Features

Enum — A Misnomer
Daniel James

How To Pick Your Programming Language
Pete Goodliffe

Introduction to std.datetime in D
Jonathan Davis

Intelligent Software
Simon Salter

Memories of Learning C
Anthony Williams

Review of Effective C# Item 15
Paul Grenyer

Regulars

Baron Muncharris
Code Critique
Desert Island Books
Inspirational (P)articles
Book Reviews
Language Barrier

I reckon I’m pretty safe in the assumption that most readers of this magazine already knew the name and reputation of Dennis Ritchie, who died in October 2011. He is probably best known as co-inventor of C and co-author of *The C Programming Language*, a text widely regarded as a pinnacle in technical authorship for its clarity and accessibility. One of the reasons that ‘K&R’ (as it’s affectionately known by many) is so short is that its subject is, in concept, very simple; the C language imposes only a few reserved words on the programmer. And yet, for all its apparent simplicity, it is expressive enough to model hugely complex ideas – the Unix operating system being a prime example. ANSI C89 added 5 keywords to the list defined in ‘K&R’ C, to make 32 keywords, increased by another 5 for the 1999 standard.

This was brought to mind recently at a talk given for the ACCU London event by Jon Skeet. The topic of the talk (which was excellent, by the way!) was the new asynchronous programming features of the up-coming C# 5. These features look most interesting, and have been given a great deal of thought by the designers to make them easy to use correctly, and be able to express complex ideas as simply as possible. With one small wart – the new features introduce (at least) two new keywords to C# – a language that already requires the concept of ‘Contextual Keywords’ to manage its reserved list.

I recall Bjarne Stroustrup at the outset of the process which has recently resulted in C++11 being ratified by ISO, exhorting the committee to prefer new libraries instead of language features where feasible. C++11 has 83 keywords, an increase of 10 over C++98.

C# has 98 keywords as at version 4.0 (including contextual keywords), C# 5 will have at least 100.

I wonder if we will ever again see a language that can be entirely described in a book as truly succinct as K&R.
ADVERTISE WITH US
The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS
Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from C Vu without written permission from the copyright holder.

FEATURES

3 How to Pick Your Programming Language
Pete Goodliffe helps us make an important decision.

4 Introduction to std.datetime in D
Jonathan M Davis describes his contribution to Phobos, the D Standard Lib.

10 A Game of Lucky Sevens
Baron Muncharris invites us to solve a new puzzle.

11 On a Game of Pathfinding
Our student analyses the Baron’s last challenge.

12 Review of Effective C# Item 15: Utilize using and try-finally for Resource Clean-up
Paul Grenyer gets to grips with the Dispose pattern.

14 Enum – a Misnomer
Daniel James exposes enum as unsuitable for enumeration.

17 Intelligent Software Design
Simon Salter receives divine inspiration for a satirical view of the design process.

WRITE FOR C VU
Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

DIALOGUE

18 C++ Standards Report 11
Roger Orr reports on the new C++ standard.

18 Inspirational (P)articles
Frances Love introduces Paul Grenyer.

19 Desert Island Books
Roger Orr shares the contents of his suitcase.

20 Memories of Learning C
Anthony Williams recalls his first experiences of C.

21 Code Critique Competition
Competition 72 and the answers to 71.

REGULARS

27 Book Reviews
The latest roundup of book reviews.

28 ACCU Members Zone
Reports and membership news.

SUBMISSION DATES

C Vu 23.6: 1st December 2011
C Vu 24.1: 1st February 2012

Overload 107: 1st January 2012
Overload 108: 1st March 2012

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.
How To Pick Your Programming Language

Pete Goodliffe helps us make an important decision.

START HERE

You just died. As your life flashes before you, you see...

A closing brace

Objects! They’re everywhere. They are...

D

Salvation

A richly fulfilled person

Smalltalk

Really? Wow!

Yeah!

The bits I remember, anyway

No. As if

Nails!

The on switch

For people who don’t understand function pointers

A runaway train comes steaming closer. Are you...

In a carriage

Strapped to the railway line

In a

carriage

Assembly

What’s the OO way to deal with wealth?

Inheritance

What are you looking for?

Objective-C

Like you had a choice, anyway.

Mac

Who cares?

Tk/TCL

Java

Deep loathing

No.

C

C++

C#

Do you have any shame?

Like you had a choice, anyway.

You only live once

Sounds like fun. Where’s the parachute?

Sure. As if.

I catch programming mistakes with...

#define BEGIN {

Acceptable

Give yourself a slap

Criminal offense

Fortran

Animal, vegetable, or mineral?

Animal

Mineral

Python

Ruby

Z

What mistakes?

Unit tests

A thorough proof

The QA department

The compiler

LISP

(punctuation)

Delegation

to someone else

Delight

The compiler

\texttt{BEGIN} 

Professionalism in Programming #71

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn’t wear shoes. Pete can be contacted at pete@goodliffe.net

NOV 2011 | CVU | 3
Introduction to std.datetime in D

Jonathan M Davis describes his contribution to Phobos, the D Standard Lib.

In dmd 2.052, the module std.datetime was introduced. It will be replacing std.date entirely. As such, std.date is currently scheduled for deprecation. At a later date it will be deprecated (at which point, you’ll have to compile with -d for it to work rather than simply having the compiler complain when you use it), and eventually it will be fully removed from Phobos. What this means is that all new code should be written to use std.datetime and that any code which currently uses std.date is going to need to be refactored to use std.datetime (unless you want to copy std.date to your own code and continue to use it as a non-Phobos module). This article attempts to familiarize you with std.datetime as well as give some advice on how to migrate code from std.date to std.datetime for those who have been using std.date.

std.date is essentially a C-based solution for dates and times. It uses d_time to hold time where d_time is a 64-bit integral value holding the number of milliseconds which have passed since midnight, January 1st 1970 A.D. in UTC. C, on the other hand, uses time_t to hold time where time_t is an integral value holding the number of seconds which have passed since midnight, January 1st, 1970 A.D. in UTC. Its size varies from architecture to architecture (typically 32 bits on a 32-bit machine and 64 bits on a 64-bit machine, but it varies with the OS and compiler). The exact set of functions that std.date provides for using with d_time aren’t the same as what C provides for using with time_t, but their representation of time is virtually the same.

std.datetime, on the other hand, is very much an object-oriented solution, and it’s not C-based at all. Rather, its API is based on Boost’s types for handling dates and times (though they’re far from identical). So, it’s a bit of a paradigm shift to move from std.date to std.datetime. Don’t expect much to be the same between the two. However, std.datetime is not plagued with the same bugs that std.date is plagued with (std.date being quite buggy in general), and it provides much more functionality than std.date does. So, in the long run at least, dealing with std.datetime should be much more pleasant — though migration is likely to present a bit of a hurdle in the short term.

Basic Concepts of std.datetime

Most things in std.datetime are based on and/or use three concepts:

- **Duration**: A duration of time with units. E.g. 4 days or 72 seconds.
- **Time Point**: A specific point in time. E.g. 21:00 or May 7th, 2013.

**Durations**

The duration types can actually be found in core.time. They are Duration and TickDuration. TickDuration is intended for precision timing and is used primarily with StopWatch and the benchmarking functions found in std.datetime — such as benchmark

| auto duration = TimeOfDay(17, 2) - TimeOfDay(6, 7); |
| assert(duration == d!"hours"(10) + d!"minutes"(55)); |
| assert(duration.hours == 10); |
| assert(duration.minutes == 55); |
| assert(duration.total!"hours"() == 10); |
| assert(duration.total!"minutes"() == 655); |
| assert(duration.total!"hnsecs"() == 393_000_000_000); |

—and you’re unlikely to use it outside of using them. Duration, on the other hand, you’re likely to use quite a bit.

Duration holds its time internally as hecto-nanoseconds (100 ns), so that’s its maximum precision. It has property functions for both returning the duration of time truncated to a particular unit (such as the days and seconds property functions) as well as a function for returning the total number of a particular unit in that Duration (the total property function).

Generally, a Duration is created in one of two ways: by subtracting two time points or with the core.time.dur function. So, for instance, if you subtracted a time point which represented 17:02 from a time point which represented 6:07, you’d get a Duration which represented 10 hours and 55 minutes. Or, if you wanted to create a Duration directly, then you’d use the dur function and make a call like dur!"hours"(17) or dur!"seconds"(234) (see Listing 1).

Like any number, Durations can be added together or subtracted from. However, unlike a naked number, they have units associated with them and will handle the appropriate conversions. Also, it should be noted that the various functions in drumtime and Phobos which take a duration of time take an actual Duration rather than a naked number (most currently take both, though the versions which take a naked number are going to be deprecated). For instance, core.thread.sleep takes a Duration, as does std.concurrency.receiveTimeout. So, durations are used outside of just interacting with core.time and std.datetime.

One particular thing to note here is how both dur and total take a string representing the units of time to be used. This is an idiom used throughout core.time and std.datetime. The possible units are "years", "months", "weeks", "days", "hours", "minutes", "seconds", "hnsecs", "usecs", "hnsecs", and "nsecs". It should be noted however that very few functions take "nsecs", because nothing in std.datetime, and very little in core.time, has precision greater than hnsecs (100 ns). Also, a number of functions (such as core.time.dur) do not take "years" or "months", because it is not possible to convert between years or months and smaller units without a specific date. So, while you can add a Duration to a time point, if you want to add years or months to one, you must use a separate function (such as add) to do that — and those will take "years" and "months".

**Time Points**

std.datetime has 4 types which represent time points.

- **Date**
- **TimeOfDay**
- **DateTime**
- **SysTime**

---

Jonathan M Davis is a bit of a programming language Geek and loves programming languages. Professionally, he has programmed in both C++ and Java. In his free time, he’s one of the developers for D’s standard library, Phobos. He can be contacted at jmdavisProg@gmx.com.
A `Date` represents a date and holds its year, month, and day as separate values internally. A `TimeOfDay` represents a time of day, 00:00:00 - 23:59:59, and holds its hour, minute, and second as separate values internally. A `DateTime` represents a date and time and holds its values as a `Date` and `TimeOfDay` internally. None of these types have any concept of time zone. They represent generic dates and/or times and are best-suited for cases where you need a date and/or time but don’t care about time zone. Also, because they hold their values separated internally, those values don’t have to be calculated every time that you ask for them. (See Listing 2.)

A `SysTime`, however, is an entirely different beast. It represents a date and time – similar to `DateTime` – but it goes to hnssec precision instead of only second precision, and it incorporates the concept of time zone. Its time is held internally as a 64-bit integral value which holds the number of hnssecs which have passed since midnight, January 1st, 1 A.D. in UTC. It also has a `TimeZone` object which it uses to polymorphically adjust its UTC value to the appropriate time zone when querying for values such as its year or hour.

`SysTime` is the type which is used to interface with the system’s clock. When you ask for the current time, you get a `SysTime`. And because it always holds its internal value in UTC, it never has problems with DST or time zone changes. It has most of the functions that `DateTime` has as well as a number of functions specific to it. It can be cast to the other 3 time types as well as be constructed from them, but you do risk problems with DST when creating a `SysTime` from the other 3 time points unless you specifically create the `SysTime` with a `TimeZone` which doesn’t have DST (such as `std.datetime.UTC`), since when a time zone has DST, one hour of the year does not exist, and another exists twice. You can also convert to and from unix time, which is what you’re dealing with in C with `time_t`.

The one other related type which I should mention at this point is `core.time.FracSec`. It holds fractional seconds, and it is what you get from a `Duration` or `SysTime` when you specifically ask for the fractional portion of the time. (Listing 3)

### Time Intervals

`std.datetime.Time` has nothing to correspond to time intervals, so we don’t go over them in great detail. Essentially, they’re constructed from either two time points or a time point and a duration. `Interval` is a finite time interval starting at a specific time point and going to positive infinity, and `NegInfInterval` is an infinite time interval starting at negative infinity and going to a specific time point. They have various operations for dealing with intersections and the like. It is also possible to create ranges over them if you want to operate on a range of time points. Take a look at the documentation [1] for more details.

