The magazine of the ACCU

www.accu.org

{cvu}

Volume 24 • Issue 5 • November 2012 • £3

{cvu}

Volume 24 Issue 5
November 2012
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Pete Goodliffe, Paul Grenyer, Huw Lloyd, Roger Orr, Richard Polton, Mark Radford

ACCU Chair
Alan Griffiths
chair@accu.org

ACCU Secretary
Giovanni Asproni
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

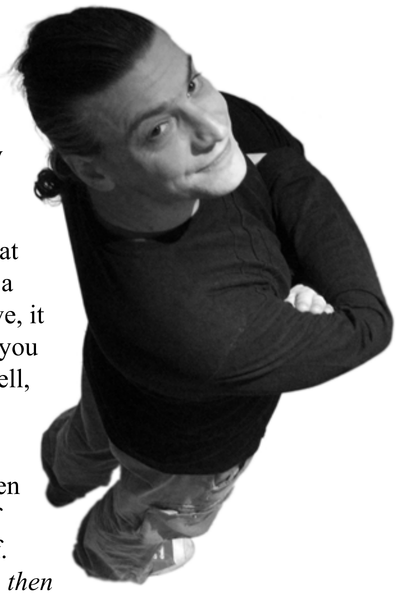
Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

Not Doing The Wrong Thing



It just goes to show, it doesn't matter how much experience you think you have, how much practice you get, how good your intentions are or how quick the hack you want to achieve is, cutting corners just doesn't pay. It's all very well telling yourself you're embarking on a throw-away scratch project with no future beyond exploring some idea or new technology. You might justify your deliberate lack of attention to detail by convincing yourself you're 'writing one to throw away' and will do it properly in the next phase. You can pretend all you like that you get enough practice at doing it the right way, and that a quick hack won't hurt. However much experience you have, it doesn't count for much if you don't put it into practice. If you consider yourself to have quite a lot of experience then, well, you really ought to know better...

Such is the diatribe I recently inflicted on myself for not writing a bunch of tests up front. Not only that, but not even structuring the code in the project with the idea in mind of being able to test it. "It's just a play project," I told myself. "Once I've explored the ideas and got *something* working, *then* I'll do it properly." Yeah. Right.

It doesn't take long before the play project has morphed into the outline of something a bit more real, that does real stuff, has a database, front end, a load of logic...and no tests. All of a sudden the realisation hits: if only you had a way of exploring why this bit didn't work without having to run the whole thing up, add some dummy data, click buttons in the UI and then watch the results in a debugger (or worse, adding a load of output to the console in order to diagnose it), you'd save a whole load of time, effort and electrical current, and be much calmer and more fulfilled.

And so, after quite a bit of time refactoring, and reminding yourself of the identity of the muppet responsible for this rubbish in the first place, you finally add some tests, discover a few things you thought **did** work but in fact didn't, and reflect that you probably don't need any more practice at doing it wrong, but you do need more practice at doing it right. First time round.

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of 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.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

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

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 24.6:	1st December 2012
C Vu 25.1:	1st February 2013
Overload 113:	1st January 2013
Overload 114:	1st March 2013

WRITE FOR C VU

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

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

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

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

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

COPYRIGHTS AND TRADE MARKS

Using the Unix ptrace API

Roger Orr exposes the Unix debugging API.

In a previous issue of *CVu* (Vol 23 Issue 1) I wrote about the Windows debugging API and promised to write a future article about the Unix debug API. Here is that article: slightly later than anticipated.

Many programmers spend a significant amount of time using debuggers but in my experience few of them have much understanding of how they work. It can be useful to look at this, for a couple of reasons:

- knowing a little about how something works can help you use it better
- you might wish to write your own tool to make use of the same API

The classic mechanism for debugging application programs in Unix is based around the **ptrace** function. This function allows the controlling process to perform a variety of tasks on the target process: for example to read and write memory, to examine and change the registers and to receive notifications of signals received by the target process. This forms the basis of the interactive debuggers **dbx** and **gdb**. Additionally Unix can invoke the **ptrace** interface whenever the target process makes a system call. The Unix **strace** utility uses this mechanism to intercept and record the system calls which are called by a process, together with the name of each system call, its arguments and its return value.

There are a number of problems with this API – it was originally designed a long time ago and some of its features have become slightly problematic since. Moreover, each flavour of Unix has made slightly different implementation decisions and often also added extensions to the basic API. There seem to be regular attempts to produce a new API without some of the peculiarities of the existing implementations, but given the widespread availability of the API and the existence of many cross-platform tools which use **ptrace** (albeit often containing platform-specific code to handle the differences) there has not yet been a clear successor. Solaris, for instance, has added other debugging and profiling mechanisms which they consider better and removed **ptrace** as a system call – but the function is still available as it is emulated using the newer functionality.

What's in the Unix debugging API?

Debugging is invoked when a process calls **ptrace** with the **PTRACE_TRACEME** argument (i.e. requesting to be debugged) or a process can try to request debugging of a target process by calling **ptrace** with the **PTRACE_ATTACH** argument. Only one process at a time can attach to a process so this call will fail if a debugger is already attached. The use of **PTRACE_ATTACH** may be restricted by security policies on various versions of Unix – typically it only works if the target process has the same user id but stronger restrictions can be – and often are – available. For example, on Mac OS X a process can use the **PTRACE_DENY_ATTACH** function code to prevent it from being debugged. This is one of the places where the age of the API shows – the security concerns often a necessary part of today's computing world were largely absent when **ptrace** was designed.

Once successfully attached to the target the debugging process typically runs in a loop, calling one of the **wait** functions to receive the next status change of the target and then using **ptrace** to inspect and/or modify the target process and then to continue its execution.

When using an interactive debugger it is common to set breakpoints in the target program – this is done by modifying the code in the target to add a breakpoint instruction and then responding to the resultant event. There is a lot of work behind the scenes in a good debugger to do such things as resolving the low level addresses and register values into high level constructs such as line numbers, source code and variables; these techniques are not covered in this article.

A debugger can detach from a debuggee by using **ptrace(PTRACE_DETACH, pid, 0, 0)**, letting the target process run freely. Unless this is done, when the debugging process terminates the target process is also automatically terminated.

Listing 1

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int idx;
  for (idx = 1; idx < argc; ++idx)
  {
    printf("** opening %s\n", argv[idx]);
    FILE *fp = fopen(argv[idx], "r");
    fclose(fp);
  }
}
```

Linux ProcessTracer

I am going to use a program that traces calls to **open** and **close** (and major events in a program's lifecycle) to provide the framework for exploring **ptrace**. Although the basic shape of the program would be the same on any flavour of Unix I have written and tested the program on Linux; I will try to point out in the text where the program is Linux-specific.

I will demonstrate the action of the process tracer by targetting an example program named **BadProgram**. The source code for **BadProgram** is very simple and is shown in Listing 1.

As is probably apparent, if any of the command line arguments refer to a file that cannot be opened for read, the call to **fclose** will segfault when it tries to access the null value of **fp**.

Here is an example of the tracing in use on the example program when it accesses a null pointer after failing to open the **noexist** file:

```
$ ./ProcessTracer ./BadProgram /dev/null noexist
open("/etc/ld.so.cache") = 3
close(3) = 0
open("/lib/x86_64-linux-gnu/libc.so.6") = 3
close(3)
** opening /dev/null
open("/dev/null") = 3
close(3)
** opening noexist
open("noexist") = -2(No such file or directory")
Segmentation violation
Terminated by signal 11
```

Here we can see *four* calls to **open**. The first two calls are from the program loader itself opening files as part of starting the program. These calls are

ROGER ORR

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

Listing 2

```
pid_t const cpid = fork();
if (cpid > 0)
{
  // In the parent
  return cpid;
}
else if (cpid == 0)
{
  // In the new child
  if (ptrace(PTRACE_TRACEME, 0, 0, 0) == -1)
  {
    // Handle error
  }
  execv( argv[0], argv );
  // Handle failing to start the new process
}
```

then followed by the two calls from the example code: a successful call to open `/dev/null` and a failed call to open **`noexist`**.

Now we've seen the process tracer in action let's go back to the beginning and build it up step by step.

Getting started

The first step in ProcessTracer is to create the child process with tracing enabled. As described above this is done by making a call to **`ptrace`** with the argument **`PTRACE_TRACEME`** in the **child** process. In Unix the standard way to create a child process is to clone the current process using **`fork`** and then call **`exec`** to replace the clone with the desired target. The usual place for the call to **`ptrace`** is therefore to put it in the code executed in the cloned child process after the call to **`fork`** which created the new process and before the **`exec`** which loads the target.

Listing 2 is an extract from ProcessTracer (without error handling, for clarity) showing this.

At this point the original process has been provided with the process ID of the new child and, since the child process has asked to be debugged, the call to **`execv`** in the child process is blocked internally waiting for the parent process to handle the associated ptrace event.

We must now take a look at the main debug loop.

The heart of the matter

In its simplest form the 'debug loop' looks like Listing 3.

This loop is purely reactive and isn't doing anything more than displaying the various debug events. Let's look at what it does and then extend it further in a moment.

The first call, to **`wait`**, halts the debugger until the next event is ready from the debuggee. On a successful return, the process id of the child raising the event is returned. The **`wait`** call returns **`-1`** to indicate there are no more debug events to process – the child process(es) have completed and we can leave the main loop.

We check the error number is ECHILD (no children) and if not raise an error. (The implementation of **`make_error`** is in the full source code for the article.)

The call returns a status value for the event, and we use a set of macros defined in `wait.h` to extract the event type and any related arguments from the status field. As can be seen from the structure of the loop the possible event types are:

- **Stopped** (**`WIFSTOPPED`**)

  The process has stopped at a signal event and the status contains the signal number. Normally you simply want to pass the signal on to the process that reported it so it can do its normal processing on the signal. One exception however is the **`SIGTRAP`** signal which is generated by a breakpoint and is also used by ptrace for some of the events that it generates. It is important that the **`SIGTRAP`** is **not** passed on to the target process as the default action is to terminate the process.

  The very first event you receive when debugging is a stop event from the child process. (In our case this occurs when the child process is inside the call to **`execv`**.) This event gives the debugger a chance to set up the debugging environment and to decide how to let the child process continue. The only exception to this is if the **`execv`** fails because, for instance, the target binary does not exist. In this case you may receive an 'Exited' event first, so for robustness you must ensure this possibility is catered for.

- **Exited** (**`WIFEXITED`**)

  The process has called **`exit`** (or otherwise terminated normally) with the specified exit status.

- **Signaled** (**`WIFSIGNALED`**)

  The process has terminated in response to the specified signal (for example, **`SIGABRT`** from calling **`abort`**).

- **Continued** (**`WIFCONTINUED`**)

  The process has resumed by means of **`SIGCONT`** (newer versions of Linux only).

The final stage of the main loop is calling **`ptrace`** with the **`PTRACE_CONT`** argument which resumes the child process until the next debug event. The code then goes back to the controlling wait call until one of the next events occurs. Other arguments are possible

Listing 3

```
void TrivialPtrace::run()
{
  int status(0); // Status of the
                 // stopped child process
  while ((pid = wait(&status)) != -1)
  {
    int send_signal(0);
    if (WIFSTOPPED(status))
    {
      int const signal(WSTOPSIG(status));
      os << "Signal: " << signal << std::endl;
      if (signal != SIGTRAP)
        send_signal = signal;
    }
    else if (WIFEXITED(status))
    {
      os << "Exit(" << WEXITSTATUS(status) << ")"
         << std::endl;
    }
    else if (WIFSIGNALED(status))
    {
      os << "Terminated: signal "
         << WTERMSIG(status)
         << std::endl;
    }
    else if (WIFCONTINUED(status))
    {
      os << "Continued" << std::endl;
    }
    else
    {
      os << "Unexpected status: " << status
         << std::endl;
    }

    ptrace(PTRACE_CONT, pid, 0, send_signal);
  }
  if (errno != ECHILD)
  {
    throw make_error("wait");
  }
}
```

when resuming the child process – for example **PTRACE_SINGLESTEP**, which resumes the child process for just a single instruction; or **PTRACE_SYSCALL**, which resumes the child process until the next time it enters or leaves a system call or generates a normal debug event; we will use this value later on in the article.

Here is the output from running this trivial **ptrace** example against our 'bad' program:

```
$ ./TrivialPtrace ./BadProgram /dev/null noexist
Signal: 5
** opening /dev/null
** opening noexist
Signal: 11
Terminated: signal 11
```

The first output '**Signal 5**' is caused by the initial **SIGTRAP** signal event returned by the **ptrace** API when the connection to the target process is first made. This first event would typically be used to initialise the debugger for the target process.

A quick check of `signal.h` shows that signal 11 (on my system) is **SIGSEGV** – segmentation violation. The full **processTracer** program adds code to map signal numbers to more 'friendly' strings.

Thread and process start and stop

The tracing program at this point suffers from what can be a fairly important restriction – it will debug only the initial target thread and process selected. We can demonstrate this by invoking a shell and then running our bad program from the shell.

```
$ ./TrivialPtrace /bin/sh
Signal: 5
$ ./BadProgram /dev/null noexist
** opening /dev/null
** opening noexist
Signal: 17
Segmentation fault (core dumped)
$ ^D
Exit(0)
```

Here we see no sign of the **SEGSEGV** signal (11) but only the **SIGCHLD** signal (17) which is received by the shell when its child process – **BadProgram** – exits. It is very common to want to follow processes (and threads) created by the target process. How can we do this with **ptrace**?

In the early days of **ptrace** the usual way to do this was to turn on system call tracing and process the calls that created new processes, such as **fork**. When **fork** returns in the parent process the return value is the **pid** of the new child process. The debugger could then issue a call to **ptrace(PTRACE_ATTACH, pid, 0, 0)**. Unfortunately this was subject to a race condition as the child process may have already executed a number of instructions before the debugger received the event via **ptrace** and was able to attach to the new **pid**. This mechanism is still available and may be used for portability reasons.

However, Linux has added some extensions to **ptrace** to improve support for debugging child process and also to provide support for debugging multiple threads. Note this is one of the places where Linux diverges from other versions of Unix as it has provided its own implementation for supporting threads, via the **clone** system call.

I found a lot of questions on the Internet about the best way to program this – it seems that the documentation is not entirely clear. Here are the steps I have found minimally sufficient.

Request tracing of child processes

When the child process is stopped at the initial stop event we set additional **ptrace** options **PTRACE_O_TRACEFORK**, **PTRACE_O_TRACEVFORK** and **PTRACE_O_TRACECLONE**. This makes the system automatically turn on tracing for processes and threads created by the three system calls **fork**, **vfork** and **clone**. Additionally, the newly created processes and threads in turn inherit these **ptrace** settings. Note that setting these options can only be done when the child process is stopped on a **ptrace** event.

Wait for all the child processes

We have to replace **wait** with a call to **waitpid** and provide a Linux-specific value **__WALL** for the third argument. If we fail to do this we do not receive signals for all the additional tasks.

Process the additional events

One we have set the options above the system will deliver additional debug events using the 'Stopped' status. The debugging process receives notifications from the parent process with signal type **SIGTRAP** with additional flags OR'd in to the status value to indicate which system call the event was raised by. Additionally the debugger also receives an initial **SIGSTOP** signal from each newly created thread or process. It seems necessary, despite what the **ptrace** documentation states, to avoid sending on the **SIGSTOP** signal to the process being debugged. Here too Linux has added a further option, **PTRACE_O_TRACEEXEC**, which can be enabled to OR an additional flag value into these initial events.

I modified the earlier example to create a **MultiPtrace** class which handles threads and processes. I refactored the handling of stopped events into a separate function: **OnStop** which is invoked in the debug loop like Listing 5. The **OnStop** function itself is shown in Listing 6.

Now when we run the previous example we receive the debugging events from *both* the direct child, the shell, and also from its child processes:

Listing 4

```
void TrivialPtrace::run()
{
  int status(0); // Status of the
                 // stopped child process
  while ((pid = wait(&status)) != -1)
  {
    int send_signal(0);
    if (WIFSTOPPED(status))
    {
      int const signal(WSTOPSIG(status));
      os << "Signal: " << signal << std::endl;
      if (signal != SIGTRAP)
        send_signal = signal;
    }
    else if (WIFEXITED(status))
    {
      os << "Exit(" << WEXITSTATUS(status) << ")"
         << std::endl;
    }
    else if (WIFSIGNALED(status))
    {
      os << "Terminated: signal "
         << WTERMSIG(status)
         << std::endl;
    }
    else if (WIFCONTINUED(status))
    {
      os << "Continued" << std::endl;
    }
    else
    {
      os << "Unexpected status: " << status
         << std::endl;
    }
    ptrace(PTRACE_CONT, pid, 0, send_signal);
  }
  if (errno != ECHILD)
  {
    throw make_error("wait");
  }
}
```

Listing 5

