**Item 2: Prefer consts, enums, and inlines to #defines.**

#define ASPECT\_RATIO 1.653

Prefer to replace the macro with a constant

const double AspectRatio = 1.653; // uppercase names are usually for macros, hence the name

// change

Why?

Macro is processed by pre-processor and expended code contains value 1.653 rather than ASPECT\_RATIO which is more meaningful. The symbolic name ASPECT\_RATIO may never be seen by compilers; it may be removed by the preprocessor before the source code ever gets to a compiler.

This can be confusing if we get an error during compilation involving the use of the constant, because the error message may refer to 1.653, not ASPECT\_RATIO.

There’s no way to create a class-specific constant using a #define, because #defines don’t respect scope.

class GamePlayer {

private:

static const int NumTurns = 5; // constant declaration

int scores[NumTurns]; // use of constant

};

As of now it has only declaration for NumTurns, not a definition.Usually, C++ requires that you provide a definition for anything you use, but class-specific constants that are static and of integral type (e.g., integers, chars, bools) are an exception.

As long as you don’t take their address, you can declare them and use them without providing a definition. If you do take the address of a class constant, or if your compiler incorrectly insists on a definition even if you don’t take the address, you provide a separate definition like this:

const int GamePlayer::NumTurns; //definition of NumTurns see below for why no value is given

// call f with the maximum of a and b

#define CALL\_WITH\_MAX(a, b) f((a) > (b) ? (a) : (b))

int a = 5, b = 0;

CALL\_WITH\_MAX(++a, b); // a is incremented twice

CALL\_WITH\_MAX(++a, b+10); // a is incremented once

You can get all the efficiency of a macro plus all the predictable behavior and type safety of a regular function by using a template for an inline function

template<typename T>

inline void callWithMax(const T& a, const T& b)

{

a > b ? a : b);

}

This template generates a whole family of functions, each of which takes two objects of the same type and calls f with the greater of the two objects. There’s no need to parenthesize parameters inside the

function body, no need to worry about evaluating parameters multiple times, etc. Furthermore, because callWithMax is a real function, it obeys scope and access rules. For example, it makes perfect sense to

talk about an inline function that is private to a class. In general, there’s just no way to do that with a macro.

**Things to Remember**

✦ For simple constants, prefer const objects or enums to #defines.

✦ For function-like macros, prefer inline functions to #defines.

**Item 3: Use const whenever possible.**

you can specify whether the pointer itself is const, the data it points to is const, both, or neither:

char greeting[] = "Hello";

char \*p = greeting; // non-const pointer, non-const data

const char \*p = greeting; // non-const pointer, const data

char \* const p = greeting; // const pointer, non-const data

const char \* const p = greeting; // const pointer, const data

If the word const appears to the left of the asterisk, what’s pointed to is constant; if the word const appears to the right of the asterisk, the pointer itself is constant; if const appears on both sides, both are constant.

void f1(const Widget \*pw); // f1 takes a pointer to a constant Widget object

void f2(Widget const \*pw); // so does f2

Constness in STL

std::vector<int> vec;

const std::vector<int>::iterator iter = vec.begin() // iter acts like a T\* const

\*iter = 10; // OK, changes what iter points to

++iter; // error! iter is const

std::vector<int>::const\_iterator cIter = vec.begin() // cIter acts like a const T\*

\*cIter = 10; // error! \*cIter is const

++cIter; // fine, changes cIter

Function with return const type:

Having a function return a constant value is generally inappropriate, but sometimes doing so can reduce the incidence of client errors without giving up safety or efficiency.

class Rational { ... };

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

For Example: Why should the result of operator\* be a const object? Because if it weren’t, clients would be able to commit atrocities like this:

Rational a, b, c;

...

(a \* b) = c; // invoke operator= on the result of a\*b

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

**Const Member Functions used for:**

It’s important to know which functions may modify an object and which may not.

They make it possible to work with const objects.

class TextBlock {

public:

...

const char& operator[](std::size\_t position) const // operator[] for const objects

{

return text[position];

//Here return type also const required, because this method

//returns reference of text string and you cont modify this later point of

//time also .

}

char& operator[](std::size\_t position) // operator[] for non-const objects

{

return text[position];

}

std::string text;

};

TextBlock’s operator [] can be used like this:

TextBlock tb("Hello");

std::cout << tb[0]; // calls non-const

const TextBlock ctb("World");

std::cout << ctb[0]; // calls const

What does it mean for a member function to be const?

There are two prevailing notions:

1. *Bitwise constness* (also known as *physical constness*) and
2. *logical constness*

***Bitwise constness***

Bitwise constness state that if any member function is declared as const then it doesn’t modify any of the bits inside the object (excluding those that are static).

C++’s definition of bitwise constness, a const member function isn’t allowed to modify any of the non-static data members of the object on which it is invoked.

***logical constness***

See the below program:

Suppose we have a TextBlock-like class that stores its data as a char\* instead of a string, because it needs to communicate through a C API that doesn’t understand string objects.

class CTextBlock {

public:

...

char& operator[](std::size\_t position) const // inappropriate (but bitwise const) declaration

//of operator[]

{

return pText[position];

}

private:

char \*pText;

};

This class (inappropriately) declares operator[] as a const member function, even though that function returns a reference to the object’s internal data .Operator[]’s implementation doesn’t modify pText in any way. As a result, compilers will happily generate code for operator[]; it is,

after all, bitwise const, and that’s all compilers check for. But look what it allows to happen:

const CTextBlock cctb("Hello"); // declare constant object

char \*pc = &cctb[0]; // call the const operator[] to get a

// pointer to cctb’s data

\*pc = ’J’; // cctb now has the value “Jello”

Surely there is something wrong when we create a constant object with a particular value and invoke only const member functions on it, yet we can still change its value.

***This leads to the notion of logical constness***

Modify member data inside const member function using *mutable* keyword.

class CTextBlock {

public:

...

std::size\_t length() const;

private:

char \*pText;

mutable std::size\_t textLength; // these data members may always be modified, even in

mutable bool lengthIsValid; // const member functions

};

std::size\_t CTextBlock::length() const

{

if (!lengthIsValid) {

textLength = std::strlen(pText); // now fine

lengthIsValid = true; // also fine

}

return textLength;

}

**Avoiding Duplication in const and Non-const Member Functions.**

We have same method for const and non const version. For example operator [], here problem with redundant code which is appear in both methods.

In this case, the const version of operator[] does exactly what the non-const version does, it just has a const-qualified return type.

class TextBlock {

public:

...

const char& operator[](std::size\_t position) const

{

// do bounds checking, log access data and verify data integrity

return text[position];

}

char& operator[](std::size\_t position)

{

// do bounds checking, log access data and verify data integrity

return text[position];

}

private:

std::string text;

};

Casting away the const on the return value is safe (not possible in vice-versa), in this case, because whoever called the non-const operator [] must have had a non const object in the first place. Otherwise they couldn’t have called a non-const function, so having the non-const operator [] call the const version is a safe way to avoid code duplication.

class TextBlock {

public:

...

const char& operator[](std::size\_t position) const // same as before

{

// do bounds checking

// log access data

// verify data integrity

if(text.length() < position){

cout<<"Invalid index ::"<<position<<endl;

return text[text.length()-1];

}

else{

cout<<"Index Validation successful\n";

}

return text[position];

}

char& operator[](std::size\_t position) // now just calls const op[]

{

// 1. add const to \*this’s type;

// 2. call const version of op[]

// 3. cast away const on op[]’s return type;

return const\_cast<char&>( static\_cast<const TextBlock&>(\*this) [position] );

}

};

**Things to Remember**

✦ Declaring something const helps compilers detect usage errors. Const can be applied to objects at

any scope, to function parameters and return types, and to member functions as a whole.

