

C++ And Threads

by Randy Charles Morin

If you've ever done multithreaded programming then most likely it was in C++. I haven't heard of many developers using the CreateThread Win32 API function from Visual Basic or even Delphi. The reason is that the advantages of C++ and multiple threads are usually the same.

You want more responsiveness from some application. How do you do it? Well you could have multiple threads in order to limit blocking of one application request by another. You could also use a more low level language.

It is highly doubtful that you'd program these days in x86 Machine Language or Assembler. So your next best choice is C++. And still today with the advent of more and more languages, i.e. Delphi, Visual Basic, Java, Perl, C# and many more, the preferred performance oriented language remain C++.

The conclusion is that performance oriented applications are generally developed in C++ and with multithreaded C++ libraries. I myself have been doing this for years. I have written dozens of C++ servers and each had some multithreaded aspects about them.

If you know one thing about me from the previous four paragraphs, you know that I'm a C++ developer. Generalizing on this theme, you should also realize that I have never worked for a company for three consecutive years. In fact, I've only worked for one company more than one year. That's because programmers are the mercenary hired gun of the third millennium. We develop an application and when you are finished, they reduce the development staff in half and thirds. Eventually, I move on.

The problem with moving from one company to another is that you have to start all over each time. You are not legally allowed to move code from company to company, unless managements concur, which rarely happens.

I once worked for the brother of my previous employer. I thought that maybe this once, I'd be able to move some code from one company to the next. Well, the two brothers couldn't make the ends meet and I was told it was not possible.

So, we developers are doomed to start all over in each project. Start form zero. All that code we wrote over the years is just not going to help. Cause we have to write it again. Or do we?

What if, I wrapped all this reusable code into a public library and made it available to everybody. Then surely, no employer could stop me from using this code at my next employ, and the next after that.

This is the intent of the article. I'm putting together a reusable library that extends above what is already available in the standard C run-time libraries and Standard Template Libraries in the current mix. Let me call it the kbcafe library. Why kbcafe? Cause I own the kbcafe.com domain.

I have named this library kbcafe.lib and it is available at the following URL http://ca.msnusers.com/KBCafecom/files.msnw?fc_p=%2Fkbcafe.lib&fc_a=0. The basics of this library are the threading classes.

- consumer
- criticalsection
- event
- monitor
- mutex
- semaphore
- thread

I will describe and show how to use each of these seven classes in this document.

thread

The first class is the basis of all the multithreading that I do in my programming efforts. The class is called kbcafe::thread and it implements the basic functionality of a Win32 thread. I have also added some ifdefs to the code where in the future I might find the time to convert this class to Solaris or other UNIX operating systems.

Listing Error! Bookmark not defined.: thread.h

```
#ifndef KBCAFE_THREAD_H
#define KBCAFE_THREAD_H

#ifdef _WIN32
#ifdef _WINDOWS_
#include <windows.h>
#endif
#else
#error _WIN32 must be defined before you include thread.h
#endif

namespace kbcafe
{

class thread
{
#ifdef _WIN32
static DWORD WINAPI ThreadFunc(LPVOID pv)
{
try
{
(reinterpret_cast<thread *>(pv))->run();
}
catch(...)
{
}
return 0;
}
#endif
#ifdef __sun
#else
#endif

public:
```

```

#ifdef _WIN32
    typedef DWORD threadid;
#elif defined(__sun)
#else
#endif

    thread()
    {
    }

    virtual ~thread()
    {
    }

static threadid getthreadid()
{
#ifdef _WIN32
    return ::GetCurrentThreadId();
#elif defined(__sun)
#else
#endif
}

    static void sleep(long milliseconds=1)
    {
#ifdef _WIN32
        ::Sleep(milliseconds);
#elif defined(__sun)
#else
#endif
    }

    threadid start()
    {
        threadid id;
#ifdef _WIN32
        ::CreateThread(NULL, 0, ThreadFunc, this, 0, &id);
#elif defined(__sun)
#else
#endif
        return id;
    }

    virtual void run()=0;
};

};

#endif

```

The class has one public non-virtual member method, one public abstract member method and two public static member methods. The public non-virtual member `start()` is called when you want to start the new thread. The code in the `start()` method is quite simple. I wrap the `this`-pointer into a `void *` parameter and call `CreateThread` passing the name of my private `ThreadFunc` and passing the `void *` parameter thru the `CreateThread` call and into the `ThreadFunc` static method. The `ThreadFunc` static method in turn, unpackages the object pointer and calls the `run` abstract method.

The public abstract member `run()` is overridden in derived classes to implement the functionality of the new thread. The public static member `getthreadid()`, retrieves the thread ID of the current executing thread. This is not the thread ID of the thread being encapsulated by the class. The public static member `sleep()` puts the current thread into sleep mode for a said amount of milliseconds. Again, this is not the thread being encapsulated by the class, but rather the current executing thread.

