zibaldone tutorial

Antonio Buccino

September 7, 2012

Events in zibaldone

Before starting [1] with a step by step sequence to define a new thread using **zibaldone**, we need first some clarifications. 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 some other 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.
Because of the above reasons, we have focused on the concept of **Event**. In the **zibaldone** way, an **Event** is something that is uniquely identified by a name (`std::string`).
Defining a new event is quite simple. As an example, let's define the new event "fooEvent":

1) inherit the new event from the class Event:

```
class fooEvent : public Event {
    //...
};
```

2) assign a **unique identifier name** to the event. This step is very important because the name is what identifies an event of a certain type inside the whole multithread process. When a thread is interested to an event, he register to the related event name 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 notify him because he is an observer of a different event. The event name has to be passed to the base class Event constructor:

```
fooEvent::fooEvent(/* ... */) : Event("fooEvent")
{
    //...
}
```

3) add to the class the following method "clone":

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

[1] for a very quick start, take a look at the code at the end ot this tutorial!

this method is mandatory, and it's used by the mediator to push a copy of the event into the event queues of the threads who has registered to that event. We cannot define this event in the base class because in that case it will clone only the base class data and we want the whole user event data to be part of the event copy that is pushed to the registered threads event queue.

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.

Before define a new event, consider the idea to inherit one of the events defined in the library. In particular the event DataEvent is useful to define all sort of rx/tx data event from/to a sap (service acess point), i.e. a serial port, a socket,...

As we told 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.

Tip: while defining a new event, be careful to respect the well known "rule of big 3", i.e. if you need one between destructor, copy constructor, operator "=" overload, then probably you'll need all of them.

Note about clone method: if we had done:

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

in the base class, only the data in the base class "Event" shall be copied into the registered thread event queue instead of the whole "fooEvent" data. This is the reason why we need a clone method as specified above.

Threads in zibaldone

Now that it's clear what is an Event in the **zibaldone** way, we can go on and have a look about the steps to define a new thread. Let's define the new thread "fooThread":

1) define the new class "fooThread" inherited from class "Thread"

```
class mythread : public Thread {
    //...
}
```

2) decide what event "fooThread" is interested to and call the method "register2Event" for every event name. As an example suppose that "fooThread" wants to receive the events whose name are the strings "fooEvents" and "ErrorEvent" (ErrorEvent is defined in the file "Thread.h" and is intended to contain some error message string). Then fooThread has to call twice the (inherited from base class Thread) method "register2Event".

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:

```
fooThread::fooThread()
{

    // ... member initializazion and other constructor operations

    register2Event("fooEvent");
    register2Event("ErrorEvent");//we could use the variable defined in Thread.cpp
                                 //and so write register2Event(ErrorEvent::ErrorEventName);
}
```

Once a thread has registered to an Event, whenever someone (in the same process address space) emits that event (i.e. istantiate an object from that class and then calls the emitEvent() method on it) then all the registered thread will receive their own copy in their event queue.

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. As we said above, when a thread is created, by default it will be automatically registered to the event with name "StopThreadEvent". All threads must handle this event by exiting the run() method and so terminating the thread itself.

3) define the method

```
void fooThread::run()
```

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 StopThreadEvent is defined in the file "Thread.h". The event name "StopEventThread" is for convenience defined in the file "Thread.cpp" as

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

This way you can use that variable to identify it instead of using directly the string of the name "StopTheradEvent". 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.

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 that 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 pointed 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 eventName() method or by means of a dynamic cast operation. 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 eventName = Ev->eventName();
        if(eventName == "fooEvent") {
            //... process "fooEvent" ...
        } else if (eventName == "ErrorEvent") {
            // ... process "ErrorEvent" ...
        } else if (eventName == "StopThreadEvent") exit = true;
        //...

        delete Ev;
    }
}
```

or, if "fooEventClass" and "ErrorEventClass" are the class (inherited from Event) relative to "fooEvent" and "ErrorEvent" we can discriminate this way:

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

        delete Ev;
    }
}
```

what is important is to register to the correct event name, after that, we can discriminate a received event in the way we are more comfortable with. Consider that discriminating by means of eventName leads to a simmetry between registration and event detection, while using dynamic cast leads to better performances.

