zibaldone tutorial

Antonio Buccino

January 18, 2014

Introduction

zibaldone is a C++ library that provides support for multithread programming.
One of the primary purposes of **zibaldone** is to make threads completely independent one from each other. We want that a thread is not required to know in advance what other threads will send or receive events from him. We want that a thread is not required to have a cross reference (pointer or something else) to other threads. The reason is that a such architecture generally needs a complex and rigid startup sequence as a thread cannot refer a pointer to another thread before the second one is ready.

Quick start

To understand how use **zibaldone**, you need to:

1. know how use events
2. know how create a thread

events

The new version of **zibaldone** manages two kind of events:

1. **A byte buffer identified by a std::string**

   To create a new event of this kind, you need only to istantiate an object of type Event. Event constructor requires as parameter a string eventId that uniquelly identifies that event.

2. **An object identified by a std::string**

   This kind of event has been introduced in the last version of **zibaldone**. Using structured object to encapsulate event data simplifies event management from a semantic point of view but requires some additional attention.

   To create a new event of this kind, you need to inherit from EventObject class and define the method clone that makes a deep copy of the event data (your inherited class). As usual, you can leave this task to the compiler if you do not allocate heap memory in your class.

   As a tip, remember to comply the well known “big 3 rule” of C++, that is if your class needs to define the destructor or the copy constructor or the copy assigment operator then it should probably explicitly define all three. If you comply that rule then if ”myEvent” is the class inherited from EventObject the clone method can be defined simply as:

   ```
   anEvent* clone()const{return new anEvent(*this);}
   ```

Take a look to the examples to have an idea on how proceed.

Event and EventObject constructor requires as parameter a string eventId that uniquelly identifies that event. An event can be fired by calling the method emitEvent(). All the threads who have registered to an event will receive a copy of that event in their event queue, whenever someone somewhere calls emitEvent() method on that event.

A thread can register to an event (thus will receive a copy of it when it's emitted) by calling register2Event method and specifying the eventId.

A thread can check out an event from his event queue by calling the method pullout. Once a thread has taken an event, it's his responibility to delete it after use.

threads

Creating a new thread is quite simple:

1. create a new class derived from class Thread. Call register2Event to register to desired event id. It's possibile to register to an event anytime during thread life, but tipically it's a good idea to perform events registration into the constructor. For example, the following code creates a new thread "foo", and register to events "goofy" and "pluto":

```
class foo : public Thread
{
    //...
public:
    foo(/*...*/)
    {
        //...
        register2Event("goofy");
        register2Event("pluto");
    }
}
```

2. define the pure virtual method run that implements the thread loop. Thread loop has to checkout events from queue and handle them. It's mandatory to handle "StopThreadId" event by gracefully terminating the thread (that is freeing allocated memory, ... and then exiting from thread loop function run.

```
void foo::run()
{
    bool exit=false;
    while(!exit) {
        Event* Ev = pullOut();//checkout the next event from event queue
        if(Ev){
            std::string eventId = Ev->eventId();
            ziblog(LOG::INFO, "received event %s", eventId.c_str());
            if(eventId == StopThreadEventId) exit = true;//it's mandatory to handle StopThreadEventId!
            else if(eventId=="goofy") handleGoofy();//handling of event "goofy"
            else if(eventId=="pluto") handlePluto();//handling of event "pluto"
            else ziblog(LOG::ERROR, "UNHANDLED EVENT %s", eventId);
            delete Ev;
        }
    }
}
```

3. instantiate an object and start the thread:

```
//...
foo fooThread;
fooThread.Start();
//...
```

Take a look to testThread.cpp example!

Detailed description.

The Event class

As told before, any event in **zibaldone** id an object istance of class Event.

