The magazine of the ACCU

www.accu.org

{cvu}

Volume 19 Issue 3 June 2007 £3

Features

Regulars

{cvu}
Volume 19 Issue 3
June 2007
ISSN 1354-3164
www.accu.org

accu

When should code be allowed to die?

As many of you know I spent quite a few years working in London. For most of that time I was working for investment banks. One of the reasons that I wanted to leave was that I kept coming across the same problems. Not exactly the same problems, but similar enough that the work was starting to become repetitive.

Personally I have a really low bordem threshold. It is one of the reasons that I'm not into computer games and don't have any games consoles when I have many friends that do, and they enjoy them. I just get bored with them. I get bored at work too. This is why I went contracting – to get a change every now and then.

The problem that kept popping up was not wanting to allow code to die. Instead companies wanted to keep it on a respirator, feed it intravenously, and hope it keeps living. Sometimes you just need to start afresh, pull the plug and let the old code go. However this is never a popular opinion in corporate circles. There are many risks when starting again, not to mention the possibility of falling behind a competitor. Companies don't like moving developers from bug fixes and incremental improvements to a new green fields project that will probably fail.

So, what can we do? When is a good time? What is the best way to mitigate the risks? Personally I believe that some systems just require some love, and radical refactoring, to breathe new life into them. Others could benefit from side by side development, creating the new system while allowing the old to limp along. Each is different.

I left the UK because I kept seeing systems that should have been laid to rest long ago, and companies and managers that couldn't see it. Now I'm sure NZ has similar systems, but thankfully I'm not working on them.

TIM PENHEY,
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 ACCU members – 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

COPY DATES

C Vu 19.4: 1st July 2007
C Vu 19.5: 1st September 2007

IN OVERLOAD

Begin your reading with 'The Policy Bridge Design Pattern' and 'Live and Learn with Retrospectives'.

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

Scripting C++ Objects

Our new gang of four makes C++ more dynamic.

No application is an island. And no team of developers can ever hope to provide all the functionality their users require. Although it is relatively easy to exchange small chunks of data between applications using the clipboard, this doesn't go far enough.

The solution to these problems is to provide an API for your application so that users can write their own solutions to particular problems. But what language, or languages, should that API use? In many cases, for CVu readers especially, the application will be written in C++. To expect users to be conversant in C++ to develop extensions is clearly not going to win any friends. This is typically an area where scripting languages such as Perl and Python are widely used. These languages are relatively easy to get started with, and many computer users are familiar with their use.

The question then becomes how to link the scripting language to application code. Fortunately this is a fairly well trodden path and there are a number of solutions already available to us:

- SWIG
- Attributed C++
- IDL
- The Preprocessor

SWIG

The Simplified Wrapper and Interface Generator [1] is a free software tool that makes it possible to build a scripting API onto C++ programs. SWIG is a compiler, of sorts, that takes C++ declarations, in the form of an 'interface file' and turns them into C code that acts as the binding between the scripting language and the application code. SWIG supports many scripting languages.

Attributed C++

The aim of Attributed C++, as introduced into Microsoft's C++ v.7.0 (Visual Studio .NET) was to allow functions and classes to be attributed with metadata within the program itself, rather than requiring a separate pre- or post-processing stage. These attributes produced hidden code which, for example, wrapped datatypes and provided interface shims.

IDL

An Interface Description Language is used to describe an interface in a language-neutral way. The most common IDLs are those used for Microsoft's COM and in CORBA.

The C++ Preprocessor

The typical C and C++ program has function declarations in header files and definitions in separate files. When brought together by the preprocessor, these usually form a 'translation unit' as defined by the Standard, i.e. a complete component that can be parsed by the compiler with no reference to external information.

Other factors affecting the choice of approach

At Accelrys, our Materials Studio™[3] application allows the user to run remote jobs both on Linux™ and Microsoft Windows™ servers. As calculations initiated by a script often take a considerable time to complete, the scripts should be runnable under both these operating systems. We needed a portable interface between our C++ code and the script.

Scripting was to be added to portions of a large existing application by a small team. The approach chosen could not require large-scale refactoring, as the test burden alone would be prohibitive. We planned that future releases would involve a wider pool of developers, not all of whom would have knowledge of the implementation mechanism and so we needed to allow as natural a C++ style as possible.

Comparison of the above

Looking at the above options, we decided to discount Attributed C++. Though it reduces the amount of IDL and associated overhead, it is not portable and its buggy implementation has in the past ruled out its use in a production system. Also, the hidden code produced by the attributes is not easily discoverable and the attribution cannot be extended arbitrarily to user-defined attributes.

With the C++ preprocessor, the split between the declaration and definition files leads to much redundancy, exacerbated when small delegation functions are used. To add an extra layer of duplication by way of pre- or post-processing compounds this. We have also found from experience that the separate pre- and post-processing tools tend to have special rules that disallow writing in a natural C/C++ style.

Looking at the two remaining options the choice is not clear cut; both of them having their strengths and weaknesses. SWIG is the front runner. Its main benefit is its support for multiple target languages. But the extra mapping files that must be written increase the development and maintenance overhead. The danger always is that the application code moves on and the interface description is not kept synchronised. And the larger the API, the bigger this issue becomes – this solution does not scale.

KEVIN DIXON

Kevin has been developing scientific applications in C and C++ since 1991, filling in the gaps with Perl. He is currently modelling polymers and other large molecular systems. He can be contacted at kdixon@accelrys.com

JOHN LEAR

John has been developing software in C++ on various platforms for the last 14 years. At Accelrys, he works on client development and infrastructure, hopefully making life easier for other developers. He can be contacted at jlear@accelrys.com

SIMON GRAY

Simon is an escaped experimental physicist who was spotted around financial programming for a few years before finding a more natural home writing scientific software. He can be contacted at Simon.Gray@Physics.org

SIMON TREW

Simon started his career as a software engineer in the defence industry. After graduating from UMIST, he produced object-oriented databases, before joining Accelrys. He can be contacted at strew@accelrys.com

The synchronisation issue applies to IDL as well. Though IDL does have its advantages, enforcing a good separation of interface from implementation, there are more downsides: It is not easily portable to other platforms and it forces a very unnatural programming style on the C++ implementation. For instance:

1. Values cannot be returned but are specified as reference parameters in a method's argument list.
2. The datatypes available are limited to PODs and cannot be easily shimmed into wrapping RAII types without costly copying.
3. Interfaces can only use raw IDL types rather than safer wrapped types.
4. Implementations cannot throw exceptions for errors. This leads to excessive flow-of-control code cluttering the implementation.
5. Interfaces cannot use qualifiers such as **`const`**.

Ideally, a solution would allow the API methods and properties to be declared within the source code of the API itself. In fact, tying the two together creates a tight development feedback loop where the API declaration cannot help but match its implementation, otherwise it won't even compile.

Rationale

The ability to write scripts to perform calculations within our application had been a long-standing user request. Since the application is written as a collection of COM objects (using the XPCOM [2] libraries for portability to Linux), these initial requirements could have been met by providing Perl modules to bind to our internal COM objects. There was concern that this would prevent us from changing our internal implementation in future and make supporting other scripting languages no easier, so we needed a more flexible approach.

Our application is already built as a series of COM components which implement a datamodel and calculation engines. To give users direct access to these would constrain future development, so our scripting framework calls through an abstraction layer. This allows us to change the underlying implementation and datamodel more freely in future as, if necessary, adaptation can be handled by this layer. Consistency of interfaces and hiding of implementation methods are also enforced through the abstraction layer.

The datamodel exposes many of its properties as key/value pairs. Rather than having to know which properties are exposed like this, and passing the key to a single get or set method, the abstraction layer coupled with the forwarding mechanism described later lets us expose these as object properties, with a **`get`** or **`put`** method for each one.

The COMLite approach

None of the previously described solutions gave the right combination of flexibility with low maintenance costs and ease of use, so we decided to pick the best techniques from what was on offer to craft our own solution.

Our novel implementation comprises three main parts:

1. A set of macros and templates that allow metadata to be easily injected into a portable C++ program using a natural style that is fully checked at compile-time.
2. A binding from that metadata to a language-independent, platform-neutral calling convention (in our case, COM).
3. A binding from that calling convention to a particular scripting language (in our case, Perl).

This approach (illustrated in Figure 1) allows APIs to be developed that are easy to use and maintain in isolation from the target language binding while still being platform neutral and useable from other C++ code in a natural way.

The choice of calling convention is fairly arbitrary. We chose COM as it is already used extensively throughout the Materials Studio application. It could easily be replaced, with SOAP say, if desired. The choice of Perl was arrived at via marketing surveys and discussions with our user base.

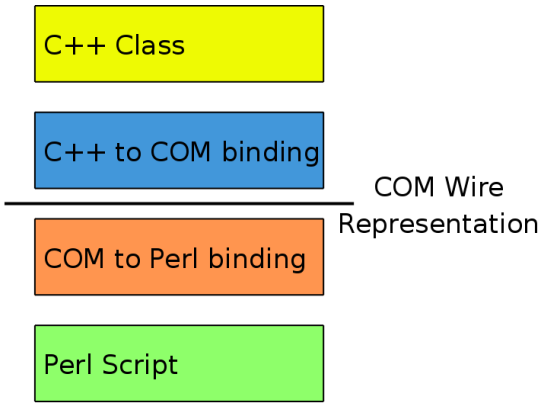


Figure 1

COM/IDispatch recap

COM is clearly a Microsoft technology. However, XPCOM, which is a Mozilla project, gives the cross-platform capability that we require. The COM facilities used are limited to interfaces and component factories.

IDispatch is one of the two core interfaces that form COM, IUnknown being the other. IUnknown is fine for statically-bound languages such as C++ that go through a compilation phase. Scripting languages are generally not compiled and statements are only interpreted immediately prior to their execution. IDispatch provides an interface where clients can dynamically discover the abilities and properties of objects at runtime. This gives us everything we need to create any new APIs for scripting support.

The IDispatch interface can be split into two parts. The first is related to processing COM type libraries which will not be covered here. The second section, and that used by COMLite, is all about discovering the methods and properties available on an object at runtime and causing them to be executed. This process takes a two stage approach:

1. Do you know about this method/property? If yes, give me a token by which to identify it.
2. Call the method/property identified by this token, possibly with a bunch of arguments.

Step 1 is achieved through a call to **`GetIDsOfNames`** and step 2 is achieved by calling **`Invoke`**, both on the IDispatch interface. Paraphrasing their real signatures, this has the form shown in Listing 1.

For every known name mentioned in the call to **`GetIDsOfNames`** a token known as a **`DISPID`** is returned. This token is used by subsequent calls to **`Invoke`** to identify the function or property accessor to be called. The **`names`** array contains the function name plus any named arguments used. (Named arguments are beyond the scope of this article and it will be assumed that **`cNames`** is always 1.) Flags are used to identify the context

Listing 1

```
typedef unsigned int DISPID;

DISPID[] GetIDsOfNames(const char       *names[],
                       const unsigned int cNames);
              //number of values in names[] array.

VARIANT Invoke(const DISPID         dispIdMember,
               const unsigned int   wFlags,
               // call reason, eg. property get
               // or property set.
               const DISPPARAMS     *pDispParams,
               // array of arguments.
               EXCEPINFO            *pExcepInfo,
               // rich error information passed
               // out on error.
               unsigned int         *puArgErr);
               // index of the parameter
               // that caused the error.
```

of the call: a function call, a property setter or a property getter. The **DISPPARAMS** structure contains the arguments to the function in the form of an array of **VARIANT**s. The final two parameters are used to pass out a rich error structure to indicate that an error occurred during the handling of the call. If no error occurred, the result of the call (if any) is returned in the **VARIANT**.

Through sequential calls to **GetIDsOfNames** and **Invoke**, client code can call methods and get/set properties on the serving object. The client is also allowed to cache the results of calls to **GetIDsOfNames**; the interface description requires that once a token has been handed out, it remains valid until the serving object dies. However, scripting clients are often ill-advised to cache the tokens as it can be hard to track whether they truly refer to the correct serving object.

The dynamic discovery of functions and properties means that an object can invent new behaviour during its lifetime. For example, it might delegate some calls to other objects, completely unknown to the client. This semblance of dynamic classification can become quite powerful when calls are delegated to other objects which also implement **IDispatch**. It is novel for C++ and most other OO languages, where an object's shape (i.e. its class) is fixed at construction time, which generally means fixed at compile time and at design time too. We discuss this in more detail later.

C++ to COM binding

One of the annoyances with writing COM classes is the need to specify the same information in several places – notably, in the IDL, the header file containing the class declaration, and the `.cpp` file containing the class implementation. (Admittedly we could simplify this by writing all definitions inline.) Although the IDL is useful for automatically generating a type library, it can become cumbersome to keep it all up to date. It also means that every interface (dual or IDispatch based) has to be named.

Scripting languages are typically weakly typed and gain no benefit from having these separate interface descriptions since they talk exclusively in terms of IDispatch and ITypeInfo. In fact, all interfaces can just be IDispatch based (often called *dispinterfaces*).

To provide a COM mapping for a C++ class we use a set of macros, templates and other classes that allow the same information as would be found in IDL to be defined in a convenient way purely in C++. This information is provided at the point of definition for each function, thus keeping all the information together.

This information is then made available to the implementation of **GetIDsOfNames** and **Invoke** to provide the type information on the fly at runtime and also implement the marshalling of the input parameters.

So in practice, how does this look? See Listing 2.

The **IDispinterfaceImpl** class implements all the methods of IDispatch and delegates calls to **Invoke** to the implementation in the derived class. The curiously recurring template pattern is used to access information contained in the derived class. **IDispinterfaceImpl** is declared as follows:

Listing 2

```
class Atom : public IDispinterfaceImpl<Atom>
{
public:
  DECLARE_PROPGET(Name, std::string) const {...}
  DECLARE_PROPPUT(Name, std::string) { ... }
  DECLARE_METHOD0(void, Delete) { ... }
  ...
  BEGIN_CLASSINFO(Atom)
    CALLINFO_PROPGET(Name)
    CALLINFO_PROPPUT(Name)
    CALLINFO_METHOD(Delete)
    ...
  END_CLASSINFO
};
```

```
template<typename impl>
   struct IDispinterfaceImpl : IDispatch
  {
    typedef impl impl_class;
    //Implementation of IUnknown
    ...

    //Implementation of IDispatch
    ...
  };
```

The basic implementation of IUnknown will not be covered here. The IDispatch implementation uses the function and property accessor information declared within the class. These declarations form two distinct sections. The first section declares the property accessors and functions. The second section injects the type information into the base class. (Note: This example gives only a static type example. Dynamic classification will be covered later.)

First of all, there is no real difference between the **PROPGET**, **PROPPUT** and **METHODx** variants of the **DECLARE_XXX** macro. Its use aids readability, it allows the injection of the **Get** and **Set** 'warts' for the C++ function name and finally it allows us to specify the **wFlags** that need to be matched against those supplied by the client code. These macros simply defer to the **DECLARE_FUNCTION0** and **DECLARE_FUNCTION1** implementations:

The dynamic discovery of functions and properties means that an object can invent new behaviour during its lifetime.

```
#define DECLARE_PROPGET(com_name, prop_type) \
   DECLARE_FUNCTION0(com_name, prop_type, \
   Get##com_name, property_get)

#define DECLARE_PROPPUT(com_name, prop_type) \
  DECLARE_FUNCTION1(com_name, void, \
  Set##com_name, prop_type, value, property_put)

#define DECLARE_METHOD0(ret_type, com_name) \
   DECLARE_FUNCTION0(com_name, ret_type, \
   com_name, method)
```

The **DECLARE_FUNCTIONx** has number of tasks to perform:

1. Create a structure to hold the static type information and type marshallers.
2. Populate that structure with the type information.
3. Create a normal C++ function declaration so that the actual implementation code may follow.

The **DECLARE_FUNCTIONx** macro is written as shown in Listing 3.

This code fragment has introduced a number of supporting types that would not normally be visible to the API writer. The most important of these is the **call_info** class. This template class acts as a repository for the information pertaining to a single function or property accessor. It also contains a C++ member function pointer to the function that will be used when satisfying the call, as shown in Listing 4.

The **mem_fun_ptr** class is simply a template class to provide some convenient type definitions:

```
template<class impl_class>
```

Listing 3

```
#define DECLARE_FUNCTION0(                              \
   com_name, ret_type, cpp_name, kind)                  \
   static mem_fun_ptr<impl_class>::                     \
   dispatch_fn_ptr                                      \
   get_dispatcher_##cpp_name##()                        \
  {                                                     \
     return &dispatcher<impl_class, ret_type,           \
        0>::dispatch<void>;                             \
  }                                                     \
                                                        \
  static const call_info<impl_class>                    \
     *get_info_##kind##_##cpp_name##()                  \
  {                                                     \
    static const variant::type return_type =            \
       static_cast<variant::type>(                      \
       variant_wrapper_traits<ret_type>                 \
       ::variant_type);                                 \
                                                        \
    static const param_infos params                     \
       = param_infos();                                 \
                                                        \
    static call_info<impl_class>::                      \
       mem_funcptr_type const fn =                      \
       reinterpret_cast<call_info<impl_class>::         \
       mem_funcptr_type>(                               \
       &impl_class::cpp_name);                          \
     static call_info<impl_class> const info =          \
       call_info<impl_class>(kind, return_type,         \
       cpp_name, params, fn,                            \
       get_dispatcher_##cpp_name##());                  \
    return &info;                                       \
  }                                                     \
                                                        \
  ret_type cpp_name()
```

Listing 4

```
template<class impl_class>
struct call_info
{
  typedef typename mem_fun_ptr<impl_class>::
     value_type mem_funcptr_type;
  typedef typename mem_fun_ptr<impl_class>::
     dispatch_fn_ptr dispatch_fn_ptr;
  call_info(kind reason,
     variant::type return_type,
     const std::string& name,
     const param_infos &params,
     mem_funcptr_type funcptr,
     dispatch_fn_ptr dispatcher) :
    m_kind(reason), m_return_type(return_type),
    m_params(params), m_funcptr(funcptr),
    m_dispatcher(dispatcher)
  {}
  ...
};
```

```
struct mem_fun_ptr
{
  typedef void (impl_class::*value_type)();
  typedef VARIANT (*dispatch_fn_ptr)(
     impl_class*, value_type,
     param_infos::positional_params_type &,
     const call_info<impl_class> &);
};
```

The **param_infos** class contains information about the parameter types used a function call. In this example, as the function takes no parameters, it is empty. The **DECLARE_FUNCTION[1-x]** variants of this macro simply populate this structure with information pertaining to the type and name of the parameter.

Listing 5

```
#define BEGIN_CLASSINFO()                               \
  static const call_info<impl_class>*                   \
     get_call_info(const std::string& name,             \
     kind reason = unknown)                             \
  {                                                     \
    static const class_info<impl_class> info =          \
     class_info<impl_class>()
#define CALLINFO_METHOD(name)                           \
           << get_info_method_##name##()
#define CALLINFO_PROPGET(name)                          \
           << get_info_property_get_##name()
#define CALLINFO_PROPPUT(name)                          \
           << get_info_property_put_##name()
#define END_CLASSINFO()                                 \
        ; return info.find(reason, name);               \
  }
```

The final supporting class used here is the **Dispatcher**. The **Dispatcher** is simply a function object whose sole role is to make the function call.

This information is then injected into the class's type information in the **CLASSINFO** section. This information is again constructed statically, using the type information provided by the methods defined by the **DECLARE_XXX** macros (Listing 5).

Listing 6 shows the final level of aggregation, **class_info**, that contains information for all property accessors and functions defined in a class, the stream insertion operator being used to allow the easy addition of the information for individual property accessors and functions.

Using the constructs described in this section we are able to build up extremely rich metadata describing the class and its methods. This information is then used at runtime to satisfy requests through calls to **GetIDsOfNames** and subsequent **Invokes**.

Listing 6

```
template<class impl_class>
struct class_info
{
  typedef std::map<std::pair<kind, std::string>,
     const call_info<class impl_class> *>
     call_infos_type;
  class_info& operator<<(
     const call_info<class impl_class>*
      call_info)
    {
      m_call_infos.insert(
         call_infos_type::value_type(
         call_infos_type::key_type(
         call_info->m_kind, call_info->m_name),
         call_info));
      return *this;
    }
  const call_info<class impl_class>* find(
     kind reason, const std::string &name) const
  {
    call_infos_type::const_iterator found;
    if (reason == unknown)
      found = find_if(m_call_infos.begin(),
        m_call_infos.end(), a_predicate(name));
    else
      found = m_call_infos.find(
        call_infos_type::key_type(reason, name));
          return (found == m_call_infos.end())
      ? 0 : found->second;
  }
  call_infos_type m_call_infos;
};
```

Listing 7

```
template<typename impl>
   struct IDispinterfaceImpl : IDispatch
{
  ...
  //IDispatch support
  typedef std::map<std::string,
     DISPID> dispid_map;
  static dispid_map m_dispids;
  static DISPID m_nextDispId = 0;

  //Implementation of IDispatch
  DISPID[] GetIDsOfNames(const char *names[],
     const unsigned int cNames)
  {
    DISPID retval[] = { DISPID_UNKNOWN };
    // first check if this name has been
    // seen before

    if(m_dispids.find(names[0]) !=
       _dispids.end() )
    {
      retval[0] = m_dispids[names[0]];
    }
    // otherwise see if the class_info
    // knows anything
    else if (get_call_info(names[0]))
    {
      dispid_map.insert(std::make_pair(names[0],
         ++m_nextDispId));
      retval[0] = m_nextDispId;
    }
    return retval;
  }
};
```

A call to **GetIDsOfNames** must return a unique identifier, or **DISPID**, for a given name. Should that same name be asked for in the future, the same **DISPID** must be returned. Using the information we have in the **class_info** structure, it is easy to satisfy these requests. A unique **DISPID** can be assigned by simply incrementing a count each time a new name is encountered and the relationship to the name stored in a map (see Listing 7).

Listing 8

```
template<typename impl_class>
   struct IDispinterfaceImpl : IDispatch
{
  ...
  VARIANT Invoke(const DISPID dispIdMember,
     const unsigned int  wFlags,
     const DISPPARAMS   *pDispParams,
     EXCEPINFO          *pExcepInfo,
     unsigned int       *puArgErr)
  {
    const call_info<impl_class>* const info =
       static_cast< const call_info
       <impl_class>*>(impl_class::get_call_info(
       dispIdMember, wFlags));

    if (info)
    {
      try
      {
        return info->Invoke(
           static_cast<mpl_class *>(this),
           pDispParams);
      }
      catch(...)
      {
        // Setup  call-specific error information
      }
    }
    //Setup generic error information
  }
};
```

Listing 9

```
template<class impl_class>
struct call_info
{
  ...
  VARIANT Invoke(impl_class* instance,
     const DISPPARAMS *pDispParams)
  {
    //Unpack the DISPPARAMS
    std::vector<VARIANT> actual_params(
       make_actual_params(pDispParams));
    //Make the call
    return m_dispatcher(
       instance, actual_params, *this);
  }
};
```

The final step is the implementation of **Invoke**. **Invoke** is called with the previously handed out **DISPID** and any arguments supplied by the caller (see Listing 8).

