operator.c", line 7: whitespace between two character assignment operators
acc: Fatal error in acomp: Bus error (core dumped)
Compilation failed
```

Incomplete Types

The ANSI C standard introduced the term *incomplete type* to formalize a fundamental, yet misunderstood, portion of C, implicit from its beginnings. This section describes incomplete types, where they are permitted, and why they are useful.

ANSI C Types

ANSI separates C's types into three distinct sets: function, object, and incomplete. Function types are obvious; object types cover everything else, except when the size of the object is not known. The Standard uses the term *object type* to specify that the designated object must have a known size. However, incomplete types other than `void` also refer to an object.

There are only three variations of incomplete types: `void`, arrays of unspecified length, and structures and unions with unspecified content. The type `void` differs from the other two because it is an incomplete type that cannot be completed; it serves as a special function return and parameter type.

Completing Incomplete Types

An array type is completed by specifying the array size in a following declaration in the same scope that denotes the same object. When an array without a size is declared and initialized in the same declaration, the array has an incomplete type only between the end of its declarator and the end of its initializer.

To complete an incomplete structure or union type, specify the content in a following declaration in the same scope for the same tag.

Declarations

Certain declarations can use incomplete types, but others require complete object types. Those declarations that require object types are array elements, members of structures or unions, and objects local to a function. All other declarations permit incomplete types. In particular, the following are permitted:

- pointers to incomplete types
- functions returning incomplete types

- incomplete function parameter types
- `typedef` names for incomplete types

The function return and parameter types are special. Except for `void`, an incomplete type used in such a manner must be completed by the time the function is defined or called. A return type of `void` specifies a function that returns no value; a single parameter type of `void` specifies a function that accepts no arguments.

Note that since array and function parameter types are rewritten to be pointer types, a seemingly incomplete array parameter type is not actually incomplete. The typical `char *argv[]` declaration in `main` as an unspecified length array of character pointers is rewritten to be a pointer to character pointers.

Expressions

Most expression operators require complete object types. The only three exceptions are the unary `&` operator, the first operand of the comma operator, and the second and third operands of the `?:` operator. Most operators that accept pointer operands also permit pointers to incomplete types, unless pointer arithmetic is required. The list includes the unary `*` operator. For example, given:

```
void *p
```

`&*p` is a valid subexpression that uses this.

Why Incomplete Types are Necessary

C might have been simpler without incomplete types. However, other than `void`, there is only one feature provided by incomplete types that C has no other way to handle, and that has to do with forward references to structures and unions. If your program has two structures that need pointers to each other, the only way to do so without resorting to potentially invalid casts is with incomplete types as follows:

```
struct a { struct b *bp; };
struct b { struct a *ap; };
```

All strongly typed programming languages that have some form of pointer and heterogeneous data types provide some method of handling this case.

Defining `typedef` names for incomplete structure and union types is frequently quite useful. If your program has a complicated bunch of data structures that contain many pointers to each other, having a list of `typedef`s to the structures in the beginning, possibly in a central header, can simplify the declarations.

```
typedef struct item_tag Item;
typedef union note_tag Note;
typedef struct list_tag List;
    .  .  .
struct item_tag { . . . };
    .  .  .
struct list_tag {
    List *next; . . .
};
```

Moreover, for those structures and unions whose contents should not be available to the rest of the program, a header can declare the tag without the content. Unless other parts of the program attempt to use any of the structure or union's members, these program parts can use pointers to the incomplete structure or union without any problems.

A frequently used incomplete type is an external array of unspecified length. You generally do not need to know the extent of an array to use its contents.

Compatible and Composite Types

With K&R C, and even more so with ANSI C, it is possible for two declarations that refer to the same entity to be other than identical. The term *compatible type* is used in ANSI C to denote those types that are similar. This section describes compatible types as well as *composite types*—the result of combining two compatible types.

- **Multiple Declarations**

  If a C program were only allowed to declare each object or function once, there would be no need for compatible types. But linkage (which allows two or more declarations to refer to the same entity), function prototypes, and separate compilation all need such a capability. Not too surprisingly, separate translation units (source files) have different rules for type compatibility than within a single translation unit.

