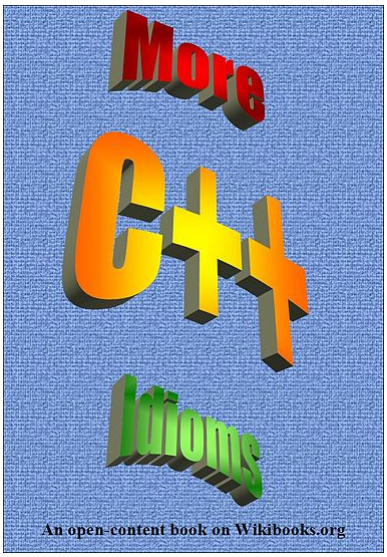
More C++ Idioms/Print Version

Preface



C++ has indeed become too "expert friendly" -- Bjarne Stroustrup, *The Problem with Programming* (http://www.technologyreview.com/Infotech/17831), Technology Review, Nov 2006.

Stroustrup's saying is true because experts are intimately familiar with the idioms in the language. With the increase in the idioms a programmer understands, the language becomes friendlier to him or her. The objective of this open content book is to present modern C++ idioms to programmers who have moderate level of familiarity with C++, and help elevate their knowledge so that C++ feels much friendlier to them. It is designed to be an exhaustive catalog of reusable idioms that expert C++ programmers often use while programming or designing using C++. This is an effort to capture their techniques and vocabulary into a single work. This book describes the idioms in a regular format: *Name-Intent-Motivation-Solution-References*, which is succinct and helps speed learning. By their nature, idioms tend to have appeared in the C++ community and in published work many times. An effort has been made to refer to the original source(s) where possible; if you find a reference incomplete or incorrect, please feel free to suggest or make improvements.

The world is invited to catalog reusable pieces of C++ knowledge (similar to the book on design patterns by GoF). The goal here is to first build an exhaustive catalog of modern C++ idioms and later evolve it into an idiom language, just like a pattern language. Finally, the contents of this book can be redistributed under the terms of the GNU Free Documentation License.

Aimed toward: Anyone with an intermediate level of knowledge in C++ and supported language paradigms

Authors

- Sumant Tambe [talk] -- The initiator and lead contributor since July 2007. See my contributions (http://en.wikibooks.org/wiki/Special:Contributions/Sutambe).
- Many other C++ aficionados who continuously improve the writeup, examples, and references where necessary.

Praise of the Book

- **"Great and valuable work!"** -- Bjarne Stroustrup (February, 2009)

Table of Contents

Note: synonyms for each idiom are listed in parentheses.

59. Object Template TODO
60. Parameterized Base Class (Parameterized Inheritance)
61. Pimpl (Handle Body, Compilation Firewall, Cheshire Cat)
62. Policy Clone (Metafunction wrapper)
63. Policy-based Design TODO
64. Polymorphic Exception
65. Polymorphic Value Types TODO
66. Recursive Type Composition TODO
67. Requiring or Prohibiting Heap-based Objects
68. Resource Acquisition Is Initialization (RAII, Execute-Around Object, Scoped Locking)
69. Resource Return
70. Return Type Resolver
71. Runtime Static Initialization Order Idioms
72. Safe bool
73. Scope Guard
74. Substitution Failure Is Not An Error (SFINAE)
75. Shortening Long Template Names TODO
76. Shrink-to-fit
77. Small Object Optimization TODO
78. Smart Pointer
79. Storage Class Tracker TODO
80. Tag Dispatching TODO
81. Temporary Base Class
82. Temporary Proxy
83. The result_of technique TODO
84. Thin Template
85. Traits TODO
86. Type Erasure
87. Type Generator (Templated Typedef)
88. Type Safe Enum
89. Type Selection
90. Virtual Constructor
91. Virtual Friend Function

Advanced idioms

These are some more advanced C++ idioms.

1. Envelope Letter TODO

Deprecated idioms

1. ~~Const auto_ptr~~

More C++ Idioms/Adapter Template

Address Of

Intent

Find address of an object of a class that has an overloaded unary ampersand (&) operator.

Also Known As

Motivation

C++ allows overloading of unary ampersand (&) operator for class types. The return type of such an operator need not be the actual address of the object. Intentions of such a class are highly debatable but the language allows it nevertheless. Address-of idiom is a way to find the real address of an object irrespective of the overloaded unary ampersand operator and its access protection.

In the example below, the `main` function fails to compile because `operator &` of `nonaddressable` class is private. Even if it were accessible, a conversion from its return type `double` to a pointer would not have been possible or meaningful.

```
class nonaddressable
{
public:
    typedef double useless_type;
private:
    useless_type operator&() const;
};

int main()
{
  nonaddressable na;
  nonaddressable * naptr = &na; // Compiler error here.
}
```

Solution and Sample Code

The Address-of idiom retrieves the address of an object using a series of casts.

```
template <class T>
T * addressof(T & v)
{
  return reinterpret_cast<T *>(& const_cast<char&>(reinterpret_cast<const volatile char &>(v)));
}
int main()
{
  nonaddressable na;
  nonaddressable * naptr = addressof(na); // No more compiler error.
}
```

C++11

In C++11 the template std::addressof (http://en.cppreference.com/w/cpp/memory/addressof), in the <memory> header, was added to solve this problem. In C++17, the template is also `constexpr`.

Known Uses

- Boost addressof utility (http://www.boost.org/doc/libs/1_47_0/libs/utility/utility.htm#addressof)

Related Idioms

References

Algebraic Hierarchy

Intent

To hide multiple closely related algebraic abstractions (numbers) behind a single generic abstraction and provide a generic interface to it.

Also Known As

- **State** (Gamma et al.)

Motivation

In pure object-oriented languages like Smalltalk, variables are run-time bindings to objects that act like labels. Binding a variable to an object is like sticking a label on it. Assignment in these languages is analogous to peeling a label off one object and putting it on another. On the other hand, in C and C++, variables are synonyms for addresses or offsets instead of being labels for objects. Assignment does not mean re-labelling, it means overwriting old contents with new one. Algebraic Hierarchy idiom uses delegated polymorphism to simulate weak variable to object binding in C++. Algebraic Hierarchy uses Envelope Letter idiom in its implementation. The motivation behind this idiom is to be able write code like the one below.

```
Number n1 = Complex (1, 2); // Label n1 for a complex number
Number n2 = Real (10); // Label n2 for a real number
Number n3 = n1 + n2; // Result of addition is labelled n3
Number n2 = n3; // Re-labelling
```

Solution and Sample Code

Complete code showing implementation of Algebraic Hierarchy idiom is shown below.

```
#include <iostream>

struct BaseConstructor { BaseConstructor(int=0) {} };

class RealNumber;
class Complex;
class Number;

class Number
{
    friend class RealNumber;
    friend class Complex;

  public:
    Number & operator = (const Number &n);
    Number (const Number &n);
    virtual ~Number();

    virtual Number operator + (Number const &n) const;
    void swap (Number &n) throw ();

    static Number makeReal (double r);
    static Number makeComplex (double rpart, double ipart);

  protected:
    Number ();
    Number (BaseConstructor);

  private:
    void redefine (Number *n);
    virtual Number complexAdd (Complex const &n) const;
    virtual Number realAdd (RealNumber const &n) const;

    Number *rep;
    short referenceCount;
};

class Complex : public Number
{
  friend class RealNumber;
  friend class Number;
```

```
  Complex (double d, double e);
  Complex (const Complex &c);
  virtual ~Complex ();

  virtual Number operator + (Number const &n) const;
  virtual Number realAdd (RealNumber const &n) const;
  virtual Number complexAdd (Complex const &n) const;

  double rpart, ipart;
};

class RealNumber : public Number
{
  friend class Complex;
  friend class Number;

  RealNumber (double r);
  RealNumber (const RealNumber &r);
  virtual ~RealNumber ();

  virtual Number operator + (Number const &n) const;
  virtual Number realAdd (RealNumber const &n) const;
  virtual Number complexAdd (Complex const &n) const;

  double val;
};

/// Used only by the letters.
Number::Number (BaseConstructor)
: rep (0),
  referenceCount (1)
{}

/// Used by static factory functions.
Number::Number ()
  : rep (0),
    referenceCount (0)
{}

/// Used by user and static factory functions.
Number::Number (const Number &n)
: rep (n.rep),
  referenceCount (0)
{
  std::cout << "Constructing a Number using Number::Number" << std::endl;
  if (n.rep)
    n.rep->referenceCount++;
}

Number Number::makeReal (double r)
{
  Number n;
  n.redefine (new RealNumber (r));
  return n;
}

Number Number::makeComplex (double rpart, double ipart)
{
  Number n;
  n.redefine (new Complex (rpart, ipart));
  return n;
}

Number::~Number()
{
  if (rep && --rep->referenceCount == 0)
    delete rep;
}

Number & Number::operator = (const Number &n)
{
  std::cout << "Assigning a Number using Number::operator=" << std::endl;
  Number temp (n);
  this->swap (temp);
  return *this;
}

void Number::swap (Number &n) throw ()
{
```

```
  std::swap (this->rep, n.rep);
}

Number Number::operator + (Number const &n) const
{
  return rep->operator + (n);
}

Number Number::complexAdd (Complex const &n) const
{
  return rep->complexAdd (n);
}

Number Number::realAdd (RealNumber const &n) const
{
  return rep->realAdd (n);
}

void Number::redefine (Number *n)
{
  if (rep && --rep->referenceCount == 0)
    delete rep;
  rep = n;
}

Complex::Complex (double d, double e)
  : Number (BaseConstructor()),
    rpart (d),
    ipart (e)
{
  std::cout << "Constructing a Complex" << std::endl;
}

Complex::Complex (const Complex &c)
  : Number (BaseConstructor()),
    rpart (c.rpart),
    ipart (c.ipart)
{
  std::cout << "Constructing a Complex using Complex::Complex" << std::endl;
}

Complex::~Complex()
{
  std::cout << "Inside Complex::~Complex()" << std::endl;
}

Number Complex::operator + (Number const &n) const
{
  return n.complexAdd (*this);
}

Number Complex::realAdd (RealNumber const &n) const
{
  std::cout << "Complex::realAdd" << std::endl;
  return Number::makeComplex (this->rpart + n.val,
                              this->ipart);
}

Number Complex::complexAdd (Complex const &n) const
{
  std::cout << "Complex::complexAdd" << std::endl;
  return Number::makeComplex (this->rpart + n.rpart,
                              this->ipart + n.ipart);
}

RealNumber::RealNumber (double r)
  : Number (BaseConstructor()),
    val (r)
{
  std::cout << "Constructing a RealNumber" << std::endl;
}

RealNumber::RealNumber (const RealNumber &r)
  : Number (BaseConstructor()),
    val (r.val)
{
  std::cout << "Constructing a RealNumber using RealNumber::RealNumber" << std::endl;
}

RealNumber::~RealNumber()
```

```
{
  std::cout << "Inside RealNumber::~RealNumber()" << std::endl;
}

Number RealNumber::operator + (Number const &n) const
{
  return n.realAdd (*this);
}

Number RealNumber::realAdd (RealNumber const &n) const
{
  std::cout << "RealNumber::realAdd" << std::endl;
  return Number::makeReal (this->val + n.val);
}

Number RealNumber::complexAdd (Complex const &n) const
{
  std::cout << "RealNumber::complexAdd" << std::endl;
  return Number::makeComplex (this->val + n.rpart, n.ipart);
}

namespace std
{
  template <>
  void swap (Number & n1, Number & n2)
  {
    n1.swap (n2);
  }
}

int main (void)
{
  Number n1 = Number::makeComplex (1, 2);
  Number n2 = Number::makeReal (10);
  Number n3 = n1 + n2;

  std::cout << "Finished" << std::endl;

  return 0;
}
```

Known Uses

Related Idioms

- Handle Body
- Envelope Letter

References

Advanced C++ Programming Styles and Idioms by James Coplien, Addison Wesley, 1992.

Attach by Initialization

Intent

Attach a user-defined object to a framework before program execution begins.

Also Known As

- Static-object-with-constructor

Motivation

Certain application programming frameworks, such as GUI frameworks (e.g., Microsoft MFC) and object request brokers (e.g., some CORBA implementations) use their own internal message loops (a.k.a. event loops) to control the entire application. Application programmers may or may not have the freedom to write the application-level `main` function. Often, the `main` function is buried deep inside the application framework (e.g, `AfxWinMain` in case of MFC). Lack of access to `main` keeps programmers from writing application-specific initialization code before the main event loop begins. Attach-by-initialization idiom is a way of executing application-specific code before the execution of framework-controlled loop begins.

Solution and Sample Code

In C++, global objects and static objects in global namespace are initialized before `main` begins execution. These objects are also known as objects of static storage duration. This property of objects of static storage duration can be used to attach an object to a *system* if programmers are not allowed to write their own `main` function. For instance, consider the following (smallest possible) example using Microsoft Foundation Classes (MFC).

```
///// File = Hello.h
class HelloApp: public CWinApp
{
public:
    virtual BOOL InitInstance ();
};
///// File = Hello.cpp

#include <afxwin.h>
#include "Hello.h"
HelloApp myApp; // Global "application" object
BOOL HelloApp::InitInstance ()
{
  m_pMainWnd = new CFrameWnd();
  m_pMainWnd->Create(0,"Hello, World!!");
  m_pMainWnd->ShowWindow(SW_SHOW);
  return TRUE;
}
```

The above example creates a window titled "Hello, World!" and nothing more. The key thing to note here is the global object `myApp` of type `HelloApp`. The `myApp` object is default-initialized before the execution of `main`. As a side-effect of initializing the objects, the constructor of `CWinApp` is also called. The `CWinApp` class is a part of the framework and invokes constructors of several other classes in the framework. During the execution of these constructors, the global object is attached to the framework. The object is later on retrieved by `AfxWinMain`, which is an equivalent of regular `main` in MFC. The `HelloApp::InitInstance` member function is shown just for the sake of completeness and is not an integral part of the idiom. This function is called after `AfxWinMain` begins execution.

Global and static objects can be initialized in several ways: default constructor, constructors with parameters, assignment from a return value of a function, dynamic initialization, etc.

Caveat

In C++, objects in the same compilation unit are created in order of definition. However, the order of initialization of objects of static storage duration across different compilation units is not well defined. Objects in a namespace are created before any function/variable in that namespace is accessed. This may or may not be before main. Order of destruction is the reverse of initialization order but the initialization order itself is not standardized. Because of this undefined behavior, *static initialization order problem* comes up when a constructor of a static object uses another static object that has not been initialized yet. This idiom makes it easy to run into this trap because it depends on a object of static storage duration.

Known Uses

- Microsoft Foundation Classes (MFC)

Related Idioms

References

- Proposed C++ language extension to improve portability of the Attach by Initialization idiom (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1995/N0717.htm)

Attorney-Client

Intent

Control the granularity of access to the implementation details of a class

Motivation

A `friend` declaration in C++ gives complete access to the internals of a class. Friend declarations are, therefore, frowned upon because they break carefully crafted encapsulations. Friendship feature of C++ does not provide any way to selectively grant access to a subset of private members of a class. Friendship in C++ is an all-or-nothing proposition. For instance, the following class `Foo` declares class `Bar` its friend. Class `Bar` therefore has access to **all** the private members of class `Foo`. This may not be desirable because it increases coupling. Class `Bar` cannot be distributed without class `Foo`.

```
class Foo
{
private:
  void A(int a);
  void B(float b);
  void C(double c);
  friend class Bar;
};

class Bar {
// This class needs access to Foo::A and Foo::B only.
// C++ friendship rules, however, give access to all the private members of Foo.
};
```

Providing selective access to a subset of members is desirable because the remaining (private) members can change interface if needed. It helps reduce coupling between classes. Attorney-Client idiom allows a class to precisely control the amount of access they give to their friends.

Solution and Sample Code

Attorney-client idiom works by adding a level of indirection. A *client* class that wants to control access to its internal details, appoints an *attorney* and makes it a friend --- a C++ friend! The `Attorney` class is crafted carefully to serve as a proxy to the `Client`. Unlike a typical proxy class, `Attorney` class replicates only a subset of `Client`'s private interface. For instance, consider class `Foo` wants to control access to its implementation details. For better clarity we rename it as `Client`. `Client` wants its `Attorney` to provide access to `Client::A` and `Client::B` only.

```
class Client
{
private:
  void A(int a);
  void B(float b);
  void C(double c);
```

```
  friend class Attorney;
};

class Attorney {
private:
  static void callA(Client & c, int a) {
    c.A(a);
  }
  static void callB(Client & c, float b) {
    c.B(b);
  }
  friend class Bar;
};

class Bar {
// Bar now has access to only Client::A and Client::B through the Attorney.
};
```

The `Attorney` class restricts access to a cohesive set of functions. The `Attorney` class has all inline static member functions, each taking a reference to an instance of the `Client` and forwarding the function calls to it. Some things are idiomatic about the `Attorney` class. Its implementation is entirely private, which prevents other unexpected classes gaining access to the internal of `Client`. The `Attorney` class determines which other classes, member functions, or free functions get access to it. It declares them as `friend` to allow access to its implementation and eventually the `Client`. Without the `Attorney` class, `Client` class would have declared the same set of friends giving them unrestrained access to the internals of `Client`.

It is possible to have multiple attorney classes providing access to different sets of implementation details of the client. For instance, class `AttorneyC` may provide access to the `Client::C` member function only. An interesting case emerges where an attorney class serves as a mediator for several different classes and provides cohesive access to their implementation details. Such a design is conceivable in case of inheritance hierarchies because friendship in C++ is not inheritable, but private virtual function overrides in derived classes can be called if base's private virtual functions are accessible. In the following example, the Attorney-Client idiom is applied to class `Base` and the `main` function. The `Derived::Func` function gets called via polymorphism. To access the implementation details of `Derived` class, however, the same idiom may be applied.

```
#include <cstdio>

class Base {
private:
  virtual void Func(int x) = 0;
  friend class Attorney;
public:
  virtual ~Base() {}
};

class Derived : public Base {
private:
  virtual void Func(int x)  {
    printf("Derived::Func\n"); // This is called even though main is not a friend of Derived.
  }

public:
  ~Derived() {}
};

class Attorney {
private:
  static void callFunc(Base & b, int x) {
    return b.Func(x);
  }
  friend int main (void);
};

int main(void) {
  Derived d;
  Attorney::callFunc(d, 10);
}
```

Known Uses

- Boost.Iterators library (http://www.boost.org/doc/libs/1_50_0/libs/iterator/doc/iterator_facade.html#iterator-core-access)
- Boost.Serialization: class boost::serialization::access (http://www.boost.org/doc/libs/1_50_0/libs/serialization/doc/serialization.html#member)

Related Idioms

References

Friendship and the Attorney-Client Idiom (Dr. Dobb's Journal) (http://drdobbs.com/184402053)

Barton-Nackman trick

Intent

Support overloaded operators without relying on namespaces or function template overload resolution.

Also Known As

The inventors originally referred to it as *Restricted Template Expansion*, though this term has never been widely used.

Motivation

John Barton and Lee Nackman first published this idiom in 1994 in order to work around limitations of the C++ implementations available at the time. Though it is no longer necessary for its original purpose, the current standard retains support for it.

At the time Barton and Nackman originally developed the idiom, C++ did not support overloading of function templates and many implementations still didn't support namespaces. This caused problems when defining operator overloads for class templates. Consider the following class:

```
template<typename T>
class List {
   // ...
};
```

The most natural way to define the equality operator is as a non-member function at namespace scope (and since compilers at the time didn't support namespaces, therefore at global scope). Defining `operator==` as a non-member function means that the two arguments are treated symmetrically, which doesn't happen if one argument is a `this` pointer to the object. Such an equality operator might look like this:

```
template<typename T>
bool operator==(List<T> const & lft, List<T> const & rgt) {
   //...
}
```

However, since function templates couldn't be overloaded at the time, and since putting the function in its own namespace wouldn't work on all platforms, this would mean that only one class could have such an equality operator. Doing the same thing for a second type would cause an ambiguity.

Solution and Sample Code

The solution works by defining an operator in the class as a friend function:

```
template<typename T>
class List {
 public:
    friend bool operator==(const List<T> & lft,
                           const List<T> & rgt) {
        // ...
    }
};
```

Instantiating the template now causes a *non-template* function to be injected into global scope with the argument types being concrete, fixed types. This non-template function can be selected through function overload resolution the same way as any other non-template function.

The implementation can be generalised by providing the friend functions as part of a base class that is inherited from via the Curiously Recurring Template Pattern:

```
template<typename T>
class EqualityComparable {
public:
    friend bool operator==(const T & lft, const T & rgt) { return lft.equalTo(rgt); }
    friend bool operator!=(const T & lft, const T & rgt) { return !lft.equalTo(rgt); }
};

class ValueType :
    private EqualityComparable<ValueType> {
 public:
    bool equalTo(const ValueType & other) const;
};
```

Known Uses

- Boost.Operators library (http://www.boost.org/doc/libs/1_50_0/libs/utility/operators.htm)

Related Idioms

- Curiously Recurring Template Pattern

References

- Barton-Nackman trick on Wikipedia (http://en.wikipedia.org/wiki/Barton–Nackman_trick)

Base-from-Member

Intent

To initialize a base class from a data-member of the derived class.

Also Known As

Motivation

In C++, base classes are initialized before any member of the derived classes. The reason for this is that members of a derived class may use *base* part of the object. Therefore, all the *base* parts (i.e., all the base classes) must be initialized before members of the derived class. Sometimes, however, it becomes necessary to initialize a base class from a data member that is available only in the derived class. It sounds contradictory to the rules of C++ language because the parameter (a member of derived class) that is passed to the base class constructor must be fully initialized. This creates circular initialization problem (an infinite regress (http://en.wikipedia.org/wiki/Infinite_regress)).

The following code, obtained from Boost[1] library, shows the problem.

```
#include <streambuf>  // for std::streambuf
#include <ostream>    // for std::ostream

namespace  std {
  class streambuf;
  class ostream {
    explicit ostream(std::streambuf * buf);
    //...
  };
}
class fdoutbuf   // A customization of streambuf
    : public std::streambuf
{
public:
    explicit fdoutbuf( int fd );
    //...
};

class fdostream
    : public std::ostream
{
protected:
    fdoutbuf buf;
public:
    explicit fdostream( int fd )
        : buf( fd ), std::ostream( &buf )
        // This is not allowed: buf can't be initialized before std::ostream.
        // std::ostream needs a std::streambuf object defined inside fdoutbuf.
    {}
};
```

The above code snippet shows a case where the programmer is interested in customizing the `std::streambuf` class. He/she does so in `fdoutbuf` by inheriting from `std::streambuf`. The `fdoutbuf` class is used as a member in `fdostream` class, which is-a kind of `std::ostream`. The `std::ostream` class, however, needs a pointer to a `std::streambuf` class, or its derived class. The type of pointer to `buf` is suitable but passing it makes sense only if `buf` is initialized. However, it won't be initialized unless all base classes are initialized. Hence the infinite regress. The base-from-member idiom addresses this problem.

Solution and Sample Code

This idiom makes use of the fact that base classes are initialized in the order they are declared. The derived class controls the order of its base classes, and in turn, controls the order in which they are initialized. In this idiom, a new class is added just to initialize the member in the derived class that is causing the problem. This new class is introduced in the base-class-list before all other base classes. Because the new class comes *before* the base class that needs the fully constructed parameter, it is initialized first and then the reference can be passed as usual. Here is the solution using base-from-member idiom.

```
#include <streambuf>  // for std::streambuf
#include <ostream>    // for std::ostream

class fdoutbuf
    : public std::streambuf
{
public:
    explicit fdoutbuf(int fd);
```

```
    //...
};

struct fdostream_pbase // A newly introduced class
{
    fdoutbuf sbuffer; // The member moved 'up' the hierarchy.
    explicit fdostream_pbase(int fd)
        : sbuffer(fd)
    {}
};

class fdostream
    : protected fdostream_pbase // This class will be initialized before the next one.
    , public std::ostream
{
public:
    explicit fdostream(int fd)
        : fdostream_pbase(fd),   // Initialize the newly added base before std::ostream.
          std::ostream(&sbuffer) //  Now safe to pass the pointer
    {}
    //...
};
int main(void)
{
  fdostream standard_out(1);
  standard_out << "Hello, World\n";

  return 0;
}
```

The `fdostream_pbase` class is the newly introduced class that now has the `sbuffer` member. The `fdostream` class inherits from this new class and adds it before `std::ostream` in its base class list. This ensures that `sbuffer` is initialized before and the pointer can be passed safely to the `std::ostream` constructor.

Known Uses

- Boost Base from Member (http://www.boost.org/doc/libs/1_47_0/libs/utility/base_from_member.html)

Related Idioms

References

1. Boost Utility http://www.boost.org/doc/libs/1_61_0/libs/utility/doc/html/base_from_member.html

Boost mutant

Intent

Reverse a pair of plain old data (POD) types without physically reorganizing or copying the data items.

Also Known As

Motivation

The need of this idiom is best motivated using the Boost.Bimap (http://www.boost.org/doc/libs/1_43_0/libs/bimap/doc/html/index.html) data structure. Boost.Bimap is a bidirectional maps library for C++. In bimap<X,Y>, values of types X and Y both can serve as keys. The implementation of such a data structure can be optimized using the boost mutant idiom.

Solution and Sample Code

Boost mutant idiom makes use of *reinterpret_cast* and depends heavily on assumption that the memory layouts of two different structures with identical data members (types and order) are interchangeable. Although the C++ standard does not guarantee this property, virtually all the compilers satisfy it. Moreover, the mutant idiom is standard if only POD types are used.[1] The following example shows how boost mutant idiom works.

```
template <class Pair>
struct Reverse
{
    typedef typename Pair::first_type  second_type;
    typedef typename Pair::second_type first_type;
    second_type second;
    first_type first;
};

template <class Pair>
Reverse<Pair> & mutate(Pair & p)
{
  return reinterpret_cast<Reverse<Pair> &>(p);
}

int main(void)
{
  std::pair<double, int> p(1.34, 5);

  std::cout << "p.first = " << p.first << ", p.second = "  << p.second << std::endl;
  std::cout << "mutate(p).first = " << mutate(p).first << ", mutate(p).second = "  << mutate(p).second << std::endl;
}
```

Given a `std::pair<X,Y>` object of POD data members only, the layout of the `Reverse<std::pair<X,Y>>` is identical to that of pair's on most compilers. The `Reverse` template reverses the names of the data members without physically reversing the data. A helper `mutate` function is used to easily construct a `Reverse<Pair>` reference, which can be considered as a *view* over the original pair object. The output of the above program confirms that the reverse view can be obtained without reorganizing data:

p.first = 1.34, p.second = 5

mutate(p).first = 5, mutate(p).second = 1.34

Known Uses

- Boost.Bimap (http://www.boost.org/doc/libs/1_43_0/libs/bimap/doc/html/index.html)

Related Idioms

References

- Boost.Bimap (http://www.boost.org/doc/libs/1_43_0/libs/bimap/doc/html/index.html) library, Matias Capeletto

1. http://beta.boost.org/doc/libs/1_43_0/libs/bimap/test/test_mutant.cpp

Calling Virtuals During Initialization

Intent

Simulate calling of virtual functions during object initialization.

Also Known As

Dynamic Binding During Initialization idiom

Motivation

Sometimes it is desirable to invoke virtual functions of derived classes while a derived object is being initialized. Language rules explicitly prohibit this from happening because calling member functions of derived object before derived part of the object is initialized is dangerous. It is not a problem if the virtual function does not access data members of the object being constructed. But there is no general way of ensuring it.

```
class Base {
public:
  Base();
  ...
  virtual void foo(int n) const; // often pure virtual
  virtual double bar() const;    // often pure virtual
};

Base::Base()
{
  ... foo(42) ... bar() ...
  // these will not use dynamic binding
  // goal: simulate dynamic binding in those calls
}

class Derived : public Base {
public:
  ...
  virtual void foo(int n) const;
  virtual double bar() const;
};
```

Solution and Sample Code

There are multiple ways of achieving the desired effect. Each has its own pros and cons. In general the approaches can be divided into two categories. One using two phase initialization and other one using only single phase initialization.

Two phase initialization technique separates object construction from initializing its state. Such a separation may not be always possible. Initialization of object's state is clubbed together in a separate function, which could be a member function or a free standing function.

```
class Base {
public:
  void init();  // may or may not be virtual
  ...
  virtual void foo(int n) const; // often pure virtual
  virtual double bar() const;    // often pure virtual
};

void Base::init()
{
  ... foo(42) ... bar() ...
  // most of this is copied from the original Base::Base()
}

class Derived : public Base {
public:
  Derived (const char *);
  virtual void foo(int n) const;
  virtual double bar() const;
};
```

- using non-member function

```
template <class Derived, class Parameter>
std::auto_ptr <Base> factory (Parameter p)
```

```
{
  std::auto_ptr <Base> ptr (new Derived (p));
  ptr->init ();
  return ptr;
}
```

A non-template version of this approach can be found here. The factory function can be moved inside base class but it has to be static.

```
class Base {
  public:
    template <class D, class Parameter>
    static std::auto_ptr <Base> Create (Parameter p)
    {
       std::auto_ptr <Base> ptr (new D (p));
       ptr->init ();
       return ptr;
    }
};
int main ()
{
  std::auto_ptr <Base> b = Base::Create <Derived> ("para");
}
```

Constructors of class Derived should be made private to prevent users from accidentally using them. Interfaces should be easy to use correctly and hard to use incorrectly - remember? The factory function should then be friend of the derived class. In case of member create function, Base class can be friend of Derived.

- Without using two phase initialization

Achieving desired effect using a helper hierarchy is described here but an extra class hierarchy has to be maintained, which is undesirable. Passing pointers to static member functions is C'ish. Curiously Recurring Template Pattern idiom can be useful in this situation.

```
class Base {
};
template <class D>
class InitTimeCaller : public Base {
  protected:
    InitTimeCaller () {
       D::foo ();
       D::bar ();
    }
};
class Derived : public InitTimeCaller <Derived>
{
  public:
    Derived () : InitTimeCaller <Derived> () {
        cout << "Derived::Derived()\n";
    }
    static void foo () {
        cout << "Derived::foo()\n";
    }
    static void bar () {
        cout << "Derived::bar()\n";
    }
};
```

Using Base-from-member idiom more complex variations of this idiom can be created.

Known Uses

Related Idioms

Capability Query

Intent

To check at runtime whether an object supports an interface

Also Known As

Motivation

Separating interface from implementation is a good object oriented software design practice. In C++, the Interface Class idiom is used to separate interface from implementation and invoke the public methods of any abstraction using runtime polymorphism. Extending the example in the interface class idiom, a concrete class may implement multiple interfaces as shown below.

```
class Shape { // An interface class
   public:
    virtual ~Shape();
    virtual void draw() const = 0;
    //...
};
class Rollable { // One more interface class
  public:
    virtual ~Rollable();
    virtual void roll() = 0;
};
class Circle : public Shape, public Rollable { // circles roll - concrete class
    //...
    void draw() const;
    void roll();
    //...
};
class Square : public Shape { // squares don't roll - concrete class
    //...
    void draw() const;
    //...
};
```

Now if we are given a container of pointers to abstract Rollable class, we can simply invoke the roll function on every pointer, as described in the interface class idiom.

```
std::vector<Rollable *> rollables;
//  Fill up rollables vector somehow.
for (vector<Rollable *>::iterator iter (rollables.begin());
     iter != rollables.end();
     ++iter)
{
  iter->roll();
}
```

Sometimes it is not possible to know in advance whether or not an object implements a particular interface. Such a situation commonly arises if an object inherits from multiple interface classes. To discover the presence or absence of the interface at runtime, capability query is used.

Solution and Sample Code

In C++, a capability query is typically expressed as a `dynamic_cast` between unrelated types .

```
Shape *s = getSomeShape();
if (Rollable *roller = dynamic_cast<Rollable *>(s))
  roller->roll();
```

This use of `dynamic_cast` is often called a **cross-cast,** because it attempts a conversion across a hierarchy, rather than up or down a hierarchy. In our example hierarchy of shapes and rollables, `dynamic_cast` to `Rollable` will succeed only for `Circle` and not for `Square` as the later one does not inherit from `Rollable` interface class.

Excessive use of capability queries is often an indication of bad object-oriented design.

Known Uses

Acyclic Visitor Pattern (http://www.objectmentor.com/resources/articles/acv.pdf) - Robert C. Martin.

Related Idioms

- Interface Class
- Inner Class

References

Capability Queries - C++ Common Knowledge by Stephen C. Dewhurst

Checked delete

Intent

Increase safety of delete expression.

