Exploring Interfaces

A C++ Perspective

2006 Edition

Mark Radford

twoNine
Computer Services Ltd

mark.radford@twonine.co.uk
www.twonine.co.uk

Introduction

- This talk:
  - Is *not* a prescriptive treatise on class/function interface design
  - Aims to be an (interesting) discussion of the factors that influence interface design in modern C++

Topics

- Design by contract
- Exception safety
- Value based programming
- Templates
- Interface classes

Design by Contract

- Contents
  - Contract anatomy
  - Pre and Post conditions
  - Contract Specification in C++
  - Idioms
  - Const

Contract Anatomy

- Elements of contracts can be broken down as follows:
  - Static - applicable at compile time
  - Dynamic - applicable at run time
- The C++ standard library contains examples of contractual requirements with both static and dynamic aspects

Pre and Post Conditions

Pre condition:

The state in which a function call expects to find the program

Post condition:

The state the call promises to leave the program in following its return

```
template <typename type> class array
{
public:
  size_type size() const;
  type const& at(size_type index) const;

  type const& front() const
  { return at(0); }

  void pop_front();

...
};
```

Pre: `size()` > 0

Pre:
`size()` > 0;
size_prior = **`size()`**
Post:
size() == (*size_prior* – 1)

Contract Specification in C++

- Complete contracts can rarely be expressed using only C++ language features
- Therefore, other means of expression must be relied upon
  - External documentation
  - Idioms

An Example Idiom – *Whole Value*

```
enum tag_type { hour_tag, minute_tag, second_tag };

template <
  typename numeric_type, numeric_type first, numeric_type last, tag_type tag>
class numeric_range
{
public:
    explicit numeric_range(numeric_type n);
    // ...
};

typedef numeric_range<unsigned int, 0, 23, hour_tag>   hour;
typedef numeric_range<unsigned int, 0, 59, minute_tag> minute;
typedef numeric_range<unsigned int, 0, 59, second_tag> second;

class time_of_day
{
public:
    time_of_day(hour in_hour, minute in_minute, second in_second);
// ...
};

void f()
{
    time_of_day now(hour(14), minute(12), second(45));
//...
}
```

Compiler checks correct type use

Const

- Const-qualified reference/pointer parameters promise that arguments' state will not change
  - To an extent, the compiler can see that promises are kept
  - But the possibility of casting away **`const`** implies the need for trust
- Clients can only use **`const`** effectively with the support of the interfaces they make calls to

`const` & API Design

```
struct time { hour h; minute m; second s; };
```

```
void time_now(time& r);

void f()
{
  time t;
  time_now(t);
...
}
```

```
time time_now();

void f()
{
  const time t = time_now();
...
}
```

The opportunity for client code to store the result as a const object is…

…denied by a function that passes it back via a reference argument

…afforded by a function that passes it back as a return value

Logical vs. Physical `const`

```
class data_structure
{
public:
  typedef size_t size_type;

  size_type size() const
  {
    if (!size_valid)
    {
        its_size = calculate_size();
        size_valid = true;
    }

    return its_size;
  }
...
private:
  mutable size_type  its_size;
  mutable bool       size_valid;

  size_type calculate_size() const;
...
};
```

const **qualified member functions offer the promise that following a call to them, the user will perceive no change in the state of the objects.**

Attribute values visible to the client remain constant, but their representation values may change.

The `size()` member function may be viewed as *logically* – as opposed to *physically* – `const` **qualified.**

Exception Safety

- Contents
  - Exceptions
  - A sample problem and solution
  - Adapting to concurrency
  - Exception safety guarantees
  - The role of the nothrow guarantee

Exceptions

- Exceptions are woven into the infrastructure of C++
  - E.g. the new operator can't allocate memory it throws **`std::bad_alloc`**
- Compare with other modern C++ features such as templates
  - Which can be used or just left in the "toolbox"

A Sample Problem and Solution

```
template <typename T> class stack
{
public:
  T pop();

//...
};
```

```
void f(stack<my_type>& s)
{
  my_type mt;
  // ...
  mt = s.pop();
}
```

This interface design leaves the programmer with a problem and no solution...

