The magazine of the ACCU

www.accu.org

{cvu}

Volume 23 • Issue 4 • September 2011 • £3

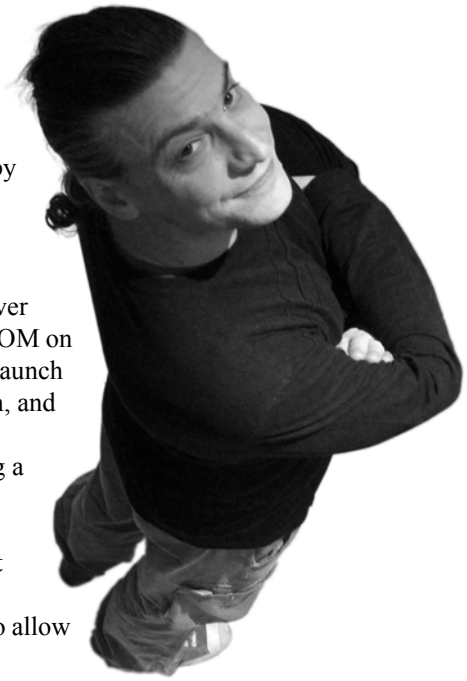
Features

Regulars

{cvu}
Volume 23 Issue 4
September 2011
ISSN 1354-3164
www.accu.org

accu

Forgotten Old Hat



Many years ago now I began life as a programmer working with real-time operating systems and the embedded systems they control. It all began as part of my University Degree, where my ambition was to write my own operating system. Ok, so the result was very simple, and ran on a simple 8-bit single-board machine, powered by an 8086 micro-controller, with serial and parallel interfaces connected to a switch box and an LCD display respectively.

The OS itself had to be linked directly with whatever program was to be run, and then burned to an EPROM on the board. The program would use the OS API to launch processes, and the OS in turn would marshall them, and do the necessary 'magic' to run them pseudo-concurrently – which essentially meant performing a context-switch by saving the necessary registers away and loading the register values of another process. For those interested in this sort of thing, it used priority-driven pre-emptive scheduling and provided limited memory managament – enough to allow processes to allocate and free heap space.

To allow data to be shared between running processes, I used an Exchange, whereby any process could leave a message addressed for another to consume at some later stage. A system-level global semaphore protected the one-and-only exchange.

I had all-but forgotten my first forays into the world of real-time OSs and Sequential Processes – both Dijkstra and Hoare varieties – until fairly recently, listening to a talk by Russel Winder on the benefits of message passing instead of memory sharing in multi-threaded programs. I had thought much of what I'd learned of real-time OSs no longer relevant. It seems this old hat is being dusted off for new duties.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 23.5: 1st October 2011
C Vu 23.6: 1st December 2011

Overload 106: 1st November 2011
Overload 107: 1st January 2012

WRITE FOR C VU

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

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

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

Enumerating Experiences

Matthew Wilson uncovers some traps and pitfalls with enumerations in C and C++.

This article considers some aspects of the definition and use of enumerations that affect the quality and maintainability of C and C++ code. It covers the naming of enumeration elements and the assigning of values along with suggested good practices for each, and presents some good and bad ideas for the inter-conversion of enumeration types with character strings.

Introduction

I'm currently finishing up a very long engagement with one client, and in the middle of a short arch/design/code review for another. Coincidentally, both tasks have recently raised the issue of the proper definition and use of enumerations, most particularly in respect of validity and inter-conversions with string forms.

Three things in particular stand out. First, the default way in which a compiler assigns values if the programmer has not done so explicitly can lead to problems in filtering. Second, some mechanisms for the inter-conversion of enumerations and strings can be trickier, or more fragile, than is appropriate. Finally, the notion of whether an enumeration value is valid or not, and where and how to validate it, appears to be a matter of confusion. I'll attempt to cover all of these issues, along with brief descriptions of the purposes, anatomies, and suggest some good practices for naming, assigning values, and for making inter-conversion with strings as safe and DRY [1] as possible.

Purpose

An enumeration introduces a new type that models a concept that can be represented by named constants, each of which represents a different possible values/state/aspect of that concept, the so-called enumerated type [2].

Consider the enumeration shown in Listing 1. Its name is **`Fruit`**, and it contains three constants, known as enumerators, called **`Apple`**, **`Banana`**, and **`Orange`**.

Use as enumerated types is the primary purpose of enumerations, and likely the most widely used. But there are other important uses:

- cross-language constants (C&C++)
- (grouped) flags
- member constants
- member flags
- combinations of the above

They're used for cross-language (C ↔ C++) constants without resorting to the pre-processor (as **`#define`** symbols). A common case is for return codes, as in the example shown in Listing 2. This allows for overloading on the type when used from C++, something that's very handy for diagnostics.

Listing 1

```
enum Fruit
{
  Apple,
  Banana,
  Orange,
};
```

Listing 2

```
enum MYAPI_RC
{
  MYAPI_RC_OK,
  MYAPI_RC_OUTOFMEMORY,
  MYAPI_RC_BADHANDLE,
  . . .
};
struct MYAPI_XYZ_T;
#ifdef __cplusplus
dump_to_log(MYAPI_XYZ_T const* xyz);
dump_to_log(MYAPI_RC const& rc);
#endif /* C++ */
```

They used to be commonly used in C++ to define member constants, and still are for any code that must work for compilers that are non-conformant with C++-98 in that regard, as shown in Listing 3.

The disadvantage in this case is that it's limited to the values representable by **`int`**, though that's very rarely a problem.

We'll look at flags and combinations shortly, after discussing naming and values.

Naming

An enumeration definition consists of the **`enum`** keyword, an optional tag (name), and an enumerator list, which is a list of one or more enumerators specified between curly braces.

In C, the type name of an enumerator **`X`** is the phrase **`enum X`**. So, to use **`Fruit`** in C, you'd have to write code such as in:

```
enum Fruit f1 = Apple;
enum Fruit f2 = Banana;
```

This rapidly becomes painful, so it is customary to define a **`typedef`** of the enumeration name to itself, as in:

```
typedef enum Fruit Fruit;
```

If you don't specify a name, you define what is (self-evidently) called an anonymous enumeration, as in Listing 4.

In this case, you can't declare an instance of the enumeration type, since it's anonymous, and must instead use the values in a less type-safe manner, with an integer variable. Anonymous enumerations are seldom used for enumerated types, but are more common when defining flags, as we'll see later, or when defining member constants, as is seen in Listing 4.

Listing 3

```
class MyClass
{
public:
  enum { MAX_GIZMO_DIM = 12 };
  . . .
};
```

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.

Listing 4

```
enum
{
  Apple,
  Banana,
  Orange,
};
```

Listing 5

```
enum MyFruit
{
  Apple,
  Banana,
  Orange,
};
enum YourFruit
{
  Cumquat,
  Lemon,
  Orange, /* COMPILE ERROR HERE */
};
```

One of the most annoying (to me at least) aspects of enumeration naming is that the enumerators have the same scope as their defining enumeration. This means that the symbols **Apple**, **Banana**, and **Orange** exist in the same namespace as the type **enum Fruit**. This means that if any other enumeration has defined any enumerators with the same name, they will clash (see Listing 5). This applies to both C and C++.

One option (and the only reasonable one in C) is to prefix the enumerator names with the enumeration name, as in Listing 6.

One trick I commonly use when writing enumerations (as enumerated types) is to enclose them in a namespace with the same name, as in Listing 7.

This works well to disambiguate the enumerator names. In my opinion, it also gives a clear look to the use of the enumerators, as in:

```
switch(fr)
{
  case MyFruit::Apple:
    . . .
```

The downside with it is that you must repeat yourself a little with the type when defining the variables, as in:

```
MyFruit::MyFruit fr = MyFruit::Banana;
void milkshake(MyFruit::MyFruit fr);
```

But that seems a pretty small price to pay for the other benefits. Obviously, this is only suitable for enumerated types that are exclusively for use by C++ code; if you need to support both languages, you need to use the prefix form.

Values

If, as in the previous definition of **Fruit**, the enumerators are not given explicit values, then the compiler provides them with values, according to the following rules:

Listing 6

```
enum MyFruit
{
  MyFruit_Apple,
  MyFruit_Banana,
  MyFruit_Orange,
};
enum YourFruit
{
  YourFruit_Cumquat,
  YourFruit_Lemon,
  YourFruit_Orange,
};
```

Listing 7

```
namespace MyFruit
{
  enum MyFruit
  {
    Apple,
    Banana,
    Orange,
  };
} // namespace MyFruit
namespace YourFruit
{
  enum YourFruit
  {
    Cumquat,
    Lemon,
    Orange,
  };
} // namespace YourFruit
```

1. If the first enumerator is not given an explicit value, it is given the value 0.
2. Any other enumerator that is not given an explicit value is given the value 1 + v, where v is the value (explicit or implicit) of the previous enumerator in the list.

Hence, we can know at compile-time what the enumerator values are:

```
assert(0 == Apple);
assert(1 == Banana);
assert(2 == Orange);
```

If we choose to do so, we can specify some/all of the enumerator values explicitly, as shown in Listing 8.

The values are still known at compile time, but may no longer be 0-based or contiguous.

```
assert(-1 == Apple);
assert(2 == Banana);
assert(3 == Orange);
```

There are good reasons to beware using 0-based enumerator values. Enumerations can (and often do) get (un)marshalled to/from binary formats, and it is not uncommon to find that conversion from a malformed binary stream appears to convert correctly to a known value simply because it contained a zero in the right spot. Obviously, any number can be erroneously converted, but 0 seems to be particularly vulnerable. As a consequence, I strongly caution clients to avoid defining enumerated types where the 0 enumerator is a valid state/value, so would start the real fruit numbering at 1, something like Listing 9.

Conversely, it is possible (i.e. the compiler won't stop you) in C++ to write code such as the following, even when the enumerator values are those specified in Listing 10 (i.e. there is no enumerator with value 0).

```
Fruit fr1 = Fruit(); // or
Fruit fr2 = Fruit(0)
```

Listing 8

```
enum Fruit
{
  Apple = -1,
  Banana = 2,
  Orange,
};
```

Listing 9

```
enum Fruit
{
  Apple = 1,
  Banana,
  Orange,
};
```

Listing 10
```
enum Fruit
{
  Unknown = 0,
  Apple,
  Banana,
  Orange,
};
```

Listing 11
```
enum Fruit
{
  Unknown = 0,
  Apple,
  Orange_IS_NO_LONGER_SUPPORTED,
  Banana,
};
```

Listing 12
```
enum Fruit
{
  Unknown = 0,
  Apple = 1,
  Banana = 3,
  Orange = 2,
};
```

As a consequence of these two related factors, I always define enumerated types with a first enumerator that represents an invalid, unknown, or otherwise reasonable default value, and have the actual domain values start at 1, as in Listing 10.

There are other risks in assigning values relying on automatic numbering. In the **`Fruit`** type so far (Listing 10), we have three fruits represented, and they are given the values 1, 2, and 3 implicitly. Because what's already here is in alphabetical order, it is all too human to, say, insert a **`Melon`** before **`Orange`** to preserve the apparent consistency of the alphabetical ordering. The problem is, that changes the value of **`Orange`** from 2 to 3. Any code that relies on the values must be recompiled in order to prevent it working with a **`Banana`** when it thinks it's working with an **`Orange`**.

A similar problem occurs if an enumerator is removed, since all subsequent enumerators shift down one value.

In any circumstances where you are not *100% sure* that all affected source will be rebuilt – such as when an enumeration is exposed via an API – you should always add new enumerators at the end, and never remove or change the values of existing ones. If you need to deprecate one, remove its name by renaming it, preferably to something unambiguously off-putting, as in Listing 11.

Alternatively, you can eschew the implicit value assignments, and do them all manually, as in Listing 12.

This brings us to the last gotcha with values of enumerated types: duplicates. If you are assigning values manually, it is all too easy to inadvertently assign a duplicate, as in Listing 13. Be sure to double-check that your values are unique.

Of course, sometimes a duplicate is desired, as in the (pedagogical) example given in Listing 14.

Listing 13
```
enum Fruit
{
  Unknown = 0,
  Apple = 1,
  Banana = 3,
  Lemon = 4,
  Orange = 2,
  Pomegranate = 3, /* OOPS! */
};
```

Listing 14
```
enum Fruit
{
  Unknown = 0,
  Apple = 1,
  Banana = 3,
  Orange = 2,
  Nana = 3, /* pour les enfants */
};
```

Listing 15
```
enum Fruit
{
  Unknown = 0,
  Apple = 1,
  Banana = 3,
  Orange = 2,
  Nana = Banana, /* pour les enfants */
};
```

Listing 16
```
enum Fruit
{
  Unknown = 0,
  Apple,
  Banana,
  Orange,
  MAXIMUM_Fruit_VALUE
};
```

If that is intended, it is far better to assign from the original enumerator (see Listing 15), as it helps any future maintainer of the code to see that it was intended.

Often it's helpful to know how many enumerators are in the enumeration. Unfortunately, neither C nor C++ provide any built-in help in this regard. A reasonable workaround [3] is to use a last-maximum-value, as shown in Listing 16. Then when you add a new fruit you always insert it *before* the last-maximum-value (which is the only exception to the rule that you never change the value of an existing enumerator), as shown in Listing 17.

If, as in these two listings, the first enumerator starts at 0, and all the rest are implicitly valued, the value of **`MAXIMUM_<my-enum>_VALUE`** will be equal to the number of enumerators (excluding itself) in the enumeration, which can be a handy thing to know, as we'll see shortly.

Flags and combinations

Because you can specify any value you want to in an enumerator, it is not uncommon to see them used as flag values, as in the extract from the **`unixstl::glob_sequence`** file-system search sequence class from the STLSoft libraries [4] shown in Listing 18.

Each enumerator has a value corresponding to a single, unique bit, and therefore can be specified to the constructor in any combination to determine what kinds of file-system elements are searched for (directories and/or files; include the dots directories?), and in what form they are presented (directories marked with trailing slash; relative or absolute path?).

Listing 17
```
enum Fruit
{
  Unknown = 0,
  Apple,
  Banana,
  Orange,
  Lemon,
  MAXIMUM_Fruit_VALUE
};
```

Listing 18

```
// file: unixstl/filesystem/glob_sequence.hpp
class glob_sequence
{
public: // Member Types
  . . . value_type, const_iterator, etc.
public: // Member Constants
  enum search_flags
  {
    includeDots   = 0x0008
  , directories   = 0x0010
  , files         = 0x0020
  , noSort        = 0x0100
  , markDirs      = 0x0200
  , absolutePath  = 0x0400
  . . .
  };
public: // Construction
  template <typename S>
  explicit glob_sequence(
    S const& directory
  , int      flags = noSort
  );
  . . .
};
```

Listing 19

```
class glob_sequence
{
. . .
  enum search_flags
  {
    includeDots   = 1 << 0
  , directories   = 1 << 1
  , files         = 1 << 2
  , noSort        = 1 << 3
  , markDirs      = 1 << 4
  , absolutePath  = 1 << 5
  . . .
  };
  . . .
};
```

Listing 20

```
enum B64_FLAGS
{
    B64_F_LINE_LEN_USE_PARAM    = 0x0000
  , B64_F_LINE_LEN_INFINITE     = 0x0001
  , B64_F_LINE_LEN_64           = 0x0002
  , B64_F_LINE_LEN_76           = 0x0003
  , B64_F_LINE_LEN_MASK         = 0x000f
  . . .
};
```

Listing 21

```
namespace Fruit
{
  enum Fruit
  {
    Unknown = 0,
    Apple,
    Banana,
    Orange,
    Lemon,
    MAXIMUM_Fruit_VALUE
  };
} // namespace Fruit
```

Listing 22

```
char const*
FruitToString(
  Fruit::Fruit const& fr
)
{
  switch(fr)
  {
    // Valid values
    case Fruit::Apple:
      return "Apple";
    case Fruit::Banana:
      return "Banana";
    case Fruit::Orange:
      return "Orange";
    case Fruit::Lemon:
      return "Lemon";
    // Invalid values
    case Fruit::Unknown:
      return "<unknown-fruit>";
    default:
      return "<invalid-fruit>";
  }
}
int main()
{
  puts(FruitToString(Fruit::Apple));
  puts(FruitToString(Fruit::Banana));
  puts(FruitToString(Fruit::Orange));
  puts(FruitToString(Fruit::Lemon));
  puts(FruitToString(Fruit::Unknown));
  puts(FruitToString(
     Fruit::MAXIMUM_Fruit_VALUE));
  puts(FruitToString(
     Fruit::Fruit(-1010)));
  return 0;
}
```

It's also common to have a mask enumerator to help the component's author and/or user to clearly delineate different enumerator roles. For example, **glob_sequence** might also have defined **contentMask** (with value **0x00ff**) and **styleMask** (with value **0x0f00**).

There are various schemes for specifying the bit-flags. As well as using hex, which I strongly prefer and recommend, you can also do it using the left-shift operator, as in Listing 19.