- **Separate Compilation Compatibility**

  Since each compilation may look at different source files, most of the following rules for compatible types across separate compiles are structural in nature.

  - Matching scalar (integer, floating, and pointer) types must be compatible, as if they were in the same source file.

- Matching structures, unions, and enums must have the same number of members and each matching member must have a compatible type in the separate compilation sense, including bit-field widths.
- Matching structures must have the members in the same order (the order of union and enum members does not matter).
- Matching enum members must have the same value.
- The names of members, including the lack of names for unnamed members, must match for structures, unions, and enums, but not necessarily their respective tags.

- **Single Compilation Compatibility**

When two declarations in the same scope describe the same object or function, the two declarations must specify compatible types. These two types are then combined into a single composite type that is compatible with the first two.

Compatible types are defined recursively. At the bottom are type specifier keywords. These are the rules that say that `unsigned short` is the same as `unsigned short int`, and that a type without type specifiers is the same as one with `int`. All other types are compatible only if the types from which they are derived are compatible. For example, two qualified types are compatible if the qualifiers (`const` and `volatile`) are identical and the unqualified base types are compatible.

- **Compatible Pointer Types**

For two pointer types to be compatible, the types they point to must be compatible, and the two pointers must be identically qualified. Recall that the qualifiers for a pointer are specified after the `*`, so that these two declarations:

```
int *const cpi;
int *volatile vpi;
```

declare two differently qualified pointers to the same type, `int`.

- **Compatible Array Types**

For two array types to be compatible, their element types must be compatible, and, if both array types have a specified size, they must match. This last part means that an incomplete array type is compatible with another incomplete array type, as well as with an array type with a specified size. See "Incomplete Types" earlier in this chapter for more information.

- **Compatible Function Types**

For two function types to be compatible, their return types must be compatible. If either or both function types have prototypes, the rules get more complicated.

For two function types with prototypes to be compatible, they also must have the same number of parameters (including use of the ellipsis (`...`) notation), and the corresponding parameters must be parameter-compatible.

For an old-style function definition to be compatible with a function type with a prototype, the prototype parameters must not end with an ellipsis (`...`), and each of the prototype parameters must be parameter-compatible with the corresponding old style parameter after application of the default argument promotions.

For an old-style function declaration to be compatible with a function type with a prototype, the prototype parameters must not end with an ellipsis (`...`); all of the prototype parameters must have types that are unaffected by the default argument promotions.

For two types to be parameter-compatible, the types must be compatible after the top level qualifiers, if any, have been removed, and after a function or array type has been converted to the appropriate pointer type.

- **Special Cases**

  Not all cases perform by the aforementioned rules. For example, `signed int` behaves the same as `int` except possibly for bit-fields, in which an `int` may denote an unsigned-behaving quantity.

  Also, each enumeration type must be compatible with some integral type. In portable programs, this means that enumeration types are effectively separate types and, for the most part, the ANSI C standard views enumeration types in that manner.

- **Composite Type**

  The construction of a composite type from two compatible types is also recursively defined. Compatible types can differ from each other by either incomplete arrays or old-style function types. As such, the *composite type* is the type compatible with both of the original types, including every available array size and every available parameter list from the original types.

Internationalization Support

Multibyte Characters and Wide Characters

At first, the internationalization of ANSI C affected only library functions. However, the final stage of internationalization—multibyte characters and wide characters—also affected the language proper.

- **Asian Languages Require Multibyte Characters**

The basic difficulty in an Asian-language computer environment is the huge number of ideograms needed for I/O. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes. The associated operating systems, application programs, and terminals understand these byte sequences as individual ideograms. Moreover, all of these encodings allow intermixing of regular single-byte characters with the ideogram byte sequences. Just how difficult it is to recognize distinct ideograms depends on the encoding scheme used.

