The magazine of the ACCU

www.accu.org

{cvu}

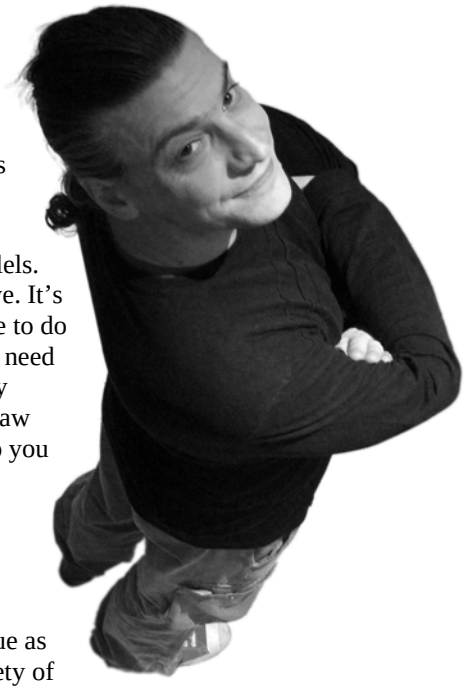
Volume 22 Issue 3 July 2010 £3

Features

{cvu}

Volume 22 Issue 3
July 2010
ISSN 1354-3164
www.accu.org

Software Construction Site



A colleague was recently comparing the task of managing a software development team with that of organising a construction site. The idea was prompted by a large construction project going on near our office; he sits near the window overlooking it, which makes me wonder if any of the site managers look up at our office and compare what they do with managing a software team.

I quite like the analogy; it certainly strikes some parallels. There's certainly more in both fields than meets the eye. It's much more than just ensuring you have enough people to do the work – although estimating how many people you need can certainly be a trial. On a construction site, it's very expensive to have people idle, so they must have the raw materials always to hand. But on a large site, where do you put two hundred tonnes of sand so that you're not stopping someone's work just by it being there? Then there are the logistical challenges of moving the materials around as work in one area completes, and needs to start somewhere else.

The people problem might seem a straightforward issue as well, until you consider that a project will need a variety of different skill sets, and at different phases of completion. As with the raw materials, you need to ensure that the skills are there at the right time to avoid holding other parts up.

All in all, an interesting enough comparison I thought I'd share with you. So, I have a question for you. Can we learn more about our own craft by thinking about comparisons like these? Or are they at best a distraction from the unique challenges software development presents?

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

FEATURES

REGULARS

SUBMISSION DATES

C Vu 22.4: 1st August 2010
C Vu 22.5: 1st October 2010

Overload 99: 1st September 2010
Overload 100:1st November 2010

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

Implementing One-to-Many Relations in C++

Xavier Nodet looks at managing relationships in C++.

This paper presents a memory-ecient implementation of One-to- Many relations in C++. Automating relations relives the programmers from tedious and error-prone manual management of back-pointers and notications when objects are inserted or removed from a relation, or destroyed altogether. This implementation only uses a minimal amount of memory: a container of pointers in the owner, and one pointer in each owned object.

Thanks to the use of the powerful mechanisms of C++ templates, these relations are perfectly type-safe, without the need for down-casting, very exible, and often as ecient as hand-written code.

Introduction

The need to manage relations between objects is at the core of many business applications: objects refer to each other with various semantics, with or without inverse links, for various durations.

A very common case is the One-to-Many relation: an **`Owner`** object has relationships with several **`User`** objects. The owner must know which objects it is related to, and the users have an inverse link to their owner. When a user is added to or removed from its owner, both links should be updated. When the owner is destroyed, all the owned objects are also destroyed. Such relations are extremely common, and often implemented with ad-hoc code.

This paper proposes a set of classes to implement those One-to-Many relations with a minimal amount of code to be written in the **`Owner`** and **`User`** classes, and performance equivalent to hand-written code, both in terms of memory and CPU.

One-to-many relations

For the purpose of this paper, a One-to-Many relation between classes **`Owner`** and **`User`** is defined as the following:

- An **`Owner`** instance refers to one or more instances of **`User`**.
- A given instance of **`User`** can appear at most once in any **`Owner`**.
- A **`User`** refers to 0 or 1 instance of **`Owner`**.
- A **`User`** refers to an **`Owner`** if and only if this **`Owner`** has the address of **`User`** in its list.

Preliminary implementation in C++

The core idea of the proposed implementation is that each relation is represented by two classes **`RelationOwner`** and **`RelationUser`**, one for the **`Owner`**, and one for the **`User`**. For this preliminary implementation, an instance of **`User`** is necessarily owned by one instance of **`Owner`**. The **`RelationOwner`** class stores pointers to the instances of **`User`** which it has a relationship with, and the **`RelationUser`** class stores a pointer to its owner. The classes **`Owner`** and **`User`** inherit from **`RelationOwner`** and **`RelationUser`**. This is all that's needed to implement the relation, but programmers may choose to hide the API provided by **`RelationOwner`** and **`RelationUser`** through methods of **`Owner`** and **`User`** delegating to their base class.

You will find in listing 1 a preliminary version of the **`RelationOwner`** class. It defines a container to hold pointers to instances of **`User`**, and methods to attach a **`User`** to its **`Owner`** and detach it, and iterate over the content of the container.

Note the destructor that first swaps the **`_owned`** container with an empty one before deleting all the objects that it owns. Iterating on a container that's not the one that stores all the pointers to the users is necessary because, when deleted, the users will try to remove themselves from this container. Copying would be less efficient in itself, and erasing from an empty container is quicker...

Listing 1

```
template <class Owner,class User>
class RelationUser;

template <class Owner, class User>
class RelationOwner {
  friend class RelationUser<Owner, User>;
public:
  typedef std::set<User_> Container;
  typedef
    typename Container::const_iterator iterator;
public:
  ~RelationOwner();
  iterator begin() const {return _users.begin();}
  iterator end() const {return _users.end();}
private:
  void attach(User_ u) {_users.insert(u);}
  void detach(User_ u) {_users.erase(u);}
private:
  Container _users;
};
template <class Owner, class User>
RelationOwner<Owner,User>::~RelationOwner() {
  Container temp;
  swap(_users, temp);
  for ( iterator it (temp.begin());
      it != temp.end(); ++it) {
    delete _it ;
  }
}
```

The **`RelationUser`** class is presented in listing 2. The interesting thing to notice is the implementation of the **`user()`** method, that returns the address of the **`User`** object. It uses the CURIOUSLY RECURRING TEMPLATE PATTERN and this was suggested to me on Stack Overflow (see [1]). The CRTP idiom occurs when the base class **`RelationUser`** is templated on the type of its derived class **`User`**

Listing 2

```
template <class Owner, class User>
class RelationUser {
public:
  explicit RelationUser(Owner_ o)
    : _owner(o)
  { _owner..>attach(user()); }
  ~RelationUser() { _owner..>detach(user()); }
private:
  User_ user() {return static_cast<User_>(this);}
private:
  Owner_ _owner;
};
```

XAVIER NODET

Xavier has been programming for almost as long as he can remember, and is still as eager to learn as the day he first saw a ZX81. So he's always on the look for new stuff to read, to try or to tinker with...
He can be contacted at xavier.nodet@gmail.com.

Listing 3

```
class User;
class Owner;

class Named {
public:
  Named(const std::string& name)
    : _name(name) {}
  const std:: string& name() const {
    return _name;
  }
  ~Named() {
    std :: cout << "Object '" << _name
                << "' destroyed" << std::endl;
  }
private:
  std :: string _name;
};

class Owner
  : public RelationOwner<Owner, User> {
public:
  void show();
};

class User
  : public RelationUser<Owner, User>
  , public Named {
public:
  User(Owner_ owner, const std::string& name)
    : RelationUser<Owner,User>(owner)
    , Named(name)
  {}
};

void Owner::show() {
  std :: cout << "Users of Owner: ";
  for (Owner::iterator it (begin());
       it != end(); ++it) {
    std :: cout << (_it)..>name() << " ";
  }
  std :: cout << std::endl;
}

void test() {
  Owner_ owner (new Owner());
  User_ user1 (new User(owner, "1"));
  User_ user2 (new User(owner, "2"));
  User_ user3 (new User(owner, "3"));
  owner..>show();

  delete user2;
  owner..>show();
  delete owner; // All users deleted
}
```

(see [2]). Indeed, we need the **`user()`** method to return the address of the **`User`** instance, not of its **`RelationUser`** base class. See 'Smart pointers' on page 7 for a complete discussion on this topic.

A simple example

Listing 3 is an example of use, and the printed output is:

```
Users of Owner: 1 2 3
User '2' destroyed
Users of Owner: 1 3
User '1' destroyed
User '3' destroyed
```

The net result is that just by inheriting from those classes, users get automatic management of ownership.

Adding functionalities

Several relations to the same class

The most pressing issue with the code above is that it is *not* possible to add two or more relations from one class to another... Suppose the user wants to implement a link between two objects, with a 'preceding' and 'following' node for each link. To define the **`Node`** class that should hold the **`Links`**, one would need to inherit twice from the **`RelationOwner<Node,Link>`**, but this is clearly impossible.

The solution is to add a *type marker* template parameter to the **`RelationOwner`** template class, e.g. an empty struct. Once two dierent types **`Preceding`** and **`Following`** have been defined, the **`Node`** can inherit from both **`RelationOwner<Node,Link,Preceding>`** and **`RelationOwner<Node,Link,Following>`**, which are different classes from the compiler's point of view, even if the rest of their definition is exactly the same. Providing a default value allows programmers that don't need this feature to simply ignore it. See Listing 4.

Such a template parameter is also added to **`RelationUser`** so that each **`RelationUser`** instantiation can refer to its **`RelationOwner`** counterpart: now that the owner may have several **`RelationOwner`** base classes, it has several **`attach()`** and **`detach()`** methods. This implies that **`RelationUser`** must be able to access the correct method: the one that corresponds to the other side of the relation, but not another. Calls to those methods must thus be qualied, which implies to know the exact type of **`RelationOwner`**.

Flexibility using policies

Another issue is that storing the users in an **`std::set`**, which is the default, is not necessarily always the best choice. It may be the case that the same user should appear several times in the same relation. Or that, on the contrary, it can never be the case that a user is inserted twice, and an **`std::vector`** would be a better choice.

As defined by Andrei Alexandrescu in [3], 'a policy defines a class interface or a class template interface. [They] have much in common

Listing 4

```
template <class Owner, class User,
          class RelId = void>
class RelationOwner {
  ...
};

template <class Owner, class User,
          class RelId = void>
class RelationUser {
  typedef typename
    RelationOwner<Owner,User,RelId> RelOwner;
public:
  RelationUser(Owner owner)
    : _owner(owner)
  {
    _owner..>RelOwner::attach(user());
  }
  ~RelationUser() {
    owner()..>RelOwner::detach(user());
  }
  ...
};

struct Preceding {};
struct Following {};

class Owner
  : public RelationOwner<Owner,User,Preceding>
  , public RelationOwner<Owner,User,Following> {
  ...
};
```

Listing 5

```
template <class L>
struct SetValuesPolicy {
  typedef std::set<L> Container;
  static void insert(Container& c, L l) {
    c. insert ( l );
  }
  static void remove(Container& c, L l) {
    c. erase( l );
  }
};

template <class L>
struct VectorNoCheckValuesPolicy {
  typedef std::vector<L> Container;
  static void insert(Container& c, L l) {
    c.push_back(l);
  }
  static void remove(Container& c, L l) {
    c. erase(std :: remove(c.begin(), c.end(), l
),
             c.end());
  }
};

template <class Owner, class User,
         class RelId = void,
         class Policy = SetValuesPolicy<User> >
class RelationOwner {
public:
  typedef typename Policy::Container Container;
public:
  void attach(User owned) {
    Policy :: insert (_users,owned);
  }
  void detach(User owned) {
    Policy :: remove(_users,owned);
  }
  ...
private:
  typename Policy::Container _users;
};
```

with traits, but differ in that they put less emphasis on types and more emphasis on behavior'.

Policy for container of users

A policy can define the type of the container to use in the **`RelationOwner`** template, and methods to insert and remove objects from this container (see Listing 5).

The **`Policy`** template parameter is, by default, the **`SetValuesPolicy<User>`** class. This class denes a **`Container`** type, and methods **`insert()`** and **`remove()`**. Another class can dene those four members so that an **`std::vector`** is used, as in **`VectorNoCheckValuesPolicy`**, or any other data-structure.

Policy for deletion behavior

If a **`User`** instance is not necessarily tied to an **`Owner`** instance but can live on its own, another policy can be defined to determine the behavior of the **`RelationOwner`** when it is destroyed: should it also destroy all the users? Should it do something else? (Listing 6 contains a policy for deletion behavior.

By default, destruction of a **`RelationOwner`** instance triggers the destruction of all the users in this relation. But providing another **`operator()`** allows to choose another behavior. Note that the user is always 'detached': as the owner is being destroyed, one does not want to have pointers to it any more... A **`static_cast`** is needed for this call, so

Listing 6

```
template <class L> struct DefaultEnder {
  void operator()(L l) { delete l; }
};

template <class Owner, class User,
         class RelId = void,
         class Policy = SetValuesPolicy<User>,
         class UserEnder = DefaultEnder<User> >
class RelationUser {
public:
  void reset() {
    if (_owner) {
      _owner..>detach(user());
      _owner = 0;
    }
  }
  ...
  private:
    User user() {return static_cast<User>(this);}
    RelOwner _owner;
  };

  template <...>
  class RelationOwner {
    ...
  public:
    ~RelationOwner() {
    Container temp;
    swap(_users, temp);
    for ( iterator it (temp.begin());
        it != temp.end(); ++it) {
      static_cast<RelUser>(it)..>reset();
      UserEnder()(it);
    }
  }
  ...
private:
  typename Policy::Container _users;
};
```

that the call to **`detach()`** is not ambiguous when **`User`** implements several relations.

Array-like API for access to users

It is very easy to provide an array-like API on **`Owner`** (Listing 7).

Listing 7

```
template <class L>
struct VectorValuesPolicy {
       ...
  static L elem(const Container& c, size_t i) {
    return c[i ];
  }
  static size_t nbElem(const Container& c) {
    return c.size ();
  }
};

template <class RelationUser>
class RelationOwner {
  ...
  User user(int i ) const {
    return Policy::elem(_users,i);
  }
  size_t nbUsers() const {
    return Policy::nbElem(_users);
  }
  ...
};
```

Listing 8

```
#include "relations.h"

using namespace std;
class Named {
public:
  Named(const string& name)
    : _name(name)
  {}
  const string& name() const { return _name; }
  ~Named() {
    cout << "Object '" << _name
         << "' destroyed" << endl;
  }
private:
  string _name;
};

using namespace relations;

struct Preceding {};
struct Following {};

class User;
class Owner;

typedef RelationUser<Owner,User,Preceding,
  VectorNoCheckValuesPolicy<User> >
  PrecedingUser;
typedef RelationOwner<PrecedingUser>
  PrecedingOwner;
typedef RelationUser<Owner,User,Following>
  FollowingUser;
typedef RelationOwner<FollowingUser>
  FollowingOwner;

class Owner
: public PrecedingOwner
, public FollowingOwner
, public Named {
public:
typedef PrecedingOwner Preceding;
typedef FollowingOwner Following;
Owner(const string& name) : Named(name) {}
};

class User
  : public PrecedingUser
  , public FollowingUser
  , public Named {
```

What's interesting is the **Policy** class does not need to define those **elem** and **nbElem** methods as long as methods **user(size_t)** and **nbUsers()** on **RelationOwner** are not used. Those will be compiled only if used somewhere in the code, and this compilation will only succeed if **Policy** has the correct API. Users of set-based relations (for which the policy does not provide the API) will thus not be able to access elements with the array-like API, and won't have to wait for run-time before discovering this...

Example of use

Listing 8 shows usage of the **RelationOwner** and **RelationUser** classes. The output for this example is:

```
owner1 is related as Preceding to 'user1'
owner1 is not related as Following to any item
owner2 is not related as Preceding to any item
owner2 is related as Following to 'user1' 'user2'
Deleting owner1
owner2 is not related as Preceding to any item
owner2 is related as Following to 'user2'
```

Listing 8 (cont'd)

```
public:
  typedef PrecedingUser Preceding;
  typedef FollowingUser Following;
  User(Owner owner1,
    Owner owner2,
    const string& name)
    : Preceding(owner1)
    , Following(owner2)
    , Named(name)
  {}
};

template <class Rel>
void printRel(typename Rel::_Owner owner,
              const string& relName) {
  if (owner..>Rel::begin() == owner..>Rel::end())
{
    cout << owner..>name()
         << " is not related as "
         << relName << " to any item";
  } else {
    cout << owner..>name()
         << " is related as "
         << relName << " to ";
    for (Rel :: iterator it
(owner..>Rel::begin());
         it != owner..>Rel::end(); ++it) {
      cout << "'" << (it)..>name() << "' ";
    }
  }
  cout << endl;
}

void test() {
  Owner owner1 (new Owner("owner1"));
  Owner owner2 (new Owner("owner2"));
  User user1 (new User(owner1, owner2, "user1"));
  User user2 (new User(0, owner2, "user2"));
  printRel<Owner::Preceding>(owner1,
"Preceding");
  printRel<Owner::Following>(owner1,
"Following");
  printRel<Owner::Preceding>(owner2,
"Preceding");
  printRel<Owner::Following>(owner2,
"Following");

  cout << "Deleting "
       << owner1..>name() << endl;
delete owner1;
printRel<Owner::Preceding>(owner2, "Preceding");
printRel<Owner::Following>(owner2, "Following");
}
```

Other approaches

The traditional approach

The usual way to manage those links is to have each object hold a pointer (or a container of pointers) to the object(s) that it may have a relation with. E.g. in a Belongs-To relation between **Plant** and **ProductionLine** classes, each **Plant** holds a list (or array, or any other container) of pointers to its **ProductionLine** instances, while the **ProductionLine** instances have a pointer to their **Plant**. Each time a relation is implemented this way, some code must be added in both classes to manage this relation: setting the back-pointer when a new object is being referred to, resetting it when the object is no longer referred to, making sure that the pointer is removed from the container if the object referred to gets destroyed, and notifying or even destroying the objects referred to when the containing object gets destroyed. All this code gets repeated for

each relation, which is bug-prone and decreases the cohesiveness of the classes. In particular, the destructor of **`Owner`** needs to be updated (so that it notifies the owned objects) each time a relation is added, which is a change that's very easy to forget.

