Using the New Features in ANSI C
Mark Gass, SAS Institute Inc., Cary, NC

ABSTRACT
The C programming language has evolved since its original definition by Kernighan and Ritchie (1978). The proposed ANSI C Language Standard incorporates many new features and defines the language with greater precision. This paper describes and demonstrates how to use the most significant new features. It also explains ANSI terminology so that advanced programmers can study the proposed Standard.

INTRODUCTION
There have been many changes in the C programming language since it was first described by Kernighan and Ritchie (1978). The original language had many strengths including efficiency, portability, and flexibility, but improvements were still possible. Bell Labs’ UNIX® operating system compilers pioneered many of the enhancements. In 1983, the American National Standards Institute formed the X3J11 Technical Committee on the C Programming Language with the goal of creating a standard for the C language, officially known as the Draft Proposed American National Standard for Information Systems - Programming Language C.

This committee has defined the language with greater precision, standardized the Bell Labs enhancements, and introduced additional useful features. The ANSI Committee is also drafting a Rationale document, which explains some of their decisions but is not intended to be part of the official Standard.

The new features offer many benefits. C compilers can now catch more programmer errors and generate more efficient code. The language itself can be written more portably and with less coding effort. This tutorial demonstrates how to take advantage of many of the improvements now available over the original Kernighan and Ritchie description of the C programming language. It also discusses the terminology and organization of the Draft Standard so that advanced programmers can study it and evaluate competing vendor’s conformance claims.

What follows attempts to cover most of the important changes, but it does not attempt to be totally comprehensive. Many of these features may already be present in your compiler, but few or no compilers implement all of them.

SOME ANSI TERMINOLOGY
Becoming familiar with the ANSI Standard’s terminology is a first step for programmers who would like to understand ANSI C in detail. Besides appearing in the Standard and Rationale, ANSI terminology is also likely to be used in vendors’ documentation and advertising.

In ANSI terminology, any particular compiler and library product is called an implementation. Thus, the SAS/C™ compiler for IBM® mainframes is one implementation, while the Lattice® C compiler for MS-DOS® is another. Furthermore, every distinct combination of compiler options can be thought of as constituting a different implementation.

Valid C programs fall into two categories: strictly conforming and conforming. A strictly conforming program is one that uses only features prescribed by the Standard. It uses no machine- or vendor-specific extensions and does not exceed the Standard’s minimum requirements in any area. A strictly conforming program would not, for example, be dependent on an underlying operating system’s filename format or assume that the value of a plain int could exceed 32k without overflowing (it generally cannot on 16-bit machines) or use more than 509 characters in a string literal (which is the most that the Standard requires of an implementation). Put simply, a strictly conforming program is one that is fully portable.

A conforming program is one that is not necessarily portable. It can exceed some limit, have some operating system dependencies, or use some compiler’s language extensions. Of course, in all other respects it must be a valid C program (the test of which is that it is accepted by some implementation). MS-DOS programs that use the near keyword or SAS/C programs on MVS/IXA that allocate objects of 100 megabytes (the maximum required by the Standard is only 32k) are conforming programs that are not strictly conforming.

A conforming implementation is a compiler and library that accepts and translates correctly any strictly conforming program. A conforming implementation is what is informally called an ANSI C compiler and library. Thus, the programmer knows how to write code that will work on any conforming implementation and is able to use the compiler vendor’s extensions, if the benefits that they provide are more important than absolute portability.

Certain aspects of the compiler’s behavior are implementation-defined. The format of filenames, the range of values that can be accommodated by a double, and whether or not case is significant in external identifiers are all examples of implementation-defined behavior. The compiler vendor is given latitude in selecting these behaviors but is required to document the behavior chosen.

Some types of behavior, such as the order of expression evaluation or the representation of floating-point types, are termed unspecified. Unspecified behavior applies only to valid constructs. Undefined behavior, on the other hand, applies to invalid constructs or erroneous data. Overflow handling is a good example of this. The Standard does not require the implementation to handle overflow in any particular way. One conforming compiler may always return a defined result from overflow and guarantee that an overflow signal will be raised, while another could produce garbage results or even terminate the program with no message or opportunity for recovery. Unlike implementation-defined behavior, the implementation is not required to document unspecified or undefined behavior. A strictly conforming program cannot rely on any of these behaviors. No program should ever rely on unspecified or undefined behavior unless its intended compiler documents that a certain type of behavior is guaranteed.

Besides translating valid C programs, a conforming implementation must also diagnose syntactic and semantic problems in incorrect programs. When it does notice such a problem, the compiler must issue a diagnostic, or some type of message. Besides saying that the program has not been accepted, there are no other requirements. Thus, buying an ANSI compiler does not assure you of having helpful error messages. The Standard also lists a set of common warnings (for example, that a state-
ment is unreachable), which it suggests that good compilers will issue, but these are not required.

The term constraint refers to an aspect of an incorrect program that the compiler must diagnose. For example, it is a constraint that a function cannot return an array. Thus, the compiler must issue a diagnostic when it sees this error.

**LEXICAL FEATURES**

**New Keywords**

Despite a desire on the committee's part to minimize the number of new keywords, the following were added:

- enum
- signed
- void
- const
- volatile
- noalias

Since keywords are reserved words, you will have to change any programs that use them as identifier names. Their meanings are described later in this paper.

