Transition Guide

SPARCompiler™ C

Version 2.0

SunPro

A Sun Microsystems, Inc. Business

SPARCompilers C 2.0.1 Transition Guide

A Sun Microsystems, Inc. Business

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No:800-6579-11
Revision A, October 1992

2550 Garcia Avenue, Mountain View, California 94043-1100

The OPEN LOOK and the Sun Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 (October 1988) and FAR 52.227-19 (June 1987).

The product described in this manual may be protected by one or more U.S. patents, foreign patents, and/or pending applications.

TRADEMARKS

The Sun logo, Sun Microsystems, Sun Workstation, NeWS, and SPARCompilers are registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SunCD, SunInstall, SunOS, SunView, NFS, and OpenWindows are trademarks of Sun Microsystems, Inc.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

X Window System is a product of the Massachusetts Institute of Technology.

SPARC is a registered trademark of SPARC International, Inc. Products bearing the SPARC trademark are based on an architecture developed by Sun Microsystems, Inc. SPARCstation is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.

All other products or services mentioned in this document are identified by the trademarks, service marks, or product names as designated by the companies who market those products. Inquiries concerning such trademarks should be made directly to those companies.

Portions © AT&T 1983-1990 and reproduced with permission from AT&T.

Contents

Tables

Preface

C Transition Guide covers the following areas:

- features of ANSI C, such as internationalization and prototyping
- differences between ANSI standard-conformant C and other versions of C

Refer to these other manuals for more information on programming in ANSI C:

- *C 2.0.1 Programmer's Guide*
  A reference manual to the C language and the ANSI C compiler.
- *Profiling Tools*
  Information on many helpful programming tools, such as `prof`(1), `gprof`(1), and various profiling tools.

We recommend two texts for programmers new to the C language: Kernighan and Ritchie, *The C Language*, Second Edition, 1988, Prentice-Hall; Harbison and Steele, *C: A Reference Manual*, Second Edition, 1987, Prentice-Hall. For implementation-specific details not covered in this book, refer to the *Application Binary Interface* for your machine.

This manual uses the following conventions:

`Bold face typewriter font`
  Indicates commands that you should type in exactly as printed in the manual.

`Regular typewriter font`
  Represents what the system prints on your workstation screen, as well as keywords, identifiers, program names, filenames and names of libraries.

Italic font
Indicates variables or parameters that you should replace with an appropriate word or string. It is also used for emphasis.

`$`
Represents your system prompt for a non-privileged user account.

Transitioning to ANSI C

Introduction

This guide describes techniques for writing new and upgrading existing C code to comply with the ANSI C language specification. The information is presented as a series of articles, each covering a specific transition topic. These articles were originally written for an in-house AT&T newsletter by David Prosser, Distinguished Member of Technical Staff, AT&T Bell Laboratories. Comments by Vijay Tatkar and Walter Nielsen of Sun Microsystems have also been incorporated.

This guide 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 / Sun ANSI C Differerences," 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 5.0).

Appendix C, "-Xs Differences for Sun C and ANSI C," describes the differences in compiler behavior when using the -Xs option.

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 intrepretation.

`-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 computer warns about all language constructs that have differing 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.

Mixing Old and New Style Functions

ANSI C's most sweeping change to the language is the function prototype borrowed from the C++ language. By specifying for each function the number and types of its parameters, not only does every regular compile get the benefits of argument/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. ANSI C includes rules that govern the mixing of old- and new-style function declarations since there are many, many lines of existing C code that could and should be converted to use prototypes.

Writing New Code

When you write an entirely new program, use new-style function declarations (function 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, we suggest you use the macro __STDC__ (which is defined only for ANSI C compilation systems) in both header and source files, as we show later.

Because an ANSI C conforming compiler must issue a diagnostic whenever two incompatible declarations for the same object or function are in the same scope, 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 — thus eliminating one of the most common C programming mistakes.

Updating Existing Code

If you have an existing application and want the benefits of function prototypes, there are a number of possibilities for updating, 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 precisely the ones that require detailed knowledge of the rules for mixing of old and new styles.

Mixing Considerations

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

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.

Just what should be used instead of an `unsigned short` leads us into the final complication. The one biggest incompatibility 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. (See "Promotions: Unsigned vs. Value Preserving" on page 10.) 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 (even though it violates the spirit of choices 2 and 3 above) 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 its value to a local variable with the narrower type, if necessary, after you enter the function).

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; {...}
```

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 (as with choice 3 above). The local function still uses an old style definition, but a prototype is included for newer compilers:

```
source.c:

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

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`.

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 file name, or a line number, or both, and these are followed by format and arguments, similar to those of `printf()`, 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. First, the header to include depends on the compilation system:

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

(`stdio.h` is included because we call `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 us to specify any fixed parameters (at least not officially), we need 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.

Sun ANSI C, as an extension, 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()` could be:

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

Promotions: Unsigned vs. 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 our code.

Background

According to Kernighan and Ritchie, *The C Programming Language* (First Edition, hereafter referred to as K&R) `unsigned` specified exactly one type; there were no `unsigned chars`, `unsigned shorts`, or `unsigned longs`, 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* — was 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, came to be called *value preserving*, in which 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 would be `unsigned int`. The *value preserving* rule produces the *least surprise* arithmetic result for most expressions.

