A Timer Queue is an Abstract Data Type (ADT) that allows applications to manage a set of timers. Common operations on a Timer Queue include schedule, cancel, and expire. This part of your programming assignment focuses upon building the following two implementations of a Timer Queue:

- **Timer_List** – An unbounded implementation of Timer Queue using the Ordered List implementation from your previous assignment.
- **Timer_Heap** – An implementation of Timer Queue using a heap data structure.

The interface for the abstract base class `Timer_Queue` is defined as follows:

```cpp
class Timer_Queue
{
public:
    // = Initialization and termination methods.
    Timer_Queue (void);
    // Default constructor.
    virtual ~Timer_Queue (void);
    // Destructor - make virtual for proper destruction of inherited
    // classes.
    virtual int is_empty (void) const = 0;
    // True if queue is empty, else false.
    virtual const Time_Value &earliest_time (void) const = 0;
    // Returns the time of the earlier node in the Timer_Queue.
    virtual int schedule (Event_Handler *event_handler,
                          const void *arg,
                          const Time_Value &delay) = 0;
    // Schedule an <event_handler> that will expire after <delay> amount
    // of time. If it expires then <arg> is passed in as the value to
    // the <event_handler>’s <handle_timeout> callback method. This method
    // returns a <timer_id> that uniquely identifies the <event_handler>
    // in an internal list. This <timer_id> can be used to cancel an
    // <event_handler> before it expires. The cancellation ensures that
    // <timer_ids> are unique up to values of greater than 2 billion
    // timers. As long as timers don’t stay around longer than this
    // there should be no problems with accidentally deleting the wrong
    // timer. Returns -1 on failure (which is guaranteed never to be a
    // valid <timer_id>).
    virtual int cancel (int timer_id, const void **arg = 0) = 0;
    // Cancel the single <Event_Handler> that matches the <timer_id>
};;
```
As shown above, the following are a number of classes associated with the Timer Queue:

class Time_Value
  // = TITLE
  // Operations on "timeval" structures.
{
  public:
    // = Useful constants.
    static const Time_Value zero;
    // Constant "0".

    // = Initialization methods.
    Time_Value (long sec = 0, long usec = 0);
    // Constructor.

    Time_Value (const Time_Value &tv);
    // Copy constructor.

    // = The following are accessor/mutator methods.
    long sec (void) const;
    // Get seconds.

    void sec (long sec);
    // Set seconds.

    long usec (void) const;
    // Get microseconds.

    void usec (long usec);
    // Set microseconds.

    // = Helper method
    static Time_Value gettimeofday (void);

    // = The following are arithmetic methods for operating on
    // Time_Values.
    void operator += (const Time_Value &tv);
    // Add <tv> to this.

    void operator -= (const Time_Value &tv);
    // Subtract <tv> to this.

    friend Time_Value operator + (const Time_Value &tv1, const Time_Value &tv2);
    // Adds two Time_Value objects together, returns the sum.

    friend Time_Value operator - (const Time_Value &tv1, const Time_Value &tv2);
    // Subtracts two Time_Value objects, returns the difference.
friend int operator < (const Time_Value &tv1, const Time_Value &tv2);
// True if tv1 < tv2.

friend int operator > (const Time_Value &tv1, const Time_Value &tv2);
// True if tv1 > tv2.

friend int operator <= (const Time_Value &tv1, const Time_Value &tv2);
// True if tv1 <= tv2.

friend int operator >= (const Time_Value &tv1, const Time_Value &tv2);
// True if tv1 >= tv2.

friend int operator == (const Time_Value &tv1, const Time_Value &tv2);
// True if tv1 == tv2.

friend int operator != (const Time_Value &tv1, const Time_Value &tv2);
// True if tv1 != tv2.

private:
void normalize (void);
// Put the timevalue into a canonical form.

long tv_sec_;    
// Seconds.

long tv_usec_;   
// Microseconds.

