

1.6.2 C++ Class *Thread* for Pthreads

Listing 1.7 shows C++ classes *Runnable* and *Thread* for Pthreads.

- Method *run()* can return a value.
- A call to *T.join()* blocks the caller until thread T's *run()* method completes. Method *join()* returns the value that was returned by *run()*.
- Class *Runnable* simulates Java's *Runnable* interface.
- C++ *Threads* can be created on the heap or on the stack. (Java *Thread* objects, like other Java objects, are never created on the stack.)
- Java has a built-in *join()* operation that is useful in Java when one thread needs to make sure that other threads have completed before, say, accessing their results. Java's *run()* method cannot return a value so results must be obtained some other way.

Listing 1.7 shows C++ classes *Runnable* and *Thread* for Pthreads.

The program in Listing 1.6 illustrates the use of C++ classes *Thread* and *Runnable*. It is designed to look like the Java programs in Listings 1-1 and 1-2.

```

class Runnable {
public:
    virtual void* run() = 0;
    virtual ~Runnable() = 0;
};
Runnable::~~Runnable() {} // a function body is required for pure virtual destructors

class Thread {
public:
    Thread(auto_ptr<Runnable> runnable_, bool isDetached = false);
    Thread(bool isDetached = false);
    virtual ~Thread();
    void start(); void* join();
private:
    pthread_t PthreadThreadID; // thread ID
    bool detached; // true if thread created in detached state; false otherwise
    pthread_attr_t threadAttribute;
    auto_ptr<Runnable> runnable;
    Thread(const Thread&);
    const Thread& operator= (const Thread&);
    void setCompleted();
    void* result; // stores return value of run()
    virtual void* run() {}
    static void* startThreadRunnable(void* pVoid);
    static void* startThread(void* pVoid);
    void PrintError(char* msg, int status, char* fileName, int lineNumber);
};

```

Listing 1.7 C++/Pthreads classes *Runnable* and *Thread*.

```

class simpleRunnable: public Runnable {
public:
    simpleRunnable(int ID) : myID(ID) {}
    virtual void* run() {
        std::cout << "Thread " << myID << " is running" << std::endl;
        return reinterpret_cast<void*>(myID);
    }
private:
    int myID;
};

class simpleThread: public Thread {
public:
    simpleThread (int ID) : myID(ID) {}
    virtual void* run() {
        std::cout << "Thread " << myID << " is running" << std::endl;
        return reinterpret_cast<void*>(myID);
    }
private:
    int myID;
};

int main() {
    std::auto_ptr<Runnable> r(new simpleRunnable(1));
    std::auto_ptr<Thread> thread1(new Thread(r));
    thread1->start();
    std::auto_ptr<simpleThread> thread2(new simpleThread(2));
    thread2->start();
    simpleThread thread3(3);
    thread3.start();
    // thread1 and thread2 are created on the heap; thread3 is created on the stack

    int result1 = reinterpret_cast<int>(thread1->join()); // wait for the threads to finish
    int result2 = reinterpret_cast<int>(thread2->join());
    int result3 = reinterpret_cast<int>(thread3.join());

    std::cout << result1 << ' ' << result2 << ' ' << result3 << std::endl;
    return 0;
    // the destructors for thread1 and thread2 will automatically delete the
    // pointed-at thread objects
}

```

Listing 1.6 Using C++ classes *Runnable* and *Thread*.

```

Thread::Thread(auto_ptr<Runnable> runnable_, bool isDetached) :
    runnable(runnable_),detached(isDetached){
    if (runnable.get() == NULL) {
        std::cout << "Thread::Thread(auto_ptr<Runnable> runnable_,
            bool isDetached) failed at " << ' ' << __FILE__ << ":"
            << __LINE__ << " - " << " runnable is NULL " << std::endl; exit(-1);
    }
}

```

```

Thread::Thread(bool isDetached) : runnable(NULL), detached(isDetached){ }
void* Thread::startThreadRunnable(void* pVoid){
// thread start function when a Runnable is involved
    Thread* runnableThread = static_cast<Thread*> (pVoid);
    assert(runnableThread);
    runnableThread->result = runnableThread->runnable->run();
    runnableThread->setCompleted();
    return runnableThread->result;
}