Code generation

Generating the code to handle the relations from a description of the BOM is a possible solution. Compared to the traditional approach, there is much less risk of mistakes. But it requires that this description is available, and external tools generate the code.

Smart pointers

A smart pointer class can be devised with the goal of being used in the **`User`** class to refer to the **`Owner`** instance. Destruction of the smart pointer (when the **`User`** instance is destroyed) would trigger the notication of the **`Owner`** so that it removes the address of the **`User`** from its list. The main drawback of this approach is that there is no way for the smart pointer instance to know the address of the **`User`** instance it belongs to, other than getting it at construction time and storing it.

The proposed approach solves this problem efficiently. Recall the code to retrieve the address of the **`User`** instance from inside the **`RelationUser`** class:

```
template <class, class User>
class RelationUser {
                    ...
  User_ user() {return static_cast<User_>(this);}
};
```

The reasons this code works are the following:

- The compiler provides the **`this`** pointer, which is not of type **`User*`**, but of type **`RelationUser<...>*`**, as we're in a method of the latter.
- It is only at the point of instantiation (i.e. when the compiler actually encounters a call to **`RelationUser<...>::user()`**) that the compiler inserts the code of the method.
- The method is called in the destructor of **`RelationUser`** to notify the **`Owner`** of the destruction of an object it owns. This destructor is instantiated from the destructor of **`User`**. The **`User`** class is obviously known to the compiler at this point.
- As **`User`** is a template parameter of **`RelationUser`** the former is known to the compiler when instantiating the code for **`RelationUser<...>::user()`**. The compiler can thus generate the code to convert a pointer to the base class **`RelationUser`** into a pointer to the derived class **`User`**. Et voilà...

The fact that the compiler provides the **`this`** pointer, plus the fact that **`User`** inherits from **`RelationUser`** (something the compiler knows as **`User`** is a template parameter of **`RelationUser`**), allows to retrieve the address of the **`User`** object without incurring the cost of storing it. In terms of CPU, the implementation of the cast is a no-op if the **`RelationUser`** class is the first super-class of the **`User`** and simply the substraction of a constant in the other cases. Pretty cheap...

Now suppose that the **`User`** class stores, as a member, an instance of a smart pointer. There is no way [4] this smart pointer would be able to retrieve the address of **`User`** without getting it from **`User`** itself and storing it. The inheritance solution proposed in this paper is thus more efficient in terms of memory.

Relationship manager

One proposed solution to these issues, as in [5], is to have all relations managed and stored into a 'central' **`RelationhipManager`** class. This approach increases the cohesiveness of the classes in the BOM, but does so at the expense of the additional memory needed to store the information: the **`Owner`** and **`User`** classes store a pointer to the relationship manager (instead of a pointer to the other object in the relation), but the latter must also use additional memory to hold all the information about which object has a relation with which. This approach is also less efficient in terms of CPU.

Conclusion

This paper shows an efficient implementation of One-to-Many relations in C++.

Thanks to the template mechanisms, this implementation is type-safe, flexible and often as efficient as hand-written code. ■

Acknowledgements

Thanks to Kevlin Henney and Georges Schumacher for their encouragements and very useful comments.

References and notes

[1] http://stackoverflow.com/questions/709790/how-can-i-know-the-address-of-owner-object-in-c/709996#709996

[2] The Curiously Recurring Template Pattern: http://c2.com/cgi/wiki?CuriouslyRecurringTemplate

[3] Andrei Alexandrescu. *Modern C++ Design* (Addison-Wesley, 2001)

[4] Except if allowing non-portable code using undefined behavior

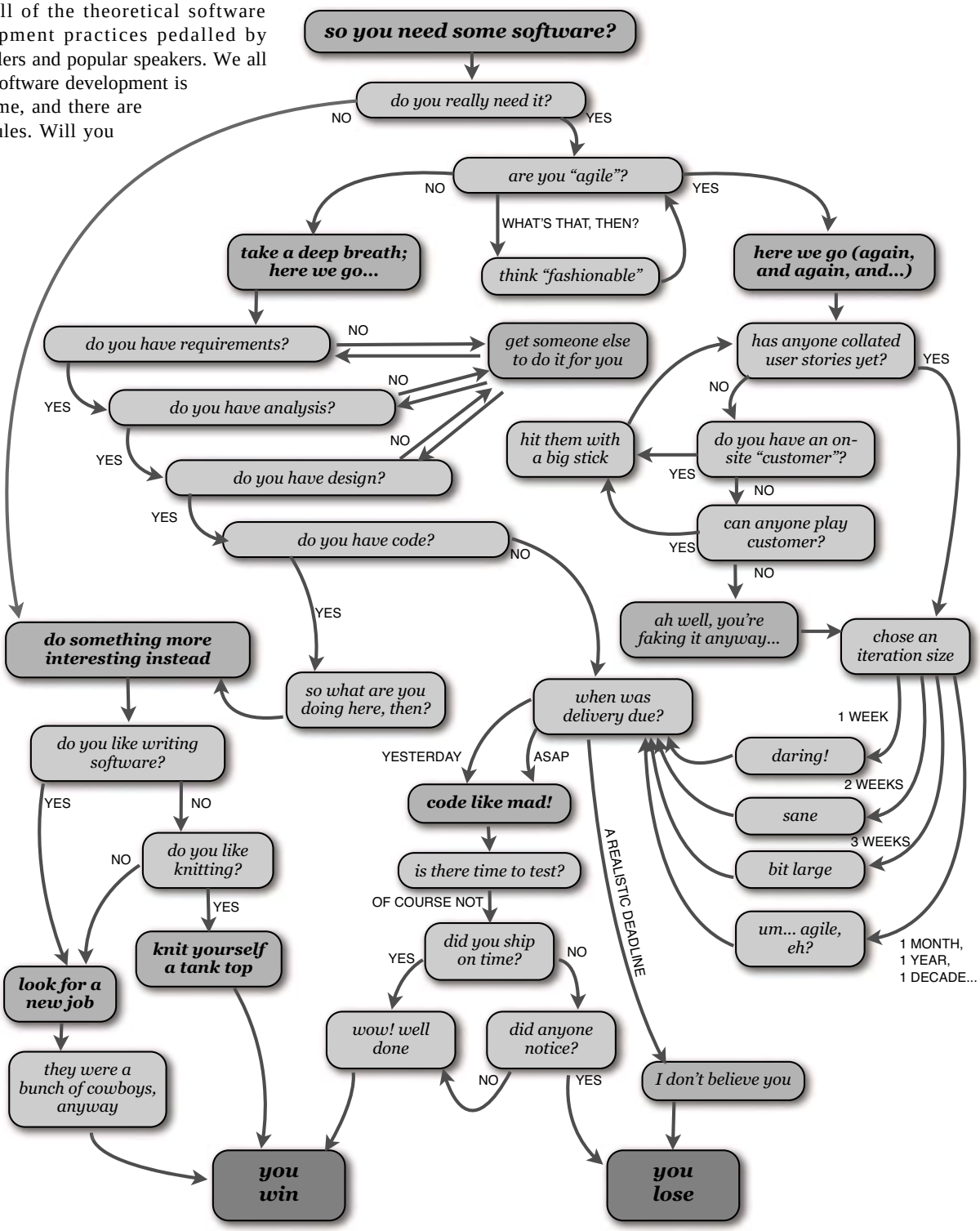
[5] Andy Bulka, Relationship Manager, http://www.andypatterns.com/index.php?cID=44

The full code is available at http://xavier.nodet.free.fr/Relations/

Software Development in 2010

Pete Goodliffe helps shows us how to develop winning software.

Forget all of the theoretical software development practices pedalled by booksellers and popular speakers. We all know that software development is really a game, and there are very few rules. Will you win or lose?



PETE GOODLIFFE

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

Experiments in String Switching

Matthew Wilson examines switching on strings.

This article describes my recent efforts in starting to synthesise succinct, simple, syntax for selecting between string cases. In so doing, it discovers a new use for the old trick for returning arrays from functions.

Introduction

In the last couple of years I've been developing a lot of command-line software, and I've become mighty tired of boilerplate coding for the handling of command-line arguments. One product of this is a new (and I think superior) command-line argument sorting and parsing (CLASP) library, which I hope to introduce to the ACCU community in a forthcoming CVu article (after its first public-domain release). Right now, I want to discuss a smaller, separate component that happens to be used in evaluating the values obtained via CLASP, or any other command-line argument.

One of the programs that I've been reworking lately is a source analysis tool that can examine any C-family language. By default, it infers the programming language from the file extension (and some other heuristics), but occasionally it's necessary or more convenient to specify the language explicitly. The command-line switch involved, **`--language`** (short form alias **`-L`**), understands values specifying the language and, optionally, the language version, as in:

```
--language=C,99
--language=D
--language=C++,98
--language=Java,1.2
```

The CLASP library provides the ability to test for the presence of a command-line option and elicit its value, as in:

```
int tool_main(
  clasp::arguments_t const* args
)
{
  ...
  char const* val;
  if(clasp::check_option(args,
     "--language", &val, NULL))
  {
    ... // parse language and version from val
  }
  ...
```

That's nice and expressive, but of course that's only the easy part. Now we have to split the value by a comma and, optionally, a period (or full-stop to the British/Antipodeans amongst you). Once that's done, we then have to recognise the language string fragment, and convert the version-hi and version-lo string fragments to integers. (We *could* treat the version fragment atomically, and convert it to a floating-point value, but I am one of those programmers who know just enough about floating-point programming to avoid it wherever I possibly can.)

Apart from the 'recognise the language' part, the rest of this programming is pretty straightforward. I'll just show you how I've done it, implemented in terms of common STLSoft components, and you can substitute whatever you might use instead. Listing 1 shows all the processing except the language recognition and verification.

Listing 1

```
enum language_t
{
  SSSALC_LANG_NEUTRAL = 0x0000,
  SSSALC_LANG_C       = 0x0100,
  . . .
  SSSALC_LANG_JAVA    = 0x1000,
};

int tool_main(
  clasp::arguments_t const* args
)
{
  ...
  language_t  language  = SSSALC_LANG_NEUTRAL;
  int         langVerHi = 0;
  int         langVerLo = 0;
  char const* val;
  if(clasp::check_option(args, "--language",
     &val, NULL))
  {
    string_t  langStr;
    string_t  verStr;
    string_t  verHiStr;
    string_t  verLoStr;

    stlsoft::split(val, ',', langStr, verStr);
    stlsoft::split(verStr, '.', verHiStr,
       verLoStr);

    ... // detect and verify language

    if( !stlsoft::try_parse_to<int>(
       langVerHiStr, &langVerHi) ||
       !stlsoft::try_parse_to<int>(
       langVerLoStr, &langVerLo))
    {
      ff::fmtln(std::cerr,
         "Invalid language version specified:
         {0}", val);
      return EXIT_FAILURE;
    }
  }
```

There's no need to check whether either split operation succeeds, because if we fail to elicit a language string it will be detected in the detect/verify step, and if we elicit an invalid language version that'll be picked up by the string-to-integer checks.

All that's left is to implement the check of the language string fragment and translate it into the corresponding language enumerator (in this case **`language_t`**), or to report an invalid language and return **`EXIT_FAILURE`**. There are a variety of possibilities.

One would be to have a giant **`if-else`** chain.

MATTHEW WILSON

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

```
if("C" == langStr)
{
  language = SSSALC_LANG_C;
}
else
if("C++" == langStr)
{
  language = SSSALC_LANG_CPLUSPLUS;
}
else
. . .
```

Yuck!

The way I am accustomed to doing this is to define a simple dedicated aggregate structure consisting of two fields – one of type **`char const*`** and the other of type **`language_t`** – and a static constant literal array of them, as in:

```
struct language_map_t
{
  char const* name;
  language_t  lang;
};
static const language_map_t languages[] =
{
  { "C", SSSALC_LANG_C },
  { "C++", SSSALC_LANG_CPLUSPLUS },
  { "D", SSSALC_LANG_D },
  . . .
}
```

Then, we'd iterate over the array, comparing the name member with the received language string, and setting the language variable to the lang member of the corresponding element. If we get to the end of the array, then the user has specified an invalid language. It's simple to do, and hard to get wrong:

```
size_t i;
for(i = 0; i != NUM_ELEMENTS(languages); ++i)
{
  if(languages[i].name == langStr)
  {
    language = languages[i].lang;
  }
}
if(i == NUM_ELEMENTS(languages))
{
  ff::fmtln(std::cerr,
     "Invalid language specified: {0}", val);
  return EXIT_FAILURE;
}
```

But the problem is, it's so incredibly tedious to do! There's no challenge, nothing fun, just wearisome repetitive boilerplate. In a fit of pique I decided to seek an alternative. The result is the **`stlsoft::string_switch()`** function, and the associated **`stlsoft::string_cases()`** function, both of which appear in the latest alpha release of STLSoft 1.10. They are used as in Listing 2.

To be sure, it's not rocket surgery, but it's highly expressive and, if I say so myself, a pleasure to use. And that's important, because boring boilerplate work has a habit of getting itself wrong.

Let's now look at the implementation, and you can consider whether it's worth the effort.

Implementation

To get this to work with multiple compilers took me more effort than I'd care to admit, involving a blind alley involving fixed array class templates (STLSoft's **`fixed_array_1d`** [1], if you're interested) and workarounds involving **`std::vector`**. As is often the case, I slept on it, and the next

Listing 2

```
#include <stlsoft/util/string_switch.hpp>

static int tool_main(
  clasp::arguments_t const* args
)
{
  . . .
  char const* val;
  if(clasp::check_option(args, "--language",
     &val, NULL))
  {
    . . .
    stlsoft::split(val, ',', langStr, verStr);
    stlsoft::split(verStr, '.',
       verHiStr, verLoStr);

    if(!stlsoft::string_switch(
        langStr.c_str()
      , &language
      , stlsoft::string_cases(
          "C",    SSSALC_LANG_C
        , "C++",  SSSALC_LANG_CPLUSPLUS
        , "C#",   SSSALC_LANG_CSHARP
        , "D",    SSSALC_LANG_D
        , "Java", SSSALC_LANG_JAVA
        )
      )
    )
    {
      ff::fmtln(std::cerr,
         "Invalid language specified: {0}", val);
      return EXIT_FAILURE;
    }

    ...
```

day hacked it back to the bare bones to get the current solution in just an hour or two. Fred Brooks' wisdom on writing two versions strikes again [2].

Although I plan to enhance them to work with arbitrary string types in the future, at the moment the functions work only with C-style strings. Consequently, they're pretty straightforward. Let's first look at the easy one, **`string_switch()`**, in Listing 3.

Listing 3

```
// in namespace stlsoft
template<
  typename C
, typename E
, size_t   N
>
inline bool string_switch(
  C const* s
, E*       result
, ximpl::string_case_item_array_t<C, E, N>
    const& cases
)
{
  { for(size_t i = 0; i != cases.size(); ++i)
  {
    ximpl::string_case_item_t<C, E>
       const& case_ = cases[i];
    if(0 == ::strcmp(case_.name, s))
    {
      *result = case_.value;
      return true;
    }
  }}
  return false;
}
```

Listing 4

```
// in namespace stlsoft
namespace ximpl
{
  template<
    typename C
  , typename E
  >
  struct string_case_item_t
  {
    C const* name;
    E        value;
  };

  template<
    typename C
  , typename E
  , size_t   N
  >
  struct string_case_item_array_t
  {
    string_case_item_array_t(
      string_case_item_t<C, E> const* p
    , size_t n
    )
      : len(n)
    {
      STLSOFT_ASSERT(N == n);
      { for(size_t i = 0; i != n; ++i, ++p)
      {
        ptr[i].name   = p->name;
        ptr[i].value  = p->value;
      }}
    }
    size_t size() const
    {
      return N;
    }
    string_case_item_t<C, E> const& operator
[](size_t i) const
    {
      STLSOFT_ASSERT(i < N);
      return ptr[i];
    }
    private: // Fields
      size_t const             len;
      string_case_item_t<C, E> ptr[N];
  };
} /* namespace ximpl */
```

I hope it's obvious what's going on. It's effectively equivalent to the loop of the hand-written solution shown earlier.

The next part is where it gets trickier. For various reasons – including the desire to incur no more costs than the hand-written form – I did not want to allocate memory. Consequently, **`string_cases()`** is a suite of ten overloads, handling arity from 1 to 10. I'll show you the implementation of the ternary version along with the supporting data structures (see Listing 4). (Note: if you're creating function (template) suites, you should either generate them with Python/Ruby or use a mix of layout and IDE keyboard macros to ensure that you're not actually writing them all by hand. That, as I'm sure you can appreciate, is a recipe for tedium-induced mistakes to creep in.)

The key to the problem is the trick of how to return an array from a function. As I'm sure you are aware, gentle readers, it's not valid C or C++ to return a naked array:

```
int[10] bad_wolf(); // Not allowed!
```

But it's very easy to achieve. Just put it in a structure:

Listing 4 (cont'd)

```
template<
  typename C
, typename E
>
inline ximpl::string_case_item_array_t<C, E, 3>
 string_cases(
  C const*  name0
, E         value0
, C const*  name1
, E         value1
, C const*  name2
, E         value2
)
{
  ximpl::string_case_item_t<C, E> items[] =
  {
      { name0, value0 }
    , { name1, value1 }
    , { name2, value2 }
  };
  return ximpl::string_case_item_array_t<C,
     E, STLSOFT_NUM_ELEMENTS(items)>(items,
     STLSOFT_NUM_ELEMENTS(items));
}
```

```
struct snarl
{
  int ar[10];
};
snarl bad_wolf();
```

All the templaty-looking magic in Listing 4 really amounts to nothing more than populating a matching-nary array of **`string_case_item_t<>`** (wrapped inside an instance of **`string_case_item_array_t<>`**).

As I mentioned earlier, it's a bit scrappy. There are several points at which there's an ostensibly runtime check of the size of the array, when it could be compile-time. And the so-called character-genericity is, for the moment, subverted by the use of **`strcmp()`**; this will be easily improved by using character/string traits.

More seriously, it only works with C-style strings. I already have a good idea how to apply string access shims [1] [3] to **`string_switch()`**, to enable its first argument to be any type representable as a string. But making the arguments to **`string_cases()`** generic will take a fair bit more deviousness, I think. I'd be happy to hear from anyone with ideas.

