# overload 94

DECEMBER 2009 £3

## Quality Matters

We continue to consider the correctness, robustness, and reliability of our code

## Creating a Framework for the iPhone

How to sidestep Apple SDK limitations and ship useful iPhone OS libraries

## Shadow Data Types

A technique for hiding implementation details in C, avoiding common problems

## Project-Specific Language Dialects

Today's languages force a one-size-fits-all approach. We see how to be a bit more flexible.

A magazine of ACCU

ISSN 1354-3172

**OVERLOAD 94**

**December 2009**

ISSN 1354-3172

**Editor**

Ric Parkin
overload@accu.org

**Advisors**

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Simon Farnsworth
simon@farnz.co.uk

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Anthony Williams
anthony.ajw@gmail.com

**Advertising enquiries**

ads@accu.org

**Cover art and design**

Pete Goodliffe
pete@goodliffe.net

**Copy deadlines**

All articles intended for publication in Overload 95 should be submitted by 1st January 2010 and for Overload 96 by 1st March 2010.

**Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website:
www.accu.org**

## ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

# A Crack in Time

Encoding messages has a long history. Ric Parking looks back at how this affected computing.

A couple of weeks ago, I attended the new ACCU autumn conference held at Bletchley Park – home of the Enigma Code crackers. I won't go into much detail – there should be some writeups in the next CVu – but suffice to say it was a splendid day, full of fascinating details of the history of computing, held at what could be described as its birthplace.

One remarkable part of the day was going around the National Museum of Computing, which is based in some of the maze of sprawling huts that housed much of the original code breaking activity. You can imagine that wandering around with around 90 fellow computer geeks made for some interesting anecdotes, especially once you got to the rooms holding the earliest home and personal computers – coos and sighs of "Ah I had one of them" abounded. But one thing that really stuck in the mind was how limited those computers were compared to now, and yet how the designers and users were so ingenious at getting around those limitations. My personal favourite was on the WITCH [WITCH] – it could use punched paper tapes to load a program to run from its central store, but as that store was very small they also used loops of tapes to hold subroutines that could be run directly from the paper (this had one side effect that you'd have to write your program so that a subroutine mustn't get called too many times – after about 500 calls the paper would break!)

But the main thrust of the day was all about codes and code breaking, and it is significant that the first computers were created to make and break codes. At the most simple level, it is because of the sheer amount of repetitive actions that are involved, which have to be performed accurately. But I think there's something deeper going on here: codes and cyphers are all about manipulation of symbolic representations, which when you think about it, is all that a computer ever does – even the simplest number type is actually stored as a pattern of bits, and it's the **interpretation** of that pattern that makes that pattern 'be' the number. The fact that when you tell the computer to add two 'numbers', it actually does manipulations of the symbols, such that the new interpreted pattern will be as if it had added the two interpreted input numbers. (This idea is not exactly new – see sidebar) How it achieves it can be very simple, or very complex. An example would be 'multiply by two' – a compiler could use the knowledge that numbers are stored as binary to turn that into a bitshift operation, or invoke a large multiplication routine.

This sort of thinking can be very useful at times. For example, when I was trying to get my head around the various flavours of Unicode, what really made things fall into place was realising that a 'Character' is really just an interpretation of one or more 'Values' – years of generally only using 7 bit US ASCII had lulled me into conflating the two. Realising that the interpretation is just as important as the actual values involved suddenly made everything make sense. This then also led me to 'get' a lot of the 8-bit extended ASCII issues.

And it goes further. How we choose to represent information can have a significant effect on what we can do with it. Choose the right data structure and a program can perform quickly in a small memory footprint. Conversely a poor choice can mean it runs slowly, consume excessive resources, and perhaps it might not even run at all – a totally unsuitable choice may make implementing an algorithm too difficult, such that it'll either never be completed or be too buggy to use. Take as an example the problem of organising 24-hour support cover. Everyone in your team volunteers the hours they can cover, and you need to check that someone will be on call at any time. One approach would be to make a list for each person of the blocks of hours they can do, then iterate over them and remove that hour from a list of uncovered hours (taking into account that the hour might already be covered). That's rather complex, will probably have lots of memory allocations/deallocations, and will have some slow lookups. Might even contain some bugs. An alternative is to represent the hours a person can cover as a bitmask, then checking that all hours are

## Symbolic Computing and Alan Turing

Precursors go all the way back to Babbage, but the purest form is surely that of Alan Turing (who of course is the most famous of code breakers at Bletchley), in his paper 'On Computable Numbers, With an Application to the Entscheidungsproblem' [Turing]. In it he proposed a thought experiment of a machine that worked on an infinitely long paper tape divided into cells that could either be empty or be filled with a symbol. It had a readwrite head that would be positioned on a single cell, read the contents, and could write or erase a symbol. It could also move the tape left or right. A program was a simple table of rules. Each rule consisted of what actions to do depending on the symbol at the current position. Each action would erase or write a symbol at the current location, then move the tape left, right or stay still, and then change which rule to now apply.

Despite its extreme simplicity, Turing and Alonzo Church suggested that any computable algorithm could be written on a Turing machine (while not completely proven, this is now pretty much accepted as true). Turing went further and wrote a set of rules whose first act would be to read in from a tape the description of a second set of rules, and then process the rest of the tape as if it were the machine described by that input. In other words, a programmable computer.

**Ric Parkin** has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

One of the few pictures of Alan Turing as an adult, running in 1946.

**Figure 1**

covered is just a matter of ORing together all the masks, and checking all the bits are set.

So in a sense, everything in a computer is already in code, it's just that we know how to interpret it. Encryption is only slightly different in that the author is now actively trying to hide the method of interpretation. (Although technically what's usually being hidden is the key used to drive an encryption algorithm, as it's easy to create a new key but very very hard to create a new algorithm). How hard they try to hide it depends on how much it costs to do the encryption, and how valuable the information is. Let's quickly look at those two variables.

The cost of encryption is usually due to difficulty of knowing what to do, inconvenience in doing it, and the cost of actually doing so, in time and opportunity cost (that is the value of what else you could have been doing instead). Many of these costs have plummeted over time, as encryption algorithms have been publicised, and automated, and computing power has increased so it doesn't take too long to encrypt to a fairly strong standard.

The value of the information can vary widely and depends on many things. Simple monetary value is obvious. From Bletchley we learn that government and military information can be very valuable, as knowing where a U-boat pack is can mean the difference between a convoy getting through with vital equipment, or being sunk. This also hints at the temporal aspect of information's value – it's not much use knowing where those subs were a week after they've sunk your ships. So the value of information often decreases over time, sometimes very rapidly. This is a very useful guide for deciding how strongly to encrypt something – you want the information to be worthless by the time it is cracked. There's also a rather subtle value of the mere existence of a message. This is exploited by traffic analysis, which tries to gather information from the patterns of messages over time and where they are sent, so you don't even need to decrypt the messages to get information. For example using mobile phone records to track a gang planning a robbery – from who calls whom you can often learn about the command structure, and from number and time of the calls you can spot that something is about to happen. You can defeat traffic analysis by hiding the patterns in noise, such always sending a message at the same time each day even if it only contains the equivalent of 'This page left intentionally blank', but this increases the cost of hiding your information.

Putting these two costs together, you should choose to encrypt your messages such that the cost to you of encryption is less than the expected cost of that information being disclosed at the time it can be deciphered.

So if the costs are so cheap, why aren't we all encrypting as a matter of course? Well, for several decades, governments have been trying to control access to cryptography on the grounds that they need to be able to read your messages in the name of law enforcement. In the 90s the US government tried to impose the Clipper chip [clipper] to give them a backdoor to voice communications. There was also the classification of encryption as a munition and so liable to strict export controls, which led to the absurdity of the T-shirt that would land you in prison for wearing it as you left the US, as it had a four line implementation of the RSA algorithm printed on it [Back]. And now, we recently had the full Regulation of Investigatory Powers Act come into force, which makes it a serious offence not to disclose your encryption keys. And which dangerous criminal or terrorist was the first to be jailed for this? A rocket hobbiest who distrusted the police [RIPA]. Best not forget those passwords.

## References

[Back] http://www.cypherspace.org/adam/shirt/

[Clipper] http://en.wikipedia.org/wiki/Clipper_chip

[RIPA] http://www.theregister.co.uk/2009/11/24/ripa_jfl/

[Turing] http://www.thocp.net/biographies/papers/turing_oncomputablenumbers_1936.pdf

[WITCH] http://en.wikipedia.org/wiki/WITCH_(computer)

# Shadow Data Types

Hiding implementation details is a good idea. Jon Jagger shows a technique in C that avoids some of the common problems.

Suppose we have a type called **wibble** defined as a concrete data type (that is, a type whose representation is fully visible) as shown in Listing 1.

```
#include "grommet.h"
#include "flange.h"

typedef struct
{
    grommet g;
    flange f;
} wibble;

void wopen(wibble * w, int i);
...
void wclose(wibble * w);
```

Listing 1

The definition of **wibble** exposes the types involved in its representation (**grommet** and **flange** in this made up example), and hence requires a **#include** for those types in its header file. This exposure has a price. One cost is that any change to the **grommet** or **flange** header files, or any header files they in turn **#include**, at any depth, will require a recompilation of any source file that **#include**s `wibble.h` (either directly or transitively). Another cost is that client code can easily become reliant on the exposed representation rather than relying solely on the functional api. Note that in C++ you can avoid this latter problem by declaring your data members private.

These costs are sufficiently high that software developers have invented techniques to hide a type's representation: turn it into an Abstract Data Type. This is simply a type that does not reveal its representation; a type that abstracts away its representation. In this article I'll look at two abstract data type implementation techniques: Opaque Data Types, and Shadow Data Types.

## Opaque data types

The term Opaque Data Type is a well established term for the technique of exposing only the name of the type in its header file. This is done with a forward type declaration. This certainly has the effect of not exposing any representation!

```
typedef struct wibble wibble;

wibble * wopen(int);
...
void wclose(wibble *);
```

**Jon Jagger** is a self-employed software consultant-trainer-coach-mentor-programmer who works on a no-win no-fee basis. He likes the technical aspects of software development but mostly enjoys working with people. He can be contacted at jon@jaggersoft.com.

A definite downside with this approach is that clients cannot create objects themselves.

```
#include "wibble.h"

void eg(void)
{
  wibble * ptr; // ok
  wibble value; // constraint violation
  ptr = malloc(sizeof *ptr);
              // constraint violation
}
```

The **wibble** type's representation is defined in its source file and so only code in the source file can create wibble objects. Furthermore, these wibble objects have to be returned to users as pointers. These two constraints mean the created objects cannot have auto storage class. This is a great loss since auto storage class alone of the three storage class options allows the clients to decide where the objects live which can greatly improve locality of reference.

So how can **wibble**'s source file actually create them? The first possibility is to use an object with auto storage class:

```
...
wibble * wopen(int value)
{
  wibble opened;
  ...
  return &opened; // very very bad
}
...
```

A second possibility is to create the objects with static storage class:

```
...
static wibble wstorage[42];
static size_t windex = 0;
...
wibble * wopen(int value)
{
  wibble * opened = wstorage[windex];
  windex++;
  ...
  return opened;
}
...
```

The final possibility is to create the objects with allocated storage class. That is, to create the objects dynamically on the heap:

```
...
wibble * wopen(int value)
{
  wibble * opened = malloc(sizeof *opened);
  ...
  return opened;
}
...
```

The static and the allocated approaches have opposing advantages and disadvantages. Static storage is very fast and doesn't fragment the memory but the type has to decide the maximum number of objects the application will need. That might be a dependency going in the wrong direction. Allocated storage is much slower and can create memory fragmentation issues, but the application decides how many objects it needs.

In short, the classic ADT technique creates an abstraction that is very opaque and pays a hefty price for this 'over-abstraction'. Abstracting away the representation also abstracts away the size details of a type and it is the loss of the size information that creates the storage class restrictions. The Shadow Data Type implementation technique attempts to rebalance these forces of abstraction by separating size abstraction from representation abstraction.

## Shadow Data Types

The term Shadow Data Type, in contrast to Opaque Data Type, is not a well established term. The technique has probably been around for a long time, it's just that it doesn't seem to have ever been documented anywhere and so a term for it has never become established. I've chosen the term Shadow Data Type to try and convey the idea that when you shine a light on an object it casts a shadow which reveals something of the size of the object but nothing of the details of the object. In other words, a Shadow Data Type has a 'full' type declaration (rather than a forward type declaration) but one revealing only the size of type.

```
typedef struct
{
  unsigned char size_shadow[16];
} wibble;

void wopen(wibble *, int);
...
void wclose(wibble *);
```

The 'true' definition of the type (together with its accompanying **#include**s) moves into the source file (Listing 2).

However, there are two problems needing attention.

## Synchronized alignment?

Firstly, there is no guarantee the two types (**wibble** and **wibble_rep**) are alignment compatible. We can solve this problem. The trick is to create a union containing all the basic types. We don't know which basic types have the strictest alignments but if the union contains them all the union must also have the strictest alignment.

```
typedef union
{
  // one of each of all the basic types go here
  // including data pointers and function pointers
} alignment;
```

```
#include "wibble.h"
#include "grommet.h"
#include "flange.h"
#include <string.h>
typedef struct
{
    grommet g;
    flange f;
} wibble_rep;
// sizeof(wibble) >= sizeof(wibble_rep)
void wopen(wibble * w, int value)
{
    wibble_rep rep;
    ...
    memcpy(w, &rep, sizeof rep);
}
...
```

Listing 2

We redefine **wibble** to be a union with two members; one member to take care of the memory footprint and one member to take care of the alignment:

```
#include "alignment.h"

typedef union
{
  unsigned char size_shadow[16];
  alignment universal;
} wibble;
...
```

The main problem with **wibble** being a union is that unions are rare. Suppose you want to forward declare the **wibble** type in a header. You're quite likely to forget it's a union.

```
typedef struct wibble wibble; // Oooops
```

We can fix this by simply putting the union inside a struct!

```
#include "alignment.h"

typedef struct
{
  union
  {
    unsigned char size[16];
    alignment universal;
  } shadow;
} wibble;
...
```

This is now sufficiently tricky to warrant an abstraction of its own (Listing 3).

## Synchronized size?

The second problem is hinted at by the comment in `wibble.c`

```
// sizeof(wibble) >= sizeof(wibble_rep)
```

This comment, like all comments, has no teeth. Ideally we'd like an assurance that if the sizes lose synchronization we're told about it. This can be done by asserting the relationship inside a unit test of course. The problem with this is the possibility that the runtime check inside a unit-test won't get run. Or, more likely, that the unit-test simply won't get written at all. Fortunately in this case we can check the relationship using a compile time assertion. We start with the fact that you cannot declare an array of negative size:

```
extern char wont_compile[-1];
extern char will_compile[+1];
```

Now we have to select a size of either +1 or -1 if the asserted expression is true or false respectively.

```
// may or may not compile
extern char compile_time_assert[sizeof(wibble)
   >= sizeof(wibble_rep) ? +1 : -1];
```

```
#ifndef SHADOW_TYPE_INCLUDED
#define SHADOW_TYPE_INCLUDED
#include "alignment.h"
#define SHADOW_TYPE(name, size)  \
  typedef struct                 \
  {                              \
    union                        \
    {                            \
      unsigned char bytes[size]; \
      alignment universal;       \
    } shadow;                    \
  } name
#endif
#include "shadow_type.h"
SHADOW_TYPE(wibble, 16);
```

Listing 3

```
#define COMPILE_TIME_ASSERT(description,         \
   expression) extern char                       \
   description[ (expression) ? 1 : -1 ]
...
#include "compile_time_assert.h"
...
COMPILE_TIME_ASSERT(
sizeof_wibble_is_not_less_than_sizeof_wibble_rep,
sizeof(wibble) >= sizeof(wibble_rep));
...
```

Listing 4

Hiding this mechanism behind a macro inside a dedicated header helps to make the code more Intention Revealing (Listing 4).

Note that the assertion uses `>=` rather than `==`. This allows binary compatibility with any alternative smaller representation.

It's worth spending a few moments to think about alignment carefully. The `wibble` type contains a union to give us the strictest alignment. This means a single `wibble_rep` and a single `wibble` can overlay each other in either direction.

If we create an array of `wibble`s the compiler will ensure the address of each `wibble` is suitably aligned. To do this it may add trailing padding to the `wibble` type but this padding will be reflected by `sizeof(wibble)`. Similarly, any padding for the `wibble_rep` type will also be reflected by `sizeof(wibble_rep)`.