Most of the work is delegated to the **call_info** class which was setup by the **DECLARE_XXX** macro in the implementation class (see Listing 9). This has a number of tasks to perform:

1. Marshall the input parameters, supplied as **Variants** in the **DISPARAMS** structure, into the native C++ types specified.
2. Call the function.

The call to **make_actual_params** unpacks the **DISPPARAMS** structure into a simple vector of **VARIANT**s. The dispatcher is then responsible for converting those types into their natural C++ types. The dispatcher for a function call is templated on the actual C++ arguments for that function call and this how the **VARIANT**s get unpacked. A **Dispatcher** class has the following form:

```
template<class impl_class, typename ret_type,
 unsigned params> struct dispatcher;
```

Using template specialisation an actual **Dispatcher** for a single argument function call would look like Listing 10.

Using a simple table of traits classes allows conversions to be specified independently and extended as required. Here as an example traits class for unpacking type **int**s:

```
template<> struct variant_wrapper_traits<int>
{
  typedef const int wrapper_type;
  enum { var_type = VT_I4 };
  static int get_value(
     const VARIANT &v) { return v.intVal; }
};
```

Observations on the implementation

The example here covers a very simple subset of what is possible. The number and type of parameters supported can easily be extended by adding to the macro definitions and type traits table. In over a year's worth of use we have only come up with one situation that required the support of more than three function parameters. The macros can also be extended to support a number of extra facilities.

- Different variants of C++ functions can be defined dependent upon whether the parameter has an in (const ref) or out (pointer) attribute.

Proactive Laziness

Simon Sebright explains why it's good to be lazy.

Introduction

This article describes a pattern I have recognized in real life as well as development process and implementation.

Proactive Laziness refers to doing something up front to avoid problems later in time. The key thing is that the up-front activity is something different to the downstream activity, it instead facilitates it. As such, it is different to the pattern which might be called 'Keeping on top of things'.

For example, we all have to pay bills. Paying a bill early is the same activity as paying it too late, so does not count (even though paying late might have penalties attached) as Proactive Laziness. Rather, it is simply keeping your affairs in order. However, setting up a direct debit is Proactive Laziness, because it is a different activity. In this case, the up-front activity automatically ensures that the desired result occurs.

Of course, without the Proactive Laziness device, in the real world, we can have benefits simply by keeping on top of things – making payments, opening post, buying enough milk for your tea. In the development world, this keeping on top of things can become difficult, if not impossible, and has a tendency to distort. For example, not using RAII classes means tracking all exit points of a function, notwithstanding the fact that exceptions can occur in some languages.

Often, the upfront activity requires more effort than the initial problem it is avoiding, but has the ability to solve the problem multiple times, thus as time goes by, the payback period is exceeded and we start to gain.

I want you to go away from this article thinking what you can do better in your work, where you can save time and money by putting in place practices which help you help yourself. Things which mean you can leave for home in the evening with a feeling of confidence, and drink your beer in peace.

Be Proactively Lazy!

Domains

This is an article for technical people, so most of the domains are to do with software development. I also look first at a few real-world examples to show that this pattern is not purely about writing code. I then move on to look at development process, development itself and lastly user interfaces.

SIMON SEBRIGHT

Simon has been programming for 10 years, mainly in multi-tier C++ application development. Recently, he has been designing and developing web- and database-based application using C# and asp.net. He can be contacted at simonsebright@hotmail.com

Scripting C++ Objects (continued)

- Including help text for use in type library generation.
- Supporting virtual functions in C++.

There are also a significant number of plus points in using this approach. As can be seen, it is very straightforward to create a class with a dispinterface. Also, the **DECLARE_XXX** macros declare a normal C++ function that can be called directly from C++:

Listing 10

```
template<class impl_class, typename ret_type>
struct dispatcher<sink, ret_type, 1>
{
  template<typename p1_type> VARIANT dispatch(
            impl_class* sink,
            const std::vector<VARIANT> &params,
            const call_info<impl_class> &info)
  {
    typename variant_wrapper_traits<p1_type>::
       wrapper_type wrapper_p1(
    variant_wrapper_traits<p1_type>::get_value(
       params[0]));
    typedef ret_type (impl_class::*func_ptr)
      (variant_wrapper_traits<p1_type>::
       wrapper_type);
    return sink->*(

reinterpret_cast<func_ptr>(fn))(wrapper_p1);
  }
  ...
}
```

```
std::string atomName(atom->GetName());
```

Code can also be separated into public, private and protected sections for C++ clients.

Coming in Part 2...

The first part of this article has shown how to provide a platform-independent and language-neutral calling convention based on metadata contained in C++ objects. In the second part, we will show how to use these features and provide a target language binding for Perl. We will also demonstrate a novel technique for reducing verbosity in the target language which also facilitates the API's implementation through C++ mixin techniques. ■

Acknowledgements

The authors would like to thank their colleagues at Accelrys for their valuable editorial contributions to this article.

References

[1] SWIG – http://www.swig.org
[2] XPCOM – http://mozilla.org/projects/xpcom
[3] Materials Studio –5.5 http://www.accelrys.com/products/mstudio

Mechanisms

For each example, I have identified the main mechanism underlying its ability to be proactive. It turns out that there are common mechanisms in the domains identified:

Availability Having information available when needed

Automation When something occurs automatically – we set something up, and it happens how and when it should.

Deterrent The next best thing to Enforcement, where we *try* to stop bad things happening

Enforcement Stopping the things we don't want to happen from happening, usually by design

Flexibility Offering components which when combined together create more numerous, powerful and elegant solutions than otherwise possible when functionality is locked up in one entity

Real world examples

Here are some things you may or may not do in day-to-day life which fall under the practice of proactive laziness.

Direct debit payments

As mentioned in the introduction, taking the time to set up an automatic payment mechanism means you will never forget to pay your bills on time and thereby avoid incurring penalties.

Mechanism Automation – automation of the desired payment activity. You rely on the systems of your suppliers and financial institutions to make the correct payments at the correct time.

Risk You have to trust the suppliers and financial bodies controlling the transactions. Personally, I am quite happy with the service I have had over the many years I've been using it.

Store phone number in address book

We often need to ring new people. If you have the number either on screen, a business card, or just a piece of paper, you need to type it into your phone. Taking the time to add it to the address book before you call the person means that you won't have to remember it any more, or take the time to retype it, or risk losing it. (Often a mobile phone will keep a record of calls made, so you might be able to retrieve it later, but they usually drop off the list after time).

Mechanism Availability – we always know we can retrieve the phone number whilst using the device in question.

Risk Of course, you could forget the phone with the number stored on it, change the chip, it might break, etc. The number may also change; only a central updated repository can deal with that.

Shopping list

If you go shopping for something, particularly household shopping where many items need to be purchased at once, then make a shopping list. That way, you don't have to remember everything and run the risk of forgetting things you need, or wasting money on things you don't.

It's best to make the list as the time goes by from the last shopping trip, as it becomes apparent that things will be needed. You might have another fixed list for things you always need such as milk, eggs, bread, etc. to avoid clutter.

Mechanism Availability – the list of things needed is collated in one place. An alternative is to go round the shop looking at everything deciding whether or not you need it. The latter works better for people living alone!

Risk List could be lost, difficult to read or ambiguous. Also, if this list is rigidly applied without intelligence, we don't cover the cases of lack of availability, special offers, spontaneous decisions on meal plans, etc.

Application in software This can be a way to introduce performance increases in applications. A computer does not get bored if you tell it to visit every item and see if it's needed, but as the number of items increases, so does the time to find out. If a process keeps a separate list of things to process, then if that list is a small fraction of the total available, we can potentially save a lot of time.

Development process examples

These are things we do in the process of producing software. It might be part of a methodology or an actual activity. The general aim is to make sure that the process is robust and reproducible.

Turn up compiler warnings

The compiler is your friend. It will tell you when you are doing things which might cause errors in your program. Turn up the warning level to maximum. Ideally, do this at the start of a project and compile with no warnings. I recommend even in the middle of a project that this is a good practice and have often taken the time to eliminate warnings. Comment out unused parameters and use appropriate casts where truncation may be occurring (or change the datatypes).

One problem here is library headers. Warnings coming from them are not in your control. Some environments allow you to introduce `#pragma` statements to suppress these. Do not suppress warnings globally in your own code though (apart from rare annoying ones that can never be relevant for you, for example the Microsoft performance warning on converting `BOOL` to `bool`).

That time when you don't refer to a caught exception local variable might be a bug – as you won't be referencing the nature of the error. That time you don't reference a parameter might mean you incorrectly implement a function.

Mechanism Enforcement – you use a tool which tries to limit your misuse (intentional or otherwise) of the language.

Risk You bypass it by ignoring warnings on too large a scale with `#pragmas`.

You become blasé about putting in C-style casts whenever a narrowing conversion or similar is diagnosed.

Treat warnings as errors

Stop yourself cheating, or new colleagues unknowingly breaking your clean build. Tell the compiler to treat warnings as errors, and you simply won't be able to build a non-clean code base. Again, it is best to do this up front and have a clear strategy for library files.

Mechanism Enforcement – like turning the warning level up, only this time, you are not allowed to proceed without warnings being addressed.

TDD (Test-Driven Development)

Take the time to introduce tests which your code passes as you develop. In terms of proactive laziness, this means you have a much better time when you make changes to the code, including refactoring. You don't have to manually run through tests, or analyse in so much detail to be certain that you haven't broken anything.

In addition, you have provided some level of documentation of your code, and may find that by writing it from a client perspective, that the interfaces are cleaner and easier to use than by starting to write functionality in the `cpp` files.

Mechanism Enforcement – the code has to pass the tests to proceed through the process of check-in and build. Ideally, the build process should automatically run the tests and fail if any test fails.

Automatic builds

By writing a script to get from source control and build your releases, you eliminate the risk of forgetting to check in files, or having local copies of extra files which affect the result when you test your own code on your machine.

Only ever release (even to internal recipients) products which have been compiled from a labelled version of the source code. This way, you can always be sure what they got, and can repeat it in the future.

In addition, incorporating automatic tests as part of the build detects problems early, particularly it might pick up on issues caused by code integration, if developers have not been building their changes against the latest codebase.

Mechanism Enforcement – you cannot deliver a version of the product if the files have not been checked in properly, or if the code has build errors

Risk In some cases, missing code may not cause a failed build, particularly where classes perform some kind of self-registration. In this case, automated tests will help, because they should cover all the cases your product is designed to handle.

Development implementation examples

Finally, when writing code, whether the design of core interfaces in a large system, or the minutiae of a particular function, do what you can to prevent things from going wrong, being misused, etc.

Encapsulation

This is one of the classic tenements of OO (Object Orientation). Objects are instances of Classes, and classes can have both data members and function members (in some languages Properties and others too). This means we can bundle pieces of data with the functionality required to manipulate it, and stop other functionality misusing it. We only expose the functionality we want the client to use, thus abstracting the implementation details away.

Taking the time to encapsulate data and functionality means that you know who can and can't use it. When members become non-public, you can write your code with certainty that abuse is difficult. At least your intention has been clearly signalled, if someone does nasty things to get at your data, it's their own fault if things go wrong.

Avoid `set` functions 'just for completeness', have the interface of a class truly reflect what you want it to do, not what it could possibly do given the data members you have. See the 'Easy to use, hard to break' section below.

Mechanism Enforcement – you cannot access member data or functions which are marked as private, and only derived classes will be able to access protected members.

Risk It may be the case that some functionality hidden from clients may at some future point really be needed. Particularly where frameworks are built, and therefore the end functionality is not known, it can be tempting to expose more than is strictly necessary. For example, the framework MFC is notorious for having protected and even public data members. The wider the applicability of a framework, the more this is likely to happen.

Of course, if you are writing your framework to use yourself, you can keep things as tight as possible until particular requirements arise which change things.

RAII classes

RAII (Resource Acquisition Is Initialisation) refers to using the scope/lifetime of objects to automatically control access to a resource. In this case, the 'resource' is anything which might need special handling, usually with respect to clearing up when we have finished with it, for example, freeing up memory, closing a file, releasing a mutex, etc.

In a language with deterministic destruction, like C++, one can wrap pairs of function calls in a class, one occurring in the constructor, the other in the destructor. For example open/close, lock/unlock, new/delete. These symmetrical pairs then automatically get invoked when an object is constructed in a scope, and when it is destructed at the end of scope

In an exception-enabled program, they are in fact required for correct function, because you can't be sure that you will ever reach the line of code which does the clearing up.

In a language with non-deterministic destruction such as C#, you can use the using construction. This is a lot messier than the C++ constructor/destructor mechanism because you have to implement IDisposable. This interface only has the `Dispose()` function on it, but to implement it properly involves being aware of finalizers being called multiple times, and taking into account the fact that the client may forget to use the using construct at all. One nice feature of the C++/CLI language here is that the C++-style destructor of a managed class automatically gets the using construct equivalent set up, and automatically implements IDisposable properly, so you can use it from other .net languages.

Mechanism Automation – by simply declaring an instance of a RAII class in a scope, you automatically get the resource cleared up when and however the scope is exited.

Risk These classes are intended for use on the stack, so that the stack unwinding mechanism at scope exit invokes the destructors. Someone might allocate an instance on the heap, and it may then not get destructed, and probably at the wrong time. You can protect against this by overriding operator new for that class to either be not implemented/private, or throw an exception or anything else which would cause the misuser to struggle.

Easy to use, hard to break

When developing a class, you must strive to make that class easy to use. [1] That is understandable – you want the clients to adopt it, and therefore having a clear interface, with minimal complexity is a good idea. The functions they call should be as obvious as possible, and you should strive to avoid compulsory sequences. For example, use a one-stop constructor rather than a default constructor followed by an initialisation function.

Hard to break takes this a step further. You want to minimize the chance that your users will either inadvertently or deliberately misuse your class. Therefore, take the time to protect it from improper use.

Devices include cutting down the interface to the bare minimum, or not publicly inheriting from an implementation-biased base class. Derive from `NonCopyable` if you don't want your objects to be copied.

If it's a base class, make the constructors protected and make the destructor either pure virtual or protected.

Where the language supports it, make classes either abstract or sealed so that correct class hierarchies are enforced. C++ users can make the destructor pure virtual to designate an abstract class. Marking constructors and possibly the destructor as protected achieves a similar effect.

minimize the chance that your users will either inadvertently or deliberately misuse your class

Be careful of implicit conversions and make constructors explicit where appropriate.

For value classes, be sure to implement the expected functionality in a standard way. Constructors, assignments, etc. should be canonical in form, 'Do as the ints'!

Proactive use of `const` on both method signatures and parameters will help to avoid inadvertent change of state where not necessary, as well as giving clear indication of intent.

Mechanism Enforcement – by deliberately restricting the things someone can do with your class to those things which it intended to do, you enforce correct use. In addition, by offering an interface which is clean and easily understood, you encourage people to use this class and not attempt to roll their own.

Risk If the interface is too restrictive, people may be put off using it. In the case where there is an alternative (for example direct access to an OS API), this may be worse than having a more powerful wrapper, because they will stick to what they know, with all the inherent risks of memory mismanagement, etc.

Refactor

How many times have you seen code copied and pasted? Here is a prime chance to factor out a common function. Doing so takes the risk out of

copying that code incorrectly, or needing to make future changes in all places.

Other forms of refactoring also fit the proactive laziness bill, particularly if you have a test suite built up.

As I mentioned in an Overload article [2], factors such as having access to files in the source control system can be a hindrance to this kind of activity, particularly where functions need to be added to header files, or new files added to the project in the case of a new class.

However you work or live, have a think what you can do today that saves you time not just tomorrow, but the day after and the day after that...

Mechanism There is a mixture of ways in which refactoring helps. For example factoring common snippets into a function acts a Deterrent of error when coding something needing it next time. This is a lesser form of Enforcement, because you cannot stop the developer copying and pasting again.

Risk Any code changes pose a risk. Incorrect implementations or simply reaching some limit on something may cause failure. Having a set of tests for your code will help to stop this. Indeed, when used as part of TDD , Refactor is an encouraged part of the process.

Separation of algorithm and data

C++ developers will know (or should know) the standard library. One of the mainstays of this is the relationship between containers and algorithms. By abstracting storage behind a common interface (use of iterators), we can write any number of algorithms to do things with ranges of data. For example, we can sort, find, and perform calculations based on a sequence of data.

This may seem at odds with the idea of encapsulation, which tries to keep functionality and data together. It isn't, though, because the actual storage mechanisms of the various containers (e.g. `vector`, `map`, `string`), is still encapsulated, we just offer a common interface for running through the sequence of elements in the form of iterators.

Thus, developers have a handy library of functions they can apply the containers that come with the standard library, or containers they may write (through the use of the `iterator` concept, a little like writing an adapter for a class). In addition, algorithms can be compounded to make more powerful functionality

Mechanism Flexibility – this is achieved through separation of concerns, allowing us to combine things more freely.

Risk Having generic algorithms could lead to inefficiencies if the data structure could be more efficiently operated on with greater knowledge. To combat that, certain containers might offer their own versions of specific algorithms. `std::map` offers `find()` for example.

User interface

HCI (Human-Computer Interaction) is a large subject. There is much literature on usability, but some points in particular are relevant to this discussion, because some effort from the UI designers, or developers can help prevent the user getting into error situations, thus saving them and your support function time and effort.

Poka-Yoke devices

Taken from the Japanese car industry, this concept (pronounced 'POH-kah YOH-keh') is a family of mechanisms which seek to stop error conditions happening in the first place. The Toyota car manufacturing company sought to find ways to reduce the number of mistakes happening on the production line. They identified common problems (such as missing bolts, welds not made, etc.) and then thought of mechanisms by which these could be spotted up front. Then, the situation was fixed at source, avoiding costly downstream action.

We can take the same idea and apply it to user interfaces. If there is a piece of data which has to be in a particular format, ensure that is the case before processing it. E.g. if the user needs to enter an integer, you can restrict the characters enterable into a text field to those. Masked text boxes can be used to ensure data is in a pre-defined format. Or, one step later, you can validate the fields on a form before accepting it. When done best, this does not let you submit a form which contains incorrect data. It should be accompanied by a suitable help message somewhere (preferably close to the source of error) to let the user know what they have done wrong.

Mechanism Enforcement – you make sure the user can only enter correct data before attempting to process it.

Risk You can be over-restrictive if you don't analyse the situation well. For example, expecting telephone numbers in a country-specific format is bad if that application is to be used by people abroad. Similarly, not all countries have a zip or postal code, and they are not all the same format.

Undo/redo

This is now something users expect in a normal product running on their machines, but is only just becoming more widespread in web-based applications. It allows the user to backtrack a sequence of actions, and if so, to replay them again. These actions could be editing a document, or navigating to a different web page in a browser.

Mechanism Flexibility – by allowing the user to explore in the confidence of being able to get back to where they came from

Conclusion

We have seen several examples of 'Proactive Laziness'; where we take steps up front to think how we might do something better or avoid problems in the future. When we do so, we set in place more security about how things will be executed or used in the systems we are involved with.

There are many more examples in many more domains. However you work or live, have a think what you can do today that saves you time not just tomorrow, but the day after and the day after that… ■

References

[1] Parkin/Meyers – Semantic Programming, ACCU 2007 Conference by Ric Parkin, referring to one of Scott Meyers' Item 18 in Effective C++ - Make interfaces easy to use correctly and hard to use incorrectly

[2] 'Up Against the Barrier', *Overload 75* October 2006

How 'professional' *are* you?

Pete Goodliffe helps us to work out our skill level.

So how 'professional' a developer *are* you? Are you an awesome codesmith? Or can't you code for toffee? Are you a vastly experienced veteran, or a fresh-faced recruit? Are you a conscientious über-programmer who only ever produces robust, verifiable, bug-free code? Are you an coding imbecile who only ever creates embarrassing software spaghetti? Or are you somewhere in between?

Up until today, there was no real way to be sure. Sure, you could ask someone's opinion – but how well do your colleagues *really* know you? Heck, they still think that the bugs you put in your code were accidents (obviously, you were only injecting flaws to test their powers of observation, and to ensure that the QA process is working adequately).

You could just decide how marvellous you are for yourself. But, let's be fair, you might be a bit biased. Or far too modest.

So it's hard to gauge how professional you *really* are.

But fear not! Goodliffe swings to the rescue with a simple five minute quiz that will categorically determine how good a programmer you are. The answers are accurate to sixteen decimal places and five character flaws. Don't say I'm not good to you.

Grab yourself a drink and a comfortable seat and plough through the following test. (You've heard of doing *test-driven programming*, haven't you? If you complete this test then you can claim to be one step beyond that: a *test-driven programmer*. Stick that on your CV and smoke it.)

The questions

Work through all three of the sections below, and for each question mark the answer that applies to you most. When you get to the end, use the score table to work out your overall score (um, your 'unique professional qualification scale rating identifier'). Then refer to the description section at the end to find out exactly how marvellous (or not) you are.

Section 1: Personal

1. Are you an excellent programmer?
   a) Yes
   b) No
   c) I don't know
2. Do you have:
   a) Boy bits
   b) Girl bits
   c) Not entirely sure
3. Do you have a blog?
   a) No
   b) Yes
   c) Yes, and I host it on my own server
   d) What's a blog?
4. How do you prefer to communicate?
   a) Talking, like God intended
   b) Phone, like God intended for ugly people
   c) In writing (with a quill and pig's blood if possible)
   d) Email, so I have an audit trail to shirk blame
   e) Instant messaging
   f) VOIP (I like anything with a SHOUTY acronym)
   g) Video conferencing
   h) Morse code
5. When did you last wash?
   a) This morning
   b) This week
   c) This month
   d) I can't remember
   e) None of your business
6. Who is this?
   a) Her Majesty The Queen of England
   b) His Majesty Bill Gates
   c) Linux Torvalds
   d) F. P. Brooks

7. How would you describe your eating habits?
   a) I eat a little bit
   b) I venture a cautious nibble
   c) I take a byte
   d) I eat my own words
   e) Eh?
8. How do you learn about new techniques?
   a) Google
   b) Wikipedia
   c) C Vu and Overload (or other magazines)
   d) I buy books when I need to
   e) I don't need to learn anything new. I know programming.

