**Chaptor-5 Implementation**

**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 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 times 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 no throw guarantees.

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