



Applied Intelligence
6324 Southwood Ave
Suite 1W
Clayton, MO 63105
(314) 863-4770

A Trick for Integrating Threads with Template Classes

An Applied Intelligence White Paper

October 3, 2000

Charles R. Calkins

© Applied Intelligence, Inc., 2000

Abstract

This paper, printed in the May 2000 issue of C/C++ User's Journal, describes one method of encapsulating threads in C++ classes when parameterized templates are involved. While the example code is geared for Linux and the pthreads thread library, minor modifications allow this technique to be used in other environments, such as the Win32 platform.

Keywords

- pthreads
- templates
- Linux
- C++

Table of Contents

| | |
|--|---|
| EXECUTIVE SUMMARY..... | 1 |
| A TRICK FOR INTEGRATING THREADS WITH TEMPLATE CLASSES..... | 1 |
| FIGURE 1..... | 3 |
| FIGURE 2..... | 3 |
| FIGURE 3..... | 4 |

Executive Summary

This paper, printed in the May 2000 issue of C/C++ User's Journal, describes one method of encapsulating threads in C++ classes when parameterized templates are involved. While the example code is geared for Linux and the `pthread` thread library, minor modifications allow this technique to be used in other environments, such as the Win32 platform.

A Trick for Integrating Threads with Template Classes

Developing software for modern systems can be a complex undertaking. Software systems are large, consist of many components, and are often multithreaded, especially as multiprocessor systems become more and more common. To help manage this complexity, object oriented strategies, and object oriented languages such as C++ have been developed.

C++ has no language constructs for threading, so developers must use system libraries such as `pthread` on UNIX-like systems or functions such as `CreateThread()` as part of the Win32 API. These libraries and functions, however, are not class-based but contain C-style mechanisms. In order to more easily apply an object oriented design methodology, it is desirable to treat threads as objects; methods of encapsulating thread API functions is needed. Once encapsulated, active objects containing threads can be used in a similar way as passive objects in the system.

One such encapsulation is given by Figure 1, but code that would be needed in a production system such as thread control or error checking is not present as it would only complicate this discussion.

Class `Thread` contains a public `Create()`, a protected `Run()`, and a destructor. `Create()` calls the `pthread` function `pthread_create()` to create a new thread in the joinable state and start it running by invoking the function `thread_func()`. A pointer to the `Thread` object is the last argument to `pthread_create()`, and it is passed as the pointer argument to `thread_func()`. A thread in the joinable state allows it to be synchronized with later via `pthread_join()` as done here in the destructor, though a thread may be created (or later changed to be) in the detached state which does not require a join to be performed.

The thread function is what provides the means for the thread encapsulation. Instead of containing the thread behavior, it makes use of the object pointer to call `Thread`'s `Run()` method. This allows the behavior of the thread to remain in the thread object and not separated. Here, `Run()` is implemented by class `Thread`, though it could also be made virtual and implemented by subclasses of `Thread` if only one thread in the object is desired. If the object requires multiple threads, it may be more convenient to not use `Thread` as an abstract base class but instead directly implement multiple `Run()`-style methods as needed.

It is important to note that the thread function must be a C-style function and not a class method, hence the need for the roundabout way of calling `Run()`. To reduce namespace pollution, `thread_func()` could be a static function in class `Thread`, but the same problem remains - it cannot be a method.

This structure is fine for a basic thread class, but suppose the class requires a template parameter - just such a case occurred when the author developed a particular object for a code library for his company. A problem now arises as the encapsulation above no longer works as shown in Figure 2.

In the non-template case, the thread function uses the passed "this" pointer of the object to call its `Run()` method. As this pointer is received as a `void *`, it is safely cast to a `Thread *` and the call to `Run()` made. In the template case, it must be cast to a `Thread<T> *` and this is where the problem lies. The thread function must now become a template function itself (as it

needs a generic T), but as stated before, the thread function must be a C-style function to be compatible with the thread library.

One method to overcome this problem is to use an intermediate object to wrap the Thread pointer, and then to rely on the behavior of C++ as shown in Figure 3.

The Thread pointer wrapper is composed of two parts, a base class and a subclass. The base class Wrapper contains a single method `Wrap()` and a virtual destructor. `Wrap()` is pure virtual, and this is the basis of the "trick."