The advantage of this form is that it's arguably clearer to ensure that you don't have clashes, if you're not that comfortable in hex. The disadvantage is that you cannot easily specify particular values, or leave bit-gaps for future expansion: any of that and the scheme is easily disrupted. And, personally, I find it harder to digest, but that might just be because I am in the habit of using the explicit hex form.

Less common, but still prevalent, is to define enumerations as a combination of enumerated type and flags. Consider the **B64_FLAGS** enumeration, from the **b64** library [5], shown in Listing 20. The bits 0–3 are interpreted as an enumerated type with which you specify only one of the **B64_F_LINE_LEN_???** flags to determine line lengths, whereas the enumerator **B64_F_LINE_LEN_MASK** is a flag enumerator that is used to mask off other flags when, say, wanting to **switch** on the length type.

Achieving correct specifications of enumerated type ranges and flags is a lot more difficult in combination, and this form is generally best avoided wherever good alternatives are available (which, with hindsight, may have been the case with **b64**).

Inter-conversion with strings

Conversion to string

It's a normal requirement that types be representable as strings, and enumerations are no exception. In my experience, the most common way to do this is via a **switch** statement, such as that shown in Listing 22, to

Listing 23

```
std::string
FruitToString(
  Fruit::Fruit const& fr
)
{
  switch(fr)
  {
    // Valid values
    case Fruit::Apple:
      return "Apple";
    case Fruit::Banana:
      return "Banana";
    case Fruit::Orange:
      return "Orange";
    case Fruit::Lemon:
      return "Lemon";
    // Invalid values
    case Fruit::Unknown:
      return "<unknown-fruit>";
    default:
      break;
  }
  char const fmt[] =
    "<invalid-fruit-value:%d>";
  char buff[50]; // can't exceed 43
  sprintf(buff, fmt, fr);
  return std::string(buff);
}
```

Listing 24

```
std::string
FruitToString(
  Fruit::Fruit const& fr
)
{
  switch(fr)
  {
    // Valid values
    case Fruit::Apple:
      return "Apple";
    case Fruit::Banana:
      return "Banana";
    case Fruit::Orange:
      return "Orange";
    case Fruit::Lemon:
      return "Lemon";
    // Invalid values
    case Fruit::Unknown:
      return "<unknown-fruit>";
    default:
      break;
  }
  char const fmt[] =
    "invalid fruit value:%d";
  char buff[50]; // can't exceed 41
  sprintf(buff, fmt, fr);
  throw bad_fruit_exception(buff);
}
```

convert to string form the version of **Fruit** that is presented in Listing 21 (this is the final version that'll be assumed for the rest of this article).

Where developers appear to differ slightly is in the kind of string they return, and differ markedly in what they have such functions do when presented with invalid values.

The issue of the string type is simplest. In the previous case I used a return type of multibyte C-style string (**char const***) because every possible branch in my **switch** statement returned a pointer to a literal string. There are no memory issues, no re-entrancy issues. This form cannot fail, and may readily be tested correct.

If you need to localise strings, returning a C-style string (in preference to **std::string**, or equivalent) is hard to do. If you need to capture the integer value of unrecognised fruit variables, it is harder still. In such cases the simplest tack is to return a string class instance, as shown in Listing 23 (with **sprintf()** failure handling elided).

Where things get more complex is in what happens for the invalid values. In the second form of **FruitToString()**, we're paying the (potential – *small-string optimisation* may be in play) cost of allocating memory to hold the copy of the literal string for all valid values, just so we can capture any invalid value (except for **Fruit::Unknown**) for what may be presumed to be diagnostic purposes.

Some programmers go further, and throw an exception for unrecognised invalid values, as in Listing 24.

These contingent actions may or may not be appropriate. It depends on what you are using these conversions for. To shed light on that, we need to talk about validity of enumeration variables in the next section.

Validity

Because enumerated types that are used to represent domain concepts are often (un)marshalled to/from binary (or textual) representations, it is quite common to have to deal with invalid values. They must therefore be checked. In *contract programming* terminology, this is known as *filtering* [6], whereby potentially invalid values brought in from 'outside' are validated, and subsequently may be assumed correct in 'inside' code elements.

A common approach is to have an **IsValidXxxx()** function, employing a simple **switch** statement, as shown in Listing 25.

In this program design, **IsValidFruit()** is part of the filtering layer, and therefore must take account of invalid **Fruit** values, whereas **UseFruit()** is part of internal program logic, and therefore can assume that it will never be passed a bad apple (or any other type of fruit.) (Ouch, that was a rotten plum, er, pun!) This is one of the primary essences of contract programming, and is utterly in opposition to the practice of checking everything at every level that is advocated by *defensive programming*. I am a very strong proponent of contract programming, and assert that the delineation it requires is profoundly important in (i) making responsibilities unambiguous, (ii) keeping code as small and clear as it can be, (iii) improving efficiency, and (iv) relieving programmers of thinking that they have to do everything, including the impossible. Furthermore, as we will see in this case, it can avoid nasty programming errors that can result in hung or crashed programs.

Any function that is in internal logic, and can only receive 'good' fruit, need not take account of invalid values. Hence, the definition of **UseFruit()** is entirely appropriate, since it only takes account of **Apple**, **Banana**, **Orange**, **Lemon** (which are the only ones it knows how to deal with). Nonetheless, it is fragile to change: there's nothing stopping someone adding a new **Fruit**. Some compilers are able to offer warnings if enumerators are missed out of a **switch** statement, but use of an 'unknown' (0) value, such as **Fruit::Unknown**, renders this impotent. As a consequence, I would always advise having a **default** branch when **switch()**-ing on an enumeration, but rather than attempting some contingent action to supply some useful functionality (which cannot even be achieved in the general case), it should have an active contract enforcement (to catch the design violation). Hence, I would rather write **UseFruit()** as in Listing 26.

Some programmers approach their equivalents of **FruitToString()** by making similar assumptions, as shown in Listing 27 or Listing 28. This is totally righteous, and fully and appropriately in keeping with contract programming principles. *Except that in the case of enumeration to string conversions it can come back to bite you in a big way.*

Consider that your program transfers marshalled fruit (in text form) between peers by networking, then (i) you're going to have to be doing

Listing 25

```
typedef std::vector<byte> flesh_t;
bool
IsValidFruit(
  Fruit::Fruit const& fr
)
{
  switch(fr)
  {
    // Valid values
    case Fruit::Apple:
    case Fruit::Banana:
    case Fruit::Orange:
    case Fruit::Lemon:
      return true;
    // Invalid values
    case Fruit::Unknown:
    default:
      return false;
  }
}
Fruit::Fruit
ReadInFruit(
  flesh_t& flesh
);
void
UseFruit(
  Fruit::Fruit const& fr
, flesh_t const&      flesh
)
{
  assert(IsValidFruit(fr));
  switch(fr)
  {
    case Fruit::Apple:
      makeApplePie(flesh);
      break;
    case Fruit::Banana:
      makeBananaBread(flesh);
      break;
    case Fruit::Orange:
      makeOrangeJuice(flesh);
      break;
    case Fruit::Lemon:
      makeLemonade(flesh);
      break;
  }
}
int main()
{
  flesh_t      flesh;
  Fruit::Fruit fr = ReadInFruit(flesh);
  // Filtering
  if(IsValidFruit(fr))
  {
    // Internal logic
    UseFruit(ff, flesh);
  }
  return 0;
}
```

string inter-conversions, and (ii) you're going to have to be doing validation in a filtering layer. If you use (as you should) diagnostic logging throughout the domain-specific and/or complex parts of your the codebase, you might see something along the lines of Listing 29.

In this case, `FruitToString()` is part of contingent action in the filtering layer, and therefore might be called with an invalid fruit. Indeed, by the definition of `IsValidFruit()`, `FruitToString()` will be given an invalid fruit in this case. Unfortunately, given its previous

Listing 26

```
void
UseFruit(
  Fruit::Fruit const& fr
, flesh_t const&      flesh
)
{
  assert(IsValidFruit(fr));

  switch(fr)
  {
    case Fruit::Apple:
      makeApplePie(flesh);
      break;
    case Fruit::Banana:
      makeBananaBread(flesh);
      break;
    case Fruit::Orange:
      makeOrangeJuice(flesh);
      break;
    case Fruit::Lemon:
      makeLemonade(flesh);
      break;
    default:
      ReportAndTerminate("unknown fruit");
  }
}
```

Listing 27

```
std::string
FruitToString(
  Fruit::Fruit const& fr
)
{
  switch(fr)
  {
    // Valid values
    . . .

    // Invalid values
    case Fruit::Unknown:
    default:
      break;
  }
  log(SEV_EMRG, "invalid fruit value %d", fr);
  ::exit(1);
}
```

Listing 28

```
std::string
FruitToString(
  Fruit::Fruit const& fr
)
{
  switch(fr)
  {
    // Valid values
    . . .

    // Invalid values
    case Fruit::Unknown:
    default:
      break;
  }
  log(SEV_EMRG, "invalid fruit value %d", fr);
  char const fmt[] =
    "invalid fruit value:%d";
  char buff[50]; // can't exceed 41
  sprintf(buff, fmt, fr);
  throw bad_fruit_exception(buff);
}
```

Listing 29

```
int main()
{
  flesh_t      flesh;
  Fruit::Fruit fr = ReadInFruit(flesh);
  // Filtering
  if(!IsValidFruit(fr))
  {
    log(L_ERROR, FruitToString(fr));
    . . .// do contingent reporting to user, and
         // other important program logic actions
  }
  else
  {
    // Internal logic
    UseFruit(ff, flesh);
  }
  return 0;
}
```

definition it'll treat the invalid fruit as a design violation, and kill the process (or throw an exception), and the user won't receive the intended contingent reporting, and other important program logic actions will be skipped.

Consequently, I recommend always writing enumeration to string conversions as if they're intended for the filtering layer. It's not their business to do filtering, that's what **`IsValidFruit()`** is for, and they needn't be written to catch design violations, since contract enforcements are always removable.

Conversion from string

The picture when converting from a string to an enumeration is largely the inverse of what we've just seen (without all the contract programming vs. defensive programming hoopla). Commonly, it's a series of **`if`**-statements, but it may be a lookup into a table or a hash/dictionary of names.

Listing 30

```
Fruit::Fruit
FruitFromString(
  std::string const& s
)
{
  // Valid values
  if("Apple" == s)
  {
    return case Fruit::Apple;
  }
  else
  if("Banana" == s)
  {
    return se Fruit::Banana;
  }
  else
  if("Orange" == s)
  {
    return se Fruit::Orange;
  }
  else
  if("Lemon" == s)
  {
    return se Fruit::Lemon;
  }
  // Invalid values
  else
  {
    return Fruit::Unknown;
  }
}
```

Listing 31

```
Fruit::Fruit
FruitFromString(
  std::string const& s
)
{
  if(0)
  {}
  // Valid values
#define VALID_VALUE(nm, val) \
  else if((nm)==s){return(val);}
  VALID_VALUE("Apple", Fruit::Apple)
  VALID_VALUE("Banana", Fruit::Banana)
  VALID_VALUE("Orange", Fruit::Orange)
  VALID_VALUE("Lemon", Fruit::Lemon)
#undef VALID_VALUE
  // Invalid values
  else
  {
    return Fruit::Unknown;
  }
}
```

For example, we might convert back from string to **`Fruit`** as shown in Listing 30.

It's not glamorous, and it wouldn't be efficient if there were a large number of enumerators, but it is effective. It can be reasonably improved with a macro, as in Listing 31.

The argument against throwing an exception (or terminating) is less strong than in the **`FruitToString()`** scenario, but I prefer to do neither because (i) the semantics are simple and clear (and language independent) and (ii) they're symmetrical with **`FruitToString()`**.

There's an obvious maintenance problem when implementing these complementary functions: it's all too easy to forget to update one or the other in light of new enumerators. (Of course, it's also easy to forget to update both!) Worse, it's possible to update both functions but make a mistake and have them use different strings (e.g. **`"Apple"`** and **`"Apple"`**, **`"Lemon"`** and **`"lemon"`**).

One thing to bear in mind is that you might want to be permissive with case, and maybe even with leading/trailing whitespace. Even though you control the precise output form (in **`FruitToString()`**) it's conceivable that out there in the world the case/spacing might change. For example, an XML file may be manually edited by someone debugging their client of your server, and inadvertently change the case/spacing in a way that should (given the flexibility of human-based formats) be permitted. Obviously, this has to be judged on a case-by-case basis.

Hazardous efficient conversions

If your enumeration is 0-based, and if it has contiguous values, and if it has no duplicates, and if there's no chance that it'll be changed – famous last words? – and if you do not need to recognise invalid enumerator values, then you can use simple indexing to access a string literal corresponding to an enumerator, as shown in Listing 32. Its advantage is that it has extremely low runtime costs, just a few cycles of pointer arithmetic.

I've seen this advocated, and even used on occasion, but there are serious problems if any of those ifs become buts. At best, you'll have a rapid crash after returning a pointer to Bob-knows-where; at worst, you'll return the wrong 'fruit name' and carry on in ignorance that your program has violated its design. I strongly recommend that you ignore any temptation to use this mechanism.

A robust and flexible compromise

Interestingly, there's a compromise, which allows for efficient indexing in the case where it is appropriate without any of the risks attendant with a pure indexing approach. It involves using a table of value-string pairs,

Listing 32

```
char const*
FruitToString( Fruit::Fruit const& fr )
{
  static char const* const strings[] =
  {
    "<unknown-fruit>",
    "Apple",
    "Banana",
    "Orange",
    "Lemon",
  };
  return strings[int(fr)];
}
```

attempting a checked index into it, otherwise doing a linear search, with a default response if not found. This same table can be used for implementing the opposite conversion. For **`Fruit`**, it would look like Listing 33.

Hopefully the advantages of this solution are clear:

- There is only one set of strings, shared by both conversion to and from string forms.
- The conversion to string is (almost) as fast as the fragile indexed version for 0-based contiguous enumerations, but will work correctly regardless.
- Because you've got control of the 'default' case, you can choose to return **`std::string`** and capture the bad value.

The only issues are that it won't work for duplicates (because that's impossible) and can't (as it's written) break apart flag combinations (e.g. **`SearchFlagsToString(directories | files | absolutePath)`**), so is best suited to enumerated types. (C# programmers will know that the .NET enumeration class is able to do that; I'll leave it as an exercise for the reader to devise a scheme for C++.)

Note, if you have a lot of enumerators and really need speed, consider ordering them lexicographically (with respect to the label), and use a binary chop.

C++0x enum class

The type-safety of an enum in C++ is pretty weak (and it's even weaker in C). Any enumeration can be implicitly converted to an **`int`**, although that's not so in reverse. For example, the following code will compile, even though we doubtless wish it would not:

```
abs(Apple);
```

Nonetheless, the (C++) compiler is still able to do some useful things. We can overload a function for any number of enumeration types (along with other types), and get the behaviour we want, as in:

```
enum Vegetable { . . . };
dump_to_log(char const* s);
dump_to_log(double d);
dump_to_log(Fruit f);
dump_to_log(Vegetable v);
```

The problems arise if, say, we'd provided the **`int`** overload, but not one for the specific enumeration we wanted. Again, the following code will compile when we likely wish it would not.

```
enum Vegetable { . . . };
dump_to_log(char const* s);
dump_to_log(double d);
dump_to_log(Fruit f);
dump_to_log(int i);
Vegetable veg = . . .
dump_to_log(veg);
```

One way to get around this is to have **`dump_to_log()`** be a function template, and apply the pedantic pointer idiom [7], along the lines of Listing 34.

Listing 33

```
namespace
{
  struct Fruit_pair_t
  {
    Fruit::Fruit  fruit;
    char const*   label;
  };
  static Fruit_pair_t const Fruit_pairs[] =
  {
    { Fruit::Unknown, "<unknown-fruit>" },
    { Fruit::Apple, "Apple" },
    { Fruit::Banana, "Banana" },
    { Fruit::Orange, "Orange" },
    { Fruit::Lemon, "Lemon" },
  };
} // anonymous namespace
char const*
FruitToString(
  Fruit::Fruit const& fruit
)
{
  size_t const N = (sizeof(Fruit_pairs) /
                    sizeof(0[Fruit_pairs]));
  int const ix = static_cast<int>(fruit);
  // Fast lookup with check
  if( ix >= 0 &&
    size_t(ix) < N)
  {
    Fruit_pair_t const& fp = Fruit_pairs[ix];
    if(fp.fruit == fruit)
    {
      return fp.label;
    }
  }
  // Linear search
  { for(size_t i = 0; i != N; ++i)
  {
    Fruit_pair_t const& fp = Fruit_pairs[i];
    if(fp.fruit == fruit)
    {
      return fp.label;
    }
  }}
  // Not found
  return "<invalid-fruit>";
}
Fruit::Fruit
FruitFromString(
  std::string const& s
)
{
  size_t const N = (sizeof(Fruit_pairs) /
                    sizeof(0[Fruit_pairs]));
  // Linear search
  { for(size_t i = 0; i != N; ++i)
  {
    Fruit_pair_t const& fp = Fruit_pairs[i];
#ifdef FRUIT_IGNORE_CASE
    if(fp.label == s)
#else /* ? FRUIT_IGNORE_CASE */
    if(fp.label == s)
#endif /* FRUIT_IGNORE_CASE */
    {
      return fp.fruit;
    }
  }}
  // Not found
  return Fruit::Unknown;
}
```

That works because even though a `Vegetable` is implicitly convertible to an `int`, a `Vegetable const*` is not the least convertible to `int const*`. I use this technique in the `FastFormat` formatting library API to avoid unwanted implicit conversions [7].