✦ Compilers enforce bitwise constness, but you should program using logical constness.

✦ When const and non-const member functions have essentially identical implementations, code

duplication can be avoided by having thenon-const version call the const version.

**Item 4: Make sure that objects are initialized before they’re used**

class PhoneNumber { ... };

class ABEntry { // ABEntry = “Address Book Entry”

public:

ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones);

private:

std::string theName;

std::string theAddress;

std::list<PhoneNumber> thePhones;

int numTimesConsulted;

};

ABEntry::ABEntry(const std::string& name, const std::string& address,

const std::list<PhoneNumber>& phones)

{

theName = name; // these are all assignments,

theAddress = address; // not initializations

thePhones = phones;

numTimesConsulted = 0;

}

The rules of C++ stipulate that data members of an object are initialized *before* the body of a constructor is entered. Inside the ABEntry constructor, theName, theAddress, and thePhones

aren’t being initialized, they’re being *assigned*. Initialization took place earlier — when their default constructors were automatically called prior to entering the body of the ABEntry constructor.

ABEntry::ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones)

: theName(name),

theAddress(address), // these are now all initializations

thePhones(phones),

numTimesConsulted(0)

{} // the ctor body is now empty

The assignment-based version first called default constructors to initialize theName, theAddress, and thePhones, then promptly assigned new values on top of the default-constructed ones. All the work performed in those default constructions was therefore wasted. The member initialization list approach avoids that problem, because the arguments in the initialization list are used as constructor arguments for the various data members. In this case, theName is copy-constructed from name, theAddress is copy-constructed from address, and thePhones is copy-constructed from phones.

if ABEntry had a constructor taking no parameters, it could be implemented like this:

ABEntry::ABEntry()

: theName(), // call theName’s default ctor;

theAddress(), // do the same for theAddress;

thePhones(), // and for thePhones;

numTimesConsulted(0) {} // but explicitly initialize numTimesConsulted to zero

Sometimes the initialization list *must* be used, even for built-in types. For example, data members that are const or are references must be initialized; they can’t be assigned.

Base classes are initialized before derived classes , and within a class, data members are initialized in the order in which they are declared. In ABEntry, for example, theName will always be initialized first, theAddress second, thePhones third, and numTimesConsulted last. This is true even if they are listed in a different order on the member initialization list.

*Initialization of static object & data members.*

Static objects inside functions are known as *local static objects* (because they’re local to a function), and the other kinds of static objects are known as *non-local static objects*(i.e., an object that’s global, at namespace scope, or static in a class or at file scope)

Static objects are destroyed when the program exits, i.e., their destructors are called when main finishes executing.

Suppose there is at least two separately compiled source files, each of which contains at least one nonlocal static object. And the actual problem is this:

if initialization of a non-local static object in one translation unit uses a non-local static object in a different translation unit, the object it uses could be uninitialized, because *the relative order of initialization of nonlocal static objects defined in different translation units is undefined*.

**Before proceeding need to understand the extern keyword role.**

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

**First File: main.c**

#include <stdio.h>

int count;

extern void write\_extern(); //Here only declaration, definition is somewhere else

main() {

count = 5; write\_extern();

}

**Second File: support.c**

extern int count; //Here only declaration, definition is somewhere else

void write\_extern(void) {

printf("count is %d\n", count);

}

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

$gcc main.c support.c

It will produce the executable program a.out. When this program is executed, it produces the following result −

count is 5

We want to create a special object at global or namespace scope representing the single file system

class FileSystem

{ // from your library’s header file

public:

...

std::size\_t numDisks() const; // one of many member functions

...

};

extern FileSystem tfs; // declare object for clients to use (tfs =“the file system” ) // definition is in some .cpp file in your library

A FileSystem object is decidedly non-trivial, so use of the tfs object before it has been constructed would be disastrous.

Now suppose some client creates a class for directories in a file system. Naturally, their class uses the tfs object:

class Directory { // created by library client

public:

Directory ( params );

...

};

Directory::Directory( params )

{...

std::size\_t disks = tfs.numDisks(); // use the tfs object

...

}

Directory tempDir( params ); // directory for temporary files

Now the importance of initialization order becomes apparent:

tfs is initialized before tempDir, because tempDir’s constructor will attempt to use tfs. But tfs and tempDir were created by different people at different times in different source files — they’re non-local static objects defined in different translation units. How can you be sure that tfs will be initialized before tempDir?

Determining the “proper” order in which to initialize non-local static objects is hard. Very hard. Unsolvably hard.

**Solution**:

Fortunately, a small design pattern (Singleton) change eliminates the problem entirely. All that must be done is to move each non-local static object into its own function, where it’s declared static. These functions return references to the objects they contain. Clients then call the functions instead of referring to the objects. *In other words, non-local static objects are replaced with local static objects.*

C++’s guarantee that local static objects are initialized when the object’s definition is first encountered during a call to that function.

So if you replace direct accesses to non-local static objects with calls to functions that return references to local static objects, you’re guaranteed that the references you get back will refer to initialized objects.

As a bonus, if you never call a function emulating a non-local static object, you never incur the cost of constructing and destructing the object, something that can’t be said for true non-local static objects.

class FileSystem { ... }; // as before

FileSystem& tfs() // this replaces the tfs object; it could be

{ // static in the FileSystem class

static FileSystem fs; // define and initialize a local static object

return fs; // return a reference to it

}

class Directory { ... }; // as before

Directory::Directory( params ) // as before, except references to tfs are

{ // now to tfs()

...

std::size\_t disks = tfs().numDisks();

...

}

Directory& tempDir() // this replaces the tempDir object; it

{ // could be static in the Directory class

static Directory td( params ); // define/initialize local static object

return td; // return reference to it

}

now client refer to tfs() and tempDir() instead of tfs and tempDir. These functions are returning references to objects instead of using the objects themselves.

**Things to Remember**

✦ Manually initialize objects of built-in type, because C++ only sometimes initializes them itself.

✦ In a constructor, prefer use of the member initialization list to assignment inside the body of the

constructor. List data members in the initialization list in the same order they’re declared in the class.

✦ Avoid initialization order problems across translation units by replacing

non-local static objects with local static objects.

**Item 5: Know what functions C++ silently writes and calls.**

C++ compiler creates own versions of a copy constructor, a copy assignment operator, and a destructor. Furthermore, if you declare no constructors at all, compilers will also declare a default constructor for you. All these functions will be both public and inline.

class Empty{};

it’s essentially the same as if you’d written this:

class Empty {

public:

Empty() { ... } // default constructor

Empty(const Empty& rhs) { ... } // copy constructor

~Empty() { ... } // destructor — see below for whether it’s virtual

Empty& operator=(const Empty& rhs) { ... } // copy assignment operator

};

Note that the generated destructor is non-virtual unless it’s for a class inheriting from a base class that itself

declares a virtual destructor (in which case the function’s virtual ness comes from the base class).

suppose NamedObject were defined like this, where nameValue is a *reference* to a string and objectValue is a const T:

template<typename T>

class NamedObject {

public:

NamedObject(std::string& name, const T& value);

...