Also Known As

Motivation and Sample Code

The C++ Standard allows, in 5.3.5/5, pointers to incomplete class types to be deleted with a delete-expression. When the class has a non-trivial destructor, or a class-specific operator delete, the behavior is undefined. Some compilers issue a warning when an incomplete type is deleted, but unfortunately, not all do, and programmers sometimes ignore or disable warnings.

In the following example, main.cpp can see the definition of Object. However, main() calls delete_object() -- defined in deleter.cpp -- which *does not* see the definition of Object, but only forward declares it. Calling `delete` on a partially defined type like this is undefined behavior which some compilers do not flag.

```
////////////////////
// File: deleter.hpp
////////////////////
// declares but does not define Object
struct Object;
void delete_object(Object* p);

////////////////////
// File: deleter.cpp
////////////////////
#include "deleter.hpp"

// Deletes an Object without knowing its definition
void delete_object(Object* p) { delete p; }

////////////////////
// File: object.hpp
////////////////////
struct Object
{
```

```
  // this user-defined destructor won't be called when delete
  // is called on a partially-defined (i.e., predeclared) Object
  ~Object() {
     // ...
  }
};

////////////////////
// File: main.cpp
////////////////////
#include "deleter.hpp"
#include "object.hpp"

int main() {
  Object* p = new Object;
  delete_object(p);
}
```

Solution and Sample Code

The checked delete idiom relies on calls to a function template to delete memory, rather than calls to `delete`, which fails for declared but undefined types.

The following is the implementation of boost::checked_delete, a function template in the Boost Utility library. It forces a compilation error by invoking the `sizeof` operator on the parameterizing type, `T`. If `T` is declared but not defined, `sizeof(T)` will generate a compilation error or return zero, depending upon the compiler. If `sizeof(T)` returns zero, checked_delete triggers a compilation error by declaring an array with -1 elements. The array name is *type_must_be_complete*, which should appear in the error message in that case, helping to explain the mistake.

```
template<class T>
inline void checked_delete(T * x)
{
    typedef char type_must_be_complete[ sizeof(T)? 1: -1 ];
    (void) sizeof(type_must_be_complete);
    delete x;
}
template<class T>
struct checked_deleter : std::unary_function <T *, void>
{
    void operator()(T * x) const
    {
        boost::checked_delete(x);
    }
};
```

NOTE: This same technique can be applied to the array delete operator as well.

WARNING: std::auto_ptr does not use anything equivalent to checked delete. Therefore, instantiating std::auto_ptr using an incomplete type may cause undefined behavior in its destructor if, at the point of declaration of the std::auto_ptr, the template parameter type is not fully defined.

Known Uses

Related Idioms

References

http://www.boost.org/libs/utility/checked_delete.html

Clear-and-minimize

Intent

Clear a container and minimize the capacity of the container.

Also Known As

This is sometimes called the *swap with temporary* idiom.

Motivation

Standard library containers often allocate more memory than the actual number of elements in them. Such a policy results in an optimization of saving some allocations when a container grows in size. On the other hand, when size of the container reduces, there is often leftover capacity in the container. The leftover capacity of the container can be unnecessary wastage of memory resources. clear-and-minimize idiom has been developed to clear a container and reduce the extra capacity to a minimum of zero and thereby saving memory resources.

Solution and Sample Code

Clear-and-minimize idiom is as simple as the one given below.

```
std::vector <int> v;
//... Lots of push_backs and then lots of remove on v.
std::vector<int>().swap (v);
```

The first half of the statement, std::vector<int>() creates a temporary vector of integers and it is guaranteed to allocate zero raw memory or an implementation minimum. The second half of the statement swaps the temporary vector with v using non-throwing swap idiom, which is efficient. After swapping, the temporary created by the compiler goes out of scope and the chunk of memory originally held by v is released.

Solution in C++14

Since C++11, some containers declare the function `shrink_to_fit()`, e.g. vector, deque, basic_string. `shrink_to_fit()` which is a non-binding request to reduce `capacity()` to `size()`. Thus, using `clear()` and `shrink_to_fit()` is a non-binding request to clear-and-minimize.

Known Uses

Related Idioms

- Shrink-to-fit
- Non-throwing swap

References

- Programming Languages — C++ (http://open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf) draft standard.

Coercion by Member Template

Intent

To increase the flexibility of a class template's interface by allowing the class template to participate in the same implicit type conversions (coercion) as its parameterizing types enjoy.

Also Known As

Motivation

It is often useful to extend a relationship between two types to class templates specialized with those types. For example, suppose that class D derives from class B. A pointer to an object of type D can be assigned to a pointer to B; C++ supports that implicitly. However, types composed of these types do not share the relationship of the composed types. That applies to class templates as well, so a `Helper<D>` object normally cannot be assigned to a `Helper<B>` object.

```
class B {};
class D : public B {};
template <class T>
class Helper {};

B *bptr;
D *dptr;
bptr = dptr; // OK; permitted by C++

Helper<B> hb;
Helper<D> hd;
hb = hd; // Not allowed but could be very useful
```

There are cases where such conversions are useful, such as allowing conversion from `std::auto_ptr<D>` to `std::auto_ptr<B>`. That is quite intuitive, but isn't supported without using the Coercion by Member Template Idiom.

Solution and Sample Code

Define member template functions, in a class template, which rely on the implicit type conversions supported by the parameter types. In the following example, the templated constructor and assignment operator work for any type U, for which initialization or assignment of a `T *` from a `U *` is allowed.

```
template <class T>
class Ptr
{
  public:
    Ptr () {}

    Ptr (Ptr const & p)
      : ptr (p.ptr)
    {
      std::cout << "Copy constructor\n";
    }

    // Supporting coercion using member template constructor.
    // This is not a copy constructor, but behaves similarly.
    template <class U>
    Ptr (Ptr <U> const & p)
      : ptr (p.ptr) // Implicit conversion from U to T required
    {
      std::cout << "Coercing member template constructor\n";
    }

    // Copy assignment operator.
    Ptr & operator = (Ptr const & p)
    {
      ptr = p.ptr;
      std::cout << "Copy assignment operator\n";
      return *this;
    }

    // Supporting coercion using member template assignment operator.
```

```
    // This is not the copy assignment operator, but works similarly.
    template <class U>
    Ptr & operator = (Ptr <U> const & p)
    {
      ptr = p.ptr; // Implicit conversion from U to T required
      std::cout << "Coercing member template assignment operator\n";
      return *this;
    }

    T *ptr;
};

int main (void)
{
   Ptr <D> d_ptr;
   Ptr <B> b_ptr (d_ptr); // Now supported
   b_ptr = d_ptr;         // Now supported
}
```

Another use for this idiom is to permit assigning an array of pointers to a class to an array of pointers to that class' base. Given that D derives from B, a D object **is-a** B object. However, an array of D objects **is-not-an** array of B objects. This is prohibited in C++ because of slicing. Relaxing this rule for an array of pointers can be helpful. For example, an array of pointers to D should be assignable to an array of pointers to B (assuming B's destructor is virtual). Applying this idiom can achieve that, but extra care is needed to prevent copying arrays of pointers to one type to arrays of pointers to a derived type. Specializations of the member function templates or SFINAE can be used to achieve that.

The following example uses a templated constructor and assignment operator expecting `Array<U *>` to only allow copying Arrays of pointers when the element types differ.

```
template <class T>
class Array
{
  public:
    Array () {}
    Array (Array const & a)
    {
      std::copy (a.array_, a.array_ + SIZE, array_);
    }

    template <class U>
    Array (Array <U *> const & a)
    {
      std::copy (a.array_, a.array_ + SIZE, array_);
    }

    template <class U>
    Array & operator = (Array <U *> const & a)
    {
      std::copy (a.array_, a.array_ + SIZE, array_);
    }

    enum { SIZE = 10 };
    T array_[SIZE];
};
```

Many smart pointers such as std::auto_ptr, boost::shared_ptr employ this idiom.

Caveats

A typical mistake in implementing the Coercion by Member Template Idiom is failing to provide the non-template copy constructor or copy assignment operator when introducing the templated copy constructor and assignment operator. A compiler will automatically declare a copy constructor and a copy assignment operator if a class does not declare them, which can cause hidden and non-obvious faults when using this idiom.

Known Uses

- std::auto_ptr
- boost::shared_ptr

Related Idioms

- Generic Container Idioms

References

- std::auto_ptr Implementation (http://www.josuttis.com/libbook/util/autoptr.hpp.html)

Intent

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

References

Computational Constructor

Intent

- Optimize return-by-value
- Allow Return Value Optimization (RVO) on compilers that cannot handle Named Return Value Optimization (NRVO)

Also Known As

Motivation

Returning large C++ objects by value is expensive in C++. When a locally created object is returned by-value from a function, a temporary object is created on the stack. The temporaries are often very short-lived because they are either assigned to other objects or passed to other functions. Temporary objects generally go out-of-scope and hence destroyed after the statement that created them is executed completely.

Over the years compilers have evolved to apply several optimizations to avoid creation of temporaries because it is often wasteful and hampers performance. Return Value Optimization (RVO) and Named Return Value Optimization (NRVO) are two popular compiler techniques that try to optimize away the temporaries (a.k.a. copy elision). A brief explanation of RVO is in order.

Return Value Optimization

The following example demonstrates a scenario where the implementation may eliminate one or both of the copies being made, even if the copy constructor has a visible side effect (printing text). The first copy that may be eliminated is the one where `Data(c)` is copied into the function `func`'s return value. The second copy that may be eliminated is the copy of the temporary object returned by `func` to `d1`. More on RVO is available on Wikipedia (http://en.wikipedia.org/wiki/Return_value_optimization)

```
struct Data {
  Data(char c = 0)
  {
    std::fill(bytes, bytes + 16, c);
  }
  Data(const Data & d)
  {
    std::copy(d.bytes, d.bytes+16, this->bytes);
    std::cout << "A copy was made.\n";
  }
private:
  char bytes[16];
};

Data func(char c) {
  return Data(c);
}

int main(void) {
   Data d1 = func('A');
}
```

Following pseudo-code shows how both the copies of `Data` can be eliminated.

```
void func(Data * target, char c)
{
  new (target) Data (c);  // placement-new syntax (no dynamic allocation here)
  return;                 // Note void return type.
}
int main (void)
{
   char bytes[sizeof(Data)];                  // uninitialized stack-space to hold a Data object
   func(reinterpret_cast<Data *>(bytes), 'A'); // Both the copies of Data elided
   reinterpret_cast<Data *>(bytes)->~Data();   // destructor
}
```

Named Return Value Optimization (NRVO) is a more advanced cousin of RVO and not all compilers support it. Note that function `func` above did not name the local object it created. Often functions are more complicated than that. They create local objects, manipulate its state, and return the updated object. Eliminating the local object in such cases requires NRVO. Consider the following somewhat contrived example to emphasize the *computational* part of this idiom.

```
class File {
private:
  std::string str_;
public:
  File() {}
  void path(const std::string & path) {
    str_ = path;
  }
  void name(const std::string & name)  {
    str_ += "/";
    str_ += name;
  }
  void ext(const std::string & ext) {
    str_ += ".";
    str_ += ext;
  }
};
```

```
File getfile(void) {
  File f;
  f.path("/lib");
  f.name("libc");
  f.ext("so");
  f.ext("6");

  // RVO is not applicable here because object has a name = f
  // NRVO is possible but its support is not universal.
  return f;
}

int main (void) {
  File  f = getfile();
}
```

In the above example, function `getfile` does a lot of *computation* on object `f` before returning it. The implementation cannot use RVO because the object has a name ("f"). NRVO is possible but its support is not universal. Computational constructor idiom is a way to achieve return value optimization even in such cases.

Solution and Sample Code

To exploit RVO, the idea behind computational constructor idiom is to put the computation in a constructor so that the compiler is more likely to perform the optimization. A new four parameter constructor has been added just to enable RVO, which is a *computational constructor* for class `File`. The `getfile` function is now much more simple than before and the compiler will likely apply RVO here.

```
class File
{
private:
  std::string str_;
public:
  File() {}

  // The following constructor is a computational constructor
  File(const std::string & path,
       const std::string & name,
       const std::string & ext1,
       const std::string & ext2)
    : str_(path + "/" + name + "." + ext1 + "." + ext2) { }

  void path(const std::string & path);
  void name(const std::string & name);
  void ext(const std::string & ext);
};

File getfile(void) {
  return File("/lib", "libc", "so", "6"); // RVO is now applicable
}

int main (void) {
  File  f = getfile();
}
```

Consequences

A common criticism against the computational constructor idiom is that it leads to *unnatural* constructors, which is partly true for the class `File` shown above. If the idiom is applied judiciously, it can limit the proliferation of *computational* constructors in a class and yet provide better run-time performance.

Known Uses

Related Idioms

References

- Dov Bulka, David Mayhew, “Efficient C++; Performance Programming Techniques”, Addison Wesley

Concrete Data Type

Intent

To control object's scope and lifetime by allowing or disallowing dynamic allocation using the free store (heap)

Also Known As

Motivation

C++ provides two ways of controlling lifetime of an object and binding it with a program level identifier (variable). First is a scoped variable and an object, which are destroyed immediately after the scope ends (*e.g.,* function scope integers). Second is a scoped variable (often a pointer) and a dynamically allocated object in the free store. In this case, at the end of the scope of the variable, the variable ceases to exist but the object's lifetime continues (*e.g.,* singletons, a window object). It is possible to force the choice of the lifetime of an object either first way or the second using the Concrete Data Type idiom.

Solution and Sample Code

This idiom simply uses class level access modifiers (private, protected) to achieve the goal. The following code shows how a MouseEventHandler class forces dynamic allocation.

```
class EventHandler
{
  public:
    virtual ~EventHandler () {}
};
class MouseEventHandler : public EventHandler // Note inheritance
{
  protected:
    ~MouseEventHandler () {} // A protected virtual destructor.
  public:
    MouseEventHandler () {} // Public Constructor.
};
int main (void)
{
  MouseEventHandler m; // A scoped variable is not allowed as destructor is protected.
  EventHandler *e = new MouseEventHandler (); // Dynamic allocation is allowed
  delete e;  // Polymorphic delete. Does not leak memory.
}
```

Another way to force dynamic allocation is to prevent direct access to the constructor and instead provide a static function instance() to return a dynamically allocated object. It is in many ways similar to the Singleton design pattern. Moreover, it is not strictly necessary to use polymorphic delete to reclaim memory. A member function destroy() can serve the purpose saving the space required for a v-table pointer.

```
class MouseEventHandler // Note no inheritance
{
  protected:
    MouseEventHandler () {} // Protected Constructor.
    ~MouseEventHandler () {} // A protected, non-virtual destructor.
  public:
    static MouseEventHandler * instance () { return new MouseEventHandler(); }
    void destroy () { delete this; }  // Reclaim memory.
};
```

An opposite extreme of this idiom is to force scoped variable (a.k.a. automatic variable) only. It can be achieved using a private new operator.

```
class ScopedLock
{
  private:
    static void * operator new (size_t size); // Disallow dynamic allocation
    static void * operator new (size_t, void * mem);  // Disallow placement new as well.
};
int main (void)
{
   ScopedLock s; // Allowed
   ScopedLock * sl = new ScopedLock (); // Standard new and nothrow new are not allowed.
   void * buf = ::operator new (sizeof (ScopedLock));
   ScopedLock * s2 = new(buf) ScopedLock;  // Placement new is also not allowed
}
```

ScopedLock object can't be allocated dynamically with standard uses of new operator, nothrow new, and the placement new.

Known Uses

Related Idioms

References

- Concrete Data Type (http://www.laputan.org/pub/sag/coplien-idioms.doc) - J. Coplien.

Const auto_ptr

Intent

To prevent transfer of ownership of a held resource. Note: auto_ptr has been deprecated and is replaced by shared_ptr, unique_ptr and weak_ptr in C++11, thereby this idiom is no longer suggested.

Also Known As

Motivation

Often it is desirable to enforce a design decision of non-transferable ownership in code and enforce it with the help of compiler. Ownership in consideration here is of any resource such as memory, database connections and so on. const auto_ptr idiom can be used if we don't want ownership of the acquired resource to be transfered outside the scope or from one object to the another.

auto_ptr without any cv-qualifier (fancy name for const and volatile) has move semantics as described in the Move Constructor idiom. It basically means that ownership of memory is unconditionally transferred from right hand side object to the left hand side object of an assignment, but it ensures that there is always a single owner of the resource. const auto_ptr can prevent the transfer.

Solution and Sample Code

Declare the auto_ptr holding memory resource as const.

```
const auto_ptr <X> xptr (new X());
auto_ptr <X> yptr (xptr); // Not allowed, compilation error.
xptr.release();           // Not allowed, compilation error.
xptr.reset( new X() );      // Not allowed, compilation error.
```

Compiler issues a warning here because the copy-constructor of yptr is not really a copy-constructor but in fact it is a move constructor, which takes a non-const reference to an auto_ptr, as given in Move Constructor idiom. A non-const reference can't bind with a const variable and therefore, compiler flags an error.

Consequences

- An undesirable consequence of const auto_ptr idiom is that compiler can't provide a default copy-constructor to a class that has a const auto_ptr member in it. This is because the compiler generated copy-constructor always takes a const RHS as a parameter, which can't bind with a non-const move constructor of auto_ptr. The solution is to use Virtual Constructor idiom or use **boost::scoped_ptr**, which explicitly prohibits copying by denying access to assignment and copy-constructor.

Known Uses

Related Idioms

- Move Constructor

References

- http://www.gotw.ca/publications/using_auto_ptr_effectively.htm
- http://www.boost.org/libs/smart_ptr/scoped_ptr.htm

Construct On First Use

Intent

Ensure that an object is initialized before its first use. Specifically, ensure that non-local static object is initialized before its first use.

Also Known As

Lazy construction/evaluation

Motivation

Static objects that have non-trivial constructors must be initialized before they are used. It is possible to access an uninitialized non-local static object before its initialization if proper care is not exercised.

```
struct Bar {
  Bar () {
    cout << "Bar::Bar()\n";
  }
  void f () {
    cout << "Bar::f()\n";
  }
};
struct Foo {
  Foo () {
    bar_.f ();
  }
  static Bar bar_;
};

Foo f;
Bar Foo::bar_;
```

```
int main () {}
```

In the above code, Bar::f() gets called before its constructor. It should be avoided.

Solution and Sample Code

There are 2 possible solutions, which depends upon whether the destructor of the object in consideration has non-trivial destruction semantics. Wrap the otherwise static object in a function so that function initializes it before it is used.

- **Construct on first use using dynamic allocation**

```
struct Foo {
  Foo () {
    bar().f ();
  }
 Bar & bar () {
    static Bar *b = new Bar ();
    return *b;
 }
};
```

If the object has a destructor with non-trivial semantics, local static object is used instead of dynamic allocation as given below.

- **Construct on first use using local static**

```
struct Foo {
  Foo () {
    bar().f ();
  }
 Bar & bar () {
    static Bar b;
    return b;
 }
};
```

Known Uses

- Singleton pattern implementations often use this idiom.
- ACE_TSS<T> class template in Adaptive Communication Environment (ACE) for creating and accessing objects in thread specific storage (TSS) uses this idiom.

Related Idioms

- Nifty/Schwarz Counter

References

Construction Tracker

Intent

To identify the data member that throws an exception when initialization of two or more objects in the constructor's initialization list can throw the same exception type

Also Known As

Motivation

When two or more objects are initialized in a constructor's initialization list and all of them can throw the same exception (std::exception), tracking which one of them failed become a tricky issue as there can be only one try block surrounding the initialization list. Such a try block has a special name called 'constructor try block', which is nothing but a 'function-try block'.

Solution and Sample Code

Construction Tracker idiom uses a simple technique to track successful construction on objects in the initialization list. A counter is simply incremented as constructors of objects finish successfully one-by-one. It cleverly uses bracket operator to inject the counter increments in between calls to the constructors all being invisible to the user of the class.

```
#include <iostream>
#include <stdexcept>
#include <cassert>

struct B {
    B (char const *) { throw std::runtime_error("B Error"); }
};
struct C {
    C (char const *) { throw std::runtime_error("C Error"); }
};
class A {
   B b_;
   C c_;
   enum TrackerType { NONE, ONE, TWO };
public:
   A( TrackerType tracker = NONE)
   try    // A constructor try block.
     : b_((tracker = ONE, "hello")) // Can throw std::exception
     , c_((tracker = TWO, "world")) // Can throw std::exception
     {
        assert(tracker == TWO);
        // ... constructor body ...
     }
   catch (std::exception const & e)
     {
        if (tracker == ONE) {
           std::cout << "B threw: " << e.what() << std::endl;
        }
        else if (tracker == TWO) {
           std::cout << "C threw: " << e.what() << std::endl;
        }
        throw;
     }
};

int main (void)
{
    try {
        A a;
    }
    catch (std::exception const & e) {
          std::cout << "Caught: " << e.what() << std::endl;
    }
    return 0;
}
```

The double parentheses is how the bracket operator is used to place in the assignment to the tracker. This idiom critically depends upon the constructor of B and C taking at least one parameter. If class B and C does not take parameters, then an adapter class needs to be written such that it the adapter class will accept a dummy parameter and calling the default parameters of B and C. Such an adapter can be written using More C++

Idioms/Parameterized Base Class idiom using mixin-from-below technique. The adapter class can also be completely encapsulated inside class A. In the consturctor of class A, the tracker parameter has a default value and therefore it does no bother the user.

Known Uses

Related Idioms

References

- Smart Pointers Reloaded (III): Constructor Tracking (http://erdani.org/publications/cuj-2004-02.pdf)

Copy-and-swap

Intent

To create an exception safe implementation of overloaded assignment operator.

Also Known As

Create-Temporary-and-Swap

Motivation

Exception safety is a very important corner stone of highly reliable C++ software that uses exceptions to indicate "exceptional" conditions. There are at least 3 types of exception safety levels: basic, strong, and no-throw. Basic exception safety should be offered always as it is usually cheap to implement. Guaranteeing strong exception safety may not be possible in all the cases. The copy-and-swap idiom allows an assignment operator to be implemented elegantly with strong exception safety.

Solution and Sample Code

Create a temporary and swap idiom acquires new resource before it forfeits its current resource. To acquire the new resource, it uses RAII idiom. If the acquisition of the new resource is successful, it exchanges the resources using the non-throwing swap idiom. Finally, the old resource is released as a side effect of using RAII in the first step.

```
class String
{
    char * str;
public:
    String & operator=(const String & s)
    {
        String temp(s); // Copy-constructor -- RAII
        temp.swap(*this); // Non-throwing swap

        return *this;
    }// Old resources released when destructor of temp is called.
    void swap(String & s) throw() // Also see the non-throwing swap idiom
    {
        std::swap(this->str, s.str);
    }
};
```

Some variations of the above implementation are also possible. A check for self assignment is not strictly necessary but can give some performance improvements in (rarely occurring) self-assignment cases.

```
class String
{
    char * str;
public:
    String & operator=(const String & s)
    {
        if (this != &s)
        {
            String(s).swap(*this); //Copy-constructor and non-throwing swap
        }

        // Old resources are released with the destruction of the temporary above
        return *this;
    }
    void swap(String & s) throw() // Also see non-throwing swap idiom
    {
        std::swap(this->str, s.str);
    }
};
```

copy elision and copy-and-swap idiom

Strictly speaking, explicit creation of a temporary inside the assignment operator is not necessary. The parameter (right hand side) of the assignment operator can be passed-by-value to the function. The parameter itself serves as a temporary.

```
String & operator = (String s) // the pass-by-value parameter serves as a temporary
{
   s.swap (*this); // Non-throwing swap
   return *this;
}// Old resources released when destructor of s is called.
```

This is not just a matter of convenience but in fact an optimization. If the parameter (s) binds to an lvalue (another non-const object), a copy of the object is made automatically while creating the parameter (s). However, when s binds to an rvalue (temporary object, literal), the copy is typically elided, which saves a call to a copy constructor and a destructor. In the earlier version of the assignment operator where the parameter is accepted as const reference, copy elision does not happen when the reference binds to an rvalue. This results in an additional object being created and destroyed.

In C++11, such an assignment operator is known as a **unifying** assignment operator because it eliminates the need to write two different assignment operators: copy-assignment and move-assignment. As long as a class has a move-constructor, a C++11 compiler will always use it to optimize creation of a copy from another temporary (rvalue). Copy-elision is a comparable optimization in non-C++11 compilers to achieve the same effect.

```
String createString(); // a function that returns a String object.
String s;
s = createString();
// right hand side is a rvalue. Pass-by-value style assignment operator
// could be more efficient than pass-by-const-reference style assignment
// operator.
```

Not every class benefits from this style of assignment operator. Consider a String assignment operator, which releases old memory and allocates new memory only if the existing memory is insufficient to hold a copy of the right hand side String object. To implement this optimization, one would have to write a custom assignment operator. Since a new String copy would nullify the memory allocation optimization, this custom assignment operator would have to avoid copying its argument to any temporary Strings, and in particular would need to accept its parameter by const reference.

Known Uses

Related Idioms

- Resource Acquisition Is Initialization
- Non-throwing swap

References

Copy-on-write

Intent

Achieve lazy copy optimization. Like lazy initialization, do the work just when you need because of efficiency.

Also Known As

- COW (copy-on-write)
- Lazy copy

Motivation

Copying an object can sometimes cause a performance penalty. If objects are frequently copied but infrequently modified later, copy-on-write can provide significant optimization. To implement copy-on-write, a smart pointer to the real content is used to encapsulate the object's value, and on each modification an object reference count is checked; if the object is referenced more than once, a copy of the content is created before modification.

Solution and Sample Code

```
#ifndef COWPTR_HPP
#define COWPTR_HPP

#include <boost/shared_ptr.hpp>

template <class T>
class CowPtr
{
    public:
        typedef boost::shared_ptr<T> RefPtr;

    private:
        RefPtr m_sp;

        void detach()
        {
            T* tmp = m_sp.get();
            if( !( tmp == 0 || m_sp.unique() ) ) {
                m_sp = RefPtr( new T( *tmp ) );
            }
        }

    public:
        CowPtr(T* t)
            :   m_sp(t)
        {}
        CowPtr(const RefPtr& refptr)
            :   m_sp(refptr)
        {}
        const T& operator*() const
        {
            return *m_sp;
        }
        T& operator*()
        {
```

```
            detach();
            return *m_sp;
        }
        const T* operator->() const
        {
            return m_sp.operator->();
        }
        T* operator->()
        {
            detach();
            return m_sp.operator->();
        }
};

#endif
```

This implementation of copy-on-write is generic, but apart from the inconvenience of having to refer to the inner object through smart pointer dereferencing, it suffers from at least one drawback: classes that return references to their internal state, like

```
char & String::operator[](int)
```

can lead to unexpected behaviour.[1]

Consider the following code snippet

```
CowPtr<String> s1 = "Hello";
char &c = s1->operator[](4); // Non-const detachment does nothing here
CowPtr<String> s2(s1); // Lazy-copy, shared state
c = '!'; // Uh-oh
```

The intention of the last line is to modify the original string `s1`, not the copy, but as a side effect `s2` is also accidentally modified.

A better approach is to write a custom copy-on-write implementation which is encapsulated in the class we want to lazy-copy, transparently to the user. In order to fix the problem above, one can flag objects that have given away references to inner state as "unshareable"—in other words, force copy operations to deep-copy the object. As an optimisation, one can revert the object to "shareable" after any non-const operations that do not give away references to inner state (for example, `void string::clear()`), because client code expects such references to be invalidated anyway.[1]

Known Uses

- Active Template Library
- Many Qt classes (implicit sharing) (http://archive.is/20121222041326/cdumez.blogspot.tw/2011/03/implicit-explicit-data-sharing-with-qt.html)

Related Idioms

References

1. Herb Sutter, *More Exceptional C++*, Addison-Wesley 2002 - Items 13–16

- wikipedia:Copy-on-write

Counted Body/Reference Counting (intrusive)

Intent

Manage logical sharing of a resource/object, prevent expensive copying, and allow proper resource deallocation of objects that use dynamically allocated resources.

Also Known As

- Reference Counting (intrusive)
- Counted Body

Motivation

When Handle/Body idiom is used, quite often it is noticed that copying of bodies is expensive. This is because copying of bodies involves allocation of memory and copy construction. Copying can be avoided by using pointers and references, but these leave the problem of who is responsible for cleaning up the object. Some handle must be responsible for releasing memory resources allocated for the body. Usually it is the last one. Without automatic reclamation of memory resources of memory, it leaves a user-visible distinction between built-in types and user-defined types.

Solution and Sample Code

The solution is to add a reference count to the body class to facilitate memory management; hence the name "Counted Body." Memory management is added to the handle class, particularly to its implementation of initialization, assignment, copying, and destruction.

```
#include <cstring>
#include <algorithm>
#include <iostream>

class StringRep {
  friend class String;

  friend std::ostream &operator<<(std::ostream &out, StringRep const &str) {
    out << "[" << str.data_ << ", " << str.count_ << "]";
    return out;
  }

 public:
  StringRep(const char *s) : count_(1) {
    strcpy(data_ = new char[strlen(s) + 1], s);
  }

  ~StringRep() { delete[] data_; }

 private:
  int count_;
  char *data_;
};

class String {
 public:
  String() : rep(new StringRep("")) {
    std::cout << "empty ctor: " << *rep << "\n";
  }
  String(const String &s) : rep(s.rep) {
    rep->count_++;
    std::cout << "String ctor: " << *rep << "\n";
  }
  String(const char *s) : rep(new StringRep(s)) {
    std::cout << "char ctor:" << *rep << "\n";
  }
  String &operator=(const String &s) {
    std::cout << "before assign: " << *s.rep << " to " << *rep << "\n";
    String(s).swap(*this);  // copy-and-swap idiom
    std::cout << "after assign: " << *s.rep << " , " << *rep << "\n";
    return *this;
  }
```

```
  ~String() {  // StringRep deleted only when the last handle goes out of scope.
    if (rep && --rep->count_ <= 0) {
      std::cout << "dtor: " << *rep << "\n";
      delete rep;
    }
  }

 private:
  void swap(String &s) throw() { std::swap(this->rep, s.rep); }

  StringRep *rep;
};
int main() {

  std::cout << "*** init String a with empty\n";
  String a;
  std::cout << "\n*** assign a to \"A\"\n";
  a = "A";

  std::cout << "\n*** init String b with \"B\"\n";
  String b = "B";

  std::cout << "\n*** b->a\n";
  a = b;

  std::cout << "\n*** init c with a\n";
  String c(a);

  std::cout << "\n*** init d with \"D\"\n";
  String d("D");

  return 0;
}
```

Gratuitous copying is avoided, leading to a more efficient implementation. This idiom presumes that the programmer can edit the source code for the body. When that's not possible, use Detached Counted Body. When counter is stored inside body class, it is called intrusive reference counting and when the counter is stored external to the body class it is known as non-intrusive reference counting. This implementation is a variation of shallow copy with the semantics of deep copy and the efficiency of Smalltalk name-value pairs.

The output should be as below:

```
*** init String a with empty
empty ctor: [, 1]

*** assign a to "A"
char ctor:[A, 1]
before assign: [A, 1] to [, 1]
String ctor: [A, 2]
dtor: [, 0]
after assign: [A, 2] , [A, 2]

*** init String b with "B"
char ctor:[B, 1]

*** b->a
before assign: [B, 1] to [A, 1]
String ctor: [B, 2]
dtor: [A, 0]
after assign: [B, 2] , [B, 2]

*** init c with a
String ctor: [B, 3]

*** init d with "D"
char ctor:[D, 1]
dtor: [D, 0]
dtor: [B, 0]
```

Consequences

Creation of multiple reference counts will result into multiple deletion of the same body, which is undefined. Care must be taken to avoid creating multiple reference counts to the same body. Intrusive reference counting easily supports it. With non-intrusive reference counting, programmer discipline is required to prevent duplicate reference counts.

Known Uses

- boost::shared_ptr (non-intrusive reference counting)
- boost::intrusive_ptr (intrusive reference counting)
- std::shared_ptr
- the Qt toolkit, e.g. QString

Related Idioms

- Handle Body
- Detached Counted Body (non-intrusive reference counting)
- Smart Pointer
- Copy-and-swap

References

C++ Idioms by James O. Coplien (http://web.archive.org/web/20101024134005/http://users.rcn.com/jcoplien/Patterns/C++Idioms/EuroPLoP98.html#CountedBody)

Curiously Recurring Template Pattern

Intent

Specialize a base class using the derived class as a template argument.