Still, it's more than a little verbose, and a pain to have to go to these lengths. Thankfully C++0x has taken this on board, and has introduced an enumeration type that is more strongly typed / less convertible. Simply apply the `class` keyword along with `enum`, as follows, and you have a type that refuses to play with `int` unless casts are applied.

```
enum class Vegetable
{
  Unknown
, Broccoli
, Carrot
, Cauliflower
, Potato
};
Vegetable veg = Vegetable::Unknown;
dump_to_log(veg); // Compile error
```

(Note: this new enumeration form also has scoped enumerator names: their names must be qualified by the enumeration, as shown above.)

Summary

Naming

Use a prefix on enumerator names if you are using C, or need to be compatible with both C and C++; use enum-named namespace if you need only be compatible with C++.

Values

For enumerated types:

- Make sure the 0 value represents an invalid state/value (except when defining a return/status code enumeration [8])
- Use a `MAX_<myenum>_VALUE` sentinel
- Never change/remove/move existing enumerators, except the `MAX_<myenum>_VALUE` sentinel
- Always insert at the end of the list (just before the `MAX_<myenum>_VALUE` sentinel).

For flag types (and combinations):

- Prefer hexadecimal if you're comfortable with it. (And maybe get comfortable with it if you're not.)
- Consider refactoring to avoid combinations, if it's not too late. If it is, use masks to help authors and users alike.

Validity

If your enumeration is exposed to the outside world, validate it in a dedicated function. Don't perform validation in conversion functions, though by all means enforce contracts in other areas of the codebase.

Conversion

Don't log from within an enum→string conversion function when an unrecognised enumerator value is found, as it may be being called from a log statement.

Don't exit/throw from within an enum→string conversion function when an unrecognised enumerator value is found, because it may be being called in the filtering layer. Enum→conversion functions are not there for validation, that's what an `IsValid<MyEnum>()` is for. They should, in this regard, be 'validity agnostic'.

Never use the raw indexed version: it's too fragile.

Consider whether to be permissive in string→enum conversions.

Consider using a table-based conversion mechanism, since it supports both enum→string and string→enum with a single set of labels, can do indexed lookup when possible, and is otherwise robust. ■

Listing 34

```
void do_dump_(char const* s, char const* const*);
void do_dump_(double d, double const*);
void do_dump_(int i, int const*);
void do_dump_(Fruit f, Fruit const*);
template <typename T>
void dump_to_log(T const& v);
{
  do_dump_(v, &v);
}
```

References

[1] *The Pragmatic Programmer*, Andy Hunt and Dave Thomas, Addison-Wesley, 1999.

[2] http://en.wikipedia.org/wiki/Enumerated_type

[3] *Code Complete*, 2nd Edition, Steve McConnell, Microsoft Press, 2004.

[4] The design of this class, and general discussions for how to map various search API paradigms to STL sequence concepts, are given in *Extended STL, volume 1: Collections and Iterators*, Matthew Wilson, Addison-Wesley, 2007.

[5] http://synesis.com.au/software/b64.html

[6] *Object-Oriented Software Construction*, 2nd Edition, Bertrand Meyer, Prentice Hall, 1997.

[7] 'An Introduction to FastFormat (Part 2): Custom Argument and Sink Types', Matthew Wilson, *Overload 90*, April 2009.

[8] When defining an enumeration that is to stand for a status code, it is usual for the 0-value to be the no-error value and is therefore also valid.

A Game of Path Finding

Baron Muncharris sets a challenge.

Welcome Sir R-----! Pray shed your overcoat and come dry yourself by the fire. I am told that these spring showers are of inestimable benefit to farming folk, but I fail to grasp why they can't show the good manners to desist until noblemen have made their way indoors.

Will you join me in a warming measure and perchance a small wager?

I had no doubt sir!

I've a mind for a game oft played by the tribesmen of Borneo upon the cobbled floors of their homes as a means of practice for their legendary talent in forging paths through the dense forests in which they dwell.

As you have no doubt heard tell, their arboreal paradise is haunted by the spirit of a rather cantankerous old fellow with bright orange hair; one who does not gladly share his territory with the living.

Any unfortunate soul who finds himself in some spot that this flame-haired shade has chosen for his own is liable to be assaulted with a hailstorm of the gigantic nuts that grow in those parts. The tribesmen have learned full well that a pile of those nuts upon the forest floor is a certain indication that another path should be sought and it is a testament to the improving quality of their game that they can unerringly do so.

Oh, but I have told you nothing of its rules!

I have chalked out a board upon the tiles of the hearth (Figure 1).

Take this stock of oak counters and I shall have this stock of pine.

I shall begin play by claiming a tile for my own and placing one of my counters upon it. You shall follow by likewise claiming some other tile for yourself.

Play shall continue in this fashion with our taking turns laying claim to untaken tiles.

I shall have the game, and a coin from your purse, if I can weave an unbroken path of counters from the left edge to the right, made up of neighbouring tiles, like so (Figure 2).

You shall have the game, and a coin from mine, if you can frustrate my efforts so that, even after every tile has been claimed, I have no such path (Figure 3).

When I told that tedious student of this game he mumbled something about a top quality curse having rendered him bored of his station, although quite what excitement he found in the musty halls of academe before this supposed affliction I cannot begin to imagine.

But there is no profit in recounting his ramblings; come, take a fresh glass and muse upon your strategy! ■

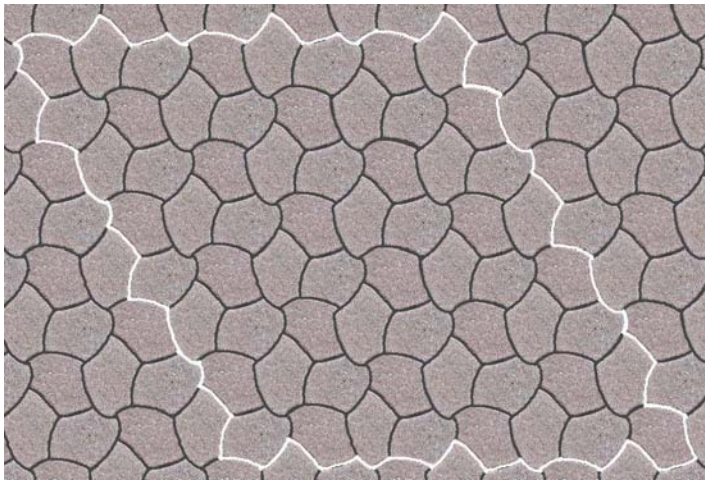


Figure 1

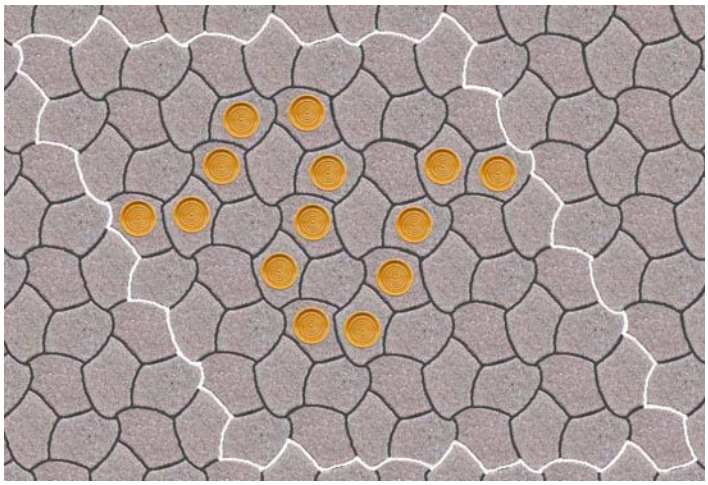


Figure 2

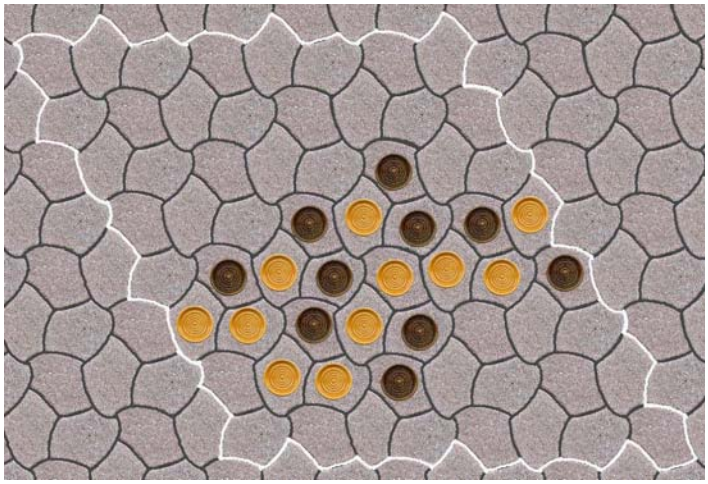


Figure 3

Figures

Figure 1: The board

FIgure 2: A path for the Baron

Figure 3: Sir R----- loses!

BARON MUNCHARRIS

In the service of the Russian military Baron Muncharris has travelled widely in this world, and many others for that matter, defending the honour and the interests of the Empress of Russia. He is renowned for his bravery, his scrupulous honesty and his fondness for a wager.

On a Game of One Against Many

A student performs his analysis.

Recall that the Baron proposed a pair of dice contests in which Sir R----, were he to best the Baron's score, stood to win a bounty of thirteen coins.

Upon paying his stake Sir R----- was to cast his die but, if unhappy with its outcome, could pay a further coin to cast it again. Likewise, if he were not satisfied with the second cast, he could elect to cast a third time for a further two coins. He could continue in this fashion for as long as he pleased with the cost rising by one coin for each additional cast of his die.

The Baron was to have but a single cast of his die, with Sir R----- to determine whether after or before his own play according to his stake; seven coins for the former and eight for the latter.

In order to figure the fairness of these wagers it is first necessary to recognise that there will inevitably come a point beyond which it is not in Sir R-----'s interest to draw out his play, for the cost of doing so rises inexorably whilst the prize, should he best the Baron, rests in constancy.

Specifically, Sir R----- should refrain from casting his die if he should expect on the average to be further out of pocket in doing so. If we figure when this will be we can work backwards from his final cast including the advantage, if any, he might derive from further casts as we proceed.

I said as much to the Baron, but I do not believe that I had his full attention.

Now, Sir R----- will be most compelled to recast his die if his will assuredly lose the wager if he does not. Indeed, he should only choose not if his expected winnings should be less than the cost of casting. Fortunately for our reckoning we need not consider further casts after this since, given that their cost shall grow ever the greater, their outcomes for Sir R----- shall grow ever the worser.

In the first game, the probability that Sir R----- will best the Baron if he casts a one is zero. If he should cast a two, he has one chance in six of winning. If a three, then two chances in six, &c.

The probability that he should win on his final cast is therefore simply

$$\tfrac{1}{6}\times 0+\tfrac{1}{6}\times\tfrac{1}{6}+\tfrac{1}{6}\times\tfrac{2}{6}+\tfrac{1}{6}\times\tfrac{3}{6}+\tfrac{1}{6}\times\tfrac{4}{6}+\tfrac{1}{6}\times\tfrac{5}{6}=\tfrac{15}{36}=\tfrac{5}{12}$$

His average expected winnings from that cast should consequently be

$$\tfrac{5}{12}\times 13=5\tfrac{5}{12}$$

and he should in no circumstance elect to proceed if the cost of doing so exceeds five coins.

From this it is readily apparent that Sir R----- can expect, on the average, a prize of five twelfths of a coin for his sixth cast of the die.

To figure the winnings Sir R----- might expect from his fifth cast, we shall multiply the formula for his probability of winning by the prize

$$\begin{aligned}&\left(\tfrac{1}{6}\times 0+\tfrac{1}{6}\times\tfrac{1}{6}+\tfrac{1}{6}\times\tfrac{2}{6}+\tfrac{1}{6}\times\tfrac{3}{6}+\tfrac{1}{6}\times\tfrac{4}{6}+\tfrac{1}{6}\times\tfrac{5}{6}\right)\times 13\\&\quad=\tfrac{1}{6}\times 0+\tfrac{1}{6}\times\tfrac{13}{6}+\tfrac{1}{6}\times\tfrac{26}{6}+\tfrac{1}{6}\times\tfrac{39}{6}+\tfrac{1}{6}\times\tfrac{52}{6}+\tfrac{1}{6}\times\tfrac{65}{6}\end{aligned}$$

Now, if Sir R----- were to cast a one, it would be in his interest to cast again since his expected winnings would be greater than the zero bounty he stood to take.

If we figure this into our calculation we have instead an expected prize of

$$\begin{aligned}&\tfrac{1}{6}\times\tfrac{5}{12}+\tfrac{1}{6}\times\tfrac{13}{6}+\tfrac{1}{6}\times\tfrac{26}{6}+\tfrac{1}{6}\times\tfrac{39}{6}+\tfrac{1}{6}\times\tfrac{52}{6}+\tfrac{1}{6}\times\tfrac{65}{6}\\&\quad=\tfrac{5}{72}+\tfrac{26}{72}+\tfrac{52}{72}+\tfrac{78}{72}+\tfrac{104}{72}+\tfrac{130}{72}\\&\quad=\tfrac{395}{72}\\&\quad=5\tfrac{35}{72}\end{aligned}$$

Finally, we must subtract the cost of four coins to yield Sir R-----'s net gain from a fifth cast of one seventy second part of a coin shy of one and a half coins.

To figure the value of this wager to Sir R----- we must simply continue in this fashion, replacing expected winnings smaller than can be got from continuing play, until we reach the first cast of the die.

For the fourth cast we consequently have a net average take of

$$\begin{aligned}&\tfrac{1}{6}\times\tfrac{107}{72}+\tfrac{1}{6}\times\tfrac{13}{6}+\tfrac{1}{6}\times\tfrac{26}{6}+\tfrac{1}{6}\times\tfrac{39}{6}+\tfrac{1}{6}\times\tfrac{52}{6}+\tfrac{1}{6}\times\tfrac{65}{6}-3\\&\quad=\tfrac{107}{432}+\tfrac{156}{432}+\tfrac{312}{432}+\tfrac{468}{432}+\tfrac{624}{432}+\tfrac{780}{432}-3\\&\quad=\tfrac{2447}{432}-3\\&\quad=5\tfrac{287}{432}-3\\&\quad=2\tfrac{287}{432}\end{aligned}$$

Likewise, for the third

$$\begin{aligned}&\tfrac{1}{6}\times\tfrac{1151}{432}+\tfrac{1}{6}\times\tfrac{1151}{432}+\tfrac{1}{6}\times\tfrac{26}{6}+\tfrac{1}{6}\times\tfrac{39}{6}+\tfrac{1}{6}\times\tfrac{52}{6}+\tfrac{1}{6}\times\tfrac{65}{6}-2\\&\quad=5\tfrac{1223}{1296}-2\\&\quad=3\tfrac{1223}{1296}\end{aligned}$$

and for the second

$$\begin{aligned}&\tfrac{1}{6}\times\tfrac{5111}{1296}+\tfrac{1}{6}\times\tfrac{5111}{1296}+\tfrac{1}{6}\times\tfrac{26}{6}+\tfrac{1}{6}\times\tfrac{39}{6}+\tfrac{1}{6}\times\tfrac{52}{6}+\tfrac{1}{6}\times\tfrac{65}{6}-1\\&\quad=6\tfrac{1439}{3888}-1\\&\quad=5\tfrac{1439}{3888}\end{aligned}$$

and, finally, for the first

$$\begin{aligned}&\tfrac{1}{6}\times\tfrac{20879}{3888}+\tfrac{1}{6}\times\tfrac{20879}{3888}+\tfrac{1}{6}\times\tfrac{20879}{3888}+\tfrac{1}{6}\times\tfrac{39}{6}+\tfrac{1}{6}\times\tfrac{52}{6}+\tfrac{1}{6}\times\tfrac{65}{6}-0\\&\quad=7\tfrac{143}{7776}\end{aligned}$$

Sir R----- will clearly earn, on average, a bounty greater than his stake and I would therefore in good conscience have recommended that he take the first of the Baron's wagers.

In the second of the Baron's wagers he was to cast his die before Sir R----- began his play. Sir R-----'s strategy should therefore have been informed by the Baron's score.

