

From Idea To Implementation: SDL Timers in C++

Christopher C. Eineke, chris@chriseineke.com

May 13, 2009

Introduction to SDL and SDL timers

The SDL (Simple Direct Media Library) is a C library designed mainly for cross-platform game development and includes features like synchronization primitives (mutexes), 2D graphics (sprites and surfaces), 3D graphics via OpenGL, an event queue, and timers.

Timers are functions that are called after a delay. One-shot timers are invoked once, periodic timers are invoked repeatedly, and varying timers are invoked repeatedly but with changing intervals.

With the SDL's aptly named functions `SDL_AddTimer` and `SDL_RemoveTimer`, we can add timers to and remove timers from the current process:

```
1  SDL_TimerID  SDL_AddTimer(Uint32  interval , SDL_NewTimerCallback  callback ,
   void  *param ) ;
2  SDL_bool  SDL_RemoveTimer(SDL_TimerID  id ) ;
```

Listing 1: API for SDL Timers

The first parameter, *interval*, tells the SDL how long to wait to invoke the timer for the first time. The second parameter, *callback*, is the function we want the SDL to call for us. And finally, the last parameter, *param*, is an optional ¹ parameter for us to supply to the callback.

The SDL provides a type definition for timer callback that our function must match:

```
1  typedef  Uint32  (*SDL_NewTimerCallback)( Uint32  interval , void  *param ) ;
```

Listing 2: Signature of callback functions for SDL Timers

The callback receives its parameters from the SDL. The *interval* parameter, together with the callback's return value, control the rate at which the timer is fired. We return *zero* in the callback to implement one-shot timers, *interval* to implement periodic timers, or return a different value altogether to implement varying timers. *Param* is the optional data we supplied in the call to `SDL_AddTimer`.

For example, to print "Hello, world!" every half a second, we first write a callback function that prints a string to the screen:

¹Optional meaning if you don't use it you must pass NULL or 0.

```

1 #include <unistd.h>
2 Uint32 hello_world_callback_c(Uint32 interval, void* param)
3 {
4     printf("%s\n", (char*)param);
5     return interval;
6 }

```

Listing 3: Callback function that prints *Hello world!* – C version

```

1 #include <iostream>
2 #include <string>
3 Uint32 hello_world_callback_cpp(Uint32 interval, void* param)
4 {
5     using namespace std;
6     cout << static_cast<string*>(param) << endl;
7     return interval;
8 }

```

Listing 4: Callback function that prints *Hello world!* – C++ version

Then, after we initialize the SDL, we add a periodic timer that fires every 500ms:

```

1 #include <SDL/SDL.h>
2 int main()
3 {
4     SDL_Init(SDL_INIT_TIMER) == -1 ? exit(EXIT_FAILURE), 0 : 0;
5     SDL_TimerID hello_world = SDL_AddTimer(
6         500,
7         &hello_world_callback_c,
8         "Hello, _world!");
9     hello_world == 0 ? exit(EXIT_FAILURE), 0 : 0;
10    sleep(3);
11    SDL_Quit();
12    return EXIT_SUCCESS;
13 }

```

Listing 5: *Hello, world!* main function – C version

```

1 #include <SDL/SDL.h>
2 int main()
3 {
4     SDL_Init(SDL_INIT_TIMER) == -1 ? exit(EXIT_FAILURE), 0 : 0;
5     SDL_TimerID hello_world = SDL_AddTimer(
6         500,
7         &hello_world_callback_cpp,
8         &std::string("Hello, _world!"));
9     hello_world == 0 ? exit(EXIT_FAILURE), 0 : 0;
10    sleep(3);
11    SDL_Quit();
12    return EXIT_SUCCESS;
13 }

```

Listing 6: *Hello, world!* main function – C++ version

and so our program will print the string "Hello, world!" about six times.

SDL timers in C++

We have been introduced to the SDL Timers API and how we can use it in C. If we want to use SDL timers in a C++-friendly manner, we will need to work around small gaps in the API.

In the C version, we defined a function at the file scope to provide a callback to `SDL_AddTimer`. Feigning ignorance, let's try to pass in a pointer to a member function (a method):

```
1 cannot convert
2     Uint32 (HelloWorld::*)(Uint32, void*) to
3     Uint32 (*)(Uint32, void*) for argument 2 to
4     _SDL_TimerID* SDL_AddTimer(Uint32, Uint32 (*)(Uint32, void*), void*)
```

Listing 7: Trying to use a member function as a SDL timer callback results in a syntax error.