1) Defining a new event

Any event id uniquelly identified in the whole multithread process by a string eventId. When a thread is interested to an event, he registers to the related event identifier and after this operation a copy of that event is pushed in his event queue every time a thread (in the same process) emits that event. Notice that this is a one-to-multi relationship, i.e. if two threads are registered to the same event then each one receives his own copy. The event name should be chosen carefully. As an example we can define the event "rxData_ttyS0" to identify the event that incapsulates the received byte stream on the serial port ttyS0 ("fooEvent" is not a good event name!). As a tip consider the idea of using a const global string variable to hold an event name, to be then sure all threads refer to the same name when they want to deal with the same event. For example, if we define the event "rxData_ttyS0" but by mistake a thread who wants to receive that event registers to "RxData_ttyS0" (capital "R" instead of "r" in "rxData_ttyS0") then he will never receive that event because the mediator will not notify him because he is an observer of a different event. The event identifier (`std::string eventId`) has to be passed to the base class Event constructor:

2) emitting an event

Whenever someone wants to emit an event, he has only to instantiate the related object and call the method "emitEvent()". When this happens every thread registered to that event receives his own copy of the event emitted in his event queue.

As we said above, an event is something that can be emitted by anyone. If we want an event to be emitted only by a specified thread, we can obtain this by means of scope rules, making that event (i.e. the related class) visible only by the thread who are allowed to emit: this way other thread cannot instantiate that event and so they cannot emit it.

The Thread class

The Thread class is the base class for every thread in **zibaldone**. The available methods are:

1) `void run()`

It's mandatory to implement this method (it's defined as pure virtual).

This method is the thread loop function (it's what the thread does during its life cycle). The thread will stop and die when this function ends.

In the thread loop function we wait for events and process them. The thread will receive only the events he has registered to (see previuos step) plus the event "StopThreadEvent". This event can be received anytime during thread life cycle and **must** be processed terminating the thread as soon as possibile. A good way to process this event is to define a boolean class private member variable "exit" with initial value set to "false". The thread loop then will terminate by setting "exit" to true whenever the event "StopThreadEvent" is received (you can also use the classic for(;;) loop and return when receiving "StopThreadEvent" ... this is only a tip to terminate the thread in a well known point (begin of each loop cycle) to simplify avoiding memory leak and dangling reference (remember you have do delete the received event pointer)

The std::string “StopThreadEvent” is for convenience defined in the file “Thread.h” as

```
const std::string StopThreadEventId = "StopThreadEvent";
```

This way you can use that variable to identify it instead of using directly the string of the eventId “StopThreadEvent”. This may help prevent typing error in event name. It’s like when in C language we used the `#define` to assign a rem name to an integer constant: it’s easy to remember, prevents error and makes it easy to change event name if we want to: you can modify only this variable instead of look all the source code and change every statement that contains the event name.

2) `void register2Event(std::string eventId)`

After this method has been called, whenever someone emits an event identified by eventId (that is whenever someone calls emitEvent on an object instante of Event with eventId equal to that passed as parameter to register2Event) a copy of that event will be pushed into the event queue. To clarify, if a Thread T1 calls register2Event with parameter “fooEvent”, anytime an object istance of Event with eventId=“fooEvent” is emitted (that is someone calls emitEvent on that object in the same address space) a copy of that event will be pushed into T1 event queue. Obviuosly one can register to as many event as he wants to. A good way to do thread Event registration phase is into the constructor, so that when the thread object is istantiated it’s clear what event he has to deal with:

Note that the registration phase is very important, and we have to do it carefully because we will be notified only about the events with **exactly** the name specified in the registration. All threads must handle event “StopThreadEvent” by exiting the run() method and so terminating the thread itself.

3) The method “pullOut” allows the thread to look for events in his event queue.

A call to pullOut sets the thread in sleeping mode. Note that this sleeping mode is not a busy waiting so the O.S. can use the processor and run other task/thread.