// as above, assume no operator= is declared

private:

std::string& nameValue; // this is now a reference

const T objectValue; // this is now const

};

Now consider what should happen here:

std::string newDog("Persephone");

std::string oldDog("Satch");

NamedObject<int> p(newDog, 2); // when I originally wrote this, our dog Persephone was

//about to have her second birthday

NamedObject<int> s(oldDog, 36); // the family dog Satch (from my childhood) would be 36 if

// she were still alive

p = s; // what should happen to the data members in p?

**Item 6: Explicitly disallow the use of compiler generated functions you do not want.**

Declare the copy constructor and the copy assignment operator private and both are not defined.

class Test {

Test(const Test &rhs); //If you are calling, then only need to provide body.

//Here want to prevent to call, don’t provide

//definition (body).

Test& operator=(const Test &rhs);

public:

Test(){}

};

Other Ways:

int main() {

Test t1, t2;

t1 = t2; //Complile time error, function inaccessable.

Test t3(t1); //Compile time error. function inaccessable.

}

class Uncopyable {

protected: // allow construction

Uncopyable() {} // and destruction of

~Uncopyable() {} // derived objects...

private:

Uncopyable(const Uncopyable&); // ...but prevent copying

Uncopyable& operator=(const Uncopyable&);

};

class TestUncopyable :public or private Uncopyable {

};

TestUncopyable p1, p2;

p1 = p2; //Canot be refrenced its deleted function.

**Things to Remember**

To disallow functionality automatically provided by compilers, declare the corresponding member functions private and give no implementations. Using a base class like Uncopyable is one way to do this.

**Item 7: Declare destructors virtual in polymorphic base classes.**

TimeKeeper base class along with derived classes for different approaches to timekeeping

class TimeKeeper {

public:

TimeKeeper();

~TimeKeeper();

...

};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };

Many clients will want access to the time without worrying about the details of how it’s calculated, so a factory function — a function that returns a base class pointer to a newly-created derived class object can be used to return a pointer to a timekeeping object:

TimeKeeper \*ptk = getTimeKeeper(); // get dynamically allocated object from TimeKeeper hierarchy

... // use it

delete ptk; // release it to avoid resource leak

The problem is that getTimeKeeper returns a pointer to a derived class object (e.g., AtomicClock), that object is being deleted via a base class pointer (i.e., a TimeKeeper\* pointer), and the base class (TimeKeeper)

has a non-virtual destructor.

This is a recipe for disaster, because C++ specifies that when a derived class object is deleted through a pointer to a base class with a non-virtual destructor, results are undefined.

In such case derived object not be destroyed, nor would the AtomicClock destructor run. However,the base class part would be destroyed, thus leading to a curious “partially destroyed” object. This is an excellent way to leak resources, corrupt data structures, and spend a lot of time with a debugger.

**Solution: Keep base class destructor virtual.**

If a class does not contain virtual functions, that often indicates it is not meant to be used as a base class. When a class is not intended to be a base class, making the destructor virtual is usually a bad idea. Consider a class for representing points in two-dimensional space:

class Point { // a 2D point

public:

Point(int xCoord, int yCoord);

~Point();

private:

int x, y;

};

Here the size of Point object is 64 bit.

Cost of virtual function:

The implementation of virtual functions requires that objects carry information that can be used at runtime to determine which virtual functions should be invoked on the object. This information typically takes the form of a pointer called a vptr (“virtual table pointer”). The vptr points to an array of function pointers called a vtbl (“virtual table”); each class with virtual functions has an associated vtbl. When a virtual function is invoked on an object, the actual function called is

determined by following the object’s vptr to a vtbl and then looking up the appropriate function pointer in the vtbl.

So, it would require too much complexity and memory also. *If the class is not meant for base class (inheritance) then avoid putting virtual mechanism into it.*

In fact, many people summarize the situation this way: declare a virtual destructor in a class if and only if that class contains at least one virtual function ( i,e polymorphic base class)

class SpecialString: public std::string {

// bad idea! std::string has a non-virtual destructor

…..

};

SpecialString \*pss =new SpecialString("Impending Doom");

std::string \*ps;

...

ps = pss; // SpecialString\* ⇒ std::string\*

...

delete ps; // undefined! In practice,

Here SpecialString resources will be leaked, because the SpecialString destructor won’t be called.

**Things to Remember**

✦ Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it

should have a virtual destructor.

✦ Classes not designed to be base classes or not designed to be used polymorphically should not

declare virtual destructors

**Item 8: Prevent exceptions from leaving destructors.**

Consider the below Program:

class Widget {

public:

...

~Widget() { ... } // assume this might emit an exception

};

Int main(){

std::vector<Widget> v;

...

} // v is automatically destroyed here

When the vector v is destroyed, it is responsible for destroying all the Widgets it contains. Suppose v has ten Widgets in it, and during destruction of the first one, an exception is thrown. The other nine Widgets still must be destroyed, so v should invoke their destructors. But suppose that during those calls, a second Widget destructor throws an exception. Now there are two simultaneously active exceptions. But C++ does not support multiple active exception, thus result is undefined behavior.

Another Example:

class DBConnection {

public:

...

static DBConnection create(); // function to return DBConnection objects; params

// omitted for simplicity

void close(); // close connection; throw an exception if closing fails.

};

RAII Approch to redefine the above class.

class DBConn { // class to manage DBConnection objects

public:

...

static DBConnection create();

~DBConn(){ // make sure database connections are always closed

db.close();

}

private:

DBConnection db;

};

{ // open a block

DBConn dbc(DBConnection::create()); // create DBConnection object and turn it over to a DBConn

// object to manage use the DBConnection object via the DBConn // interface

}

At end of block, the DBConn object is destroyed, thus automatically calling close on the DBConnection object.

This is fine if the call to close succeeds, but if the call yields an exception, DBConn’s destructor will propagate that exception, i.e., allow it to leave the destructor. That’s a problem, because destructors that throw mean trouble.

There are two primary ways to avoid the trouble. DBConn’s destructor could:

**Terminate the program** if close throws, typically by calling abort:

DBConn::~DBConn()

{

try { db.close(); }

catch (...) {

make log entry that the call to close failed;

std::abort();

}

}

**Swallow the exception** arising from the call to close:

DBConn::~DBConn()

{

try { db.close(); }

catch (...) {

make log entry that the call to close failed;

}

}

In general, swallowing exceptions is a bad idea, because it suppresses important information — something failed! Sometimes, however, swallowing exceptions is preferable to running the risk of premature program termination or undefined behavior. For this to be a viable option, the program must be able to reliably continue execution even after an error has been encountered and ignored.

Neither of these approaches is especially appealing. The problem with both is that the program has no way to react to the condition that led to close throwing an exception in the first place.

Better approach to minimize the risk of exception form destructor. Test every possibility to not generate the exception. For ex- Lets redesign the DBConn class. This time we will consider below points also:

* We know the if the connection is already close and again try to close the connection it may cause the exception.
* If connection is interrupted with SQL server and then try to close the connect again lead to exception.

Now our DBConn interface like:

class DBConn {

public:

...

void close() // new function for client use

{

db.close();

closed = true;

}

~DBConn()

{

if (! closed && ConnectionEatablished) {

try { // close the connection if the client didn’t

db.close();

}

catch (...) { // if closing fails,

*make log entry that call to close failed;* // note that and

... // terminate or swallow

}

}

}

private:

DBConnection db;

bool closed;

};

Still there is a chance to throw the exception form DBConn destructor, however, we’d be

back to terminating or swallowing. We don’t have any other choice.