For each of the Baron's possible scores we can figure the final cast that Sir R----- should countenance in exactly the same fashion as we did for the first game.

For example, had the Baron cast a six, Sir R----- should have immediately cut his losses since he should have surely lost the wager no matter how many casts he took.

If the Baron had instead cast a five, then on each cast Sir R-----'s should have had one chance in six of besting him. On his final cast he should therefore have expected an average prize of

$$\tfrac{1}{6}\times 13=2\tfrac{1}{6}$$

and he should have in no event elected to cast his die more than three times.

Given that his third cast would have cost two coins his expected net gain would have been one sixth part of a coin.

Should he not best the Baron on his second cast he should most certainly cast again, so his expected winnings are given by

$$\tfrac{1}{6}\times 13+\tfrac{5}{6}\times\tfrac{1}{6}-1=\tfrac{78}{36}+\tfrac{5}{36}-\tfrac{36}{36}=\tfrac{47}{36}=1\tfrac{11}{36}$$

and, on his first cast by

$$\tfrac{1}{6}\times 13+\tfrac{5}{6}\times 1\tfrac{11}{36}-0=\tfrac{468}{216}+\tfrac{235}{216}-0=\tfrac{703}{216}=3\tfrac{55}{216}$$

Now, if the Baron should have cast a four then Sir R----- would have had two chances in six of winning on each cast and would have consequently had an average prize of

$$\tfrac{2}{6}\times 13=4\tfrac{1}{3}$$

and should therefore not have cast his die more than five times.

Working backwards from his fifth cast we have

$$5:\quad 4\tfrac{1}{3}-4=\tfrac{1}{3}$$
$$4:\quad \tfrac{2}{6}\times 13+\tfrac{4}{6}\times\tfrac{1}{3}-3=1\tfrac{5}{9}$$
$$3:\quad \tfrac{2}{6}\times 13+\tfrac{4}{6}\times 1\tfrac{5}{9}-2=3\tfrac{10}{27}$$
$$2:\quad \tfrac{2}{6}\times 13+\tfrac{4}{6}\times 3\tfrac{10}{27}-1=5\tfrac{47}{81}$$
$$1:\quad \tfrac{2}{6}\times 13+\tfrac{4}{6}\times 5\tfrac{47}{81}-0=8\tfrac{13}{243}$$

Next, we must consider the case of the Baron casting a three. Sir R-----'s expected winnings on his final cast will be

$$\tfrac{3}{6}\times 13=6\tfrac{1}{2}$$

and thus his last cast should be his seventh.

Working retrograde once again we have

$$7:\quad 6\tfrac{1}{2}-6=\tfrac{1}{2}$$
$$6:\quad \tfrac{3}{6}\times 13+\tfrac{3}{6}\times\tfrac{1}{2}-5=1\tfrac{3}{4}$$
$$5:\quad \tfrac{3}{6}\times 13+\tfrac{3}{6}\times 1\tfrac{3}{4}-4=3\tfrac{3}{8}$$
$$4:\quad \tfrac{3}{6}\times 13+\tfrac{3}{6}\times 3\tfrac{3}{8}-3=5\tfrac{3}{16}$$
$$3:\quad \tfrac{3}{6}\times 13+\tfrac{3}{6}\times 5\tfrac{3}{16}-2=7\tfrac{3}{32}$$
$$2:\quad \tfrac{3}{6}\times 13+\tfrac{3}{6}\times 7\tfrac{3}{32}-1=9\tfrac{3}{64}$$
$$1:\quad \tfrac{3}{6}\times 13+\tfrac{3}{6}\times 9\tfrac{3}{64}-0=11\tfrac{3}{128}$$

In the same fashion we can figure that should the Baron cast a two, Sir R----- should throw his die no more than nine times and his expected winnings should be

$$9:\quad \tfrac{4}{6}\times 13-8=\tfrac{2}{3}$$
$$8:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times\tfrac{2}{3}-7=1\tfrac{8}{9}$$
$$7:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times 1\tfrac{8}{9}-6=3\tfrac{8}{27}$$
$$6:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times 3\tfrac{8}{27}-5=4\tfrac{62}{81}$$
$$5:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times 4\tfrac{62}{81}-4=6\tfrac{62}{243}$$
$$4:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times 6\tfrac{62}{243}-3=7\tfrac{548}{729}$$
$$3:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times 7\tfrac{548}{729}-2=9\tfrac{548}{2,187}$$
$$2:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times 9\tfrac{548}{2,187}-1=10\tfrac{4,922}{6,561}$$
$$1:\quad \tfrac{4}{6}\times 13+\tfrac{2}{6}\times 10\tfrac{4,922}{6,561}-0=12\tfrac{4,922}{19,683}$$

Finally and, given the growing complexity of the figures thus far, most dauntingly we must reckon the value of the wager should the Baron have cast a one.

$$11:\quad \tfrac{5}{6}\times 13-10=\tfrac{5}{6}$$
$$10:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times\tfrac{5}{6}-9=1\tfrac{35}{36}$$
$$9:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 1\tfrac{35}{36}-8=3\tfrac{35}{216}$$
$$8:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 3\tfrac{35}{216}-7=4\tfrac{467}{1,296}$$
$$7:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 4\tfrac{467}{1,296}-6=5\tfrac{4,355}{7,776}$$
$$6:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 5\tfrac{4,355}{7,776}-5=6\tfrac{35,459}{46,656}$$
$$5:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 6\tfrac{35,459}{46,656}-4=7\tfrac{268,739}{279,936}$$
$$4:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 7\tfrac{268,739}{279,936}-3=9\tfrac{268,739}{1,679,616}$$
$$3:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 9\tfrac{268,739}{1,679,616}-2=10\tfrac{3,627,971}{10,077,696}$$
$$2:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 10\tfrac{3,627,971}{10,077,696}-1=11\tfrac{33,861,059}{60,466,176}$$
$$1:\quad \tfrac{5}{6}\times 13+\tfrac{1}{6}\times 11\tfrac{33,861,059}{60,466,176}-0=12\tfrac{275,725,763}{362,797,056}$$

We have one last and most tedious calculation to perform. To figure Sir R-----'s expected bounty we must average these results.

$$\tfrac{1}{6}\times\left(0+3\tfrac{55}{216}+8\tfrac{13}{243}+11\tfrac{3}{128}+12\tfrac{4,922}{19,683}+12\tfrac{275,725,763}{362,797,056}\right)$$

That figuring the result of this average was a Herculean feat of arithmetic hardly bears mention!

Have I not erred then Sir R----- should expect from this wager, on the average, a bounty of

$$7\tfrac{1,937,927,123}{2,176,782,336}$$

and I could not therefore have recommended it to him for a stake of eight coins.

That the reckoning of the fairness of the Baron's second wager was substantially more difficult than that of his first is all the more curious if we consider a slight change to these games. Specifically, let us imagine that the Baron had shaken his die in a cup and upended it upon the table before Sir R----- began his play.

If he revealed his score after Sir R----- had declared that he was done then the reckoning of the game would be identical to that of the first of the Baron's wagers; if before, then identical to that of the second.

The Baron's score in both games is determined at the outset; the only difference being when Sir R----- learns of it.

That the earlier informed Sir R----- is of the state of play, the harder it is for him to determine the consequences of entering into the Baron's wager has caused the more philosophically minded of my fellows some small consternation.

For my part, I am content; it is so and that is reason enough! ■

Smarter, Not Harder

Pete Goodliffe helps us to pick our battles.

Battles are won by slaughter and manoeuvre.
The greater the general, the more he contributes in manoeuvre, the less he demands in slaughter.

– Winston Churchill

Let me tell you a story. It's all true. A colleague, working on some UI code, needed to overlay pretty rounded arrows over his display. After he struggled to do it programmatically using the drawing primitives provided, I suggested he just overlay a graphic on the screen. That would be much easier to implement.

So off he went. He fired up Photoshop. And fiddled. And tweaked. And fiddled some more. In this, the Rolls-Royce of image composition applications, there is no quick-and-easy way to draw a rounded arrow that looks halfway decent. Presumably an experienced graphic artist could knock one up in two minutes. But after almost an hour of drawing, cutting, compositing, and rearranging, he still didn't have a convincing rounded arrow.

He mentioned it to me in frustration as he went to make a cup of tea.

On his return, tea in hand, he found a shinny new rounded arrow image sitting on his desktop ready for use.

'How did you do that so quickly?' he asked.

'I just used the right tool.' I replied, dodging a flying mug of tea.

Photoshop *should* have been the right tool. It's what most image design work is done in. But I knew that Open Office provides a handy configurable rounded arrow tool. I had drawn one in ten seconds and sent him a screenshot. It wasn't elegant. But it worked.

The moral?

There is a constant danger of focusing too closely on one tool, or on a singular approach to solve a problem. It's tantalisingly easy to lose hours of effort exploring its blind alleys when there's a simpler, more direct route to your goal.

So how can we do better?

Pick your battles

To be a productive programmer, you need to learn to work *smarter* rather than *harder*. One of the hallmarks of experienced programmers is not just their technical acumen, but how they solve problems and pick their battles.

Good programmers get things done quickly. Now, they *don't* bodge things like a shoot-from-the-hip cowboy coder. They just work smart. This is not necessarily because they are more clever; they just know how to solve problems *well*. They have an armoury of experience to draw from that will guide them to use the correct approach. They can see lateral solutions – the application of an unusual technique that will get the job done with less hassle. They know how to chart a route around looming obstacles. They can make informed decisions about where best to invest effort.

Battle tactics

Here are some simple ideas to help you work smarter:

- Don't write a lump of code yourself when you can use an existing library, or can repurpose code from elsewhere.

  Even if you have to pay for a third-party library, it is often far more cost effective to take an off-the-shelf implementation than to write your own. And test your own. And then debug your own...

  Overcome *'not invented here' syndrome*. Many people think that they can do a much better job themselves, or fashion a more appropriate version for their specific application. Is that *really* the case? Even if the other code isn't designed exactly how you prefer, just use it. You don't necessarily need to rewrite it if it's working already. Make a facade around it if you must to integrate into your system.

- Don't work out how to do a task yourself if someone already knows how to do it. You might like to bask in the glory of the accomplishment. You might like to learn something new. But if someone else can give you a leg-up, or complete the job much faster than you, then it may be better to put the task in their work queue instead.

- Consider sacrilege: Do you need to refactor? Do you need to unit test? I'm a firm advocate of both practices, but sometimes they might not be appropriate or a worthwhile investment of your time. Yes, yes: refactoring and unit testing do both bring great benefits and should never be tossed aside thoughtlessly. However, if you're working on a small prototype, or exploring a possible functional design with some throw-away code, then you might be better off saving the theologically correct practices for later.

  If you do avoid being tainted by a lack of unit tests, instead consider exactly *which* tests to write. A stubborn test-every method approach is not sensible. (Often you'll think you have better coverage than you expect). For example, it's not high priority to test every single getter and setter in your API [1]. Instead, focus your testing efforts on the places you would likely expect brittleness. Pick your testing battles.

- If you're presented with multiple design options and you're not sure which solution to pick don't waste hours cogitating about which is best if a quick 'spike' solution (a throw-away prototype) might generate more useful answers in minutes. To make this work well, set a specific Pomodoro-like time window within which you will perform the spike [2]. Stop when the time elapses. (And in true Pomodoro style, get yourself a nice hard-to-ignore wind-up timer to force you to stop.) Use tools that will help you backtrack quickly (e.g. effective version control tools).

- Prioritise your work list. Do the most important things first.

  Be rigorous about this. Don't get caught up on unimportant minutiae; it's incredibly easy to do. Especially when one simple job turns out to depend on another simple job. Which depends on another simple job, which depends on... After two hours you'll surface from a rabbit hole and wonder why on earth you're reconfiguring the mail server on your computer when what you wanted to do was add a method to add an item to a list. In computer folklore, this is referred to as *Yak Shaving* [3].

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

- Do one thing at a time. It's hard to focus on more that one job at once (especially for men with our uni-tasking brains). If you try to work concurrently, you'll do both jobs badly. Finish one job then move on to another. You'll get both jobs completed in a shorter space of time.
- When you are given a new task, check what's *really* needed now. What does the customer actually need you to deliver? Don't implement the Rolls-Royce full bells-and-whistles version if it's not necessary. Even if the work request asks for it, push back and verify what is genuinely required. To do this, you need to know the context your code lives in.

  This isn't just laziness. There is a danger in writing too much code too early. The Pareto Principle [4] implies that 80% of required benefit could come from just 20% of the full implementation. Do you really need to write the remainder of that code, or could your time be better employed elsewhere?
- Keep your code and design as small and as simple as possible. Otherwise, you'll just add a lot more code that will cost you time and effort to maintain in the future.

  You *will* need to change it; you can never foretell exactly what the future requirements are. Predicting the future is an incredibly inexact science. It is easier and smarter to make your code malleable to change now, than it is to build in support for every possible future feature on day one.
- Some things that are hard (like code integration) should *not* be avoided because they are hard. Many people do so, to try to minimise the pain. It sounds like picking your battles, doesn't it?

  In reality, the smarter thing is to do them sooner and face the pain when it is smaller. It's easier to integrate small pieces of code early on, and then to frequently integrate the subsequent changes, than it is to work on three major features for a year and then try to stitch them together at the end. The same goes for unit testing: write tests now, alongside your code. It'll be far harder and less productive if you wait until the code is 'working' until you write the tests. As the saying goes: *If it hurts, do it more often*.
- Remember the classic advice: *if you have to do it more than once, write a script to do it for you*. Clearly, automating a common tedious task could save you many hours of effort. Consider this also for a single task that has a high degree of repetition. It might be faster to write a tool and run it once, than to do the job by hand yourself.

  This has the added advantage that it helps others to work smarter too. If you can run a build with one command, rather than a script of 15 complex commands and button presses, then your entire team can build more easily, quickly, and newcomers can get up-to-speed faster.

  To aid this automation, the experienced programmer will naturally pick automatable tools, even if they don't intend to automate anything right now. Favour workflows that produce plain text, or XML formatted intermediate files. Select tools that have a command-line interface as well as (or instead of) an inflexible GUI panel.

  It can be very hard to know ahead of time if it's worth writing a script for a task. Obviously, if you are likely to do it again then it's a good candidate. But other times, you might end up *wasting* time writing the script.
- Find errors sooner, so you don't spend too long doing the wrong thing. To achieve this:
  - Show your product to customer early and often, so you'll find out quickly if you're building them the wrong thing.
  - Discuss your code design with others, so you'll find out if there's a better way to structure your solution earlier. Don't invest effort in bad code if you can avoid it.
  - Code review small, understandable bits of work often, not large dense bits of code.
  - Unit test code from the outset. Ensure the unit tests are run frequently to catch errors before they bite you.
- Learn to communicate better. Learn to ask the right questions to understand unambiguously, so you don't end up re-working later on or having to wait for more answers to outstanding questions.

  And learn how to run productive meetings so your life is not sucked out by the demons who sit in the corners of meeting rooms.
- Don't burn yourself out working silly hours, leading people to expect unrealistic levels of work from you all the time. Make it clear if you are moving beyond the call of duty, so people learn not to expect it too often!

If someone else can give you a leg-up, or complete the job much faster than you, then it may be better to put the task in their work queue instead

- Beware of the many small tasks you do that aren't that important. Email, paperwork, phone calls; the administrivia. Instead of doing those little things throughout the day, interrupting and distracting you from your flow on important tasks, batch them together and do them at one (or a few) set times each day.

  You may find it helps to write these tasks down on a small 'to do' list, and at a set time start processing them as quickly as possible. Ticking them off your list – the sense of accomplishment can be a motivating reward.
- Always look out for new tools that will power-boost your workflow. But don't become a slave to finding new software. Often new software has sharp edges that could cut you. Favour tried-and-tested tools that many people have used. You can't put a price on the collected knowledge of these tools available via Google.

Conclusion

Pick your battles. Work smarter, not harder. They are trite maxims. But true.

Of course, this doesn't mean *don't work hard*. Unless you want to get fired. But that's not smart.

Questions

1. How do you determine the right amount of testing to apply to your work? Do you rely on experience or guidelines? Look back over your last month's work; was it really tested adequately?
2. How good are your at prioritising your workload? How can you improve?
3. How do you ensure you find issues as soon as possible? How many errors or re-workings have you had to perform that could have been avoided?
4. Do you suffer from *not invented here* syndrome? Is everyone else's code rubbish? Could you do better? Can you stomach incorporating other's work in your own? ■

Notes

[1] It's another issue whether you should have getters and setters in your APIs in the first place...

[2] The Pomodoro Technique. http://www.pomodorotechnique.com/

[3] Yak Shaving. http://catb.org/jargon/html/Y/yak-shaving.html

[4] The Pareto Principle. For many events, roughly 80% of the effects come from 20% of the causes http://en.wikipedia.org/wiki/Pareto_principle

Concurrency, Parallelism and D

David Simcha explains message passing for parallel programs in D.

