**Cheptor:4 Designs and Declarations**

**Item 18: Make interfaces easy to use correctly and hard to use incorrectly.**

Developing interfaces that are easy to use correctly and hard to use incorrectly requires that you consider the kinds of mistakes that clients might make.

For example, see the below interface:

class Date { //class representing date.

public:

Date(int month, int day, int year);

...

};

But client might pass blow invalid date, which is absolutely correct with our designed interface.

Date d(30, 3, 1995); // Oops! Should be “3, 30” , not “30, 3”

Date d(3, 40, 1995); // Oops! Should be “3, 30” , not “3, 40”

It is better pass primitive datatype int to simple wrapper types to distinguish days, months, and years, then use these types in the Date constructor:

struct Day { struct Month { struct Year {

explicit Day(int d) explicit Month(int m) explicit Year(int y)

: val(d) {} : val(m) {} : val(y){}

int val; int val; int val;

};

Our new design looks like:

class Date {

public:

Date(const Month& m, const Day& d, const Year& y);

...

};

This time harder to commit the mistake.

Date d(30, 3, 1995); // error! wrong types

Date d(Day(30), Month(3), Year(1995)); // error! wrong types

Date d(Month(3), Day(30), Year(1995)); // okay, types are correct

sometimes be reasonable to restrict the values of those types. For example, there are only 12 valid month values, so the Month type should not cross the limit. There are several ways to achieve this:

We can declare the enum for all month, but enums not type safe. For example, enums can be used like ints (see Item 2). A safer solution is to predefine the set of all valid Months like:

class Month {

public:

static Month Jan() { return Month(1); } // functions returning all valid

static Month Feb() { return Month(2); } // Month values; see below for

... // why these are functions, not

// objects.

static Month Dec() { return Month(12); }

... // other member functions

private:

explicit Month(int m); // prevent creation of new Month

// values month-specific data

};

Date d(Month::Mar(), Day(30), Year(1995));

Lets suppose a,b and c are rational numbers. Inside the class operator \* declared like this:

Rational operator \* (Rational & rhs);

if (a \* b = c) {…} // oops, meant to do a comparison!

We can easily remove this mistake by defining correct interface of operator \* like:

Const Rational operator \* (Rational & rhs); //Now syntax will not compile.

Any interface that requires that clients remember to do something is prone to incorrect use, because clients can forget to do it. For example, remember (Item 13) return pointers to dynamically allocated objects in an Investment hierarchy:

Investment\* createInvestment(); // Client need to delete the returned

// pointer.

In Item 13, client need to store the returned pointer into auto\_ptr or shared\_ptr. But what if clients forget to use the smart pointer?

Design interface like below solve our problem.

std::tr1::shared\_ptr<Investment> createInvestment();

Note: shared\_ptr allows a resource release function — a “deleter” — to be bound to the smart pointer when the smart pointer is created. (auto\_ptr has no such capability.)

**Things to Remember**

✦ Good interfaces are easy to use correctly and hard to use incorrectly. You should strive for these characteristics in all your interfaces.

**Item 19: Treat class design as type design**

Defining a new class defines a new type. This means you’re not just a class designer, you’re a type designer. Overloading functions and operators, controlling memory allocation and deallocation, defining object initialization and finalization — it’s all in your hands.

Whenever we design a new class/type need to take care of below questions:

■ **How should objects of your new type be created and destroyed?**

Concern about constructors and destructor, as well as its memory allocation and deallocation functions (operator new, operator new[], operator delete, and operator delete[])

■ **How should object initialization differ from object assignment?**

Determines the behaviour of and the differences between your constructors and

your Assignment operators. Never confuse to initialization with assignment,

because they correspond to different function calls.

■ **What does it mean for objects of your class (new type) to be passed by value?**

*Remember, the copy constructor defines how pass-by value is implemented for a type (Not understand)*

■ **What are the restrictions on legal values for your class (new type)?** *(not*

*understand)*

■ **Does your new type fit into an inheritance graph?**

If we inherit from existing classes, we are constrained by the design of those

classes, particularly by whether their functions are virtual or nonvirtual.

