Multithreaded applications (1)

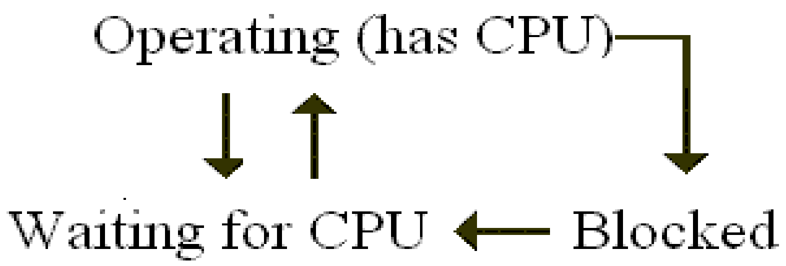
A thread of execution is the smallest sequence of programmed instructions that an operating system can manage independently.

Multitasking: several processes run concurrently (or seem to run concurrently). Between processes the computer resources (like memory, ports, etc.) are not shared, For example each process has its own address space.

Multithreading: several threads run concurrently (or seem to run concurrently) within one process. Several threads can share the same resource, for example a section of memory.

On single processor systems the processor switches from one thread to another. When the time slice allocated for a thread has expired, the processor starts to execute instructions from another thread. As the time slices are short, the user percieves that the threads are running on the same time. On multiprocessor (multicore) systems each processor may run a particular thread, i.e. the threads are actually running on the same time.

A thread may be in three states:



When a thread needs a resource currently occupied by another thread or waits for data from external sources, it is in the blocked state. The other threads continue to work.

Multithreaded applications (2)

When you must create threads:

1. You have some subtasks that take a large amount of time (searching, printing, etc.).
2. You need to communicate with external data sources (web, devices connected to your computer, etc.) which may send data to you or read data you want to send them at occasional moments. It may even happen that they do not want to communicate with you. A single-thread application waiting for data from external source freezes and becomes uncontrolable.
3. You have subtasks with different priority.
4. You have user interface which must be responsive - i.e. react to the user actions (for example mouse clicks) immediately.

main() is in the primary (or main) thread. The main thread may initialize and launch another threads, those in turn can also have their own child threads and so on.

Each thread must have its thread entry point function which corresponds to the *main()* of the primary thread.

Each thread has its own stack for local variables. The global variables are common to all the threads.

STL threads (1)

```
#include <thread> // See http://www.cplusplus.com/reference/thread/
using namespace std;
```

To declare a thread and launch it, write:

```
thread thread_object_name(entry_point_function_name,
                          list_of_input_parameters);
```

The entry point function may have any set of input parameters, but no return value.

Example: suppose we have function

```
void PrintTextFile(string FileName);
```

To organize the printing in background write

```
thread BackgroundPrinting(PrintTextFile, string("c:\\temp\\data.txt"));
```

The thread stops when its entry point function quits. The question is what happens if the thread object is destroyed (automatically as a local variable or by *delete* operator) before the end of entry point function. For example:

```
int main()
{
  thread BackgroundPrinting(PrintTextFile, string("c:\\temp\\data.txt"));
  ……………………………
  return 0; // still printing, but BackgroundPrinting thread object goes out of scope
}
```

STL threads (2)

There are two solutions:

```
void fun()
{
   thread BackgroundPrinting(PrintTextFile, string("c:\\temp\\data.txt"));
   ……………………………
   BackgroundPrinting.join(); // main thread waits until the PrintTextFile returns
   return;
}
void fun()
{
   thread BackgroundPrinting(PrintTextFile, string("c:\\temp\\data.txt"));
   BackgroundPrinting.detach(); // we get a thread that runs independently.
   ……………………………
   return;
}
```

If a thread is detached (called also as daemon thread), it just keeps running in the background until the end of entry point function. Destroying of the thread object does not stop the detached thread. However, when the process dies (i.e. the *main()* returns), it kills all the running detached threads.

If a thread is not detached and we destroy the thread object without applying *join*, we'll get a run-time error.

STL threads (3)

If the entry point function is a member of a class:

```
thread thread_object_name(&class_name::entry_point_function_name,
                                         pointer_to_object, list_of_input_parameters);
```

Example: suppose we have class

```
class Printers
{
  ………………………………….
   PrintTextFile(string FileName);
  ……………………………………
};
```

and object:

```
Printers *pHP_LaserJet = new Printers;
```

then to create and launch the background printing write:

```
thread BackgroundPrinting(&Printers::PrintTextFile, pHP_laserJet,
                                         string("c:\\temp\\data.txt"));
```

If the function launching thread and the entry point function are members of the same class and belong to the same object, *pointer_to_object* is of course pointer *this*.

STL threads (4)

The entry point may be specified by lambda expressions:

```
thread thread_object_name(lambda_expression);
```

Example:

```
const char *pTextFile = "c:\\temp\\data.txt";
thread BackgroundPrinting([&]() { PrintTextFile(pTextFile); });
```

Also, it is possible to write a new class and apply functors. Example:

```
class PrintThread
{
 private: const char *pFile;
 public: PrintThread(const char *p) : pFile(p) { } // parameter is specified in constructor
            operator() () const { PrintTextFile(pFile); }
};
PrintThread pt("c:\\temp\\data.txt"); // create functor object
thread BackgroundPrinting(pt); // functor object presents the entry point function
```

or

```
thread BackgroundPrinting(PrintThread("c:\\temp\\data.txt")); // create nameless functor
```

or with uniform initialization

```
thread BackgroundPrinting { PrintThread("c:\\temp\\data.txt") }; // here we create object
// BackgroundPrinting and initialize it with nameless functor
```

STL threads (5)

Alternative:

```
class PrintThread
{
 public: // no constructor, just operator()
      operator() (const char *p) const { PrintTextFile(p); }
};
thread BackgroundPrinting(PrintThread(), "c:\\temp\\data.txt"); // create nameless object
```

or

```
PrintThread pt;
thread BackgroundPrinting(pt, "c:\\temp\\data.txt"); // use previously defined object
```