If you've followed advances in computer architecture over the past half-decade, you've probably noticed that more and more cores are showing up on CPUs. The transistor budgets of CPUs are still increasing exponentially as Moore's Law predicts, but CPU designers are finding it increasingly difficult to translate the increased transistor budgets into increased single thread performance. Why the sudden change in paradigm? Under the hood, modern CPUs execute code in parallel. Examine the following source code:

```
i++;
j++;
```

This probably compiles down to two assembly instructions on the x86 architecture, assuming `i` and `j` live in EAX and EDX respectively:

```
inc EAX;
inc EDX;
```

Neither instruction depends on the result of the other. They can be executed in parallel, and this is exactly what the hardware does. This is called instruction level parallelism, or ILP. The problem with ILP is that it doesn't scale past a certain point. There's only so much parallelism available at such a low level. Even where ILP exists, the CPU needs to prove on the fly that the relevant instructions can be executed in parallel, and work around issues like branching and register sharing to make parallel execution happen. (For an excellent article that discusses these issues in a more detailed but still accessible way see [1].) What's a CPU designer to do when he/she can't find any more ways to translate an increased transistor budget into proportionally increased performance? Punt that task to the programmer, of course. The responsibility for exploiting parallelism in your code has shifted from the hardware and by extension the hardware designer to you, the programmer.

Notice that the word 'concurrency' hasn't been mentioned once yet, except in the title. Wikipedia defines concurrency as '…a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.' [2]. At the lowest levels we definitely have concurrency. Two increment operations are being executed simultaneously, yet somehow even assembly language programmers don't need to have the slightest clue about deadlocks, race conditions or any of the other standard concurrency concepts, even if they want to write highly optimized code that benefits maximally from ILP. Concurrency has been abstracted away as an implementation detail of parallelism. In its place is the higher level language of parallelism, which speaks in terms of dependencies. Two pieces of code that do not depend on each other's results, including side effects, may be executed in parallel. See [3] for an excellent and detailed discussion on why concurrency and parallelism are not the same thing.

There are other applications of concurrency where abstracting away its existence is not desirable, let alone practical. If you're writing a server, handling multiple requests simultaneously is a fundamental part of the problem you're solving, not an implementation detail. If you're writing a game, simultaneously processing graphics and sound is not an implementation detail. If you're writing an operating system, multitasking is not an implementation detail. There are two traditional ways to handle explicit concurrency: Multiple processes and multiple threads. Threads share an address space. This means that all of the memory in the process is implicitly shared between threads. Change a value from one thread and all of the other threads 'know' about it. This makes communication between concurrently executing tasks easy, but makes ensuring correctness of the code hard, since any memory address may be read or written to by any thread, in a nondeterministic order. Processes don't share an address space and all sharing is explicit. The downsides are that interprocess communication has substantial overhead and passing complex object graphs (i.e. the general case of objects that have pointers to other objects) between address spaces is a non-trivial problem.

Listing 1

```
import std.stdio, std.concurrency;
void fun() {
  writeln("Printing from a regular function.");
}

void main(string[] args) {
  void fun2() {
    writeln("Printing from a closure.");
  }

  auto t1 = spawn(&fun);   // Works.
  // Error.  Taking the address of a nested
  // function produces a delegate and allows
  // variables declared in the outer function to
  // be accessed from the nested function's body.
  // Therefore, the local variables of main()
  // would be accessible from multiple threads
  // if this were allowed.
  auto t2 = spawn(&fun2);
}
```

D's approach to the general case of concurrency can be summarized as isolation via the type system plus message passing and limited, explicit memory sharing. If you use `std.concurrency` for all your multithreading needs and don't use unsafe casts to subvert the type system, there can be no implicit sharing of mutable data between threads and no low-level data races. This is accomplished using several features of D's type system and the design of `std.concurrency`. First, `std.concurrency`'s `spawn()` function can only take a function pointer, not a delegate or a class instance. (For those not familiar with D, a delegate is a 'fat pointer' that holds a pointer to a function and a pointer to a context, such as a class or struct instance, or a closure.) See Listing 1.

Second, the arguments to functions started by `spawn` and messages passed between threads must not have unshared aliasing (using the standard library's terminology). This means that only data marked immutable or shared may be transitively reachable via pointers or references passed into a spawned function or passed as a message. (The immutable and shared type constructors are transitive, meaning all data reachable via pointer/reference indirection from immutable/shared data is also immutable/shared.) See Listing 2.

Third, all global or static variables not explicitly marked as shared or `__gshared` are implicitly thread-local. (`__gshared` variables are classic C-style global variables, are intentionally ugly looking, are not

DAVID SIMCHA

David Simcha is a Ph.D. student in Biomedical Engineering at Johns Hopkins University. His research interests include bioinformatics. This work has led him towards parallel computing as a secondary research interest. He can be reached at dsimcha@gmail.com

Listing 2

```
import std.stdio, std.concurrency;

void fun(string str) {
  writeln(str);
}

void fun2(char[] str) {
  writeln(str);
}

void fun3(int i) {
  writeln(i);
}

void main() {
  string str1 = "foo";
  // string is an alias for immutable(char)[]

  char[] str2 = "foo".dup;
  auto t1 = spawn(&fun, str1);
  // Works. Pointers to immutable data.

  auto t2 = spawn(&fun2, str2);
  // Error:  Pointers to mutable data.

  auto t3 = spawn(&fun3, 1);
  // Works. No pointer indirection at all.

  send(t1, str1);
  // Pass a message. Works.
  // Pointers to immutable data.

  send(t1, str2);
  // Error:  Pointers to mutable data.

  send(t1, 1);
  // Works. No pointer indirection at all.
}
```

allowed in code marked **@safe**, are easily greppable and are regarded as a similar to using an unsafe I-know-what-I'm-doing cast. **__gshared** is a storage class, not a type constructor, so **__gshared** variables have the same type as non-**__gshared** variables.) Immutable variables are implicitly shared, since there are no concurrency issues when sharing immutable data.

Fourth, the compiler guarantees sequential consistency of all access to data marked shared. Sequential consistency means that all reads and writes from a given thread happen in the order they were issued, as seen from all threads. The details of shared for classes and structs are too complicated to discuss here. The last chapter of Andrei Alexandrescu's book [4], discusses them in detail. Basically, shared classes and structs provide shared-memory concurrency that's limited, explicit (because the type must be marked shared and designed for thread safety) and safe from low-level data races, though it does not prevent concurrency-related bugs in high-level invariants.

Such a scheme provides the best of both worlds for concurrency use cases where the units of concurrency are not tightly coupled and complex state doesn't need to be passed frequently. You can do practical concurrent programming without the complexities of locks, atomic operations or memory models. You don't need to give up mutable state, which makes sense since mutable state isn't a problem for concurrency as long as it's private to a thread. Communication between threads is cheap and even complex object graphs can be easily, cheaply and safely shared as long as they're immutable. Finally, if the rules need to be broken occasionally, memory can be shared in a limited way that's guaranteed free from low-level data races.

Sometimes loose coupling between units of concurrency is an unaffordable luxury

Locks are still useful

I occasionally use locks with parallel foreach loops when I want low-level control over how a reduction is performed, especially with respect to memory usage. To me this is like using a **goto**: Usually more structured, higher level primitives are better but there are occasional oddball cases. Therefore, like **goto**, parallel **foreach** loops with locks to update shared data structures should be discouraged but not banned.

What's this got to do with parallelism? Sometimes loose coupling between units of concurrency is an unaffordable luxury. One such case is when implementing fine-grained parallelism. (For the purpose of this article fine-grained parallelism is defined as parallelism where communication between units of concurrency happens several times per second.) Unless the type system becomes so complex that computer science Ph.Ds can't wrap their heads around it or major sacrifices are made in efficiency or expressiveness, this probably can't be made statically checkable. Nonetheless, there are safer ways of exploiting parallelism than using threads directly, while fitting into D's current type system and preserving efficiency and expressiveness.

std.parallelism [5] is a module I've been prototyping for several years that was recently accepted into Phobos, the D standard library. To introduce some terminology before discussing this module in more detail, **std.parallelism** is based on the concept of a **Task**, which is the fundamental unit of work and may be executed in parallel with any other **Task**. A **TaskPool** encapsulates a task queue and worker threads, which execute the **Task** at the front of the task queue. The globally accessible instance of this class is called **taskPool**. A task queue is a FIFO queue where **Task**s are submitted for execution. A **Task** can be used explicitly for future/promise parallelism, and is used under the hood to implement higher level primitives.

The fundamental compromise **std.parallelism** makes is that it's up to you the programmer to understand how your program works and identify pieces of code that have no data dependencies and can be safely executed in parallel. In the interest of preserving efficiency, expressiveness and flexibility **std.parallelism** doesn't try to hold your hand here. However, once you've correctly identified parallelizable code, **std.parallelism** provides primitives that automatically handle the low-level concurrency issues related to parceling out the work to worker threads and getting the results back to the thread where they're needed. If you need to think about locks, race conditions, atomic operations, dining philosophers, sleeping barbers, or generally black magic of low-level concurrency, you're probably doing it wrong.

One way to think of **std.parallelism**'s model is that it uses a weaker version of the isolation principle of **std.concurrency**. If used idiomatically, every piece of data is owned by a single thread at any given instant or not mutable at that instant. No two threads may even attempt to update the same piece of data at the same time. If a piece of data were protected by a lock, this lock would never be contested. The twist is that this ownership may change during the life of the program. A memory fence is automatically inserted anytime data may change ownership. Unfortunately some discipline is required to write code in idiomatic **std.parallelism** form. The concept of all data being owned by a single thread at any given time is a high level invariant that is not statically checkable. **std.parallelism** provides the mechanisms to make this model easy to implement without using locks or atomic operations, or generally thinking about concurrency or low level memory model issues. For example, the following code computes the logarithm of every number from 1 to 1,000,000 in parallel and stores the results in a single array (Listing 3).

Listing 3

```
import std.math, std.parallelism;
void main() {
  auto logs = new double[1_000_000];
  // A parallel foreach loop is just like a
  // regular foreach loop, except its body may be
  // executed in parallel.
  foreach(i, ref num; taskPool.parallel(logs)) {
    num = log(i + 1.0);
  }
}
```

Let's look at this code from the perspective of our weak isolation model. Before the parallel **foreach** loop is entered, logs is owned by the program's main thread. The parallel **foreach** loop logic sends different loop iterations to different threads for execution. The loop body is written such that all iterations are independent. No two iterations write to the same piece of memory, and no iteration reads any piece of memory that another writes. If worker thread A performs iteration 0, then worker thread A 'owns' **logs[0]** while this iteration is occurring. Adjacent elements may be owned by different threads, though in practice the design minimizes false sharing [6]. Once the loop is complete, ownership of all elements of logs is transferred back to the main thread. The worker threads hold no reference to it and either terminate, sleep or execute unrelated work.

Let's take a look at another example, this time using the parallel reduction function **taskPool.reduce**. (Listing 4: thanks to Russel Winder for this example.)

iota(n) returns a range (basically a pair of iterators for those from a C++ background) that holds all numbers in **[0, n)**. **std.algorithm.map** lazily computes **getTerm()** for each element of **iota(n)**. The data it contains is sliced up and sent to multiple threads. Each temporarily owns a summation variable and computes its part of the sum. Then, ownership of all of these summation variables is transferred to the main thread and the final reduction of each thread's result is performed serially. (The fact that the terms of summation are computed lazily instead of stored is an implementation detail. Conceptually, the elements of the range are still parceled out to multiple threads.)

In the reduce example, **getTerm()** could write to a global variable or do several other nasty things. **std.parallelism** could require that the reduction function be pure, but I made the decision to favour flexibility over safety. First, a function may have unimportant side effects. It might allocate and free memory from a custom thread-safe allocator. It might generate random numbers from a thread-local random number generator instance. It might really be pure but not marked as such because the discipline required to recursively mark it and all functions it calls pure doesn't scale much better than any other form of discipline in programming.

Listing 4

```
import std.algorithm, std.parallelism, std.range;
void main() {
  immutable n = 1_000_000_000;
  immutable delta = 1.0 / n;

  // Calculate pi by quadrature. getTerm() gets
  // the individual terms in the summation and
  // is evaluated in parallel by taskPool.reduce,
  // by reading from the std.algorithm.map range.
  real getTerm(int i) {
    immutable x = ( i - 0.5 ) * delta;
    return delta / ( 1.0 + x * x ) ;
  }

  // The string "a + b" is a template parameter
  // and is a shorthand way of writing a simple
  // lambda function in D.  It's equivalent to a
  // function that takes two arguments named a
  // and b and returns a + b.
  immutable pi = 4.0 * taskPool.reduce!"a + b"(
    std.algorithm.map!getTerm(iota(n))
  );
}
```

Listing 5

```
import std.parallelism, std.file : write;
void main() {
  auto writeTask = task!write("foo.txt",
     "Writing from a task.");
  writeTask.executeInNewThread();
  write("bar.txt",
     "Writing from the main thread.");
  writeTask.yieldForce();
  // Wait for writeTask to finish.
}
```

Similarly in the case of future/promise parallelism, tasks may have side effects as long as there's no dependency. This example writes to two different files in parallel (Listing 5).

The result of these efforts is a compromise: a library that allows you to program in parallel, fully utilizing your new-fangled multicore hardware, if you understand parallelism but not concurrency. Concurrency only leaks out in that, if you try to parallelize something that can't be parallelized, bugs will manifest themselves as erratic, non-deterministic behaviour. An approach that favoured safety over efficiency and flexibility might try to determine what could be safely executed in parallel automatically, avoiding the requirement that the user understand parallelism. In the extreme it might do away with mutable state entirely, making this problem trivial. (This is what purely functional languages do.) Such approaches would always be conservative because proving lack of data dependency in the general case is equivalent to solving the halting problem and the compiler doesn't generally have access to the whole codebase. Furthermore, I find the concept of writing parallel programs that look almost the same as their serial counterparts and don't require a restrictive language design quite elegant. In the end, D is a systems language and with great power comes great responsibility. ■

References

[1] Modern Microprocessors: A 90-Minute Guide http://www.lighterra.com/papers/modernmicroprocessors/

[2] http://en.wikipedia.org/wiki/Concurrency_%28computer_science%29

[3] http://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency/

[4] *The D Programming Language* (1 ed.) Andrei Alexandrescu (2010), Addison-Wesley Professional

[5] http://digitalmars.com/d/2.0/phobos/std_parallelism.html, source at https://github.com/D-Programming-Language/phobos/blob/master/std/parallelism.d

[6] http://en.wikipedia.org/wiki/False_sharing

Code Patterns

Adam Petersen sees value in the visual shape of the code.

This year marks the 10th anniversary of my Test-Driven Development (TDD) career. It's sure been a bumpy ride. If a decade of experience taught me anything, it is that the design context resulting from TDD is far from trivial. TDD is a high-discipline methodology littered with pitfalls. In this article I'll look at the challenges involved in introducing and teaching TDD. I'm gonna investigate something we programmers rarely reflect over, the form and physical layout of our code, and illustrate how it may be used as a teaching-tool.

The human parser

Did you ever think about how fast a decent programmer assesses the quality of any given piece of code? In most cases it's a matter of seconds. But, in that short frame of time, what is it that we actually assess? What thought-process do we rely on? Do we parse the code at hypersonic speed in our mind while rapidly calculating the cyclomatic complexity as we go along and arrive at a formally well-founded decision? Probably not. And if we don't, why does it matter?

The good, the bad, the Java

Last year I held a couple of workshops with the aim of introducing unit tests and TDD in Java. Since I do believe TDD to be a design technique with a potentially huge payoff (at least for most statically typed OO languages), I always start with the benefits. After all, it's like I got something to sell. In an inspired moment of pedagogical high I even launched a Common Lisp REPL (Read-Eval-Print-Loop) to demonstrate the benefits of interactive development that TDD enables. Yet, as reality hit the fan, without any mentors aboard, the TDD experience of the team turned out to be far from the rosy development dream my presentation hinted at.

I've seen similar failures of applying TDD before. Not only does TDD require a tremendous amount of discipline. It also immediately highlights flaws and insufficient design in the code under test. That immediate feedback, while praised as the big-win of all lean technologies, at the same time, it poses part of the main problem with TDD. How come?

Well, a developer starting with TDD always has a history. It probably ranges from more or less successful projects, but somewhere along the line we've all managed to deliver some code that actually works (at least for some definition of 'works'). Put in other words: we all know how to program. As we start with TDD, the initial response is a lot of problems slowing development to a crawl. Complicated set-up code, tricky logic not easily expressed in a test, private data I cannot access from a test, heavy dependencies towards third-party products like databases and GUI, etc. Faced with all these obstacles, the testcases quickly degenerate and TDD is seen as quicksand upon which the team is trying to build a project. Pretty soon, the testcases are commented away or even shutoff in order to 'deliver'. More time passes and all that remains of TDD is a collective memory of a dysfunctional technique. A bridge is burned.

