C11: The New C Standard

by Thomas Plum

The committee that standardizes the C programming language (ISO/IEC JTC1/SC22/WG14) has completed a major revision of the C standard. The previous version of the standard, completed in 1999, was colloquially known as "C99." As one might expect, the new revision completed at the very end of 2011 is known as "C11."

Concurrency

C11 standardizes the semantics of multi-threaded programs, potentially running on multi-core platforms, and lightweight inter-thread communication using atomic variables.

The header `<threads.h>` provides macros, types, and functions to support multi-threading. Here is a summary of the macros, types, and enumeration constants:

Macros: `thread_local, ONCE_FLAG, TSS_DTOR_ITERATIONS cnd_t, thrd_t, tss_t, mtx_t, tss_dtor_t, thrd_start_t, once_flag.`

Enumeration constants to pass to `mtx_init`: `mtx_plain, mtx_recursive, mtx_timed.`

Enumeration constants for threads: `thrd_timedout, thrd_success, thrd_busy, thrd_error, thrd_nomem.`

Functions for condition variables:

```
call_once(once_flag *flag, void (*func)(void));
cnd_broadcast(cnd_t *cond);
cnd_destroy(cnd_t *cond);
cnd_init(cnd_t *cond);
cnd_signal(cnd_t *cond);
cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx, const struct
timespec *restrict ts);
cnd_wait(cnd_t *cond, mtx_t *mtx);
```

The `mutex` functions:

```
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx,
const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
```

Thread functions:

```
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
noreturn void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
int thrd_sleep(const struct timespec *duration, struct timespec
*remaining);
void thrd_yield(void);
```

Thread-specific storage functions:

```
int tss_create(tss_t *key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

These standardized library functions are more likely to be used as a foundation for easier-to-use APIs than as a platform for building applications. (See "When Tasks Replace Objects", by Andrew Binstock, March 28, 2012, for discussion of higher-level APIs.) For example, when using these low-level library functions it is very easy to create a data race, in which two or more threads write (or write-and-read) to the same location without synchronization. The C (and C++) standards permit any behavior whatever if a data race happens to some variable `x`. For example, some bytes of the value of `x` might be set by one thread and other bytes could be set by another thread ("torn values"), or some side-effect that appears to take place after assignment to `x` might (to another thread or another processor) appear to take place before that assignment. Here is a short program that contains an obvious data race, where the 64-bit integer (`long long`) named `x` is written and read by two threads:

```
#include <threads.h>
#include <stdio.h>
#define N 100000
char buf1[N][99]={0}, buf2[N][99]={0};
long long old1, old2, limit=N;
long long x = 0;

static void do1()  {
   long long o1, o2, n1;
   for (long long i1 = 1; i1 < limit; ++i1) {
      old1 = x, x = i1;
      o1 = old1;  o2 = old2;
      if (o1 > 0) { // x was set by this thread
         if (o1 != i1-1)
            sprintf(buf1[i1], "thread 1: o1=%7lld, i1=%7lld, o2=%7lld", o1, i1, o2);
      } else {      // x was set by the other thread
         n1 = x, x = i1;
         if (n1 < 0 && n1 > o1)
```

```
            sprintf(buf1[i1], "thread 1: o1=%7lld, i1=%7lld, n1=%7lld", o1, i1, n1);
      }
   }
}

static void do2()  {
   long long o1, o2, n2;
   for (long long i2 = -1; i2 > -limit; --i2) {
      old2 = x, x = i2;
      o1 = old1; o2 = old2;
      if (o2 < 0) { // x was set by this thread
         if (o2 != i2+1)
            sprintf(buf2[-i2], "thread 2: o2=%7lld, i2=%7lld, o1=%7lld", o2, i2, o1);
      } else {      // x was set by the other thread
         n2 = x, x = i2;
         if (n2 > 0 && n2 < o2)
            sprintf(buf2[-i2], "thread 2: o2=%7lld, i2=%7lld, n2=%7lld", o2, i2, n2);
      }
   }
}