Class *this_thread*

Class *this_thread* has several static methods allowing the current thread to control itself.

```
this_thread::sleep_for(duration);
```

blocks the current thread for the specified time interval. Example:

```
this_thread::sleep_for(chrono::seconds(5));
```

```
this_thread::sleep_until(time_point);
```

blocks the current thread until the specified moment. Example:

```
time_t now_t = chrono::system_clock::to_time_t(chrono::system_clock::now());
struct tm now_tm;
localtime_s(&now_tm, &now_t);
now_tm.tm_sec = now_tm.tm_min = 0;
now_tm.tm_hour++;
this_thread::sleep_until(chrono::system_clock::from_time_t(mktime(&now_tm)));
 // for example, if time is now 14:05:00 then the thread resumes its work at 15:00:00
```

Exceptions in threads (1)

Suppose that our main thread has launched another thread. An exception has occurred in this thread but for some reasons the handling of it in the launched thread itself is not possible and should be transferred to the main thread. For solution the problem we need:

- An object of class *exception_ptr*, used for storing the exception. This object is actually a smart pointer. It must be accessible for the launched thread as well as for the main thread.
- Function *current_exception()* creating from ordinary exception an object of class *exception_ptr* (actually storing the exception).
- Function *rethrow_exception()* allowing to transform the object of class *exception_ptr* back to ordinary exception.

See also http://www.cplusplus.com/reference/exception/exception_ptr/

Exceptions in threads (2)

```
int main()
{
  exception_ptr Ex_ptr = nullptr; // smart pointer Ex_ptr will store the exception
  ………………………………….
  thread Task(TaskMain, &Ex_ptr, …….);
  Task.join(); // wait for end of thread Task
  try
 {
      if (Ex_ptr)
     { // was there an exception in the thread?
          rethrow_exception(Ex_ptr); // transform back to normal exception
      }
  }
   catch (const exception& e)
  {   // process the restored exception
       cout << e.what() << endl;
   }
   return 0;
}
```

Exceptions in threads (3)

In thread:

```
void TaskMain(exception_ptr *pEx_ptr, ……)
{
  ……………………..
  try
 {
        ………………….
  }
  catch (exception)
 {
    *pEx_ptr = current_exception(); // instead of handling store the exception
     return;
 }
 ………………………………….
}
```

Race conditions (1)

Mostly, threads share some resources. Suppose we have two threads and they share an integer *static int x = 1;* One of them increments and the other decrements it. Those operations are performed in 3 steps: retrieving the value from memory, incrementing or decrementing and sending back into the memory. As the thread scheduling mechanism can swap between threads at any time, the final result is unpredictable:

Thread 1	Thread 2
Retrieve x (value 1)	
Increment x	
Store x (value 2)	
	Retrieve x (value 2)
	Decrement x
	Store x (value 1)

Thread 1	Thread 2
	Retrieve x (value 1)
	Decrement
	Store x (value 0)
Retrieve x (value 0)	
Increment	
Store x (value 1)	

Thread 1	Thread 2
Retrieve x (value 1)	
	Retrieve x (value 1)
	Decrement
	Store x (value 0)
Increment	
Store x (value 2)	

Thread 1	Thead 2
Retrieve x (value 1)	
Increment	
	Retrieve x (value 1)
	Decrement
Store (value 2)	
	Store (value 0)

Race conditions (2)

From https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks:

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

For us the thread execution is nondeterministic, therefore we cannot control the time or order of execution. It may happen that the "race" does not occur at all (tables 1 and 2). But if it does occur, the "winner" may be any of the threads and therefore the final result is unpredictable (tables 3 and 4).

The solution is to lock the shared memory. It means that when one of the threads is operating with the shared memory field, the other thread(s) must stop and wait for the first one to release the shared memory.

Race conditions (3)

More generally, we have to synchronize the threads:

- When one of the threads owns a resource (array, linked list, disk file, COM port, etc.), the other threads can access this resource for reading only. The have no right to change anything on that resource.
- If a thread needs a resource owned by another thread for writing (i.e. changing data), it must wait until the resource is released.
- When the thread owning a resource does not need it more, it has to release the resource and signal about it to the other threads.

Key problems: communication between threads, locking resources.

The code must be thread-safe.

Mutexes (1)

Let us have two threads:

1. The producer thread retrieves data from an external source or generates data in some other way. If a package of data is ready, it stores it in a buffer. Time interval needed to create a package is occasional.
2. The consumer thread reads the data from the buffer it shares with the producer and processes it and / or views on the screen. Time interval needed to process a package and view it is also occasional.

It may happen that the consumer starts to process data when the producer has not yet finished the storing of new package. It may also happen that the producer starts to store the new package when the consumer has not yet finished the processing of previous package.

Consequently, to synchronize the work of threads we need a mechanism guaranteeing that

1. When the producer is accessing the shared buffer, the consumer has no right to access it. If the consumer tries to access the buffer, it will be blocked until the producer releases the buffer.
2. When the consumer is accessing the shared buffer, the producer has no right to access it. If the producer tries to access the buffer, it will be blocked until the consumer releases the buffer.