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 eventName, int maxWaitMsec = NOWAIT);
```

This variant works as the previous one but ignores events other than the specified eventName. 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 "eventName" is notified only if there isn't a StopThreadEvent (if both "StopThreadEvent" and "eventName" 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 eventName returns the first available event in the eventNames list (with the same conditions as the pullOut version with only one eventName, i.e. maxWaitMsec cannot be set to indefinite wait and "StopThreadEvent" has the precedente on all other events)

4) create the thread object and start the thread.

and now... fire to the powder! We have to create an object from the class fooThread and 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. 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 have to call the Join() method so the caller hangs and wait for the thread to finish.

Note that we need a Start() method instead of automatically starting the thread loop function into the base class Thread constructor. Infact the thread loop function (run) is defined in the inherited class and it can use member variabiles of the inherited class. If we trigger the loop thread in the base class constructor, the thread loop may use not yet initialized subclass member variabiles (base constructor is called before subclass constructor) leading to impredicible behaviour.

To stop a thread we have to call Stop() method on it. This method sends a "StopThreadEvent" to the target thread (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.

We couldn't embed the Stop() into the Thread base class destructor because this way the thread loop function would continue running untill the base class destructor had not been called. As well known the base class destructor is called after the inherited class destructor, and so the thread loop would continue his life for a little time between the end of the inherited class destructor and the base class destructor call. This means that the thread loop function could use again some member variabiles (defined in the subclass) yet destroyed by subclass destructor.

As we said abovem Stop() method hangs untill the thread has terminated so when the Stop() method returns it's safe to destroy the related object incapsulatinf thread. But it's not safe to send a broadcast StopThreadEvent instantiating a StopThreadEvent object and calling emitEvent() on it because this way we cannot know when we can destroy the related objects as we have no confirmation about the end of each thread loop function.

Moreover, even thinking of a way to overcome the above mentioned problem (I've thought about some workaround using mutex) embedding the Stop() into the destructor of the thread incapsulating class, we had to set Stop() method as protected to avoid collisions and that means that the only way to stop a thread associated to an automatic variabile object would have been to leave the related scope (we cannot delete a variable that isn't in the heap memory!!!)

Another way to stop a thread is by means of some condition (i.e. some external event, like a key press, trapped by the thread loop).

If we destroy a Thread object without prior stopping the thread loop by calling Stop() method, then the thread loop function will continue his life even if no events will be notified (the destructor removes all thread registration to Events). This is not good, because this wastes processor time doing nothing useful.

For the above mentioned reasons, we have made the choice of not canceling a thread in the destructor by means of pthread_cancel because this destroys the thread loop function without considering if that thread was terminating some operation (that may also involve other threads) then that may cause both memory leaks (if the thread loop suddenly break had allocated some memory but had killed before freeing that memory) or other unpredictable behaviour if some other thread was waiting for some notification/event that only the destroyed thread was able to send. The right way to stop a thread is then to decide it knowingly, and the only right way to do that is use the Stop() method.

Finally, while it may seem obvious, the **zibaldone** user should care of not calling the Stop() method nor the Join() method inside the thread loop function "run()" itself. Clearly it's a non sense to join a thread to himself, but someone may think about stopping a thread itself while in the run() loop function by calling Stop() there instead of exiting the loop cycle (break, return or something else!). This wold lead to a deadlock because as explained above, the Stop() method hangs and wait for the loop cycle to end, but if we call it inside the loop function itself.....

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().

As we'll see in the next examples, 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, start, are intrinsically linked to multithread programming instead of beeing **zibaldone** limitations.

Note:

- **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.
- Event identification by name is needed by threads to declare what event type the want to receive. When a thread receives an event, he can discriminate what event is it both by event name variable member and by a dynamic cast operation.

Example

Let’s now put all togheter in an example:

file “example.cpp”

```
/*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 2 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see http://www.gnu.org/licenses/.
*/