int main(int argc, char *argv[])
{
   thrd_t thr1;
   thrd_t thr2;
   thrd_create(&thr1, do1, 0);
   thrd_create(&thr2, do2, 0);
   thrd_join(&thr2, 0);
   thrd_join(&thr1, 0);
   for (long long i = 0; i < limit; ++i) {
      if (buf1[i][0] != '\0')
         printf("%s\n", buf1[i]);
      if (buf2[i][0] != '\0')
         printf("%s\n", buf2[i]);
   }
   return 0;
}
```

If you had an implementation that already conformed to the C11 standard, and you compiled this program for a 32-bit machine (so that a 64-bit `long long` is written in 2 or more memory cycles), you could expect to see confirmation of the data race, with a varying number of lines of output such as this:

```
thread 2: o2=-4294947504, i2=    -21, o1=  19792
```

The traditional solution for data races has been to create a lock. However, using atomic data can sometimes be more efficient. Loads and stores of atomic types are done with sequentially consistent semantics. In particular, if thread-1 stores a value in an atomic variable named `x`, and thread-2 reads that value, then all other stores previously performed in thread-1 (even to non-atomic objects) become visible to thread-2. (The C11 and C++11 standards also provide other models of memory consistency, but even experts are cautioned to avoid them.)

The new header `<stdatomic.h>` provides a large set of named types and functions for manipulation of atomic data. For example, `atomic_llong` is the typename provided for atomic `long long` integers. Similar names are provided for all the integer types. One of these typenames, `atomic_flag`, is required to be lock free. The standard includes a macro named `ATOMIC_VAR_INIT(n)`, for initialization of atomic integers, as shown below.

The data race in the previous example can be cured by making `x` an `atomic_llong` variable. Simply change the one line that declares `x` in the above listing:

```
#include <stdatomic.h>
```

```
atomic_llong x = ATOMIC_VAR_INIT(0);
```

By using this atomic variable, the code operates without producing any data-race output.

A Note on Keywords

The C committee prefers not to create new keywords in the user name space, as it is generally expected that each revision of C will avoid breaking older C programs. By comparison, the C++ committee (WG21) prefers to make new keywords as normal-looking as the old keywords. For example, C++11 defines a new `thread_local` keyword to designate static storage local to one thread. C11 defines the new keyword as `_Thread_local`. In the new C11 header <threads.h>, there is a macro definition to provide the normal-looking name:

```
#define   thread_local    _Thread_local
```

In these articles, I will assume that you include the appropriate headers, so I will show the normal-looking names.

The `thread_local` Storage Class

This new `thread_local` storage class provides static storage that is unique to each new thread, and is initialized before the thread begins execution. However, there are no safeguards to prevent you from taking the address of a thread_local variable and passing it to other threads; what happens next is implementation-defined (i.e., not portable). Each thread has its own copy of `errno` in `thread_local` storage.

Threads are Optional

C11 has designated several features as optional. For example, if the implementation defines a macro named `__STDC_NO_THREADS__`, then it will presumably not provide a header named `<threads.h>` nor any of the functions defined therein.

Politics, Design, and Incomplete Information

As a general rule, WG21 entrusts Bjarne Stroustrup with the overall design-and-evolution responsibility; do an online search for “camel is a horse designed by committee” to understand the reasons for this approach. However, there is one design principle that motivates both WG14 and WG21: don’t leave room for a more-efficient systems-programming language underneath our language (C or C++).

Some participants (call them “Group A”) expect that atomic data will remain a seldom-used specialty, but others (call them “Group B”) believe that atomic data will become a crucial feature, at least for a systems-programming language.

Over the past decades, various higher-level languages have been built based on C: Java, C#, Objective C, and of course, C++ and subsets-or-supersets based on C++ (such as D and Embedded C++). Many

companies that participate in WG14 and WG21 have made decisions regarding the languages in which their apps will be written. Those companies that chose C++ as their upper-level app language (call them "Group D") are often content for C to be stabilized (or for WG21 to control its standardization), whereas companies that chose other languages (call them "Group C") sometimes regard C as a crucial foundation under their upper-level app language.

With this much background, I can give an account of the evolution of atomics in C11. The design of atomics in C++11 made crucial use of templates, such that `atomic<T>` is the simple and universal way of getting the atomic version of any type `T`, even if `T` is a `class` or `struct` type; and `atomic<T*>` retains all the compile-time type information of what `T*` points to. However, for several years, the C design used only the several dozen named types (such as `atomic_llong` shown above). One advantage of the named-type approach is that it requires no changes to the compiler itself; it can be implemented in a library-only solution, which invokes system-dependent intrinsic functions at the very lowest level. However, the named-type approach precludes creating an atomic for any C `struct` (no matter how small) or for a `T*` pointer (for a general `T` which is known to the compiler). Largely due to the influence of Group B and Group C opinions within WG14, a decision was made to require a C11 compiler to recognize an atomic `T` for any type `T`.

There was also a subsequent controversy within WG14 about the compiler syntax for specifying an atomic `T`. One approach ("atomic-parenthesis") was motivated by compatibility with C++: let `_Atomic(T)` be the syntax for designating an atomic `T`. Then that same source program could be compiled as C++ simply by defining one macro:

```
#define _Atomic(T)    atomic<T>
```

The other side of the controversy preferred to create a new type-qualifier (analogous to the C99 treatment of `_Complex`); using this syntax ("atomic-space"), the type atomic `T` would be written as "`_Atomic T`". A program written using that syntax could not directly be compiled as C++ (without making use of compatibility macros that would look essentially like the atomic-parenthesis approach).

Both sides of this controversy agreed that, once a team commits to modifying the compiler for this feature of C11, it's a relatively minor amount of incremental work to implement both the atomic-parenthesis syntax and the atomic-space syntax. In the end, that's the position that prevailed in WG14. In the meantime, the price of that decision is that only the named-type approach is available today (until compilers implement the C11 syntaxes), and the most vocal Group D participants can grumble about the decisions of WG14 creating incompatibilities with C++.

Getting the C11 Standard

C11 and C++11 are available from the ANSI store. For C11, use this link: http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2fISO%2fIEC+9899-2012. For C++11, use this link: http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2fISO%2fIEC+14882-2012 . Each costs $30 USD for PDF. The whole process takes only a few minutes.

. . . . . .

C11 and Cybersecurity

C and C++ are members of the same family of languages. The evolutionary boldness of C++ removes some of the marketplace pressure on C; the people who are continually pushing for innovation are naturally drawn to the C++ development process. Each language had a coherent original design (by Dennis Ritchie and Bjarne Stroustrup, respectively), followed by successive refinement in a very competitive marketplace of ideas. Both languages share an extreme concern for performance, with the slogan "don't leave space for a more-efficient systems-programming language underneath our language (C or C++)", as I mentioned last month. It's unfair to complain that the original designs assigned too little importance to cybersecurity; both languages pre-date the beginnings of concern for security. But in recent years the marketplace has started to emphasize cybersecurity, and the programming languages are responding in several ways.

In early 2002, Bill Gates' "battleship-turning" memo made cybersecurity a top goal for Microsoft. About a year later, Microsoft proposed a new "bounds-checking" library to WG14, which eventually became Technical Report 24731-1; now it has become part of C11 as the (optional) Annex K. (Look here for an almost-final draft of C11.)

The C11 Annex K functions

I'll start my tour of Annex K with the `fopen_s` function. The main innovation is that files are opened with exclusive (also known as non-shared) access. Furthermore, if the `mode` string doesn't begin with `'u'` (and of course, if the code is being remediated from using the older `fopen`, then `mode` doesn't begin with `'u'`), then to the extent that the underlying system supports it, the file gets a file permission that prevents other users on the system from accessing the file.

In this article, I'll sequentially enumerate the security benefits of these "_s" functions; the new semantics illustrate pattern #1, "least privilege". This "exclusive" mode was previously available in the Posix open() function, but the ISO standard for C doesn't standardize system-dependent low-level I/O. See Robert Seacord's book Secure Coding in C and C++ for detailed discussion of these various security benefits of the Annex K library.

If a file is opened with `'x'` as the last character in the `mode` argument, and the requested filename is already in use, the `fopen_s` function fails (as opposed to truncating the existing file, which is presumably already being used by someone). If the application program had been required first to check whether the file was in use and then to create the new file, this would illustrate the "time-of-check versus time-of-use (TOCTOU)" vulnerability; the Annex K version assists with pattern #2, "minimize TOCTOU vulnerability".

The `mode` argument is passed to `fopen_s` as a `const char*` pointer, as is the `filename` argument. Requiring these pointers to be non-null is one of the *runtime-constraints* of the `fopen_s` function, to use the C11 terminology.

If any of the runtime-constraints are violated, the library function (`fopen_s` in this case) invokes the run time-constraint handler. (In Visual Studio, this handler is known as the invalid parameter handler – same concept, different name.) This is pattern #3: invoke the runtime-constraint handler if any

runtime-constraint is violated. Usually, a runtime-constraint violation would have resulted in an undefined behavior if not caught.

If the runtime-constraints were not violated, then `fopen_s` returns the resulting `FILE*` pointer through an argument, rather than producing it as the returned value of the function. If `fopen_s` fails for any of several reasons, it returns a nonzero value according to the conventions encoded in `<errno.h>`; the various Annex K headers provide the typedef name `errno_t` for this `int` returned value. This, then, is pattern #4: reduce the inconsistency of return-value idioms to the greatest extent possible, by uniformly returning `errno_t` for erroneous conditions that didn't violate a runtime-constraint.

This initial discussion about `fopen_s` has introduced the first four patterns of the Annex K library: (1) provide least privilege; (2) minimize TOCTOU vulnerability; (3) reduce the return value variability using `errno_t` returned values; (4) use runtime-constraint handlers for logic errors.

In the original C standard, and in C++ still today, most library functions specify something like "if copying takes place between objects that overlap, the behavior is undefined". In C99 and C11, there is a syntactic way to specify this restriction, the **`restrict`** keyword. As a result of all these various design decisions, the calling sequence for `fopen_s` looks like this:

```
errno_t fopen_s(FILE * restrict * restrict streamptr,
     const char * restrict filename,
     const char * restrict mode);
```

Designing the runtime-constraint handler provides the implementation and the project team a range of choices. The logically simplest handler simply invokes `abort()`. A somewhat more complex architecture gives the user a choice between aborting or debugging, potentially preserving the full state of the stack frames and global variables. (From a standards-conformance point of view, the invocation of an interactive debugger is equivalent to invoking `abort()` —anything that happens subsequently is under the control of the interactive user.)

Other forms of handlers could be used. In an application that never terminates, the handler could reinitialize, flush the current transaction, start a new transaction, and so forth. In a specialized testing situation, the handler could log the failures.

The `freopen_s` function illustrates the same patterns as `fopen_s`, including the `'x'` and `'u'` mode flags.

Continuing with the file oriented functions, consider `tmpnam_s`:

```
errno_t tmpnam_s(char *s, rsize_t maxsize);
```

The function illustrates pattern #5:

> In the calling sequence of the function, every pointer through which the function might modify an array is immediately followed by the number of elements which the function is permitted to modify.

In the case of `tmpnam_s`, the second argument specifies a maximum for the number of characters that can be modified by `tmpnam_s`. The type of the second argument is `rsize_t`, designating a "restricted `size_t`" value. The intent is to prevent the common error of inadvertently passing a negative value, which after conversion to an unsigned type, becomes a huge number, and in this case, defeating the purpose of bounds-checking the string written into `s`. This common error is intended to be caught within `tmpnam_s` by comparing `maxsize` against `RSIZE_MAX` and invoking the runtime-constraint handler if it's larger. (I've said "intended" several times, because Annex K makes it optional whether `RSIZE_MAX` is any smaller than `RSIZE_MAX`.) This manner of designating bounding values with the type `rsize_t` is pattern #6 of the Annex K library.

Next, consider the `tmpfile_s` function:

```
errno_t tmpfile_s(FILE * restrict * restrict streamptr);
```

It could be invoked like this:

```
FILE *myTempFile = 0;
errno_t err = tmpfile_s(&myTempFile);
```

There is a window of TOCTOU vulnerability between obtaining a filename from `tmpnam_s` and subsequently creating that file with `fopen_s`; using `tmpfile_s` eliminates that particular vulnerability. Consequently, `tmpfile_s` illustrates patterns 1, 2, 3, and 4.

Pattern #7 is easy to describe: "eliminate the `%n` format". For the details, refer to Seacord and the original Rationale for the library that become Annex K. The basic problem with `%n` is that the `printf` family of functions are intuitively thought of as "output" functions, but the `%n` format can be used to modify memory, and therefore provides an attack surface.

These, then, are the "`_s`" versions of the formatted output functions: `fprintf_s, printf_s, snprintf_s, sprintf_s, vfprintf_s, vprintf_s, vsnprintf_s, vsprintf_s, fwprintf_s, snwprintf_s, swprintf_s, vfwprintf_s, vsnwprintf_s, vswprintf_s, vwprintf_s, wprintf_s.`

Pattern #8 is rather technical, but significant: if the various formatted functions produce overlapping stores, the resulting behavior is not undefined, but is merely unspecified. Implementing patterns 4 (handlers for e.g. null arguments), 5 (buffer sizes), 6 (RSIZE_MAX), and 8 (overlapping stores), we have the "`_s`" versions of the formatted input functions: `fscanf_s, scanf_s, sscanf_s, vfscanf_s, vscanf_s, vsscanf_s, fwscanf_s, swscanf_s, vfwscanf_s, vswscanf_s, vwscanf_s, wscanf_s`

Pattern #9 is even more technical: when the time-and-date functions produce a "year" value, it should be bounded to the interval [0, 9999]; added to the other patterns, this produces the time-and-date functions: `asctime_s, ctime_s, gmtime_s, localtime_s.`

Pattern #10 guarantees that `memset_s` will over-write the argument array, even if the optimizer thinks that those stores are "useless", such as when over-writing a password before leaving a function.

Pattern #11 provides an extra argument to keep track of previous state information, to avoid static buffers that would prevent re-entrancy or use in a multi-threaded environment: `bsearch_s`, `qsort_s`, `strtok_s`, `wcstok_s`.

Pattern #12 is to chop (or zero-fill) the resulting string if a runtime-constraint error happens: `gets_s`, `getenv_s`, `wctomb_s`, `mbstowcs_s`, `wcstombs_s`, `memcpy_s`, `memmove_s`, `strcpy_s`, `strncpy_s`, `strcat_s`, `strncat_s`, `strerror_s`, `strnlen_s`, `wcscpy_s`, `wcsncpy_s`, `wmemcpy_s`, `wmemmove_s`, `wcscat_s`, `wcsncat_s`, `wcsnlen_s`, `wcrtomb_s`, `mbsrtowcs_s`, `wcsrtombs_s`

Pattern #13 is to provide the bounds that will be needed to allocate buffers: the `strerrorlen_s` function tells how many characters will be needed to store the locale-specific error message for one specific `errno` value.

Finally, pattern #14 is illustrated by all the functions in Annex K: permit a localized remediation of existing code, without global design changes. Each of the various "`_s`" functions can replace its previous version by changing only one or two lines of the existing code.

The Annex K functions are widely available on Visual Studio and a few other places; still, there's no reason why they shouldn't already be available on all platforms. Perhaps there is some degree of "not invented here" resistance; I hope these articles will help create greater marketplace demand. Talk to your compiler/library providers.

Annex L

