Tutorial – Map and Multimap

Introduction:

In this tutorial we will revisit the *ResourceManager* class we created during the tutorial for the session on Resource Management.

We’ll replace the *Resource* class and array v*ector* with a *map*, to simplify and reduce the code we write.

If you have not yet completed the tutorial for the Resource Management session, you will need to do that before attempting this tutorial. Only the code that needs to be updated will be shown.

Design Goals:

If we recall back to the original tutorial, you may remember our design goals being as follows:

* Allow us to specify the file path of the resource we want to load
* Load the resource if it has not already been loaded
* Return a reference to a loaded resource if the resource has already been loaded
* Automatically track how many references to the resource exist
* Automatically track when a resource is not being used
* Allow us to unload all unused resources

These goals haven’t changed.

What has changed is our use of an associative container in implementing the *ResourceManager* class.

The *map* class will allow us to associate some data (i.e., a loaded resource, like a sound or a texture) with a *key* (i.e., the resource’s file path, stored as a string). This will eliminate the need for the *Resource* class and will simplify both how we add resources to the manager, and how the manager searches for loaded resources.

The ResourceManager Class:

Before we modify the *ResourceManager* class, go ahead and delete the *Resource* class.

We’ll replace the *m\_resources* vector with a *map*. The *key* for the map will be the resources’ file path (i.e., a string), and the *mapped value* will be the resource itself, stored as a *shared\_ptr*.

As with the old *ResourceManager* class, we’ll keep it templated. This means we’ll have one instance of the *ResourceManager* for each type of resource we use in our game.

Update the *ResourceManager* class as follows:

template< class T >

class ResourceManager

{

~~std::vector< std::shared\_ptr<Resource<T>> > m\_resources;~~

std::map<std::string, std::shared\_ptr<T>> m\_map;

// keep the copy constructor and assignment allocator private

ResourceManager(const ResourceManager&) {};

ResourceManager& operator=(const ResourceManager&) {};

public:

ResourceManager() {};

~ResourceManager() {};

std::shared\_ptr<Resource<T>> get(const std::string filename) { };

int getCount();

void collectGarbage() {};

};

We can now go ahead and update the *get()* and *collectGarbage()* functions (the modification to *getCount()* are trivial, and you should be able to update that function without guidance).

**get()**

The *get()* function actually becomes easier to write when using a map.

We can use the map’s *find()* function to determine if the *key* (the filename) is in our map. If it is, that means the resource has been loaded and we can simply return the *shared\_ptr* containing the resource.

If our resource hasn’t been loaded, we load the resource (in a similar way to the previous version of our class) and insert it into out map using the filename as the *key*.

The new *get()* function is as follows:

std::shared\_ptr<T> get(const std::string filename)

{

std::map<std::string, std::shared\_ptr<T>>::iterator it = m\_map.find(filename);

if (it == m\_map.end())

{

std::shared\_ptr<T> resource(new T(filename.c\_str()));

it = m\_map.insert(it, std::pair< std::string,

std::shared\_ptr<T> >(filename, resource));

}

return (\*it).second;

}

When searching a map using *find()*, an interator containing the key-value pair is returned. However, if the key was not found, then the iterator equates to *map.end()*. This gives us an easy way to determine whether or not the resource has been loaded.

Notice now that when we create the new resource object (by calling *new* on the template type) that we no longer need an instance of the *Resource* class.

The original *Resource* class was present only so that we could associate a filename with a loaded resource. This association is now performed by the *map* itself, so we’ve eliminated the need for one class.

As mentioned in the lecture slides, the map iterators point to instances of the *std::pair* class, which contain the key-value pairs. To access the *key*, you use the *first* variable (i.e., *(\*it).first*), and to access the *value* we use the *second* variable (i.e., *(\*it).second*).

**collectGarbage()**

Finally we have the *collectGarbage()* function. This is very similar to the *vector* version, except there are different rules surrounding iterators in map classes.

When we erase a value in a *map*, only the iterators and pointers for that value become invalidated (contrast this with a *vector*, where all iterators and pointers to elements after and including the element deleted become invalid).

This means we don’t get a new iterator returned by the *erase()* function.

To ensure out iterator is updated to the next element in the *map* after the erase, we need to post-increment the iterator when we pass it to the *erase()* function (as seen in the code below).

Update the *collectGarbage()* function as follows:

void collectGarbage()

{

for (std::map<std::string, std::shared\_ptr<T>>::iterator it = m\_map.begin(); it != m\_map.end(); )

{

if (it->second.use\_count() == 1)

m\_map.erase(it++);

else

++it;

}

}

Pay special attention to how we increment the iterator when erasing a value.

Variadic Templates:

So far our templated *ResourceManager* class works great… as long as we only want to load resources that can be loaded using only their filename.

For sounds and textures we have no problem. But as soon as we want to create a *ResourceManager* to handle fonts, we run into a problem. To create a font we need to give the constructor both a filename *and* a font size, yet the creation code in the *ResourceManager*’s *get()* function assumes just the one argument (the filename) when it calls the resource’s constructor.

We need a way to pass the font size as an argument to the *get()* function when we have an instance of the *ResourceManager* that handles the *Font* type, and yet not affect instances of the class handling other types.

The solution is to use variadic templates – an advanced feature of C++ that lets us have a variable number of template types. Combine this with a variable argument list for the *get()* function, and we have a way to handle our special-case font type while keeping the generic flexibility of the *ResourceManager* class.

When declaring the list of templated types for the class, we specify a variable number of types using an ellipsis, as follows:

template< class T, class... Targs>

class ResourceManager

{

This definition indicates that we must have one type, called T, followed by a variable list of types (we could have 0, 1, or more other types in this list).

We can then update the *get()* function as follows:

std::shared\_ptr<T> get(const std::string filename, Targs ... args)

{

std::map<std::string, std::shared\_ptr<T>>::iterator it = m\_map.find(filename);

if (it == m\_map.end())

{

std::shared\_ptr<T> resource(new T(filename.c\_str(), args...));

it = m\_map.insert(it, std::pair< std::string, std::shared\_ptr<T> >(filename, resource));

}

return (\*it).second;

}

Our *get()* function now accepts a string (the filename), followed by 0, 1 or more arguments. We define the type and order of these arguments when creating an instance of the *ResourceManager* class.

The *get()* function then passes this variable list of arguments to the constructor of the resource. As long as the type and order of these arguments matches, everything will compile and run without errors.

We would now create instances of our updated *ResourceManager* class as follows:

ResourceManager<aie::Texture> m\_images;

ResourceManager<aie::Font, int> m\_fonts;

m\_std::shared\_ptr<aie::Font> font = m\_fonts.get("./font/consolas.ttf", 32);

m\_std::shared\_ptr<aie::Texture> grass = m\_images.get("./textures/grass.png");

Where To Now?:

Variadic templates are a novel solution, and depending on the needs of your game this may be a perfectly adequate solution.

But are there still problems? Well, yes. What if we wanted a version of our *consolas* font at 12px and 32px. There’s no scope in our *ResorurceManager* for that.

We’ve also (arguably) decreased the understandability of our code. We’re assuming that any user of our *ResourceManager* class (including us) sufficiently understands not only templates, but variadic templates, so as not to get themselves into trouble when using our class.

Is it possible to come up with an alternate design for our *ResourceManager*? Absolutely. Could this new design be more flexible and functional? Possibly. Would it be better suited to the game I’m making? Without a doubt. What does it look like? That’s entirely up to you.

There are many possible design choices you could make when creating a resource manager class. But hopefully, you’ve seen how an associative container class like *map* can be used to efficiently store and search the resources this class is managing.