**Longer Identifiers**

ANSI C adds flexibility in the use of identifiers. Most identifiers can now be of any length, with the first thirty-one characters being significant. Unfortunately, names that are visible to the linker (identifiers declared with the keyword extern or outside of any function without the keyword static) are an exception. An implementation is only required to distinguish them to six characters and need not differentiate uppercase and lowercase. If you do not anticipate running on an operating system that has such a short maximum length for external identifiers, you can assume that eight (case-insensitive) characters will probably provide sufficient portability to any mainstream operating system. (However, this prevents your program from being strictly conforming.)

The ANSI Standard also states that identifiers beginning with two underscores or one underscore and a lowercase letter are reserved for the implementation (that is, the compiler vendor). It also requires that all external identifiers beginning with an underscore (whether or not the next character is an uppercase or lowercase letter) be reserved. Thus, your program should never use such an identifier unless you are doing so in a manner directed by your implementation. For example, when using the SAS/C library, you can set the variable _style to control whether, by default, filenames are assumed to specify DDnames or data set names.

**New Name Spaces**

A much needed enhancement to Kernighan and Ritchie is the creation of the following separate namespaces for identifiers:

- structure, union, and enumeration tags (enumerations are described below)
- structure and union member names (one namespace for each type of structure or union)
- label names
- ordinary identifiers, typedef names, and enumeration constants.

Separate name spaces eliminate much arbitrary twisting of identifier names in order to ensure uniqueness. For example,

```c
#include <time.h> /* Need for time_t definition. */
struct bird_sighting {
    time_t when;
    unsigned type;
    unsigned number_seen;
};
struct nest_sighting {
    time_t when;
    unsigned type;
    unsigned number_seen;
    unsigned total_eggs;
};
typedef struct nest_sighting nest_sighting;
```

The member names in the two structure objects can be the same because they are in different types of structures. Similarly, the typedef name nest_sighting can have the same name as a structure tag.

**Adjacent String Concatenation**

Lexically adjacent string literals are now accepted. The compiler is required to concatenate them into a single string literal. This is particularly useful for long strings, for example,

```c
char*mem_title="Welcome to the Integrated Ornithology System"
"nt"
"Please choose one of the following options:"
```

**New Escape Sequences**

The ANSI Standard provides for new escape sequences in character constants and string literals. You can specify hexadecimal values. The annoying requirement to specify unprintable characters in octal is thus eliminated. In the following example,

```c
#define CLEAR_SCREEN "\x1b\23"
#define ESCAPEDA "\x1b A" /* Not \x1bA */
```

characters following the x are used to form a hex value that is then assigned to the next char in the string. The hex value is formed out of all successive hex digit characters (0-9,A-F,a-f). Thus, on the second line of the example above, the A (which is not to be taken as a hex digit) must be separated into a different string literal to avoid the first character in the string being interpreted as hex 01BA. This uses the adjacent string concatenation feature described above. Most compilers that accept hex escape sequences do not follow these rules. This is another of many features that have changed in successive versions of the Standard.

The `\a` (alert) character escape sequence has been added. As with all the special characters, its effect is implementation-defined, but it is intended to be equivalent to the ASCII BELL character.

**DECLARATIONS**

ANSI C reflects many enhancements to the declarations described in Kernighan and Ritchie. For example, there are new data types, type qualifiers that can modify existing types, and totally reworked function declarations and definitions.

**New Ways to Declare Arithmetic Types**

You can use the keyword signed with any of the integral types (including bit fields). Its use is fully analogous to the current use of unsigned. For the most part, this just allows you to be explicit in declaring the type of an integral object (signed int is really no different from int), but it does introduce signed bit fields as a new type and requires compilers that previously supported only unsigned characters to support signed characters as well.
A new floating data type called long double has been introduced to allow greater precision on machines such as the IBM 370 that have three floating point formats. Putting the letter L after a floating constant (as in 3.14159L) causes it to be assigned a type of long double.

The addition of this data type is the most significant of several enhancements that the committee has made with a goal of making C a better language for mathematical programming. Note that while conforming implementations are required to support the syntax, they may treat the long double object as having exactly the same precision and range as a double (not all machines have three floating point formats). The following examples show the use of the signed keyword and long double.

```c
signed long ytd_seagull_population_increase;
/* as with the unsigned keyword, int is implied here. */
signed md.seagull_population_increase;

struct single.seas.ag.counts {
    unsigned initial_count; 4;
    signed int one_year_change; 4;
};

long double very_precise_number = 1.0L; /* Note use of l suffix. */
```

### New Data Types with Void

The new keyword void allows for more new data types. The void type by itself is used to indicate that an expression (most often a function call) has no value and that it is being used for its side effects only.

The type void * (pointer to void) specifies a generic pointer. A pointer to void is one that cannot be construed to point to any specific type of data object. This is particularly useful for passing parameters to and returning pointer values from functions such as malloc() that are not specific to any type of object.

Prior to the introduction of void * , char * was used as the generic pointer type, but the ambiguities that this caused were an impediment to good type checking. It is reasonable to expect that compilers supporting void * will issue no errors or warnings when converting any data pointer to or from pointer to void. Furthermore, you can make your desire to use a generic pointer obvious.