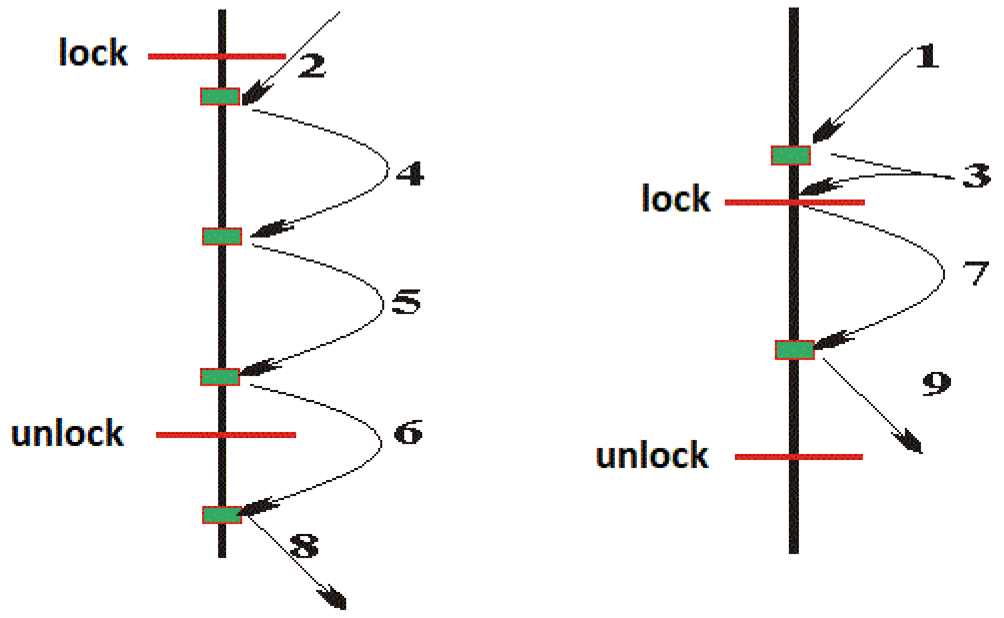
In C++ this mechanism is implemented by mutexes (mutual exclusion classes).

Mutexes (2)

```
#include <mutex> // see http://www.cplusplus.com/reference/mutex/mutex/
mutex mx; // must be accessible in both threads
```

```
// Producer:
mx.lock();
// code in which buffer is accessed
mx.unlock();
```

```
// Consumer:
mx.lock();
// code in which buffer is accessed
mx.unlock();
```



Actually, we do not lock the shared memory field but mark the section of code in which the shared memory is accessed.

If the threads need the shared memory only for reading, mutexes are unnecessary.

Mutexes (3)

```
class Producer
{  // task: create random data on an occasional moment
public:
   vector<int> *pBuf; // shared resource
   mutex *pMx;
   Producer(vector<int> *pv, mutex *pm) : pBuf(pv), pMx(pm) { } // constructor
   void operator() ()
   {
        default_random_engine generator;
        uniform_int_distribution<int> delay_distribution(0, 2000);
        uniform_int_distribution<int> value_distribution(0, 100);
        pMx->lock();
        this_thread::sleep_for(chrono::milliseconds(delay_distribution(generator)));
                 // pause, random duration
        generate(pBuf->begin(), pBuf->end(),
               [&]() { return value_distribution(generator);});
                 // fill the vector with random numbers
        pMx->unlock();
   }
};
```

Mutexes (4)

```
class Consumer
{ // task: print the created data
 public:
   vector<int> *pBuf; // shared resource
   mutex *pMx;
   Consumer(vector<int> *pv, mutex *pm) : pBuf(pv), pMx(pm) { } // constructor
   void operator() ()
   {
     this_thread::sleep_for(chrono::milliseconds(2500)); // see comment on the next slide
     pMx->lock();
     for_each(pBuf->begin(), pBuf->end(), [](int i) { cout << i << ' '; });
     cout << endl;
     pMx->unlock();
    }
};
```

Mutexes (5)

```
int main()
{
  mutex mx;
  vector<int> buf(32);
  thread ProducerThread { Producer(&buf, &mx) }; // uniform initialization, see slide
                                                  // STL Threads (4)
  thread ConsumerThread { Consumer(&buf, &mx) };
      // see slide STL threads (4)
  ProducerThread.join();
  ConsumerThread.join();
  return 0;
}
```

Comment:

The producer starts first but we cannot be sure that it locks the mutex before the consumer applies the locking. If the consumer is faster, it prints zeroes and then allows the producer to work. Therefore we have blocked the consumer for a while. This is not a good way to solve the problem. Later we'll see how to solve it with conditional variables.

Mutexes (6)

Method *try_lock* works in the following way:

- If the mutex is not locked by another thread, it will be locked
- If the mutex is locked by another thread, it returns immediately with *false*.

Usage example:

```
while (true)
{
    if (mx.try_lock())
    {
        .................................. // critical section operations
        mx.unlock();
        break;
    }
    else
    {
        ................................... // do something else, then try once more to enter
    }
}
```

Mutexes (7)

If a thread is in critical section and hangs, the mutex is never unlocked and the other threads cannot continue. The solution is *timed_mutex* that has methods *try_lock_for* and *try_lock_until:*

- If the mutex is not locked by another thread, it will be locked immediately.
- If the mutex is locked by another thread, it tries to lock it until the specified time interval has elapsed or the specified time point reached. If the locking is still impossible, *false* is returned.

Usage example:

```
timed_mutex tmx; // see http://www.cplusplus.com/reference/mutex/timed_mutex/
chrono::milliseconds timeout(1000); // timeout duration
if (tmx.try_lock_for(timeout))
{           // successful locking
            .................................. // critical section operations
            tmx.unlock();
}
else // 1000ms has elapsed but locking is still not possible
{
            cout << " Problems: the other thread has frozen" << endl;
}
```

Timed mutexes can operate as ordinary mutexes, i.e. they support also methods *lock()* and *try_lock()*.

Mutexes (8)

To simplify locking and unlocking with mutexes use class lock_guard:

```
lock_guard<mutex> lock_guard_name(associated_mutex_name);
```

Example:

```
#include <mutex> // see http://www.cplusplus.com/reference/mutex/lock_guard/
mutex mx;
lock_guard<mutex> lock(mx); // constructor for object lock of class lock_guard
```

The *lock_guard* has only two methods: contructor and destructor.

- When the constructor is called, the *lock()* method of the associated mutex is also called.
- When the destructor is applied, the *unlock()* method of the associated mutex is also called.

With manual locking, you have to ensure that the mutex is unlocked correctly on every exit path from the region where you need the mutex locked, including when the region is exited due to an exception. Having a local *lock_guard* object (i.e. auto memory class) it is not a problem: even if an exception is thrown, the destructor is applied and the mutex released. The mutex itself may be a global variable or input parameter of the thread entry point function.