/* example of using zibaldone library. This example contains two simple
   thread: the keyLogger Thread stands listening on keyboard press. When
   a key is pressed, keyLogger prints the message "keyLogger > you have
   pressed the key <key> and then sends an event, containing the pressed
   key, to the second thread "parrot" who echoes this key prepending to it
   the message "parrot > keyLogger told me you pressed <key>". For exmaple
   if the user presses the key 'A' then we should see:
   "keylogger > you have pressed the key A"
   "parrot > keyLogger told me you pressed A"
*/

#include "Thread.h"

// keyLogger Thread ------------------------------------------------------------------------
class keyLogger : public Thread {//to make keyLogger be a thread it's
                                 //sufficient to inherit from Thread

    //note: keyLogger does not receive any event (only reads from console
    //by means of a std::cin) and so we don't need do register to an event.
    //Therefore we can use the default constructor generated by compiler

    void run();//the thread loop function
    bool exit;

public:
    class keyPressedEvent : public Event {//An event can be defined in any scope.
                                          //We define it here because keyPressedEvent
                                          //concerns keyLogger Thread
        char key;
    public:
        static const std::string keyPressedEventName;//instead of remembering what event name will
                                                      //we assign to keyPressedEvent, we store it here
                                                      //(it's like when in C language we used the #define
                                                      //to assign a name to a integer constant instead of
                                                      //using directly the number itself

        Event* clone() const{return new keyPressedEvent(*this);}//mandatory clone method

        keyPressedEvent(char key) : Event(keyPressedEventName), key(key){}//constructor
```

```
        char data(){return key;}
    };
};

const std::string keyLogger::keyPressedEvent::keyPressedEventName = "keyPressedEvent";

void keyLogger::run()
{
    //this is keyLogger thread loop function...
    char cmd;
    exit = false;
    std::cout<<"keyLogger Thread is starting ... \n press any key, or 'q' to quit the program\n";
    Event* Ev = NULL;
    while(!exit)
    {
        //keyLogger have to listen for keys from keyboard and for the only event he can receive:
        //StopThreadEvent (keyLogger has not registered to any event, then the only one event he
        //can receive is StopThreadEvent, always receivable by default). To manage this, we use
        //pullOut with 1 sec timeout, this way we poll keyboard input and event queue periodically
        //every second.

        Ev = pullOut(1000);//chech the event queue for an event. If nothing is received
                           //within 1 sec, pullOut stop blocking and returns NULL.
        if(Ev) {
            //an event has been received... The only one event we can receive is StopThreadEvent...
            if(Ev->eventName() == StopThreadEvent::StopThreadEventName) {
                std::cout<<"StopThreadEvent received... keyLogger says Bye Bye!!!!\n";
                exit = true;
            }
            else std::cerr<<"something wrong has happened.... event "<<Ev->eventName()<<"is not expected to be here!";
        } else {
            //no event in queue. gonna check if user has pressed a key.
            if(std::cin.peek() != std::iostream::traits_type::eof()) {
                std::cin>>cmd;
                if(cmd == 'q' || cmd == 'Q') {
                    std::cout<<"keyLogger says Bye bye!\n";
                    exit = true;
                } else {
                    keyPressedEvent kev(cmd);
                    kev.emitEvent();
                }
            }
        }
    }
}

//-----------------------------------------------------------------------------------------


// parrot Thread ----------------------------------------------------------------------------
class parrot : public Thread {
    void run();
    bool exit;
public:
    parrot();
};

parrot::parrot()
{
exit = false;
register2Event(keyLogger::keyPressedEvent::keyPressedEventName);//register to event keyPressedEvent. Everytime this event is
                                                                //emitted somewhere in the same address space (i.e. whithin the
                                                                //same multithread process) parrot will receive his own copy of
                                                                //that event in his event queue and can extract it by means of
                                                                //the method pullOut.
}

void parrot::run()
{
    //this is parrot thread loop function...
    std::cout<<"parrot Thread is starting ... \n";
    Event* Ev = NULL;
    while(!exit)
    {
        Ev = pullOut();//stay here untill there is an event for me...
        std::string eventName = Ev->eventName();//note: if we are here, Ev cannot be NULL!
        if(eventName == StopThreadEvent::StopThreadEventName) {
            std::cout<<"StopThreadEvent received... parrot says Bye Bye!!!!\n";
            exit = true;
        } else if(eventName == keyLogger::keyPressedEvent::keyPressedEventName){
```

