1. (a) Explain what C++ header files are, and why they are needed in building C++ software, such as game engines. Define what a translation unit is, and explain how C++ supports separate compilation. Describe the relevant C++ language features and keywords needed. Then give a short C++ example program that uses header files and multiple translation units, and explain it.

Answer:

Header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. (Tutorialspoint) They are needed in building C++ Software as we can include the header file in our source file if we require its function prototypes, instead of copying the content of the header file directly to our source files which are error prone.

The source file together with all the headers and source files included via preprocessing directive #include is known as preprocessing translation unit. After preprocessing, they are then called a translation unit. (C99 Documentation) Separate compilation is a process where pieces of the program are compiled independently through the two stage approach of compilation and then linking. (Derek Molloy) As C++ program need not be translated at the same time, they can be compiled individually and the resulting compiled code (.o or .obj files) are combined through the use of the linker.

//main.cpp

```
#include <iostream>

#include “foo.h”

#include “foo2.h”

int main(void) {

foo();

}
```

//foo.h

```
void foo()

{

   std::cout << "Hello, world! From foo";}
```

//foo2.h

```
void foo2()

{

   std::cout << "Hello, world! From foo2";

}
```

I have three files named main.cpp and foo.h, foo2.h, where foo.h and foo2.h each has one function called foo() and foo2(), my main function would be able to invoke the two functions foo() and foo2() by just including the two header files.

Reference(s):

1) Tutorialspoint,https://www.tutorialspoint.com/cprogramming/c_header_files.htm#:~:text=A%20header%20file%20is%20a,that%20comes%20with%20your%20compiler.
2) Derek Molloy, http://www.eeng.dcu.ie/~ee553/ee402notes/html/ch03s14.html#ftn.d4e3056
3) C99 Documentation http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

(b) Give practical advice (style rules) how to write header files; i.e., what should be placed (written) into header files and what should not. Give explanations and justifications for your style rules.

Answer:

In a medium to large project, the header files should always contain the source code documentation, such as the purposes of the various parameters, the uses of the functions, the return values of the functions so that we are able to organise our projects easily.

We should also always use header guards so that the header file would not be included twice, since processing the contents twice will result in an error. So we have to enclose the entire contents of the file in a conditional with #ifndef HEADER_FILE #define HEADER_FILE … #endif

The header file should contain only declarations, templates, and inline function definitions, and is included by the .cpp file for the module.(C++ header file guidelines) For example, we should put the structure and class declarations, function prototypes, and global variable extern declarations, in the .h file and put the function definitions and global variable definitions and initializations in the corresponding .cpp file. The .cpp file for the module must include the .h file so that the compiler will detect any discrepancies between the two, and thus help ensure consistency.

We should use #include moderately in the header files, i.e. we should not #include a file in a header file if the header file doesn't need the other file. This is to prevent recompilation of things that don't need to be recompiled and might cause errors.

As said above, the header files should be well documented and thus the header files itself should be self-explanatory, which means that the purposes/uses of the header files should be well defined. Hence we should not include lots of structure definitions in one header file, we need to split those up into separate headers.

Reference(s):

1) C++ header file guidelines. http://umich.edu/~eecs381/handouts/CppHeaderFileGuidelines.pdf

2. Describe the different alternative ways to create and properly initialize modules in a layered game engine architecture. Compare them to each other, and choose one of them to implement. Sketch out a C++ implementation, and explain it. Defend and justify your own choice: analyze its benefits and (potential) drawbacks. (15 p.)

Answer: In a layered game engine architecture, we need the interacting subsystems to be configured and initialized in a specific order as the interdependencies between the subsystems implicitly define the order in which they must be started to have a proper working game engine. The first way (albeit incorrect) of doing is to have a singleton class for each subsystem, in which the global and static class instances were constructed and destroyed in the constructor.