Compilation Behavior

Only in the transition or pre-ANSI modes (`-Xt` or `-Xs`), will the ANSI C compiler use the unsigned preserving promotions; in the other two modes, conforming (`-Xc`) and ANSI (`-Xa`), the value preserving promotion rules will be used. No matter what the current mode may be, the compiler will warn you about each expression whose behavior might depend on the promotion rules used.

This warning is not optional since 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 below.

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 code above causes the compiler to issue the following:

```
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
```

Because this expression can be viewed as ambiguous (since differing compilers chose different meanings for the same code), 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 the two 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 list below 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 will warn you about any expression whose behavior 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 `int`s 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 will probably produce an error message with older compilers.)

Tokenization and Preprocessing

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 tokens, ready to parse. 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.

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) and are three-character sequences that name a character not in the ISO 646-1983 character set:

Table 1-1 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/new-line 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.
4. Every preprocessing directive is handled and all macro invocations are replaced. Each `#included` source file is run through the earlier phases before its contents replace the directive line.

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 these 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/new-line 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/new-line 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/new-line pair will not behave as expected.

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 tutorial 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.

Using Strings

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

In ANSI C, the examples below 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.

```
#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)
```

The above 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 was similar to the following:

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

which produced

```
(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)
```

Again, ANSI C could not sanction either approach to what they believed to be an important capability. In ANSI C, both the above invocations would produce the two separate tokens `x` and `1`. The second of the above two methods can be rewritten for ANSI C by using the `##` macro substitution operator:

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

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

Since `##` 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.

Using `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

`const` and `volatile` are part of an identifier's type, not its storage class. However, they are peculiar in that they are often removed from the top-most 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 identical to the above declaration in its effect.)

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, but `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, `cpi`'s value will not be changed by a correct program, but it can be used to modify the object to which it points. Notice that `const` comes after the `*` in the above 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* `readonly`

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

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

are easily understood to mean that the second parameter will only be used to read character values, while the first parameter will undoubtedly overwrite the characters to which it points. Furthermore, despite the fact that in the above example the type of `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 it 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), and may cause attempts to modify this presumably invariant data to be detected immediately by means of some sort of memory protection fault, as the data resides in a read-only portion of memory.

The second use will most likely help 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's conceptually simpler. But what does `volatile` really mean? To the programmer, it has multiple meanings. To a compiler writer (remember that the ANSI C committee was principally composed of implementors) it has one meaning: take no code generation shortcuts when accessing such an object. Moreover in ANSI C, it is a programmer's responsibility to declare every object that has the appropriate special properties with a `volatile` qualified type.

Examples of `volatile` *Usage*

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 turns out to be 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 just 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.

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, no matter what encoding scheme is employed. All multibyte characters are members of the so called *extended character set*. (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 camps. The first is one in which each multibyte character is self-identifying, or, in other words, any multibyte character can simply 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. An example is 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

Some of the inconvenience of handling multibyte characters would be eliminated if all characters were of a uniform number of bytes or bits. Since there can be thousands or tens of thousands of ideograms in such a character set, a 16-bit or 32-bit sized integral value should be used to hold all members. (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 regular 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:

Table 1-2 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 setlocale() Function" on page 30.)

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 even more flexibility to the programmer in an Asian-language environment, ANSI C provides wide character constants and wide string literals. These have the same form as their non-wide versions except that they are immediately prefixed by the letter `L`:

'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

Notice 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:

```
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`, and 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 even required to complain if it does not accept such concatenations.

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 the writing of 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

In order 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 that no program could be guaranteed to compile from one release to another, let alone port from one vendor's implementation to another.

Thus the 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 will cause 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 standard headers are as follows:

Table 1-3 Standard 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 the macro `_POSIX_SOURCE` to be `#defined` prior to the inclusion of any header to guarantee that 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.) X/Open, in its *Portability Guide*, has also used 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 `#included` before or after it, and each standard header can be `#included` more than once without causing problems. The Standard also requires that its headers be `#included` 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 certain patterns:

Table 1-4 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 above lists, 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` or `_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` or `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 `errno.h` is `#include`-d.)

As noted earlier, most implementations will continue to add names to the standard headers by default. Therefore these rules are just a guideline to follow.

Internationalization

A previous transition topic from this series introduced the *internationalization* of the standard libraries. (See "Multibyte Characters and Wide Characters" on page 23.) This section discusses the affected library functions and gives some hints on how programs should be written to take advantage of these features.

Locales

At any time, a C program has a current *locale* — a collection of information that describes the conventions appropriate to some nationality, culture, and language. Locales have names that are strings and the only two standardized locale names are `"C"` and `""`. Each program begins in the `"C"` locale which unsurprisingly causes all library functions to behave just like they have historically. The `""` locale is the implementation's best guess at the correct set of conventions appropriate to the program's invocation. (Of course `"C"` and `""` can cause identical behavior.) Other locales may be provided by implementations.

For the purposes of practicality and expediency, locales are partitioned into a set of categories. A program can change the complete locale (all categories) or one or more categories leaving the other categories unchanged. Generally each category affects a set of functions disjoint from the functions affected by other categories, so temporarily changing one category for a little while can make sense.

The `setlocale()` *Function*

The `setlocale()` function is the interface to the program's locale. In general, any program that wishes to use the invocation country's conventions should place a call such as

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

early in the program's execution path. This causes the program's current locale to change to the appropriate local version (if possible), since LC_ALL is the macro that specifies the entire locale instead of one category. The following are the standard categories:

Table 1-5 `setlocale()` Standard Categories

LC_COLLATE	sorting information
LC_CTYPE	character classification information
LC_MONETARY	currency printing information
LC_NUMERIC	numeric printing information
LC_TIME	date and time printing information

Any of these macros can be passed as the first argument to `setlocale()` to specify just that category.

The `setlocale()` function returns the name of the current locale for a given category (or LC_ALL) and serves in an inquiry-only capacity when its second argument is a null pointer. Thus, code along the lines of the following can be used to change the locale or a portion thereof for a limited duration:

```
#include <locale.h>
/*...*/
char *oloc;
/*...*/
oloc = setlocale(LC_cat, NULL);
if (setlocale(LC_cat, "new") != 0)
{
          /* use temporarily changed locale */
    (void)setlocale(LC_cat, oloc);
}
```

Most programs will never need this capability.

Changed Functions

Wherever possible and appropriate, existing library functions were extended to include locale-dependent behavior. These functions came 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 (for example, `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 other than `"C"`. In a Spanish locale, `isalpha('ñ')` should be true. Similarly the character conversion functions (`tolower()` and `toupper()`) should appropriately handle any extra alphabetic characters identified by the `isalpha()` function. (As an implementation note, 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 table(s) to the new locale's values, and therefore there is no performance impact.)

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

Certain locale-dependent capabilities were added as new standard functions. Besides `setlocale()` which allows control over the locale itself, 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

and the multibyte functions previously discussed (`mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, and `wcstombs()`).

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.

(This is the only function whose behavior 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. As this comparison is not necessarily as inexpensive as `strcmp()`, the `strxfrm()` function can be used to transform a string into another, such that any two such after-translation strings can be passed to `strcmp()` and get an ordering analogous to what `strcoll()` would have returned if passed the two pre-translation strings.

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

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 to the *The C Programming Language* by Kernighan and Ritchie. However, ANSI C does not grant compilers this same freedom.

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 are performed with a particular grouping. For the above expression, the updating of `i` and `p` must occur after the previous statement and by the `;` of this expression statement; the calls to the functions can occur in either order, any time after the previous statement, but before their return values are used. In particular, note that the operators that cause memory to be updated have no requirement 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 above 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, these three groupings will behave differently if one of the additions 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.

The Parentheses are Special Mistake

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 simply 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 shifting) are allowed to be rearranged on any machine because there is simply 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.

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 article describes incomplete types, where they are permitted, and why they are useful.

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, but it is important to know that 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 in that it is an incomplete type that cannot be completed, and 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. (Also, 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.)

An incomplete structure or union type is completed by specifying 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 and 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 declaration of `main`'s `argv` (namely, `char *argv[]`) 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, even though some may find this surprising. For example, given:

```
void *p
```

`&*p` is a valid subexpression that makes use of this.

Justification

C would have been simpler without incomplete types. Why are they necessary? Ignoring `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 one 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:

```
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.

Examples

Defining `typedef` names for incomplete structure and union types is frequently quite useful. If one has a complicated bunch of data structures that contain many pointers to each other, having a list of `typedefs` to the structures up front (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. Other parts of the program can use pointers to the incomplete structure or union without any problems (unless they attempt to use any of its members).

A frequently used incomplete type is an external array of unspecified length. It generally is not necessary to know the extent of an array to make use of 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 *close enough*. 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 probably looks at different source files, most of the rules for compatible types across separate compiles are structural in nature:

- Matching scalar (integral, 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.

  An additional requirement is that the names of members (including the lack of names for unnamed members) 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. More about composite types later.

The 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 (see "Incomplete Types" on page 36) is compatible both with another incomplete array type and an array type with a specified size.

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 (not a definition) to be compatible with a function type with a prototype, the prototype parameters must not end with an ellipsis (. . .) and all of the prototype parameters must have types that would be 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

There are a few surprises in this area. For example, `signed int` behaves the same as `int` except possibly for bit-fields, in which a plain `int` may denote an unsigned-behaving quantity.

Another interesting note is that each enumeration type must be compatible with some integral type. For portable programs this means that enumeration types effectively are separate types, and, for the most part, the ANSI C standard views them in that manner.

Composite Type

The construction of a composite type from two compatible types is also recursively defined. The ways compatible types can differ from each other are due either to incomplete arrays or to old style function types. As such, the simplest description of the composite type is that it is the type compatible with both of the original types, including every available array size and every available parameter list from the original types.

Sun C / Sun ANSI C Differences

A.1 Introduction

In this chapter we describe the differences between the previous (K&R) Sun C and Sun ANSI C, as implemented on SunOS 5.0.

"Sun C Incompatibilities with Sun ANSI C" on page 44 describes previous Sun C features that are incompatible with Sun ANSI C. These differences should be addressed when porting source code written for the Sun C compiler to Sun ANSI C.

"Keywords" on page 51 lists reserved words used by the ANSI C standard, Sun ANSI C, Sun C, and those defined by the Sun ANSI and Sun C preprocessors.