The term *multibyte character* is defined by ANSI C to denote a byte sequence that encodes an ideogram in whatever encoding scheme is employed. All multibyte characters are members of the *extended character set*. (Note that a regular single-byte character is just a special case of a multibyte character.) Essentially the only requirement placed on the encoding is that no multibyte character can use a null character as part of its encoding.

ANSI C specifies that program comments, string literals, character constants, and header names are all sequences of multibyte characters.

- **Encoding Variations**

  The encoding schemes fall into two categories. The first is one in which each multibyte character is self-identifying; any multibyte character can be inserted between any pair of multibyte characters.

  The second scheme is one in which the presence of special *shift bytes* changes the interpretation of subsequent bytes, such as the method used by most fancy character terminals to get in and out of line drawing mode. For programs written in multibyte characters with a shift-state-dependent encoding, ANSI C has the additional requirement that each comment, string literal, character constant, and header name must both begin and end in the unshifted state.

- **Wide Characters**

  There can be tens of thousands of ideograms in such a character set. Therefore, you should use a 16-bit or 32-bit sized integral value to hold all members. For example, the full Chinese alphabet includes more than 65000 ideograms. ANSI C includes the `typedef` name `wchar_t` as the implementation-defined integral type large enough to hold all members of the extended character set.

  For each wide character there is a corresponding multibyte character and vice versa; the wide character that corresponds to a single-byte character is required to have the same value as its single-byte value, including the null character. However, there is no guarantee that the value of the macro `EOF` can be stored in a `wchar_t`, just as `EOF` might not be representable as a `char`.

- **Conversion Functions**

ANSI C provides five library functions that manage multibyte characters and wide characters; they are shown in Table 2-7.

Table 2-7 Multibyte Character Conversion Functions

`mblen()`	length of next multibyte character
`mbtowc()`	convert multibyte character to wide character
`wctomb()`	convert wide character to multibyte character
`mbstowcs()`	convert multibyte character string to wide character string
`wcstombs()`	convert wide character string to multibyte character string

The behavior of all of these functions depends on the current locale. See the chapter in this book on "Internationalization" for information on locales.

It is expected that vendors providing compilation systems targeted to this market will supply many more string-like functions to simplify the handling of wide character strings. However, for most application programs, there is no need to convert any multibyte characters to or from wide characters. Programs such as `diff`, for example, will read in and write out multibyte characters, needing only to check for an exact byte-for-byte match. More complicated programs (such as `grep`) that use regular expression pattern matching, may need to understand multibyte characters, but only the common set of functions that manages the regular expression needs this knowledge. The program `grep` itself requires no other special multibyte character handling.

- **C Language Features**

To give you more flexibility in an Asian-language environment, ANSI C provides wide character constants and wide string literals. These character constants, shown in Table 2-8, have the same form as their non-wide versions except that they are immediately prefixed by the letter `L`.

Table 2-8 Wide Character Constants and String Literals

Character	Representation
`'x'`	regular character constant
`'¥'`	regular character constant
`L'x'`	wide character constant
`L'¥'`	wide character constant
`"abc¥xyz"`	regular string literal
`L"abcxyz"`	wide string literal

Note that multibyte characters are valid in both the regular and wide versions. The sequence of bytes necessary to produce the ideogram ¥ is encoding-specific, but if it consists of more than one byte, the value of the character constant `'¥'` is implementation-defined, just as the value of `'ab'` is implementation-defined. A regular string literal contains exactly the bytes (except for escape sequences) specified between the quotes, including the bytes of each specified multibyte character.

When the compilation system encounters a wide character constant or wide string literal, each multibyte character is converted (as if by calling the `mbtowc()` function) into a wide character. Thus the type of `L'¥'` is `wchar_t` and the type of `abc¥xyz` is array of `wchar_t` with length eight. (Just as with regular string literals, each wide string literal has an extra zero-valued element appended, but in these cases it is a `wchar_t` with value zero.)