There has been a tendency to approach the requirements for safety-critical, zero defects, and cybersecurity with the same developmental methods, producing high-integrity applications at a correspondingly high cost. However, cybercriminal exploits tend to focus on the most popular apps, which are often produced under less-than-ideal schedule and budget constraints. The languages chosen for hopefully-popular apps are frequently C and C++.

Within WG14 there have been several initiatives to improve software security without sacrificing the efficiency advantages of C, or the developmental methodologies that organizations are already familiar with.

Within the world of safety-critical development methods, it is common to target the elimination of all undefined behaviors (UBs) in C, on the grounds that compilers are free to do anything whatever when an app produces UB. However, compiler developers are very influential within WG14, and they know that in almost all cases of UB, the hardware actually produces a benign result, and that often when some corner case is identified as UB, the standard is marking it as "non-portable", and not as "dangerous".

The Analyzability Annex of C11 (Annex L) identifies a small number of UBs as "critical UB", classifying all the others as "bounded UB", and imposes some implementation constraints on the resulting behavior. The net result is that when an implementation provides this analyzable behavior, and the app is subjected to static analysis, the actual app when executed does implement the source code of the program as analyzed.

With the benefit of hindsight, I have found one improvement that I will suggest for the Analyzability Annex: an implementation should be permitted by Annex L to generate code that violates the constraints in the Annex, provided that it produces a warning message when it does so. After all, "analyzability" relies on a project methodology that focuses attention upon the warnings generated by the static analyzer and the compiler, so guaranteeing the production of a warning is all that should be required by Annex L.

The C Secure Coding Rules project

ISO and IEC currently define a Technical Specification (TS) to have less than the official status of an International Standard (IS). WG14 has used this less-formal approach to several topic areas (including the Bounds-Checking Library TR 24731-1 mentioned above) for specifications that may benefit from experience in the marketplace before being standardized.

Further work is under way within WG14, a Technical Specification (TS 17961) for "C Secure Coding Rules" (CSCR). Most of the TSs (and ISs) produced by the programming language committees target the compiler-and-library marketplaces, but the CSCR TS primarily targets the static analyzers marketplace.

. . . . .

C11 Ease of Use

The 2011 revision of the ISO standard for C ("C11") provides several ease-of-use features, most of which are compatible with C++11. In order to use the normal-looking names I'll show here, you need to include these headers: `<stddef.h>`, `<stdlib.h>`, `<assert.h>`, `<complex.h>`, `<stdnoreturn.h>`, `<uchar.h>`, `<stdatomic.h>`, and `<stdalign.h>`.

Alignment

C11, and C++11, provide new syntax for specifying alignment. The expression `alignof(`*type-name*`)` a designates the alignment of *type-name*; it is a constant expression, as is the familiar `sizeof(`*type-name*`)`. (There's one exception in C: applying `sizeof` to a variable length array, or VLA, produces a non-constant expression.) The expression `alignof(char)` is, of course, always 1.

There is a similar syntax for declarations:

```
int alignas(double) b;
```

specifies that `b` is an `int`, but is aligned suitably for a `double`. Or for a more realistic example,

```
alignas(double) unsigned char buf[sizeof(double)];
```

specifies that `buf` is a array of `unsigned char`, whose size and alignment would be suitable to hold a `double`.

Alignment can be specified as an integer: `alignas(`*constant-expression*`)` specifies the *constant-expression* as an alignment. Thus, `alignas(`*type-name*`)` means the same thing as `alignas(alignof(`*type-name*`))`.

For each target platform, there is some type which has the largest alignment requirement; that type can be named by the typedef `max_align_t`, so a declaration that specifies `alignas(max_align_t)`

requests an alignment that will be suitable for any type on that platform. If a program requests an alignment that is greater than `alignof(max_align_t)`, the program is not portable, because support for an *over-aligned type* is optional.

The C11 library provides `aligned_alloc(size_t bound, size_t nbytes)`, which allocates `nbytes` of storage aligned on a `bound` boundary. The most common use case heard by the committee was to request a buffer aligned on a cache boundary (typically 32k or 64k); however, you have to check your own compiler's manual, because the implementation gets to determine the valid alignments.

Unicode strings and constants

The new `u8` prefix for strings creates a string (i.e., an array of `char`) which is encoded using the UTF-8 encoding. If your text editor and your compiler are using the ASCII representation (most are), then the string `u8"John Doe"` will contain the same characters as the ordinary string `"John Doe"`. The crucial difference comes when your program needs to represent international characters beyond the basic 7-bit ASCII (English) characters. If your text editor and compiler can handle the characters, then your program could contain a string like `u8"α Ä Ǽ Ω"`, and pass that string to the various C library functions that handle ordinary strings (arrays of `char`).

The UTF-8 encoding is increasingly popular; for example, it is the default encoding for XML. To the extent that you have a choice about character representations, it appears to me, with benefit of decades of hindsight, that UTF-8, using the `u8` strings, is the simplest and best choice.

However, you may not have this simple choice, so C11 (and C++11) also provide several other Unicode representations. A string like `u"αΩ"`creates an array of `char16_t` values (encoded in UTF-16); similarly, a string like `U"αΩ"`creates an array of `char32_t` values (encoded in UTF-32). Also, there are character constants for `char16_t` and `char32_t` values, written as `u'α'` and `U'α'`. Unfortunately, if you need to use these more complex features, you may need to know about endian-ness, surrogate characters, differences between Windows and UNIX/Linux representations, and this overview article couldn't provide enough details to address all those issues.

Type-generic Macros

The C99 standard introduced type-generic macros into the standardized library; for example, you could invoke `fabs(x)`, where `x` is either `float`, `double`, or `long double`. What happened auto-magically was that invocation of the type-generic macro `abs` would cause invocation of one of three separate library functions `fabsf(float)`, `fabs(double)`, or `fabsl(long double)`. However, in C99 you had no opportunity to use the same magic for your own purposes. Now, in C11, you could create an `fabs(x)` that would be portable to any other C11 compiler:

```
#define fabs(X) _Generic( (X),                  \
                          long double: fabsl,   \
                          default: fabs,        \
                          float: fabsf          \
                        ) (X)
```

This method defines a macro named `fabs`, which will cause the invocation of several different named library functions, using the new C11 syntax for the `_Generic` keyword. That `fabs` macro is an ordinary macro defined in the preprocessor. For example, `fabs` could be undefined (using `#undef`). As you see, type-generic macros provide only a tiny portion of the full-blown overloading that is available in C++, but it's enough for purposes such as the type-generic math library.

Miscellaneous Ease-of-use Features

It is now possible, in C11 and C++11, to inform the compiler that a function will not return. For example. `exit` is a function that does not return, so it can be declared like this:

```
noreturn void exit(int status);
```

Using `noreturn` in this way can assist the compiler's optimizer, possibly eliminating unnecessary warnings.

C11 and C++11 provide `static_assert(`*constant-expr, string-literal*`)`; if the *constant-expr* is zero, then a diagnostic message containing the text of the *string-literal* will be printed. As the name implies, the static_assert is evaluated at compile time, so it can prevent compilation with incompatible options; for example

```
static_assert(sizeof(void*) == 4,
     "64-bit code generation not supported");
```

One common use of `static_assert` is to verify that resource configuration is adequate:

```
static_assert(NUMBER_OF_BUCKETS < 16,
     "NUMBER_OF_BUCKETS must be at least 16");
```

The message produced by `static_assert` will contain, besides the *string-literal* argument, the file name, line number, and function name (if any).

The C11 standard provides three macros that are helpful for C/C++ compatibility for programs that use complex floating-point values:

```
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
```

Your C++ version of the program could create corresponding macro definitions (but the C++11 standard does not provide these):