ADAM PETERSEN

Adam Petersen is a programmer and graduate student. His interests include Lisp, Erlang, parallel programming martial arts, music and modern history. He can be contacted at: adam@adampetersen.se

Mentor-driven design

Obviously we all recognize the problems above as symptoms of flawed designs caused by the neglect of solid design principles. After all, feedback in any form is good only if acted upon. Perhaps TDD is best seen as a messenger; if something's hard to test we don't have a testing problem but rather a design problem. Is a team ever going to succeed with TDD, the design skills of the programmers on the team have to be raised. And this is the actual point where I deem a mentorship absolutely crucial to success. I mean, the mechanics themselves behind a unit test are trivial: tag certain functions as testcases, express the observable behaviour of the system as assertions and press a button. If the design aspect was as simple as that, Fred Brooks wouldn't have written an entire book [1] about it and we would probably all be writing dry business code in some cobolesque language by now.

Guidance

Unit tests mirror the quality of the code they're testing. It's virtually impossible to separate unit testing and design. So what methods and techniques are useful to a mentor in guiding developers towards maintainable unit tests and thus a sound underlying design?

Some years ago, I had the opportunity to work with a group of mentors. Our goal was to develop a conceptual framework intended to make the transition to TDD as smooth as possible for the organization. A key part of the framework was a set of rules for unit tests. The rules were never an end in themselves. Rather, their purpose was to stimulate a discussion between developer and mentor. Each rule reflected one fundamental design principle concretized and augmented with specific examples from actual production code. We were deliberately oversimplifying. To give one example, the rules prohibited the use of explicit conditional logic a la if-else in the unit tests. The rationale was that an if-else-chain probably hints at low cohesion; the testcase probably covers multiple responsibilities and these should be separated. More often than not, it mixed normal flow of control with exceptional cases.

The rule set basically served as a learning tool and was never intended to live on. Come the day when we all understand why the rules are there and the organization has reached a point where it is actually okay to break the rules.

While we did have some success with the idea I continued to look for ways of further simplifying the rule set. There's something about enforcing rules upon my fellow programmers that I never quite liked. Besides, I had the idea there had to be some common, alternative way of capturing the rules.

Patterns

Let's return to the original question asked at the beginning of this article: no, we don't really parse code with hypersonic speed in our mind when determining its qualities. Our brain has a much more powerful tool: the visual system allows us to process a tremendous amount of information at literally a glance. What we actually do is comparing the physical, visual shape of the code against our experience. And even if we aren't consciously aware of it, years of coding has taught us how 'good' code looks in our beloved Emacs buffers.

Have a look at the following two code layouts. Both of them reflect the form of a small unit test suite. Which one of them would you prefer to maintain and extend?

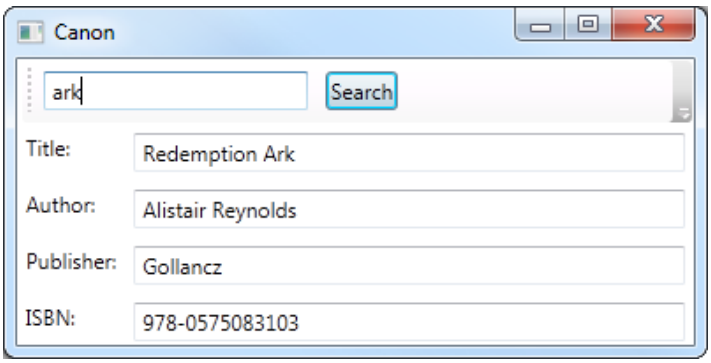
Have you choosen your favourite? Here's my take on it. Independent of programming language, the differences in complexity are quite striking

An Introduction to the Windows Presentation Foundation with the Model-View-ViewModel (Part 2)

Paul Grenyer wraps up the introduction to WPF.

In part 1 of 'An Introduction to the Windows Presentation Foundation with Model-View-ViewModel' [1] I introduced the Canon application, the source doe which can be downloaded from my website [2]. I used it to introduce you to simple WPF UI development and the MODEL-VIEW-VIEWMODEL [3] pattern including simple binding and commands. Figure 1 shows the GUI for Cannon (a successful search).

Figure 1



Part 2 is focused around making the GUI look more aesthetically pleasing and introducing menus and tool bars and demonstrating system commands. I'll start off by introducing images.

Images

Currently the Canon application uses the standard icon in its title bar. It doesn't really make Canon stand out from any other Windows application.

PAUL GRENYER

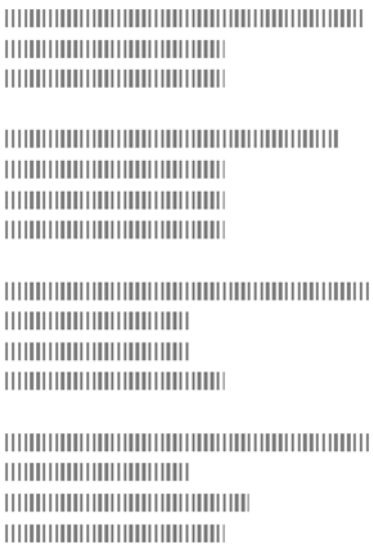
Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com

Code Patterns (continued)

and immediate. And that visual contrast serves as a tool for discussing design and highlighting basic principles. It's not limited to the educational aspect; I've found that the patterns work well during code reviews as an effective way of encouraging discussions about a given solution. If we take a high-level view, what's the shape of this very piece of code? Are there any parts of the design that diverge from the rest? Any signs of growing complexity? If so, why's that and is there a better way to attack the problem?

Obviously there are many valid reasons to design differently and the trade-offs may indeed motivate and result in radically different visual shapes than the ones above. The challenge is to make it an active decision rather than an ad-hoc solution. At the end, it's all about reflecting upon current practice, sharing knowledge within the team and continuously improve. And if you manage to get a team to actively discuss different design aspects on a daily basis, you're halfway there.

Figure 1



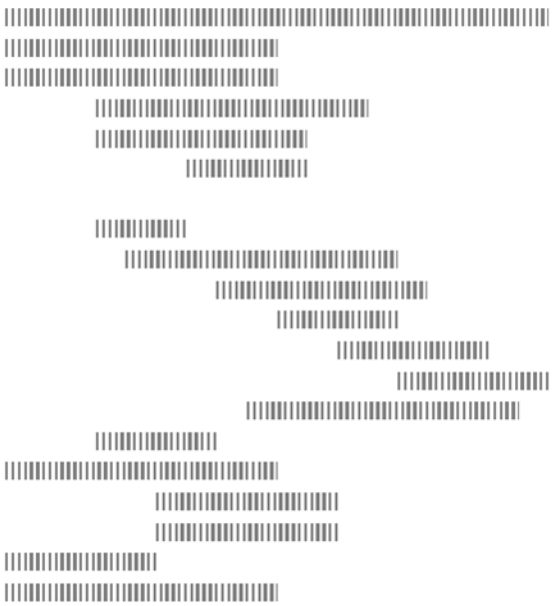
Summary

Test-Driven Development is a powerful yet unforgiving design technique. In order to address the design flaws that TDD inevitably highlights, it's important to recognise them as such and to act upon the feedback. As presented above, visual code patterns serve as a tool for discussing designs, perhaps by collecting a small library of patterns for different kinds of code and programming languages. A first step would be to print out the examples, tape them to the walls and let them compete for attention with the obligatory Dilbert strips. Discussing and comparing different code patterns together with other programmers and mentors provides an excellent learning opportunity. ■

References

[1] *The Design of Design* by Frederick P. Brooks, 2010, Addison-Wesley Professional

Figure 2



Listing 1
```
<Window x:Class="Canon.View.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="{Binding AppTitle}"
    MinHeight="230"
    Height="230"
    MinWidth="450"
    Width="450"
    FocusManager.FocusedElement="{Binding ElementName=searchBox}"
    Icon="/Canon;component/images/lightbulb.png">
```

If you look on your (Windows 7 at least) task bar you'll see that all the applications you have open have an icon. If they all had the standard icon it would be difficult to tell them apart.

I use free icon libraries, like Silk Icon Set [4], available on the internet for icons. I usual put images into an `Images` folder at the project level, so create one for the Canon project. Paste a suitable image (e.g. a 16x16 PNG) for the Canon icon into it and add the image to the project in the usual way. Make sure its Build Action property is set to **`Resource`**. Adding the image as an icon to the main window is done by setting the **`Icon`** attribute in the **`Window`** element (see Listing 1).

The format of the **`Icon`** attribute value is Microsoft Pack URI [5] and consists of the following tokens:

- **`/Canon`** The name of the resource file, including its path, relative to the root of the referenced assembly's project folder.
- **`;component`** Specifies that the assembly being referred to is referenced from the local assembly.
- **`/images/lightbuilb.png`** The relative path to the image file.

Listing 2
```
<DockPanel>
  <Menu DockPanel.Dock="Top">
    ...
  </Menu>
  <ToolBarTray DockPanel.Dock="Top">
    ..
  </ToolBarTray>
</DockPanel>
```

Listing 3
```
<Menu DockPanel.Dock="Top">
  <MenuItem Header="_File">
    ...
  </MenuItem>
</Menu>
```

Listing 4
```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">
      <MenuItem Header="_Save"
                Command="{Binding RunSave}"/>
    </MenuItem>
  </Menu>
```

Menus and tool bar icons

As it stands the Canon application is not very useful as it only allows us to search for the two preloaded books. What it needs to be able to do next is save updates to those books and create new ones. Save actions are often invoked by a menu and/or tool bar button or via a keyboard shortcut. Next I'll show you how to add a menu, with menu items bound to commands, which share an icon with a tool bar button we'll add to a new tool bar. First add a menu to the top section of the dock panel (Listing 2).

Menus are declared in their parent component with the **`Menu`** element. In the case of a **`DockPanel`** they also inherit the **`DockPanel.Dock`** attribute which is set to **`Top`** to put it in the same place as the tool bar tray. The order of child elements is important. If you put the menu below the tool bar tray the menu will appear below the tool bar tray. Add a drop down menu by adding a **`MenuItem`** with the **`Header`** attribute set (Listing 3).

The underscore in front of the **`F`** in **`File`** specifies that **`F`** is the short cut key for the File menu. To add an item to the drop down menu, add a child **`MenuItem`** element (Listing 4).

The **`Header`** attribute specifies the name of the item and the command binding is the same as a button command binding. We also need to add the command to the view model (Listing 5).

Listing 5
```
public class MainWindowViewModel : PropertyChangeEventBase
{
  ...
  public ICommand RunSave { get; private set; }
  ...
  public MainWindowViewModel(IBookRepository repo)
  {
    ...
    RunSave = new RelayCommand(o => Save(), o => canSave());
  }
  private bool canSave()
  {
    return true;
  }
  private void Save()
  {}
}
```

The **`canSave`** method just returns true for the time being. We'll put it to better use later. Menu items can also have images and the same image can be used for a tool bar button too. You could repeat the location of the image for both the menu item and the tool bar button, but a better solution is to add a resource (Listing 6).

Listing 6
```
<DockPanel>
  <DockPanel.Resources>
    <BitmapImage x:Key="SaveImage" UriSource="/Canon;component/images/disk.png" />
  </DockPanel.Resources>
</DockPanel>
```

This resource is added to the dock panel. Resources can be added to most components and are in scope within that component and all of its children. Before you add the **`DockPanel.Resources`** element, make sure you add a suitable image, called something like `disk.png`, to the images folder the name must match the name specified in **`UriSource`**. You can add all sorts of resources including the **`BitmapImage`** shown above. The **`x:Key`** attribute specifies the name that the resource will be referred to by when

Listing 7
```
<MenuItem Header="_Save" Command="{Binding RunSave}">
  <MenuItem.Icon>
    <Image Source="{StaticResource SaveImage}"/>
  </MenuItem.Icon>
</MenuItem>
```

Listing 8

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
    <StackPanel Orientation="Horizontal">
      ...
    </StackPanel>
  </ToolBar>
  <ToolBar>
    <Button Command="{Binding RunSave}">
      <Image Source="{StaticResource SaveImage}" />
    </Button>
  </ToolBar>
</ToolBarTray>
```

it's used by other components. The `UriSource` attribute is the path to the resource. It also uses Pack URI.

The `MenuItem.Icon` and `Image` child elements are required to add an image to a menu item (see Listing 7).

The image to use is specified by the `Source` attribute of the `Image` element which maps to the `x:Key` attribute of the resources. The image is bound to the resource, so uses curly braces. The resource is static as it is known at compile time, so uses the `StaticResource` keyword followed by the name of the resource. If you run the application now you will see the image next to the new menu item. The same image can be used as a tool bar icon. Add a new tool bar under the existing one. Add a button with a `Command` binding to the tool bar and an `Image` element that binds to the save image. (See Listing 8 and Figure 2.)

The save menu item and button do not currently save. The simplest way to save a book is to create a new `Book` instance, initialise it from the UI fields and pass it to the `Save` method of the repository (Listing 9).

Can you spot the flaw? The `Id` is not set, which means every time you save a new book instance will be created, even if it has exactly the same field values as an existing one. To get around this, we need to keep a reference to the loaded book (Listing 10).

To hold the reference we add a book field called `currentBook` to the `MainWindowViewModel`. We default initialise it in the constructor to make sure it is valid even if a book has not been loaded yet. Then if we find a book when we search for one we set the `currentBook` reference to the new book. Finally when we save the new book we use the `Id` from `currentBook` to create a new book instance. After a successful save we set `currentBook` to the new book instance. Try it out and see if you can spot the further flaw.

The only way to create a new book is to enter values into all the fields and save before searching for a book and even then you can only do it once. What we need is a new book menu item, image and tool bar button (Listing 11) and a new Command like `RunSave` and `RunSearch`. The difference with `RunNew` is that it does not need a `canNew` method as it is always permitted to create a new book:

```
RunNew = new RelayCommand(o => New());
```

You could create a `canNew` method hard coded to return `true` for consistency if you wanted too. The implementation of `New` looks like Listing 12.

The `Update` method is duplication of the code in the `Search` method, so the `Search` method can be refactored to remove the duplication (Listing 13).

If you run the application now you can create, save and search for new books.

System commands

WPF supports a range of system commands for operations including cutting, copying and pasting. This means you can add standard functionality without having to implement the details. For example you can add an edit menu (Listing 14).

You can of course add images and a corresponding tool bar in the way already described. Here we've replaced the command bindings with the

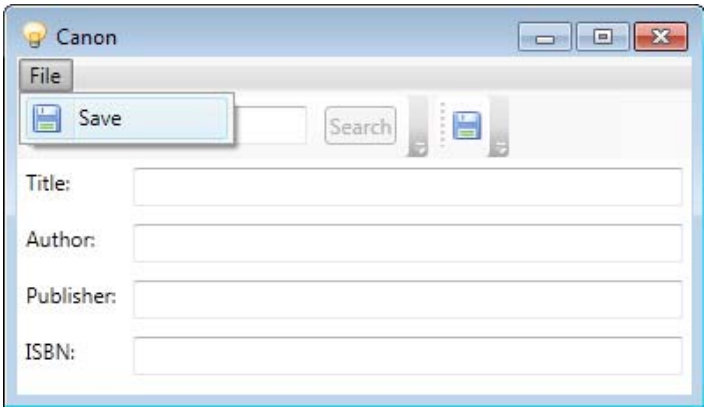


Figure 2

Listing 9

```
private void Save()
{
  repo.Save(new Book{Title = Title,
                     Author = Author,
                     Publisher = Publisher,
                     ISBN = ISBN});
}
```

Listing 10

```
public class MainWindowViewModel :
PropertyChangeEventBase
{
  ...
  private Book currentBook;
  ...
  public MainWindowViewModel(IBookRepository repo)
  {
    ...
    currentBook = new Book();
    ...
  }
  private void Search()
  {
    var book = repo.Search(SearchText);
    if (book != null)
    {
      currentBook = book;
      Title = book.Title;
      Author = book.Author;
      Publisher = book.Publisher;
      ISBN = book.ISBN;

      OnPropertyChanged("Title");
      OnPropertyChanged("Author");
      OnPropertyChanged("Publisher");
      OnPropertyChanged("ISBN");
    }
  }
  ...
  private void Save()
  {
    currentBook = repo.Save(new Book
    {
      Id = currentBook.Id,
      Title = Title,
      Author = Author,
      Publisher = Publisher,
      ISBN = ISBN
      });
  }
}
```

Listing 11

```
<DockPanel>
  <DockPanel.Resources>
    <BitmapImage x:Key="SaveImage" UriSource="/Canon;component/images/disk.png" />
    <BitmapImage x:Key="NewImage" UriSource="/Canon;component/images/add.png" />
  </DockPanel.Resources>
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">
      <MenuItem Header="_New" Command="{Binding RunNew}">
        <MenuItem.Icon>
          <Image Source="{StaticResource NewImage}"/>
        </MenuItem.Icon>
      </MenuItem>
      <MenuItem Header="_Save" Command="{Binding RunSave}">
        <MenuItem.Icon>
          <Image Source="{StaticResource SaveImage}"/>
        </MenuItem.Icon>
      </MenuItem>
    </MenuItem>
  </Menu>
  <ToolBarTray DockPanel.Dock="Top">
    ...
    <ToolBar>
      <Button Command="{Binding RunNew}">
        <Image Source="{StaticResource NewImage}" />
      </Button>
      <Button Command="{Binding RunSave}">
        <Image Source="{StaticResource SaveImage}" />
      </Button>
    </ToolBar>
  </ToolBarTray>
```

Listing 12