```
if (WIFSTOPPED(status))
{
  send_signal = OnStop(WSTOPSIG(status),
                       status >> 16);
}
```

Listing 6

```
int MultiPtrace::OnStop(int signal, int event)
{
  if (!initialised)
  {
    initialised = true;
    long const options =
      PTRACE_O_TRACEFORK |
      PTRACE_O_TRACEVFORK |
      PTRACE_O_TRACECLONE;
    if (ptrace(PTRACE_SETOPTIONS, pid,
              0, options) == -1)
    {
      throw make_error("PTRACE_SETOPTIONS");
    }
    return 0;
  }

  if (event)
  {
    os << "Event: " << event << std::endl;
  }
  else
  {
    os << "Signal: " << signal << std::endl;
  }
  return (signal == SIGTRAP || signal == SIGSTOP)
     ? 0 : signal;
}
```

```
$ ./MultiPtrace /bin/sh
$ ./BadProgram /dev/null noexist
Event: 1
Signal: 19
Signal: 5
** opening /dev/null
** opening noexist
Signal: 11
Terminated: signal 11
Signal: 17
Segmentation fault (core dumped)
$ ^D
Exit(0)
```

System calls

The next step is to look at tracing system calls; which is enabled by simply replacing **`PTRACE_CONT`** with **`PTRACE_SYSCALL`** in the main debugging loop. Having done this we get two more events on every system call; one event on entry to the system call and one event just before exiting the system call. The event returned is a 'stopped' event with the stop signal value **`SIGTRAP`**. This is the same value used for a software breakpoint event and while this makes sense of what is occurring it can mean additional work by the debugger to differentiate between the two cases.

This overloading of the **`SIGTRAP`** processing is unnecessary – a distinct value for a system call event would have been a cleaner design. (Those who read the earlier article on the Windows debugging interface may recall a similar overload with the so-called 'initial breakpoint' event sent to notify the debugger that the initialisation has completed.) Some later versions of **`ptrace`** now support an additional option – **`PTRACE_O_TRACESYSGOOD`** – which, when set, changes the signal value by ORing it with 0x80 and hence removes the ambiguity. Prior to this option (and for systems not supporting it) you normally disambiguate the two cases by calling **`ptrace`** with the **`PTRACE_GETSIGINFO`** argument to return information about the underlying signal – the returned value of **`si_code`** will be **`SIGTRAP`** for the system call entry/exit case.

Now we can expand a little further on the initial **`SIGTRAP`** event we saw in the **`TrivialPtrace`** example: this is actually a system call exit event for the **`execve`** call in the child process.

The events generated on entry and exit to a system call are distinguished by the value of the register used for the return code, which contains a special value of **`-ENOSYS`** on entry and the actual return value or error code on exit. Note that this technique works as no system call returns **`-ENOSYS`** on failure.

This information is not provided by the **`ptrace`** event itself, the debugger has to make an additional call using the **`PTRACE_GETREGS`** function code to read the register values for the target process. We then have to get down to the specifics of the architecture – which register will contain the return code we are we looking at? The documentation for each hardware platform includes a section on calling conventions and the system calling convention gives us the information we are looking for.

System call conventions on x86 and x64

The Linux convention on Intel x86 hardware is to use the **`eax`** register for the system call number on entry and the return code on exit. The arguments to the system call (up to six arguments depending on the call) are passed in **`ebx`**, **`ecx`**, **`edx`**, **`esi`**, **`edi`** and **`ebp`**.

As described above, when a syscall entry event occurs the **`eax`** value is overwritten by the special value **`-ENOSYS`**. The **`ptrace`** interface therefore saves the original **`eax`** value in **`orig_eax`** so the debugger still has access to the system call number. It also does this on the system call exit case so the debugger knows *which* system call is exiting.

The basic logic looks something like Listing 7.

The actual handling of each function will obviously involve decoding the actual value of each of the valid arguments to the function call.

For the x64 API the **`rax`** register replaces the **`eax`** register and the system call arguments are passed in **`rdi`**, **`rsi`**, **`rdx`**, **`r10`**, **`r8`** and **`r9`** with the original system call number returned in **`orig_rax`**. The resultant code is very similar to that shown in Listing 7 for the x86 case.

Turning a register value into a file name

The process tracer program in this article is interested in just two system calls: **`open`** and **`close`**. The corresponding system call numbers are **`__NR_open`** and **`__NR_close`**.

The **`close`** call takes a single argument, the file handle to close, and so displaying this is easy. The **`open`** call is more complicated as we are interested in the first argument ("**`const char *`** ***`path`***") and this argument is the **address** of the character string.

The return code from the system call is negative on error and the error code is the negation of this number. It can be displayed as a text string by using

Listing 7

```
struct user_regs_struct regs;
if (ptrace(PTRACE_GETREGS, pid, 0, &regs) == -1)
{
  throw make_error("ptrace(PTRACE_GETREGS)");
}
int const rc = regs.eax;
int const func = regs.orig_eax;
if (rc == -ENOSYS)
{
  OnCallEntry(func, regs.ebx, regs.ecx, ...);
}
else
{
  OnCallExit(func, rc);
}
```

Listing 8

```
std::string ProcessTracer::readString(long addr)
{
  std::string result;
  int offset = addr % sizeof(long);
  char *peekAddr = (char *)addr - offset;
  // Loop round to find the end of the string
  bool stringFound = false;
  do
  {
    long const peekWord =
      ptrace( PTRACE_PEEKDATA, pid,
              peekAddr, NULL );
    if ( -1 == peekWord )
    {
      throw make_error
         ("ptrace(PTRACE_PEEKDATA)");
    }
    char const * const tmpString =
      (char const *)&peekWord;
    for (unsigned int i = offset;
         i != sizeof(long); i++)
    {
      if (tmpString[i] == '\0')
      {
        stringFound = true;
        break;
      }
      result.push_back(tmpString[i]);
    }
    peekAddr += sizeof(long);
    offset = 0;
  } while ( !stringFound );
  return result;
}
```

the standard **strerror()** function. (This is a slight simplification: only ‘small’ negative numbers indicate an error.)

We need to invoke **ptrace** again to read the string, this time using the **PTRACE_PEEKDATA** function code. This function can be slightly awkward to use as it only provides access to memory one word at a time.

In order to read a **NUL**-terminated string we must bear in mind that the start of the string may not be aligned on a word boundary, so we may need to read (and ignore) a few bytes before the string starts. The algorithm then reads words one at a time and processes characters from each word until the terminating null is located.

Listing 8 is an implementation of this algorithm.

A further difficulty with the access mechanism occurs when reading arbitrary data (this issue doesn’t occur when reading printable character strings) as the return code of -1 may be caused by an error reading from the target **or** by reading the value of -1 from the target process. The only way to differentiate between these two cases is to set **errno** to zero before making the call and examine its value afterwards: it will still be zero for successful completion and the error number on failure.

This is another place where the original design of **ptrace** is showing signs of age. The mechanism is unwieldy to use and causes performance problems when reading larger data structures as each word of memory read involves a separate system call.

Listing 9

```
std::string ProcessTracer::readString(long addr)
{
  std::string result;  std::ostringstream os;
  os << "/proc/" << pid << "/mem";
  std::ifstream mem(os.str().c_str(),
                    std::ios::binary);
  mem.exceptions(std::ios::failbit);
  mem.seekg((std::streampos)addr);
  std::getline(mem, result, '\0');
  return result;
}
```

Many versions of Unix now provide other ways to access memory from another process. In Linux we can make use of the `/proc` pseudo-filesystem to access the memory of the target process using the filesystem API to read `/proc/`*`pid`*`/mem` for the target process id.

Listing 9 is the same method reimplemented in terms of this filesystem.

In production code it is likely that the open file stream would be cached for performance.

Extending process tracer

We have now completed the implementation of the simple process tracer demonstrated at the beginning of the article. The basic framework is in place to handle the debug events generated by the child process(es) and thread(s) and we have looked at how to access data in the target process.

The standard system tool **strace** performs all this, and more, and for most of the process tracing needs you might have this is likely to be the right solution! It is already configured to understand and display the various system call argument types and has a range of options available.

However, there are times when a more specific tool is required and techniques like the ones described in this article can be used to implement such tools.

Conclusion

I have covered only the basics of a call tracer in this article and there is obviously a lot more that must be added to write a proper interactive debugger. However I hope that the overview of the debug API that I have presented here has given you some understanding of the bare bones of the interaction between the debugger and the target; and some sympathy for the complexity of the task involved in providing a tool such as **gdb** or **strace**. ■

Acknowledgements

Many thanks to Irfan Butt for reviewing this article and correcting a number of mistakes.

Source code

The full source code for this article can be found at:
http://www.howzatt.demon.co.uk/articles/LinuxProcessTracer.zip

Functional Decomposition

Richard Polton looks at more functional techniques for reducing duplication.

One of the most-frequently occuring meta-patterns (that's a word I use to describe a pattern of patterns ;-) that I have come across in my most recent position is the imperative loop. Generally speaking, the loop will have a simple body, possibly containing some decision branches, and almost always will result in a value of some sort. For example,

```
var l = new List<U>();
for(int i=0; i<myContainer.Count; ++i)
{
  l.Add(Convert.ToTypeU(myContainer[i]));
}
return l;
```

(and yes, I know this could be done in a **foreach** statement as well, but believe it or not, this was how it was).

This commonly occuring pattern is a map, in that it takes an input container, **myContainer** in this case, and transforms each element into another type before inserting it into a new container, **l**. Using .NET2.0, we have to create our own **map** function, but of course with the introduction of all that wonderful LINQ-iness, Microsoft have kindly removed most of the hand-cranking for us. But let's stick to .NET2.0 for now – it's easier to see how things work.

So, with .NET2.0, we declare a **delegate** and a function:

```
public delegate B map_fn<A,B>(A a);
public static List<B> map<A,B>(map_fn<A,B> f,
   List<A> input)
```

In case you are wondering, this parameter ordering style is often associated with functional programming, and we will use these functions later on to demonstrate functional composition. So, we have:

```
public static List<B> map<A,B>(map_fn<A,B> f,
   IEnumerable<A> input)
{
  List<B> r = new List<B>();
  foreach(A a in input) r.Add(f(a));
  return r;
}
```

Lovely! Now we can rewrite our original code as

```
return map(
  delegate(T t)
  {
    return Convert.ToTypeU(t);
  },
  myContainer);
```

So, armed thusly we can strike out across the codebase replacing the numerous transformations with our **map** function. However, while this is easy to say it quickly becomes less easy to do. One reason for that is the dreaded predicate. Oh woe! someone sneaked in a little **if** statement into our clean loop (see Listing 1). What to do?

Lackaday! Now how should we progress? We cannot use the **map** function because that is a 1-to-1 transformation, meaning that one element in the result set is produced for every element in the input set. What we need is a filter. So, again, define a **delegate** and a function., as in Listing 2.

And so now we can rewrite the loop in terms of **map** and **filter**.

```
T limit;
return
  map(delegate(T t) {return Convert.ToTypeU(t);},
  filter(delegate(T t) {return t<=limit;},
  myContainer));
```

Hmm, well, it does what we want but it's not very efficient is it? Having to loop through the input container twice is not good at all (although it is more in keeping with C#'s long-windedness). What we need now is some mechanism which allows us to present each individual element in the input container to both the filter and the map at once. The simplest, and most C#, way of doing this is to return **IEnumerable** from the **map** and

Listing 1

```
T limit; // imagine it has a value, please ;-)
var l = new List<U>();
for(int i=0; i<myContainer.Count; ++i)
{
  if(myContainer[i] <= limit)
     l.Add(Convert.ToTypeU(myContainer[i]));
}
return l;
```

Listing 2

```
public delegate bool filter_fn<A>(A a);
public static List<A> filter<A>(filter_fn<A> f,
   IEnumerable<A> input)
{
  List<A> r = new List<A>();
  foreach(A a in input)
     if(f(a)) r.Add(a);
  return r;
}
```

Listing 3

```
public static
  IEnumerable<B> map<A,B>(map_fn<A,B> f,
  IEnumerable<A> input)
{
  foreach(A a in input) yield return f(a);
}

public static
  Enumerable<A> filter<A>(filter_fn<A> f,
  Enumerable<A> input)
{
  foreach(A a in input)
    if(f(a)) yield return a;
}
```

RICHARD POLTON

Richard has enjoyed functional programming ever since discovering SICP and feels heartened that programming languages are evolving back to LISP. He likes 'making it better' and enjoys riding his bike when he can't. He can be contacted at richard.polton@shaftesbury.me

Listing 4

```
public delegate OptionType<B> choose_fn<A,
   B>(A a) where B:class;

public static IEnumerable<B>
   choose<A,B>(choose_fn<A> f,
   IEnumerable<A> input) where B:class
{
  foreach(A a in input)
  {
    OptionType<B> intermediate = f(a);
    if(!intermediate.None) yield
       return intermediate.Some;
  }
}
```

filter functions instead of **List**. This means that the output container will be produced as it is consumed. So, rewrite **map** and **filter** as shown in Listing 3.

Now, with these new definitions for **map** and **filter**, the input container is only traversed once. Bonus! Albeit with the downside of having to explicitly force the **IEnumerable** to a list with a constructor call or similar, eg

```
return new List<U>(
  map(delegate(T t) {return Convert.ToTypeU(t);},
   filter(delegate(T t) { return t<=limit; },
   myContainer)));
```

Of course, we are not required to build a list. We could change the calling code to consume an **IEnumerable** instead, remembering that such a collection can only be traversed once, but for the purposes of this article we are making the minimum drop-in changes.

Now we have a one-liner which describes our pattern, a filtered transformation, and we have implemented it throughout the codebase. Everything is hunky-dory and looking great, but we have this nagging feeling that we could improve this, and indeed we could. There is a composite pattern called 'choose' that we can implement which rolls the **filter** and the **map** into a single function (Listing 4).

In this way, it is clear that the **input** container is only traversed once. Now we can rewrite our original looping code as

```
return choose<T,U>(
  delegate(T t)
  {
    return t<=limit ? new OptionType<U>
      (Convert.ToTypeU(t)) : OptionType<U>.None;
  },
  myContainer);
```

We now feel quite pleased with ourselves, having trimmed much duplication from our codebase and replaced it with standard patterns. One might very well ask what comes next and how can we build upon this? Another common pattern which is related to the above is the **find** pattern. The behaviour of this pattern should come as no great surprise; it is used to identify the first element in a container which satisfies some predicate. Imagine we find code like this:

```
T r;
foreach(T a in myContainer)
{
  if(a>limit)
  {
    r=a;
    break;
  }
}
```

Clearly, if no element of **myContainer** is greater than the limit then **r** will be null otherwise **r** will contain the first element which is greater than the limit. While less common in our codebase, this did occur reasonably frequently. Before creating a new function **find** which replaces this pattern, we need to decide how to implement the failure scenario, that is the case where no element exists in the container that satisfies the predicate. We choose to throw a custom exception of type **KeyNotFoundException** (in fact, following from the F# practice) although we could have returned an **OptionType<T>.Null** [1] as long as our input container could be guaranteed not to contain that element.

We have already defined a **filter_fn** which we shall reuse here.