A.2 Sun C Incompatibilities with Sun ANSI C

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 1 of 8)

Topic	Sun C	Sun ANSI C
envp argument to `main()`	Allows `envp` as third argument to `main()`.	Allows this third argument; however, this usage is not strictly conforming to the ANSI C standard.
keywords	Treats the identifiers `const`, `volatile`, and `signed` as ordinary identifiers.	`const`, `volatile`, and `signed` are keywords.
`extern` and `static` functions declarations inside a block	Sun C promotes these function declarations to file scope.	The ANSI standard does not guarantee that block scope function declarations are promoted to file scope.
identifiers	Allows dollar signs (`$`) in identifiers.	`$` not allowed.
`long float` types	Accepts `long float` declarations and treats these as `double`(s).	Does not accept these declarations.

multi-byte character constants	`int mc = 'abcd';`
	
	yields
	
	`abcd`
		`int mc = 'abcd';`
		
		yields
		
		`dcba`

integer constants	Accepts 8 or 9 in octal escape sequences.	Does not accept 8 or 9 in octal escape sequences.
assignment operators	Treats the following operator pairs as two tokens, and as a consequence, permits whitespace between them: `*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	Treats them as single tokens, and therefore disallows whitespace in-between.
unsigned preserving semantics for expressions	Supports *unsigned* preserving. That is, `unsigned char/shorts` are converted into `unsigned int`(s).	Supports *value-preserving*. That is, `unsigned` char/short(s) are converted into `int`(s).

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 2 of 8)

Topic	Sun C	Sun ANSI C
single / double precision calculations	Promotes the operands of floating point expressions to `double`. In Sun C, functions which are declared to return `floats` always promote their return values to `doubles`.	Allows operations on `floats` to be performed in single precision calculations. Allows `float` return types for these functions.
name spaces of `struct/union` members	Allows struct, union, and arithmetic types using member selection operators ('.', '->') to work on members of other struct(s) or unions.	Requires that every unique struct/union have its own unique namespace.
a *cast* as an lvalue	Supports casts as lvalue(s). For example: `(char *)ip = &char;`	Does not support this feature.

implied `int` declarations	Supports declarations without an explicit type specifier. A declaration such as `num;` is treated as implied `int`. For example:
	
	`num;  /* num implied as an int */`
	
	`int num2;  /* num2 explicitly declared an int */`
		The `num;` declaration (without the explicit type specifier `int`) is not supported, and generates a syntax error.

empty declarations	Allows empty declarations: `int;`	Except for *tags*, disallows empty declarations.

type specifiers on type definitions	Allows type specifiers such as `unsigned, short, long` on typedefs declarations. For example:
	
	`typedef short small;`
	`unsigned small x;`
		Does not allow type specifiers to modify `typedef` declarations.

types allowed on bitfields	Allows bitfields of all integral types, including unnamed bit fields. **NOTE:** The ABI requires support of unnamed bit fields and the other integral types, as provided in Sun C.	Supports bitfields only of the type `int`, `unsigned int` and `signed int`. Other types are undefined.

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 3 of 8)

Topic	Sun C	Sun ANSI C

treatment of tags in incomplete declarations	Since Sun C ignores the incomplete type declaration, in the below example `f1` refers to the outer struct:
	`struct x { . . . } s1;`
	`{`
	`  struct x;`
	`  struct y {struct x f1; } s2;`
	`  struct x { . . . };`
	`}`
		In an ANSI conforming implementation, an incomplete `struct` or `union` type specifier hides an enclosing declaration with the same tag.

mis-match on `struct/union /enum` declarations	Allows a mismatch on the `struct/enum/union` type of a tag in nested `struct/union` declarations. In the example below the second declaration is treated as a `struct`:
	`struct x { . . . } s1;`
	`{`
	`  union x s2;`
	`  . . .`
	`}`
		Will treat the inner declaration as a new declaration, hiding the outer tag.

labels in expressions	Treats labels as `(void *)` lvalues.	Does not allow labels in expressions.
`switch` condition type	Allows `float`(s) and `double`(s) by converting them to `int`(s).	Evaluates only integral types (`int, char`, and enumerated) for the switch condition type.
syntax of conditional inclusion directives	The Sun C preprocessor ignores trailing tokens after an `#else` or `#endif` directive.	Disallows such constructs.

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 4 of 8)

Topic	Sun C	Sun ANSI C
token pasting and the ## preprocessor operator	Doesn't recognize the ## operator. In Sun C, token pasting is accomplished by placing a comment between the two tokens being pasted: `#define PASTE(A,B)A/*any comment*/B`	Defines ## as the preprocessor operator that performs token pasting, as shown in the example below. `#define PASTE(A,B)A##B` Furthermore, the Sun ANSI C preprocessor doesn't recognize the Sun C method. Instead, it treats the comment between the two tokens as whitespace.

preprocessor rescanning	The Sun C preprocessor will recursively substitute:
	
	`#define F(X) X(arg)`
	`F(F)`
	
	yields
	
	`arg(arg)`
		A macro will not be replaced if it is found in the replacement list during the rescan:
		
		`#define F(X) X(arg)`
		`F(F)`
		
		yields:
		
		`F ( arg )`

typedef names in formal parameter lists	Allows one to use typedef names as formal parameter names in a function declaration, and would, in effec, "hide" the typedef declaration.	Disallows the use of an identifier declared as a typedef name as a formal parameter.

implementation specific initializations of aggregates	Uses a bottom-up algorithm when parsing and processing partially elided initializers within braces:
	
	`struct { int a[3]; int b; } \`
	`  w[] = { {1}, 2};`
	
	yields
	
	`sizeof(w) = 16`
	`w[0].a = 1, 0, 0`
	`w[0].b = 2`
		Uses a top-down parsing algorithm. For example:
		
		`struct { int a[3]; int b; } \`
		`  w[] = { {1}, 2};`
		
		yields
		
		`sizeof(w) = 32`
		`w[0].a = 1, 0, 0`
		`w[0].b = 0`
		`w[1].a = 2, 0, 0`
		`w[1].b = 0`

comments spanning include files	Allows comments which start in an `#include` file to be terminated by the file that includes the first file.	Comments are replaced by a white-space character in the translation phase of the compilation, which occurs before the `#include` directive is processed.