There is no confusion as to whether you actually intend to use the pointer as generic or as a pointer to a character or string. A review of the library function synopses in the standard shows that many functions that previously accepted or returned char * now use void * instead. The following are some examples of the use of void:

```c
#include <time.h> /* Need for time_t definition. */
#include <stdio.h> /* Defines NULL. */
#include <stdlib.h>

/* Function newpage does not accept parameters or return a value. */
void newpage(void) {
    putchar('');
}

struct bird.sighting {
    time_t when;
    unsigned type;
    unsigned number_seen;
};

#define BIRD_SIGHTING_RECORD 20

#define HL_SIZE 1000
/* An array of pointers to different types of things. */
void *heterogeneous_list[HL_SIZE];
short heterogeneous_list_type[HL_SIZE];
/* Define a function which will find the first bird.sighting */

#include <time.h> /* Need for time_t definition. */
enum bird.type {
    robin, turkey, eagle, scarlet_lbird
};

/* Bird sighting record using an enum. */
struct bird.sighting {
    time_t when;
    enum bird.type type;
    unsigned number_seen;
};

/* This function notes that the bird seen was an eagle. */
void saw.eagle(ps) {
    ps->type=eagle;
    last.bird.seen=eagle;
}
```

In the example above, the identifier eagle is an enumeration constant and the structure field type is a bird..type enumeration. Note that unlike a structure or union member, an enumeration constant can be used alone. In the example above, eagle is simply treated by the compiler as another name for the constant 2 (robin is 0, turkey is 1, and so on).

You can also assign explicit values to enumeration constants. The declaration

```c
enum bird.type {
    robin=10, turkey=20, eagle=30, scarlet_lbird=40
};
```

assigns values to each of the constants.

Strictly speaking, there is no requirement on what integral value is assigned to an enumeration. You can assign to it using an arbitrary expression. You can even assign to it using an enumeration constant from a different enumeration. However, you are likely to get a warning message from your compiler if you assign (or compare) to an enumeration using an expression that is neither an enumeration of the same type nor one of the enumeration's constants. If you do assign arbitrary values to the enumeration, you must also be careful about assigning ones that are too large to fit in the type selected by the compiler.
In the past, the usual way to handle a set of named integral values was by using #define to assign names to constant values. Enumerations are a good alternative to this because they allow better type checking (which can catch some of your errors at compile-time). They are also better because they avoid the context-independent substitution of text performed by the preprocessor.

The exact size of the enumeration (char, short or long int) and whether it is unsigned or signed is implementation-defined. The type checking (which can catch some of your errors at compile-time) is also better because they avoid the context-independent substitution of text performed by the preprocessor.

Enumerations have their shortcomings. There are no successor or predecessor functions that, given one enumeration value, will return the next or prior one. Furthermore, unlike structure members, an enumeration’s constants do not have their own name space. This means that when you use an identifier as an enumeration constant, that identifier cannot be used for another plain identifier, enumeration constant, or typedef name in the same scope.

Qualifying Types with Const, Volatile, and Noalias Keywords

The ANSI standard introduces the concept of qualified types. Three new keywords can be used to add information to the type in a declaration.

Use const in a declaration to indicate that you do not want to modify the declared object. You can also use it with a pointer to indicate that the pointer will not be used to modify an object.

Use volatile to indicate that the value of an object may change in a way that is independent of the function’s generated code. Objects that correspond to memory-mapped machine status registers should be declared volatile, as should ones that may be affected by a signal handling routine.

The noalias keyword is now available to indicate that an object will only be accessed in one way. For example, if x is an integer that is modified by assignments to the identifier y, but never through a pointer, then it is modified in only one way. This is often something that the compiler cannot determine itself, and which, once determined, allows it to generate more efficient code. In particular, you can often expect that a value declared with noalias can be kept in a register longer than otherwise possible. This may also allow global optimizers to find optimizations (such as global common subexpression elimination) more often than without it.

See Aliasing later in this paper for more details.

Type qualifiers are a new syntactic category. Their placement is not the same as storage classes (such as static) or type specifiers (such as char). The following is an example using a type qualifier:

```
#include time.h
// Needed for time_t definition. */

/* Here are some simple objects with type qualifiers. */
volatile time_t *bird_sensor_time; // Updated by an asynchronous signal
noalias nest_type;

/* Here is a more complicated example using pointers. */
float eggs_per_nest;
const float nest_check_ptr[4][eggs_per_nest];
float * const eggs_per_nest_ptr[4][eggs_per_nest];
```

In the example above, eggs_per_nest can be changed by the program directly, but it will not do so using nest_check_ptr. The value of eggs_per_nest can also be changed using eggs_per_nest_ptr. The positioning of const keyword in the eggs_per_nest_ptr declaration indicates that the actual value of the pointer will not change. This is different from the previous declaration, where the positioning of const indicated that the pointer would not be used to change the value of the pointed-to object. The application of volatile and noalias to pointers is analogous.

Better Ways to Declare and Define Functions

In the past, providing an argument of the wrong type to a function has been one of the greatest sources of errors in C programs. A utility such as lint can be used to detect this. But not every environment has such a utility; it is also expensive to run and cannot check calls to non-C functions.