I should mention that Chris Oldwood, one of my reviewing friends, suggested that this is overkill (which has some truth) and offers nothing over the alternative of defining the cases in an array and passing that to a function similar to **`string_switch()`**. The reasons I want to persist with **`string_cases()`** are that it is still quite a bit more succinct, and the cases are defined where they're used. Also, once I've made it more generic, it'll be able to use case arguments of arbitrary string types, some of which may, if required, be variables. ■

Acknowledgements

I'd like to thank Steve Love, the CVu editor, for dealing patiently with this chronically deadline-slipped author, and my friends and review panel resident experts, Garth Lancaster and Chris Oldwood.

References

[1] *Imperfect C++*, Matthew Wilson, Addison-Wesley, 2004

[2] *The Mythical Man Month*, Frederick P. Brooks, Addison-Wesley, 1995

[3] 'An Introduction to FastFormat, part 2: Custom Argument and Sink Types', Matthew Wilson *Overload* 90, April 2009

Competency Scale

Ian Bruntlett thinks about expertise.

The differing levels of C++ expertise have been some discussed on accu-general where 0 is no knowledge and 10 is Bjarne Stroustrup. It has also been noted that the average programmer rates themselves higher than average. In order to gauge my own expertise in programming in general, I decided to expand on this. Here is the competency scale:

0	No knowledge
Novice	
1	Done a "Hello World" program from a magazine/web site article.
2	Novice/Tourist – relies on 'phrase books' (e.g. O'Reilly's books).
3	Novice – less reliant on books.
Practised	
4	Gaining confidence – books / man pages used for reference.
5	Average – knows the ins and outs of the language/topic.
6	Fluent – above average, becoming an expert.
Expert	
7	Expert.
8	Lead programmer.
9	Mentor.
10	Guru (e.g Bjarne Stroustrup for C++).

This scale can be used on accu-contacts etc to list the minimum acceptable competence for key skills. One particular use could be to illustrate the current skill levels. For instance, someone could have had C++(7) in the past but, due to working with other languages for a period of time, slip down to C++(Was 7, now 5).

There are other skills to be measured as well, not just programming skills.

Refactoring – Do you just copy and paste code into functions (0) or do you put it into suitable functions. (Similar things apply to classes and templates).

Other skills, left as an exercise for the reader, would be: Design, Coding, Literacy (reading about the topic), Literacy (writing about the topic), use of Standard Libraries, use of other related libraries, knowledge of the application/business domain.

My own ratings are in the table below. ■

Skill	Score
bash (Linux shell scripting)	3
awk	2
Python	2
Perl	3
C	7
C++	was 7, now 4
Creating solution domain libraries	was 8, now 6
Business domain (energy trading and forecasting)	4
Refactoring	7
Literacy (C++, reading)	6
C++ Builder	was 7 now 5
Visual C++ / MFC	2
Hardware troubleshooting	6
Guiding novice users	7
Boost	2
Algorithms	6
STL	5 was 6
Assembly Language	was 8 now 4

IAN BRUNTLETT

On and off, Ian has been programming for some years. He is a volunteer system administrator for a mental health charity called Contact (www.contactmorpeth.org.uk). As part of his work, Ian has compiled a free Software Toolkit (http://contactmorpeth/wikispaces.com/SoftwareToolkit).

Implication Assert

John Fraser describes a novel use of assert.

Assertions are very useful for documenting and checking software contracts; preconditions, post-conditions and invariants. They provide a functional, referentially transparent (i.e. stateless) expression that describes the state of a system at a particular point in time. The expression that is evaluated by an assert statement can include any number of unary or binary operators, for example, `Assert(!condition1&&condition2 j| condition3)`. There is however another useful binary operator called the **`implication`** operator that is not directly supported by most languages but can be formed by using negation (NOT) and disjunction (OR). In this article I will show you why the implication operator is useful and how to use it in an assert statement.

Notation

Before describing the implication operator and its use in assert statements it is necessary to define some notation that will be used throughout the rest of this article.

Symbol	Description	Example
$\vee$	Disjunction (OR): this acts like the **`||`** operator in C++	$p \vee q$
$\wedge$	Conjunction (AND): this acts like the **`&&`** operator in C++	$p \wedge q$
$\neg$	Negation (NOT): this acts like the **`!`** operator in C++	$\neg p$

The assert statement

First, let us look at the **`assert`** statement and describe its purpose and behaviour. The code below shows a method and a precondition defined using an **`assert`** statement.

```
void SomeMethod(int parm1, int parm2)
{
 assert( parm1 != 0 && parm2 != parm1 );
 ...
 // now parm1 and parm2 can be used in the body
 // of the method knowing that parm1 != 0 and
 // parm2 != parm1
 ...
}
```

To anyone reading the code shown above, the **`assert`** statement acts as clear documentation for the responsibility of the caller of **`SomeMethod`**; the caller of **`SomeMethod`** must ensure that **`parm1`** does not equal zero and that **`parm1`** does not equal **`parm2`**. Likewise the **`assert`** statement documents the responsibility of **`SomeMethod`** in that **`SomeMethod`** does not need to check that the values of **`parm1`** and **`parm2`** are valid before using them; the code in the body of **`SomeMethod`** after the **`assert`** statement can be read with the assumption that **`parm1`** and **`parm2`** contain valid values. Using **`assert`** statements in this way can simply the code of **`SomeMethod`** and the caller of **`SomeMethod`**. **`SomeMethod`** can be written without having to check that the values of **`parm1`** and **`parm2`** are valid before they are used. If the caller of **`SomeMethod`** ensures that the values of **`parm1`** and **`parm2`** meet the precondition before calling **`SomeMethod`** then the caller will not have to handle an exception resulting from invalid parameters passed to **`SomeMethod`**.

At runtime (typically in a debug build) the expression passed as parameter to the **`assert`** is evaluated and if the result is **`true`** then the assertion is valid and no action needs to be is taken. If the result of the expression is **`false`** then the action that occurs is typically platform dependent and configurable. Some implementations of **`assert`** will halt execution of the program invoking the debugger, alternatively an **`assert`** can throw an exception and/or log an error message to file or the console.

Assert with conditional

Now let us look at what problem the implication operator can solve with regard to **`assert`** statements. The author of the code shown below would like to make the assertion that if the **`interrupt`** variable is not equal to zero then the value of **`data`** should not be equal to zero. The point of the conditional statement around the assertion is that the code should only check the assertion that data does not equal zero if **`interrupt`** does not equal zero, in other words, if **`interrupt`** equals zero then we do not care if **`data`** is zero or not therefore the **`assert`** should not halt execution of the program.

```
void EventHandler( int interrupt, int data )
{
  if( interrupt != 0 )
  {
    assert( data != 0 );
  }
  ...
  ...
}
```

Whilst the code above will work, there are two issues with the way this code is structured. One of the issues is that the conditional **`if`** statement will not be compiled out when a release build is made; typically **`assert`** statements are not a part of the final executable and are compiled out when a release build is made. The other issue is that the assert statement does not represent a complete expression of what should be asserted, for the reason that a conditional is used before the assert. This makes the assertion difficult to read because the assertion is not only about **`data`** but also about **`interrupt`** and **`interrupt`** is not a part of the assert expression. It also makes the assertion difficult to parse, for example, a tool that performs static analysis or displays documentation about each method along with its software contracts would not know to include the conditional statement as a part of the precondition, it would just use the expression that is a parameter to the **`assert`** statement.

Implication operator

The implication operator is a binary operator meaning that it takes two operands as parameters. You have already seen an example of two binary operators in the notation table at the beginning of this article; the disjunction and conjugation operators both take two operands as input. The symbol for the implication operator is an arrow ($\rightarrow$). The truth table for the implication operator is shown below. Notice that when p is **`false`** as in the 3rd and 4th rows of the truth table, the result of $p \rightarrow q$ (p implies q) is always **`true`**; it is this behaviour that we will make use when forming the expression for an **`assert`** statement.

JOHN FRASER

John Fraser has been developing software for 20 years mainly in C++ and C#. He currently works for the Institute of Cancer Research in London writing software that will help medical staff visualize the effectiveness of treatments. John can be contacted at logic_cube@btopenworld.com

p	q	p→q
true	true	true
true	false	false
false	true	true
false	false	true

The implication operator is not directly supported by most languages but it can be represented in terms of other operators that are supported. The implication operator → can be represented in terms of *negation* (not) and *disjunction* (or). For example, the expression $p \rightarrow q$ (*p* implies *q*) can be expressed as $\neg p \vee q$ (i.e. not *p* or *q*). To prove this, the following table compares $p \rightarrow q$ with $\neg p \vee q$ and shows that the result of each expression is the same for each combination of *p* and *q*.

p	¬p	q	¬p∨q	p→q
true	false	true	true	true
true	false	false	false	false
false	true	true	true	true
false	true	false	true	true

Have a look at the second row of the truth table. Notice that the value in the column for $p \rightarrow q$ contains *false*. Having a look at the rest of the rows you can see that the only time the implication is *false* is when *p* is *true* and *q* is *false*, otherwise the implication is *true*. In rows three and four the value in the implication column is *true* but have a look at the values for *q*. In row three the value of *q* is *true* and in row four the value is *false*; this means it doesn't matter what the value of *q* is if the value of *p* is *false*, the implication will always be *true*. Now we can use the behaviour of the implication operator shown in the truth table to rewrite the **assert** statement using a single expression without the need for the conditional **if** statement around it.

```
void EventHandler( int interrupt, int data )
{
  assert( !(interrupt !=0) || data !=0 );
  ...
  ...
}
```

As you can see, the expression in the **assert** statement has been written based upon column four in the truth table, i.e. **assert(!(***p***) || (***q***))** where *p* is **interrupt != 0** and *q* is **data != 0**. Let us test the behaviour of the expression for two cases to see what it evaluates to for different values of *p* and *q*. If **interrupt !=0** is **true** and data != 0 is **false** then the negation of **interrupt !=0** achieved using the **!** (not) operator in the code above would evaluate to **false**, so **!(interrupt !=0)** evaluates to **false**. As **data != 0** also evaluates to **false**, the result of **!(interrupt !=0) || data !=0** would also evaluate to **false** and the assertion would halt execution of the program. This is the behaviour that we want; we only want the assertion to halt execution of the program if **interrupt** is not equal to zero and **data** equals zero. If **interrupt !=0** is **false** (i.e. **interrupt** equals 0) and **data != 0** is **false** then the negation of **interrupt !=0** evaluates to **true**. Now because in this case the **interrupt** variable equals zero we actually don't care what the result of **data != 0** evaluates to. In this instance **data != 0** is **false** meaning that the result of **!(interrupt !=0) || data !=0** evaluates to **true** and the **assert** will not halt execution of the program.

Although the code now uses an assertion that includes **interrupt** and **data**, the assertion is difficult to read, i.e. the intention of the assertion, which is to make use of the implication operator may be unclear. This can be solved by writing an **assert** macro that indicates the intention, making the assertion easier to read by formalising the premise and conclusion of the implication using two parameters.

```
#define assert_implication(p,q) assert(!(p)||(q))
void EventHandler( int interrupt, int data )
{
  assert_implication( interrupt != 0,
     data != 0 );
  ...
  ...
}
```

Using the macro the code is easier to read; the relationship between **interrupt** and **data** has also been made explicit due to the naming of the macro (**assert** implication). The expression can now be read as *interrupt != 0* → *data != 0*. Using the implication operator the *assertion* will only halt execution of the program in the case where **interrupt != 0** is **true** and **data != 0** is **false** (i.e. **interrupt** has some value other than zero and **data** is equal to zero). If **interrupt != 0** evaluates to **false** then we do not care about the value of **data** and so because of the use of the implication operator the assertion will evaluate to **true** and not halt execution of the program.

Summary

For *assertions* that are required to be evaluated conditionally, it is possible to use a conditional **if** statement followed by the actual assertion. However, this article has pointed out two issues with this approach and has shown a way for the conditional **if** to be removed and formed as a part of the assertion expression by making use of the implication operator. A macro has also been introduced that makes the intention of the assertion explicit with the result that the assertion is easier to understand. ■

A Game of Guesswork

Baron Muncharris sets a challenge.

Greetings Sir R-----. I trust that I find you in good spirits this evening? Will you take a glass of this excellent porter and join me in a little sport?

Splendid!

I propose a game that is popular amongst Antipodean opal scavengers as a means to improve their skill at guesswork.

Opals, as any reputable botanist will confirm, are the seeds of the majestic opal tree which grows in some abundance atop the vast monoliths of that region. Its mouth-watering fruits are greatly enjoyed by the Titans on those occasions when, attracted by its entirely confused seasons, they choose to winter thereabouts.

Having gorged themselves upon these fruits, the atrociously mannered Titans cast the scraps deep into the surrounding deserts leaving their stones scattered haphazardly therein.

Given the consequent lack of predictability of the location of opals it is small wonder that a talent for guesswork is so highly prized amongst their scavengers and that they are willing to expend so much of their spare time in practice of it.

Finding myself stranded in a camp of these hardy fellows after a misadventure involving a very great number of rabbits, upon which I have no desire to expand, I discovered that I have some small talent in their industry. During the course of an afternoon I chanced upon some several thousand stones to which I added some few thousand more at their game after our labour.

But I digress.

See here, I have placed two upturned cups on the table. Beneath one of them is a token and if you pay me a stake of 1 and 7/8ths of a coin you make guess which it is. If you are correct you shall win a prize of 1/8th of a coin.

Now this may strike you as a somewhat unworthy bounty but pray hear me out!

If you guess correctly you may play a second round of the game for the same stake. In this round there are three cups rather than two and the prize is doubled.

If you guess correctly again you may likewise enter into a third round in which the cups number four and the prize is triple that of the second round.

Play continues in this fashion with every correct guess giving you the option to pay your stake again and enter into another round in which the number of cups is increased by one and the prize multiplied by the number of cups employed in the round just played.

An incorrect guess brings the game to an end, but you may elect to begin again with another first round if you so desire.

When I explained these rules to that disreputable student acquaintance of mine he began blathering on, in his usual witless fashion, about how the harmonies of Sirius are oft lengthier than expected, although quite what bearing he imagines the endless dirges of the Canicular peoples might have upon this game entirely escapes me. Perhaps a touch of dog day sunstroke has accelerated the deterioration of his already meagre faculties.

Now come recharge your glass and think on your thirst for a wager! ■

Listing

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

Listing 1

```
size_t
rnd(const size_t n)
{
  return size_t(double(n) * double(rand()) /
     (double(RAND_MAX)+1.0));
}

void
play()
{
  const double stake = 1.0 + 7.0/8.0;
  double prize = 1.0/8.0;
  size_t cups = 2;
  double balance = 0.0;

  char again = 'Y';
  while(toupper(again)=='Y')
  {
    balance -= stake;
    const size_t cup = 1 + rnd(cups);
    size_t picked = 0;
    while(picked==0 || picked>cups)
    {
      std::cout << "Pick a cup from 1 to "
         << cups << " : ";
      std::cin  >> picked;
      std::cin.clear();
      std::cin.ignore(
         std::numeric_limits<int>::max(), '\n');
    }

    if(picked==cup)
    {
      balance += prize;
      prize *= double(cups);
      ++cups;

      std::cout << "Right! Balance = "
         << balance << std::endl;
    }
    else
    {
      prize = 1.0/8.0;
      cups = 2;
      std::cout << "Wrong! Balance = "
         << balance << std::endl;
    }

    again = '\0';
    while(toupper(again)!='Y'
       && toupper(again)!='N')
    {
      std::cout << "Play again? [Y/N] : ";
      std::cin  >> again;

      std::cin.ignore(
         std::numeric_limits<int>::max(), '\n');
      }
    }
  }
```

On a Game of Nerve

The Baron's acquaintance performs his analysis.

The Baron's latest game consists of up to four turns throwing a pair of dice and costs nine coins to play. After each turn the player may elect to stop playing and collect their sum as winnings.

On hearing these rules, it immediately occurred to me that he should only continue the game if he has thrown a sum less than that he might expect to win in future turns.

We can thus reckon the expected winnings *before* throwing the dice by considering the expected winnings after throwing them conditional upon the cases of having done better and having done worse than expected in future turns.

I explained this insight to the Baron, but fear I may not have done so with sufficiently clarity since I was struck with the impression that he had not fully understood me. Hopefully I shall do better with this exposition!

Let us assume that the expected winnings when there are a number, let us say n, turns left is equal to e_n. The expected winnings when there are n+1 turns left can then be figured as the sum of en times the probability that we roll less than or equal to e_n and the expected sum of a throw given that it is greater than e_n times the probability that we make such a throw.

We write this as

$$e_{n+1} = p(x \le e_n) \times e_n + p(x > e_n) \times E[x|x > e_n]$$

where x represents possible sums of the pair of dice, p the probability of observing the event that follows it and E the expected value of the symbol before the bar assuming it satisfies the condition after it.

We can figure this conditional expectation with the formula

$$E[x|x > e_n] = \frac{\sum_{y > e_n} y \times p(x = y)}{p(x > e_n)}$$

where the capital sigma stands for the sum of the terms that follow it for every value y greater than e_n.

The expected sum of a roll of a pair of dice is 7, so

$$e_1 = 7$$

We can therefore figure the expected winnings in a game of two turns with the formula

$$e_2 = p(x \le 7) \times 7 + p(x > 7) \times E[x|x > 7]$$

Noting that the probabilities of rolling sums from two to twelve are equal to

Sum	2	3	4	5	6	7	8	9	10	11	12
Probability	1/36	2/36	3/36	4/36	3/36	6/36	5/36	4/36	3/36	2/36	1/36

we can rewrite this as

$$e_2 = \left(\frac{1}{36} + \frac{2}{36} + \frac{3}{36} + \frac{4}{36} + \frac{5}{36} + \frac{6}{36}\right) \times 7 + \left(\frac{5}{36} + \frac{4}{36} + \frac{3}{36} + \frac{2}{36} + \frac{1}{36}\right) \times \frac{\left(8 \times \frac{5}{36} + 9 \times \frac{4}{36} + 10 \times \frac{3}{36} + 11 \times \frac{2}{36} + 12 \times \frac{1}{36}\right)}{\left(\frac{5}{36} + \frac{4}{36} + \frac{3}{36} + \frac{2}{36} + \frac{1}{36}\right)}$$

$$= \frac{21}{36} \times 7 + \left(\frac{40}{36} + \frac{36}{36} + \frac{30}{36} + \frac{22}{36} + \frac{12}{36}\right)$$

$$= \frac{147}{36} + \frac{140}{36} = \frac{287}{36} = 7\frac{35}{36}$$

Given this result, we can figure the expected winnings when there are three turns remaining and, given that, the expected winnings when there are four.