}

class Event_Handler
// = TITLE
// Provides an interface to an Event Handler;
{
public:
   virtual int handle_timeout (const Time_Value &,
                const void *arg) = 0;

};

The Timer List Implementation

The following class provides a specialization of Timer_Queue that uses an Ordered_List of absolute times. Therefore, in the average- and worst-case, scheduling and canceling Event_Handler timers is $O(N)$ (where $N$ is the total number of timers) and expiring timers is $O(K)$ (where $K$ is the total number of timers that are $<$ the current time of day).

#include "Timer_Queue.h"
class Timer_List : public Timer_Queue
// = TITLE
// Provides a simple implementation of timers using a linked list.
{
public:
   // Initialization and termination methods.
   Timer_List (void);
   // Default constructor.

   virtual ~Timer_List (void);
   // Destructor

   virtual int is_empty (void) const;
   // True if queue is empty, else false.
virtual const Time_Value &earliest_time (void) const;
// Returns the time of the earlier node in the <Timer_List>.

virtual int schedule (Event_Handler *event_handler,
         const void *arg,
         const Time_Value &delay);
// Schedule an <event_handler> that will expire after <delay> amount
// of time. If it expires then <arg> is passed in as the value to
// the <event_handler>'s handle_timeout callback method. This method
// returns a <timer_id> that uniquely identifies the <event_handler>
// in an internal list. This <timer_id> can be used to cancel an
// <event_handler> before it expires. The cancellation ensures that
// <timer_ids> are unique up to values of greater than 2 billion
// timers. As long as timers don’t stay around longer than this
// there should be no problems with accidentally deleting the wrong
// timer. Returns -1 on failure (which is guaranteed never to be a
// valid <timer_id>).

virtual int cancel (int timer_id, const void **arg = 0);
// Cancel the single <Event_Handler> that matches the <timer_id>
// value (which was returned from the <schedule> method). If arg is
// non-NULL then it will be set to point to the “magic cookie”
// argument passed in when the <Event_Handler> was registered. This
// makes it possible to free up the memory and avoid memory leaks.
// Returns 1 if cancellation succeeded and 0 if the <timer_id>
// wasn’t found.

private:
  // You fill in here.
};

The Timer Heap Implementation

The following class provides another specialization of Timer_Queue that uses a Heap of absolute times. A Heap is an “almost complete, partially ordered binary tree.” Heaps are very efficient since in the average- and worst-case, scheduling and canceling Event_Handler timers is \( O(\log N) \) (where \( N \) is the total number of timers) and expiring timers is \( O(\log N) \) (where \( N \) is the total number of timers that are \(<\) the current time of day). Note that this is substantially faster than the Timer_List implementation.

Please refer to your handouts for an explanation of how to implement a Timer_Heap.

#include "Timer_Queue.h"

class Timer_Heap : public Timer_Queue
  // = TITLE
  //   Provides an optimization implementation of timers using a heap.
{
public:
  // = Initialization and termination methods.
  Timer_Heap (size_t size);
  // Default constructor, which creates a heap with <size> elements.

  virtual ~Timer_Heap (void);
  // Destructor

  virtual int is_empty (void) const;
  // True if queue is empty, else false.

  virtual const Time_Value &earliest_time (void) const;
  // Returns the time of the earlier node in the <Timer_Heap>.
virtual int schedule (Event_Handler *event_handler,
    const void *arg,
    const Time_Value &delay);
// Schedule an <event_handler> that will expire after <delay> amount
// of time. If it expires then <arg> is passed in as the value to
// the <event_handler>’s <handle_timeout> callback method. This method
// returns a <timer_id> that uniquely identifies the <event_handler>
// in an internal list. This <timer_id> can be used to cancel an
// <event_handler> before it expires. The cancellation ensures that
// <timer_ids> are unique up to values of greater than 2 billion
// timers. As long as timers don’t stay around longer than this
// there should be no problems with accidentally deleting the wrong
// timer. Returns -1 on failure (which is guaranteed never to be a
// valid <timer_id>).