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 5 of 8)

Topic	Sun C	Sun ANSI C

formal parameter substitution within a character constant	Substitutes characters within a character constant when it matches the replacement list macro:
	
	`#define charize(c) 'c'`
	`charize(Z)`
	
	yields
	
	`'Z'`
		The character is not replaced:
		
		`#define charize(c) 'c'`
		`charize(Z)`
		
		yields
		
		`'c'`

formal parameter substitution within a string constant	The Sun C preprocessor will substitute a formal parameter when enclosed within a string constant:
	
	`#define stringize(str) 'str'`
	`stringize(foo)`
	
	yields
	
	`"foo"`
		For string substitution in Sun ANSI C, the # preprocessor operator should be used.
		
		`#define stringize(str) 'str'`
		`stringize(foo)`
		
		yields
		
		`"str"`

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 6 of 8)

Topic	Sun C	Sun ANSI C

preprocessor built into the compiler "front-end"	Compiler calls `cpp`(1).	Preprocessor (`cpp`) is built directly into `acomp`, so `cpp` is not directly involved (except in `-Xs`) mode.
		
		Following are the components used in compiling:
		
		Sun C / Sun ANSI C
		
		`cpp` — `cpp (-Xs mode only)`
		`ccom` — `acomp`
		— `basicblk` (only with `-ql`)
		`iropt` — `iropt`
		`cg` — `cg`
		`inline` — `inline`
		`as` — `fbe`
		`ld` — `ld`
		
		Note: `iropt` and `cg` will be invoked only with the following options:
		
		`-O -xO2 -xO3 -xO4 -xa -fast`
		
		`inline` will be invoked only if an inline template file (*file*`.il`) is provided

line concatenation with backslash	Does not recognize the backslash character in this context.	Requires that a new-line character immediately preceded by a backslash character be spliced together.
trigraphs in string literals	Does not support this ANSI C feature.	
`asm` keyword	`asm` is a keyword.	`asm` is treated as an ordinary identifier.

linkage of identifiers	Does not treat uninitialized `static` declarations as tentative declarations. As a consequence, the second declaration (in the example below) will generate a 'redeclaration' error:
	
	`static int i = 1;`
	
	`static int i;`
		Treats uninitialized `static` declarations as tentative declarations.

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 7 of 8)

Topic	Sun C	Sun ANSI C
name spaces	Distinguishes only three : `struct/union/enum` tags, members of `struct/union/enum`, and everything else.	Recognizes four distinct name spaces: label names, tags (the names that follow the keywords `struct, union` or `enum`), members (of `struct/union/enum`), and ordinary identifiers.
`long double` type	Not supported.	Allows `long double` type declaration.
floating point constants	The floating point suffixes, `f`, `l`, `F`, and `L`, are not supported.	
unsuffixed integer constants can have different types	The integer constant suffixes `u and U` are not supported.	
wide character constants	Does not accept the ANSI C syntax for wide character constants, as shown below: `wchar_t wc = L'x';`	Supports this syntax.
`'\a'` and `'\x'`	Treats them as the characters 'a' and 'x'.	Treats `'\a'` and `'\x'` as special escape sequences.
Concatenation of string literals	The ANSI C concatenation of adjacent string literals is not supported in Sun C.	
Wide-character string literal syntax	The ANSI C wide-character-string literal syntax shown in the example below is not supported: `wchar_t *ws = L"hello";`	Supports this syntax.
pointers: `void *` *vs.* `char *`	The ANSI C `void *` feature **is supported**.	
unary plus operator	This ANSI C feature **is not supported**.	
function prototypes - ellipses	Not supported.	ANSI C defines the use of ellipses "..." to denote a variable argument parameter list.

Table A-1 Sun C Incompatibilities with Sun ANSI C (Sheet 8 of 8)

Topic	Sun C	Sun ANSI C
type definitions	Disallows typedefs to be redeclared in an inner block by another declaration with the same type name.	Allows typedefs to be redeclared in an inner block by another declaration with the same type name.
initialization of extern variables	Does not support the initialization of variables explicitly declared as `extern`.	ANSI C treats the initialization of variables explicitly declared as `extern`, as definitions.
initialization of aggregates	Does not support the ANSI C initialization of unions or automatic structures.	
prototypes	This ANSI C feature (of prototyping) is not supported.	
syntax of preprocessing directive	Recognizes only those directives with a '#' in the first column.	ANSI C allows leading whitespace characters before a '#' directive.
the # preprocessor operator	The ANSI C # preprocessor operator is not supported.	
`#error` directive	This ANSI C feature is not supported by the preprocessor.	
preprocessor directives	Supports two pragmas, `unknown_control_flow` and `makes_regs_inconsistent` along with the #ident directive. Moreover, the preprocessor will issue warnings when it finds unrecognized pragmas.	ANSI C does not specify its behavior for unrecognized pragmas. (See the *C 2.0.1 Programmer's Guide* for recognized pragmas.)

predefined macro names	The ANSI C-defined macro names shown below are not defined in Sun C.
	