Also Known As

- CRTP
- Mixin-from-above

Motivation

To extract out a type independent but type customizable functionality in a base class and to mix-in that interface/property/behavior into a derived class, customized for the derived class.

Solution and Sample Code

In CRTP idiom, a class T, inherits from a template that specializes on T.

```
class T : public X<T> {…};
```

This is valid only if the size of X<T> can be determined independently of T. Typically, the base class template will take advantage of the fact that member function bodies (definitions) are not instantiated until long after their declarations, and will use members of the derived class within its own member functions, via the use of a `static_cast`, e.g.:

```
template <class Derived>
struct base
{
    void interface()
    {
        // ...
        static_cast<Derived*>(this)->implementation();
        // ...
    }

    static void static_interface()
    {
        // ...
        Derived::static_implementation();
        // ...
    }

    // The default implementation may be (if exists) or should be (otherwise)
    // overridden by inheriting in derived classes (see below)
    void implementation();
    static void static_implementation();
};

// The Curiously Recurring Template Pattern (CRTP)
struct derived_1 : base<derived_1>
{
    // This class uses base variant of implementation
    //void implementation();

    // ... and overrides static_implementation
    static void static_implementation();
};

struct derived_2 : base<derived_2>
{
    // This class overrides implementation
    void implementation();

    // ... and uses base variant of static_implementation
    //static void static_implementation();
};
```

Known Uses

Barton-Nackman trick

Related Idioms

- Parameterized Base Class Idiom
- Barton-Nackman trick

References

Curiously Recurring Template Pattern on Wikipedia (http://en.wikipedia.org/wiki/Curiously_Recurring_Template_Pattern)More C++ Idioms/Detached Counted Body

Empty Base Optimization

Intent

Optimize storage for data members of empty class types

Also Known As

- EBCO: Empty Base Class Optimization

- Empty Member Optimization

Motivation

Empty classes come up from time to time in C++. C++ requires empty classes to have non-zero size to ensure object identity. For instance, an array of `EmptyClass` below has to have non-zero size because each object identified by the array subscript must be unique. Pointer arithmetic will fall apart if `sizeof(EmptyClass)` is zero. Often the size of such a class is one.

```
class EmptyClass {};
EmptyClass arr[10]; // Size of this array can't be zero.
```

When the same empty class shows up as a data member of other classes, it consumes more than a single byte. Compilers often align data on 4-byte boundaries to avoid splitting. The four bytes taken by the empty class object are just placeholders and serve no useful purpose. Avoiding wastage of space is desirable to save memory and help fit more objects in the cpu cache lines.

Solution and Sample Code

C++ makes special exemption for empty classes when they are inherited from. The compiler is allowed to flatten the inheritance hierarchy in a way that the empty base class does not consume space. For instance, in the following example, `sizeof(AnInt)` is 4 on 32 bit architectures and `sizeof(AnotherEmpty)` is 1 byte even though both of them inherit from the `EmptyClass`

```
class AnInt  : public EmptyClass
{
   int data;
};   // size = sizeof(int)

class AnotherEmpty : public EmptyClass {};  // size =  1
```

EBCO makes use of this exemption in a **systematic** way. It may not be desirable to naively move the empty classes from member-list to base-class-list because that may expose interfaces that are otherwise hidden from the users. For instance, the following way of applying EBCO will apply the optimization but may have undesirable side-effects: The signatures of the functions (if any in E1, E2) are now visible to the users of class `Foo` (although they can't call them because of private inheritance).

```
class E1 {};
class E2 {};

// **** before EBCO ****

class Foo {
  E1 e1;
  E2 e2;
  int data;
}; // sizeof(Foo) = 8

// **** after EBCO ****

class Foo : private E1, private E2 {
  int data;
}; // sizeof(Foo) = 4
```

A practical way of using EBCO is to combine the empty members into a single member that flattens the storage. The following template `BaseOptimization` applies EBCO on its first two type parameter. The `Foo` class above has been rewritten to use it.

```
template <class Base1, class Base2, class Member>
struct BaseOptimization : Base1, Base2
{
   Member member;
   BaseOptimization() {}
   BaseOptimization(Base1 const& b1, Base2 const & b2, Member const& mem)
       : Base1(b1), Base2(b2), member(mem) { }
};

class Foo {
  BaseOptimization<E1, E2, int> data;
}; // sizeof(Foo) = 4
```

With this technique, there is no change in the inheritance relationship of the `Foo` class. And, because `BaseOptimization` declares no member functions, it also avoids the problem of accidentally overriding a function from the base classes. Note that in the approach shown above it is critical that the base classes do not conflict with each other. That is, `Base1` and `Base2` are part of independent hierarchies.

Caveat

Object identity issues do not appear to be consistent across compilers. The addresses of the empty objects may or may not be the same. For instance, consider the below.

```
template <class Base1, class Base2, class Member>
struct BaseOptimizationWithFunctions : Base1, Base2
{
   Member member;
   BaseOptimizationWithFunctions() {}
   BaseOptimizationWithFunctions(Base1 const& b1, Base2 const & b2, Member const& mem)
       : Base1(b1), Base2(b2), member(mem) { }
   Base1 * first()  { return this; }
   Base2 * second() { return this; }
};
```

The pointers returned by the `first` and `second` member functions may be the same on some compilers and different on others. See more discussion on StackOverflow (http://stackoverflow.com/questions/7694158/boost-compressed-pair-and-addresses-of-empty-objects)

Known Uses

- boost::compressed_pair (http://www.boost.org/doc/libs/1_47_0/libs/utility/compressed_pair.htm) makes use of this technique to optimize the size of the pair.
- A C++03 emulation of unique_ptr (http://home.roadrunner.com/~hinnant/unique_ptr03.html) also uses this idiom.

Related Idioms

References

- The Empty Base Class Optimization (EBCO) (http://www.informit.com/articles/article.aspx?p=31473&seqNum=2)
- The "Empty Member" C++ Optimization (http://www.cantrip.org/emptyopt.html)
- Internals of boost::compressed_pair (http://www.ibm.com/developerworks/aix/library/au-boostutilities/index.html)More C++ Idioms/Emulated Exception

enable-if

Intent

Allow function overloading based on arbitrary properties of type

Also Known As

- Explicit Overload Set Management

Motivation

The **enable_if** family of templates is a set of tools to allow a function template or a class template specialization to include or exclude itself from a set of matching functions or specializations based on properties of its template arguments. For example, one can define function templates that are only enabled for, and thus only match, an arbitrary set of types defined by a traits class. The **enable_if** templates can also be applied to enable class template specializations. Applications of **enable_if** are discussed in length in the literature.[1] [2]

Sensible operation of template function overloading in C++ relies on the SFINAE (substitution-failure-is-not-an-error) principle:[3] if an invalid argument or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set instead of causing a compilation error. The following example[1] demonstrates why this is important:

```
int negate(int i) { return -i; }

template <class F>
typename F::result_type negate(const F& f) { return -f(); }
```

Suppose the compiler encounters the call *negate(1)*. The first definition is obviously a better match, but the compiler must nevertheless consider (and instantiate the prototypes) of both definitions to find this out. Instantiating the latter definition with *F* as *int* would result in:

```
int::result_type negate(const int&);
```

where the return type is invalid. If this was an error, adding an unrelated function template (that was never called) could break otherwise valid code. Due to the SFINAE principle the above example is not, however, erroneous. The latter definition of negate is simply removed from the overload resolution set.

The **enable_if** templates are tools for controlled creation of the SFINAE conditions.

Solution and Sample code

The enable_if templates are very simple syntactically. They always come in pairs: one of them is empty and the other one has a `type` typedef that forwards its second type parameter. The empty structure triggers an invalid type because it contains no member. When a compile-time condition is false, the empty `enable_if` template is chosen. Appending `::type` would result in an invalid instantiation, which the compiler throws away due to the SFINAE principle.

```
template <bool, class T = void>
struct enable_if
{};

template <class T>
struct enable_if<true, T>
{
  typedef T type;
};
```

Here is an example that shows how an overloaded template function can be selected at compile-time based on arbitrary properties of the type parameter. Imagine that the function **T foo(T t)** is defined for all types such that T is arithmetic. The enable_if template can be used either as the return type, as in this example:

```
template <class T>
typename enable_if<is_arithmetic<T>::value, T>::type
foo(T t)
{
  // ...
  return t;
}
```

or as an extra argument, as in the following:

```
template <class T>
T foo(T t, typename enable_if<is_arithmetic<T>::value >::type* dummy = 0);
```

The extra argument added to foo() is given a default value. Since the caller of foo() will ignore this dummy argument, it can be given any type. In particular, we can allow it to be void *. With this in mind, we can simply omit the second template argument to **enable_if**, which means that the **enable_if<...>::type** expression will evaluate to *void* when **is_arithmetic<T>** is true.

Whether to write the enabler as an argument or within the return type is largely a matter of taste, but for certain functions, only one alternative is possible:

- Operators have a fixed number of arguments, thus **enable_if** must be used in the return type.
- Constructors and destructors do not have a return type; an extra argument is the only option.
- There does not seem to be a way to specify an enabler for a conversion operator. Converting constructors, however, can have enablers as extra default arguments.

Known Uses

Boost library, C++ STL, etc.

Related Idioms

- SFINAE

References

1. Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. C/C++ Users Journal, 21(6):25--32, June 2003.
2. Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In Frank Pfennig and Yannis Smaragdakis, editors, Generative Programming and Component Engineering, volume 2830 of LNCS, pages 228--244. Springer Verlag, September 2003.
3. David Vandevoorde and Nicolai M. Josuttis. C++ Templates: The Complete Guide. Addison-Wesley, 2002.

External links

Boost Libraries enable-if documentation (http://www.boost.org/libs/utility/enable_if.html)More C++ Idioms/Envelope Letter

Erase-Remove

Intent

To eliminate elements from a STL container to reduce the size of it.

Also Known As

Remove-Erase (based on the order of execution, rather than the typed order on the source code line)

Motivation

std::remove algorithm does not eliminate elements from the container! It simply moves the elements not being removed to the front of the container, leaving the contents at the end of the container undefined. This is because std::remove algorithm works only using a pair of forward iterators (Iterator Pair idiom) and generic concept of forward iterators does not know how to eliminate data elements from an arbitrary data structure. Only container member functions can eliminate container elements as only members know the details of internal data structure. Erase-Remove idiom is used to really eliminate data elements from a container.

Solution and Sample Code

std::remove algorithm returns an iterator to the beginning of the range of "unused" elements. It does not change the end() iterator of the container nor does the size of it. Member erase function can be used to really eliminate the members from the container in the following idiomatic way.

```
template<typename T>
inline void remove(std::vector<T> & v, const T & item)
{
    v.erase(std::remove(v.begin(), v.end(), item), v.end());
}


std::vector<int> v;
// fill it up somehow
remove(v, 99);// really remove all elements with value 99
```

The order of evaluation of v.end() and invocation of std::remove is unimportant here because std::remove algorithm does not change end() iterator in any way.

Known Uses

http://www.boost.org/doc/libs/1_43_0/libs/range/doc/html/range/reference/algorithms/new/remove_erase.html

Related Idioms

References

Effective STL, Item 32 - Scott MeyersMore C++ Idioms/Examplar

Execute-Around Pointer

Intent

Provide a smart pointer object that transparently executes actions before and after each function call on an object, given that the actions performed are the same for all functions.[1] This can be regarded as a special form of aspect oriented programming(AOP).

Also Known As

Double application of smart pointer.

Motivation

Often times it is necessary to execute a functionality before and after every member function call of a class. For example, in a multi-threaded application it is necessary to lock before modifying the data structure and unlock it afterwards. In a data structure visualization application might be interested in the size of the data structure after every insert/delete operation.

```
using namespace std;
class Visualizer {
    std::vector <int> & vect;
  public:
    Visualizer (vector<int> &v) : vect(v) {}
    void data_changed () {
       std::cout << "Now size is: " << vect.size();
    }
};
int main () // A data visualization application.
{
  std::vector <int> vector;
  Visualizer visu (vector);
  //...
  vector.push_back (10);
  visu.data_changed ();
  vector.push_back (20);
  visu.data_changed ();
  // Many more insert/remove calls here
  // and corresponding calls to visualizer.
}
```

Such a repetition of function calls is error-prone and tedious. It would be ideal if calls to visualizer could be automated. Visualizer could be used for std::list <int> as well. Such funcitonality which is not a part of single class but rather cross cuts multiple classes is commonly known as aspects. This particular idiom is useful for designing and implementing simple aspects.

Solution and Sample Code

```
class VisualizableVector {
  public:
    class proxy {
      public:
        proxy (vector<int> *v) : vect (v) {
          std::cout << "Before size is: " << vect->size ();
        }
        vector<int> * operator -> () {
          return vect;
        }
        ~proxy () {
          std::cout << "After size is: " << vect->size ();
        }
      private:
        vector <int> * vect;
    };
    VisualizableVector (vector<int> *v) : vect(v) {}
    proxy operator -> () {
       return proxy (vect);
    }
  private:
    vector <int> * vect;
};
int main()
{
  VisualizableVector vecc (new vector<int>);
  //...
  vecc->push_back (10); // Note use of -> operator instead of . operator
```

```
  vecc->push_back (20);
}
```

Overloaded -> operator of visualizableVector creates a temporary proxy object and it is returned. Constructor of proxy object logs size of the vector. The overloaded -> operator of proxy is then called and it simply forwards the call to the underlying vector object by returning a raw pointer to it. After the real call to the vector finishes, destructor of proxy logs the size again. Thus the logging for visualization is transparent and the main function becomes free from clutter. This idiom is a special case of Execute Around Proxy, which is more general and powerful.

The real power of the idiom can be derived if we combine it judiciously with templates and chain the overloaded -> operators.

```
template <class NextAspect, class Para>
class Aspect
{
  protected:
    Aspect (Para p): para_(p) {}
    Para  para_;
  public:
    NextAspect operator -> ()
    {
      return NextAspect (para_);
    }
};

template <class NextAspect, class Para>
struct Visualizing : Aspect <NextAspect, Para>
{
  public:
    Visualizing (Para p)
       : Aspect <NextAspect, Para> (p)
    {
    std::cout << "Before Visualization aspect" << std::endl;
    }
    ~Visualizing ()
    {
    std::cout << "After Visualization aspect" << std::endl;
    }
};
template <class NextAspect, class Para>
struct Locking : Aspect <NextAspect, Para>
{
  public:
    Locking (Para p)
       : Aspect <NextAspect, Para> (p)
    {
        std::cout << "Before Lock aspect" << std::endl;
    }
    ~Locking ()
    {
    std::cout << "After Lock aspect" << std::endl;
    }
};
template <class NextAspect, class Para>
struct Logging : Aspect <NextAspect, Para>
{
  public:
    Logging (Para p)
        : Aspect <NextAspect, Para> (p)
    {
        std::cout << "Before Log aspect" << std::endl;
    }
    ~Logging ()
    {
    std::cout << "After Log aspect" << std::endl;
    }
};
template <class Aspect, class Para>
class AspectWeaver
{
public:
    AspectWeaver (Para p) : para_(p) {}
    Aspect operator -> ()
```

```
    {
       return Aspect (para_);
    }
private:
    Para para_;
};

#define AW1(T,U) AspectWeaver <T <U, U>, U >
#define AW2(T,U,V) AspectWeaver <T < U <V, V> , V>, V >
#define AW3(T,U,V,X) AspectWeaver <T < U <V <X, X>, X> , X>, X >

int main()
{
  AW3(Visualizing, Locking, Logging, vector <int> *)
      X (new vector<int>);
  //...
  X->push_back (10); // Note use of -> operator instead of . operator
  X->push_back (20);
  return 0;
}
```

Known Uses

Related Idioms

Smart Pointer

References

1. Execute Around Sequences - Kevlin Henney

Exploding Return Type

Intent

An idiom to return either by error code or by throwing an exception.

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

References

- Choose Your Poison (http://accu.org/content/conf2007/Alexandrescu-Choose_Your_Poison.pdf)
- Overload no. 53 (http://accu.org/var/uploads/journals/overload53-FINAL.pdf)More C++ Idioms/Export Guard Macro

Expression Template

Intent

- To create a domain-specific embedded language (DSEL) in C++
- To support *lazy evaluation* of C++ expressions (e.g., mathematical expressions), which can be executed much later in the program from the point of their definition.
- To pass an expression -- not the result of the expression -- as a parameter to a function.

Also Known As

Motivation

Domain-specific languages (DSLs) is a way of developing programs where the problem to be solved is expressed using notation that is much closer to the domain of the problem rather than the usual notation (loops, conditionals, etc.) provided by procedural languages. Domain-specific embedded languages (DSELs) is a special case of DSLs where the notation is **embedded** in a host language (e.g., C++). Two prominent examples of DSELs based on C++ are the Boost Spirit Parser Framework (http://spirit.sourceforge.net) and Blitz++ (http://www.oonumerics.org/blitz) scientific computing library. Spirit provides a notation to write EBNF grammar directly into a C++ program whereas Blitz++ allows a notation to perform mathematical operations on matrices. Obviously, such notation is not provided in C++ natively. The key benefit of using such notation is that the program captures the intention of the programmer quite intuitively making the program much more readable. It reduces the development as well as maintenance costs **dramatically**.

So, how do these libraries (Spirit and Blitz++) achieve such a leap in the abstraction-level? The answer is -- you guessed it right -- **Expression Templates**.

The key idea behind expression templates is **lazy evaluation** of expressions. C++ does not support lazy evaluation of expressions natively. For example, in the code snippet below, the addition expression (x+x+x) is executed before the function foo is called.

```
int x;
foo(x + x + x); // The addition expression does not exist beyond this line.
```

Function foo never really knows how the parameter it receives is computed. The addition expression never really exists after its first and the only evaluation. This default behavior of C++ is necessary and sufficient for an overwhelmingly large number of real-world programs. However, some programs need the expression later on to evaluate it again and again. For example, tripling every integer in an array.

```
int expression (int x)
{
  return x + x + x; // Note the same expression.
}
// .... Lot of other code here
const int N = 5;
double A[N] = { 1, 2, 3, 4, 5};

std::transform(A, A+N, A, std::ptr_fun(expression)); // Triples every integer.
```

This is the conventional way of supporting lazy evaluation of mathematical expressions in C/C++. The expression is wrapped in a function and the function is passed around as a parameter. There is overhead of function calls and creation of temporaries in this technique, and quite often, the location of the expression in the source code is quite far from the call site, which adversely affects the readability and maintainability. Expression templates solve the problem by inlining the expression, which eliminates the need for a function pointer and brings together the expression and the call site.

Solution and Sample Code

Expression templates use the Recursive Type Composition idiom. Recursive type composition uses instances of class templates that contain other instances of the same template as member variables. Multiple repetitive instantiation of the same template gives rise to an abstract syntax tree (AST) of types. Recursive type composition has been used to create linear Type lists as well as binary expression trees used in the following example.

```
#include <iostream>
#include <vector>

struct Var {
  double operator () (double v) { return v; }
};

struct Constant {
  double c;
  Constant (double d) : c (d) {}
  double operator () (double) { return c; }
};

template < class L, class H, class OP >
struct DBinaryExpression {
  L l_;
  H h_;
  DBinaryExpression (L l, H h) : l_ (l), h_ (h) {}
  double operator () (double d) { return OP::apply (l_ (d), h_(d)); }
};

struct Add {
  static double apply (double l, double h) { return l + h; }
};

template < class E >
struct DExpression {
  E expr_;
  DExpression (E e) : expr_ (e) {}
  double operator() (double d) { return expr_(d);  }
};

template < class Itr, class Func >
void evaluate (Itr begin, Itr end, Func func)
{
  for (Itr i = begin; i != end; ++i)
    std::cout << func (*i) << std::endl;
}

int main (void)
{
  typedef DExpression <Var> Variable;
  typedef DExpression <Constant> Literal;
  typedef DBinaryExpression <Variable , Literal, Add> VarLitAdder;
  typedef DExpression <VarLitAdder> MyAdder;

  Variable x ((Var()));
  Literal l (Constant (50.00));
  VarLitAdder vl_adder(x, l);
  MyAdder expr (vl_adder);

  std::vector <double> a;
  a.push_back (10);
  a.push_back (20);

  // It is (50.00 + x) but does not look like it.
  evaluate (a.begin(), a.end(), expr);

  return 0;
}
```

An analogy to the Composite design pattern is useful here. The template DExpression can be considered as the abstract base class in the Composite pattern. It captures the commonality in the interface. In expression templates, the common interface is the overloaded function call operator. DBinaryExpression is a real composite as well as an adaptor, which adapts Add's interface to that of DExpression. Constant and Var are two

different types of leaf nodes. They also stick to the DExpression's interface. DExpression hides the complexity of DBinaryExpression, Constant and Var behind a unified interface to make them work together. Any binary operator can take place of Add, for example Divide, Multiply etc.

The above example does not show how recursive types are generated at compile-time. Also, expr does not look like a mathematical expression at all, but it is indeed one. The code that follows show how types are recursively composed using repetitive instantiation of the following overloaded + operator.

```
template< class A, class B >
DExpression<DBinaryExpression<DExpression<A>, DExpression<B>, Add> >
operator + (DExpression<A> a, DExpression<B> b)
{
  typedef DBinaryExpression <DExpression<A>, DExpression<B>, Add> ExprT;
  return DExpression<ExprT>(ExprT(a,b));
}
```

The above overloaded operator+ does two things - it adds syntactic sugar and enables recursive type composition, bounded by the compiler's limits. It can therefore be used to replace the call to evaluate as follows:

```
evaluate (a.begin(), a.end(), x + 1 + x);
/// It is (2*x + 50.00), which does look like a mathematical expression.
```

Known Uses

- Blitz++ Library (http://www.oonumerics.org/blitz)
- Boost Spirit Parser Framework (http://spirit.sourceforge.net)
- Boost Basic Linear Algebra (http://www.boost.org/libs/numeric/ublas)
- LEESA: Native XML Programming using an Embedded Query and Traversal Language in C++ (http://www.dre.vanderbilt.edu/LEESA)
- *std::valarray* as implemented by GNU libstdc++ (http://gcc.gnu.org/viewcvs/trunk/libstdc%2B%2B-v3/include/std/valarray?view=co), LLVM libc++ (http://llvm.org/svn/llvm-project/libcxx/trunk/include/valarray), and others.

Related Idioms

- Recursive Type Composition

References

- Expression Templates (http://www.angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm) - Klaus Kreft & Angelika Langer
- Expression Templates (http://web.archive.org/web/20050315091836/http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html) - Todd Veldhuizen
- Faster Vector Math Using Templates (http://www.flipcode.com/archives/Faster_Vector_Math_Using_Templates.shtml) - Tomas Arce

Intent

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

References

Intent

Increase performance of Handle Body idiom.

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

- Handle Body (aka pimpl)

References

The Fast Pimpl Idiom (http://www.gotw.ca/gotw/028.htm)

Final Class

Intent

- Partially simulate the *final* class feature found in other languages.
- Partially prevent a class from further subclassing or inheritance.

Also Known As

Motivation

Class designers may want to enforce that a specific class can not be further extended or subclassed by the user of the class. Other object-oriented languages such as Java and C# provide this feature to the class designers. In Java the keyword is called *final* whereas in C#, it is called *sealed*. Final class idiom is a way of partially simulating this effect in C++.

Solution and Sample Code

Final class idiom makes use of virtual inheritance and a friend class to create the effect of a *final* class. The idiom depends on the following C++ rule: the constructor (and destructor) of a virtually inherited class is called directly by the derived-most class. If access to the constructor or destructor of such a virtually inherited class is prevented, the class can not be subclassed further.

```
class MakeFinal
{
  MakeFinal() {}  // private by default.
  friend class sealed;
};

class sealed : virtual MakeFinal
{ };

class test : public sealed
{ };

int main (void)
{
  test t; // Compilation error here.
}
```

In the above example, the *test* class inherits from the *sealed* class and the *main* function tries to instantiate an object of type *test*. The instantiation fails because the *test* class can not access the private destructor of the *MakeFinal* class because it is defined private and inherited virtually. However, friendship is not inheritable and therefore, objects of type *test* can't be created.

Note that the said error occurs only when the *test* class is instantiated. This behavior is different from how final classes behave in Java and C#. In fact, this idiom does not prevent inheritance of static methods defined in the *sealed* class. As long as the *test* class is not instantiated and it accesses only the static members of the *sealed* class, compiler does not complain.

C++11

The C++11 standard provides the *final* modifier, which can be used to prevent a class from being subclassed.

```
class Base final { };

class test : public Base { };  // incorrect
```

Known Uses

Related Idioms

References

Free Function Allocators

Intent

Allow containers to use custom allocators without creating a new type

Also Known As

Motivation

C++ standard allocators have serious problems, because they change the underlying type of the container.

Solution and Sample Code

This idiom is superior to the way that std::allocators work the whole idiom is outlined here.

```
struct user_allocator_nedmalloc
{
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;

    static inline char* malloc(const size_type bytes) {
        return reinterpret_cast<char*>(nedmalloc(bytes));
    }

    static inline void free(char* const block) {
        nedfree(block);
    }
};
```

Known Uses

- boost::ptr_container

Related Idioms

References

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1850.pdf

Attorney-Client

Intent

Control the granularity of access to the implementation details of a class

Motivation

A `friend` declaration in C++ gives complete access to the internals of a class. Friend declarations are, therefore, frowned upon because they break carefully crafted encapsulations. Friendship feature of C++ does not provide any way to selectively grant access to a subset of private members of a class. Friendship in C++ is an all-or-nothing proposition. For instance, the following class `Foo` declares class `Bar` its friend. Class `Bar` therefore has access to **all** the private members of class `Foo`. This may not be desirable because it increases coupling. Class `Bar` cannot be distributed without class `Foo`.

```
class Foo
{
private:
  void A(int a);
  void B(float b);
  void C(double c);
  friend class Bar;
};

class Bar {
// This class needs access to Foo::A and Foo::B only.
```

```
// C++ friendship rules, however, give access to all the private members of Foo.
};
```

Providing selective access to a subset of members is desirable because the remaining (private) members can change interface if needed. It helps reduce coupling between classes. Attorney-Client idiom allows a class to precisely control the amount of access they give to their friends.

Solution and Sample Code

Attorney-client idiom works by adding a level of indirection. A *client* class that wants to control access to its internal details, appoints an *attorney* and makes it a friend --- a C++ friend! The `Attorney` class is crafted carefully to serve as a proxy to the `Client`. Unlike a typical proxy class, `Attorney` class replicates only a subset of `Client`'s private interface. For instance, consider class `Foo` wants to control access to its implementation details. For better clarity we rename it as `Client`. `Client` wants its `Attorney` to provide access to `Client::A` and `Client::B` only.

```
class Client
{
private:
  void A(int a);
  void B(float b);
  void C(double c);
  friend class Attorney;
};

class Attorney {
private:
  static void callA(Client & c, int a) {
    c.A(a);
  }
  static void callB(Client & c, float b) {
    c.B(b);
  }
  friend class Bar;
};

class Bar {
// Bar now has access to only Client::A and Client::B through the Attorney.
};
```

The `Attorney` class restricts access to a cohesive set of functions. The `Attorney` class has all inline static member functions, each taking a reference to an instance of the `Client` and forwarding the function calls to it. Some things are idiomatic about the `Attorney` class. Its implementation is entirely private, which prevents other unexpected classes gaining access to the internal of `Client`. The `Attorney` class determines which other classes, member functions, or free functions get access to it. It declares them as `friend` to allow access to its implementation and eventually the `Client`. Without the `Attorney` class, `Client` class would have declared the same set of friends giving them unrestrained access to the internals of `Client`.

It is possible to have multiple attorney classes providing access to different sets of implementation details of the client. For instance, class `AttorneyC` may provide access to the `Client::C` member function only. An interesting case emerges where an attorney class serves as a mediator for several different classes and provides cohesive access to their implementation details. Such a design is conceivable in case of inheritance hierarchies because friendship in C++ is not inheritable, but private virtual function overrides in derived classes can be called if base's private virtual functions are accessible. In the following example, the Attorney-Client idiom is applied to class `Base` and the `main` function. The `Derived::Func` function gets called via polymorphism. To access the implementation details of `Derived` class, however, the same idiom may be applied.

```
#include <cstdio>

class Base {
private:
```

```
  virtual void Func(int x) = 0;
  friend class Attorney;
public:
  virtual ~Base() {}
};

class Derived : public Base {
private:
  virtual void Func(int x)  {
    printf("Derived::Func\n"); // This is called even though main is not a friend of Derived.
  }

public:
  ~Derived() {}
};

class Attorney {
private:
  static void callFunc(Base & b, int x) {
    return b.Func(x);
  }
  friend int main (void);
};

int main(void) {
  Derived d;
  Attorney::callFunc(d, 10);
}
```

Known Uses

- Boost.Iterators library (http://www.boost.org/doc/libs/1_50_0/libs/iterator/doc/iterator_facade.html#iterator-core-access)
- Boost.Serialization: class boost::serialization::access (http://www.boost.org/doc/libs/1_50_0/libs/serialization/doc/serialization.html#member)

Related Idioms

References

Friendship and the Attorney-Client Idiom (Dr. Dobb's Journal) (http://drdobbs.com/184402053)More C++ Idioms/Function Object

Generic Container Idioms

Intent

To create generic container classes (vector, list, stack) that impose minimal requirements on their value types. The requirements being only a copy-constructor and a non-throwing destructor.

Motivation

Developing generic containers in C++ can become complex if truly generic containers (like STL) are desired. Relaxing the requirements on type T is the key behind developing truly generic containers. There are a few C++ idioms to actually achieve the "lowest denominator" possible with requirements on type T.

Lets take an example of a Stack.

```
template<class T>
class Stack
{
    int size_;
```

```
    T * array_;
    int top_;
  public:
    Stack (int size=10)
      : size_(size),
        array_ (new T [size]), // T must support default construction
        top_(0)
    { }
    void push (const T & value)
    {
      array_[top_++] = value; // T must support assignment operator.
    }
    T pop ()
    {
      return array_[--top_]; // T must support copy-construction. No destructor is called here
    }
    ~Stack () throw() { delete [] array_; } // T must support non-throwing destructor
};
```

Other than some array bounds problem, above implementation looks pretty obvious. But it is quite naive. It has more requirements on type T than there needs to be. The above implementation requires following operations defined on type T:

- A default constructor for T
- A copy constructor for T
- A non-throwing destructor for T
- A copy assignment operator for T

A stack ideally, should not construct more objects in it than number of push operations performed on it. Similarly, after every pop operation, an object from stack should be popped out and destroyed. Above implementation does none of that. One of the reasons is that it uses a default constructor of type T, which is totally unnecessary.

Actually, the requirements on type T can be reduced to the following using *construct* and *destroy* generic container idioms.

- A copy constructor
- A non-throwing destructor.

Solution and Sample Code

To achieve this, a generic container should be able to allocate uninitialized memory and invoke constructor(s) only once on each element while "initializing" them. This is possible using following three generic container idioms:

```
#include <algorithm>
// construct helper using placement new:
template <class T1, class T2>
void construct (T1 &p, const T2 &value)
{
  new (&p) T1(value); // T must support copy-constructor
}

// destroy helper to invoke destructor explicitly.
template <class T>
void destroy (T const &t)  throw ()
{
  t.~T(); // T must support non-throwing destructor
}

template<class T>
class Stack
{
    int size_;
    T * array_;
    int top_;
  public:
```

```
    Stack (int size=10)
      : size_(size),
        array_ (static_cast <T *>(::operator new (sizeof (T) * size))), // T need not support default construction
        top_(0)
    { }
    void push (const T & value)
    {
      construct (array_[top_++], value); // T need not support assignment operator.
    }
    T top ()
    {
       return array_[top_ - 1]; // T should support copy construction
    }
    void pop()
    {
      destroy (array_[--top_]);     // T destroyed
    }
    ~Stack () throw()
    {
      std::for_each(array_, array_ + top_, destroy<T>);
      ::operator delete(array_); // Global scope operator delete.
    }
};
class X
{
  public:
    X (int) {} // No default constructor for X.
  private:
    X & operator = (const X &); // assignment operator is private
};
int main (void)
{
  Stack <X> s; // X works with Stack!

  return 0;
}
```

operator new allocates uninitialized memory. It is a fancy way of calling malloc. The construct helper template function invokes placement new and in turn invokes a copy constructor on the initialized memory. The pointer p is supposed to be one of the uninitialized memory chunks allocated using operator new. If end is an iterator pointing at an element one past the last initialized element of the container, then pointers in the range end to end_of_allocation should not point to objects of type T, but to uninitialized memory. When an element is removed from the container, destructor should be invoked on them. A destroy helper function can be helpful here as shown. Similarly, to delete a range, another overloaded destroy function which takes two iterators could be useful. It essentially invokes first destroy helper on each element in the sequence.