### Interfacing with C

Hopefully, you can do everything that you need to do using the types in `core.time` and `std.datetime`, but if you do need to interface with C code, then you can. C’s `time_t` uses ‘unix time’ (seconds since midnight, January 1st, 1970 A.D. in UTC), whereas `SysTime` uses what it calls ‘std time’ (hnssecs since midnight January 1st, 1 A.D. in UTC). Translating between the two is fairly straightforward. To get a `time_t` from a `SysTime`, simply call `toUnixTime` on the `SysTime`. To convert the other way around, you first need to convert a `time_t` to std time, then pass that value to `SysTime`’s constructor. And if you ever simply need a `SysTime`’s std time for any reason, then use its `stdtime` property.

Hecto-nanoseconds were chosen as the internal representation of `Duration` and `SysTime`, because that is the highest precision that you can use with a 64-bit integer and still cover a reasonable amount of time (`SysTime` covers from around 29,000 B.C. to around 29,000 A.D.). It also happens to be the same internal representation that C# uses, so if you need to interface with C# for any reason, converting between its representation of time and `std.datetime`’s representation is extremely easy, since no conversion is necessary. C#’s `DateTime` uses both the same units and epoch for its internal representation (which it calls `Ticks`) as `SysTime`, though unlike `SysTime`, it doesn’t work with negative values (which would be B.C.) and doesn’t go past the end of 9,999 A.D. Most programs are unlikely to care about values outside that range however. Regardless, hnssecs make the most sense for `std.datetime`, which tries to have the highest precision that it reasonably can, so that’s why they were picked.

### Recommendations on Using std.datetime

Whether `Date`, `TimeOfDay`, `DateTime`, or `SysTime` is more appropriate in a particular situation depends very much on that situation. `Date`, `TimeOfDay`, and `DateTime` generally make the most sense when you’re dealing with generic dates and times that have nothing to do with time zones, but if you’re dealing with time zones at all or are dealing with anything which needs to worry about DST, you should use `SysTime`. Because it keeps its time internally in UTC, it avoids problems with DST. And while it does have a time zone component, it defaults to `std.datetime.LocalTime` (which is the time zone type for the local time of the system), so you don’t generally have to deal directly with time zones if you don’t want to.

If you do want to deal with time zones, then the time zone types in `std.datetime` are `LocalTime`, `UTC`, `SimpleTimeZone`, `PosixTimeZone`, and `WindowsTimeZone` – or if for some reason, they don’t do what you need, you can always create your own time zone class derived from `TimeZone`. That’s unlikely to be necessary, however (and if you think that you have come up with such a class which would be generally useful, please bring it up in the digitalmars.D newsgroup, since if it’s truly generally useful, we may want some version of it in `std.datetime`). Read their documentation for more details. Most applications shouldn’t have to worry about time zones though, beyond perhaps using `UTC` instead of `LocalTime` in some cases.

When it comes to saving a time point to disk or a database or something similar, I would generally recommend using the `toISOString` or `toISOStringExtString` functions, since both are standard ISO formats for date-time strings (`toISOStringExtString` is likely better in the general case, since it’s more humanly readable, but they’re both standard). You can then use `fromISOString` or `fromISOStringExtString` to recreate the appropriate time type later. `toString` uses the `toSimpleString`
One area with saving times as strings which gets a bit awkward is time zones. The time zone is included in the string as part of the ISO standard, but all it contains is the total offset from UTC at that particular date and time, so you can’t generally use an ISO string (extended or otherwise) to get the exact time zone which the `SysTime` originally had. Rather, it will be restored with a `SimpleTimeZone` with the given offset from UTC (except in the case of UTC, where it can restore UTC). On the other hand, if you’re using `LocalTime`, then the time zone is not part of the string (per the ISO standard), and restoring the `SysTime` will restore it to whatever the current time zone is on the box, regardless of what the original time zone was. However, because in all cases, except for `LocalTime`, the UTC offset is included in the string, it is generally possible to get the exact UTC time that the `SysTime` was for. But you can’t usually restore the original time zone from just the ISO string. (Listing 5)

To summarize, UTC and `SimpleTimeZone` can be restored exactly using an ISO or ISO extended string. However, none of the other `TimeZone` can be `LocalTime` is restored with the same date and time but in the local time zone of the computer it’s restored on, so its std time may differ. Other time zones end up with the restored `SysTime` having the same std time as the original, but the new time zone is a `SimpleTimeZone` with the same total UTC offset which the original time zone had at the given std time, but you don’t get the original time zone back. That works just fine if you don’t ever need to change the value of that `SysTime` or need to know the name of the original time zone, but it is inadequate if you need to do either of those, since the rules for the new time zone won’t match those of the original.

So, if you don’t care about the time zone or if the restored `SysTime` has the same std time as the original, then `LocalTime` is fine. However, if you want the std time to be consistent, then avoid `LocalTime`. In most cases, I’d advise simply using UTC as the time zone when saving the time. And if you want to restore the time zone such that it’s the same time zone with the same rules as it was prior to saving the time, then you’re going to need to save that information yourself. With a `WindowsTimeZone` or a `PosixTimeZone`, all you have to do is save the time zone’s name (which for them is the TZ database name and the Windows time zone name of that time zone respectively). That can be used to restore the time zone. If it’s a `SimpleTimeZone` or UTC, then you don’t have to do anything, because the ISO string will be enough. If you’re using `LocalTime`, however, you’re in a bit of a bind.

The restored time zone will be `LocalTime`, so if you want it to be whatever the local time of the computer you’re doing the restoring on is, then you’re fine. But if you want to be able to have the same actual time zone restored regardless of the local time of the computer restoring the time, you’ll need to figure out what the time zone’s TZ database name or Windows time zone name is on the original computer so that you can use it to get its corresponding `WindowsTimeZone`, odds are that restoring the time with the correct std time value and correct UTC offset will be enough (and if it’s not, you can always save the time zone’s name to restore the correct `WindowsTimeZone`). It’s just `LocalTime` that has the problem. However, if a function to accurately determine the TZ database name of the local time zone on Posix systems is ever devised, then it will be added to `std.datetime`.

The other, more compact, option for saving a `SysTime` is to just save its std time as a 64-bit integer. It’s not really humanly readable like an ISO or ISO extended string would be, but it takes up less space if saved as an actual number rather than a string. However, you do then have to worry about the time zone yourself entirely if you wish to be able to restore it. But if you save the current UTC offset (meaning the UTC offset with the DST offset applied – such as an ISO string would include), that would be enough to correctly give what the time would have been in the original time zone even if you can’t restore that time zone.

Well, that’s probably more than enough on time zones. In most cases, you shouldn’t need to care about them (`SysTime` is designed to make it so that you shouldn’t have to worry about them if you don’t want to), but `std.datetime` strives to give the best tools possible for handling time zones when you actually want to. Regardless, by far the biggest gain that `SysTime` gives you is that its internal time is always in UTC, so regardless of whether you try and do anything with time zones explicitly, you won’t have any problems with DST changes when dealing with `SysTime`.

One last suggestion on using `SysTime` would be that if you need to query it for more than one of its properties (e.g. day or hour), or if you need to do it many times in a row, and the `SysTime` isn’t going to change, then you should probably cast it to another time point type (probably

```
$a = new DateTime("2011-06-26", new DateTimeZone("America/New_York"));
$b = new DateTime("2011-06-26", new DateTimeZone("America/Los_Angeles"));
$a->setTimezone(new DateTimeZone("America/Sao_Paulo"));
$b->setTimezone(new DateTimeZone("America/Argentina/Buenos_Aires"));
```

But casts are expensive, so if you’re only going to use a single property, then I’d suggest sticking with just a `LocalTime` object. You can get away with saving time zone names with a `WindowsTimeZone` and doing restores with `DateTime::createFromFormat` if you’re using `WindowsTimezone`, but that’s a bit of a kludge.

That’s a pretty long list, so I’ll leave the other features to the next issue.
are scheduled to be deprecated and have replacement functions which take
what

converting your code from using
documentation), but the big question for many is how best to handle

in
to deal with conversions or just doing them yourself, since the functionality
for dealing with the time from the system’s clock, so
(Systems). Those changes are quite straightforward and not particularly disruptive. Of greater concern are the
formats that times are printed or saved in and how time zones are dealt with.
If you were saving the integral d_time value anywhere, then you’re either
going to have to switch to saving a value that SysTime would use as
discussed previously (such as its std time or its ISO string), or you’re going
to have to be converting between d_time and SysTime.
At present, the functions std.datetime.SysTimeToDTime and
std.datetime.DTimeToSysTime will do those conversions for you.
So, converting between the two formats is easy. However, because
d_time is going away, those functions will be going away. That means
that you either need to refactor your code so that those functions aren’t
necessary, or you need to copy them to your own code to continue to use them.
As for formatted strings, std.datetime currently only supports ISO
strings, ISO extended strings, and Boost’s simple string. Eventually, it
should have functions for custom strings, but a well-designed function
for creating custom strings based on format strings is not easy to design, and
it hasn’t been done for std.datetime yet (it’s on my todo list, but it
could be a while before I get to it). So, in general, you’re either going
to have to switch to using one of the string formats that std.datetime
supports, or you’re going to have to generate and parse the string
format that you want yourself. In some cases, you should be able to adjust
the string that core.datetime.datetime gives you, and in others, you may
be able to use
toISOExtString
and adjust what it gives you, but there’s
a decent chance that you’re going to have to just create and parse the strings
yourself using the various properties on SysTime. One major difference
between the string functions in std.date and those in std.datetime
to note is that unlike std.date, aside from Boost’s simple strings,
nothing in std.datetime prints the names of months or weekdays,
because that poses a localization issue. So, unless you’re using
to get those values, you’re going to have to create the names yourself.
Now, if you were doing anything with time zones with std.date, odds
are that you were doing all of those conversions yourself (since that’s one
of the areas where std.date is buggy). That being the case, you probably
have the offset from UTC and the offset adjustment for DST for whatever
time zone you’re dealing with. What is likely the best way to handle that
is to create
SimpleTimeZone
using those values. Simply calculate the
total UTC offset (so add in the DST offset if it applies for the date in
question) in minutes and create a
SimpleTimeZone
with that. Note that
std.datetime treats west of UTC as negative (for some reason, some
systems – particularly Posix stuff – use a positive offset from UTC when
west of UTC, in spite of the fact that when talking about time zones,
negative is always used for west of UTC, and that’s what the ISO standard
strings do). So, you may have to adjust your values accordingly.
Regardless, be very careful to make sure that you understand what the

SysTime
hold its time internally in UTC in a manner similar to
within the system’s clock, so
DTime
is generally
what
d_time
should be replaced with. Functions in Phobos which previously took or returned a
d_time
now take or return a
SysTime
or are scheduled to be deprecated and have replacement functions which take
or return a
SysTime.
Generally, in the case where a function took a
d_time
that function is now overloaded with a version which takes a
SysTime, but in cases where a function could not be overloaded (such as
when it simply returned a
d_time), a new function has been added to
replace the old one (so as to avoid breaking existing code). The module
that this impacts the most is
std.file.
For an example, see Listing 8.
With all such functions, it’s simply a matter of changing the type of the
argument that you’re passing to the function or assigning its return value
to and possibly changing the function name that you’re calling so that it’s
the version that returns a
SysTime. Those changes are quite
straightforward and not particularly disruptive. Of greater concern are the
formats that times are printed or saved in and how time zones are dealt with.