From lecture notes, we have

```
class GRenderEngine { // game renderer (or whatever)
 public:
        GRenderEngine () { /* start up the render manager */ }
        ~ GRenderEngine () { /* shut down the manager */ }
 };
  GRenderEngine Renderer;   //  A global instance of the class (singleton design pattern)
```

However, we can notice that the constructors and destructors of global objects in different compilation units are executed in unpredictable order, so this approach would not work for interdependent subsystems.

We can leverage the approach above by having the static singleton variable to be declared within a function so that we would be able to control the order of construction of the global singletons. However, this still does not give us ways to control the destruction orders, and it is still difficult to predict exactly when the singleton will be constructed.

There are alternatives such as having each singleton managers to register itself into a global priority queue and then walk this queue to start up all the managers in the proper order, define the manager-to-manager dependency graph by having each manager explicitly list the other managers upon which it depends and then write some code to calculate the optimal start-up order. (Game Engine Architecture) However, I would still stick to the singleton managers for our subsystems and use a brute-force approach to explicitly define the start-up and shut-down functions for each singleton manager class. And we would mak it so that the constructor and destructors of each singleton manager class do nothing. This way the start-up and shutdown functions can be explicitly called in the required order for our subsystem to work.

For example, we can have

```
Class RManager
{
Public:
        RManager()
        {// do nothing
```

```
        }
        ~RManager()
        {// do nothing
        }
        void startUp(){ // start up the manager}
        void shutdown() {// shut down the manager}
}
// Then we can have other classes with the similar structures as above.
class MeshManager {// similar as above}
class AudioManager {// similar as above}
…
```

and we can initialise an instance of each class

```
RManager gRManager;
MeshManager gMManager;
AudioManager gAManager;
…
```

And in our main we can start them up/shut them down in the orders required for their interdependencies for example,

```
Int main (){
        gRManager.startUp();
        gAManager.startUp();
        gMManager.startUp();
        …..
        gMManager.shutDown();
        gAManager.shutDown();
        gRManager.shutDown();
}
```

This is a simple approach that works, an obvious benefit is that it is easy to implement. Secondly, it is explicit and the codes are self-explanatory, so we can see and understand the start-up/shut-down order immediately from the codes. Thirdly, it is easy to debug and maintain since we can easily change the orders of the subsystems to see whether which is not working as intended. The potential drawback of this is that we might accidentally shut things down in an order that isn't strictly the reverse of the start-up order if we were to be careless.

Reference(s):

1) **Lecture Notes**
2) **Game engine architectures page 418-422**

3. (a) Describe and explain how to create and destroy "objects" in C++, in all possible different ways. In each case, explain how the lifetime ("life cycle") of the object is determined. Provide short illustrative examples, and explain them.

Answer:

A constructor is a method that is automatically called when an object of a class is created, destructor on the other hand is a method that is automatically invoked when the object is destroyed. Their purpose in c++ is to initialize the class object and destroy the class object.

After an object is created, the memory is allocated and then the constructor runs. When the constructor runs to completion, the lifetime or the scope of the object begins, the object is also initialized with some default values.If the object goes out of scope or the lifetime of the object ends, the destructor function is called, in which the memory will be freed.

For example,

Global variable c++ can be defined by the following, then it will be accessible to all the files that include the header.

In the Global_obj.cpp

Class obj;

In the header file Global_obj.h

Extern Class obj;

In Main.cpp

#include “global_obj.h”

You can create a local obj in the function call, and the life cycle of the object would end after the function call is finished.

Foo (){

Class obj;

}

You can dynamically allocate and deallocate the objects this way

```
using namespace std;

int main()

{

Class* a = new Class[4];

delete [] a;

return 0;

}
```

By having a pointer, we just need to delete the array of objects a then we can allocate and deallocate the objects dynamically. Note that the dynamic objects can be created with the new operator. For a single object we can use MyClass *obj = new MyClass;, For an array of object we can use MyClass *oArray = new MyClass [50]{}; And to destroy them we can use delete [] oArray and delete obj for the object and array.

Reference(s):

1) Lecture notes

(b) How can we assure that such C++ objects really are appropriately destroyed, i.e., how to prevent memory leaks? What are the practical problems and dangers in C++ for achieving adequate memory management? Describe the relevant language features and run-time support. Finally, give your advice for proper programming techniques and idioms for successful memory management. (15 p.)