In addition, the traditional function definition syntax also needed improvement since it required repeating the identifier both in the parentheses following the function name and in the actual declaration list. Besides being unnecessarily verbose, this allowed confusion over the order of parameters and separated the type information from the parameter list. There was also the aesthetic problem that function declarations and definitions had dissimilar forms.

ANSI C overcomes the first problem by allowing the types of the parameters to be specified in a function declaration. It overcomes the problems with function definitions by describing a new syntax. The result is that the syntax for function declarations and definitions is now identical. Both specify the function name and return type as before. The difference is that full declarations of the parameters are accepted in the parentheses following the function name. The new form of declaration is called a prototype.

The following is an example of the improved methods of declaring and defining functions.

```
enum color { brown, white, scarlet, grey, blue, green
};
/* Here is the declaration of a function to find the */
/* wing color of a given type of bird. */
void find_wing_color(enum bird_type type, enum color *wing_color);

struct bird_colors {
    enum color head, back, wings, tail, breast;
};
/* An array of pointers to bird_colors records. */
/* indexed by bird_type */
extern struct bird_colors **color_info;

/* Here is the corresponding definition */
void find_wing_color(enum bird_type type, enum color *wing_color)
    *wing_color=color_info[type]->wing_color;
```

Note that the function declaration and definition have the same format. Both contain full parameter declarations inside the parentheses. In the definition, these give the actual identifier names for the parameters. The parameter declaration list that used to precede the opening brace is no longer included. In the prototype, on the other hand, the only purpose of the identifier is to document the parameter’s use. Identifiers are optional in a prototype. When they occur, the names have no significance other than as a comment to document the intended use of the parameter.

Many compilers do not yet support prototypes that contain identifiers. Thus

```
void find_wing_color(enum bird_type, enum color *);
```

is functionally equivalent to the declaration above and works with more compilers. Compilers that do not yet support identifiers in prototypes are also unlikely to support the style of function definition.

Use prototypes! They will catch many errors and also reduce the need for casts. You should get a warning or error whenever a function call uses parameters types that do not match those declared in the prototype. The following is a good example of a case where prototypes are helpful:

```
const float typical_eagle_weight=5.5;
```
This example works fine when a prototype is available because the compiler knows that the integer constant 1 will be promoted to double. Then a double constant 1 would be put in the parameter list. This will not work without a prototype since an integer 1 will be placed on the parameter list, where the called function will try to access it as a double.

You may find that there are times when you want the prototype to do a conversion for you (for example, from double to int), but the compiler generates an unwanted warning message. The SAS/C and Lattice C compilers generate such a message to warn you since the full value of the double cannot necessarily be represented in the int. In these cases, coding an explicit case to int is generally a good way to overcome the warning.

The new style of function definition acts as a prototype for all calls in the remainder of the module, whereas the old style of function definition does not. This plus the improved syntax and the fact that the committee has declared the old style obsolescent provide good reasons to begin using the new style of function definition once it is available on all compilers that you use.

CONVERSIONS AND EXPRESSIONS

Value Preserving Rules

In C, when types shorter than int (such as a bit field, a char, or a short in an implementation in which the plain int is long) are used, they are generally converted to int.

Prior to the ANSI Standard, there was no clear consensus on whether an unsigned char, short, or bit field should be promoted to a plain (signed) int or to an unsigned int. Either of these two choices is reasonable, since both destination types can represent all the values of the original type. Compilers that promote to unsigned int are said to use unsigned preserving rules. Those that promote to signed int are said to use value preserving rules.

The ANSI Standard requires that a conforming implementation use value preserving rules. If you use bit fields (which before ANSI were always unsigned) or write code for an EBCDIC environment (in which case characters are unsigned) these rules are more likely to affect you.

The following is a test program that determines which type of rules your compiler uses:

```c
#include <stdio.h>

int main()
{
    printf("SUCCESS\n");
    return 0;
}
```

The main function always needs a prototype since, in function calls, the called function will try to access it as a double.

Before macro expansion, this converts to the following:

```c
#define SUCCESS 0
```

After macro expansion, this converts to the following:

```c
return (float) number_seen / square_miles_covered;
```

This function always uses a prototype since, in function calls, float is still promoted to double in the absence of a prototype.

Regrouping Expressions

Before ANSI, compilers were allowed to freely regroup commutative expressions in an attempt to optimize them. Consider the following:

```c
#define AVERAGE_DUCK_FAMILY_SIZE 2.5f
#define AVERAGE_DUCK_NEST_DENSITY 4.0f
#define DUCK_NEST_COUNT(s) (AVERAGE_DUCK_NEST_DENSITY * s)

float expected_ducks(float sq_miles)
{
    return AVERAGE_DUCK_FAMILY_SIZE * DUCK_NEST_COUNT(sq_miles);
}
```

After macro expansion, this converts to the following:

```c
float expected_ducks(float sq_miles)
{
    return 2.5f * (4.0f * sq_miles);
}
```

It would be nice if the compiler could rearrange the expression so as to compute the product of 2.5 and 4.2 at compile time. According to the latest version of the ANSI Standard, it cannot.
In ANSI C, parentheses cause grouping. This causes a loss of optimization potential in code such as the above, where macros are used. The expression cannot be rewritten by the programmer without eliminating the macro, and lots of parentheses are needed in macros to cause correct evaluation.