//These are the same offsets as
// America/Los_Angeles.
auto utcoffset = -8 * 60;
auto dstOffset = 60;
immutable tzWithDST =
new SimpleTimeZone(utcOffset + dstOffset);
immutable tzWithoutDST =
new SimpleTimeZone(utcOffset);
values you’ve been using represent in units of time and whether you need to be adding or subtracting them to convert them to what SimpleTimeZone expects for its offset from UTC: the minutes to add to the time in UTC to get the time in the target time zone. (Listing 9)

The last thing that I have to note is some differences in numerical values between std.date and std.datetime. std.date.Date’s weekday property gives Sunday a value of 1, but std.date.weekDay gives Sunday a value of 0. std.datetime.DayOfWeek gives Sunday a value of 0. So, depending on which part of std.date you’re dealing with it, it may or may not match what std.datetime is doing for the numerical values of weekdays. Months have a similar problem. std.date.Date’s month property gives January a value of 1 – which matches what std.datetime.Month does – but std.date.monthFromTime gives January a value of 0. So, just as with the days of the week, you have to be careful with the numerical values of the months. Whether std.datetime matches what std.date is doing depends on which part of std.date you’re using. And as you’ll notice, it’s not even consistent as to whether std.date.Date or the free function in std.date is the one which matches std.datetime. So, you should be very careful when converting code which uses numerical values for either the days of the week or the months of the year.

std.date symbols and their std.datetime counterparts
A table giving std.date symbols and their std.datetime counterparts can be found below.

<table>
<thead>
<tr>
<th>std.date</th>
<th>std.datetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_time</td>
<td>The closest would be SysTime</td>
</tr>
<tr>
<td>d_time_nan</td>
<td>There is no equivalent. SysTime.init, which has a null TimeZone object, would be the closest, but once CTFE (Compile-Time Function Execution) advances to the point that you can new up class objects with it, SysTime.init’s timezone will be LocalTime, so don’t rely on SysTime.init being invalid. std.datetime in general tries to avoid having any invalid states for any of its types. It’s intended that creating such values be impossible</td>
</tr>
<tr>
<td>Date</td>
<td>SysTime.dayOfWeek – but note that the values are off by 1.</td>
</tr>
<tr>
<td>Date.year</td>
<td>SysTime.year</td>
</tr>
<tr>
<td>Date.month</td>
<td>SysTime.month</td>
</tr>
<tr>
<td>Date.day</td>
<td>SysTime.day</td>
</tr>
<tr>
<td>Date.hour</td>
<td>SysTime.hour</td>
</tr>
<tr>
<td>Date.minute</td>
<td>SysTime.minute</td>
</tr>
<tr>
<td>Date.second</td>
<td>SysTime.second</td>
</tr>
<tr>
<td>Date.ms</td>
<td>SysTime.fracSec.msecs</td>
</tr>
<tr>
<td>Date.weekday</td>
<td>SysTime.dayOfWeek – but note that the values are off by 1.</td>
</tr>
<tr>
<td>Date.tzcorrection</td>
<td>immutable tz = sysTime.timezone; auto diff = tz.utcToTZ(sysTime.stdTime) - sysTime.stdTime; auto tzcorrection = convert!(&quot;hnsecs&quot;, &quot;minutes&quot;)(diff); However, it looks like tzcorrection is broken, so you’re probably not using it in your code anyway.</td>
</tr>
<tr>
<td>Date.parse</td>
<td>SysTime.fromISOString, SysTime.fromISOExtString, and SysTime.fromSimpleString, but the formats of the strings differ from what std.date.Date.parse accepts.</td>
</tr>
<tr>
<td>ticksPerSecond</td>
<td>There is no equivalent. It’s only relevant to d_time.</td>
</tr>
<tr>
<td>toISO8601YearWeek</td>
<td>SysTime.isoWeek</td>
</tr>
<tr>
<td>hourFromTime</td>
<td>SysTime.hour</td>
</tr>
<tr>
<td>minFromTime</td>
<td>SysTime.minute</td>
</tr>
<tr>
<td>secFromTime</td>
<td>SysTime.second</td>
</tr>
<tr>
<td>daysInYear</td>
<td>sysTime.isLeapYear ? 366 : 365</td>
</tr>
<tr>
<td>dayFromYear</td>
<td>(sysTime - SysTime(Date(1970, 1, 1), UTC())).total!&quot;days&quot;()</td>
</tr>
<tr>
<td>yearFromTime</td>
<td>SysTime.year</td>
</tr>
<tr>
<td>inLeapYear</td>
<td>SysTime.isLeapYear</td>
</tr>
<tr>
<td>monthFromTime</td>
<td>SysTime.month – but note that the values are off by 1.</td>
</tr>
<tr>
<td>dateFromTime</td>
<td>SysTime.day</td>
</tr>
<tr>
<td>weekDay</td>
<td>SysTime.dayOfWeek</td>
</tr>
<tr>
<td>UTCtoLocalTime</td>
<td>SysTime.toUTC</td>
</tr>
<tr>
<td>dateFromNthWeekdayOfMonth</td>
<td>There is no equivalent. Listing 10 (below) is a possible implementation.</td>
</tr>
<tr>
<td>daysInMonth</td>
<td>SysTime.endOfMonthDay: Actually, this name is overly easy to confuse with endOfMonth – which returns a SysTime of the last day of the month. I will probably rename this to daysInMonth. But if I do, it won’t be until the next release (2.054), and this name will be around until it’s gone through the full deprecation cycle.</td>
</tr>
<tr>
<td>UTCtoString</td>
<td>There is no equivalent. You could probably parse and recombine core.std.time.ctime and SysTime.toISOExtString to create it though. However, this function appears to be fairly buggy in the first place, so odds are that your code isn’t using it anyway.</td>
</tr>
<tr>
<td>toUTCString</td>
<td>There is no equivalent. You could probably parse and recombine core.std.time.ctime and SysTime.toISOExtString to create it though.</td>
</tr>
</tbody>
</table>
### std.date vs std.datetime

<table>
<thead>
<tr>
<th>std.date</th>
<th>std.datetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>toDateString</td>
<td>There is no equivalent. You could probably parse and recombine <code>core.stdc.time.ctime</code> and <code>SysTime.toISOExtString</code> to create it though. However, this function appears to be fairly buggy in the first place, so odds are that your code isn’t using it anyway.</td>
</tr>
<tr>
<td>toStringDate</td>
<td>There is no equivalent. You could probably parse and recombine <code>core.stdc.time.ctime</code> and <code>SysTime.toISOExtString</code> to create it though. However, this function appears to be fairly buggy in the first place, so odds are that your code isn’t using it anyway.</td>
</tr>
<tr>
<td>parse.parse</td>
<td><code>SysTime.fromISOString</code>, <code>SysTime.fromISOExtString</code>, and <code>SysTime.fromSimpleString</code>, but the formats of the strings differ from what <code>std.date</code> parse accepts.</td>
</tr>
<tr>
<td>getUTCtime</td>
<td><code>Clock.curTime(UTC())</code> if you want the <code>SysTime</code> to have its time zone be UTC. More likely though, you’ll just use <code>Clock.curTime()</code>. Its internal time is in UTC regardless.</td>
</tr>
<tr>
<td>DosFileTime</td>
<td><code>DosFileTime</code></td>
</tr>
<tr>
<td>toDtime</td>
<td><code>DosFileTimeToSysTime</code></td>
</tr>
<tr>
<td>toDosFileTime</td>
<td><code>SysTimeToDosFileTime</code></td>
</tr>
<tr>
<td>benchmark</td>
<td><code>benchmark</code></td>
</tr>
</tbody>
</table>

Note that I’m not an expert on what does and doesn’t work in `std.date`, so while I have noted some of the functions that I know to be broken, just because a function isn’t labeled as broken in the above table does not mean that it works correctly. And any function which doesn’t work correctly is obviously not going to give the same results as the `std.datetime` equivalent, since it’s almost certain that the `std.datetime` version isn’t buggy, let alone buggy in the same way (if it is buggy, the bug is almost certainly going to be far more subtle than any bug in `std.date`, since `std.datetime` is quite thoroughly unit tested).

### Conclusion

Hopefully this article has improved your understanding of `std.datetime` and will get you well on your way to being able to migrate your code from `std.date` to `std.datetime`. If you have any further questions, please ask them on the digitalmars.D.learn newsgroup. And if there’s a major use case of `std.date` which is not easy to convert over to `std.datetime` which I missed in this article and you think should be in it, please feel free to bring it up on the digitalmars.D newsgroup, and if need be, I’ll update the online version of this article with the relevant information.

### References


Greetings Sir R-----! This evening’s chill wind might be forgiven some of its injurious assault upon me by delivering me some good company as I warm my bones. Come, shed your coat and join me in a glass of this rather delightful mulled cyder!

Might you be interested in a little sport whilst we recover?
Excellent!

This foul zephyr puts me in mind of the infantile conflict between King Oberon and Queen Titania that was in full force during my first visit to the faerie kingdom. I had arrived there quite by accident but fortunately my reputation was sufficient to earn me an invitation to dine at the King’s table. That the fare was sumptuous beyond the dreams of mortal man goes without saying, but the conflict between the King and his consort cast something of a shadow upon the evening.

I resolved that I might ease the tension, and improve the terrible weather that was its consequence, by arranging some diversion that might afford the royal couple an opportunity to resolve their dispute. I therefore made my way back to the Earthly realm and employed a troupe of actors to put on a play for the faerie court. To my very great shame they revealed themselves to be utterly inadequate upon the night; the lead actor, one Nick Bottom, faring so badly that he made a comedy of Pyramus and Thisbe.

My reconciliatory efforts having been so thoroughly unsuccessful I retired to a faerie tavern and whiled my hours away at a game most popular in that realm.

But I must tell you of its rules!

Here I have a pair of fresh decks running from Ace to King, each suit in its turn. I shall set one deck unmolested before me and the other thoroughly shuffled before you. I shall then take my top card and, if it be a seven keep it for my hand, if not discard it. You shall then do likewise and we shall continue taking turns in such manner until one of us holds a trick of four sevens. If it is my good fortune to have it, you shall give me a bounty of eleven coins. If, on the other hand, you prevail, I shall give you nine.

When I described the game to that odious student whose company I am cursed to endure, he became somewhat agitated regarding the mention of that oafish Mr Bottom, perhaps unsurprisingly given his own oafish nature.

But let us not put a tarnish upon this night with talk of that feeble-minded fellow; take another glass and consider your chances!

Listing 1 shows a C++ implementation of the game.

```cpp
void play()
{
    const char names[13][10] = {
        "an Ace\0", "a Deuce\0", "a three\0", "a four\0",
        "a five\0", "a six\0", "a seven\0",
        "an eight\0", "a nine\0", "a ten\0",
        "a Jack\0", "a Queen\0", "a King\0"};

    deck_type b_deck = deck();
    deck_type r_deck = deck();

    std::reverse(b_deck.begin(), b_deck.end());
    std::random_shuffle(r_deck.begin(), r_deck.end());

    size_t b_hand = 0;
    size_t r_hand = 0;

    deck_type::const_iterator b_card = b_deck.begin();
    deck_type::const_iterator r_card = r_deck.begin();

    while(b_hand!=4 && r_hand!=4)
    {
        if(*b_card==6) ++b_hand;
        std::cout << "The Baron drew " << names[*b_card];
        std::cout << " and has a hand of " << b_hand << " seven";
        if(b_hand!=1) std::cout << 's';
        else std::cout << '.';
        std::cout << std::endl;

        if(*r_card==6) ++r_hand;
        std::cout << "You drew " << names[*r_card];
        std::cout << " and have a hand of " << r_hand << " seven";
        if(r_hand!=1) std::cout << 's';
        else std::cout << '.';
        std::cout << std::endl;

        ++b_card;
        ++r_card;
    }

    if(b_hand==4) std::cout << "The Baron wins!" << std::endl;
    else std::cout << "You win!" << std::endl;
}
```

Baron Muncharris

In the service of the Russian military Baron Muncharris has travelled widely in this world, and many others for that matter, defending the honour and the interests of the Empress of Russia. He is renowned for his bravery, his scrupulous honesty and his fondness for a wager.
On a Game of Path Finding
Our student analyses the Baron’s last challenge.

