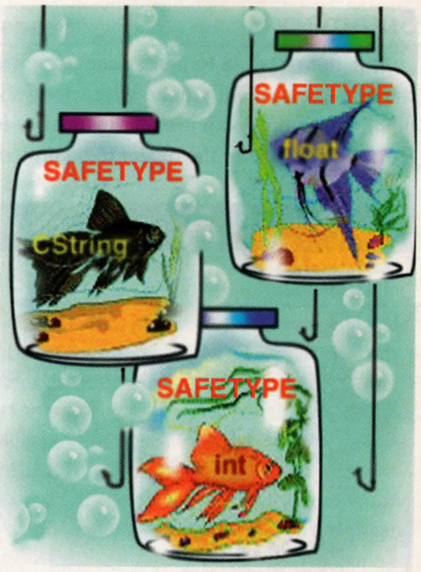
Juval Lowy

Making Primitive Objects Thread Safe

All sorts of things need thread locks. A fairly simple template or two can do the job.



Multithreading can bring many advantages to an application: the user interface can remain responsive while processing goes on in the background; the application can take advantages of multiple CPUs, serve multiple clients, and prioritize tasks. But these benefits come with a price — you have to worry about synchronization issues, deadlocks, reentrancy, and managing the state of objects that are being accessed on multiple threads.

The non-OO way to protect resources, is to have a collection of locks (semaphores or critical section objects), that a thread must acquire before accessing the resource. If the resource is being accessed by another thread, the thread trying to acquire the lock will block. This solution is very error prone. It relies on developers having the necessary discipline, it couples the clients of the resources to the synchronization requirements and proper usage (e.g., what lock goes with what object), and it could result in deadlock. Changing the sychronization mechanism is difficult.

The object-oriented solution to synchronization problems is conceptually simple — make the resource or the object itself thread safe by encapsulating a lock inside the object it is supposed to protect. For example, if you have a member variable in your class that is a linked list, and multiple clients on multiple threads can access your object, you encapsulated the list lock inside the list. The list has a synchronization lock inside it as a data member. Every time you access the list API, say via function `AddHead`, the method implementation tries to acquire the internal lock. If it is available, the `AddHead` method locks the object, accesses the list pointer, adds a new element to the head, and releases the lock. If the lock is not available, the call blocks until it is.

However, suppose you have member variables in your class that are primitive types, such as `int` or `float`. In C++ you cannot extend primitive types by inheritance. You will end up in the scope of your class managing locks for those primitive members variables and be susceptible to the same problems as if you were programming in C!

The solution I've come up with is a template class called `SAFETYPE`. Figure 1 shows portions of the header file that defines `SAFETYPE`. The complete source files (`safetype.h`, `safetype.inl`, and `safetype.cpp`) are available on the *CUJ* ftp site (see p. 3 for downloading instructions). I also make the files `safelist.h` and `safelist.inl` available, as an example of the classic OO way of making an object thread safe. `SAFETYPE` provides synchronization at the atomic level, both for primitive types and for complex classes such as `CString`. `SAFETYPE` implements almost all of the C operators. It has only two member variables — an object of type `T` (the type specified by the template parameter), and a lock (a Win32 mutex).

The implementation of a given operator is straightforward. It locks the object, executes the "real" operator that is defined for the `T` type, and unlocks the object. Figure 2 shows an example. The `SingleLock` method shown simply calls the Win32 function `WaitForSingleObject`. If the operator is a binary, it calls a method `DoubleLock`, which in turn calls `WaitForMultipleObjects`.

I have implemented `SAFETYPE` on Windows, but you should be able to port it easily to other platforms by providing your own implementation for the `SingleLock/DoubleLock` methods using the target platform's synchronization support.

Using SAFETYPEs

Using `SAFETYPE` is just like using the original type, assuming the original type has all the operators that you want to call defined. Just declare the `SAFETYPE` variable and use it:

```
SAFETYPE<int> nValue;
int nBug;

nValue +=3; //This is thread safe
nBug   +=3; //This is not thread safe
```

By making all your class data members thread safe, you can automatically eliminate synchronization problems. This is a much more simplified programing model than the non-OO way presented above. Figure 3 shows an example of using `SAFETYPE` in a C++ object that is being accessed by multiple threads. Note that all the data members are safe.

Juval Lowy is a software architecture manager at KLA-Tencor, a fortune 500 company. Juval also manages a program for reusable software components across the corporation. He conducts classes on Object-Oriented Design, Win32 multithreading, COM, and advanced COM. He can be reached at `juval.lowy@kla-tencor.com`.