Use of parentheses to cause grouping is valuable for controlling the evaluation of expressions that might overflow if evaluated in the wrong order. Before the recent adoption of this feature, the unary plus operator (+) was used to suppress commutative rearrangement (for example, a + +(b*c)). It is still supported in the Standard (for the purpose of compatibility), but the construct was so unnatural that the use of parentheses for this purpose was added, despite the negative impacts on optimizations. Unary plus is now effectively obsolete.

**Aliasing**

The C programming language allows a program to modify an object both directly through its identifier name and indirectly through one or more pointers. In such cases, each pointer is said to be an alias for the object.

Aliases cause problems for optimizers. Consider the following:

```c
extern float max_wingspan, min_wingspan;
void update_error_counter(int);

void check_wingspan(float *span)
{
  if (*span >= max_wingspan)
    update_error_counter();
  if (*span <= min_wingspan)
    update_error_counter();
}
```

At the first if statement, an optimizing compiler would normally try to keep the value of the double variable pointed to by span in a register for use in the second if statement. Unfortunately, the compiler cannot know if this pointer is also accessible to the `update_error_counter()` function (this pointer could have been saved in an external variable before the call to `check_wingspan()`). This problem is more acute on machines such as the IBM 370 with large enough register sets to keep values in registers for a long period of time.

If the `update_error_counter()` function does modify span through a pointer, any value pointed to by span that is loaded into a register at the first if statement is no longer valid at the second if statement. Assuming that the function `update_error_counter()` is in a separate compilation, the compiler cannot know what it actually does. Thus, the compiler must generate code to reload span from memory at the second if statement. The possible existence of an alias for span has caused the compiler to generate an extra load from memory.

Use of the `noalias` keyword can remedy this problem. If the function definition is

```c
void check_wingspan(noalias float *span)
{
  ...
```

then the compiler can assume that, for the duration of the `check_wingspan()` function, no other alias is used to access the double variable pointed to by span.

This problem is, in fact, more general than the example above suggests. In most cases in which an object is accessed through a pointer, it is either very difficult or impossible for the compiler to tell if an alias is being used. In these cases the compiler must make assumptions, and, to insure that correct code is always generated, it must often generate less efficient code than would be necessary if it knew that the accessed object was not aliased.

The fact that the C language allows pointers to one type of object to be copied into pointers to another type complicates the problem. If this type of pointer conversion were not allowed, it would be safe for the compiler to assume that a pointer to one type of object can never be a pointer to (or alias for) another type of object. The use of a pointer to one type in accessing another type is called type punning. Before the ANSI Standard there were no restrictions on type punning and the compiler needed to allow for it. Compilers that did allow for it were said to use worst-case aliasing assumptions.

Any of the following objects might be aliased:

- externs
- objects accessed via pointers
- autos whose address has been taken.

Compilers that use worst-case aliasing assumptions generally need to flush any registers containing the values of such objects before a pointer of any type is used to read a value. Any time any pointer is used to modify an object, compilers that use worst-case aliasing assumptions need to reload subsequent references to such an object from memory. Since these effects occur almost any time a pointer is used, register utilization can be quite poor.

Recall that the root of this problem is that before the ANSI Standard any type of pointer could be used to access any type of object (in other words, there were no restrictions on type punning). The ANSI Standard has modified the rules so that type punning is allowed only with pointers to char or with pointers to integral types of the same size and opposite signedness of an integral type (that is a pointer to char can be used to access any object, and a pointer to an unsigned int can be used to access a signed int). You should not attempt unrestricted type punning with ANSI compilers. If you do, your code may not work as expected (which would be your fault, not the compiler’s).

**Structure Assignment, Passing, and Return**

Kernighan and Ritchie did not allow the assignment of structures or passing and returning them by value in functions. However, they did suggest that this might be allowed in future versions of C. These features have been in many compilers for several years. The ANSI Standard makes their use official.

Structures assignment can be done as long as the left- and right-hand side are the same type of structure. Structures can be passed to functions, in which case, a copy of the entire structure is passed. Functions can also return structure values.

**THE PREPROCESSOR**

The ANSI Standard has enhanced the preprocessor in several ways. It has defined several preprocessing directives that were not in Kernighan and Ritchie. It has added new capabilities in the macro expansion process, and it has added several predefined macros.

**New Directives**

The ANSI Standard includes the `#elif` directive to handle multiway decisions in conditional inclusion initiated by `#if`. In the past, nested `#if` statements were needed for this.

The ANSI Standard also defines the `#error` directive, which issues a compile error. There is a new preprocessor expression operator, defined, that you can use to create arbitrary expressions that test whether one or more symbols is defined. This
effectively obsoletes the #ifdef and #ifndef directives (although compilers will continue to support them for compatibility with old code).