Mutexes (9)

The unique_lock is more flexible:

```
unique_lock<mutex> unique_lock_name(associated_mutex_name);
                    // as lock_guard – the constructor locks the associated mutex.
unique_lock<mutex> unique_lock _name(associated_mutex_name, std::defer_lock);
                    // the constructor does not lock the associated mutex
```

To lock later you may use methods:

- *lock()* as the *lock()* of *mutex*
- *try_lock()* as the *try_lock()* of *mutex*
- *try_lock_for* as the *try_lock_for* of *timed_mutex*
- *try_lock_until* as the *try_lock_until* of *timed_mutex*

The destructor of *unique_lock* unlocks the associated mutex. But you may also unlock with *unlock()* method. Example:

```
mutex mx; // see http://www.cplusplus.com/reference/mutex/unique_lock/
unique_lock<mutex> lock (mx, std::defer_lock);
.....................................................................
lock.lock(); // locks the mutex mx
……………………………..
if (…)
   return; // unlocks mutex mx
..........................
lock.unlock(); // unlocks the mutex mx
```

Mutexes (10)

Some terminology:

```
mutex mx;
...........................................................
mx.lock();      // the thread acquires the mutex
..................... // the thread owns the mutex
mx.unlock();  // the thread releases the mutex

unique_lock<mutex> lock(mx, std::defer_lock);
..................................................................
lock.lock();        // the thread acquires the lock
.........................   // the thread owns the lock
lock.unlock();    // the thread releases the lock
```

Call a function just once (1)

Suppose we have two threads sharing a common but not initialized yet resource. The code of both threads starts with the call to initializing function. For example, one of the threads sends data to an external device, the other receives data and the shared device is a COM port or a TCP/IP socket. It is unknown which of the threads will call the initializer first. But it is clear that the initializer must be called only once.

```
void ::initializer();
                    // initializes the common resource, for simplicity let it to be out of classes
class Thread_1 { …. void operator()() { initializer(); … } …..};
class Thread_2 { …. void operator()() { initializer(); … } …..};
```

The most effective solution to similar problems is to apply the *call_once* mechanism:

1. Define variable of type *once_flag*, it must be accessible for both threads:
   ```
   once_flag flag_name;
   ```
2. In both threads call the initializer in this way:
   ```
   call_once(flag, pointer_to_initializer, list_of_initializer_parameters);
   ```

If the first thread wants to call the initializer when the second thread is currently intializing, it blocks until the end of initializing and then continues without stepping into initializer. If the first thread wants to call the initializer when the second thread has already finished the initializing, it continues immediately. Thus the faster thread performs the initialization and the slower omits this step.

Call a function just once (2)

If you have several functions to be called only once you need also several flags.

The initializer may be a function, class method or functor. About details see http://www.cplusplus.com/reference/mutex/call_once/ .

Example:

```
#include <mutex>
void ::OpenFile(const char *pFilename, fstream **pFile) {……………………….}
class Calculate_1
{
public:
   once_flag *pFlag;
   fstream *pSharedFile;
   Calculate_1(once_flag *p1, fstream *p2) : pFlag(p1), pSharedFile(p2) { }
   void operator() ()
   {
        call_once(*pFlag, OpenFile, "c:\\temp\\data.txt", &pSharedFile);
         ……………
   }
   …………………………………………..
};
// class Calculate_2 is similar
```

Call a function just once (3)

```
int  main()
{
  once_flag flag;
  fstream *pDataFile;
  thread thread_1{ Calculate_1(&flag, pDataFile) };
  thread thread_2{ Calculate_2(&flag, pDataFile) };
  thread_1.join();
  thread_2.join();
  return 0;
}
```

One of the threads (impossible to say which) opens the file

Atomic variables (1)

If a single variable is shared, we may lock the operations with it using mutexes. But it is more efficient to declare them as atomic variables.

An atomic operation is a machine instruction that is always executed without interruption. A sequence of two or more machine instructions isn't atomic since the operating system may suspend the execution of the current sequence of operations in favour of another task. A C++ statement is atomic if the compiler translates it into a single machine instruction. But each hardware architecture may translate the same C++ statement in its own different way. So it is wise to say that none of the C++ is statements is atomic. See also slide *Race conditions (1)*.

An operation with atomic variables of course needs a sequence of machine instructions. But the execution of this sequence is not interrupted. In other words, an operation with atomic variables is locked although we need not to declare any mutexes for them.

```
#include <atomic> // see http://www.cplusplus.com/reference/atomic/
using namespace std;
atomic<int> i(0);
atomic<char> c('A');
atomic<long> l(100000);
.............................. // all the integral types are allowed
```

i, *c*, *l* etc. are objects. Their initialization is compulsory.

Atomic variables (2)

Due to operator functions we may write simply:

```
atomic<int> i = 0;
atomic<char> c = 'A';
atomic<long> l = 100000;
```

Functions declared in atomic classes are interlocked functions. It means that if one of the functions associated with an atomic variable is running and another thread tries to call this or some other function of the same atomic variable, this thread has to wait. In other words, the functions of atomic classes lock and unlock their inner contents automatically. For example:

```
i.store(1);              // or due to operator functions we may write also i = 1;
if (i.load() == 1) ..... // or due to operator functions we may write also if (i == 1)
```

Some examples about the other atomic class functions:

```
i.fetch_add(1);          // or due to operator functions we may write also i++;
i.fetch_add(10);         // or due to operator functions we may write also i+=10;
i.fetch_sub(1);          // or due to operator functions we may write also i--;
```

But:

```
atomic<int> i1 = 10;
atomic<int> i2 = i1; // error
i2 = i1; // error
i2.exchange(i1); // meaning: i2 = i1
```

Atomic variables (3)

Similarly:

```
atomic<int> i1 = 10, i2 = 20, i3 = 0;
i3.exchange(i1 + i2); // i3 = i1 + i2
cout << i3 << endl;
i3.exchange(i1 * i2); // i3 = i1 * i2
```

Atomic pointers are also allowed, for example:

```
atomic<vector<int> *> pv = new vector<int>(10);
atomic<int *> pi = new int[10];
```