Importantly, since `sizeof(wibble_rep)` may be strictly less than `sizeof(wibble)` we cannot overlay an array of either type directly onto an array of the other type.

However, we are only concerned with creating an array of wibbles since that is the type the client uses. There should never be any need to create an array of `wibble_reps`. Nevertheless, the `.c` file implementation must always do any array pointer arithmetic in terms of `wibble`s and never in terms of `wibble_rep`s.

Note also that using `>=` rather than `==` allows binary compatibility with any alternative smaller representation.

## Casting the shadow

Inside the source file we can create a helper function to overlay the true representation onto the client's memory (the fragment in Listing 5 uses the dot designator syntax introduced in c99).

Careful use of `memcpy` can help to make the wibble functions behave atomically from the users perspective. That is to say, the function can do the work off to the side in a local `wibble_rep`, and copy back into the shadow only if everything is successful.

An alternative to `memcpy` is to cast the pointer on each access (Listing 6).

```
static inline void shadow(wibble * dst,
wibble_rep * src)
{
    memcpy(dst, src, sizeof *src);
}

bool wopen(wibble * w, const char * name)
{
    wibble_rep rep =
    {
        .g = ...,
        .f = ...,
    };
    shadow(w, &rep);
    ...
}
```

Listing 5

```
static inline wibble_rep * light(wibble * w)
{
  return (wibble_rep *)w;
}
void wclose(wibble * w)
{
  wibble_rep * rep = light(w);
  rep->g = ...;
  rep->f = ...;
  ...
}
```

Listing 6

### Constness?

It makes no sense to declare a `wibble` object with a `const` modifier unless the object can be initialized.

```
void pointless(void)
{
  const wibble w; // :-(
  // ... ?
}
```

However, this is not an issue since the `wibble` type is opaque anyway. Nevertheless, a slight redesign can accommodate `const wibble` objects if desired, at the cost of copying struct objects (Listing 7).

## Summary

In C it is impossible to expose a type's size without also exposing its representation. It is possible to explicitly specify a concrete type's representation as being 'unpublished' but since C does not offer the C++ private keyword using the representation is always possible and remains a constant temptation.. Once one piece of client code succumbs more are sure to follow and like a dam bursting the client and implementation quickly become tightly coupled and any separation is washed away.

Completely hiding a type's representation behind an opaque pointer/handle removes the temptation and creates a powerful abstraction but at the price of hiding the size of the type and the consequent restriction on the storage class of client memory.

A shadow data type offers a half-way house where a type is effectively split into two, with one part exposing the size and the other part holding the representation. The alignment and sizes of the two parts must correspond. Client code is then able to use all three storage class options. The implementation code takes the full load of the extra complexity mapping/overlaying between the split parts. One interesting observation is that the client code would be unaffected (other than needing recompilation) if the representation was moved back into the client side type (to try and improve performance perhaps).

No mechanism is universally applicable and the shadow data type is no exception! Experience and time alone will tell if and how useful it is. Caveat emptor. ■

```
...
wibble wopen(int value)
{
  wibble_rep rep = { ...value... };
  wibble w;
  memcpy(&w, &rep, sizeof rep);
  return w;
}
void ok(void)
{
  const wibble w = wopen(42);
  ...
}
```

Listing 7

# Creating a Framework for the iPhone

Apple's iPhone SDK doesn't allow you to create a Framework. Pete Goodliffe explains how to build one manually.

Apple's iPhone SDK and the Xcode development environment are a powerful and very easy way to develop incredible mobile applications. The facilities they provide (in ease development, debugging, and the rich Cocoa Touch libraries) far exceed what was available on desktop platforms only a few years ago.

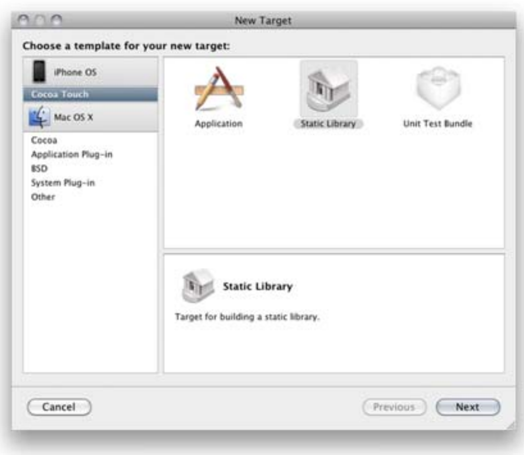
However, there is still a natural bias to simple stand-alone application development – you cannot build your own shared libraries to reduce memory footprint; you can only create applications or simple static libraries. There are some good reasons for this, security and ease of installation being two of the most obvious.

However, seasoned Apple developers are not used to simple static libraries; they are used to Apple's OS X Frameworks (essentially a shared library on steroids, see the sidebar) which are a very convenient method for code sharing and reuse.

Despite the restriction, with a little elbow work and some simple scriptery it is possible to enjoy most of the benefits of a Framework on the iPhone platform. This article explains how to build your own framework for Apple's iPhone OS. I presume a level of familiarity with static and shared libraries. You'll also need to understand bash shell scripting. Understanding the rudiments of Apple development is useful, particularly the Xcode development environment.

## The problem

Apple's Xcode development environment does not let programmers create their own framework for use in iPhone OS applications. As you can see from Figure 1, when targeting the iPhone you can only create an *Application*, a *Static Library*, or a *Unit Test Bundle* (I have to give Apple credit for including the last item, and for its integrated support of unit tests in the Xcode IDE. But that's a different article). This has caused many iPhone developers great frustration, although the restriction is for fairly sensible reasons (Figure 1).



Target for building a static library.

**Figure 1**

So why this restriction?

A framework usually contains a dynamically loaded shared library, and the associated header files a client application requires to be able to access its facilities. iPhone OS keeps applications very separate from one another, and so there is no concept of a user-created dynamic library shared between applications. There is no central library install point accessible to the developer. Indeed, managing such a software pool would be rather complex on iPhone-like devices (the OS hides the file system from developer and user alike). Preventing developers from installing their own shared frameworks neatly sidesteps a whole world of painful shared library compatibility issues, and simplifies the application uninstall process.

It's one, fairly final, way to avoid DLL hell [DLL]!

All applications may link to the blessed, system-provided frameworks[1]. The only other libraries they may use must be standard static libraries, linked directly to the application itself.

### What is a framework?

Although the Apple build technologies for Mac OS and iPhone OS are essentially Unix-like (using the gcc compiler and binutils linker) Apple have applied a number of tucks and tweaks. The addition of Frameworks is one such addition; support for Frameworks has been added into the Apple versions of gcc and binutils.

Frameworks are hierarchical directory structures grouping related but separate items, for example: dynamic libraries, header files, user interface assets and documentation. They provide an internal versioning facility (defined by the directory hierarchy). The OS provides support for loading items from a framework directory, ensuring only one copy is in memory at any time.

Almost all Mac OS and iPhone SDK services are packaged as Frameworks and most third party Mac OS libraries also ship as frameworks. Using a framework in Xcode is as simple as a drag-and-drop operation; header paths and linkage are looked after for you automatically.

Frameworks (and the "Bundle" directory format they are based on) date back to the NextStep platform, which was acquired by Apple and used as the foundation for OS X. More on information frameworks can be found at Apple's Developer Connection site [Apple].

1. Indeed, they may only use the public, documented methods of the system frameworks. Applications that discover and use undocumented APIs will not be allowed onto Apple's carefully policed App Store. This seems draconian, but again, is for fairly obvious reasons.

**Pete Goodliffe** is a software developer, columnist, speaker and author who never stays at the same place in the software food chain. Pete's popular book, Code Craft, is a practical and entertaining investigation of the entire programming pursuit. In about 600 pages. No mean feat! He has a passion for curry and doesn't wear shoes.

**As you can see, static library usage on the iPhone is clumsy. But fear not, there is a way...**

For most simple application developers this situation is perfectly fine. However, those of us who'd like to supply functionality to other users in library form are left at somewhat of a disadvantage. Most Apple-savvy application developers are used to the simplicity of dragging a framework bundle onto their application target in Xcode, and not worrying about header paths or link issues. **`#include`** magically works, and the linkage issues are sorted out under the covers.

As a library provider, it is nowhere near as neat to have to provide a static library and a set of associated header files in a separate flat directory. Doing this requires your clients to work out how to integrate your library in their application by hand. And it'll make integrating a new version of your library into the application more work. Granted, it's not hard (for people who know what they're doing), but it is tedious. That's not the *Apple Way*, is it?!

When shipping a static library, you will also have to ship a library version for each platform the developer will need (at the very least, an ARM code library for use on the iPhone OS device itself, and an i386 build for them to use in the iPhone simulator).

As you can see, static library usage on the iPhone is clumsy. But fear not, there is a way...

## How to build your own framework

I have worked out how to create a usable Framework that you can ship to other iPhone OS application writers. You *can* ship libraries that are easy to incorporate into other projects, and can exploit the standard framework versioning facilities.

There is one caveat: because of iPhone OS limitations the framework will *not* be a shared library; it will only provide a statically linked library. But the application writer need not be concerned about this issue. As far as they're concerned everything will just work as if they were using a standard OS framework.

Here's how to do it:

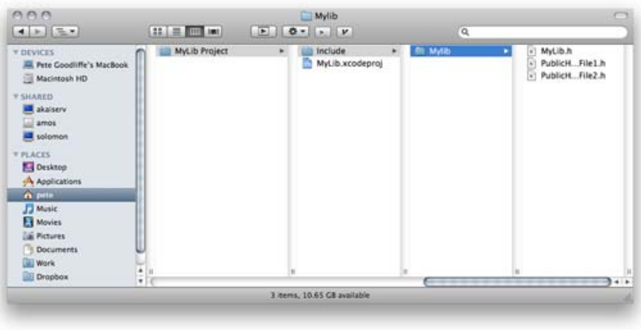
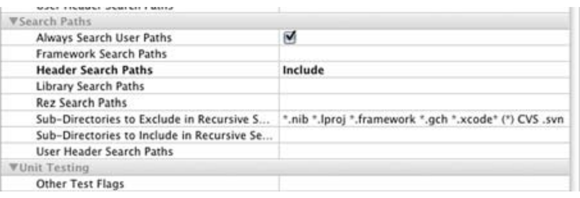


Figure 2



| | |
|---|---|
| ▼Search Paths | |
| Always Search User Paths | ✓ |
| Framework Search Paths | |
| **Header Search Paths** | **Include** |
| Library Search Paths | |
| Rez Search Paths | |
| Sub-Directories to Exclude in Recursive S... | *.nib *.lproj *.framework *.gch *.xcode* (*) CVS .svn |
| Sub-Directories to Include in Recursive Se... | |
| User Header Search Paths | |
| ▼Unit Testing | |
| Other Test Flags | |

Figure 3

### 1. Structure your framework's header files.

Let's say your library is called **MyLib**. Structure your project with a top-level directory called `Include`, and inside that make a `MyLib` subdirectory. Put all your public header files in there.

To be idiomatic, you'll want to create an umbrella header file `Include/MyLib/MyLib.h` that includes all the other headers for the user's convenience. See Figure 2.

Set up your Xcode project **`Header Search Paths`** build parameter to include `Include` (note, do *not* include the `MyLib` subdirectory) as in Figure 3.

Now your source files can happily **`#import <MyLib/MyLib.h>`** in the same way they'd use any other framework. Everything will include properly.

### 2. Put your source files elsewhere

I create a `Source` directory containing subdirectories `Source/MyLib` and `Source/Tests`. You can put your implementation files (and private header files) wherever you want. Just, obviously, not in the `Include` directory!

### 3. Create a static library target

Create an iPhone OS static library target that builds all your library sources. Call this target **`MyLib`**, and by default it will create a static library called `libMyLib.a`.

### 4. Create the framework plist file

Create a plist file that will be placed inside your framework, describing it. Plist files are *Property Lists*, used to store settings about an object (in this case the details about this framework).

I keep my plist file in Resources/Framework.plist. It's a piece of XML joy that should look like Listing 1.

### 5. Construct the framework by hand

Now this is where the real magic happens. Create a shell script to build your framework. I have a `Scripts` top-level directory that contains it, because I like to keep things neat like that. Make sure your script file is executable[2].

2. Type **`man chmod`** in Terminal for details.

**There is one caveat: because of iPhone OS limitations the framework will not be a shared library**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0
  //EN" "http://www.apple.com/DTDs/PropertyList-
  1.0.dtd">
<plist version="1.0">
  <dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleExecutable</key>
    <string>MyLib</string>
    <key>CFBundleIdentifier</key>
    <string>com.MyLovelyDomain.MyLib</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundlePackageType</key>
    <string>FMWK</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>1.0</string>
  </dict>
</plist>
```

Listing 1

The first line is the canonical hashbang [Shebang]:

```
  #!/bin/bash
```

Following this there are two parts to the file...

### 5a. Build all the configurations that you need your framework to support

There must be at least a build for **armv6** for the iPhone device itself, and an **x386** build for the iPhone simulator. Application developers will require both of these to be able to work. You'll want these to be Release configuration libraries.

```
  xcodebuild               \
    -configuration Release \
    -target "MyLib"        \
    -sdk iphoneos3.0
  xcodebuild               \
    -configuration Release \
    -target "MyLib"        \
    -sdk iphonesimulator3.0
```

So that’s our libraries built. That was the simple bit. Now...

### 5b. Piece it all together

With a little understanding of the canonical structure of a framework directory, our ability to write a plist, and the knowledge that putting a static library in the framework instead of a dynamic library works fine, you can create your framework using the script in Listing 2. The comments in the listing describe exactly what’s going on.

```
# Define these to suit your nefarious purposes
                 FRAMEWORK_NAME=MyLib
              FRAMEWORK_VERSION=A
      FRAMEWORK_CURRENT_VERSION=1
FRAMEWORK_COMPATIBILITY_VERSION=1
                     BUILD_TYPE=Release

# Where we'll put the build framework.
# The script presumes we're in the project root
# directory. Xcode builds in “build” by default
FRAMEWORK_BUILD_PATH="build/Framework"

# Clean any existing framework that might be there
# already
echo "Framework: Cleaning framework..."
[ -d "$FRAMEWORK_BUILD_PATH" ] && \
  rm -rf "$FRAMEWORK_BUILD_PATH"

# This is the full name of the framework we'll
# build
FRAMEWORK_DIR=$FRAMEWORK_BUILD_PATH/
$FRAMEWORK_NAME.framework

# Build the canonical Framework bundle directory
# structure
echo "Framework: Setting up directories..."
mkdir -p $FRAMEWORK_DIR
mkdir -p $FRAMEWORK_DIR/Versions
mkdir -p $FRAMEWORK_DIR/Versions/$FRAMEWORK_
   VERSION
mkdir -p $FRAMEWORK_DIR/Versions/$FRAMEWORK_
   VERSION/Resources
mkdir -p $FRAMEWORK_DIR/Versions/$FRAMEWORK_
   VERSION/Headers

echo "Framework: Creating symlinks..."
ln -s $FRAMEWORK_VERSION $FRAMEWORK_DIR/Versions/
   Current
ln -s Versions/Current/Headers $FRAMEWORK_DIR/
   Headers
ln -s Versions/Current/Resources $FRAMEWORK_DIR/
   Resources
ln -s Versions/Current/$FRAMEWORK_NAME $FRAMEWORK_
   DIR/$FRAMEWORK_NAME

# Check that this is what your static libraries
# are called
FRAMEWORK_INPUT_ARM_FILES="build/$BUILD_TYPE-
   iphoneos/libMyLib.a"
FRAMEWORK_INPUT_I386_FILES="build/$BUILD_TYPE-
   iphonesimulator/libMyLib.a
```

Listing 2

**If calling scripts from the command line scares you, you may choose to make a ‘Run Script Build Phase’ in your Xcode project**

```
# The trick for creating a fully usable library is
# to use lipo to glue the different library
# versions together into one file. When an
# application is linked to this library, the
# linker will extract the appropriate platform
# version and use that.
# The library file is given the same name as the
# framework with no .a extension.
echo "Framework: Creating library..."
lipo \
  -create \
  -arch armv6 "$FRAMEWORK_INPUT_ARM_FILES" \
  -arch i386 "$FRAMEWORK_INPUT_I386_FILES" \
  -o "$FRAMEWORK_DIR/Versions/Current/
$FRAMEWORK_NAME"