The following example demonstrates how some old conditional compilation directives can be improved by the use of the new features:

```c
#ifdef OSYS
    /* OSYS-specific code. */
#else
    /* CMS-specific code. */
#endif
#ifdef MSOS
    /* MS-DOS-specific code. */
#else
    #ifdef OS2
        /* OS2-specific code. Happens to be the same as the code for MS-DOS. */
#else
    #error "unsupported operating system"
#endif
#endif
```

The code above can be converted to

```c
#define OSYS #if defined(OSYS) # else #endif
#define CMS #if defined(CMS) # else #endif
#define MSOS #if defined(MSOS) # elif defined(OS2) # else
    #error "unsupported operating system"
#endif
```

You can also indent directives, such as #error above, for greater readability.

The ANSI Standard has also added the #pragma directive. This allows the program to pass implementation-defined information to the compiler. For example, when using the SAS/C compiler, the directive

```c
#pragma title Integrated Ornithology System
```
places the given title at the top of all subsequent listing pages.

Token Pasting and Stringizing

The ANSI Standard has added two features that increase the flexibility of the macro substitution process. The first, *token pasting*, allows two or more separate macro arguments to be combined into one token after substitution. This is controlled by the token pasting operator.

The second feature, *stringizing*, allows a macro parameter to be converted into a character string. Consider the following example:

```c
int kestrel_count;
#define PLSIGHT(bird) printf("%d \"s seen\n",bird,"s_count)
PLSIGHT(kestrel);
/* This expands to
printf("%d \"kestrel\" s seen\n",kestrel_,count); */
```

The # operator before the identifier bird in the macro definition indicates that double quotes should be put around the parameter when it is substituted. The # operator between the bird and _count indicates that the substituted argument should be pasted together with _count after the substitution for bird. This example also relies on the concatenation of adjacent strings (as do many uses of stringization).

Miscellaneous Changes

The ANSI Standard now requires that macro names be omitted from further macro substitution. For example,

```c
#define sizeof(x) {intI sizeof(x)
```

might be used to help in converting pre-ANSI programs that relied on sizeof returning signed int. This does not cause an infinite loop in macro expansion because the macro name sizeof() must be omitted from substitution during the rescanning process.

The predefined symbols that you can use include the following:

- __LINE__ - the current source file line number
- __FILE__ - the current source file name
- __DATE__ - date of compilation in “Mmm dd yyyy” format
- __TIME__ - time of compilation in “h:mm:ss” format
- __STDC__ - defined and given value 1 for conforming implementations.

THE LIBRARY

Kernighan and Ritchie did little to standardize the C library. UNIX operating system implementations have never had to worry about this since the UNIX environment provided the library. Implementations on other operating systems attempted to provide as much compatibility with the UNIX operating systems' library as possible. There were essentially no standards that stated what library functions must be implemented or described their semantics. The proliferation of versions of UNIX operating systems exacerbated this problem.

By far the greatest value of the ANSI Standard for the C library is that it specifies which functions must be implemented and defines their semantics in an operating system-independent manner.

The ANSI committee's function descriptions are derived from the 1984 /usr/group Standard. The committee is developing library standards in coordination with the IEEE POSIX committee (which is currently developing a vendor-independent UNIX operating system standard). The ANSI C committee is defining functions that are machine-independent and expected to be found in all C implementations. The POSIX committee is concentrating its efforts on UNIX-dependent functions.

Header Files

Every library function is declared in a header. Although it is permissible in many cases to simply declare the function without including a header, it is much better to always include the proper header. There are many good reasons for doing this:

- It eliminates the possibility of errors in the declaration of the function.
- It can give you the benefit of a prototype for all calls to the function.
- It assures that any related types have been properly defined.
- The implementation may require it for functions with variable argument lists.
- It documents which library features the module uses.
It allows you to take advantage of built-in functions.

Each of the following sections discusses functions and types related to one header. There is no attempt to discuss all the changes in detail, but many of the most important changes are covered. Not all headers are discussed. The following is a complete list of ANSI Standard headers:

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

Errors: `<errno.h>`
The macros ERM0, EDOM, and ERANGE are now defined in `<errno.h>`. Other values for errno may also be found here. In the past, your library may have used `<error.h>` for this purpose.

Note that errno is a macro. You should not try to declare it as an external variable. The ANSI Standard requires that the macro expand to a modifiable lvalue, so you may set it to zero as before.

Limits: `<float.h>` and `<limits.h>`
The implementation's limits for values of various data types are available as macros defined in these two header files. For example, the maximum value for a signed int is defined in `<limits.h>` using the name INT_MAX. The Standard lists all macros that are available along with the minimum acceptable values.

Localization: `<locale.h>`
The ANSI committee is making an effort to increase the usefulness of C in countries other than the United States. Data types have been added for multi-byte character set support that allows for the implementation of complicated alphabets such as those used in China and Japan. The library is required to implement a set of types and functions that allow the program to discern the local conventions for representations of elements such as the currency symbol and decimal point. This information is known as the locale. The functions and types that handle locale dependences are defined in `<locale.h>`.

Signals: `<signal.h>`
Signals allow unpredictable events to be handled. They have been available (in various forms) in UNIX operating systems and other implementations for some time. The ANSI Standard requires that a conforming implementation have some support for signals.

The following signals must be defined:

- SIGABRT - abnormal termination
- SIGFPE - erroneous arithmetic operation, such as divide by zero
- SIGILL - attempt to execute an illegal instruction
- SIGINT - attention signal from a user terminal
- SIGSEGV - memory protection exception
- SIGTERM - a termination request sent to the program.