Summary

It is not possible to make access to primitive types thread safe the way it is typically done with user-defined types in object-oriented programs. With user-defined types it is fairly easy to encapsulate a lock within the object to be protected. This is not possible with primitive types. The SAFETYPE template class provides a convenient way to make access to primitive objects safe in a multithreading environment. You can use SAFETYPE on complex user-defined types as well. ❑

Figure 1: ***The SAFETYPE template class***

```
template<class T>
class SAFETYPE
{
   public:
      SAFETYPE();
      SAFETYPE(const SAFETYPE<T>& SAFETYPEsrc);
      SAFETYPE(const T& t);

      // member methods for atomic lock/unlock
      void SingleLock()const;
      void SingleUnlock()const;

      // operators
      operator T() const;
      const SAFETYPE<T>&
      operator=(const SAFETYPE<T>& SAFETYPEsrc);
      const SAFETYPE<T>&
      operator+=(const SAFETYPE<T>& SAFETYPEsrc);
      const SAFETYPE<T>&
      operator-=(const SAFETYPE<T>& SAFETYPEsrc);

      /* the rest of the C operators ... */

   protected:
      HANDLE m_hMutex; // access protection
      T m_t;           // The data
};
```

— End of Figure —

Figure 2: ***Implementation of the SAFETYPE operator- for unary negation***

```
template<class T>
const SAFETYPE<T> SAFETYPE<T>::operator-()
{
   T t;
   SingleLock();
   t = -m_t;
   SingleUnlock();
   return t;
}
```

— End of Figure —

Figure 3: ***Using SAFETYPE in a C++ class***

```
class CDog
{
   public:
      void Fetch();
      void Bark();
      BOOL HasShots();
   protected:
      SAFETYPE<CString> m_sName;
      SAFETYPE<BOOL>    m_bHasShots;
};
```

— End of Figure —

CodeMapper

It's never been so easy before!

For Windows ™ 95/98/NT

- C/C++ source code analyzing and automatic documentation building and export (RTF)
- Class hierarchy configurable scheme and image export
- Class browser for fast code navigation
- Project management for complicated file structures
- Automatic code change recognition
- Supports ANSI C/C++
- User-definable for non-ANSI C/C++
- Ready to use with MS Visual C++ ™, Borland C++ Builder ™, UNIX C++
- User templates support

$218

E-mail now to **orbitech@techno-link.com** *for your FREE evaluation copy*

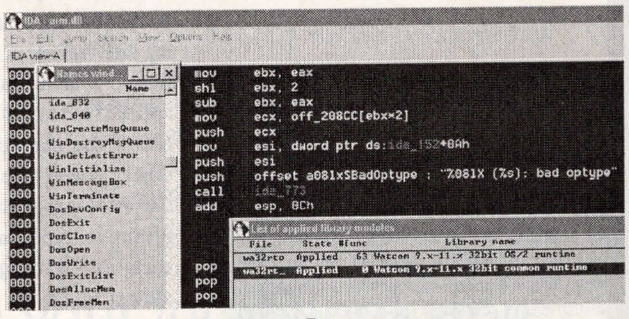
OrbiTech® System Graphics
Making the world visible

Product info:
Phone: (+359 2) 971-36-26
Fax: (+359 2) 971-14-89
e-mail: orbitech@techno-link.com

IDA Pro Interactive Disassembler

Y2K Problem ? Lost Source ? time CAN be reversed !

Intel 80x86, AVR, Z80, Z380, PIC, 8051, 860, H8, 6502, 680x, 68K, Mips, SH-3, SH-4, PPC, TMS320C6x....



```
mov     ebx, eax
shl     ebx, 2
sub     ebx, eax
mov     ecx, off_208CC[ebx*2]
push    ecx
mov     esi, dword ptr ds:ida_152+8Ah
push    esi
push    offset a081xSBadOptype ; "%081X (%s): bad optype"
call    ida_773
add     esp, 8Ch
pop
pop
pop
```

```
ida_832
ida_840
WinCreateMsgQueue
WinDestroyMsgQueue
WinGetLastError
WinInitialize
WinMessageBox
WinTerminate
DosDevConfig
DosExit
DosClose
DosOpen
DosWrite
DosExitList
DosAllocMem
DosFreeMem
```

```
File     State   #func  Library name
wa32rto  Applied    63  Watcom 9.x-11.x 32bit OS/2 runtime
wa32rt_  Applied     0  Watcom 9.x-11.x 32bit common runtime
```

Download our Demo

www.datarescue.com