If an exception is thrown during the assignment, the element on the top of the stack will be lost forever!

```
template <typename T> class stack
{
public:
  const T& top() const;
  void pop();
//...
};
```

```
void f(
    stack<my_type>& s)
{
  my_type mt;

  // ...

  mt = s.top();
  s.pop();
}
```

A solution is to ensure that if an exception is thrown, the pop() function is never called...

Therefore, separate the query and pop operations.

Adapting to Multi-Threading

```
template <
  typename type,
  typename locker
  >
class thread_safe_stack
{
public:
  pop(type& value)
  {
    locker lock;
    value = st.top();
    st.pop();
  }
...
private:
  std::stack<type> st;
};
```

Returning the value and popping it from the top of the stack must be done while other threads are locked out...

This requires both operations to be combined into one single function

Note that the result can not be returned by value

Exception Safety Guarantees

- The *basic* guarantee
  - If an operation throws an exception, no resources will leak as a result
- The *strong* guarantee
  - Additionally, the program's state remains unchanged
- The *nothrow* guarantee
  - An operation guarantees never to propagate an exception under any circumstances

The Role of the *Nothrow* Guarantee

Example Problem: *Inserting elements into a vector, honouring the strong exception safety guarantee.*

```
template <typename type> void f_unsafe(std::vector<type>& v)
{
  // ...
  v.insert(v.end(), first, last);

  // ...
}
```

If an exception is thrown during `insert()`, `v` is left in an indeterminate state.

Solution: *Create a working copy, insert the new elements into it, then swap the working copy with the original.*

```
template <typename type> void f_safe(std::vector<type>& v)
{
  std::vector<type> v_temp(v);
  v_temp.insert(v.end(), first, last);

  v.swap(v_temp);

  // ...
}
```

`vector`'s `swap()` member is carries a *nothrow* guarantee – therefore, having constructed the new state, it can be safely swapped into the original.

Templates

- Contents
  - Generic programming
  - A simple string class template…
  - Traits

Generic Programming

- A highly flexible mechanism for expressing the compile-time commonality and variability needed for Generic Programming
  - Well illustrated by the STL, where types, data structures, operations and control flow are independently interchangeable

A Simple String Class

```
template<typename charT>
class rudimentary_string
{
public:
  typedef charT   char_type;
  typedef size_t  size_type;

  size_type size() const;

  void copy(
    rudimentary_string<char_type> const& s)
  {
    // resize buffer to fit, then...
    std::copy(
        s.buffer, s.buffer+size(), buffer);
  }
...
private:
  char_type *buffer;
...
};
```

The character type charT can be any type - subject to the provision of certain operations. For example, it may need to be *copyable*.

Using `std::copy()` is convenient but leaves problems to be solved - for example, if a particular platforms provides a high performance byte copying function, how are we to use it? ...

Traits

```
template <typename charT> struct char_traits
{
  typedef charT  char_type;
  typedef size_t size_type;

  static char_type* copy(char_type* destination,
    char_type const* source, size_type n_to_copy)
  {
    char_type const* begin = source;
    char_type const* end   = source + n_to_copy;

    return std::copy(begin, end, destination);
  }
};
```

Delegating the copy operation to a separate class template makes it possible to provide an explicit specialisation for *char*.

```
template <> struct char_traits<char>
{ ...
  static char_type* copy(
    char_type *dest, char_type const *source, size_type n_to_copy)
  {
    void* result = std::memmove(dest, source, n_to_copy);

    return static_cast<char_type*>(result);
  }
};
```

Traits At Work

```
template<typename charT,
typename traitsT=char_traits<charT> >
class rudimentary_string
{
public:
  typedef traitsT traits;

  size_type size() const;

  void copy(
    const rudimentary_string<char_type>& s)
  {
    // resize buffer to fit, then...
    traits::copy(buffer, s.buffer, s.size());
  }
...
private:
...
};
```

Users can select the desired *traits*, while the use of default template arguments means they don't *have* to