Other signals may be supported by the implementation. For example, the SAS/C library raises the signal SIGMEM when there is no more space on the run-time stack.

The only functions defined by the ANSI Standard are signal() (which defines a function as a signal handler) and raise() (which raises a signal artificially).

Note that the ANSI Standard does not specify that the implementation must raise the defined signals except in response to a call to raise(). Sometimes this is a result of hardware or operating system deficiencies. For example, since an 8086 has no memory protection, SIGSEGV would never be raised by the hardware. In other cases, the particular library may not provide access to hardware and operating system signals that do exist. This is another example of the fact that ANSI conformance does not always imply quality of implementation.

Variable-Length Argument Lists: `<stdarg.h>`
The ANSI Standard has defined a portable mechanism for defining and calling programs with variable-length argument lists: using the macros va_start, va_arg, and va_end. These are defined in `<stdarg.h>`.

A Few Important Definitions: `<stddef.h>`
Four important macros are defined in `<stddef.h>`:

- NULL
- size_t - the type of a sizeof expression
- ptrdiff_t - the type of a pointer subtraction expression
- offsetof(type,membername) - a macro that returns the offset of a member from the beginning of a structure.

Note one important point: the ANSI Standard Requires that size_t be an unsigned integral type. Not all compilers currently meet this requirement. If you have such a compiler, this may break your code. The unsignedness of size_t often propagates through an entire expression and changes the resulting value (in much the same way as happened in the unsigned preserving example).

I/O: `<stdio.h>`
The ANSI Standard defines only standard I/O functions (such as fopen() and fseek()). It does not define the UNIX operating systems' I/O functions (such as open() and lseek()). This implies that portable programs should not use the UNIX operating systems' I/O. Given that it is a goal of many non-UNIX implementations to provide a high degree of compatibility with UNIX operating systems, many non-UNIX libraries will continue to support UNIX operating systems' I/O. But, doing so is not required by ANSI.
Besides omitting UNIX operating systems' I/O from the ANSI Standard, the most important changes to I/O have been careful work to define the semantics in a portable manner. The fundamental concept here is the distinction between text and binary streams (controlled by the second argument to fopen()). You establish a text stream when you open a file whose contents you intend to process as textual data. You should do so when you do not expect to be working with unprintable characters and want the library to convert newlines into your operating system's means of distinguishing records (for example, a carriage return/newline pair under MS-DOS).

When you use a binary stream the library must record characters exactly as they are written. When writing binary data, you should not assume that newlines mark the end of records and therefore transform them to something else. For example, if your data compose an object program, the newline character may be meant to represent an opcode. Inserting a carriage return/newline pair in its place will corrupt the program.

The text/binary distinction is controlled by a mode string that you supply to fopen(). If the mode string contains a b, as in rb (read binary file) or w+b (create binary file for update), then the library uses a binary stream to access the file. If it does not contain a b, as in r (read text file) or w+ (create text file for update), the library uses a text stream.

The ANSI committee has also addressed the problem that many operating systems have useful file types that do not support the full seeking capability of UNIX operating system disk files. For example, under MVS, files are generally addressable by block and, when using variable length records, each block can contain different numbers of bytes. The result is that it is impossible to seek to an arbitrary byte position in the file without reading the file from the beginning to that point.

A similar situation occurs with files accessed via text streams under MS-DOS, because although it is possible to seek to an arbitrary byte in the file, the fact that the C library must view carriage return/newline pairs as a single character means that the nth byte, as seen by DOS, is not the same as the nth byte that the library must find.

This problem has been addressed in two ways. First, the ANSI Standard states that fseek() should not be used to seek to an arbitrary byte of a text stream. For text streams, fseek() can only be used to seek to a position previously returned by ftell() or lseek() instead.

Second, two new functions, fsetpos() and fgetpos(), are defined. Instead of returning a long as a file position indicator, they return the type fpos_l. They do not support seeking to an arbitrary byte. Instead they allow you to get the current file position (using fgetpos()), save it, and then, at some later time, return to the position (using fsetpos()).

These functions work on almost all types of files (terminals and printers are exceptions). You can use them with text or binary access. They even work on files larger than 2 gigabytes (with which fseek() and ftell() have real problems). If positioning to an arbitrary byte is not a requirement, you should use these functions; expanding the number of file types that your program supports will enhance your ability to use data files created by other programs such as text editors and non-C applications.

Various Utility Functions and Macros : <stdlib.h>

This header controls a diverse group of useful functions including the following:

- memory allocation (such as malloc() and free())
- string-to-number conversion (such as atof())
- random number generation functions
- operating system communication functions (including system(), exit(), and abort())
- the sorting and searching functions qsort() and bsearch()
- functions for absolute value and division of both integral and floating types.

The most important things to note are that the UNIX operating systems' dependent functions such as fork() are not required by the ANSI Standard (the semantics of fork() are not implementable in many environments) and that abs() takes only integer arguments. The closest equivalent to fork() is the system() function that, when provided with a character string of an implementation-defined format, attempts to execute it as a command in an implementation-defined way. As with any implementation-defined feature, check your documentation for details.