The compiler will complain loudly about our attempt at supplying a pointer to a non-static member function where a pointer to a regular function is needed.

Conceptually, every method is passed a pointer to the object (most prominently visible when we use *this* to access something from within the scope of an instance) it is invoked on. In the above error message, it is the `(HelloWorld::*)` part telling us that we got is a pointer to a method belonging to the class `HelloWorld`.

A non-static member function is therefore incompatible with the callback type expected by the `SDL_AddTimer` function, but there are some straightforward and some not-so-straightforward solutions. How can we integrate SDL timers into C++?

Solution 0: Integration, Schmintegration. Functions at the file scope are good enough.

Every time we add a timer to our program, we have to write one callback at the file scope. We could

- Pass an object as the *param* parameter to `SDL_AddTimer`, correctly cast it in our callback, and invoke a method as usual. We lose the data pointer and our code becomes less robust because we need to cast manually and the add more house-keeping than necessary.

```
1 #include <SDL/SDL.h>
2 #include <iostream>
```

```

3  #include <string>
4  using namespace std;
5  struct MsgPrinter {
6      MsgPrinter(string msg) : _msg(msg) { }
7      void print() { cout << _msg << endl; }
8      string _msg;
9  };
10
11  Uint32 message_printer(Uint32 interval, void* param)
12  {
13      MsgPrinter* mp = static_cast<MsgPrinter*>(param);
14      mp->print();
15      return interval;
16  }
17
18  int main()
19  {
20      MsgPrinter mp("Hello ,_world!");
21      SDL_Init(SDLINIT_TIMER) == -1 ? exit(EXIT_FAILURE), 0 : 0;
22      SDL_TimerID hello_world = SDL_AddTimer(
23          500,
24          &message_printer,
25          &mp);
26      hello_world == 0 ? exit(EXIT_FAILURE), 0 : 0;
27      sleep(3);
28      SDL_Quit();
29      return EXIT_SUCCESS;
30  }

```

Listing 8: Using the user data pointer for delegating to a member function.

- Store an object at the file scope and use the user data pointer as usual. We have the data pointer available to us, but we still have to cast manually. We also clutter up the namespace and have to initialize the file-scope pointer on time.

Solution 1: Register a static member function.

Static member functions don't require a pointer to an instance. We can see that by the way call we them in our programs:

```

1  int main() {
2      MyTextPrinter::print("Hello ,_World!");
3  }

```

Listing 9: Example of invoking a static member function on class MyTextPrinter.

This call to print treats the class MyTextPrinter as a namespace. There is no instance associated with it.

It follows that within the class-static scope we cannot access instance variables, but it also means that we can usually² cast static member functions to the regular, non-member functions.

With that information in mind, we can write a class that contains a static member function that'll become the callback for the timer. If we want to add more than a single timer, this scheme will quickly become tedious because we'll either need to write a callback for every timer or need to distinguish between the timers that invoke the single callback.

```
1 #include <SDL/SDL.h>
2 #include <iostream>
3
4 class HelloWorld
5 {
6     public:
7         HelloWorld(char* hi)
8         {
9             timerID = SDL_AddTimer(period, &callback, hi);
10            if (timerID == 0) {
11                /* Handle the error appropriately. */
12            }
13        }
14
15        static Uint32 callback(Uint32 interval, void* param)
16        {
17            using namespace std;
18            cout << static_cast<char*>(param) << endl;
19            return interval;
20        }
21
22        private:
23            static const int period = 500; /* milliseconds */
24            static SDL_TimerID timerID;
25    };
26
27    SDL_TimerID HelloWorld::timerID = 0;
28
29    int main()
30    {
31        SDL_Init(SDL_INIT_TIMER) == -1 ? exit(EXIT_FAILURE), 0 : 0;
32        HelloWorld foo("Hello, _World!");
33        sleep(5);
34        return 0;
35    }
```

Listing 10: Registering a static member function.

If we put the timer starting code into the constructor, we can instantiate the class safely only once. Each new instance would overwrite the previous instance's timer ID and

²See the C++ FAQ's item 33.1 for more information.

we would not be able to stop any other instances' timers. In essence, this is the very first example's C code transposed into a class namespace.

Solution 2: Use a Timer Manager class.