Just as regular string literals can be used as a short-hand method for character array initialization, wide string literals can be used to initialize `wchar_t` arrays, as shown in the following example:

```
wchar_t *wp = L"a¥z";
wchar_t x[] = L"a¥z";
wchar_t y[] = {L'a', L'¥', L'z', 0};
wchar_t z[] = {'a', L'¥', 'z', '\0'};
```

In the above example, the three arrays `x`, `y`, and `z`, as well as the array pointed to by `wp`, have the same length and all are initialized with identical values.

Finally, adjacent wide string literals will be concatenated, just as with regular string literals. However, adjacent regular and wide string literals produce undefined behavior. A compiler is not required to complain if it does not accept such concatenations.

Changed Functions

Wherever possible and appropriate, existing library functions have been extended to include locale-dependent behavior. These functions come in two groups: those declared by the `ctype.h` header (character classification and conversion), and those functions that convert to and from printable and internal forms of numeric values, such as `printf()` and `strtod()`.

All `ctype.h` predicate functions except `isdigit()` and `isxdigit()` are allowed to return nonzero (true) for additional characters when the `LC_CTYPE` category of the current locale is not "`C`." For example, in a Spanish locale, `isalpha('ñ')` is true. Similarly, the character conversion functions `tolower()` and `toupper()` handle any extra alphabetic characters identified by the `isalpha()` function. The `ctype.h` functions are almost always macros that are implemented using table lookups indexed by the character argument. Their behavior is changed by resetting the tables to the new locale's values, thus causing no impact on software performance.

Those functions that write or interpret printable floating values may change to use a decimal-point character other than period (`.`) when the `LC_NUMERIC` category of the current locale is other than "`C`." There is no provision for converting any numeric values to printable form with thousands separator-type characters, but when converting from a printable form to an internal form, implementations are allowed to accept such additional forms, again in other than the "`C`" locale. Those functions that make use of the decimal-point character are the `printf()` and `scanf()` families, `atof()`, and `strtod()`. Those functions that are allowed implementation-defined extensions are `atof()`, `atoi()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, and the `scanf()` family.

New Functions

Several locale-dependent capabilities were added as new standard functions. In addition to `setlocale()` which allows control over the locale itself and the multibyte functions previously discussed (`mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, and `wcstombs()`), the Standard includes the following new functions:

`localeconv()`	numeric/monetary conventions
`strcoll()`	collation order of two strings
`strxfrm()`	translate string for collation
`strftime()`	formatted date/time conversion

The `localeconv()` function returns a pointer to a structure containing information useful for formatting numeric and monetary information appropriate to the current locale's `LC_NUMERIC` and `LC_MONETARY` categories.

The `localeconv()` function is the only function with behavior that depends on more than one category. For numeric values the structure describes the decimal-point character, the thousands separator, and where the separator(s) should be located. There are fifteen other structure members that describe how to format a monetary value!

The `strcoll()` function is analogous to the `strcmp()` function except that the two strings are compared according to the `LC_COLLATE` category of the current locale. Because this comparison is sometimes more expensive than `strcmp()`, you can also use the `strxfrm()` function to transform a string into another. Using this method, any two such translated strings passed to `strcmp()` will return an ordering analogous to what `strcoll()` would have returned if it had been passed the two pre-translation strings.

The `strftime()` function provides formatting of the values in a `struct tm`. This formatting is similar to that used with `sprintf()`. This function also provides date and time representations that depend on the `LC_TIME` category of the current locale. The `strftime()` function is based on the `ascftime()` function released as part of UNIX System V Release 3.2.

Bibliography

The C Language, Second Edition, Kernighan and Ritchie, Prentice Hall, Englewood Cliffs, NJ.

C: A Reference Manual, Second Edition, Harbison and Steele, Prentice Hall, Englewood Cliffs, NJ.

System V ANSI C and Programming Support Tools, UNIX System Laboratories, Prentice Hall, Englewood Cliffs, NJ.

SPARCompilers C 2.0 Transition Guide, Part Number 800-6579-05, SunPro, Inc., Mountain View, CA.

SPARCompiler C 2.0 Programmer's Guide, Part Number 800-6578-10, SunPro, Inc., Mountain View, CA.