You will recall that the Baron’s latest game consists of seeking to build a path of counters across a board chalked out upon the tavern’s hearth. The Baron was to place the first counter and strive to construct a path from the left to the right and Sir R---- was to follow and strive for one from top to bottom. They would continue to take turns in this fashion until a path had been forged, whereupon the victor would have a coin from the loser’s purse. Figure 1 shows a path for the Baron.

The first thing to note is that since each tile on the hearth is adjacent to six others, they are topologically identical to hexagons and the playing area is consequently equivalent to that of the board game Hex [1]. I said as much to the Baron, but I fear I may not have done so with sufficient clarity. The Hex board is shown in Figure 2.

Now, this game cannot end in a draw since the only means by which either player can block all of the other’s paths across the board is to make one of his own.

Noting this property, it was proven by one Mr Nash that the second player cannot force victory if the first player keeps his wits about him. He did so by first proposing that the second player had in mind a strategy that would ensure a win and then suggesting that the first player steal it. That is to say, he should place his first counter at random and thereafter use the second player’s strategy, or again at random if that strategy demands he places a counter on his random spare. The first player could thusly steal the second’s guaranteed win.

That mathematics abhors a contradiction is sufficient to demonstrate that the second player can have no such strategy, although it provides no hint as to how the first player might prevail.

I should therefore have advised Sir R---- not to take up the Baron’s challenge, unless of course, he believed he had the wits to overcome this disadvantage.

An interesting consequence of Mr Nash’s proof is that for any symmetric game in which the first player may elect to pass it is impossible for the second to guarantee success.

References
Review of Effective C# Item 15: Utilize using and try-finally for Resource Clean-up

Paul Grenyer gets to grips with the Dispose pattern.

The original Effective C++ series from Scott Meyers was a real revelation for C++ programmers. It grouped together many idioms from the wildly diverse and complex language and made them understandable. It identified many of the pitfalls and made them avoidable. It was a must read for every serious C++ programmer.

Since then all the major language seems to have an effective series. You would think this was a good idea, but most languages are not as wildly complex as C++, with fewer idioms and pitfalls. They’re still there, but the languages have been designed with the idioms in mind, and the introductory texts teach them, and with a lot of the pitfalls already avoided. Consequently most effective series for modern languages are smaller and contain a lot of patterns as well. For example, Effective Java starts off with the builder pattern. To my mind that belongs in a patterns book and it certainly should not be the first Java idiom described.

I am currently reading Effective C# by Bill Wagner. I’ve got as far as chapter 18 and so far it is full of good advice, but, in my opinion, is extremely poorly explained. Items 6 and 7 cover equality and GetHashCode. These are complex concepts in predominantly reference based languages, like C#, and after I’d finished reading the items I didn’t feel I understood them much better.

Items 12 to 20 cover resource management. This is a real passion of mine, so naturally I’m quite critical of what’s written here, as well as how it’s actually written. Luckily most of what’s written is sound, but part of Item 15 gives, in my opinion, some just plain bad advice. The following item, 16, is another exceptionally badly written item, all though the advice is sound, but I’ll leave that for another time.

**Item 15: Utilize using and try-finally for Resource Clean-up**

Resource management is probably the biggest Achilles heel of garbage collected languages. As such, it should probably be the subject of the first section of any effective series, but item 15 out of 50 isn’t too bad.

How and why resources need to be managed in C# is explained satisfactorily by the item, so I won’t go over it again. But, in my opinion, is extremely poorly explained. Items 6 and 7 cover equality and GetHashCode. These are complex concepts in predominantly reference based languages, like C#, and after I’d finished reading the items I didn’t feel I understood them much better.

Items 12 to 20 cover resource management. This is a real passion of mine, so naturally I’m quite critical of what’s written here, as well as how it’s actually written. Luckily most of what’s written is sound, but part of Item 15 gives, in my opinion, some just plain bad advice. The following item, 16, is another exceptionally badly written item, all though the advice is sound, but I’ll leave that for another time.

The author points out that you’ve effectively written the construct in Listing 3.

**Listing 1**

```csharp
public void ExecuteCommand(string connString, string commandString)
{
    using (var myConnection = new SqlConnection(connString))
    {
        using(var myCommand = new SqlCommand(commandString, myConnection))
        {
            myConnection.Open();
            myCommand.ExecuteNonQuery();
        }
    }
}
```

**Listing 2**

```csharp
public void ExecuteCommand(string connString, string commandString)
{
    using (var myConnection = new SqlConnection(connString))
    using(var myCommand = new SqlCommand(commandString, myConnection))
    {
        myConnection.Open();
        myCommand.ExecuteNonQuery();
    }
}
```

**Listing 3**

```csharp
public void ExecuteCommand(string connString, string commandString)
{
    SqlConnection myConnection = null;
    SqlCommand myCommand = null;
    try
    {
        myConnection = new SqlConnection(connString);
        try
        {
            myCommand = new SqlCommand(commandString, myConnection);
            myConnection.Open();
            myCommand.ExecuteNonQuery();
        }
        finally
        {
            if (myCommand != null)
                myCommand.Dispose();
        }
    }
    finally
    {
        if (myConnection != null)
            myConnection.Dispose();
    }
}
```

Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com
As he finds it ugly, when allocating multiple objects that implement `IDisposable`, he prefers to write his own `try/finally` blocks (Listing 4).

I have two problems with this. The first is that if the FINALLY FOR EACH RELEASE pattern, as described by Kevlin Henney in *Another Tale of Two Patterns* [1], is correctly implemented, the null checks, which are a terrible code smell and often the cause of bugs if they get forgotten, would be completely unnecessary (see Listing 5).

If the nested try blocks are a problem for you, another method can be introduced, shown in Listing 6.

However, the real problem is that you are only effectively implementing this construct. If you stick with the original nested using blocks, the compiler creates the construct for you and you don’t see it. Which means that it really doesn’t matter how ugly it might be and ditching the using blocks and writing your own construct just creates the ugliness. Maybe the root of the author’s aesthetic objection is the nesting. Again, this is easily overcome by introducing another function, shown in Listing 7.

### References

[1] *Another Tale of Two Patterns*: http://www.two-sdg.demon.co.uk/curbralan/papers/AnotherTaleOfTwoPatterns.pdf
Enum – a Misnomer

Daniel James exposes enum as unsuitable for enumeration.

I read Matthew Wilson’s recent article on enums in C and C++ [1] with some interest. Wilson gives a good overview of the capabilities of enum types in these languages and some helpful tips on their use, especially on their use as enumerations.

As I read, though, I found myself asking why such an article was necessary at all. Isn’t an enum just about as simple as any datatype you can have … apart maybe from int? Shouldn’t all of this be obvious? Clearly it isn’t obvious – or there would be no need for such an article – and I would suggest that the reason we need to do articles like this is that the C++ enum type is not an enumeration type. That’s quite a claim: The C++ standard says that it isn’t one?

My dictionary [2] says that the verb ‘enumerate’ is connected with census taking, and means to ‘specify as in a list (a number of things) one by one’ or to ‘ascertain the number of’, and that ‘enumeration’ means ‘The action of enumerating’ or ‘A list, a catalogue’. The concept embodied in this definition is that of a fixed set of entities with distinct values that can be examined in turn.

In programming, the notion of an enumerated type in programming was first introduced by Niklaus Wirth in the Pascal programming language [3]. Enumerated types in Pascal have very similar semantics to subranges of integral types (which Pascal also supports) with the additional feature of enumerating’ or ‘A list, a catalogue’. The concept embodied in this definition is that of a fixed set of entities with distinct values that can be examined in turn.

Enum – a Misnomer

Daniel James exposes enum as unsuitable for enumeration.

In programming, the notion of an enumerated type in programming was first introduced by Niklaus Wirth in the Pascal programming language [3]. Enumerated types in Pascal have very similar semantics to subranges of integral types (which Pascal also supports) with the additional feature of defining a named constant for each value in the range. Pascal’s enums therefore implicitly represent ordered sequences of consecutive integer values. Pascal’s enums are never implicitly converted to or from integer types, but Pascal provides a full set of comparison operations between enum values and the FRED and SUCC functions which return the predecessor and successor values within the type, to enable iteration over the set of values. There are also ORD and VAL functions which are effectively casts to convert between enumeration values and their ordinal positions within the type. Listing 1 shows a Pascal enum definition for the same Fruit type as in Wilson’s C++ example.

I’d say that Pascal’s enum is pretty-much what a programmer with no preconceptions would expect an ‘enumeration’ to be, based on the dictionary definition. There is no way to ask Pascal for the number of values in the type, nor any way to ask for the minimum or the maximum value, but it is the nature of Pascal that you generally have to know these things to be able to use the type at all. Listing 2 shows a typical snippet of Pascal code that defines an enum representing days of the week, a subrange of that enum representing weekdays, and an array of integers that can be subscripted by weekday values. Listing 3 shows a procedure that initializes a value of the array type. Note that the original Pascal is so strongly typed that only an array of that exact type can be passed, so having to hard-code variables that only an array of that exact type can be passed, so having to hard-code Monday to Friday is not as limiting as it seems (though it could be a maintenance headache if the number of days in a week were to change). More modern Pascal dialects – especially those based on Borland’s Object Pascal (aka the Delphi language) – offer much richer functionality.

Pascal’s definition of enumerated types was the first, and it was refined and extended in other ‘Wirth’ languages that followed on from Pascal, and further still in Ada. All these languages add features, but the essential design of an enumerated type is that of an ordered set of named integer values is

Daniel James

Daniel James owns Sondata Limited, a one-man software consultancy based in Maidenhead. He has been learning programming languages for over 30 years, and has yet to find an entirely satisfying one so continues to use C++ for most of his work.

Listing 1

Listing 2

Listing 3

Listing 4

unchanged. Modula-2 [4] adds functionality to determine the highest and lowest enumerators of a type, so code can be more robust in the face of possible changes to the definition of the enum type. Listing 4 shows a trivial example using the new syntax, which will continue to work correctly even if new fruit types are added to the enum. Modula-2 also adds new INC and DEC operators, as shortcuts which modify a variable or an enum type (or any other scalar type) in situ in very much the same way as operator++ and operator-- do in C++, and are useful in writing loops other than the simple FOR to loop over all the values of a type.

Ada provides similar functionality, but in a slightly different way. In Ada the maximum and minimum values of an enumeration are given by the first and last attributes of the type, and values have succ and pred attributes that provide access to their successor and predecessor.
values, and an ‘image’ attribute to obtain their string representation. The string representation is taken from the enumerator name in the sourcecode (converted to upper case, as Ada sourcecode is case insensitive), and is only really useful for logging and debugging, because it makes no allowance for internationalization.

Listing 5 shows a simple Ada example based around the familiar Fruit type that prints out the names of the fruits and their ordinal positions in the type. To print the names I’ve used the generic Ada package Ada_Text_IO.enumeration_IO which in turn uses the ‘image’ attribute mentioned above. Note the use of the ‘width’ attribute of the type, which gives the length of the longest ‘image’ attribute that can be returned by the type, for aligning output. (Note also that it’s only useful if you’re using a fixed-pitch font!)

In Pascal and Modula-2 there’s no great magic going on, nor much runtime calculation to cause overheads. The languages are strongly enough typed that the actual type of any enum variable is known at compile time and the MAX and MIN ‘functions’ in Modula-2 are compile-time constants, while things like the SUCC and PRED functions are simply increments and decrements (perhaps with a range check, depending on the implementation and debug level). Converting an enum value to its enumerator name requires a lookup into a table of enumerator name strings, so that table does represent a space overhead in implementations that offer it.

In Ada there is an added complication that the programmer can use a representation clause to specify the actual bit pattern stored in memory for a given enum value (e.g. for access to memory-mapped hardware ports). This looks a bit C-like, but really it isn’t – the bit patterns are almost entirely invisible to the program once they have been set up. Listing 6 shows a variation of the program from Listing 5 that enumerates the items in a hypothetical Command type whose underlying values might be command bytes to be sent to a motor controller, where a byte with bit 5 set controls the on/off state of the motor and a byte with bit 6 set controls the motor speed, and in each case the value of bit 0 indicates on or off. Although the values used for storage have been changed, the program still behaves as if the enum had values 0..3, and the only way to access the actual stored bit pattern is by aliasing it with another type.