If we wish to allow other classes to inherit from our class, that affects whether

the functions we declare are virtual, especially our destructor.

■ **What kind of type conversions are allowed for your class (new type)?**

***Discuss Type casting***

If we wish to allow objects of type T1 to be *implicitly* converted into objects of

type T2, we will want to write either a type conversion function in class T1 (e.g.,

operator T2) or a non-explicit constructor in class T2 that can be called with a

single argument.

If you wish to allow *explicit* conversions only, you’ll want to write functions to

perform the conversions, but you’ll need to avoid making them type conversion

operators or non-explicit constructors that can be called with one argument.

#define PI 3.14

class Circle {

public:

Circle(float r):radious(r){}

Circle(Circle &c):radious(c.radious){}

float getArea() {

return PI\*radious\*radious;

}

float getParameter() {

return 2 \* PI\*radious;

}

private:

float radious;

};

class Square {

public:

Square(float s):side(s){}

Square(Square &rhs):side(rhs.side){}

//Implicit conversion from circle to square

Square(Circle &c) {

side = (c.getParameter())/4;

}

float getArea() {

return side\*side;

}

float getParameter() {

return 4 \* side;

}

private:

float side;

};

■ **What operators and functions make sense for the new type?**

Some functions will be member functions, but some will not based on our choice.

■ **What standard functions should be disallowed?**

Those are the ones you’ll need to declare private (compiler generated function).

■ **Who should have access to the members of your new type?**

This question helps you determine which members are public, which are

protected, and which are private. It also helps you determine which classes and/or functions should be friends, as well as whether it makes sense to nest one class inside another.

■ **What is the “undeclared interface” of your new type?**

What kind of guarantees does it offer with respect to performance, exception

safety and resource usage? (e.g., locks and dynamic memory)?

■ **How general is your new type?**

Perhaps we are not really defining a new type. Perhaps you’re defining a whole

*family* of types. If so, you don’t want to define a new class, you want to define a

new class *template*.

■ **Is a new type really what you need?**

If we are defining a new derived class only so that we can add functionality to an

existing class, better to defining one or more non-member functions on existing

class.

These questions are difficult to answer, so defining effective classes can be challenging.

**Things to Remember**

✦ Class design is type design. Before defining a new type, be sure to consider all the issues discussed in this Item.

**Item 20: Prefer pass-by-reference-to-const to pass-by value.**

By default, C++ passes objects to and from functions by value. Function parameters are initialized with copies of the actual arguments, and function callers get back a copy of the value returned by the function. These copies are produced by the objects’ copy constructors. This can make pass-by-value an expensive operation.

For example, consider the following class hierarchy:

class Person {

public:

Person(); // parameters omitted for simplicity

virtual ~Person();

...

private:

std::string name, address;

};

class Student: public Person {

public:

Student(); // parameters again omitted

virtual ~Student();

...

private:

std::string schoolName, schoolAddress;

};

Below function that takes a Student argument (by value) and returns whether it has been validated:

bool validateStudent(Student s); // function taking a Student by value

Student plato; // Plato studied under Socrates

bool platoIsOK = validateStudent(plato); // call the function

Cost of call by value calculation:

Passing a Student object by value leads to one call to the Student copy constructor, one call to the Person copy constructor, and four calls to the string copy constructor (member variable string). When the copy of the student object is destroyed, each constructor call is matched by a destructor call, so the overall cost of passing a Student by value is six constructors and six destructors.

it would be nice if there were a way to bypass all those constructions and destructions.

There is: pass by reference-to-const:

bool validateStudent(const Student& s);

This is much more efficient: no constructors or destructors are called, because no new objects are being created. But why we need const parameter declaration?

The original version of validateStudent took a Student parameter by value, so callers knew that any changes the inside the validateStudent would not be modify its actual parameter passed in. validateStudent would be able to modify only a copy of it.

Now, that the Student is being passed by reference, it’s necessary to also declare it const, because otherwise callers would have to worry about validateStudent making changes to the Student they passed in.

Passing parameters by reference also avoids the slicing problem when a derived class object is passed (by value) as a base class object, the base class copy constructor is called, which sliced off derived class object and we are only left with a simple base class object.