Pointer operations are supported with overloaded operators, for example

```
*pi = 10;
*(pi + 1) = 20;
```

Atomic variables (4)

Example: thread control using polling.

```
class Worker
{
 public:
        vector<int> &buf;
        atomic<bool> &stop;
       Worker (vector<int> &v, atomic<bool> &b) : buf(v), stop(b) { }
       void operator() ()
       {
          default_random_engine generator;
          uniform_int_distribution<int> delay_distribution(0, 10000);
          uniform_int_distribution<int> value_distribution(0, 100);
           while (!stop) // worker runs until stop becomes true (polls value of variable stop)
          {
              this_thread::sleep_for(chrono::milliseconds(delay_distribution(generator)));
              generate(buf.begin(), buf.end(),
                                   [&]() { return value_distribution(generator); });
              …………………… // do something with the generated numbers
            }
         }
};
```

Atomic variables (5)

```
class Controller
{
 public:
          atomic<bool> &stop;
           Controller(atomic<bool> &b) : stop(b) { }
           void operator() ()
          {
               _getch(); // the human operator has to press a key to stop worker
               stop = true;
           }
};
int main()
{
          atomic<bool> stop = false;
          vector<int> buf(32);
          thread ControllerThread { Controller(stop) };
          thread WorkerThread { Worker(buf, stop) };
          WorkerThread.join();
          ControllerThread.join();
          return 0;
}
```

Conditional variables (1)

Inter-thread communication in C++ is implemented by conditional variables. Let us have two threads:

1. The producer thread retrieves data from an external source or generates data in some other way. If a package of data is ready, it stores it in a buffer. Time interval needed to create a package is occasional.
2. The consumer thread reads the data from the buffer shared with the producer and processes it and / or views on the screen. Time interval needed to process a package and view it is also occasional. Problem: the processing may start only when the producer has finished its task.

```
#include <condition_variable> // see http://www.cplusplus.com/reference/condition_variable//
mutex mx; // must be accessible in both threads
condition_variable cv; // must be accessible in both threads, cooperates with mutex
```

```
// Producer:
{
  unique_lock<mutex> lock(mx);
  …. // code in which buffer is accessed
  cv.notify_one();
}
```

```
// Consumer:
{
  unique_lock<mutex> lock(mx);
  cv.wait(lock);
  …. // code in which buffer is accessed
}
```

Conditional variables (2)

wait(unique_lock_name);

The *wait* method can be called only when the thread is already locked by *unique_lock* managing the associated mutex. It:

1. Automatically releases the lock (atomically calls *unlock()* of the associated *unique_lock*), thus allowing the other thread to run.
2. Blocks its own thread and starts to wait for a notification.
3. Unblocks its thread when the other thread has emitted the notification.
4. Locks again (before return atomically calls *lock()* of the associated *unique_lock)*.

Notification is emitted by method *notify_one()* called from another thread.

If several threads are waiting, *notify_one()* releases only one of them (impossible to say which). Method *notify_all()* sends notification to all the threads stopped by the current conditional variable. If there are no waiting threads, those functions do nothing.

```
{  // Producer:
  unique_lock<mutex> lock(mx);
  ….. // code in which buffer is accessed
  cv.notify_one();
        // releases consumer

}
```

```
{ // Consumer:
  unique_lock<mutex> lock(mx);
  cv.wait(lock);
        // blocks consumer and releases
        // the lock, producer can start
  …. // code in which buffer is accessed
}
```

Conditional variables (3)

wait(unique_lock_name, predicate);

The predicate may be a pointer to function, lambda expression or functor. It cannot have any arguments and it must return *true* or *false*.

wait() with predicate:

1. If the predicate has returned *true*, does nothing and returns immediately.
2. If the predicate has returned *false*, releases the lock (atomically calls *unlock()* of the associated *unique_lock*), thus allowing the other thread to run.
3. Also, if the predicate has returned *false* blocks its thread and starts to wait for a notification.
4. If the notification has arrived, calls the predicate once more.
5. If the predicate still returns *false*, the thread stays in blocked state.
6. If the predicate returns *true*, unblocks its thread and locks the associated *unique_lock*.

std::notify_all_at_thread_exit(conditional_variable_name, unique_lock_name);

This function not belonging to any classes is mostly called before *join* to release threads that may be have got stuck.

Conditional variables (4)

Example:

```
int main()
{
  vector<int> buf; // empty vector
  mutex mx;
  condition_variable cv;
  thread ProducerThread { Producer (buf, mx, cv, 32) };
  thread ConsumerThread { Consumer (buf, mx, cv) };
  ProducerThread.join();
  ConsumerThread.join();
  return 0;
}
// the order of launching threads is meaningless
```

Conditional variables (5)

```
class Producer
{
public:
    vector<int> &buf;
    mutex &mx;
    condition_variable &cv;
    int buf_size;
    Producer (vector<int> &v, mutex &m, condition_variable &c, int n) :
                    buf(v), mx(m), cv(c), buf_size(n) { }
   void operator() ()
   {
        default_random_engine generator;
        uniform_int_distribution<int> delay (0, 10000);
        uniform_int_distribution<int> random_number(0, 100);
        unique_lock<mutex> lock(mx); // locks the mutex
        buf.resize(buf_size);
        this_thread::sleep_for(chrono::milliseconds(delay(generator)));
        generate(buf.begin(), buf.end(), [&]() { return random_number(generator); });
        cv.notify_one(); // releases the comsumer
   } // variable "lock" deleted, mutex automatically unlocked
};
```

Conditional variables (6)

```
class Consumer
{
public:
   vector<int> &buf;
   mutex &mx;
   condition_variable &cv;
   Consumer(vector<int> &v,  mutex &m, condition_variable &c) :
                  buf(v), mx(m), cv(c) { }
   void operator() ()
   {
          unique_lock<mutex> lock(mx); // locks the mutex
          cv.wait(lock, [&]() { return !buf.empty(); });
                   // If the buffer is empty, unlocks the mutex allowing the producer to work
                   // and starts to wait for the notification from the producer.
                   // The producer sends notification when the buffer is filled. Consequently
                   // predicate then returns true and the thread will be unblocked.
          for_each(buf.begin(), buf.end(), [](int i) { cout << i << ' '; });
          cout << endl;
   }
}; // destructor called, mutex automatically unlocked
```