Listing 7 shows the result of running this program. Note that although we’ve set the hardware values corresponding to the different enumerators to non-contiguous values it is still possible to enumerate all the values in software, and their positions are still reported as 0..3. Clearly there must be some overhead in implementing the ‘pred’ and ‘succ’ attributes in this case, as they no longer represent a simple increment or decrement of the internal enum value.

So how does all this relate to the enum type in C and C++?

I don’t know when the concept of enumerated types was first implemented in C. They do not appear in the first edition of K&R [5] but are described in the second edition [6] ten years later, with a note that they had been implemented by some vendors some years previously. C’s implementation of enums differs quite noticeably from that of the Wirth languages in two ways:

- The values of the enumerators can be assigned by the programmer, and are not constrained to be ordered or consecutive.
- There is no mechanism for looping over all the legal values (and only the legal values) of the type.

The first difference makes the C enum more than an enumerated type, the second makes it less than one.

To a large extent the second difference is a consequence of the first: if the values are not ordered and sequential it’s not trivial to loop over them. C++ could have been designed to support the same sort of enumeration mechanism that we have seen with Ada enums that have representation clauses – but that would be an uncharacteristically high-level construct for C, and would require some runtime overhead, so the language doesn’t support it.

The C enum can be used for enumeration so long as the enumerator values are selected so as to be consecutive, and as C enum variables are simply integers they can be incremented and decremented using the same arithmetic operations as any other integers. Listing 8 shows a simple C program that illustrates that an enum can be used for enumeration in C about as easily as in Pascal as long as the requirement for the enumerators to be consecutive is observed.

Stroustrup says [7] that when designing C++ he had no particular desire to include an enumeration type beyond the need for compatibility with C, so the C++ enum was basically the same as the C one, but with added type safety. Type safety is good – Pascal has type safety between enums and other types – but making the enum type distinct from integer types makes integer operations, in particular increment and decrement, unavailable to enums. The simple solution is to cast between enum and int, as seen in Listing 9 which is the C++ version of Listing 8. Note that this simple example was written to show the awkwardness of the cast so I have not included any of the refinements that Wilson advocates, such as Unknown_Fruit and Maximum_Fruit_Value members in the enum.

Of course, it’s possible to define an operation that wraps up the increment operator for an enum type in a clean and typesafe way, we can even use a template to do it for all enum types. If we’re using the new C++11 scoped
enums we can also write generic code to enumerate over the values of any enum type that has been written to support enumeration as Listing 10 shows. Note that without scoped enums we could not have the same MaxValue enumerator name in more than one enum type, and there would be no generic way to write the end() function.

Scoped enums can be made to offer just about all the functionality that one might expect from a real enumeration type, but only as long as one follows the conventions that Wilson describes and is prepared to write a little code. There is no generic way to get the enumerator names as strings as Ada does – that would require language support that C++ just doesn’t offer – but I don’t see that as necessary functionality for an enumerated type, as one would normally want more functionality than the compiler-generated names would offer (internationalization, different forms of name for singular and plural (one cherry but two cherries), etc.).

I started by saying that the C++ enum was both less and more than an enumerated type. So far I’ve talked about the less, but I should say a few words about the more as well. Because a C or C++ enum isn’t constrained to contain a sequence of values starting from zero, as is a Pascal enum, it can be used to define and name any group of compile-time constants. Because an enum is a proper type, unlike a preprocessor symbol, it follows the language scoping rules and its visibility can be limited to a given struct or function (in C) or class or namespace (in C++) – these benefits are well known. These things are also true of const int values, of course, but the grouping of related names constants into an enum serves to document the relationship between them and aids code readability. Also, older compilers don’t allow initialization of const int values within class definitions, so it may (still) be necessary to use enum values to define compile-time constants within the class.

```cpp
template <typename T>
typename boost::enable_if< boost::is_enum<T>, T>::type
operator++( T & t )
{
    return t = static_cast<T>(
        static_cast<int>(t)+1 );
}
```
Intelligent Software Design

Simon Salter receives divine inspiration for a satirical view of the design process.

This was revealed to me the other morning. I was walking the dog and found myself luminescent in a beam of sunlight which penetrated the stormy clouds and infused my inner being with a deep, wise and irrefutably true revelation concerning software architecture. A new paradigm for creating absolutely correct and inimitable software.

Intelligent software design is the one true path for the righteous programmer.

Project managers struggling with devilish delivery schedules will be blessed in the divine knowledge that intelligent software design is guaranteed to take only six days to create everything. On the seventh day you can have a rest and contemplate your creation.

Unlike some blasphemous design methodologies objects are not mutable in any way. They cannot evolve and iteration is a heresy. Objects always appear fully formed and absolutely correct. They may later get corrupted by memory overwrites, also known as the greed of man, but this can be avoided by using the dogmatic interface.

There is only one true singleton which is of irreducible complexity. You may attempt to make others (the idol pattern) but your program will crash, the computer will burn and there may be a faint smell of sulphur.

Praying during acceptance testing is not only allowed but actively encouraged. The debugging prayer is particularly easy to learn and always works. Guaranteed. Never fails.

Asynchronous messages may originate from any sufficiently old object and these can always be interpreted in a way that is convenient.

Counting is done two by two, a technique which can handle all things bright and beautiful up to numbers of truly biblical proportions.

The useful disciple logic function supports arbitrary repeat counts. You can call this as many times as needed to ensure that it must be true.

Expressions which use a canonical form are prefixed with St.

Objects which consist entirely of pure, virtuous functions are marked with a halo.

Trinity value types (true, false, spiritual) can be used to map real entities to virtual functions.

Passing by reference is achieved using the chapter, verse, line notation.

Transcendental functions are used to access higher level objects.

Mistakes never need to be fixed and the confessed exception handler ensures program execution can continue unhindered by any previous activity or state.

Collections are managed through the powerful Chalice class, considered by many to be the holy grail of STL containers.

An acolyte function can be used where an over developed and complex object has difficult side effects; this is technically referred to as sublimation (while never documented this is actually an implementation of the choir boy pattern).

A master of these techniques is revered as a divine architect.

Enum – a misnomer

Enumerators of enum types occupy a unique place within the language in that they are pukka language elements with meaningful values, but they are strictly compile-time quantities. They occupy no storage and so have no address, which is why they are so suitable for holding values in template metaprogramming. One can use static const integers instead of enums in TMP, but there may be an overhead. Listing 11 shows the well-known TMP factorial code example, rewritten to use static const unsigned values instead of enums. The code works as expected, and in this case the compiler optimizes away the storage allocated for the constants in the two templates (when built with optimization turned on) so there is no penalty.

```cpp
template <unsigned N>
struct Factorial
{
    static const unsigned value = N * Factorial<N - 1>::value;
};
template <>
struct Factorial<0>
{
    static const unsigned value = 1;
};
```

It is not an enumeration type (at least not by any sane (and non-recursive) definition) because it allows the definition of values that are unordered and non-contiguous, and which therefore do not lend themselves readily to enumeration. It is nevertheless possible – Wilson has shown us how – to use an enum as the basis for a datatype that can be enumerated.

The properties that make enum unsuitable for enumeration do have their uses, however, for defining groups of named compile-time constant values without allocating any storage.

References


Code samples accompanying this article were compiled and tested with gcc (v4.4.5), g++, fpc, gnat, or gm2, as appropriate. The scoped enum code was compiled with the -std=c++0x switch.
C++ Standards Report 11
Roger Orr reports on the new C++ standard.

As anticipated in the last CVu we have a new C++ standard (snappily known as ‘ISO/IEC 14882:2011’) and the PDF can now be purchased from the ISO website for a ‘mere’ 352 CHF. National bodies will also make the standard available in due course and we are exploring trying to get the standard printed in the UK for a somewhat more realistic price (the 2003 standard was published by Wiley this way for £35). Watch this space!

We have also had a blog posting from Microsoft with news of the features of C++11 that will be in the next version of Visual Studio (http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx).

While it is good that a number of ‘C++0x’ features are already in VS 2010, I was rather disappointed with the small number of extra features that will be added for the next release. I suspect enough customers will have to ask about the missing features before there is enough commercial pressure to implement them. Users of g++ have much better coverage (http://gcc.gnu.org/projects/cxx0x.html).

Call for work items
The August C++ standards committee meeting was a bit more relaxed than the previous few meetings had been since the ink is barely dry on new standard.

The main business in the core language working group was with wording changes, mostly minor, to remove ambiguities or small problems. People trying to implement the new C++11 features had raised some of these issues as they uncovered problems (or potential problems) with the new wording. Work on any more substantial new core language features will wait for further meetings to decide which (if any) such items should be considered.

Walter Brown presented a paper describing an ambiguity in the C++ handling of conversion operators – one of his examples was a zero_init class such that:

```cpp
zero_init<int> i;
```

```cpp
i = 7;
switch( i ) { ... } // fails to compile
switch( i+0 ) { ... } // compiles
```

Perhaps this code will appear in a future Code Critique column!

There was more new work in the library working group as they discussed five papers for future library extensions: filesystems, shared_lock, permutations of partial elements of set, I/O for duration types and date time. They also issued the following general call for proposals:

The C++ committee Library Working Group welcomes proposals for library extensions which will be considered starting in the February 2012 meeting. We have not yet set out an overall timeline for future library extensions, but are ready to consider new proposals at this point.

To increase the chances of your proposal being accepted by the committee, we strongly recommend that a committee member willing to champion your proposal (this could be you yourself, or a delegate) attend upcoming meetings to help shepherd your proposal through the process.

Please take note of this if you have a C++ library that would be suitable for standardisation and would be prepared to put in some of the work for the process. You could contact me (or other members of BSI C++ panel) in the first instance. As with the C++11 standard, it is likely that many of the new library features will be implemented in publicly available repositories, such as boost, to provide experience and feedback from use in real programs.

ROGER ORR
Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

Inspirational (P)articles
Frances Love introduces Paul Grenyer.

This time Paul Grenyer tries to disagree with Steve Freeman and Nat Pryce, and realising the error of his ways changes his perspective and writes his database tests in a new way. Engaging with a subject often involves arguing with people only to realise they are right. Have you changed perspective on something recently, or tried arguing only to realise you were wrong? Then write CVu an inspiration (p)article sharing the details. Contact frances.buontempo@gmail.com.

So Long and Thanks for all the Transactions
I’m a big fan of integration testing data access layers using transactions. Very simply you start a transaction, clear out put any necessary data into the database, run the test against it and roll back the transaction putting the database back to its original state. The advantage is a known database state every time you run the test.

I was very surprised when I read in Growing Object Orientated Software Guided by Tests that integration tests should not be run in a transaction. The database should still be cleaned out and have necessary data inserted, but it is not rolled back after the tests. The two main advantages are that the tests run in an environment closer to production and resulting database state can be used to help solve those awkward bugs.

Another advantage of transactions is that manual and automated system tests can use the same database as the integration tests without the integration tests destroying all the data needed for the other tests. I started discussing why I liked the transaction approach with Steve & Nat. Nat pointed out that the solution was to have two databases. One for system and manual testing and one for integration testing. The creation of the database for your application should be scripted and automated anyway, so keeping the structure, reference data and test data synchronised should be easy.

This was clearly the solution and completely changed how I felt about transactions. Where possible my future projects will have two databases and fewer integration test transactions.
Desert Island Books
Roger Orr shares the contents of his suitcase.

I can’t actually remember the moment I met Roger Orr for the first time. I thought it was probably at a C++ panel meeting or an ACCU conference. Roger thinks it was on the platform at Oxford station on the way back from one of my first conferences. I’m ashamed to say I cannot remember this meeting at all, but I was still drinking then and everything immediately following a conference was usually quite hazy.