```
private void New()
{
  Update(new Book());
}
private void Update(Book book)
{
  currentBook = book;
  Title = book.Title;
  Author = book.Author;
  Publisher = book.Publisher;
  ISBN = book.ISBN;
  OnPropertyChanged("Title");
  OnPropertyChanged("Author");
  OnPropertyChanged("Publisher");
  OnPropertyChanged("ISBN");
}
```

Listing 13

```
private void Search()
{
  var book = repo.Search(SearchText);
  if (book != null)
  {
    Update(book);
  }
}
```

system commands for cut, copy and paste. If you run the application you'll find cut, copy and paste just work as expected. WPF In Action [6], the book in Introduced in part 1, goes into the system commands in more detail in Chapter 10, Commands.

Not all system commands are as straight forward. Unfortunately if you add the system **Close** command to the file menu:

```
<MenuItem Header="Close"
          Command="ApplicationCommands.Close" />
```

it is not enabled and does not close the application. What is missing is a command binding and handler methods (Listing 15).

The **CommandBinding** element uses its **Command** and **Executed** attributes to map the **Close** system command to the **CloseCommandHandler** handler, which is defined in the **MainMindow** class (Listing 16).

CloseCommandHandler just calls the **Close** method on the window to close it and consequently the application.

Detecting changes

Do you remember that earlier on we implemented a not particularly helpful binding for the Canon window title? Do you also remember the **canSave** method that always returns **true**? It would be far more useful for the user to only be able to save when there were changes to be saved and for the window title to indicate when there are changes to be saved (Listing 17).

IsDirty is a boolean property the indicates if any changes have been made. In the case of **AppTitle** it is used to determine whether an asterisk should be appended to the title when there are changes and in the case **canSave** it is just returned to indicate if the command should be enabled. (See Listing 18.)

The **IsDirty** property compares the current book fields against the equivalent UI fields to determine if there are any changes. Unfortunately this leads to some more verbose changes to the UI field properties to get the title and save command to update in real time (Listing 19).

I have only shown the changes for the **Title** property, but the **Author**, **Publisher** and **ISBN** properties must be changed in the same way. Instead of using the default property implementation we have to implement our own **set** method so that when the property is updated we can tell WPF to also update the window title. This means we also need to separately store the property value, which is initialised to an empty string to match the default **Book** instance, and implement a **get** method too. One advantage is that we can also move the WPF notification that the property has changed to the property itself so that we don't need to remember to call **OnPropertyChanged** anywhere else in the code where we assign the property. So the **Update** method is reduced to Listing 20.

The window title also needs to be updated when a book is saved as there are no longer any changes (Listing 21).

Finally

This is where this second article leaves the Canon application. There is more to do, but that falls outside the scope of an introductory article. In part 1 I already introduced you to simple WPF UI development and the Model-View-ViewModel pattern including simple binding and commands. In this part I explained how to make WPF GUIs more aesthetically pleasing with the use of images and more user friendly with the use of menus and toolbars and showed how to implement those menus and toolbars with custom and system commands.

Listing 14
```
<MenuItem Header="_Edit">
  <MenuItem Header="Cut" Command="ApplicationCommands.Cut" />
  <MenuItem Header="Copy" Command="ApplicationCommands.Copy" />
  <MenuItem Header="Paste" Command="ApplicationCommands.Paste" />
</MenuItem>
```

Listing 15
```
<Window>
  <Window.CommandBindings>
     <CommandBinding Command="ApplicationCommands.Close" Executed="CloseCommandHandler"/>
  </Window.CommandBindings>
    ...
</Window>
```

Listing 16
```
private void CloseCommandHandler(object sender, ExecutedRoutedEventArgs e)
  {
    Close();
  }
```

In future articles I will cover unit testing and patterns for maintaining the separation between the view model and the view when you want to display message boxes and child windows or use custom controls. ■

References

[1] 'In Introduction to the Windows Presentation Foundation with Model-View-ViewModel – Part 1': http://paulgrenyer.net/Introduction_to_WPF_with_MVVM_-_Part_1.pdf

[2] Canon 0.0.1 Source Code: http://paulgrenyer.net/dnld/Canon-0.0.1.zip

[3] 'WPF Apps With The Model-View-ViewModel Design Pattern' by Josh Smith. *MSDN Magazine*: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

[4] Silk Icon Set from Mark James: http://www.famfamfam.com/lab/icons/silk/

[5] Pack URIs in WPF: http://msdn.microsoft.com/en-us/library/aa970069.aspx

[6] *WPF In Action with Visual Studio* 2008 by Arlen Feldman and Maxx Daymon. Manning. ISBN: 978-1933988221

Listing 17
```
public string AppTitle
{
  get
  {
    return string.Format("Canon{0}",
       IsDirty ? " *" : "");
  }
}
...
private bool canSave()
{
  return IsDirty;
}
```

Listing 18
```
public bool IsDirty
{
  get
  {
    return
      !currentBook.Title.Equals(Title) ||
      !currentBook.Author.Equals(Author) ||
      !currentBook.Publisher.Equals(Publisher) ||
      !currentBook.ISBN.Equals(ISBN);
  }
}
```

Listing 19
```
private string title = string.Empty;
public string Title
{
  get
  {
    return title;
  }
  set
  {
    title = value;
    OnPropertyChanged("Title");
    OnChange();
  }
}
...
private void OnChange()
{
  OnPropertyChanged("AppTitle");
}
```

Listing 20
```
private void Update(Book book)
{
    currentBook = book;

    Title = book.Title;
    Author = book.Author;
    Publisher = book.Publisher;
    ISBN = book.ISBN;
}
```

Listing 21
```
private void Save()
{
  currentBook = repo.Save(new Book
  {
    Id = currentBook.Id,
    Title = Title,
    Author = Author,
    Publisher = Publisher,
    ISBN = ISBN
  });
  OnChange();
}
```

Code Critique Competition 71

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

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

Last issue's code

I've written a simple arithmetic expression evaluator – it works left to right but does supports brackets, a bit like old fashioned calculators. But when I try to test it I get this output:

```
cc70>test "1+2" "1/3 * 3"
"1+2" = 1.78744e-307
"1/3 * 3" = 1.78744e-307
```

Can you explain what's wrong (and also comment generally on the implementation)?

- The test program is in Listing 1
- The header (expr.h) is in Listing 2
- The implementation (expr.cpp) is in Listing 3.

Listing 1

```
#include "expr.h"

#include <iostream>
#include <sstream>

// evaluate and print supplied expression
int test(std::string const &s)
{
  try
  {
    expr e(std::istringstream(s).ignore(0));
    std::cout << "\"" << s << "\" = "
      << e.value() << std::endl;
    return 0;
  }
  catch (std::exception & ex)
  {
    std::cerr << "\"" << s << "\" failed: "
      << ex.what() << std::endl;
    return 1;
  }
}int main(int argc, char **argv)
{
  int ret(0);
  for (int idx = 1; idx != argc; ++idx)
  {
    ret += test(argv[idx]);
  }
  return ret;
}
```

Listing 2

```
#include <istream>
/* Left to right expression parser */
class expr
{
public:
  /* Parse a stream */
  expr(std::istream &is);
  /* Get the value */
  double const & value() const { return val; }
private:
  /* A term in the expression */
  class term
  {
  public:
    term(std::istream &is);
    term(double v);
    operator double() const { return val; }
    void operator+=(term const& rhs);
    void operator-=(term const& rhs);
    void operator*=(term const& rhs);
    void operator/=(term const& rhs);
    void operator%=(term const& rhs);
  private:
    double val;
  };
  term val;
};
```

Listing 3

```
#include "expr.h"
#include <functional>
#include <iostream>
#include <stdexcept>
#include <string>
expr::expr(std::istream & is) : val(is)
{
  char op;
  while (is >> op && op != ')')
  {
    if (op == '+') val += is;
    if (op == '-') val -= is;
    if (op == '*') val *= is;
    if (op == '/') val /= is;
    if (op == '%') val %= is;
  }
}
// Read a number or a bracketed sub-expression
expr::term::term(std::istream & is)
{
  char op;
  is >> op;
  if (op == '(')
    val = expr(is).val;
  else if (!(is.unget() >> val))
  {
    std::string error("Bad parse at: ");
    throw std::runtime_error(error + op);
  }
}
```


ROGER ORR

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

Listing 3 (cont'd)

```
// ctor from double
expr::term::term(double v) : val(v) {}
void expr::term::operator+=(term const& rhs)
{
  val += rhs;
}
void expr::term::operator-=(term const& rhs)
{
  val -= rhs;
}
void expr::term::operator*=(term const& rhs)
{
  val *= rhs;
}
void expr::term::operator/=(term const& rhs)
{
  val /= rhs;
}
void expr::term::operator%=(term const& rhs)
{
  val = std::modulus<long>()(val, rhs);
}
```

Critiques

Peter Sommerlad <peter.sommerlad@hsr.ch>

The first observation is that the numbers output seem to be very close to the smallest representable value greater than zero.

This can be checked through **#include**-ing **<limits>** and outputting **std::numeric_limits<double>::min()**, which seems to be "2.22507e-308" on my system.

However, on my Mac and the compiler I am using the output of the test program is 'correct' and the value given by the question is never given:

"1+2" = 3

"1/3 * 3" = 1

Now, compiling it gives a more interesting result:

```
In file included from ../expr.cpp:1:0:
../expr.h: In member function 'const double&
 expr::value() const':
../expr.h:12:40: warning: returning reference to
temporary [enabled by default]

In file included from ../main.cpp:1:0:
../expr.h: In member function 'const double&
expr::value() const':
../expr.h:12:40: warning: returning reference to
temporary [enabled by default]
```

that refers to the line

```
double const & value() const { return val; }
```

If **val** would be a member variable of type **double** the signature would be fine. However, it is of type **expr::term** and this type contains the underlying **double** but is not identical. What happens is that the operator **double()** of class **expr::term** is applied automatically by the compiler resulting in a temporary value of type **double**, whose reference is passed out to the caller of **expr::value()**. When **value()** returns the temporary's lifetime is over and the reference is invalid resulting in undefined behaviour. Fortunately for me, my compiler is aware of that problem and seems to cleverly handle that situation for me, without formatting my hard^h^h^h^hflash-disk.

Now, we can get to the main issue of this program:

It tries to rely on C++'s automatic conversion mechanisms too much.

My typical advice to my students is to declare all single-parameter (aka conversion) constructors as **explicit**. With the new C++ standard even constructors with more than one argument could and sometimes should be declared explicit, but explaining that is beyond the code critique.

The opposite direction of conversion operators can be a clever way to mix and match types without the need to overload operators in all mixed type applications, i.e., if I have type **T** and type **E** and both can be combined in expressions, let's say by **+** one would need to provide overloads of **operator+** with all combinations of **T** and **E** as input arguments, resulting in 4 overloads required. Providing automatic conversion from **T** to **E**, by either having a non-explicit conversion constructor of **E**, or a conversion **operator E()** on **T**, would allow to get away with just one overload of **operator+(E const&, E const&)**.

Since such conversion operators or constructors are applied automatically by the C++ compiler, if a single step conversion can make an expression or function call valid, they can be handy. On the other hand this is a double-edged sword, in that you can cut yourself easily. First, if there is more than one way to make your expression valid by applying different conversions your code becomes invalid again and you need to explicitly state what conversion you want to apply. That can become tricky and effortful, especially in the case, when you call template functions. Second, if your conversion involves elementary types, your class can behave strangely, when combined with such types indadvertedly, e.g., through a mis-spelled variable name.

Now, what can we fix about it. First, instead of the clever operator **double()** in class **expr::term** an accessor to the member variable **val** could be used, or the member variable be made public, it is in a private class anyway. But since I will give some discussion about that later, let us opt to change **operator double()** to an accessor instead:

```
double value() const { return val; }
```

OOPS, that gives us about half a dozen compile errors... Well those are exactly the places where the automatic conversion took place. So let us change those places as well:

- in `expr.h`:

  ```
  double value() const { return val.value(); }
  ```

- in `expr.cpp`:

  ```
  val = expr(is).val.value();
  ```

  aha this is another problem. Well we are in a member function of class term and try to access a private member of class **expr**. That is invalid and Linticator recognized that, but my compiler silently allowed it. But we corrected **expr::value()** before, so let us use that instead:

  ```
  val = expr(is).value();
  ```

all operators need adjustment too, like with **operator+** we need to call **rhs.value()** instead of relying on the automatic conversion.

```
val += rhs.value();
```

Now, things look fixed. However, it is bad practice today to write code without unit tests. It is also hard to test class **expr::term**, since that is a private member of class **expr**. As an example of a better tested expression evaluator, I'll attach one that I created in a TDD manner in an ACCU session led by Hubert Matthews some years ago. During that session an evaluator including variables was created including a very good test coverage in C++.

Before we go to that, there are other minor issues within the code that we can look at as well:

```
expr::expr(std::istream & is) : val(is)
{
  char op;
  while (is >> op && op != ')')
  {
    if (op == '+') val += is;
    if (op == '-') val -= is;
    if (op == '*') val *= is;
    if (op == '/') val /= is;
    if (op == '%') val %= is;
  }
}
```

in addition to not being **explicit** this constructor is a bit obfuscated and does more than a typical constructor does. It is not only initializing the left-hand term in the initializer list, but also continues to parse to the end or a syntax error on the right hand side. Bad practice is, that while **switch(op)** could have been used a sequence of **if** statements is used. This looks simple, but there is no guard against a syntax error here, so an operation character **op** that does not match any of the given 5 or the closing parentheses is silently ignored without telling the user. At least a final else, or a **switch(op)** with a **default** case should be used. In addition here again, the code relies on silently applying the non-explicit **ctor** of **expr::term(istream&)** which again can be confusing. So making that ctor **explicit** and giving an **else** case is good practice. Because the throwing of a 'bad parse' exception is already implemented, we extract that part out of the corresponding function of **term** into a static member of **expr** (static because it does not rely on any member variable).