Conditional variables (7)

Comments:

1. Suppose that the producer steps into the critical section first. In that case the consumer must wait until the producer has left the critical section. i.e. until the end of producer thread. The first step of the consumer is the call to *wait()*. As the buffer is already full, the predicate returns *true* and *wait()* returns immediately - the consumer may start to run. The notification sent by the producer is ignored.
2. Now suppose that the consumer steps into the critical section first and thus blocks the producer. The first step of the consumer is the call to *wait()*. As the buffer is empty, the predicate returns *false*. Now:
   a. *wait()* unlocks the mutex, the producer may start to run.
   b. *wait()* does not return but starts to wait for the notification. So the consumer is blocked and the producer may complete its task.

   Before return the producer sends notification. *wait()* in consumer calls once more the predicate. As the buffer is already filled, the predicate returns *true*. Now:
   a. *wait()* locks the mutex, but as the producer has already finished, it has no consequences.
   b. *wait()* returns and thus the consumer may start to run.

So, in this solution we may be sure that at first the producer creates the data and only after that the consumer processes the data. Compare it with solution on slides *Mutexes (3)* ... *Mutexes (5).*

Conditional variables (8)

Example: classical producer – consumer problem in which the producer has to inform the the consumer that a data package is prepared, wait until the consumer has finished the processing and start to prepare the next package.

```
int main()
{
  atomic<bool> stop = false;
  vector<int> *pBuf = new vector<int>;
  mutex mx;
  condition_variable cv;
  thread ControllerThread{ Controller(stop) };  // see slide Atomic variables (5)
  thread ProducerThread{ Producer(pBuf, &stop, &mx, &cv, 32) };
  thread ConsumerThread{ Consumer(pBuf, &stop, &mx, &cv) };
  ConsumerThread.join();
  ProducerThread.join();
  ControllerThread.join();
  return 0;
}
```

Conditional variables (9)

```
class Producer {
public: vector<int> *pBuf;
        atomic<bool> *pStop;
        mutex *pMx;
        condition_variable *pCv;
        int buf_size;
        Producer (vector<int> *pv, atomic<bool> *pb, mutex *pm, condition_variable *pc,
                  int n) : pBuf(pv), pStop(pb), pMx(pm), pCv(pc), buf_size(n) { }
        void operator() () {
            default_random_engine generator;
            uniform_int_distribution<int> delay (0, 10000);
            uniform_int_distribution<int> random_number(0, 100);
            while (!*pStop) {
                unique_lock<mutex> lock(*pMx); // locks the mutex
                pBuf->resize(buf_size);
                this_thread::sleep_for(chrono::milliseconds(delay(generator)));
                generate(pBuf->begin(), pBuf->end(), [&]() { return random_number(generator); });
                pCv->notify_one(); // releases the comsumer
                pCv->wait(lock); // blocks the producer
             }
        }
};
```

Conditional variables (10)

```
class Consumer {
public: vector<int> *pBuf;
        atomic<bool> *pStop;
        mutex *pMx;
        condition_variable *pCv;
        Consumer (vector<int> *pv, atomic<bool> *pb, mutex *pm, condition_variable *pc) :
                pBuf(pv), pStop(pb), pMx(pm), pCv(pc) { }
        void operator() () {
            while (!*pStop) {
                unique_lock<mutex> lock(*pMx); // locks the mutex
                pCv->wait(lock, [&]() { return !pBuf->empty(); });
                for_each(pBuf->begin(), pBuf->end(), [](int i) { cout << i << ' '; });
                pBuf->resize(0);
                cout << endl;
                pCv->notify_one(); // releases the producer
            }
        }
};
```

Conditional variables (11)

```
wait_for(unique_lock_name, duration);
wait_until(unique_lock, time_point);
```

and

```
wait_for(unique_lock_name, duration, predicate);
wait_until(unique_lock_name, timepoint, predicate);
```

Those two methods block the thread until notified or until the specified timeout has elapsed or timepoint has been reached. The return value is *cv_status::timeout* or *cv_status::no_timeout*. About the usage of predicates see slide *Conditional variables (3)*.

async and futures (1)

Suppose our program consists of 3 tasks. The final task 3 needs data prepared by tasks 1 and 2. There are no any dependences between tasks 1 and 2: they do not need request data from each other, do not share resources, etc. If we have a multiprocessor computer, we may put task 2 into a separate background thread and force it run concurrently with task 1: thus we may get better performance.

There is another way: use function *std::async* and futures:

```
future <entry_point_function_return_value_type> future_name =
                                async(entry_point_function_name, list_of_input_parameters);
```

The task entry point function may have any set of input parameters. Example: suppose the entry point function for task 2 is:

```
bool ReadData(unsigned char *, int);
```

To start asynchronous reading task:

```
#include <future> // see http://www.cplusplus.com/reference/future/async/
                  // see http://www.cplusplus.com/reference/future/future/
unsigned char *pBuf = new unsigned char[1024 * 10];
future<bool> result = async(ReadData, pBuf, 1024 * 10); // task 2 is now in a separate thread
…………………………………….. // program continues with task 1
if (result.get()) // method get() returns the return value of task entry point function
{  // you can call get on a specific future only once!
  ………………………… // program starts to execute task 3
}
```

async and futures (2)

async does not guarantee that a separate thread with the specified entry point function will start immediately. When the future's *get()* method is called, it is possible that:

- The asynchronous task has finished and *get()* returns the output value immediately.
- The asynchronous task is still running. In that case *get()* blocks the current thread until the return value has become available.
- The asynchronous task has not started yet. Then it is forced to start, the current thread blocks. Actually it means that the ansynchronous executing of tasks has failed. If the call to *get()* is omitted, it may happen that the asynchronous task never starts.

