Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #1

Manager & LogManager

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 6.0). Information online at: `http://dragonfly.wpi.edu/book/`

4.2 Managers

Managers are the support systems for game engines, handling crucial tasks. This includes handling input, rendering graphics, logging data, managing the game world and game objects and more. Logically, the different functions can be broken up into different managers. The main class, Manager, is not instantiated. Instead, it serves as a base class for all derived game engine managers. Refer to Table 4.1 on page 50 for details.

The interface for the Dragonfly Manager class is shown in Listing 4.1. The Manager class provides `startUp()` and `shutDown()` methods, allowing the game programmer to control initialization and termination of all derived manager objects. For the base Manager, `startUp()` sets `is_started` to `true` and `shutDown()` sets `is_started` to `false`. The method `isStarted()` allows for a query to check if the manager has been successfully started (via `startUp()`). As this is the base class, in Manager, the methods for starting up and shutting do not do any "real" work – instead just manipulating the `is_started` boolean variable. The method `setType()` sets the private attribute `type` to the name `"Manager"`. The method is `protected` since only the base class and derived classes are allowed to change a manager's type (typically, this is done in the constructor for each derived Manager).

Listing 4.1: Manager.h

```
0  namespace df {
1
2  class Manager {
3
4   private:
5    std::string m_type;        // Manager type identifier.
6    bool m_is_started;         // True when started successfully.
7
8   protected:
9    // Set type identifier of Manager.
10   void setType(std::string type);
11
12  public:
13   Manager();
14   virtual ~Manager();
15
16   // Get type identifier of Manager.
17   std::string getType() const;
18
19   // Startup Manager.
20   // Return 0 if ok, else negative number.
21   virtual int startUp();
22
23   // Shutdown Manager.
24   virtual void shutDown();
25
26   // Return true when startUp() was executed ok, else false.
27   bool isStarted() const;
28 };
29
30 } // end of namespace df
```

Line 0 of Listing 4.1 defines the Dragonfly namespace using the `df::` tag. This requires code outside of the namespace (e.g., game code) to use `df::` to access elements inside the namespace (e.g., `setSolidness(df::SPECTRAL)`). Typically, large, 3rd-party libraries (such as a game engine) use namespaces to help developers avoid conflicts in names their own code may use with names the libraries use. The Dragonfly namespace is meant to prevent potential name conflicts with game code.

Note, for brevity in all future Listings in this book (with the exception of `Logfile.h`), the Dragonfly namespace `df::` is not shown, but it does appear in the actual engine `.h` files.

Many managers depend upon each other, so the startup order of individual managers matters. For example, a logfile manager is often needed first since all the other managers write log messages upon startup, either noting successful startup in the logfile or reporting errors in the logfile when there are problems. Or, a display manager may need memory allocated for sprites, so a memory manager would need to be invoked before the display managers.

In addition, unlike many game objects, it often makes sense to have only one instance of each manager. For example, having two display managers simultaneously writing to the graphics card/display device may not make sense nor even be supported by the hardware/operating system. Similarly, having two independent managers handle input from the keyboard and mouse may yield undesirable (or at least unpredictable) results.

Managers are generally global in scope because the service the manager provides may be sought in many places in both game and engine code. For example, the engine code and game code may both write messages to the logfile via the LogManager, and both the engine code and game code may draw characters on the screen via the DisplayManager. Given this, a natural inclination may be to make the manager instances global variables, as depicted in Listing 4.2.

Listing 4.2: Managers declared as global variables

```
0 // Outside main(), these are global variables.
1 df::DisplayManager display_manager;
2 df::LogManager log_manager;
3
4 main() {
5   ...
6 }
```

While declaring managers as global variables does provide for global scope, it does not give control over the order of invocation. The order of instantiation of global variables is determined by the compiler and not by the program order. For example, if the code in Listing 4.2 is compiled and run, the `log_manager` may be instantiated first and then the `display_manager` or vice versa. Moreover, using global variables from the engine – say, if DisplayManager wanted to write to the logfile – would *require* the game programmer to use the same names as expected by the engine, making the game code more brittle.