```
public class KeyNotFoundException : Exception;
public static A find<A>(filter_fn<A> f,
                        IEnumerable<A> input)
{
  foreach(A a in input)
    if(f(a)) return a;
  throw new KeyNotFoundExpception();
}
```

Now the example code can be rewritten using the **find** function as

```
T r;
try
{
  r = find(delegate(T t) { return t>limit; },
     myContainer);
}
catch(KeyNotFoundExpception)
{
}
```

Not ideal, because it's not declarative, but it's better. We can of course improve on this but that can wait for another time. Now we have 'find', we can trawl the codebase looking for instances of this pattern and replace them. In the same manner as we did with 'choose' earlier, we can replace the combined 'map' and 'find' with a single composite pattern 'pick'. That is, if we have

```
U r;
foreach(T a in myContainer)
{
  if(a>limit)
  {
    r=new U(a);
    break;
  }
}
```

we define

```
public static B pick<A, B>(choose_fn<A, B> f,
   IEnumerable<A> input) where B : class
{
  foreach (A a in input)
  {
    OptionType<B> intermediate = f(a);
    if (!intermediate.None)
      return intermediate.Some;
  }
  throw new KeyNotFoundException();
}
```

and so we can write

```
U r;
try
{
  r = pick(delegate(T t)
  { return t>limit ? new OptionType<U>(new U(t))
    : OptionType<U>.Null;}, myContainer);
}
catch(KeyNotFoundException)
{
}
```

Listing 5

```
public delegate A try_fn<A>();

public static A try_<A,Ex>(try_fn<A> try_f,
   try_fn<A> catch_f) where Ex:Exception
{
  try
  {
    return try_f();
  }
  catch(Ex)
  {
    return catch_f();
  }
}
```

Listing 6

```
U r = try_<U,KeyNotFoundException>(
  delegate()
    {
      return pick(delegate(T t)
        {
          return t>limit ? new OptionType<U>
          (new U(t)) : OptionType<U>.Null;
        },
        myContainer);
    },
  delegate()
    {
      return new U();
    });
```

Gah! This **try** ... **catch** is annoying, isn't it? Let's do something about it now. We can't wait forever for high-quality code! Analysing the structure we can see that **try** ... **catch** should take two functions as parameters, each of them returning a type **U**, in our examples. So, we write Listing 5 and now we can replace the **pick** example with Listing 6.

At this point, we have converted large swathes of repetitive code into a functional style and in so doing will almost certainly have observed marked similarities between chunks of code, thus further enhancing quality by removing repetition (the Evil Copy-and-Paste demon certainly is tempting). There are two more patterns which are of particular use although there are several more which have not been required here. These last two patterns are referred to by the names 'fold' and 'partition'. Let us start with 'fold'.

The 'fold' is an accumulation operation. That is to say, the **fold** function is given an initial state and an accumulator function, and returns the value after applying the accumulator over each element in the container in turn. One example that occurs frequently in this codebase was the calculation of the sum of certain record fields. For example, we find

```
double tot=0.0;
foreach(Record r in myRecords)
{
  tot+=r.Value;
}
return tot;
```

and

```
DateTime dt = DateTime.MinDate;
foreach(Record r in myRecords)
{
  if(r.Date>dt) dt=r.Date;
}
return dt;
```

The pattern to be recognised here is the iteration over the sequence while making use of a previously-calculated value. Let us create a 'fold' function (Listing 7).

Listing 7

```
public delegate B fold_fn<A,B>(B state, A a);
public static B fold<A,B>(fold_fn<A,B> f,
   B initialState, IEnumerable<A> input)
{
  B state = initialState;
  foreach(A a in input)
  {
    state = f(state, a);
  }
  return state;
}
```

Now, using this function we can rewrite the two examples above as

```
return fold(delegate(double tot, Record r) {
  return tot+r.Value; }, 0.0, myRecords);
```

and

```
return fold(delegate(DateTime dt, Record r) {
  return r.Date > dt ? r.Date : dt; },
  DateTime.MinDate, myRecords);
```

In a similar manner, the 'fold' function can be used to find the minimum of a sequence and the count, as well as slightly more complex cases like compiling a string from a list of integers.

The final pattern to be explored here is the partition. This has not been observed often in this codebase but when it occurs it is invariably mashed in with some other pattern and so the extraction of the two can be tricky. Obviously, YMMV. Anyway, the partition function does exactly what it appears to do, that is, given a predicate and an input sequence, it returns two sequences in which the elements in the first satisfy the predicate and the elements in the second do not. Given the definition of **tuple2** in Listing 8, we can implement **partition** as shown in Listing 9 and now we can split our sequence of records, for example, into two. One sequence should hold those elements which are less than a date and the remainder should be in the second sequence.

```
DateTime limit;
tuple2<List<Record>,List<Record>>
   val = partition(delegate(Record r)
   { return r.Date < limit; }, myRecords);
```

Listing 8

```
public struct tuple2<A, B>
{
  public tuple2(A a, B b)
  {
    field1 = a;
    field2 = b;
  }
  public A field1;
  public B field2;
}
```

Listing 9

```
public static tuple2<List<A>,
  List<A>>     partition<A>
  (filter_fn<A> f, IEnumerable<A> input)
  {
    var left = new List<A>();
    var right = new List<A>();
    foreach (A a in input)
      if (f(a))
        left.Add(a);
      else
        right.Add(a);
    return new tuple2<List<A>,
       List<A>>(left, right);
}
```

Functional composition in C#

Now that we have some functions to perform common tasks, it's time to talk about another aspect of functional programming as applied to C#; functional composition. Coming from a C++ background with a pre-conceived idea of how to do this we were quickly disabused of the notion by the syntactic constraints. But first, before launching into a technical explanation of how functional composition was implemented, let us briefly describe what it is and why we want to do it, so to speak.

So, the mighty Wikipedia [2] gives this for mathematical function composition:

> In mathematics, function composition is the application of one function to the results of another. For instance, the functions f: X → Y and g: Y → Z can be composed by computing the output of g when it has an argument of f(x) instead of x. Intuitively, if z is a function g of y and y is a function f of x, then z is a function of x.

and, from a different link [3], this for programmatic functional composition:

> In computer science, function composition (not to be confused with object composition) is an act or mechanism to combine simple functions to build more complicated ones. Like the usual composition of functions in mathematics, the result of each function is passed as the argument of the next, and the result of the last one is the result of the whole.
>
> Programmers frequently apply functions to results of other functions, and all programming languages allow it. In some cases, the composition of functions is interesting as a function in its own right, to be used later. Such a function can always be defined but languages with first-class functions make it easier.
>
> The ability to easily compose functions encourages factoring (breaking apart) functions for maintainability and code reuse. More generally, big systems might be built by composing whole programs.

So, with these definitions in mind, what we would like to be able to write is something akin to:

```
t > (f >> g)
```

where **>** should be interpreted as an operator which takes a function on the right-hand side and its argument(s) on the left-hand side, in other words it reverses the arguments; and **>>** should be interpreted as the functional composition operator that returns the output of `g` given that the output of `f` is passed as the input to `g`. So, as a first step, let us define two functions `fn1` and `fn2` as

```
public static R1 fn1(T t);
public static R2 fn2(R1 r);
```

C# gives us the `delegate` type which is analogous to a strongly-typed C function pointer. This means that we can create a pair of `delegate`s

```
public delegate R1 dfn1(T t);
public delegate R2 dfn2(R1 r);

public static dfn1 fn1_d = fn1;
public static dfn2 fn2_d = fn2;
```

Now we have the components but how can we combine them? Ideally, we would like to be able to extend the delegate type by adding a pair of operators, namely **>** and **>>**. Unfortunately, C# 2.0 does not allow this because, for some unknown (to the author) reason, the delegate type is neither a struct nor a class and so cannot be extended. Hmmm ... So, what should we do? The default response is to add a layer of indirection and, in this case, we will create a class which mimics the behaviour of a `delegate` and encapsulates the actual `delegate` we wish to access.

Create a class called `Func` and store within it a `delegate`, as shown in Listing 10.

So far so good. Now we can create a `Func` object and initialise it with the `delegate` of our choosing, like this:

```
Func<T,R1> fn1_f = new Func<T,R1>(fn1_d);
Func<T,R1> fn2_f = new Func<R1,R2>(fn2_d);
```

Listing 10

```
public class Func<A,RetType>
{
  private System.Func<A,RetType> _f;
  public Func(System.Func<A,RetType> f)
  {
    _f = f;
  }
}
```

We are now in a position to extend the `Func` class further to access the underlying `delegate`. For now, let us define a function `act` which executes the underlying function.

```
public RetType act(A a)
{
  return _f(a);
}
```

Okay. So now we can write

```
T t = new T();
R2 r2 = fn2_f.act(fn1_f.act(t));
```

which is close; it's composition of a sort. What we need to do next is to reverse the order of the components in the declaration. As illustrated above, we already have an idea of how we wish to represent the composed functions so it is necessary to consult the C# language specification. Taking a look through the MDSN library [4], we see that we cannot create new operators! Boo hiss! Additionally, if we implement the **>** operator we are required to implement the **<** operator as well! This presents us with two new problems but, with a little thought, it will be possible to work around them.

In the case of the proposed composition operator, we suggest a new function be added to the `Func` interface. Let us call this new function 'Then'. We suggest that this function would transform the goal into

```
t > f.Then(g)
```

which, while not ideal, still captures the meaning in a succinct and clear manner. The only real issue we have with this modified solution is the manner in which multiple functions are composed together. That is, if we want to write

```
t > (f >> g >> h)
```

then we have to write

```
t > f.Then(g).Then(h)
```

This by itself is fine, and means the same thing as

```
t > f.Then(g.Then(h))
```

which we prefer not to use. So, having modified our desired goal to be the creation of

```
t > f.Then(g)
```

let us define 'Then'. In class `Func<A,RetType>` define

```
public Func<A,C> Then(Func<RetType,C> g)
{
  return new Func<A,C>(delegate(A a)
  { return g.act(act(a)); });
}
```

Now we can define a composed function `fc`, say, as

```
Func<A,C> fc = f.Then(g);
```

Progress! The final step is the **>** operator. As previously discussed this is complicated by the requirement of the C# language specification that the opposite operator, **<**, is also declared. While the **<** operator could be used to preserve the usual order of the function arguments (which is how it is used in F#), unfortunately the C# language spec also requires that both operators are implemented using the same signature! Doh!

Listing 11

```
public static RetType operator >(A a, Func<A,
                                RetType> f)
{
  return f.act(a);
}

/// <summary>Not implemented because it makes no
sense. C# compiler requires the matching operator
but logic does not</summary>
public static RetType operator <(A a, Func<A,
                                RetType> f)
{
  throw new NotImplementedException();
}
```

This is an issue which we decide to tackle by virtue of throwing a `NotImplementedException`. There seems to be little else we can do to avoid this and so commenting and a run-time failure is the best we can achieve at present. Therefore, the `>` and `<` operators are implemented in class `Func<A, RetType>` as shown in Listing 11.

And now we can finally write code in the desired manner, that is

```
t > f.Then(g)
```

Note that we could also have written this equivalently as

```
t > f > g
```

And so, having created the structure an example seems like a good idea. The example we will present takes a fairly common task, that of a command-line parser which distinguishes between mandatory and optional arguments, and transforms it into a reusable pattern.

In Listing 12 is the somewhat long-winded code as it existed prior to transformation. Imagine that `ParseCommandLine` is a function which accepts a string array and returns a dictionary of strings, that `Functional.map` is a 1-to-1 transformation operation (as described above) and that `StringUtils.Split` is a safe wrapper around `string.Split`.

As can be seen from the code, there is a lot of repetition. We aim to reduce, or remove, this using functional composition. The first task is to recognise that there are two types of command line parameters: mandatory and optional. So we create two dictionaries, one for each type, and name them appropriately. As we are using .NET2.0 here, we have created some utility functions for dictionary and list creation. We have `toDict` which takes an array and returns a `Dictionary`, `toList` which takes an array and returns a `List`, `init` which creates a list using a generator function and `Augment`, which concatenates an element and a container of the same type. Listing 13 shows the transformed code using functional composition.

Clearly, in Listing 13, we have included some generally applicable functions into the class definition but, in the real world, you would expect to see `SplitAndPopulateArguments`, the two instances of `SplitArgsInto` and `Validate` in some public static utility class.

In Listing 13, `ParseCommandLine` is the function of interest and the line of particular interest is

```
bool result = args > SplitArgsInto(mandatoryArgs,
  optionalArgs).Then(Validate(mandatoryArgs));
```

This is where all the magic happens; the rest of it is setting up and tearing down, if you like. This is the pattern that we reuse throughout our codebase.

Currying

We have demonstrated composition of functions but one thing that went unsaid was a concept referred to as currying. Now it is important to point out that this has nothing to do with the UK's national dish but instead is concerned with partial function calls. That is to say, if we have a function `f` which expects two parameters `x` and `y` then `f(x)` is a curried call to `f`. See Wikipedia's entry on 'currying' [5] by way of reference.

If we momentarily forget C# syntactic issues, we could write

```
R1 f(T1 t1, T2 t2) { return new R1(t1,t2); }
  T1 t1 = new T1();
  <some type> f1 = f(t1);
  T2 t2 = new T2();
  R1 r1 = f1(t2);
```

and the final line would be the point at which 'f' is actually called. At the start of this article when we discussed meta-functions, the ordering of function arguments was mentioned. Taking 'map' for example, we expect to present the transformation function as the first argument and the container over which the transformation will be applied as the second argument. This is for currying purposes. For example, in C#, suppose we can curry functions and suppose we have

```
IEnumerable<B> map<A,B>(Func<A,B> f,
IEnumerable<A> input)
```

Then, if we have defined the `>` operator as above, we can write

```
IEnumerable<double> d =
  new double[]{1.2,2.1,3.7,4.8,5.0}
  > map( Math.Floor);
```

Clearly, in C# as it stands this is not possible but, as illustrated implicitly in the command-line parsing example above, with a little work it can be achieved. In order to make this example work in C#, define a second function also called `map` as

```
Func<IEnumerable<A>,IEnumerable<B>>
  map<A,B>(Func<A,B> f)
{
  return new Func<IEnumerable<A>,IEnumerable<B>>
    (delegate (IEnumerable<A> input)
    { return map(f, input); });
}
```

If we create the analogous 'filter' function we can re-implement the 'choose' function in terms of 'map' and 'filter'. 'choose' is equivalent to `filter(predicate).Then(map(transformation))` as long as we ensure that the input sequence is traversed only once.

Composition in action

Having presented a simple example and demonstrated the neatness and reusability of the composition pattern, you are doubtless wondering how this might be applicable in other areas. One area that is of interest to the author, which shall be very quickly illustrated here, is combinations.

In the example in Listing 14, there is an ordered set of arrays of functions, each array containing functions whose input type is the output type of the functions in the prior array. For example, suppose we have Listing 14, then output is a list of 432 doubles which have been obtained by composing all of the functions in all of the allowable combinations. Of course this code can be reduced further using .NET3.5 syntax but we elect to leave it as it is for ease of comprehension. Equally obviously the arrays of functions do not have to be fixed, they could be dynamically generated. The possibilities are endless :-) ■

References

[1] See article in *C Vu* 24.4
[2] Wikipedia: http://en.wikipedia.org/wiki/Function_composition
[3] Wikipedia: http://en.wikipedia.org/wiki/Function_composition_%28computer_science%29
[4] MSDN: http://msdn.microsoft.com/en-us/library/ms228593
[5] Wikipedia: http://en.wikipedia.org/wiki/Currying

Listing 12

```
class ArgumentParser
{
  private const char _separator = ',';
  public static Arguments
    ParseCommandLine(string[] args)
  {
    string statusString = string.Empty;
    string folioString = string.Empty;
    string allotmentsString = string.Empty;
    string entitiesString = string.Empty;
    string counterpartiesString = string.Empty;
    string boStatusString = string.Empty;
    string businessEventsString = string.Empty;
    string acceptEventString = "291";
    string dbAccessString =
      "user/password@testServer";

    try
    {
      foreach (KeyValuePair<string,
         string> p in ParseCommandLine(args))
      {
        // Skip leading "-"
        string key = p.Key.StartsWith("-")
          ? p.Key.Substring(1) : p.Key;
        string value = p.Value;

        switch (key.ToLower())
        {
          case "status":
            statusString = value;
            break;
          case "folios":
            folioString = value;
            break;
          case "allotments":
            allotmentsString = value;
            break;
          case "businessevents":
            businessEventsString = value;
            break;
          case "entities":
            entitiesString = value;
            break;
          case "counterparties":
            counterpartiesString = value;
            break;
          case "bocommentstring":
            boStatusString = value;
            break;
          case "kernelevent":
            acceptEventString = value;
            break;
          case "dbdetails":
            dbAccessString = value;
            break;
          default:
            throw new ArgumentException
            ("Wrong commandline parameter :",
             key);
        }
      }
      List<int> statusList =
         SplitAndPopulateArguments(statusString);
      List<int> portfolioList =
         SplitAndPopulateArguments(folioString);
      List<int> allotmentList =
         SplitAndPopulateArguments
         (allotmentsString);
```

Listing 12 (cont'd)

```
      List<int> businessEventsList =
         SplitAndPopulateArguments
         (businessEventsString);
      List<int> entityList =
         SplitAndPopulateArguments
         (entitiesString);
      List<int> counterpartyList =
         SplitAndPopulateArguments
         (counterpartiesString);