`WrapperSub` inherits from `Wrapper`, but unlike `Wrapper`, `WrapperSub` is a template class and it has the same template parameters as does `Thread`. The purpose of `WrapperSub<T>` is twofold. It first stores `Thread`'s "this" pointer when it is constructed for later use - as it has the same template parameters, storing a `Thread<T> *` can be done without difficulty. It also provides a means to call `Thread`'s `Run()` method. That is, `WrapperSub<T>` implements a function `Wrap()` which now has the same code as `thread_func()` had previously - it uses the stored "this" pointer to call `Thread`'s `Run()` method. It is important to recognize that `Wrap()` in class `WrapperSub<T>`, even though `WrapperSub<T>` is a template class, still overloads `Wrap()` from its non-template base class `Wrapper`.

The call to `pthread_create()` is modified to not pass "this" directly, but instead a dynamic instantiation of `WrapperSub<T>`. Keep in mind that `WrapperSub<T>` stores the "this" pointer of `Thread`.

The thread function has also changed. The pointer received is to an instance of class `WrapperSub<T>`, but the pointer is safely cast to one pointing to class `Wrapper`, syntactically eliminating the template parameter and allowing `thread_func()` to remain a C-style function. The function `Wrap()` is called from that pointer, but as `WrapperSub<T>`'s `Wrap()` function overrode `Wrapper`'s `Wrap()` function, it is `Wrap()` of `WrapperSub<T>` that is called. This function then uses the stored "this" pointer and calls `Run()` of `Thread<T>` as was desired. When `Run()` returns, `Wrap()` returns, and the instance of `WrapperSub<T>` is freed.

Although slightly cumbersome, this method does solve the template problem and relies on the language features of C++ to do it. This trick is also applicable to other C-based threading libraries. For instance, eliminating `pthread_join()`, replacing `pthread_create()` by `_beginthread()` and changing the return type of `thread_func()` to a void instead of `void *` allows the same structure to encapsulate Win32 API thread creation. I'm certain this isn't the only method which can address the problem, and the author would be interested in learning of others.

ADDENDUM

Comments received from C/C++ User's Journal readers to the above article fell into two main categories. Some noted that a member function of a template class can be made static and used as an argument to a library thread creation function. One other commented that library functions that accept function pointers as parameters require those pointers to have C linkage, and making a member function static (nor providing a separate function outside of a class, but not marking it with C linkage) doesn't fit the bill. The tested compilers (Visual C++ 6.0 and egcs 2.95), however, did not have a problem with static member functions or regular functions with C++ linkage as thread creation function parameters. The method presented in this paper skirts the issue, although `thread_func()` should be marked to have C linkage and not C++ linkage as the sample programs indicate.

FIGURE 1

```
#include <pthread.h>

class Thread {
protected:
    pthread_t _threadptr;
    void Run() {}
public:
    void Create() { pthread_create(&_threadptr, NULL, thread_func, this); }
    ~Thread() { pthread_join(_threadptr, NULL); }
    friend void *thread_func(void *);
};

void *thread_func(void *p) {
    Thread *t=reinterpret_cast<Thread *>(p);
    if (t)
        t->Run();
    return 0;
}

int main() {
    Thread t;
    t.Create();
    return 0;
}
```

FIGURE 2

```
#include <pthread.h>

template <class T>
class Thread {
protected:
    pthread_t _threadptr;
    void Run() {}
public:
    void Create() { pthread_create(&_threadptr, NULL, thread_func, this); }
    ~Thread() { pthread_join(_threadptr, NULL); }
    friend void *thread_func(void *);
};

// error - can't make the thread function a template function
template <class T>
void *thread_func(void *p) {
    Thread<T> *t=reinterpret_cast<Thread<T> *>(p);
    if (t)
        t->Run();
}

main() {
    Thread<int> t;
    t.Create();
    return 0;
}
```

FIGURE 3

```
#include <pthread.h>

template <class T>
class WrapperSub;

template <class T>
class Thread {
protected:
    pthread_t _threadptr;
    void Run() { }
public:
    void Create() { pthread_create(&_threadptr, NULL, thread_func, new
        WrapperSub<T>(this)); }
    ~Thread() { pthread_join(_threadptr, NULL); }
    friend void *thread_func(void *);
    friend class WrapperSub<T>;
};

class Wrapper {
public:
    virtual void Wrap()=0;
    virtual ~Wrapper();
};

template <class T>
class WrapperSub : public Wrapper {
protected:
    Thread<T> *_t;
public:
    WrapperSub(Thread<T> *t) : _t(t) {}
    void Wrap() {
        _t->Run();
    }
};

void *thread_func(void *p) {
    Wrapper *w=reinterpret_cast<Wrapper *>(p);
    if (w) {
        w->Wrap();
        delete w;
    }
}

int main() {
    Thread<int> t;
    t.Create();
    return 0;
}
```