In doing so, we find that the expected winnings in the Baron's game are 8 coins plus 7349 of 7776th parts of a coin, or 427 of 7776th parts of a coin less than the cost to play. The game is thus biased in favour of the Baron and as such, unless Sir R----- had had a particularly keen appetite for a wager, I should have advised him to refrain from play.

Incidentally, I recently ran into Monsiour L----- and, on proudly recounting to him the tale of how I and my fellow students played the Baron's game of strategy for 48 hours straight to demonstrate that it could always have been won by Sir R-----, he pointed out that this was obviously true if Sir R----- plays so that one half turn of the board leaves the coins in the same positions, albeit with tails exchanged for heads.

If, after such a turn of the board, Sir R----- flips the coins over it would have no effect on the state of the board and hence both players must control the same number of squares. Sir R----- must therefore always be able to force a draw.

It is with some shame that I must admit that I have not the nerve to report this to my fellow students for fear that they shall vent upon me their frustration at having utterly wasted their time. ■

Listings

- Listing 1: The probabilities of sums
- Listing 2: The conditional expectation of sums no less than the minimum
- Listing 3: The expected winnings in a game of *n* turns

Listing 1

```
double
p_equal(unsigned long i)
{
  static const unsigned long n[13] =
     {0, 0, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1};
  return i<=12 ? double(n[i])/36.0 : 0.0;
}

double
p_less_equal(unsigned long i)
{
  static const unsigned long n[13] =
     {0, 0, 1, 3, 6, 10, 15, 21, 26, 30, 33, 35,
     36};
  return i<=12 ? double(n[i])/36.0 : 0.0;
}
```

1st Annual UK Vintage Computing Festival

This was held at the National Museum of Computing, Bletchley Park on 19th and 20th June 2010.

On the 19th of June, Richard (my partner) and I went to the 1st Annual UK Vintage Computing Festival. We'd been looking forward to this for the best part of 9 months and so we got an early night and an early start (for us at least). Fortunately, the lovely thing about Bletchley is that it's only about 30 minutes by train from London Euston Station so by lunchtime we were strolling up the drive to the Mansion House.

First stop had to be the vendors' tents but I'm afraid they were something of a disappointment. If you were in the market for a Sinclair or Amiga or Acorn, then you were in luck but anything more exotic on display was not for sale. :-(That said, there were some very interesting displays from enthusiasts, particularly the riscos guys [1] who were rocking some hot stuff on beagleboards [2]. Sadly, yet again we were thwarted in our desire to acquire this nifty bit of kit. The other interesting thing that I learned is that the Amiga OS is still going strong and seems to have a dedicated cadre of followers [3].

Having exhausted the shopping possibilities, we took a spin through The National Museum of Computing. Since last year's Autumn Conference at Bletchley, TNMOC has been making great strides in improving its offering. They have now added a room on networking and a room on analogue computing with some fascinating early punchcard equipment and early computers. In addition, the WITCH project [4] is moving apace and should be nearing completion by the time the next ACCU/Bletchley Autumn Lecture happens (mark it in your calendar for November 6th!!!).

Before we left (having neglected to buy tickets to see the evening entertainment – a show by Orchestral Manoeuvres in the Dark), we ducked into the ABC Enigma Cinema to see a 25 minute film about the LEO. It was fabulous! Not only was the film extremely well done but it was the original reel-to-reel from 50 years ago being played on an utterly enchanting piece of original kit. I definitely need to spend more time examining the vintage projectors on my next visit!

Anyway, to summarise, the festival is a great start and I can imagine it growing into a valuable resource for vintage computing enthusiasts in the future.

Asti Byro

References

[1] riscos: http://www.riscos.com/
[2] Beagleboards: http://beagleboard.org/
[3] Amiga OS: http://www.amiga25.com/
[4] The WITCH project: http://www.tnmoc.org/126/section.aspx/105

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

If you've been out-and-about and have come across something you think other readers might find interesting, share the experience...especially if it's something they can look out for next year!

And if you attend any of the regional meetings, let your fellow ACCU-ers know what they are missing!

On a Game of Nerve (continued)

Listing 2

```
double
conditional_expectation(unsigned long min)
{
  double p = 0.0;
  double x = 0.0;

  for(unsigned long i=min;i<=12;++i)
  {
    p += p_equal(i);
    x += p_equal(i) * double(i);
  }
  return x/p;
}
```

Listing 3

```
double
expected_value(unsigned long n)
{
  if(n==0)  return 0.0;
  if(n==1)  return 7.0;

  const double        play_expected(
     expected_value(n-1));
  const unsigned long stick(ceil(play_expected));
  const double        stick_expected(
     conditional_expectation(stick));

  const double p_play(p_less_equal(stick-1));
  const double p_stick(1.0-p_play);

  return p_play*play_expected
     + p_stick*stick_expected;
}
```

Regional Meetings

A round-up of happenings across the country.

ACCU London

Two regional meetings have been held in London since the last CVu was published.

May 2010 – Kevlin Henney: Rethinking unit testing in C++

The London branch of the ACCU met up once again in May at one of the SkillsMatter offices for a presentation by Kevlin Henney. There was a good turnout of around 20, with a few new faces to boot. That wasn't overly surprising given that Kevlin is a renowned speaker – I've seen him speak a number of times at the ACCU conference and found his material to be very insightful, with a dash of dry humour and controversy to keep you on your toes.

For the most part I think the C++ suffix in the presentation's title was a little superfluous. Yes, he was demonstrating with examples in C++, but much of the message was clearly language agnostic. The first half illustrated the evolution of unit testing from a bunch of random chunks of 'test' code through to an organised set of tests that had a significant air of 'Formal Specification' about it. The initial tests were really just a stream of consciousness laid out sequentially in a single scope with no attention paid to structure and consequently future maintenance costs. This was then evolved into a procedural style whereby the tests were partitioned around the (member) functions under test. One of the main issues with this style is that it's impossible not to use the same functions in other test methods, which makes the idea of isolated tests and order independence somewhat moot.

static languages require *as much* test code as dynamic ones, because it's difficult *not* to test for type-safety even though the compiler does that for you

The next progression was coined 'contextual' whereby the partitioning was biased towards 'behaviours' or 'features' rather than the implementation. By this point the names of the tests started to follow natural language rather than the more terse method naming we use daily and Kevlin described how some developers find this transition difficult because they're not used to such expression. Although not as rigid, the style was reminiscent of the BDD Given/When/Then form with the test name resembling a proposition rather than an exclamation. The judicious application of macros meant that the source file was taking on a more formal shape with a SPECIFICATION at the top, followed by a series of PROPOSITIONs. But, not only did the code appear more formal, the output from the tests had a more intention revealing feel about it. Nat Pryce and Steve Freeman gave a talk a few months back about TDD in which they highlighted the importance of the test diagnostics and these ideas certainly enhanced that message.

A brief diversion into unit testing C++ code with NUnit & C++/CLI then followed. Once again the aid of a few macros helped to hide away the test method attributes and static Assert invocations so that visually the code appeared just like the native versions. Kevlin admitted that there are limitations to this technique, but it was a good first approximation that showed a nice meshing of the two 'meta' techniques – .Net reflection and C/C++ pre-processing.

For his grand finale Kevlin showed how he could capture the expression being asserted. However this was not only in textual form, but also in programmatic form so that the output for a test failure would display the failed expression and the values of the variables used within that expression. Naturally the audience wanted to see behind the curtain and we quickly found ourselves staring into Expression Template territory. Kevlin only had a partial implementation due to time constraints, but there was already a significant body of template code. The most interesting aspect was the choice of operator used to allow the code to work its magic by 'capturing' the leftmost part of the assert expression – it needed to be overloadable, practically unused and higher than those you would normally find inside assert expressions. The proverbial rabbit that he pulled out of the hat was the `->*` operator which I readily confess to not knowing, but promptly Googled when I got home.

Not that I departed for home the moment the bell went as there was still the matter of post-discussion beer which was satisfied by The Slaughtered Lamb. Fortunately we had a few of the non-members join us in the pub afterwards which gave us ample opportunity to slip The King's Shilling into their pint…

June 2010 – Giles Thomas: IronPython at large

12 or 13 people gathered at Cititec's offices near Old Street for this month's talk, given by Giles Thomas, co-founder and managing director of Resolver Systems. IronPython is a version of Python for the .Net environment, and Giles gave us a quick demonstration of it by launching a window by calling into Windows.Forms from a (Iron)Python console, and adding controls and event handlers while the program was running! This was the first time for many years I'd seen somebody patching running code...

The main product at Resolver Systems is a spreadsheet called Resolver One, and Giles introduced it by talking about how many MS Excel 'applications' he'd seen and supported where the VBA code had become so unmanageable as to make any changes or improvements almost impossible. Resolver One is designed to expose the scripting capability directly, thus making Python the language behind a spreadheet. More than that, the spreadsheet /is/ the Python code. When you edit a cell and add a formula, the code window updates showing the Python code, so you can edit it and define your own functions etc. Plus, IronPython exposes all of .Net too. Resolver One is also written entirely in IronPython, a fact many of those attending found surprising. Giles explained that they had initially thought they would encounter performance issues, and have to write libraries in C# or perhaps C to handle possible bottlenecks, but it turns out they've not encountered the need for that yet.

As a development house, they use Extreme Programming principles, and one of the 'key facts' was the ratio between lines of code and lines of test code was on the order of 3 or 4 test lines to 1 line of app code. An important part of this is the code is developed using stricty TDD. Giles quoted some stats gathered in statically typed languages, where the ratio is approximately the same. A common perception of dynamic languages like Python is the apparent need for more tests to make up for the lack of compile-time type-safety. Giles explained that this was not their experience, and that static languages require *as much* test code as dynamic ones, because it's difficult *not* to test for type-safety even though the compiler does that for you. We didn't get time to explore this further, but I wonder what other teams' experiences are in this area.

Alas Giles was unable to join us for a drink after the talk, but he had brought along a colleague from Resolver Systems' development team. Glenn was

Desert Island Books

Phil Bass heads for the life boats.

Phil Bass has been an ACCU member for well over ten years. I have only met him a couple of times, outside of accu-general, at the conference. It has always been a matter of pride for me that someone as knowledgeable as Phil is a firm supporter of the mentored developers.

I am also very happy to say that Phil tells another very imaginative story behind his desert island books.

Phil Bass

Attention all passengers ...

The ship is sinking. Three days into a round-the-world cruise and this happens. It was going to be the trip of a lifetime, an adventure, something to look back on in my retirement. I can't believe it.

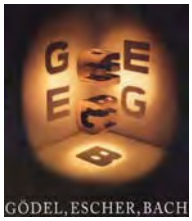
There's a rumour we hit an iceberg. This far south? A certain amount of panic is forgiveable, but that's insane. Of course, we couldn't get anything sensible out of the crew. 'There's a technical problem', they said, yesterday. 'There's no need to worry. We're going to stop off at the next port for some minor repairs.' Apparently it was a problem with the engine or the navigation systems or the heating/ventilation; everyone had a different story.

If you were a book title, which would you be?

But the captain has just announced that the ship is taking on water and that we won't reach port in time. All passengers are to transfer to the lifeboats. There's an island on the horizon; we'll set up camp there until we can be rescued. Unfortunately, both the communications and navigation systems are severely damaged. We have only a rough idea of where we are and we haven't been able to radio for help. We may be marooned for some time.

Heroes and aspirations

We can take some clothes and a single piece of hand luggage with us. I have quite a few books and CDs in the cabin – it was supposed to be a three month voyage – and I can't bear to see them all go to the bottom of the sea. Which ones shall I choose?



As I started to pack my bag I remembered a conversation I had at dinner on our first night on board. There was a teacher at our table with a passing interest in psychology. She asked me, 'If you were a book title, which would you be?' Strange question, I thought. The sort of question you might hear at a speed-dating session. I wondered, briefly, if she was evaluating me for a holiday romance, but no, that was just my ego talking. Then it struck me that if any book title summed up who I am it was probably *Gödel, Escher, Bach: An Eternal Golden Braid*.

To me Gödel's mathematical theorems, Bach's music and Escher's drawings are works of astonishing originality and beauty. I'm a 'vertical', straight-line thinker and I simply can't imagine where they got their ideas. I think that is why Gödel, Escher and Bach all have a place in my personal catalogue of great men alongside the likes of Newton, Einstein, Babbage and Darwin.

I grabbed *Gödel, Escher, Bach* and thrust it into the bag. It was a book I'd been meaning to get for years, but have never read. I'd bought it specially for the voyage and it must be saved at all costs!

Regional Meetings (continued)

able to enlighten us on much more about the product and his team over a beer or two.

Many thanks to Stefano Cicu and Cititec for hosting the event and providing us with the facilities for another very interesting presentation.

Software East

Paul Grenyer provides an overview of the latest meeting in the east.

Allan Kelly : The falling off a log theory

This was this first Software East presentation I have attended. It was hosted at the Redgate offices just off the A14. They were very nice, if not as technologically advanced as Morgan Stanley. It just so happened that Allen Kelly, the speaker, was arriving as I phoned him from the car park, so we wondered in together to find no receptionist and a sign directing us to the Seagull Suite on the first floor. From the first floor there was no indication of where the Seagull Suite was, so Allan gave Mark Dalgarno, the event organiser a call and he showed us to the suite via the 'SQL Servery', Red Gate's appropriately named cafeteria .

The presentation was scheduled to take place between 6.30pm and 8.30pm. Two hours is a long time for these events, even though I can imagine Allan speaking for two hours without a problem. I found out that the first half hour is for networking and buffet eating, so I tucked in and chatted to Allan and Pete Goodliffe.

Allan spoke about setting up your own business. He believes that it's as easy as falling of a log and it is! He pointed out that all you need to do is pay an accountant and they set it up and do all the legal and financial stuff for you. I know this to be true as I have done it.

Allan's main thrust though was that there needs to be at least two of you, a techie and a salesman. That way you may just make it to through your first year. Again, the same thing had occurred to me. Whenever I have thought of setting up my business and going it alone I've always had two problems, I'm no good at selling and I don't have an idea for a product. A salesman wouldn't necessarily have an idea for a product, but they should be able to sell!

Allan went on to explain how he missed out on making a billion dollars by not developing an idea for a product he had early in his career that someone else later did. I did have some sympathy, but mostly envy.

Allan over ran and after questions at the end he finished about 8.30pm. The most significant question asked from my point of view was where do us techies find a salesman and, unfortunately, no one really had an answer.

The rest of the group stayed to polish off the buffet and do some more networking while I sloped off for the drive back to Norwich.

A few objects more

I suppose a man is defined to a large extent by what he does for a living. I've been a programmer for all my adult life and no book collection of mine would be complete without one or two books on the noble art of computer programming. Knuth, however, is not in the cabin and I wouldn't take him to a desert island if he was. I'm going to enjoy my stay on that rock out there if I can and I need something lighter than that.

The first book that taught me something about program design was Michael A Jackson's *Principles of Program Design*. I remember the idea that the structure of the code should follow naturally from the structure of its inputs and outputs, which made sense in the days when batch processing was the norm. Later I read some books on structured design by Tom De Marco, Ed Yourdon and others. I learnt about data flow diagrams and, perhaps the most important lesson of all, that we should strive to minimise coupling and maximise cohesion. Although these taught me many things no one book from this period stands out in my memory.

Then came object-oriented design, a new paradigm that spawned a plethora of books. In spite of that embarrassment of riches there is one OO book that had a significant impact on my approach to software development: Bertrand Meyer's *Object-Oriented Software Construction*. It starts by considering what 'quality' means when applied to software, lists some criteria by which we can judge the quality of our code and states some principles that must be adhered to in order to build good quality software. It then describes a programming language designed to support those principles. That language is Eiffel.

Eiffel is both an object-oriented language and a programming environment. The language has classes, parameterised types, exceptions and support for preconditions, postconditions and invariants. The environment provides a garbage collector, compiler/linker/dependency manager and several other tools to aid the Eiffel programmer. In short, it has pretty much everything today's programmers have come to expect. It's a bit too object-oriented for my taste and has a number of features that I don't like, but I think it deserves to have a much larger following. I've never used Eiffel, Java or C# (no, really!), but given a choice from those three I'd try Eiffel first simply because of the clarity of the reasoning in Meyer's book.

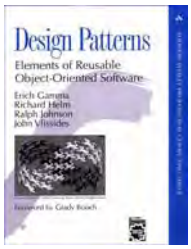
Object-Oriented Construction goes into the bag.

Pretty patterns

The tannoy blares out again. 'Will all passengers, please, assemble on the sun deck for evacuation.' I have just a few minutes to choose what else to pack.

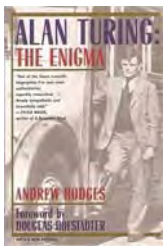
Instinctively, I reach for *Design Patterns – Elements of Reusable Object-Oriented Software*. With an Escher print on the cover and a foreword by Grady Booch (one of the foremost proponents of object-oriented programming) it sits very comfortably alongside my earlier choices. But, more than that, it is the book that has had the greatest influence on my approach to programming.

By the time *Design Patterns* was published I had been programming for 20 years and, although I wasn't consciously aware of it, I had built up a small catalogue of tricks and techniques that could be applied to a wide range of programs. As I read each chapter I kept coming across problems I'd wrestled with in the past and solutions that I recognised from my own accumulated experience. This book brought them all together, gave them names, clearly defined the problems and explained when and why a particular solution works. It was a revelation.



Man and machine

There is no clock in the cabin, nothing to measure the passage of time, but I can feel the seconds ticking by as I scan the bookshelf. It's getting hard to make decisions.

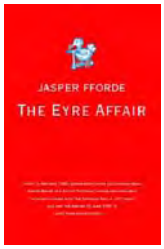


'No more technical books', I tell myself. There won't be any opportunity to write software where we're going now. That narrows down the choice enough for me to pick *Alan Turing – The Enigma*, a biography by Andrew Hodges.