Section two: Programming

9. What is programming most like?
   a) An art
   b) A science
   c) Engineering
   d) A craft
   e) A game
   f) A way to pay the bills
10. Which OS do you use regularly?
    a) Windows
    b) Mac OS
    c) Linux
    d) Unix
    e) Other
    f) 2 of the above
    g) 3 or more of above

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@cthree.org

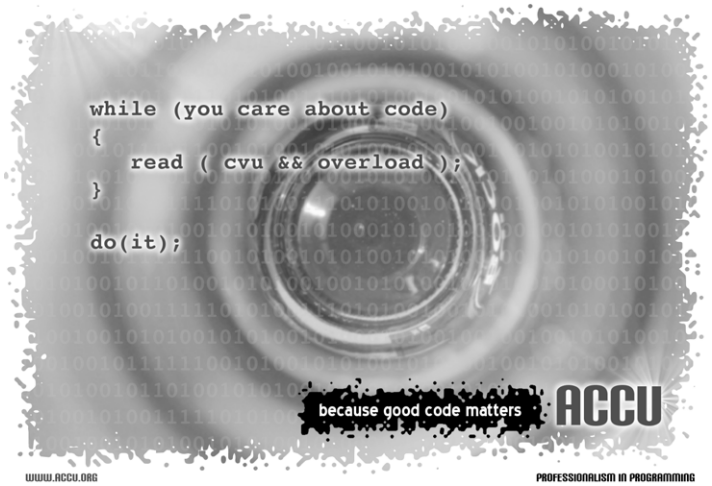
11. Which OS do you program on regularly?
    a) Windows
    b) Mac OS
    c) Linux
    d) Unix
    e) Other
    f) 2 of the above
    g) 3 or more of above
12. Your coding style:
    a) Fast and furious: there's code flying around all over the place
    b) I shoot from the hip: I write what comes to mind as it comes to mind
    c) Slow and steady: I think carefully and hone each line precisely
    d) Test-driven: as long as the tests pass, it's good
    e) Coding? Eek! Run away as fast as you can.
13. You're stuck. Your first reaction is to:
    a) Ask someone for help
    b) Google it
    c) Ask someone to 'pair program'
    d) Ignore the entire problem, and feign surprise when someone files a bug report
14. You need to add some new variables to a function. What do you call them?
    a) a, b, c
    b) tmp, tmp2
    c) foo, bar, baz
    d) elephant, flamingo, duck
15. You need to look at someone else's code. Do you:
    a) Involuntarily vomit
    b) Relish the opportunity to learn from someone else's coding style
    c) Open source files tentatively, and try not to make any modifications
    d) Get someone else to do it
16. There's a nasty bug. Do you:
    a) Ignore it. Call it a feature.
    b) Ignore it and hope someone else fixes it
    c) Try to fix it as quickly as possible, with minimum impact on what you were doing
    d) Drop all other work until you've found and fixed it
17. Have you done any assembly programming?
    a) Yes, as part of a programming course
    b) Yes, for fun
    c) Yes, as part of a main programming task
    d) It's all I ever do
    e) Assembly? Is that like Meccano?
    f) What's Meccano?
18. Which programming language is better?
    a) C
    b) C++
    c) Java
    d) C#
    e) Anything on .NET
    f) Python
    g) Ruby
    h) Any dynamic language rocks
    i) Any static language rocks

Section 3: Process, teams, and other tedium

19. How well do you cope with team work?
    a) I can drink with the best of them
    b) Other people frighten me
    c) I'm the best programmer I know
    d) As long as they do what I say, everything will be OK
20. Which source control system do you use?
    a) What's source control?
    b) None
    c) Visual Source Safe
    d) CVS
    e) Subversion
    f) Another client/server system
    g) Another distributed system
21. How do you check code in to your source control system?
    a) I check in little and often, as I work
    b) I save up big chunks of work, and check it all in at once
    c) I only check in when the code builds OK
    d) I only check in when the code works
    e) Source control? No, still not with you.
22. When do you release software?
    a) When it's ready
    b) Late
    c) The schedule says we'll ship next Monday. So next Monday it is, then.
    d) The schedule says we'll ship next Monday, but we all know that it'll slip about a month. Or two.
    e) The schedule says we'll ship next Monday, but we all know that it'll slip about a year. Or two.
    f) We're continually releasing
    g) We're continually release high-quality non-beta versions
23. Your methodology of choice:
    a) Waterfall (we're old school)
    b) Iterative and incremental (we're modern)
    c) Agile (we're fashionable)
    d) Whatever – we just kinda do it till it works (we've not got a clue)

Scoring

So you got through all the questions. Well done! Now add up your score using the table overleaf.



Stand and Deliver

Ric Parkin discovers that talking at the ACCU conference can be fun.

Five minutes to go. How did I get to be here? How will it go? Will the audience get to double figures? Do I have a point to make? Agggh, they'll hate me!

Rewind six months to September and my monthly one-to-one meeting with my line manager. 'So have you any objective you'd like to suggest?'. 'Well, the ACCU has just asked for submissions for talks for the conference. Perhaps I should try to think of a topic and put it forward?'. 'OK, good idea'.

I mean, how hard could it be?

Two days later, and still no ideas. Tim wanders over, giggling. 'Have you seen this?' and he shows me a Google Codesearch for a silly programming error where parameters are passed in the wrong order. 'Pah' I snort 'if only the signature was more typesafe, that wouldn't even compile! Hmm'.

I realised that this chimed in with an idea I've had for a while, but never really seen articulated before. Perhaps... A couple of hours later I've a title, a quick description, and hardest of all, the speaker biography. And off it goes. What **have** I done?

RIC PARKIN

Ric learnt BASIC on a teletype attached to a mainframe in 1980 and has been programming professionally since 1991, mainly using C++. He believes that programmers should continually strive to be lazy and is always on the lookout for ways to get a computer to do the work for him.

Professionalism in Programming (continued)

If you had any sense, you'd've abstained from answering some of the questions. But if you didn't you *must* now include those scores in your total! So what score did you get?

They say that there are 10 types of programmer. The ones that understand binary, and the ones that don't. In the Real World there are actually a few more. Find out what kind of programmer you are here:

I lost count: You are a muppet. You are either far too embarrassed about your score to admit it, you couldn't be bothered to add it all up, or you've got a brain like a sieve. If it's the last then get a pocket calculator, or grow more fingers. Otherwise, try harder next time.

Less than zero: You are a liability. Take a long hard look at yourself in a mirror. You are almost certainly a massive danger to the world of coding. Consider a change of career. You'll never make it as a rocket scientist, but perhaps a travelling circus might have oportunities for people of your calibre.

0-25: You are a programming Luddite, a Neanderthal. But as long as you recognise that then all is not lost. Keep learning. Keep trying. Keep practising. If you're lucky, you'll avoid a career in management.

25-44: You show great promise, but you have room for improvement. As long as you never get complacent with your current skillset, or think that you've learnt all there is to learn, you'll have a long, happy, successful programming life ahead of you. Keep it up.

45-55: You are a liar. No one's that good.

More than 55: You can't add up.

That's all, folks!

So there you are. You now know how much of a social misfit or a programming god you are. Go forth and recurse. ■

	A	B	C	D	E	F	G	H	I
1	0	2	3						
2	1	1	0						
3	1	2	3	3					
4	0	2	3	2	2	1	0	0	
5	3	2	0	1	2				
6	0	0	0	0					
7	0	1	2	3	-3				
8	2	2	3	3	0				
9	2	2	2	2	2	0			
10	1	1	1	1	1	2	2		
11	1	1	1	1	1	2	3		
12	2	1	2	1	0				
13	2	2	2	0					
14	0	0	0	0					
15	1	3	1	0					
16	0	0	1	2					
17	1	2	2	1	0	-4			
18	-3	-3	-3	-3	-30	-3	-3	-3	-3
19	3	1	0	0					
20	-1	-100	0	1	2	2	2		
21	3	1	2	3	0				
22	2	1	2	1	1	2	3		
23	2	2	2	0					

Starting to put it together

I do nothing for a while but vaguely think of ideas, then around the start of November, the email arrives: they've accepted it, the mad fools!

So I start to collect ideas from two different directions – lots of examples that I feel might illustrate my point; and a rough outline of the sort of stages I think my talk should take in. I've done a single article for the ACCU Journals before, and it took a **long** time to come together. The single most important breakthrough was finding a cohereant 'narrative' that acted as a scaffold upon which the rest of it would hang. Once that is there, the rest flows really easily.

Christmas? That's when the three weeks outstanding holiday has to be used up. At least I'll get a lot of my talk done.

Riiiight!

I did a bit more of the background ideas, but not much more

Then I got back to work and found the next email – time slots had been allocated. Overall I think it was good news – first slot after the first morning keynote. Hooray, I'll be fresh, not hungover, and alert! Bad news, so will my audience. And a hint of doom – while no rooms had been allocated, a rough track system was in place with the relative announcements: my talk was so general, it was placed first. If the layout was like the previous conferences, that means I'm in a big room, perhaps 100-200 people? EEEEEK!

Really time to get going. So by the start of March I had a big load of slides ready. But could I get up and do it? Well, I've done small scale presentations to a few collegues and found I could do them although with a certain amount of hyperactive bouncing. Perhaps I could do better?

Write and rewrite

So I went on a two day presentation course. It was mainly aimed at doing formal pitches, but showed up a lot of things I should be aware of: damn that video camera! But I now knew to try and restrain the handwaving lest it distract from the points I was trying to get across, and how best to position the visual aids around you. And most interesting in this PowerPoint-centric world – that you should tell your audience what the next slide is going to show them. And of course, it's **good** to have pauses – it gives the audience time to think about what you've just said, and you time to think about what to say next.

I came out with some great things I could use, and I did them straight away. I completely rewrote my talk, mainly to pull all the detail I needed to say down into 'cheat notes' and off the slides – they were there to introduce the ideas, not to do them to death. That was my job!

I knew to avoid excessive slide transistion animations, so made sure the few there were simple, non-distracting, and had a point to make.

And in the last few days, I tried to introduce pictures. A million code-drenched slides made for a fairly dull talk but getting something colourful in can really emphasise a point. But be careful of overdoing it, and I tried to make sure the pictures were relevant.

But I missed the deadline for materials to be on the CD. I had really set that as a personal deadline so felt failure, but also knew that what I'd just improved meant the CD would have had the rough alpha. If I'd had a better idea of my talk up front or done it to multiple groups, then yes it would have been there, but this was new!

But most of all, I needed the self-confidence that I'd be able to do it – get up and talk for an hour and a half to people who really knew their stuff. Daunting, but achievable so long as you **believed** you could do it.

Look! No hands!

Ah, toys! In my presentation course, we had to talk about a talk that someone had done that had really made an impact. I remembered Herb Sutter at an ACCU conference where he'd used a wireless gadget to change his slides – no more wandering over to the laptop, hitting the wrong button etc, and the slides behind would emphasise the point he was making *as he said it* without a pause. Neat.

So I got the company to get us one. Which partially died after one use, but the main next button worked. And as a tool, it **really** worked – I just put it in my pocket, and just touching my leg as if I'd dropped my hands naturally would go to the next slide, leaving both hands to point at the screen, audience etc

I know you are meant to do this but I avoided practicing much. I had the opportunity to do it to people at work, but made excuses – I needed encouragement I could do it and positive feedback, and was fearful of a full criticism. But I needed timing help – I've seen people with far too few slides, or far too many struggle against the time slot. So I practiced in front of my girlfriend to get some idea, and the night before in the hotel, to get detailed timings of how close I was. Thankfully it didn't seem too far off, although I was worried my 41 slides in 90 minutes would be too many with the amount of explaining I would going to do. But best of all, going through it properly in front of a mirror, made what I could say easier to remember.

And a few final last minute changes crept in past midnight...

Show time!

I was up ridiculously early to get breakfast and help set up our company display stand, but also to make sure I had everything I needed. I even slipped out of the preceeding keynote to get to my room and see what was there. I moved the screen from the inevitable centre-stage to stage right, and set up a flip chart stage left. This would leave me in the middle so I would be speaking centre stage, which after all is where most of the talking would happen. And ***lots*** of mucking around with PowerPoint to try and get it to go to the next slide on both the main screen ***and*** my notes. This should be basic functionality but I could't get it to work for ages.

And then someone walked in with fifteen minutes to go! I hide. Two more come in. Five to. Is that it? Then the world goes mad and about a hundred people cram into a room for sixty-five.

Time to do it....

...I was nervous – I don't naturally like public speaking, who does? – but something kicked in and it just *flowed*. Yes, I missed bits I meant to say, but who else knew that? A few interruptions, but good questions and I managed to answer them, even the one that spotted the syntax error – fortunately solved by a suggestion two slides further on!

And then finish a little early for lunch, which kept people happy

It always seemed a little odd to me when people went up to the speakers after a talk, and now it happened to me! The first was to ask how I changed the slides (respect to the gadget!), others to get copies of the slides – more requests turned up via email. Some good discussions of things I'd suggested, and a couple of well-observed critisisms of techniques. Best of all were the comments that 'I've been trying to tell my company that!', and a reference to my talk in another session later that week [Roger Or] – pity he forgot who said it.

Go on, you know you want to

Would I do it again? Sure. Just need to to find a good idea, plenty of time to hone it, and the self confidence to do it again. But the second time will be easier now I've done it – I know some of the thngs to do better – really start earlier, better techniques, practice more, better song and dance routines.

What do I get out of it? A damn good line on my CV; confidence that people I respect actually care to listen, and even agree with me; and a cheap way to actually go to the conference if I had to pay for myself. Fortunately my current employer is happy, but I've been places in the past where it would have really helped me go. And most of all, it was surprsingly fun to do.

Would I recommend it to someone else? Yes! ■

Customising a Diskless Linux

Silas Brown finds alternative tools.

Most readers will have heard of Knoppix (www.knoppix.net), a live bootable Linux CD. As long as you can boot a system from CD, getting a full-featured Linux desktop is simply a matter of booting from the Knoppix CD and waiting; no hard disk access is required (although if you want it can access the hard disk and even install itself to it).

The main problem with Knoppix is that, unless you have a great deal of RAM, it is almost constantly accessing the CD, which means it's quite slow (bad for demos) and the CD-ROM drive is always occupied (and if that drive is on a laptop's docking station then you can't take the laptop out of the docking station).

Puppy Linux (www.puppylinux.org) is a smaller live Linux CD which loads itself entirely into RAM (as long as the system has more than 128M) and then runs completely diskless, unless you want to save your work, and if you do then you can save it to a USB key and dispense with mechanical disks altogether apart from at bootup. You can even boot from USB if the BIOS supports it, but many old systems don't.

Running a system entirely from RAM does have its advantages. For one thing, hard disks are fragile. If you've ever wanted to hold a laptop at an angle or carry it around when it's running, but have been afraid of causing disk crashes, then running without the disk may be useful (just remember to shut down the disk by typing **`hdparm -y /dev/hda`**, or even take it out of the system altogether before booting from CD). Hard disks are also one of the first things to break in a laptop, so Puppy can let you use a laptop that's otherwise broken (unless you want to replace the hard disk, but I'm tempted to think that it's a false economy to throw expensive new components at old laptops). The CD itself is needed only during the bootup and can thereafter be freed up or removed.

Unbootable PCMCIA

A problem is that Puppy's hardware detection is not as comprehensive as that of Knoppix. On some systems (like Sony Viao laptops where the CD-ROM is a PCMCIA device in the docking station) Puppy cannot detect the CD that it is booting from, which makes things difficult. Like many live CD Linux systems, Puppy first uses the BIOS to load a small 'initial RAM disk', and then from this RAM disk it loads kernel modules and other software that will help it to access the rest of its filesystem on the CD. This latter stage fails if it's a PCMCIA drive (and the DSL distribution doesn't do much better; it just complains of not being able to find its KNOPPIX filesystem, although Knoppix itself has been able to boot from PCMCIA since at least version 3.4 which was released in 2004, so perhaps its derivatives like DSL haven't been keeping up with its developments). You might have thought that, as Puppy runs entirely in RAM, all its data can go on the initial RAM disk without the need for a second stage, but it does need to do some sanity checking before loading everything (for example there is an option to run from disk if RAM is very short) and another problem is complications in kernel support for RAM disks.

You can put the Puppy files on a USB device and, if the system cannot boot directly from USB, use the CD for the first stage of the boot and let it find the rest of the system on USB (specify **`PMEDIA=usbhd`** at the boot prompt, or edit the text file on the CD image to make this the default). This works, but it means the USB device must be connected all the time (it doesn't just load itself and dismount the device like it does with CD), and in some cases you can't take the laptop out of its docking station without breaking the USB connection and hence crashing the system. Moreover in many cases old hardware is very slow at running Puppy from USB. So I wanted to try customising the CD.

SILAS BROWN

Silas is partially sighted and is currently undertaking freelance work assisting the tuition of computer science at Cambridge University, where he enjoys the diverse international community and its cultural activities. Silas can be contacted at ssb22@cam.ac.uk

Customising the initial RAM disk

I started by going back to Puppy version 1.x which is simpler and has more in the initial RAM disk. Its scripts are programmed to look on the RAM disk for the `usr_cram.fs` file which contains the rest of the system, and to check on the CD as a fallback. So if that `usr_cram.fs` file could be added to the initial RAM disk then the kernel won't need to be able to see the CD drive to find it. The old 1.x releases can be hard to find for download these days, but I managed to get version 1.09 (Community Edition) via distrowatch.com. On another Linux box, I mounted the ISO file (the CD image) with the loop option (**`-o loop`**). (It's useful if you can make lots of directories to act as mountpoints, because we will be doing quite a few loop mounts.) After mounting the ISO file, I found `image.gz` (the initial RAM disk) and copied it to a different directory (necessary because ISO filesystems are always mounted read-only), then decompressed (`gzip -d image.gz`) and mounted 'image' with the loop option (it's in ext2 format). Then I created a file just large enough to hold both `usr_cram.fs` and the decompressed RAM disk image (plus a little for overheads), ran **`mke2fs`** on that file to make a filesystem and mounted it with the loop option as well, and copied everything from the first image plus `usr_cram.fs` into the new image. Finally I unmounted everything, gzipped the new image, and created a replacement ISO for the CD. This last step was a bit subtle due to the requirement to make the CD bootable (and you can't just append a session to a multisession bootable CD because the boot information has to be in the LAST session, so you still need to consider booting); I eventually figured out that the required parameters are:

```
mkisofs -no-emul-boot -boot-load-size 4 -boot
-info-table -b isolinux.bin -c boot.cat -o
isofile.iso to-cd/
```

where **`to-cd`** is the directory containing the files to go on the CD – `boot.msg`, `image.gz`, `isolinux.bin`, `isolinux.cfg` and `vmlinuz` – obviously `usr_cram.fs` isn't needed now it's included in `image.gz`.

There are a couple more subtleties however. Firstly, the Linux kernel (at least the version that Puppy uses) needs to be told the size of the RAM disk at bootup, and if you've made it bigger then you need to tell the kernel so. You can specify this either at compile time or on the kernel command line; obviously the command line is better because it avoids having to configure and compile a replacement kernel. Either type in a full command line at boot, or (more convenient) edit `isolinux.cfg` before you make the ISO filesystem. The parameter to set is called **`ramdisk_size`** and you should set it to a little bit higher than the number of kilobytes of the uncompressed image (not the compressed image). For example, set **`ramdisk_size=72000`** on the **`append`** lines of `isolinux.cfg`.

Secondly, Puppy's `/sbin/init` is a shell script which copies the contents of the RAM disk into a different filesystem called tmpfs (which can, among other things, grow and shrink more easily) before passing control to the all-in-one embedded Linux utility 'busybox' to do the real init. If you look in `/sbin/init` you'll see a whole block of **`cp -a`** commands, and the new `usr_cram.fs` file needs to be added to that block, otherwise it won't be copied to the tmpfs and the later scripts won't see it. This addition to `/sbin/init` should be done from within the

mounted image before doing the previously-mentioned **umount**, **gzip** and **mkisofs**.

However, on a 128M machine, a 72M RAM disk is too big, because at bootup, the compressed image is read into RAM and decompressed into another part of RAM, which means the RAM has to be big enough to hold both the compressed and the uncompressed versions at the same time, as well as the kernel code. Otherwise the kernel will probably hang (but you might get a diagnostic with some versions). So we need to cut down the size a little, and that basically means cutting down `usr_cram.fs`.

Customising /usr

Looking at the name and some of the comments, you might think that `usr_cram.fs` is a CramFS filesystem, but it isn't. As can be seen from `/etc/rc.d/rc.sysinit` in `image.gz`, the Puppy developers switched from CramFS to SquashFS and didn't update all the references. So you will need SquashFS to edit `usr_cram.fs`.

At the time of writing, SquashFS has yet to be integrated into the standard Linux kernel, but it is available as a patch from squashfs.sourceforge.net or you might be able to use your distribution's package management. Once you've installed it, make sure to turn on the SQUASHFS option and compile (and if you set it to build as a module, do a **modprobe squashfs**). Then you will be able to mount the `usr_cram.fs` file with the loop option.

As the squashfs mount is (like an ISO mount) read-only, you will need to copy it in its entirety to another directory before you can start customising. Then you might want to do **du -h** to see how the space is currently being used. When you have finished customising, you will need to use the **mksquashfs** utility, which just needs as parameters the source directory and the resulting file which in our case will be called `usr_cram.fs`.

If you just want to put `usr_cram.fs` on the CD then you're done; just run the **mkisofs** command with the boot options as above. If you want to put `usr_cram.fs` on the initial RAM disk (as I did) then you will also need to follow the above steps for customising that RAM disk after you have created the new `usr_cram.fs`. It wasn't too difficult to get `/usr` down to just over 100 megabytes, which led to a `usr_cram.fs` of about 40 megabytes and a **ramdisk_size** value of 56000 was adequate; this meant a 128M system could boot the CD (and have RAM left over once is has freed the memory after its decompressing and copying).

Back to Knoppix

While I had a working Puppy 1.09 system that boots entirely from the initial RAM disk, the next problem was that I needed a more up-to-date kernel and userspace utilities to be able to do what I wanted with certain USB devices. I could have taken a Puppy 2 system and tried to do the same customisation, but it would be much more complicated (look at Puppy 2's `/sbin/init` script to see what I mean).

However, the original Knoppix (which, as mentioned before, knows how to boot from a PCMCIA drive) is also capable of booting entirely to RAM, if you have enough RAM. If you don't, then you can create a 're-mastered' version of Knoppix, somewhat like creating your own version of DSL but with a more up-to-date version of the Knoppix boot process as a base.