The copy() member functions now delegates to the *traits*' copy() function – this is transparent to the user

Value Based Programming

- Contents
  - Value based types
  - C++ value type interfaces
  - Swap semantics
  - Whole values
  - Quantities

Value Based Types

- Characterised by ...
  - Dominant informational content (state) and transparent identity
- Take the form of concrete classes in C++, therefore…
  - They do not inherit publicly from other types
  - Direct copy semantics make sense

C++ Value Type Interfaces

- General characteristics related (dominant state and transparent identity)
  – Accessors, mutators and Constructors (for conversion, and for initialising state)
- C++ language representation related
  – Copy constructor, Copy assignment

Swap Semantics

- Necessary to support common idioms which support the strong exception safety guarantee
  - Therefore, the swap function must carry a *nothrow* guarantee
- May be provided either ...
  - As a member function
  - As a freestanding function
  - By relying on `std::swap()`

Whole Values

```
class minutes
{
public:
  explicit minutes(int initial);
  ...
};
```

```
class transaction
{
public:
  void timeout(const minutes& duration);
  ...
};
```

```
void f(transaction* current)
{
  current->timeout(minutes(10));
  ...
}
```

- In C++ classes should be used to implement first class data abstractions
  - The compiler is empowered to do stronger type checking
  - The code communicates application domain vocabulary

Quantities

- E.g. minutes, voltages, kilograms
  - Quantities account for much/most of the values in programmes
- They have their own additional specific characteristics, e.g…
  - They have *units*
  - A value of zero/empty makes sense

Quantity Abstraction Pitfall

```
class time_interval
{
public:
  struct from_seconds {};
  struct from_minutes {};

  time_interval(
    int value, const from_sec&);
  time_interval(
    int value, const from_min&);
  ...
};

std::ostream& operator<<(
  std::ostream& os,
  const time_interval& value);
```

Not only does initialisation require a very artificial looking interface design, but …

The need to please everyone threatens to bloat the interface out of control…

And what exactly <u>is</u> inserted into the stream!?

Abstract Quantities as Units

```
class minutes
{
public:
  explicit minutes(int initial);
  minutes();
  ...
};


std::ostream& operator<<(
  std::ostream& os,
  const minutes& value);
```

Initialisation is now straightforward – either by default or from a single value

Operator overloads come naturally, and their meaning is automatically clear

Initialisation & Conversion

```
class minutes
{
public:
  typedef int value_type;

  minutes();

  explicit
    minutes(value_type initial_value);

  value_type value() const;
    ...
};
```

Default initialisation value of zero is a feature of quantity types

Conversion *must* be explicit to convey meaning

Conversion back to underlying type must possible and must be invoked *deliberately*

Note: the provision of* `value()` *is necessary for the range of supported operations to be extensible – e.g. for the provision of conversion functions

Interface Classes

- Contents
  - Interface class
  - Implementation only class
  - Mixins

Interface Class

```
class shape
{
public:
  virtual ~shape();

  virtual void move_x(xinterval x) = 0;
  virtual void move_y(yinterval y) = 0;
  virtual void rotate(radians r)   = 0;

//...
private:
  shape& operator=(const shape&);
};
```

The *interface class* defines a protocol for usage – all functions are pure virtual and not implemented (except for the destructor which will have an empty implementation).

Null Object

```
class group
{
public:
  virtual unsigned int num_in_group() const = 0;
...
};
```

The Null Object provides an implementation that behaves as if the object is absent

```
class null_group : public group
{
public:
  virtual unsigned int num_in_group() const
    { return 0; }
...
};
```

```
class drawing
{
public:
  drawing() : selected(zero_selected),
  ...
  unsigned int num_selected() const
  {
    return selected->num_in_group();
  }
private:
  group* selected;
  static null_group* zero_selected;
};
```

Client code no longer needs to be concerned with checking for nullness – calls to the *null object* do the right thing