Using the class involves instantiating an derived instance of the kbcafe::thread class and calling the start method. The start method invokes and runs the new thread. When the thread starts the run method is called. When the run method completes, the thread exits. A simple example follows.

Listing Error! Bookmark not defined.: thread sample

```
#include <iostream>
#include "thread.h"

class MyThread : public kbcafe::thread
{
public:
    virtual void run()
    {
        std::cout << "Hello" << std::endl;
    };
};

int main(int argc, char* argv[])
{
    MyThread thread;
    thread.start();
    ::Sleep(1000);
    return 0;
}
```

This threading class is likely a little lighter than you may be use to. The typical heavyweight threading class available in most libraries is likely too complex for general re-use. I was thinking about extending the class, but I haven't found any extensions worth preserving yet. Buy I have extended the class on many occasions to deal with threading issues beyond the norm.

One such extension I have considered is to stop the thread when the destructor of the thread object is called. I have yet found a perfect way to do this. It is for this reason that the sample above calls the Sleep function after starting the thread. Failing to call Sleep could cause the thread class to fall out of scope before the run method is invoked.

criticalsection

Once you have the capability of executing multiple threads in your application, the concern is then raised on how to synchronize access to critical resources available to more than one thread. The following code exemplifies the synchronization requirements of applications with more than one thread.

Listing Error! Bookmark not defined.: Lack of Synchronization

```
#include <iostream>
#include "thread.h"

class MyThread : public kbcafe::thread
{
public:
    virtual void run()
    {
        for (int i=0;i<100;i++)
            std::cout << "Hello" << std::endl;
    };
};

int main(int argc, char* argv[])
{
    MyThread thread;
```

```
    thread.start();
    for (int i=0;i<100;i++)
        std::cout << "Hello" << std::endl;
    return 0;
}
```

If you run the previous code, then you'll generate some unexpected output on some lines.

Listing Error! Bookmark not defined.: Output

```
Hello
lo
Hello
Hello
HelHello
Hello
```

Because there is no synchronization to the cout object, the output will eventually get corrupted as out thread is swapped out and another thread continues execution midway thru the results of the other thread.

In order to synchronize the output of the two threads, you have to use a synchronization object to control access to the object. The most basic of such synchronization objects is the critical section. The critical section only allows one thread to access a resource or section of code. My critical section class follows.

Listing Error! Bookmark not defined.: criticalsection.h

```
#ifndef KBCAFE_CRITICALSECTION_H
#define KBCAFE_CRITICALSECTION_H

#ifdef _WIN32
#ifdef _WINDOWS_
#include <windows.h>
#endif
#else
#error _WIN32 must be defined before you include criticalsection.h
#endif

namespace kbcafe
{

class criticalsection
{

#ifdef _WIN32
    typedef CRITICAL_SECTION cs;
#elif defined(__sun)
#else
#endif
    cs m_cs;

public:
    criticalsection()
    {
#ifdef _WIN32
        ::InitializeCriticalSection(&m_cs);
#elif defined(__sun)
#else
#endif
    }

    ~criticalsection()
    {
#ifdef _WIN32
        ::DeleteCriticalSection(&m_cs);
#elif defined(__sun)
#else
#endif
    }
};
};
```

```

#endif
}

    void enter()
    {
#ifdef _WIN32
        ::EnterCriticalSection(&m_cs);
#elif defined(__sun)
#else
#endif
    }

    void leave()
    {
#ifdef _WIN32
        ::LeaveCriticalSection(&m_cs);
#elif defined(__sun)
#else
#endif
    }
};

};

#endif

```

The critical section class wraps a few Win32 API functions in order to implement a critical section. When a thread calls the enter method, it becomes the only thread which can access the critical section until it calls the leave method.

The following code section shows the same code as listing 3, but with synchronization code that prevents simultaneous access to the cout object.

Listing Error! Bookmark not defined.: Synchronized

```

#include <iostream>
#include "thread.h"
#include "criticalsection.h"

kbcafe::criticalsection cs;

class MyThread : public kbcafe::thread
{
public:
    virtual void run()
    {
        for (int i=0;i<100;i++)
        {
            cs.enter();
            std::cout << "Hello" << std::endl;
            cs.leave();
        }
    };
};

int main(int argc, char* argv[])
{
    MyThread thread;
    thread.start();
    for (int i=0;i<100;i++)
    {
        cs.enter();
        std::cout << "Hello" << std::endl;
        cs.leave();
    }
    return 0;
}

```

Conclusion

I hope this thread and thread synchronization code helps you. In future articles, I hope to talk about the various other threading classes available in my library. See you then.

About the Author

Randy Charles Morin is the Lead Architect of SportMarkets Development from Toronto, Ontario, Canada and lives with his wife and two kids in Brampton, Ontario. He is the author of the www.kbcafe.com website, author of Wiley's Programming Windows Services book and co-author of many other programming books.