4.2.1 Singletons

The *singleton* design pattern can be used for the game engine manager to solve all the above problems: 1) the singleton restricts instantiation of a class to one, and only one, object; 2) the singleton allows control of the order of manager initialization for dependency cases where the order matters; and 3) the singleton allows for global access.

In order to restrict instantiation to one (and only one) instance, the singleton class needs to disallow typical operations that enable object creation from a class. In particular, access must be denied for public access to the constructor, copy and assignment operators – otherwise, a programmer can use them to make additional instances of the class. Restricting creation is done by making the specific operations `private` to the class.

In order to instantiate a singleton class in C++, the keyword `static` is used as a modifier to the variable representing the one instance of the class. Remember, `static` variables retain their value even after the function terminates – in effect, the lifetime extends across the entire run of the program. However, a `static` variable is not allocated until the function is first called. This last feature allows explicit control as to when the manager is started up. Also remember, the keyword `static` in front of a method or function is quite different. A `static` method does not require an instance of the class to use it, and a `static` function (or a `static` global variable) has a scope that restricted to the `.cpp` file it is declared in.

Listing 4.3 depicts the class template for a singleton class.

Listing 4.3: A Singleton class

```
0  class Singleton {
1   private:
2     Singleton();                          // No constructing.
3     Singleton(Singleton const &copy);     // No copying.
4     Singleton&(Singleton const &assign); // No assigning.
5   public:
6     static Singleton &getInstance();      // Return instance.
7  };
8
9  // Return the one and only instance of the class.
10 Singleton &Singleton::getInstance() {
11   // Note, a static variable persists after method ends.
12   static Singleton single;
13   return single;
14 }
```

While the singleton class guarantees there will be one and only one instance of the class, when a manager is actually instantiated (the first time `getInstance()` is called for that class), there can sometimes be a lot of work to be done. Thus, most of the initialization work for any manager is done in the `startUp()` method, called after the first `getInstance()` call.

Each specific manager class (e.g., DisplayManager) inherits from the base Manager class using the singleton template. The virtual methods `startUp()` and `shutDown()` are defined in the derived class and are specific to that particular manager. For example, the logfile manager might open the logfile for writing, while the display manager might ready the display for graphical output.

4.3 Logfile Management

If a game is working well, meaning all game engine code and game programmer code is doing what it is supposed to, all meaningful output typically goes to the screen in the form of game actions – characters moving, bullets flying, menus popping up, etc. However, during development this is often not the case, as code (even game engine code) can have bugs, or confirmation of working code is needed before proceeding. While debuggers are essential for effective programming, many game programmers do not have the luxury of having the source code for the game engine so a debugger cannot trace through the engine code. Moreover, some bugs are timing dependent meaning they only happen at full speed or are caused by a long sequence of events, making them hard to trace by a debugger.

What is helpful in these cases is a game engine that provides meaningful output as to the workings (or not) of the engine, and also provides a flexible, easy-to-use mechanism for a game programmer to get output. However, standard methods of printing to the screen can often interfere with the game itself or are not even possible when a display device is in graphics mode. In order to get around this limitation, logfiles are often used, where descriptive messages from the engine are written to a file, and the engine provides a flexible, easy-to-use mechanism for their own messages. This is the essence of the LogManager, often the first manager developed since all other engine components make use of it.

For base functionality, upon startup the LogManager opens up the logfile, making sure writing is allowed to the appropriate directory. Advanced features could allow appending or overwriting of previous logfiles, name the logfile with a timestamp, and check if there is sufficient disk space for normal operations. Upon shutdown, the logfile is closed, effectively flushing any outstanding data to the disk.

For attributes, the LogManager only needs a file structure (e.g., `FILE *`) for access to the logfile.