**Things to Remember**

✦ Destructors should never emit exceptions. If functions called in a destructor may throw, the

destructor should catch any exceptions, then swallow them or terminate the program.

✦ If class clients need to be able to react to exceptions thrown during an operation, the class should

provide a regular (i.e., non-destructor) function that performs the operation.

**Item 9: Never call virtual functions during construction or destruction.**

class Transaction { // base class for all transactions

public:

Transaction(){

logTransaction(); //virtual function called from constructor

}

virtual void logTransaction() const = 0; // make type-dependent log entry

...

}

class BuyTransaction: public Transaction { // derived class

public:

virtual void logTransaction() const; // how to log transactions of this type

...

};

class SellTransaction: public Transaction { // derived class

public:

virtual void logTransaction() const; // how to log transactions of this type

...

};

Main(){

BuyTransaction b;

}

Note: *Transaction has pure virtual function that means it is an abstract class. We are unable to create the object directly but when we inherit and create the object of derived class, its base class ctor will execute.*

Here BuyTransaction constructor will be called, but first, a Transaction constructor must be called(because of base class construct first). The last line of the Transaction constructor calls the virtual function logTransaction, but this is where the surprise comes in. The version of logTransaction that’s called is the one in Transaction, *not* the one in BuyTransaction — even though the type of object being created is BuyTransaction.

During base class construction, virtual functions never go down into derived classes.

Because base class constructors execute before derived class constructors, derived class data members have not been initialized when base class constructors run. If virtual functions called during base class construction went down to derived classes, the derived class functions would almost certainly refer to local data members, but those data members would not yet have been initialized. This would lead the undefined behavior of a program.

This program will not compile it would give the linker error:

Because the logTransaction function is pure virtual in Transaction. Unless it had been defined, the program wouldn’t link: the linker would be unable to find the necessary implementation of Transaction::logTransaction.

*Error LNK2019 unresolved external symbol "public: virtual void Transaction::logTransaction(void) const" referenced in function "public: Transaction::Transaction(void)"*

*However we can remove this error by defining base class logTransaction() method.*

void Transaction::logTransaction()const {

std::cout << "Trx\n";

}

*Even this method is pure virtual, complier will happily accept definition of this method, but overall output of this program is wrong.*

*We are creating object of BuyTransaction* and expecting the derived class method of *logTransaction() will called, but it always call its base class variant method which is totally unacceptable.*

Consider the below program, which is conceptually same as above.

class Transaction {

public:

Transaction() {

init(); //virtual function called from constructor method init

std::cout << "Trx created::\n";

}

virtual void logTransaction() const = 0; // make type-dependent log entry

private:

void init() {

logTransaction();

}

};

This code is conceptually the same as the earlier version, but this time it is more dangerous, because it will typically compile and link without complaint. In this case, because logTransaction is pure virtual in Transaction, most runtime systems will abort the program when the pure virtual is

Called.

However, if logTransaction were a “normal” virtual function (i.e., not pure virtual) with an

implementation in Transaction, that version would be called.

*The only way to avoid this problem is to make sure that none of your constructors or destructors call virtual functions on the object being created or destroyed and that all the functions they call*

*obey the same constraint.*

**Things to Remember**

✦ Don’t call virtual functions during construction or destruction, because such calls will never go to

a more derived class than that of the currently executing constructor or destructor.

**Item 10: Have assignment operators return a reference to \*this.**

int x, y, z;

x = y = z = 15; // chain of assignments

Assignment is right-associative, so the above assignment chain is parsed like this:

x = (y = (z = 15));

Here, 15 is assigned to z, then the result of that assignment (the updated z) is assigned to y, then the result of that assignment (the updated y) is assigned to x.

The way this is implemented is that assignment returns a reference to its left-hand argument, and that’s the convention you should follow when you implement assignment operators for your classes:

class Widget {

public:

Widget& operator=(const Widget& rhs) // return type is a reference to the current class

{

...

return \*this; // return the left-hand object

}

Widget& operator+=(const Widget& rhs) // the convention applies to +=, -=, \*=, etc.

{

...

return \*this;

}

};

**Things to Remember**

✦ Have assignment operators return a reference to \*this.

**Item 11: Handle assignment to self in operator=.**

An assignment to self occurs when an object is assigned to itself:

class Widget { ... };

Widget w;

...

w = w; // assignment to self, this looks silly, but it’s legal.

Another valid example is:

If the two objects need not even be declared to be of the same type if they’re from the same hierarchy, because a base class reference or pointer can refer or point to an object of a derived class type:

class Base { ... };

class Derived: public Base { ... };

void doSomething(Base& rb, Derived\* pd); // \*rb and \*pd might actually be point

// the same object.

If we try to manage resources our self, then possible to the trap of accidentally releasing a resource before we are done using it. For example, suppose you create a class that holds a raw pointer to a dynamically allocated bitmap:

class Bitmap { ... };

class Widget {

...

Widget& Widget::operator=(const Widget& rhs) // unsafe impl. of operator=

{

delete pb; // stop using current bitmap

pb = new Bitmap(\*rhs.pb); // start using a copy of rhs’s bitmap

return \*this;

}

private:

Bitmap \*pb; // ptr to a heap-allocated object

};

Self-assignment problem here is that inside operator=.

w=w;

Here the \*this and rhs both point the same object w. Inside the operator=(), we are deleting the object which is same as rhs. And at the end of function call we returned the deleted object.

The traditional way to prevent this error is to check for assignment to self via an identity test at the top of operator=:

Widget& Widget::operator=(const Widget& rhs)

{

if (this == &rhs) return \*this; // identity test: if a self-assignment, do nothing

delete pb;

pb = new Bitmap(\*rhs.pb);

return \*this;

}

**Things to Remember**

✦ Make sure operator= is well-behaved when an object is assigned to itself.

✦ Make sure that any function operating on more than one object behaves correctly if two or more

of the objects are the same.

**Item 12: Copy all parts of an object.**

only two functions copy objects: copy constructor and copy assignment operator.

We’ll call these the copying functions. Compilers will generate the copying functions, and compiler-generated versions do precisely what we do expect, they copy all the data of the object being copied.

When you declare your own copying functions, you are indicating to compilers that there is something about the default implementations you don’t like.

void logCall(const std::string& funcName) {

std::cout << funcName.c\_str() << "\n";

}

class Customer {

public:

Customer(const std::string str):name(str) {}

Customer(const Customer& rhs):name(rhs.name) {

logCall("Customer copy constructor");

}

Customer& operator=(const Customer& rhs) {

logCall("Customer copy assignment operator");

name = rhs.name; // copy rhs’s data

return \*this;

}

private:

std::string name;

};

Customer c1("Rajeev");

Customer c2("Suresh");

Suppose we added more field on class Customer Date lastTransaction. At this point, the existing copying functions are performing a partial copy. The conclusion is obvious: if you add a data member to a class, you need to make sure that you update the copying functions, too.

This issue can arise is through inheritance also.

class PriorityCustomer: public Customer { //Derived class

public:

...

PriorityCustomer(const PriorityCustomer& rhs);

PriorityCustomer& operator=(const PriorityCustomer& rhs);

...

private:

int priority;

};

//Below copy ctor not compile, It will ask about the base class default ctor

//which we have don’t created.

PriorityCustomer(const PriorityCustomer& rhs):priority(rhs.priority){

logCall("PriorityCustomer copy constructor");

}

//This assignment overload missed base class member copy.