If we have an object of a built-in type (e.g., an int), it’s often more efficient to pass it by value than by reference.

In general, the only types for which you can reasonably assume that pass-by-value is inexpensive are built-in types and STL iterator and function object types. For everything else, follow the advice of this Item and prefer pass-by-reference-to-const over pass-by-value.

**Things to Remember**

✦ Prefer pass-by-reference-to-const over pass-by-value. It’s typically more efficient and it avoids the slicing problem.

✦ The rule doesn’t apply to built-in types and STL iterator and function object types. For them, pass-by-value is usually appropriate.

**Item 21: Don’t try to return a reference when you must return an object.**

Consider a class for representing rational numbers, including a function for multiplying two rational together:

class Rational {

public:

Rational(int numerator = 0, int denominator = 1); // ctor isn’t declared explicit

...

private:

int n, d; // numerator and denominator

friend const Rational operator\*(const Rational& lhs, const Rational& rhs);

};

This version of operator\* is returning its result object by value. Can we return it by reference?

Always remember that a reference is just a *name*, a name for some *existing* object. Whenever we see the declaration for a reference, should immediately ask yourself what it is another name for, because it must be another name for *something*.

In the case of operator\*, if the function is to return a reference, it must return a reference to some Rational object that already exists and that contains the product of the two objects that are to be multiplied together.

Expected result is:

Rational a(1, 2); // a = 1/2

Rational b(3, 5); // b = 3/5

Rational c = a \* b; // c should be 3/10

What is happening if operator \* return by reference, should c have value 3/10. Let see:

Answer is No, if operator\* is to return a reference to such a number, it must create that number object itself. A function can create a new object in only two ways: on the stack or on the heap. Creation on the stack is accomplished by defining a local variable see below code.

const Rational& operator\*(const Rational& lhs, const Rational& rhs) // warning! bad code!

{

Rational result(lhs.n \* rhs.n, lhs.d \* rhs.d);

return result;

}

A more serious problem is that this function returns a reference to result, but result is a local object, and local objects are destroyed when the function exits. So, returned object no longer exists in a memory. Plus, cost of constructor also included in this implementation.

Now, heap-based objects come into being through using new operator, so we might write a heap-based operator\*like this:

const Rational& operator\*(const Rational& lhs, const Rational& rhs) // warning! more bad

// code!

{

Rational \*result = new Rational(lhs.n \* rhs.n, lhs.d \* rhs.d);

return \*result;

}

Here who will apply delete to the object conjured up by your use of new? You think about client, see the below scenario:

Rational w, x, y, z;

w = x \* y \* z; // same as operator\*(operator\*(x, y), z)

Here, there are two calls to operator\* in the same statement, hence two uses of new that need to be undone with uses of delete. Yet there is no reasonable way for clients of operator\* to make those calls, because there’s no reasonable way for them to get at the pointers hidden behind the references being returned from the calls to operator\*. This is a guaranteed resource leak.

Cost of constructor also included in this implementation.

We noticed that both the on-the-stack and on-the-heap approaches suffer from having to call a constructor for each result returned from operator\*. Perhaps, our initial goal was to avoid such constructor invocations.

Think about the s*tatic* Rational object, one defined *inside* the function operator \*

const Rational& operator\*(const Rational& lhs, const Rational& rhs) // warning! yet more

// bad code!

{

static Rational result; // static object to which a

// reference will be returned

result = ... ; // multiply lhs by rhs and put

// the product inside result.

return result;

}

To see its deeper flaw, consider this perfectly reasonable client code:

bool operator==(const Rational& lhs, const Rational& rhs); // an operator == for

// Rationals

Rational a, b, c, d;

...

if ((a \* b) == (c \* d)) {

do whatever’s appropriate when the products are equal;

} else {

do whatever’s appropriate when they’re not;

}

Guess what? The expression ((a\*b) == (c\*d)) will *always* evaluate to true, regardless of the values of a, b, c, and d!

This revelation is easiest to understand when the code is rewritten in its equivalent functional form:

if (operator==(operator\*(a, b), operator\*(c, d)))

Notice that when operator== is called, there will already be *two* active calls to operator\*, each of which will return a reference to the static Rational object inside operator\*. Thus, operator== will be asked to compare the value of the static Rational object inside operator\* with the value of the static Rational object inside operator\*. It would be surprising indeed if they did not compare equal. Always.