The Timer Manager class encapsulates a static function (the *trampoline*) that invokes the actual callback and a mapping from `SDL_TimerID`s to objects that implement a callback interface. Every object that wants to add or remove timers from the running program needs to use the manager interface to do this. Every time the SDL fires a timer, the manager's trampoline is invoked which looks up in a map the object that is associated with the `SDL_TimerID` and notifies it.

```
1 #include <SDL/SDL.h>
2 class SDL_Callback
3 {
4     public:
5         virtual Uint32 invoke(Uint32 interval, void* param, SDL_TimerID
6             timerID) = 0;
7         virtual ~SDL_Callback() { }
```

Listing 11: Declaration of callback interface.

```
1 #include <SDL/SDL.h>
2 #include <map>
3
4 class SDL_TimerManager
5 {
6     public:
7         static SDL_TimerManager* getInstance();
8
9         SDL_TimerID addTimer(Uint32 interval, SDL_Callback* cb, void* param)
10            ;
11         bool removeTimer(SDL_TimerID id);
12
13     private:
14         typedef struct {
15             void* param;
16             SDL_Callback* cb;
17             SDL_TimerID id;
18         } SDL_ExtUserParam;
19         typedef std::map<SDL_TimerID, SDL_ExtUserParam*> SDL_ParamMap;
20
21         static SDL_TimerManager* INSTANCE;
22         SDL_TimerManager(); // made private for Singleton pattern
23         ~SDL_TimerManager(); // made private for Singleton pattern
24
25         Uint32 forward(Uint32 interval, SDL_ExtUserParam* ext);
26         static Uint32 trampoline(Uint32 interval, void* param);
```

```

27     SDL_ParamMap paramMap;
28 };

```

Listing 12: Declaration of timer manager class.

```

1  #include "../sdl_callback_interface.h"
2  #include "../sdl_timermanager.h"
3  #include <iostream>
4  #include <utility>
5
6  using namespace std;
7
8  SDL_TimerManager* SDL_TimerManager::getInstance()
9  {
10     if (INSTANCE == 0) {
11         INSTANCE = new SDL_TimerManager();
12     }
13
14     return INSTANCE;
15 }
16
17 SDL_TimerID SDL_TimerManager::addTimer(UINT32 interval, SDL_Callback* cb,
18 void* param)
19 {
20     if (cb == 0) {
21         return 0;
22     }
23
24     SDL_ExtUserParam* ext = new SDL_ExtUserParam;
25     ext->param = param;
26     ext->cb = cb;
27     SDL_TimerID id = SDL_AddTimer(interval, &trampoline, ext);
28     if (id == 0) {
29         delete ext;
30         return 0;
31     }
32     ext->id = id;
33
34     pair<SDL_ParamMap::iterator, bool> storage = paramMap.insert(make_pair(
35         id, ext));
36     if (storage.second == false) {
37         SDL_RemoveTimer(id);
38         delete ext;
39         return 0;
40     }
41     return id;
42 }
43
44 bool SDL_TimerManager::removeTimer(SDL_TimerID id)
45 {
46     if (id == 0) {
47         return false;

```

```

46     }
47
48     SDL_ParamMap::iterator param_iter = paramMap.find(id);
49     if (param_iter == paramMap.end()) {
50         return false;
51     }
52
53     SDL_RemoveTimer(param_iter->first);
54     delete param_iter->second;
55     paramMap.erase(param_iter);
56
57     return true;
58 }
59
60 SDL_TimerManager* SDL_TimerManager::INSTANCE = 0;
61
62 SDL_TimerManager::SDL_TimerManager() :
63     paramMap()
64 {
65 }
66
67 SDL_TimerManager::~~SDL_TimerManager()
68 {
69     for (SDL_ParamMap::iterator iter = paramMap.begin(); iter != paramMap.
70         end(); iter++) {
71         delete iter->second;
72         paramMap.erase(iter);
73     }
74     delete INSTANCE;
75 }
76
77 Uint32 SDL_TimerManager::forward(Uint32 interval, SDL_ExtUserParam* ext)
78 {
79     SDL_ParamMap::iterator iter = paramMap.find(ext->id);
80     if (iter == paramMap.end()) {
81         return interval;
82     }
83     return iter->second->cb->invoke(interval, ext->param, ext->id);
84 }
85
86 Uint32 SDL_TimerManager::trampoline(Uint32 interval, void* param)
87 {
88     return SDL_TimerManager::getInstance()->forward(interval, static_cast<
89         SDL_ExtUserParam*>(param));
90 }

```

Listing 13: Definition of timer manager class.

```

1 class MyCallback : public SDL_Callback
2 {
3     public:
4         Uint32 invoke(Uint32 interval, void* param, SDL_TimerID timerID)

```

```

5         {
6             using namespace std;
7             cout << *static_cast<int*>(param) << endl;
8             return interval;
9         }
10    };
11
12    int main()
13    {
14        SDL_Init(SDL_INIT_TIMER);
15
16        MyCallback foo;
17        int bar = 15;
18        SDL_TimerID enk = SDL_TimerManager::getInstance()->addTimer(500, &foo, &
19            bar);
20        cout << enk << endl;
21        sleep(5);
22        cout << SDL_TimerManager::getInstance()->removeTimer(enk) << endl;
23
24        SDL_Quit();
25        return 0;
26    }

```

Listing 14: Example using timer manager class.