Known Uses

All STL containers employ similar techniques. They have minimal possible requirements on the template parameter types. On the other hand, some popular C++ libraries have stricter requirements on parameterizable types than necessary.

Related Idioms

There are several other generic container idioms.

- Non-throwing swap
- Copy-and-swap
- Iterator Pair
- Coercion by Member Template
- Making New Friends

References

Designing Exception Safe Generic Containers (http://portal.acm.org/citation.cfm?id=331173) -- Herb Sutter

Include Guard Macro

Intent

To allow inclusion of a header file multiple times.

Also Known As

Motivation

Including the same header file more than once in the same compilation unit can lead to the violation of a basic rule of C++: One Definition Rule (ODR). A header may get included multiple times because of direct and indirect inclusion.

Solution and Sample Code

Include Guard macro idiom is an old idiom, which is also applicable in a C program. It uses simple #define to allow the inclusion of a header file multiple times in a compilation unit. The idiom ensures that after preprocessing the guarded content of the header file is only seen once by the compiler. More precisely, the compiler gets to see the guarded content where the header file gets included for the very first time. The following macros are put at the very beginning and at very end of a header file:

```
#ifndef MYHEADER_H_ // beginning
#define MYHEADER_H_
...
#endif // MYHEADER_H_ // end
```

Some compilers support

```
#pragma once
```

as an efficient alternative to include guards. It does not require to open the header file more than once, unlike traditional include guard macro in some compilers. On many modern compilers like GCC4 or MSC++2008 #pragma once will not give better compile time performance as they recognize header guards.

Known Uses

Virtually all header files in the world!

Related Idioms

- Inline Guard Macro
- Export Guard Macro

References

#pragma once (http://en.wikipedia.org/wiki/Pragma_once) in Wikipedia.

Inline Guard Macro

Intent

To conveniently control inline-ness of functions using a compiler command line macro definition switch.

Also Known As

Motivation

For debugging purpose, it is often necessary to turn off inlining of functions throughout the program. But for release version, inline functions are desirable. This indicates a need of a quick way of turning inline-ness on/off as and when desired. Moreover, such functions should be defined in header files when they are inlined and otherwise should be in the source (.cpp) file. If non-inline functions are in header files, almost always it ends up creating multiple definitions of the function. On the other hand, if inline functions are not in header files then compilation units can't find them. In both the cases linker throws errors.

Therefore, a flexible way of inlining is often desirable but C++ language does not support it without some macro magic. The Inline Guard Macro idiom achieves this.

Solution and Sample Code

The solution is to put all the inline functions in a separate file called .ipp file and decorate each function with a macro INLINE. Header file and the implementation file is create as usual and the .ipp file in selectively included in one of the two files (header or implementation) depending upon whether inlining is desired. An example of a class Test is given below.

```
// test.ipp file
INLINE void Test::func()
{}
```

```
// test.hpp file
#ifndef MYPROJECT_TEST_H // Note include guards.
#define MYPROJECT_TEST_H

class Test
{
  public:
    void func();
};

#ifdef MYPROJECT_INLINE_ENABLED
#define INLINE inline // Define INLINE as inline (the keyword)
#include "test.ipp"   // It is included only when MYPROJECT_INLINE_ENABLED is defined, i.e. inlining is enabled.
#endif

#endif  // MYPROJECT_TEST_H
```

```
//test.cpp file
#include "test.hpp" // Include header file as usual.

#ifndef MYPROJECT_INLINE_ENABLED
#define INLINE      // INLINE is defined as empty string
#include "test.ipp" // It is included only when MYPROJECT_INLINE_ENABLED is NOT defined, i.e. inlining is disabled.
#endif
```

The effect of using Include Guard Macro is that depending upon whether MYPROJECT_INLINE_ENABLED is defined or not, test.ipp gets #included in either test.cpp or test.hpp. When it gets merged with test.cpp, functions are not inlined because INLINE is defined as an empty string. On the other hand when test.ipp is merged with test.hpp, INLINE is defined as inline (the keyword).

Now, what remains is to define the MYPROJECT_INLINE_ENABLED macro when desired. Generally, all modern C/C++ compilers allow defining macros at command line. For example to compile the above program on gcc using inlining, the argument **-DMYPROJECT_INLINE_ENABLED** option is added to the compiling command line. If no such macro is defined, the functions are automatically treated as non-inline and the program compiles OK.

Known Uses

- ACE (Adaptive Communication Environment)
- TAO (The ACE ORB)

Related Idioms

- Include Guard Macro
- Export Guard Macro

Inner Class

Intent

- Implementing multiple interfaces without multiple inheritance and yet provide natural looking up-casting.
- Provide multiple implementations of the same interface in a single abstraction.

Also Known As

Motivation

Signature of a virtual function in two independent interfaces provided by two independent class libraries may collide. It is a problem especially when a single class has to implement both the colliding functions in different ways depending upon the interface you consider. For example,

```
class Base1 /// Provided by Moon
{
  public:
      virtual int open (int) = 0;
      /* virtual */ ~Base1() {}  // No polymorphic deletion allowed
};

class Base2 /// Provided by Jupitor
{
  public:
      virtual int open (int) = 0;
      /* virtual */ ~Base2() {}  // No polymorphic deletion allowed
};

class Derived : public Base1, public Base2
{
  public:
    virtual int open (int i)
    {
      // Call from which base class?
      return 0;
    }
    /* virtual */ ~Derived () {}
};
```

The inner class idiom can help solve this problem.

Solution and Sample Code

Leaving the interface classes, Base1 and Base2 unchanged we can implement the Derived class as follows.

```
#include <iostream>
class Base1  /// Provided by Moon
{
 public:
  virtual int open() = 0;
  /* virtual */ ~Base1() {}  // No polymorphic deletion allowed
};

class Base2  /// Provided by Jupitor
{
 public:
  virtual int open() = 0;
  /* virtual */ ~Base2() {}  // No polymorphic deletion allowed
};

class Derived  // Note no inheritance
{
  class Base1_Impl;
  friend class Base1_Impl;
  class Base1_Impl : public Base1  // Note public inheritance
  {
   public:
    Base1_Impl(Derived* p) : parent_(p) {}
    int open() override { return parent_->base1_open(); }

   private:
    Derived* parent_;
  } base1_obj;  // Note member object here.

  class Base2_Impl;
  friend class Base2_Impl;
  class Base2_Impl : public Base2  // Note public inheritance
  {
   public:
    Base2_Impl(Derived* p) : parent_(p) {}
    int open() override { return parent_->base2_open(); }

   private:
    Derived* parent_;
  } base2_obj;  // Note member object here

  int base1_open() { return 111; }  /// implement
  int base2_open() { return 222; }  /// implement

 public:

  Derived() : base1_obj(this), base2_obj(this) {}

  operator Base1&() { return base1_obj; }  /// convert to Base1&
  operator Base2&() { return base2_obj; }  /// convert to Base2&
};                                         /// class Derived

int base1_open(Base1& b1) { return b1.open(); }

int base2_open(Base2& b2) { return b2.open(); }

int main(void) {
  Derived d;
  std::cout << base1_open(d) << std::endl;  // Like upcasting in inheritance.
  std::cout << base2_open(d) << std::endl;  // Like upcasting in inheritance.
}
```

Note the use of conversion operators in class Derived. (Derived class is really not a derived class!) The conversion operators allow conversion of Derived to Base1 even though they don't share inheritance relationship themselves! Use of member objects base1_obj and base2_obj eliminates the concern of object lifetime. Lifetime of member objects is same as that of the Derived object.

Known Uses

Related Idioms

- Interface Class
- Capability Query

References

Thinking in C++ Vol 2 - Practical Programming --- by Bruce Eckel.

Int-To-Type

Intent

- To treat an integral constant as a type at compile-time.
- To achieve static call dispatch based on constant integral values.

Also Known As

Integral constant wrappers

Motivation

Function overloading in C++ is based on different types, which prevents compile-time integral constants taking part into function overload resolution. There are at least two different mechanisms to perform static dispatching based on integral constants. First is enable-if idiom and the other one is the int-to-type idiom described below.

Solution and Sample Code

A simple template, initially described in Dr. Dobb's Journal (http://www.ddj.com/cpp/184403750) by Andrei Alexandrescu provides a solution to this idiom.

```
template <int I>
struct Int2Type
{
  enum { value = I };
};
```

The above template creates different types for different integer values used to instantiate the template. For example, *Int2Type<5>* is a different type from *Int2Type<10>*. Moreover, the integral parameter is saved in the associated constant *value*. As each integral constant results in a different type, this simple template can be used for static dispatching based on integral constants as shown below.

Consider an *Array* class that encapsulates a fixed size array very similar to that of the standard array class from TR1. In fact, our Array class is implemented as a derived class of the standard TR1 array class with the only difference of a *sort* function.

We intend to dispatch sort function at compile-time based on the size of the array to achieve some performance optimizations. For example, sorting an array of size zero or one should be a no-op. Similarly, arrays smaller than size 50 should be sorted using the insertion-sort algorithm whereas as larger arrays should be sorted using the quick-sort algorithm because insertion-sort algorithm is often more efficient than quick-sort algorithm for small data size. Note that, this selection of the sorting algorithm can be trivially done using run-time *if* condition. However, int-to-type idiom is used to achieve the same effect at compile-time as shown below.

```
#include <iostream>
#include <array>

template <int I>
```

```
struct Int2Type
{
  enum { value = I };
};

template <class T, unsigned int N>
class Array : public std::array <T, N>
{
   enum AlgoType { NOOP, INSERTION_SORT, QUICK_SORT };
   static const int algo = (N==0) ? NOOP :
                           (N==1) ? NOOP :
              (N<50) ? INSERTION_SORT : QUICK_SORT;
   void sort (Int2Type<NOOP>) { std::cout << "NOOP\n"; }
   void sort (Int2Type<INSERTION_SORT>) { std::cout << "INSERTION_SORT\n"; }
   void sort (Int2Type<QUICK_SORT>) { std::cout << "QUICK_SORT\n"; }
 public:
   void sort()
   {
     sort (Int2Type<algo>());
   }
};
int main(void)
{
  Array<int, 1> a;
  a.sort(); // No-op!
  Array<int, 400> b;
  b.sort(); // Quick sort
}
```

Few more associated types and constants can be defined with *Int2Type* template to increase its usability. For example, enumeration *value* is used to retrieve the integer constant associated with the type. Finally, other typedefs such as, *next* and *previous* are used to find other types *in order* such that *Int2Type<7>::next* is the same type as *Int2Type<9>::previous*.

```
template <int I>
struct Int2Type
{
  enum { value = I };
  typedef int value_type;
  typedef Int2Type<I> type;
  typedef Int2Type<I+1> next;
  typedef Int2Type<I-1> previous;
};
```

Known Uses

- Integral constant wrappers (bool_, int_, long_) (http://www.boost.org/doc/libs/1_36_0/libs/mpl/doc/refmanual/integral-constant.html) in Boost.MPL
- Dimensional Analysis (http://www.boost.org/doc/libs/1_36_0/libs/mpl/doc/tutorial/representing-dimensions.html)

Related Idioms

- enable-if
- Type Generator

References

[1] Generic<Programming>: Mappings between Types and Values (http://www.ddj.com/cpp/184403750) -- Andrei Alexandrescu

Interface Class

Intent

- To separate an interface of a class from its implementation.
- Invoke implementation of an abstraction/class using runtime polymorphism.

Also Known As

Motivation

Separating an interface of a class from its implementation is fundamental to good quality object oriented software design/programming. For object oriented programming, the principal mechanism of separation is the Interface Class. However C++ (when compared to, say, Java) provides no exclusive mechanism for expressing such a separation. In Java, **interface** keyword is used to specify only the public methods that are supported by an abstraction. C++ does not have such a keyword but its functionality can be expressed closely using the Interface Class idiom. The idea is to express only the public methods of an abstraction and provide no implementation of any of them. Also, lack of implementation means no instances of the interface class should be allowed.

Solution and Sample Code

An interface class contains only a virtual destructor and pure virtual functions, thus providing a construct similar to the interface constructs of other languages (e.g. Java). An interface class is a class that specifies the polymorphic interface i.e. pure virtual function declarations into a base class. The programmer using a class hierarchy can then do so via a base class that communicates only the interface of classes in the hierarchy.

```
class shape   // An interface class
{
  public:
    virtual ~shape();
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void draw() = 0;
//...
};

class line : public shape
{
  public:
    virtual ~line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    virtual void draw(); // implements draw
  private:
    point end_point_1, end_point_2;
//...
};

int main (void)
{
  std::vector<shape *> shapes;
  //  Fill up shapes vector somehow.
  for (vector<shape *>::iterator iter (shapes.begin());
       iter != shapes.end();
       ++iter)
  {
    (*iter)->draw();
  }
  //  Clean up shapes vector. (Normally we would use something like boost::shared_ptr to automate cleanup,
  //  this is for illustration only)
}
```

Every interface class should have a virtual destructor. Virtual destructor makes sure that when a shape is deleted polymorphically, correct destructor of the derived class is invoked. Otherwise there is a good chance of resource leak. Benefit of expressing design using interface classes are many:

- New shape abstractions can be added without changing the code that depends only on shape interface. For example, Square can inherit from shape and implement the interface methods in its own way. The function main() needs no changes at all.
- Separation of interface from implementation prevents recompilation of parts of the program that depend only on the interface classes.
- Dependency Inversion Principle (DIP) states that implementation classes should not depend on each other. Instead, they should depend on common abstraction represented using an interface class. DIP reduces coupling in a object-oriented system.

Known Uses

Nearly all good object-oriented software in C++!

Related Idioms

- Capability Query
- Inner Class

References

C++ Interface Classes - An Introduction (http://www.twonine.co.uk/articles/CPPInterfaceClassesIntro.pdf)

Iterator Pair

Intent

Specify a range of data values without worrying about the underlying data structure used by the data values.

Also Known As

Sometimes this is referred to as an Iterator Range.

Motivation

It is well understood that it is useful to create a vector<int> from another vector<int> using a copy constructor. Similarly, it is useful to create a vector<double> from a vector<int> using Coercion by Member Template idiom applied on a member template constructor. A code example is given below.

```
template <class T>
class vector
{
  public:
    vector (const vector<T> &); // copy constructor
    template <class U>
    vector (const vector<U> &); // constructor using Coercion by Member Template Idiom.
};
```

The vector interface is still not flexible enough for some needs. For example, A vector can't create itself from a list or a set or a POD array.

```
template <class T>
class vector
{
  public:
    vector (const list<T> &);
    // constructor must know the interface of list<T> (not necessarily std::list)
```

```
    vector (const set<T> &);
    // constructor must know the interface of set<T> (not necessarily std::set)
    vector (const T * pod_array);
    // another constructor - does not know where pod_array ends - too inflexible!
};
```

Iterator-pair is an idiom that addresses this challenge. It is based on the Iterator design pattern (obviously!) Iterator pattern intent: Provide an object which traverses some aggregate structure, abstracting away assumptions about the implementation of that structure.

Solution and Sample Code

A pair of iterators is used to designate a beginning and an end of a range of values. By virtue of the iterator design pattern whoever (in our example vector) uses iterator pair idiom can access the range without worrying about the implementation of the aggregate data structure. The only requirement is that the iterators should expose a fixed, minimal interface such as a pre-increment operator.

```
template <class T>
class vector
{
    T * mem;
  public:
    template <class InputIterator>
    vector (InputIterator begin, InputIterator end) // Iterator-pair constructor
    {
      // allocate enough memory and store in mem.
      mem=new T[std::distance(begin, end)];
      for (int i = 0; begin != end; ++i)
      {
        mem[i] = *begin;
        ++begin;
      }
    }
};
int main (void)
{
  std::list<int> l(4);
  std::fill(l.begin(),l.end(), 10);    // fill up list using iterator pair technique.
  std::set<int> s(4);
  std::fill(s.begin(),s.end(), 20);    // fill up set using iterator pair technique.

  std::vector<int> v1(l.begin(), l.end());  // create vector using iterator pair technique.
  std::vector<int> v2(s.begin(), s.end());  // create another vector.
}
```

Iterator-pair idiom is often combined with member templates because the exact type of the iterators is not known apriori. It could be set<T>::iterator or list<T>::iterator or a POD array. Irrespective of the type, any generic algorithm written in terms of the iterator pairs works. It is often useful to indicate the concept that iterator types are supposed to model. In the example above, the iterators are required to model at least the InputIterator concept. More information about iterator categories (tags) and their uses are described in Tag Dispatching idiom.

Sometime iterator pair idiom is unavoidable. For example, to construct a std::string from a buffer of characters with embedded null characters iterator-pair idiom is unavoidable.

```
char buf[] = { 'A', 'B', 0, 'C', 0, 'D'};
std::string str1 (buf); // only creates "AB"
std::string str2 (buf, buf + sizeof (buf) / sizeof (*buf)); // Using iterator pair. Creates "AB_C_D"
// buf is start of the range and buf + sizeof (buf) / sizeof (*buf) is the end of the range.

std::cout << str1 << " length = " << str1.length() << std::endl; // AB length = 2
std::cout << str2 << " length = " << str2.length() << std::endl; // AB_C_D length = 6
```

Known Uses

All standard containers

Related Idioms

- Coercion by Member Template
- Tag Dispatching

References

Making New Friends

Intent

To simplify creation of friend functions for a class template.

Motivation

Friend functions are often used to provide some auxiliary additional interfaces for a class. For example, insertion (<<), extraction (>>) operators and overloaded arithmetic operators are often friends. Declaring friend functions of a class template is a little more complicated compared to declaring a friend function for a non-template class. There are four kinds of relationships between classes and their friends when templates are involved:

- One-to-many: A non-template function may be a friend to all template class instantiations.
- Many-to-one: All instantiations of a template function may be friends to a regular non-template class.
- One-to-one: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- Many-to-many: All instantiations of a template function may be a friend to all instantiations of the template class.

The one-to-one relationship is of interest here because setting that up in C++ requires additional syntax. An example follows.

```
template<typename T>
class Foo {
   T value;
public:
   Foo(const T& t) { value = t; }
   friend ostream& operator<<(ostream&, const Foo<T>&);
};

template<typename T>
ostream& operator<<(ostream& os, const Foo<T>& b) {
   return os << b.value;
}
```

The above example is no good for us because the inserter is not a template but it still uses a template argument (T). This is a problem since it's not a member function. The operator<<( ) must be a template so that distinct specialization for each T can be created.

Solution here is to declare a insertion operator template outside the class before friend declaration and adorn the friend declaration with <>. It indicates that a template declared earlier should be made friend.

```
// Forward declarations
template<class T> class Foo;
template<class T> ostream& operator<<(ostream&,
```

```
                                    const Foo<T>&);
template<class T>
class Foo {
   T value;
public:
   Foo(const T& t) { value = t; }
   friend ostream& operator<< <>(ostream&, const Foo<T>&);
};

template<class T>
ostream& operator<<(ostream& os, const Foo<T>& b)
{
   return os << b.value;
}
```

A disadvantage of the above solution is that it is quite verbose.

Solution and Sample Code

Dan Saks suggested another approach to overcome the verbosity of the above solution. His solution is known as "Making New Friends" idiom. The idea is to define the friend function inside the class template as shown below.

```
template<typename T>
class Foo {
   T value;
public:
   Foo(const T& t) { value = t; }
   friend ostream& operator<<(ostream& os, const Foo<T>& b)
   {
      return os << b.value;
   }
};
```

Such a friend function is not a template, but the template acts as a factory for "making" new friends. A new non-template function is created for each specialization of Foo.

Known Uses

Related Idioms

References

C++ FAQ 35.16 (http://www.parashift.com/c++-faq/template-friends.html)

Metafunction

Intent

- To encapsulate a complex type computation algorithm
- To generate a type using compile-time type selection techniques

Also Known As

Motivation

Templates is a powerful feature of C++, which can be used to perform arbitrary computations at compile-time, which is known as template metaprogramming. Some of the basic examples of the computations performed at compile-time are: (1) selection of a type based on compile-time constants or (2) computing factorial of a

number. As a matter of fact, C++ templates is a turing complete (http://ubiety.uwaterloo.ca/~tveldhui/papers/2003/turing.pdf) sub-language of C++. Metafunction idiom is the principal way of writing compile-time algorithms in C++.

Algorithms -- compile-time or run-time -- should be encapsulated so that they are easier to use and reuse. Conventionally, run-time algorithms are encapsulated in functions that are invoked, obviously, at run-time. Metafunctions, on the other hand, are compile-time analogs of run-time functions. Traditional functions accept values/objects as parameters and return values/objects. However, metafunctions accept types and compile-time constants as parameters and return types/constants.

Solution and Sample Code

A metafunction, contrary to its name, is a class template. Implementation of metafunctions is often based on template specializations. For example, consider the following *IF* metafunction, which is compile-time equivalent of run-time *if* statement. Depending upon the value of the first parameter, *IF* metafunction yields either an *int* or a *long* in the example below.

```
template <bool, class L, class R>
struct IF
{
  typedef R type;
};

template <class L, class R>
struct IF<true, L, R>
{
  typedef L type;
};

IF<false, int, long>::type i; // is equivalent to long i;
IF<true,  int, long>::type i; // is equivalent to int i;
```

Factorial metafunction below is another example showing how a recursive factorial computation algorithm can be encapsulated using C++ templates. This metafunction yields an integral value rather than a type.

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0>
{
    enum { value = 1 };
};

// Factorial<4>::value == 24
// Factorial<0>::value == 1
void foo()
{
    int x = Factorial<4>::value; // == 24
    int y = Factorial<0>::value; // == 1
}
```

Metafunction and Type Generator

Metafunction is a more general idiom than the type generator idiom. The intent of the metafunction idiom is to encapsulate compile-time computation whereas, type generator simplifies specification of a type. Metafunctions that produce type(s) as a result of a compile-time computation are type generators, but not every metafunction is a type generator. For example, the *Factorial* metafunction shown before produces an integral value, not a type. Metafunctions are often implemented using compile-time control structures or other metafunctions.

Libraries such as Boost.MPL (http://www.boost.org/doc/libs/release/libs/mpl) provide a large collection of metafunctions and compile-time data structures to simplify C++ template metaprogramming.

Higher order metafunctions

These are metafunctions that accept other metafunctions as parameters and use them during computation. This is conceptually is similar a function accepting a pointer to another function or a function object as a parameter at run-time. Only difference is that metafunctions exist only at compile-time. *boost::mpl::transform* is an example of such a higher order metafunction.

Known Uses

Boost.MPL (http://www.boost.org/doc/libs/release/libs/mpl)

Related Idioms

- Type Generator

References

A Deeper Look at Metafunctions (http://www.artima.com/cppsource/metafunctions.html) -- David Abrahams and Aleksey Gurtovoy

Move Constructor

Intent

To transfer the ownership of a resource held by an object to another object **in C++03**. Note: In C++11, move-constructors are implemented using the built-in rvalue reference feature.

Also Known As

- Colvin-Gibbons trick
- Source and Sink Idiom

Motivation

Some objects in C++ exhibit so called move semantics. For example, `std::auto_ptr`. In the code that follows `auto_ptr b` ceases to be useful after the creation of object `a`.

```
std::auto_ptr <int> b (new int (10));
std::auto_ptr <int> a (b);
```

The copy constructor of `auto_ptr` modifies its argument, and hence it does not take a `const` reference as a parameter. It poses no problem in the above code because object `b` is not a `const` object. But it creates a problem when a temporary is involved.

When a function returns an object by value and that returned object is used as an argument to a function (for example to construct another object of the same class), compilers create a temporary of the returned object. These temporaries are short lived and as soon as the statement is executed, the destructor of the temporary is called. The temporary therefore owns its resources for a very short time. The problem is that temporaries are

quite like `const` objects (it makes little sense to modify a temporary object). Therefore, they can not bind to a non-const reference and as a result, can not be used to call the constructor taking a non-`const` reference. A move constructor can be used in such cases.

Solution and Sample Code

```
namespace detail {

template <class T>
struct proxy
{
  T *resource_;
};

} // detail

template <class T>
class MovableResource
{
  private:
    T * resource_;

  public:
    explicit MovableResource (T * r = 0) : resource_(r) { }
    ~MovableResource() throw() { delete resource_; }   // Assuming std::auto_ptr like behavior.

    MovableResource (MovableResource &m) throw () // The "Move constructor" (note non-const parameter)
      : resource_ (m.resource_)
    {
      m.resource_ = 0; // Note that resource in the parameter is moved into *this.
    }

    MovableResource (detail::proxy<T> p) throw () // The proxy move constructor
      : resource_(p.resource_)
    {
      // Just copying resource pointer is sufficient. No need to NULL it like in the move constructor.
    }

    MovableResource & operator = (MovableResource &m) throw () // Move-assignment operator (note non-const parameter)
    {
      // copy and swap idiom. Must release the original resource in the destructor.
      MovableResource temp (m); // Resources will be moved here.
      temp.swap (*this);
      return *this;
    }

    MovableResource & operator = (detail::proxy<T> p) throw ()
    {
      // copy and swap idiom. Must release the original resource in the destructor.
      MovableResource temp (p);
      temp.swap(*this);
      return *this;
    }

    void swap (MovableResource &m) throw ()
    {
      std::swap (this->resource_, m.resource_);
    }

    operator detail::proxy<T> () throw () // A helper conversion function. Note that it is non-const
    {
      detail::proxy<T> p;
      p.resource_ = this->resource_;
      this->resource_ = 0;     // Resource moved to the temporary proxy object.
      return p;
    }
};
```

The move constructor/assignment idiom plays an important role in the code snippet shown next. The function `func` returns the object by value i.e., a temporary object is returned. Though `MovableResource` does not have any copy-constructor, the construction of local variable `a` in `main` succeeds, while moving the ownership away from the temporary object.

```
MovableResource<int> func()
{
  MovableResource<int> m(new int());
  return m;
}
int main()
{
  MovableResource<int> a(func()); // Assuming this call is not return value optimized (RVO'ed).
}
```

The idiom uses a combination of three interesting and standard features of C++.

- A copy-constructor need not take its parameter by `const` reference. The C++ standard provides a definition of a copy-constructor in section 12.8.2. Quoting -- A non-template constructor for class X is a copy constructor if its first parameter is of type `X&`, `const X&`, `volatile X&` or `const volatile X&`, and either there are no other parameters or else all other parameters have default arguments. -- End quote.
- A sequence of conversions via the `detail::proxy<T>` object is identified automatically by the compiler.
- Non-const functions can be called on temporary objects. For instance, the conversion function `operator detail::proxy<T> ()` is non-const. This member conversion operator is used to modify the temporary object.

The compiler seeks a copy-constructor to initialize object `a`. There is a copy-constructor defined but it accepts its parameter by non-const reference. The non-const reference does not bind to the temporary object return by function `func`. Therefore, the compiler looks for other options. It identifies that a constructor that accepts a `detail::proxy<T>` object is provided. So it tries to identify a conversion operator that converts the `MovableResource` object to `detail::proxy<T>`. As a matter of fact, such a conversion operator is also provided (`operator detail::proxy<T>()`). Note also that the conversion operator is non-const. That's not a problem because C++ allows invoking non-const functions on temporaries. Upon invocation of this conversion function, the local `MovableResource` object (m) loses its resource ownership. Only the `proxy<T>` object knows the pointer to `T` for a very brief period of time. Subsequently, the converting constructor (the one that takes `detail::proxy<T>` as a parameter) successfully obtains the ownership (object `a` in `main`).

Let's look into the details of how temporary objects are created and used. In fact, the above steps are executed not once but twice in exactly the same way. First to create a temporary `MovableResource` object and later to create the final object `a` in `main`. The second exceptional rule of C++ comes into play when `a` is being created from the temporary `MovableResource` object. The conversion function (`operator detail::proxy<T>()`) is called on the temporary `MovableResource` object. Note that it is non-const. A non-const member function can be called on a temporary object unlike real `const` objects. Section 3.10.10 in C++ ISO/IEC 14882:1998 standard clearly mentions this exception. More information on this exception can be found here (http://cpptruths.blogspot.com/2009/08/modifying-temporaries.html). The conversion operator happens to be a non-const member function. Therefore, the temporary `MovableResource` object also loses its ownership as described above. Several temporary `proxy<T>` objects also are created and destroyed when the compiler figures out the right sequence of conversion functions. It is possible that the compiler might eliminate certain temporaries using return value optimization (RVO). To observe all the temporaries, use --no-elide-constructors option to the gcc compiler.

This implementation of the move constructor idiom is useful to transfer the ownership of resources (e.g., heap-allocated memory) in and out of a function. A function that accepts `MovableResource` by-value serves as a *sink* function because the ownership is transferred to a function deeper inside the call-stack whereas a function that returns `MovableResource` by-value serves as *source* because the ownership is moved to an outer scope. Hence the name *source-and-sink* idiom.

Safe Move Constructor

Although the source-and-sink idiom is quite useful at the boundaries of a function call, there are some undesirable side-effects of the above implementation. Particularly, simple expressions that look like copy-assignment and copy-initialization do not make copies at all. For instance, consider the following code:

```
MovableResource mr1(new int(100));
MovableResource mr2;
mr2 = mr1;
```

`MovableResource` object `mr2` looks like it has been copy-assigned from `mr1`. However, it silently steals the underlying resource from `mr1` leaving it behind as if it is default initialized. The `mr1` object no longer owns the heap allocated integer. Such behavior is quite surprising to most programmers. Moreover, generic code written assuming regular copy semantics will likely have very undesirable consequences.

The semantics mentioned above are commonly known as *Move Semantics*. It is a powerful optimization. However, part of the problem of this optimization, implemented in the above form, is the lack of error messages during compilation of assignment and copy-initialization of expressions that look like a copy but in fact perform a move. Specifically, implicitly copying one `MovableResource` from another *named* `MovableResource` (e.g., `mr1`) is unsafe and therefore should be indicated by an error.

Note that stealing resources from a temporary object is perfectly acceptable because they do not last long. The optimization is meant to be used on *rvalues* (temporary objects). Safe move constructor idiom is a way of achieving this distinction without language-level support (rvalue references in C++11 (http://en.wikipedia.org/wiki/C%2B%2B11#Rvalue_references_and_move_constructors)).

The safe move constructor idiom prevents implicit move semantics from named objects by declaring private constructor and assignment operator functions.

```
private:
  MovableResource (MovableResource &m) throw ();
  MovableResource & operator = (MovableResource &m) throw ();
```

Key thing to note here is that the parameter types are non-const. They bind to named objects only and not to temporary objects. This little change alone is sufficient to disable implicit moves from named objects but allow moves from temporary objects. The idiom also provides a way to do explicit move from a named object, if desired.

```
template <class T>
MovableResource<T> move(MovableResource<T> & mr) throw() // Convert explicitly to a non-const reference to rvalue
{
  return MovableResource<T>(detail::proxy<T>(mr));
}
MovableResource<int> source()
{
  MovableResource<int> local(new int(999));
  return move(local);
}
void sink(MovableResource<int> mr)
{
  // Do something with mr. mr is deleted automatically at the end.
}
int main(void)
{
  MovableResource<int> mr(source()); // OK
  MovableResource<int> mr2;
  mr2 = mr;                          // Compiler error
  mr2 = move(mr);                    // OK
  sink(mr2);                         // Compiler error
  sink(move(mr2));                   // OK
}
```

The `move` function above is also quite idiomatic. It must accept the parameter as a non-const reference to the object so that it binds to l-values only. It turns a non-const `MovableResource` lvalue (named object) into an rvalue (temporary object) via conversion to `detail::proxy`. Two explicit conversions are necessary inside the

`move` function to avoid certain language quirks (http://stackoverflow.com/questions/8450301/the-move-function-in-unique-ptr-c03-emulation). Ideally, the `move` function should be defined in the same namespace as that of `MovableResource` class so that the function can be looked up using argument dependent lookup (ADL).

Function `source` returns a local `MovableResource` object. Because it is a named object, `move` must be used to turn it into an rvalue before returning. In `main`, first `MovableResource` is initialized directly from the temporary returned by `func`. Simple assignment to another `MovableResource` fails with a compiler error but explicit move works. Similarly, implicit copy to the `sink` function does not work. Programmer must explicitly express its interest in moving the object to the `sink` function.

Finally, it is also important that the key functions (construction from and to `detail::proxy`) be non-throwing to guarantee at least basic exception guarantee. No exceptions should be thrown in the meanwhile, otherwise there will be resource leaks.

C++11 provides rvalue references (http://www.artima.com/cppsource/rvalue.html), which support language-level move semantics, eliminating the need for the Move Constructor idiom.

Historical Alternatives

An old (but inferior) alternative is to use a `mutable` member to keep track of ownership to sneak past the const-correctness checks of the compiler. The following code shows how the `MovableResource` may be implemented alternatively.