```
#define CMPLX(x, y)  std::complex((double)x, (double)y)
#define CMPLXF(x, y) std::complex((float)x, (float)y)
#define CMPLXL(x, y) std::complex((long double)x, (long double)y)
```

Finally, there's a committee decision that was inadvertently left out of the published standard: the pre-defined macros _ _STDC_VERSION_ _ and _ _STDC_LIB_EXT1_ _ are defined to be 201112L.

Several other corrections and improvements were made, which won't be itemized here.

Compatibility with C++

Compatibility with the evolving standard for C++ was a high priority, and a serious challenge, since both standards were completed in the fourth quarter of 2011.

All the new features described in this ease-of-use article can be used in a C11 program, or a C++11 program, with all the same semantics (except, of course, for the type-generic macro).

Last month's article ("cybersecurity") described the C11 Annexes K and L (bounds-checking interfaces and analyzability), both of which are not part of C++11. However, the Annex K library is widely available from C++ environments that provide additional libraries for use by the C++ application; certainly Microsoft's Visual Studio has been in this category for several years.

The first article in this series ("threading and atomics") covers several compatibility challenges. I've been told, by people whose expertise I respect, that when you incorporate multi-threading into your design, you should put all the threading control into the C++ components, or alternatively put all the threading control into the C components. A mix-and-match approach to the threading controls raises serious issues. And if the app contains both C and C++ components, there are other (unrelated) good reasons to make the `main` program to be a C++ program (initialization of statics, setting up the exception-handling mechanism, etc.). A further consideration is that some compilation environments are fairly close to the C++11 standard for multi-threading, where the C11 implementation may be lagging behind somewhat.

However, with regard to accessing atomic data, the compatibility situation is somewhat more favorable. Depending upon your compiler-and-library environment, you should fairly soon be able to use the named `atomic_*` basic atomic types in both C11 and C++11. Somewhat more ambitiously, you could use the `_Atomic(T)` syntax ("_Atomic parenthesis"), and if your C++11 environment doesn't already support this syntax, you could create your own compatibility header containing this definition:

```
#define   _Atomic(T)   atomic<T>
```

However, I'm uncertain as to when, or whether, any particular C11 implementation will support the full type-qualifier syntax of `_Atomic T` ("_Atomic space").

I'm just about ready to summarize the compatibility situation, but first some discussion about conformance testing and optional subsets. From the very beginning of the C standards process (going back to the 1980's), the marketplace for C compilers has been attentive to, and concerned about, the various agencies and processes for formal certification of conformance to the ISO/IEC C standard. Even now in the third decade of the successive C standards, a formal certification process is provided by The Open Group for POSIX operating systems, which requires a C compiler. By contrast, there has never yet been a formal certification for C++ compilers; it's probably true that no commercial C++ compiler has ever fully conformed to any ISO/IEC C++ standard.

Given this greater attention to conformance details, it's worth noting that the C11 standard has identified eight different portions of the standard as optional: variable length arrays (VLAs), complex types, IEC 60559 ("IEEE") floating-point (Annex F), IEEE complex (Annex G), bounds-checking interfaces (Annex K), analyzability (Annex L), multithreading, and atomic. A compiler could conform to C11 and provide all of these, or none of these, or some portion of these. The minimum requirement of C11 is, of course, to provide none of these. The message from the marketplace, or so it seems to me, is that C99 imposed a number of requirements that were needed only by small portions of the marketplace (what some on the committee referred to as "boutique" markets).

I'll conclude with a table that describes the C/C++ compatibility issues of these eight optional C11 features:

C11 Features	C++ Compatibility Issues
1. Variable length arrays (VLAs)	Will never be part of a C++ standard
2. Complex types <complex.h>	User program can be compatible, with effort
3. IEC 60559 floating-point arithmetic (Annex F)	Could be provided by a C++ environment, but not required
4. IEC 60559 complex arithmetic (Annex G)	Could be provided by a C++ environment, but not required
5. Bounds-checking interfaces (Annex K)	Could be provided by a C++ environment, but not required
6. Analyzability (Annex L)	Could be provided by a C++ environment, but not required
7. Multithreading <threads.h>	Choose C11 or C++11; don't mix
8. Atomic primitives and types <stdatomic.h>	User program can be compatible, with effort

---

Dr. Thomas Plum is Vice President of Technology and Engineering at Plum Hall, Inc., and is a member of the C and C++ committees which developed C11 and C++11. He can be reached at tplum@plumhall.com. The author gratefully acknowledges helpful suggestions from Pete Becker (the project editor of the 2011 C++ standard), Robert Seacord of CERT, and David Keaton (chairman of the US committee for C language).