There are re-mastering instructions available for Knoppix at http://www.knoppix.net/wiki/Knoppix_Remastering_Howto but it's still rather difficult to get Knoppix down to size especially if you want X11. However you probably don't need all the different X servers and fonts, and in some applications you can even do without a window manager and basic X clients as long as you make an `/etc/X11/xinit/xinitrc` that does something sensible (this is probably the best place to put application-specific startup logic when you're on a stripped-down system). Also, Knoppix's compression is better than that of Puppy Linux, and it doesn't have to go in the initial RAM disk, so you don't have to get it down to 100 megabytes; you can aim for 200-250 megabytes (excluding `/proc`) if you want it to fit in 128M RAM. You can save the last few megabytes by removing the Debian packaging infrastructure and documentation, although it's probably best to leave this until last and take a backup first.

Finally you'll be wanting to change the default boot commandline (at least to add the **toram** option); to do that you need to edit `isolinux.cfg` in the boot directory before making the CD image.

Re-mastering a small version of Knoppix takes longer than customising Puppy 1. The Knoppix option involves a 700+ megabyte download, a write to a CD-R (the downloaded image is too big to fit on a 650M CD-RW, and so you won't be able to test that it boots on old hardware that can't read your CD-R; you have to wait until you have the small re-mastered version before you can try it), at least two reboots of your working Linux system, about 4 gigabytes of spare hard disk space, a lot more packages and files to trim down (you can start saving space fairly quickly but after a while it becomes harder and harder to figure out what else you can safely do without), and a longer-running compression process at the end. The good news is that the remastering instructions still mostly work with old versions, so if you already have an older version of Knoppix on CD then you can (if it does what you want) re-master that instead of downloading a new one. So if Puppy 1 can do what you want then you might be better off sticking with that, otherwise start with a version of Knoppix that works.

Other remarks

Note if you are booting any live CD that some older laptops have drives that can read DVDs and CD-RWs but not CD-Rs; other systems can read CD-Rs but not CD-RWs. So if you don't know what hardware you'll be facing, it's best to have both types of CD as well as a USB key.

Running a system entirely from RAM is certainly very fast, although if the system's RAM is limited then you can't do too much in it (don't try compiling the Boost libraries). However, I can think of a few applications:

- when you need to use a laptop at an unusual angle, such as on a lectern while giving a speech. Many lecture and conference facilities are laptop-friendly but there are instances when they're not, and running with the hard drive turned off takes away some of the worry.
- as a rescue tool that's faster than Knoppix (but it's best to have Knoppix as a backup in case Puppy doesn't work)
- when you want to minimise noise as much as possible, and you already have quiet fans (or are running a laptop whose fan is off most of the time) so the hard drive is the loudest thing, it can be good to work without it (however, modern hard drives are quieter than they used to be)
- when you're on a budget and need to make the most of old hardware

If you are very well-funded then you could run a high-capacity solid state drive instead of a hard drive, or even a DRAM drive functioning as a hard drive, but that's for very high-end applications (and is priced accordingly) whereas the boot-to-RAM live CD approach makes more sense on more common hardware.

Sting in the tail

I mentioned above that if you're stuck on an old laptop where Puppy can't see the CD-ROM drive, then you could use USB for the second stage of the boot, but that this might mean you can't then take the laptop out of its docking station because it would break the USB connection. I thought I was setting up one of those laptops, but after I had done everything, I found that there was in fact another USB port on the opposite side, and that this other port was not obstructed by the docking station and therefore you could plug a device into it and don't have to break the connection when you take the laptop out of its docking station. So on that particular laptop I could have saved a lot of trouble by simply customising Puppy 2's default boot parameters (and perhaps adding a few lines to the Xsession script or something) and making sure there is a USB key with the rest of the system inserted into the right slot, and it would still be able to come out of its docking station after the inital boot (and sooner at that, although application start-up may be slower) and moreover it would be much easier to administrate. It's rather annoying to realise that you just wasted about 2 days' work just because you didn't notice a second USB slot. Oh well, at least the experience might be useful in other situations. ■

ACCU Conference 2007

Pete Goodliffe rounds up a retrospective of this year's awesome ACCU conference.

Another year rolls around, and another ACCU conference has been and gone. And what an excellent event it was. One of the hallmarks of the ACCU is that it is a participatory organisation, and the conference is an exceptional example of this. It is an incredible event – with world-renowned speakers, an electric atmosphere, and plenty of things going on. It's hard to not be stimulated, and you could never claim to be bored!

This year the conference moved venue to Oxford's Paramount Hotel, which was almost universally agreed as a great improvement over previous locations. This venue offered were bigger conference rooms and more breakout rooms, more space to mill around in, aircon, (Oh yes! *Aircon*! That works!), and sports facilities, too.

The programme was as wide and varied as ever, an incredible mix of programming technique, methodology, design, and more. This year's keynote speakers were Mary Poppendieck (the renowned Lean Software Development expert), Hans Boehm (a leading C++ authority), Mark Shuttleworth (found of the Ubuntu project), and Pete Goodliffe (just some guy!). The sessions covered topics from C++ to Python (to Groovy, to C#, to...), from development practices to historical computing, from designing to testing, from tools to techniques. There was a full timetable of presentations, "birds of a feature" sessions, technology demonstrations, evening entertainment (including the infamous *Grumpy Old Programmers* panel discussion), and plenty of socialising (and drinking – more on this later). The programme also boasted a banquet on Friday evening, a lively *Vi vs Emacs* squash competition, and a competition to win an X Box (and, it should be remembered, a copy of *Code Craft*). Phew! It's no wonder that after a few days most delegates had over-exerted their brains, their stomachs, and their ability to stay awake.

As ever, to give a flavour of the event for those who couldn't attend, to encourage those who are thinking about it next year, and to act as a memory jog for those who are *still* recovering from the conference, here are some write-ups from various attendees. These are little tastes of what people thought of the event (the good and the bad), summaries of some of the things that went on, and synopses of the sessions that were attended.

Paul Grenyer <paul.grenyer@gmail.com>

For me the conference gets better every year! That's how I'm going to justify saying this was the best conference ever. The new venue was nice, but lacked the character and location of the Randolph (no free booze for me this year either). It was good to be able to park the car easily and nearby, though, and as we went in large groups the taxi into Oxford never cost more than £2 each.

I talked to a lot of people at the conference this year and there were two things I heard consistently:

- Those who weren't already on accu-general find it very difficult to get on and assume that they are, but that the traffic is low as they don't receive any emails. I think something must wrong with the sign up process.
- The ACCU is still perceived as the Association of C and C++ users and it's certainly true that, although other languages are creeping in, the majority of the conference is still C++ focussed. I think we need to seriously change our perception. Not just of the conference, but of the association. Maybe even a name change. Even if you refer to it as just the ACCU (or the horrid "aque"), you can never get away from explaining what it used to stand for. I don't think C++ is dying yet, but other languages are growing and most people agree we want our membership to grow.

Every session I went too was very well presented. Particular highlights for me are Pete Goodliffe's key note (where does he get his material?), Andrei Alexandrescu's 'Choose your Poison: Exceptions or Error Codes' and of course Russel's attempt to write off C++. Was I convinced? Not in the slightest, but it was very entertaining and Aeryn got a back-handed plug, so I can't complain. I must thank Kevlin for plugging Aeryn too and the rumours of me giving Peter Sommerlad a hard time about CUTE in his session are slightly exaggerated.

Roll on next year...

James Slaughter <slaughter@acm.org>

This was the sixth ACCU conference I've attended. Though I'm not currently employed, I didn't hesitate to book my place for the four full days, as I've always found the event to be exceptional value for money – and a lot of fun! This year certainly didn't disappoint on either count. There seemed to be a healthy mix of new and familiar faces, both as speakers and as delegates, and all the talks I saw were of a high calibre.

There are three things that don't seem to get any easier from year to year: picking which sessions to attend when there are so many to choose from, getting a table at the Pizza Express on Cornmarket Street, and going to bed at a sensible time. One of the "big-name" speakers in particular made two of those exceptionally hard this year, and the new location didn't help with the other, but I can't really complain! Next year's conference has a lot to live up to, and I'm looking forward to it.

Semantic Programming (Ric Parkin)

Ric Parkin's talk was a welcome reminder that sometimes a little bit more effort spent on a function or class's interface can make code more readable and often safer. The talk consisted entirely of practical measures to improve code written in C++; Ric started his presentation with an illustration using `memset`, adeptly demonstrating that even a function this simple is prone to accidental misuse in real-world code, before leading the packed room through a surprising number of ways to exploit features of both the language and of modern development environments to reduce this likelihood for an equivalent function.

After exhausting the `memset` example, Parkin proceeded to a variety of others, always emphasizing that the best possible outcome is one in which incorrect code doesn't compile. The audience remained interested throughout and occasionally contributed alternative techniques for a topic that was clearly both important and familiar to them.

Fingers in the air: a gentle introduction to software estimation (Giovanni Asproni)

Giovanni Asproni's often light-hearted presentation tackled the potentially dull topic of estimating project time-scales in a way that captivated the

audience. Asproni's talk combined insight, common sense and anecdotes, and his audience responded likewise with interesting comments and questions throughout, including those of whether producing estimates is really addressing the underlying problem and who should be involved.

The session covered concepts such as the difference between estimates and commitments, and made much of the degree of uncertainty with respect to progress. Various techniques for producing and interpreting the amount of work remaining and tracking work to date were discussed, along with advice for managing situations where commitment to other projects and time 'lost' due to external factors is hard to predict.

Anna-Jayne Metcalfe<anna@riverblade.co.uk>

Wednesday: Tar'ed and Feathered

The first keynote session was 'The Software Development Pendulum' with Mary Poppendieck. The conference hall was packed, and we managed to find a couple of seats near the front. Mary presented an interesting history of software development – and in particular some of the early failures. A recurring theme was that the vision for larger systems has consistently been at or beyond the limits of the hardware and software capability of the time. One thing which came out very strongly was the contrast between large systems, smaller systems and software products – the latter are not prone to failure in the same way, and indeed my own experience bears this out – none of the projects I've worked on have failed, and all have been delivered. At the end of the session Mary touched on Lean Software Development, which is something I feel we should explore further.

the vision for larger systems has consistently been at or beyond the limits of the hardware and software capability of the time

Next up was 'Coaching Software Development Teams' with Michael Feathers. This was a session I've been looking forward to. Michael is the author of *Working Effectively with Legacy Code*, a book which I've found phenomenally useful in bringing Visual Lint under unit test and using TDD techniques to develop the product further. His topic for this talk was how to coach teams to to use agile and quality centric techniques to help teams improve their processes and thus the quality and fitness of their products. Invariably, a central topic of this talk was human behaviour, and how it can affect team dynamics. The values and motivations of team members are something a coach must recognise in order to achieve their aims. Changing the values of an organisation in a significant way is extremely difficult (and therefore likely to fail), but building on the values of a team and/or its organisation can be very successful. It was very notable that the approach presented was analytical rather than human centric.

When a delegate asked how to formally evaluate the values of the team I couldn't help thinking that something fundamental was being missed. To understand a team, you have to know them – and to do that you need to meet them both in and out of their everyday environment – in other words get to know them as people as well as professionally. I couldn't help thinking that much of the techniques being presented were ones I've been using for some time, but it was useful to see it disseminated in a structured setting, nevertheless.

After a relaxed lunch and chat out on the grass we headed back in for the first of the afternoon sessions – 'Reviewing the C++ Toolbox: Identifying tools that support Agile development in C++' with Alan Griffiths. During the introduction he identified the wide variation of tools in C++, with no consensus between groups (or even teams!) as to which tools are best suited for each role. This was an interactive session – the idea being that delegates would share their experiences of tools in particular domains and suggest alternatives:

- Build systems/source control/continuous integration
- Test frameworks
- Refactoring
- Code documentation
- Modelling + round trip
- Editors/IDEs
- Code analysis
- Debugging
- Instrumentation/coverage/profiling/performance analysis

We broke into 6 groups to discuss various areas from the list above. I must have stuck my hand up once too often, because I ended up leading (with another developer who worked on QA C++ until recently) the group discussing code analysis. It's the first time I've presented in front of a group of peers for some time, and to be honest I was not too sure how well I came across at the time – I felt quite nervous, but it seemed to go down well.

Next up was 'Linting Software Architectures' with Bernhard Merkle. We've been talking with Bernhard by email over the last couple of weeks, so this is a session we've been looking forward to. The focus of this talk was architectural analysis, and in particular at tools which can be used to automate such analysis. The need for such tools is of course a result of the phenomenon of 'architectural decay' – a problem which should be familiar to any developer who has had to maintain working software systems.

- **Architectural analysis** works on layers, graphs, subsystems, components, interfaces etc., and assesses metrics such as coupling, dependency etc. as well as things like consistency analysis and detection of anti-patterns.
- **Consistency analysis** is based on the premise of comparing the codebase with a 'gold standard' model. Key to this sort of analysis is of course how results are presented (to be honest, the same considerations apply in code analysis).
- A further analysis type described was **Rating of architecture**, which assesses characteristics such as cycles, coupling, stability and the presence (or hopefully absence) of anti-patterns.

Afterwards everybody gradually congregated in the hotel bar to socialise. As tends to happen, a consensus on where to go next gradually arose, as a result of which by 8pm a bunch of us were piling into taxis for the trip into town. The initial plan was to eat at the Randolph, but as ever things changed at the last minute and we found ourself in the Eagle and Child waiting for the remainder of our contingent, where good food, much beer (and the occasional red wine) flowed amid an equal volume of hilarity.

Thursday: Forgive me Father, for I have singleton'ed

(or: 'When patterns meet anti-patterns, do they annihilate?')

We had a late start this morning, skipping the opening session because we needed a break after the marathon yesterday (I crashed out with a headache at 6pm yesterday, only waking up to go to the bar to eat at 8pm).This morning we met up with a couple of other developers at breakfast and had

a lively discussion about multithreading, the idiosyncrasies of development tools and network compilation techniques (as seen in Incredibuild). We found the discussion very useful – in particular it helped us to focus some ideas on future versions of Visual Lint – and in particular on the potential for integration with build and continuous integration systems.

The first session this morning was 'Choose your poison: Exceptions or Error Codes?' with Andrei Alexandrescu. Coming from a Win32 background as we do, I suspect we tend to use the latter a little too much, so we saw this as an opportunity to learn an alternative viewpoint. Andrei's delivery was humorous and entertaining. One early point was that in many cases it can actually makes sense to implement both schemes in a library. Ultimately, the consumer is best placed to evaluate which scheme is appropriate for a given situation. Another way is to categorise into soft errors and hard errors. The former can safely be ignored; the latter cannot. Another consideration is the state of the system – if the system is left in an undefined state, an exception is almost certainly appropriate. The old C standard library function `atoi()` was used as an example of a library function designed without consideration of error conditions – neither has an error code return nor throws an exception on failure, instead returning `0` (arguably the most common return value!) on error. Andrei presented four solutions to return error information:

- Set a global state (the '`errno`' approach). This approach (also used in Win32 as `GetLastError()`) is easy **but** has big issues with threading and makes error handling far too optional for many tastes
- Encode the error code as a reserved return value. A reserved value conveys very little information, and cannot support centralised handling. It also requires error values to be reserved in the normal return code, which is not possible for `atoi()`, for example
- Encode the error information as a value of a distinct type (an error code). This approach (commonly used in Win32) works quite well but does not lend itself to centralised handling – its success is entirely reliant on the caller checking the returned error code
- Exceptions (effectively 'covert return values'). Exceptions are good at supporting centralised error handling. Local handling is also possible, but more long winded.

A key consideration is that only certain callers understand certain errors – the local caller may not, but a higher level method which ultimately invoked it may be better placed to understand the context of the error. The converse can often be true. From a personal perspective, I have to say that debugging exceptions is not as easy as it could be, and furthermore if you use exceptions you **must** ensure that declared exceptions are caught *somewhere* (far too many developers don't, sadly).The top three issues with exceptions discussed were:

- Metastable states – the user must ensure transactional semantics, or the system can be left in an undefined state.
- Local handling is unduly verbose.
- They are hard to analyse and debug.

Andrei then presented an intriguing proposal – an alternative approach which combines some of the better characteristics of both. I won't go into it further other than to say it was an intriguing proposal involving a template type called `Likely<T>`.

During the lunch break the Perforce guys were hosting a sponsor session called 'Branching and Mergine without a Safety Net'. We went along out of interest (SourceSafe really is showing its age now, and we're always on the lookout for new tools) and I'm very glad we did. The delegates' pack (a rather useful ACCU 2007 bag containing lots of flyers and a rather fat A4 pad) also includes a CD with a fully functional 2-user copy of Perforce, so I dare say we'll try it out when we get the chance.

After lunch the world and his dog squeezed into a far too small room for what turned out to be a highly entertaining session by Kevlin Henney entitled 'Pattern Connections' (incidentally, the title of this post is a quote from the session if you didn't realise it). I can't even begin to do it justice here. Suffice it to say that if you get a chance to see him present – just go! Kevlin is apparently a brilliantly entertaining speaker at the best of times, and today he was definitely 'on form'.

Next I headed for the session 'Test Driven Development with C# and NUnit', which was actually an extract from Learning Tree 511 course (*.NET Best Practices*). We had a very small contingent for this session, but at least that meant we could feel the air conditioning for a change! Encouragingly, I found that the vast majority of the content of the session covered topics and practices I was already familiar with. As a result, I didn't learn much, but it was a useful confirmation that we are heading in the right direction with TDD. Given that we only started using it back in December, that is extremely encouraging. During the break we congregated at the Perforce stand for a brief hands on demo of their product; effectively a follow up from their sponsor presentation earlier. Suffice it to say that we were **very** impressed with what we saw, and we are quite likely to 'jump ship' from VSS in the reasonably near future as a result.

the natural tendency to structure a large team around functional blocks is one which runs counter to the agile aim of keeping the system working at all times

After the break was 'Grumpy Old Programmers – The Ultimate IT Chat' or (more accurately) 'We Want Beer!'. This was a freeform, irreverent and **very** funny discussion, frequently interrupted by calls of 'More beer!' from the panel.

Friday: What do you mean, it's morning?

Suffice it to say that this morning we were a bit slow! The first session this morning was 'This Software Stuff' with Pete Goodliffe (the author of *Code Craft*) – a light-hearted but incisive look at what developers who care about their trade should be doing. Appropriately, this session was anything but serious, which was I think exactly what we needed after the fun of last night. If you get a chance to hear Pete speak, I'd highly recommend going. Just remember to ask him about the fizzy milk and alphabetti custard...

After the interval I headed for the Cherwell Suite for 'Global – Yet Agile – Software Development' with Jutta Eckstein. The session discussed techniques for running Agile processes in large and often geographically dispersed software teams. Obviously, these have major implications for communication – a keystone of agile methodologies. One key point that emerged was that the natural tendency to structure a large team around functional blocks is one which runs counter to the agile aim of keeping the system working at all times and delivering each feature complete *throughout the system* at the end of each iteration or sprint. The project is of course far more likely to succeed if a team or subteam is given complete responsibility for a feature all of the way *from requirement to acceptance*.

The session discussed various considerations and techniques for overcoming the many hurdles which a distributed agile team faces. Communication and synchronisation is key; how you do it (IM, phone, video conferencing etc.) isn't particularly important – but it **must** happen – and ultimately, there is no substitute for face to face contact. It was an interesting session, and although I've not worked in such an environment I think I can visualise the issues clearly.

Over lunch there was a Visual C++ session with Steve Teixeira, the Group Programme Manager for Visual C++. The session started with an informal poll, which illustrated how many people in the room are using Visual C++, how few people are using managed code. and the reasons (versioning, distribution etc.) why they are using native code rather than managed. Steve stated that the first priority of the Visual C++ team was native code, followed by enabling interop to allow use of newer technologies from Visual C++. This is illustrated by the roadmap for Visual C++ Orcas and beyond:

- Renewed investment in native libraries (MFC/ATL)
- Making it easier to interoperate between platform paradigms
- Innovation in areas such as concurrency etc.

there is very little open source software out there which has sufficient support and active development to be used in a defence environment

Interestingly, Steve admitted MFC and ATL have been neglected by Microsoft since managed code emerged. That is now recognised within Microsoft as a mistake and is changing. As a result we can expect to see significant new functionality in the native frameworks (e.g. WPF support in MFC) in future Visual C++ releases. We can also expect significant IDE improvements in future. C++ IDE support is currently lagging behind those for managed code at the moment, and (interestingly enough) the Visual C++ team see managed IDEs as their 'productivity competitors' rather than the functionality offered by add-ins such as Visual Assist. The new Visual C++ features in Orcas include:

- MFC support for new Vista common controls (sysLink, IPv6 compatible network address control, split/drop button and command link.
- Vista UAC support in IDE and projects. Interestingly, the registration of ATL components is now by default in `HKEY_CURRENT_USER` rather than `HKEY_LOCAL_MACHINE`.
- New Vista SDK and APIs.
- STL/CLR – an STL which can be used from managed code, and allows STL interfaces to be used to work with managed collections.
- A marshalling library to make marshalling data between native and managed types.
- Metadata based incremental managed builds and concurrent module compilation (improved dependency checking).
- .NET framework multi-targetting (i.e Orcas can target both .NET 2.0 and 3.5).
- The C++ class designer is back! Unfortunately, in Orcas it is read only – you can't edit class diagrams for C++ projects in this version, only view them.
- ATL Server is now shared source on CodePlex (it is no longer Microsoft proprietary).
- The removal of Win9x targeting. Orcas built projects can target Win2k and above only.

Sadly, Orcas will **not** include MFC support for the `TaskDialog()` API. To me, this is very disappointing given that we really should not be using `MessageBox()` in new projects (WTL 8 already has it, incidentally).

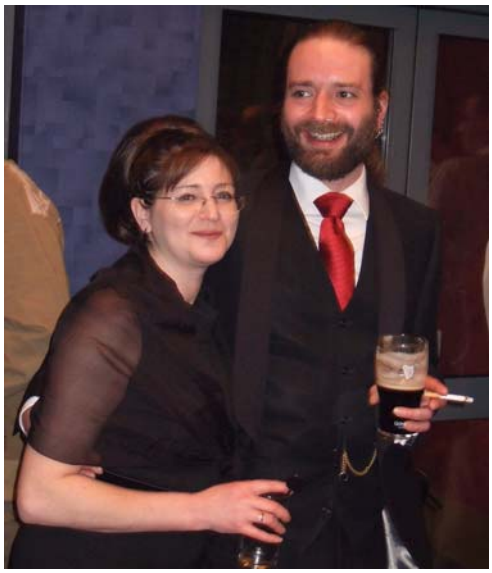
The session finished at almost exactly 2pm so I rushed upstairs to Peter Hammond's 45 minute 'Open Architecture vs Open Source in defence systems' session, which I thought might be interesting given my past experience. The core theme of the session was that in most cases the reality is that there is very little open source software out there which has sufficient support and active development to be used in a defence environment. Projects such as MySQL really are the exception rather than the rule. As ever, there is no magic bullet.