```
            //process this event...
            keyLogger::keyPressedEvent* e = (keyLogger::keyPressedEvent* ) Ev;
            std::cout<<"parrot > keyLogger told me you pressed the key "<<e->data()<<std::endl;
        } else std::cout<<"Unexpected event "<<eventName<<std::endl;
        delete Ev;
    }
}
//---------------------------------------------------------------------------------------------

//now let's create the thread object and start them
int main()
{
    keyLogger klt;//prepare a keyLogger Thread
    parrot pt;//prepare parrot Thread

    //let's start both of them:
    klt.Start();
    pt.Start();

    /*
    now we have to Join main() function to one thread.

    If we don't do that, main function will end (return) and both of klt and pt will stop
    their execution immediately. If this happens (thread killed instead of stopped in the
    right way) we cannot predict what shall happen: it may happen that the thread function
    was accessing a member data variabile that has been destroyed and so we can have a
    segmentation fault...

    what thread do we Join() to? klt or pt? At first sight it might seem that it's the same.
    but as we'll see next, it's not indifferent what thread join to.

    Suppose we join to pt, then:

    pt.Join();//programs hangs here untill pt Thread doesn't stop his execution. When this
              //line will be overcame, it means that pt has stopped and we have only to stop
              //the other thread: klt

    klt.Stop();


    Good! Ok!

    ....but it doesn't work: main blocks to pt.Join() instruction and never goes on.
    What's wrong with that?

    the problem is that pt thread (the one we have joined to) never ends! Infact pt stops execution only
    when he receives the StopThreadEvent (take a look at "void parrot::run()" method), but nobody sends
    this event to him! Klt exits when someone presses 'q" or 'Q' (look at keyLogger loop function), but
    klt doesn't send a StopThreadEvent to pt! (nor he is expected to: he neither have to know that pt exists!)
    So the right thing to do is to join to a thread that someone can stop! (here is the user who presses
    keys on the keyboard).

    Then we have to join to klt, and stop pt when klt ends!

    */

    klt.Join();//programs hangs here untill klt Thread doesn't stop his execution. When this
               //line will be overcame, it means that klt has stopped and we have only to stop
               //the other thread: pt

    pt.Stop();

    //Note that if you don't know what to do (but this is not a good thing!!! Especially in multithread programming...)
    //sometime you can join to both and stop both (joining to a dead thread or stopping an already stopped thread has
    //no consequences: nothing literally will be done in these cases). However, in this specific example (we have chosen
    //it not randomly!!!!) if we join pt before klt, because of what explained above, the programs hangs because pt
    //will never stop and the source line "pt.Join()" will be never overcame!

    return 0;
}
```

makefile

N.B.: before compliling and linking the example you have to compile and link the zibaldone library. In the following makefile we will identify with “path_to_zibaldoneLib” the path to the directory containing libzibaldone.so and with “path_to_zibaldoneHeaders” the path to the directory containing the zibaldone header files.

```
CC = g++
DEBUG = -g

ifeq ($(shell uname -m), x86_64)
    ARCH_CFLAGS = -fPIC
else
    ARCH_CFLAGS =
endif

CFLAGS = -Wall -Werror $(DEBUG) $(ARCH_CFLAGS)

IFLAGS = -I. \
         -Ipath_to_zibaldoneHeaders\Thread

LFLAGS = -Lpath_to_zibaldoneLib \
         -lzibaldone \
         -lpthread \
         -lrt #beware that -lpthread and -lrt have to be put after -lzibaldone!

# dependencies
DEPS =

# objects
OBJ = example.o

# targets
%.o: %.cpp $(DEPS)
    @echo "building $@"
    $(CC) -c -o $@ $< $(CFLAGS) $(IFLAGS)

EXAMPLE: $(OBJ)
    @echo "building $@"
    $(CC) -o $@ $^ $(LFLAGS)

clean:
    rm -f ./EXAMPLE $(OBJ)
```