# Now copy the final assets over: your library
# header files and the plist file
echo "Framework: Copying assets into current
   version..."
cp Include/$FRAMEWORK_NAME/* $FRAMEWORK_DIR/
   Headers/
cp Resources/Framework.plist $FRAMEWORK_DIR/
   Resources/Info.plist
```

Listing 2 (cont’d)

In summary, this script:

- Cleans up any existing Framework (this is cleaner than simply building over the top of anything that may be already there)
- Creates the canonical directory structure for a Framework.
- Creates a single library file that supports all necessary platforms using the **`lipo`** tool.
- Copy header files into the `Headers` directory.
- Copy the plist file into the Framework.

This script generates a fully usable Framework bundle in the `build/Framework` directory. It is called **`MyLib.framework`**. This bundle can be shipped to your external application developers. They can incorporate it into their iPhone OS applications like any other framework.

## Other remarks

I have presented here the most basic structure of a shell file. My production version includes more robust error handling, and other steps that are relevant to my particular project.

I also have a build script that automatically creates documentation for the framework that I can ship with it. Indeed, I have a release script that applies versioning information to the project, builds the libraries, creates a framework, assembles the documentation, compiles release notes and packages the whole thing in a pretty DMG.

If calling scripts from the command line scares you, you may choose to make a ‘Run Script Build Phase’ in your Xcode project to call your framework script from there. Then you can create a framework without having to creep to the command line continually.

In summary, the final file layout of my project looks like Figure 4.

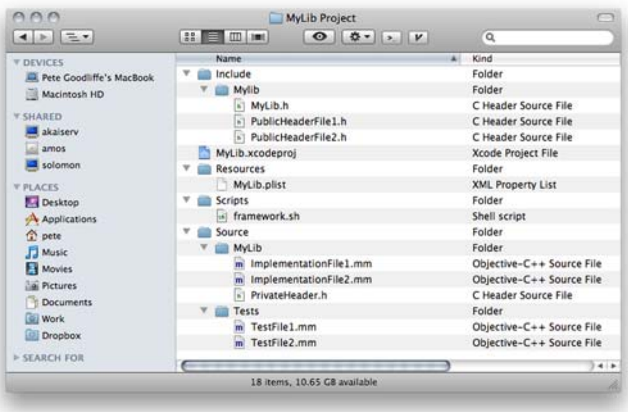
I hope you have found this tutorial useful. ■

## References

[Apple] ‘What are Frameworks?’ *Apple Developer Connection*. http://developer.apple.com/mac/library/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html

[DLL] ‘DLL Hell’, *Wikipedia*. http://en.wikipedia.org/wiki/DLL_hell

[Shebang] ‘Shebang (Unix)’, *Wikipedia*. http://en.wikipedia.org/wiki/Shebang_%28Unix%29



| Name | Kind |
|---|---|
| Include | Folder |
| Mylib | Folder |
| MyLib.h | C Header Source File |
| PublicHeaderFile1.h | C Header Source File |
| PublicHeaderFile2.h | C Header Source File |
| MyLib.xcodeproj | Xcode Project File |
| Resources | Folder |
| MyLib.plist | XML Property List |
| Scripts | Folder |
| framework.sh | Shell script |
| Source | Folder |
| MyLib | Folder |
| ImplementationFile1.mm | Objective-C++ Source File |
| ImplementationFile2.mm | Objective-C++ Source File |
| PrivateHeader.h | C Header Source File |
| Tests | Folder |
| TestFile1.mm | Objective-C++ Source File |
| TestFile2.mm | Objective-C++ Source File |

18 items, 10.65 GB available

Figure 4

# The Model Student: A Primal Skyline (Part 3)

The prime factors of the integers show some repeating patterns. Richard Harris investigates whether they have fractal properties.

In part 1 of this investigation I introduced some of the great questions, both answered and unanswered, about those elitists of the integers, the primes; those whole numbers greater than or equal to 2 that will not suffer to be wholly divided by any other than themselves and the singularly noble 1.

We took a look at Euclid's impressively straightforward proof of the infinitude of the primes and introduced the prime number theorem which states that the number of primes less than or equal to a given number $n$, denoted by the function $\pi$, is approximately equal to

$$\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1$$

where the lim term denotes the limit of the fraction as $n$ grows larger and larger.

Moving on, we considered the factorisations of the integers; the unique sets of primes that when multiplied together result in any given integer greater than 0. I pointed out that 1 can be considered the product of no primes and that 0 can be considered the product of negative infinity of them, or equivalently of dividing 1 by infinitely many primes.

We then took a look at a relative of $\pi$, the function that counts the number of prime factors of its argument, $\Omega$. For example, the number 42 can be represented as the product of the primes $2\times3\times7$ and hence $\Omega(42)=3$.

Recall that $\Omega$ counts repeated prime factors, so that 84, which has the factorisation $2^2\times3\times7$ has a value of $\Omega$ equal to $\Omega(84)=4$.

Building upon this, and in pursuit of a pattern in the factorisations of the integers, I introduced the function $\daleth_n$, defined for non-negative integer $n$, real number arguments greater than or equal to 0 and less than or equal to 1 and the expression

$$\daleth_n(x) = \frac{2^{\Omega(\lfloor 2^n x \rfloor)}}{2^n}$$

Figure 1 illustrates two example graphs of $\daleth_n$ ($\daleth_5$ and $\daleth_7$).

We next showed that, for any $n$ greater than 0, $\daleth_n$ and $\daleth_{n+1}$ are coincident for half the points in the range 0 to 1 and also that the infinite limit, $\daleth_\infty$, can be entirely recovered from an arbitrarily small range of arguments, both of which bear a resemblance to the properties of many fractals.

In part 2, we gave fractals the sound definition of being objects that have a fractional dimension. Of the many definitions of dimension we chose the Minkowski, or box-counting dimension, defined in terms of the number of rulers of length $\varepsilon$ required to cover a curve, $N(\varepsilon)$, by

$$d = \lim_{\varepsilon \to 0} \frac{\ln N(\varepsilon)}{\ln \frac{1}{\varepsilon}}$$

where the lim term means the limit of the expression to its right as $\varepsilon$ tends to 0.

As a motivating example, I introduced the Koch curve, generated by iteratively replacing every straight in its fundamental element, illustrated in figure 2 (the basic element of the Koch curve), with a scaled down version of itself.

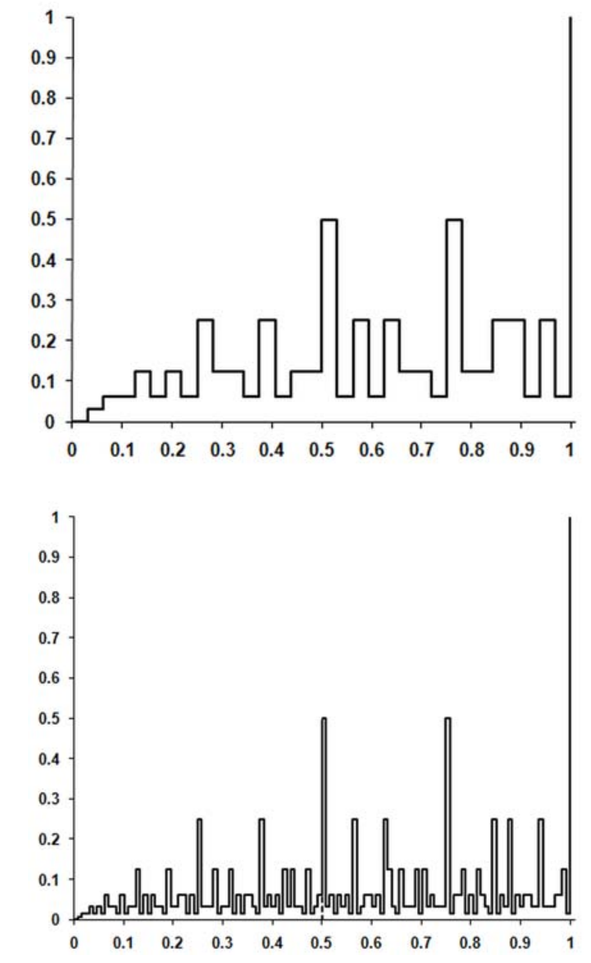


Figure 1

A later iteration in the construction of this curve is illustrated in figure 3.

Noting that every time we divide the length of the ruler by 3, we increase the number of them required to cover the curve by a factor of 4, we can

**Richard Harris** has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

it's not at all obvious exactly what that limit might be

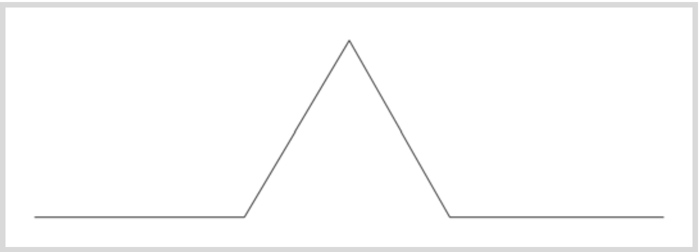


Figure 2

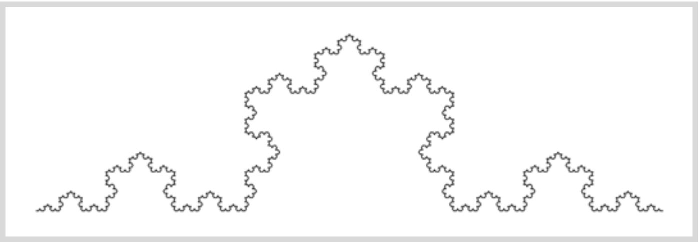


Figure 3

deduced that the fractal dimension of the Koch curve is approximately 1.2619.

We also demonstrated that the number of rulers of length $\varepsilon_n$, equal to 1 divided by $2^n$, required to cover the graph of $ℸ_n$ would probably exceed the maximum value of an **`unsigned long`** and so implemented an accumulator class to keep track of them, as illustrated in listing 1.

We reused the **`count_factors`** function we implemented in the first part, declared as shown below.

```
template<class FwdIt>
unsigned long
count_factors(unsigned long x, FwdIt first_prime,
  FwdIt last_prime);
```

Finally, taking great care to ensure that we couldn't possibly be laid low by integer wrap-around, even for *n* equal to the number of bits in an

```
class accumulator
{
public:
  accumulator();

  accumulator & operator+=(unsigned long n);
  operator double() const;

private:
  std::vector<unsigned long> n_;
};
```

Listing 1

| n | d | n | d | n | d | n | d |
|---|---|---|---|---|---|---|---|
| 1 | 2.0000 | 9 | 1.4745 | 17 | 1.3612 | 25 | 1.2913 |
| 2 | 1.5000 | 10 | 1.4575 | 18 | 1.3505 | 26 | 1.2845 |
| 3 | 1.5283 | 11 | 1.4411 | 19 | 1.3405 | 27 | 1.2781 |
| 4 | 1.5000 | 12 | 1.4258 | 20 | 1.3311 | 28 | 1.2720 |
| 5 | 1.5229 | 13 | 1.4114 | 21 | 1.3222 | 29 | 1.2662 |
| 6 | 1.5181 | 14 | 1.3976 | 22 | 1.3138 | 30 | 1.2606 |
| 7 | 1.5039 | 15 | 1.3846 | 23 | 1.3059 | 31 | 1.2554 |
| 8 | 1.4906 | 16 | 1.3725 | 24 | 1.2984 | 32 | 1.2503 |

Figure 4

**`unsigned long`**, we implemented a function to calculate the required number of rulers, whose declaration is given again below:

```
double count_rulers(unsigned long n);
```

## The box-counting dimension of $ℸ_\infty$

You will no doubt be relieved to read that we are at long last ready to investigate whether $ℸ_\infty$ is fractal or not.

The results of the successive approximations of the box-counting dimension calculation using $ℸ_n$ for *n* from 1 to 32 (the number of bits in an **`unsigned long`** on my, and I suspect almost everyone's, machine) are given in figure 4.

Clearly it has not yet reached its limit and, as illustrated by the graph of these values in figure 5, it's not at all obvious exactly what that limit might be.

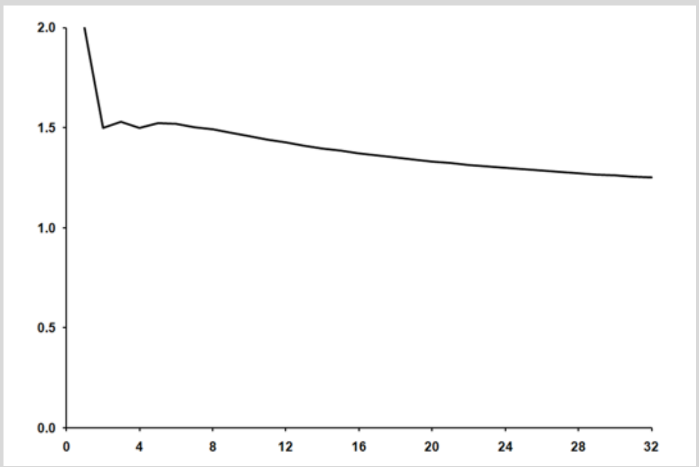


Figure 5

**The length of a straight line is constant no matter what the length of the ruler**

That said, since the graph changes fairly smoothly for the larger values of $n$, we may very well be able to extrapolate it to the limit of $n$ equal to infinity, or equivalently $\varepsilon_n$ equal to 0, and hence determine the fractal dimension of $\rceil_\infty$.

## The length of $\rceil_n$

Rather than consider the successive approximations to the box-counting dimension of $\rceil_\infty$, hereafter denoted by $D_n$, we shall instead consider the lengths of $\rceil_n$ for each $n$, given by

$$l_n = \varepsilon_n \times N(\varepsilon_n)$$

Note that the two values are related by the equations

$$l_n = \varepsilon_n \times e^{D_n \times \ln\frac{1}{\varepsilon_n}}$$

$$D_n = \ln\frac{l_n}{\varepsilon_n} \Big/ \ln\frac{1}{\varepsilon_n}$$

Plotting the length of $\rceil_n$ against $n$, we discover a suspiciously familiar graph, as illustrated in figure 6.

In fact, this curve is reasonably well approximated by the formula

$$l_n \approx \frac{1}{4}n^2 + 1$$

and is *very* well approximated by

$$l_n \approx 0.26397450n^2 - 0.45141160n + 1.85600512$$

especially so for large $n$.

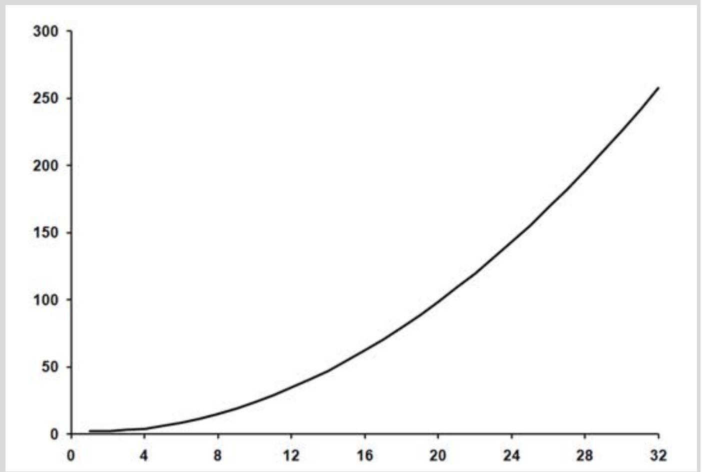


Figure 6

$$\begin{aligned}
D_n &\approx \ln\frac{0.26397450n^2 - 0.45141160n + 1.85600512}{\varepsilon_n} \Big/ \ln\frac{1}{\varepsilon_n} \\
&= \ln\frac{0.26397450n^2 - 0.45141160n + 1.85600512}{2^{-n}} \Big/ \ln\frac{1}{2^{-n}} \\
&= \frac{\ln(0.26397450n^2 - 0.45141160n + 1.85600512) - n\ln 2}{-n\ln 2}
\end{aligned}$$

As $n$ grows ever larger, so the linear and constant terms in the argument of the first logarithm becomes ever less significant, implying that

$$\begin{aligned}
D_n &\approx \lim_{n\to\infty}\frac{\ln 0.26397450n^2 - n\ln 2}{-n\ln 2} \\
&= \lim_{n\to\infty}\frac{2\ln n + \ln 0.26397450 - n\ln 2}{-n\ln 2}
\end{aligned}$$

and since, for $n$ greater than 1, $\ln n$ is always absolutely smaller than $n$, and significantly so for large $n$, we have

$$D_\infty \approx 1$$

Derivation 1

This implies that the approximate fractal dimension $D_n$ is itself accurately approximated by 1, as shown in derivation 1.

## So it's *not* a fractal?

Well, whilst I haven't provided anything remotely close to a proof, I very much suspect that it isn't.

Nevertheless, the fact that its length is a function of the length of the ruler used to measure it is still an interesting property. Specifically, since $n$ can be recovered from $\varepsilon_n$, we have

$$n = \frac{\ln(1/\varepsilon_n)}{\ln 2}$$

$$l_n \approx 1 + \left(\frac{\ln(1/\varepsilon_n)}{2\ln 2}\right)^2$$

The length of a straight line is constant no matter what the length of the ruler, and is in some sense less fractal-like than $\rceil_\infty$.

Since a fractal has a dimension greater than 1, let's say $d$, it has a length that is an exponential function of the logarithm of the reciprocal of the length of the ruler, as shown in derivation 2, and is in the same sense more fractal-like than $\rceil_\infty$.

## And what of simple curves?

Recall that we derived the fractal dimension of the circumference of the unit circle using inscribed polygons as illustrated in figure 7.

Now as our ruler gets smaller and smaller, the measured length of the circumference gets closer and closer to $2\pi$. Any formula relating the log of the length of the ruler to the measured length of the circumference must

**the log of the ruler length grows increasingly negative as the ruler length decreases**

$$\begin{aligned} l_n &= \varepsilon_n \times e^{d \times \ln(1/\varepsilon_n)} \\ &= e^{-\ln(1/\varepsilon_n)} \times e^{d \times \ln(1/\varepsilon_n)} \\ &= e^{-\ln(1/\varepsilon_n) + d \times \ln(1/\varepsilon_n)} \\ &= e^{(d-1) \times \ln(1/\varepsilon_n)} \end{aligned}$$

**Derivation 2**

be dominated by the constant term of $2\pi$, since the log of the ruler length grows increasingly negative as the ruler length decreases. A rough mathematical treatment of this is given in derivation 3.

## Mocktals? Or faketals?

What we appear to have found is a curve that sits somewhere between a fractal and a simple curve; a curve whose length increases without limit as the length of the ruler shrinks, but is governed by a polynomial, rather than an exponential, function of the logarithm of the reciprocal of the length of the ruler.

I shall christen these curves mocktals, or perhaps faketals (I can't decide), and use the power of the dominant term in the polynomial function that determines their length, which I shall call the mocktal (or faketal) order or the curve, to compare them to each other.

Note that this order is exactly the sense in which $\daleth_\infty$ is more fractal-like than a simple curve. A simple curve, tending as it does to a constant value,

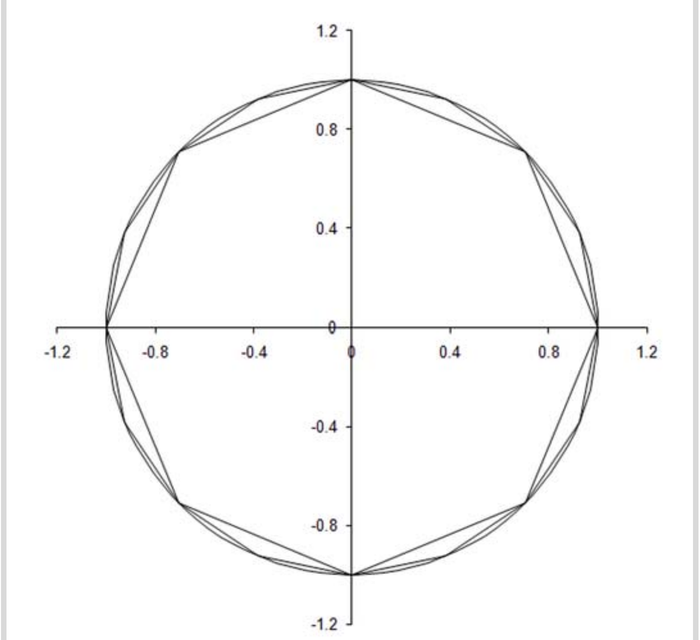


**Figure 7**

Note than an $n$-sided polygon has sides of length

$$\varepsilon_n = 2 \times \sin\left(\frac{\pi}{n}\right)$$

Recall how we used Taylor's theorem to approximate a function for small arguments in previous articles

$$f(x) \approx f(0) + f'(0)x + \frac{1}{2}f''(0)x^2 + \frac{1}{6}f'''(0)x^3$$

Using these first 4 terms to approximate the sides of the polygon, we have

$$\varepsilon_n \approx \frac{2\pi}{n} - \frac{1}{3}\left(\frac{\pi}{n}\right)^3$$

The length of the circumference is therefore

$$l_n = n \times \varepsilon_n \approx 2\pi - \frac{\pi^3}{3n^2}$$

Using the first term in our approximation of $\varepsilon_n$ to estimate $n$ in terms of the log of its reciprocal, we have

$$n \approx 2\pi e^{\ln(1/\varepsilon_n)}$$

and hence

$$l_n \approx 2\pi - \frac{\pi}{12e^{2\ln(1/\varepsilon_n)}}$$

**Derivation 3**

has a mocktal order of 0. A fractal, having a length equal to an exponential function of the logarithm of the reciprocal of the ruler length, has infinite mocktal order. This is because the exponential function can be exactly defined for all arguments, using Taylor's expansion about 0 again, by a polynomial with an infinite number of terms. This is such an important result that the expansion has its own name; the exponential series. The relationship between $e$, $i$, $\pi$ and -1 that we briefly mentioned in the first part of this article can be demonstrated using it, for example.

Of course, thus far we have seen but one example; the mocktal of order 2, $\daleth_\infty$. Is there a family of such curves, or is our $\daleth_\infty$ a solitary fellow?

## A mocktal sinusoid

Focusing on the partial self similarity exhibited by $\daleth_\infty$, I propose that we begin our hunt for another mocktal with a function that displays the same property:

$$f(x) = \frac{1}{2}x \times \left(1 + \cos\frac{2\pi}{x}\right) \qquad x \in [0, 1]$$

Like $\daleth_\infty$, this passes through both (0,0) and (1,1), as illustrated in figure 8.

Now, unfortunately, this curve isn't so easy to measure with fixed length rulers. We can, however, calculate a reasonable approximation of its length the distances between subsequent points in the graph **increase** as we move from left to right, **greatly mitigating** our exposure to loss of precision

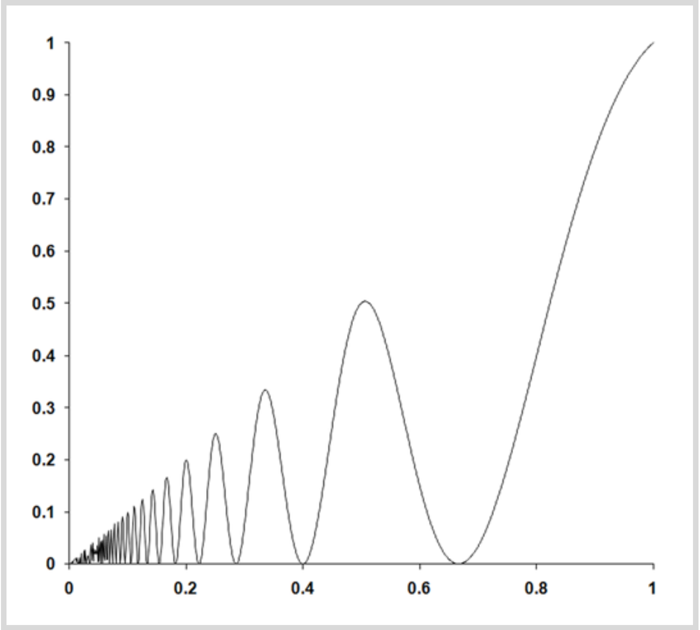


Figure 8

by summing the distances between points on the graph at fixed steps along the *x* axis. Specifically, with

$$l_n = \sum_{i=1}^{2^n} \sqrt{\varepsilon_n^2 + (f(i \times \varepsilon_n) - f((i-1) \times \varepsilon_n))^2}$$

where, once again, $\varepsilon_n$ is equal to 1 divided by $2^n$.

Since we do not have a fixed length ruler in this case, we shall instead use the average distance between the points we use to calculate the length of the curve as an analogue for the ruler length:

$$\frac{1}{2^n} \sum_{i=1}^{2^n} \sqrt{\varepsilon_n^2 + (f(i \times \varepsilon_n) - f((i-1) \times \varepsilon_n))^2} = \varepsilon_n \times l_n$$

Because of this lack of a fixed length ruler, we cannot use our **`accumulator`** to count the number of rulers and so shall have to accept the potential precision issues that might arise during the calculation and settle instead for accumulating the length of the curve with a **`double`**. Fortunately, the distances between subsequent points in the graph increase as we move from left to right, greatly mitigating our exposure to loss of precision.

The code to calculate the length of this sinusoid is a reasonably straightforward adaptation of the **`count_rulers`** function, as illustrated in listing 2.

Plotting the lengths returned by this function for *n* from 1 to 32 against the logarithm of the reciprocal of the average distance between the points quite strikingly reveals the mocktal order of this curve, as illustrated in figure 9.

```
double
sinusoid_length(unsigned long n)
{
  static const int dig =
     std::numeric_limits<unsigned long>::digits;
  static const double pi = 2.0*acos(0.0);

  if(n>dig)  throw std::invalid_argument("");
  const unsigned long upper_bound =
    ((n==dig) ? 0UL : (1UL<<n))-1UL;
  const double step = pow(2.0, -double(n));
  double length = 0.0;
  double prev = 0.0;
  unsigned long i = 0;