In the evening those of us who weren't booked in for the Speakers Dinner (a tad expensive at £50 per head, we thought) wandered off in different directions to eat/drink/be merry. Beth and I walked into Wolverton with a couple of the guys for a gorgeous meal at the Trout. Yum.

Saturday: A Qt way to eat breakfast

At breakfast time yesterday morning we were sitting in a conference room listening to a seminar by Trolltech – the people behind the Qt cross platform C++ framework. Although we use WTL for our current projects, we are always looking to learn new techniques – and of course the lack of cross platform support is the big Achilles heel of frameworks such as WTL (and of course MFC). It was an interesting presentation, and although we don't have any direct application for it at the moment, it is certainly something we will bear in mind for the future.

The first session of the day was 'Towards a Memory Model for C++' with Hans-J. Boehm from HP Labs. Multithreading is increasingly important in modern software systems, and the difficulties of writing multithreading support are well known. Java and C# define threads as a core part of the language, but even there getting the semantics of threading right is exceptionally difficult. C++ (as ever) presents a different set of problems and considerations, and traditionally C++ has addressed this using libraries. Hans discussed why this is not adequate, using Pthreads and the Win32 threading model as illustrations. The impact of these failings is clear – multithreaded development in C++ is harder (and more error prone) than it needs to be.As a result of this efforts are under way to ensure that the C++0x standard will define a memory model describing visibility of memory accesses to other threads. Herb Sutter is leading a similar effort for Microsoft platforms, and the intention is that the outcomes of the two efforts will be compatible. That can only be good news for C++ developers. The detail of the proposals was beyond the scope of the session, but one obvious point is that a standard threading API will be provided as part of the C++0x language. The full proposal for this area of the standard can be read on the web at: http://www.hpl.hp.com/personal/Hans_Boehm/c++mm.



After a short break we both headed to the Bladon suite for 'Better Bug Hunting' with Roger Orr. The session started with an introduction discussing the high cost of bugs and the wide variation in the effectiveness of individual developers at finding bugs quickly and not introducing

unnecessary bugs in fixing them. Bugs come in many forms, ranging from inconsistent or non-standard UI and badly specified feature, to poor performance or instability. These are however, only symptoms – the root causes are often more subtle defects or flaws. A simple approach to bug hunting might include:

- Understand the system and how it works
- Reproduce the failure. Unit testing can help a great deal here
- Identify where the problem really is – not just where the symptom is
- Change **one** thing at a time
- Keep an audit trail (so you know what you changed and why)
- Canvass views from others on the defect and possible fix
- If you didn't fix it, it ain't fixed. Don't be tempted to 'hide it under the rug'

More typically, it goes something like this:

- Hope it goes away
- Blame someone else
- Open a debugger and poke around in the vague hope of finding something
- Try random changes to see if the bug goes away
- Fix the symptoms, but ignore the underlying defect.

Obviously, being able to reproduce the bug is absolutely key, and often developers have very limited information to go on. This increases costs and makes it less likely that the developer will correctly identify the root cause. Communicating to testers and end users what information is required when a defect is identified is essential. An obvious improvement is to write scripts or code to collect the supporting information we need in order to triage the bug. Included in this category are of course the ubiquitous crash dumps. Log files also have their place – but only if the logs are easy to find and their contents are comprehensible. A bad log message is worse than none at all. Any defect may relate to others which are already documented. As a result it is always worth looking for patterns in the defect tracking database. Even when you think you've identified the cause, it is worth taking a step back. Could there be another possible cause? Is your fix really the right one or do you need to look a bit deeper? Some classes of defects are of course much harder than others:

- You can't reproduce the defect
- The defect affects widely separated blocks of code
- Memory corruption
- Timing related
- Environment related (e.g. permissions)
- It was not the fault you thought it was.

Some techniques which can be used to increase effectiveness include:

- Adding tracing
- Refactoring areas of the code where you think the defect may lie
- Running the system under a virtual machine in a configuration which is representative of that on which the bug was previously seen
- Deliberately stressing the code to see if the failures match the observed symptoms
- Use static or dynamic analysis.

both the general public and developers (who really should know better) will give 'gut instinct' precedence over contrary evidence

From the Conference Chair

Ewan Milne <ewan.milne@gmail.com>

This year's conference was a big event for several reasons: we marked the tenth anniversary of the event, moved to a new venue, and attracted of highest turnout, around 380 delegates. As many of you know, it was also my last in the role of Chair. From next year, Giovanni Asproni will be chairing the event, and I will stay on as a member of the conference committee, most likely for the next two years.

Over the past four years I have gained an enormous amount of satisfaction from being at the centre of such an exciting event, the people I have met and the ideas that have been generated have been incredibly stimulating. But for the conference to stay fresh I feel it is important for me now to take a step back. Not to mention that I am looking forward to enjoying accu2008 in a more, shall we say, relaxed mood. Each year I promise myself I will do better at sitting still for whole 90 minute periods at a time: I rarely manage it, though this year I probably managed to be in sessions nearly 50% of the time, perhaps.

I must stress that organizing the conference is emphatically not a one man job. I need to thank Francis Glassborow, my predecessor, for developing the conference form its inception ten years ago, and everyone who has helped as part of the conference committee: Giovanni, Allan Kelly, Kevlin Henney, Alan Lenton, Tim Penhey and Andy Robinson. Also my wife who has been tremendously patient over the demands on our spare time, and I cannot forget Julie Archer, who has actually worked harder than anyone else to make sure the event has gone from strength to strength in recent years.

Highlights of this year's conference? Being so closely involved with the programme, I am always loathe to pick favourites. But one speaker did unmistakably stamp his mark on the event this year, in the bar as much as the lecture room. So much so that a new word was coined: I was one of many to find myself one evening Lakosed (verb: to find oneself in an advanced state of inebriation, having had large quantities of tequila thrust upon one by a loud, persuasive New Yorker).

- When the cause of a bug is identified, it makes sense to take a step back to determine how to more easily identify and fix bugs of this class.

After a brief and Spartan lunch (all the hotel provided were a few sandwiches) I headed back to our room to rest for a while. Conferences really are exhausting! The final session of the conference was 'C/C++ Programmers and Truthiness' with Dan Saks, an entertaining look at how both the general public and developers (who really should know better) will give 'gut instinct' precedence over contrary evidence. Dan's major example was the placement of `const` in variable declarations (i.e. whether you should use `const T *` or `T const *`), a cause he's been fighting for some years. It got interesting when Herb Sutter and Bjarne Stroustrup both got involved in the discussion... Personally, I think we've got bigger battles to fight. There are far too many developers out there who *don't even use* `const`, and with the trickle of people from languages without `const` (for example C#) into ocasasional C++ coding I keep running across this. It is not likely to get better in the foreseeable future.

Mark Dalgarno <mark@software-acumen.com>

This was my first ACCU conference, a bigger affair than my usual SPA conference with a stronger focus on C / C++ / Java talks, although like SPA there is a good selection of sessions on general software deveopment issues.

Chaos aids learning – coaching software development teams

Of the four sessions I attended on Wednesday I got the most from Michael Feathers talk entitled 'Coaching Software Development Teams'. This presented a number of techniques that Michael has used when working with software development teams over the past few years.

Michael began by describing the role of coach as a person who helps a team produce a desired effect – so 'coaches aid and provoke change', and a big

part of their work is to find teachable moments i.e. occasions where the team can learn from some individual or team issue.

This learning is best achieved in tension / release cycles where a problem gradually emerges and starts creating tension within a team and then a solution is found and the tension is released. Letting a team discover for itself the solution leads to a significantly better (learning) result than providing a solution for them and the biggest problems (chaos) lead to the biggest learning experiences. One audience member compared this to the selling process – whereby a customer finds for themselves how a tool or service can address a problem they may not have fully known about or had a name for.

Duplication is anathema to simplicity, cut and paste is evil

Some of the techniques Michael identified include:

- **Go Sideways** – When problems don't yield to pressure, help people switch gears by showing them a similar or smaller problem. Often this helps the original problem yield.
- **Ask the room** – If a team is tempted to break a rule then ask them to huddle in the centre of the room to discuss the problem. This helps build a cohesive team and builds a strong sense of how the team works and is also a good way of identifying hidden talents within the team as great designers and leaders can emerge from these discussions.
- **Make it physical** – Take the abstract and make it tangible. Michael cited the example of a team that had reorganised their desks so that the QA people sat around the build machine. Developers had to physically walk a gauntlet to contribute code to the build and the presence of the QA people reinforced the goal of only checking in high-quality code.
- **The Flounce** – Identify a hidden problem by asking pointed questions, soliciting comments and then ending with silence. This builds tension until the team eventually puts a name to the problem (release).
- **Push in the water** – Ask people to go beyond their limits and try atypical solutions. Can have great results but take care to be supportive if necessary and handle breaches of organisational behaviour with care.

Michael also talked about the ethical and personal issues involved in coaching – talking about issues like whether every intervention should be 'above board' for example. James Coplien (in the audience) noted that in his experience teams would still react positively if they knew the facts and so there was no need to hide things from them.

In terms of audience demographics there are certainly more people from the embedded space here at ACCU than at SPA. I guess this is a strong reflection of the C / C++ roots of ACCU and the types of organisations (still) using these languages. However there were a few familiar faces at the event including Giovanni Asproni and Kevlin Henney, Charles Weir and John Pagonis.

The evening was rounded off with a sponsor reception and then around 40 of us descended on Pizza Express much to the concern of the staff. I met up with Bernhard Merkle to quiz him on his session on 'Linting Software Architectures' and it turns out he has a background in Model-Driven Development and Software Product Lines and worked with Axel Uhl, one of my 'Code Generation 2007' keynotes, on ArcStyler.

Simplicity in Software

Day 2 of the ACCU Conference saw me at Peter Sommerlad, Kevlin Henney, and Giovanni Asproni's session on the value of simplicity in software development.

This was a workshop session with the aim of working towards a Manifesto for Software Simplicity along the lines of the Agile Manifesto. With a keen and knowledgeable audience we stood a high chance of success...

Kevlin, Peter and Giovanni began by talking about their previous work in this area and what brought them together for this session, for example Peter is hosting a wiki on the topic.

As background to the workshop, the Laws of Simplicity, as formulated by John Maeda, were introduced:

- **Reduce** – Shrink, hide, embody (by abstraction) to make things simpler.
- **Organise** – Make a system of many appear fewer. (The system then becomes easier to abstract and has a higher degree of regularity). Patterns / Themes, Grouping and Clustering are key tactics here.
- **Time** – Savings in time feel like simplicity. When you deal with something, if you can save time by dealing with it in an easy way (or automating it) then things are simpler.
- **Learn** – Knowledge makes everything simpler – build models of domains to make your life simpler – you can identify patterns that you've solved before etc.
- **Differences** – Simplicity and complexity need each other – contrast makes simplicity appear good.
- **Context** – What lies in the periphery of simplicity is definitely not peripheral. If you understand the context then you can make things simpler – if you don't then you can't. Defining the (domain) rules can help communicate ideas and improve learning.
- **Emotions** – More emotions are better than less. Individuality comes through simplicity – making things simpler is inherently creative and satisfying. Emotions have an impact on what you think is simplest in a given context. Acknowledges humanity in coding activity. Gut reactions to code based on expertise and experience.
- **Trust** – In simplicity we trust. You trust things to do the work you expect them to do – but if they're complex you don't trust them. If you don't trust stuff then you make things more complex by erecting elaborate barriers between things.
- **Failure** – Some things can never be made simpler. Recognising that you can't make something simpler saves time (and so makes things simpler).

What lies in the periphery of simplicity is definitely not peripheral

- **The One** – Simplicity is about subtracting the obvious and making it meaningful.

Three Key Patterns were also presented:

- **away** – More appears like less by simply moving it far, far away. (This relates to levels of abstraction).
- **open** – Openness simplifies complexity. Leads back to learning.
- **power** – Use less, gain more. Less code => more software.

After the introduction we divided into groups for a bit of brainstorming around our beliefs about simplicity in software and then reviewed and discussed what each group had come up with.

My own best contribution was 'Simplicity is Satisfying'; other things that resonated around the room were (paraphrasing):

- Duplication is anathema to simplicity, cut and paste is evil – a good argument for (software) reuse.
- Automation is good because of time-saving and its capability to hide complexity – rewriting poor code is leads to more simplicity than refactoring it.

- Doing one thing well leads to simplicity, constraining scope is good – there could be some relevance for people developing Software Product Lines.
- Component / Library / Compiler complexity is favoured over code complexity. Code should be human readable. It's interesting that compiler was named specifically here – I wonder if a generalisation can be made in favouring tool sophistication over code complexity?
- We believe in not crossing bridges until we come to them (in case you don't have to) – there was an interesting discussion here around the trade-off between planning and simplicity.
- Design by contract makes things simpler – both by making assumptions explicit and by providing a form of contractual framework that enforces consistency and simplicity.
- Dynamic languages are preferable to static languages (for simplicity) – a theme that also emerged at SPA 2007.

The full session outputs are now available on Peter's wiki.

Generative Programming in the large – applied meta-programming

The highlight of Day 3 of the ACCU Conference (for me at least) was Schalk Cronjé's talk on 'Generative Programming'. Schalk has run a sequence of sessions at past ACCU conferences and this session focussed on the real problems of applying Generative Programming. His focus on doing things with today's tools and on practical problem solving was a welcome change. According to Schalk, C++ Generative programming is a vastly untapped field.

We believe in not crossing bridges until we come to them (in case you don't have to)

Schalk began by noting that many engineers build components with little reuse potential. This leads to excessive time when adapting the software for another system. Sometimes when developing there may be artefacts that you can reuse or generate to save time later. You just need to think about this when starting out.

One question he is commonly asked is – 'Is it too difficult?'. His answer is that this isn't the issue; the problem is usually motivation. If the semantics are correct then the concepts are straightforward. An audience member noted that the organisation must value code reuse in order to implement these approaches.

As an example of creating generative library Schalk described the process of building a rule evaluator: Web and mail protocols are very different but there is an essential need to provide a custom, optimised evaluator for different protocols. I'm hoping that Schalk will write up his talk for the Code Generation Network so I'll miss out the technical details here and focus on some of his general comments.

Schalk began by noting that Generative C++ can lead to a large number of configuration classes or meta-classes being generated – it is not uncommon to work with 50+ tag classes within a single type list. Effective integration of meta-programming, compile-time paradigms and run-time paradigms is required. Typical problems if done wrongly in C++ are code bloat or long symbolic names. In some cases you can't always use a pure template-driven approach.

Schalk uses the Boost libraries for meta-programming as he feels this is the only practical C++ meta-programming environment and all his examples made use of Boost. In particular the Boost metaprogramming preprocessor has a number of macros to deal with the complexity of generated C++ (e.g. commas between typenames in templates). Use of Boost is also in line with Schalk's 'Principle of least surprise' – stick to what other people have been doing especially in a pretty new field.

After taking us through 5 idioms for C++ Template Metaprogramming Schalk closed his session with a number of observations:

- Generative Programming pushes C++ skills beyond the knowledge of existing practitioners. This explains the slow uptake of such techniques and also raises the question of how to recruit people with these skills?
- Compiler time reduction is a big issue – his current builds run at around 45 minutes – and what happens if you hit a compiler error after this amount of time? Some compilers also can't deal with the complexity of some of the generated programs.
- Introducing a new technology can be political:
  - Some people will have a mental block about the technology
  - There is a distrust of tools
- Generative Programming teaches very good skills outside the C++ domain Separation of generative stuff from non-generated stuff
- Am I in the wrong language? (Schalk's answer – Yes)
  - The problem is that languages aren't good enough to solve these complex problems.
  - Some languages are easier to generate in than others. An audience member – not me – noted that 'All languages will become Lisp eventually.'
- Generative Programming in C++ shows requirements for future languages:
  - Powerful syntax
  - Simplicity of expression

Mobile convergence – it's nice because there's so much of it...

I took part in Charles Weir's 'Supporting Many Platforms' session at the ACCU conference yesterday.

Charles heads up Penrillian, a company specialising in porting applications between mobile platforms and between different implementation languages.

The first question was 'why port? why not just start again from scratch?'. The answer is that rewriting from scratch is in most cases very expensive, it's almost impossible to reimplement from a specification (or rather to rewrite the specification from what you've implemented). Furthermore, while you're rewriting you're not advancing and so you lose time in which other software may be moving forward in the marketplace. (Internet Explorer overtook Netscape while Netscape was being rewritten.) In a Software Product Line setting you have to produce so many product variants that it's simply not practical to rewrite from scratch each time.

Charles presented three key techniques to perform the porting process:

- **Code Triage** – Begin by dividing the codebase to be ported into 'portable code', 'potentially portable code' (which would be portable if it was refactored) and 'platform-specific code' (examples of this are typically found where the code talks to the outside world – e.g. device hardware, networks, UIs etc.). Charles also noted that this triage process implies some architectural decisions to minimize the size of the non-portable code.
- **Refactoring to produce portable code** – Much code could be portable if it was refactored – business logic and rendering code are prime examples. Typical issues Penrillian have encountered include UI event handlers that do business logic, rendering code for a fixed-size screen, communication interfaces that aren't abstracted.
- **Test-driven porting** – Changing code introduces bugs and so you need a safety net. The process used at Penrillian starts by ensuring that the code is under test on the original platform. When this has been achieved logging is used to record calls and responses at the points where the program calls non-portable code. This logging data is then used to generate mock objects to mock-out code for your new platform. Once you have this in place you can begin your reimplementation of platform-specific code safe in the knowledge that your safety harness will catch any problems in your new implementation.

The First (Unofficial) Annual Boat Race

Kevlin Henney <kevlin@curbralan.com>

Tension had been building throughout the conference, and a final showdown on Friday night was inevitable. Tim 'Hobbit Feet' Penhey and Pete 'Foot Craft' Goodliffe had been competing at one event or other all week. Tim 'emacs' Penhey had trounced Pete 'vi' Goodliffe three times: downing a pint the fastest on Wednesday night, the Xbox competition on Thursday evening, and the vi-versus-emacs squash tournament on Friday before the conference dinner.

And so it came to pass that the Annual Boat Race appears to have been instituted. Tim 'Massed Armies of Middle Earth' Penhey and Pete '10,000 Monkeys' Goodliffe captained teams of sturdy, hard-drinking geeks or, failing that, anyone who happened to be around and was foolish enough to say 'yes' (which is how this reporter came to be embedded in the event).

In the spirit of all good grudge matches, a theme was chosen, and what could be more traditional than a language war? Pete 'C++' Goodliffe compiled the static languages team and Tim 'Python' Penhey scripted the dynamic languages team.

The rules were quite simple but most of the participants proved to be simpler, so much explanation and re-explanation was needed. Two teams of eight lined up on opposite sides of a table, with one pint of beer in front of each competitor. Under starter's orders – Allan Kelly quickly adding this position of responsibility to his CV – the first drinker downs a pint and then, relay style, each successive drinker knocks back a pint, starting only once the previous drinker in line has completed, with completion indicated by placing the empty glass upturned on one's head and then slamming (but not smashing) it against the table. The team captains were the last in line and would sprint for the finish. The crowd surrounding the table were variously bemused, amused or attempting to express the algorithm in STL.

I ended up as the pole-position drinker on the dynamic languages team, where I was hoping that the cost of my average performance would be amortised by better drinkers further down the line. And this almost happened! The dynamic languages team gave a pretty consistent performance, and were even ahead by a person/pint at one point. However, the static languages team managed to overcome their leaks and put in an impressively optimised performance towards the end of their pipeline, ultimately handing victory to their team captain, Pete 'I've finally beaten Tim at something!' Goodliffe.

There were a number of performances worthy of note. A key stage in the static team's pipeline optimisation was Richard Harris, who seemingly teleported his beer from glass to stomach without apparently having it pass through any point in between. Richard admitted later that he was a bit of a ringer as he used to race pints (which I'm presuming is harder than racing pigeons due to questions of spillage, mobility and evaporation, amongst other issues). Another key moment for that team was Dietmar Kuehl's speed draining of his glass. This was all the more impressive because Dietmar does not normally drink: he put his speed down to his dislike of beer, so he wanted to get it over with as quickly as possible. All this meant that Pete started his pint before Tim. In a French-Connection-eat-your-heart-out high-speed chase, Tim almost caught up, losing only by a fraction of a second. At the other end of the scale on the static languages team, was Jason McGuiness's apparent desire to represent COBOL whilst all around him were emulating the performance of C, C++ and Fortran. Jason enjoyed a leisurely pint, which would have been more understandable had the beer been better.

Of course, this event was about speed. Endurance and distance awards for the conference go, without question, to John Lakos. But that's another story or ten. The conclusion is that Tim 'Eat my shorts' Penhey can still drink faster than Pete 'Attend my keynote' Goodliffe, but that team events are more fun and offer more opportunities for levelling the playing field (as well as the players). The scene is now set for the Second Annual Boat Race!

Charles concluded by noting the two worst mistakes when porting:

- Thinking something isn't needed when you port – it usually will be.
- Not using testing to support your construction process.

At the end of the talk I asked everyone what his or her thoughts were on device convergence in the mobile space. I've participated in a few of the sessions run by Cambridge Wireless's Mobile Games group and these always seem to end with the hope for greater convergence in the device space. As I've previously noted, 50% of the development budget for a mobile game can be consumed by porting costs.

The first observation was that convergence is nice because there's so much of it. More seriously it was also observed that vendor lock-in relies on divergence and it's my own view that the pressures operating in this market will force continued fragmentation in device platforms and capabilities. The forces for convergence are simply not strong enough yet...

Penrillian has a number of links to mobile porting resources on their web site at http://www.penrillian.com/porting.

Nicola Musatti <Nicola.Musatti@fastwebnet.it>

Semantic Programming – Ric Parkin

Ric's presentation wasn't focussed on introducing revolutionary techniques, but rather on reminding his audience to take advantage of tried and tested ones. The heart of it was the idea that by defining new, specific types for our function parameters we can take advantage of the compiler's help in spotting common mistakes. Ric started off with the following example:

```
memset( &buffer, sizeof(buffer), 0 );
```

Here the filler character and the buffer size have been swapped. By wrapping these parameters in specific classes such as:

```
class BufferSize {
  public:
    explicit BufferSize(std::size_t s) : siz(s)
{}
    operator size_t() { return siz; }
  private:
    std::size_t siz;
};
```

and declaring **memset** as:

```
void *memset(void *s, FillerChar c, BufferSize n);
```

we make it necessary to specify the argument type exlicitly as in

```
memset( &buffer, FillerChar(0),
   BufferSize(sizeof(buffer)) );
```

so that the compiler will issue an error message if we get the argument order wrong. This could be further improved by combining the buffer pointer and the buffer size in a single class. In some cases even enums are enough to improve arguments' type safety.