```
expr::expr(std::istream & is) : val(is)
{
  char op;
  while (is >> op && op != ')')
  {
    if (op == '+') val += term(is);
    else if (op == '-') val -= term(is);
    else if (op == '*') val *= term(is);
    else if (op == '/') val /= term(is);
    else if (op == '%') val %= term(is);
    else throw_bad_parse(op);
  }
}
void expr::throw_bad_parse(char op)
{
  std::string error("Bad parse at: ");
  throw std::runtime_error(error + op);
}
```

This also results in adjustment of term's constructor like the following:

```
expr::term::term(std::istream & is)
{
  char op;
  is >> op;
  if (op == '(')
    val = expr(is).value();
  else if (!(is.unget() >> val))
  {
    expr::throw_bad_parse(op);
  }
}
```

While dense, the **!(is.unget() >> val)** is still a bit obfuscated. What it does, is resetting the current read pointer just before the read character op, since it is not a parenthesis it reads it again and parses the ongoing sequence as a **double** number. If that fails the underlying stream **is** is put into 'fail mode' and the **operator!** of the stream will return **true**, which results in throwing an exception.

With all those changes it becomes clear through Linticator that the second constructor of **term** is never used (Figure 1). Since it is private to **expr** anyway, we can just get rid of it.

A final remark on **expr**'s design is the use of a modulus operator on **double** values. Modulus is usually not defined on doubles and using the version of **long** is an interesting implementation option. Just consider the interesting aspect that 10*PI%PI is not zero in that case!

Now back to the main program:

```
expr e(std::istringstream(s).ignore(0));
```

is also interesting. It has two issues. The bigger one, also recognized by Lint is shown in Figure 2.

Figure 1

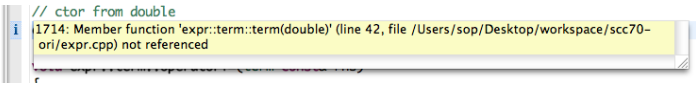


Figure 2

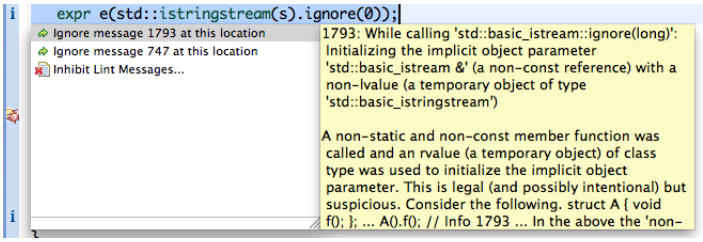
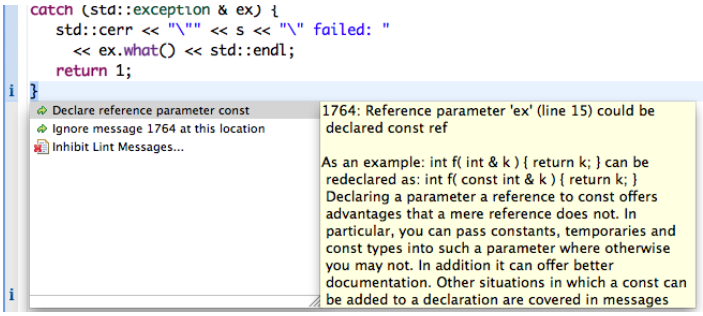


Figure 3



The code passes a temporary object by reference not only to the call of **ignore** which is a non-const member function of **istream** but also to the constructor of **expr**. This is bad practice and would be disastrous if **expr** would keep the **istream** reference as a member, because that would be dangling.

The second issue is calling **istream::ignore()** which is an 'unformatted input function' and can be used to skip over characters. However, ignoring 0 characters looks like a no-op. Under the hood some things happen, and if the stream object would throw an exception it might throw, but I believe it is superfluous, since extracting nothing means nothing. However, **ignore(0)** will check for a bad stream, but I can not think of an **istringstream** freshly constructed that would be bad, even if **s** is empty.

However, the reason for calling **ignore(0)** becomes more obvious, when we delete it from the code:

```
expr e(std::istringstream(s));
```

is not a definition of variable **e** being an **expr**, but a declaration of function **e** taking a **istringstream** as argument named **s** and returning an **expr**. This is a C++ parsing surprise, many programmers get trouble with, especially since the following code using **e** will result in surprising error messages further on. A rescue would be either using **'='** to initialize **e** or with the new C++ to use curly braces instead of the function parentheses:

```
expr e=std::istringstream(s);
// doesn't work, because of explicit ctor
```

or

```
expr e{std::istringstream(s)};
// C++11 aka C++0x, doesn't work, because of
// reference to temporary
```

However, as noted in the comments, both versions do not work, because now the temporary passed as constructor reference parameter becomes more obvious. The cure is not calling **.ignore(0)**, but to make a separate **istringstream** variable and use that. Then we can make sure that the object lifetime is no longer a potential problem.

Since the original programmer seems to be eager to save source code real estate we can eliminate a pair of parenthesis and make the function body a try-catch block instead. In addition we can follow Linticator's suggestion to make the reference to the exception object a const-reference, since we don't want to change the exception object anyway:

A further ugliness is the function name **test** that requires a comment to explain what it does, better we name the function accordingly and thus remove the need for the comment.

Applying rename and the shown quick fix results in the following code:

```
int evaluate_and_print(std::string
    const &expression)
try {
```

```
    std::istringstream is(expression);
    expr e(is);
    std::cout << "\"" << expression << "\" = "
              << e.value() << std::endl;
    return 0;
  } catch (std::exception const & ex) {
    std::cerr << "\"" << expression <<
      "\"failed: " << ex.what() << std::endl;
    return 1;
  }
```

There is not much to say about **main()**, except that it plays nicely by returning a non-zero number when detecting an error. However, since we do not know about how many arguments are given a too large return code might be generated that is wrapped around, depending on your operating system. The only thing the standard provides are two macros with the values **EXIT_SUCCESS** and **EXIT_FAILURE**, so the 'standard' way would be to write

```
  return ret?EXIT_FAILURE:EXIT_SUCCESS;
```

to leave **main()**.

Now for the teased, here is the code of the expression evaluator developed in a TDD way at ACCU 200x:

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"
#include <cctype>
#include <map>
class Parser {
  typedef int value_type;
  typedef std::vector<value_type> valuestack;
  typedef std::vector<char> opstack;
  typedef std::map<std::string,value_type>
    memory;
public:
  memory variables;
  private:
  void evaluateSingleOperator(
    char op, value_type &result,
    value_type operand) {
    switch(op) {
      case '+': result += operand; break;
      case '-': result -= operand; break;
      case '*': result *= operand; break;
      case '/': result /= operand; break;
      default: throw("invalid operand");
    }
  }
  void evaluateStacks(valuestack &values,
    opstack &ops) {
    while(ops.size() && values.size()>1) {
      char op = ops.back(); ops.pop_back();
      value_type operand = values.back();
      values.pop_back();
      evaluateSingleOperator(op,values.back(),
        operand);
    }
  }
  bool higherPrecedenceOrLeftAssociative(
    char last, char current) {
    return (last == current)||(last == '*'
      || last == '/');
  }
  bool shouldEvaluate(char op,
    opstack const &ops) {
    return ops.size() > 0 &&
      higherPrecedenceOrLeftAssociative(
        ops.back(),op);
  }
  std::string parseVariableName(
    std::istream &is) {
    std::string variable;
    char nextchar=0;
    while ((is >> nextchar) &&
      isalpha(nextchar)) {
    variable += nextchar;
  }
  if (variable.size() == 0) throw
    std::string("internal parse error");
  is.unget();
  return variable;
}
int peekWithSkipWhiteSpace(std::istream &is) {
  int nextchar = EOF;
  while(isspace(nextchar = is.peek()))
    is.get();
  return nextchar;
}
value_type getOperand(std::istream &is) {
  int nextchar = peekWithSkipWhiteSpace(is);
  if (nextchar == EOF) throw std::string
    ("syntax error operand expected");
  if (isdigit(nextchar)){
    value_type operand=0;
    if (!(is >> operand)) throw std::string
      ("syntax error getting number");
    return operand;
  } else if ('(' == nextchar) {
    is.get();
    return parse(is);
  } else if (isalpha(nextchar)) {
    std::string variable=
      parseVariableName(is);
    if( parseAssignmentOperator(is)) {
        variables[variable] = parse(is);
      } else {
        if (!variables.count(variable))
          throw std::string
          ("undefined variable: ")+variable;
      }
      return variables[variable];
  }
  throw std::string("syntax error");
}
bool parseAssignmentOperator(
  std::istream &is) {
  int nextchar = peekWithSkipWhiteSpace(is);
  if ('=' != nextchar) {
    return false;
  }
  is.get();
  return true;
}
public:
value_type parse(std::istream &is) {
  is >> std::skipws;
  valuestack values;
  opstack  ops;
  values.push_back(getOperand(is));
  char op=')';
  while((is  >>op) && op != ')') {
    if (shouldEvaluate(op, ops)) {
      evaluateStacks(values, ops);
    }
    values.push_back(getOperand(is));
    ops.push_back(op);
  }
  evaluateStacks(values,ops);
  return values.back();
}
value_type eval(std::string s) {
  std::istringstream is(s);
```

```
    return parse(is);
  }
};
int eval(std::string s) {
  return Parser().eval(s);
}
void shouldThrowEmptyExpression() {
  ASSERT_THROWS(eval(""),std::string);
}
void shouldThrowSyntaxError() {
  ASSERT_THROWS(eval("()"),std::string);
}
void testSimpleNumber() {
  ASSERT_EQUAL(5,eval("5"));
}
// ... other test definitions elided
void runSuite(){
  cute::suite s;
    s.push_back(CUTE(shouldThrowEmptyExpression));
  s.push_back(CUTE(shouldThrowSyntaxError));
  s.push_back(CUTE(testSimpleNumber));
  s.push_back(CUTE(testSimpleAdd));
  s.push_back(CUTE(testMultiAdd));
  s.push_back(CUTE(testSimpleSubtract));
  s.push_back(CUTE(testTenPlus12Minus100));
  s.push_back(CUTE(testMultiply));
  s.push_back(CUTE(testDivision));
  s.push_back(CUTE(testAddThenMultiply));
  s.push_back(CUTE(testAddSubSub));
  s.push_back(CUTE(testAddThenMultiplyAdd));
  s.push_back(CUTE(testSimpleParenthesis));
  s.push_back(CUTE(testSimpleOperandParenthesis));
  s.push_back(CUTE(testParenthesis));
  s.push_back(CUTE(testNestedParenthesis));
  s.push_back(CUTE(testDeeplyNestedParenthesis));
  s.push_back(CUTE(testSimpleAssignment));
  s.push_back(CUTE(testLongerVariables));
  s.push_back(CUTE(shouldThrowUndefined));
  cute::ide_listener lis;
  cute::makeRunner(lis)(s, "The Suite");
}
int main(){
  runSuite();
}
```

Commentary

Peter's critique demonstrates, among other things, some of the benefits of using tools to perform static checking of your code. In my experience such tools need 'training' to eliminate false positives but once this is done they can be added to the build process and they will catch a number of problems at compilation time.

The winner of CC 70

There was only one entrant, so Peter gets the prize. If you haven't done one before, why not try writing a critique for the problem below?

Code Critique 71

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

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

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

`circularList.h` is in Listing 4 and `circularListTest.cpp` is in Listing 5.

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.

Listing 4

```
#include <list>

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

Standards Report C++11

Roger Orr reports on the new C++ standard.

By the time you read this the ballot among national bodies (ANSI, BSI, DIN, AFNOR, etc) for the new C++ standard will have completed and we should have a new standard.

So what happens next?

C++ compilers will gradually become available with increasing coverage of all the C++11 features and there will be books about the new features (either updates to existing books, or brand new ones).

It will be interesting to see which of the new features will prove to be the most popular and useful ones. I suspect some of this will depend on the individual programmer – related to both their level of skill and which other computer languages they are familiar with.

How quickly this will affect you will also depend on the organisation for which you work and the client base you are supplying: some projects are, for example, able to track new releases of gcc whereas developers working in other environments may still be supporting older compilers and have no immediate plans to update.

Future committee work

The C++ standards committee continue to meet, with two meetings a year still scheduled. We are currently looking at the idea of hosting the Spring 2013 meeting in the UK the week after the ACCU conference: we have done this in the past and it resulted in additional members of the committee being available to present sessions at the conference. If you have any contact with possible sponsors for the week please get in touch with me!

Some of the future work will consist of fixing defects in the C++11 standard: work of this size and complexity is bound to contain a number of problems, many of which will not be uncovered until people start to use the new features in their own projects. In particular there may well be some 'integration bugs' where two or more new features interact with each other in confusing or ambiguous ways. A number of have already been found during the latter stages of preparing the C++11 standard but there are almost certainly more still to find.

However there is new material to work on too, for the next standard (tentatively called C++1y). There are a number of additions to the C++ standard library, such as filesystem support (based on that in boost), that were proposed too late for inclusion in the standard and also new proposals, such as various suggestions for a network library.

Further work in the language is also on the table: including extensions to the lambda syntax and a 'modules' proposal from Daveed Vandevoorde to support cleaner partitioning of C++ code.

There is also the thorny question of what to do with concepts, which was initially added to the C++0x standard and then removed when it became clear that there were enough unresolved issues to significantly delay the new standard. Should this be worked on for inclusion in the next standard?

Finally there are also some 'meta' questions to discuss:

- What is the best mechanism to deliver the next standard?
- Should we separate some of the library into its own standard?
- Do we need to formalise policy regarding inclusion of new features?
- Is the close relationship between C++ and C still relevant?

Initial discussion on these questions will have occurred at the August meeting in Bloomington, Indiana and the various national bodies, such as our own BSI panel, have also been discussing this.

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

Code Critique Competition 71 (continued)

Listing 4 (cont'd)

```
    iterator operator--(int)
    {
      iterator it(*this);
      --*this;
      return it;
    }
    iterator operator-(int n)
    {
      iterator result(*this);
      return result -= n;
    }
  };
};
```

Listing 5

```
#include "circularList.h"
#include <algorithm>
#include <iostream>
#include <string>
using std::string;
void test(circular<string> s)
{
  circular<string>::iterator it =
    std::find(s.begin(), s.end(), "Wed");
  std::cout << "Today is " <<
    *it << std::endl;
  circular<string>::iterator yest = it - 1;
  std::cout << "Yesterday was " <<
     *yest << std::endl;
  int const n = 5;
  it += n;
  std::cout << n << " days time will be "
    << *it << std::endl;
}
int main()
{
  circular<string> s;
  s.push_back("Sun");
  s.push_back("Mon");
  s.push_back("Tue");
  s.push_back("Wed");
  s.push_back("Thu");
  s.push_back("Fri");
  s.push_back("Sat");
  test(s);
}
```

Listing 5 (cont'd)

View From The Chair

Hubert Matthews
chair@accu.org

It's done; the long wait is over. After what seems an interminable time the new ISO C++ Standard (usually known as C++0x) has been approved. C++ may no longer be the hottest language in town – it is a mature technology – but it is still important for many companies, developers and ACCU members.

It is perhaps worthwhile taking some time to reflect on how the fundamental tools we use are developed and how their futures are controlled. Some languages are fleet-of-foot and change rapidly based on what their communities want, such as Ruby. Agility is one of the things that the Ruby community values along with expressiveness and so evolution there is rapid. Other languages are controlled primarily by vendors and change more slowly (Java and C# spring to mind). Yet others are controlled by large, rather slow international standards committees mostly staffed by volunteers (C++ and Fortran, for instance). C++0x is the product of a lot of work by a large number of people, some of them ACCU members and others who are ACCU friends. They deserve our thanks for the time and effort they have put in to this process to bring us new tools, ideas and techniques whilst protecting our investment in existing code. Languages that try to evolve without regard for the existing code base can back themselves into a tight corner. One merely has to look at Perl 6 to see what happens if you try to be too revolutionary with a popular entrenched language.

One of the things that tends to happen when a new version of a programming language comes out is that there is a flurry of interest in the new features and what you can do with them. After that comes a period when people start to realise the full implications of the new features and new idioms appear. I therefore fully expect there to be a resurgence in interest in C++ as people get to grips with key new features such as auto, lambdas, rvalue references and variadic templates. Just as the subject of exception safety caused a wholesale rethink of the way people programmed in C++, things like move semantics will lead to new ideas. I personally look forward to learning more about these and I hope to be able to share and pass that knowledge on to others. The ACCU has a long tradition of sharing information and a desire to learn is one of the key defining characteristics of our members.

Those ACCU members who don't program in C++ may be bemused or disinterested in the new standard and see it as irrelevant to them. However, a large proportion of the tools that they use – compilers, virtual machines, browsers, image manipulation programs, etc – are programmed in C++ so they too will benefit, if only indirectly. The new C++ standard represents an incremental change in the foundations of many of the things we rely on; it will take time to see just how much that affects and benefits all of our members.

Lack of space means we do not have room for any book reviews in this issue. They will be back in November, so keep sending them in.

Regional Meetings: ACCU London

Chris Oldwood reflects on a recent London event.

Once again we returned to the location where I first experienced an ACCU London talk – the offices of 7 City at Moorgate. The July meet-up saw us listening to Ed Sykes talk about Mocking in C++. As usual there was a good turnout with a few new faces to try and recruit in the bar opposite afterwards.

Ed started with a straw poll or two to try and gauge the audience and for once I fell into the minority as most of them were actively doing C++ and doing some form of modern unit testing. This allowed him to cut to the chase and get on with looking into the two mocking frameworks that he was using in his current position – MockItNow and Hippo Mocks. He acknowledged later that other frameworks had matured in the meantime, but that these two were the ones his team had focused on at the time they had started investigating the use of a 3rd party mocking framework for use with their C++ code-base.

First up was MockItNow which he clearly saw as a solution most suitable for those targeting the Microsoft compiler and trying to use mocking with a legacy code-base. C++ programmers have a natural tendency to want to look under the hood and fortunately he answered the obvious nagging question many of us immediately had, which was 'How does it work?' Once this minor distraction was resolved we could get on with just enjoying how the tests were expressed and what features it provided. His examples were simple enough to touch on the salient points, yet still managed to show it in action by actually running a set of tests with VS2010.

The second offering Ed explored was Hippo Mocks, which is a newer and more actively developed framework. Whereas MockItNow provides good support for legacy code-bases where concrete classes may be the norm, Hippo Mocks targets a more modern style of C++ code where the use of Interfaces is more prevalent to separate concerns. The same examples were carried over from the previous discussion to provide continuity, but with some minor changes required to allow the mocked type to be passed via the constructor. Whereas the MockItNow examples had a dash of macro magic the Hippo Mocks ones seemed a little more conventional.

It's a brave presenter who does live coding during a talk but Ed rose to the challenge to try and answer some of the questions from the audience by adapting his existing tests. Hippo Mocks showed its header-only, template-based nature nicely by spewing some unpleasant compiler errors as Ed tried to rework the tests but the speculative questions were never really going to be answered easily with the small amount of time left. Still, we did get to look inside this implementation too and Ed explained that his team had extended it relatively easily.

Those who have worked in other modern, reflection-capable languages like C# and Java will have been spoilt for years with mature mocking frameworks. Ed confidently showed us that C++ is finally catching up, even if some of the tools are platform specific or not quite as flexible as those available for its curly-brace cousins.