4.3.1 Variable Number of Arguments

The most frequently used LogManager method is to support writing general-purpose messages to the logfile – whether the messages come from other parts of the engine or from the game programmer – via a `writeLog()` method. For example, the game programmer may want to write a string such as "Player is moving", which is effectively one argument to `writeLog()`. Or, the game programmer may want to write "Player is moving to (x, y)" where x and y are `float` variables that are passed in. In other words, the number of arguments that `writeLog()` supports is not known ahead of time, but can be one or more.

A function that supports a variable number of arguments is depicted in Listing 4.4. Note the "..." characters in the parameter list for the function. Handling a variable number of arguments in this way requires `#including` the system header file `stdarg.h`, and the system header file `stdio.h` is needed for `fprintf()`. In the body of the function, a `va_list` structure is created, then initialized with arguments in the `va_start` command, provided with the name of the last known argument (`fmt`, in this case). At this point, the function is ready to call a `printf()` to produce output, but instead of a fixed string, the `va_list` structure has the formatting parameters, so `vfprintf()` is used instead. When finished, the `va_end()` must be called to clean up the stack.

Listing 4.4: Function taking variable number of arguments

```
#include <stdio.h>
#include <stdarg.h>

void writeMessage(const char *fmt, ...) {
  fprintf(stderr, "Message: ");
  va_list args;
  va_start(args, fmt);
  vfprintf(stderr, fmt, args);
  va_end(args);
}
```

Note, Listing 4.4 uses standard error (`stderr`), which typically defaults to the console Window, while a game engine (e.g., Dragonfly) will usually write to a file. The code can be adjusted, accordingly, to use a file.

4.3.2 Human-friendly Time Strings (optional)

For a long running game, it is often helpful to have timestamps associated with messages in the logfile. These times can be in "game time", such as game loop iterations, or in "wall-clock time" corresponding to the actual time of the day. Dragonfly does both, displaying a human-friendly time and game loop iteration in front of each message.

An easy way to associate a time with a written message is to have a function, say `getTimeString()`, that `writeLog()` calls to get a string with a timestamp. The `getTimeString()` method uses the `time()` system call, which returns the number of seconds since January 1, 1970. In order to turn that big number into something that is easier for humans to read, the `localtime()` system call converts the seconds into calendar time, allowing extraction of hours, minutes and seconds. Listing 4.5 depicts the `getTimeString()` method. Note, no error checking is provided for the system calls.[1] The function `sprintf()` on line 12 is similar to `printf()`, but instead of printing to `stdout`, `sprintf()` prints to a string,[2] in this case `time_str`.

Listing 4.5: Function to provide human-readable time string

```
// Return a nicely-formatted time string: HH:MM:SS
char *getTimeString() {

  // String to return, made 'static' so persists.
  static char time_str[30];

  // System calls to get time.
  time_t now;
  time(&now);
  struct tm *p_time = localtime(&now);

  // '02' gives two digits, '%d' for integers.
  sprintf(time_str, "%02d:%02d:%02d",
  p_time -> tm_hour,
  p_time -> tm_min,
```

[1] *All* system calls should be error-checked, and errors handled appropriately, in case they fail.
[2] The 's' in front of `printf()` is for 'string.'

```
15    p_time -> tm_sec);
16
17    return time_str;
18  }
```

A complementary message in the logfile is the "game clock" – the number of iterations of the game loop – obtained from the GameManager. This can be displayed as an integer pre-pended to the log message. See the GameManager in Section 4.4.4 on page 4.4.4 for details.

The presence of both the time string and the game clock in the logfile can be setup to be controlled by the game programmer by having the LogManager keep two boolean variables, `log_time_string` and `log_step_count`, which, if `true`, pre-pend the time string or game clock, respectively, to the game programmer's log message.

While the printing of time has been presented in the context of the logfile, functions like `getTimeString()` are useful beyond the LogManager class and do not access any attributes of the class. As such, they should be placed in a file called `utility.cpp` (with a corresponding `utility.h`). Other functions that provide utility services, but are not part of any class definitions, will reside in `utility.cpp` as they are created.