PriorityCustomer& operator=(const PriorityCustomer& rhs) {

logCall("PriorityCustomer copy assignment operator");

priority = rhs.priority;

return \*this;

}

Every PriorityCustomer also contains a copy of the data members it inherits from Customer, and

those data members are not being copied at all. PriorityCustomer’s copy constructor specifies no arguments to be passed to its base class constructor. Similar thing happened with assignment operator overload also.

Solution: derived class copying functions must invoke their corresponding base class functions. Correct implementation is:

PriorityCustomer(const PriorityCustomer& rhs): Customer(rhs), // invoke base class

// copy ctor

priority(rhs.priority)

{

logCall("PriorityCustomer copy constructor");

}

PriorityCustomer& operator=(const PriorityCustomer& rhs)

{

logCall("PriorityCustomer copy assignment operator");

Customer::operator=(rhs); //assign base class parts

priority = rhs.priority;

return \*this;

}

When you’re writing a copying function, be sure to

(1) copy all local data members and

(2) invoke the appropriate copying function in all base classes.

We can observe one thing is code duplication while implementing the both version of copying function.

But, there is no sense to have the copy assignment operator call the copy constructor, because we are trying to construct an object that already exists.

On other hand having the copy constructor call the copy assignment operator — is equally nonsensical. A constructor initializes new objects, but an assignment operator applies only to objects that have already been initialized.

Instead, if you find that your copy constructor and copy assignment operator have similar code bodies, eliminate the duplication by creating a third member function that both calls. Such a function is typically private and is often named init(). This strategy is a safe, proven way to eliminate code duplication in copy constructors and copy assignment operators.

**Things to Remember**

✦ Copying functions should be sure to copy all of an object’s data members and all of its base class

parts.

✦ Don’t try to implement one of the copying functions in terms of the other. Instead, put common

functionality in a third function that both calls.

**Item 13: Use objects to manage resources.**

class Investment { ... }; // root class of hierarchy of investment types

below method return ptr to dynamically allocated object in the Investment hierarchy the caller must delete it (parameters omitted for simplicity)

Investment\* createInvestment();

void f()

{

Investment \*pInv = createInvestment(); // call factory function

... // use pInv

delete pInv; // release object

}

There might be a premature return statement somewhere inside the “...” part of the function.

If such a return were executed, control would never reach the delete statement. Result memory leak.

The standard library’s auto\_ptr is tailor made for this kind of situation. auto\_ptr is a pointer-like object (a smart pointer) whose destructor automatically calls delete on what it points to. Here’s how to use auto\_ptr to prevent f’s potential resource leak:

void f()

{

std::auto\_ptr<Investment> pInv(createInvestment()); // call factory function

... // use pInv as before

} // automatically delete pInv via auto\_ptr’s dtor

This simple example demonstrates the two critical aspects of using objects to manage resources:

* **Resources are acquired and immediately turned over to resource-managing objects.**

The idea of using objects to manage resources is often called *Resource Acquisition Is Initialization* (RAII).

* **Resource-managing objects use their destructors to ensure that resources are released.**

auto\_ptr automatically deletes what it points to when the auto\_ptr is destroyed, it’s important that there never be more than one auto\_ptr pointing to an object. If there were, the object would be deleted more than once, and would lead the undefined behavior of program . To prevent such problems, auto\_ptrs have an unusual characteristic:

Copying them (via copy constructor or copy assignment operator) sets them to null, and the copying pointer assumes sole ownership of the resource.

std::auto\_ptr<Investment> pInv1(createInvestment()); // pInv1 points to the object returned from

//createInvestment

std::auto\_ptr<Investment> pInv2(pInv1); // pInv2 now points to the object pInv1 is now null

pInv1 = pInv2; // now pInv1 points to the object, and pInv2 is null

Resources managed by auto\_ptrs must never have more than one auto\_ptr pointing to them.

***STL containers require that their contents exhibit “normal” copying behavior, so containers of auto\_ptr aren’t allowed.***

An alternative to auto\_ptr is a *reference-counting smart pointer* (RCSP). An RCSP is a smart pointer that keeps track of how many objects point to a particular resource and automatically deletes the resource when nobody is pointing to it any longer. As such, RCSPs offer behavior that is similar to that of garbage collection.

void f()

{

...

// pInv1 points to the object returned from createInvestment

std::tr1::shared\_ptr<Investment> pInv1(createInvestment());

std::tr1::shared\_ptr<Investment> pInv2(pInv1); // both pInv1 and pInv2 now

// point to the object.

pInv1 = pInv2; // ditto — nothing has changed

...

} //pInv1 and pInv2 are destroyed, and the object they point to is automatically deleted.

***Because copying shared\_ptrs works “as expected,” they can be used in STL containers and other contexts where auto\_ptr’s unorthodox copying behavior is inappropriate.***

Both auto\_ptr and tr1::shared\_ptr use delete in their destructors, not delete []. That means that using

auto\_ptr or tr1::shared\_ptr with dynamically allocated arrays is a bad idea.

std::auto\_ptr<std::string> aps(new std::string[10]); // bad idea! the wrong

// delete form will be used

std::tr1::shared\_ptr<int> spi(new int[1024]); // same problem

That’s because vector and string can almost always replace dynamically allocated arrays in c++.

**Things to Remember**

✦ To prevent resource leaks, use RAII objects that acquire resources in their constructors and

release them in their destructors.

✦ Two commonly useful RAII classes are tr1::shared\_ptr and auto\_ptr. tr1::shared\_ptr is usually the

better choice, because its behavior when copied is intuitive. Copying an auto\_ptr sets it to null.

**Item 14: Think carefully about copying behavior in resource-managing classes.**

RAII is the backbone of resource-managing classes. Not all resources are heap-based, however, for such resources, smart pointers like auto\_ptr and tr1::shared\_ptr are generally inappropriate as resource handlers.

For that need to create your own resource managing classes.

For example, suppose we are using a C API to manipulate mutex objects of type Mutex offering functions lock and unlock:

void lock(Mutex \*pm); // lock mutex pointed to by pm

void unlock(Mutex \*pm); // unlock the mutex

To make sure that you never forget to unlock a Mutex you’ve locked, you’d like to create a class to manage locks. The basic structure of such a class is dictated by the RAII principle that resources are acquired during construction and released during destruction:

class Lock {

public:

explicit Lock(Mutex \*pm): mutexPtr(pm){

lock(mutexPtr); // acquire resource

}

~Lock() {

unlock(mutexPtr); // release resource

}

private:

Mutex \*mutexPtr;

};

Clients use Lock in the conventional RAII fashion:

Mutex m; // define the mutex you need to use

...

{ // create block to define critical section

Lock ml(&m); // lock the mutex

... // perform critical section operations

} // automatically unlock mutex at end of block

This is fine, but what should happen if a Lock object is copied?

Lock ml1(&m); // lock m

Lock ml2(ml1); // copy ml1 to ml2 — what should

what should happen when an RAII object is copied?

Most of the time, you’ll want to choose one of the following possibilities:

1. **Prohibit copying:**
2. **Reference-count the underlying resource:**

Sometimes it’s desirable to hold on to a resource until the last object using it has been

Destroyed like a shared\_ptr.