There are some possibilities to control the behavior of asynchronous task:

```
future< entry_point_function_return_value_type> future_name =
    async(launch_policy, entry_point_function_name, list_of_input_parameters);
```

The launch policy may be:

1. *launch::async* – try to launch a new thread immediately. May be risky because if it is not possible, exception will be thrown.
2. *launch::deferred* – launch when *get()* is called, i.e. no concurrent executing. May be useful as temporary setting for debugging.
3. *launch::async* | *launch::deferred* – default setting, launch time is set by system.

async and futures (3)

Class *future* has method:

wait();

This function behaves similarly to *get():* if the asynchronous task has not finished it blocks the current thread until the asynchronous task has ended. Also, if necessary it forces the asynchronous task to start. Later, the output value may be retrieved by *get()*.

Class *future* has also methods:

- wait_for(duration);
- wait_until(timepoint);

Those functions block the current thread until the asynchronous task has finished or timeout elapsed or the specified timepoint reached. But they do not force the asynchronous task to start. Their return value may be:

1. *future_status::deferred* - the asynchronous task has not started yet.
2. *future_status::timeout* - the asynchronous task is running but waiting time has elapsed.
3. *future_status::ready* - the asynchronous task has finished.

Example:

```
if (result.wait_for(chrono::seconds(60)) != future_status::ready) {
{
    cout << "It seems that the asynchronous task has problems" << endl;
    return;
}
```

wait_for(chrono::seconds(0)) returns us the current status of the asynchronous task.

async and futures (4)

If the asynchronous task has thrown an unhandled exception, *get()* catches it and rethrows:

```
try
{
      if (result.get())
     {
           ………………. // program starts to execute task 3
     }
}
catch (exception &e)
{
  ……………………..
}
```

Method *wait()* does not rethrow exceptions.

Method *get()* may be called only once. After get() the future becomes invalid. To check the state of future use method *valid()* – it returns false if the future has become not useable.

In a complicated application with a lot of threads multiple calls to *get()* may be needed. In that case use *shared_future* (see https://cplusplus.com/reference/future/shared_future/).

async and futures (5)

Instead of entry point function we may use functor or lambda. Example:

```
class Producer
{
public:
    list<int> &buf; // reference, see slide "Initializing (4)" from chapter "Advanced C++"
    int lower, upper;
    Producer(list<int> &v, int l, int u) : buf(v), lower(l), upper (u) { }
    void operator() ()
    {
            default_random_engine generator;
            uniform_int_distribution<int> delay(0, 10000);
            uniform_int_distribution<int> random_number(lower, upper);
             this_thread::sleep_for(chrono::milliseconds(delay(generator)));
             generate(buf.begin(), buf.end(),
                    [&]() { return random_number(generator); });
     }
};
```

async and futures (6)

```
int main()
{
    list<int> buf1(10);
    list<int> buf2(10);
    future<void> f1 = async(Producer(buf1, 0, 100));
    future<void> f2 = async(Producer(buf2, -100, 0));
    f1.get();  // here the thread returns nothing but we need to wait until it quits
               // so get() from class future behaves as join from class thread
    f2.get();
    buf1.sort();
    buf2.sort();
    buf1.merge(buf2);
    for_each(buf1.begin(), buf1.end(), [](int i) { cout << i << ' '; });
    cout << endl;
    return 0;
}
```

Shortly: a *future* and *async* provide facilities to retrieve values and / or exceptions from a function that is located in background thread and is currently executing or has already executed or will be executed (in the last case, they may force to start executing).

async and futures (7)

Example (compare with solution on slides *Atomic (4)* and *Atomic (5)*):

```
class Worker
{
public:
 vector<int> &buf;
 future<void> &fut;
Worker (vector<int> &v, future<void> &f) : buf(v), fut(f) { }
void operator() ()
{
   default_random_engine generator;
   uniform_int_distribution<int> delay_distribution(0, 10000);
   uniform_int_distribution<int> value_distribution(0, 100);
   while (fut.wait_for(chrono::seconds(0)) != future_status::ready) {
        // worker runs until the keyboard async thread has not exited
        this_thread::sleep_for(chrono::milliseconds(delay_distribution(generator)));
        generate(buf.begin(), buf.end(),
                              [&]() { return value_distribution(generator); });
        …………………… // do something with the generated numbers
     }
  }
};
```

async and futures (8)

```
void Keystroke()
{
     _getch(); // waits for keystroke, then exits
}

int main()
{
    vector<int> buf(32);
    future<void> fut = async(Keystroke);
    thread WorkerThread { Worker(buf, fut) };
    WorkerThread.join();
}
```

Packaged tasks

It is possible to prepare tasks beforehand and invoke them later (or not invoke at all if they come out to be unnecessary):

```
packaged_task<entry_point_value_return_type(list_of_parameter_types)>
task_name(entry_point_function_name);
```

Each packaged_task contains a future. To retrieve it:

```
future<entry_point_value_return_type> future_name = task_name.get_future();
```

To invoke task:

```
task_name(actual_parameters_list);
```

and after that apply the associated future's *get()* to retrieve the result.

Example: suppose the entry point function (functors and lambdas also allowed) for a task is:

```
bool ReadData(unsigned char *, int);
```

To create the associated task package:

```
#include <future> // see http://www.cplusplus.com/reference/future/packaged_task/
packaged_task<bool(unsigned char *, int)> read_task(ReadData);
………………………………….. // program continues
if (data_needed) {
  unsigned char *pBuf = new unsigned char[1024 * 10];
  future<bool> read_result = read_task.get_future(); // retrieves the future
  read_task(pBuf, 10240) // program starts to execute task, if possible then in separate thread
  bool success = read_result.get();  // blocks until the end of task
}
```

Promises (1)

We can store the data created by a thread into a promise and later use future to retrieve it into another thread:

```
promise<task_return_value_type> promise_name;
promise_name.set_value(data_to_store); // can be called only once
```

Each promise has a future. When we need data stored in promise, retreive it:

```
future<return_value_type> future_name = promise_name.get_future();
```

and then call *get()*.