  while(i!=upper_bound)
  {
    ++i;
    const double x = double(i)*step;
    const double curr = 0.5*x*(1.0+cos(2.0*pi/x));
    length += sqrt(
       step*step + (curr-prev)*(curr-prev));
    prev = curr;
  }
  length += sqrt(
     step*step + (1.0-prev)*(1.0-prev));
  return length;
}
```

Listing 2

As the ruler reaches its smallest values, the graph approaches a straight line with a slope of approximately 1.05. Hence, I believe we can be reasonably

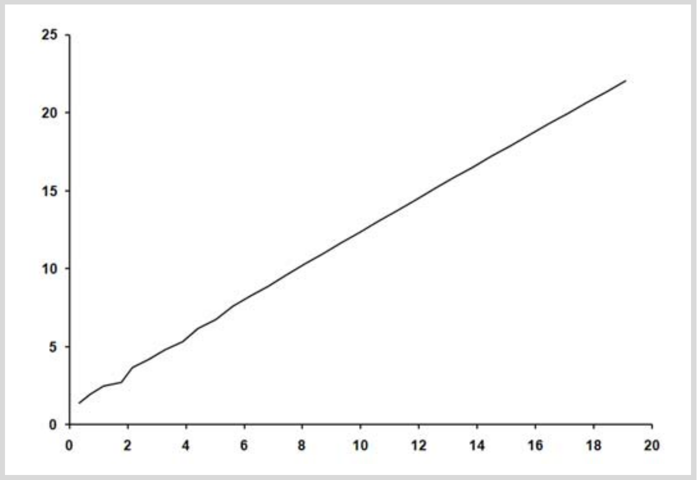


Figure 9

**the infinitesimal numbers have enjoyed something of a resurgence**

confident that this curve has a mocktal order of 1 and consequently falls precisely between a simple curve and ף$_\infty$.

## To be a curve, in the summertime, close to a fractal

The mocktal orders of these two curves are infinitely smaller than that of a true fractal, since fractals have infinite mocktal order, but crucially they are not equal to 0. In this sense, these curves can be thought of as infinitesimally fractal-like.

Infinitesimal numbers were introduced in the original, somewhat hand-waving, definition of the calculus. Defined as numbers smaller than any of the real numbers, but nevertheless greater than 0, their very existence has always been in question. A great deal of effort was expended to define the calculus without them during the 19th century, a time during which the modern, relentlessly rigorous approach to mathematics was adopted.

Those of you who have studied the mathematical subject of analysis will appreciate just how much more convoluted the calculus is without them.

## Non-standard numbers

In more recent times, the infinitesimal numbers have enjoyed something of a resurgence with both the surreal numbers [Knuth74] and the non-standard numbers [Robinson74] providing them with a secure footing.

Just as the real numbers can be represented with an infinite sequence of integers, so the non-standard numbers can be represented with an infinite sequence of reals.

The familiar real numbers are those sequences which, at least after some point in the sequence, have elements forever equal to a given real number.

For example, the non-standard number

$$(1,2,3,\pi,\pi,\pi,\pi,\ldots)$$

is equal to $\pi$.

Similarly, we can strictly order the non-standard numbers by defining one to be less than another if, after some point in both their sequences, the elements of the first are always less than the elements of the second. Strictly speaking, given two non-standard numbers $a$ and $b$ defined by

$$a = (a_0,a_1,a_2,a_3,a_4,...)$$

$$b = (b_0,b_1,b_2,b_3,b_4,...)$$

we say that $a$ is less than $b$ if there is some $n$ for which

$$a_i < b_i \qquad \forall i > n$$

where the upside down A means ‘for all’.

Finally, we define arithmetic operations on the non-standard numbers by applying the operations element by element to the sequences that represent them. For example, adding $a$ and $b$ yields

$$a + b = (a_0+b_0,a_1+b_1,a_2+b_2,a_3+b_3,a_4+b_4,...)$$

Given these definitions we can construct non-standard numbers that are greater than 0, but less than any real number. For example, the sequence

$$\left(1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots\right)$$

consists of elements that are always greater than 0, but growing ever smaller, must after some element always be less than any given real number.

Note that we can also define strictly ordered infinities in the same fashion. For example, inverting the infinitesimal above yields

$$(1,2,4,8,16,...)$$

which by our definitions is greater than any real number.

## Mocktals, fractals, infinities and infinitesimals

If we choose a universal sequence of ruler lengths, $\varepsilon_n$, we can relate straight lines to the non-standard representation of the real numbers since the length of a straight line measured by those rulers is an infinite sequence of identical numbers.

Simple curves, whose length gets closer and closer to some limit as the ruler gets shorter, can be similarly related to non-standard numbers that differ infinitesimally from that limit.

Finally, the mocktals are in this sense equivalent to non-standard infinities ranked by their mocktal order and hence all are smaller than the non-standard infinities to which we can map the fractals.

Dividing any of the mocktal non-standard infinities by any of the fractal non-standard infinites yields an ultimately ever decreasing sequence of numbers which is, by definition, a non-standard infinitesimal.

So there we have it. A mathematically sound mapping by which the mocktals can be described as both infinitesimally fractal-like and yet infinitely more so than simple curves, and an object lesson in the care that must be taken when pushing C++ integers to their very limits to boot.

In parting, I should like to pose the question of whether we can construct curves with any given mocktal order, even non-integer. I haven’t found any more yet, but then I haven’t really looked very hard.

If you do, dear reader, I would be delighted to hear about them. ■

## Acknowledgements

With thanks John Paul Barjaktarevic and Lee Jackson for proof reading this article.

## References and further reading

[Devlin05] Devlin, K., *The Millenium Problems*, Granta Books, 2005.

[Knuth74] Knuth, D., *Surreal Numbers*, Addison-Wesley, 1974.

[Robinson74] Robinson, A., *Non-Standard Analysis*, Princeton University Press, 1974.

# Project-Specific Language Dialects

Today's languages force a one-size-fits-all approach on projects. Yaakov Belch, Sergey Ignatchenko and Dmytro Ivanchykhin suggest a more flexible solution.

*'When I use a word,' Humpty Dumpty said in a rather a scornful tone, 'it means just what I choose it to mean – neither more nor less.'*
Lewis Carroll, Through the Looking Glass

When we started our most recent project, we had quite substantial discussions on the programming languages and libraries to be used and to our surprise found that, despite our very different backgrounds, we all agreed that existing programming languages are rather inadequate for certain aspects of otherwise pretty straightforward programming tasks. In our case, discussion started with rather routine issues like support for serialization and configuration, but eventually extended to much more complicated issues like the ability to write the very same code only once for several different programming languages (like C++ and Java) and compile-time detection of certain types of multithreading bugs.

The most substantial issue we have found in modern mainstream programming languages like C/C++/Java, was that they are mostly languages aiming to instruct a computer what it should do, but what we needed was a language which provides a straightforward and easy way to express our thoughts. While certain thoughts can, indeed, be directly expressed with modern mainstream programming languages, there is still a wide range of thoughts and concepts which require rather artificial, inefficient or outright mindless and repetitious coding to be done. We feel that the worst part about it is that it tends to affect developers' way of thinking, causing a developer to pay more attention to how to get around language/compiler/library problems rather than to think what the program logic should do, with the worst case being the developers' attempts to redefine the original task, and degrading the end-user experience merely because of language or library limitations[1].

Another way to represent the same basic idea is to consider it as a way to shift the burden of repetitive, mundane tasks from developer to compiler, freeing developers' time for more creative work. As a common wisdom says, 'Computers will never do anything really smart for you. But they can do something dumb, freeing you some time to do something really smart'.

As a result of all those discussions, we have tried to find out if there is some way to address the limitations of modern mainstream programming languages while keeping their positive aspects substantially intact, and taking into account certain practical aspects of organization of medium- to large-scale programming projects.

1. The idea that the language used by people can affect the way they think is nothing new, and in non-programming world is known as 'Sapir-Whorf Hypothesis'. While we're not aware of extensive discussions of its applicability to programming languages and developers, some observations implying such applicability were made by Iverson [Iverson79] and Graham [Graham03].

## Current practice – project-specific vocabularies, guidelines and problems resulting from lack of their enforcement

In practice, every medium- to large-scale software project has its own project-specific vocabulary as well as formal or informal usage conventions and guidelines.

Such vocabularies include class hierarchies, APIs, macros and templates. For example, for Apache projects there is an 'Apache Portable Runtime' library, which forms a substantial part of Apache projects' vocabulary, and the 'Linux Kernel API' forms a substantial portion of the vocabulary for writers of Linux drivers. Current programming languages provide means to manage and control such vocabularies.

However, conventions and guidelines for the correct use of project-specific vocabulary are no less important, and existing programming languages usually don't provide much help for managing and controlling them. In an average medium-size software project there are many implicit conventions as well as formal or informal guidelines, which are at best documented (within source code comments or a separate document), and at worst exist as an undocumented bunch of 'everybody knows it' rules which are passed from one generation of developers to another, often only via trial and error. Even if there is the will to enforce those conventions and guidelines, missing support from the programming language easily leads to dilution of those guidelines, in extreme cases up to the point of the whole project becoming one big plate of spaghetti code, with the need to throw it away and restart the whole project from scratch.

For example, if the project is designed to be cross-platform but is compiled only on one platform for the time being, there is usually a guideline 'never ever use platform-specific APIs'. Another typical example of guideline is 'Resource Allocation Is Initialization' (RAII), which aims to reduce/eliminate resource leaks. Unfortunately, as there is no way of enforcing these guidelines, project architects tend to find that the burden of such strict self-discipline proves to be too much for at least some of developers. As a result, in the case of the 'no platform-specific API calls' guideline, architects either need to spend a significant portion of their time to 'police' the code for inappropriate API calls, or find when they want to compile the project elsewhere that it will require major refactoring, up to the point of complete rewrite. Effects of violating 'RAII' guideline are usually less devastating, but still can easily lead to many months spent on isolating and fixing resource leaks. To make things worse, not enforcing guidelines

**Dr. Yaakov Belch** joined Blue Whale Software to turn Sergey's vision of C+– into a reality. He can be contacted at yaakov@yaakovnet.net

**Dmytro Ivanchykhin** is currently working primarily on system-level programming, with a focus on chipping off all unnecessary material. He can be contacted at di@bluewhalesoftware.com

**Sergey Ignatchenko** has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

**Developments in programming languages are very often related to adding more and more features to the language**

leads to new developers seeing code which does not comply with the guidelines, assuming that this code is OK and then using it as a model. The very same thing happens with virtually any non-enforced guideline – it takes significant effort to keep it from being violated, and if this effort is not made, it usually means a downward spiral towards a complete ignoring of the guideline.

Therefore, it seems beneficial to provide some way to enforce project-specific guidelines; we will discuss more detailed requirements for such control below.

## Requirement – need to restrict language features at least in certain cases

Developments in programming languages are very often related to adding more and more features to the language. As one of the most prominent examples, the new programming language 'D' [Alexandrescu09] takes C++ (which is already very far from being simple) and adds an impressive number of new features to it (from garbage collection to closures). Planned developments for the C++ standard [ISO09], while are less impressive in scope, also include a number of new features, including lambda-functions. While we agree that there are some projects which do need those features, we want to show that restricting available language features can be a good thing, at least in some cases.

First of all, let's take a look at the non-programming world. Even at first glance it becomes apparent that excluding certain words from vocabulary can help to keep language clarity; one notable example is 'Seven dirty words' banned on American TV; while their prohibition obviously restricts available vocabulary, it is quite difficult to argue that at least in some contexts agreement on not using them indeed promotes language clarity rather than impeding the expression of thoughts[2].

Now let's see how it applies to programming languages. Take as an example a project designed to be cross-platform and the unenforced guideline 'never ever use platform-specific code' (see above). It seems quite obvious that the ability to enforce this guideline (restricting programmers from using platform-specific language features) would benefit that hypothetical project.

Moreover, from our observations of trends within the industry we have found that, ironically, the more features a programming language provides and the bigger the project it is used for, the more project architects will be cautious and reluctant to adopt it exactly because of increased efforts to enforce guidelines. For example, we see the (in)famous Linus Torvalds' post 'C++ is a horrible language' [Torvalds] as a prominent example of such reluctance of a project architect to move to a language with more features. In particular, Linus wrote: 'You invariably start using the 'nice' library features of the language like STL and Boost and other total and utter crap, that may 'help' you program, but causes... [here follows list of problems]'; we think that if Linus could choose *which* features of C++ to allow into git or Linux kernel and which *not* to allow, he would be *much* less reluctant to allow a feature or two from C++ (but just those 2 features he needs, not more) into Linux kernel or git.

2. Late during review the real-world example of Simplified English was suggested [SE].

## Requirement – need to provide more language features

Another problem of existing project-specific vocabularies is that usually they are just vocabularies, not real languages. It means that basically you can specify 'words' to be used, but cannot really specify the patterns they can belong to. Even for now ubiquitous classes, while you can easily specify acceptable APIs, there is usually no way to enforce at compile-time that, for example, the function **`init()`** must always be called before any other function (excluding the constructor), or that the function **`deinit()`** must always be called right before the destructor. When dealing with features which cannot be easily described in terms of classes, the situation is even worse. For example, let's consider MFC's 'message cracking' which uses macros like **`DECLARE_MESSAGE_MAP`**, **`BEGIN_MESSAGE_MAP`** and **`END_MESSAGE_MAP`**. While it indeed provides a way to define message maps, they are far from being easily readable, and the error messages the compiler gives about malformed message maps tend to be perfectly useless (which is inevitable as compiler operates at the stage after the preprocessor). Add to the mix the not-100%-efficient code it generates (which was the case last time we checked), and you'll get a typical pattern of the effects of emulating a missing language feature without the direct support of the language.

There are many such missing features in different languages, with the obvious examples of reflection and serialization being just the tip of the iceberg.

Adding all of those new features into the same language doesn't look like a good option either. Take serialization, for instance. Some can say, 'wait, languages like Java already have it, so let's just switch to Java', but unfortunately it won't always help. Java indeed provides built-in serialization, but the problem is that it is only *one of many possible serializations*, and if you need, for example, ASN.1 [Larmouth99] serialization (and don't forget about at least the BER and DER variations), or JSON serialization, or even IIOP serialization (not 'RMI over IIOP', but real IIOP) at some point, developers will still need to code it manually (which requires substantial effort even for medium-sized projects). It means that to satisfy all requirements for all possible projects, language will need to provide support for *all* possible serializations, which most likely is not feasible.

## Resolving requirements conflict – a call for cheap creation of project-specific languages

At this point we seem to have two conflicting sets of requirements for the programming language. One set of requirements calls for restricting features, another one asks for adding more and more new features. Fortunately, it seems that there is a way to satisfy both those sets of requirements simultaneously; it is to allow different projects to have their own different languages. While such an option to create project-specific languages has existed for a long time (using tools like YACC), apparently

**Currently, choosing a programming language is basically a 'once and forever' decision, which is made very early in the development process**

the cost of language creation was too high for real-world projects. It means that the way to create such a new project-specific language should be *substantially* cheaper than that of YACC to become usable in practice.

Also we should emphasize that here we're not speaking about *domain-specific* languages, we're speaking about *project-specific* languages, where every single project should have its own programming language (or more likely, programming language dialect). It obviously makes the requirement for development of such a language to be cheap even more important.

## Requirement – support for architect's role and control over features

In practice, in order to succeed in building a software project with more than 2–3 developers, a project usually has one or more project architects. The distinction between architect and developer roles is vital to the success of the project, but unfortunately there is no direct support for such distinction in modern software languages. From our own experience and discussions with project architects within the industry, it becomes quite clear that architects would clearly appreciate having more control over the language features allowed for use in their specific project.

The most important reasons why architects want to control language features are:

- Ability to enforce a common style for the project, reducing potential misunderstandings between team members. The bigger the project, the more likely developers will need to work with code written earlier by some other programmer. Even if some piece of code is trivial to the author, it may well be incomprehensible by others, especially if their coding style is substantially different. And here we don't mean 'style' as the way to indent curly brackets, but rather 'style' as the approach to solving certain types of problems.
- Ability to enforce common requirements for the project, aiming to stimulate a more efficient coding style, where efficiency can be measured in terms of CPU/RAM, bugs or security flaws per thousand lines of code, or development time spent on a certain feature. The problem here is that different projects have very different aims, and no approach works universally well for everybody; eg a coding style which is good for writing a Flash-based Tetris game will probably be devastating for Linux kernel (and vice versa).
- Ability to prevent 'vendor lock-in'. When you start using some non-standard feature of one platform or tool, you may soon be unable to switch to a competing platform or tool. The advantage of this one feature right now may turn out to be much smaller than the advantage of using a more appropriate tool later on.

One of the important aspects of control over language features is the ability to use different sets of rules for different parts of the project. For example, it is fairly obvious that the set of rules and/or guidelines for server-side business logic code will be quite different than those for the UI code within the same project.

It is worth noting that support for the other roles which exist in modern projects (most notably the 'Business Analyst' role) can also be achieved using the same mechanism, creating special dialects easily usable by the target group.

## Requirement – 'Agile programming language'

In recent years, agile software development [Agile01] [Newkirk01] has become more and more popular among software developers, and we think this is no coincidence. One of the biggest reasons for its efficiency is that modern business requirements tend to evolve much more rapidly than the program can possibly be developed, which implies that the ability to react to the changing requirements is extremely important for the success of the project.

We feel that the very same logic should apply to programming languages too. Currently, choosing a programming language is basically a 'once and forever' decision, which is made very early in the development process (essentially this is a 'waterfall' decision with no ability to change it later). Ideally, we think that the project language should be able to evolve as the needs of project grow.

For example, one of us as an architect prefers to start projects with a minimal vocabulary provided to the developers, and then when a requirement for a new feature arises the developer is able to come to the architect and to argue that a certain language feature should be introduced. Our ideal 'Agile programming language' should support this development model (allowing to introduce new language features along the road) and as well should support any other model when the programming language needs to evolve with the project.

Extending our earlier analogy, we should note that language on TV also evolves as the time goes on. Certain things, which were off limits 10 years ago, have become mainstream now, certain words have gone out of circulation, and new words have been invented. The very same process is natural for any successful software project which lives for many years.

## Requirements – industry 'use cases' for project-specific languages

One of the important factors to consider when trying to design something is to understand its potential uses within the target industry. Our preliminary analysis has revealed several areas where project-specific languages could be useful. This analysis was the basis for our programming language discussed later, which (as we hope) should be able to cover all these areas by using language extensions/dialects. These areas include:

### Bigger projects, where keeping the language clean is a significant concern

Actually, we think that almost any project which has more than one developer can benefit from enforcing currently informal (and therefore unenforced) guidelines. Still, usually the bigger the project, the more significant the requirements to keep the code clean tend to become. Ability

**enforcing certain existing guidelines can substantially improve both program security and reliability**

to enforce guidelines will help improve code readability and clarity, while keeping necessary requirements like portability (if it exists) under control, without spending ongoing significant effort on enforcing them.

Examples of requirements for such projects can include:

- Portability: don't use features that are not provided by all target platforms
- Avoiding vendor lock-in
- Abstractions at the proper level and efficient implementation on each platform
- Avoiding constructs which are deemed inefficient by the project architect (as discussed above, definition of efficiency depends on the project, ranging from CPU efficiency to development efficiency)
- Enforcing naming conventions
- Forbidding use of confusing language features (with the list of confusing features being up to project architect)
- Replacing macro-preprocessor and/or templates by more predictable mechanisms
- Replacing pointers with alternative abstractions, like references and arrays. We understand that this item is rather controversial and will probably cause a lot of opposition from existing C/C++ programmers, but as long as it is only an optional feature, we don't see it as a big problem.

### Projects with high requirements for security and/or reliability

We feel that enforcing certain existing guidelines can substantially improve both program security and reliability. Examples of such guidelines include:

- Limiting access to certain resources (like the file system and network)
- Preventing buffer overflows
- Addressing resource leaks
- Preventing at least some kinds of multithreading bugs (which tend to be extremely difficult to find)

### Projects with a need to extend an existing language

It is fairly common that projects are happy with C/C++ or Java, and need just a few minor adjustments to make life easier. Examples of such new features include:

- different serialization mechanisms (from IIOP and JSON to custom storage-optimized or legacy-system-compatible ones)
- Built-in testing support
- Design By Contract
- Introspection
- Nested functions
- Anonymous functions
- Closures and lambda-functions

### Projects which need inter-language portability

Sometimes a project needs to be compilable across multiple languages. Usually it applies to the C++/Java pair, to make sure essentially the same logic can run optimally on both C/C++ platforms like Windows/Linux/Solaris/... and on Java-only platforms like in-browser JVM, Android or BD-J. Achieving this goal will most likely require to use a 'Replace pointers' dialect.

### Projects with a need for user-definable scripts

Usually projects which need user-definable scripts, tend either to invent their own script language, or to use an existing one (such as JavaScript). Ideally though, it often should be a rather close dialect of the very same language within the program itself, and in part allowed to the user. Example requirements for such projects can include:

- Should be easy to learn
- Should be easy to integrate
- Should be easy to transfer features from the 'compiled-into-the-program' domain to 'user-definable' one.
- User dialect might even need to be a weakly typed one

### UI projects

We feel that current state of programming is pretty sad in the field of UI projects. For example, all the UI code written for Apple Cocoa API, is essentially useless for any other platform, and the very same is true for most of the platform-specific APIs (obviously including the Windows API). One can argue that Java provides a good solution for a cross-platform UI, but our understanding of ideal cross-platform UI is much wider then just an ability to run a client-based UI on different platforms.

Within our philosophy that 'language is to express thoughts', we understand portability in much wider sense. We think that in most cases it is indeed possible to create a UI which is suitable not only for an application on an end user's PC, but also for a completely different media.

Example requirements for such projects can include:

- Portability across different platforms
- Portability to use with remote-access protocols like VNC
- Portability between client-based UI and web-based UI
- Portability to text UI where applicable (obviously, you cannot make PhotoShop work in a text-only window, but we feel that the UI to install an OS security upgrade should translate into text easily).

## Summarizing requirements

It seems that now we can summarize requirements for a programming language that will address the issues we have outlined above. This programming language should:

- allow creation of project-specific language dialects, including one of our requirements is to **keep readability** for **users of existing programming languages**

  - ability to restrict certain existing language features;
  - ability to add new language features;
  - ability to apply somewhat different requirements to different subprojects
- have low cost of creation for project-specific language dialects mentioned above;
- be 'agile', allowing ability to create project-specific language dialects as the need arises, not necessarily at the very beginning of the project;
- have explicit (but optional) support for 'Architect'/'Developer' distinction
- keep the positive aspects of existing programming languages;
  - preferably including easy readability for those with experience of existing languages.

In the rest of this article, we will propose a way to address all of those requirements. It is to define a 'basic language' (based on some existing and popular language) and to allow extensions to be written for it *easily* (much more easily than it can be done now with YACC). As the library of such publicly available extensions grows, project architects will be able to choose their project-specific dialect mostly by choosing which extensions to this 'basic language' they want and which ones they don't want; this should make dialect creation even cheaper.

## Consideration – comparing programming languages' popularity

As one of our requirements is to keep readability for users of existing programming languages, we need some data on programming language popularity to understand what kind of syntax is the most popular one (and therefore will be the most easily recognized). We took the popularity of projects on SourceForge [Labelle] as a baseline (adding a new point for 2009, see Figure 1.), and have found that at least over last 8 years, programming languages with C-like syntax[3], were used for at least 80–90% of all the SourceForge projects. This data is also corroborated by independent research [DedaSys]. Therefore, we can safely assume that C-like syntax is quite universally recognized in the industry, and using it as a baseline will have substantial benefits at least because of this universal recognition.

Based on this research, we have decided to use a subset of a C++ (close to 'C with classes', [Stroustrup94]) as our 'basic language', and to name it 'C+–', to show that it provides options to be either more feature-rich, or to be less feature-rich than C or C++. C+– will also allow language extensions (to form language dialects) but they will be restricted to similar syntactic patterns.

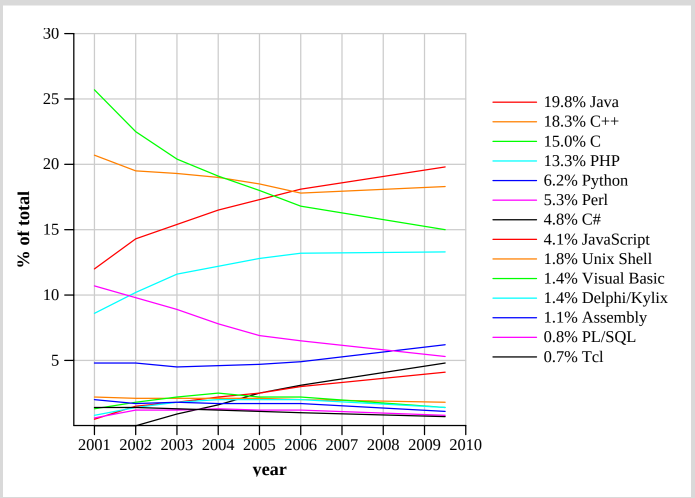


Figure 1

It's important to note that in C+–, we do not support all of the subtleties described in the C++ standard (even the C standard is not 100% implemented, though most of the features can be reinstated using C+– extensions). Instead, in certain situations we require the programmer to use simpler alternative methods to express the same thought. According to our experience, most programmers[4] already tend to avoid most of these complexities and opt for simpler alternatives. Hence, we feel that our limitations don't substantially reduce the usability of C+–. This issue will be discussed in more detail below.

## Consideration – common extension types

Based on the 'Industry use cases' above, we tried to analyze what different types of extensions might be needed, and our analysis has revealed that virtually all language extensions and dialects we could think of fall into one of two classes:

- Limit or forbid use of a certain feature/feature combination in C++. Sometimes, this involves complex program-scale checks to detect such usage (for example, it might involve memory leak detection during compile-time; while it's not always possible, some memory leaks can indeed be detected, with the expected number of cases detected being significantly higher than that of LINT [Kunst88]).
- Copy a proven feature from another language. In some cases, this will require removing conflicting features from C/C++.

Researchers from the field of domain-specific languages may be surprised by the virtual lack of demand for completely new features. This industry inertia may be explained by the following observations:

3. Languages C, C++, C#, Java, PHP, Perl, JavaScript (and many more) all are using common syntactic structure borrowed from classic C: operators with generally accepted precedence levels, nested curly brackets and commonly accepted control structures like `if-else`, `while`, `for` etc.

4. Obviously excluding those who're competing in the 'Obfuscated C Contest' [OCC]

there are already **lots of languages** which '**tell computers what to do**' rather efficiently

- The chances of misunderstandings and inconsistencies grow quickly with project and team size. Enforcing even simple rules in medium to large projects is a much more pressing concern than in small projects.
- Even in the most innovative projects, the innovation is usually contained in a few modules and complemented by a larger amount of existing industry code.
- It is natural for managers to limit the risk of a large project by mainly using ideas and tools that have been tested in smaller projects before. Even when they are not perfect, this experience will help to avoid problems in planning and to resolve problems when they occur.

Technically, C+– does allow completely innovative extensions, but we expect more conservative extensions to dominate.

## Consideration – what to compile to?

Whenever somebody wants to develop new programming language, they usually face a tough question of 'how we're going to compile it on all existing platforms?'.

Fortunately, as we feel that there are already lots of languages which 'tell computers what to do' rather efficiently, we didn't aim to compile C+– code directly into binary code. Instead, we are aiming to compile C+– into the source code of a certain 'target programming language', such as a C/C++/Java; we also aim, where possible, to make this 'target programming language' code be human-readable, and to correspond line-to-line to the original C+– code. This approach has the additional benefit that the same framework can be used to compile C+– code into languages like C and Java, which are similar source-wise but are rather different binary-code-wise. This approach does not preclude us (or anybody else) from developing a 'native' compiler at some point later.

## Implementation – 'Basic C+–', extensions and dialects

C+– is essentially an extensible language, consisting of 'Basic C+–' plus all kinds of different extensions to it. Language extensions can be combined together to form 'C+– dialects', specific to the project. Extensions themselves are also written in one such C+– dialect (specific to the task of writing language extensions).

It is important to note that technically a very wide range of extensions can be created for C+–, with some extensions even breaking the overall 'feel' and readability of C+–. To address this problem and avoid too much dilution of the meaning of 'C+–', we intend to disallow certain extensions from being named 'C+– extensions' (such extensions and dialects will still be possible, but without 'C+–' being attached to the name of the resulting language/dialect). We also intend to discourage different extensions from doing essentially the same thing and encourage authors to consolidate their efforts to avoid unnecessary duplication. This corresponds to our feeling that differences in language dialects should be motivated by different needs, not by a need to differ.

## Implementation – 'Basic C+–' as a subset of existing C/C++

We tried to make Basic C+– more or less 'the least common denominator' of the most popular programming languages; this logic has lead us to making our language rather close to 'C with Classes' [Stroustrup94], but with certain technical incompatibilities with C.

These incompatibilities include:

1.
   ```
   a*b; // error - binary expression as a statement
   ```

2.
   ```
   A x[3][]; // OK
   int (*ptrToFunc)( int, int ) = NULL; // error:
     // "complex" type in variable declaration;
     // need to use the following instead:
   typedef int (*)( int, int ) FuncPtr; // OK
   FuncPtr ptrToFunc = NULL; // OK
   ```

3.
   ```
   A* xx = (A*)x; // error:
    // C-style cast is not supported,
    // need to use the following instead:

   A* xx = c_cast<A*>( x ); // OK, c_cast<>
    // is similar to the C++ *_cast<> family of casts
   ```

4.
   ```
   int x = sizeof(a*b);// error:
     // sizeof(expression) is not supported;
     // only sizeof(type) is supported now
   ```

While we have quite strong feelings about items (1) and (2) and they are unlikely to be introduced later (as we don't feel obligated to support what we feel is a 'cumbersome and obfuscated coding style'), items (3) and (4) can be reinstated at some point if there is enough pressure from the community to do so.

As of now though none are supported by C+–, which made the initial implementation much easier; in particular, *these restrictions allowed the grammar of C+– to be a LALR(1) grammar*, substantially reducing the cost of initial implementation.

Another problem such an extensible language can face is the pollution of the namespace of global keywords with extension-introduced keywords. To address this issue, we plan to impose the following guideline on official C+– extensions (those which can have 'C+–' in their name): any keyword accepted by the parser must either:

- comply with the current C, C++ standard, or
- start with a leading `@`

While exceptions are possible (for example, **`c_cast<>`** is likely to be introduced without `@` to be consistent with the C++-style **`*_cast<>`** family of casts), in general we're going to apply this guideline both to our own and to 3rd-party extensions.

the very next problem developers face is that of **making sure that they didn't forget** to **protect all accesses** to all variables which need to be protected

## Implementation – extension example

C+– aims to achieve the agility and flexibility requirements via the wide use of language extensions. Let us consider one rather simple (but practical) C+– extension.

There is one common problem with multithreaded programming, which C+– can help with. Let's assume that we have a C++ program with the following model of synchronization between threads. There is class **`Mutex`** and class **`Lock`** with constructor **`Lock( Mutex& mx )`**. **`Lock()`** acquires **`mutex mx`** in the constructor, and releases it in the destructor; this simple technique protects developers from forgetting to release **`Mutex`**. But the very next problem developers face is that of making sure that they didn't forget to protect all accesses to all variables which need to be protected, by creating an instance of **`Lock`** for the appropriate **`Mutex`**. In practice, such mistakes can live unnoticed for many months and will manifest themselves at the worst possible time, causing a lot of time to be spent figuring out what went wrong. As the job of checking that all accesses to all relevant variables are protected looks rather mechanical, we will try to write a C+– extension to handle it.

First, let us describe what we want to achieve. We want to create an extension which will allow us to write a modifier

```
@protected_by <mutex_name>
```

for any data member, and it should then become the job of the compiler to check that every function which accesses one of those 'protected_by' variables, has a **`Lock`** object created for the relevant **`Mutex`** (in practice, more sophisticated analysis of the call graph will be necessary, but for the purposes of this article we will restrict the task definition to a single function only).

Then our hypothetical extension **`protected_by`** will look something like Listing 1.

It is rather obvious, that such an extension (even when production-quality code) will not have 100% accuracy in detecting both mistakes and absence of mistakes. It is fairly easy to write code which will make any such static analysis impossible, leaving room for situations for which it cannot possibly be decided for sure if they provide locking or not. Our approach in this (and many similar cases, like detecting memory leaks) is to:

- admit that for any such analysis there are 3 possible outcomes: 'good', 'bad' and 'not sure'
- in general, aim for '100% safe' code, treating 'not sure' the same way as 'bad'. This behaviour can be overridden by the project architect if *really* necessary. Our estimates show that in at least 90% of cases it should be possible to rewrite the code into a 'good' form (as an additional benefit, such a rewrite tends to make the code cleaner). In those rare cases when the code indeed needs to be so complicated that 'not sure' situations are indeed necessary, such code can always be moved to a separate subproject with a different set of restrictions, or in some cases extensions might need to be customized for the specific needs of specific project.

Obviously, many other types of extension are possible within C+–. It includes extensions to add new language constructs like functional-style **`map()`**, **`reduce()`** and **`filter()`**, though the ability to affect operator precedence or introduce new operators is not currently planned both because of the language dilution issues and because of technical complexity.

## Implementation – combining extensions and agility

As we hope, most of the power of C+– extensions will come from the publicly available library of extensions, and project architects will mostly just select a set of features they want for their specific project (or a

```
@extension protected_by {

@additional_node_member
  string protected_by default "";
  // provides us with a data member 'protected_by'
  // in each node of the parsed semantic tree

@data_member_modifier @protected_by IDENTIFIER
  { protected_by = identifier_name( $1 ); }
  // assigning data member defined a few lines
  // above

@data_member_access_hook( Node& node )
{
  Node& decl = find_data_member_declaration( node
);
  if ( decl.protected_by == "" )
    return;

  for ( Node::going_up_code_iterator it =
          node.begin_going_up_code();
        it != node.end_going_up_code(); ++it )
  // going_up_code_iterator goes "up" the code
  // until it encounters function definition
  {
    Node& n = *it;
    if ( n.nodeType() == Node::ObjectDeclaration
      && n.objectType().name() == "Lock"
      && n.nParameters() == 1
      &&
      test_reference_to_data_member_equivalency(
                       n.parameters[ 0 ], node )
    )
    return;
  }

  report_error( ... );
}

};// @extension protected_by
```

Listing 1

project architects will mostly just **select a set of features** they want for their specific project

subproject) forming a project (or subproject) dialect. This creates a very agile language, where certain constructs can be added as easily as by checking a checkbox and recompiling the compiler. Obviously, we cannot hope that 100% of all cases for all projects will be covered by existing extensions, and from time to time some project-specific extensions will need to be written; we still hope that with all the measures we have taken to make writing such extensions simpler, it will still be quite within the abilities of even rather small projects (especially as it will always be possible to start without certain extensions and introduce them later as the need arises).

Such variety of extensions means that the problems of combining extensions will be very important for the future of C+–. Fortunately, it seems that as long as:

- all extensions start with an extension-specific keyword
- extensions are limited either to restricting a single feature, or to introducing a new one, inter-extension interaction will be reduced to a minimum, essentially allowing most extension combinations to be valid. We have about 30 extensions we ourselves would like to have on our list, and almost all of them can be combined with the others easily (with one notable exception being an extension to replace pointers with references and arrays).

In any case, C+– extensions will be checked for incompatibilities as early as possible, and project architects will know that they selected an impossible set of extensions at the stage of selection.

An essential part of C+– is an ability to have different subprojects with different dialects and still be able to compile it all together. To deal with this, C+– will require that each source file starts with a line declaring the language dialect used in this file. The set of available dialects and their names are specified by the project architects.

## Implementation – 'BetterCC'

To enable the writing of C+–, we needed to create a comiler to generate the dialect compiler; we have named it 'BetterCC'.

Basically, BetterCC is a platform to create different languages and dialects, with C+– being just one of a multitude of possible languages. We intend to license BetterCC for free under an open source licence. On the other hand, as it was already mentioned above, we feel that we need to exercise control over C+– extensions to avoid unnecessary language dilution.

BetterCC is implemented using common approaches and consists of the following stages (with extensions allowed to interact with this process via various hooks):

- lexer: split sources into tokens
- parser: detect language structures; build a semantic tree
- resolver: build symbol tables and attach full semantic information to the nodes of the semantic tree
- target writer: convert semantic tree to the target language (e.g. C++)
- target compiler: compile target code to objects or executables.

While some of these stages (e.g. lexer and parser) may interleave in execution time, we avoid as much as possible pushing information back into previous parts of the pipeline. In particular, we do not allow the lexer to read symbol tables to determine the semantics of a non-keyword identifier.

It's interesting to note that most of the projects will involve two very separate runs of the stages above. The first run happens when project architects have defined which extensions they want, and then BetterCC is used to compile these specifications (written in special C+– dialect designed for writing extensions) into a 'project compiler'. The second run happens within this 'project compiler' when developers compile their C+– code (in the dialect defined by project architects) into the target language.

## Implementation – C+– and preprocessor

While we consider the preprocessor as a relatively minor issue, we expect it to be rather frequently asked about, so we'll try to address it quickly here. Basically, we feel that a preprocessor as 'something that runs before the compiler' is not a good thing. On the other hand, we recognize the need for things like conditional compilation. To deal with it, we performed some analysis of existing code, and found that in well-organized and well-disciplined projects developers normally use the preprocessor as a just yet another idiom from the 'project vocabulary'. For example, constructs like

