Semantic Programming

Ric Parkin

A gentle start

- A motivating example
- Lots of ways to make things better
  - Some will be familiar
  - Some may be new
  - Some may be ideas you’ve never tried
- A step back to look for unifying concepts
- Conclusions

A motivating example

```
void* memset( void* ptr, int fill,
              size_t count );

memset( buffer, buffersize, 0 );
```

You think it can't happen?

/* Code Search */

memset\(\ \w*,\ \w*,\ 0\ \) Search Advanced Code Search

Code Results **1** - **10** of about **50**. (**0.86** seconds)

systems/Core5.7/IM1Decoders/AAC/AACDecoder.cpp - 4 identical

```
129: #ifdef SILENCE_PLEASE
            ::memset( PUptr, m_PUlen, 0 );
            PUlen = m_PUlen;
```

```
136:   {
             ::memset( PUptr, m_PUlen, 0 );
             PUlen = m_PUlen;
```

standards.iso.org/.../c036089_ISO_IEC_14496-5_2001_Amd_1_2002_Reference_Software.zip - Unknown License - C++

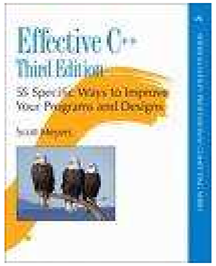
And my favourite

```
#define ZeroMemory(obj,size) memset(obj,size,0)
```

So what is the problem?

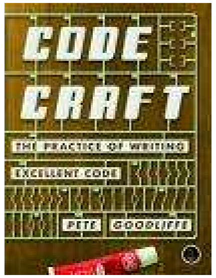
1) The programmer has written something they do not really mean
2) It looks okay so rereading might not spot it
3) The syntax is correct so compiler accepted
4) At runtime it *might* trigger, and if so we *might* spot the resulting bug, otherwise we'll release with it, and the user *might* find it

Inspiration



Scott Meyers Effective C++ Item 18

Make interfaces easy to use correctly and hard to use incorrectly.



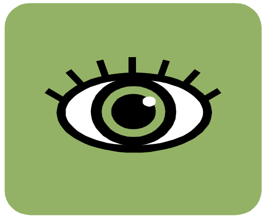
Pete Goodliffe: CodeCraft Chapter 6

If you can pull forward any tests to compile time, then do so. The sooner you detect and rectify an error, the less hassle it can cause.

What would help?

When you write

It should be easy to express what you want to happen



When you read

It should be clear what the code actually does

When you compile

The computer should catch mistakes

Aim to make it so that...

Correct code is

easy

clear

compiles

Incorrect code is

hard

ugly

an error

Aside: language applicability

- Key feature is the *compiler* reasons about the code before you run it and can reject it
- Key technique is to make new types
  - Encapsulation
- Mainly strongly typed languages
  - eg C, C++, C#, Java
- Things that are nice to have
  - Overloading on type
  - Operator overloading
  - Templates

Helping you get it right

- Education
- Add comments
- Documentation tools

  Doxygen, Docbook, JavaDoc etc

  Would be nice if your editor will show it, or hook into help

```
/// Fill an area of memory with a repeated byte
/// \param p Start of memory area to fill
/// \param c Fill byte (as int, converts to uchar)
/// \param i Size of memory area to fill
/// \return Returns passed in buffer
void* memset( void* p, int c, size_t i );
```

Name the variables/parameters

- What does it *represent*?

  Hungarian?