The right way to write a function that must return a new object is to have that function return a new object. For Rational’s operator\*, that means either the following code or something essentially equivalent:

inline const Rational operator\*(const Rational& lhs, const Rational& rhs)

{

return Rational(lhs.n \* rhs.n, lhs.d \* rhs.d);

}

But what is about cost of constructing and destructing operator\*’s return value?

C++ allows compiler implementers to apply optimizations to improve the performance of the generated code without changing its observable behavior. construction and destruction of operator\*’s return value can be safely eliminated.

**Things to Remember**

✦ Never return a pointer or reference to a local stack object, a reference to a heap-allocated object, or a pointer or reference to a local static object if there is a chance that more than one such object will be needed. (returning a reference to a local static is

reasonable, at least in single-threaded environments).

**Item 22: Declare data members private.**

So, public data members. Why not?

If data members aren’t public, the only way for clients to access an object is via member functions.

If everything in the public interface is a function, clients won’t have to scratch their heads trying to remember whether to use parentheses when they want to access a member of the class.

If you make a data member public, everybody has read-write access to it, but if you use

functions to get or set its value, you can implement no access, read only access, and read-write access.

class AccessLevels {

public:

...

int getReadOnly() const { return readOnly; }

void setReadWrite(int value) { readWrite = value; }

int getReadWrite() const { return readWrite; }

void setWriteOnly(int value) { writeOnly = value; }

private:

int noAccess; // no access to this int

int readOnly; // read-only access to this int

int readWrite; // read-write access to this int

int writeOnly; // write-only access to this int

};

Many data members should be hidden. Rarely does every data member need a getter and setter.

Think about the encapsulation. If you hide your data members from your clients (i.e., encapsulate them), you can ensure that class invariants are always maintained, because only member functions can affect them.

If the data members are not private then, an unlimited number of functions can access them. They

have no encapsulation at all. For data members that *are* private, the number of functions that can access them is the number of member functions of the class plus the number of friend functions, because only members and friends have access to private members.

**Things to Remember**

✦ Declare data members private. It gives clients syntactically uniform access to data, affords fine-grained access control allows invariants to be enforced, and offers class authors implementation flexibility.

**Item 23: Prefer non-member non-friend functions to member functions.**

A class for representing web browsers:

class WebBrowser {

public:

...

void clearCache();

void clearHistory();

void removeCookies();

...

};

Many users will want to perform all these actions together, so Web-Browser might also offer a function to do just that:

class WebBrowser {

public:

...

void clearEverything(); // calls clearCache, clearHistory, and removeCookies

...

};

Of course, this functionality could also be provided by a non-member function that calls the appropriate member functions:

void clearBrowser(WebBrowser& wb)

{

wb.clearCache();

wb.clearHistory();

wb.removeCookies();

}

So which is better, the member function clearEverything or the non member function

clearBrowser?

Object-oriented principles dictate that data and the functions that operate on them should be bundled together, and that suggests that the member function is the better choice.

Unfortunately, this suggestion is incorrect. It’s all about the misunderstanding of OODP. Object-oriented principles dictate that data should be as *encapsulated* as possible. The member function clearEverything offers *less* encapsulation than the non-member clearBrowse. Non-member function allows for greater packaging flexibility for WebBrowser-related functionality, fewer compilation dependencies and an increase in WebBrowser extensibility.

If something is encapsulated, it’s hidden from view. The more something is encapsulated, the fewer things can see it. The fewer things can see it, the greater flexibility we have to change it, because our changes directly affect only those things that can see what we change.

That’s the reason we value encapsulation in the first place: it affords us the flexibility to change things in a way that affects only a limited number of clients.

Given a choice between a *member function* and a *non-member non-friend function* providing the same functionality, the choice yielding greater encapsulation is the non-member non-friend function, because it doesn’t increase the number of functions that can access the private parts of the class. This explains why clearBrowser (the nonmember non-friend function) is preferable to clearEverything (the member function). It provides greater encapsulation in the WebBrowser class.