```
template<class T>
class MovableResource
{
  mutable bool owner;
  T* px;

public:
  explicit MovableResource(T* p=0)
       : owner(p), px(p) {}

  MovableResource(const MovableResource& r)
       : owner(r.owner), px(r.release()) {}

  MovableResource & operator = (const MovableResource &r)
  {
    if ((void*)&r != (void*)this)
    {
      if (owner)
        delete px;
      owner = r.owner;
      px = r.release();
    }
    return *this;
  }

  ~MovableResource() { if (owner) delete px; }
  T& operator*() const { return *px; }
  T* operator->() const { return px; }
  T* get()    const { return px; }
  T* release()  const { owner = false; return px; } // mutable 'ownership' changed here.
};
```

This technique, however, has several disadvantages.

- The copy constructor and copy-assignment operators do not make logical **copies**. On the contrary, they transfer the ownership from right hand side object to `*this`. This fact is not reflected in the interface of the second implementation of `MovableResource`. The first `MovableResource` class accepts a `MovableResource` object by non-const reference, which prevents the use of the object in contexts where a copy is expected.
- The Const auto_ptr idiom, which is C++03's way of preventing transfer of ownership is simply not possible with the second implementation of the `MovableResource` class. This idiom depends on the fact

that `const` auto_ptr objects cannot bind to the non-const reference parameters of the move-constructor and move-assignment operators. Making the parameters `const` references defeats the purpose.
- The `boolean` flag 'owner' increases the size of the structure. The increase in size is substantial (essentially double) for classes like `std::auto_ptr`, which otherwise contain just a pointer. Doubling of the size is due to compiler-enforced alignment of data.

Known Uses

- Boost.Move (http://www.boost.org/doc/html/move.html) library
- std::auto_ptr uses "unsafe" version of move constructor.
- The unique_ptr C++03 emulation (http://home.roadrunner.com/~hinnant/unique_ptr03.html) by Howard Hinnant uses the safe move constructor idiom.

Related Idioms

- Resource Acquisition Is Initialization (RAII)
- Scope Guard
- Resource Return
- Const auto_ptr

References

- Move Constructors - by M. D. Wilson (http://www.synesis.com.au/resources/articles/cpp/movectors.pdf)
- auto_ptr and auto_ptr_ref - by Nicolai Josuttis (http://www.josuttis.com/libbook/auto_ptr.html)
- Change Of Authority and Thread Safe Interface goes synchronized - by Philipp Bachmann (http://hillside.net/plop/2005/proceedings/PLoP2005_pbachmann1_0.pdf)
- A Brief Introduction to Rvalue References (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html)

Multi-statement Macro

Intent

To write a multi-statement (multi-line) macro.

Also Known As

Motivation

Sometimes it is useful to group two or more statements into a macro and call them like a function call. Usually, an inline function should be the first preference but things such as debug macros are almost invariably macros rather than function calls. Grouping multiple statments into one macro in a naive way could lead to compilation errors, which are not obvious at first look. For example,

```
#define MACRO(X,Y) { statement1; statement2; }
```

would fail in an if statement if a semi-colon is appended at the end.

```
if (cond)
   MACRO(10,20); // Compiler error here because of the semi-colon.
else
   statement3;
```

The above statement expands to

```
if (cond)
   { statement1; statement2; }; // Compiler error here because of the semi-colon.
else
   statement3;
```

giving a compiler error. Therefore, people came up with a widely used idiom for multi-statement macros, which is based on a do-while loop.

Solution and Sample Code

Here is an example of the multi-statement macro idiom.

```
#define MACRO(arg1, arg2) do {  \
  /* declarations, if any */    \
  statement1;                   \
  statement2;                   \
  /* ... */                     \
  } while(0)    /* (no trailing ; ) */
```

When the caller appends a semicolon, this expansion becomes a single statement regardless of the context. Optimizing compilers usually remove any dead tests, such as while(0). This idiom is not useful when macro is used as a parameter to a function call. Moreover, this idiom allows return statement.

```
func(MACRO(10,20)); // Syntax error here.
```

Known Uses

ACE_NEW_RETURN, ACE_NEW_NORETURN macros in Adaptive Communication Environement (ACE).

```
#define ACE_NEW_RETURN(POINTER,CONSTRUCTOR,RET_VAL) \
      do { POINTER = new (ACE_nothrow) CONSTRUCTOR; \
      if (POINTER == 0) { errno = ENOMEM; return RET_VAL; } \
    } while (0)
```

Related Idioms

References

- What's the best way to write a multi-statement macro? (http://c-faq.com/cpp/multistmt.html)More C++ Idioms/Multiple Member Initialization

Member Detector

Intent

To detect the presence of a specific member attribute, function or type in a class.

Also Known As

Motivation

Compile-time reflection capabilities are the cornerstone of C++ template meta-programming. Type traits libraries such as Boost.TypeTraits and TR1 <type_traits> header provide powerful ways of extracting information about types and their relationships. Detecting the presence of a data member in a class is also an example of compile-time reflection.

Solution and Sample Code

Member detector idiom is implemented using the Substitution Failure Is Not An Error (SFINAE) idiom. The following class template *DetectX<T>* is a meta-function that determines whether type *T* has a data or function member named *X* in it or not. Note that the type of the data member *X* does not matter nor does the return value and arguments of the member function (if it is one).

```
template<typename T>
class DetectX
{
    struct Fallback { int X; }; // add member name "X"
    struct Derived : T, Fallback { };

    template<typename U, U> struct Check;

    typedef char ArrayOfOne[1];  // typedef for an array of size one.
    typedef char ArrayOfTwo[2];  // typedef for an array of size two.

    template<typename U>
    static ArrayOfOne & func(Check<int Fallback::*, &U::X> *);

    template<typename U>
    static ArrayOfTwo & func(...);

  public:
    typedef DetectX type;
    enum { value = sizeof(func<Derived>(0)) == 2 };
};
```

This idiom works by creating a controlled ambiguity during compilation and recovering from that using the SFINAE idiom. First proxy class, *Fallback*, has a data member of the same name that we want to detect the presence of. Class *Derived* inherits from both *T* and *Fallback*. As a result, *Derived* class will have at least one data member named *X*. Potentially, *Derived* class may have two *X* data members if *T* also has one.

The *Check* template is used to create controlled ambiguity. *Check* template takes two parameters. First is a type parameter and the second is an instance of that type. For example, *Check<int, 5>* is be a valid instantiation. Two overloaded functions named *func* create an overload-set as often done in the SFINAE idiom. The first *func* function can be instantiated only if the address of data member *U::X* can be taken unambiguously. Address of *U::X* can be taken if there is exactly one *X* data member in the *Derived* class; i.e., *T* does not have data member *X*. If *T* has *X* in it, the address of *U::X* can't be taken without further disambiguation and therefore the instantiation of the first *func* fails and the other function is chosen, all without an error. Note the difference between the return types of the two *func* functions. The first function returns a reference to array of size one whereas the second one returns a reference to an array of size two. This difference in the sizes allows us to identify which function was instantiated.

Finally, a boolean *value* is exposed, which is true only if the sizeof the return type of the function is two. That is, when the second *func* is instantiated only because *T* has *X* data member.

For every different member that is to be detected, the above class template needs to change. A macro would be preferable in such cases. The following sample code demonstrates the use of a macro.

```
#define CREATE_MEMBER_DETECTOR(X)                                                 \
template<typename T> class Detect_##X {                                           \
    struct Fallback { int X; };                                                   \
    struct Derived : T, Fallback { };                                             \
                                                                                  \
    template<typename U, U> struct Check;                                         \
```

```
                                                                        \
    typedef char ArrayOfOne[1];                                         \
    typedef char ArrayOfTwo[2];                                         \
                                                                        \
    template<typename U> static ArrayOfOne & func(Check<int Fallback::*, &U::X> *); \
    template<typename U> static ArrayOfTwo & func(...);                 \
  public:                                                               \
    typedef Detect_##X type;                                            \
    enum { value = sizeof(func<Derived>(0)) == 2 };                     \
};

CREATE_MEMBER_DETECTOR(first);
CREATE_MEMBER_DETECTOR(second);

int main(void)
{
  typedef std::pair<int, double> Pair;
  std::cout << ((Detect_first<Pair>::value && Detect_second<Pair>::value)? "Pair" : "Not Pair");
}
```

Using C++11 features, this example can be rewritten so that the controlled ambiguity is created using the `decltype` specifier instead of a *Check* template and a pointer to member. The result can then be wrapped inside a class inheriting from `integral_constant` to provide an interface identical to the type predicates present in the standard header `<type_traits>`.

```
#include <type_traits> // To use 'std::integral_constant'.
#include <iostream>    // To use 'std::cout'.
#include <iomanip>     // To use 'std::boolalpha'.


#define GENERATE_HAS_MEMBER(member)                                     \
                                                                        \
template < class T >                                                    \
class HasMember_##member                                                \
{                                                                       \
private:                                                                \
    using Yes = char[2];                                                \
    using  No = char[1];                                                \
                                                                        \
    struct Fallback { int member; };                                    \
    struct Derived : T, Fallback { };                                   \
                                                                        \
    template < class U >                                                \
    static No& test ( decltype(U::member)* );                           \
    template < typename U >                                             \
    static Yes& test ( U* );                                            \
                                                                        \
public:                                                                 \
    static constexpr bool RESULT = sizeof(test<Derived>(nullptr)) == sizeof(Yes); \
};                                                                      \
                                                                        \
template < class T >                                                    \
struct has_member_##member                                              \
: public std::integral_constant<bool, HasMember_##member<T>::RESULT>    \
{                                                                       \
};


GENERATE_HAS_MEMBER(att)  // Creates 'has_member_att'.
GENERATE_HAS_MEMBER(func) // Creates 'has_member_func'.


struct A
{
    int att;
    void func ( double );
};

struct B
{
    char att[3];
    double func ( const char* );
};

struct C : A, B { }; // It will also work with ambiguous members.
```

```
int main ( )
{
    std::cout << std::boolalpha
              << "\n" "'att' in 'C' : "
              << has_member_att<C>::value // <type_traits>-like interface.
              << "\n" "'func' in 'C' : "
              << has_member_func<C>() // Implicitly convertible to 'bool'.
              << "\n";
}
```

Detecting member types

The above example can be adapted to detect existence of a member type, be it a nested class or a typedef, even if it is incomplete. This time, the ambiguity would be created by adding to *Fallback* a nested type with the same name as the macro argument.

```
#include <type_traits> // To use 'std::integral_constant'.
#include <iostream>    // To use 'std::cout'.
#include <iomanip>     // To use 'std::boolalpha'.


#define GENERATE_HAS_MEMBER_TYPE(Type)                                              \
                                                                                    \
template < class T >                                                                \
class HasMemberType_##Type                                                          \
{                                                                                   \
private:                                                                            \
    using Yes = char[2];                                                            \
    using  No = char[1];                                                            \
                                                                                    \
    struct Fallback { struct Type { }; };                                           \
    struct Derived : T, Fallback { };                                               \
                                                                                    \
    template < class U >                                                            \
    static No& test ( typename U::Type* );                                          \
    template < typename U >                                                         \
    static Yes& test ( U* );                                                        \
                                                                                    \
public:                                                                             \
    static constexpr bool RESULT = sizeof(test<Derived>(nullptr)) == sizeof(Yes); \
};                                                                                  \
                                                                                    \
template < class T >                                                                \
struct has_member_type_##Type                                                       \
: public std::integral_constant<bool, HasMemberType_##Type<T>::RESULT>             \
{ };                                                                                \


GENERATE_HAS_MEMBER_TYPE(Foo) // Creates 'has_member_type_Foo'.


struct A
{
    struct Foo;
};

struct B
{
    using Foo = int;
};

struct C : A, B { }; // Will also work on incomplete or ambiguous types.


int main ( )
{
    std::cout << std::boolalpha
              << "'Foo' in 'C' : "
              << has_member_type_Foo<C>::value
              << "\n";
}
```

This is much like the member detection macro that declares a data member of the same name as the macro argument. An overload set of two *test* functions is then created, just like the other examples. The first version can only be instantiated if the type of *U::Type* can be used unambiguously. This type can be used only if there is exactly one instance of *Type* in *Derived*, i.e. there is no *Type* in *T*. If *T* has a member type *Type*, it is garanteed to be different than *Fallback::Type*, since the latter is a unique type, hence creating the ambiguity. This leads to the second version of *test* being instantiated in case the substitution fails, meaning that *T* has indeed a member type *Type*. Since no objects are ever created (this is entirely resolved by compile-time type checking), *Type* can be an incomplete type or be ambiguous inside *T*; only the name matters. We then wrap the result in a class inheriting from `integral_constant`, just like before to provide the same interface as the standard library.

Detecting overloaded member functions

A variation of the member detector idiom can be used to detect existence of a specific member function in a class even if it is overloaded.

```
template<typename T, typename RESULT, typename ARG1, typename ARG2>
class HasPolicy
{
    template <typename U, RESULT (U::*)(ARG1, ARG2)> struct Check;
    template <typename U> static char func(Check<U, &U::policy> *);
    template <typename U> static int func(...);
  public:
    typedef HasPolicy type;
    enum { value = sizeof(func<T>(0)) == sizeof(char) };
};
```

The *HasPolicy* template above checks if *T* has a member function called *policy* that takes two parameters *ARG1*, *ARG2* and returns *RESULT*. Instantiation of *Check* template succeeds only if *U* has a *U::policy* member function that takes two parameters and returns *RESULT*. Note that the first type parameter of *Check* template is a type whereas the second parameter is a pointer to a member function in the same type. If *Check* template cannot be instantiated, the only remaining *func* that returns an *int* is instantiated. The size of the return value of *func* eventually determines the answer of the type-trait: true or false.

Known Issues

Will not work if the class checked for member is declared final (C++11 keyword).

Will not work to check for a member of a union (unions cannot be base classes).

Related Idioms

- Substitution Failure Is Not An Error (SFINAE)

References

- Substitution failure is not an error, part II (https://cpptalk.wordpress.com/2009/09/12/substitution-failure-is-not-an-error-2), Roman Kecher

Named Constructor

Intent

- To have a readable and intuitive way of creating objects of a class
- To impose certain constraints while creating objects of a class

Also Known As

Motivation

In C++, constructors are distinguished from each other only based on the type, the order and the number of parameters. Of course when a class has multiple constructors, each constructor has a different purpose. However, in C++ it is hard to capture that "semantic" difference in the interface of the class because all the constructors have the same name and only parameters can distinguish between them. Reading code with lots of constructor calls only differing in the type/order/number of parameters is quite unintuitive except for the original developer of the class. Named constructor idiom addresses the problem.

Solution and Sample Code

The named constructor idiom uses a set of static member functions with meaningful names to create objects instead of constructors. Constructors are either private or protected and clients have access only to the public static functions. The static functions are called "named constructors" as each unique way of creating an object has a different intuitive name. Consider the example below:

```
class Game
{
  public:
    static Game createSinglePlayerGame() { return Game(0); } // named constructor
    static Game createMultiPlayerGame() { return Game(1); }  // named constructor
  protected:
    Game (int game_type);
};
int main(void)
{
   Game g1 = Game::createSinglePlayerGame(); // Using named constructor
   Game g2 = Game(1); // multiplayer game; without named constructor (does not compile)
}
```

Without using the named constructor idiom in the class above, it is difficult to convey the meaning of what *Game(1)* and *Game(0)* means. The idiom makes it loud and clear! Additionally, it is possible to put certain constraints on the object creation process using this idiom. For example, named constructors could always create an object dynamically using *new*. In such a case, Resource Return idiom could be helpful.

Known Uses

Related Idioms

Resource Return

References

http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.8More C++ Idioms/Named External Argument

Named Loop

Intent

To partially simulate the *labeled loop* feature found in other languages.

Also Known As

Labeled loop

Motivation

Some languages, such as Java and Perl, support labeled looping constructs. In these languages, *break* and *continue* keywords can optionally specify a previously-defined loop label to control the flow of the program. A labeled break brings the control-flow out of the specified loop and similarly, a labeled continue starts with the next iteration. Labels allow breaking and continuing not just the innermost loop, but any outer loop that is labeled. Labeled break and continue can improve the readability and flexibility of complex code which uses nested loops. Named loop idiom in C++ provides partial support for this feature.

Solution and Sample Code

The named loop idiom is implemented using gotos. Macros are used to hide the explicit use of gotos, which is often frowned upon. Only labeled break statements can be simulated using the following two parameterized macros.

```
#define named(blockname) goto blockname; \
                         blockname##_skip: if (0) \
                         blockname:

#define break(blockname) goto blockname##_skip
```

The *named(X)* macro defines two goto labels *X* and *X_skip*. The break macro is a simple goto to the *X_skip* label. An example follows.

```
struct test
{
  std::string str;
  test (std::string s) : str(s) {
    std::cout << "test::test()::" << str << "\n";
  }
  ~test () {
    std::cout << "~test::" << str << "\n";
  }
};

int main(void)
{
  named (outer)
  for (int i = 0; i < 10; i++)
  {
    test t1("t1");
    int j = 0;
    named(inner)
    for (test t2("t2"); j < 5; j++)
    {
      test t3("t3");
      if (j == 1) break(outer);
      if (j == 3) break(inner);
      test t4("t4");
    }
    std::cout << "after inner\n";
  }
  return 0;
}
```

The gotos do not interfere with the proper construction and destruction of objects as confirmed by the output of the above program.

test::test()::t1

test::test()::t2

test::test()::t3

test::test()::t4

~test::t4

~test::t3

test::test()::t3

~test::t3

~test::t2

~test::t1

Known Uses

Related Idioms

References

Named Parameter

Intent

Simulate named (key-value pair) parameter passing style found in other languages instead of position-based parameters.

Also Known As

Method chaining, Fluent interface

Motivation

When a function takes many parameters, the programmer has to remember the types and the order in which to pass them. Also, default values can only be given to the last parameters, so it is not possible to specify one of the later parameters and use the default value for former ones. Named parameters let the programmer pass the parameters to a function in any order and they are distinguished by a name. So the programmer can explicitly pass all the needed parameters and default values without worrying about the order used in the function declaration.

Solution and Sample Code

Named parameter idiom uses a proxy object for passing the parameters. The parameters of the function are captured as data members of the proxy class. The class exposes set methods for each parameter. The set methods return the **this* object by reference so that other set methods can be chained together to set remaining parameters.

```
class X
{
  public:
    int a;
    char b;
```

```
    X() : a(-999), b('C') {}  // Initialize with default values, if any.
    X & setA(int i) { a = i; return *this; } // non-const function
    X & setB(char c) { b = c; return *this; } // non-const function

    static X create() {
      return X();
    }
};

std::ostream & operator << (std::ostream & o, X const & x)
{
    o << x.a << " " << x.b;
    return o;
}

int main (void)
{
  // The following code uses the named parameter idiom.
  std::cout << X::create().setA(10).setB('Z') << std::endl;
}
```

Known Uses

- bgl_named_params template in the Boost Graph Library (BGL) (http://beta.boost.org/doc/libs/1_33_1/libs/graph/doc/bgl_named_params.html)
- boost::parameter (http://boost.org/libs/parameter/doc/html/index.html)

Related Idioms

- Named Template Parameters

References

- Named Parameter Idiom (http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.20), Marshal ClineMore C++ Idioms/Named Template Parameters

Nifty Counter

Intent

Ensure a non-local static object is initialized before its first use and destroyed only after last use of the object.

Also Known As

Schwarz Counter

Motivation

When static objects use other static objects, the initialization problem becomes more complex. A static object must be initialized before its use if it has non-trivial initialization. Initialization order of static objects across compilation units is not well-defined. Multiple static objects, spread across multiple compilation units, might be using a single static object. Therefore, it must be initialized before use. One example is std::cout, which is typically used by a number of other static objects.

Solution and Sample Code

The "nifty counter" or "Schwarz counter" idiom is an example of a reference counting idiom applied to the initialization of static objects.

```
// Stream.h
#ifndef STREAM_H
#define STREAM_H

struct Stream {
  Stream ();
  ~Stream ();
};
extern Stream& stream; // global stream object

static struct StreamInitializer {
  StreamInitializer ();
  ~StreamInitializer ();
} streamInitializer; // static initializer for every translation unit

#endif // STREAM_H
```

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

#include <new>          // placement new
#include <type_traits> // aligned_storage

static int nifty_counter; // zero initialized at load time
static typename std::aligned_storage<sizeof (Stream), alignof (Stream)>::type
  stream_buf; // memory for the stream object
Stream& stream = reinterpret_cast<Stream&> (stream_buf);

Stream::Stream ()
{
  // initialize things
}
Stream::~Stream ()
{
  // clean-up
}

StreamInitializer::StreamInitializer ()
{
  if (nifty_counter++ == 0) new (&stream) Stream (); // placement new
}
StreamInitializer::~StreamInitializer ()
{
  if (--nifty_counter == 0) (&stream)->~Stream ();
}
```

The header file of the Stream class must be included before any member function can be called on the Stream object. An instance of the StreamInitializer class is included in each compilation unit. Any use of the Stream object follows the inclusion of the header, which ensures that the constructor of the initializer object is called before the Stream object is used.

The Stream class' header file declares a reference to the Stream object. In addition this reference is extern, meaning it is defined in one translation unit and accesses to it are resolved by the linker rather than the compiler.
The implementation file for the Stream class finally defines the Stream object, but in an unusual way: it first defines a static (i.e. local to the translation unit) buffer. This buffer is both, properly aligned and big enough, to store an object of type Stream. The reference to the Stream object defined in the header is then set to point to this buffer.
This buffer workaround enables fine-grained control of when the Stream object's constructor and destructor are called. In the example above, the constructor is called within the constructor of the first StreamInitializer object - using placement new to place it within the buffer. The Stream object's destructor is called, when the last StreamInitializer is destroyed,

This workaround is necessary because defining a Stream variable within Stream.cpp - be it static or not - will define it after the StreamInitializer, which is defined by including the header. Then, the StreamInitializer's constructor is run before Stream's constructor and even worse, the initializer's destructor is run after the Stream object's destructor. The buffer solution above avoids this.

Known Uses

Standard C++ iostream library std::cout, std::cin, std::cerr, std::clog.

Related Idioms

- Reference Counting

References

- http://www.petebecker.com/js/js199905.html

Non-copyable Mixin

Intent

To prevent objects of a class from being copy-constructed or assigned to each other.

Also Known As

Motivation

Many times it makes sense to prevent copying of objects of a class. For example, a class that encapsulates network connections. Copying can't be meaningfully defined for such classes. So it should be explicitly prevented without relying on guidelines or discipline on the programmer's part. The intent should also be easily identifiable just by looking at the declaration of a class to improve readability.

Solution and Sample Code

A class called non-copyable is defined which has a private copy constructor and copy assignment operator.

```
class NonCopyable
{
  protected:
    NonCopyable () {}
    ~NonCopyable () {} /// Protected non-virtual destructor
  private:
    NonCopyable (const NonCopyable &);
    NonCopyable & operator = (const NonCopyable &);
};
class CantCopy : private NonCopyable
{};
```

CantCopy objects can't be copied because the copy constructor and copy assignment operators of the private base class NonCopyable are not accessible to the derived class. The traditional way to deal with these is to declare a private copy constructor and copy assignment, and then document why this is done. But deriving from noncopyable is simpler and clearer, and doesn't require additional documentation. private members of NonCopyable need not be defined. NonCopyable can also be categorized as a mixin-from-above because it defines a reusable module that "mixes-in" the feature of "non-copyability" into the derived class from "above". A CRTP based solution is given below. (This allows for Empty Base Optimization with multiple inheritance [see the Discussion page].)

```
template <class T>
class NonCopyable
{
  protected:
```

```
    NonCopyable () {}
    ~NonCopyable () {} /// Protected non-virtual destructor
  private:
    NonCopyable (const NonCopyable &);
    NonCopyable & operator = (const NonCopyable &);
};
class CantCopy : private NonCopyable <CantCopy>
{};
```

Explicit copy-constructor

It is worthwhile to note that C++ allows one more way to control copy-construction. An **explicit** copy-constructor will prevent the compiler from calling a copy-constructor implicitly. I.e., passing an object by value and returning an object by value will be disabled for the class having an explicit copy-constructor. However, making explicit copies is allowed. On gcc, use -fno-elide-constructors option to disable copy-elision. This option is useful to observe (actually not observe!) the effects of Return Value Optimization. It is generally not recommended to disable this important optimization.

```
struct NoImplicitCopy
{
  NoImplicitCopy () {}
  explicit NoImplicitCopy (const NoImplicitCopy &) {}
};

NoImplicitCopy foo()   // Compiler error because copy-constructor must be invoked implicitly to return by value.
{
  NoImplicitCopy n;
  return n;
}

void bar(NoImplicitCopy n)   // Compiler error because copy-constructor must be invoked implicitly to pass by value.
{
}

int main(void)
{
  NoImplicitCopy n;
  NoImplicitCopy x(n);       // This is fine. explicit copy.
  n = foo();
  bar(n);
}
```

Known Uses

boost::noncopyable (http://www.boost.org/libs/utility/utility.htm#Class_noncopyable)

Related Idioms

Curiously Recurring Template Pattern

References

1. ISO/IEC 14882:2003 C++ Standard §12.8/4, §12.8/7More C++ Idioms/Non-member get

More C++ Idioms/Non-member Non-friend Function

Non-throwing swap

Intent

- To implement an exception safe and efficient swap operation.
- To provide uniform interface to it to facilitate generic programming.

Also Known As

- Exception safe swap

Motivation

A typical implementation of swap could be given as follows:

```
template<class T>
void swap (T &a, T &b)
{
  T temp (a);
  a = b;
  b = temp;
}
```

This can be problematic for two reasons:

Performance
: Swapping of two large, complex objects of the same type is inefficient due to acquisition and release of resources for the intermediate temporary object.

Exception-safety
: This generic swap implementation may throw if resources are not available. (Such a behavior does not make sense where in fact no new resources should have been requested in the first place.) Therefore, this implementation cannot be used for the Copy-and-swap idiom.

Solution and Sample Code

Non-throwing swap idiom uses Handle Body idiom to achieve the desired effect. The abstraction under consideration is split between two implementation classes. One is handle and other one is body. The handle holds a pointer to a body object. The swap is implemented as a simple swap of pointers, which are guaranteed to not throw exceptions and it is very efficient as no new resources are acquired or released.

```
namespace Orange {
class String
{
    char * str;
  public:
    void swap (String &s) // throw ()
    {
      std::swap (this->str, s.str);
    }
};
}
```

Although an efficient and exception-safe swap function can be implemented (as shown above) as a member function, non-throwing swap idiom goes further than that for simplicity, consistency, and to facilitate generic programming. An explicit specialization of std::swap should be added in the std namespace as well as the namespace of the class itself.

```
namespace Orange { // namespace of String
  void swap (String & s1, String & s2) // throw ()
  {
    s1.swap (s2);
  }
}
namespace std {
  template <>
  void swap (Orange::String & s1, Orange::String & s2) // throw ()
  {
    s1.swap (s2);
```

```
  }
}
```

Adding it in two places takes care of two different common usage styles of swap (1) unqualified swap (2) fully qualified swap (e.g., std::swap). When unqualified swap is used, right swap is looked up using Koenig lookup (provided one is already defined). If fully qualified swap is used, Koenig lookup is suppressed and one in the std namespace is used instead. It is a very common practice. Remaining discussion here uses fully qualified swap only. It gives a uniform look and feel because C++ programmers often use swap function in an idiomatic way by fully qualifying it with **std::** as shown below.

```
template <class T>
void zoo (T t1, T t2) {
//...
int i1, i2;
std::swap(i1,i2); // note uniformity
std::swap(t1,t2); // Ditto here
}
```

In such a case, the right, efficient implementation of swap is chosen when zoo is instantiated with String class defined earlier. Otherwise, the default std::swap function template would be instantiated -- completely defeating the purpose of defining the member swap and namespace scope swap function. This idiom of defining explicit specialization of swap in std namespace is particularly useful in generic programming.

Such uniformity in using non-throwing swap idiom leads to its cascading use as given in the example below.

```
class UserDefined
{
    String str;
  public:
    void swap (UserDefined & u) // throw ()
    {
      std::swap (str, u.str);
    }
};
namespace std
{
  // Full specializations of the templates in std namespace can be added in std namespace.
  template <>
  void swap (UserDefined & u1, UserDefined & u2) // throw ()
  {
    u1.swap (u2);
  }
}
class Myclass
{
    UserDefined u;
    char * name;
  public:
    void swap (Myclass & m) // throw ()
    {
      std::swap (u, m.u);       // cascading use of the idiom due to uniformity
      std::swap (name, m.name); // Ditto here
    }
}
namespace std
{
   // Full specializations of the templates in std namespace can be added in std namespace.
   template <>
   void swap (Myclass & m1, Myclass & m2) // throw ()
   {
     m1.swap (m2);
   }
};
```

Therefore, it is a good idea to define a specialization of std::swap for the types that are amenable to an exception safe, efficient swap implementation. The C++ standard does not currently allow us to add new templates to the std namespace, but it does allow us to specialize templates (e.g. std::swap) from that

namespace and add them back in it.

Caveats

Using non-throwing swap idiom for template classes (e.g., Matrix<T>) can be a subtle issue. As per the C++98 standard, only the full specialization of std::swap is allowed to be defined inside std namespace for the user-defined types. Partial specializations or function overloading is not allowed by the language. Trying to achieve the similar effect for template classes (e.g., Matrix<T>) results into overloading of std::swap in std namepspace, which is technically undefined behavior. This is not necessarily the ideal state of affairs as indicated by some people in a spectacularly long discussion thread on comp.lang.c++.moderated newsgroup.[1] There are two possible solutions, both imperfect, to this issue:

1. Standard-compliant solution. Leveraging on Koenig lookup, define an overloaded swap function template in the same namespace as that of the class being swapped. Not all compilers may support this correctly, but this solution is compliant to the standard.[2]
2. Fingers-crossed solution. Partially specialize std::swap and ignore the fact that this is technically undefined behavior, hoping that nothing will happen and wait for a fix in the next language standard.

Known Uses

All boost smart pointers (e.g., boost::shared_ptr)

Related Idioms

- Copy-and-swap

References

1. "Namespace issues with specialized swap". comp.lang.c++.moderated (Usenet newsgroup). 12 March 2000. http://groups.google.ca/group/comp.lang.c++.moderated/browse_thread/thread/b396fedad7dcdc81.
2. Sutter, Herb; Alexandrescu, Andrei (25 October 2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth. Addison Wesley Professional. ISBN 0-321-11358-6. Item 66: Don't specialize function templates.

Non-Virtual Interface

Intent

- To modularize/refactor common before and after code fragments (e.g., invariant checking, acquiring/releasing locks) for an entire class hierarchy at one location.

Also Known As

- Template Method - a more generic pattern, from the Gang of Four's Design Patterns book.

Motivation

Pre- and post-condition checking is known to be a useful object-oriented programming technique particularly at development time. Pre- and post-conditions ensure that invariants of a class hierarchy (and in general an abstraction) are not violated at designated points during execution of a program. Using them at development time or during debug builds helps in catching violations earlier. To maintain consistency and ease of

maintenance of such pre- and post-conditions, they should be ideally modularized at one location. In a class hierarchy, where its invariants must be held before and after every method in the subclasses, modularization becomes important.

Similarly, acquiring and releasing locks on a data structure common to a class hierarchy can be considered as a pre- and post-condition, which must be ensured even at production time. It is useful to separate the responsibility of lock acquiring and releasing from subclasses and put it at one place - potentially in the base class.

Solution and Sample Code

Non-Virtual Interface (NVI) idiom allows us to refactor before and after code fragments at one convenient location - the base class. The NVI idiom is based on 4 guidelines outlined by Herb Sutter in his article named "Virtuality"[2]. Quoting Herb:

- Guideline #1: Prefer to make interfaces nonvirtual, using Template Method design pattern.
- Guideline #2: Prefer to make virtual functions private.
- Guideline #3: Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.
- Guideline #4: A base class destructor should be either public and virtual, or protected and nonvirtual. - Quote complete.