      Arguments arguments =
         new Arguments(statusList,
                       portfolioList,
                       allotmentList,
                       businessEventsList,
                       entityList,
                       counterpartyList,
                       boStatusString,
                       acceptEventString,
                       dbAccessString);
      SanityCheckArguments(arguments);
      return arguments;
    }

    catch(Exception ex)
    {
      throw new ArgumentException(ex.Message);
    }
  }

  private static List<int>
    SplitAndPopulateArguments
    (string toSplit)
  {
    return string.IsNullOrEmpty(toSplit)
        ? new List<int>()
        : Functional.map<string,int>
          (delegate(string param)
          { return int.Parse(param); },
        StringUtils.Split(toSplit,_separator));
  }

  // This method checks some mandatory fields are
  // populated otherwise it throws an
  // ArgumentException message
  private static void
    SanityCheckArguments(Arguments arguments)
  {
    if (arguments.GetStatuses().Count <= 0)
    {
      throw new ArgumentException
        ("The user must provide at least one
         valid status");
    }
    if (arguments.GetPortfolios().Count <= 0)
    {
      throw new ArgumentException
        ("The user must provide at least one
         valid portfolio id");
    }
    if
      (arguments.GetAcceptEventStr().Length == 0)
    {
      throw  new ArgumentException
         ("BO event is cannot be null");
    }
  }
}
```

Listing 13

```
class ArgumentParser
{
  public static Arguments ParseCommandLine
    (string[] args)
  {
    const string defaultEvent = "291";

    try
    {
       Dictionary<string, string> mandatoryArgs =
         Functional.array.toDict<string>
         (new string[] {"status", "folios",
         "dbdetails"});

       Dictionary<string, string> optionalArgs =
         Functional.array.toDict<string,
         string>(new string[] {"kernelevent",
         "allotments", "businessevents",
         "entities", "counterparties",
         "bocommentstring"},

         Functional.seq.toList
            (Enumerators.Augment(defaultEvent,
             Functional.init(7, delegate(int i) {
             return string.Empty; }))));

         bool result = args > SplitArgsInto
            (mandatoryArgs, optionalArgs).
            Then(Validate(mandatoryArgs));

         const char sep = ',';

         return new Arguments
            SplitAndPopulateArguments(sep,
               mandatoryArgs["status"]),
            SplitAndPopulateArguments(sep,
               mandatoryArgs["folios"]),
            SplitAndPopulateArguments(sep,
               optionalArgs["allotments"]),
            SplitAndPopulateArguments(sep,
               optionalArgs["businessevents"]),
            SplitAndPopulateArguments(sep,
               optionalArgs["entities"]),
            SplitAndPopulateArguments(sep,
               optionalArgs["counterparties"]),
            optionalArgs["bocommentstring"],
               optionalArgs["kernelevent"],
               mandatoryArgs["dbdetails"]);
    }
    catch(Exception ex)
    {
      throw new ArgumentException(ex.Message);
    }
  }

  public static List<int>
    SplitAndPopulateArguments
    (char separator, string toSplit)
  {
    return string.IsNullOrEmpty(toSplit)
       ? new List<int>()
       : Functional.map<string,int>
         (delegate(string param)
           { return int.Parse(param); },
         StringUtils.Split(toSplit, separator));
  }
```

Listing 13 (cont'd)

```
  public static string SplitArgsInto
    (Dictionary<string, string> mandatory,
     Dictionary<string, string> optional,
     IEnumerable<string> args)
  {
    foreach (KeyValuePair<string,
    string> kv in ParseCommandLine
    (args.ToArray()))
    {
      string k = (kv.Key.StartsWith("-") ?
         kv.Key.Substring(1) : kv.Key).ToLower(),
         v = kv.Value;

      if (mandatory.ContainsKey(k))
         mandatory[k] = v;
      else if (optional.ContainsKey(k))
         optional[k] = v;
      else return "Unrecognised commandline
         parameter :" + kv.Key);
    }
    return string.Empty;
  }

  public static
    Functional.Func<IEnumerable<string>,
    string> SplitArgsInto
    (Dictionary<string, string> mandatory,
     Dictionary<string, string> optional)
    {
       return
         new Functional.Func<IEnumerable<string>,
         string>(delegate
         (IEnumerable<string> args)
         { return SplitArgsInto(mandatory,
          optional, args); });
  }

  public static
     Functional.Func<string, bool>
     Validate(Dictionary<string,
     string> mandatory)
  {
    return new Functional.Func<string,
               bool>(delegate (string result)
    {

      if (!string.IsNullOrEmpty(result))
         throw new ArgumentException(result);

      foreach (KeyValuePair<string,
          string> kv in Functional.filter(
        delegate (KeyValuePair<string,
                  string> kv)
        {
          return string.IsNullOrEmpty(kv.Value);
        },
        mandatory))

        throw new ArgumentException("The user
           must provide a valid " + kv.Key);
      return true;
    });
  }
}
```

Listing 14

```
using System.Math;

Func<int,int> f1 = new Func<int,int>(delegate (int a) { return a+2; });
Func<int,int> f2 = new Func<int,int>(delegate (int a) { return a-2; });
Func<int,int> f3 = new Func<int,int>(delegate (int a) { return a*2; });
Func<int,int> f4 = new Func<int,int>(delegate (int a) { return a/2; });

Func<int,double> g1 = new Func<int,double>(delegate (int a) { return Sin(a); });
Func<int,double> g2 = new Func<int,double>(delegate (int a) { return Asin(a); });
Func<int,double> g3 = new Func<int,double>(delegate (int a) { return Cos(a); });
Func<int,double> g4 = new Func<int,double>(delegate (int a) { return Acos(a); });
Func<int,double> g5 = new Func<int,double>(delegate (int a) { return Tan(a); });
Func<int,double> g6 = new Func<int,double>(delegate (int a) { return Atan(a); });

Func<double,double> h1 = new Func<double,double>(delegate (double a) { return Sinh(a); });
Func<double,double> h2 = new Func<double,double>(delegate (double a) { return Cosh(a); });
Func<double,double> h3 = new Func<double,double>(delegate (double a) { return Tanh(a); });

Func<int,int>[] arr1 = new Func<int,int>[] {f1, f2, f3, f4};
Func<int,double>[] arr2 = new Func<int,double>[] {g1, g2, g3, g4, g5, g6};
Func<double,double>[] arr3 = new Func<double,double>[] {h1, h2, h3};

int[] initialiser = new int[]{1,3,6,-10,-12,0};