Regardless of how we met, I have this feeling that Roger has just always been there providing the steady hand of common sense. He is the current organiser of the ACCU Canary Wharf lunch. Most recently, although in different teams, Roger and I have been working for the same bank. And of course he also edits CVu’s very own code critique and takes an active role in the C++ standards committee.

Roger Orr
I like the idea of ‘Desert Island Books’ and it has been fascinating to hear about the wide range of books people have listed. Inevitably enough there are a few books that just keep coming up: and so my first book duplicates one of the ones that Chris O’Dell covered in May’s CVu.

I had a six month job, way back in 1978, as a ‘trainee computer programmer’ in the gap between finishing at school and starting at Uni and I worked in a team of ten or so programmers writing programs on a mini computer. I was assigned to someone in the team to teach me to program and, looking back, I was incredibly fortunate in having the mentor I got. Dave Buckle was a good programmer and was a very good example from whom to learn the craft of programming. In addition to learning from him in person, over those six months he lent me several books on programming – I couldn’t now recall the content of all but one. There was Jackson’s Structured Programming book and a book on writing structured programs in Fortran using ‘RatFor’ (Rational Fortran: a preprocessor for Fortran). Although these particular books are now quite old and technology has moved on a long way, the lessons Dave taught me about keeping interested in the subject and reading more broadly than just your current experience of programming have stood me in good stead since.

He also lent me my first Desert Island choice: The Mythical Man Month by Fred Brooks, which is still just as relevant as it was when I first read it as a relatively new book. I consider this is one of the ‘must read’ books in the programming world. It is a bit sad to realise that although it was written in 1975 we still haven’t learned the lessons. For those of you who haven’t read it … put down the magazine now and go and order it! It was reissued as an anniversary edition in 1995 and is still for sale.

Fred Brooks was one of the first people to point out that adding people to a delayed software project makes it later and also coined the term ‘second system syndrome’ to describe the danger of over-engineering the second version of a program by trying to fit all the features there wasn’t time to put into the first version. Both these problems still seem to be alive and well in today’s world of software engineering.

My second book comes a few years later when I was programming in ‘C’ on PCs. I read a book review in the EXE programming magazine where the actual text of the review made a shape on the page and the review ended with something like ‘whatever you do, buy this book’. I was soon the proud owner of Obfuscated C and other mysteries by Don Libes – a book unlike any other I’d come across. The chapters alternated between describing entries in the annual obfuscated C competition (now defunct, but see www.ioccc.org) and chapters on general C programming issues such as ‘Keeping Track of Malloc’ and ‘Byte Ordering’. The book is an entertaining read but also very informative. It also opened my mind to some of the creative things you could do with a programming language and how you can abuse the rules and (sometimes) get away with it.

(Incidentally EXE magazine is the reason I’m a member of ACCU: Francis Glassborow’s regular column in that magazine kept mentioning this organisation called ‘ACCU’ and eventually I decided to investigate further. The rest, as they say, is history. I wonder how many other members of ACCU arrived down the same route!)

Programming in C gradually morphed into programming in C++ and this leads me on to my third book, commonly known as the ARM. The Annotated C++ Reference Manual by Bjarne Stroustrup and Margaret Ellis was for some years the definitive book for the C++ programming language. It has now been superseded by the ISO standard which first appeared in 1998 but for much of the 1990s this book defined the C++ language.

It was the book I learned the language from. Now I know most normal people would learn a language from a tutorial book but I was perfectly happy to read through the reference manual and try to make sense of it. Little did I suspect that later on I’d be sitting in committees revising the wording of the ISO C++ standard!

One of the things I enjoy most about programming is finding and fixing bugs; particularly when the relationship between the symptom and the cause is obscure. It’s a bit like a puzzle: you know there must be an explanation for the behaviour you observe in the program but it just doesn’t make sense…yet. There are very few good books about debugging despite the large amount of time that most programmers spend doing it.

I was delighted to come across my fourth book, simply titled Debugging by David Agans. He wrote ‘I intended

---

What’s it all about?

Desert Island Discs is one of Radio 4’s most popular and enduring programmes. The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml).

The format of ‘Desert Island Books’ is slightly different from the Radio 4 show. You choose about five books, one of which must be a novel, and up to two albums. Some people even throw in the odd film. Quite a few ACCUers have chosen their Desert Island Books to date and there are plenty more to go.

The rules aren’t too strict but the programming books must have made a big impact on your programming life or be ones that you would take to a desert island. The inclusion of a novel and a couple of albums helps us to learn a little more about you. The ACCU has some amazing people and Desert Island Books has proved we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.
Memories of Learning C
Anthony Williams recalls his first experiences of C.

I studied Physics at college, and there was very little programming taught as part of the course. That didn’t bother me though; I’d taught myself to program up until then, and I wasn’t going to stop now. The big benefit I got from computing at college was access to the internet, and access to C and C++ compilers. I could program in BASIC, Pascal and a couple of forms of assembly language, and I’d eagerly read Stan Lippman’s C++ Primer and written out (on paper!) some C++ code, but I hadn’t yet had a C++ compiler to try out my programs on.

I wrote several C++ programs before I even considered writing a plain C program, but I probably typed in and compiled the classic printf("hello world\n"); program to check everything was working before I compiled any C++.

Usenet
My strongest memories about learning C are about learning from usenet. Though I had access to C compilers at college, access to experts was not so readily available unless you were studying computing. With access to the internet, I didn’t need local experts though – usenet provided access to experts from across the world. I read comp.lang.c and comp.lang.c++ avidly, and taught myself both languages together. The usenet community was invaluable to me. The wealth of knowledge that people had, and their willingness to share with newbies was something I really appreciated.

I remember struggling over file handling, and getting the arguments to scanf right; I remember puzzling over the poor performance of a program and having someone kindly point out that my code was doing malloc and free calls in a tight loop. Though I tend to answer more questions than I ask these days, I still hang out on newsgroups such as comp.lang.c++ today. It seems that for many people StackOverflow has replaced usenet as the place to go for help, but the old-style newsgroups are still valuable.

Ubiquity
Back then, C++ compilers were in their infancy. Templates didn’t work on every compiler, there was no STL, and many platforms didn’t have a working C++ compiler at all. I consequently wrote a lot of C – every platform had a C compiler, and my C code would work on the college PCs, my PC (when I saved up enough to buy one), the University’s Unix machine, and the Physics department workstations. The same could not be said for C++.

The ubiquity of C is something I still appreciate today, and this is only possible because Dennis Ritchie designed his language to be portable to multiple platforms. Though ‘implementation defined’ behaviour can be frustrating when the implementation defines it a different way to how you would like, it is this that enables the portability. You want to write code for a DSP that only handles 32-bit data? Fine: make char, short, int and long all 32-bits.

What made it, for me, a great book for teaching the essential and universal techniques of debugging ... I demonstrated this by using examples in various disciplines including hardware and software, cars, houses, and even human bodies.” His point was that a lot of debugging comes down to some simple rules that are applicable to many different kinds of problem solving. I suspect, on this mythical desert island, I could use the same techniques to solve some of the problems I would be likely to come across thrown up by daily life.

I thought about what novel I would like to take to my desert island, and decided that it was too hard to decide: there are so many I’d like to have with me! So I decided I’d do something different and take a poetry anthology: Seven Ages: Poetry for a Lifetime by David Owen. This was the book I took with me on a trip to Tanzania in 2000 and is a great book if you can only take one: there are hundreds of poems in it loosely grouped around the seven ages of man from Shakespeare’s famous poem starting ‘All the world’s a stage, And all the men and women merely players, They have their exits and entrances, And one man in his time plays many parts, His acts being seven ages.’ What made it, for me, a great book for travelling with was it could take a very long time to read; you read a poem, perhaps read it again, and then think about it.

Finally some music. I’m really not sure what to choose. I spent much of my student years working to various Beatles compilations; that would still be one option. Alternatively my most recent purchase might be an interesting one to pick: Tres Tres Fort by Staff Benda Bilili. This band are a group of paraplegic musicians living around the Zoo in Kinshasa. Many of the instruments are made or adapted by themselves and the music is quite unusual, but full of life. Perhaps it would inspire me to try and make some music myself with whatever I could find on my desert island!?
Code Critique Competition 72
Set and collated by Roger Orr. A book prize is awarded for the best entry.

Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Last issue’s code
I’m trying to write a simple circular list — it is like a std::list but it wraps at each end just like days of the week do. However, when I try to go on five days from Wednesday I reach Sunday, not Monday. Please help!

cc71>circularListTest
Today is Wed
Yesterday was Tue
5 days time will be Sun

circularList.h is in Listing 1 and circularListTest.cpp is in Listing 2.

```cpp
#include <list>

template <typename T>
class circular : public std::list<T>
{
  typedef std::list<T> list;
public:
  class iterator;
  circular() {}
  template <typename IT>
circular(IT beg, IT end) : list(beg, end) {}
  iterator begin() { return iterator(*this); }
  iterator end() { return --begin(); }
  // iterator implementation details
  class iterator : public list::iterator
  {
    list & parent;
    typedef typename list::iterator super;
    public:
      iterator(list & par) : parent(par), super(par.begin()) {}
      using super::operator=;
      iterator& operator++()
      {
        if (*this == parent.begin())
          *this = parent.end();
        else
          this->super::operator++();
        return *this;
      }
      iterator& operator+=(int n)
      {
        while (n-- % parent.size())
          ++*this;
        return *this;
      }
      iterator& operator--()
      {
        if (*this == parent.begin())
          *this = parent.end();
        else
          this->super::operator--();
        return *this;
      }
      iterator& operator-=(int n)
      {
        while (n-- % parent.size())
          --*this;
        return *this;
      }
      // derived operators
      iterator operator+(int n)
      {
        iterator result(*this);
        return result += n;
      }
      iterator operator-(int n)
      {
        iterator result(*this);
        return result -= n;
      }
      iterator operator++(int)
      {
        iterator it(*this);
        ++*this;
        return it;
      }
      iterator operator+(int n)
      {
        iterator result(*this);
        return result += n;
      }
      iterator operator--(int)
      {
        iterator it(*this);
        --*this;
        return it;
      }
      iterator operator-(int n)
      {
        iterator result(*this);
        return result -= n;
      }
  };
};
```

Listing 2
```cpp
#include "circularList.h"
#include <algorithm>
#include <iostream>
#include <string>
using std::string;
void test(circular<string> s)
{
  circular<string>::iterator it =
  std::find(s.begin(), s.end(), "Wed");
}
```

ROGER ORR
Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk
std::cout << "Today is " << *it << std::endl;
circular<string>::iterator yest = it - 1;
std::cout << "Yesterday was " << *yest << std::endl;
int const n = 5;
it += n;
std::cout << n << " days time will be " << *it << std::endl;
}
int main()
{
    circular<string> s;
s.push_back("Sun");
s.push_back("Mon");
s.push_back("Tue");
s.push_back("Wed");
s.push_back("Thu");
s.push_back("Fri");
s.push_back("Sat");
test(s);
}

Critiques

Paul Floyd <Paul_Floyd@mentor.com>

This is an ‘out by one’ style error. The problem is with operator++

```
iterator& operator++()
{
    if (*this == parent.end())
        *this = parent.begin();
    else
    {
        this->super::operator++();
        return *this;
    }
}
```

which is saying ‘if it’s at the end, wrap to the beginning, otherwise increment’. When this is called at end() -1, it increments to end() and returns that. A circular list should have no end! It is only on the next call to operator++ that the wrapping takes place. So it takes two calls to operator++ to wrap from end() -1 to begin().