Consider to our example, tr1::shared\_ptr’s default behavior is to delete what it points to

When the reference count goes to zero, and that’s not what we want. When we’re done with

a Mutex, we want to unlock it, not delete it.

shared\_ptr allows specification of a “***deleter***” — a function or function object to be called when the reference count goes to zero. (This functionality does not exist for auto\_ptr, which *always* deletes its pointer.) The deleter is an optional second parameter to the shared\_ptr constructor, so the code would look like this:

class Lock {

public:

explicit Lock(Mutex \*pm) // init shared\_ptr with the Mutex

: mutexPtr(pm, unlock) // to point to and the unlock function

{ // as the deleter

lock(mutexPtr.get()); //Get the raw pinter on Mutex.

}

private:

std::tr1::shared\_ptr<Mutex> mutexPtr; // use shared\_ptr instead of raw pointer

};

In this example, notice how the Lock class no longer declares a destructor. That’s because

there’s no need to(It automatically invokes the destructors of the class’s non-static data

members.). mutexPtr’s destructor will automatically call the tr1::shared\_ptr’s deleter

— unlock, in this case when the mutex’s reference count goes to zero.

1. **Copy the underlying resource:**

Sometimes we need the multiple copies of a resource, and the only reason you need a resource-managing class is to make sure that each copy is released when you’re done with it. In that case, copying the resource-managing object should also copy the resource and copying a resource-managing object performs a “deep copy.”

1. **Transfer ownership of the underlying resource:**

Sometimes it is required that only one RAII object refers to a raw resource and that when the RAII object is copied, ownership of the resource is transferred from the copied object to the copying object (Like auto\_ptr).

**Things to Remember**

✦ Copying an RAII object entails copying the resource it manages, so the copying behavior of the

resource determines the copying behavior of the RAII object.

✦ Common RAII class copying behaviors are disallowing copying and performing reference

counting, but other behaviors are possible.

**Item 15: Provide access to raw resources in resource managing classes.**

std::tr1::shared\_ptr<Investment> pInv(createInvestment()); //Item 13 for Investment class.

Suppose that a function like to use when working with Investment objects is this:

int daysHeld(const Investment \*pi); // return number of days investment has been held

You’d like to call it like this,

int days = daysHeld(pInv); // error!

But the above code won’t compile: daysHeld wants a raw Investment\* pointer, but we are passing an object of type shared\_ptr<Investment>

We need a way to convert an object of the RAII class (shared\_ptr) into the raw resource it contains (Investment\*). There are two general ways to do it:

1. **Explicit conversion**

Both tr1::shared\_ptr and auto\_ptr offers to get member function to perform an explicit conversion, i.e., to return (a copy of) the raw pointer inside the smart pointer object:

int days = daysHeld(pInv.get()); // fine, passes the raw pointer in pInv to daysHeld

Other example consider this RAII class for fonts.

FontHandle\* getFont(); // Params omitted for simplicity

void releaseFont(FontHandle \*fh);

class Font { // RAII class

public:

explicit Font(FontHandle \*fh) : f(fh){} // acquire resource

~Font() { releaseFont(f ); } // release resource

FontHandle\* get() const { return f; } // explicit conversion function

... // handle copying (see Item14)

private:

FontHandle \*f; // the raw font resource

};

void changeFontSize(FontHandle f, int newSize); // Change font method

Font getFont() // Return new Font

Font f(getFont()); // Create new font

int newFontSize;

...

changeFontSize(f.get(), newFontSize); // explicitly convert Font to FontHandle

1. **Implicit conversion**

shared\_ptr and auto\_ptr also overload the pointer dereferencing operators (operator-> and operator\*), and this allows implicit conversion to the underlying raw pointers:

class Investment { // root class for a hierarchy of investment types

public:

bool isTaxFree() const;

...

};

Investment\* createInvestment(); // factory function

std::tr1::shared\_ptr<Investment> pi1(createInvestment()); // have tr1::shared\_ptr manage a

// resource

bool taxable1 = !(pi1->isTaxFree()); // access resource via operator->, Implicit

//conversion shared\_ptr to raw pointer.

...

std::auto\_ptr<Investment> pi2(createInvestment()); // have auto\_ptr manage a resource

bool taxable2 = !((\*pi2).isTaxFree()); // access resource via operator\*

**Things to Remember**

✦ APIs often require access to raw resources, so each RAII class should offer a way to get at the

resource it manages.

✦ Access may be via explicit conversion or implicit conversion. In general, explicit conversion is

safer, but implicit conversion is more convenient for clients.

**Item 16: Use the same form in corresponding uses of new and delete.**

std::string \*stringArray = new std::string[100];

...

delete stringArray;

The program’s behavior is undefined. At least 99 of the 100 string objects pointed to by stringArray

are not properly destroyed, because their destructors will probably never be called.

When we employ a new *expression,* two things happen:

First, memory is allocated (via a function named operator new) and second, one or more constructors are called for that memory.

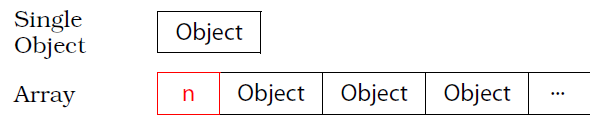
When you employ a delete *expression*, similarly two other things happen:

One or more destructors are called for the memory, then the memory is deallocated (via a function named operator delete).

The big question for delete is this: *how many* objects reside in the memory being deleted?

The answer to that determines how many destructors must be called (pointer being deleted point to a single object or to an array of objects). It’s a critical question, because the memory layout for single objects is generally different from the memory layout for arrays.

The memory for an array usually includes the size of the array, thus making it easy for delete to know how many destructors to call. The memory for a single object lacks this information.



Widget \*w= new Widget();

….

Delete [] w;

The result is undefined. Assuming the layout above, delete would read some memory and interpret what it read as an array size, then start invoking that many destructors which is not in this case at least.

**Things to Remember**

✦ If you use [] in a new expression, you must use [] in the corresponding delete expression. If you

don’t use [] in a new expression, you mustn’t use [] in the corresponding delete expression.

**Item 17: Store newed objects in smart pointers in standalone statements.**

Suppose we have two functions:

int priority(); // Reveal our processing priority.

void processWidget(shared\_ptr<Widget> pw, int priority); // Do some processing on a dynamically

//allocated Widget in accord with a //priority:

processWidget uses a smart pointer (shared\_ptr) for the dynamically allocated Widget it processes.

Now call like:

processWidget(new Widget, priority());

It won’t compile. shared\_ptr’s constructor taking a raw pointer is explicit, so there’s no implicit conversion from the raw pointer returned by the expression “new Widget” to the shared\_ptr required by processWidget.

The following code, however, It will compile:

processWidget(std::tr1::shared\_ptr<Widget>(new Widget), priority());

**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 mistek.

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 behavior 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.

**Item 26: Postpone variable definitions as long as possible.**

(Lazy evolution)

Consider the following function, which returns an encrypted version of a password, provided the password is

long enough. If the password is too short, the function throws an exception of type logic\_error, which is defined in the standard C++ library.

std::string encryptPassword(const std::string& password)

{

using namespace std;

string encrypted;

if (password.length() < MinimumPasswordLength) {

throw logic\_error("Password is too short");

}

... // do whatever is necessary to place an

//encrypted version of password in //encrypted.

return encrypted;

}

*String encrypted completely* unused when an exception is thrown and we are paying cost of construction

and destruction of it. As a result, better postponing encrypted’s definition until we know, need of it.