List<double> output = new List<double>();
foreach(int i in initialiser)
  foreach( Func<int,int> f in arr1)
    foreach(Func<int,double> g in arr2)
      foreach(Func<double,double> h in arr3)
        output.Add(i > f.Then(g).Then(h));
```

The Contradictions of Technical Recruitment

Huw Lloyd reflects on the interview process.

Dig through the archives of any long-standing software development forum and you're likely to find discussions on how to conduct interviews for the purposes of increasing the project headcount.

Usually the hirers will have a particular set of activities in mind that need to be undertaken, and an appreciation for the technologies they would like the candidates to use in fulfilling the work. From this the hirers formulate a description of the candidates they wish to hire. There then follows, provided the hirers are fortunate, a steady flow, or trickle, of candidates to interview.

Interviewing the candidates presents a small but surmountable problem of verifying their technical competence, which may be resolved by a battery of test activities. If the candidate navigates the tests and they presented themselves well, they may then find themselves on the final shortlist for potential hiring or employment.

This is the usual scene for technical recruitment. To elaborate, here's a brief job description that an agent sent to me earlier today:

C# – Greenfield – Front Office – Contract – London – £650pd

I am looking for expert C# Developers to work on 2 Greenfield projects in a leading city based investment bank:

1) Collateral Optimization Engine (Fixed Income) – Up to £650 – City based – Expert front-to-back C# (not asp.net) – Must have Fixed Income or Futures front office experience
2) FX Options eTrading – Up to £650 – City based – Expert server-side C# (C++/Java) background – Must have investment banking/hedge fund experience.

For completeness, here are some details the agent omitted:

1. The client is expecting you to work flat out for a year at least. Ideally they will want to hold onto you until the next time they have to make some sweeping budget cuts. Four years would be 'solid'.
2. They want you to know everything about the technology that is required and they want you to undertake any jobs they see as urgent.
3. Despite the greenfield adjective, you're well aware that in this kind of culture you don't get to spend much time tinkering with design options, or looking into space.

Now, there are different ways to read this brief description, which constitutes the first gambit of an agent to acquire a fresh batch of CVs and phone numbers for this and other, less glamorous, projects. Let's assume you've had some experience with this situation and that you qualify for the role or, rather, that you can see a way of presenting yourself such that you may be construed as qualifying. Let's also assume you're aware of what 'must have investment banking experience' means. Here's the question: does this 'usual scenario' seem reasonable, where surely here the case is quite a straightforward one of hours worked in a professional manner for high pay, with anything extra being a bonus?

Irrespective of whether it seems reasonable, it is, I assure you, normal. In the following sections, I present several interrelated contradictions to this model of recruitment.

HUW LLOYD

Huw Lloyd is a software developer, tutor and researcher in educational psychology. He can be reached at huw.softdesigns@gmail.com

The contradictions of the expertise and development

The first and basic contradiction that manifests itself here, and elsewhere (repeatedly), is development. How did you, the expert programmer, acquire the skills that are being sought by this agent? What personality traits were central to this activity? If amongst those traits you omitted curiosity, learning, elegant design, insight and creativity, you're going to fall short of being a bona fide expert. If, on the other hand, you do exhibit these, and other, traits, what kinds of impoverishing things are going to happen during those years of high pay when these traits are denied expression?

Perhaps the best way is to present yourself as an expert and then proceed to become the expert on the job

There are ways to resolve this contradiction, of course. Perhaps the best way, given the all round naïvety, is to present yourself as an expert and then proceed to become the expert on the job, this way there'll be things to learn. But this doesn't seem to be what the hirers want, does it? On the other hand, perhaps you have acquired all the syntactic and empirical knowledge required for passing the technical tests, consider yourself an expert, and other than being a bit confused as to why some people go on about design, collaboration and something called concepts, you've no reason to let reflection get in the way of rolling up your sleeves and getting on with the work. Is this the expert that we want to unleash on core company architecture?

Let's go back to the original need to increase the project headcount and contrast this with the source of our basic contradiction, which is development. Ah, the Newtonian model of project management has distracted us. What we really need is 'to increase the numbers on the team, such that all of the team members experience some improvement in their working environment, with increased satisfaction in doing the work that is required, whilst being presented with the kinds of challenges that meet with their daily personal conduct, such that they will be working as satisfied productive team members for several years, if not more'.

Great, progress. That seems reasonable as a first effort, but we're in a hurry, so let's refine this so that an agent can recite it, and then do our interviews.

The contradictions of testing and integration

We have interviews to conduct! That means we'll need those technical tests from last time again, because without them, we might end up with someone not suitable...

Technical tests... Not suitable...

Could it be?

Is it possible?

Could it be that those technical tests mean that we end up with someone not suitable? Is that possible? Yes, it is perfectly possible. Here are some reasons why.

Firstly, there may be significant inconsistencies amongst the interviewers. It may well happen that an interviewee will score well with one technical interviewer and not with another. One conclusion to draw from this is that the interviewee knows how to solve one set of problems but not the kinds of problems presented by the second interviewer. That's one way of looking at it. But what if these two interviewers have very different models of what is required?

I recall an interview I attended for a large consultancy. Upon greeting me the interviewer, a senior consultant with the firm, turned to the whiteboard, drew four dots in a square formation on the board and said, "What's this?" I looked at him whilst weighing up my various answers, "four dots on a whiteboard" was probably the best compromise between accuracy and patronization. For this interviewer silence was evidently a sign of ignorance, clearly I did not know what a scope operator was.

The person we are looking for is going to integrate into a team and the team as a whole will hopefully work better once he or she is fully on board

On another occasion, with a different consultancy, I was requested to create and work through an entity relationship model for the interviewer, who was 'the client'. "When I am sitting down", the interviewer said, "I am the client." He paused, looked around casually, stood up and said, "And when I'm standing up, I am the interviewer."

Clearly these two interviewers had elaborate models and expectations. The question is whether the other interviewers agreed with them!

Note also, that these assumptions also relate to the other themes, for instance as assertions of authority and seniority. In the case of the interviewer as client, do you, as the interviewee and potentially more junior employee, agree to pretend to pretend, whilst the interviewer gets to write the script and pull the rug away when he so chooses, or do you confront him?

A second condition of unsuitable testing, is that you're testing (exploring would be a better word) the wrong thing.

Let's return to the developmental mode of consideration. The person we are looking for is going to integrate into a team and the team as a whole will hopefully work better once he or she is fully on board. How do we determine whether someone is going to integrate well?

The standard approach to interview testing is to determine what the candidate does and doesn't know. Yet this will not tell you anything about how the candidate works with others and whether he or she will integrate with the team.

Furthermore you will be fooled if you believe that the 'not knowing' results from such a test equate to 'can't do'. 'Not knowing', rather, means 'can't do on their own in this situation'. Now this reach, of what can be done with some assistance, is actually a variable property. Some people with a little extra assistance can do a great deal more, whilst others may not gain the same benefits – their knowledge and methods of activity may be closer to the binary model that testing implies. This means that some of the candidates who do very well at the technical tests are unlikely to integrate very well on things that they don't know how to do, and are more likely to take particular, potentially divisive, strategies regarding the kinds of 'collaboration' they are used to. [1]

For some scientific researchers of psychology this notion of cognitive reach contributes instrumentally to studying development. This potential for development, mediated by appropriate instruction, is often called the "zone of proximal development" [2]. Hence, how far you are able to reach in an unfamiliar setting contributes to informing us beyond what in this particular instance you're able to do. It tells us that you can listen carefully to others, that you can identify what is not known, and that you can build upon relevant information that is provided such as through auxiliary symbolic models.

A second interesting aspect of cognitive reach is that it has reciprocity too. If you are able to conduct assisted activities successfully far beyond your usual current independent limits, it is likely that you will also be equipped to provide the right kind of information and orientation to help others reach beyond theirs. This sounds like exactly the kinds of skills we'd like for team integration, doesn't it?

More contradictions of expertise and development

Hirers and, by proxy, interviewers are often focused on short term project needs and the competencies of the candidates, rather than on longer term development. If, as an employer, you have an urgent need for a developer, then you have something wrong with your enterprise, and it isn't the absence of developers.

There are plenty of competent developers who would like to work in, or for, a place that honoured their own development. An urgent need for developers is a clear message about the kind of culture being recruited for. For example, I have seen many job descriptions labelled urgent that also stipulate the necessity for a particular syntax or library as a 'must have'. If you're looking to hire someone for many years, then what is a month to gain a working memory familiarity with your particular syntax of choice amortized over this duration?

This issue leads to a third and more profound point about cognitive reach, which is that the things we are able to do with assistance foreshadow what we will be able to do independently in the future. This is worth reflecting on.

Fundamentally the mindset for this kind of mutual support in reaching is grounded in a helping mindset which is often anathema to a culture of demonstrating cleverness and claims to authority.

The basis of the helping mindset is one of process; it's about how we do things rather than what, in particular, we do. If, as a helper, you are focused on telling others what to do, you are acting as an expert [3]. If your only mode of helping is telling others what to do, collaboration and team integration are not going to be improving noticeably, because you're not providing a framework for others' development.

In other words, to play the expert role without attending to development, is actually tantamount to denying the opportunity for others to develop. Instead, you are contributing to establishing a hierarchy of 'knowledge power': a culture of silos, where experts are urgently required at high rates of pay.

The contradictions of assessment and understanding

Interviewers often employ a school model of knowledge, which arguably isn't right for schools either.

One of the things many of us learnt at school is that the exams we took and the means of passing them do not have that much to do with whether a pupil understands or appreciates the subject. The exams are incidental to genuine learning. Yet many people persist with the notion that tests demonstrate whether the critical capacities for knowledge are present. Whether they can repeat an answer, formulaically. Yet this unthinking answer is as far from a real, conceptual, answer as correctly reciting the syntax for an SQL statement is from an appreciation of normalisation. Such a situation would be laughed at were it not for the state of the whole industry and our wider culture.

More specifically on the theme of technical assessment, the interviewer may not know the important difference between empirical concepts and theoretic (or scientific) concepts [4]. It is quite possible that the primary interviewer for these high paying jobs does not know the difference. In such circumstances, they will be unable to discriminate between superficial technical knowledge and technical insights that pave the way towards orders of power of magnitude in improvements of software capability.

If your only mode of helping is telling others what to do, collaboration and team integration are not going to be improving noticeably

Knowledge is certainly key to the successful activity of software development. However, some of this knowledge is far more superficial and subject to change, such as knowing the particular syntax for an API for instance.

Other forms of knowledge are more efficacious: knowing when to probe around to find out how others have dealt with a particular problem, knowing how to partition a system or write a class, knowing the various constraints and affordances of particular means of writing software such as scripts or programming languages, knowing how the procedures and standards of the organisation influence the choice of means of implementation, knowing what it means to persist data or what impedance mismatch is, or having a notion of the concurrency capacity of a particular server instance. These forms of knowledge are more significant than recalling the particular syntax for a left outer join.[5]

This insightful knowledge is theoretic and, as has been explained by noteworthy scholars such as Dewey [6] and Piaget [7], these ideas cannot be communicated directly. Rather they must be constructed personally to appreciate them. This means that the methods employed by a 'non-technical' interviewer must be skewed. The damage would be less if we recognised this. However, many technically competent people take the interview methods of the non-technical as de facto standard practices, thereby discarding any means of attaining a high quality, nuanced evaluation.

perhaps the interviewer has a hidden agenda: to some people, the smarter you are, the more of a threat you are

The contradictions of investment and value

What is the programmer's attitude as she leaves her desk to interview a candidate, in order to give some feedback to the hirers about the technical competence of the candidate? Perhaps, if she has done some preparation, she will have a portfolio of programming tasks or puzzles to solve. If she has done less work, she may have a few sheets of code in which the object is to find the bugs. The basic point is that the test presented will be more a reflection of the tester's notion of what programming is about and in particular where they spend much of their time.

Where, psychologically, people spend much of their time is often conflated with what is valued, which is reflected in attitude. If you poll people for what they value most (such as their favourite book [8]) many people will select the activities that take the longest. It is processes of this sort, a variant of cognitive dissonance, that lead programmers to test another programmer's debugging skills.

I do not wish to devalue the skills inherent in debugging of a real kind, which is often a highly conceptual activity, yet I have not encountered this kind of real debugging during interviews. Interview debugging is often mocked up and artificial.

Then there are the programming interviews about demonstrating cleverness which, as we've seen, is often an indirect reference to claims of authority. Yet much of all the variants of this politicized access to knowledge is merely a familiarity with protocols, such as knowing where to go to diagnose a particular problem in a particular system. (Polyani [9] describes this as "absorbing elements through an operation".) Anyone can acquire this kind of expertise provided they stick around long enough. This certainly satisfies an older generation's notion of seniority within a company based upon length of service, but is that the kind of knowledge and respect we aspire to?

Note also, that if you have a culture of expertise along these lines there will be vested interests in not changing the way things are done or, at a push, certainly not changing them quickly. This applies even if there are clearly better ways of conducting the work. Provided the work is sufficiently muddled, in an expert culture there will be people who believe they stand to gain by keeping things that way. [10]

The contradictions of formality and intuition

The interviewer may be looking for something to fail the candidate on, because there's something else not right. Perhaps the interviewer has intuitions which are difficult to express using their interview model. Or perhaps the interviewer has a hidden agenda: to some people, the smarter you are, the more of a threat you are. Either way, a simple way forward, for this style of interview, is to continue presenting technical tests until the interviewee makes a mistake.

The role of intuition in the workplace is pervasive. Intuition entails a non-conscious mode of directing ones awareness, rather like the experience of having a word or answer on the tip of one's tongue. If we design our processes such that intuition is disregarded, we deny a powerful source of creativity and a powerful means of overcoming frustrating circumstances. For example, the side effects of a managerial response to change efforts, by requesting a well articulated alternative may lead into quicksand. This is simply because the necessity for a well articulated description coupled with an absence of resources and conditions to formulate the articulation results in status quo.

Widening channels of communication to encourage the translation of intuitions and speculations into clear explanations can be given a boost by incorporating this awareness into the process of recruitment. Workplace awareness for intuitive processes has significant cultural ramifications for facilitating where the creative work takes place in an organisation. Is it to be found in the half-starved processes that are reduced to a few private moments between meetings, or can it be facilitated as the core activity of those collaborations, that energise the work of a whole team?

Achieving a culture of creative participation of this form is no easy endeavour. However, even small shifts in the direction of inclusiveness and mutual permission for this creative attitude can have considerable benefits. Giving credence to these processes during interviews and recruitment is an example of such a shift.

The contradictions of partitioning and sharing work

From a technical stand point, the manner of conducting interviews has tangible implications for the enterprise, including the structure of the code. This is, essentially, the point of Conway's law [11]. A fragmented or partitioned team will, at best, produce a fragmented and partitioned software product. But this is not the kind of partitioning that good design entails. This is a partitioning where one subsystem is built without any internal relatedness to the other subsystems.

As many companies begin to explore further the potential for distributed work, these issues stemming from cutting corners and paying lip-service to technical problems, by only heeding empirical practice, will come increasingly and frequently to the fore. Typically there exists a spectrum of approaches to remediate this situation. At one end presides the acknowledgement of development along with all that it implies. The opposite end, which some might refer to as the conservative approach, entails imposing yet greater and more rigid constraints.

Conclusion

I have enumerated key contradictions inherent in the problem of recruitment, which implicate team cohesion, effectiveness and development. These and other contradictions are interrelated. They comprise a complex whole, whereby each facet cannot be treated successfully in exclusion to the others.

A central theme has been the critique of the normative methods of recruitment that lead, in a self-fulfilling manner, to a perpetual and often urgent search for technical expertise. In juxtaposition to the expertise model, I have presented an alternative, viable model of development. The developmental model presents challenges too, yet I believe the arguments in favour of it are compelling.

I hope to have provided some readers with a pause for thought regarding their practices and culture. I also hope to have provided some pointers for improving the recruitment process, particularly the implications for the developmental alternative. For an effective organisation, where team integration is genuinely sought, the architects of the organisation, which include all those involved in the process of recruitment, will need to confront these issues. ■

Acknowledgements

Thanks to Shane Hastie, Kevlin Henney and Helen Lloyd for their reviews.

Thanks also to the recruiter, who gave permission to quote the job details.

References, notes and further reading

[1] I am fully aware of the need to create circumstances and supportive environments for those people labelled with particular psychological symptoms. In a supportive environment discrimination means 'to notice differences'.

[2] Vygotsky, L. S., (1987) 'Thinking and Speech', in *The Collected Works of L. S. Vygotsky*, Volume 1. Plenum Press.

[3] Schein, E., (1999) *Process Consultation Revisited: Building the Helping Relationship*. Addison-Wesley Longman.

[4] Davydov, V. V., (2008) Problems of Developmental Instruction, A Theoretical and Experimental Psychological Study. Nova

[5] Caveat lector. Not all syntaxes are equally arbitrary. Some syntaxes such as the equals sign in mathematics indicate concepts. A clinical precision in their use has more profound implications.

[6] Dewey, J., (1967) 'Psychology', in John Dewey, *The Early Works 1882–1898*, Volume 2. Southern Illinois University Press.

[7] Piaget, J. (1976) *To Understand is to Invent*. Penguin.

[8] *Lord of the Rings* came top of the BBC's 'The Big Read', a poll conducted in 2003. See http://www.bbc.co.uk/arts/bigread/

[9] Polyani, M. (1958) *Personal Knowledge: Towards a Post-Critical Philosophy*, p61. Routledge, Kegan & Paul.

[10] This is the tip of the iceberg, the domain name for which is 'organisational development'.

[11] Conway, M. E. (1968), 'How do Committees Invent?', *Datamation* 14 (5): 28–31

Festinger, L. (1957) *A Theory of Cognitive Dissonance*. Stanford University Press

This article was first published on infoq (http://www.infoq.com/articles/contradictions-technical-recruitment), August 2012

MSc in

Software Engineering

(part-time)

UNIVERSITY OF OXFORD

- a flexible programme in software engineering leading to an MSc from the University of Oxford
- a choice of over 30 different courses, each based around an intensive teaching week in Oxford
- MSc requires 10 courses and a dissertation, with up to four years allowed for completion
- applications welcome at any time of year, with admissions in October, January, and April

www.softeng.ox.ac.uk

The Advanced Coding Test

Pete Goodliffe drives his point home.

Logic will get you from A to B.
Imagination will take you everywhere else.
~ Albert Einstein

For your intellectual delight and mental edification, I present a train of thought my mind idled over one autumn evening. I won't claim that it's a mature thesis; it is merely an interesting thought experiment. But it will perhaps help you ponder your present programming prowess. And potentially provide a plan to polish it.

Second nature

After enough experience, we find that the practice of software development becomes *second nature* for a programmer. Have you reached this stage?

Once you become familiar with the syntax of your programming language, with the concepts of good program design, and have learnt to appreciate the difference between good and bad code, you find yourself naturally making good (or at least reasonable) coding decisions without any discernible effort.

Most of the time you do not unduly tax the grey cells. A lot of coding activities and 'design in the small' becomes instinctive. Correct syntax comes out of the finger's muscle memory, and the code structure you construct appears to be 'obvious'.

This state of working isn't necessarily mindless coding – the bad practice of 'shooting from the hip'. (Although it is a symptom of how some poor programmers mindlessly work.) It's just the way high-calibre programmers work.

It's a good thing. This is what experience gives you.

This state is defined by the Four Stages of Competence model [1] as unconscious competence; something we are able to do without consciously thinking about it. We can perform this task effectively, without even realising exactly what we're doing and how difficult it is.

Driving the point home

There are many activities in which we achieve a state of unconscious competence. Some are professional. Some are far more mundane. Most humans can walk and eat without carefully considering each movement, and without making complex prior plans to enable them to do so.

Another great example is driving a car.

Driving is an interesting analogue of programming; there are some interesting lessons we can pull out from a comparison of the two.

It takes a significant amount of learning to become a competent driver. It requires effort to learn the mechanics of the car, as well as learning the etiquette and rules of the road. Driving well requires a concert of actions and skills; it's an intricate process. You have to invest a lot of effort and practice to achieve competence at this.

When a new driver first passes their driving test they are at the *conscious competence* stage of learning. They know they can drive, and they have to pay attention to carefully coordinate all the contending forces. The selection of a new gear is a conscious process (for those enlightened drivers with manual transmission). Mastery of the clutch requires thoughtful balance.

But after more experience, a lot of these mechanics become automatic. We gain confidence. The controls and handling of the vehicle become second nature. We become accustomed to how the vehicle responds to the controls we adjust. We naturally adopt the correct road positioning. We become masters of the operation of the vehicle.

Once a driver reaches this stage, their attention is freed up to concentrate on the remaining unknown – the road itself, and the decisions that it constantly presents.

How much does this sound like programming?

Some people are better drivers than others. Some people have better tools (their vehicles) than others. Some people have more natural ability than others.

How much does this sound like programming?

The majority of problems on the road – the accidents, delays, etc – are due to *driver error*. Crashes happen to cars but they are caused by the people who learnt to use them.

Seriously: how much does this sound like programming?

A little knowledge is dangerous

The state of conscious competence can, if you're not careful, lead to a complacency. Rather than concentrating on the road, you end up 'coasting', driving on automatic (in the metaphorical sense, not the transmission system). Rather than looking out for hazards on the road ahead, you're thinking about what to eat for dinner.

It is important to overcome this kind of complacency to become a better driver. Otherwise, frankly, you are a liability. You could very easily do more harm than good.

How much does this sound like programming?

Do you have what it takes?

Before you are let loose in a vehicle, you have to prove that you are capable. You have to pass a *driving test*. It's illegal to drive on public roads without having first passed this test. The driving test proves that you have the necessary skills, and the responsibility, to drive. It demonstrates that you can not only handle a car, but can make good decisions under the pressure of the conditions of the road.

Now, there isn't a direct equivalent of a driving test in the programming world; certification is not a legal prerequisite to write code (nor should it be, in the author's opinion). But, to enter gainful employment you *do* usually have to demonstrate a reasonable level of skill: having passed a course, or be able to prove demonstrable prior experience.

In the UK, and in many other countries, there is an extra level of certification that a driver can earn, a far more rigorous test. This is the *advanced driving test*.

To pass this test you have to demonstrate a superior level of control of the vehicle in more demanding driving conditions. You must possess a far greater level of awareness of road conditions, of what's going on around you, and where potential hazards are. An 'advanced' driver concentrates more fully on their driving, and has better anticipation and planning.

It is a higher standard of driving.

This advanced test is not mandatory. But some occupations *require* drivers to have earned a higher qualification before they can drive professionally.

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

PROGRAMMING IS LIKE DRIVING
IT REQUIRES SKILL AND PRACTICE
PRACTICE MAKES PERFECT
« syntax error at line 1982 »
AND CARELESSNESS INEVITABLY LEADS TO CRASHES

For example, police drivers have to undergo far more rigorous training and pass a very stringent driving test.

Even if it is not mandatory professionally, achieving an ‘advanced’ driver status imparts many benefits:

- You learn greater skills.
- You run a greatly reduced risk of accidents.
- A gain a better awareness of the limitations of the equipment you use.
- An understanding of the dangers and pitfalls inherent in driving.
- The ability to make better decisions, with greater confidence; to be more decisive.
- To be more considerate to other people.
- Also, there is a selfish motivation: advanced drivers have reduced insurance premiums because they are far less likely to be involved in accidents!

It’s not tenable to expect every driver to achieve ‘advanced’ level before they are allowed on the road. Indeed, it doesn’t make sense.

But the advanced level is definitely something to aspire to.

The equivalent?

So here’s the thought experiment: what would the equivalent of an ‘advanced’ driving test for software developers look like?

We often argue about the true value of certification in our industry. A lot of the certification peddled by training organisations is pure bunk, snake oil that helps you tick boxes in job application forms.

But some of it *is* valuable.

Would a physical test be of any use? What would it look like? How specific would it have to be to specific technology areas? Would this specialisation make it impractical to run? I’m sure there are coders that you respect, and who you *recognise* as advanced. But is it possible, or practical, or useful, to realistically certify them as such?

Even if we don’t try to define a strict ‘advanced test’, what skills would you expect of an ‘advanced’ coder?

As we’ve seen, the majority of advanced programmer skills are gained by experience garnered on the job. But not every long-serving coder continues to learn and hone their skills. Time on the job is not enough. Indeed, advanced coding skills are orthogonal to developer promotion path. If you serve faithfully in a job for *n* years your company might give you a pay rise and allow you to climb another step up the corporate ladder. But this doesn’t necessarily mean you’re any better a programmer than when you joined.

An advanced programmer will demonstrably:

- Be confident in their abilities. And confidently know when they’ve reached the limit of their skills.
- Make better decisions.
- Be more mindful of the work they are doing.
- Have a greater set of skills, and mastery of the tools they are using.
- Anticipate what other coders might find ‘unexpected’ and already have a plan to deal with these situations

In many respects, this is an analogue of the list of characteristics of an advanced driver that we saw above.

One of the classic skills an advanced driver must perform is driving whilst giving a running commentary. The driver mindfully describes what they can see, what they’re thinking, and what they plan to do *as they drive*. This shows that they have identified potential hazards and have already considered how they will react.

How might we do that as programmers? How well would design reviews before coding help foster a similar state? Does pair programming, and the constant conversation that this develops, give a similar kind of benefit?

Conclusion

Think for yourself and let others enjoy the privilege of doing so too.

~ Voltaire

This is a little thought experiment. Nothing more. But it’s interesting to think about this kind of thing, to provide a framework that might help us become better programmers.

It’s certainly very valuable to consider the stages in our coding abilities. To determine when you’ve moved from conscious competence to unconscious competence. And to recognise that you can progress from a ‘standard’ level to more ‘advanced’.

Questions

1. What is the programmer’s equivalent of a driving test? Could there be such a thing?
2. Are your programming skills at the ‘standard test’ level or at the ‘advanced level’? Do you think you frequently achieve the *unconscious competence* level?
3. Do you want to maintain your current skill level? Do you want to improve it? How will do you this?
4. How could you test a programmer’s ability to perform an ‘emergency stop’?!
5. Is there any extra value to be gained from investing in your skills? If advanced drivers enjoy lower insurance premiums, how does being an ‘advanced coder’ materially benefit you?
6. If coding is like driving, do we treat code testers like crash test dummies? ■

References

[1] Four Stages of Competence: http://en.wikipedia.org/wiki/Four_stages_of_competence

Standards Report

Mark Radford reports the latest from the next version of C++.

The international C++ Standards' Committee met in Portland, Oregon during the week Monday 15th – Friday 19th October. There were around eighty attendees, which means attendance is on the up (typically meetings were attended by forty to fifty people during C++2011 development).

Working towards C++2017, the time scale is quite aggressive and there is much work to be done. Therefore, study groups have formed, each with its own remit. There are six currently active study groups: concurrency, modules, the file system, networking, transactional memory, and numerics. These are in addition to the normal workings groups, consisting of the core (CWG), library (LWG) and evolution (EWG) working groups. The idea is that the study group carries out initial investigation work, before detailed specifications are drawn up at CWG, LWG or EWG level.

The working groups met daily, as did the concurrency study group. However, owing to there being only one more more room available, other study groups each met for half a day at a time. The exception to this was the transactional memory study group: while they only met occasionally, their meetings were co-located with the concurrency group, because of the overlap in the two subject areas.

While all this was going on, there were still issues and defect reports (relating to the current standard) to be processed. These were handled by the CWG or LWG as appropriate. Meanwhile the EWG was handling requests and proposals for new features. Again, this is where study groups become relevant: after looking at a proposal or new feature request, one course of action for the EWG is to delegate the next phase of the work to a study group.

You have probably figured out from the above, that concurrency is where much of the action is at the moment. For that reason, much of the material I will now discuss is concurrency related.

Papers

Obviously many papers were discussed at Portland, and, owing to time and space limitations, I can only mention a very few of them. However, all the papers can be found at http://www.open-std.org/JTC1/SC22/WG21/docs/papers, and in particular the pre-Portland mailing can be found at http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2012/#mailing2012-09.

Some of the issues associated with output streams in multi-threaded code are examined in *C++ Stream Mutexes* (N3395). This proposal addresses the problem that stream output operations, while currently guaranteeing that they will not produce race conditions, do not guarantee that the effect will be sensible. This is problematic because many applications want to write spontaneously to an output stream, such as when writing debug information to a log file. The concurrency group was in general agreement on the need to provide this stream locking facility. The consequence of not doing so is likely to be that **`printf()`** is used instead (because Posix provides some threading guarantees).

The current standard library component **`std::queue`** is designed for use only in sequential code. *C++ Concurrent Queues* (N3434) addresses requirements of queues in concurrent code. Some of the operations look the same, or similar, to those already familiar from the existing **`std::queue`**. The differences are in the interface's operational aspects. For example, pushing an element onto the queue may not be possible immediately in a concurrent context. The push operations described in the paper wait until it is possible to push the element onto the queue. Note the presence of member functions such as **`try_push()`**: these execute an operation only if it can be executed soon, and if it can't be executed soon it is not executed at all. That is to say, if the queue is full then waiting is avoided, however **`try_push()`** will block if it is just a matter of waiting for a mutex lock to become available. These functions return a status to indicate whether or not the operation was actually executed. The interface described is an abstract one so, for example, concrete classes could implement bounded or unbounded queues. There was some discussion of this paper, for example, on the possibility of also having a concurrent stack. Another point of discussion was on the **`value_pop()`** member function, and how to make it exception safe. The conclusion was that the basic exception guarantee is the best that could be offered. Just to explain that last point a little: remember how **`std::queue`** (and **`std::stack`**) have separate functions for querying and popping the front/top element, because that is the only way to be exception safe? The problem is, that approach doesn't work in a concurrent context, because the element at the front/top may be changed (by another thread) between the query and pop operations.

There are many other interesting concurrency papers that I would like to cover but can't. For example, you might want to look up *Shared locking in C++* (N3427), and *async and ~future* (N3451). The former is a proposal for improving the thread locking library to facilitate multi-reader/single-writers, while the latter addresses the problem of surprising blocking behaviour in **`~future`**. The latter has caused a lively discussion, which I hope to come back to. However, before I end this column I was to briefly move out of the concurrency domain.

The paper *A Database Access Library* (N3415) begins to address the need for, well, a database access library, in C++. This is something C++ will need if it is to provide a full set of general purpose libraries – and, provide a full set of general purpose libraries is something it needs to do in order to remain useful as a general purpose language. This proposal was discussed by the EWG. Although it has not been done yet, there was interest in forming a new study group to work in this field.

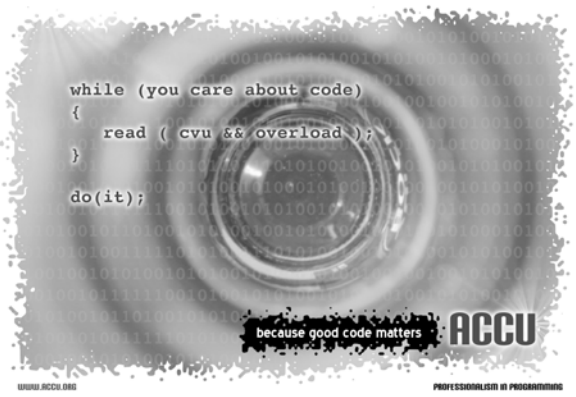
Finally

That concludes my column featuring the Portland meeting, sadly with more left out than went in. There is much of interest for me to (hopefully) come back to in future columns.

I would like to thank Roger Orr and Alisdair Meredith for posting updates to the BSI C++ Panel reflector during the week of the meeting. Also I would like to thank the CVu editor, Steve Love, for his patience that allowed me to get coverage from Portland into this column.

MARK RADFORD

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk


while (you care about code)
{
read (cvu && overload);
}
do(it);
because good code matters
ACCU
WWW.ACCU.ORG
PROFESSIONALISM IN PROGRAMMING

Agile East Anglia: A Short History

Paul Grenyer gives us a potted history of a local group from its inception in December 2011.

It's well know within the ACCU that I was born in Norwich, Norfolk, UK. After some years working away, I am now back home and working in Norwich. I've always enjoyed the banter about Norwich being a backwater and not somewhere for someone who wants a career in software development because until recently I believed it too. However, in the last year the technology community in Norwich has got together to make itself known.

Agile East Anglia was started by me as an Extreme Tuesday Club (xTc) inspired pub meet on Monday 5th December 2011 at the Coach and Horses on Thorpe Road in Norwich. Attended by a handful of people from local firm Aviva it was followed on Monday 9th January 2012 by a less well attended meeting at the same place.

On a cold and snowy Monday 6th February, with sponsorship from Ipswich based consultancy firm Smart421, Agile East Anglia put on a presentation on Agile User Stories given by well known Agile consultant Rachel Davies at The Assembly House in Norwich. Around 20 people attended, predominantly from Aviva, but also people from other firms such as Archant, Smart421, Axon Active AG and Proxama.

On Monday 26th March, Agile East Anglia put on a second Agile presentation: this time it was a Dialogue Sheets workshop given by Agile consultant Allan Kelly. Again it was at the Assembly House in Norwich and sponsored by Smart421. Around 20 people attended. This time there were more people from local East Anglian companies such as Ifftner (Ipswich), Redgate (Cambridge), Call Connection (Ipswich) and Purple Tuesday, as well as people from Archant, Smart421 and Axon Active AG.

On Tuesday 10th April, I presented The Walking Skeleton for the newly formed facebook group, Norwich Developers Community. The group was created and run by Stephen Pengilley to bring together Norwich's software developers.

On Thursday 19th April, Agile East Anglia regular John Fagan gave a presentation on the Lean Startup book for the Norwich Startups meetup group. The group was created and run by Juliana Meyer to bring entrepreneurs, developers, and anyone interested in startup companies in Norwich together. Starting in September 2011, Norwich Startups had already had a number of well attended meetings and was well established.

It was about this time that John Fagan and I independently had the idea to bring all of the groups together with the aim of making something bigger and even better. Purple Tuesday cofounder Seb Butcher had also expressed interest in getting involved with Agile East Anglia. Stephen Pengilley and Juliana Meyer were both approached with a view to merging Norwich Developer community and Norwich Startups with Agile East Anglia.

On Thursday 3rd May at the Gunton Arms near Cromer, John Fagan, Seb Butcher, Juliana Meyer, Stephen Pengilley and I got together to discuss and shape the new group, which had the working title "On The Code City".

On Thursday 7th June Agile East Anglia held a meeting at the Hog and Armour in Norwich. Nearly 30 people came to hear Agile consultant Liz Keogh speak about behavior Driven Development. Smart421 continued their sponsorship and there were attendees from Ifftner, Redgate, Call Connection, Purple Tuesday, Smart421, Axon Active AG and newcomers Silo18 (Norwich). During the introduction it was announced that Agile East Anglia would be merging with the Norwich Developers Community and Norwich Startups under the name "SyncNorwich". Seb Butcher, Stephen Pengilley and John Fagan were introduced to the Agile East Anglia group and the date of the first SyncNorwich meeting was announced as 5th July, with a presentation from Colm McMullan on his one man startup.

With the formation of SyncNorwich, this became the last Agile East Anglia event. Future meetings had already been planned, including Lightning Talks, a presentation on Kanban from Benjamin Mitchell, a session on iteration planning from Simon Cromarty and a workshop on GIT from Pete Goodliffe. All of these sessions were adopted into the SyncNorwich programme.

In less than a year SyncNorwich has attracted over 300 members and regularly gets in excess of 60 people to its events. Plans are currently underway for SyncNorwich's first conference in February 2013.

PAUL GRENYER

Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com

Code Critique Competition 78

Set and collated by Roger Orr. A book prize is awarded for the best entry.

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

Last issue's code

I'm trying to get started with Python ... tell me why the programme below doesn't work. The program is supposed to find a nine digit number using all of the digits 1 to 9 which is divisible by 9 (not difficult!) but is such that

- removing the last digit gives an 8 digit number divisible by 8
- then removing the last digit gives a 7 digit number divisible by 7 … and so on down to
- then removing the last digit gives a 2 figure number divisible by 2
- then removing the last digit gives a 1 digit number divisible by 1 (not difficult!).

Why does it tell me that `i` is not iterable:

```
$ ./program.py
Traceback (most recent call last):
  File "./program.py", line 8, in <module>
    for i in x:
TypeError: 'float' object is not iterable
```

The program is in Listing 1.

Editor's note: to make this printable I've used one space, not the recommended four, for each indentation and split long lines with a continuation character (\)].

```
#!/usr/bin/python
x=[1,3,7,9]
y=[2,4,6,8]
e=5
for a in x:
 for c in x:
  for g in x:
   for i in x:
    if a<>c and a<>g and a<>i and c<>g and \
       c<>i and g<>i:
     for b in y:
      for d in y:
       for f in y:
        for h in y:
         if b<>d and b<>f and b<>h and d<>f \
            and d<>h and f<>h:
          z=10**8*a+10**7*b+10**6*c+ \
            10**5*d+10**4*e+ \
            10**3*f +100*g+10*h+i
          p,q,r,s,t,u,v,w= \
           z-z%10,z-z%100,z-z%1000, \
           z-z%10000,z-z%100000, \
           z-z%1000000,z-z%10000000, \
           z-z%100000000
          p,q,r,s,t,u,v,w= \
           p/10.,q/100.,r/1000.,s/10000., \
           t/100000.,u/1000000.,v/10000000., \
           w/100000000.
          p,q,r,s,t,u,v= \
           p%8,q%7,r%6,s%5,t%4,u%3,v%2,
          x=p+q+r+s+t+u+v
          if x==0:
           print z
```

Listing 1

Critiques

Ian Bolland <ian.bolland@nut.eu.com>

The error looks mysterious if you read from the top and stop at the line where it is reported. It looks as though three iterations over **x** have succeeded, while the fourth has failed.

If you carry on as far as the line:

```
x=p+q+r+s+t+u+v
```

then the mystery is solved. Initially **x** refers to the list [1,3,7,9] (or in Python terminology is bound to the list). Lists are iterable, so the program successfully starts each of the loops. During the course of the iterations **x** is reset to refer to a numeric value (in Python terminology it is rebound). So the next time the program re-enters one of the inner loops it tries to iterate over a numeric value rather than a list, and so it crashes.

In Python you can bind a name to an object of any type, and can rebind a name to an object of a different type. However, you have to ensure that you never use the name in a way that is incompatible with the type of object that is currently bound to it. This is much easier if you don't rebind a name to an object of a radically different type.

In this case the rebinding is clearly unintentional. The author has used single-character names up to **w**, and wanting a new name has chosen **x**, forgetting that he has already used it to refer to the list [1,3,7,9]. Needless to say, this sort of naming convention is a bad idea in any language.

The **TypeError** can be fixed in various ways. The simplest is to remove the rebinding of **x** and use the RHS expression directly in the **if** test, i.e.:

```
if p+q+r+s+t+u+v == 0 :
 print z
```

With this change the program runs, and prints

```
381654729
```

A couple of minutes with a calculator (or the Python shell) verifies that this is the correct answer. This surprises me – if I ever write code this complicated, it seldom contains only one error. So let,s see how we can simplify the code. The author has gone to some trouble to avoid generating obviously impossible permutations by observing that:

- the first 5 digits are divisible by 5, and so digit 5 must be 5
- the first 2 digits are divisible by 2, and hence digit 2 must be even. For the same reason, digits 4, 6 and 8 must be even
- digits 1, 3, 7 and 9 must be the remaining odd digits.

A comment to explain this would have been useful, even if the rest of the code does not need commenting.

ROGER ORR

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

I'm not convinced of the importance of efficiency in this case, but for the moment I will play along with the assumption, and will start by improving the implementation of the algorithm as written.

To start with, there are a few examples of dubious style:

1. The **`<>`** operator is obsolete: use **`!=`** instead.
2. I don't like the way that the calculations involving **`p`**, **`q`**, **`r`**, **`s`**, **`t`**, **`u`**, **`v`** and **`w`** are combined into single statements. It makes the code less readable than writing each calculation as a separate statement, and incurs a slight run-time overhead constructing a temporary tuple to hold the calculated values.
3. The calculations use three different ways to represent powers of 10. I would use **`10**n`** in each case, to avoid counting zeros. And there is no good reason to use floating-point constants at all.
4. I also dislike the test controlling the **`print z`** statement. I would have written this as:

   ```
   if ( p==0 and q==0 ... ):
   ```

   which more clearly expresses the condition that all the remainders must be zero. Better still, I would have combined it with the previous statement and written the test as:

   ```
   if ( p%8==0 and q%7==0 ... ) :
   ```

And if efficiency is important, it is worth observing that the algorithm only generates permutations for which **`s%5`** and **`v%2`** are zero. In fact **`s`**, **`v`** and **`w`** need never have been calculated – these calculations slow down the program by about 20%.

Moving on to more significant improvements, we note that the two sets of loops and the subsequent tests generate permutations of the sequences **`x`** and **`y`**. This can be done more simply by using the standard library function **`itertools.permutations`** to replace 8 loops and 2 tests by:

```
from itertools import permutations
for a, c, g, i in permutations ( [1,3,7,9] ):
    for b, d, f, h in permutations([2,4,6,8]):
```

and as a bonus we can revert to using the standard 4-space indentation.

The calculation of the values of the first **`n`** digits can also be simplified by holding the digits in an array and calculating the values using the recurrence relation:

```
value[n+1] = value[n]*10 + digits[n]
```

Not only is this simpler, it also allows us to test for divisibility by n at each step, stopping as soon as the current permutation fails one of these tests. This gives:

```
from itertools import permutations

#   Standard tests for divisibility imply that
#   in any valid solution, digits 2,4,6,8 are
#   even, and digit 5 is 5.
#   Only generate permutations which satisfy
#   these conditions.