Here is some code that implements the NVI idiom following the above 4 guidelines.

```
class Base {
private:
    ReaderWriterLock lock_;
    SomeComplexDataType data_;
public:
    void read_from( std::istream & i)  { // Note non-virtual
      lock_.acquire();
      assert(data_.check_invariants()); // must be true

      read_from_impl(i);

      assert(data_.check_invariants()); // must be true
      lock_.release();
    }
    void write_to( std::ostream & o) const { // Note non-virtual
      lock_.acquire();
      write_to_impl(o);
      lock_.release();
    }
    virtual ~Base() {}  // Virtual because Base is a polymorphic base class.
private:
    virtual void read_from_impl( std::istream & ) = 0;
    virtual void write_to_impl( std::ostream & ) const = 0;
};
class XMLReaderWriter : public Base {
private:
    virtual void read_from_impl (std::istream &) {
      // Read XML.
    }
    virtual void write_to_impl (std::ostream &) const {
      // Write XML.
    }
};
class TextReaderWriter : public Base {
private:
    virtual void read_from_impl (std::istream &) {}
    virtual void write_to_impl (std::ostream &) const {}
};
```

The above implementation of the base class captures several design intents that are central to achieving the benefits of the NVI idiom. This class intends to be used as a base class and therefore, it has a virtual destructor and some pure virtual functions (read_from_impl, write_to_impl), which must be implemented by all the

concrete derived classes. The interface for clients (i.e., read_from and write_to) is separate from the interface for the subclasses (i.e. read_from_impl and write_to_impl). Although the read_from_impl and write_to_impl are two private functions, the base class can invoke the corresponding derived class functions using dynamic dispatch. These two functions give the necessary extension points to a family of derived classes. However, they are prevented from extending the client interface (read_from and write_to). Note that, it is possible to call interface for clients from the derived classes, however, it will lead to recursion. Finally, the NVI idiom suggests use of exactly one private virtual extension point per public non-virtual function.

Clients invoke only the public interface, which in turn invokes virtual *_impl functions as in the Template Method design pattern. Before and after invoking the *_impl functions, lock operations and invariant checking operations are performed by the base class. In this way, hierarchy wide before and after code fragments can be put together at one place, simplifying maintenance. Clients of the Base hierarchy still get polymorphic behavior even though they don't invoke virtual functions directly. Derived classes should ensure that direct access to the implementation functions (*_impl) is disallowed to the clients by making them private in the derived class as well.

Consequences

Using the NVI idiom may lead to fragile class hierarchies if proper care is not exercised. As described in [1], in Fragile Base Class (FBC) interface problem, subclass's virtual functions may get accidentally invoked when base class implementation changes without notice. For example, the following code snippet (inspired by [1]) uses the NVI idiom to implement CountingSet, which has Set as a base class.

```
class Set {
    std::set<int> s_;
  public:
    void add (int i) {
      s_.insert (i);
      add_impl (i); // Note virtual call.
    }
    void addAll (int * begin, int * end) {
      s_.insert (begin, end);   //  --------- (1)
      addAll_impl (begin, end); // Note virtual call.
    }
  private:
    virtual void add_impl (int i) = 0;
    virtual void addAll_impl (int * begin, int * end) = 0;
};
class CountingSet : public Set {
  private:
    int count_;
    virtual void add_impl (int i) {
      count_++;
    }
    virtual void addAll_impl (int * begin, int * end) {
      count_ += std::distance(begin,end);
    }
};
```

The above class hierarchy is fragile in the sense that during maintenance, if the implementation of the addAll function (indicated by (1)) is changed to call public non-virtual add function for every integer from begin to end, then the derived class, CountingSet, breaks. As addAll calls add, the derived class's extension point add_impl is called for every integer and finally addAll_impl is also called couting the range of integers twice, which silently introduces a bug in the derived class! The solution is to observe a strict coding discipline of invoking exactly one private virtual extension point in any public non-virtual interface of the base class. However, the solution depends on programmer discipline and hence difficult to follow in practice.

Note how the NVI idiom treats each class hierarchy as a tiny (some may like to call trivial) object-oriented framework, where inversion of control (IoC) flow is commonly observed. Frameworks control the flow of the program as opposed to the functions and classes written by the client, which is why it is known as inversion of

control. In NVI, the base class controls the program flow. In the example above, the Set class does the required common job of insertion before calling the *_impl virtual functions (the extension points). The Set class must not invoke any of its own public interface to prevent the FBC problem.

Finally, the NVI idiom leads to a moderate degree of code bloat in the class hierarchy as the number of functions double when the NVI is applied. Size of the refactored code in the base class should be substantial to justify the use of NVI.

Known Uses

Related Idioms

- Interface Class
- Thread-Safe Interface, from Pattern-Oriented Software Architecture (volume 2) by Douglas Schmidt et al.
- Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals [1] (http://www.parashift.com/c++-faq-lite/strange-inheritance.html#faq-23.3)

References

[1] Selective Open Recursion: Modular Reasoning about Components and Inheritance - Jonathan Aldrich, Kevin Donnelly.

[2] Virtuality! (http://www.gotw.ca/publications/mill18.htm) -- Herb Sutter

[3] Conversations: Virtually Yours (http://www.ddj.com/cpp/184403760) -- Jim Hyslop and Herb Sutter

[4] Should I use protected virtuals instead of public virtuals? (http://www.parashift.com/c++-faq/protected-virtuals.html) -- Marshall Cline

nullptr

Intent

To distinguish between an integer 0 and a null pointer.

Also Known As

Motivation

For many years C++ had an embarrassment of not having a keyword to designate a null pointer. C++11 has eliminated that embarrassment. C++'s strong type checking makes C's NULL macro almost useless in expressions, e.g.:

```
// if the following were a valid definition of NULL in C++
#define NULL ((void *)0)

// then...

char * str = NULL; // Can't automatically convert void * to char *
void (C::*pmf) () = &C::func;
if (pmf == NULL) {} // Can't automatically convert from void * to pointer to member function.
```

Instead,

```
#define NULL 0
```

or

```
#define NULL 0L
```

are valid definitions for NULL in C++. See below.

The crux of the matter, in fact, is that C++ disallows conversions from void *, even when the value is a constant zero, **but**, for constant zero, introduces a special case anyway: *int* to pointer (actually several of them: *short* to pointer, *long* to pointer, etc.). This is, in fact, even worse than allowing (for the constant case) the former exception, the more so given the support of function overloading in the language.

```
void func(int);
void func(double *);
int main()
{
  func (static_cast <double *>(0)); // calls func(double *) as expected
  func (0); // calls func(int) but the programmer might have intended double *, because 0 IS also a null pointer constant
}
```

The use of the macro NULL has its own set of problems as well. C++ requires that macro NULL be defined as an integral constant expression having the value of 0. So unlike in C, NULL cannot be defined as (void *)0 in the C++ standard library. Furthermore, the exact form of definition is left to the particular implementation, which means that e.g. both 0 and 0L are viable options, among some others. This is a trouble as it can cause confusion in overload resolution. Worse, the way confusing overload resolution manifests itself will vary depending on the compiler and its settings used. An illustrative case is shown in this slight modification of the example above:

```
#include <cstddef>

void func(int);
void func(double *);

int main()
{
  func (static_cast <double *>(0)); // calls func(double *) as expected
  func (0); // calls func(int) but double * may be desired because 0 IS also a null pointer

  func (NULL) // calls func(int) if NULL is defined as 0 (confusion, func(double *) was intended!) - logic error at runtim
              // but the call is ambiguous if NULL is defined as 0L (yet more confusion to the unwary!) - compilation erro
}
```

Solution and Sample Code

The nullptr idiom solves some of the above problems and can be put in reusable form, as a library solution. It is a very close approximation of a "null keyword", by using only pre-C++11 standard features.

The following is such a library solution as mostly found in Effective C++ by Scott Meyers, Second Edition, Item 25 (it is not present in the third edition of the book).

```
const // It is a const object...
class nullptr_t
{
  public:
    template<class T>
```

```
    inline operator T*() const // convertible to any type of null non-member pointer...
    { return 0; }

    template<class C, class T>
    inline operator T C::*() const   // or any type of null member pointer...
    { return 0; }

  private:
    void operator&() const;  // Can't take address of nullptr

} nullptr = {};
```

The following code illustrates some usage cases (and assumes the class template above has already been #included).

```
#include <typeinfo>

struct C
{
  void func();
};

template<typename T>
void g( T* t ) {}

template<typename T>
void h( T t ) {}

void func (double *) {}
void func (int) {}

int main(void)
{
  char * ch = nullptr;        // ok
  func (nullptr);             // Calls func(double *)
  func (0);                   // Calls func(int)
  void (C::*pmf2)() = 0;      // ok
  void (C::*pmf)() = nullptr; // ok
  nullptr_t n1, n2;
  n1 = n2;
  //nullptr_t *null = &n1;    // Address can't be taken.
  if (nullptr == ch) {}       // ok
  if (nullptr == pmf) {}      // Valid statement; but fails on g++ 4.1.1-4.5 due to bug #33990
// for GCC 4: if ((typeof(pmf))nullptr == pmf) {}
  const int n = 0;
  if (nullptr == n) {}        // Should not compile; but only Comeau shows an error.
  //int p = 0;
  //if (nullptr == p) {}      // not ok
  //g (nullptr);              // Can't deduce T
  int expr = 0;
  char* ch3 = expr ? nullptr : nullptr; // ch3 is the null pointer value
  //char* ch4 = expr ? 0 : nullptr;     // error, types are not compatible
  //int n3 = expr ? nullptr : nullptr;  // error, nullptr can't be converted to int
  //int n4 = expr ? 0 : nullptr;        // error, types are not compatible

  h( 0 );               // deduces T = int
  h( nullptr );         // deduces T = nullptr_t
  h( (float*) nullptr ); // deduces T = float*

  sizeof( nullptr );    // ok
  typeid( nullptr );    // ok
  throw nullptr;        // ok
}
```

Unfortunately, there seems to be a bug (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=33990) in gcc 4.1.1 compiler that does not recognize the comparison of nullptr with point to member function (pmf). The above code compiles on VC++ 8.0 and Comeau 4.3.10.1 beta.

Note that nullptr idioms makes use of the Return Type Resolver idiom to automatically deduce a null pointer of the correct type depending upon the type of the instance it is assigning to. For example, if nullptr is being assigned to a character pointer, a char type instantiation of the templatized conversion function is created.

Consequences

There are some disadvantages of this technique and are discussed in the N2431 proposal (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2431.pdf) draft. In summary, the disadvantages are

- A header must be included to use nullptr. In C++11, nullptr itself is a keyword and requires no headers (although std::nullptr_t does).
- Compilers have historically produced (arguably) unsatisfactory diagnostics when using the code proposed.

Known Uses

Related Idioms

- Return Type Resolver

References

- ISO C++11
- Effective C++, 2nd edition
- N2431: nullptr proposal (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2431.pdf)

Object Generator

Intent

To simplify creation of objects without explicitly specifying their types. (This is not the factory method pattern)

Also Known As

Motivation

In C++ template programming, types of objects can get really large and incomprehensible even in small programs. For example, the following type (*Wrapper*) is a standard unary function object that wraps the member function *read_line* in class *File*.

```
struct File
{
  int read_line (std::string);
};
typedef std::mem_fun1_t<int, File, std::string> Wrapper;
```

Reading a collection of files using the *for_each* STL algorithm, without object generators, looks like the following:

```
void read_lines(std::vector<File *> files)
{
   typedef std::mem_fun1_t<int, File, std::string> Wrapper;
   std::string arg;
   for_each(files.begin(), files.end(),
      std::binder2nd<Wrapper>(Wrapper(&File::read_line), arg));
}
```

The above code is pretty much unreadable and more bloated than necessary. Even typedefs don't improve readability as placeholder typedefs like *Wrapper* are distracting. The *object generator* idiom alleviates the situation.

Solution and Sample Code

In the *object generator* idiom, a template function is created whose only job is to construct a new object from its parameters. It is based on a useful property of function templates which class templates don't have: The type parameters of a function template are deduced automatically from its actual parameters. For example, consider a simple object generator defined in STL: *make_pair*.

```
template <class T, class U>
std::pair <T, U>
make_pair(T t, U u)
{
  return std::pair <T, U> (t,u);
}
```

make_pair returns an instance of the *pair* template depending on the actual parameters of the *make_pair* function. For example, *make_pair(1,1.0)* creates an object of type: *std::pair<int, double>* by automatically deducing the types of the objects being passed to the object generator function. *make_pair* is particularly handy in the case where a generated pair object does not need to be stored in a local variable.

```
map <int, double> m;
m.insert (make_pair(1,1.0)); // No need to know how pair template is instantiated.
```

The C++ standard library defines several object generators to avoid code bloat. *std::bind2nd* and *std::mem_fun* are two such standard object generators that can be used to avoid code bloat in the example shown in the motivation section above.

```
void read_lines(std::vector<File *> files)
{
   std::string arg;
   for_each(files.begin(), files.end(), bind2nd(mem_fun(&File::read_line), arg));
}
```

Known Uses

C++ standard library (mem_fun, make_pair, bind1st, bind2nd etc.)

Related Idioms

Type Generator

References

Object Generator (http://www.boost.org/community/generic_programming.html#object_generator)

Intent

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

References

Intent

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

References

Parameterized Base Class

Intent

To abstract out an aspect in a reusable module and combine it in a given type when required.

Also Known As

- Mixin-from-below
- Parameterized Inheritance

Motivation

A certain aspect can be abstracted out from requirements and be developed as templates (e.g., object serialization). Serialization is a cross-cutting concern that many classes/POD types in an application may have. Such a concern can be abstracted out in a manageable reusable module. By addition of an aspect, substitutability with the original type is not broken so another motivation is to have a IS-A (public inheritance) or WAS-A (private inheritance) relationship with the type parameter.

Solution and Sample Code

```
template <class T>
class Serializable : public T,   /// Parameterized Base Class Idiom
                     public ISerializable
{
  public:
    Serializable (const T &t = T()) : T(t) {}
    virtual int serialize (char *& buffer, size_t & buf_size) const
    {
        const size_t size = sizeof (T);
        if (size > buf_size)
          throw std::runtime_error("Insufficient memory!");

        memcpy (buffer, static_cast<const T *>(this), size);
        buffer += size;
        buf_size -= size;
        return size;
    }
};
```

Serializable <T> can be used polymorphically as a T as well as a ISerializable. Above example works correctly only if T is a user-defined POD type without pointers.

Known Uses

Related Idioms

- Curiously Recurring Template Pattern

References

Idioms

Handling self-assignment in an assignment operator

`T::operator=` that handles the case where the LHS and the RHS refer to the same object.

```
T& operator= (const T& that)
{
    if (this == &that)
        return *this;

    // handle assignment here

    return *this;
}
```

Notes:

- Keep in mind the differences between *identity* (the LHS and the RHS are the same object) and *equality* (the LHS and the RHS have the same value). `T::operator=` has to protect itself against *identity* since then, the code for assigning can conveniently and safely assume that the LHS and the RHS refer to different objects.
- There are other techniques that are superior but not applicable in all situations. For example, if all the members of the class `T` (say, `mem1`, `mem2`, ..., `memN`) provide a `swap` function, one could use the following code instead:

```
T& operator= (T that)
{
    // that is constructed by the copy constructor

    mem1.swap (that.mem1);
    mem2.swap (that.mem2);
```

```
    ...

    memN.swap (that.memN);

    // now what were originally this->mem1, this->mem2, etc. get
    // destroyed when that gets destroyed, and that.mem1, etc. are
    // retained in *this

    return *this;
}
```

Pointer To Implementation (pImpl)

The "pointer to implementation" (pImpl) idiom, also called the "opaque pointer" idiom, is a method of providing data and thus further implementation abstraction for Classes.

In C++ you have to declare member variables within the class definition which is then public and that this is necessary so that an appropriate memory space is allocated means that abstraction of implementation is not possible in "all" classes.

However, at the cost of an extra pointer dereference and function call, you can have this level of abstraction through the Pointer to Implementation.

```
class Book
{
public:
  void print();
private:
  std::string  m_Contents;
}
```

So somebody who works with the Book class only really needs to know about print(), but what happens if you wish to add more detail to your book class.

```
class Book
{
public:
  void print();
private:
  std::string  m_Contents;
  std::string  m_Title;
}
```

Now all the users of book have to recompile because the object they know about got bigger, yet they still only call print().

pImpl would implement the following pattern so that this would not be a problem.

```
/* public.h */
class Book
{
public:
  Book();
  ~Book();
  void print();
private:
  class BookImpl;
  BookImpl* const m_p;
}
```

and in a separate 'internal' header

```
/* private.h */
#include "public.h"
#include <iostream>
class Book::BookImpl
{
public:
  void print();
private:
  std::string  m_Contents;
  std::string  m_Title;
}
```

The body of the Book class then would look something like

```
Book::Book(): m_p(new BookImpl())
{
}

Book::~Book()
{
  delete m_p;
}

void Book::print()
{
  m_p->print();
}

/* then BookImpl functions */

void Book::BookImpl::print()
{
  std::cout << "print from BookImpl" << std::endl;
}
```

then invoked from a main function:

```
int main()
{
  Book b;
  b.print();
}
```

You could also use std::unique_ptr<BookImpl> or equivalent to manage the internal pointer.

Further reading

- More C++ Idioms
- Wikipedia: opaque pointer

Policy Clone

Intent

Instantiate a policy class with many different possible types without ad-hoc limitations on type of the policy classes.

Also Known As

- Meta-function wrapper idiom

Motivation

Highly reusable, flexible and extensible classes can be built using policy based class design techniques. Sometimes, the host class of the policies needs to make an exact replica of one of its policies which is instantiated with a different type parameter. Unfortunately, the writer of the host class template does not know the template name to instantiate beforehand. Moreover, the policy class may or may not be a template in the first place. If it is a template, then the host class may not know how many minimum type parameters are required to instantiate the parameterized policy class. If the policy class is not a template, it may not be able to participate as a policy class. This situation is quite analogous to the situation in the Factory Method (GoF) pattern where type of the object to be created is not known a priori.

```
template <class Apolicy>
class Host
{
  Apolicy direct_policy_use;
  Apolicy <SomeInternalType> InternalClone;  // Problem 1: Can't do this
};

template <class T, template <class T> class Apolicy>
class Host2
{
  Apolicy <T> common_use;
  Apolicy <SomeInternalType> InternalClone;
  // Can do this now but
  // Problem 2: policies that require more than one type parameter can't participate.
};
```

Solution and Sample Code

A member template struct (called rebind) is used to pass a different type parameter to the policy class template. For example,

```
template <typename T>
class NiftyAlloc
{
  public:
    template <typename Other>
    struct rebind // The Policy Clone idiom
    {
       typedef NiftyAlloc <Other> other;
    };
    //...
};

template <typename T, class Alloc = NiftyAlloc <T> >
class Vector
{
  public:
    typedef typename Alloc::template rebind<long>::other ClonePolicy;
    // Here, Alloc may not be a template class or a parametrized instantiation of
    // a class that takes unknown number of type parameters.
};
```

Here, the Container template needs a replica of the allocation policy it is instantiated with. Therefore, it uses the rebind mechanism exposed by the NiftyAlloc policy. The type Alloc::template rebind<long>::other is same as NiftyAlloc<long>. Essentially, it says, "I don't know what kind of allocator this type is, and I don't know what it allocates, but I want an allocator just like it that allocates longs." Using concepts（it has been removed from C++0x), the Vector class can write type concept that checks whether the Alloc policy type supports rebind concept.

To keep the compiler happy, we have to use both the keywords typename and template in the ClonePolicy typedef. The rule is as follows: If the name of a member template specialization appears after a ., ->, or :: operator, and that name has explicitly qualified template parameters, prefix the member template name with the

keyword template. The Keyword typename is also necessary in the typedef because "other" is a type, not a variable.

Known Uses

- Standard Template Library
- Compilers that do not support template template parameters

Related Idioms

Meta-function wrapper idiom is a more powerful idiom than policy Clone. Policy Clone idiom indicates its purpose in a more abstract fashion than meta-function wrapper. The rebind template is essentially the meta-function wrapper.

References

- Modern C++ Design - by Andrei Alexandrescu.
- C++ Understand rebind (http://meditation-art.blogspot.com/2007/11/c-understand-rebind.html)More C++ Idioms/Policy-based Design

Polymorphic Exception

Intent

- To create an exception object polymorphically
- To decouple a module from the concrete details of the exceptions it may throw

Also Known As

Motivation

Dependency Inversion Principle (DIP) (http://en.wikipedia.org/wiki/Dependency_inversion_principle), a popular object-oriented software design guideline states that higher level modules should not depend directly on lower level modules. Instead, both should depend on common abstractions (captured in the form of well-defined interfaces). For example, an object of type *Person* (say John) should not create and use an object of type *HondaCivic* but instead John should simply commit to a *Car* interface, which is an abstract base class of *HondaCivic*. This allows John to upgrade to a *Corvette* easily in future without any changes to the class *Person*. John can be "configured" with a concrete instance of a car (any car) using dependency injection (http://en.wikipedia.org/wiki/Dependency_injection) technique. Use of DIP leads to flexible and extensible modules that are easy to unit test. Unit testing is simplified by DIP because real objects can be easily replaced with mock objects using dependency injection.

However, there are several occasions when DIP is violated: (1) while using the Singleton pattern and (2) while throwing exceptions! Singleton pattern breaks DIP because it forces the use of the concrete class name while accessing the static *instance()* function. A singleton should be passed as a parameter while calling a function or a constructor. A similar situation arises while dealing with exceptions in C++. The *throw* clause in C++ requires a concrete type name (class) to raise an exception. For example,

```
throw MyConcreteException("Big Bang!");
```

Any module that throws exceptions like this immediately results into a violation of DIP. Naturally, it is harder to unit test such a module because real exception objects cannot easily be replaced with mock exception objects. A solution like the one below fails miserably as *throw* in C++ uses static typing and knows nothing about polymorphism.

```
struct ExceptionBase { };
struct ExceptionDerived : ExceptionBase { };

void foo(ExceptionBase& e)
{
   throw e; // Uses static type of e while rasing an exception.
}
int main (void)
{
  ExceptionDerived e;
  try {
    foo(e);
  }
  catch (ExceptionDerived& e) {
    // Exception raised in foo does not match this catch.
  }
  catch (...) {
    // Exception raised in foo is caught here.
  }
}
```

Polymorphic exception idiom addresses the issue.

Solution and Sample Code

Polymorphic exception idiom simply delegates the job of raising the exception back to the derived class using a virtual function *raise()*

```
struct ExceptionBase
{
  virtual void raise() { throw *this; }
  virtual ~ExceptionBase() {}
};
struct ExceptionDerived : ExceptionBase
{
  virtual void raise() { throw *this; }
};

void foo(ExceptionBase& e)
{
   e.raise(); // Uses dynamic type of e while raising an exception.
}
int main (void)
{
  ExceptionDerived e;
  try {
    foo(e);
  }
  catch (ExceptionDerived& e) {
    // Exception raised in foo now matches this catch.
  }
  catch (...) {
    // not here anymore!
  }
}
```

The *throw* statement has been moved into virtual functions. The *raise* function invoked in function *foo* is polymorphic and selects the implementation in either *ExceptionBase* or *ExceptionDerived* class depending upon what is passed as a parameter (dependency injection). Type of **this* is obviously known at compile-time, which results into raising a polymorphic exception. The structure of this idiom is very similar to that of Virtual Constructor idiom.

Propagating a polymorphic exception

Quite often an exception is handled in multiple catch statements to treat it differently in different layers of the program/library. In such cases, the earlier catch blocks need to rethrow the exception so that the outer catch blocks, if any, may take the necessary actions. When a polymorphic exception is involved, inner catch block(s) may modify the exception object before passing it on to the catch blocks up in the stack. In such cases, care must be taken to ensure that the original exception object is propagated. Consider a seemingly innocuous looking program below, which fails to do that.

```
try {
    foo(e); // throws an instance of ExceptionDerived as before.
  }
  catch (ExceptionBase& e) // Note the base class. Exception is caught polymorphically.
  {
    // Handle the exception. May modify the original exception object.
    throw e; // Warning! Object slicing is underway.
  }
```

The *throw e* statement does not throw the original exception object. Instead, it throws a sliced copy (only ExceptionBase part) of the original object because it considers the static type of the expression in front of it. Silently, the derived exception object was not only lost but also translated into the base type exception object. The catch blocks up in the stack do not have access to the same information this catch had. There are two ways to address the problem.

- Simply use *throw;* (without any expression following it). It will rethrow the original exception object.
- Use polymorphic exception idiom again. It will throw a copy of the original exception object because the *raise()* virtual function uses *throw *this*.

Use of *throw;* should be strongly preferred in practice, because depending on the implementation, it may preserve the original throw-location in case the exception goes unhandled and the program dumps core, thus easing post mortem analysis of the problem.

```
try {
    foo(e); // throws an instance of ExceptionDerived as before.
  }
  catch (ExceptionBase& e) // Note the base class. Exception is caught polymorphically.
  {
    // Handle the exception. May modify the original exception object.
    // Use only one of the following two.
    throw;      // Option 1:  Original derived exception is thrown.
    e.raise();  // Option 2:  A copy of the original derived exception object is thrown.
  }
```

Known Uses

Related Idioms

Virtual Constructor

References

- How do I throw polymorphically? (parashift) (http://www.parashift.com/c++-faq-lite/exceptions.html#faq-17.10)
- How do I throw polymorphically? (NCTU) (http://www.cis.nctu.edu.tw/chinese/doc/research/c++/C++FAQ-English/exceptions.html#faq-17.10)

Polymorphic Value Types

Intent

Support run-time polymorphism of value types without requiring inheritance relationship.

Also Known As

Run-time Concept

Motivation

Solution and Sample Code

Known Uses

Adobe Poly Library (http://stlab.adobe.com/group__poly__related.html)

Related Idioms

- Type Erasure

References

Sean Parent: Value Semantics and Concepts-based Polymorphism (https://github.com/boostcon/cppnow_presentations_2012/blob/master/fri/value_semantics/value_semantics.pdf?raw=true) @ C++ Now! 2012, Aspen, CO

Inheritance Is The Base Class of Evil (https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil)

Recursive Type Composition

Intent

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

References

Compile Time Recursive Objects in C++ (http://portal.acm.org/citation.cfm?id=832912) - Jaakko Jarvi

Requiring or Prohibiting Heap-based Objects

Intent

- Require or prevent creation of objects on heap

Also Known As

Motivation

C++ supports different ways to create objects: stack-based (including temporary objects), heap-based, and objects of static storage duration (e.g., global objects, namespace-scope objects). Sometimes it is useful to limit the ways in which objects of a class can be created. For instance, a framework may require users to create only heap-based objects so that it can take control of the lifetime of objects regardless of the function that created them. Similarly, it may be useful to prohibit creation of objects on the heap. C++ has idiomatic ways to achieve this.

Solution and Sample Code

Requiring heap-based objects

In this case, programmers must use `new` to create the objects and prohibit stack-based and objects of static duration. The idea is to prevent access to one of the functions that are always needed for stack-based objects: constructors or destructors. Preventing access to constructors, i.e., protected/private constructors will prevent heap-based objects too. Therefore, the only available way is to create a protected destructor as below. Note that a protected destructor will also prevent global and namespace-scope objects because eventually the objects are destroyed but the destructor is inaccessible. The same is applicable to temporary objects because destroying temporary objects need a public destructor.

```
class HeapOnly {
  public:
    HeapOnly() {}
    void destroy() const { delete this; }
  protected:
    ~HeapOnly() {}
};
HeapOnly h1;     // Destructor is protected so h1 can't be created globally
HeapOnly func()  // Compiler error because destructor of temporary is protected
{
  HeapOnly *hoptr = new HeapOnly; // This is ok. No destructor is invoked automatically for heap-based objects
  return *hoptr;
}
int main(void) {
  HeapOnly h2; // Destructor is protected so h2 can't be created on stack
}
```

Protected destructor also prevents access to `delete HeapOnly` because it internally invokes the destructor. To prevent memory leak, `destroy` member function is provided, which calls `delete` on itself. Derived classes have access to the protected destructor so `HeapOnly` class can still be used as a base class. However, the derived class no longer has the same restrictions.

Prohibiting heap-based objects

Dynamic allocation of objects can be prevented by disallowing access to all forms of class-specific `new` operators. The `new` operator for scalar objects and for an array of objects are two possible variations. Both should be declared protected (or private) to prevent heap-based objects.

```
class NoHeap {
protected:
```

```
  static void * operator new(std::size_t);      // #1: To prevent allocation of scalar objects
  static void * operator new [] (std::size_t);  // #2: To prevent allocation of array of objects
};
class NoHeapTwo : public NoHeap {
};
int main(void) {
  new NoHeap;        // Not allowed because of #1
  new NoHeap[1];     // Not allowed because of #2
  new NoHeapTwo[10];  // Not allowed because of inherited protected new operator (#2).
}
```

The above declaration of protected `new` operator prevents remaining compiler-generated versions, such as placement new and nothrow new. Declaring just the scalar `new` as protected is not sufficient because it still leaves the possibility of `new NoHeap[1]` open. Protected `new []` operator prevents dynamic allocation of arrays of all sizes including size one.

Restricting access to the `new` operator also prevents derived classes from using dynamic memory allocation because the new operator and delete operator are inherited. Unless these functions are declared public in a derived class, that class inherits the protected/private versions declared in its base preventing dynamic allocation.

Known Uses

Related Idioms

References

Resource Acquisition Is Initialization

Intent

- To guarantee release of resource(s) at the end of a scope
- To provide basic exception safety guarantee

Also Known As

- Execute-Around Object
- Resource Release Is Finalization
- Scope-Bound Resource Management

Motivation

Resources acquired in a function scope should be released before leaving the scope unless the ownership is being transferred to another scope or object. Quite often it means a pair of function calls - one to acquire a resource and another one to release it. For example, new/delete, malloc/free, acquire/release, file-open/file-close, nested_count++/nested_count--, etc. It is quite easy to forget to write the "release" part of the resource management "contract". Sometimes the resource release function is never invoked: this can happen when the control flow leaves the scope because of return or an exception. It is too dangerous to trust the programmer that he or she will invoke resource release operation in all possible cases in the present and in the future. Some examples are given below.

```
void foo ()
{
  char * ch = new char [100];
  if (...)
     if (...)
        return;
```

```
    else if (...)
           if (...)
  else
    throw "ERROR";

  delete [] ch; // This may not be invoked... memory leak!
}
void bar ()
{
  lock.acquire();
  if (...)
    if (...)
      return;
  else
    throw "ERROR";

  lock.release(); // This may not be invoked... deadlock!
}
```

This is a general control flow abstraction problem. Resource Acquisition is Initialization (RAII) is an extremely popular idiom in C++ that relieves the burden of calling "resource release" operation in a clever way.

Solution and Sample Code

The idea is to wrap the resource release operation in a destructor of an object in the scope. The language guarantees that the destructor will always be invoked (of a successfully constructed object) when control flow leaves the scope because of a return statement or an exception.

```
//  Private copy constructor and copy assignment ensure classes derived
//  from class NonCopyable cannot be copied.
class NonCopyable
{
   NonCopyable (NonCopyable const &); // private copy constructor
   NonCopyable & operator = (NonCopyable const &); // private assignment operator
};
template <class T>
class AutoDelete : NonCopyable
{
  public:
    AutoDelete (T * p = 0) : ptr_(p) {}
    ~AutoDelete () throw() { delete ptr_; }
  private:
    T *ptr_;
};

class ScopedLock : NonCopyable// Scoped Lock idiom
{
  public:
    ScopedLock (Lock & l) : lock_(l) { lock_.acquire(); }
    ~ScopedLock () throw () { lock_.release(); }
  private:
    Lock& lock_;
};

void foo ()
{
  X * p = new X;
  AutoDelete<X> safe_del(p); // Memory will not leak
  p = 0;
  // Although, above assignment "p = 0" is not necessary
  // as we would not have a dangling pointer in this example.
  // It is a good programming practice.

  if (...)
    if (...)
      return;

  // No need to call delete here.
  // Destructor of safe_del will delete memory
}
void X::bar()
{
  ScopedLock safe_lock(l); // Lock will be released certainly
  if (...)
```

```
    if (...)
      throw "ERROR";
  // No need to call release here.
  // Destructor of safe_lock will release the lock
}
```

Acquiring resource(s) in constructor is not mandatory in RAII idiom but releasing resources in the destructor is the key. Therefore, it is also known (rarely though) as **Resource Release is Finalization idiom**. It is important in this idiom that the destructor should not throw exceptions. Therefore, the destructors have no-throw specification but it is optional. std::auto_ptr and boost::scoped_ptr are ways of quickly using RAII idiom for memory resources. RAII is also used to ensure exception safety. RAII makes it possible to avoid resource leaks without extensive use of try/catch blocks and is widely used in the software industry.