Mock Object

Assume a `drawing` object has been created and loaded with five shapes

```
class repository
{
public:
  virtual void save(const shape* s) = 0;
...
};
```

```
class drawing
{
public:
  void save(repository& r) const;
...
};
```

```
class counting_repository :
  public repository
{
public:
  counting_repository() : count(0) {}
  virtual void save(const shape* s)
  { ++count; }
  unsigned int num_saved() const
  { return count; }
...
private:
  unsigned int count;
};
```

`counting_repository` is a simple example of a *mock object*

```
void unit_test(const drawing& d)
{
  counting_repository counter;
  d.save(counter);

  assert(counter.num_saved()
        == 5);
}
```

In this simple illustration, a *mock* implementation of `repository` is used to test that the correct number of shapes are saved

Implementation Only Class

```
class line : public shape
{
public:
  line(point end_point_1, point end_point_2);
//...
private:
  virtual ~line();
  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);

//...
};
```

The separation of concerns afforded by the interface class can be strengthened by making derived classes *implementation only* – constructors are public, everything else is private.

Making the derived class *implementation only* allows client code to create instances, while calls are permitted only through pointer/reference to the *interface class*

Mixins

```
class storable
{
public:
  virtual void load(istream& in) = 0;
  virtual void save(ostream& out) = 0;
protected:
  ~ storable();
private:
  storable& operator=(const storable &);
};
```

```
class shape : public storable
{
public:
  virtual ~shape();
...
// No declarations of load() or save()
// in this class
};
```

Sometimes a class must support functionality outside its mainstream design remit

(Note the protected non-virtual destructor – mixins are not deletion types)

Using a mixin interface class keeps these extra concerns separate and reduces the level of intrusion

Multiply-Inherited Mixin

```
struct coordinate { unsigned int x,y; };

class coordinated
{public:
  virtual coordinate where() const = 0;
...
};
```

```
class event
{public:
  virtual ~event() = 0;
...
};
```

```
class timer :
  public event
{
  ...
};
```

```
class mouse_click : public event,
                    public coordinated
{
...
private:
  virtual coordinate where() const
  { return c; }

  coordinate c;
};
```

When functionality added by a mixin is not relevant to the whole hierarchy, the mixin can not be the base class

In such cases multiple inheritance can be used to include the mixin interface

Do You Support This Interface?

```
class event
{public:
  virtual ~event() = 0;
...
};
```

```
class mouse_click :
  public event, public coordinated
{
...
private:
  virtual coordinate where() const
  { return c; }
  coordinate c;
};
```

```
void process(event* e)
{ const coordinated* c =
    dynamic_cast<coordinated*>(e);
  if (c)
  {
    const coordinate where = c->where();
    //...
  }
}
```

By using dynamic_cast<> to navigate across a hierarchy, it is possible to ask an object if it supports a particular mixin interface

There is an analogy here with Smalltalk – in which an object can be sent a message, and will either act, or reply saying it does not understand the message

Private Mixin

```
class notification_client
{
public:
    virtual void update() = 0;

protected:
    ~notification_client();
};
```

```
class notifier
{
public:
  virtual void register_client(
             notification_client* c)=0;
    ...
};
```

The notifier only needs access only to the behaviour expressed by the notification_client interface

```
class my_window : public window, private notification_client
{
public:
    void register_for_notifications(notifier& n)
    { n.register_client(this); }
    ...
};
```

private inheritance renders this interface (normally) inaccessible

Within a member function the conversion is accessible

Private Operation Interface

```
template <typename persistent>
class persistent_ptr
{
  database_query<persistent> >* query;
  persistent* object;
public:
  persistent* operator->() const
  { if (!object)
      object = query->execute():
    return object;
  }
 ...
};
```

```
template <typename persistent>
class database_query
{
 friend class persistent_ptr<persistent>;
 virtual persistent* execute() const = 0;
 ...
};
```

There is no support provided by C++ for *package* level access

A degree of specific access can be obtained by using *private* class members in conjunction with *friend*

Summary

- C++ offers a rich toolset supporting several approaches to design/programming
  - E.g. Value based, object oriented and generic programming are all supported
- With the richness of the toolset comes a wide range of choices
  - This places responsibility for managing complexity firmly with the designer