// this function postpones encrypted’s definition until it’s truly necessary

std::string encryptPassword(const std::string& password)

{

using namespace std;

if (password.length() < MinimumPasswordLength) {

throw logic\_error("Password is too short");

}

string encrypted;

encrypted = password; // assign to encrypted

encrypt(encrypted); // do whatever is necessary to place an

// encrypted version of password in encrypted

return encrypted;

}

Still need improvement. “encrypted” is defined without any initialization arguments. That means its default constructor will be used after that assignment operator is used for initialization. Both collectively more expensive then copy constructor. So, we have better syntax like:

string encrypted(password);

encrypt(encrypted);

Not only should need to postpone a variable’s definition until right before have to use the variable, also should also try to postpone the definition until you have initialization arguments for it. By doing so, we can avoid constructing and destructing unneeded objects.

**Things to Remember**

✦ Postpone variable definitions as long as possible. It increases program clarity and improves

program efficiency.

**Item 27: Minimize casting**

In a C++ casting have some special power then other language. C++ also offers four new cast forms (often called new-style or C++-style casts.

* **const\_cast** is typically used to cast away the constness of objects. It is the only C++-style cast that can do this.
* **dynamic\_cast** is primarily used to perform “safe downcasting,” i.e. to determine whether an object is of a particular type in an inheritance hierarchy. It is also the only cast that may have a significant runtime cost. It is also the only cast that may have a significant runtime cost.
* **reinterpret\_cast** is intended for low-level casts that yield implementation-dependent (i.e., unportable) results, e.g., casting a pointer to an int. Such casts should be rare outside low-level code.
* **static\_cast** can be used to force implicit conversions (e.g., non-const object to const object, int to double, etc.). It can also be used to perform the reverse of many such conversions (e.g.,

void\* pointers to typed pointers, pointer-to-base to pointer-to-derived). It cannot cast from const to non-const objects. (Only const\_cast can do that.)

Why we prefer C++ style of cast:

First, they’re much easier to identify in code. Second, always there is specific cast for that specific needs, none of other cast operator will work. For example, if we try to cast away constness using a new-style cast other than const\_cast, your code won’t compile.

See the below code:

class Widget {

public:

explicit Widget(int size);

...

};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15)); // create Widget from int with function-style cast

doSomeWork(static\_cast<Widget>(15)); // create Widget from int with C++-style cast

In a function cast, object creation doesn’t “feel” like a cast, New C++ style of static\_cast gives better look & feel.

Many time we feel, casts do nothing but tell compilers to treat one type as another, but this is mistaken. Type conversions of any kind (either explicit via casts or implicit by compilers) often lead to code that is executed at runtime.

int x, y;

...

double d = static\_cast<double>(x)/y; // divide x by y but use floating point division.

The cast of the int x to a double almost certainly generates code at run time, because on most architectures, the underlying representation for an int is different from that for a double. Consider another example:

class Base { ... };

class Derived: public Base { ... };

Derived d;

Base \*pb = &d; // implicitly convert Derived\* ⇒ Base\*

Here the base class pointer holds the derived class object. An offset is applied *at runtime* to the Derived\* pointer to get the correct Base\* pointer value. In such example single object (e.g., an object of type Derived) might have more than one address (e.g., its address when pointed to by a Base\* pointer and its address when pointed to by a Derived\* pointer). That can’t happen in C, JAVA, C# etc.

Casting object addresses to char\* pointers and then using pointer arithmetic on them almost always yields undefined behavior.

An interesting thing about casts is that it’s easy to write something that looks right but is wrong. For example, require that virtual member function implementations in derived classes call their base class counterparts first.

class Window { // base class

public:

Window():state(true){}

virtual void onResize() { // base onResize impl

….

state=false;

}

...

Private:

Bool State;

};

class SpecialWindow: public Window { // derived class

public:

virtual void onResize() { // derived onResize impl;

static\_cast<Window>(\*this).onResize(); // cast \*this to Window then call its

//onResize; this doesn’t work!

... // do SpecialWindow-specific stuff

}

...

};

As we expected, the code casts \*this to a Window. The resulting call to onResize therefore invokes Window::onResize. What you might not expect is that it does not invoke that function on the current object!

Still the state is true only.

*Note: SpecialWindow sw; //Not related to above example*

*w = &sw; //Implicit Cast*

*w = static\_cast<Window\*>(&sw); //Explicit cast. Same as above.*

Instead, the cast create a new, temporary *copy* of the base class part of \*this, then invokes onResize on the copy. It doesn’t call Window::onResize on the current object and then perform the SpecialWindow-specific actions on that object — it calls Window::onResize on a *copy of the base class part* of the current object before performing SpecialWindow-specific actions on the current object.

If Window::onResize modifies the current object, the current object won’t be modified. Instead, a *copy* of that object will be modified. If SpecialWindow::onResize modifies the current object, however, the current object *will* be modified, which invalidate the current object state, one where base class modifications have not been made, but derived class ones have been.

So, how we can call a base class version of onResize on the current object? The solution is to eliminate the cast.

class SpecialWindow: public Window {

public:

virtual void onResize() {

Window::onResize(); // call Window::onResize on \*this (current object)

….

}

...

};

Now move into dynamic\_cast: It is very slow. A deeper hierarchy or one using multiple inheritance would be more expensive (at least required class comparison, dynamic linking mechanism).

The need for dynamic\_cast generally arises because we want to perform derived class operations through the base class pointer or reference (through which to manipulate the object).

**Things to Remember**

✦ Avoid casts whenever practical, especially dynamic\_casts in performance- sensitive code. If a

design requires casting, try to develop a cast-free alternative.

✦ When casting is necessary, try to hide it inside a function. Clients can then call the function

instead of putting casts in their own code.

✦ Prefer C++-style casts to old-style casts. They are easier to see, and they are more specific about

what they do.

**Item 28: Avoid returning “handles” to object internals.**

class Point { // class for representing points

public:

Point(int a, int b) :x(a), y(b) {}

void setX(int newVal) { x = newVal; }

void setY(int newVal) { y = newVal; }

private:

int x,y;

};

struct RectData { // Point data for a Rectangle

RectData(const Point &p1, const Point &p2) :

ulhc(p1), lrhc(p2) {}

Point ulhc; // ulhc = “ upper left-hand corner”

Point lrhc; // lrhc = “ lower right-hand corner”

};

class Rectangle {

public:

Rectangle(const Point &p1, const Point &p2) : pData(new RectData(p1,p2 )) { }

Point& upperLeft() const { return pData->ulhc; }

Point& lowerRight() const { return pData->lrhc; }

private:

std::tr1::shared\_ptr<RectData> pData;

};

Now, Rectangle clients want to determine the extent of a Rectangle. The class provides the upperLeft and lowerRight functions which return the reference of user-defined type Point (we know that passing user-defined types by reference is typically more efficient than passing them by value).

Both upperLeft and lowerRight are declared to be const member functions, because they are designed to support read only information for client so. On the other hand, both functions return references to private internal data. Returned references that callers can use to modify that internal data. That a big problem. For example:

Point coord1(0, 0);

Point coord2(100, 100);

const Rectangle rec(coord1, coord2); // rec is a const rectangle from (0, 0) to (100, 100)

rec.upperLeft().setX(50); // now rec goes from (50, 0) to (100, 100)

Rectangle upperLeft() is able to returned reference to one of rec’s internal Point data members. And we can modify it easily. But rec is supposed to be const!

This immediately leads to two lessons.

1. Follow the rule of encapsulation, always declared the data member as a private to class.

In above example, ulhc and lrhc are supposed to be private in Rectangle class, but they’re effectively public and public function upperLeft and lowerRight return references of it.