Friends have the same access to a class’s private members that member functions have, hence If have impact on encapsulation.

In C++, a more natural approach would be to make clearBrowser a nonmember function in the same namespace as WebBrowser:

namespace WebBrowserStuff {

class WebBrowser { ... };

void clearBrowser(WebBrowser& wb);

...

}

**Item 24: Declare non-member functions when type conversions should apply to all parameters.**

Classes support implicit type conversions is generally a bad idea. But some time it is good. Like if you’re designing a class to represent rational numbers, allowing implicit conversions from integers to rationals

doesn’t seem unreasonable.

class Rational {

public:

Rational(int numerator = 0, int denominator = 1); // ctor is deliberately not explicit;

// allows implicit int-to-Rational

// conversions

int numerator() const; // accessors for numerator and

int denominator() const; // denominator

const Rational operator\*(const Rational& rhs) const; //operator \*

private:

...

}

Rational oneEighth(1, 8);

Rational oneHalf(1, 2);

Rational result = oneHalf \* oneEighth; // fine

result = result \* oneEighth; // fine

But we want like to support mixed-mode operations, where Rationals can be multiplied with int, for example:

result = oneHalf \* 2; oneHalf.operator\*(2) // fine

result = 2 \* oneHalf; 2\*operator\*(oneHalf) // error!

Since the multiplication is supposed to be commutative, both syntaxes will work.

The object oneHalf is an instance of a class that contains an operator\*, so compilers call that function. However, the integer 2 has no associated class, hence no operator\* member function. Compilers will also look for non-member operator\*s (i.e., ones at namespace or global scope) that can be called like this:

result = operator\*(2, oneHalf); // error!

But in this example, there is no non-member operator\* taking an int and a Rational, so the search fails.

const Rational temp(2); // create a temporary Rational object from 2

result = oneHalf \* temp; // same as oneHalf.operator\*(temp);

Compilers do this only because a non-explicit constructor is involved. If Rational’s constructor were explicit, neither of these statements would compile:

result = oneHalf \* 2; // error! (with explicit ctor);

// can’t convert 2 to Rational

result = 2 \* oneHalf; // same error, same problem

If Rational’s constructor were explicit, neither of the above statements would compile.

If we still like to support mixed-mode arithmetic, make operator\* a non-member function, thus allowing compilers to perform implicit type conversions on *all* arguments.

class Rational {

... // contains no operator\*

};

const Rational operator\*(const Rational& lhs, const Rational& rhs)

// now a non-member function

{

return Rational(lhs.numerator() \* rhs.numerator(), lhs.denominator() \* rhs.denominator());

}

Rational oneFourth(1, 4);

Rational result;

result = oneFourth \* 2; // fine

result = 2 \* oneFourth; // hooray, it works!

Now one question comes in our mind: ***Should operator\* be made a friend of the Rational class?***

In this case, the answer is no, because operator\* can be implemented entirely in terms of Rational’s public interface. The code above shows one way to do it. ***That leads to an important observation: the opposite of a member function is a non-member function, not a friend function****. M*any C++ programmers assume that if a function is related to a class and should not be a member, it should be a friend. This example demonstrates that such reasoning is flawed. Whenever you can avoid friend functions, you should, because, much as in real life, friends are often more trouble than they’re worth.

**Things to Remember**

✦ If you need type conversions on all parameters to a function (including the one that would otherwise be pointed to by this pointer), the function must be a non-member.

**Item 25: Consider support for a non-throwing swap.**

Swap is an interesting function. Originally introduced as part of the STL. Because swap is so useful, it’s important to implement it properly.

To swap the values of two objects is to give each the other’s value. By default, swapping is accomplished via the standard swap algorithm. Its typical implementation is exactly same as expected:

namespace std {

template<typename T> // typical implementation of std::swap;

void swap(T& a, T& b) // swaps a’s and b’s values

{

T temp(a);

a = b;

b = temp;

}

}

As long as our types/class support copying (via copy constructor and copy assignment operator), the default swap implementation will let objects of our types be swapped correctly.

