**CConstructors, Destructors, and Assignment Operators.**

**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).

* Compiler generated version of copy constructor and the copy assignment operators are simply copy each non-static data member of the source object to the target object.
* If we define any parameterised constructor for a class, the compiler refuse to creates the default constructor.
* Compiler will generate the operator=, only if resulting code is legal with respect to assignment operation. See the below example where C++ compiler refuse to create the default operator= function.

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?

**Things to Remember ????????????????????????**

✦ Compilers may implicitly generate a class’s default constructor, copy

constructor, copy assignment operator, and destructor.

**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(){}

};

Test t1, t2;

t1 = t2; //Compile time error, function inaccessible.

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

Other Ways:

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 behaviour.

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 behaviour. 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:

// invoke base class copy ctor

PriorityCustomer(const PriorityCustomer& rhs): Customer(rhs), 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 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.