Automatic resource management as a C++ design pattern

Luis Valente
Aura Conci

Universidade Federal Fluminense – Instituto de Computação
Rua Passo da Pátria, 156 Bloco E Sala 350 - 24.210-240 – Niterói – RJ
{lvalente,aconci}@ic.uff.br

Abstract

Traditionally, developers must manage all resources used in an application directly. As applications grow in complexity, this task may become too complicated and prone to errors, which may raise several kinds of problems. This poster proposes a solution to automate this task, as a C++ design pattern.

Keywords: *Resource management, C++, design patterns*

1 Introduction

Resource management is one of the most common tasks held in any computer program. A resource may be defined as something that exists in the system in a limited supply, and must be acquired from the Operating System before use and must be returned to it after utilization. That definition qualifies examples as memory (the most common resource), sockets, threads and files, among others.

This poster proposes a C++ design pattern, which can be used to minimize errors related to resources and raise software's quality.

2 Design patterns

Design patterns describe solutions to problems which occur over and over in many systems. Those solutions are formulated in such a way so as they can be applied in different systems. A pattern, besides describing the solution to a particular problem, also states the context in which it can be applied and explains the reasons that make its solution successful.

According to [1], a design pattern has four essential elements:

- Name. An identifier which summarizes the problem, the solution and the consequences of applying the pattern in one expression.
- Problem. Identifies the main aspects that distinguish the problem to be solved. The context where the pattern should be applied may be described as well. The context may include preconditions and restrictions (e.g. an interaction between two components of the system must be done using a specific communication protocol). Additionally, the context explains why the problem occurs, when and how it manifests itself.
- Solution. Demonstrates how to solve the problem, or how to balance all the preconditions and restrictions. The solution may be unable to resolve all the prerequisites, specially if they are contradictory.
- Consequences. Describes the results and costs of applying the pattern. The consequences may include the impacts of the solution in the system's stability and flexibility, for instance.

3 Problem

The main problem of resource management is that developers must manually accomplish this task. If resource management is not done properly, many different kinds of problems can manifest themselves, like resource leaks, deadlocks, “random bugs” (an error that is hard to reproduce), invalid memory accesses, exhaustion of system resources, among others.

As a simple example, the following code listing:

```
void foo () {
  int * p = new int;
        p = new int;
}
```

The first block of memory is lost at the second assignment, whereas the second block of memory allocated at the second line is lost when the execution flow exits the function's

scope. In other words, there's no automation in returning the resources to the system.

4 Solution

A key concept to automatic resource management is the definition of ownership of resources. A resource owner is the one which is responsible to returning it to the system.

Every C++ class has two special methods named constructors and destructors. The constructor is called when an instance of the class is created and the destructor is called when an instance is destroyed.

For any class used in a program, the following situations apply:

- An object is declared inside a block of code. When the execution flow enters the block's scope, the object's constructor will be called (the object will be created). When the execution flow exits the block's scope, the object's destructor will get called. The effect is, the object will be destroyed.
- An object A is declared as a member of another object B. So, when object A is created, object B will be created as well. Likewise, object A's destructor will be called by object B's destructor, whenever it is destroyed.
- An object is declared as either a variable or constant with global scope. The object will be created at program startup, and destroyed after program shutdown.

If the concepts of constructors, destructors and scope are combined, it's possible to write classes that own resources, by acquiring resources in constructors and releasing them in destructors. This idiom is known as "resource acquisition is initialization" [2]. Hence, compilers are able to automate resource management. A generic class could be declared as this one (pseudo-code):

```
class AutoResource {
  public:
    AutoResource () { Acquire (); }

    ~AutoResource () { Release (); }

  private:
    resource _r; // real resource
};
```

However, there is another issue left, the ownership transfer. The following example should help to clarify it:

```
AutoResource r1;
AutoResource r2;
r2 = r1;
```

In this case, to ensure that there are no resource leaks, the resource allocated by `r2` must be returned to the system, and the resource previously allocated by `r1` must be transferred to `r2`. In this context, the assignment operation becomes a resource transfer operation. If done like that, the property of ownership of resources is retained. That operation would be implemented like this:

```
class AutoResource {
  public:
    AutoResource () { Acquire (); }

    ~AutoResource () { Release (); }

    void operator = (AutoResource & obj) {
       Release ();
       _r = obj.TransferResource ();
    }

    AutoResource (AutoResource & obj)
     : _r (obj.TransferResource () )
      {}

  private:
    resource  TransferResource () {
      resource  temp = _r;
      _r = null reference;
      return temp;
    }

    resource _r;   //  real resource
};
```

In the former example, "`null reference`" means a neutral value, like 0 would be if the managed resource was memory.

5 Variations

A simple variation can be implemented to achieve more flexibility:

```
class AutoResource {
  public:

   AutoResource (resource rs) : _r (rs) {}

   void Reset (resource rs) {
     Release (); _r = rs;
   }

   ... // same as above

};
```

That example does not break the rules as long as the resource `rs` is supplied directly from an acquisition (like using operator **`new`**).

However, there may be situations where the resource should be shared among many objects. In this case, a kind of reference counter to the resource could be implemented.