A simple call to pullOut without parameters sets the thread in sleeping mode until he receives an event, that is untill someone emits an Event the thread has registered to.

```
void fooThread::run()
{
    while(!exit)
    {
        /* ... */
        Event* Ev = pullout();//run sleeps here until someone emits an
                              //event that fooThread has registered to
    }
}
```

when pullOut returns, Ev contains the pointer of one event between those the thread has registered to. Ev points to the thread own copy of the event. It’s responsibility of the thread to deallocate the referenced memory area once he has processed the event:

```
void fooThread::run()
{
    while(!exit)
    {
        /* ... */
        Event* Ev = pullout();//run sleeps here until someone emits en
                              //event that fooThread has registered to
        //...
        //process event
        //...
        delete Ev;
    }
}
```

When we receive an event, we can discriminate what event is it by using the eventId() method of Event class. For example, if fooThread has registered to “fooEvent” and “ErrorEvent” he can discriminate what event has received, in the following way:

```
void fooThread::run()
{
    /* ... */
    while(!exit)
    {
        Event* Ev = pullout();//run sleeps here until someone emits en event that fooThread has registered to
        std::string eventId = Ev->eventId();
        if(eventId == "fooEvent") {
            //... process "fooEvent" ...
        } else if (eventId == "ErrorEvent") {
            // ... process "ErrorEvent" ...
        } else if (eventId == "StopThreadEvent") exit = true;
        //...

        delete Ev;
    }
}
```

It’s important to register to the correct eventId: if not, the thread will never receive the event he’s waiting for.

The method pullOut has the following variants:

```
Event* pullOut(int maxWaitMsec = WAIT4EVER);
```

This variant returns the first available event in the thread queue waiting at least for “maxWaitMsec” msecs. If no event is in the queue within that time the method returns a NULL pointer. The default value “WAIT4EVER” makes the thread wait indefinitely until there is an event in his queue. In the opposite, to peek the queue to see if is there an event without waiting it’s possible call pullOut with maxWaitMsec set to zero.

```
Event* pullOut(std::string eventId, int maxWaitMsec = NOWAIT);
```

This variant works as the previous one but ignores events other than the specified eventId. The other events are simply left in the queue. The only one event that is notified to the caller is the “StopThreadEvent”. Keep in mind that “StopThreadEvent” has priority over all other events, so “eventId” is notified only if there isn’t a StopThreadEvent (if both “StopThreadEvent” and “eventId” are present into the thread queue then it’s “StopThreadEvent” that pullOut returns to the caller). The default value for maxWaitMsec is now set to zero (no wait). This decision is due to avoid ambiguous situation (what if the desidered event is not in queue? Have we to wait indefinitely and ignore all other events? Clearly not: this would lead to starvation, then if we want to check the queue for a specified event then we have also to give a finite max wait time).

```
Event* pullOut(std::deque <std::string> eventNames, int maxWaitMsec = NOWAIT);
```

The last variant is similar to the previous one but instead of waiting for a specified eventId returns the first available event in the eventIds list (with the same conditions as the pullOut version with only one eventId, i.e. maxWaitMsec cannot be set to indefinite wait and “StopThreadEvent” has the precedente on all other events)

4) `void Start()`

And now... fire to the powder! We have to call the method Start() to start the thread.

Once a Thread class object is istantiated, the associated thread loop function doesn’t start automatically. This choice is due principally to two reasons. The first one is that to start automatically a thread we should do it in the base constructor, but while the run() loop thread cycle function is defined in the inherited class,

we would start something that is not yet ready; moreover, since the thread loop function shall use some member variables defined in the inherited class, if we start the loop in the base class constructor this function could access memory areas not yet initialized (base constructor is called before subclass constructor) leading to impredicible behaviour. The second reason is that calling explicitly a Start() method give more control to the **zibaldone** user who decide exactly when start a thread (he can prepare all, istantiate the object and start effectively the thread some time later).

The method Start() triggers the thread loop. Keep in mind that the thread is created as son of the process who calls the Start() method. To keep alive this process and make the thread be alive we can call the Join() method so the caller hangs and wait for the thread to finish.