4.3.3 Flushing Output

Generally, writing data to a file does not immediately write the data out to the disk. The operating system typically buffers data, writing when the device is idle or when internal memory buffers are filled. Such buffering generally improves overall system performance, but can cause unexpected output (or, more precisely, lack of it) if a program, say a game engine, crashes before all data is written. For example, if the line `log_manager.writeLog("Doing stuff")` is executed and then the program crashes (e.g., from a segfault), the string "Doing stuff" may not appear in the logfile even though that line has been executed. This can make it hard to trace where, exactly, the error (in this case, the error that caused the segfault) might have occurred.

To force the operating system to immediately write out buffered data to the disk, the `fflush()` system call can be used after each write. Used during development, this helps provide complete logfiles even during system crashes. Note, this does decrease efficiency (speed) somewhat, so might not be used when game development (or game engine development) is complete. Thus, the LogManager can provide the game programmer with an option to flush the logfile after each disk, or not.

The attributes and methods for the LogManager can now be described in Listing 4.6. The destructor closes the file if `p_f` is not `NULL`.

Listing 4.6: LogManager attributes and methods

```
0  private:
1    bool do_flush;  // True if fflush after each write.
2    FILE *p_f;      // Pointer to logfile structure.
3
4  public:
5    // If logfile is open, close it.
6    ~LogManager();
7
```

```
8   // Start up the LogManager (open logfile "dragonfly.log").
9   int startUp();
10
11  // Shut down the LogManager (close logfile).
12  void shutDown();
13
14  // Set flush of logfile after each write.
15  void setFlush(bool do_flush=true);
16
17  // Write to logfile. Supports printf() formatting.
18  // Return number of bytes written, -1 if error.
19  int writeLog(const char *fmt, ...);
```

4.3.4 Conditional Compilation

Once implemented, using the newly-minted LogManager throughout the engine (or in game code) will quickly reveal a problem – the LogManager header file, `LogManager.h`, likely gets included by the compiler pre-processor multiple times, resulting in compiler warnings about "redeclaration of class LogManager". In order to fix this, directives to the pre-processor can limit class (and other) definitions to be included only once by having code only compiled during certain conditions. For the LogManager (and other Dragonfly header files), this is done by using an `#ifndef` wrapper and a unique identifier. Consider the sample code in Listing 4.7. When `foo.h` is seen by the compiler the first time, `FILE_FOO_SEEN` is not defined, so the pre-processor defines it in the next line and proceeds to parse and processes the foo file (and defining `class Foo`), normally. The next time `foo.h` is seen by the pre-processor, `FILE_FOO_SEEN` is already defined so the contents of the foo file are not included, avoiding a duplicate definition of `class Foo`.

Listing 4.7: Once-only header files

```
0  // File foo.h
1  #ifndef FILE_FOO_SEEN
2  #define FILE_FOO_SEEN
3
4  // The entire foo file appears next.
5  class Foo {};
6
7  #endif // !FILE_FOO_SEEN
```

Such conditional compilation directives are often used for platform-specific parts of code. Listing 4.8 shows a code stub that would compile Linux-specific code if `LINUX` was defined (say, with a `-DLINUX` flag to a `g++` compiler), or Windows-specific code if either `_WIN32` or `_WIN64` was defined.

Listing 4.8: Conditional compilation for platform-specific code

```
0  #if defined(_WIN32) || defined(_WIN64)
1
2  // Windows specific code here.
3
4  #elif defined(LINUX)
5
```

```
6 // Linux specific code here.
7
8 #endif
```

Note, there is no real functional difference between `#ifdef NAME` and `#if defined(NAME)`, but `#ifdef` can only use a single condition while `#if defined` can use compound conditions (as in the example in Listing 4.8).

When using `#define` directives in Dragonfly for literal replacement (e.g., for the engine version number), the convention is to prefix names with a `DF_` (e.g., `DF_VERSION`). This naming convention is to reduce the risk of potential namespace conflicts between the engine programmer and the game programmer.