```
#ifdef THREADS
....
#else
....
#endif
```

usually becomes ubiquitous all over the project, to denote 'part of code which is compiled only if we're compiling for multithreaded mode', which makes it essentially an idiom which belongs to the 'project-specific language', rather than a preprocessor trick.

Based on this analysis, we have decided that there will be no preprocessor in C+–, but there will be an extension which will allow for easy creation of language idioms like the one shown above. Among other things, this approach will improve project discipline and also will allow for much stricter enforcement of certain rules at compile-time.

## Implementation – compatibility with existing languages

Obviously, when one starts a new language, there is always an issue of reusing existing libraries written in different languages. In this area we plan to allow a project architect to choose one of two approaches:

- rely on cross-platform and cross-language C+– libraries, ensuring portability, avoiding vendor lock-in, etc. While we plan to provide a certain set of such libraries, C+– will need support from the programming community to make this set of libraries comprehensive enough.

save valuable developer time from working on issues which are purely mechanical

- to reuse existing libraries in existing programming languages. While this approach does not ensure a high degree of portability, and we hope to be able to discourage using it at some point, C+– will need to support it at least for a while. To facilitate it, we plan to have an import tool which will import, for example, C/C++ headers into C+– headers; then, when compiling C+– into C/C++, it will generate source code which is C/C++ and will directly use appropriate C/C++ functions/classes. This approach won't be restricted to C/C++; the very same thing can be done with Java (though compiling to Java as such will require some special extensions to C+–, eliminating pointers from C+–).

## Conclusion

We think that we have managed to find a solution which, while is not absolutely universal, can help both academics and industry with common problems. The advantages of the proposed approach compared to existing programming languages are:

- cheap creation of project-specific language dialects
- 'agility' to easily add/restrict language features as necessary
- explicit (but optional) support for 'Architect'/'Developer' distinction

All of that was achieved without affecting the existing positive aspects of successful existing programming languages too much. We think that we have managed to keep most of the most popular syntax, and most of the concepts, while moving towards the 'a programming language is to express thoughts, not to tell a computer what to do' paradigm. We hope that going this way will save valuable developer time currently spent working on issues which are purely mechanical or can be done mechanically, and start spending it on tasks and algorithms in hand, which we think will be more interesting for most developers and useful from the point of view of end-results too.

Currently we are in the process of implementing these ideas in practice and hope to present the first working implementation of C+– soon. ■

## References

[Agile01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland and Dave Thomas, 'Agile Manifesto', http://agilemanifesto.org/ (2001).

[Alexandrescu09] Andrei Alexandrescu, *The D Programming Language, Rough Cuts*, Addison-Wesley, 2009 [work in progress].

[DedaSys] DedaSys LLC, 'Programming Language Popularity', http://langpop.com/

[Graham03] Paul Graham, 'Five Questions about Language Design', http://www.paulgraham.com/langdes.html (2003).

[ISO09] ISO, *Working Draft, Standard for Programming Language C++*, ISO/IEC, N2914, 2009

[Iverson79] Kenneth E. Iverson, 'Notation as a Tool of Thought', 1979, *ACM Turing Award Lecture*.

[Kunst88] Frans Kunst, 'Lint', a C Program Checker, Vrije Universiteit Amsterdam 1988

[Labelle] François Labelle, 'Programming Language Usage Graph', http://www.cs.berkeley.edu/~flab/languages.html

[Larmouth99] Prof. John Larmouth, *ASN.1 Complete*, Open Systems Solutions, 1999

[Newkirk01] James W. Newkirk, Robert C. Martin, *Extreme Programming in Practice*, Addison-Wesley, 2001.

[OCC] International Obfuscated C Code Contest, http://www.ioccc.org/

[SE] Simplified English: http://en.wikipedia.org/wiki/Simplified_English

[Stroustroup94] Bjarne Stroustroup, *The Design and Evolution of C++*, Addison-Wesley, 1994

[Torvalds] Linus Torvalds, 'C++ is a horrible language', http://thread.gmane.org/gmane.comp.version-control.git/57643/focus=57918

# Quality Matters: A Case Study in Quality

How do we assess quality? Matthew Wilson takes a look at the design of one library.

A few years ago I was tasked with the architecture/design and implementation of a suite of middleware daemons, to arbitrate between the external (point of sale) ingest lines and the processing cores, old and new, for a large Australian insurer. The system uses the AS2805 [AS2805] financial protocol, one of the more arcane and truculent data-exchange protocols I've had the pleasure of working with. It comprises variable-length fields, different character encodings (ASCII and EBCDIC) and Base-64 encoded data [BASE-64]. We hunted around for an open-source implementation of Base-64, but couldn't find anything that met our criteria, so I knocked one up. (Not on the client's time, of course.)

In this instalment, I'll be looking at the design and implementation of the resulting library, **b64** [B64], and will also discuss changes I've been planning to make in light of matters arising in this column.

## Base-64 encoding

Before we look at the library, let's first do a quick refresher on Base-64 encoding/decoding: I'm covering only the basics sufficient to be able to talk about the design of the library; if you want to know all the in-and-outs you'll have to do further reading.

As the name implies, Base-64-encoded data is encoded into 64-possible values from the range character range represented by the string

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

Each value, obviously, requires 6-bits to represent the value. Binary data is encoded in groups of three bytes, totalling 24-bits, into four Base-64 values. This may be shown pictorially (see Figure 1).

For example:

```
0x000000 // => "AAAA"
0x010000 // => "AAAB"
0x190000 // => "AAAZ"
0x330000 // => "AAAz"
0x3d0000 // => "AAA9"
0x3e0000 // => "AAA+"
0x3f0000 // => "AAA/"
0x400000 // => "AABA"
0x000001 // => "AQAA"
```

If the source data does not contain a multiple of three bytes, then one or two 0-bytes are used to pad out the last triple, and `'='` characters used in the encoded output. For example:

```
0x0000   // => 0x000000 => "AAA="
0x0100   // => 0x000100 => "AAE="
0x00     // => 0x000000 => "AA=="
0x01     // => 0x000001 => "AQ=="
```

**Matthew Wilson** is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au

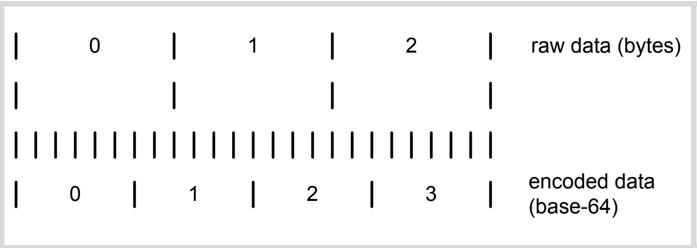


Figure 1

## Perfection?

When I'm talking to developers about software quality, I often use **b64** as an example of near perfection. It scores highly in all the following:

- correctness
- efficiency
- modularity
- portability
- discoverability (though with some deficiencies; discussed later)
- flexibility (in C++ API)
- expressiveness (in C++ API)

(In my opinion, it also scores well in transparency, but I cannot be sure that I'm not biased in that respect.)

Now, before you all accuse me of towering arrogance, I must point out that several of these are an almost inevitable consequence of the simplicity of the problem domain: these are credits by default, to be lost through haste or carelessness, rather than won by skill or insight. (There's also the not-inconsequential fact that I fluffed the first implementation by following too-old RFCs on Base-64 conversion, and so had to release a modified version that still bears the scars to this day; see sidebar.)

Nonetheless, it's a good little library, and its popularity, absent any serious effort at popularisation prior to this article, is justified, as is our looking at how and why it achieves its quality. Let's now consider each in turn, and discuss to what degree they're to be expected and what design decisions can influence their win/loss.

## Design

The design of the library is informed by the following characteristics of Base-64 conversion:

1. Given known input data size, the maximum required output data size can always be calculated accurately.
2. The conversion to/from Base-64 format relies on a fixed, well-known algorithm that is entirely predictable, and free from any configuration or runtime influence.

These characteristics influence the design in several ways, including the implementation of the conversion algorithms, the library API, the way in which correctness (or robustness [QM-2]) will be assured, error indication/handling, even the choice of implementation language.

I have not (yet) heard of an application where **Base-64 conversion** is required to operate at **MIP-blistering speed**

The main design features are:

- The implementation of the core library is in C; a C++ API is provided separately in header-only form
- The library functions do not allocate memory; it must be supplied by the caller
- No diagnostic logging is used in the library; quality is assured by automated testing and (optional) contract enforcement

### Implementation language: C vs C++

The middleware suite was mainly C++, with a few modules written in C. All the parts that would touch Base-64 were C++. Despite this, I chose to implement it in C. As I've said before [!C^C++], absent any strong reasons in favour of C++ – such as the need to use containers in the implementation – I believe that C should be the default choice of implementation language. There are many reasons for this, dependent on the type of software entity being considered. In this case, the choice was based on portability and on need.

By implementing in C there's a modest advantage in portability. By presenting a C API, there's a considerable increase in the potential client applications and languages, including those written in C, C++, D, .NET, to name a few.

As far as need, there's just no clear (to me anyway) requirement to implement such conversion logic in C++. This may be because I have a bias against unnecessary use of C++, and prefer C APIs with C++ wrappers over pure C++ libraries whenever appropriate; if any readers can think of an advantage to writing **b64** in C++ I'd be genuinely interested to hear it.

The choice of language has other impacts. One of the more obvious ones is that we don't have to worry about exceptions; we'll see how return codes are handled shortly.

### Memory allocation

In cases where one may choose, the choice between having a library allocate memory and requiring users to provide memory depend on a number of factors, including:

- Whether the amount of memory to be used by the callee can be accurately predicted/determined by the caller
- For how long the memory may be required by the caller

This is also complicated by factors such as ensuring memory is allocated and released by the same memory pool/allocator [IC++].

In the case of Base-64 conversion, we've already noted that the amount of memory can be predicted by the caller based on the size of the data to calculate. The **b64** API helps out here, in returning the maximum required size to the caller if they pass `NULL` for the destination to any encode/decode method (and does so in O(1)). Thus, in C, you might see a call sequence such as in Listing 1.

```
size_t n = b64_encode(src, srcLen, NULL, 0);
// get max required size
void*  p = malloc(n);
. . . // handle allocation failure here
n = b64_encode(src, srcLen, p, n);
. . .
```

Listing 1

At face value, this complicates the life of the client coder: at least it appears to add more code. However, by removing memory allocation entirely from the library, the responsibilities are totally clear. And, even better, we can implement **b64** entirely without dependencies on any other functions (including those of the standard library), which means that, discounting cosmic rays in the processor, its ability to perform its task correctly is dependent only on the code within.

Readers who managed to stay awake to the end of the last instalment should recognise the significance of this. When we factor in that Base-64 conversion is deterministic, and therefore highly amenable to automated testing, we realise that **b64** is a software library for which we can demonstrate correctness (as opposed to robustness): a highly desirable characteristic of software.

(And, for C++ clients at least, we can and do provide useful, expressive wrappers that minimise/remove the intrusion on the transparency of client code, as I'll discuss later.)

### Quality assurance

Although I have not (yet) heard of an application where Base-64 conversion is required to operate at MIP-blistering speed, it does seem kind of obvious that it shouldn't have much fat in it. Consequently, one original design decision – still upheld in my recent updates – is that it should not have diagnostic logging. Of course, performance is not the only, or even the most significant, factor in determining whether a software entity should include diagnostic logging. (And if you believe the propaganda of, er, me

### Version 1 API

In my haste to implement a version of the library that would meet the requirements of our commercial work, I, er, forgot to read the RFC specs properly. One part of the MIME encoding requirements is that a Base-64-encoded sequence be broken up with newline sequences (`"\r\n"`) such that each line have a maximum 76 characters (not including the newline). Other and, if memory serves, earlier specifications require maximums of 64 and 1000 characters.

Consequently, the first version consisted solely of a pair of API functions:

```
size_t b64_encode(void const *src,
   size_t srcSize, char *dest, size_t destLen);
size_t b64_decode(char const *src,
   size_t srcLen, void *dest, size_t destSize);
```

Each takes only two pairs of parameters, one for input data and one for output data. Only later, when the error of my ways was pointed out by the user Adam McLaurin, did I have to amend the API to what is described in the main text. Doh!

Adding diagnostic logging is simply a waste of time here

– in respect of the Pantheios diagnostic logging API library, anyway – you don’t even need to sacrifice performance to have logging statements compiled into released code. But that’s for another day …)

Back on point: the major factor at play here is that the algorithms are self-contained and deterministic. For a given input, we must have a given output, regardless of whether we’re running on a mainframe, Windows PC, Linux quad-core server, iPhone, whatever. Adding diagnostic logging is simply a waste of time here, because the diagnostic logging that will be in the application in the code that calls into **b64** will be sufficient to (i) record its failure, and (ii) record the precipitating input data; the latter can be replayed into the library at any later time to test the veracity of the expected result.

## The b64 C API

The **b64** C API consists of two enumerations and six functions, enclosed (in C++ compilation) in the `b64` namespace. Listing 2 shows a truncated form of the `b64/b64.h` header, including all the essential elements.

```
/* b64/b64.h */
#include <stddef.h>

#if !defined(B64_NO_NAMESPACE) && \
    !defined(__cplusplus)
# define B64_NO_NAMESPACE
#endif /* !B64_NO_NAMESPACE && !__cplusplus */

#ifndef B64_NO_NAMESPACE
namespace b64
{
#endif /* !B64_NO_NAMESPACE */

typedef char b64_char_t;

enum B64_RC
{
    B64_RC_OK
  , B64_RC_INSUFFICIENT_BUFFER
  , B64_RC_DATA_ERROR
};
#ifndef __cplusplus
typedef enum B64_RC B64_RC;
#endif /* !__cplusplus */
enum B64_FLAGS
{
    B64_F_LINE_LEN_USE_PARAM    = 0x0000
  , B64_F_LINE_LEN_INFINITE     = 0x0001
  , B64_F_LINE_LEN_64           = 0x0002
  , B64_F_LINE_LEN_76           = 0x0003
  , B64_F_LINE_LEN_MASK         = 0x000f
  , B64_F_STOP_ON_NOTHING       = 0x0000
```

Listing 2

```
  , B64_F_STOP_ON_UNKNOWN_CHAR  = 0x0100
  , B64_F_STOP_ON_UNEXPECTED_WS = 0x0200
  , B64_F_STOP_ON_BAD_CHAR      = 0x0300
};
#ifndef __cplusplus
typedef enum B64_FLAGS  B64_FLAGS;
#endif /* !__cplusplus */

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

size_t b64_encode(
  void const* src
, size_t      srcSize
, b64_char_t* dest
, size_t      destLen
);
size_t b64_encode2(
  void const* src
, size_t      srcSize
, b64_char_t* dest
, size_t      destLen
, unsigned    flags
, int         lineLen /* = 0 */
, B64_RC*     rc      /* = NULL */
);
size_t b64_decode(
  b64_char_t const* src
, size_t            srcLen
, void*             dest
, size_t            destSize
);
size_t b64_decode2(
  b64_char_t const*   src
, size_t              srcLen
, void*               dest
, size_t              destSize
, unsigned            flags
, b64_char_t const**  badChar /* = NULL */
, B64_RC*             rc      /* = NULL */
);
char const *b64_getErrorString(B64_RC code);
size_t b64_getErrorStringLength(B64_RC code);

#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
#ifndef B64_NO_NAMESPACE
} /* namespace b64 */
#endif /* !B64_NO_NAMESPACE */
```

Listing 2 (cont’d)

**Strangely, for me, I've never actually tested its performance**

There are several aspects to note:

- The API has include-dependency only on `stddef.h`, which will be available on every compiler you come across.
- If compiling for C, the namespace is suppressed. Otherwise, unless **`B64_NO_NAMESPACE`** is defined, all elements will be within the **`b64`** namespace, according to best practice.
- The **`b64_char_t`** typedef is used in case a widestring port will ever be required. (This hasn't come up yet.)
- **`B64_RC`** is a value enumeration, representing result codes; **`B64_FLAGS`** is a flags enumeration, used for moderating the behaviour of encoding/decoding. The use of typedefs on the enumeration names ensures that the same names are available in C and C++, and the qualifying prefix **`enum`** is not required in C [!C^C++].
- The 2-variants cater to the needs of the RFC specifications in working with newline-split input/output encoded sequences.
- The aforementioned mistake is evident in the presence of **`b64_encode()`**/**`b64_decode()`**, which are actually implemented in terms of the corresponding 2-variants. More seriously, it means that the return type of the 2-variants serves double duty, indicating the number of elements (to be) converted and, by returning **`0`**, that an error may have occurred, in which case the caller must check the out-parameter **`rc`**. Although we'll see later that with the C++ API implementation that this becomes unimportant, it is nonetheless a detraction from discoverability for users of the C API. (-1 point here.)
- Conversion of **`B64_RC`** to string form (and length thereof) are provided for using **`B64_RC`** with a diagnostic logging library (such as, er, Pantheios); see [STRERROR] for details of how this works.

Let's briefly examine my claims regarding the software quality characteristics, before we move on to consider the implementation and the C++ API.

## Correctness

This is established by automated tests, which come with the library and are built and executed by **`"make test"`**.

## Efficiency

Strangely, for me, I've never actually tested its performance. Since it allocates no memory, uses static lookup tables for converting between encoded and unencoded form, and doesn't do anything more than once, I've simply assumed that it has good performance. Were performance a major criterion, or had anyone ever reported an issue, I guess I would do so, but absent that, why bother?

## Modularity

Here's where **`b64`** rises above just about any other non-trivial library: it has 100% modularity in a release build; in debug form it relies on the runtime's **`assert()`** support and a couple of string functions. Obviously, the acclaim for this is largely due to the characteristics of the problem area, which then enable the choice to eschew memory allocation.

## Portability

The implementation relies only on the standard headers `string.h` and `assert.h` (in addition to **`b64`**/`b64.h`), so we can also claim that **b64** also has near perfect portability. In this case, it is a by-product of the problem area; one would have to try hard to introduce non-portability. The only non-portable aspect is the use of English result code strings, but this is not a true break of localisation, as discussed in [STRERROR].

## Expressiveness

The C API does no better or worse in terms of expressiveness than any such API could, which is to say not very much. As we'll see shortly, expressiveness is dealt with in the C++ API, commensurate with the (my, anyway) hypothesis that the major reason people choose C++ over C (or any one language over another) is expressiveness.

## Flexibility

The C API is not flexible at all, since few C APIs are able to offer any bona fide flexibility, and those that do – e.g. the variadics – tend to be inherently unsafe. Again, the C++ API takes care of flexibility.

## Discoverability

Other than the result-code-as-out-parameter ugliness already discussed, I contend that the API is largely discoverable. Ignoring the extensive per-function documentation (not shown), the API functions are self-documenting in the following ways:

- The enumerators for both enumerations are pretty self-explanatory. A function might succeed, or fail due to insufficient buffer or data error (i.e. asking **`b64_decode2()`** to decode arbitrary text data consisting of non-Base-64 characters). Encoding line length can be a fixed infinite (no newline), 64 or 76, or a user specified length. Decoding can be asked to stop on an unknown character, and/or on unexpected whitespace, or simply ignore bad characters and proceed.
- The encoding/decoding functions represent input and output memory ranges as matched pairs of length+pointer. This leaves only three parameters to grok. **`flags`** should be obvious: a combination of **`B64_FLAGS`**. **`rc`** is an optional (as denoted by the comment) pointer to a variable to receive the return code. **`badChar`** is an optional (as denoted by the comment) pointer to a (non-mutating) pointer to a character, which will be set to refer to the bad character that stops the decoding. **`lineLen`** specifies the required maximum line length, but will be ignored unless