5) `void Stop();`

To stop a thread we have to call Stop() method on it. This method pushes a "StopThreadEvent" into the target thread queue (the one incapsulated by the object who holds the method) and hangs until the thread loop function has exited. Normally the hang time is very small, because the StopThreadEvent has priority on all other events and so that time is what thread loop function needs just to complete the current loop cycle and exit. This is important because the thread loop may have allocated some memory (for example the heap area related to an event) and before terminating we need to allow the thread loop to free that memory to prevent memory leak. After the Stop() method has return it's safe to destroy the related object (that incapsulates the thread). It's not safe to destroy the incapsulating thread object before the thread loop function has terminated because that loop may use some member variables of the incapsulating object, and if we destroy it while running we cause thread loop have unpredictable behaviour because he thinks to access some object member variable that has been deleted.

As we said above Stop() method hangs untill the thread has terminated so when the Stop() method returns it's safe to destroy the related object incapsulating thread.

Summarizing: after creating the Thread object, we have to call the Start() method, the Join() method (if needed) and to exit the Stop() method. For example we can have two thread, A and B. We start both and Join on B. B stops when receives some external event and when this happens then the program continues to the next instruction to B.Join() that can be A.Stop().

These rules are a low price to pay to have very simple Thread creation and management by means of **zibaldone**. Infact most of the limitations about thread termination and start, are intrinsically linked to multithread programming instead of beeing **zibaldone** limitations.

Timers

Timer class allows user to easily start a timeout event. Timer constructor takes two parameters: the first one, mandatory, is a string that uniquely identifies the timer. The second one, optional, sets a default timer duration. When the method Start of a timer object is called, an event with identifier equal to timer string identifier will be emitted at duration expiration. Timer duration can be set in constructor of timer object and/or overwritten on Start call.

Take a look at testTimer.cpp example to understand how use class Timer.

Inter Process Communication

Interprocess communication allows communication between different process. As well known two different process does not share address space and then it cannot be used shared memory to communicate. As stated before, **zibaldone** event management is intended to be used in the same address space, that is between threads who belong to the same process and that thus can share memory. **zibaldone** IPC is obtained by means of socket, that can be of type PF_LOCAL or AF_INET. The second one (AF_INET) is the classic tcp socket

and can be used both in the same machine or in different machine connected by a network. The first one (PF_LOCAL) can be used only for inter process communication in the same machine and it's a little bit more efficient than the other one (works like a pipe). Both of the two kind of IPC share the same interface, and once created and connected allow on both side of communication side to transmit/receive byte stream encapsulated in **zibaldone** event.

Take a look at testSocket.cpp example that shows how use Socket class.

Log

class Log allows to write log messages both on file and terminal console. To log a message you have to call ziblog function. This function works like the C printf, except that the first parameter sets the log level of the following messages. The log levels, ordered by severity are: DEBUG, INFO, WARNING and ERROR. To enable log, you have prior to call LOG::set() method, where you can declare log level: all message log whose level is upper than the one set will be shown. LOG::set method allows to disable log message visualization on terminal console (default is true). Once LOG::set has been called all ziblog will be shown. LOG::disable method allows to disable log messages.

Serial Port

SerialPortHandler class implements a thread that handles a serial port and can be used to read/write on it. When someone wants data to be transmitted on serial port, he has only to emit an event containing these data. The event Id must be set to the identifier obtained by calling getTxDataEventId on an object instance of SerialPortHandler or by calling the static namesake method getTxDataEventId with port as parameter. Similarly, threads who wants to receive data from serial port have to register to event identifier obtained by calling getRxDataEventId on an object instance of SerialPortHandler or by calling the static namesake method getRxDataEventId with port as parameter.

Take a look at testSerialPort.cpp example to understand how use class SerialPortHandler.

Tools

zibTools class offers miscellaneous methods. Currently there is only a method to obtain the list of available serial port. Take a look at testZibTools.cpp example to understand how use class zibTools.

Compile and Linking Note:

Linux version of **zibaldone** needs "lpthread" and "lrt" to work, so you need to link them togheter with libzibaldone to compile and link your programs. Remember that linking order (depending on the linker) maybe important, and if it's then you have to link librt and libpthread before libzibaldone.