All in all, Ric's was an interesting and well delivered presentation, full of sound common sense. Note that I made these examples up as I don't have Ric's notes and so I can't present the original ones. They should be very close, though.

Supporting Many Mobile Platforms: Making Your Killer App Dominate the Mobile World – Charles Weir

This was another low rocket science, high common sense presentation. Despite its title, the topics Charles presented were of wider application than the mobile world. His approach to porting is very pragmatic: you start by examining the code ("triaging" in Charles's terms) to distinguish the parts that are presumably directly portable, from those that could be made portable with relatively little effort, e.g. by reordering and those that would require a full rewrite. Then you write or extend your test suite, to ensure that your refactoring doesn't break anything. All this takes place on the original platform. At this you factor out the portions of code that are inherently non portable and replace them with mock objects that simulate

just enough of the non portable code to let you run your tests. At this point you are ready to port your code to the destination platform and fix what breaks, basing on your test results, which should by now run unmodified on the source and destination platform When you have the portable code in place you're ready to proceed with the rewrite of the non-portable parts. This approach helps you avoid what Charles described as one of the worst possible mistakes, i.e. the full rewrite. According to Charles (and not only him!) by restarting from scratch you risk losing all that subtle knowledge that invariably gets embedded into the code as an application evolves and you tie all your resources in an all or nothing venture, effectively making it impossible for you to respond to your competition's moves. Another interesting session, pleasantly delivered.

There is a world of difference between active participation and disruption

Mark Easterbrook <mark@easterbrook.org.uk>

This was the year the ACCU went into space – figuratively: Rockets, computers in space, and even a spaceman in the form of Mark Shuttleworth featured. This year was also a new venue and I saw none of the problems that marked previous out-of-town venues, mainly due to a fleet of taxis ferrying delegates between the hotel and Oxford town centre. Apparently the organisers are considering a shuttle bus service for next year, which will make things even easier. There was just one negative comment, but a big one, the hotel just didn't seem to be able to get food to the delegates at lunchtime. Some wag even mentioned going back to the Randalf for the food! ACCU2007 also seemed to be much more Anglo-German centric than previous years: the lack of a Python track reduced the European presence and the North Americans seemed to be more subdued than usual – were they perhaps outclassed by a certain New Zealander, or have most of them been to Oxford so many times they are becoming Anglicised? There were several new faces presenting this year, but none made their presence felt than John Lakos who not only tried to buy the bar dry but introduced a new verb into the ACCU language: "To be Lakosed". Yet again the conference committee have done an excellent job and I look forward to next year.

Semantic Programming – Ric Parkin

Ric starts by using `memset(&buffer, sizeof(buffer), 0)` to show that although compilers are good at checking syntax they fail to spot semantic problems even in strongly typed languages unless the programmer encodes semantic information into types. The presentation covered ways to do this and discussed the extra work required compared with the gain in correctness of the code. This was a useful presentation for anyone working with C, C++ or similar languages.

C++ has no useful purpose – Russel Winder

The provocative title resulted in a packed room for this discussion of typed languages versus dynamic languages, with occasional diversion to traditional "scripting" and functional languages. Russel defused the audience by stating the aim of the 90 minutes was to decide if the title should end with a question mark or exclamation mark, and thus the audience decides. A clever ploy as this means the title is either a statement or a loaded question and thus biased against C++. Languages such as Python will always win out over heavyweight languages when given small problems such as will fit onto a presentation slide and thus the type of problem Russel used to try and prove his point. Despite the agenda, many an interesting 90 minutes raising many serious points about all types of languages.

Choose your Poison: Exceptions or Error Codes? – Andrei Alexandrescu

The choice of exceptions or error codes is not only difficult, but it is a design time decision meaning that it needs to be resolved early (early binding), often before enough information is available to make the right decision. After comparing and contrasting error codes and exceptions, Andrei introduced a technique that allowed the choice to be deferred. Especially powerful was the way in which the decision on how to handle an error in library code could be made by the library user.

Introduction to Component-Level Testing – John Lakos

Although an "introduction" talk, this was introduction at maximum velocity and covered an amazing amount of material in two 90 minute sessions. John started with highlighting the importance of component testing in large systems and showed how the wrong dependencies can undermine the whole testing system. This was followed by illustrations as to why white-box testing is essential and that why the implementation should drive the test data. He ended looking at test coverage: given that it is not possible to test everything, how should testing be ordered to maximum coverage whilst minimising test cases.

The appliance of science... – Andrei Alexandrescu

Andrei covered four subjects from computer science that may be useful in production programming. Dynamic Programming is about optimisation by turning exponential solutions into polynomial and has practical uses in searching. Then he looked at the downsides of Garbage Collection if added to a language such as C++, which means a loss of performance if extra memory is not available. The third subject was Machine Learning for when the function is not known and has to be determined by training. Finally Transaction Memory which applies atomic and roll-back semantics at the memory-CPU interface.

Network Programming – Alan Lenton

This was a look at the lower levels of network programming and evolved into an interesting discussion group showing that despite libraries that are supposed to take the hard work out of network there is an interest in the details. Alan is a games developer so there was a particular bias towards the special needs of network games, but nevertheless useful information for anyone moving away from HTTP and similar well known uses of networks.

Better Bug Hunting – Roger Orr

At last year's conference Roger told us how to put bugs into our programs. This talk was about finding them. This was a well presented talk on an important subject that was both entertaining and practical. I think even the most experienced bug hunter went away with something new. At least for those who made the mistake of being elsewhere, Roger provides excellent notes.

Chris Southern <cdsouthern@theiet.org>

As ever, there is room for imrovement, and Chris wasn't universally happy with the event.

Professionalism in Participation

As an old timer, and having attended a few other conferences and training courses, I appreciate the audience engagement I associate with the ACCU conference. There is nothing quite like the dialogue between a speaker and the relevant member of the ISO panel sitting in the front (or back) row. Back row is good, then they have to be loud enough for all to hear, and less experienced.

Now we come to the grumpy old programmer bits.

There is a world of difference between active participation and disruption. I heard at least two complaints, and of course I have my own, we also had disruptive non-participation.

Why should we care? The conference must not be considered unprofessional. Companies will not send staff, sponsors would leave. The financial viability of the conference would be compromised. On an individual level one might rather have the speaker at least get to even if not through the interesting looking last third of their talk.

What should we not do? Lose the essential nature of the conference, vague as such a concept might be. Expect everybody to wear suits. Expect nobody to wear a suit. Enforce a vow of silence.

What can we do? Strive for active participation rather than disruption of course. Keeping the banter for the bar and the coffee breaks.

Active participation:

- Asking for clarification of a difficult point
- Observing that something is illegal or undefined
- Pointing out a missing inversion

Disruption:

- Anticipating the thrust
- Attempting to demonstrate that you understand the slide
- Pointing out a missing brace or parenthesis
- Attempting to reverse the speaker's position mid-presentation
- Me-too war stories

Now I am quite aware that this is more of a continuum than a clear divide. However, I think that there are things we all can do to steer our behaviour towards participation and away from the disruptive. We must recognize that humour on the part of speakers is part of a good speaker's technique and not an invitation to tell our own jokes. We should wait a few slides, or for an obvious signal of a change in topic before asking about a missing implementation detail or if a missing concept had been considered. If one is not willing to show non-comprehension, do not ask the question.

The last point deserves a few more words. It is tempting to frame a question in a leading fashion, even if you are not actually guilty of the sin of attempting to demonstrate your intelligence. After all, if you did understand it, you look less of a fool. But you could be mistaken for one of those awful sinners, or introduce an erroneous and possibly confusing explanation. Now the speaker has to do twice as much work. First they have to leave their own frame of reference to understand your statement and then produce the correct explanation. A good speaker would have produced the original restated and clarified given a much simpler question. We have lost three ways. The question took too long to ask, was probably more difficult to answer and other listeners may now have the incorrect explanation mixed into their understanding of the talk.

If you feel so strongly that a speaker needs to reverse their position, how about an article for C Vu?

Last Grump. Disruptive non-participation was my bugbear. The small rooms were frequently crowded and having chairs occupied by attendees who were coding, surfing or using email struck me as plain rude. I am not saying that people misjudged the presentation level and rather than leave started to doodle electronically and quietly. No, some went equipped, and typed hard and loud.

David Carter-Hitchin <david@carter-hitchin.clara.co.uk>

I enjoyed the conference this year immensely, much more than I thought I would. I came to the pre-conference day to learn about Agile development with Kevlin Henney which was really useful. Kevlin adopted the Test Driven Development (TDD) approach which was a revelation to me and something I will definitely use in my working day. I've only been to one other ACCU conference before so I haven't got much to compare it with like-for-like, but it was by far the best conference I'd been to anywhere. There is a real sense of community in the ACCU and it was nice to see some old familiar faces. I thought the talks were really good – they started off a bit slowly to begin with, but towards the end I was hard pressed to choose between two or three really interesting ones. I chose to stick mainly with the C++0x track, as I thought it wise to start getting up to speed with changes that will hit us in the next few years. Alan Griffith's open talk on C++ tools was very interesting and useful too. John Lakos's talk on testing was brilliant. I enjoyed some of the keynotes – especially Pete Goodliffe's as it is a rare thing to hear anyone talk about the 'human side' of programming. There were a lot of big names this year, largely due to the C++ ISO committee which was convening the week after the conference, so that made things extra special. I was chatting to PJ Plauger at one time – it was only afterwards that I realised he wrote a book with Brian Kernighan and is president of Dinkumware! For anyone who didn't go, who perhaps found the cost prohibitive, I would say to them that the price is definitely worth it, and plenty of people stayed in cheaper accommodation nearby to help with the costs. For anyone who seriously can't afford it, but really wants to come then there are discounts available if you are prepared to do a talk...

Looking forward to ACCU 2008 already.

Conclusion

Phew! There you have it – a view of the ACCU 2007 conference from the floor. As ever, our thanks go to the organisers (the conference committee, the speakers, and Julie and her team at Archer Yates for organising it so smoothly).

We hope to see you there next year! ■

Standards Report

Lois Goldthwaite brings news from the C standard committee.

The international C++ and C standards committees met in Oxford and London in April, each for a week, and both enjoyed productive, successful meetings. The UK was represented by four delegates at the C committee (WG14), and by what must be a record thirteen at the C++ committee (WG21). The Force is strong in us!

WG21 buckled down to the task of putting together a Final Committee Draft document for international balloting at the end of the year. We are at the hard graft stage of converting lots of wonderful ideas into precisely-worded specifications. You can see the newly-revised Working Draft at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2284.pdf.

Among the new items: a comprehensive reworking of all the standard library clauses to embrace the techniques known as 'rvalue references' and 'move semantics'. In a nutshell, these techniques are used to provide a significant speedup of code in circumstances which previously required making a full copy of a short-lived variable, such as a temporary or function return value. Now instead of copying, the library code simply extracts the internal contents of the temporary and inserts them directly into the variable which will be around a while. The same techniques can of course be used in your own code, once C++0x compilers become available.

Through a **`<system_error>`** header, C++0x will provide better integration with diagnostics reported by operating system APIs (for example, **`ECONNREFUSED`** and **`ECONNRESET`** relating to sockets).

A new language feature (and one which caused a second set of updates to the library clauses) is variadic templates. These use an ellipsis syntax to indicate that an arbitrary number of template arguments can be used:

```
template <class... Types> class tuple;
```

This removes the need to write a large number of overloaded templates with different numbers of template parameters, as at present.

The Oxford meeting also voted to seek ISO approval of a new standard for the mathematical special functions which were published in TR 19768, the 'Technical Report on Library Extensions for C++' (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf). These provide C and C++ bindings for the functions defined in ISO 31-11, 'Mathematical signs and symbols for use in the physical sciences and technology'. The other bits of TR 19768 have been incorporated into the standard library for C++0x, but the package of special maths functions was considered too large and too specialised to be imposed on every implementation. On the other hand, there is a sizeable scientific and engineering community interested in using these functions, free qood-quality implementations exist, and there was no wish to let the work which has gone into this document be lost. Moving this library into a stand-alone international standard of its own may set a precedent on how standard C++ can expand to serve other communities in future.

Several times Bjarne Stroustrup urged WG21 to avoid making timid decisions about the development of C++. 'If we had not taken the bold step into the unknown in 1995 of adopting STL,' he said, 'I bet half of us would not be in this room today.' His exhortations persuaded the committee to incorporate several useful libraries into C++0x which otherwise would have been delayed until a future Technical Report.

LOIS GOLDTHWAITE

Lois has been a professional programmer for over 20 years. She is convenor of the C++ and Posix standards panels at BSI. One of her hobbies is representing the UK at international standards meetings! Lois can be contacted at standards@accu.org.uk

Not only is C++ evolving, but we can also expect to see a new version of the C standard in a few years' time. WG14 spent a whole day in London discussing what C99 got right and where it fell short, and establishing priorities and principles to guide the work.

Some C99 features, such as the **`bool`** type, // comments, expanded compiler limits, and variable declarations anywhere in a block, have been widely adopted by vendors and users. Takeup of other aspects, in particular the new types and functions in support of numerical programming, has been disappointing. Several vendors commented that they have not put much effort into implementing these because their customers have not demanded them. If it were not that Posix compliance requires a C99 compiler, there would be even less interest. This focus on numerical programming came from the perception in the late 1990s that C's main competitor was Fortran, and the language had to become more like Fortran to serve the needs of its community. Today there are other languages in competition with C, and other programming problems are at the forefront of attention.

Unusually for a standards body, there was a general willingness in WG14 to prune some current features from C1x, as well as add new ones to address new problems that have become prominent since C was invented over 30 years ago. Topping the list of those new concerns are threading, dynamic libraries, and above all security. Embedded environments are a major market for the C language. Improving portability of code across compilers and chip architectures would provide a valuable service to these programmers.

Philosophically, WG14 showed no inclination to follow WG21's example and be bold in expanding the C language. Their traditional conservative emphasis on standardising only existing practice with implementation experience will continue.

Before embarking on the C99 revision, WG14 adopted a charter (http://www.open-std.org/jtc1/sc22/wg14/www/charter) reaffirming the fundamental principles of the original drafting committee. Briefly:

1. Existing code is important, existing implementations are not.
2. C code can be portable, and C compilers have been implemented for a wide variety of computers and operating systems.
3. C code can be non-portable; the ability to write machine-specific code is one of the strengths of C.
4. Avoid 'quiet changes'.
5. A standard is a treaty between implementor and programmer.
6. Keep the traditional spirit of C:
   a) Trust the programmer.
   b) Don't prevent the programmer from doing what needs to be done.
   c) Keep the language small and simple.
   d) Provide only one way to do an operation.
   e) Make it fast, even if it is not guaranteed to be portable.

In these days when security holes are discovered and exploited every day, people look a bit askance at 6a and 6b. Maybe 'trust but verify' should be the new motto. Or 'trust those who are trustworthy'. Or even just 'a language should not be accident-prone', or perhaps 'the language should be verifiable'. Twenty, or even ten, years ago you didn't have to worry about Russian gangsters hacking into your toaster and turning it into a spam bot. On the other hand, maybe we should just recognise that C is the way it is and not handicap programmers too much. ■

Code Critique Competition 46

Set and collated by Roger Orr.

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 am building a random sentence generator which will construct a sentence from four arrays containing verbs, nouns, etc. The sentence is built by using a random index for each of the arrays. There is one slight problem – the three calls in my test program produce exactly same value! If I run the program again I get a different sentence, but again repeated three times. Of course, I want three different sentences within the *same* run.

I've tried to follow the code through with a debugger, but it **does** produce different sentences when I single step through the code. Can anyone help me out?'

```
#include <string>
#include <vector>

class RandomSentence
{

public:
  RandomSentence() { sentence.resize(6); }
  void createRandomSentence();
  std::vector<std::string> & getSentence()
  { return sentence; }

private:
  std::vector<std::string> sentence;
  static std::string article[5];
  static std::string noun[5];
  static std::string verb[5];
  static std::string preposition[5];
};

#include <time.h>

void RandomSentence::createRandomSentence(){

  int randNum;

  for(int i = 0; i <= 5; i++) {
    srand(time(0));
    randNum = (rand()%5);
    switch(i){
      case 0:
        sentence[i] = article[randNum];
        break;

      case 1:
        sentence[i] = noun[randNum];
        break;

      case 2:
        sentence[i] = verb[randNum];
        break;

      case 3:
        sentence[i] = preposition[randNum];
        break;

      case 4:
        sentence[i] = article[randNum];
        break;

      case 5:
        sentence[i] = noun[randNum];
        break;
    }
  }
}

std::string RandomSentence::article[5] =
  {"the", "a", "my", "your", "his"};
std::string RandomSentence::noun[5] =
  {"pig", "cup", "phone", "TV", "letter"};
std::string RandomSentence::verb[5] =
  {"ate", "sat", "flew", "ran", "lay"};
std::string RandomSentence::preposition[5] =
  {"by", "in", "with", "over", "on"};

#include <iostream>

int main() {
  RandomSentence rsc;
  for ( int i = 0; i < 3; i++ ) {
    rsc.createRandomSentence();
    for ( int j = 0; j != 6; j++ ) {
      std::cout << rsc.getSentence()[j];
      if ( j == 5 ) std::cout << std::endl;
      else std::cout << " ";
    }
  }
}
```

Critiques

You will notice that we have a lot of critiques this issue – many thanks to all the readers who put finger to keyboard! All the entrants identified the main problem correctly, so in an attempt to reduce repetition I have 'refactored' the solutions to extract the common analysis, and then left under each author's name the additional points they raised.

The initial problem (from Jim Hague's entry)

Programs that work perfectly under the debugger but fail when run normally are one of those distressing facts of programming life. The introduction of the debugger into the environment can cause many subtle changes in the program execution environment, any of which may make the bug not go away, but change symptoms.

ROGER ORR

Roger has been programming for 20 years, most recently in C++ and Java for various investment banks in Canary Wharf. He joined ACCU in 1999 and the BSI C++ panel in 2002.

He may be contacted at rogero@howzatt.demon.co.uk

Before charging blithely onwards to other debugging techniques, first apply the most important debugging tool available to you; think carefully about what is going wrong, and see if you can come up with a way you could intentionally make the program behave that way. In this case, the program is outputting the same sentence three times on each run. Did you run the program more than once? If so, you'll see that the sentence does change between runs. And you also know that if you single-step through the program the sentence will change within a run. Now, for the same sentence to be output, the sentence generator must be using the same random numbers each time. But somehow the sequence of random numbers the program is getting does change over time. So a close look at code associated with getting random numbers is a possible line of enquiry, particularly there is anything time-related there.

So Suspect No. 1 must be the call `srand(time(0))`. Checking the documentation for `srand()`, you'll find it sets the seed used by `rand()`. This isn't the place for a full discussion of randomness – for that, I'll refer you to any good algorithms text. Briefly, `rand()` works by taking a seed value, permuting it, returning the permuted value and keeping that value as its seed for the next call. If you don't call `srand()`, or call `srand()` at the start of the program with a fixed value, then `rand()` will return the same sequence of random numbers on each run of the program. This can be rather useful in debugging.

So the immediate cause of the problem is that the random seed is being reset to the value returned by `time()` on each call to `createRandomSentence()`. That value will only change once per second. Calling `srand(time(0))` is a common way of setting the random seed to a different value on each program run, but should happen only once, typically at program startup.

Headers (from David Carter-Hitchin's entry)

The placement of `#include <iostream>` above `main()` is rather odd, since it would prevent `cout` being used in the class (if that was required). Generally header includes are best clumped together right at the top of source files. The same goes for `<time.h>` – this should be at the top and it should also be `<ctime>` as `time.h` is the C header, not the C++ one.

[Ed: Having got headers out of the way, I've omitted them from the entries below to save space.]

From Reg. Charney <charney@charneyday.com>

To fix the problem, you need to change the seed value for the pseudo random number generator. The easiest way is to use a high resolution timer, if one is available (perhaps something that works on internal clock 'ticks'). This may be implementation defined. However, this means only one or two lines need to change.

Another way of doing this is to move the call to `time()` outside the loop, save the value, and use an incremented value as the seed. Thus,

```
for(int i = 0; i <= 5; i++) {
    srand(time(0));
    randNum = (rand()%5);
```

becomes:

```
time_t curTime; // place to store current time
time(&curTime); // get the current time in
                // seconds and store it in curTime
++sentenceNum;  // increment sentence number
                // (see below)
for (int i = 0; i<= 5; i++) {
  srand(curTime+sentenceNum+i); // guaranteed to
                // change the seed (see note below)
  randNum = (rand()%5);
```

As you can see, this solution depends on the sentence number. This is easy to fix in this application. In the definition of the `RandomSentence` class make one basic change: add the following static declaration to the private part of the class:

```
static unsigned int sentenceNum = 0;
```

(for older compilers the initialization may need to be separated out)

From Ian Bruntlett <ianbruntlett@hotmail.com>

After referring to my copy of *C in a Nutshell* for information about `rand()` and `srand()` I modified the method `RandomSentence::createRandomSentence()` so that the call to `srand()` could be moved into `main()`. `srand()` guarantees 'For each value of the seed passed to `srand()`, subsequent calls to `rand()` yield the same sequence of 'random' numbers'.

I modified `main()` to call `srand()` with the seed value of `time(0)`. Without that, every invocation of this program would behave as if `srand(1)` had been called resulting in the program printing the same strings time and time again.

```
class RandomSentence {

public:
  RandomSentence() { sentence.resize(6); }
  void createRandomSentence();
  std::vector<std::string> & getSentence()
  { return sentence; }

private:
  std::vector<std::string> sentence;
  static std::string article[5];
  static std::string noun[5];
  static std::string verb[5];
  static std::string preposition[5];
};

void RandomSentence::createRandomSentence(){

  int randNum;

  for(int i = 0; i <= 5; i++) {
// IRB:- This is the smoking gun :)
//    srand(time(0));
//     std::cout << "Would call srand "
//       << time(0) << std::endl;
    randNum = (rand()%5);
    switch(i){
    // rest of function as before
    }
  }
}

// article, noun, verb, prepositions as before

int main() {
  srand(time(0));
  std::cout << "main : does call srand " <<
    time(0) << std::endl;

  RandomSentence rsc;
  for ( int i = 0; i < 3; i++ ) {
    rsc.createRandomSentence();
    for ( int j = 0; j != 6; j++ ) {
      std::cout << rsc.getSentence()[j];
      if ( j == 5 ) std::cout << std::endl;
      else std::cout << " ";
    }
  }
}
```

From Silas S. Brown <ssb22@cam.ac.uk>

In this application, you could just move `srand()` out of the loop. Put it a line earlier than where it is, or even put it at the beginning of `main()`. In applications where you need more genuine randomness (i.e. cryptography) then you will need to do something stronger (e.g. /dev/random or OpenSSL libraries) but that is beyond our scope.