Instead, when this is called at end() -1, it should increment then test whether it is at end() and if so wrap:

```
iterator& operator++()
{
    this->super::operator++();
    if (*this == parent.end())
        *this = parent.begin();
    return *this;
}
```

There is a similar problem with operator-- which should be

```
iterator& operator--()
{
    if (*this == parent.begin())
    {
        *this = parent.end();
        this->super::operator--();
    }
    else
    {
        this->super::operator--();
        return *this;
    }
```

Peter Sommerlad <peter.sommerlad@hsr.ch>

The main problem of our student this time is, that he or she doesn’t understand the concept of C++’s iterators well enough. Luckily or unfortunately, STL’s designer(s) did choose a design that mimics the behaviour of pointers. In contrast to, for example, Java or D, that define a single object to denote a range of iteration, C++’s STL uses two iterator objects or two pointers to denote a range representing its beginning and ending. The interesting part then is that the end is marked by a pointer/iterator that is one PAST the end of the underlying sequence. This means accessing a container’s end() iterator is usually undefined behaviour. And that seems to be the problem of the code on first look. So the own iterator uses that undefined end() iterator as one of its steps, this then results in ending one weekday too early. Fortunately, the test program doesn’t access the *end() element and thus does not format my disk or rain pink elephants which might be the case in doing such undefined behaviour.

So the iterator range between begin() and end() actually seems to be something like:

Sun Mon Tue Wed Thu Fri Sat <<(end) place>> Sun Mon Tue ...

That explains, why adding 5 days to Wednesday results in Sun instead of the expected Mon.

A bigger problem is that the code inherits from standard library classes that are not intended to be inherited from. Since the containers and iterators are not using virtual member functions, such inheritance can be dangerous, especially if additional member variables are defined, as is the case with circular::iterator.

Linticator also shows some warning about that:

```
template <typename T>
class circular : public std::list<T>
{
    friend class std::list<T>::iterator;
    // ... test(s);
}
```

Also using the super class’ assignment operator is dangerous as well in such a case, because the new member variable never gets overwritten with such an assignment.

But now, let us start a more thorough analysis:

A quick observation is, that the header file lacks an #include guard. That can be an omission to save space in cvu, but shouldn’t be in real header files to avoid double inclusion and thus violation of the one-definition-rule of C++. Some compilers support #pragma once instead, but I would prefer to stick with the standard mechanism of an internal #include guard. Some ‘classics’ recommend external #include guards, but the reason for them is no longer valid, since modern compilers automatically detect multiple includes and will not parse the files with internal #include guards again.

Now my compiler tells me the following (slightly simplified):

```
g++ -o0 -g3 -Wall -c -fmessage-length=0 -MDM
-mp -MF"scc71.d"-MT"scc71.o"-o"scc71.o"
"../scc71.cpp"
```

In file included from ../scc71.cpp:1:0:
```
../circularList.h: In constructor
    'circular<T>::iterator::iterator(...)':
    ..../circularList.h:18:42: instantiated from here
    'circular<T>::iterator::iterator(...)':
    ..../circularList.h:11:22: instantiated from here
```

This strange message provides the information that the initializer list of the constructor iterator(list &) is in the wrong order. First all base-class elements are initialized for a class, then the member variables in the order of their definition. A further problem of that constructor is that the list is passed and kept as a reference. This can be problematic, if an iterator object survives its ‘parent’ list.

Nevertheless, exchanging the order of the initializer list elements cures that warning:
However, that doesn’t fix the behaviour, just the compilation warning.
The inheriting iterator class is one of the major problems. So let us first encapsulate the list::iterator as a member instead of inheriting it. This way we loose the inherited operator*, operator! and operator==, but those are trivially to implement. In addition we need to inherit from std::iterator to obtain the traits required for the standard algorithms:

```cpp
// iterator implementation details
class iterator : public std::iterator<
    std::bidirectional_iterator_tag,T>
{
    typename list::iterator iter;
    list & parent;
    typedef typename list::iterator super;
public:
    iterator(list & par)
    : iter(par.begin()),parent(par) {}
    typedef typename list::value_type
    value_type;
    value_type operator*() const { return *iter; }
    bool operator==(iterator const &other) const
    { return iter == other.iter; }
    bool operator!=(iterator const &other) const
    { return iter != other.iter; }
};
```

Now, before we fix everything, let us first simplify the code. To write your own iterators I recommend to my students using boost’s iterator helpers from boost/operator.hpp.

This will eliminate some of the operator overloads. Inheriting from boost::bidirectional_iterator_helper<T> will give us the overloads for == and post-fix ++ and --. In addition we can use boost::addable and boost::subtractable to obtain + and - from ++ and --.

```cpp
class iterator : public
    boost::bidirectional_iterator_helper<
        typename list::value_type>
    , public boost::addable<iterator,int>
    , public boost::subtractable<iterator,int>
{
    iterator end() { return --begin(); }
}
```

Now let us use Linticator again to figure out some of the problems:

```cpp
    iterator end() { return --begin(); }
    std::advance(it,-1); return it;
```

Further explanations of Linticator shows us that we are using a temporary for applying operator--(). A better approach and a generically working one, would be to use a local iterator variable instead and std::advance to navigate it:

```cpp
    std::advance(it,begin());
    return it;
```

Now let us fix the logic. To safely wrap around, we must never reach the underlying list’s end() iterator. This can be done as follows:

```cpp
    iterator operator++()
    { ++iter; 
      if (iter == parent.end())
        iter = parent.begin();
      return *this;
    }
```

```cpp
    iterator operator--()
    { 
      if (iter == parent.begin()) { 
        iter = parent.end();
      }--iter;
      return *this;
    }
```

ct voilà, the output now is:

```cpp
    Today is Wed
    Yesterday was Tue
    5 days time will be Mon
```

Now some further cleanup, e.g., getting rid of the potentially error-prone loops, especially if n is negative (which might still be an error, but I ignore that for now):

```cpp
    value_type& operator*() {
        return *this;
    }
    operator++() {
        ++iter;
        if (iter == parent.end())
            iter = parent.begin();
        return *this;
    }
    operator--() {
        if (iter == parent.begin()) {
            iter = parent.end();
        }
        --iter;
        return *this;
    }
```

There are further issues, like considering making iterator a const_iterator instead, or providing that in addition. A non-const overload of operator*() is missing:

```cpp
    value_type& operator*() {
        return *this;
    }
```

And the circular class is still inheriting from std::list. The test() function is not nearly close to a good set of unit tests, one can write with CUTE. In addition there are further lint messages one can deal with, but for the moment I’ve chosen the option to ignore them, because I’ve run out of time and the deadline is too close for me to further fix that. Maybe some other submitter explains that to you as well.

**Balog Pal <pasa@lib.hu>**

This entry won my heart with its first sentence. It so much resembles the favorite approach from clients, asking for a very simple something, that is exactly like some existing program or feature, just has a small difference. And obviously expect me to have one in no time built on top of the designated candidate.

A software engineer certainly must have experience to call an immediate red alert, all shields up and slow to impulse. A difference may look small expressed in English, or drawn in paper, but in software design it may be breaking a fundamental. Murphy’s rule suggests that it normally does too. So our first task here is to clear the mind, step back to requirements, find an implementation candidate using tabula rasa, only then look at the original proposal on similarity. If we could really use the ‘similar’ thing, hooray, and if the change is really small, trivial. But it does not happen very often.

So here we’re suggested that a ‘circular list’ is exactly like a linear list. You know, a circle is exactly like a line. Just bend it, and join the ends, so it forms a circle. In geometry you would use that similarity, do you? Or in topology? Well? We can list similarities! Just the list will not be so huge.
Our patient fell in the trap. Here we indeed have similarities, that can be expressed as “we want the interface of this collection be like std::list’s.” Just as many collections in STL have similar interface, list and vector both have many matching functions and concepts. Certainly they are not implemented in terms of each other.

In the presented code we see public derivation from list and list::iterator to our more specific class. Most STL classes are not created to act as base classes. They don’t have virtual destructors or virtual functions. It is not strictly forbidden to use them as such, but we shall be extremely careful even in cases where we actually have an is-a relation. In use we may face slicing, Janus-faced behavior and other confusions. Before going that way we should read up the rationales of the common do-not suggestions and make sure our case plays as exceptions, or we mitigated the problems some way. We shall re-visit this question after we decided that subclassing would be good in theory.

So, first fundamental question is to see whether circular-list is A std::list really. std::list is a ‘reversible sequence container’. Reading the requirements in the standard I set warning flags on begin(), end(), size(). At reversible container section I have serious doubts our code took it into consideration. Then I get to section 23.1.1, ‘Sequences’. Starting ‘A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement.’

This sounds like a show-stopper unless we can come up with really good explanation how a circle will look strictly linear. In our circular list, any closed subrange indeed looks like a list. But what about beginning and end? A true circle does not have that notation at all.

We may try to force our idea, to chose an arbitrary element to call beginning. That will designate a matching end too. It would even work fine. We may try to force our idea, to chose an arbitrary element to call end? A true circle does not have that notation at all.

closed subrange indeed looks like a list. But what about beginning and end? A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement.’

The purpose of the whole thing was to have our iterators wrap around. So that +=closed_end() returns begin() and we can go ahead; similarly in other direction. In operator + of the iterator we face a dilemma. What to give from our virtual end element? Go to begin() by the circular requirement, or fall to list::end() for the sequence container’s.

I see no resolution to this dilemma and that sends the whole implementation idea down the drain. And really not just the implementation, but the design too. We certainly can design and implement a circular container that works fine. But you must go the extra mile to draw its interface. I suggest removing elements that would make it look like a STL sequence and cause confusion. Possibly have them by a different, distinctive name.

I don’t submit a solution, as it would take huge amount of code, and implementations for circular lists must exist freely accessible. Not even just a design, as it is an open-ended thing, the patient shall make decisions.

The test results are now as expected. However, this test is only the start. There is plenty still to do. I aim to find the implementation’s vulnerabilities through adding more tests.

Well done to the developer for a good first stab at this. It is a neat idea to use the list container and its iterator type. It is a good start to base the implementation on reusing well known code and interfaces. First I’ll deal with the error seen in the test results. The error is in the iterator’s increment operator. It considers ‘end’ to be the last item in the sequence rather than ‘one past the end’ that it really is. The iterator has a reference to the std::list, not the circular<> concrete type. A simple re-write of the pre-increment operator puts this right:

```
iterator& operator++()
{
    this->super::operator++();
    if (*this == parent.end())
        *this = parent.begin();
    return *this;
}
```

The test results are now as expected. However, this test is only the start. There is plenty still to do. I aim to find the implementation’s vulnerabilities through adding more tests.

The code looks like it might be a little error prone here due to the circular::end method not behaving as a standard end method. However, my test empty_list function does pass. I think that it is not reasonable to perform circular iteration operations on empty lists or invalid iterators. An exception should be thrown in this instance.

Ok, another attempt to break the code – what about searching for the last item in the list, “Sat”? I expected this might be incorrect – again because the circular end method returns the last item, not the ‘one past the end’ item. However, this test also works. Why is this? I would have expected the find algorithm to stop the search at the list end, and also for the ‘it == end’ validation check to be true i.e. the iterator is invalid. It turns out there is a bug in the operator--() method. When we’re at the beginning of the sequence, we want to skip to the last item in the sequence, but the code actually assigns the value to end (one past the end). A simple fix is:

```
iterator& operator--()
{
    if (*this == parent.begin())
    {
        *this = parent.end();
    }
    this->super::operator--();
    return *this;
}
```

I would not include a function named end() with that name (by collection requirements). I’d keep begin() only with a compelling use case.

- if the implementation is done using list, I’d be tempted to provide seq_begin() and seq_end(). At least for a tamed form, returning list::const_iterator for ‘special use’ only like dumping the content, etc. It is a double-edged sword with some pros and many cons. It breaks encapsulation and locks in the implementation, but is worth careful consideration.
- do comprehensive tests on rbegin() and company. The original version just left them as inherited from list, I doubt with sensible results ;-)"
My next test (searching for a non-existent string) shows that the above isn’t good enough. The std::find algorithm returns the end iterator given to it, which is actually the last item, not the ‘one past the end’. So we are asking something quite unreasonable from the algorithm.

At this point, it is worth thinking about the design. It is a usable interface? It does the circular iteration just fine, but cannot realistically be used where there is a chance of the iterator being or becoming invalid e.g. find, erase etc. It also shows that it is not suitable where we would want to perform some algorithm on the sequence e.g. std::find.