We have made it necessary for us to keep two copies of every `SDL_TimerID`, once in the manager and once in a client. We also have reduced the overall performance of the scheme, because we need to look up the mapping between ID's and the respective object. Lastly, we need to manually cast the data pointer again.

Solution 3: Use templates to achieve type safety and quick dispatch.

The last solution combines all of the advantages of the above solutions, but with a slight increase in memory footprint. We can simplify our code, improve type safety, and still have a fast notification mechanism — all with C++ templates. We let the `Timer` template dictate the type of data that is sent with the call to `SDL_AddTimer` and restrict the callbacks to exactly those methods that take the matching template type parameter. Every `Timer` has a static method that notifies the respective client. This scheme allows us to either let clients inherit from the `SDL_Callback` template and implement the *invoke* method, or make up new classes that inherit from it.

```

1  #ifndef SDL_TIMER_H_INCLUDED
2  #define SDL_TIMER_H_INCLUDED
3
4  /*
5   Copyright (c) 2009, Christopher Eineke <chris@chriseineke.com>
6   All rights reserved.
7

```

```

8   Redistribution and use in source and binary forms, with or without
9   modification, are permitted provided that the following conditions are met:
10  * Redistributions of source code must retain the above copyright
11  * notice, this list of conditions and the following disclaimer.
12  * Redistributions in binary form must reproduce the above copyright
13  * notice, this list of conditions and the following disclaimer in the
14  * documentation and/or other materials provided with the distribution.
15  * Neither the name of the author nor the
16  * names of its contributors may be used to endorse or promote products
17  * derived from this software without specific prior written permission.
18
19  THIS SOFTWARE IS PROVIDED BY Christopher Eineke 'AS IS' AND ANY
20  EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
21  WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
22  DISCLAIMED. IN NO EVENT SHALL Christopher Eineke BE LIABLE FOR ANY
23  DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
24  (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES
25  ;
26  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
27  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
28  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
29  THIS
30  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
31  */
32  #include <SDL/SDL.h>
33  #include <utility>
34
35  template <typename T> class SDL_Callback;
36  template <typename T> class SDL_Timer;
37
38  template <typename T>
39  class SDL_Callback
40  {
41  public:
42      virtual Uint32 invoke(SDL_Timer<T>* timer, Uint32 interval, T param)
43      = 0;
44      virtual ~SDL_Callback() { }
45  };
46
47  template <typename T>
48  class SDL_Timer
49  {
50  public:
51      SDL_Timer(SDL_Callback<T>* callback) :
52          id(0),
53          cb(callback),
54          ext(0)
55      {
56      }
57  };

```

```

56     ~SDL_Timer()
57     {
58         removeTimer();
59     }
60
61 }
62
63 bool addTimer(Uint32 interval, T param = 0)
64 {
65     removeTimer();
66     ext = new SDL_ExtUserParam(this, param);
67     id = SDL_AddTimer(interval, &trampoline, static_cast<void*>(ext)
68         );
69     if (id == 0) {
70         delete ext;
71         ext = 0;
72         return false;
73     }
74     else {
75         return true;
76     }
77 }
78
79 bool removeTimer()
80 {
81     bool result = SDL_RemoveTimer(id);
82     id = 0;
83     delete ext;
84     ext = 0;
85     return result;
86 }
87
88 private:
89     typedef std::pair<SDL_Timer<T>*, T> SDL_ExtUserParam;
90
91     SDL_TimerID id;
92     SDL_Callback<T>* cb;
93     SDL_ExtUserParam* ext;
94
95     static Uint32 trampoline(Uint32 interval, void* param)
96     {
97         SDL_ExtUserParam* ext = static_cast<SDL_ExtUserParam*>(
98             param);
99         SDL_Timer<T>* timer = ext->first;
100         T real_param = ext->second;
101         return timer->cb->invoke(timer, interval, real_param);
102     }
103 };
104
105 #endif /* SDL_TIMER_H_INCLUDED */

```

Listing 15: The finished solution using templates.

I believe that this is a good solution to the SDL Timer problem in C++. I hope you had as much fun reading as you will have fun using the code.