1.6.1 C++ Class *Thread* for Win32

Listing 1.5 shows C++ classes *Runnable* and *Thread* for Win32:

- 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.

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() {} // function body required for pure virtual destructors

class Thread {
public:
  Thread(std::auto_ptr<Runnable> runnable_);  
  Thread();
  virtual ~Thread();
  void start();  // starts a suspended thread
  void* join();  // wait for thread to complete

private:
  HANDLE hThread;
  unsigned winThreadId;  // Win32 thread ID
  std::auto_ptr<Runnable> runnable;
  Thread(const Thread&);
  const Thread& operator=(const Thread&);
  void setCompleted();  // called when run() completes
  void* result;  // stores value returned by run()
  virtual void* run() {return 0; }
  static unsigned WINAPI startThreadRunnable(LPVOID pVoid);
  static unsigned WINAPI startThread(LPVOID pVoid);
  void PrintError(LPTSTR lpszFunction, LPSTR fileName, int lineNumber);
};

Listing 1.5 C++/Win32 classes Runnable and Thread (header files).
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(std::auto_ptr<Runnable> runnable_) : runnable(runnable_) {
    if (runnable.get() == NULL) PrintError(…);
    hThread = (HANDLE)_beginthreadex(NULL,0,Thread::startThreadRunnable,
        (LPVOID)this, CREATE_SUSPENDED, &winThreadID);
    if (!hThread) PrintError("_beginthreadex failed at ",__FILE__,__LINE__);}
}

Thread::Thread() : runnable(NULL) {
    hThread = (HANDLE)_beginthreadex(NULL,0,Thread::startThread,
        (LPVOID)this, CREATE_SUSPENDED, &winThreadID);
    if (!hThread) PrintError(…);
}

unsigned WINAPI Thread::startThreadRunnable(LPVOID pVoid){
    Thread* runnableThread = static_cast<Thread*> (pVoid);
    runnableThread->result = runnableThread->runnable->run();
    runnableThread->setCompleted();
    return reinterpret_cast<unsigned>(runnableThread->result);
}

unsigned WINAPI Thread::startThread(LPVOID pVoid) {
    Thread* aThread = static_cast<Thread*> (pVoid);
    aThread->result = aThread->run(); aThread->setCompleted();
    return reinterpret_cast<unsigned>(aThread->result);
}

Thread::~Thread() {
    if (winThreadID != GetCurrentThreadId()) {
        DWORD rc = CloseHandle(hThread);
        if (!rc) PrintError(…);
    } // note that the runnable object (if any) is automatically deleted by auto_ptr.
}

void Thread::start() {
    assert(hThread != NULL);
    DWORD rc = ResumeThread(hThread);
    // thread was created in suspended state so this starts it running
    if (!rc) PrintError("ResumeThread failed at ",__FILE__,__LINE__);}

void* Thread::join() {
    /* a thread calling T.join() waits until thread T completes; see section 3.7.4. */
    return result; // return the void* value that was returned by method run()
}

void Thread::setCompleted() {
    /* notify any threads that are waiting in join(); see section 3.7.4. */
}

void Thread::PrintError(LPTSTR lpszFunction, LPSTR fileName, int lineNumber)
    { /* see Listing 1.4 */}
Listing 1.5 (continued).
When a C++ Thread is created, the corresponding Thread constructor calls function _beginthreadex() with the following arguments:

- NULL: This is the default value for security attributes.
- 0: This is the default value for stack size.
- 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.
- (LPVOID) 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.
- CREATE_SUSPENDED: A Win32 thread is created to execute the startup method, but this thread is created in suspended mode, so the startup method does not begin executing until method start() is called on the thread.

Since the Win32 thread is created in suspended mode, the thread is not actually started until method Thread::start() is called.

- Method Thread::start() calls Win32 function ResumeThread(), which allows the thread to be scheduled and the startup method to begin execution.
- The startup method is either startThread() or startThreadRunnable(), depending on which Thread constructor was used to create the Thread object.

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.