	`__STDC__`
	`__TIME__`
	`__DATE__`
	`__LINE__`
		Defined.

A.3 Keywords

The four tables below list the keywords for the ANSI C Standard, the Sun ANSI C compiler, the Sun C compiler, and the preprocessor keywords that are

therefore reserved words when using either compiler.

The first table lists the keywords defined by the ANSI C standard.

Table A-2 ANSI C Standard Keywords

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

Sun ANSI defines one additional keyword, `asm`. However, `asm` is not supported in `-Xc` mode.

Keywords in Sun C are listed below.

Table A-3 Sun C (K&R) Keywords

`asm`	`auto`	`break`	`case`
`char`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`fortran`	`goto`
`if`	`int`	`long`	`register`
`return`	`short`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`while`	

The following are predefined macros defined by the preprocessor for both Sun C and Sun ANSI C. (The `-Xc` mode of the Sun ANSI C compiler does not define these names.)

Table A-4 Preprocessor-Defined Keywords

`sparc`	`sun`
`sun4`	`unix`

The `-sys5` option predefines __SYS5__.

Comparison of `cc` *Options*

The following table shows a comparison between `cc` options supported in Sun C on SunOS 4.1, in Sun ANSI C on SunOS 4.1 and in ANSI C on SunOS/SVR4.

Please note the following:

- A *yes* in any column indicates that the option is supported by that driver.
- A *no* is shown in the column if the option is not supported by that driver.
- If the option has changed, the new functionality is listed.
- A + in the ANSI C SunOS/SVR4 column indicates an option required by SVID specifications.

Table B-1 Comparison of `cc` Options (Sheet 1 of 5)

Option or Flag	Sun C (SunOS 4.x)	ANSI C (SunOS 4.x)	ANSI C (SunOS 5.0)	Description
`-#`	`-v`	`-v`	yes	Verbose mode
`-###`	`-dryrun`	`-dryrun`	yes	Show compiler steps but don't execute
`-A`	no	yes	yes	Preprocessor predicate assertion
`-a`	yes	yes	`-xa`	Count # basic block executions
`-align`	yes	yes	no	Page align (`ld`(1))
`-B`*binding_option*	yes	yes	yes	Specify binding type (dynamic, static)
`-C`	yes	yes	+yes	Preprocessor comments left in
`-c`	yes	yes	+yes	Produce `.o` file
`-cg87`	yes	yes	no	Sets `fp` option to `-cg87`
`-cg89`	yes	yes	no	Sets `fp` option to `-cg89`
`-D`*symbol*`[=`*def*`]`	yes	yes	yes	Define *symbol* to *def*
`-d[y|n]`	`-B`*x*	`-B`*x*	yes	Dynamic linking[yes | no]
`-dalign`	yes	yes	yes	Assume doubles are doubleword aligned
`-dryrun`	yes	yes	`-###`	Show commands constructed by driver
`-E`	yes	yes	+yes	Run source through preprocessor
`-F`*option*	no	no	yes	Reserved for floating point
`-f`*option*	yes	no	no	Floating-point generation code
`-fast`	yes	yes	yes	Options for best performance
`-flags`	`-help`	`-help`	yes	Print available options
`-fsingle`	yes	yes	yes	Float are single precision
`-fsingle2`	yes	no	no	Pass float as float not double
`-fnonstd`	yes	yes	yes	Non-standard float option
`-fstore`	yes	no	no	Force writes on store
`-G`	no	no	yes	`-dy`, but no `crt1.o` is linked
`-g`	yes	yes	+yes	Generate info for `dbx`
`-go`	yes	no	no	Generate info for `adb`
`-H`	yes	yes	yes	Print paths of included files

Table B-1 Comparison of `cc` Options (Sheet 2 of 5)

Option or Flag	Sun C (SunOS 4.x)	ANSI C (SunOS 4.x)	ANSI C (SunOS 5.0)	Description
`-h`	no	no	yes	Name a shared dynamic library
`-help`	yes	yes	`-flags`	Lists options
`-I`*x*	yes	yes	+yes	Add *x* to include path
`-i`	no	no	yes	Ignore `LD_LIBRARY_PATH` setting
`-J`	yes	no	no	Generate long offset for `switch|case`
`-KPIC`	`-PIC`	`-PIC`	yes	Position independent code
`-Kpic`	`-pic`	`-pic`	yes	`PIC` with short offsets
`-keeptmp`	no	yes	yes	Retain temporary files
`-libmil`	yes	yes	`-xlibmil`	Pass `libm.il` as part of `-fast`
`-l`*x*	yes	yes	yes	Read object library (for `ld`)
`-L`*x*	yes	yes	yes	Add *x* to `ld` library path
`-M`	yes	yes	`-xM`	Collect dependencies (calls preprocessor)
`-misalign`	yes	yes	yes	Handle misaligned Sun-4 data
`-native`	yes	yes	no	Use appropriate `-cg` option
`-nolibmil`	yes	yes	`-xnolibmil`	Don't pass `libm.il` with `-fast`
`-noqueue`	no	no	yes	Don't queue license requests
`-o` *file*	yes	yes	+yes	Set name of output file
`-O[1,2,3,4]`	yes	yes	`-xO[1,2,3,4]`	Generate optimized code
`-O`	yes	yes	+yes	Generate optimized code
`-P`	yes	yes	+yes	Run source thru preprocessor, output to `.i`
`-PIC`	yes	yes	`-KPIC`	Generate `pic` code with long offset
`-p`	yes	yes	+yes	Collect data for `prof`
`-pic`	yes	yes	`-Kpic`	`pic` code with short offset
`-pipe`	yes	no	no	Use pipes instead of temp files
`-pg`	yes	yes	`-xpg`	Collect data for `gprof`
`-Qdir` *x*	yes	yes	`-Yc` *, dir*	Look for compiler passes in *x*
`-Qoption cpp` *x*	yes	yes	use -W option	Pass option *x* on to program `cpp`