When using conditional compilation for header files, the convention is for system utilities to use underscores before and after the name (e.g., `_STRING_H_`), while user code (game code) should never use initial/post underscores. This naming convention is to avoid potential namespace conflicts between the engine developer and the game programmer. For Dragonfly, a double initial underscore and double post underscore is used.

4.3.5 The LogManager

The complete header file for the LogManager is shown in Listing 4.9. Notice the `#ifndef` and `#define` statements at the top for conditional compilation.[3]

The `#include <stdio.h>` on line 6 is for the `FILE` variable, `p_f`. `LOGFILE_NAME` on line 13 provides the name of the logfile, "dragonfly.log".

The methods in the `private` section that allow implementation of the singleton pattern. The attributes provide the file descriptor and whether or not to flush output after each write. Whether flushing is done or not is specified in the `setFlush()` method, but defaults to not flushing (`do_flush` is `false`).

The LogManager constructor should set the type of the Manager to "LogManager" (i.e., `setType("LogManager")`. As in most classes, the constructor should also initialize all attributes, in this case `p_f` to `NULL` and `do_flush` to `false`.

While `startUp()` and `shutDown()` are defined in the Manager class, they are redefined in the LogManager to open the logfile and close the logfile, respectively. Manager `startUp()` and Manager `shutDown()` should be called from LogManager `startUp()` and LogManager `shutDown()`, respectively. Remember, in C++ even if a method is defined in a derived class (e.g., `startUp()` in the LogManager), the parent method can still be called explicitly (e.g., `Manager::startUp()`).

Listing 4.9: LogManager.h

```
0 // The logfile manager.
1
2 #ifndef __LOG_MANAGER_H__
3 #define __LOG_MANAGER_H__
4
5 // System includes.
6 #include <stdio.h>
7
```

[3] For brevity, subsequent Dragonfly header files are not shown with `#ifndef` directives.

```
// Engine includes.
#include "Manager.h"

namespace df {

const std::string LOGFILE_NAME = "dragonfly.log";

class LogManager : public Manager {

 private:
  LogManager();                       // Private since a singleton.
  LogManager(LogManager const&);      // Don't allow copy.
  void operator=(LogManager const&); // Don't allow assignment.
  bool m_do_flush;                   // True if flush to disk after each write.
  FILE *m_p_f;                       // Pointer to logfile struct.

 public:
  // If logfile is open, close it.
  ~LogManager();

  // Get the one and only instance of the LogManager.
  static LogManager &getInstance();

  // Start up the LogManager (open logfile "dragonfly.log").
  int startUp();

  // Shut down the LogManager (close logfile).
  void shutDown();

  // Set flush of logfile after each write.
  void setFlush(bool do_flush=true);

  // Write to logfile. Supports printf() formatting of strings.
  // Return number of bytes written, -1 if error.
  int writeLog(const char *fmt, ...) const;
};

} // end of namespace df
#endif // __LOG_MANAGER_H__
```

Note, for code readability, from here on, a macro is created for each derived manager, providing a two letter acronym for accessing the singleton instance of each manager. For example, Listing 4.10 shows the definition for the `LM` acronym, which should be placed in the `LogManager.h` header file. With this in place, a game programmer could then invoke `LM.writeLog()` without needing to call `getInstance()`.

Listing 4.10: Two-letter acronym for accessing LogManager singleton

```
// Two-letter acronym for easier access to manager.
#define LM df::LogManager::getInstance()
```

4.3.6 Controlling Verbosity (optional)

Logfile messages can be invaluable for debugging, performance tuning or verifying that engine code or game code is working properly. However, logfiles can also be "noisy," with many, many innocuous lines of information hiding the ones that may offer true value. This is especially true of messages that are printed each step of the game loop (typically, 30 times per second), or for messages printed each step of a loop that iterates through all game objects!