Other hints about the code:

1. The `getSentence()` method is pointless as it is. You might as well make `sentence` a public member, because `getSentence()` offers no constraints like `const` to justify having the method. I recommend writing `const std::vector<std::string> & getSentence() const` instead (if you don't know what `const` means and why it's used then please read about it).
2. There are rather a lot of number 5's in the code. If this changes then you'll have an awkward time. You could write something like:

   ```
   enum { NumWordsOfEachType = 5};
   ```

   and then use `NumWordsOfEachType` everywhere, but it might be better to support different lengths for the different arrays – why not make them all vectors and call `rand()` modulus the length of the vector? (see STL documentation on vectors for how to get this)
3. Related to this, you don't really need that switch. You could have an array of references to vectors, i.e.

```
const std::vector<std::string>& sentencePattern[] =
{article, noun, verb, preposition, article, noun};
```

   Then for each element of `sentencePattern`, dereference it with `[]` to get the vector of words from which you can pick a random element.

   Doing it this way makes it much, much easier to change things: not only can you have different numbers of each type of word (most languages have a lot more nouns than determiners for example) but also you can quite easily change the sentence pattern. You could even make `sentencePattern` a vector, and you can (if you want) have several different sentence patterns that are picked at random. All of that would be rather difficult with the switch. (If you really feel like doing something advanced, try generating sentences recursively using clauses and noun phrases and so on, but I won't go into that here.)

From Jonas Hammarberg <hammarberg@computer.org>

To get it working you just need to move the initialisation of the random generator. Preferable into `main()`, before the line `RandomSentence rsc`. Alternatively you can place it in the constructor of `RandomSentence`. The latter approach has a drawback as the generator might be initialized several times by different constructors but in your very specific case it would work.

A point of critique would be your naming of the class. Honestly, does it really produce a sentence (as you need to do quite a lot of work to print it out)? Another point would be the knowledge needed in `main` to extract each word – you have to know the exact number of words.

I took the liberty of writing an alternative solution. It's of course done 'my way' but please note with what ease you can extract a random sentence, just get it. ;)

If we look at the method for getting a sentence (I merged `get` and `create`) we see that I have gotten rid of the `switch`-statement. A `switch` does have its place, but it shouldn't be your first answer to a problem.

You might also notice that I've no 'magic numbers'. Actually, you can have different number of samples for each atom. Yes, I'm using macros as in this case it removes a number of possible typing errors (misspelling, mixing arrays etc).

Now I need to take the dogs for a walk so I'll have to leave the rest of the analysis in your hands. Please also remember that most sentences start with a capital letter. ;)

```
char* articles[] =
  {"the", "a", "my", "your", "his"};
char* nouns[] =
  {"pig", "cup", "phone", "TV", "letter"};
char* verbs[] =
  {"ate", "sat", "flew", "ran", "lay"};
char* prepositions[] =
  {"by", "in", "with", "over", "on"};

class RandomSentence {
public:
  RandomSentence() {
#define NUMBER_OF_ELEMENTS(a) \
  (sizeof(a) / sizeof(a[0]))
#define COPYSAMPLE(a) \
 stringvector(a, a + NUMBER_OF_ELEMENTS(a))
    articles_ = COPYSAMPLE(articles);
    nouns_ = COPYSAMPLE(nouns);
    verbs_ = COPYSAMPLE(verbs);
    prepositions_ = COPYSAMPLE(prepositions);
#undef COPYSAMPLE
#undef NUMBER_OF_ELEMENTS
  }

  std::string getSentence() {
    std::stringstream ss;
    ss << capitalize(article()) << " "
       << noun() << " " << verb() << " "
       << preposition() << " " << article()
       << " " << noun() << ".";
    return ss.str();
  }

private:
  typedef std::vector<std::string> stringvector;

  stringvector articles_;
  stringvector nouns_;
  stringvector verbs_;
  stringvector prepositions_;

  std::string capitalize(const std::string& s)
  { return s.empty()? s : std::string(1,
      std::toupper(s[0])) + s.substr(1); }

  size_t rnd(const size_t range)
   { return rand() % range; }

  std::string word(const stringvector& a)
  { return a.empty()? "" : a[rnd(a.size())]; }

  std::string noun() { return word(nouns_); }
  // ditto for articles/verbs/prepositions
};

int main(void) {
  std::srand((unsigned int)std::time(NULL));
  RandomSentence rsc;
  for(int i(0), end(3); i != end; ++i) {
    std::cout << rsc.getSentence()
      << std::endl;
  }
  return 0;
}
```

From Peter Pichler <peter.pichler@sophos.com>

On a broader scale, the code seems a bit too complicated for such a simple task. The use of OOP is superfluous for something that is essentially a procedural problem. You do not need to keep the intermediate result when you can print them right away. The use of a `switch` inside a loop increases

the line count by adding no obvious benefit. The method `getSentence()` is not named properly for what it does.

There are some style issues too. Mixing `#include` with lines of code, inconsistent brace style, declaring local variables outside the scope they are used in. Also, `createRandomSentence()` could be combined with the `RandomSentence` constructor. The function `main()` is correctly declared as `int`, but does not return a value. [Ed: this is in fact valid for `main`] Finally, there is one article/pronoun missing in each generated sentence, before the last noun.

My solution is below. I have taken the liberty of increasing the vocabulary. Like the original, it still produces mostly meaningless sentences, but it achieves it with a much simpler code. It can be simplified even further, but probably at the expense of a reduced readability by a novice programmer.

```
static const char * article[] =
  { "the", "a", "my", "your", "his", "her" };
static const char * noun[] =
  { "pig", "cup", "phone", "TV", "letter" };
static const char * verb[] =
  { "ate", "sat", "flew", "ran", "lay",
    "walked", "stray" };
static const char * preposition[] = {
 "by", "in", "with", "over", "on", "beyond" };

const char * getWord (
  const char *dictionary[], int dictSize) {
  int r = rand() % dictSize;
  return dictionary[r];
}

#define GW(x) getWord(x, sizeof x/sizeof x[0])

void printSentence () {
  std::cout << GW(article) << ' '
            << GW(noun) << ' '
            << GW(verb) << ' '
            << GW(preposition) << ' '
            << GW(article) << ' '
            << GW(noun) << std::endl;
}

int main() {
  srand((int)time(0));
  for (int i = 0; i < 3; ++i)
    printSentence();
  return 0;
}
```

From Jim Hague <jim.hague@acm.org>

Job done? That depends. Since you've been inspecting the code anyway, you should always be on the lookout for opportunities to make improvements to the code, particularly if you found the code difficult to understand for no good reason. But just before a release isn't the time to start making big changes; in this case, file a bug to remind you to come back later.

My first observation is that the class `RandomSentence` could be better named. A `RandomSentence` object isn't of itself a random sentence; it's a means of generating a random sentence. I'd rename the class `RandomSentenceGenerator`.

Next, the class interface is somewhat curious. You need to call `createRandomSentence()` to generate a sentence and then `getSentence()` to get the sentence just generated. I'd consider collapsing them into `getSentence()`, so that each call to `getSentence()` returns a new random sentence.

I'm also concerned about the return value from `getSentence()`. It returns a vector of words in the sentence. Does it really need to, or could it just return a single string containing the sentence? That's what I would immediately expect from a method called `getSentence()`. If you really do need each word individually, I'd call the method `getSentenceWords()`.

The code as it stands contains a worrying number of magic numbers, `5`'s and `6`. Bare numbers are worrying in code. They make it brittle, likely to break on a small change. For example, the code in `main()` assumes that it always gets a vector of 6 words, never more, never less. A small change to the generator would break it. In this case `6` should be replaced by the size of the vector. Raw numbers also make it harder to reason about what the code is doing. There's a particularly dangerous `5` in the `for` loop in `createRandomSentence()`; 5 is used almost exclusively to mean the number of entries in each word array, but here it's counting up to the number of words in a sentence. At the very least I'd create a constant `WORDS_IN_SENTENCE` and use that throughout the class when referring to the number of words in a generated sentence.

The code assumes that the number of articles is the same as the number of verbs, the number of nouns etc. This seems unnecessarily restrictive.

Should this generator be a class? C++ doesn't force you into the object world if it isn't appropriate. The generator class isn't doing much here; the possible words are fixed, and the whole class could be easily replaced with a single function. Or the code could be rearranged so that word lists could be supplied externally; in that case having different generator objects with created with different word lists would be an appropriate use of classes.

From Thaddaeus Frogley <tfrogley@climaxgroup.com>

The author may also consider a change from `time` (which returns a number of seconds) to the use of `clock`, which returns sub-second accurate timing (`CLOCKS_PER_SEC`), but this would most likely still not work as expected, since multiple instructions can be executed in a single `CLOCKS_PER_SEC`, and there is no guarantee that sufficient time will have passed in a single iteration of the loop for the return value of clock to have changed. Had the author done this in the first place he may have encountered the scary world of 'debug builds work, release builds do not', or even worse 'it works on my (slow) laptop, but when my client runs the program (on fast server platform) it doesn't work'.

That said, I believe it's traditional to explore design improvements in the CVu code critiques, and not just explain the bug and provide a fix.

In this case the design issue that is most closely linked to the authors bug, is the use of c library functions with hidden global state, this is to say, calls to `rand`/`srand`.

I would advise programmers who are writing classes that need a source of random data to accept parameterisation of that data source, probably by defining a random number generator base class within their framework, and having pointer or reference to it passed in as an argument to their class constructor, or perhaps via template parameterisation. This allows isolation of specific random number sources for specific uses, and enables better flexibility during debugging or unit testing, by allowing guaranteed reproducibility by for example, taking 'random' numbers from a predetermined source (say a debug-random file), whilst allowing deployed builds to use true random number sources from entropic hardware sources.

The finer details of how to implement such a scheme are, of course, left as an exercise for the reader.

From Nevin :-] Liber <nevin@eviloverlord.com>

Bias: there is a subtle issue here, in that the distribution of random numbers being used will be slightly biased. `rand()` returns a value that is uniformly distributed across [0..2^^32-1]. Because 5 is not a power of 2, `rand()%5` will bias towards the lower numbers of the range [0..4]. Removing this bias is non-trivial.

- Encapsulation: class `RandomSentence` doesn't really encapsulate much of anything useful. Its only non-static member variable is `sentence`, which is just a cache of the result of calling `RandomSentence::createRandomSentence()`. Its static data are just implementation details. This class can easily be replaced with a function.

- Coupling: `RandomSentence::createRandomSentence` 'knows' that there are only five words in each of the sentence parts (article, noun, etc.). `main()` 'knows' that the sentence structure consists of exactly six words. If any of that changes, code all over the place needs to change, too.
- Efficiency: On one hand, the C-string literals are converted to `std::string`s, for an efficiency penalty and no benefit. On the other hand, `RandomSentence::getSentence()` returns a reference to `sentence`, which can be more efficient but at high cost of exposing an implementation detail, thus breaking encapsulation.

My solution:

```
template<typename T, std::size_t N>
T* begin(T (&array)[N])
{ return array; }

template<typename T, std::size_t N>
T* end(T (&array)[N])
{ return array + N; }

template<typename T, std::size_t N>
std::size_t size(T const (&)[N])
{ return N; }

// http://graphics.stanford.edu/~seander/
// bithacks.html#RoundUpPowerOf2
unsigned powerOf2Ceil(unsigned v) {
  --v;
  v |= v >> 1;
  v |= v >> 2;
  v |= v >> 4;
  v |= v >> 8;
  v |= v >> 16;

 return ++v;
}

unsigned randCeil(unsigned v) {
  unsigned roundUpPowerOf2(powerOf2Ceil(v));
  do {
    unsigned r(rand() % roundUpPowerOf2);
    if (r < v) return r;
  }
  while (true);
}

typedef std::vector<char const*> Words;

Words createRandomSentence() {
  static char const* const article[] = {
    "the", "a", "my", "your", "his",
  };

  static char const* const noun[] = {
    "pig", "cup", "phone", "TV", "letter",
  };

  static char const* const verb[] = {
    "ate", "sat", "flew", "ran", "lay",
  };

  static char const* const preposition[] = {
    "by", "in", "with", "over", "on",
  };
  static Words articles(
    begin(article), end(article));
  static Words nouns(
    begin(noun), end(noun));
  static Words verbs(
    begin(verb), end(verb));
  static Words prepositions(
    begin(preposition), end(preposition));

  static Words const* const structure[] = {
    &articles, &nouns, &verbs,
    &prepositions, &articles, &nouns,
  };

  Words sentence;
  sentence.reserve(size(structure));

  for (Words const* const* w =
    begin(structure); w != end(structure);
    ++w) {
    Words const& part(**w);
    sentence.push_back(
      part[randCeil(part.size())]);
  }
  return sentence;
}

int main() {
  srand(time(0));

  for (int i = 0; 3 != i; ++i) {
    Words sentence(createRandomSentence());
    if (!sentence.empty()) {
      std::copy(&sentence.front(),
        &sentence.back(),
        std::ostream_iterator
        <Words::value_type>(std::cout, " "));
      std::cout << sentence.back();
    }
    std::cout << std::endl;
  }
}
```

Notes:

- `begin()`, `end()` and `size()` are simple helper functions for using arrays in ways similar to STL containers.
- `powerOf2Ceil(v)`: This function returns the smallest power of 2 not less than v (assumes 32-bit 2s complement machine with v > 0). It will be used to remove the bias in calculating the random number. (I grabbed this off the Internet; credit is in the comment.)
- `randCeil(v)`: This function removes the bias in `rand() % v`. Basically, it makes sure that the mod happens with `powerOf2Ceil(v)`, and then throws away any values greater than or equal to `v`.
- `typedef std::vector<char const*> Words`: Because all the words are generated from C-string literals, I store a sequence of words as a `std::vector<char const*>`. If some future version of this program needed to store them as `std::string`s (say, a version that read the word lists from a file), all I would have to change is this `typedef` to reflect that, and the rest of the code would still just work.
- `createRandomSentence()`: This function returns a randomly generated sentence every time it is called.
- `article`, `noun`, `verb`, `preposition`: These static arrays have unspecified sizes. That way, if one wants to change the number of words in any of those categories, all one has to do is add or remove words from the array.
- `articles`, `nouns`, `verbs`, `prepositions`: In order to allow for each of the categories to contain different numbers of words, each category of words needs to be stored in a container whose size can be queried. `Words` is one such container.

- `structure`: In the quest to be more data driven, structure describes the sentence structure. The function just iterates over structure, picks a random word from each category, and appends it to sentence.
- `srand(time(0))`: This is at the top of main, and only called once to seed the random number generator.
- `std::copy()`: This line copies all but the last word to `std::cout`, appending each word with a `" "`.

From John Penney <J.Penney@servicepower.com>

In his commentary on CC44, Roger noted that both solutions had slightly different behaviour to the original code. I reflected back to the start of my week at ACCU 07, when I attended a Test Driven Development (TDD) workshop run by Kevlin Henney, and further back to an excellent book I looked at last year, *Working Effectively with Legacy Code* by Michael Feathers. In this book Feathers (who was also at ACCU07) suggests a number of strategies for getting nasty legacy code into the comforting arms of a unit test.

So I thought I'd take a different approach to tackling CC45: can we use unit tests to find and fix the bug? Doing so would then give us a framework to carry out further non-functional refactorings suggested here and by other Code Critiquers – safe in the knowledge that we haven't changed functionality.

Making the code unit-testable

Just last week a colleague said (of our simulated annealing algorithm) "Oh, we can't unit test that – its results are random". And this is indeed a problem with the code supplied: it behaves in a random way. (Let's just ignore the fact that it's actually currently quite random and that what you actually want it to be very random! Every run gives different results, but all the sentences produced in a run are the same.) How can you unit test code that does random stuff? Well in this case we can decouple the responsibility for random-number generation from the responsibility for sentence-building. This first change we have to do without the aid of our unit test safety net, so we want to change as little as possible. The simplest change I could think of doing was to pass the six random numbers which generate sentence indices in as parameters instead of having them generated inline, so:

```
typedef int SentenceIndices[6];
void RandomSentence::createRandomSentence(
  const SentenceIndices& sentenceIndices){

    int randNum;
    for(int i = 0; i <= 5 ; i++) {
        //srand(time(0));
        //randNum = (rand()%5);
        randNum = sentenceIndices[i];
        switch(i){
```

Getting some unit tests in place!

Now we have something we can unit test. Maybe you have a unit test framework to hand… even if you don't think you do, you do! As Kevlin Henney demonstrated, plain ol' assert will do:

```
#include <cassert>
void testNormalSentenceConstruction() {
  RandomSentence rsc;
  const SentenceIndices indexSet1 =
    {0, 0, 0, 0, 0, 0};
  rsc.createRandomSentence(indexSet1);
  assert(rsc.getSentence().size() == 6);
  assert(rsc.getSentence()[0] == "the");
  assert(rsc.getSentence()[1] == "pig");
  assert(rsc.getSentence()[2] == "ate");
  assert(rsc.getSentence()[3] == "by");
  assert(rsc.getSentence()[4] == "the");
  assert(rsc.getSentence()[5] == "pig");
}
```

Now you can write a few of these tests – but there's a lot of repetition here (every character up to column 40 in fact, on later lines). Let's refactor our tests, so we end up with a rather more elegant-looking test as follows:

```
#include <cassert>
std::string buildSentence(int i1, int i2,
  int i3, int i4, int i5, int i6) {
  RandomSentence rsc;
  const SentenceIndices indices = {i1, i2, i3,
    i4, i5, i6};
  rsc.createRandomSentence(indices);
  const std::vector<std::string>& s =
    rsc.getSentence();
    return s[0] + " " + s[1] + " " + s[2]
      + " " + s[3] + " " + s[4] + " " + s[5];
}
void testNormalSentenceConstruction() {
  assert(buildSentence(0, 0, 0, 0, 0, 0) ==
    "the pig ate by the pig");
  assert(buildSentence(4, 4, 4, 4, 4, 4) ==
    "his letter lay on his letter");
  assert(buildSentence(3, 0, 1, 1, 2, 1) ==
    "your pig sat in my cup");
  assert(buildSentence(1, 2, 3, 3, 2, 0) ==
    "a phone ran over my pig");
}
```

Bug? What bug?

Now you can write lots of these tests and they'll all pass. Why? Because the code has no bug in it… we took it out in our very first refactoring! Think back to the original published version of `RandomSentence`. Being a battle-scarred veteran, my suspicious eyes were drawn to the statement `srand(time(0))`. Here's a solution that re-uses the `buildSentence()` method we wrote above to support our unit tests:

```
int main() {
  testNormalSentenceConstruction();

  srand(time(0));
  const int numSentences = 3;
  for (int sentenceNum = 0 ;
       sentenceNum < numSentences ;
       ++sentenceNum) {
    std::cout << buildSentence(
      (rand()%5),(rand()%5),(rand()%5),
      (rand()%5),(rand()%5),(rand()%5))
      << std::endl;
  }
}
```

Don't ttop (thinking about tomorrow)

We don't want to stop now, just having fixing the bug. Let's take advantage of that useful unit testing and refactor the code to make it as clean, maintainable and simple as possible. Here are some ideas:

- Rename `i` and `j` with nice names like `wordNum` etc.
- Be more scrupulous about `const`-ness. For example, make `RandomSentence::getSentence()` `const` and declare `randNum` at the point of use, and make it `const`:

  ```
  const int randNum = sentenceIndices[5];
  ```

- Think about the role of the classes: `RandomSentence` looks to me a lot more like a `RandomSentenceGenerator`
- Those C arrays of words could become STL containers which can then be of different sizes (you're currently constrained to choosing exactly 5 words in each class).

- Prefer `++i` to `i++` (aka Don't Pessimize Prematurely – use the prefix `++` operator as a matter of course)
- Add some tests to exercise the indices in `SentenceIndices` being out of bounds (<0 or >6) and then add the protection to your class.

Each of these changes is best made in small steps, running the unit tests through each time to confirm you've not broken anything.

I hope you can see that by working in this cyclic manner the code moves a long way away from the original, becoming simpler, more elegant and acquiring new functionality as it goes, yet all the time maintaining functional integrity.

From Jean-Marc Bourguet <jm@bourguet.org>

Now, we can go in either of two directions: the design of your program and the use of random numbers in C++. Even if I think that there are things to be commented in the design, I won't do it as I can come with several alternative designs and I don't know enough about the context to decide which is best.

The first thing you may want to do when using a pseudo-random number generator in a program is to inform the user of the seed used to initialize it – for example in a log file – and allowing the user to set the seed – for example with an argument to `main()`. Even if most users won't take advantage of that feature, it will enable you to debug problems which depend on the generated sequence.

Then, there are some problems when you try to reduce the range of the generated numbers. The easiest way is to use the remainder operator as you have done it. It works well enough in most cases but you need to be aware of two caveats.

The first caveat is common to whatever generator you use. If the number of cases you are interested – five here – is not a divisor of `RAND_MAX+1`, you'll get the small values generated more often than the bigger one. How much more often? That depends on the respective values, the bigger `RAND_MAX` is, the smaller is the bias. If you were using a dice – with a `RAND_MAX` of 5 – 0 would be generated twice as often as the other 4 cases. The only way to cure this is to drop some values.

The second problem is that some popular pseudo-random number generators generate numbers whose less significant bits are 'less random' than the most significant one. Using the remainder operator uses precisely those bits. In the worst case I've seen was using `rand() % 2` and get an alternating sequence of 0 and 1. Now, the library providers are aware of this and usually take some corrective measures, but you can take care of it by using a division instead of a remainder.

The code I use to reduce the range is the following:

```
int alea(int n) {
  assert (0 < n && n <= RAND_MAX);
  int partSize = n == RAND_MAX ? 1
    : 1 + (RAND_MAX-n)/(n+1);
  int firstToBeDropped = partSize * n
    + partSize;
  int draw;
  do {
    draw = rand();
  } while (draw >= firstToBeDropped);
  return draw/partSize;
}
```

It takes care of these two issues and also won't trigger overflow if `RAND_MAX` happens to be equal to `INT_MAX`.

Note that if those issues – especially the second – are important to you, you probably don't want to rely on a generator of unknown quality. The technical report on library – and boost, and probably the next standard – provides additional way to get random numbers. There are several generators using known algorithms, and interfaces to provide some standard distributions. Using them, the function above could be rewritten as this:

```
std::tr1::mt19937 generator;

int alea(int n) {
   std::tr1::uniform_int distribution(0, n);
   return distribution(generator);
}
```

From Joe Wood <joew@aleph.ndo.co.uk>

Hi, sure I'll have a look for you, but I should warn you before we begin that I'm not a C++ programmer. But it's late and the lab is nearly deserted so let's see what we can do.