It's quite a few years since I read Hodges' account of the life and work of Alan Turing, so my recollection is rather hazy. It does include some technical information, including a concise description of Turing machines, but it is mainly a sympathetic exploration of Alan Turing himself. As Douglas Hofstadter said in an early review, '... it is hard to imagine a more thoughtful and warm biography than this one'. The title, of course, simultaneously refers to Turing's work on the German Enigma machines, to the necessarily secretive nature of a gay man working for the government in the 1940s and to the uncertainty surrounding Turing's suicide.

Pure escapism

Glancing through the port hole I can see a calm blue sea, the sun glinting off the gentle waves. For a moment a profound sense of peace and tranquillity pervades my thoughts. But something about that view through the window isn't right. There's no horizon! With the cold logic of Sherlock Holmes I deduce that the ship is listing noticeably now.



For the first time the full implications of our predicament sink in. It could be hell on that island and I'm going to need something that will transport me, metaphorically speaking, to a better place. Jasper Fforde's *The Eyre Affair* comes to mind immediately, I search for its bright red cover and shove it hastily into the bulging bag.

But something about that view through the window isn't right. There's no horizon!

The Eyre Affair is set in an alternative universe in which literary debate is so fierce that it leads to gang wars and murder. In this world 'Jane Eyre' ends (lamely) with Jane accompanying her cousin to India to help with his missionary work. The villain of *The Eyre Affair*, Acheron Hades, uses a Prose Portal to enter works of fiction, threatening to kidnap their characters as a form of blackmail. It falls to literary detective, Thursday Next, to pursue Hades into 'Jane Eyre' and stop him. Eventually she succeeds, but in the process the ending is changed so that Thornfield Hall burns down, Rochester's mad wife dies and Rochester himself is badly injured. Returning to her real world Thursday Next discovers that public reaction to the new ending is positive, but her employers are not best pleased with her efforts and the book ends with Thursday facing an uncertain future.

The Eyre Affair is a wild, wacky, witty and extremely funny book in the style of Terry Pratchett. Indeed, Pratchett himself commented, 'Ingenious – I shall watch Jasper Fforde nervously'.

Watching the birds

The engines have stopped. There's just time to grab a couple of CDs and then I must go. Apart from a fascination with science in all its forms, my other passion is music. My CD collection covers a fairly broad spectrum; there are rock, pop, folk, jazz and classical albums that I really treasure. Leaving them behind will be heart-breaking but, strangely, it's not too hard to select just two to preserve my sanity on the island.

A sea bird flies past the porthole as I pull *Through the Window Pane* by Guillemots from the rack. For those who don't know Fyfe Dangerfield's compositions they're hard to

describe because they don't fit into any well-defined category. Many of the tracks would make excellent film soundtrack material, but that's not what they are. On my web site I say: 'This has everything: memorable melodies, irresistible rhythms, sweet harmonies, epic arrangements; sometimes all in the same song'. Every one of the 12 tracks seems to be a window into Fyfe's varied emotions. There's wistfulness, sadness, anger, despair, joy, love, playfulness and even a touch of humour. Very few artists have that wide a repertoire and very few bands can provide a vehicle for expressing it as well as Guillemots.

There's wistfulness, sadness, anger, despair, joy, love, playfulness and even a touch of humour

Five-pointed star

There's a knock on the door. A middle-aged man in a smart nautical uniform politely tells me I must vacate the room and join the other passengers on deck. With trembling fingers I pick out *Light Flight: The Anthology* by Pentangle and stow it hastily in my sanity bag.

Quoting from my web site again, *Light Flight* contains 'some of the most beautiful folk songs performed by the most accomplished folk/jazz group there's ever been'. By adding this album to *Window Pane* I can cover a reasonably large subset of musical styles and, at the same time, have songs whose poignant and exquisite beauty never fails to move me.

Back in the 70s when I was a student I went to see Pentangle at the New Theatre, Oxford. It was a disappointing performance and they split up shortly afterwards. The following year (I think) a friend of a friend invited me to a private gig at St. Catherine's college where John Renbourn and Jacqui McShee (both ex-Pentangle) were to play. It was an intimate setting, ideally suited to John's fine acoustic guitar playing and Jacqui's clear, mellow voice. It was a magical evening and my Pentangle CD reminds me of that night, too.

Regrets – too few to mention

As I lower myself down into the lifeboat I can't help thinking about the books and CDs I left behind, either on the ship or back at home.

There are many programming books that have been invaluable to me in my career. The single most important book for me as a C++ programmer is, of course, *The C++ Programming Language* by Bjarne Stroustrup. Scott Meyers' *Effective C++* and *More Effective C++*, Herb Sutter's *Exceptional C++*, Andrei Alexandrescu's *Modern C++ Design* and Vandevoorde & Josuttis' *C++ Templates* have all been of direct practical use in my professional work. *Generative Programming* by Czarnecki and Eisenecker has broadened my programming horizons. I might have chosen any of those for a desert island with all mod cons including an internet connection, but I'm not going to need them on that God-forsaken rock in the distance.

I would have liked to have brought some old favourites for bedtime reading. Something by John Wyndham, Arthur C Clarke, Ray Bradbury or Iain Banks, perhaps. *Alice in Wonderland* or *The World of Pooh* would add variety. Then there are the epic stories of Tolkein or Stephen Donaldson. Sandi Toksvig writes entertaining novels. And there are more Jasper Fforde books, not to mention lots of Terry Pratchet.

I shall sorely miss listening to Elbow performing *The Seldom Seen Kid*. I've played albums by Genesis, Yes, King Crimson and Soft Machine so often that they will never be forgotten, but I shall pine for them just same. Of the jazzier CDs in my collection remembering those by Back Door, Weather Report and Brand X brings a tinge of regret. Then there are odd ones like Tom Griesgraber's *A Whisper in the Thunder* (atmospheric Chapman Stick music), Gorillaz *Demon Days* and the free folk song downloads from Kray Van Kirk. I could go on, but I'm getting a lump in my throat.

No, I wouldn't swap any of those for the five books and two CDs in the little bag I'm carrying. As I hand the bag to the lifeboat crew for safe keeping I hear myself saying, 'Careful with that, mate; it's precious.'

The books

Gödel, Escher, Bach: An Eternal Golden Braid, Douglas Hofstadter.

Object-Oriented Software Construction, Bertrand Meyer.

Design Patterns – Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Alan Turing – The Enigma, Andrew Hodges.

The Eyre Affair, Jasper Fforde.

The albums

Through the Window Pane, Guillemots.

Light Flight: The Anthology, Pentangle.

Next issue: Chris Oldwood

What's it all about?

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

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

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

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

Code Critique Competition 64

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. A book prize is awarded to the winning entry.

Thanks to Louis Lavery, who wrote:

> Fixing some one else's, often poorly written, code does not appeal to me, sorry, but that's how it is (and you did say 'I'm sure you all have things to say about this code').
>
> So, I'd like to suggest an alternative, something that does appeal to me and, I feel sure, will to others. Well I would say that, being as it's my idea!
>
> The idea is to state a problem together with a solution and ask for improvements (which, in some cases, might lead to a full rewrite due to taking a different view or to generalising and in others to the conclusion there's no way to improve it).
>
> I reckon this'll have a greater appeal than the 'Code Critique' as it's a natural thing for programmers to do. You see a short piece of code, you know what it's supposed to do (and it does do it) but there's something about it that makes you think it can be done a little better, maybe the improvement would just be that the code is easier to under stand[1].
>
> Of course trying to think up bits of code like that isn't easy, probably nigh on impossible. But such code does exist. We all come across it in our daily work.

He also supplied an example of such a piece of code, which I intend to use in the next critique to see how the idea fares. Maybe this appeals to you, and you have some suitable code to send me for a future issue!

Last issue's code

We have a little piece of straight C code this time (Listing 1).

The following program is designed to sort a list of names, but does something very strange. Here is what I get when I try to run it:

```
C:> cc63
Anthony Hopkins
John Guilgud
Michael Caine
Charlie Chaplin
^Z
0: Caine Anthony
1: Charlie Chaplin
2: Hopkins Guilgud
3: Michael John?O3
```

Help!

Critiques

Matthew Wilson <stlsoft@gmail.com>

Version 0:

```
Anthony Hopkins
John Guilgud
Michael Caine
Charlie Chaplin
^Z
0: Caine Anthony²²²²
1: Charlie²²²² Chaplin
2: Hopkins Guilgud
3: Michael²²²² John²²²²²²²
Press any key to continue . . .
```

Given a specification that assumes each line of input can always be guaranteed to contain at least one space and will never contain more than 79 characters, then there are only two faulting defects and three lurking defects.

Listing 1

```
/**
 * Program to read first_name last_name
 * and print sorted. See also cc50
 */
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
typedef
int (*compar)(const void*, const void*);
typedef struct name
{
  char *ln; // last name
  char *fn; // first name
} name;
// order last names before first names
int compare(name *n1, name *n2)
{
  int t = strcmp(n1->ln, n2->ln);
  if (!t)
    t = strcmp(n1->fn, n2->fn);
  return t;
}
int main()
{
  char line[80];
  struct name *names = NULL;
  int n = 0;
  int i;
  while (gets(line))
  {
    struct name *nm;
    char *p;
    n = n + 1;
    names = (name*)
      realloc(names, n * sizeof(name));
    nm = names + n - 1;
    p = strstr(line, " ");
    nm->fn = (char*)malloc(p - line);
    strncpy(nm->fn, line, p - line);
    nm->ln = strdup(p+1);
  }
  qsort(names, sizeof(name),
    n, (compar)compare);
  for (i = 0; i < n; i++)
  {
    printf("%i: %s %s\n",
      i, names[i].fn, names[i].ln);
  }
}
```

ROGER ORR

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

Faulting defects

Let's consider the two faulting defects first.

Faulting Defect 1 – corrupted surname

The use of **strncpy()** does not result in a nul-terminated copy of the source string

```
nm->fn = (char*)malloc(p - line);
strncpy(nm->fn, line, p - line);
```

changed to

```
nm->fn = (char*)malloc(1 + (p - line));
memcpy(nm->fn, line, p - line);
nm->fn[p - line] = '\0';
```

Now the output is

```
Anthony Hopkins
John Guilgud
Michael Caine
Charlie Chaplin
^Z
0: Caine Anthony
1: Charlie Chaplin
2: Hopkins Guilgud
3: Michael John
Press any key to continue . . .
```

Faulting Defect 2 – mixed names after sort

The arguments to **sort()** are the wrong way around:

```
qsort(names, sizeof(name), n, (compar)compare);
```

changed to

```
qsort(names, n, sizeof(name), (compar)compare);
```

Lurking defects

The three lurking defects all pertain to the exhaustibility of memory:

- If **malloc()** fails, **nm->fn** will be **NULL**, and the subsequent **memcpy()** / **strncpy()** call will produce undefined behaviour (hopefully a fatal fault)
- Same for **strdup()** and **nm->ln** [there's more to say about this later ...]
- The same for **realloc()** and names. Furthermore, the direct assignment to **names** means that even if it's checked, the previous memory block held by **names** will be lost. It's customary to **realloc()** to a new (temporary) pointer, check the result, and if successful then assign to the regular holding pointer.

Other issues

There are other issues, divided into serious and benign/trivial. First, the serious:

- the result of **strstr()** is not checked, so an input line that does not contain a space will cause undefined (and likely fatal) behaviour in the subsequent evaluation and copying of name information
- the program will very likely overflow, due to the use of an input buffer of only 80 characters, at which point undefined behaviour will occur. Changing this to a large number reduces the chances of it happening in normal use but does not remove the vulnerability. **gets()** should not be used
- **strdup()** is non-standard
- the use of three lines (two in the original) to copy the forename is ugly: too much work being done in the application logic. This should be factored out into a helper function, e.g. **strndup()**, as in:

```
/* assuming p has been checked */
nm->fn = strndup(line, p - line);
```

The benign issues are:

- the use of **strstr()** over **strchr** is unnecessary. Could instead be **strchr(line, ' ');**
- the name **name** for the type is parlously unambiguous. Why not **name_t**
- the names **fn** and **ln** are unnecessarily terse. Why not **forename** and **surname**?
- the **compare()** function should take its arguments as pointers to **const**. (This is only benign in so far as it will not cause any problems in the current implementation. If changed, it could be faulting/lurking.)
- the syntax **if(!t)** implies that **t** is treated as boolean. Rather, the result is effectively tri-state [>0, 0, <0], and I find the boolean suggestion confusing. It should be explicitly **if(0 == t)**
- none of the memory is cleaned up at the end of the program. On systems that clean up after processes exit, this is not a problem. But if the code is used on a system that does not do so, leaks would occur. This goes from being possible to being a near certainty if the code is transposed into a larger program, that may live for longer and for which memory leaks would be a practical problem.
- rather than casting from one function type to another, I prefer to pass an existing **compare_void(void const*, void const*)**, which would do the casts and invoke a **compare_name(name_t const*, name_t const*)**, defined as per the current **compare()** function. I believe that's much less prone to accidental inappropriate casting.

Summary

I've offered advice on what to do about all problems save one: the overflow of **gets()**. Whenever I use C for reading lines from **FILE*** streams, I avoid the Streams library like the plague, instead – pardon the plug – using the **cstring_readLine()** function from my cstring library (http://synesis.com.au/software/cstring).

This would look something like:

```
cstring_t line = cstring_t_DEFAULT;
CSTRING_RC rc;

while(CSTRING_RC_SUCCESS ==
  (rc = cstring_readline(stdin, &line, NULL)))
{
  … // process each line, via ptr and
    // len members
  cstring_truncate(&line, 0);
}
if(CSTRING_RC_EOF != rc)
{
  // deal with error other than EOF
  …
}
cstring_destroy(&line);
```

[cstring_readLine() handles all the buffer issues]

So, were I to have to write this program for a production system I'd use **cstring** for the input, define **name_t** as containing two **cstring_t** members, write helper functions for splitting the **cstring_t** line into **name_t**'s two members, handle all possible allocation failures, and detect (and stop) if an invalid line was read.

Paul Stephenson <p.j.stephenson@gmail.com>

So, you're the new developer on the team, are you? And I see you've been given that pesky 'sort actors' names' utility that nobody else wanted to do. Well, thanks for coming to me with your bug report.

Firstly, I'll see if I can reproduce your error, and then we'll try to fix the bugs so that the utility behaves correctly as far as we can tell. Finally, I'll give you some tips on your style, and how we keep code maintainable in this team.

Reproduce it

When given a bug report of this kind, the first thing I like to do is reproduce it. So let's paste the code into Emacs and see what happens with me. My gcc compiler warns about **gets** being dangerous – let's leave that for the moment – but manages to produce me a cc63 executable. OK, let's try your example:

```
$ ./cc63
Anthony Hopkins
John Guilgud
Michael Caine
Charlie Chaplin
^D
0: Caine Anthony
1: Charlie Chaplin
2: Hopkins Guilgud
3: Michael JohnðÙ
```

Great! I get almost the same output as you, which means we're pretty much on the same page, and gives me confidence that any fixes I make are likely to work for you too. The only difference between your run and mine was in the weird characters at the end. The presence of these almost always suggests a problem with null-termination of strings or buffer overruns, where arbitrary bits of memory are unintentionally written to the output, so you might expect them to vary between systems.

Apart from the weird characters, the first obvious error with this example is that the first and last names of the actors are all mixed up, which leads me to wonder: when you say 'designed to sort a list of names', are they meant to be sorted by first name (as typed in) or by last name? It is useful to be clear about such things when reporting the bug. Luckily, if I dig into the code I can see from the comment to the **compare** function that they should be sorted primarily by last name, so I'll assume that's the requirement.

Now, I expect I'll be testing this code quite often, and it'll get tedious typing in all those names every time. Let's write a little bash script that will run the test on a given input file. While we're there, we might as well use it to compare the output with what we expected.

```
#!/bin/bash

input=$1
expected_output=$2
thisdir=`dirname $0`
output=`cat $input | $thisdir/cc63`
diffoutput=`echo "$output" | diff -u -
$expected_output`

if [ -z "$diffoutput" ]; then
  echo "Test passed.  Output:"
  cat $expected_output
else
  echo "Test FAILED!"
  echo "---- Expected output ----"
  cat $expected_output
  echo "---- Actual output ----"
  echo "$output"
  echo "---- Diff ----"
  echo "$diffoutput"
fi
```

Now I'll write the example input into `actors.txt` and the expected output for cc63 into `actors-result.txt`. This is what it looks like when the test script is run:

```
$ ./cc63-test actors.txt actors-result.txt
Test FAILED!
---- Expected output ----
0: Michael Caine
1: Charlie Chaplin
2: John Guilgud
3: Anthony Hopkins
---- Actual output ----
0: Caine Anthony
1: Charlie Chaplin
2: Hopkins Guilgud
3: Michael JohnPÿ
---- Diff ----
--- -   2010-04-27 14:39:49.862546883 +0100
+++ actors-result.txt   2010-04-27
14:23:20.000000000 +0100
@@ -1,4 +1,4 @@
-0: Caine Anthony
+0: Michael Caine
 1: Charlie Chaplin
-2: Hopkins Guilgud
-3: Michael JohnPÿ
+2: John Guilgud
+3: Anthony Hopkins
```

Interesting. The test script seems to work, but I've noticed something else: the output has the mixed-up names in the same order as before, but the weird characters have changed from last time. It seems that in this case, the output corruption even varies between runs.

Fixing the obvious

Now we've got our handy tester script set up, let's start looking at the code. Rather than pick each line apart right now and get distracted by side issues, I'd first like to find out why the names are being strangely mixed up. The creation of the **names** array mostly seems to do what I think you meant it to do, although there are some issues we'll come back to later. What about the **qsort**? Well, it looks like you're passing the required arguments, but it's not a function I use very often, so let's just check the man page:

```
void qsort(void *base, size_t nmemb,
  size_t size,
  int(*compar)(const void *, const void *));
```

Aha, you had **n** and **sizeof(name)** the wrong way round: you need the number of members second and the size of each member third. Let's fix that and inch a little closer to correctness:

```
qsort(names, n, sizeof(name),
  (compar)compare);
```

After a recompilation, how does our output look now?

```
$ ./cc63-test actors.txt actors-result.txt
Test FAILED!
---- Expected output ----
0: Michael Caine
1: Charlie Chaplin
2: John Guilgud
3: Anthony Hopkins
---- Actual output ----
0: Michael Caine
1: Charlie Chaplin
2: John@ Guilgud
3: Anthony Hopkins
---- Diff ----
--- -   2010-04-27 14:45:06.907776877 +0100
+++ actors-result.txt   2010-04-27
14:23:20.000000000 +0100
@@ -1,4 +1,4 @@
 0: Michael Caine
 1: Charlie Chaplin
-2: John@ Guilgud
+2: John Guilgud
 3: Anthony Hopkins
```

Right, we are getting somewhere. The first and last names have stayed together, and the resulting list is indeed sorted by last name. John Guilgud is still suffering though. Why could that be? The corruption looks like it's

on the 'John' (it was in the original example too), so let's look how first names are stored:

```
p = strstr(line, " ");
nm->fn = (char*)malloc(p - line);
strncpy(nm->fn, line, p - line);
```

Everything from the beginning of the line to the first space is copied into **nm->fn**, which has just enough space (**p - line** bytes). I said earlier there might be a null-termination problem, and here it is.