While messages can be removed to decrease some of this noise, sometimes messages are useful during later debugging and take time to put back into place. So, instead of removing messages what is often better is to control the verbosity level of messages written to the log – with low verbosity, only essential messages are printed, while with high verbosity, all messages are printed. One way to do this is to have an explicit verbosity setting, depicted in Listing 4.11, via an attribute named `log_level`, with get/set methods, in the LogManager. Verbosity is controlled by changing the log level, even dynamically during run time, with logfile messages only written out when the LogManager log level is greater than or equal to that of the message.

Listing 4.11: Controlling verbosity with log level

```
  // (attribute of LogManager)
 private:
  int log_level;                // Logging level.
  ...

// (Modify  writeLog to take log level as a parameter.)
// Write to logfile.
// Only write if indicated log level >= LogManager log level.
// Supports printf() formatting of strings.
// Return number of bytes written (excluding pre-pends), -1 if error.
void LogManager::writeLog(int log_level, char *fmt, ...)

  // Only print message when verbosity level high enough.
  if log_level > this.log_level then
    va_list args
    ...
  end if
```

In order to avoid repeating code, the version of `writeLog()` without the `log_level` should just invoke the `writeLog()` in Listing 4.11, providing `INT_MAX`, the maximum integer (found in `limits.h`), as the log level.

The method of controlling verbosity in Listing 4.11 is effective but does have a bit of additional overhead, notably a comparison check against the global variable holding the verbosity level for each call. It is not likely this overhead is onerous, but it can be significant, particularly for messages written each step and for each iteration of an object list.

An alternative method is to use conditional compilation, as described in Section 4.3.4 (page 58), with `#ifdef` directives used to decide whether or not to compile in logfile messages. An example is shown in Listing 4.12.

Listing 4.12: Controlling verbosity with conditional compilation

```
#ifdef DEBUG_1
LogManager &log_manager = LogManager::getInstance();
log_manager.writeLog("WorldManager::markForDelete(): deleting object %d",
    p_o -> getId());
#endif
```

The first line indicates the following lines (that actually write the logfile message) are to be compiled in *only* if `DEBUG_1` is defined. The developer can define `DEBUG_1` when testing code, looking for bugs, verifying functionality and so on. When the game is ready to ship, `DEBUG_1` can be left undefined and the code is not compiled into the game. This approach has none of the overhead in Listing 4.11 when messages are not to be written since the messages to write to the logfile are not included at all. As a downside, code can be slightly less readable if there are many `#ifdef` messages.

4.3.7 Development Checkpoint #1!

At this point, development of Dragonfly should commence! The setup from Chapter 2, used for the tutorial, can be used to setup the development environment for your engine. Steps:

1. Setup your development environment, as specified in Chapter 2. Successfully compiling the first tutorial game in Section 3.3.1 on page 15 will ensure that the necessary tools are in place and configured.

2. Discard the pre-compiled Dragonfly libraries and header files from step 1 and prepare a directory structure for your own engine development.

3. Create a Manager base class, both `Manager.h` and `Manager.cpp`. See Listing 4.1 for details on the class definition.

4. Create a LogManager derived class, inheriting from the Manager class. Use Listing 4.9 as a reference.

5. Implement a `writeLog()` function, initially, not part of the LogManager. Have `writeLog()` just produce output to the screen (standard output). Test thoroughly, with no arguments, single arguments and multiple arguments. Be sure to test with different data types (`int`s, `float`s, etc.), too.

6. Move `writeLog()` into the LogManager class as a method. In `game.cpp`, have `#include "LogManager.h"` at the top of the file to include the class definition. Then, instantiate (via `getInstance()`) and start up (via `startUp()`) the LogManager. Check the return values to ensure the calls are working.

7. Test with various calls to `writeLog()` in the LogManager are working, testing single and multiple arguments with different types. Verify that the expected output appears in the logfile, "dragonfly.log".

8. Implement and test any optional elements (e.g., `getTimeString()`), as desired.

Make sure all the code is working thoroughly and is clearly written (indented and commented). As suggested earlier, the LogManager is used heavily during development, both for engine code and for game code, and needs to be robust and reliable before moving forward. This is true of each Development Checkpoint – make sure code is debugged and tested thoroughly before proceeding!