Answer:

We can assure that such objects are appropriately destroyed by ensuring that any dynamically allocated memory within the class is destroyed by calling delete in the destructor of the class. After deallocating the object, you can also assign null to the pointer so that the pointer does not point to the same memory location.

Another technique is to use a smart pointer in which we can define the pointers in the memory header file, they ensure proper destruction of dynamically allocated objects and are automatically deleted.

The practical problems and dangers in C++for achieving adequate memory management are that it is very hard to manage a program with lots of memory managements (International Organization for Standardization C++ Documentation). We are inevitably going to mess up somewhere and get leaks, stray pointers etc, no matter how conscientious we are with our allocations, since eventually the complexity of the code will overcome the performance of the system/ the time and efforts we are going to put for maintaining the code base. Hence, we should have ideally as few new/delete calls at the program level as possible. Furthermore, the problems arise when C++ programmers are not careful with the usage of new keyword and delete/delete[] operator. The delete operator should be used to free a single allocated memory pace, whereas the delete[] operator should be used to free an array of data values.

For advices, we should reduce the use of c style raw pointer and employ the use of smart pointers if possible since they will ensure that the memories are freed after the pointer goes out of scope. We should write the new keyword and delete keyword first and write all codes between them to ensure that we don't have memory leaks by forgetting to deallocate the memory. Generally, we would use references over pointer but if we want to decouple compilation units to improve compilation time we can use C pointer, the useful property of a C pointer is that we only require a forward declaration of the pointed-to type, which will improve our compilation time. If we want to use pointers dynamically, we can make use of the new operator in C++ in which it will assign an address to the pointer itself, however we should remember to deallocate the memory on our own during the program execution.

Reference(s):

1) My own answers from exercises
2) https://www.geeksforgeeks.org/memory-leak-in-c-and-how-to-avoid-it/

3) Lecture notes
4) ISO C++ Documentation https://isocpp.org/wiki/faq/freestore-mgmt#memory-leaks

4. Explain the motivation of the "Model-View-Controller" (MVC) architecture for a game program. Explain the problems it can solve and the (potential) achieved benefits. How should this architecture be applied for implementing video games? What are the different parts: model, controller, and views in this case? When implementing such a MVC architecture, what additional other design patterns are necessarily required to make the MVC architecture work? Explain why they are needed. Describe these alternative design patterns and how they work.

Answer:

Using versions of the MVC is simple and helps us to decouple the classes with each other (lecture notes), by having one class that maintains a list of pointers to instances of some interface (the observers in this case), which would help in observing the list of observers and sending the notifications to them once some triggering conditions have been met.

The most obvious use of MVC is to build user interfaces of the game (Lecture notes), the MVC can decouple the user interface designs to increase the flexibility and facilitate reuse. So the model in this case would be the application object, the view would be its screen presentation and the controller would b the way the user interface reacts to user input to manage the view and the model.

Likewise, we can apply this architecture to the game achievements system in the game, in this case, the model would be the classes that represent the game data, the views would be the current interface/game state rendered, and controllers would be the user's input or triggering conditions. The game would wait for the user's input or other triggering condition, in which that will invoke the notification of those events to be sent and updated the model.

When implementing such MVC architecture, we can consider additional design patterns such as using an asynchronous communication method using Event Queue (Game Programming Pattern), since synchronous communication might lock up the games for too long and could potentially deadlock the game, especially in a highly threaded game. And if we are not careful mixing observers with threading and explicit locks, the observers might try to grab a lock that the subject has, which would deadlock the game.

Another design pattern in addition of MVC is that we can have an object pool that we reallocate to avoid dynamic allocation of the observers, so that we can have a fixed-size pile of list nodes of observers to use and reuse as we need without having to deal with manually allocating/deallocating memory.

Reference(s):

1) **Game Programming Pattern Observer Section**
2) **Lecture notes**