Table B-1 Comparison of `cc` Options (Sheet 3 of 5)

Option or Flag	Sun C (SunOS 4.x)	ANSI C (SunOS 4.x)	ANSI C (SunOS 5.0)	Description
`-Qoption acomp` *x*	yes	yes	use -W option	Pass option *x* on to program `acomp`
`-Qoption iropt` *x*	yes	yes	use -W option	Pass option *x* on to program `iropt`
`-Qoption cg` *x*	yes	yes	use -W option	Pass option *x* on to program `cg`
`-Qoption inline` *x*	yes	yes	use -W option	Pass option *x* on to program `inline`
`-Qoption as` *x*	yes	yes	use -W option	Pass option *x* on to program `as`
`-Qoption ld` *x*	yes	yes	use -W option	Pass option *x* on to program `ld`
`-Qpath` *x*	yes	yes	`-Yc`, *dir*	Same as `Qdir`
`-Qproduce .o`	yes	yes	no	Produce type `.o` file (Object file)
`-Qproduce .s`	yes	yes	no	Produce type `.s` file (Assembler source)
`-Qproduce .c`	yes	yes	no	Produce type `.c` file (C source)
`-Qproduce .i`	yes	yes	no	Produce `.i` file (source after preprocessor)
`-Q[y|n]`	no	no	yes	Add, don't add version stamp info
`-qdir` *x*	yes	yes	`-Yc`, *dir*	Look for compiler passes in *x*
`-ql`	no	no	+yes	Collect data for `lprof`
`-qoption cpp` *x*	yes	yes	use -W option	Pass option *x* on to program preprocessor
`-qoption acomp` *x*	yes	yes	use -W option	Pass option *x* on to program `acomp`
`-qoption iropt` *x*	yes	yes	use -W option	Pass option *x* on to program `iropt`
`-qoption cg` *x*	yes	yes	use -W option	Pass option *x* on to program `cg`
`-qoption inline` *x*	yes	yes	use -W option	Pass option *x* on to program `inline`
`-qoption as` *x*	yes	yes	use -W option	Pass option *x* on to program `as`
`-qoption ld` *x*	yes	yes	use -W option	Pass option *x* on to program `ld`
`-qp`	`-p`	`-p`	+yes	Collect data for `prof`
`-qpath` *x*	yes	yes	-Yc, *dir*	Same as `Qdir`
`-qproduce .o`	yes	yes	no	Produce type `.o` file (Object file)
`-qproduce .s`	yes	yes	no	Produce type `.s` file (Assembler source)
`-qproduce .c`	yes	yes	no	Produce type `.c` file (C source)
`-qproduce .i`	yes	yes	no	Produce `.i` file (source after preprocessor)

Table B-1 Comparison of cc Options (Sheet 4 of 5)

Option or Flag	Sun C (SunOS 4.x)	ANSI C (SunOS 4.x)	ANSI C (SunOS 5.0)	Description
-R	yes	yes	no	Merge data into text segment
-R*dir*[:*dir*]	no	no	yes	Specify library search directories for ld
-S	yes	yes	+yes	Product .s file only
-s	yes	yes	yes	strip (4.1); pass to ld (5.0)
-sb	yes	yes	-xsb	Collect info for code browser
-sbfast	no	yes	-xsbfast	Collect info for code browser, no compile
-strconst	no	yes	-xstrconst	Insert string literals in text segment
-temp=dir	yes	yes	no[a]	Set directory for temps to <dir>
-time	yes	yes	no	Report the execution times
-U*x*	yes	yes	+yes	Undefine preprocessor symbol *x*
-v	yes	yes	-#	Verbose mode
-vc	no	yes	-v	Stricter semantic checking
-V	yes	yes	+yes	Report versions of programs
-W	no	no	+yes	arguments to other components
-w	yes	yes	yes	Do not print warnings
-X[s]	no	yes	yes	Sun C compatibility option
-X[t,a,c]	no	yes	yes	Compatibility options
-xa	-a	-a	yes	Collect data for tcov
-xF	no	no	yes	Produce reorder-able code
-xlibmil	-libmil	-libmil	yes	Use inline templates (used with -fast)
-xlicinfo	no	yes	yes	Return license status information
-xM	-M	-M	yes	Collect makefile dependencies
-xnolibmil	-nolibmil	-nolibmil	yes	Don't use inline templates
-xO[1,2,3,4]	-O[1-4]	-O[1-4]	yes	Generate optimized code
-xpg	-pg	-pg	yes	Collect data fro gprof
-xs	no	no	yes	Put all stabs in .stabs section
-xsb	-sb	-sb	yes	Collect SourceBrowser info

Table B-1 Comparison of `cc` Options (Sheet 5 of 5)

Option or Flag	Sun C (SunOS 4.x)	ANSI C (SunOS 4.x)	ANSI C (SunOS 5.0)	Description
`-xsbfast`	no	-sbfast	yes	Same as `-xsb` but no compilation
`-Yc,` *dir*	no	no	+yes	Change pathname to components
`-YI,` *dir*	no	no	yes	Change search directory for include files
`-YP,` *dir*	no	no	yes	Change default directory for library files
`-YS,` *dir*	no	no	yes	Change default directory for start up files

a. Replaced by the environment variable `TMPDIR`.