```

```

void* Thread::startThread(void* pVoid) {
    // thread start function when no Runnable is involved
    Thread* aThread = static_cast<Thread*> (pVoid);
    assert(aThread);
    aThread->result = aThread->run(); aThread->setCompleted();
    return aThread->result;
}

```

```

Thread::~~Thread() { }

```

```

void* Thread::join() {
    int status = pthread_join(PthreadThreadID,NULL);
    // result was already saved by thread start functions
    if (status != 0) { PrintError(...); exit(status);}
}

```

```

void Thread::setCompleted() {/* completion was handled by pthread_join() */}

```

```

void Thread::PrintError(char* msg, int status, char* fileName, int lineNumber) {
/*see Listing 1.4 */}

```

Listing 1.7 (cont.) C++/Pthreads classes *Runnable* and *Thread*.

```

void Thread::start() {
    int status = pthread_attr_init(&threadAttribute); // initialize attribute object
    if (status != 0) { PrintError(...); exit(status); }
    status = pthread_attr_setscope(&threadAttribute,
    PTHREAD_SCOPE_SYSTEM);
    if (status != 0) { PrintError(...); exit(status); }
    if (!detached) {
        if (runnable.get() == NULL) {
            int status = pthread_create(&PthreadThreadID,&threadAttribute,
            Thread::startThread,(void*) this);
            if (status != 0) { PrintError( ... ); exit(status); }
        }
        else {
            int status = pthread_create(&PthreadThreadID,&threadAttribute,
            Thread::startThreadRunnable, (void*)this);
            if (status != 0) {PrintError(...); exit(status);}
        }
    }
    else {
        // set the detachstate attribute to detached
        status = pthread_attr_setdetachstate(&threadAttribute,
        PTHREAD_CREATE_DETACHED);
        if (status != 0){
            PrintError(...);exit(status);
        }
        if (runnable.get() == NULL) {
            status = pthread_create(&PthreadThreadID,&threadAttribute,
            Thread::startThread, (void*) this);
            if (status != 0) {PrintError(...);exit(status);}
        }
        else {
            status = pthread_create(&PthreadThreadID,&threadAttribute,
            Thread::startThreadRunnable, (void*) this);
            if (status != 0) {PrintError(...); exit(status);}
        }
    }
    status = pthread_attr_destroy(&threadAttribute);
    if (status != 0) { PrintError(...); exit(status); }
}

```

Listing 1.7 (cont.) C++/Pthreads classes *Runnable* and *Thread*.

```
pthread_create(&PthreadThreadID,&threadAttribute,  
             Thread::startThread,(void*) this);
```

When method *start()* is called, function *pthread_create()* is called with the following arguments:

- *&PthreadThreadID*: the address of the memory location that will receive an identifier assigned to the thread if creation is successful.
- *&threadAttribute*: This address of the thread attribute variable
- The third argument is either *Thread::startThread()* or *Thread::startThreadRunnable()*. Method *startThread()* is the startup method for threads created by inheriting from class *Thread*. Method *startThreadRunnable()* is the startup method for threads created from *Runnable* objects.
- *(void*) this*: The fourth argument is a pointer to this *Thread* object, which is passed through to method *startThread()* or *startThreadRunnable()*. Thus, all threads execute one of the startup methods, but the startup methods receive a different *Thread* pointer each time they are executed.

Method *startThread()* casts its *void** pointer parameter to *Thread** and then calls the *run()* method of its *Thread** parameter.

- When the *run()* method returns, *startThread()* calls *setCompleted()* to set the thread's status to completed and to notify any threads waiting in *join()* that the thread has completed.
- The return value of the *run()* method is saved so that it can be retrieved in method *join()*.

Static method *startThreadRunnable()* performs similar steps when threads are created from *Runnable* objects. Method *startThreadRunnable()* calls the *run()* method of the *Runnable* object held by its *Thread** parameter and then calls *setCompleted()*.

In Listing 1.6, we use *auto_ptr<>* objects to manage the destruction of two of the threads and the *Runnable* object *r*.

- When `auto_ptr<>` objects *thread1* and *thread2* are automatically destroyed at the end of the program, their destructors will automatically invoke `delete` on the pointers with which they were initialized.
- This is true no matter whether the *main* function exits normally or by means of an exception.

Note that startup functions *startThreadRunnable()* and *startThread()* are static member functions.

Note also that calls to method *join()* are simply passed through to method *pthread_join()*.