With the input "John Guilgud", **p** points to the space and **p - line** is 4 bytes. That's enough space for "John" but not the zero byte to terminate the string. We need to allocate an extra byte for it:

```
nm->fn = (char*)malloc(p - line + 1);
```

What's more, checking the man page for **strncpy**, it won't add a terminating byte into **nm->fn** for us, because we're only copying up to the space. We'll have to do it ourselves:

```
p = strstr(line, " ");
nm->fn = (char*)malloc(p - line + 1);
strncpy(nm->fn, line, p - line);
nm->fn[p - line] = '\0';
```

With this fix, let's try our example once more:

```
$ ./cc63-test actors.txt actors-result.txt
Test passed.  Output:
0: Michael Caine
1: Charlie Chaplin
2: John Guilgud
3: Anthony Hopkins
```

A closer look at gets

OK, we finally have the desired output from our single test case of the example input! Are we finished now? Well, no, there's a lot more to do before I'll let this into any production system. While I've been compiling the various fixes above, gcc has constantly been reminding me about **gets**, so let's start by looking at that. What was the warning again?

```
$ gcc -o cc63 cc63.c
/tmp/ccy8Mdlq.o: In function `main':
cc63.c:(.text+0x138): warning: the `gets'
function is dangerous and should not be used.
```

Wow, that's pretty strong stuff for a compilation at the default warning level. Why is it dangerous? Well, let's have a look at the **line** array: it's 80 bytes big. That should be plenty for most actors' names, shouldn't it? Although **gets** is quick and easy to use, it doesn't notice if some naughty person tries to pretend there's someone with a stupidly long name out there. Or maybe there really is such a name ... [pause for Google search] ... let's put just a fraction of this German American's name[1] into a new input file called `long-name.txt`. The expected output is in `long-name-result.txt`:

```
$ ./cc63-test long-name.txt long-name-result.txt
Test FAILED!
---- Expected output ----
0: Hubert
Wolfeschlegelsteinhausenbergerdorffvoralternwarenge
wissenhaftschaferswesenchafewarenwholgepflegeundsor
gfaltigkeit...
---- Actual output ----

---- Diff ----
--- -   2010-04-27 14:56:31.516277996 +0100
+++ long-name-result.txt    2010-04-27
14:54:46.000000000 +0100
@@ -1 +1 @@
-
+0: Hubert
Wolfeschlegelsteinhausenbergerdorffvoralternwarenge
wissenhaftschaferswesenchafewarenwholgepflegeundsor
gfaltigkeit...
```

What, no actual output at all? Let's look at it without the test script:

```
$ cat long-name.txt | ./cc63
Segmentation fault
```

Boom! I don't know whether Mr Wolfe (as he chose to be known) ever dabbled in amateur dramatics, but perhaps our sneaky user wasn't sure, or maybe he was searching for a security hole and not really wanting to sort actors' names *at all*.

So **gets** is trying to fit the 123 characters of input into the 80 bytes allocated for **line**, which just doesn't go. Instead, it scribbles all over some other areas of memory as well, including some we're not allowed to touch, causing the crash. How do we fix this? We have to use **fgets** instead:

```
while (fgets(line, 80, stdin))
```

The warning has now disappeared, which is a good start. First let's test the original example again:

```
$ ./cc63-test actors.txt actors-result.txt
Test FAILED!
---- Expected output ----
0: Michael Caine
1: Charlie Chaplin
2: John Guilgud
3: Anthony Hopkins
---- Actual output ----
0: Michael Caine

1: Charlie Chaplin

2: John Guilgud

3: Anthony Hopkins
---- Diff ----
--- -   2010-04-27 15:01:49.857578926 +0100
+++ actors-result.txt   2010-04-27
14:23:20.000000000 +0100
@@ -1,7 +1,4 @@
 0: Michael Caine
-
 1: Charlie Chaplin
-
 2: John Guilgud
-
 3: Anthony Hopkins
```

Oh dear, we've broken it again. What are those extra newlines doing there? It seems that **fgets** doesn't behave *exactly* the same as **gets**: newlines are now kept in the **line** buffer unless the input reaches 79 bytes. This is awkward, and points out that the 'getting a name from the input stream' operation should really be its own function so that fiddly bits like this can be abstracted away from the rest of the code.

OK, so let's write a new function **get_input_line**. We overwrite the final newline if and only if the input is less than the maximum (which itself is one less than the buffer size):

```
char *get_input_line(char *line,
  int line_size, FILE *stream)
{
  char *result;
  result = fgets(line, line_size, stream);
  if (result)
```

```
  {
    size_t input_len = strlen(line);
    if (input_len < line_size - 1)
    line[input_len - 1] = '\0';
  }
  return result;
}
```

We can then simply replace **fgets** with **get_input_line** in the **main** function:

```
while (get_input_line(line, 80, stdin))
```

Note that I've left in **stream** as an argument to **get_input_line**, even though I could have hard-coded **stdin** within the function. This is because most utilities of this sort are written to accept either standard input or a filename on the command line. If this one ever gets updated to accept a filename, we won't have to modify **get_input_line**, which will help maintenance.

Anyway, rerunning with `actors.txt` now passes the test again.

Finding the space

I have a niggling feeling that this line is going to cause us trouble:

```
p = strstr(line, " ");
```

What is this saying? In order to split the input line into first and last names, we simply find a pointer to the space between them and use its position to create copies of each one. Note that we could just as well use **strchr()** to search for a single space character rather than a substring, which feels more efficient, but to be honest the outcome is equivalent and the differences are negligible so I'll leave it as it is. Anyway, the approach sounds sensible, until you realise that since this input comes from the user, it could contain anything. What if there are two spaces, like in Simon Russell Beale? Or no spaces, like Madonna?

Let's deal with Simon Russell Beale first. The **strstr()** function will find the first space, and so split his name into "Simon" and "Russell Beale", sorting on "Russell Beale" as his surname. This is wrong, but could easily be rectified by finding the last space with **strrchr()** instead. The problem is that then our cheeky user will go and enter "Robert De Niro". Extracting the surname from an arbitrary full name is actually very difficult for software to accomplish, if not impossible. I'll therefore leave the code as it is and not worry about it any more, unless our stakeholders express a preference for how these cases should be handled.

It's less obvious from the code what will happen with Madonna, so let's try it:

```
$ echo Madonna > no-space.txt
$ cat no-space.txt | ./cc63
[...slight pause...]
Segmentation fault
```

Oh dear. What's happening here? Well, that **strstr** line is trying to locate a pointer to the first space in "Madonna". Reasonably enough, it returns **NULL** to indicate the absence of a space. The bug is on the next line:

```
nm->fn = (char*)malloc(p - line + 1);
```

With **p** having the value **NULL**, the expression **p - line + 1** is going to have a crazy value. Just for fun, let's see what it is. I inserted the following line to print out some memory addresses:

```
printf("line = %08x, p = %08x,\n"
  "p - line + 1 = %u\n",
  line, p, p - line + 1);
```

Now I get this output:

```
$ cat no-space.txt | ./cc63
line = bfe6d51c, p = 00000000,
p - line + 1 = 1075391205
Segmentation fault
```

So we are allocating around 1GB memory to hold Madonna's first name. This actually works on my machine, but the following line then copies over a billion bytes from **line** into **nm->fn**. This causes the crash, since line is, as we noted before, only 80 bytes big.

So, how to deal with Madonna? It's even a valid input, so it's not as though you could claim that such 'far-fetched theoretical scenarios' will never be triggered in practice. Clearly, an actor with a single name must be sorted as though that is her surname. We can probably assume that she has a blank first name, but if we are in any doubt it would be a good time to consult with our users or stakeholders.

We need to rewrite the code so that, in the event of no space appearing in the input, the whole string is copied into the last name field, and the first name field is initialised to the empty string.

```
p = strstr(line, " ");
if (p)
{
  nm->fn = (char*)malloc(p - line + 1);
  strncpy(nm->fn, line, p - line);
  nm->fn[p - line] = '\0';
  nm->ln = strdup(p+1);
}
else
{
  nm->fn = "";
  nm->ln = strdup(line);
}
```

What happens now?

```
$ cat no-space.txt | ./cc63
0:  Madonna
```

No crash. Looks good. Hang on, wasn't I meant to be using a test script, at least when segfaults have been eliminated?

```
$ ./cc63-test no-space.txt no-space-result.txt
Test FAILED!
---- Expected output ----
0: Madonna
---- Actual output ----
0:  Madonna
---- Diff ----
--- -   2010-04-27 15:18:22.477023604 +0100
+++ no-space-result.txt 2010-04-27
15:10:14.000000000 +0100
@@ -1 +1 @@
-0:  Madonna
+0: Madonna
```

Oh, that's annoying. By having a blank first name, Madonna is being printed out with a leading space. I might easily have missed that without **cc63-test** screaming in my face. It looks like time for pulling another bit of the code into its own function, this time for printing a name:

```
void print_name(int index, struct name *nm)
{
  // nm->fn[0] is zero if nm->fn is empty
  if (nm->fn[0] == '\0')
    printf("%i: %s\n", index, nm->ln);
  else
    printf("%i: %s %s\n", index, nm->fn,
      nm->ln);
}
```

So the loop now looks like this:

```
for (i = 0; i < n; i++)
{
```

```
    print_name(i, &names[i]);
  }
```

As a result, Madonna is happier:

```
$ ./cc63-test no-space.txt no-space-result.txt
Test passed.  Output:
0: Madonna
```

Let's remember to check the original as well:

```
$ ./cc63-test actors.txt actors-result.txt
Test passed.  Output:
0: Michael Caine
1: Charlie Chaplin
2: John Guilgud
3: Anthony Hopkins
```

Good, good. We had another test file, didn't we? Oh yes, that German name:

```
$ ./cc63-test long-name.txt long-name-result.txt
Test FAILED!
---- Expected output ----
0: Hubert
Wolfeschlegelsteinhausenbergerdorffvoralternwarenge
wissenhaftschaferswesenchafewarenwholgepflegeundsor
gfaltigkeit...
---- Actual output ----
0: Hubert
Wolfeschlegelsteinhausenbergerdorffvoralternwarenge
wissenhaftschaferswes
1: enchafewarenwholgepflegeundsorgfaltigkeit...
---- Diff ----
--- -   2010-04-27 15:28:23.620914006 +0100
+++ long-name-result.txt        2010-04-27
14:54:46.000000000 +0100
@@ -1,2 +1 @@
-0: Hubert
Wolfeschlegelsteinhausenbergerdorffvoralternwarenge
wissenhaftschaferswes
-1: enchafewarenwholgepflegeundsorgfaltigkeit...
+0: Hubert
Wolfeschlegelsteinhausenbergerdorffvoralternwarenge
wissenhaftschaferswesenchafewarenwholgepflegeundsor
gfaltigkeit...
```

Hmm. You probably noticed I forgot to retest Mr Wolfe after fixing the **`gets`** bug, which was remiss of me. Ideally all the test files would be wrapped up in a single suite, the whole of which would be run regularly so this sort of thing wouldn't happen.

So, what's the problem here? Looking at the actual output, it seems that our **`fgets`** fix has successfully stopped crashes from happening but still doesn't give us a useful result in the event of someone really having such a long name. How should we fix this? Well, to be honest, if the intent of the utility is to sort names on input, then anything that only accepts part of the input isn't going to give the right answer. If we want to cope with arbitrary-length names then we have to do some clever tricks to append each piece of input to the current **`name`** structure somehow. Consult the stakeholders here: if the utility is only going to be used internally then they may well agree that the effort required is too great for virtually no practical gain. I will therefore leave the matter here. At this point I should put `long-name.txt`'s *actual* result into `long-name-result.txt`.

That way, although the output is fairly nonsensical, the test scripts will still notice if we were to re-introduce a crashing bug at a later date.

Further improvements

We seem to have a fairly solid piece of code now. What else could be improved? At this point I'm mainly thinking about Bernard, the future developer who has to maintain this utility after you've disappeared to Barbados with your lottery winnings.

Looking at the structure of the **`main()`** function, there are three basic operations: get the input, sort the names, and print the output. When you can break a function down so easily with this kind of analysis, then it is a prime candidate for refactoring the code into small functions, each with its own clear responsibility. Since each function has to have a name too, then you improve the documentation without even adding any comments. Here's what my **`main()`** function might look like:

```
int main()
{
  struct name *names = NULL;
  int num_names = 0;

  if (!get_input_names(&names, &num_names))
    return EXIT_FAILURE;
  sort_names(names, num_names);
  print_names(names, num_names);
  return EXIT_SUCCESS;
}
```

Here, we've managed to move two of the local variables, **`line`** and **`i`**, out of the **`main()`** function, leaving only the **`names`** array and the number of names. The increases comprehension of the code as there's less for Bernard to think about at any one time. I made **`get_input_names()`** a boolean function, returning a **`false`** value if there's a problem with the input, in order to allow the utility to abort, and note that the **`&`** in front of **`names`** and **`num_names`** makes it blindingly obvious that these are intended to be output parameters.

I've added return codes to **`main()`**– when I compile with the **`-Wall`** option, gcc points out their absence – and since the convention for returning zero (false) on success is slightly counter-intuitive, I've used the constants **`EXIT_SUCCESS`** and **`EXIT_FAILURE`** from `stdlib.h` instead.

The functions **`sort_names()`** and **`print_names()`** are fairly robust: both work when names is **`NULL`**, so long as **`num_names`** is zero to match, and **`print_names()`** doesn't require **`names`** to be sorted, and so can be re-used in other places, for example to print out what the array looks like before the sort.

Why create the **`sort_names()`** function at all? After all, it's only one line of code! There are two reasons. Firstly, we've made **`main()`** clearer: the function name shows that it's not just any old sorting operation, it's sorting *names*, and by hiding **`sizeof(name)`** and **`(compar)compare`** we've removed some implementation details that might confuse poor Bernard on first reading. Secondly, we might decide in future that **`qsort`** isn't good enough. By abstracting away the algorithm used, we are free in future to change it to something better, with the confidence that only a small piece of nicely encapsulated code will have to be modified.

You may have noticed I renamed **`n`** to **`num_names`** in my restructuring of **`main()`**. Why would I do that? I tend to think that cryptic variable names like **`nm`**, **`fn`**, **`ln`**, **`n`** and so on are unnecessary. In these days of large screens and auto-completion by text editors, it's not much effort to make Bernard's life a lot easier by using **`current_name`**, **`first_name`**, **`last_name`** and **`num_names`** instead. I'll let you change the others throughout the code.

Final points

What other comments should I make? We haven't looked at the **`compare()`** function at all. Apart from the unhelpful variable names, I would change one other thing. By writing **`if (!t)`** it looks like you're saying 'if the call to **`strcmp`** failed', as its return value **`t`** is treated as a boolean. I know **`strcmp`** is a super-standard function and lots of C programmers instinctively know about its behaviour, but for Bernard's sake I would write **`if (t == 0)`**, which is more likely to conjure up mental images of equality than of failure.

I would avoid having both **struct name** and the **name** typedef. It is confusing to mix the two within the code, especially as they have the same, er, name. To be honest, using **struct name** throughout is not a hardship, but if you really want to lose the **struct** then I would use **NAME** or **name_t** as the typedef, to distinguish it easily as a type, rather than, say, a variable.

I haven't mentioned the use of dynamic memory allocation at all. You are relying on the weird feature of **realloc()** to call **malloc()** if names starts out being **NULL**. It makes the code shorter but it does mean I had a momentary panic and had to look up **realloc()**'s documentation. We don't free *any* memory in this utility, which is OK as it stands, since the whole program exits after printing the result, but it does ring a few alarm bells. This actor-sorting code couldn't easily be dropped in as part of a larger system without additional work. It is always worth considering creating static buffers for simplicity: in this case we decided to keep the 80-character line limit for the input anyway, so unless memory is really tight or datasets are expected to be huge then **struct name** could simply contain two 80-character arrays and save quite a lot of hassle. Of course, if memory is really tight then you need a good deal more error-checking on your calls to **malloc()** and **realloc()** as it is.

Finally, remember that, given the considerable odds against that lottery win and your trip to Barbados going ahead, who is Bernard the maintainer most likely to be? It's you, in 18 months' time, struggling to remember why you couldn't have made this code just a little bit easier to understand.