```
B64_F_LINE_LEN_USE_PARAM
     == (flags & B64_F_LINE_LEN_MASK).
```

The things you can glean only from the documentation are:

- If you specify a lineLen of less than 0 (when **`B64_F_LINE_LEN_USE_PARAM == (flags & B64_F_LINE_LEN_MASK)`**), the line length defaults to 64.
- If you specify a **`lineLen`** of 0, you get an infinite line length.
- If you specify **`NULL`** for dest, the function returns the maximum possible amount of memory (characters or bytes) required.

### Transparency

In the previous instalment [QM-2] I gave a sample of the implementation of **`b64_encode2()`**, from which you may make your own judgements. In researching this instalment I had occasion to go back and try and understand the implementation. Since the last modification of any significance was done nearly a year ago, I therefore experienced the transparency anew. Acknowledging a few minutes to acclimatise, I was able to find my way pretty well. Of course, transparency is a very subjective thing – coding styles (layout, naming), choice of idioms, and so on all influence – and I'm the author! It's quite conceivable you might not find the implementation transparent and the only thing I can say for certain is that it's one of the more transparent implementation that I've written.

## The b64 C++ API

I don't have space here to include the full implementation of the library. As mentioned above, the essential elements from **`b64_encode2()`** were shown in the last instalment [QM-2]; and you can download the library to look at it if you wish. In any case it's not really necessary. Given the design decisions and the API outlined above, I think any competent C programmer could produce a correct, portable, efficient and (reasonably) transparent implementation.

What I think is of interest is in how the low-level C API can be mapped into C++ with a good dose of expressiveness and flexibility. In my opinion, the biggest issue C++ users will have with the C API are:

- The need to manually manage memory
- The abstraction dissonance [QM-1, Monolith] encountered when having to work with 'raw' types such as ranges of characters and bytes

Both of these are exemplified by what I consider would be the common types – **`std::string`** and **`std::vector<unsigned char>`** – and preferred usage patterns of the library in C++ code:

```
typedef std::vector<unsigned char> bytes_t;

bytes_t     raw     = . . .
// read in from somewhere

std::string encoded = AcmeBase64::encode(raw);
```

**b64**'s C API cannot support this. Instead, the user will be forced to write something along the (possibly pseudo-code) lines in Listing 3.

Truly hideous stuff. Thankfully, **b64**'s C++ API does not require this of the poor programmer. Consider the simplified listing of `b64/b64.hpp` in Listing 4.

Once again, there are several points to note:

- The API function names reflect the fact that they're inhabiting the **`b64`** namespace, and dispense with the **`b64_`**-prefix of the C API. Users now write the clearly digestible **`b64::encode()`** and **`b64::decode()`**.
- The API raises the level of abstraction and deals with strings and "blobs" (sequence of bytes), rather than size+pointer pairs.
- The units of currency for string and "blob" types are assumed, respectively, to be **`std::string`** and **`std::vector<unsigned char>`**. But it's flexible in two ways. First, custom types may be specified at compile-time, via definition of the appropriate pre-processor symbols, for string and/or blob. Second, the use of string

```
b64::B64_RC rc;
size_t n = b64::b64_encode2(&raw[0]. raw.size(),
   NULL, 0, 0, 76, &rc);
if(0 == n && b64::B64_RC_OK != rc)
{
  throw std::runtime_error(
     b64::b64_getErrorString(rc));
}
std::string encoded;
encoded.reserve(n);
// don't lose "buff" in an assign()-throw
char*       buff = new char[n];
n = n = b64::b64_encode2(&raw[0], raw.size(),
   buff, n, 0, 76, &rc);
encoded.assign(buff, n);
delete [] buff;
```

Listing 3

```
#include <b64/b64.h>
#include <stlsoft/stlsoft.h>

#if defined(B64_USE_CUSTOM_STRING)
# include B64_CUSTOM_STRING_INCLUDE
#else /* B64_USE_CUSTOM_STRING */
# include <string>
#endif /* !B64_USE_CUSTOM_STRING */

#if defined(B64_USE_CUSTOM_VECTOR)
# include B64_CUSTOM_VECTOR_INCLUDE
#else /* B64_USE_CUSTOM_VECTOR */
# include <vector>
#endif /* !B64_USE_CUSTOM_VECTOR */

#include <stlsoft/memory/auto_buffer.hpp>
#include <stlsoft/shims/access/string.hpp>
#include <stdexcept>

#ifndef B64_NO_NAMESPACE
namespace b64
{
#endif /* !B64_NO_NAMESPACE */

class coding_exception
  : public std::runtime_error
{
public:
  coding_exception(B64_RC rc, char const*
badChar);
public:
  B64_RC      get_rc() const;
  char const* get_badChar() const;
private:
  B64_RC      m_rc;
  char const* m_badChar;
};

#if defined(B64_USE_CUSTOM_STRING)
typedef B64_CUSTOM_STRING_TYPE      string_t;
#else /* B64_USE_CUSTOM_STRING */
typedef std::string                 string_t;
#endif /* !B64_USE_CUSTOM_STRING */

#if defined(B64_USE_CUSTOM_VECTOR)
typedef B64_CUSTOM_BLOB_TYPE        blob_t;
#else /* B64_USE_CUSTOM_VECTOR */
typedef std::vector<unsigned char>  blob_t;
#endif /* !B64_USE_CUSTOM_VECTOR */
```

Listing 4

```
inline string_t encode(void const* src,
   size_t srcSize, unsigned flags,
   int lineLen = 0, B64_RC* rc = NULL)
{
  typedef
     stlsoft::auto_buffer<char, 1024> buffer_t;

  B64_RC rc_;
  if(NULL == rc)
  {
    rc = &rc_;
  }
  size_t   n = b64_encode2(src, srcSize, NULL, 0,
     flags, lineLen, rc);
  buffer_t buffer(n);
  size_t   n2  = b64_encode2(src, srcSize,
     &buffer[0], buffer.size(), flags, lineLen,
     rc);
  string_t s(&buffer[0], n2);
  if( 0 != srcSize &&
      0 == n2 &&
      rc == &rc_)
  {
    throw coding_exception(*rc, NULL);
  }
  return s;
}
inline string_t encode(void const* src,
   size_t srcSize)
{
  return encode(src, srcSize, 0, 0, NULL);
}
template <typename T, size_t N>
inline string_t encode(T (&ar)[N])
{
  return encode(&ar[0], sizeof(T) * N);
}
inline string_t encode(blob_t const &blob)
{
  return encode(blob.empty() ? NULL : &blob[0],
     blob.size());
}
inline string_t encode(blob_t const &blob,
    unsigned flags, int lineLen = 0,
    B64_RC* rc = NULL)
{
  return encode(blob.empty() ? NULL : &blob[0],
     blob.size(), flags, lineLen, rc);
}


inline blob_t decode(char const* src,
   size_t srcLen, unsigned flags,
   char const** badChar = NULL,
   B64_RC* rc = NULL)
{
  B64_RC      rc_;
  char const* badChar_;
  if(NULL == rc)
  {
    rc = &rc_;
  }
  if(NULL == badChar)
  {
    badChar = &badChar_;
  }
  size_t  n = b64_decode2(src, srcLen, NULL, 0,
     flags, badChar, rc);
```

Listing 4 (cont'd)

```
  blob_t  v(n);
  size_t  n2  = v.empty() ? 0 : b64_decode2(src,
     srcLen, &v[0], v.size(), flags, badChar,
     rc);
  v.resize(n2);
  if( 0 != srcLen &&
      0 == n2 &&
      rc == &rc_)
  {
    throw coding_exception(*rc,
       (badChar == &badChar_) ? *badChar : NULL);
  }
  return v;
}
inline blob_t decode(char const* src,
   size_t srcLen)
{
  return decode(src, srcLen, 0, NULL, NULL);
}
template <class S>
inline blob_t decode(S const &str)
{
  return decode(stlsoft::c_str_data_a(str),
     stlsoft::c_str_len_a(str));
}
inline blob_t decode(string_t const &str,
   unsigned flags = 0)
{
  return decode(stlsoft::c_str_data_a(str),
     stlsoft::c_str_len_a(str), flags, NULL,
     NULL);
}
inline blob_t decode(string_t const &str,
   unsigned flags, char const** badChar,
   B64_RC* rc = NULL)
{
  return decode(stlsoft::c_str_data_a(str),
     stlsoft::c_str_len_a(str), flags, badChar,
     rc);
}

#ifndef B64_NO_NAMESPACE
} /* namespace b64 */
#endif /* !B64_NO_NAMESPACE */
```

Listing 4 (cont'd)

access shims [IC++, XSTLv1, FF-2] means that **`b64::decode()`** may be called on instances of arbitrary string types

- Memory allocation is handled within the C++ API entirely, so users need neither concern themselves with (de-)allocating the memory nor detecting failure to allocate it (at the level of their client code).
- The library reports coding errors via exceptions if users elect not to receive the errors (by passing an address of a **`B64_RC`** variable).
- The functions handle all the issues of standards conformance, such as the fact that one cannot invoke the subscript operator on an empty **`std::vector`**, nor assume the contiguity of storage in **`std::string`**. Such things have a habit of leaking from one's (sub-)conscious onto the screen, as actually just happened to me in the pseudo-code at the start of this section!
- On reflection, the inclusion of the literal array overload of **`encode()`** is probably gratuitous; more a 'look what I can do' than 'we need this'. However, if anyone can think of a real scenario for this, please let me know.

I contend that the discoverability of these C++ API functions is good, and that the transparency is also reasonably good (assuming one understands the notion and purpose of string access shims). But where it really shines

```
unsigned char bytes0[] = { 0, 1, 2, 3, 4, 5, 6,
   7, 8 9 };
void*         bytes1   = . . .
b64::blob_t   bytes2   = . . .
// aka std::vector<unsigned char>

b64::string_t encoded0 = b64::encode(bytes0);
// aka std::string
b64::string_t encoded1 = b64::encode(bytes1);
b64::string_t encoded2 = b64::encode(bytes2);
```

Listing 5

is in the modest increase in flexibility and the substantial increase in expressiveness. We can now write code such as Listing 5 and Listing 6.

## New 'software quality' changes

When, some months ago, I'd planned an instalment of this column that would discuss **b64**, I'd planned to include an evolution of the library implementation to include the (removable) diagnostic measures [QM-1]:

- Code Coverage, via the xCover library [XCOVER]
- Contract Enforcement, via the xContract library

However, now I've got this far, I realise that I shouldn't bother. This is not, as the Overload editorial team may suspect, due to the fact that I've overrun the submission deadline for the third instalment in a row. Rather, it's because I realise that neither of these measures are necessary for **b64**. (I warned you when we started that this is a learning experience for me as well.)

First, the code coverage. The implementation of encoding has three branches and four loops. Decoding has nine branches and one loop. Although nine could be argued as a decent amount of complexity, it's the case that each branch (and loop) can be determined by visual inspection to be part of behaviour already established (by testing) as correct. Now, it's presumption bordering on hubris to say 'the requirements' won't change, or 'my project's not complex enough to need [insert useful software quality assurance technique requiring a modicum of effort here]'. So I won't claim either of those. What I will claim, however, is that, given the library has a pretty comprehensive test suite and has been in use for several years without a report of defective conversion, it's not worth the cost of introducing code coverage. As well as introducing coupling to other (admittedly open-source, and bundleable) libraries, it also introduces more choices to users of the libraries, and choice is the enemy of discoverability. Furthermore, it'd also mean more effort for me in teasing out a bit more sophistication from my already moribund, kill-yourself-rather-than-change-anything, ripe-for-rewrite makefile generator.

```
char          encoded0[] = { 'A', 'A', 'A', 'A' };
std::string   encoded1("AAE=");
char const*   encoded2   = "AQAA";

b64::blob_t   decoded0 = b64::decode(encoded0);
b64::blob_t   decoded1 = b64::decode(encoded1);
b64::blob_t   decoded2 = b64::decode(encoded2);
```

Listing 6

Things are a little less clear cut with contract enforcement. The library already has contract enforcement implemented in the 'usual way', which is to say via the C standard library's `assert()` macro. For sure, this only applies in debug builds, but contract enforcement is a removable diagnostic measure, so that's perfectly legitimate. It's not always the wisest choice to elide contract enforcements from release builds; I'll cover this a future instalment of the column, dedicated to contract programming principles and practice. But in this case, I'm again inclined to live with the status quo for the same reasons as for code coverage: **b64** has a lot of tests; it has an established history of correctness; I don't want to burden users with more choices / build hassles; I don't want to burden myself with more work for no tangible benefit.

It may seem a little perverse to be writing a column about the practical application of software quality assurance measures, and then not actually apply any. But we're talking about removable diagnostic measures, and perhaps it's useful to start out by abstinence just to ram the removability point home.

But fear not, earnest fellow appliers of software quality measures, these things will soon be introduced to a library near you! In fact, for all the reasons that **b64** does not qualify for the various (removable) diagnostic measures – logging, code coverage, contract enforcement – I have a library that needs them for the same reasons, **recls**. I've recently released the first 100%-X rewrite – a 100% .NET implementation [RECLS100.NET] – and am also planning to release a reworked version of the main C/C++ library very soon. So, the next instalment will probably contain an examination – theoretical and practical – of how I go about adding diagnostic measures to **recls** in C and C++. ■

## 'Quality Matters' online

I just want to let you know that I'm in the process of setting up a website for the column – at http://www.quality-matters-to.us/ – and hope to include blog and/or discussion forum(s) in the non-too-distant future. To start with, the website will contain definitions given in the column instalments, and useful links (including to the article instalments on the Overload site), and I invite you to check it out. (Even better, offer to do the design, and save us all from my hideous lack of visual creativity!)

## References and asides

[AS2805] http://en.wikipedia.org/wiki/AS_2805
[BASE-64] http://wikipedia.org/wiki/Base64
[B64] http://synesis.com.au/software/b64/
[!C^C++] '!(C ^ C++)', Matthew Wilson, *CVu*, November 2008
[STRERROR] 'Safe and Efficient Error Information', Matthew Wilson, *CVu*, July 2009
[IC++] *Imperfect C++*, Matthew Wilson, Addison-Wesley, 2004
[QM-2] 'Quality Matters, Part 1: Correctness, Robustness and Reliability', Matthew Wilson, *Overload 93*, October 2009
[QM-1] 'Quality Matters, Part 1: Introductions, and Nomenclature', Matthew Wilson, *Overload 92*, August 2009
[Monolith] http://breakingupthemonolith.com/
[RECLS100.NET] recls 100% .NET, Matthew Wilson, *Dr Dobb's Journal*, November 2009; http://www.ddj.com/cpp/221900083