for o1,o2,o3,o4 in permutations([1,3,7,9]):
   for e1,e2,e3,e4 in permutations([2,4,6,8]):
        digits = (o1,e1,o2,e2,5,e3,o3,e4,o4)
        value = digits[0]*10 + digits[1]
        #  value is divisible by 2 because
        #  digits[1] is even.
        #  Test divisibility by 3..9
        for n in xrange(3,10) :
            value = value*10 + digits[n-1]
            if value%n != 0 :
                break
        else :
            print value
```

The Python **`for`**/**`break`**/**`else`** construct is slightly unusual. In this case it works as follows. If **`value%n != 0`** for any iteration then the **`break`** is executed, the loop terminates completely, and the **`else`** is not executed. If no iteration executes a **`break`** then the loop terminates by executing the **`else`** clause. This means that the **`print`** statement is only executed if **`value%n == 0`** for all **`n`**. If you don't like **`for`**/**`else`** (and some people don't), you can use an extra boolean variable in a normal **`for`** loop to get the same effect.

I have written the loop over **`n`** to avoid testing divisibility by 1 and 2 (which slow the program down noticeably), but I don't mind testing for divisibility by 9, since this is only executed once.

Using the standard library module 'timeit' on the original program gives:

    100 loops, best of 3: 3.04 msec per loop

and on the improved version gives:

    1000 loops, best of 3: 714 usec per loop

So, not only is the improved version shorter and simpler, it is also much faster. Part of the speedup is caused by the more efficient permutations algorithm, but most of it is caused by the avoidance of unnecessary arithmetic, and particularly by the early termination of loops as soon as any divisibility test fails.

However, as I said, I am not convinced of the importance of efficiency in this case. Premature optimisation is a well-known pitfall in software development: the general rule is 'first make it right, and then make it fast'. In a case such as this, where the program is only needed to run once, then it needs to be very slow indeed before it is worth optimising. The optimisation effort might pay off if you can write, test, debug and run the optimised version while the straightforward version is still running. But this is very rare.

So let's look at how a straightforward version might be written. For a start, we can take the brute-force approach of testing all permutations of the digits. Factorial **`n`** looks scary, but when **`n`**=9 it is only 362880.

Secondly, the task of testing the divisors of leading digits becomes easier if we hold the digits in a string. The leading **`n`** digits of **`s`** are divisible by **`n`** if

```
int(s[:n]) % n == 0
```

So we can write the program as:

```
from itertools import permutations
for digits in permutations ( "123456789" ) :
    digits = "".join(digits)
    if all ( (int(digits[:n]) % n == 0
             for n in xrange(1,10)) ) :
        print digits
```

Here:

- The first line makes itertools permutations available.
- The second line generates all permutations of the digits (the string is treated as a sequence of single- character digits). These are returned as a tuple of characters.
- The third line is the Python idiom for concatenating a sequence of strings. Effectively this turns the tuple of digits into a single string. Note that this is a case where rebinding a name to hold a value of a different type is acceptable. The tuple returned by permutations is essentially a temporary value used only to construct the digits string, and there is no benefit in inventing a separate name for it.
- The fourth line tests whether all the leading **`n`**-digit values are divisible by **`n`**, for **`n`** in the range 1 to 9. I have used **`xrange(1,10)`** to keep the solution as close as possible to the initial problem statement. **`xrange(2,9)`** would also work in this case, but it might mislead a Python learner into thinking that **`xrange`** generates the inclusive range 2 to 9, rather than 2 to 8.

Using 'timeit' on this version gives:

    10 loops, best of 3: 1.64 sec per loop

which is less time than it would have taken me even to fire up my editor to start writing the optimised version. So I would go with the straightforward version.

Paul Evans <paul.xa.evans@barclays.com>

It's actually saying that **x** isn't iterable and that's because Python changes the dynamic binding of the variable **x** from the list [1,3,7,9] (set in line 2) to be bound to the **float p+q+r+s+t+u+v** (set in line 31) in the first iteration of the **for** loop at line 8. Simply renaming the variable **x** in lines 31 and 32 to **m** removes the bug and the program now gives the correct answer of 381654729.

The indentation problem can be solved along with a major cleanup of lines 5 to 16 by using the **permutations** function from the **itertools** module. Lines 17 to 31 can also be cleaned up by using a string to handle the 'removing the last digit' part of the problem. Python's **join**, **map**, **str**, **int** and **sum** built-ins along with some list comprehension also work better here. We can eliminate testing for divisibility by 5 and 2 as '(not difficult!)' due to the placement of 5 and the even digits. Finally we can use meaningful names, some more list comprehension and the **range** built-in. Putting parentheses around the **print** argument makes the code Python 3.x compatible to give us:

```
from itertools import permutations as perms
odds = [i for i in range(1, 10, 2) if i != 5]
evens = range(2, 10, 2)
testers = [i for i in range(3, 9) if i != 5]
for o in perms(odds):
    for e in perms(evens):
        n = (o[0], e[0], o[1], e[1], 5, \
           e[2], o[2], e[3], o[3])
        number = "".join(map(str, n))
        if sum(int(number[:i]) % i \
            for i in testers) == 0:
            print(int(number))
```

Ola M <aleksandra.mierzejewska@gmail.com>

The immediate reason for the error showing up is the re-use of the **x** variable. It is used in line 2 to denote a **range**, and changed on line 20 into a **float**. The error shows on line 9, because after full iteration in the 4 inner **for** loops the 3rd **for** loop tries to go to the next element of **x**. But in the meantime **x** was used as a **float**. After changing the second **x** to another name – the program works and gives the correct result.

Although we have working code at this point, still some of its aspects are not best practice. The variable names are meaningless, which made the original error easy to make. There is no need to convert variables **p**, **q**, **r**,... to **float**s with the **.** – they can stay integers.

Most importantly, the author of this code used some mathematical rules to save on a number of calculations, but not all of them. I think that it would be simpler and more consistent to either go the whole way with the maths approach or to simplify the algorithm and go through all possible combinations. After all there are just 9! numbers to check.

Personally I would go with the mathematical approach. In the original solution, the author notices that E must be 5 to be divisible by 5 and that every second number must be even, to be divisible by 2, 4, 6 and 8. If we go further with this approach we can see that number CD (c*10+b) must be divisible by 4, that EFG must be divisible by 8, that sum of A, B and C must be divisible by 3 and sum of D, E, F must be divisible by 3. Once all these conditions are met we are left with six numbers, for which we need to check that ABCDEFG is divisible by 7.

However, as the author mentioned the aim was to learn some Python, the above solution is not really helpful... In that case I'd go through all permutations of the given numbers – that, I believe, significantly simplifies the code. The running time is still negligible and we get the same result of 381654729.

```
#! /usr/bin/python
import itertools

def isPD(perm):
    num=perm[0]
    for i in range(len(x)-1):
        num=num*10+perm[i+1]
        if(num%(i+2)!=0):
            return 0
    return num

x=[1,2,3,4,5,6,7,8,9]
for perm in list(itertools.permutations(x)):
    res=isPD(perm)
    if(res):
        print res
```

Commentary

I was a little nervous of setting a code critique in a different language since sometimes when I've strayed away from C and C++ in the past there have been few entries – but it was good to see several critiques this time. So if you have a potential critique in another language please send it in!

The original problem has three parts – the user's problem with the runtime error, the mathematical problem they are trying to solve and the fact that they're starting to learn Python.

All three entrants did cover the three points, although the lure of improving performance (not mentioned by the original writer) proved hard to ignore!

One of the hardest things for newcomers to a language is knowing what techniques are available. While all three solutions imported **itertools**, I think it might have been useful to give some background to what this does and how you know it exists. Paul mentions the word 'built-ins' which is a useful clue for a newcomer to the language and Ian gives some details about the term 'rebinding' and why it must be done with care. It can be hard though to put yourself back in the mind of a beginner.

I think between them the critiques covered pretty well all the points in the code sample so I've nothing further to add – I hope the student would have learned some useful tips and well as having a working program!

The Winner of CC 77

There were things I liked about each critique; such as Paul's expressive variables names (odds and evens) and Ola's introduction of a function call to separate the problem cleanly into two sections. However, I felt Ian's solution was the most detailed and also his final solution was the clearest to understand, so I have awarded him the prize.

Code critique 78

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

I thought it was time to start trying out some of the new C++11 concurrency features; but it looks like the compilers might be a bit buggy still. Shouldn't the following program cleanly separate log output from the two threads? I started with MSVC 2012 and it worked Ok until I turned on optimising; so I tried g++ with -std=c++11 – both sometimes give mixed lines.

Sample output is in Listing 2 and the code is in Listing 3.

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

```
29 Sep 23:40:37 [1]: starting the calculation
29 Sep 23:40:37 [2]: starting the calculation
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [2]: next...
29 Sep 23:40:37 [129 Sep 23:40:37 [2]: next...
]: next...
29 Sep 23:40:37 [2]: next...
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [2]: next...
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [2]: ending the calculation
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [1]: ending the calculation
```

Listing 2

Two Pence Worth

An opportunity to share your pearls of wisdom with us.

One of the marvellous things about being part of an organisation like ACCU is that people are always willing to help out and put in their two-pence-worth of advice. In this new section of CVu, we'll capture some of those gems and print the best ones. If you have your own 2p to add to the collective wisdom of the group, send it to cvu@accu.org.

'In order to maximise the security of your job, write everything in Powershell.' Chris, Cambridgeshire

'Always repeat yourself: adding comments that repeat the code reinforces your point.' @UncleDave

'Ensure people can identify the provenance of bugs by putting the URL of where you nicked the code from in a comment.' Doctor Love, London

'Consider using more than one letter for variable names. That way you won't run out of names so quickly.' Donald, United States

'Pair programming is a great way to save money on desks and other equipment.' Twitter rumour

'Search and replace is a great cheap alternative to expensive refactoring tools, but make sure Case Sensitive searches are possible!' Cheap Chaz, Cheam

'When pasting code from somewhere, always keep the typos so future programmers know where you copied it from.' Doctor Love, London

'A full-featured IDE editor turns not being a touch-typist into an advantage. Amaze and confound your typing friends with word completion and auto-formatting!' Bill, USA

'Call your tests 'test1', 'test2 etc. so that people know what order to run them in.' Anon

'Upgrade to a modern C++ compiler so that you can start using threads in your code.' A man from Runcorn

Code Critique Competition (continued)

Listing 3

```
#include <iostream>
#include <mutex>
#include <string>
#include <thread>
// get now
std::string now()
{
  time_t const timeNow = time(0);
  char buffer[16];
  strftime(buffer, sizeof(buffer),
    "%d %b %H:%M:%S", localtime(&timeNow));
  return buffer;
}
// log message
void log(int id, std::string const & message)
{
  std::mutex mutex;
  mutex.lock();
  std::cout << now() << " [" << id << "]: "
    << message << std::endl;
  mutex.unlock();
}
// simulate some activity
void dosomething()
{
  int i(0);
  for (int j = 0; j != 10000; ++j)
  {
    for (int k = 0; k != 10000; ++k)
    {
      ++i;
    }
  }
}
```

Listing 3 (cont'd)

```
// calculate
void calculate(int id, int count)
{
  log(id, "starting the calculation");
  for (int idx = 0; idx != count; ++idx)
  {
    dosomething();
    log(id, "next...");
  }
  log(id, "ending the calculation");
}