Many classes that manage resources using RAII, do not have legitimate copy semantics (*e.g.,* network connections, database cursors, mutex). The *NonCopyable* class shown before prevents copying of objects that implement RAII. It simply prevents access to the copy-constructor and the copy-assignment operator by making them private. *boost::scoped_ptr* is an example of one such class that prevents copying while holding memory resources. The *NonCopyable* class states this intention explicitly and prevents compilation if used incorrectly. Such classes should not be used in STL containers. However, every resource management class that implements RAII does not have to be non-copyable like the above two examples. If copy semantics are desired, *boost::shared_ptr* can be used to manage memory resources. In general, non-intrusive reference counting is used to provide copy semantics as well as RAII.

Consequences

RAII is not without its limitations. The resources which are not memory and must be released deterministically and may throw exceptions usually aren't handled very well by C++ destructors. That's because a C++ destructor can't propagate errors to the enclosing scope (which is potentially winding up). It has no return value and it should not propagate exceptions outside itself. If exceptions are possible, then the destructor must handle the exceptional case within itself somehow. Nevertheless, RAII remains the most widely used resource management idiom in C++.

Known Uses

- Virtually all non-trivial C++ software
- std::unique_ptr (and std::auto_ptr)
- std::shared_ptr
- boost::mutex::scoped_lock

Related Idioms

- Scope Guard
- Reference Counting
- Non copyable
- Scoped Locking (http://www.cs.wustl.edu/~schmidt/PDF/ScopedLocking.pdf) idiom is a special case of RAII applied to operating system synchronization primitives such as mutex and semaphores.

References

- Resource Acquisition Is Initialization on Wikipedia
- Exception Safety: Concepts and Techniques (http://www.stroustrup.com/except.pdf), Bjarne Stroustrup
- The RAII Programming Idiom (http://www.hackcraft.net/raii)
- Sutter, Herb (1999). Exceptional C++. Addison-Wesley. ISBN 0-201-61562-2.
- C++ Patterns: Execute Around Sequences (http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ExecutingAroundSequences.pdf), Kevlin Henney

Resource Return

Intent

To convey ownership transfer (of a resource) explicitly in the return type of a factory function.

Also Known As

Motivation

Factory functions are often used to create new resources and return them to the caller. A new resource could be raw memory, dynamically allocated object, database cursors/connections, locks and so on. An important question about resources is who owns the resource and who releases it? Many times, interfaces are developed where the caller is **implicitly** responsible for resource release. If the caller is not aware of this fact or simply forgets to take correct steps, it gives rise to an easy-to-use-incorrectly kind of interface. Following code snippet shows an example.

```
struct X
{
  void foo() {}
};
X * Xfactory() // Resource ownership implicitly transferred to the caller.
{
  return new X; // Dynamically allocated instance
}
int main (void)
{
  Xfactory()->foo(); // Dynamically allocated instance of X leaks here
}
```

Resource Return Idiom provides different alternatives to rectify the situation and results into (somewhat) hard-to-use-incorrectly interfaces.

Solution and Sample Code

The solution is to wrap the resource in a resource-management smart pointer and return the smart pointer instead of the raw pointers. Simplest form of Resource Return Idiom is shown in the code snippet below.

```
struct X
{
  void foo() {}
};
std::auto_ptr<X> Xfactory() // Resource ownership explicitly transferred to the caller.
{
  return std::auto_ptr<X> (new X); // Dynamically allocated instance
}
int main (void)
{
  Xfactory()->foo(); // Dynamically allocated instance of X does not leak here
}
```

There are several issues to be considered while determining the type of resource-management smart pointer to use to return a resource. Possible options are:

- std::auto_ptr
- boost::shared_ptr
- std::unique_ptr in C++0x
- User defined Handle/Body idiom

An excellent discussion of pros and cons of choosing one over the other are discussed at length by Scott Meyer in his article The Resource Return Problem (http://www.aristeia.com/Papers/resourceReturnProblem.txt). As long as custom deletion functionality (other than plain old delete) is not required, auto_ptrs are a quick way to use the Resource Return idiom as shown above. auto_ptrs give exclusive but transferable ownership of resources which becomes very clear just by looking at the interface. For dynamically allocated pointer-returning factory functions, boost::shared_ptr is a also good choice because it offers normal copy-semantics (e.g., it can be stored in STL containers). It also allows changes to resource release policy from normal delete operation to custom deletion without disturbing clients.

When exclusive ownership is involved in case of Resource Return idiom, Move Constructor idiom is often useful while transferring ownership of resources.

Known Uses

Related Idioms

- Resource Acquisition Is Initialization (RAII)
- Move Constructor

References

- The Resource Return Problem (http://www.aristeia.com/Papers/resourceReturnProblem.txt)

Return Type Resolver

Intent

Deduce the type of the variable being initialized or assigned to.

Also Known As

Motivation

The type of the variable being initialized can be a useful information to have in certain contexts. Consider, for instance, we want to initialize STL containers with random numbers. However, we don't know the exact type of the container expected by the user. It could be std::list, std::vector or something custom that behaves like STL container. A straight-forward approach to write and use such a function would be as follows.

```
template <class Container>
Container getRandomN(size_t n)
{
  Container c;
  for(size_t i = 0;i < n; ++i)
    c.insert(c.end(), rand());
  return c;
}

int main (void)
{
   std::list<int> l = getRandomN<std::list<int> > (10);
   std::vector<long> v = getRandomN<std::vector<long> > (100);
}
```

Note that the type of the container must be passed to the function because that is the desired return type of the function. Clearly, the type must be repeated at-least twice. Return type resolver idiom can be used to address this issue.

Solution and Sample Code

Return type resolver idiom makes use of a proxy class and templatized conversion operator functions in the class. *getRandomN* function above can be implemented with a class and a member conversion function as follows.

```
class getRandomN
{
  size_t count;

public:
  getRandomN(int n = 1) : count(n) {}

  template <class Container>
  operator Container () {
    Container c;
    for(size_t i = 0;i < count; ++i)
      c.insert(c.end(), rand()); // push_back is not supported by all standard containers.
    return c;
  }
};

int main()
{
  std::set<int> random_s = getRandomN(10);
  std::vector<int> random_v = getRandomN(10);
  std::list<int> random_l = getRandomN(10);
}
```

getRandomN class has a constructor and a templatized conversion operator function. For initialization, a temporary object of *getRandomN* class is created and assigned to the desired container class. C++ compiler attempts to convert the temporary object into the container class object. The only way to do that is via the conversion operator. The conversion operator is instantiated with the type of the container that is being populated. Due to automatic resolution of the return type, the user does not have to spell it out again. Note that `insert` member function has been used instead of `push_back` because `std::set` does not support `push_back`.

Known Uses

The nullptr idiom makes use of the return type resolver idiom to automatically deduce a null pointer of the correct type depending upon the pointer variable it is assigning to.

Related Idioms

- nullptr

References

Runtime Static Initialization Order Idioms

Intent

Control the order of initialization and destruction of non-local static objects across compilation units that are otherwise ordered in an implementation dependent manner.

Motivation

- Order of initialization of static objects spread across different compilation units is not well defined. Order of destruction is the reverse of initialization order but initialization order itself is implementation defined. Bring order to this chaos.

- The destructor of static objects are non-trivial and have important side-effects that have to happen

Solution and Sample Code

The following idioms are commonly used to control the order of initialization of static objects.

- Construct On First Use
- Nifty Counter Idiom (a.k.a. Schwarz Counter Idiom)

Safe Bool

Intent

To provide boolean tests for a class but restricting it from taking participation in unwanted expressions.

Also Known As

Motivation

User provided boolean conversion functions can cause more harm than benefit because it allows them to participate in expressions you would not ideally want them to. If a simple conversion operator is defined then two or more objects of unrelated classes can be compared. Type safety is compromised. For example,

```
struct Testable
{
    operator bool() const {
          return false;
    }
};
struct AnotherTestable
{
    operator bool() const {
          return true;
    }
};
int main (void)
{
  Testable a;
  AnotherTestable b;
  if (a == b) { /* blah blah blah*/ }
  if (a < 0) { /* blah blah blah*/ }
  // The above comparisons are accidental and are not intended but the compiler happily compiles them.
  return 0;
}
```

Solution and Sample Code

Safe bool idiom allows syntactical convenience of testing using an intuitive if statement but at the same time prevents unintended statements unknowingly getting compiled in. Here is the code for the safe bool idiom.

```
class Testable {
    bool ok_;
    typedef void (Testable::*bool_type)() const;
    void this_type_does_not_support_comparisons() const {}
  public:
    explicit Testable(bool b=true):ok_(b) {}

    operator bool_type() const {
      return ok_ ?
        &Testable::this_type_does_not_support_comparisons : 0;
    }
};
```

```
template <typename T>
bool operator!=(const Testable& lhs, const T&) {
    lhs.this_type_does_not_support_comparisons();
    return false;
}
template <typename T>
bool operator==(const Testable& lhs, const T&) {
    lhs.this_type_does_not_support_comparisons();
    return false;
}
class AnotherTestable ... // Identical to Testable.
{};
int main (void)
{
  Testable t1;
  AnotherTestable t2;
  if (t1) {} // Works as expected
  if (t2 == t1) {} // Fails to compile
  if (t1 < 0) {} // Fails to compile

  return 0;
}
```

Reusable Solution

There are two plausible solutions: Using a base class with a virtual function for the actual logic, or a base class that knows which function to call on the derived class. As virtual functions come at a cost (especially if the class you're augmenting with Boolean tests doesn't contain any other virtual functions). See both versions below:

```
class safe_bool_base {
  public:
    typedef void (safe_bool_base::*bool_type)() const;
    void this_type_does_not_support_comparisons() const {}
  protected:

    safe_bool_base() {}
    safe_bool_base(const safe_bool_base&) {}
    safe_bool_base& operator=(const safe_bool_base&) {return *this;}
    ~safe_bool_base() {}
};

// For testability without virtual function.
template <typename T=void>
class safe_bool : private safe_bool_base {
  // private or protected inheritance is very important here as it triggers the
  // access control violation in main.
  public:
    operator bool_type() const {
      return (static_cast<const T*>(this))->boolean_test()
        ? &safe_bool_base::this_type_does_not_support_comparisons : 0;
    }
  protected:
    ~safe_bool() {}
};


// For testability with a virtual function.
template<>
class safe_bool<void> : private safe_bool_base {
  // private or protected inheritance is very important here as it triggers the
  // access control violation in main.
  public:
    operator bool_type() const {
      return boolean_test()
        ? &safe_bool_base::this_type_does_not_support_comparisons : 0;
    }
  protected:
    virtual bool boolean_test() const=0;
    virtual ~safe_bool() {}
};

template <typename T>
   bool operator==(const safe_bool<T>& lhs, bool b) {
      return b == static_cast<bool>(lhs);
```

```
  }

template <typename T>
   bool operator==(bool b, const safe_bool<T>& rhs) {
      return b == static_cast<bool>(rhs);
  }


template <typename T, typename U>
  bool operator==(const safe_bool<T>& lhs,const safe_bool<U>& rhs) {
      lhs.this_type_does_not_support_comparisons();
      return false;
  }

template <typename T,typename U>
  bool operator!=(const safe_bool<T>& lhs,const safe_bool<U>& rhs) {
    lhs.this_type_does_not_support_comparisons();
    return false;
  }
```

Here's how to use safe_bool:

```
#include <iostream>

class Testable_with_virtual : public safe_bool<> {
  public:
    virtual ~Testable_with_virtual () {}
  protected:
    virtual bool boolean_test() const {
      // Perform Boolean logic here
      return true;
    }
  };

 class Testable_without_virtual :
    public safe_bool <Testable_without_virtual> // CRTP idiom
 {
  public:
    /* NOT virtual */ bool boolean_test() const {
      // Perform Boolean logic here
      return false;
    }
  };

int main (void)
{
  Testable_with_virtual t1, t2;
  Testable_without_virtual p1, p2;
  if (t1) {}
  if (p1 == false)
  {
    std::cout << "p1 == false\n";
  }
  if (p1 == p2) {} // Does not compile, as expected
  if (t1 != t2) {} // Does not compile, as expected

  return 0;
}
```

In C++, address of protected members functions can't be taken in a derived class. Derived class could be a standard class, a class template or a specialization of a class template. Some implementations of safe bool idiom declare safe_bool_base::this_type_does_not_support_comparisons as protected, address of which can't be taken in the derived class - a requirement in reusable safe bool idiom.

An insightful discussion on the boost mailing list (http://lists.boost.org/Archives/boost/2011/05/182157.php) initiated by Krzysztof Czainski resulted in an implementation (http://codepaste.net/c83uuj) of safe bool idiom using CRTP as well as another using macros.

C++11

The C++11 standard provides *explicit conversion operators* as a parallel to explicit constructors. This new feature solves the problem in a clean way.[1][2][3]

```
struct Testable
{
    explicit operator bool() const {
          return false;
    }
};

int main()
{
  Testable a, b;
  if (a)      { /*do something*/ }  // this is correct
  if (a == b) { /*do something*/ }  // compiler error
}
```

Known Uses

- boost::scoped_ptr
- boost::shared_ptr
- boost::optional
- boost::tribool

Related Idioms

References

http://www.artima.com/cppsource/safebool.html

Scope Guard

Intent

- To ensure that resources are always released in face of an exception but not while returning normally
- To provide basic exception safety guarantee

Also Known As

Motivation

Resource Acquisition is Initialization (RAII) idiom allows us to acquire resources in the constructor and release them in the destructor when scope ends successfully or due to an exception. It will always release resources. This is not very flexible. Sometime we don't want to release resources if no exception is thrown but we do want to release them if exception is thrown.

Solution and Sample Code

Enhance the typical implementation of Resource Acquisition is Initialization (RAII) idiom with a conditional check.

```
class ScopeGuard
{
public:
  ScopeGuard ()
    : engaged_ (true)
  { /* Acquire resources here. */ }
```

```
  ~ScopeGuard ()
  {
    if (engaged_)
     { /* Release resources here. */}
  }
  void release ()
  {
     engaged_ = false;
     /* Resources no longer be released */
  }
private:
  bool engaged_;
};
void some_init_function ()
{
  ScopeGuard guard;
  // ...... Something may throw here. If it does we release resources.
  guard.release (); // Resources will not be released in normal execution.
}
```

Known Uses

- boost::mutex::scoped_lock
- boost::scoped_ptr
- std::auto_ptr
- ACE_Guard
- ACE_Read_Guard
- ACE_Write_Guard
- ACE_Auto_Ptr
- ACE_Auto_Array_Ptr

Related Idioms

- Resource Acquisition Is Initialization (RAII)

References

- Generic: Change the Way You Write Exception-Safe Code — Forever (http://www.ddj.com/cpp/184403758)
- SCOPE_FAILURE and SCOPE_SUCCESS in C++: ScopeGuards without need to release/commit by hands (https://github.com/panaseleus/stack_unwinding#d-style-scope-guardsactions)

SFINAE

Intent

Prune functions that do not yield valid template instantiations from a set of overloaded functions.

Also Known As

Substitution **F**ailure **I**s **N**ot **A**n **E**rror

Motivation and Solution

Strictly, SFINAE is a language feature and not an idiom. However, this language feature is exploited in a very idiomatic fashion using enable-if.

In the process of template argument deduction, a C++ compiler attempts to instantiate signatures of a number of candidate overloaded functions to make sure that exactly one overloaded function is available as a perfect match for a given function call. If an invalid argument or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set instead of causing a compilation error. As long as there is one and only one function to which the call can be dispatched, the compiler issues no errors.

For example, consider, a simple function multiply and its templatized counter-part.

```
long multiply(int i, int j) { return i * j; }

template <class T>
typename T::multiplication_result multiply(T t1, T t2)
{
  return t1 * t2;
}
int main(void)
{
  multiply(4,5);
}
```

Calling function `multiply` in `main` causes the compiler to instantiate the signature of the templatized function even though the first `multiply` function is a better match. During instantiation an invalid type is produced: `int::multiplication_result`. Due to SFINAE, however, the invalid instantiation is neglected automatically. At the end, there is exactly one `multiply` function that can be called. So the compilation is successful.

SFINAE is often exploited in determining properties of types at compile-time. For instance, consider the following `is_pointer` meta-function that determines at compile-time if the given type is a pointer of some sort.

```
template <class T>
struct is_pointer
{
  template <class U>
  static char is_ptr(U *);

  template <class X, class Y>
  static char is_ptr(Y X::*);

  template <class U>
  static char is_ptr(U (*)());

  static double is_ptr(...);

  static T t;
  enum { value = sizeof(is_ptr(t)) == sizeof(char) };
};

struct Foo {
  int bar;
};

int main(void)
{
  typedef int * IntPtr;
  typedef int Foo::* FooMemberPtr;
  typedef int (*FuncPtr)();

  printf("%d\n",is_pointer<IntPtr>::value);        // prints 1
  printf("%d\n",is_pointer<FooMemberPtr>::value);  // prints 1
  printf("%d\n",is_pointer<FuncPtr>::value);       // prints 1
}
```

The `is_pointer` meta-function above would not work without SFINAE. It defines 4 overloaded `is_ptr` functions, three of which are templates that accept one argument each: a pointer to a variable, pointer to a member variable, or a simple function pointer. All three functions return a `char`, which is deliberate. The last

`is_ptr` function is a catch-all function that uses ellipsis as it parameter. This function, however, returns a `double`, which is always greater in size compared to a character.

When the `is_pointer` is passed a type that is really a pointer (e.g., IntPtr), `value` is initialized to true as a result of the comparison of two `sizeof` expressions. The first `sizeof` expression calls `is_ptr`. If at all it is a pointer, only one of the overloaded template functions match and not others. Due to SFINAE, however, no error is raised because at least one function is found to be suitable. If none of the functions are suitable, the function with ellipsis is used instead. That function however, returns a `double`, which is larger than a character and so the `value` is initialized to false as the comparison of `sizeof` fails.

Note that, none of the `is_ptr` functions have definitions. Only declarations are sufficient to trigger SFINAE rule in the compiler. Those functions themselves must be templates, however. That is, a class template with regular functions will not participate in SFINAE. The functions that participate in SFINAE must be templates.

Known Uses

- Member Detector

Related Idioms

- enable-if

References

- SFINAE Sono Buoni (http://www.stevedewhurst.com/once_weakly/w02_SFINAE.pdf)
- Function Overloading Based on Arbitrary Properties of Types (http://www.ddj.com/cpp/184401659)
- SFINAE article on Wikipedia (http://en.wikipedia.org/wiki/SFINAE)More C++ Idioms/Shortening Long Template Names

Shrink-to-fit

Intent

Minimize the capacity of a container just enough to hold existing range.

Also Known As

"Swap-To-Fit", introduced by Scott Meyers in his book "Effective STL".

Motivation

Standard library containers often allocate more memory than the actual number of elements in them. Such a policy results in an optimization of saving some allocations when a container grows in size. On the other hand, when size of the container reduces, there is often leftover capacity in the container. The leftover capacity of the container can be unnecessary wastage of memory resources. Shrink-to-fit idiom has been developed to reduce the extra capacity to a minimum required capacity and thereby saving memory resources.

Solution and Sample Code

Shrink-to-fit idiom is as simple as the one given below.

```
std::vector<int> v;
//v is swapped with its temporary copy, which is capacity optimal
```

```
std::vector<int>(v).swap(v);
```

The first half of the statement, `std::vector<int>(v)`, creates a temporary vector of integers and it is guaranteed[4] to allocate memory just enough to hold all the elements in the parameter, `v`. The second half of the statement swaps the temporary vector with `v` using the non-throwing swap member function. `swap()` is very efficient, amounting to little more than swapping pointers between the vectors. After swapping, the temporary goes out of scope and deletes the memory originally held by `v`, whereas `v` retains the memory allocated by the temporary, so it has just enough to hold the original elements in `v`.

1. N2437: Explicit Conversion Operator Draft Working Paper (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2437.pdf)
2. http://stackoverflow.com/questions/6242768/is-the-safe-bool-idiom-obsolete-in-c11
3. http://stackoverflow.com/questions/13193766/isset-or-operator-void-or-explicit-opertor-bool-or-something-else
4. ISO/IEC 14882:1998 does not appear to document this behavior of the copy constructor. Where is this behavior guaranteed?

A more reliable solution (notably for `std::string`, which may be implemented using *reference counting*, but also for `std::vector`, whose copy-constructor might copy the other vector's excess capacity[1]) is to use the "range constructor" instead of the copy constructor:

```
std::vector<int> v;
//v is swapped with its temporary copy, which is capacity optimal
std::vector<int>(v.begin(), v.end()).swap(v);
```

1. http://www.aristeia.com/BookErrata/estl1e-errata.html (search for "string(s.begin(), s.end()).swap(s)")

Solution in C++11

In C++11 some containers declare such idiom as function `shrink_to_fit()`, e.g. vector, deque, basic_string. `shrink_to_fit()` is a non-binding request to reduce `capacity()` to `size()`.

Known Uses

Related Idioms

- Clear-and-minimize
- Non-throwing swap

References

- Programming Languages — C++ (http://open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf) draft standard.

As seen in boost::function to implement type erasure but faster than the heap allocated holder object

Smart Pointer

Intent

To relieve the burden of duplicating changes to the signature of the body class in its handle class when Handle Body idiom or Envelope Letter idiom is in use.

Also Known As

- En Masse (whole) Delegation

Motivation

When Handle/body idiom is used, it may become necessary to duplicate the interface of the body class in the handle class because handles are used by the user code. This duplication is often tedious and error prone. Smart Pointer idiom is used to relieve this burden. Smart Pointer idiom is often used along with some sort of "smartness" in the handle class such as reference counting, automatic ownership management and so on.

Solution and Sample Code

There are at least two overlapping ways of implementing the smart pointer idiom depending upon the intended use.

- Completely pointer like semantics
- Less pointer like semantics

Both of the above variations define an overloaded arrow operator in the so called "handle" class. Lets begin with the completely pointer like semantics.

```
class Body;
class Handle // Completely pointer like semantics
{
  public:
    void set (Body *b) { body_ = b; }
    Body * operator -> () const throw()
    {
      return body_;
    }
    Body & operator * () const throw ()
    {
      return *body_;
    }
  private:
    mutable Body *body_;
};
int main (void)
{
   Handle h;
   h.set(new Body());
   h->foo();    // A way of invoking Body::foo()
   (*h).foo();  // Another way of invoking Body::foo()
}
```

Using the -> operator alone mitigates the problem of duplicating interface of body class in the handle class. An alternative is to overload deference (*) operator as show in the code snippet above but it is not as natural as the earlier one. If the Handle abstraction is a some sort of pointer abstraction then both the overloaded operators should be provided (e.g., std::auto_ptr, boost::shared_ptr). If the Handle abstraction is not a pointer like abstraction then * operator need not be provided. Instead, it could useful to provide const overloaded set of arrow operators because client always interacts with the handle class objects. For the client code, handle **is** the object and hence const-ness of the handle should be propagated to corresponding body whenever it's appropriate. In general, the obscure behavior of being able to modify a non-const body object from within a constant handle should be avoided. Unlike pure pointer semantics, in some cases, automatic type conversion from Handle class to Body class is also desirable.

```
class Body;
class Handle // Less pointer like semantics
{
  public:
    void set (Body *b) { body_ = b; }
```

```
    Body * operator -> () throw()
    {
      return body_;
    }
    Body const * operator -> () const throw()
    {
      return body_;
    }
    operator const Body & () const // type conversion
    {
      return *body_;
    }
    operator Body & ()  // type conversion
    {
      return *body_;
    }
    // No operator *()
  private:
    mutable Body *body_;
};
int main (void)
{
   Handle const h;
   h.set(new Body());
   h->foo();    // compiles only if Body::foo() is a const function.
}
```

An alternative to using member conversion functions is to use the Non-member get idiom as shown below. The overloaded non-member get() functions must be in the same namespace as the Handle class according to the Interface Principle (http://www.gotw.ca/publications/mill02.htm).

```
namespace H {
class Body;
class Handle { ... }; // As per above.
Body const & get (Handle const &h)
{
  return *h.body_;
}
Body & get (Handle &h)
{
  return *h.body_;
}
} // end namespace H.
int main (void)
{
  H::Handle const h;
  h.set(new Body());
  get(h).foo(); // compiles only if Body::foo() is a const function.
}
```

Known Uses

- std::auto_ptr (Completely pointer like semantics)
- boost::shared_ptr (Completely pointer like semantics)
- CORBA Var types in C++ (TAO_Seq_Var_Base_T< T > Class in TAO (The ACE ORB) - less pointer like semantics)

Related Idioms

- Handle Body
- Envelope Letter
- Reference Counting

References

Intent

Also Known As

Motivation

Solution and Sample Code

Known Uses

Related Idioms

References

Reference

Tag Dispatching (http://www.generic-programming.org/languages/cpp/techniques.php#tag_dispatching)

Temporary Base Class

Intent

Reduce the cost of creating temporary objects.

Motivation

Temporary objects are often created during execution of a C++ program. Result of C++ operators (unary, binary, logical, etc.) and return-by-value functions always give rise to temporary objects. For built-in types, the cost of creating temporaries is minimal because compilers often use CPU registers to manipulate them. However, creation of temporary objects can be quite expensive for user-defined classes that allocate memory in the constructor and may require expensive copy operations in the copy-constructor. Temporaries are often wasteful because their lifespan is usually very short and they exist only to be assigned to a named object (lvalue). Even though their lifespan is short, the C++ language rules require the compilers to create and destroy temporaries to maintain correctness of the program. (In reality, RVO and NRVO optimizations are allowed to eliminate some temporaries). The cost of creating and destroying temporaries can adversely affect the performance of programs that use heavy objects.

Consider, for instance, a `Matrix` class that uses heap memory to store an array of integers. This class uses the usual RAII idiom to manage the resources: allocation in the constructor and de-allocation in the destructor. The copy constructor and the copy-assignment operator take care of maintaining exclusive ownership of the allocated memory.

```
void do_addition(int * dest, const int * src1, const int * src2, size_t dim)
{
  for(size_t i = 0; i < dim * dim; ++i, ++dest, ++src1, ++src2)
    *dest = *src1 + *src2;
}

class Matrix
{
  size_t dim;
  int * data;
```

```
public:

  explicit Matrix(size_t d)
    : dim(d), data(new int[dim*dim]())
  {
    for(size_t i = 0;i < dim * dim; ++i)
      data[i] = i*i;
  }

  Matrix(const Matrix & m)
    : dim(m.dim), data(new int[dim*dim]())
  {
    std::copy(m.data, m.data + (dim*dim), data);
  }

  Matrix & operator = (const Matrix & m)
  {
    Matrix(m).swap(*this);
    return *this;
  }

  void swap(Matrix & m)
  {
    std::swap(dim, m.dim);
    std::swap(data, m.data);
  }

  Matrix operator + (const Matrix & m) const
  {
    Matrix result(this->dim);
    do_addition(result.data, this->data, m.data, dim);
    return result;
  }

  ~Matrix()
  {
    delete [] data;
  }
};
```

Using objects of this class in expression such as below have several performance issues.

```
Matrix m1(3), m2(3), m3(3), m4(3);
Matrix result = m1 + m2 + m3 + m4;
```

Temporary `Matrix` objects are created as a result of every summation and they are destroyed immediately. For every pair of parenthesis in `(((m1 + m2) + m3) + m4)`, a temporary object is needed. Creation and destruction of each temporary requires memory allocation and de-allocation, which is quite wasteful.

Temporary Base Class idiom is a way of reducing the overhead of temporaries in arithmetic expression such as the above. However, this idiom has major drawbacks as described towards the end.

Solution and Sample Code

Temporary Base Class idiom (TBCI) does not change the fact that temporary objects are created but it reduces (substantially) the cost of creating them. This is achieved by recognizing the places where temporaries are created and by using a type that is lightweight to create and destroy. Unlike C++11, C++03 does not have a language supported way of distinguishing temporary objects (rvalues) from named objects (lvalues). Const reference is the only way available in C++03 for binding a temporary object to a reference.

In TBCI idiom, each class for heavy objects is represented by two classes. One class, D (for derived), is meant to represent the named objects whereas another class, B (for base), is used to represent temporary objects. Users are expected to use only D class in variable/function declaration because objects of type D behave just like regular objects. For instance, deep copies are performed when copying D objects.

The B class is used for intermediate temporary objects created in arithmetic expressions. Creation and destruction of B type objects is transparent to the user provided all the results computed by D objects are assigned to another D object. The key difference between B and D classes is the copying behavior. Whenever a B or D object is created from a D object, a deep copy (i.e., new memory allocation and copying data) is performed. On the other hand, whenever a B or D object is created from a B object, a shallow copy (i.e., just copy pointers) is performed. These rules are also applicable to the assignment operators with an exception that existing memory may have to be deleted in the left hand side object (e.g, an assignment from a B to B or D). Moreover, unlike D objects, B objects do not have exclusive ownership of the data and therefore do not destroy it when a destructor is called.

Construction and Assignment

Deep Copy	Shallow Copy
From D to B, D	From B to B, D

Interfaces of B and D classes are designed carefully to fully support conversion from one another with respect to the rules of memory management given above. The following example is a TBCI version of the `Matrix` class shown above. The `Matrix` class is the main class whereas `TMatrix` class is meant to represent temporary matrices.

```
class Matrix;
class TMatrix
{
  size_t dim;
  int * data;
  bool freeable;
  void real_destroy();

 public:

  explicit TMatrix(size_t d);
  TMatrix(const TMatrix & tm);
  TMatrix(const Matrix & m);
  TMatrix & operator = (const Matrix & m);
  TMatrix & operator = (const TMatrix & tm);
  TMatrix & operator + (const Matrix & m);
  TMatrix & operator + (TMatrix tm);
  ~TMatrix();
  void swap(TMatrix &) throw();

  friend class Matrix;
};

class Matrix : public TMatrix
{
public:
  explicit Matrix(size_t dim);
  Matrix(const Matrix & tm);
  Matrix(const TMatrix & tm);
  Matrix & operator = (const Matrix & m);
  Matrix & operator = (const TMatrix & tm);
  TMatrix operator + (const Matrix & m) const;
  TMatrix operator + (const TMatrix & tm) const;
  ~Matrix();
};
```

The interfaces of the above two classes reveal several things. Not only the classes have doubled, number of member function have also (nearly) doubled. In particular, copy-constructor, copy-assignment operator, overloaded `operator +` are declared for both `Matrix` and `TMatrix`. This is necessary to ensure that in all possible cases where `Matrix` and `TMatrix` object come together, the behavior is well-defined.

The only way `TMatrix` objects are created are through the overloaded `operator +` in the `Matrix` class. Whenever two `Matrix` classes are added, the result is returned as a `TMatrix` object. The result of any subsequent additions of `Matrix` objects are stored in the same `TMatrix` object that is a result of the first addition. This eliminates the

need to allocate and de-allocate memory for temporary matrices. For instance, TBCI achieves efficiency by executing addition of 4 matrices in the following way.

```
Matrix result = (((m1 + m2) + m3) + m4);
...
Matrix result = (((T1) + m3) + m4);
...
Matrix result = ((T1) + m4);
...
Matrix result = (T1);
```

New memory is allocated only when the `TMatrix` object is created the first time. The results of additions are stored in the temporary `TMatrix` object shown as `T1`. Finally, the result must be assigned to a `Matrix` object to ensure that memory resources do not leak.