  - 'Good' v 'bad'
  - Typeless or dynamic languages

- Return value has no name

  Function name should describe the return value

  Procedure name should describe what it does

```
void* memset(
  void*  memoryAddress,
  int    fillByte,
  size_t memorySize );
```

Name the types

- Use typedef
  - Is an *alias*, not a new type
  - Is just another form of documentation
  - Takes part of the description from the variable name

```
typedef void*  Address;
typedef int    Byte;
typedef size_t Count;

Address memset(
  Address memoryStart,
  Byte    fill,
  Count   memoryLength);
```

Introduce a new type – by hand

- Really make a *new* type
  - Carefully control conversions
  - Is a place for helpers

```
struct Byte {
  explicit Byte( unsigned char value );
  operator unsigned char() const;
  private: unsigned char value;
};

Address memset(
  Address memoryStart,
  Byte    fill,
  Count   count);
```

Introduce a new type - automatic

- Strong/opaque typedef? No

  Was a proposal (wg21 N1706 N1891 N2141) but is stalled

- But we can write our own generators

```
struct ByteTag {};
typedef Integer<unsigned char, ByteTag> Byte;

struct MyStringTraits : char_traits<char> {};
typedef basic_string<char,MyStringTraits> MyString;
```

Constructor inheritance proposal (wg21 n2203)

Interaction with literals - constructors and operators

boost.operators makes it much easier to write lots of operators

Clouds, camels, weasels and whales

We want the new type to act like the old type, except not *quite*

Ham.	*Do you see yonder cloud that 's almost in shape of a camel?*
Pol.	*By the mass, and 't is like a camel, indeed.*
Ham.	*Methinks it is like a weasel.*
Pol.	*It is backed like a weasel.*
Ham.	*Or like a whale?*
Pol.	*Very like a whale.*




Find things that 'go together'

Buffer address and size form a natural "unit"

- Combine into a composite type
  - Compare 'Up Front' verses 'Just In Time' discovery
  - Good for security issues

```
struct Memory
{
  Memory(void* address, size_t size );
  void* address;
  size_t size;
};

Memory memset( Memory memory, Byte fill );
```

Bools and flags

```
DrawText( Text(“Hello World”), Point(0,0),
          true, true, false, 3 );
```

True/false is the answer but to what question?

```
void DrawText(
   Text  text,
   Point position,
   bool  alignLeft,
   bool  alignTop,
   bool  doKerning,
   int   fontTypes );
```

Enums for bools and flags

- Introduce an enum to make the caller clearer

```
enum HAlign { HLeft, HRight };
enum VAlign { VTop, VBottom };
enum KerningSetting { NoKerning, Kerning };
enum FontTypes { FontTypeVector = 1, FontTypeTrueType
= 2, FontTypeBitmap = 4 };

void DrawText( ..., HAlign hAlign, VAlign vAlign,
KerningSetting kerning, FontTypes fonts ... );

DrawText( ..., HRight, VTop, NoKerning,
FontTypeVector | FontTypeTrueType, ... );
```

Enums, flags, and bitfields

- One-Of

  Can only take one of the enumerators

- Set-Of, ie bitfield

  Make it easy to combine several of the enumerators

  eg Meyers ACCU 2006: When C++ meets the hardware

```
enum Bit { Bit1, Bit2};
struct BitField {
  Bit value;
  BitField( Bit bit );
  bool isSet( Bit bit ) const;
  BitField& operator |= ( BitField bitfield );
  ...
};
```

Strong enums

C/C++ enumerator names leak into outer scope

- Introduce a new scope

```
namespace HAlign {
  enum Enum {
    Left,
    Middle,
    Right
  };
}
void DrawText( ..., HAlign::Enum halign, ...);

DrawText( ..., HAlign::Left, ... );
```

Future: strong enum

WG21 n2213

Add useful interactions

Undesired conversions are an important example of type interactions. Built in operators are another

But int + int is syntactically okay, even if one represents a count and the other is a velocity!