int main()
{
  try
  {
    std::thread t1(calculate, 1, 5);
    std::thread t2(calculate, 2, 4);
    t1.join();
    t2.join();
  }
  catch (std::exception const & ex)
  {
    std::cerr << "exception: " << ex.what()
      << std::endl;
  }
}
```

Bookcase

The latest roundup of book reviews.

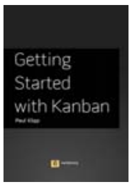
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

Getting Started with Kanban



By Paul Klipp, published by Kanbanery, self-published ebook

Reviewed by Paul Grenyer

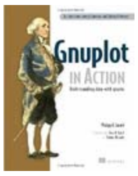
Other than Allan Kelly's 10 things to know about Kanban software development blog post, which is awesome, *Getting Started with Kanban* by Paul Klipp is the only Kanban material I have read so far. I really like these short books which seem to be coming out thick and fast at the moment. I really must get mine ready! It took me less than an hour to get through this book. I suppose it could have been presented for free as a long blog post or an article, but I'm really not bothered paying £1.54 for it. It was worth it.

I literally had no idea about Kanban other than it was a looser Agile (than something like Scrum). I enjoyed reading this book and I learnt a lot in a very short period of time. I am now comfortable with what Kanban is and how it works and I can really see the appeal. I may even have to revise my thinking that to be Agile you have to have iterations.

About half the book is dedicated to an overview of Kanban with a list of other books you should read, including Kanban by David J. Anderson which is next on my reading list, and the final half to a description of the Kanban process that Paul Klipp uses. This really helps give some context to Kanban.

If you want to learn about Kanban quickly and easily, read this book.

Gnuplot in Action: Understanding Data with Graphs



By Philipp K. Janert, published by Manning, ISBN: 978-1-933988-39-9

Reviewed by Fred Youhanaie

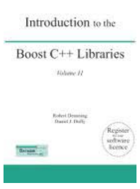
I have been using Gnuplot on and off since the early nineties. Due to its simple interface and the on-line help facility I was able to get by with all simple plotting tasks, so I never bothered with reading the manual, and as a result I never learned of its more advanced facilities. For complex tasks I would instead opt for alternative tools such as Cern's ROOT software. This book, however, has changed all that, tipping the balance in favour of Gnuplot.

The four parts of the book, each with 3–5 chapters, take the reader from a very gentle introduction, to advanced Gnuplot topics, to graphical analysis techniques. All the chapters are well written, with plenty of good examples. The largest of the four parts, Advanced Gnuplot, has dedicated chapters for topics such as 3D plotting and colour, as well as methods for generating plots, and slide-shows, using scripts.

The book is as much about Gnuplot commands and options as it is about its application to graphical data analysis, although for the latter the author also refers you to specialist books. With data driven computing (Big Data) getting more and more attention everyday, this book has its place on the data analyst's bookshelf.

As with most Manning books, those who buy the printed copy of the book are automatically entitled to download the electronic e-book versions free of charge.

Introduction to the Boost C++ Libraries Volume II – Advanced Libraries



By Robert Demming and Daniel J. Duffy, published by Datasim Education, ISBN: 978-94-91028-02-1

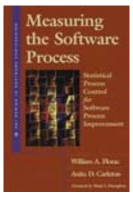
Reviewed by Paul Floyd

Fairly obviously, this books carries on where Volume 1 left off. Roughly, the libraries covered fall into three main categories: maths, network and graphs, as well as a few odds and ends.

I felt that the early mathematical chapters were a bit lacking. The examples seemed to lack purpose other than being examples. Some of the examples didn't even seem to illustrate the point being made that well. For example, in Chapter 2 'Math Toolkit: Special Functions', there are examples of using `nextafter`, `float_next` and `float_prior`. The results are streamed out to cout at the default precision whereas at least 16 places are needed to be able to see any difference.

Things improved from Chapter 3 onwards, and I felt that from then on there were no more gaps in the coverage, and the chapters on I/O networks and graphs made plenty of effort to cover the background subject matter. I would recommend this book for all but the first two chapters.

Measuring the Software Process – Statistical Process Control for Software Process Improvement



By William A. Florac, Anita D. Carleton, published by Addison Wesley, ISBN: 978-0201604443

Reviewed by Paul Floyd

What a difference there is between this book and the subject of my book review, *Practical Software Measurement*. I suppose that in a way that makes them rather complementary. This book is very much concerned with the practical side of measurement and statistical analysis.

The first two chapters cover background and planning what to measure. The core of the book then covers collecting data, analyzing it and using it for process improvement. The tone and language is similar to that of Deming, talking about the process being under statistical control (or not). If the process is not under statistical control, then it's because there are assignable causes. Once you've identified and eliminated assignable causes, then the remaining variations are inherent in the process, and you can set about reducing that variation.

The final two chapters give advice on how to improve your process based on the measurements and analysis and some practical tips covering getting started and a FAQ. The appendices mostly cover the statistical methods that are required for the analysis described in the book.

I felt that the best part of the book was the advice on how much data to collect and how soon to start using the analyses. The main thing that seemed to be lacking was real-world examples. These methods seem to be commonplace in manufacturing industry but rare in software development. There are some examples in the text, but too short to give any real feeling as to how much benefit can be had from applying measurement in this way.

Practical Software Measurement – Objective Information for Decision Makers

By John McGarry, David Card, Cheryl Jones, Beth Layman, Elizabeth Clark, Joseph Dean and Fred Hall, published by Addison Wesley Professional, ISBN: 978-0-20-171516-3

Reviewed by Paul Floyd

Not recommended.

Seven authors for 268 pages, or about 38 pages each. It does look a bit like a committee. That's not a criticism, just something I find remarkable as I'm used to books being written by one or two people.

I found the first three chapters quite good, covering background, a model for the process of measurement and planning for introducing measurement into a development process. Then I felt that the book lost its way. The next chapters cover the 'do-check-act' parts of the Deming cycle. For example, the 'Perform Measurement' chapter includes explanations of the differences between line charts, bar charts and scatter charts. There are a lot of general guidelines, most of which seemed like obvious common sense.

The appendices (over 100 pages) were better with a larger, more developed example and a couple of real world case studies.

There isn't enough technical content to be of much use to someone that will be a practitioner of measurement. That leaves management as the target audience, more or less as the book subtitle says. I hope that most managers don't need general advice that would be applicable to any major undertaking (stuff like don't alienate your staff and get high management buy-in). I'm afraid this one's heading to the back of my bookshelf.

Release It!

By Michael T Nygard, published by Pragmatic Bookshelf, ISBN:978-0-9787-3921-8

Reviewed by Chris Oldwood

I'm not exactly sure how I came across this book; it had a good reputation on the grapevine and I appeared to be spending more time working on distributed systems, so seemed a suitable choice.

As Michael states in the preface, this is a book for architects, designers and developers working on enterprise-class systems. In essence the focus is on server-based and back-end systems – not desktop apps. The book is split into 4 parts: Stability, Capacity, General Design Issues and Operations with the first two parts taking a pattern language approach, whilst the latter two are more of a collection of loosely related topics.

The first two parts both start with a 10-page case study that sets the scene for the collection of patterns and anti-patterns that follow. Besides providing a back-drop for the pattern language the case study includes an attempt to put a real cost to the business on the failures in question. This provides a sobering thought that only goes to highlight how important the subject matter really is. There are also numerous side-bars throughout the book that take a far more technical look at the issues in question, such as capturing network packets.

Specifying timeouts was already on my list of essentials, but now the terms Circuit Breaker and Bulkhead also seem to be part of my regular vernacular. The key message from the Stability chapters is about de-coupling the various parts system whereas the Capacity section is about being careful with pooling and caching. The anti-patterns in the Capacity part have a definite web bias but I suspect that's just because there are so many more different ways to host your servers.

The grab-bag of topics covered under General Design Issues includes multi-homed servers, clustering and configuration files – all interesting stuff. This leads on to the final part on Operations. Once again Michael takes a few pages to recount a story before heading into discussions around log file formats and various forms of monitoring. Chapter 18 concludes the book by proposing how to adopt a more agile approach to development so that systems will be able to mature and grow in ways that seek to minimise the chances of the anti-patterns rearing their ugly heads.

Software Configuration Management Patterns

By Stephen P Berczuk with Brad Appleton, published by Addison-Wesley, ISBN: 978-0201741179

Reviewed by Chris Oldwood

I've been blessed in that right from the start of my professional career I've always had a Version Control System behind the codebase I'm working on. Naturally in the intervening years I've formed my own opinions about codelines and good SCM policies. So why buy a book? Mainly I was looking for a deeper understanding of the forces that drives the need to isolate and join codelines and also a dictionary to help ensure I understood the correct meanings of the terminology I appeared to be using.

This is a neat little book weighing in around 160 pages after you've ignored the aging appendices, so slips nicely on the shelf. Like many books of its era, it takes a patterns-style approach to describing the various concepts, but only after going through some background notes. The first 3 chapters are all about getting a fluid development process in place and putting some key terms into the right context. I suspect most of this is *de rigueur* these days but the books listed in the further reading sections are all sound recommendations that you probably already own.

The meat of the book is a language of patterns that are both specific to the SCM world, such as 'Mainline', and also tangential like 'Unit Test'. The reason for this is that they are all part of the larger context which is the developer's Workspace – the sandbox that we play in day-to-day. Hence 'Private Workspace' is introduced early on as the pattern language loosely follows the timeline of changes from sandbox, to branch, to build system. 'Integration Build' on the other hand tackles the post-commit side of the process.

There are seven patterns that directly relate to branches of various kinds, each of which has a different policy. Curiously 'Codeline Policy' – the meta-pattern if you like – doesn't get documented until you've already met three of the concrete types. Then there are five patterns covering the build and test side which may seem out-of-place but really it's about promoting practices that ensure commits are solid as you're effectively publishing your changes at that point. This just leaves a couple of other miscellaneous SCM patterns to shore up the good practices, e.g. 'Task Level Commit'.

Sadly the two appendices highlight how far this subject has moved on in the last 10 years because although some of the main URLs in Appendix A are still valid, the list of SCM tools in Appendix B is pretty hopeless. There isn't even mention of Subversion (only CVS) let alone any of the more well-established Distributed VCSs like Git or Mercurial.

I came looking for a book that discussed branches and merging in a bit more detail than the bigger general-purpose development books cover – which I found.

REST in Practice

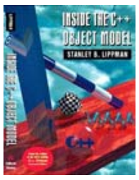
By Jim Weber, Savas Parastatidis, Ian Robinson, published by O'Reilly, ISBN: 978-0-596-80582-1

Reviewed by Paul Grenyer

I read this book because I wanted to learn about REST for a project I'm doing. This book taught me about REST. In fact it did it in the first five chapters. That's less than half the book and it could have been quicker! There is a lot of, to my mind, unnecessary detail that could have been postponed to later chapters or even an appendix. I just wanted to get to the code and to the 'how', I'm not overly interested in the 'why' in this much depth. Furthermore I found the vast majority of the diagrams incomprehensible and totally unhelpful.

However, the book did answer my questions about REST and even using the same REST framework I'd chosen for my project. Unlike a lot of other books, web service clients were not neglected. In fact they were there in as much detail as the server side components with real code! The format of the book is also great for skipping to the good parts and the parts with code.

Inside the C++ Object Model



By Stanley B Lippman, published by Addison-Wesley, ISBN: 978-0201834543

Reviewed by Chris Oldwood

I managed to pick this book up second hand from Jon Jagger's book stall at the ACCU conference this year. It's been on my wish list for a long time as I've always had a fondness for 'under the hood' style books. The big question though is how well has it aged? Surprisingly well it seems...

Just a week or so ago the age old argument about C++ being bloated and inefficient reared its ugly head again. The answer it now seems is to point the protagonists to this book as Lippman cites this common myth in his preface as one reason for writing it. I agree with him that an understanding of the implementation helps with writing more efficient code from the get go as you know the costs of various language features.

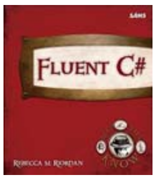
The book is split into 7 chapters and starts with the fundamental Object Model – how an object is represented in memory. He doesn't just cover what a v-table is, but instead walks through in detail how we got from a C style struct to a C++ polymorphic object, including such trade-offs as where to place the v-table. He makes continuous references to the evolution of Cfront which helps you understand the earlier constraints that drove the language.

Once the model is cemented he goes on to talk about constructors (default & copy), data members and then different types of functions (static, non-static & virtual) in the following 3 chapters. He covers the Named Return Value Optimisation in depth and provides some performance figures to back things up. What is particularly noticeable is the number of pages devoted to dealing with Virtual Base Classes – you'll begin to understand why developers often shy away from them.

Chapter 5 looks at the semantics (rather than just the mechanics) of constructors and destructors and their effects on the compiler. Chapter 6 covers new & delete and the tricky issues that initialising arrays throw up. Plus he covers temporary objects in fine detail as that provides a common source of folklore it seems.

The final chapter looks at the then 'new editions' to the language – Templates, Exception Handling and Runtime Type Identification (RTTI). Given its age this chapter is lighter on content, but the mechanics and lineage are still very useful. I also learnt *another* thing about name resolution within templates that really surprised me – this alone made purchasing it worth the price.

Fluent C#



By Rebecca Riordan, published by SAMS, ISBN: 978-0672331046

Reviewed by Adam Petersen

Let me state right up front that I'm not in the intended audience for this book. Fluent C# is a book for beginners to programming. Although I do feel like one from time to time, I've spent the better parts of my adult life hacking together software systems of all scales. Reviewing a book for novices is hard. My view will be quite different. A reader with little programming experience is likely to focus on other aspects and will probably face different problems. So why did I opt to review it? Well, this book claims to be different. It's based on principles of cognitive science and instructional design. Both topics I have an interest in. From that perspective parts of the book provide an interesting read. To a beginner interested in learning C#, the technical inaccuracies are likely to become an obstacle.

Superficially, *Fluent C#* is similar to the well-known Head-First books. Perhaps this is of little surprise since they both draw their educational principles from the same basic research. The learning concept is based on activating multiple senses. The book is littered with graphics, dialogues and attempts to provide the big picture early on. The details are filled-in as we read along. Whereas I'm a big fan of the Head-First concept, *Fluent C#* doesn't work as well. The most striking problem is the layout. All pages are set in a soft, brown tone that becomes strenuous to read. The layout definitely has to be fixed in future editions.

As far as the technical content goes, Jon Skeet has published a +60 pages document on *Fluent C#*. The document contains notes and an unofficial errata for the book. The errors range from ambiguous language, syntax errors, and all the way down to the curious case of code in a different language than the C# promised in the title (in this case some VB code got included in one example). Some amount of errors is inevitable; I mean, even Knuth makes them. But in a book aimed at beginners, the number of errors and omissions has to be considered a serious flaw. If we look beyond the errors to the actual content, we see that the bulk of the book is about writing applications with the Windows Presentation Foundation (WPF). The actual C# introduction is quite rudimentary. A lot of later additions like implicit typing and LINQ are skipped. Instead the author attempts to mix in some high-level design discussions. I find it hard to see how a novice programmer could benefit from these brief discussions on architectural design patterns. A pattern like Model View Presenter (MVP) is non-trivial in its details and requires experience beyond what *Fluent C#* provides.

I took on this book with high expectations. Far too many introductory books take a narrow, technical perspective. Programming is hard enough as it is. We need books like this, designed to actually learn from. I do hope future editions address the present problems. In its current state I cannot recommend the book.

ACCU Information
Membership news and committee reports

View From the Chair
Alan Griffiths
chair@accu.org

One of the great things about the ACCU is that volunteers do everything. People volunteer their time and skills to progress the things they feel the organisation needs. In return for this they get a feeling of accomplishment and recognition from their peers (the other members).

One of the problems with ACCU is that volunteers do everything. People doing things are all volunteers who usually have families, full time jobs and other distractions from getting ACCU business done. That means that things can happen slower than one might hope.

Writing "From the Chair" makes me reflect on what has happened and what needs to happen. I'd love to be reporting something and exciting every time but the reality is that we are making slow, steady progress. It is far from exciting.

Since my last report one of our widely recognised volunteers has decided that his other commitments are such that he has had to reduce his contribution to ACCU. Over the last decade Paul Grenyer has created our "Mentored Developers" group and kept it going. That has now changed and Chris O'Dell has taken over organising the "Mentored Developers" and has replaced Paul on the committee. I'm sure you'll join me in thanking Paul for his contribution over the years and wishing him well with his new activities and in welcoming Chris.

One other service that Paul provided for the ACCU was moderating the accu-contacts mailing list. This isn't an onerous task – there are a few emails each week to classify as "OK", "Spam" or "needs fixing" – but we don't currently have a replacement for this role. Please contact me at chair@accu.org if you are willing and able to take over.

Our "website" working group headed by Dirk Haun has started to understand and document the complexity of the problem they are facing. Like many of the problems we developers face every day it looks simple until you understand it! And like we do every day the team has started identifying things they can fix incrementally. I understand that some volunteers have stepped forward to help since my last report. Hopefully there will be more details to report next time, but until then "thanks" to those getting to grips with the problems.

You'll have seen elsewhere (such as the accu-members mailing list and the website) that Giovanni Asproni has been making sure that our processes are clear and taking steps to bring us into the internet age. Some examples are publishing the officers list and contact details and our complaints procedure on the website. He has also set out how members can participate in committee meetings.

Giovanni has also been working with Mick Brooks on updating the constitution "for the internet age" (as planned at last year's AGM). A draft has now been circulated to the committee, and will probably be made public around the time you read this. This is an important piece of work and will be voted on under the new remote voting rules introduced at last year's AGM. There will be an opportunity to discuss the draft before the motion is formally proposed – watch accu-members for updates.

We've just passed the busiest time of year for membership renewals – and Mick Brooks (the Membership Secretary) reports that membership numbers are holding steady. We can view that positively that ACCU still offers value for money in these difficult economic times.

The ACCU does a good job of keeping the loyalty of existing members so it is doing something right, but it doesn't do so well at attracting new members. There must be loads of developers "out there" who would benefit from ACCU if only they knew what it could do for them. A pity we don't have a volunteer with the skills and motivation to help them.

Secretary's Report
Giovanni Asproni
secretary@accu.org

Making the ACCU more transparent

You may have noticed that the current committee is trying to make the inner workings of the association more transparent to the membership – we started with the creation of the accu-members mailing list, and the publication on it of the minutes of each meeting after they are approved, and of other announcements of importance to the membership.

Then we published the officers page with the names and (most) emails of the committee members, the journal editors and the conference Chair.

We also published the complaints procedure – after some episodes of discontent from members with the way the association was dealing with some issues, we realised that we needed to create a clear complaints procedure.

Finally, we are opening up the participation to the committee meetings. Those meetings have always been open to the membership, but, probably, only a few members knew that.

What follows is what we have already published on the accu-members mailing list and on the members section of the ACCU website. If you have any comments, suggestions, or even criticism, please send them to secretary@accu.org.

Committee meetings: what to do if you want to attend

Committee meetings have always been open to members, but chances are only a few people knew that. That said, there are at least two problems.

First, the available space is usually limited since the meetings are often held at the home of one of the committee members, or in a room lent to us by a company or a charity – e.g., the meeting of July 2012 was held at Bletchley Park in a very small room.

Second, it may be difficult for members to attend in person either because they live far away from where the meeting is being held, or because they are interested only in part of it.

Therefore, in order to alleviate those problems, we are willing to experiment with Google hangouts – and other virtual meeting technologies – to let people attend remotely. We are already using these technologies to allow committee members to take part in the meetings when they cannot participate in person. The procedure will be as follows. The secretary will advertise the meeting time, place and agenda in advance on the accu-members list. If you are interested in attending, either in person or virtually, send an email to secretary@accu.org, and you will be sent the joining instructions.

One thing to keep in mind is that there are some technical limitations – e.g., Google hangouts allow only a maximum of ten people connected, and participants must have a google plus account – and we are still experimenting with the technology, so things may not be very smooth at the beginning and we may be forced to limit the virtual attendance to just a few, on a first come first served basis and subject to availability, with committee members getting precedence. Since the space constraints are going to stay, we will need to limit in-person attendance as well on the same first come first served basis with precedence given to the committee members.

Complaints procedure

The ACCU is an association run by volunteers who do that in their spare time for no compensation. They always try to do their best, but, sometimes, they make mistakes which may end up upsetting some members.

If you want to report an issue to the committee, you can contact any committee member – to find their names, roles and email addresses login with your details and follow the link 'Officers' under the 'Members Section' area on the left side menu – but, the best procedure is the following:

- If you want to complain about matters regarding one of the ACCU magazines, the best thing to do is to email the editor first. Their mail addresses are cvu@accu.org for CVu and overload@accu.org for Overload. If you are not satisfied with the answer you can escalate to the publications officer at publications@accu.org

- If you want to complain about something regarding the conference, you should contact the Conference Chair first at conference@accu.org
- For anything else, the best thing to do is to contact the secretary at secretary@accu.org or the ACCU chair at chair@accu.org

If you feel you need to escalate your issue further, please contact the ACCU Chair at chair@accu.org, and, if you are still not satisfied, the constitution allows you to call a general meeting. Please have a look at the constitution for the details.

If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

ACCU officers

The names, positions and (where available) the email addresses for the ACCU officers are given in the table below:

Name	Email	Role
Committee: Executive members		
Alan Griffiths	chair@accu.org	Chair
Robert Pauer	treasurer@accu.org	Treasurer
Giovanni Asproni	secretary@accu.org	Secretary
Mick Brooks	accumembership@accu.org	Membership Secretary
Seb Rose	ads@accu.org	Advertising
Astrid Byro	publicity@accu.org	Publicity
Roger Orr	publications@accu.org	Publications Officer
Committee: Non-executive Members		
Ali Cehreli		US Agent
Andrew Marlow	andrew@andrewpetermarlow.co.uk	
Chris O'Dell	christine.ann.odell@gmail.com	Mentored Developers
Dirk Haun	dirk@haun-online.de	Website
Jon Jagger	conference@accu.org	ACCU Conference Chair
Mark Radford	standards@accu.org	Standards
Matthew Jones		Local Groups
Paul Grenyer		
Silas Brown		Disabilities
Stewart Brodie		
Tom Hughes	tom@compton.nu	
Journals		
Steve Love	cvu@accu.org	CVu Editor
Frances Buontempo	overload@accu.org	Overload Editor