References

[1] http://everything2.com/title/Longest+names

Pete Disdale <pete@papadelta.co.uk>

I have seen far worse C code than this and, if I'm honest, have written some of it! There are really only two things wrong with it however, 'wrong' in the sense of preventing it from working as intended – given safe input anyway – but also quite a few things worthy of comment. The easiest way I find to do this is a line by line walk through the code, with my comments in standard `/* ... */` C comments fashion.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

typedef int (*compar) (const void*, const void*);

typedef struct name
{
    char *ln;  // last name
    char *fn;  // first name
} name;

/*
 * I don't think that this is strictly wrong, but
 * using the same symbol for the typedef and struct
 * tag name is potentially confusing. In fact, that
 * confusion is shown further down where the writer
 * has sometimes used 'name' and sometimes 'struct
 * name' when referring to the same entity. Also,
 * like many C programmers I hate typing long
 * symbol names but there are occasions when a
 * longer name makes the code more readable; I
 * would suggest using 'lastname' and 'firstname'
 * for the struct members here -- this is simply a
 * personal preference!
 */

// order last names before first names
int compare (name *n1, name *n2)
{
    int t = strcmp (n1->ln, n2->ln);
    if (!t)
      t = strcmp (n1->fn, n2->fn);
    return t;
}

/* This exemplifies the comment above about symbol
 * names; the code is fine but one has to look
 * carefully to ensure that the code is fine! Also,
 * whilst one can not say that "if (!t)" is
 * strictly wrong, I think there are times where
 * "if (t == 0)" is more meaningful to the reader,
 * and reinforces the point that strcmp() returns
 * an int rather than a boolean type.  Again
 * personal preference.
 *
 * Finally, compar was typedef'd to receive const
 * pointers, so on the grounds that compare()
 * should not change what its parameters point to,
 * they should be declared const.
 */

int main()
{
  char line[80];

/*
 * That fixed buffer size literal will come back
 * and bite someone at some future time. Much
 * better to #define a buffer size, and use it
 * whenever possible in places where a buffer
 * overrun is possible.
 *      #define BUFLEN 80
 * and
 *      char line[BUFLEN];
 * is not much extra typing to do...
 */

  struct name *names = NULL;

/*
 * See comment above. Why bother to typedef the
 * struct and not use the typedef?
 */

  int n = 0;
  int i;
  while (gets(line))

/*
 * Using gets() is a cardinal sin!
 * Much safer would be:
 *      while (fgets(line, BUFLEN, stdin) != NULL)
 * to mitigate a buffer overrun.  Remember that the
 * input to this program could be redirected from
 * file, and that file could very easily have lines
 * longer than "normal" keyboard input.
 */

  {
    struct name *nm;

/*
 * Why not use the typedef name? (see next but one
 * comment)
 */

    char *p;
    n = n + 1;

/*
 * It is not very safe to bump this element counter
 * until the new line of input has been validated
 * and successfully added to the names array.
```

```
 * Remember that this number will be passed later
 * to qsort(), which could give odd results if e.g.
 * mem allocation fails and there are fewer
 * elements available than stated.
 */

    names = (name *) realloc (names,
      n * sizeof(name));

/*
 * ... as is done here?
 *
 * One other comment: realloc() is a very useful
 * function but as used here is potentially very
 * inefficient as the number of name elements gets
 * large, and could lead to serious memory
 * fragmentation. Not to mention that one should
 * always check the return value of any malloc()
 * functions for NULL...
 *
 * A better approach might be to use a linked list
 * for storage (or better still would be a binary
 * tree as this would obviate the need to later
 * sort the list). There are plenty of examples of
 * how to implement a binary tree, even the
 * Structures chapter in K&R has sample code.
 */

    nm = names + n - 1;
    p = strstr (line, " ");

/*
 * And if there is no space character in the input?
 * Or even multiple spaces?  In the first case, the
 * malloc() below will fail spectacularly, and in
 * the second case the lastname will have some
 * leading whitespace (which will make the qsort()
 * not work as intended.
 */

    nm->fn = (char *) malloc (p - line);
    strncpy (nm->fn, line, p - line);

/*
 * Here is major problem 1: even assuming that p is
 * valid there is no allowance made for the null
 * terminator for the first name. And while strncpy
 * will not overflow nm->fn it also does not add
 * the '\0' to the end. One way around this would
 * have been to write
 *
 *      *p = '\0';
 *
 * to terminate the string and to use strdup() to
 * sort out the allocation of the right size (as is
 * done with last name below.  The code as it
 * stands will not terminate the first name string,
 * and is primarily responsible for the strange
 * output from the sample input.
 */

    nm->ln = strdup (p + 1);
  }

  qsort (names, sizeof(name), n,
    (compar)compare);

/*
 * Here is major problem 2: the calling sequence is
 * wrong! It should be:
 *
 *      qsort (names, n, sizeof(name),
 *     (compar)compare);
 *
 * As it happens, using 4 input strings (and
 * compiling in small model, as I did) this error
 * was not apparent as both n and sizeof(name) had
 * the same value. But that said, there is no
 * sanity checking at all of the input: what if n
 * were zero?  qsort() might not like that.
 */

  for (i = 0; i < n; i++)
  {
    printf ("%i: %s %s\n", i, names[i].fn,
     names[i].ln);
  }

/*
 * main() was defined above as returning int, but
 * here there is no return value. The shell will
 * presumably receive whatever was in the
 * AX/EAX/RAX register at the time; far from ideal
 * behaviour.
 */

}
```

Apart from the above comments, the code demonstrates a general lack of error checking (e.g. for null pointers) and assumes well-formed input, which might be OK for a 'one-off, for personal use only' piece of code but would be a liability for production code.

To finish up, here is my attempt at tackling some of these potential shortcomings. I haven't taken the binary tree route in order to keep the functionality similar to the original, but have tried to address the points made earlier in the comments and suggested ways to write this program more defensively. Not that this attempt could not be improved upon – even though I'm old enough for a free bus pass I'm still learning and enjoying the experience! (Pete Goodliffe hits the nail on the head with his CVu article). Maybe *this* code could used for a future Code Critique?

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

#define LINELEN 80

typedef int (*compar) (const void*,
  const void*);

typedef struct name
{
  char *lastname;
  char *firstname;
} Name;

/**
 * function prototypes
 */
int main (void);
int compare (const Name *n1, const Name *n2);
char *skipws (const char *s);
char *findws (const char *s);
char *trimline (char *line);

/**
 * main()
 * Returns 0 on success, 1 on error
 */
```

```
int main()
{
  char line[LINELEN];
  Name *names = NULL;
  int n = 0;
  int retcode = 0; /* assume all OK */

  while (fgets(line, LINELEN, stdin) != NULL)
  {
    Name *nm;
    char *p;
    char *linep = trimline (line);

    if (!*linep)
    {
      fprintf (stderr,
        "Empty input - ignored\n");
      continue;
    }

    if ((p = findws (linep)) == NULL)
    {
      fprintf (stderr,
        "\"%s\": incomplete data - ignored\n",
        linep);
      continue;
    }

    /* else terminate the firstname
       and skip to lastname */
    *p = '\0';
    p = skipws (p + 1);

    names = (Name *) realloc (names,
      (n + 1) * sizeof(Name));
    if (names == NULL)
    {
      fprintf (stderr,
        "Out of memory - aborting\n");
      exit(1);
    }

    nm = &names[n];
    nm->firstname = strdup (linep);
    nm->lastname = strdup (p);

    if (nm->firstname == NULL ||
        nm->lastname == NULL)
    {
      fprintf (stderr, "Out of memory - results"
        " are incomplete\n");
      break;
    }

    /* else all OK - safe to bump counter */
    ++n;
  }

  if (n == 0)
  {
    fprintf (stderr, "No valid data to "
      "process!\n");
    retcode = 1;
  }
  else
  {
    int i;
    /* no point sorting less than two entries */
    if (n > 1)
      qsort (names, n, sizeof(Name),
        (compar)compare);
    for (i = 0; i < n; i++)
      printf ("%i: '%s' '%s'\n", i,
        names[i].firstname,
        names[i].lastname);
    free (names);
  }

  return retcode;
}

/**
 * Given a line of input (null terminated)
 * removes any leading and trailing whitespace,
 * including any end-of-line characters.
 * Returns a pointer to the trimmed line or NULL
 * if the input was NULL.
 */
char *trimline (char *line)
{
  if (line != NULL)
  {
    line = skipws (line);
    if (*line)
    {
      char *p = line + strlen(line) - 1;
      while (p >= line &&
          strchr(" \t\r\n", *p) != NULL)
        *p-- = '\0';
    }
  }
  return line;
}

/**
 * Compares two Name elements (using strcmp so
 * case sensitive)
 * Primary comparison is lastname, secondary is
 * firstname.
 */
int compare (const Name *n1, const Name *n2)
{
  int t = strcmp (n1->lastname, n2->lastname);
  if (t == 0)
    t = strcmp (n1->firstname, n2->firstname);
  return t;
}

/**
 * Given a char pointer to a null terminated
 * string, advances it skip over any space or
 * tab characters until either a non-whitespace
 * character or null is found.
 * Returns a pointer to the first non-white char
 * or NULL if the input was NULL.
 */
char *skipws (const char *s)
{
  if (s != NULL)
  {
  while (*s && (*s == ' ' || *s == '\t'))
    ++s;
  }
  return (char *) s;
}

/**
 * Given a pointer to a string of text, returns
 * a pointer to the first occurrence of a space
 * or tab character, or NULL if not found.
 */
```

```
char *findws (const char *s)
{
  char *p, *q;

  if (s == NULL)
    return NULL;

  p = strchr (s, ' ');
  q = strchr (s, '\t');

  if (p == NULL && q == NULL)
    return NULL;

  if (p == NULL)
    return q;

  if (q == NULL)
    return p;

  return (p < q) ? p : q;
}
```

PS. My promotion of **main()** to the first function in the source is personal preference only – it obviously has no influence in the way that the code works (or doesn't....)

Terry Whiterod <terry.whiterod@beraninstruments.com>

First let's first concentrate on why the code produces the wrong result. **qsort()** is defined as qsort(array, number of elements, size of an element, compare function). The code calls **qsort** but swaps the 'number of elements' and 'size of an element' parameters. This results in 8 elements of 4 bytes each being sorted rather than 4 elements of 8 bytes. As **sizeof(name)** is 8 and there are 4 names.

The junk "?03" on the end of the line "3:Michael John?03" is due to a missing terminator introduced by **strncpy(nm->fn, line, p-line)**; **strncmp()** will not append a terminator if the source string is greater than or equal to the buffer size specified in the last parameter **p-line**. This also raises another problem, the **nm->fn = (char*)malloc(p - line);** line creates a buffer big enough for the first name but not the string terminator. This could be changed to **nm->fn = (char*)malloc((p - line) + 1);**

Fixing these three problems will produce the expected output:

```
0: Michael Caine
1: Charlie Chaplin
2: John Guilgud
3: Anthony Hopkins
```

But other code robustness problems remain:

The return values of **malloc()** and **realloc()** should be checked against **NULL** in case the memory allocation fails.

There should be matching **free()** calls for the **names** array, first and last name strings, **strdup()** will allocate a string that can be freed using **free()**.

After **while(gets(line))** the **line** variable should be checked to see if it contains an empty string. This occurs if the user just presses the return key.

The result of **strstr(line, " ");** should be checked for a **NULL** pointer. This occurs if the user enters a first name only.

It is unsafe to use a fixed buffer **char line[80];** and pass it into **gets()** as no bounds checking is performed. If the user enters more than 80 characters then the memory after the array is overwritten.

It would be best to check the **n1** and **n2** parameters to the **compare()** function against **NULL** before using them. This is true for the **fn** and **ln** elements of the structures too. This function is called by external code and is not guaranteed to supply valid pointers.

And several less serious issues:

int compare(name* n1, name* n2) should be defined as **int compare(const void* vn1, const void* vn2)** instead and the parameters cast to the expected type before use. Otherwise the function declaration does not match the specification for **qsort**:

```
void qsort ( void * base, size_t num, size_t size,
    int ( * comparator )
  ( const void *, const void * ) );
```

Pointer arithmetic is bad, I would suggest using a different function to find the space character in the name, one that returns an array index. If one does not exist in a common library then one could be created.

It would be clearer to split the main loop up into several functions, each one describing the action to be taken:

```
while (get_name(line, sizeof(line))
{
  if (validate_name(line))
  {
    if (allocate_names(&names,  ++n))
    {
      nm = &names[n - 1];
      extract_names(line, name);
    }
  }
}
sort_names(names);
print_names(names);
```

The line length (80) should be defined as a preprocessor constant e.g.

```
#define LINE_LENGTH 80
char line[LINE_LENGTH];
```

I would also prefer to see variable names lengthened e.g. **nm** to **name**, **ln** to **lastName**, **fn** to **firstName**, **n** to **numberOfNames** etc. This increases readability and would make typing mistakes much less likely. Alarm bells should ring when you see a line like this:

```
char* ln; // last name
```

as the variable name should make the comment redundant. Strive to reduce comments by making the code obvious.

```
char* lastName;
```

Well that's it, I am sure there are many other style issues that could be commented on but I have tried to cover my major concerns. Problems aren't always evident as bugs, we should try to produce elegant and readable code. Code should check return codes and take appropriate action as errors occur. If code is readable then bugs are more obvious and can be solved quickly during the coding stage and not once the code has been delivered to the customer.

Balog Pal <pasa@lib.hu>

Uh, this sample is just plain disgusting. Really. Though I rather cut the flame, as it is about the C language, especially pre-C99, and whoever still picks it to do anything when we have 2010 on calendar... Hopefully other entries will show how it is done in C++, and point out how many problems are just sidestepped that way, including memory leaks and buffer overruns.

So here I pretend the scope of the problem was just to sort the supplied list of names using **qsort**, correctly, and is allowed to do anything for other input (including lines longer than 80 chars, not having a single space), and leaking memory is not a concern either. And we do get enough memory so allocations (in **realloc** and **strdup**) never fail.

Scanning the code filtering out the just mentioned problems leaves us with just a few yellow flags:

- the **compare** function, while correct inside, lacks **const** qualification of the input pointers
- use **strstr** instead of **strchr** to look for a single character
- use of blacklisted function **strncpy**

The rest seem like 'should be what required'.

strncpy masquerades as a fixed, replacement function (like **strncat**, **snprintf**...) for **strcpy**. It does take a size parameter and promises to limit action to that length – so far so good – but the rest of the specification is a mess. On one hand, if the source string is too long, it will not zero-terminate the result. While if source is short, it will fill the rest of the buffer with zeroes. None of that behaviour fits with the original use cases of **strcpy**, so used as replacement it will do mostly harm. The general suggestion is to not use it ever, write your own **safe_strcpy** or pick an existing one.

This turns out the actual source of our problem: the duplicate of the first name is created without zero-termination. That in turn will cause undefined behavior in the **compare** function and **printf**. The simplest fix is to make this field also with **strdup** like the last name, by overwriting the located space character with 0, then pass the line:

```
p = strchr(line, ' ');
*p = 0;
nm->fn = strdup(line);
```

With that the output is correct for the input.

Commentary

I was, like Balog, half expecting an answer suggesting the use of C++ but we got various critiques of the C code instead.

As most entrants pointed out, the C runtime library contains some bad design decisions.

- **strncpy** is poorly specified as it differs from the other strn… functions in ways making it easy to use unsafely
- **realloc** tries to do too many things: allocate new buffers, or change the effective size of existing buffers. It is also hard to write proper error handling
- **gets** is dangerous (no way to prevent buffer overrun) and it is then made worse by **fgets** handling line terminators differently.

I think all three functions are best avoided. The old adage 'a bad workman blames his tools' doesn't apply here – the tools in this instance actually are a problem.

The Winner of CC 63

As is often the case there was a lot of overlap between the entries but I think that Paul's critique was the most helpful – I liked the way he started with reproducing the error and then writing a simple automated test harness. He also asked a good question over how to sort names with multiple spaces!

Code Critique 64

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

I'm trying to write a simple quadratic equation solver for the equation "a * x * x + b * x + c = 0" that writes output to a file but I am having problems getting it working. It's OK for some inputs but I'm having problems, in particular with equations that have no (real) solution.

Listing 2

```
C:> cc64
Enter quadratic coeffs: 1 0 -1
Roots: 1 and -1
C:> cc64
Enter quadratic coeffs: 1 2 3
Roots: -1.#IND and -1.#IND
Didn't match

// Solve quadratic equation.
// Save the roots.
// Read and verify they wrote OK.

#include <cmath>
#include <fstream>
#include <iostream>

void tofile(double a, double b)
{
   std::ofstream("file.dat") << a << b;
}

void fromFile(double & a, double & b)
{
   std::ifstream("file.dat") >> a >> b;
}

void verify(double rootHigh, double rootLow)
{
  double readRootHigh, readRootLow;
  fromFile(readRootHigh, readRootLow);

  if ((readRootHigh != rootHigh) ||
      (readRootLow != rootLow))
  {
     std::cout << "Didn't match" << std::endl;
  }
}

int main()
{
  std::cout <<
    "Enter quadratic coefficients: ";
  double a,b,c,rootHigh,rootLow;

  if (std::cin >> a >> b >> c)
  {
    rootHigh = (-b + sqrt(b*b - 4*a*c)) / 2*a;
    rootLow = (-b - sqrt(b*b - 4*a*c)) / 2*a;
    std::cout << "Roots: " << rootHigh <<
      " and " << rootLow << std::endl;

    tofile(rootHigh, rootLow);

    verify(rootHigh, rootLow);
  }
}
```

The code is shown in Listing 2. 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.

Bookcase

The latest roundup of book reviews.

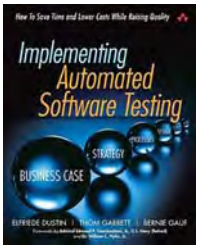
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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.

Jez Higgins (jez@jezuk.co.uk)

Implementing Automated Software



By Elfiede Dustin, Thom Garrett, Bernie Gauf, published by Addison Wesley (2009), ISBN 978-03211580511

Reviewed by Matthew Jones

I was looking forwards to reviewing this book because I've introduced a test framework at work, we use automated builds, we do some TDD, and we have some automated module tests. I was hoping that this book (with its impressive claims on the back cover) would take me to the next level. I was disappointed.

The book is vague, repetitive and far from concise. Editors and reviewers are acknowledged, but it has the feel of something that was padded to make a target, or that they were paid by the page. Nearly three quarters of all pages have one (or more) 'as we saw in Chapter X' in them, rendering the nine pages of contents almost irrelevant. It could do with some ruthlessly edited to lose 50 to 100 pages. The authors all work together in a company which specialises in AST for the defense (yes they are American) industry. No doubt the pace and presentation style reflect this, and maybe that's where the majority of the intended audience lie.

The book is divided into two sections. The first, what & why, takes 4 chapters (what is AST, why do it, business case, failures and pitfalls) and seemed to be pitched at student level: each concept is introduced in annoying detail, with lots of obvious examples and felt almost patronising. For example, one paragraph tells us that if we make a 10% improvement in a task that costs the industry billions, our savings could be millions. Amazing! Please tell me more! The second section, how, is divided into six chapters each detailing a 'key': a key point related to successful implementation of AST. What they tell us is that development of an AST implementation should be treated like any other serious software development project: it should be justified, required, planned, resourced correctly (there is an entire chapter on 'putting the right people on the job'), tested, tracked, measured, and carried out according to some defined process. That's it in a nutshell. There are 4 appendices: some lists, a case study, and one useful run down of the popular and/or effective tools that the authors have used, both FOSS and commercial. If you feel the need for a 329 page book to state these fairly obvious truths, then buy it. Otherwise, avoid. Perhaps I was expecting a more technical approach, explaining actually AST techniques, but the title is accurate: the book is all about implementation, and that is always going to be largely a project management issue.