//This will also not help (My observation).

class Rectangle {

public:

Rectangle(const Point &p1, const Point &p2) :ulhc(new Point(p1)), lrhc(new Point(p2)) {}

Point& upperLeft() const { return \*ulhc; }

Point& lowerRight() const { return \*lrhc; }

Rectangle() {

delete ulhc;delete lrhc;

}

private:

Point \*ulhc;

Point \*lrhc;

};

1. If a const member function returns a reference to data associated with an object that is stored outside the object itself, the caller of the function can modify that data.

References, pointers, and iterators are all *handles* (ways to get at other objects), and returning a handle to an object’s internals always runs the risk of compromising an object’s encapsulation. As we’ve seen, the const member functions that allow an object’s state to be modified.

Both problems we’ve identified for those functions can be eliminated by simply applying const to their return types:

class Rectangle {

public:

...

const Point& upperLeft() const { return pData->ulhc; }

const Point& lowerRight() const { return pData->lrhc; }

...

}

With this altered design, clients can read the Points defining a rectangle, but they can’t write them. Still there are other problems will be occurred when we are returning the internal handle like *“dangling handle”* handles that refer to parts of objects that don’t exist any longer.

**Things to Remember**

✦ Avoid returning handles (references, pointers, or iterators) to object internals. Not returning

handles increase encapsulation, helps const member functions act const, and minimizes the

creation of dangling handles.

**Item 29: Strive for exception-safe code.**

The GUI class is designed to be used in a threaded environment, so it has a mutex for concurrency control:

class PrettyMenu {

public:

...

void changeBackground(std::istream& imgSrc); // change background image

private:

Mutex mutex; // mutex for this object

Image \*bgImage; // current background image

int imageChanges; // No. of times image has been changed

}

Consider this possible implementation of PrettyMenu’s changeBackground function:

void PrettyMenu::changeBackground(std::istream& imgSrc)

{

lock(&mutex); // acquire mutex

delete bgImage; // get rid of old background

++imageChanges; // update image change count

bgImage = new Image(imgSrc); // install new background

unlock(&mutex); // release mutex

}

From the perspective of exception safety, this function is not good at all. There are two requirements for exception safety, and this satisfies neither. When an exception is thrown, exception-safe functions:

■ **Leak no resources**: Suppose “new Image(imgSrc)” expression yields an exception, the call to unlock

never gets executed, and the mutex is held forever.

■ **Don’t allow data structures to become corrupted**: If “new Image(imgSrc)” throws, bgImage is left

pointing to a deleted object. In addition, imageChanges has been incremented, even though it’s not

true that a new image has been installed.

First problem will easy to address using RAII approach:

void PrettyMenu::changeBackground(std::istream& imgSrc)

{

Lock ml(&mutex); // acquire mutex and ensure its later release

delete bgImage;

++imageChanges;

bgImage = new Image(imgSrc);

}

we can turn our attention to the issue of data structure corruption. Exception-safe functions offer one of three guarantees:

1. Functions offering **the basic guarantee** promise that if an exception is thrown, everything in the program remains in a valid state. No objects or data structures become corrupted, and all objects are in an internally consistent state.

For example: Suppose we design changeBackground method such a way, when exception being thrown form it, then the PrettyMenu object might continue to have the old background image, or it might have some default background image.

So, in any scenario system will remains in valid state. However, the exact state of the program may not be predictable (whether new image or default one).

1. Functions offering **the strong guarantee** promise that if an exception is thrown, the state of the program is unchanged. Calls to such functions are *atomic* in the sense that if they

succeed completely, and if they fail, the program state is as if they’d never been called.

Working with functions offering the **strong guarantee** is easier than working with functions offering only the **basic guarantee**, because after calling a function offering the strong guarantee, there are only two possible program states:

* Expected successful execution of the function, or
* In case of exception, the state of function same as the time the function was called.

In contrast, if a call to a function offering only the basic guarantee yields an exception, the program could be in any valid state (Default or new state but both are valid).

1. Functions offering **the nothrow guarantee** promise never to throw exceptions, because they always do what they promise to do. All operations on built-in types (e.g., ints, pointers, etc.) are nothrow (i.e., offer the nothrow guarantee).

This is a critical building block of exception-safe code. It might seem reasonable to assume that functions with an empty exception specification are nothrow, but this isn’t necessarily true. For example, consider this function:

int doSomething() throw(); // note empty exception spec.

This doesn’t say that doSomething will never throw an exception; it says that *if* doSomething throws an exception, it’s a serious error, and the ***unexpected function*** should be called.

In fact, doSomething may not offer any exception guarantee at all. The declaration of a function does not provide the, exception safety guarantee this is determined by its implementation.

Exception-safe code must offer one of the three guarantees above. If it doesn’t, it’s not exception-safe.

Note: *Anything using dynamically allocated memory (e.g., all STL containers) typically throws a bad\_alloc exception if it can’t find enough memory to satisfy a request.*

*changeBackground* functions easily offers **the strong guarantee** Exception-safe code. To do that need to change two things:

First, we change the type of PrettyMenu’s bgImage data member from a built-in Image\* pointer to

one of the smart pointers. Second, we reorder the statements in changeBackground so that we

don’t increment imageChanges until the image has been changed.

Here’s the resulting code:

class PrettyMenu {

...

std::tr1::shared\_ptr<Image> bgImage;

...

};

void PrettyMenu::changeBackground(std::istream& imgSrc)

{

Lock ml(&mutex);

bgImage.reset(new Image(imgSrc)); // replace bgImage’s internal pointer with the

//result of the “new Image” expression.

++imageChanges;

}

Note that there’s no longer a need to manually delete the old image, because that’s handled internally by the smart pointer. Furthermore, the deletion takes place only if the new image is successfully created. shared\_ptr::reset function will be called only if its parameter is successfully created. Delete is used only inside the call to reset, so if the function is never entered, delete is never used.

A common approach to achieve the **strong guarantee** by the strategy named “**copy and swap**.”

In this approach, we make a **copy** of the object in which, want make modification, then make all needed modification to the copy. If any of the modifying operations throws an exception, the original object remains unchanged. After all the changes have been successfully completed, **swap** the modified object with the original.

struct PMImpl { // PMImpl = “PrettyMenu

std::tr1::shared\_ptr<Image> bgImage;

int imageChanges;

};

class PrettyMenu {

...

private:

Mutex mutex;

std::tr1::shared\_ptr<PMImpl> pImpl;

};

void PrettyMenu::changeBackground(std::istream& imgSrc)

{

using std::swap;

Lock ml(&mutex); // acquire the mutex

std::tr1::shared\_ptr<PMImpl> pNew(new PMImpl(\*pImpl)); // copy obj. data

pNew->bgImage.reset(new Image(imgSrc)); // modify the copy

++pNew->imageChanges;

swap(pImpl, pNew); // swap the new data into place

} // release the mutex

The copy-and-swap strategy is an excellent way to make all-or-nothing changes to an object’s state.

**Things to Remember**

✦ Exception-safe functions leak no resources and allow no data structures

to become corrupted, even when exceptions are thrown. Such

functions offer the basic, strong, or nothrow guarantees.

✦ The strong guarantee can often be implemented via copy-and-swap

**Inheritance and Object-Oriented Design**

* Public inheritance means “is-a,”
* A virtual function means “interface must be inherited,” while a non-virtual function means “both interface and implementation must be inherited.

Failing to distinguish between these meanings has caused C++ programmers considerable grief.