I think that the standard list interface is what is required for almost all operations (except the circular iteration). The circular iterator should be used as exception rather than the rule.

With that in mind, I have re-written the circular iterator arithmetic operators as simple template functions (see below.) Although this does not give us the familiar ‘++it’ interface, it provides quite a few other benefits including:

- Compatibility with any type of iterator, including const_iterator types.
- Compatibility with any type of std container, not just std::list
- None of the issues (experienced above) with ‘one past the end’ are a problem.

```cpp
/**
 * increment circular iterator
 */
template <typename C, typename I>
I& circular_increment(C& container, I& it)
{
    if (it == container.end())
        throw BadCircularIterator("Cannot increment invalid iterator");
    if (++it == container.end())
        it = container.begin();
    return it;
}

/**
 * Add n to the iterator it.
 */
template <typename C, typename I>
I circular_add(C& container, I it, std::size_t n)
{
    while (n-- % container.size())
        it = circular_increment<C, I>(container, it);
    return it;
}
```

```cpp
/**
 * Decrement the circular iterator it
 */
template <typename C, typename I>
I& circular_decrement(C& container, I& it)
{
    if (it == container.end())
        throw BadCircularIterator("Cannot decrement invalid iterator");
    if (it == container.begin())
        it = container.end();
    --it;
    return it;
}
```

```
Frances Buontempo <frances.buontempo@gmail.com>
I can solve the initial problem with one simple change. Remove the else from the pre-increment operator.

```cpp
iterator& operator++()
{
    if (*this == parent.end())
        *this = parent.begin();
    else
        //REMOVE
        this->super::operator++();
    return *this;
}
```

Problem solved. I wonder if this means the pre-decrement operator has a similar problem? Some moments thought, or at least another test, will reveal more problems.

```cpp
circular<string>::iterator it =
    std::find(s.begin(), s.end(), "Wed");
circular<string>::iterator it_trouble =
    std::find(it, s.end(), "Tue");
if( it_trouble!=s.end() )
    std::cout << *it_trouble << '
';
else
    std::cout << "Tue not there 
";
```

The circular list does not find "Tue". Starting on Wednesday, then I would expect a circular list to find Tuesday eventually, if it circles. This is the crux of the matter. If I want to use the container in the standard algorithms, they expect to have a begin and an end. The code provides end iterator end() { return --begin(); }

```cpp
iterator& operator--()
{
    if (*this == parent.begin())
        *this = parent.end();
    else
        this->super::operator--();
    return *this;
}
```

```
So, end is actually the end of the std::list. This means the standard algorithms won’t circle round the container, unless I do something hacky to add a guard item one before where we start the algorithm, and try to remove it when I think you have finished.

We need to step back and decide how we want to use this container. If I wish to use it in an algorithm like find, I must provide an end. If we want to circle, we want end to go back to the beginning. If I then try to search for something that isn’t in the container, any algorithm that tries to walk from the beginning to the end will never terminate. This gives us two conflicting requirements. If the circular buffer provides a circular_iterator along with the normal iterator this might save the day. Though any attempt to find or walk the whole container is likely to end up iterating forever if the user isn’t very careful. Of course, we should also be providing const_iterator versions too.

Perhaps the best way forward is to provide a member find, which will do one full sweep from any starting point in the container.
bool find(iterator & first, const T & value) {
    size_t count = 0;
    for (; count != contents.size(); ++count, ++first)
        if (*first == value)
            return true;
    return false;
}

This will find Tue if we start at Wed, and will report false if we search
for something that isn’t there. This doesn’t match the usual standard find
algorithm, but a truly circular container doesn’t have an end to compare
against.

Now we have seen the main problems, we should remember Sherlock Holmes’ advice never to derive from classes without virtual destructors,
including standard containers. (C/Vu Issue 12 V 5, 2000, http://accu.org/
index.php/journals/c124 or journals/1063). Or maybe it wasn’t Sherlock
Holmes, but the point still stands. It makes more sense to *contain* a
list, or possibly vector if we want to skip to random places say by using
it += 5, rather than derive from a container.

In conclusion, you should try Python’s list slice and look at boost’s
circular_buffer for inspiration and start over.

Commentary

This problem consisted of three separate, but related issues. The first was
the ‘presenting problem’ where the iteration round the circular list
produced the wrong answer. As Paul said, this was an ‘off by one’ error
and relatively easily fixed.

The second problem was that the circular_list class interface was far
too big: using public inheritance from std::list results in too many
functions ‘leaking out’ into the public interface of the derived class. This
problem is clearly seen with rbegin() / rend() as the circular list class
needs a specific version of these functions. As Peter, Pal and Frances all
pointed out, public inheritance from standard library collection classes is
(almost always) a bad sign.

However, the third and biggest problem is, even if we can fix the syntax,
whether or not it makes sense semantically to try and treat a circular list
like a standard container. As Pal points out, there are a number of
requirements for sequences listed in the C++ standard and a circular list
does not comply with all of them.

The STL is a superb abstraction and has produced a large number of very
useful algorithms and extension points for the C++ collection classes.
However it is not a magic panacea for every possible type of data structure,
and it is important to be aware of the assumptions that the STL requires
for collections and iterators.

I think the best solution is to avoid the standard iterators completely, like
Huw’s solution does, since trying to make such circular iterators fit into
the STL model is doomed to failure.

The winner of CC 71

Paul wrote a short and sweet entry that very clearly answered the initial
question, but I think a little hint towards the bigger issues might have been
good. When answering questions like about code like this I think there is
a balance, sometimes hard to find, between just answering the specific
question and pointing to bigger issues. Peter produced a detailed analysis
of the problems (of which there were many!) with the source code
presented, and once again demonstrated how static analysis tools can help
with writing good code. I also liked Frances’ approach of adding a further
test to demonstrate a further problem with the original code. Huw’s critique
provided a very good solution to the problem with the iterators but I think
the public inheritance from std::list needs tackling too. However, in
my opinion Balog Pal’s entry was the best one because he covered more
of the design problems with the student’s code – it was the design choice
of using an STL collection to implement a circular list that led to the off
by one error in the first place and his critique covered slightly more ground
than Frances’ did.

It was encouraging to have five entrants for this issue’s critique; I hope
even more people will be prepared to try the next code critique!

Code Critique 72

(Submissions to scc@accu.org by Dec 1st)

I’ve written a function that escapes a string of UTF-8 characters using the
html entity format and I’m trying to use it with different compilers. One
compiler fails to find std::runtime_error – don’t know why – and
another compiles it but produces unexpected output. Please help! I wrote a
test program using the four example UTF-8 sequences from http://
en.wikipedia.org/wiki/UTF-8#Description.

Here is what I want:

```cpp
<test_escape_utf8>
U0024 \x24 = $
U00A2 \x2A = €
U20AC \xe2\x82\xac = €
U024B62 \xF0\xA4\xA6\xA2 = �
</test_escape_utf8>
```

Here is the unexpected output:

```cpp
<test_escape_utf8>
U0024 \x24 = $
U00A2 \x2A = €
U20AC \xe2\x82\xac = €
U024B62 \xF0\xA4\xA6\xA2 = \x24b62;
</test_escape_utf8>
```

(See Listing 3 for test_escape_utf8.cpp, Listing 4 for escape_utf8.h and
Listing 5 for escape_utf8.cpp)
Invent Your Own
Computer Games with
Python 2nd Edition
By Al Sweigart, published by
Sweigart, ISBN: 978-0982106013
Reviewed by Daniel Higgins
This software design book is
an easy to use beginner’s guide to Python. It
varies from the most simple program like just a
conversation to more complicated ones such as
a full game.
You don’t have to have the slightest of
experience of programming to use it either, you
could be a total beginner and in less than a week
you can easily have your own quiz on your hard
drive. Then in less than a month you could be
writing games such as dodger! Invent Your Own
Computer Games with Python can also be used
for children as young as ten or eleven and is an
excellent book to start off with before you get to
the really tough stuff!
Overall I would totally recommend this book no
matter how old you are! Buy this and you can
just program, program, program!
Jez Higgins:
As you may have guessed
Daniel is my son. He started using this
book at the start of the year, shortly before
his 11th birthday. He would have said he had no
previous programming experience, but was
thoroughly comfortable using a PC to video and
audio edit, do presentations, and script levels for
games. He found writing his own programs to be
really quite thrilling.
The book is available online or as a PDF from
http://inventwithpython.com/chapters/, although
this review is of the print edition.
Pragmatic Version
Control using Git
By Travis Swicegood, published by
The Pragmatic Programmers,
ISBN: 978-1934356159
Reviewed by Paul Grenyer
I was an early adopter of
Subversion after having used CVS for a little
while. I’ve come rather late to the Git party and
I wanted a book that would give me a quick, yet

Code Critique Competition (continued)

Listing 4
ifndef escape_utf8
#define escape_utf8

std::string escape_utf8(char const * utf8);
#endif

Listing 5
#include <string>
#include "escape_utf8.h"

std::string escape_utf8(char const * utf8)
{
    std::string result;
    long value;
    int multibyte(0);
    while (char const ch = *utf8++)
    {
        if (multibyte-- > 0)
        {
            if ((ch & 0xc0) != 0x80)
                throw std::runtime_error(
                    "Bad multibyte continuation");
            value <<= 6;
            value += ch - 0x80;
            if (!multibyte)
            {
                result += '&#x';
                char buff[7];
                sprintf(buff, "%lx", value);
                result += buff;
                result += ';';
            }
        }
        else if ((ch & 0xC0) == 0xc0)
            value = ch & 0x1f;
        multibyte = 1;
        if (ch & 0x20)
            multibyte++;
        if (ch & 0x10)
            value -= 0x10;
        multibyte++;
        if (ch & 0x8)
            throw std::runtime_error(
                "Bad multibyte start");
    }
    if (multibyte) // more bytes
    {
        value <<= 4;
        value += ch - 0x80;
        if (result.length() > 0)
        {
            result += ch;
        }
    }
    return result;
}
solid, introduction. Pragmatic Version Control using Git is just such a book. I really like the Pragmatic Programmer books as they tend to be short and easy to read. They allow me to absorb a lot of information in a very short period of time. The first thing that struck me was the brilliant simplicity of the example code. Many books on version control use Java as a language that is easily understood by most people. Even with Java you need a fair bit of code before you’ve got a program that does anything, even Hello, World! Swicegood uses HTML as his example code. This is perfect because everyone can understand it easily and you only need a little to do some interesting things. The HTML example is used throughout the book, in my opinion, very successfully. Git itself took me a little by surprise. Having a local copy of the whole repository felt a little extravagant at first and it took me a while to get my head around the idea of having to add a file every time I want to commit it, even if I’ve committed a previous revision. However, Swicegood explains how and why you do both of these very well and now I see the benefit of local copies of a repository and having a staging area. Branching is key to Git and Swicegood explains it in a lot of detail. The book closes with a chapter on Subversion and CVS integration and migration with Git and a chapter on setting up a Git server. The only disappointment for me was the sparse descriptions of GUI clients. I am totally addicted to TortoiseSVN and would have liked to have seen a Git equivalent explained in some detail.

Continuous Delivery

Reviewed by Jez Higgins

The basic premise of Continuous Delivery is straightforward – if continuous integration is a good thing, and it is, why stop there? If we make the effort to automate our compile and test cycle so we can reliably and repeatably build our code whenever we like, then extending our automation and tooling right through into packaging, deployment, smoke/soak/integration testing brings even greater benefits. I’ve spent a large part of the past two years working in exactly the areas this book covers. My colleagues and I have had some success too – our releases now take minutes rather than days. As such this book has served to reassure, but I’d would have loved to have had when we set out. Humble and Farley are comprehensive, covering configuration management, continuous integration, data versioning, and different flavours of deployment and rollback – the end to end of the ‘deployment pipeline’. You’re unlikely to read from cover to cover, but the two-thirds you do read will confirm, correct, comfort, and guide.