- Add *selected* interactions between new types

```
Distance operator+( Distance, Distance );
Location operator+( Location, Distance );
Location operator+( Distance, Location );
Distance operator-( Location, Location );

Distance operator+( Distance, long ); //?
Distance operator+( long, Distance ); //?
```

Type systems - 1

- Geometry

```
struct X {};
struct DeltaX {};

X operator+( X, DeltaX );
X operator+( DeltaX, X );
DeltaX operator-( X, X );
```

- Unit conversions

```
// Use in Mars Climate Orbiter and save $125m
DistanceUS::operator DistanceMetric() const
{ return m_miles * 1.609344; }
```

Type systems - 2

- Physical quantities system

  http://sourceforge.net/projects/quan

```
Encodes physical units into type eg:
template <int length, int time, int mass>
class Quantity {…}
typedef Quantity<1,0,0>  Distance;
typedef Quantity<0,1,0>  Time;
typedef Quantity<0,0,1>  Mass;
typedef Quantity<1,-1,0> Speed;
typedef Quantity<2,-2,1> Energy;

Distance operator*( Velocity, Time );
```

Compiler does dimensional analysis!

Type system generation

- When need to instantiate multiple related types
  - Having each type as own template gets tricky so instantiate the whole system as one

```
template <typename Tag>
struct Axis {
  struct Coord {};
  struct Distance {};
  friend Coord operator+( Coord, Distance );
  friend Coord operator+( Distance, Coord );
};
struct XTag {}; struct YTag {};
typedef Axis<XTag> XAxis; typedef XAxis::Coord X;
typedef Axis<YTag> YAxis; typedef YAxis::Coord Y;
```

Not copyable or assignable

Many classes should never be copied or assigned

- Prevent it from compiling/linking

```
// By Hand
class File {
private:
  File( const File& );
  File& operator=( const File& );
};
// Helper
class File : noncopyable {};
```

Encapsulates a rule about usage

ReadOnly and ReadWrite

A very common usage rule – I will give you access to an object but you can *only* look

Makes calling code easier to reason about if can assume the callee does not change things

Three main techniques:

- Documentation
- Separate interfaces

```
struct IReadOnly { int getX(); };
struct IReadWrite : IReadOnly { void setX( int x ); };
```

- Const!

```
struct IReadWrite { int getX() const;  void setX( int x ); };
```

Locking

Multithreaded code must protect access

Locking/unlocking is an obvious use of RAII class

```
Lock lock( m_crit );
```

Locking objects is an obvious extension

```
Obj* obj = getSharedObject();
Lock lock( *obj ); // Obj is a lockable object
obj->serialisedAPI();
```

But you can forget to lock…

Is there some way to enforce?

Unavoidable locking

```
template <typename Value> struct ProtectedValue {
  Crit m_crit; Value m_value;
  struct ConstAccess : private Lock {
    ConstAccess(const ProtectedValue& pv )
    : Lock(pv.m_crit ), m_pv(pv) {}
    const Value& get() const { return m_pv; }
  };
  // and non-const Access
};
typedef ProtectedValue<int> ProtectedInt;
ProtectedInt protectedInt;
cout<< ProtectedInt::ConstAccess(protectedInt).get();
```

Object lifetime and ownership

```
Mail* MailServer::create();
bool MailServer::send( Mail* );
```

What lifetime guarantees are there?

How/when should client dispose of the object?

- Encapsulate in a new type (or generate one)

```
auto_ptr<Mail> MailServer::create();


auto_ptr<Mail> mail = mailServer.create();
...
mailServer.send( mail.get() );
```

auto_ptr – a defence

auto_ptr gets a bad press - 3 versions, bugs in some implementations

But it does some things very well

- Encapsulates a heap object's lifetime in a value type
- Makes the single ownership clear
- Can transfer that ownership clearly
- Automatic transfer and cleanup code

auto_ptr – Just one of many

auto_ptr has made a *lot of decisions*:

- *Single* owner
- Can *transfer* ownership
- *When* disposal occurs
- *How* to dispose

Alternatives:

- auto_array, scoped_ptr, shared_ptr
- Write own RAII wrappers by hand

Can add extra functions, eg COM wrappers

- RAII generators

My Handle generator, MC++D smart pointer framework

- GC....sort of – recycles memory, doesn't dispose of objects

Error codes

Can too easily forget/ignore

- Wrap in type that will assert or throw

```
template < typename E > struct Error {
  Error()  : error(), isRead (true){}
  Error( E error )  : error(error), read(false){}
  Error( const Error& e )  : error(e.error), read(e.read) { e.read = true; }
  ~Error() { if (!read) throw ErrorIgnored(error); }
  operator E() const { read = true; return error;}
  void operator=( const Error& e )   {
    if ( this != &e ) { // needed
      error=e.error; read =e.isRead; e.read=true;
    }
  }
  operator void() const {read = true; }
private:
  E error;  mutable bool read;
};
```

Reliable initialisation

Separate init functions ask for trouble

- Mistake to forget to call
- Mistake to call too many times
- Extra line to prepare
- Can’t use as temporary

Obvious solution is often recommended:

- Init function is the constructor

Enforce relationships

Relationships between instances are common

Parent/child

Owner/owned

Ways of enforcing relative lifetimes

- Parent creates and returns child

  Child needs to be copyable/transferable

  Child could out-live parent…

- Child constructor takes reference to parent

  Construction cascades

A look back

What have all these examples been showing?

Making new types to represent *concepts* present in the problem or solution domain gives some benefits:

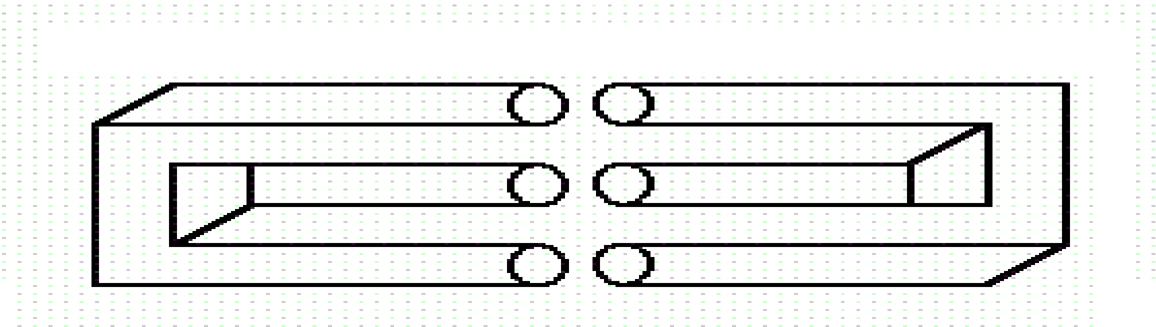
- Clarity of expression
- Better automatic checking
- Housekeeping code generation
- Closer match of the semantic and the syntactic vocabulary

When to apply…

- Big up front design – problem domain
- Agile discovery – solution domain
- If you *need* something to happen, stop and think – can you *ensure* it?
- Does the type describe its meaning?
- Does code using something look obvious?

Not a silver bullet – it *must* make things clearer

Different views



Writing libraries tends to make us concentrate on the implementation *inside* an interface

Using it sees the interface from *outside* - a very different view

Remember usability when designing interfaces

Consequences

Some interesting things happen when these ideas get used a lot

- Dull housekeeping code disappears
  - Temporary variables are often enough; no visible clear-up
  - Better exception safety tends to emerge
- Types describe *meaning*, not representation
  - Very few built-in types, usually internals. *What*, not *How*
- Important relationships become more visible
  - No longer gets lost in the clutter of *how* to do it
- Steps towards an aim become clearer

Consequences

The code takes on its own style and idioms, unique to that particular domain and application

In effect you have just written a **domain-specific language dialect** that is used to write your application.

New starters have to learn this language, but it's closer to the application concepts

Key point

Aim to encapsulate types, relationships and behaviour so that the desired *semantics* are expressed by the resulting *syntax*

Program Semantically

Questions?