I find it odd that many other reviewers generally score this book well (80%+ on Amazon). Maybe other books on testing make this one look good, but I really doubt that it 'fills a huge gap in our knowledge of software testing' (back cover comment from a professor of SW engineering). Maybe they are right and I am wrong, but don't say I didn't warn you.

In summary: tedious.

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

- **Holborn Books Ltd** (020 7831 0022) www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792) blackwells.extra@blackwell.co.uk

The Economics of Iterative Software Development

By Walker Royce, Kurt Bittner, and Mike Perrow, published by Addison Wesley (2010), ISBN 978-03211509352

Reviewed by Allan Kelly

It was the word 'economics' in the title that got my attention. It's not the first book to take liberties with the title, nor are these the first authors to abuse the word economics. You will find very little economics in this book, a shame really because a more discussion of the under supply and over demand of software would be a good thing.

What you will find is plenty of discussion about metrics, or rather the importance of measuring things. I guess that's what prompted the word economics in the title.

Really this is a book of to halves. What the third author did I don't know because it seems pretty clear the two parts were written by different people.

The first half is a very readable, short, discussion of iterative software development and why it is preferable from big functional phased development. Just don't look for the word Agile in the book, the authors base most of their examples and discussion on the (Rational) Unified Process. Perhaps unsurprising when you consider the authors have close links to IBM and Rational in particular.

This half of the book offers a few insights: their description of 'diseconomies of scale' in software development will stay with me. It is so obvious I wonder why I've never heard it before. On the whole though this is not a book of new insights, more a retelling of a popular story.

The second half is about measurement, the chapters are still short but, as so often happens with metrics, the discussion can be a bit dry. More solid metrics and worked examples would help. Despite this the author(s) know their stuff and there is a good discussion here. But when it comes to details there is too much hand waving.

Perhaps a bigger flaw in the book is the failure to cite references or research. The authors write with authority in their voice and, as far as I can tell, have a right to be authoritative. However it is hard to tell when they are offering their own experience, observations, solid research, industry data or just opinion. I find myself

asking 'Can they prove that?' time and time again.

In writing that I know I'm being a little unfair: writers who provide scrupulous detailed references and examples to prove their facts usually end up writing turgid texts. Few authors manage to provide both evidence and readability in one text.

While I'd happily recommend the first half of this book to a manager wanting a quick introduction to iterative development I can't say the same for the second half. So let your manager read the first half, but not the second; then talk to them about details and tell them RUP is not the only fruit.

iPhone User Interface Design Projects

by various authors, published by Apress (2009), ISBN 978-1430223597

Reviewed by: Pete Goodliffe

Verdict: OK

This is another book in Apress' iPhone development series. Like all the later books in the series, it's produced in black and white, which is a shame given that the book focuses on user interfaces.

It is a collection of essays from ten different authors. Each chapter stands alone. Some of the authors are clearly programmers, and their descriptions of interface design come from a techie viewpoint. Others authors are domain experts or people who hired in programmers to get their job done. These discussions are far less technical in nature.

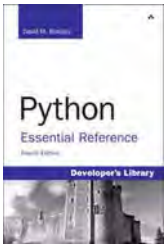
I've read quite a few of the APress iPhone books now, and although they are good, I'm starting to get tired of each chapter starting with a gushing 'the iPhone is great' section before getting into the meat of the topic. I'd like to see a little heavier editorial control if this series continues. The chapters are personal in nature, and that is part of the charm of these books. However some of the lengthy intros add nothing of value to their chapters.

The material in this book is not as strong as others in this series. This opinion does reflect my bias as a coder rather than an UI designer, but experienced UI designers won't find much essential or new information in this book. In general, good iPhone UI design requires an understanding the native iPhone idioms and of how to create compelling touch-based interfaces for small screen sizes. Some chapters go a way towards describing this; but sadly the book is by no means a compelling or thorough discourse on the subject.

I will admit my favourite chapter was a very interesting one on the Font catalogue application FontShuffle. The UI material here was somewhat thin, but it was a really interesting insight into the world of typography.

If you're starting off on some iPhone UI design work, know nothing about the topic, and fancy a chatty, but brief, introduction to subject this book is OK. If you are a UI designer then you'll probably not find much of value here.

Python Essential Reference, Fourth Edition



by David M. Beazley, published by Addison Wesley (2009), ISBN 978-0672329784

Reviewed by Garry Lancaster

Recommended

For anyone who is unfamiliar with it, Python is a dynamically typed language, with a clear and concise syntax, which is increasing in popularity. Some term it a scripting language, but others use it for very large applications.

This book's main audience is those with experience of other languages who pick things up quickly. Perhaps you've heard about Python and want to know what the fuss is about? Well, this book functions well enough as a no-waffle introduction. Later, it will stay on your desk as an initial reference source when programming. It is then that the comprehensive 78 page index will come in extremely useful. It isn't suitable for programming novices, who would be better served by a more gentle introductory text.

The first third of the 700 plus pages are devoted to explaining the core Python language in depth. This is very successful, and will usually answer any language-specific questions you might have. Most of the remainder covers the Python Library – such areas as string handling, database access, file and directory handling, and networking and web programming. Given the wide scope of the library, here the focus is necessarily more on breadth rather than depth. Some functions get just single sentence descriptions, whilst others have one sentence per parameter or more. Fortunately, there are numerous short examples. So, for in depth library insight, additional sources will sometimes be required, yet there is enough here to get you started.

Windows programmers should note that the very useful Windows specific libraries, such as for COM programming, are not covered, as coverage is restricted to the standard, cross-platform, parts of Python. The book concentrates primarily on Python version 2.6, as the majority of existing Python code is written for the 2.x branch, but does point out where version 3 differs.

Science : A Four Thousand Year History

By Patricia Fara, published by Oxford University Press (2010), ISBN 978-0199580279

Reviewed by Ian Bruntlett

According to this book, science began in the Mesopotamia region. We can look back to the Cradle of Civilisation being the founding of Babylon in about 2000 BC – but, typically for the dissemination of ideas, the Babylonians used prior knowledge – the invention of writing.

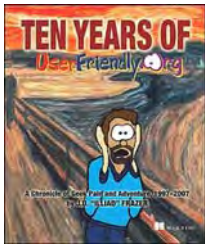
The first scientists weren't called scientists. However, one thing in common is a spectrum with one end being the application of scientific knowledge (farmers, sailors, fortune tellers – astronomy, accountants, business men, administrators – mathematics etc) and the other end of the spectrum being theorists (philosophers, wealthy amateurs – aristocrats/clergy).

These are the days of Big Science – e.g. the Hubble space telescope, the Large Hadron Collider. Science wasn't always Big. Initially they were individuals who performed wondrous feats and they were referred to as Magicians. This lengthy transition is told in this book.

Maybe I'm being a little naïve... but it has often been the cry of programmers that they don't want anything to do with politics... but after reading this book, you'll be aware of Science's competing influences – scientists themselves, industrialists, politicians, media and the military.

If you have already read Thomas S. Kuhn's *The Structure of Scientific Revolutions*, you really need to read this book. It attempts to cover 4,000 years worth of knowledge working in just under 500 pages. It does have some notes and diagrams but it also has about 17 pages listing the book's sources for diligent readers.

Ten Years of UserFriendly.org



By J.D. "Illiad" Frazer, published by Manning (2008), ISBN 978-1935182122

Reviewed by Alison Lloyd

UserFriendly holds a special place for me, as the first web comic I ever read. Detailing the exploits of the employees of a small Canadian ISP, it is also heavy on geek humour, and pretty much any technologically inclined person should be able to find something to relate to. Indeed, it has become part of the techie heritage: if you ever wondered where 'problem exists between keyboard and chair' (PEBKAC) came from, wonder no more!

This is a big, chunky volume, with one week's worth of comics per double page. In addition to the strips, J.D. Frazer has added comments alongside some strips; these range from trivial ('look how much better I am at drawing this character') to story background and planning, to insight and opinion on the real-world events that inspired certain strips or stories. There is also some background information on J.D. Frazer himself, from which we learn that his inspiration for the story was when he and some friends started a small ISP. Taken together, the reader is walked through the early development and

subsequent maturation of the comic in an engaging and light-hearted way.

The Sunday comics tend to be based on some topic of the day which had some technical relevance or interest. As you might expect, things like the SCO lawsuits and various Google happenings feature large. As this archive goes back ten years, it serves as a memory lane of many major events in technology, many annotated with the author's personal recollections and opinions.

The major let-down of this book is that the Sunday strips (which are published in colour online) are reproduced in black and white. This isn't too bad to start with, but many strips use colour as part of the punch-line (e.g. the colour worn by a particular character, or similar), which really grates when the colour is absent. I assume the decision was made for cost reasons, but it is a real pity and I feel detracts significantly from what is otherwise an excellent read.

In summary, this book will appeal to geeks everywhere. It stands up well as something to dip in and out of, and for a longer read, with the cartoon strips balanced nicely by comments and background from the author. Aside from the lack of colour, the layout is sensible and uncluttered. All in all, a pleasure to (re)read, and something that I've dipped back into more than once after (re)reading it cover-to-cover.

User Stories Applied – For Agile Software Development

By Mike Cohn, published by Addison Wesley (2004), ISBN: 978-0321205681

Reviewed by Paul Grenyer

After reading *Agile Estimating and Planning*, also by Mike Cohn, I was rather disappointed with *User Stories Applied*. Then I saw that *Agile Estimating and Planning* was published in November 2005 and *User Stories Applied* was published twenty months earlier in March 2004. A lot of the material in *User Stories Applied* forms the basis for and is expanded in *Agile Estimating and Planning*. Therefore I have come to the conclusion that Mike Cohn spent the twenty months between the two books improving as a writer! However, I think there is great scope for merging the two books and coming up with a better title. There is not enough user story based material here for a single book.

Only about half the book is actually about writing user stories. The other sections cover things like planning and testing. There is also some discussion about identifying roles within a system which, on the first read, felt a bit thin. Then when the case study came at the end and I had had chance to think about user roles in my own context I started see how useful defining them could be.

As you would expect, user stories are talked about in a reasonable amount of depth and most of the advice seems good to me. One of the main points I liked was the clear explanation of how user stories differ from tradition requirements capture and upfront design.

Mike Cohn asks questions at the end of each chapter. At the end of the book there are two appendices, one giving an introduction to XP and the other the answers to the questions.

Overall *User Stories Applied* is a little bit killer, but mostly filler.

Volume 4 Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams

By D Knuth, 261 pages, ISBN 0-321-58050-8

Reviewed by Frances Buontempo

Warning: Contains maths.

This fascicle is the second of five, 0 being the first of course. Together they will form a new 4th volume to Knuth's epic trilogy. Why would you read the short paper back versions rather than wait for the whole hard back volume? Knuth wants you to read it (and do the exercises) to provide a detailed review so that the next volume is perfect. Why might you read it? It's certainly less intimidating than a large heavy detailed technical tome, or at least not as big.

Two thirds of the pages are text and diagrams etc. The remainder is exercises and answers. Knuth frequently refers to the exercises during the course of the text, pointing out that you will get lost if you don't heed his advice and do the exercises. Knuth is right – I did read the book without doing all the exercises. In fact I only tried about 20 in total, and cheated by reading the answers on a few of those. By the end of the book I was slightly lost. The main thing I had learnt by the end of the book was that I needed to re-read it and try more exercises to understand more of it. I have spent a lot of my time reading mathematics books, and I *always* have to read them at least twice to understand them properly. Once through to get completely lost, but get to grips with the author's style and notation, and once more to actually learn the technical details in any depth. This book is no exception. It is not for the faint hearted.

The book is in two sections: the first covers a wealth of disparate information on bitwise manipulations such as bit shifting, reversing, swapping and permuting in general, tweaking several bits at once, and how to minimise the number of operations performed. The first algorithm given in the book shows how to compute the binary logarithm of a number, $\lambda \quad x = \lfloor \log_2 x \rfloor$. This is called Algorithm B. Each algorithm and theorem in the book is designated by a single letter, or primed letter. Since there is no index of algorithms and theorems, finding them when they are referred to elsewhere in the text is linear rather than logarithmic. The figures are labelled in numeric order, but they can be hard to find, since again there is no listing with page numbers. I am told this is easy to do in Latex, which ironically Knuth should know how to drive. The second part of the book is about binary decision diagrams, BDD. These are defined as 'A binary decision tree with shared sub-trees, a directed acyclic graph in which exactly two distinguished arcs emanate from every non-sink node.' Like the majority of the sentences in this book, it require a few reads through. It's a DAG with two arcs from every non-leaf node. The leaf nodes will be either TRUE or FALSE, allowing these data structures to be used for decision making. Knuth adds two restrictions: they should be minimal, so that no non-sink node's outgoing paths lead to the same node, and they should be ordered, that is if you label the nodes and traverse the BDD the labels will be in increasing order. He proves that you can order and minimise any decision diagram. He gives many examples of the use of BDD in combinatorial problems and shows how to minimise the space and time they take to complete. This book is packed with information, very hard to read, especially if you don't do enough of the exercises. Do I recommend it? I'm not sure. I enjoyed reading it, but struggled to understand several algorithms and applications. Reading a whole Knuth book was an awesome experience, though it will only be a chapter in the upcoming new volume. It seems you can pick up all 5 fascicles about £35 at the moment. Should you? Only if you do the exercises...

View From The Chair

Hubert Matthews
chair@accu.org

The not quite earth-shattering news of my election at the last AGM seems to have been somewhat overshadowed by some other election thing going on at roughly the same time. Not only that, but those other new incumbents seem to be taking a rather dim view of perks and expenses, so the ACCU private jet will have to wait for a while. And no-one wants to form a coalition with me, either.

So, what next and where do we go from here? The ACCU is successful so we must preserve that. I believe that this success is because our members are:

- passionate about programming,
- focused on learning, and
- having fun

I prefer 'passionate about programming' to 'professionalism in programming' as I feel it reflects better the enthusiasm that we have for our craft. Professionalism has overtones of wearing ties and being earnest and there are already other organisations (such as the BCS, IET, IEEE, etc) that tread this path. Passion is contagious and is much more likely to attract and interest potential new members, particularly the next generation of developers.

The emphasis on learning leads me to the next area: conferences and events. The spring conference is the jewel in the crown of the ACCU. It has grown over the years because of the work of a very dedicated group of people. We have even had an autumn conference again, something that has not happened for a number of years and something I'd like to see continue. My thanks go to all of the organisers for their hard work; we all appreciate it greatly. The spring conference, to me, is a technical forum with lots of hard-core programming content. The autumn conference could then become a forum for those other aspects of software development that are less technical.

Those who present at conferences will attest to the value of so doing. There's nothing like having to present your ideas publicly in a coherent form to get you to sort out your thinking. To this end, I would like to float the possibility of having small-scale workshops on writing or presenting to help budding authors develop their skills. I'm sure that we can find enough 'usual suspects' from within the ranks to assist. If we ran such workshops, the overall timetable of events could be a large spring technical conference, a small autumn conference on broader issues, with one or two presentation and writing workshops in between. Regular events would reinforce our already strong sense of community, as do the local ACCU group meetings around the country.

On a more mundane note, the ACCU constitution is quite old and doesn't reflect the current state of affairs. For instance, it doesn't mention the conference at all so there is no official recognition of the work of the conference team and no measure of accountability to the membership. A review seems appropriate and your input and ideas are most welcome, as always.

I am proud to be able to lead an organisation that has been a fundamental part of my development as a programmer. I sincerely hope that we can share this experience of growth and learning with as many new programmers as possible. The ACCU is full of outrageously interesting, amusing and talented people. Long may it continue.

ACCU 2010: a view from the conference chair

Giovanni Asproni
conference@accu.org

The ACCU 2010 conference was yet another memorable one. We managed to have a great programme and the usual great atmosphere, and all this despite many last minute problems. In fact, a few speakers had to pull-out for personal reasons just days before the start, and, as if that was not enough, the Eyjafjallajokull decided to erupt just after the conference started, making it impossible for some speakers, including Lisa Crispin, and Dan North (who was supposed to deliver a keynote), to be there.

Luckily, we had a contingency plan that relied on having so many great speakers around. I simply asked some of them if they want to take the free slots. So, we had Robert Martin explaining us 'WTF is a monad', Diomidis Spinellis talking about a mechanical device the Antikythera mechanism that was, effectively, a specialized computer built around 150–100 BC, and, on the Saturday, a session of 'Lightning Keynotes' – delivered by Robert Martin, James Bach, Walter Bright, and Jim Hague – to replace the missing keynote, and a very successful 'coding dojo' by Olve Maudal and Jon Jagger to fill up an empty 90 minute slot. The lightning keynotes were so successful that quite a few delegates suggested keeping them for the next editions of the conference as well.

One very positive side-effect of the volcanic eruption was that almost all the speakers were at the conference dinner, making it even more interesting than usual, and, for some, more expensive, given the money they spent in the very successful charity auction for Bletchley Park held during the dessert.

At the end I was very tired (after the conference, I needed an entire week to recover), but also very satisfied – the feedback from the attendees was exceptional, with many of them telling me that this was the best ACCU conference they had been to! Unfortunately I cannot take all the credit. In fact, most of it goes to the conference committee, the AYA organisers (Julie, Belinda and Marsha) who, as always, did an outstanding job, to the people I always consult for opinions and suggestions, Allan Kelly, Alan Griffiths and my official Consigliori Kevlin 'Hacker' Henney (who had also the idea of the lightning keynotes), and finally to all the speakers and delegates who always make the conference so special.

We are now starting the organisation for 2011. In the last few years we have been making changes to the committee every year in order to bring new ideas in and to give other ACCU members a chance to participate in the organisation of the conference. This year is no exception, and the two longest serving committee members – Ewan Milne and Alan Lenton – have stepped down down. Please, join me in thanking them for their invaluable contributions to the success of the conference. They will be replaced by Jon Jagger and Alan Griffiths, two ACCU members well known for their knowledge and experience.

If you have any feedback, comments, or suggestions regarding the conference, or even if you have an idea for a proposal, but you are not quite sure about it and want some help or feedback, feel free to email me at conference@accu.org.