Swap implementation involves copying three objects: a to temp, b to a, and temp to b. For some types, none of these copies are necessary. For such types, the default swap puts you on the fast track to the slow lane.

Consider the pimpl approach. A common manifestation of this design approach is the “pimpl idiom” (“pointer to implementation”). A Widget class employing such a design might look like this:

class WidgetImpl { // class for Widget data;

public: // details are unimportant

...

private:

int a, b, c; // possibly lots of data —

std::vector<double> v; // expensive to copy!

...

};

class Widget { // class using the pimpl idiom

public:

Widget(const Widget& rhs);

Widget& operator=(const Widget& rhs) // to copy a Widget, copy its

{ // WidgetImpl object.

...

\*pImpl = \*(rhs.pImpl);

...

}

...

private:

WidgetImpl \*pImpl; // ptr to object with this

}

To swap the value of two Widget objects, all we really need to do is swap their pImpl pointers, but the default swap algorithm has no way to know that. Result wrong answer.

template<typename T> // typical implementation of std::swap;

void swap(T& a, T& b) // swaps a’s and b’s values

{

T temp(a);

a = b;

b = temp;

}

class WidgetImpl {

public:

WidgetImpl():a(0),b(0),c(0) {}

WidgetImpl(int x, int y, int z) :a(x), b(y), c(z) {}

private:

int a, b, c;

};

class Widget {

public:

Widget(int x, int y, int z) {

impl = new WidgetImpl(x, y, z);

}

Widget(const Widget &rhs){

impl = new WidgetImpl;

impl = rhs.impl;

}

Widget& operator =(const Widget &rhs) {

if (this == &rhs) return \*this;

delete impl;

impl = new WidgetImpl;

impl = rhs.impl;

return \*this;

}

~Widget() {

delete impl;

}

private:

WidgetImpl \*impl;

};

int main() {

Widget w1(10, 20, 30),w2(100,200,300);

swap(w1,w2);

}

This Program will not work desirably.

What we’d like to do is tell std::swap that when Widgets are being swapped, the way to perform the swap is to swap their internal pImpl pointers. There is a way to say exactly that: specialize std::swap for Widget. Now change little bit the implementation of Widget class.

When Widgets are being swapped, the way to perform the swap is to swap their internal pImpl pointers. There is a way to say exactly that: specialize std::swap for Widget.

namespace std {

template<> // this is a specialized version

void swap<Widget>(Widget& a, Widget& b) // of std::swap for when T is Widget

{

swap(a.pImpl, b.pImpl); // to swap Widgets, swap their

} // pImpl pointers; this won’t

// compile

}

The “template<>” at the beginning of this function says that this is a *total template specialization* for std::swap, and the “<Widget>” after the name of the function says that the specialization is for when T is Widget.

In other words, when the general swap template is applied to Widgets, this is the implementation that should be used. We’re not permitted to alter the contents of the std namespace, but we can totally specialize standard templates (like swap) for types of our own creation (such as Widget). That’s what we’re doing here.

However, this function won’t compile. That’s because it’s trying to access the pImpl pointers inside a and b, and they’re private.

class Widget { // same as above, except for the addition of the swap mem func.

public:

...

...

void swap(Widget& other)

{

std::swap(impl, other.impl); //to swap Widgets, swap

//their pImpl pointers

}

...

private:

WidgetImpl \*impl;

};

namespace std {

template<> // revised specialization of std::swap

void swap<Widget>(Widget& a, Widget& b)

{

a.swap(b); // to swap Widgets, call their

} // swap member function

}

int main() {

Widget w1(10, 20, 30),w2(100,200,300);

w1.swap(w2);

}

This will compile also consistent with the STL containers, all of which provide both public swap member functions and versions of std::swap that call these member functions.

**Things to Remember**

✦ Provide a swap member function when std::swap would be inefficient for your type. Make sure your swap doesn’t throw exceptions.

✦ If you offer a member swap, also offer a non-member swap that calls the member. For classes (not templates), specialize std::swap, too.

✦ When calling swap, employ a using declaration for std::swap, then call swap without namespace qualification.

✦ It’s fine to totally specialize std templates for user-defined types, but never try to add something completely new to std.