On the subject of the call to `srand`, `time` should be called with a `NULL` parameter to return the current time not 0. Yes, I know and you know that `NULL` has a value of 0, but logically they are different. Also `srand` expects an unsigned parameter not a `time_t` type, so let's add a cast. Hence the call now looks like:

`srand((unsigned)time(NULL))`. You need to add `cstdlib` to your list of includes, to get access to the `NULL` macro. Incidentally, it is generally best to put all your includes at the top of file, ideally grouped by (say) system, project, local and then alphabetically. By the way, I see you are mixing old style `.h` headers with new style headers, really it is best to use the new style headers.

On the matter of file structure, as this is only an exercise its probably OK to put the entire program in a single file, but really we should put the declaration of class `RandomSentence` into one (header) file, the implementation of `RandomSentence` into one or more files as required for ease of development and our software configuration policy and finally a separate file for the main function.

Now the function `createRandomSentence` looks rather complicated. Ah, I see, you are constructing a sentence with the grammar {sentence := article noun verb proposition article noun}, and for each part of speech you have chosen one at random from their respective arrays in `RandomSentence`. But you don't need the loop and the switch statements they just obscure what is really a simple sequence. Let's create a macro called `RANDNUM` which takes an array as a parameter, then the entire for block can be replaced by

```
sentence[0] = RANDNUM(article);
sentence[1] = RANDNUM(noun);
sentence[2] = RANDNUM(verb);
sentence[3] = RANDNUM(preposition);
sentence[4] = RANDNUM(article);
sentence[5] = RANDNUM(noun);
```

which is pretty close to what our grammar looks like. Now we need to sort out the definition of `RANDNUM`. Basically it wants to be something like "Take in an array and return a random element from that array". But why have you made all the string arrays in `RandomSentence` the same size? Ah, so that you can use

```
randNum = (rand()%5);
```

in your loop. Its not a good idea to just drop 'magic' numbers into your code, use a macro instead because this eases maintenance. Now that we have got rid of the loop and introduced the `RANDNUM` macro, we don't need to limit ourselves to all the arrays being the same length, but we need to know the length of the individual arrays. Unfortunately, this is one of C++ weaknesses, arrays do not carry knowledge of their length with them unlike container classes in the STL for example `vector`. However we can get round this by defining our `RANDNUM` macro as

```
#define RANDNUM(X) \
  ::randNum(X, sizeof(X)/sizeof(X[0]))
```

and then we can create the function `randNum` as follows

```
static std::string randNum (
  std::string * choices, size_t count) {
  return *(choices+(rand()%count));
}
```

Which probably needs some explanation. The local function **randNum** takes in an array (of strings) and a length, it then returns a random element from that array. The macro **RANDNUM** calculates the length of the array as "total size of(array)/size of(first element of array)". I would put the definition of **RANDNUM** just before the function **createRandomSentence** and undefine it just after.

Now we can handle different sizes for the various parts of speech. Alternatively we could make the data member sentence into a single **std::string**, and use **RANDNUM** with the append **operator+=**. We would have to decide if a single extra space at the end of sentence was acceptable, this is not clear.

The need for the macro **RANDNUM** would be removed if arrays carried their length, or we used vectors, but I don't know now to initialise a vector from an aggregate, and using the naïve approach of inserting elements into a vector inside a loop would result in two copies of the data. Not a problem in a small program but it won't scale well.

Better yet, we could construct a vector of pairs (not a provided C++ type), something like (in pseudo C++)

```
const std::vector<pair> grammar =
  {a(article), a(noun), a(verb),
   a(proposition), a(article), a(noun)};
```

and then we could just iterate over the grammar.

It should be noted that the string returned by **RandomSentence::getSentence** allows full access to the member sentence, fixing this would require the additions of a number of **const**s, and is deemed beyond the scope of this exercise and my knowledge.

There is no need to specify the sizes of the parts of speech arrays in **RandomSentence** nor in the actual data initialisers as the compiler is much better at counting than most people, and it reduces program coupling. I would however declare the arrays as const.

Finally we can make a few small changes to **main**. In the outer **for** loop, just after the call to **createRandomSentence** we should add the line

```
std::vector<std::string> & sentence =
  rsc.getSentence();
```

which saves several unnecessary calls to **getSentence**. I think it is always a good idea to put braces around statements in for and if statements etc. because later editing is unlikely to cause nasty surprises in the execution flow. Personally, I would rewrite the inner loop as

```
size_t last_word = sentence.size()-1;

// all words bar the last are followed by a space
for ( size_t j = 0; j < last_word  ; j++ ) {
  std::cout << sentence[j] << " ";
}
// last word is followed by end of line
std::cout << sentence[last_word] << std::endl ;
```

But that's just a personal choice. It would be nice since sentence is a vector to use iterators, but I don't know how to specify the range **begin(), end()-1**.

From David Carter-Hitchin <David.Carter-Hitchin@rbos.com >

Looks like this student has fulfilled Knuth's prediction: 'Every random number generator will fail in at least one application.' (Donald Knuth, 1969).

At first sight the code looks reasonably structured and well written. Concentrating on the student's complaint first of all, the error comes to light pretty quickly. At this point I would explain to the student how pseudo-random sequences work. Namely using a seed value and by applying modular arithmetic to that seed to get the next number in the sequence. This would allow the student to understand that nature of pseudo-random numbers and the importance of resetting the random seed only once (or when appropriate). In simulation software it is often useful to set the random seed to the same value at the start of each batch of simulations, in order to get reproducible and comparable results. In the case of the random sentence generator, a different seed each time seems appropriate.

One must also consider that the sequence of such values can only ever be pseudo random. To quote one of the founding fathers of modern computing, John Von Neumann: 'Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.' (1951, 'Various Techniques used in connection with random digits' in *Monte Carlo Method*) (1)

Von Neumann worked at RAND, which spent a lot of time and money developing methods to artificially generate random numbers (besides lots of other things, like the development of Game Theory as applied to nuclear warfare).

For the program above, the choice of random number generator is insignificant, but for large scale scientific programs, for example, choices become much more critical. A good overview of methods is discussed in Chapter 7 of *Monte Carlo Methods in Finance* by Peter Jaekel, and googling for 'random number generator comparison' produces 42 million hits.

So much for pseudo randomness. The rest of the code seems a little bulky for what it does. The **switch** statement in **createRandomSentence()** seems way over the top, considering the values of **i** will be the same for each call. In other words it's equivalent to:

```
sentence[0] =     article[rand()%5];
sentence[1] =        noun[rand()%5];
sentence[2] =        verb[rand()%5];
sentence[3] = preposition[rand()%5];
sentence[4] =     article[rand()%5];
sentence[5] =        noun[rand()%5];
```

Which is much more compact and readable. Admittedly it is perhaps less flexible than a switch, which is maybe what the student had in mind.

The magic number 5 should be a constant at the top of the program, for ease of maintenance. Sentences start with capital letters and end with full stops, so these need to be incorporated.

Finally the structure of the program doesn't allow for any flexibility of random sentence structure – i.e. you always get 'article noun verb preposition article noun'. It might be more interesting to have a stock of say ten or so common sentence structures and then apply appropriate words into those structures from particular sets. One way to do this would be to store all the nouns (for example) into a vector and put that into a global word map with 'noun' as the key. Something like this:

```
std::map<std::string,
   std::vector<std::string> > words;

std::vector<std::string> > nouns;
nouns.push_back("Fred");
nouns.push_back("apple");
nouns.push_back("throttle");

words["noun"] = nouns;
```

Then a random sentence structure could be picked which would look something like:

```
std::string structure1 = "verb!";
  // e.g. "Swim!"
std::string structure2 = "noun verb noun"
  // e.g. "Dog eat dog."
```

The tokens in the structure would then be looked up in the words map and a random element of the vector of words be chosen for the resulting sentence.

(1) *Monte Carlo Method*, US Department of Commerce, National Bureau of Standards, Applied Mathematics Series 12, June 1951.

The Winner of CC 45

The bug was fairly obvious, but the fact that it didn't appear when single stepping the program with a debugger does make it a lot harder for novice programmers to resolve the problem. I picked Jim as the winner this time as I consider his advice that the best debugging tool is to *think* was the best key to cracking the problem.

I also liked John and Jean's different approaches to making the use of **`rand()`** reproducible for testing.

Code Critique 46

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

'I have built a simple singleton for logging and it seems to work, but I had to add a call to **`clear()`** the file stream after **`open()`** to get it to work properly. Does anyone know why this call is needed – I think my compiler's standard library has a bug?'

Please answer this question, but don't stop there…

```
// Logger.h
#include <string>
#include <fstream>

class Logger
{
public:
   static Logger * instance(
     std::string dest = "logfile.txt" );
   ~Logger();
   void write( std::string );
private:
   static Logger * theLogger;
   std::ofstream f;
};
```

```
// Logger.cpp
#include "Logger.h"

Logger * Logger::theLogger;

Logger * Logger::instance( std::string dest )
{
   if ( ! theLogger )
      theLogger = new Logger;
   theLogger->f.open( dest.data() );
   theLogger->f.clear(); // << Help - why??
   return theLogger;
}

Logger::~Logger()
{
   if ( this == theLogger )
      theLogger = 0;
   f.close();
}

void Logger::write( std::string line )
{
   f << line << std::endl;
}

// Example.cpp
#include "Logger.cpp"

int main()
{
   Logger::instance( "example.log" )->
     write( "Starting main" );
   //
   Logger::instance()->write( "Doing stuff" );
   //
   Logger::instance()->write( "Ending main" );
   delete Logger::instance();
}
```

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.

Don't forget to get your entries in by 1st July to scc@accu.org

Prizes provided by Blackwells Bookshops and Addison-Wesley

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-glamourous "not recommended" rating, you are entitled to another book completely free.

I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

The Unicode 5.0 Standard

By the Unicode Consortium, published by Addison-Wesley
ISBN 0-321-48091-0

Reviewed by Derek Jones

This book contains the latest version of the Unicode Standard, including all of its annexes, for a total of 1,417 pages (plus a CD).

If you have any interest in using European character sets you might be able to get away with using one or more of the ad-hoc methods that were invented before Unicode came along. However, the computing world jumped on the Unicode bandwagon nearly 10 years ago and if you plan to create text in any non-English language you will probably use Unicode and will need a copy of its standard. Buying this book is the most cost effective way of obtaining the Unicode Standard. As far as I could tell all of the material on the CD (text versions of all of the Unicode data files) is available on the Unicode web site.

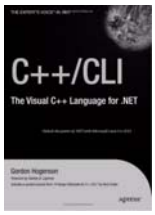
The book is a lot more than a collection of numbers and the corresponding visual representation of the characters they denote. There is extensive discussion, on a per language basis, of the problems associated with encoding the character sets of planet Earth (a proposal to support Klingon was rejected). This edition contains significantly more background material than earlier editions, but has shrunk from an A4 page size to something slightly smaller.

Individual characters rarely exist in isolation, they are combined to form words, sentences and paragraphs. The problems associated with creating and using subsets of the Unicode characters (ie, those belonging to a particular language, or combination of languages, such as inserting an English quotation, in English, into a character sequence representing another language) are covered in great detail. The detail is so great it is possible to create an implementation from it.

People not creating an implementation, wanting fewer pages and more background might like to check out *Unicode Demystified* by Richard Gillam, also published by Addison-Wesley ISBN: 0-201-70052-2.

Comparative Book Review

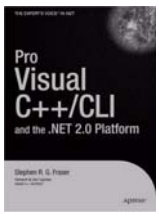
C++/CLI: The Visual C++ Language for .NET



By Gordon Hogenson, published by Apress

ISBN-13: 978-1-59059-705-7
ISBN-10: 1-59059-705-2

Pro Visual C++/CLI and the .NET 2.0 Platform



By Stephen R. G. Fraser, published by Apress

ISBN: 1-59059-640-4

Review by Richard Elderton

Microsoft Visual C++ 2005 Express is one humdinger of an integrated development environment, what is more, you can download it free of charge from Microsoft. Once one has it, one naturally thinks 'How can I make maximum use of this thing? And what's CLR all about?' I have echoes in my mind of people saying to me 'You know boy, your trouble is that you are always trying to run before you can walk!' But surely, if one has no interest in running, one is unlikely to care much about walking either. I confess though, my classic C++ is still at the toddler stage. But what better time to start thinking about a major new extension to the C++ language?

For those you don't know, CLI (Common Language Infrastructure) is an ISO/ANSI (language standard) specification for a virtual machine environment in which to run executables. The CLI specifies a 'standard intermediate language', which for .Net is the Microsoft Intermediate Language. The intermediate code is compiled by a just-in-time compiler at runtime, so the target machine has to have a version of the .Net Framework installed as a component of the Windows operating system. Microsoft's implementation of the CLI is known as the Common Language Runtime (CLR).

Since C++/CLI is essentially a superset of ISO standard C++, I innocently assumed that I would cover the gap that separates them with one small step, but it turns out that a giant leap is required, and you need expert guidance for setting the correct glide slope and velocity on landing. (Strictly speaking, this may be found in the Help system for Visual C++ 2005 Express, but that's like entering a centrifuge unless you know what you are doing already.)

The Mullah Lippman thinks C++/CLI is a jolly good idea, whilst the Chief Mufti Stroustrup has declared it to be an ungodly tendency:

> The wealth of new language facilities in C++/CLI compared to ISO Standard C++ tempts programmers to write non-portable code that (often invisibly) become [sic] intimately tied to Microsoft Windows…
> …CLI is 'language neutral' only in the sense that every language must support all of the CLI features to be 'first-class' on .Net.

[continued on back page]

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Computer Manuals** (0121 706 6000)
  www.computer-manuals.co.uk
- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

ACCU Information

Membership news and committee reports

View From The Chair

Jez Higgins
chair@accu.org

As I write this, New Year's resolution still unfulfilled[1], I still haven't recovered from this year's conference. Attending the conference s exhausting, because it just doesn't let up. You go from session to session to session, and in the gaps in between you talk. You talk about the sessions you been to, the session you plan to go to, what you're working on, what you will work on, that time when you both were on that job out in Germany and the infrastructure team ... It would be pretty tiring even if you didn't then sit up half the night drinking and talking some more. But you do, and it seems entirely natural and sensible at the time, because this is the one of the few opportunities we get to meet people we only otherwise know through articles or email lists. The discussion and story-swapping that goes on cements those friendships and builds "technical trust" (a phrase I've just made up, but hopefully you understand what I mean).

While the physical after-effects only take a day or two to overcome (even for those of us foolish enough to play squash[2]), the mental effects take that much longer to recover from. In fact, recover isn't even be the right word. Absorb maybe, or internalise. With a programme as varied as the ACCU Conference has, you don't take away a well-defined little collection of related information. Instead you carry away a smogasbord of session material, combined with all the conversations you had (both relevant and not). This great ball of "stuff" takes time to digest, to pick apart, to find and follow the threads of. Some pieces will take longer than others. I've already bought and am reading three books because of sessions I attended, but I know that months, maybe even years, later some snippet will come back to me when I need it.

From the outside, it might appear that the conference is some enormous booze-up, when it's really a deep and life changing experience. No, really!

To close, I'd like to thank all you who were able to attend this year's AGM. It was my first time in the chair and I didn't make as good a job of it as I might, but I think the issues discussed and decisions taken will work out well for the organisation. On behalf of the officers and committee I'd like to thank you for electing or reelecting us, and for the confidence you show in us.

ACCU confers on honarary memberships on members who have made a particular and long-standing contribution to the organisation. This year, the AGM recognised David Hodge, who stood down as membership secretary. David was membership secretary for 10 years, meaning for many, perhaps even most, of us he was our first point of contact with the organisation. In the time I've been involved with the committee he has been as solid and as reliable a person as you could wish for. His service to ACCU can not really be measured, and on behalf of the membership I would like to again thank David and wish him a long and happy retirement.

[1] CVu 17.1
[2] But for the good! Emacs roolz!

Membership Report

Mick Brooks
accumembership@accu.org

I'm struggling to find something to say for my first report as membership secretary proper. I think I'll choose to interpret that as indication that all is going well. The new membership system continues to work well, and we have close to 1000 members after the traditional rush of new subscriptions before the conference. Now we're in a comparative lull. Elsewhere in this issue, our new publicity officer has lots of ideas to increase awareness of the ACCU. Can you do your bit? Do consider mentioning us to your friends and colleagues. If you know a group of likely recruits and want to evangelise, drop me or David a line: we'd be happy to send spare journals and flyers for you to wave around.

As always, send details of contact info changes to me, and let me know if any of the journal mailings fail to reach you.

Publicity Officer

David Carter-Hitchin
publicity@accu.org

It is a great shame that there are many IT professionals who have never heard of the ACCU. It is one thing to know about the ACCU and have decided to not be a member, but it is an entirely different thing to not be a member because you've never heard about the organisation. I have spoken to numerous C++, Java, Python, C# programmers who have never of the ACCU, and I believe a fair proportion of them would be interested in joining and finding out what we're all about. I think it is much more common for a programmer to be ignorant of the ACCU's existence than not. This must change.

It is for this reason that I have put myself forward for the role of publicity officer, as I believe every person involved with IT and programming in particular should have heard about the ACCU and know broadly what it constitutes. I have come to the ACCU relatively late in my career, and I wish I'd known about it earlier as I believe it is an immensely valuable organisation that fosters excellence in programming through various activities. It has certainly helped me in ways that outstrip the cost of the annual membership fee.

You can help in advertising the ACCU. At the bottom of your e-mail, in your signature, please put the following message (or something similar):

ACCU - http://www.accu.org/ - Professionalism in Programming

Do this now, don't leave it until tomorrow when you might forget! If you wish to put a few more lines in describing what we do, then that's great too. If you run a website, or know people who do, see if you can put a link to the ACCU site from your site. Also tell all of your friends and colleagues when you have an opportunity to do so - the power of personal recommendation is not to be underestimated! You can also help me by volunteering a small amount of your time - I'll talk about this in a moment.

Spreading the word about the ACCU will attract more members which will have several notable benefits:

1. More quality CVu and Overload articles, on more diverse subjects.
2. More book reviews.
3. More mentors.
4. The ACCU will have more funds which will empower it to do more things. There are a large number of possibilities here, for example to provide scholarships, enhance the magazine content (perhaps with a DVD), put on more events and so on. Money buys a lot of things!
5. To diversify and enrich the membership. The ACCU will benefit from attracting members from diverse fields which are perhaps poorly or not at all represented at the moment and also to increase the numbers in those areas which are well represented. You can never have enough of a good thing. Attracting good people who care about what they do within IT is really what the ACCU is about.

To this end, I will putting a publicity campaign together over the coming months. This will consist of targeting obvious organisations such as:

- Software Houses
- Universities
- Recruitment and Training Companies
- Other programming organisations (e.g. USENIX, MSDN) Banks and other corporates.

I will sending out a fair number of letters, but I may also need volunteers to help place flyers in strategic locations or to talk to local students (for example). If you can help with this, then please get in touch via the e-mail address below.

I will also be seeking to set up reciprocal arrangements with other conferences such as the BoostCon. The idea is that we give them a space at our next conference in exchange for a space at theirs. For this to happen I am going to need help, as I can't cover all the conferences on my own! If you would like to go to a conference to represent the ACCU can you please get in touch with me and I'll keep your details on file for when the time comes.

I will also be looking to get a slot on Radio and perhaps even something on TV. I don't have many contacts in this area, so if you do, then let me know.

Finally if you have any other ideas for how we can spread the word, then please get in touch.

Bookcase (continued)

Dare I risk my mortal soul? Well, I think its worth a serious look. Whilst classic C++ programming is a more 'pure' form of the art, programming with C++/CLI promises to be an easier, more productive and practically useful activity, apart from the fact that its primary purpose is to produce applications which target the .Net platform, which is all about interoperability between different programming languages.

There seem to be few books catering to the C++/CLI novice, certainly none within a day's march, but I found two possible titles on www.apress.com: *C++/CLI: The Visual C++ Language for .NET* by Gordon Hogenson, and *Pro Visual C++/CLI and the .NET 2.0 Platform* by Stephen R. G. Fraser.

The publisher's promotional blurbs for these books are very similar, making it difficult to make a rational choice between them. The 'Pro' in Fraser's title suggests that it might be more difficult to digest, but the larger page count (961) suggests that it might have more comprehensive coverage of the subject, and /or have more detailed explanations. This book would have cost me £45.59 from Amazon.

I plumped for Hogenson's book initially. It has a total page count of 448 and cost me £29.44, new, from Amazon. It is an unusually comfortable tome to handle, for one describing a computer programming language.

This book serves well as an introduction to C++/CLI programming, but the author makes no pretence to its being fully comprehensive. He assumes the reader to be proficient in classic C++, or another language targeting .NET. The book is billed as addressing the .Net Platform version 3.0 (which is a superset of version 2.0), but as far as I could tell, the author does not expound on any of the extra class libraries it contains (chiefly concerning 3D vector graphics, communication between applications, task automation, and digital identities).

By chapter four, I was finding this book very hard work. I then discovered that you can buy PDF versions of many Apress titles, including *Pro Visual C++/CLI,* from their web site. A paper book with over 900 pages has got to be a pretty cumbersome thing to handle, and since my laptop computer keeps me warm in the winter, I thought I would give the PDF a try. It cost me £18.33.

This book does have a larger scope than Hogenson's. Fraser, for instance, has a lot to say about the very significant topic of Windows Forms, whereas Hogenson mentions it only briefly. Despite the 'Pro' in the title, Fraser's book takes things at a more leisurely pace and is kinder to the less experienced reader.

After a while I got used to this rather unnatural way of reading a book and appreciated the ease with which you can find references and copy and paste material into a notes file. If my laptop had enough grunt to drive MS Visual C++ 2005 Express I could try out the sample programs as well. Of course, I could do this on my desktop PC but after the day's work I have usually had enough of sitting at a desk.

Quite early on I needed to understand the concept of 'member properties' – one not used in classic C++. Hogenson's multi-faceted explanations were beyond my comprehension until I had read Fraser's. In only a few sentences he satisfied my curiosity and I was able to proceed, whilst accepting that I would need to return later.

Fraser shows screen shots of program output; this saves you the trouble of trying out stuff yourself (except in cases where you just don't believe him), since it adds a touch of realism. Hogenson lists program output as printed text between horizontal rule lines; this has the advantage of being more legible.

Fraser's PDF book is fairly stuffed with typing errors. I noted 30 in the first 156 pages (is this more than the national average?). They do force one to think about what the author really means, which is not a bad thing, but it can become irksome.

Hogenson's errors are more difficult to spot and fewer on the ground (averaging one per 28 pages, in the first few chapters). Being a senior Microsoft technical writer, he may care more about that kind of thing.

In conclusion, the electronic Fraser has been better value for money, whilst Hogenson's book is still entertaining when I am too far away from a mains power receptacle to give life support to my old laptop. Now that summer's coming on I just might buy Fraser's paper version too.