Note that other combinations of the arithmetic operations are also possible. In particular, when other higher precedence operators, such as binary multiplication and division are used, `TMatrix`objects may be created in different order. For the sake of simplicity, only binary addition is shown in the examples and parentheses are used to enforce precedence. For instance, consider the following order of execution.

```
((m1 + m2) + (m3 + m4))
...
((T1) + (T2))
...
(T1)
```

To obtain the above behavior, both the classes (`Matrix` and `TMatrix`) are implemented in an idiomatic way as shown below.

```
/***** Implementation *****/
void do_addition(int * dest, const int * src1, const int * src2, size_t dim)
{
  for(size_t i = 0; i < dim * dim; ++i, ++dest, ++src1, ++src2)
    *dest = *src1 + *src2;
}

void do_self_addition(int * dest, const int * src, size_t dim)
{
  for(size_t i = 0; i < dim * dim; ++i, ++dest, ++src)
    *dest += *src;
}

void populate(int *data, size_t dim)
{
  for(size_t i = 0;i < dim * dim; ++i)
    data[i] = i*i;
}

TMatrix::TMatrix(size_t d)
: dim(d), data (new int[dim*dim]()), freeable(0)
{
  populate(data, dim);
}

TMatrix::TMatrix(const TMatrix & tm)
: dim(tm.dim), data(tm.data), freeable(0)
{}

TMatrix::TMatrix(const Matrix & m)
: dim(m.dim), data(new int[dim*dim]), freeable(0)
{
  std::copy(data, data + dim*dim, m.data);
}

TMatrix & TMatrix::operator = (const Matrix & m)
{
  std::copy(m.data, m.data + (m.dim * m.dim), data);
  return *this;
}
```

```
TMatrix & TMatrix::operator = (const TMatrix & tm)
{
  real_destroy();
  dim = tm.dim;
  data = tm.data;
  freeable = 0;
  return *this;
}

TMatrix & TMatrix::operator + (const Matrix & m)
{
  do_self_addition(this->data, m.data, dim);
  return *this;
}

TMatrix & TMatrix::operator + (TMatrix tm)
{
  do_self_addition(this->data, tm.data, dim);
  tm.real_destroy();
  return *this;
}

TMatrix::~TMatrix()
{
  if(freeable) real_destroy();
}

void TMatrix::swap(TMatrix & tm) throw()
{
  std::swap(dim, tm.dim);
  std::swap(data, tm.data);
  std::swap(freeable, tm.freeable);
}

void TMatrix::real_destroy()
{
  delete [] data;
}

Matrix::Matrix(size_t dim)
: TMatrix(dim)
{}

Matrix::Matrix(const TMatrix & tm)
: TMatrix(tm)
{}

Matrix::Matrix(const Matrix & m)
: TMatrix(m)
{}

Matrix & Matrix::operator = (const Matrix &m)
{
  Matrix temp(m);
  temp.swap(*this);
  return *this;
}

Matrix & Matrix::operator = (const TMatrix & tm)
{
  real_destroy();
  dim = tm.dim;
  data = tm.data;
  freeable = 0;
  return *this;
}

TMatrix Matrix::operator + (const Matrix & m) const
{
  TMatrix temp_result(this->dim);
  do_addition(temp_result.data, this->data, m.data, dim);
  return temp_result;
}

TMatrix Matrix::operator + (const TMatrix & tm) const
{
  TMatrix temp_result(tm);
  do_addition(temp_result.data, this->data, tm.data, dim);
  return temp_result;
```

```
}

Matrix::~Matrix()
{
  freeable = 1;
}
```

Some highlights of the above code are given here. `TMatrix` objects are created only as a result of `Matrix` additions. Addition of two `Matrix` objects results in a freshly allocated `TMatrix` object. On the the other hand, when any one of the objects being added are of `TMatrix` type, the result is stored in the memory referenced by the temporary.

`TMatrix` constructor allocates memory but destroys it only when `freeable` is true. `Matrix` destructor responsible for freeing memory. Unlike copy-constructor of `Matrix`, `TMatrix` copy-constructor does a shallow copy. However, as mentioned before, construction of `TMatrix` from `Matrix` does a deep copy with its exclusive memory.

`TMatrix::operator +` is a key function in this idiom. Note that the result of the addition is stored in the temporary object itself (`do_self_addition`) thereby avoiding another allocation. The `TMatrix::operator +` that takes a `TMatrix` object as a parameter is interesting because it accepts `TMatrix` object by-value. This is necessary because an addition of two temporary objects should result into only one temporary object and the other one must be destroyed. `TMatrix` objects do manage their memory, therefore, one of the objects is explicitly destroyed using `TMatrix::real_destroy`. This would not have been possible if right-hand-side `TMatrix` is bound to a const-reference.

The `Matrix` class is implemented in terms of `TMatrix`. Construction of `Matrix` from another `Matrix` does an allocation and copy. On the other hand, construction of `Matrix` from a `TMatrix` simply copies the pointers because `TMatrix` does not delete data. The same rules are applicable to the assignment operators of the `Matrix` class. Finally, `Matrix` destructor simply changes the `freeable` flag to `true` resulting into deleting the memory. To handle some rarely occurring cases correctly, assignments to a `TMatrix` object are also handled. An assignment from a `Matrix` is simply a copy operation whereas an assignment from a `TMatrix` class requires destroying one of them and doing a shallow copy of the pointers.

Caveats

This idiom has some serious drawbacks.

- The result of computation *must* be assigned to a derived class object (`Matrix` object). Otherwise there is definite resource leak in the program. Such a convention is usually very hard to maintain.
- As a consequence, this idiom is not even basic exception safe. If any intermediate memory allocation operation throws an exception, it is very likely that there would be a resource leak.

Known Uses

The TBCI Library (http://plasimo.phys.tue.nl/TBCI/)

Related Idioms

- Move Constructor
- Computational Constructor
- Resource Acquisition Is Initialization

References

- K. Spanderen und Y. Xylander: Gut kalkuliert, "Effiziente Numerik mit C++", iX Seiten 166–173, November 1996

- Documentation of the Numerical classes for Plasma simulation (http://plasimo.phys.tue.nl/TBCI/numlib/doc.pdf), Kurt Garloff

Temporary Proxy

Intent

To detect and execute different code when the result of overloaded `operator []` is either modified or observed.

Also Known As

`operator []` proxy

Motivation

The index operator (`operator []`) is often used to provide array like access syntax for user-defined classes. C++ standard library uses `operator []` in std::string and std::map classes. Standard string simply returns a character reference as a result of `operator []` whereas std::map returns a reference to the value given its key. In both cases, the returned reference can be directly read or written to. The string or map class have no knowledge or has no control over whether the reference is used for reading or for modification. Sometimes, however, it is useful to detect how the value is being used.

For example, consider an `UndoString` class, which supports a single *undo* operation in addition to the existing std::string interface. The design must allow an undo even though the character is accessed using the index operator. As mention above, std::string class has no knowledge of whether the result of `operator []` will be written to or not. The temporary proxy can be used to solve this problem.

Solution and Sample Code

The temporary proxy idiom uses another object, conveniently called *proxy*, to detect whether the result of `operator []` is used for reading or writing. The `UndoString` class below defines its own non-const `operator []`, which takes place of std::string's non-const `operator []`.

```
class UndoString : public std::string
{
  struct proxy
  {
    UndoString * str;
    size_t pos;

    proxy(UndoString * us, size_t position)
      : str(us), pos(position)
    {}

    // Invoked when proxy is used to modify the value.
    void operator = (const char & rhs)
    {
      str->old = str->at(pos);
      str->old_pos = pos;
      str->at(pos) = rhs;
    }

    // Invoked when proxy is used to read the value.
    operator const char & () const
    {
      return str->at(pos);
    }
  };

  char old;
  int old_pos;
```

```
public:

  UndoString(const std::string & s)
    : std::string(s), old(0), old_pos(-1)
  {}

  // This operator replaces std::string's non-const operator [].
  proxy operator [] (size_t index)
  {
    return proxy(this, index);
  }

  using std::string::operator [];

  void undo()
  {
    if(old_pos == -1)
      throw std::runtime_error("Nothing to undo!");

    std::string::at(old_pos) = old;
    old = 0;
    old_pos = -1;
  }
};
```

The new `operator []` returns an object of `proxy` type. The `proxy` type defines an overloaded assignment operator and a conversion operator. Depending on the context how the proxy is used, the compiler chooses different functions as shown below.

```
int main(void)
{
  UndoString ustr("More C++ Idioms");
  std::cout << ustr[0];  // Prints 'M'
  ustr[0] = 'm';         // Change 'M' to 'm'
  std::cout << ustr[0];  // Prints 'm'
  ustr.undo();           // Restore'M'
  std::cout << ustr[0];  // Prints 'M'
}
```

In all the output expressions (`std::cout`) above, the `proxy` object is used for reading and therefore, the compiler uses the conversion operator, which simply returns the underlying character. In the assignment statement (`ustr[0] = 'm';`), however, the compiler invokes the assignment operator of the `proxy` class. The assignment operator of the `proxy` object saves the original character value in the parent `UndoString` object and finally writes the new character value to the new position. This way, using an extra level of indirection of a temporary proxy object the idiom is able to distinguish between a read and a write operation and take different action based on that.

Caveats

Introducing an intermediary `proxy` may result in surprising compiler errors. For example, a seemingly innocuous function `modify` fails to compile using the current definition of the `proxy` class.

```
void modify(char &c)
{
  c = 'Z';
}
// ...
modify(ustr[0]);
```

The compiler is unable to find a conversion operator that converts the temporary proxy object into a `char &`. We have defined only a `const` conversion operator that returns a `const char &`. To allow the program to compile, another non-const conversion operator could be added. However, as soon as we do that, it is not clear whether the result of conversion is used to modify the original or not.

Known Uses

Related Idioms

References

More C++ Idioms/The result of technique

Thin Template

Intent

To reduce object code duplication when a class template is instantiated for many types.

Also Known As

Motivation

Templates are a way of reusing source code, not object code. A template can be defined for a whole family of classes. Each member of that family that is instantiated requires its own object code. Whenever a class or function template is instantiated, object code is generated specifically for that type. The greater the number of parameterized types, the larger the generated object code. Compilers only generate object code for functions for class templates that are used in the program, but duplicate object code could still pose a problem in environments where memory is not really abundant. Reusing the same object code could also improve instruction cache performance and thereby application performance. Therefore, it is desirable to reduce duplicate object code.

Solution and Sample Code

Thin template idiom is used to reduce duplicate object code. Object code level reusable code is written only once, generally in a base class, and is compiled in a separately deployable .dll or .so file. This base class is not type safe but type safe "thin template" wrapper adds the missing type safety, without causing much (or any) object code duplication.

```
// Not a template
class VectorBase {
  void insert (void *);
  void *at (int index);
};

template <class T>
class Vector<T*> // Thin template
   : public VectorBase
{
  inline void insert (T *t) {
     VectorBase::insert (t);
  }
  inline T *at (int index) {
     return VectorBase::at (index);
  }
};
```

The base class may be fat: it may contain an arbitrary amount of code. Because this class uses only inline functions, it generates no extra code. But because the casting is encapsulated in the inline function, the class is typesafe to its users. The templated class is thin

Known Uses

Symbian OS relies on this idiom a lot. For example,

```
template <class T> class CArrayFix : public CArrayFixBase
```

where CArrayFixBase does all the work

Related Idioms

References

Symbian Essential Idioms: Thin Templates (http://www.symlab.org/main/documentation/reference/s3/sdk/GUID-497930CE-4D61-50EE-A63B-3656158EE29C.html)

Reference

- Traits (http://www.generic-programming.org/languages/cpp/techniques.php#traits)
- Introduction to Traits (http://accu.org/index.php/journals/442)
- GENERIC<PROGRAMMING> Traits: The else-if-then of Types (http://erdani.org/publications/traits.html)

Type Erasure

Intent

To provide a type-neutral container that interfaces a variety of concrete types.

Also Known As

Variant, boost::any

Motivation

It is often useful to have a variable which can contain more than one type. Type Erasure is a technique to represent a variety of concrete types through a single generic interface.

Implementation and Example

Type Erasure is achieved in C++ by encapsulating a concrete implementation in a generic wrapper and providing virtual accessor methods to the concrete implementation via a generic interface.

They key components in this example interface are var, inner_base and inner classes:

```
struct var{
   struct inner_base{
      using ptr = std::unique_ptr<inner_base>;
   };
   template <typename _Ty> struct inner : inner_base{};
private:
   typename inner_base::ptr _inner;
};
```

The var class holds a pointer to the inner_base class. Concrete implementations on inner (such as inner<int> or inner<std::string>) inherit from inner_base. The var representation will access the concrete implementations through the generic inner_base interface. To hold arbitrary types of data a little more scaffolding is needed:

```
struct var{
   template <typename _Ty> var(_Ty src) : _inner(new inner<_Ty>(std::forward<_Ty>(src))) {} //construct an internal concre
   struct inner_base{
      using ptr = std::unique_ptr<inner_base>;
   };
   template <typename _Ty> struct inner : inner_base{
      inner(_Ty newval) : _value(newval) {}
   private:
      _Ty _value;
   };
private:
   typename inner_base::ptr _inner;
};
```

The utility of an erased type is to assign multiple typed values to it so an assignment operator achieves just that:

```
struct var{
   template <typename _Ty> var& operator = (_Ty src) {
      _inner = std::make_unique<inner<_Ty>>(std::forward<_Ty>(src));
      return *this;
   }
   struct inner_base{
      using ptr = std::unique_ptr<inner_base>;
   };
   template <typename _Ty> struct inner : inner_base{
      inner(_Ty newval) : _value(newval) {}
   private:
      _Ty _value;
   };
private:
   typename inner_base::ptr _inner;
};
```

Creating an erased type and assigning it various values isn't of much use unless you can interrogate it. One useful method is to query for the underlying type info:

```
struct var{
   const std::type_info& Type() const { return _inner->Type(); }
   struct inner_base{
      using ptr = std::unique_ptr<inner_base>;
      virtual const std::type_info& Type() const = 0;
  };
   template <typename _Ty> struct inner : inner_base{
      virtual const std::type_info& Type() const override { return typeid(_Ty); }
   };
private:
   typename inner_base::ptr _inner;
};
```

Here the var class forwards calls of Type() to it's inner_base interface which is overridden by the concrete inner<_Ty> subclass which ultimately returns the underlying type. This technique of forwarding accessor methods to a virtual interface which is overridden by concrete implementations is expanded for a fully useful generic type.

Complete Implementation

```
struct var {
   var() : _inner(new inner<int>(0)){} //default construct to an integer
```

```
   var(const var& src) : _inner(src._inner->clone()) {} //copy constructor calls clone method of concrete type

   template <typename _Ty> var(_Ty src) : _inner(new inner<_Ty>(std::forward<_Ty>(src))) {}

   template <typename _Ty> var& operator = (_Ty src) { //assign to a concrete type
      _inner = std::make_unique<inner<_Ty>>(std::forward<_Ty>(src));
      return *this;
   }

   var& operator=(const var& src) { //assign to another var type
      var oTmp(src);
      std::swap(oTmp, *this);
      return *this;
   }

   //interrogate the underlying type through the inner_base interface
   const std::type_info& Type() const { return _inner->Type(); }
   bool IsPOD() const { return _inner->IsPOD(); }
   size_t Size() const { return _inner->Size(); }

   //cast the underlying type at run-time
   template <typename _Ty> _Ty& cast() {
      return *dynamic_cast<inner<_Ty>&>(*_inner);
   }

   template <typename _Ty> const _Ty& cast() const {
      return *dynamic_cast<inner<_Ty>&>(*_inner);
   }

   struct inner_base {
      using Pointer = std::unique_ptr < inner_base > ;
      virtual ~inner_base() {}
      virtual inner_base * clone() const = 0;
      virtual const std::type_info& Type() const = 0;
      virtual bool IsPOD() const = 0;
      virtual size_t Size() const = 0;
   };

   template <typename _Ty> struct inner : inner_base {
      inner(_Ty newval) : _value(std::move(newval)) {}
      virtual inner_base * clone() const override { return new inner(_value); }
      virtual const std::type_info& Type() const override { return typeid(_Ty); }
      _Ty & operator * () { return _value; }
      const _Ty & operator * () const { return _value; }
      virtual bool IsPOD() const { return std::is_pod<_Ty>::value; }
      virtual size_t Size() const { return sizeof(_Ty); }
   private:
      _Ty _value;
   };

   inner_base::Pointer _inner;
};

//this is a specialization of an erased std::wstring
template <>
struct var::inner<std::wstring> : var::inner_base{
   inner(std::wstring newval) : _value(std::move(newval)) {}
   virtual inner_base * clone() const override { return new inner(_value); }
   virtual const std::type_info& Type() const override { return typeid(std::wstring); }
   std::wstring & operator * () { return _value; }
   const std::wstring & operator * () const { return _value; }
   virtual bool IsPOD() const { return false; }
   virtual size_t Size() const { return _value.size(); }
   private:
   std::wstring _value;
};
```

Type Generator

Intent

- To simplify creation of complex template-based types
- To synthesize a new type or types based on template argument(s)
- To localize default policies when policy-based class design is used

Also Known As

Templated Typedef Idiom

Motivation

Class templates designed using policy-based class design, often result in very flexible templates with multiple type parameters. One downside of such class templates is that too many type parameters must be provided when instantiating them. Default template parameters can help in such cases. However, when the last template parameter (policy class) differs from the default, all of the intermediate template parameters must be specified.

For example, consider a case in which special purpose allocators are used with standard C++ containers. The GNU C++ compiler provides many special purpose allocators in namespace *__gnu_cxx* as extensions of the standard C++ library. The following illustrates specializing std::map with GNU's malloc_allocator:

```
std::map <std::string, int, less<std::string>, __gnu_cxx::malloc_allocator<std::string>>
```

A variation of the above map using *float* instead of an *int* requires all the unrelated type parameters to be mentioned again.

```
std::map <std::string, float, less<std::string>, __gnu_cxx::malloc_allocator<std::string> >
```

The type generator idiom is used to reduce code bloat in such cases.

Solution and Sample Code

In the type generator idiom, common (invariant) parts of a family of type definitions are collected together in a structure, whose sole purpose is to generate another type. For example, consider the *Directory* template shown below.

```
template <class Value>
struct Directory
{
  typedef std::map <std::string, Value, std::less<std::string>,
                    __gnu_cxx::malloc_allocator<std::string> > type;
};

Directory<int>::type    // gives a map of string to integers.
Directory<float>::type  // gives a map of string to floats.
```

An extra level of indirection (*struct Directory*) is used to capture the invariant part and one or two template parameters are left open for customization. A type generator usually consolidates a complicated type expression into a simple one. A type generator can be used to generate more than one type by simply adding more typedefs.

For example, consider how standard STL algorithms are applied to maps.

```
Directory<int>::type age; // This is a map.
transform(age.begin(), age.end(),
          std::ostream_iterator<string>(std::cout, "\n"),
          _Select1st<std::map<std::string, int>::value_type> ());
```

An adapter that transforms map's value_type, which is a pair, into the first element of the pair. *_Select1st* does the job of adapter in the example above. Its type is needlessly complex with ample opportunity of typing it wrong when repeated multiple times. Instead, the type generator idiom simplifies type specification of the

adapter considerably.

```
template <class Value>
struct Directory
{
  typedef map <string, Value, less<string>, __gnu_cxx::malloc_allocator<std::string> > type;
  typedef _Select1st<typename type::value_type> KeySelector;
  typedef _Select2nd<typename type::value_type> ValueSelector;
};
Directory<int>::type age;    // This is a map.
transform(age.begin(), age.end(),
          std::ostream_iterator<string>(std::cout, "\n"),
          Directory<int>::KeySelector());
```

Finally, the type generator idiom can be used to conveniently change the invariant type parameters, if needed. For example, changing *malloc_allocator* to *debug_allocator* throughout the program. The main reason why you might sometimes want to change it is to get more useful information from bounds-checking or leak-detection tools while debugging. Using type generators such a program-wide effect can be achieved by simply changing it at one place.

Known Uses

- Boost.Iterator library

Related Idioms

- Policy-based Class Design
- Meta-function wrapper
- Object Generator

References

[1] Type Generator (http://www.boost.org/community/generic_programming.html#type_generator)

[2] Policy Adaptors and the Boost Iterator Adaptor Library (http://www.oonumerics.org/tmpw01/abrahams.pdf) -- David Abrahams and Jeremy Siek

[3] Template Typedef (http://www.gotw.ca/gotw/079.htm) -- Herb Sutter

[4] The New C++: Typedef Templates (http://www.ddj.com/cpp/184403850) -- Herb Sutter

Type Safe Enum

Intent

Improve type-safety of native enum data type in C++.

Also Known As

Typed enums

Motivation

Enumerations in C++03 are not sufficiently type-safe and may lead to unintended errors. In spite of being a language-supported feature, enums also have code portability issues due to different ways in which different compilers handle the corner cases of enumerations. The problems surrounding enumerations can be categorized

in 3 ways[1]:

- Implicit conversion to an integer
- Inability to specify the underlying type
- Scoping issues

C++03 enumerations are not devoid of type safety, however. In particular, direct assignment of one enumeration type to another is not permitted. Moreover, there is no implicit conversion from an integer value to an enumeration type. However, most unintended enumeration errors can be attributed to its ability to get automatically promoted to integers. For instance, consider the following valid C++03 program. Only a few compilers such as GNU g++ issue a warning to prevent unintended errors like below.

```
enum color { red, green, blue };
enum shape { circle, square, triangle };

color c = red;
bool flag = (c >= triangle); // Unintended!
```

Other problems with C++03 enums are their inability to specify the underlying representation type to hold the values of enumerators. A side effect of this underspecification is that the sizes and the signedness of the underlying type vary from compiler to compiler, which leads to non-portable code. Finally, the scoping rules of C++03 may cause inconvenience. For instance, two enumerations in the same scope can not have enumerators with the same name.

Type safe enum idiom is a way to address these issues concerning C++03 enumerations. Note that *enum class* in C++11 will eliminate the need of this idiom.

Solution and Sample Code

Type safe enum idiom wraps the actual enumerations in a class or a struct to provide strong type-safety.

```
template<typename def, typename inner = typename def::type>
class safe_enum : public def
{
  typedef inner type;
  inner val;

public:

  safe_enum() {}
  safe_enum(type v) : val(v) {}
  type underlying() const { return val; }

  friend bool operator == (const safe_enum & lhs, const safe_enum & rhs) { return lhs.val == rhs.val; }
  friend bool operator != (const safe_enum & lhs, const safe_enum & rhs) { return lhs.val != rhs.val; }
  friend bool operator <  (const safe_enum & lhs, const safe_enum & rhs) { return lhs.val <  rhs.val; }
  friend bool operator <= (const safe_enum & lhs, const safe_enum & rhs) { return lhs.val <= rhs.val; }
  friend bool operator >  (const safe_enum & lhs, const safe_enum & rhs) { return lhs.val >  rhs.val; }
  friend bool operator >= (const safe_enum & lhs, const safe_enum & rhs) { return lhs.val >= rhs.val; }
};

struct color_def {
  enum type { red, green, blue };
};
typedef safe_enum<color_def> color;

struct shape_def {
  enum type { circle, square, triangle };
};
typedef safe_enum<shape_def, unsigned char> shape; // unsigned char representation

int main(void)
{
  color c = color::red;
  bool flag = (c >= shape::triangle); // Compiler error.
}
```

In the above solution, the actual enumerations are wrapped inside *color_def* and *shape_def* structures. To obtain a safe enumeration type from the definitions, *safe_enum* template is used. *safe_enum* template makes use of the Parameterized Base Class idiom i.e., it inherits publicly from the *def* parameter inself. As a result, the enumerations defined in the definitions are available in the *safe_enum* instantiation.

safe_enum template also supports a way to specify the underlying type (*inner*) used to hold values of the enumerators. By default it is the same as the enumeration type in its definition. Using explicit underlying representation type as the second type parameter, the size of the *safe_enum* template instantiation can be controlled in a portable way.

safe_enum template prevents automatic promotion to integers because there is no conversion operator in it. Instead, it provides *underlying()* function, which can be used to retrieve the value of the enumerator. Overloaded operators are also provided by the template to allow simple comparisons and total ordering of the type-safe enumerators.

Iterating over enumerations

In some cases where enumerations are all contiguous values, iteration over the entire set of enumeration is also possible. Strictly speaking, iteration is not a part of the idiom, but it is a useful enhancement. Following code shows the enhancements needed to add iteration capability. A static array of *safe_enum*s is created for each instantiation of *safe_enum* class. The array is populated by the *initialize* function with the original enumeration values. Note that enumeration values are assumed to be contiguous in the *initialize* function. Once the array is initialized, iteration is simply traversal from beginning to the end of the static array. The *begin* and *end* functions return iterators to the beginning and (one past) the end of the array.

```
template<typename def, typename inner = typename def::type>
class safe_enum : public def
{
  // ...
  // The stuff shown above goes here.
  // ...
private:
  static safe_enum array[def::_end_ - def::_begin_];
  static bool init;

  // Works only if enumerations are contiguous.
  static void initialize()
  {
    if(!init) // use double checked locking in case of multi-threading.
    {
      unsigned int size = def::_end_ - def::_begin_;
      for(unsigned int i = 0, j = def::_begin_; i < size; ++i, ++j)
        array[i] = static_cast<typename def::type>(j);
      init = true;
    }
  }

public:
  static safe_enum * begin() {
    initialize();
    return array;
  }

  static safe_enum * end() {
    initialize();
    return array + (def::_end_ - def::_begin_);
  }
};

template <typename def, typename inner>
safe_enum<def, inner> safe_enum<def, inner>::array[def::_end_ - def::_begin_];

template <typename def, typename inner>
bool safe_enum<def, inner>::init = false;

template <class Enum>
void f(Enum e)
```

```
{
  // ...
}
int main()
{
  typedef safe_enum<color_def, unsigned char> color;
  std::for_each(color::begin(), color::end(), &f<color>);
}
```

Known Uses

Related Idioms

- Parameterized Base Class

References

1. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2347.pdf

- Strongly Typed Enums (revision 3) (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2347.pdf), David E. Miller, Herb Sutter, and Bjarne Stroustrup

Type Selection

Intent

Select a type at compile-time based on a compile-time boolean value or a predicate.

Also Known As

Motivation

Ability to take decisions based on the information known at compile-time is a powerful meta-programming tool. One of the possible decisions to be made at compile-time is deciding a type i.e., the choice of the type may vary depending upon the result of a predicate.

For example, consider a *Queue* abstract data-type (ADT) implemented as a class template that holds a static array of Ts and the maximum capacity of the Queue is passed as a template parameter. The Queue class also needs to store the number of elements present in it, starting from zero. A possible optimization for such a queue class could be to use different types to store the size. For instance, when Queue's maximum capacity is less than 256, *unsigned char* can be used and if the capacity is less than 65,536, *unsigned short* can be used to store the size. For larger queues, *unsigned integer* is used. Type selection idiom can be used to implement such compile-time decision making.

Solution and Sample Code

A simple way of implementing the type selection idiom is the *IF* template. IF template takes three parameters. The first parameter is a compile-time boolean condition. If the boolean condition evaluates to *true* the 2nd type passed to the IF template is chosen otherwise third. Type selection idiom consists of a primary template and a partial specialization as shown below.

```
template <bool, class L, class R>
struct IF  // primary template
```

```
{
  typedef R type;
};

template <class L, class R>
struct IF<true, L, R> // partial specialization
{
  typedef L type;
};

IF<false, int, long>::type i; // is equivalent to long i;
IF<true,  int, long>::type i; // is equivalent to int i;
```

We now use the type selection idiom to implement the Queue size optimization mentioned above.

```
template <class T, unsigned int CAPACITY>
class Queue
{
  T array[CAPACITY];
  typename IF<(CAPACITY <= 256),
      unsigned char,
      typename IF<(CAPACITY <= 65536),
                  unsigned short,
                  unsigned int
                 >::type
    >::type size;
  // ...
};
```

The Queue class template declares an array or Ts. The type of the *size* data member depends on the result of two comparisons performed using the *IF* template. Note that these comparisons are not performed at run-time but at compile-time.

Known Uses

- Boost.MPL Library

Related Idioms

- Tag Dispatching

References

Virtual Constructor

Intent

To create a copy of an object or a new object without knowing its concrete type.

Also Known As

Factory Method of initialization

Motivation

Uses of calling member functions in a class hierarchy polymorphically are well known in the object-oriented programming community. It is a way of implementing the **is-a** (or more practically, **behaves-as-a**) relationship. Sometimes it is useful to call life-cycle management (creation, copy, and destruction) functions of a class hierarchy polymorphically.

C++ natively supports polymorphic destruction of objects using a virtual destructor. An equivalent support for creation and copying of objects is missing. In C++, creation of object(s) always requires its type to be known at compile-time. The Virtual Constructor idiom allows polymorphic creation of and copying of objects in C++.

Solution and Sample Code

The effect of a virtual constructor by a create() member function for creation and a clone() member function for copy construction as shown below.

```
class Employee
{
  public:
    virtual ~Employee () {}                 // Native support for polymorphic destruction.
    virtual Employee * create () const = 0; // Virtual constructor (creation)
    virtual Employee * clone () const = 0;  // Virtual constructor (copying)
};
class Manager : public Employee     // "is-a" relationship
{
  public:
    Manager ();                     // Default constructor
    Manager (Manager const &);      // Copy constructor
    virtual ~Manager () {}                  // Destructor
    Manager * create () const       // Virtual constructor (creation)
    {
      return new Manager();
    }
    Manager * clone () const        // Virtual constructor (copying)
    {
      return new Manager (*this);
    }
};
class Programmer : public Employee { /* Very similar to the Manager class */ };
Employee * duplicate (Employee const & e)
{
   return e.clone();  // Using virtual constructor idiom.
}
```

The Manager class implements the two pure virtual functions and uses the type name (Manager) to create them. The function duplicate shows how virtual constructor idiom is used. It does not really know what it is duplicating. It only knows that it is cloning an Employee. The responsibility of creating the right instance is delegated to the derived classes. The duplicate function is therefore closed for modifications even though the class hierarchy rooted at Employee gets more sub-classes added in the future.

The return type of the clone and create member functions of the Manager class is not Employee but the class itself. C++ allows this flexibility in types where the return type of the over-ridden function can be a derived type of that of the function in the base class. This language feature is known as **co-variant return types.**

To handle resource ownership properly, the Resource Return idiom should be employed for the return types of clone() and create() functions as they are factory functions. If used, the return types (shared_ptr<Employee> and shared_ptr<Manager>) are no longer covariant return types and program should fail to compile. In such a case, the virtual constructor functions in the derived class should return the exact type as in the parent class.

```
#include <tr1/memory>

class Employee
{
  public:
    typedef std::tr1::shared_ptr<Employee> Ptr;
    virtual ~Employee () {}                     // Native support for polymorphic destruction.
    virtual Ptr create () const = 0; // Virtual constructor (creation)
```

```
    virtual Ptr clone () const = 0;  // Virtual constructor (copying)
};
class Manager : public Employee     // "is-a" relationship
{
  public:
    Manager () {}                     // Default constructor
    Manager (Manager const &) {}      // Copy constructor
    virtual ~Manager () {}
    Ptr create () const       // Virtual constructor (creation)
    {
      return Ptr(new Manager());
    }
    Ptr clone () const        // Virtual constructor (copying)
    {
      return Ptr(new Manager (*this));
    }
};
```

Known Uses

std::function

Related Idioms

- Resource Return
- Polymorphic Exception

References

- Virtual Constructor (http://www.parashift.com/c++-faq-lite/virtual-functions.html#faq-20.8)

Virtual Friend Function

Intent

Simulate a virtual friend function.

Also Known As

Motivation

Friend functions are often needed in C++. A canonical example is that of types that can be printed to output streams (*e.g.,* std::cout). An overloaded left-shift operator function, which is often a friend, is needed to achieve seamless streaming capabilities. Friend functions are really an *extension* of the class's interface. However, friend functions in C++ can not be declared virtual and therefore no dynamic binding of friend functions is possible. Applying a friend function to an entire hierarchy of classes becomes awkward if an overloaded friend function is needed for every class in the hierarchy. This lack of support for dynamic binding makes it hard to justify that friend functions are in fact an *extension* of the class's interface. Virtual friend function idiom addresses this concern elegantly.

Solution and Sample Code

Virtual friend function idiom makes use of an extra indirection to achieve the desired effect of dynamic binding for friend functions. In this idiom, usually there is only one function that is a friend of the base class of the hierarchy and the friend function simply delegates the work to a helper member function that is virtual. The helper function is overridden in every derived class, which does the real job and the friend function just serves as a facade.

```
class Base {
  public:
    friend ostream& operator << (ostream& o, const Base& b);
    // ...
  protected:
    virtual void print(ostream& o) const
    { ... }
};
/* make sure to put this function into the header file */
inline std::ostream& operator<< (std::ostream& o, const Base& b)
{
  b.print(o); // delegate the work to a polymorphic member function.
  return o;
}

class Derived : public Base {
  protected:
    virtual void print(ostream& o) const
    { ... }
};
```

Known Uses

Related Idioms

References

- Virtual friend function idiom (http://www.parashift.com/c++-faq-lite/friends.html#faq-14.3), Marshall Cline

GNU Free Documentation License

Version 1.3, 3 November 2008 Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined

work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.3
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.