In the past, some implementations (such as Lattice C) have defined abs() as a macro that will work on any arithmetic type. If any of your programs rely on abs() to handle long or double, they must be changed to use labs() or labs() instead.

This header also contains the macros EXIT_FAILURE and EXIT_SUCCESS, which expand to the host operating system's return values for the exit conditions. They are useful as parameters to exit() and when returning from main().

String Functions : <string.h>

This header contains a variety of string functions. Some, such as strtok (which breaks a string into tokens), perform string processing that is relatively sophisticated. The memory-oriented functions, such as memcpy(), memcmp(), and memset(), are noteworthy because they may be much more efficient than functions such as strcpy() that work with zero-terminated strings. For example, the SAS/C compiler and Lattice C compiler implement built-in functions for these. Thus, you can expect a memory-oriented function often to be executed in just a few instructions.

USING THE NEW FEATURES

Cautions

The ANSI Standard is still in draft form. There is probably no compiler on the market that actually includes all the features described here. Major changes are still being made in every version of the Standard. Until it is officially ratified, no one can claim to be ANSI-conforming. Just to remain conforming with the current draft would require that a compiler vendor operate on a three-month release cycle.

Beware of advertising claims that a compiler is ANSI-compatible. This is a very misleading claim. Many vendors are claiming that they are ANSI-compatible without claiming that they are an ANSI-conforming implementation. This claim generally means that the compiler has many of the features required by ANSI, but not all of them. It almost certainly means that the implementation ignores the myriad details required by the ANSI Standard.

Even when a vendor does claim to be an ANSI-conforming implementation, it is hard for a customer to know that the vendor has achieved conformance in detail. Yet lack of attention to detail can cause problems when you try to port your code to another compiler.
Be aware that ANSI conformance is not the total measure of a compiler. ANSI conformance does not assure you that good diagnostics will be generated. It does not assure you of compatibility with UNIX operating systems. It does not assure you that the compiler is bug-free. It does not assure you that efficient code will be generated.

Most important, ANSI conformance says nothing about machine-or operating system-specific support that your compiler and library provide. For example, an MS-DOS compiler that supports only the small memory model can be ANSI-conforming. An ANSI-conforming MVS implementation may not support reentrancy, VSAM files, or 31-bit addressing.

Until the ANSI Standard is ratified, the best approach is to be selective about which features you use. Use of some ANSI features before they are widely available in the market may cause you problems in porting your code to other compilers. The more well-established features (such as structure assignment) should be safe to use. Features that can be isolated in header files (such as prototypes) or conditionally removed by the preprocessor (such as the noalias keyword) are also unlikely to cause portability problems. Monitor the ANSI conformance of all your compiler vendors so that you know when the features that you want will be available.

Nonstandard Features

Compilers often have nonstandard extensions. The near and far keywords on MS-DOS compilers are a good example. Nonstandard extensions should be flagged as errors by ANSI Standard-conforming implementations. This presents a problem because these useful features cannot just be eliminated. Thus, many compilers will probably implement switches to control whether such features are enabled or disabled. In many cases, additional keywords begin with two underscores to prevent conflicts with your identifiers.

Systems Application Architecture

If the new features in ANSI C are not sufficient to convince you of its importance, IBM's commitment to it might be. The C programming language is one of only three languages supported by IBM's Systems Applications Architecture (SAA). Although IBM has not yet committed itself to supporting the eventual ANSI Standard, the SAA C definition already includes most of the features from the Draft Standard.

Future Directions

Release 3.01 of the SAS/C mainframe compiler incorporates most of the features described in this paper. The main exceptions are the signed keyword, long double, new-style function definitions, dummy identifiers in prototypes, and some preprocessor features. Release 4.00 of the Lattice MS-DOS and OS/2 C compilers and Release 3.02 of the SAS/C compiler will be quite close to ANSI conformance. The releases following these are intended to be ANSI-conforming in every detail. SAS Institute is committed to have its C compiler products, SAS/C, SAS/CX, and the Lattice C compilers, conform to the ANSI C Standard.

SAS Institute is an active participant in the ANSI C Standards committee. The Institute has been incorporating parts of the ANSI Standard in SAS/C compiler releases since becoming a member in March 1985. The Institute is also intent on supporting SAA. If there are any conflicts between the ANSI and SAA C standards, SAS Institute will allow use of compiler and run-time switches and/or alternate libraries to support both.

Ordering A Copy of the Standard

To order a copy of the ANSI Standard and the accompanying Rationale, call Global Engineering Documents at 1-800-854-7179.

CONCLUSION

If you make careful use of the new features in ANSI C, you can write programs that are more efficient, portable, and easier to understand. At the same time, the compiler can catch more of your errors and you will find that less coding effort is required. If you are comfortable with most of the material in this paper, you can study the Standard for more details.

SAS/C and SAS/CX are trademarks of SAS Institute Inc., Cary, NC, USA.

IBM is a registered trademark of International Business Machines Corporation, Armonk, NY, USA.

Lattice is a registered trademark of Lattice, Inc., Glen Ellyn, IL, USA.

MS-DOS is a registered trademark of Microsoft Corp.

UNIX is a registered trademark of AT&T.