virtual int cancel (int timer_id, const void **arg = 0);
// Cancel the single <Event_Handler> that matches the <timer_id>
// value (which was returned from the <schedule> method). If arg is
// non-NULL then it will be set to point to the ‘‘magic cookie’’
// argument passed in when the <Event_Handler> was registered. This
// makes it possible to free up the memory and avoid memory leaks.
// Returns 1 if cancellation succeeded and 0 if the <timer_id>
// wasn’t found.

private:
    // You fill in here.
};

Test Driver Code

The following code implements a test driver to test your Timer Queue implementation:

#include "stdio.h"
#include "stdlib.h"
#include "assert.h"
#include "Timer_List.h"
#include "Timer_Heap.h"

// Number of iterations for the performance tests.
static int max_iterations = 100;

// Default size of the heap.
static int max_heap = 100;

class Example_Handler : public Event_Handler
{
public:
    virtual int handle_timeout (const Time_Value &,
        const void *arg)
    {
        assert ((int) arg == 42);
        return 0;
    }
};

static void
test_functionality (Timer_Queue *tq)
{
    Example_Handler eh;
assert (tq->is_empty ());
assert (Time_Value::zero == Time_Value (0));

int timer_id1;

timer_id1 = tq->schedule (&eh, (const void *) 1,
Time_Value::gettimeofday ());
assert (timer_id1 != -1);

assert (tq->schedule (&eh, (const void *) 42,
Time_Value::gettimeofday ()) != -1);
assert (tq->schedule (&eh, (const void *) 42,
Time_Value::gettimeofday ()) != -1);
assert (tq->cancel (timer_id1) == 1);
assert (tq->is_empty () == 0);

assert (tq->expire () == 2);

timer_id1 = tq->schedule (&eh, (const void *) 4,
Time_Value::gettimeofday ());
int timer_id2 = tq->schedule (&eh, (const void *) 5,
Time_Value::gettimeofday ());

assert (timer_id1 != -1 && timer_id2 != -1);
assert (tq->is_empty () == 0);

void *arg = 0;
assert (tq->cancel (timer_id2, &arg) == 1);
assert ((int) arg == 5);
assert (tq->cancel (timer_id1, &arg) == 1);
assert ((int) arg == 4);
assert (tq->expire () == 0);
}

struct Timer_Queues
{
    Timer_Queue *queue_;
    // Pointer to the subclass of <Timer_Queue> that we're testing.
    const char *name_;
    // Name of the Queue that we're testing.
};

static Timer_Queues timer_queues[] =
{
    { new Timer_List, "Timer_List" },
    { 0, "Timer_Heap" },
    { 0, 0 }
};

int
main (int argc, char *argv[])
{
    if (argc > 1)
        max_iterations = ::atoi (argv[1]);
    else if (argv > 2)
        max_heap = ::atoi (argv[2]);

    timer_queues[1].queue_ = new Timer_Heap (max_heap);

    for (int i = 0; timer_queues[i].name_ != 0; i++)
        { fprintf (stderr, "**** starting test of %s\n", timer_queues[i].name_);
            test_functionality (timer_queues[i].queue_);
            delete timer_queues[i].queue_;
        }
return 0;

Getting Started

You can get the "shells" and Makefile for part one of the program from your account on cec. These files are stored in /project/adaptive/cs242/assignment-5/. Here's a script that shows you how to set everything up and get these files:

% cd ~/cs242
% mkdir assignment-5
% cd assignment-5
% cp -r /project/adaptive/cs242/assignment-5/* .
% ls
Makefile
timer-test.C
Event_Handler.h
Time_Value.C
Time_Value.h
Timer_Heap.C
Timer_Heap.h
Timer_List.C
Timer_List.h
Timer_Queue.C
Timer_Queue.h
% make

The Makefile and various header files are written for you. All you need to do is edit the *.C files to add the methods that implement the Timer Queues. Note that you’ll need to copy the Ordered_List files from your previous assignment and add them to the Makefile.