The managed resources in this situation would be the original resource and the reference counter itself. Here is an example implementation:

```
class RefCounter {
  public:

    RefCounter ()  : _counter (1) {}
    void Increase () { ++_counter; }
    void Decrease () { --_counter; }
    int  GetCount () { return _counter; }

  private:
    int _counter;
};


class SharedRes {
 public:
  SharedRes (resource rs)
   : _r (rs), _counter (new RefCounter)
   {}

  ~SharedRes () { DecRef (); }

  SharedRes (SharedRes & obj) {
    AddRef (obj);
  }

  void operator = (SharedRes & obj) {
    if (_r != obj._r) {
      DecRef ();
      AddRef (obj);
    }
  }

  void DecRef () {
    _counter->Decrease ();

    if (_counter->GetCount() == 0) {
      delete _counter;
      Release ();

    }

  }

  void AddRef (SharedRes & obj) {
    _counter = obj._counter;
    _counter->Increase();
    _r =  obj._r;
  }


 private:
   resource     _r;
   RefCounter * _counter;
};
```

6 Consequences

Using this pattern ensures automatic resource management, except in the following cases, specific to C++. According to [2], there are two situations where the destructors of local objects (objects declared inside some function, for instance) are not called. Those situations correspond to using the functions `exit()` and `abort()`. In those cases, automatic resource management will fail.

In this work, issues related to multithreaded applications and distributed systems were not regarded.

7 Related works

Authors like [3] and libraries such as [4] and [5] provide implementations for the pattern, related to memory management (described as "smart pointers"). Other authors like [6] and [7] take a more general approach.

8 Examples

A sample program from a project [8] which applies the pattern proposed in this work is available on the Internet at the following url: http://www.ic.uff.br/~lvalente/projetoFinal.zip. The project represents a toolkit for game development implemented in C++, which employs OpenGL and DirectX.

The toolkit uses several kinds of resources, like OpenGL contexts, textures, 3d models, threads and sound effects, among others. Applying automatic resource management in this project rendered it into a robust system, as well as simplified its development and utilization.

The next example depicts loading a texture from disk, using classes implemented in the toolkit. An object which represents a reference counted texture is declared in `main()`. Then, an instance of the `Texture` class is created, which tries to read an image named "image.jpg" from disk. If any error occurs in this process, a more detailed message will be displayed to the user. At the end of the program, the resources allocated by the class will be returned to the system, automatically:

```
int main (int argc, char * argv []) {

 TextureSharedPtr tex;

 try {
   tex.reset (new Texture ("image.jpg") );
 }

 catch (exception & e) {
   cout << "Error loading image.jpg\n"
        << e.what ();
 }

 return 0;
}
```

The `Texture` class' constructor is implemented this way:

```
Texture::Texture (const string &
filename) {
```

```
 LoadImage (filename);
}
```

The `LoadImage()` method reads the image data from disk and transfers them to OpenGL, as is described here:

```
void Texture::LoadImage (const string &
filename) {

  Image image (filename);

  if (image.GetBytesPerPixel () < 3)
    throw FatalErrorException ("images
must have 24 bpp or more");

  UploadImageToOpenGL (image);
}
```

The image stored in file `filename` will have its data read from disk in `Image` class' constructor. The `Texture` class had a limitation in such that it could only handle images with at least 24 bits per pixel, so there is a test at the second line of the function. At last, the image data are transferred to OpenGL.

It's interesting to observe the application of the automatic resource management paradigm in the following situations:

- The file referenced by or `filename` does not exist. In this case, an exception will be raised by `Image`'s constructor. The execution flow will be broken. If there were objects built prior to this event, their destructors would have been called.
- The image was loaded successfully, however, it does not have the required amount of bits per pixel. In this case, an exception will be raised, and the execution flow will exit from the point where the exception was thrown. As the `image` object was successfully built, it will have its destructor called. The net effect is such that all the acquired resources (memory, image data, etc) are automatically returned to the system.
- If the method `UploadImageToOpenGL()` threw an exception (which is not the case), the end result would be analogous to the first scenario.
- The image was successfully loaded and its data were transferred to OpenGL. The method `LoadImage()` ends. The resources acquired by `image` are automatically returned to the system.

9 Conclusions

Improper resource management can lead to many problems of varying degrees of graveness, like high memory use and deadlocks.

Traditional resource management is a developer's responsibility, who should handle this process manually and ensure all resources are returned to the system at the end of the application. As applications grow in complexity, this task may become too complicated and prone to errors.

The automatic resource management paradigm is as powerful as it is simple. As all resource management is performed by specific classes and automated by the compiler, with no intervention from the developer, programming errors related to resources can be minimized. Its employment raises applications' robustness and reliability.

Some examples in this work were demonstrated in the context of games, but the pattern can be applied in many different domains.

10 References

1. Gamma, E. et all. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts, 1995.
2. Stroustrup, B. *C++ Programming Language, 3rd Edition*, Addison-Wesley, Massachusetts, 1997.
3. Alexandrescu, A. *Modern C++ Design – Generic Programming and Design Patterns Applied*, Addison-Wesley, Massachusetts, 2001.
4. SGI. *Standard Template Library Programmer's Guide*, http://www.sgi.com/tech/stl/ (03/08/04).
5. Boost.org. *Boost C++ Libraries*, http://www.boost.org (03/08/04).
6. Milewsky, B. *C++ in Action: Industrial Strength Programming Techniques, Web Edition*, http://www.relisoft.com/book (02/08/04).
7. Lippman, S. *Essential C++*, Addison-Wesley, Massachusetts, 1999.
8. Valente, L. *Um modelo orientado a objetos para jogos em computador*, Monografia de conclusão de curso, Universidade Federal Fluminense, Niterói, 2002.