Table B-2 File Suffixes

Suffix `.a`	Object library
Suffix `.il`	Inline expansion file
Suffix `.o`	Object file
Suffix `.so`	Shared object
Suffix `.s`	Assembler source
Suffix `.S`	Assembler source for preprocessor
Suffix `.c`	C source
Suffix `.i`	C source after preprocessor

-Xs Differences for Sun C and ANSI C

C.1 Introduction

In this appendix we describe the differences in compiler behavior when using the -Xs option. The -Xs option tries to emulate /bin/cc, Sun C 1.0, Sun C 1.1 (K&R style), but in some cases the emulation fails.

Table C-1 -Xs Behavior (Sheet 1 of 2)

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)`

Table C-1 -Xs Behavior (Sheet 2 of 2)

data type	Sun C (K&R)	Sun ANSI C

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

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).

Index

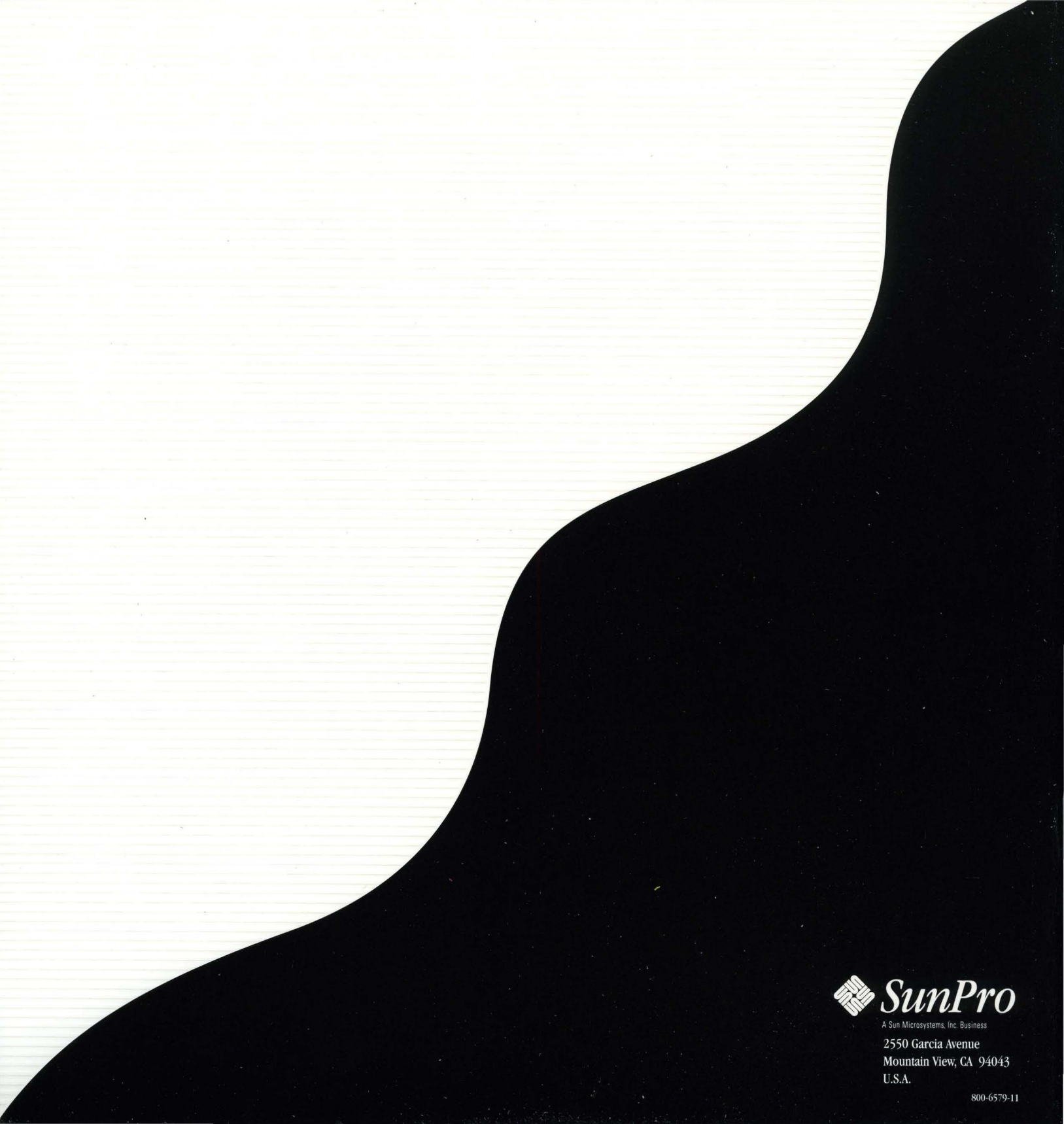
S

T

V

W

X



SunPro
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

800-6579-11