Example:

```
void Producer(list<int> *pBuf, int lower, int upper, promise<list<int> *> *pPromise)
{ // see http://www.cplusplus.com/reference/future/promise/
  default_random_engine generator;
  uniform_int_distribution<int> delay(0, 10000);
  uniform_int_distribution<int> random_number(lower, upper);
  this_thread::sleep_for(chrono::milliseconds(delay(generator)));
  generate(pBuf->begin(), pBuf->end(), [&]() { return random_number(generator); });
  pBuf->sort();
  pPromise ->set_value(pBuf); // store the result into promise
                              // turn attention that the thread entry point function
                              // does not return any value
}
```

You can call *set_value* on a specific promise only once!

Promises (2)

```
void Consumer(promise<list<int>*>* pPromise)
{
    future<list<int> *> fut = pPromise->get_future(); // future associated with promise
    list<int> *pRes = fut.get(); // retrieve the result, if necesasary, wait
    for_each(pRes->begin(), pRes->end(), [](int i) { cout << i << ' '; });
    cout << endl;
}

int main()
{  // Compare with similar example on slides Mutexes (3) … Mutexes (5)
    // Here we need neither mutexes nor shared memory fields
  promise<list<int> *> prom;
  list<int> buf(10);
  thread thr1(Producer, &buf, 0, 100, &prom);
  thread thr2(Consumer, &prom);
  thr1.join();
  thr2.join();
  return 0;
}
```

Promises (3)

The promises are very necessary if we need to get results created by detached threads.
Example:

```
void Producer(list<int> *pBuf, int lower, int upper, promise<void> *pPromise)
{
  default_random_engine generator;
  uniform_int_distribution<int> delay(0, 10000);
  uniform_int_distribution<int> random_number(lower, upper);
  this_thread::sleep_for(chrono::milliseconds(delay(generator)));
  generate(pBuf->begin(), pBuf->end(), [&]() { return random_number(generator); });
  pBuf->sort();
  pPromise ->set_value(); // inform that the thread has ended
}

void Consumer(list<int> *pBuf, promise<void>* pPromise)
{
    future<void> fut = pPromise->get_future(); // future associated with promise
    fut.get(); // wait until the end of producer
    for_each(pRes->begin(), pRes->end(), [](int i) { cout << i << ' '; });
    cout << endl;
}
```

Promises (4)

```
int main()
{
  promise<void> prom;
  list<int> buf(10);
  thread thr1(Producer, &buf, 0, 100, &prom);
  thr1.detach();
  thread thr2(Consumer, &buf, &prom);
  thr2.join();
  return 0;
}
```

Asynchronous I/O in Windows (1)

C++ input/ output standard classes (see http://www.cplusplus.com/reference/fstream/) are excellent for simple file operations. In more complicated situations, however, we have to use the Windows standard I/O mechanism.

The first step is to create the file:

```
HANDLE handle_name = // HANDLE is defined in Windows.h
CreateFileA(file_name_and_path,    // as regular C string
desired_access, // GENERIC_READ for files used for reading only
                // GENERIC_WRITE for files used for writing only
                // GENERIC_READ | GENERIC_WRITE for the both operations
share_mode,     // outside the course scope, set to 0
security_attributes, // outside the course scope, set to NULL
creation_disposition, // CREATE_ALWAYS and if already exists, at first destroy it
                      // CREATE_NEW and if already exists, the operation fails
                      // OPEN_EXISTING and if not found, the operation fails
                      // OPEN_ALWAYS and if not found creates a new one
                      // TRUNCATE_EXISTING (destroy the contents) and if not found,
                      // the operations fails
flags_and_attributes, // in our course FILE_FLAG_OVERLAPPED (discussed later)
template_file_handle); // outside the course scope, set to NULL
```

The complete specification of function *CreateFileA* is on website https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-createfilea is

Asynchronous I/O in Windows (2)

If the function fails, the return value is *INVALID_FILE_HANDLE*. To know the reason, call function *GetLastError*:

```
unsigned long error_code = GetLastError();
```

The error codes are on https://docs.microsoft.com/en-us/windows/desktop/Debug/system-error-codes.

If you do need the file any more, close it:

```
CloseHandle(handle_name);
```

Example:

```
HANDLE hFile = CreateFileA("FileExample.bin", GENERIC_READ | GENERIC_WRITE,
                           0, NULL, CREATE_ALWAYS, FILE_FLAG_OVERLAPPED, NULL);
if (hFile == INVALID_HANDLE_VALUE)
    cout << "File not created, error " <<  GetLastError() << endl;
// Important: you should always check was a file operation successful or not
…………………………………………………
CloseHandle(hFile);
```

Asynchronous I/O in Windows (3)

To read from file synchronously:

```
int result_name = ReadFile(handle_name, pointer_to_buffer_that_receives_read_data,
max_number_of_bytes_to_read, pointer_to_variable_for _number_of_bytes_actually_read,
NULL);
```

If the reading failed, the return value is *FALSE* and *GetLastError()* returns the error code. The actual number of read bytes may be less than the number of needed bytes.

Example:

```
unsigned long nBytesToRead = 1024, nReadBytes = 0;
unsigned char *pBuffer = new unsigned char[nBytesToRead];
HANDLE hFile;
int Result = ReadFile(hFile, pBuffer, nBytesToRead, &nReadBytes, NULL);
if (!Result)
     cout << "Data not read, error " <<  GetLastError() << endl;
else if (nReadBytes != nBytesToRead)
    cout << "Only " << nReadBytes << " bytes instead of " << nBytesToRead << " was read"
            << endl;
else
    cout << nReadBytes << " bytes was read" << endl;
```

The complete specification of function *ReadFile* is on website
https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-readfile

Asynchronous I/O in Windows (4)

To write into file synchronously:

int result_name = WriteFile(handle_name, pointer_to_buffer_containing_data, number_of_bytes_to_write, pointer_to_variable_for _number_of_bytes_actually_written, NULL);

If the writing failed, the return value is 0 and *GetLastError()* returns the error code. The actual number of written bytes may be less than the number of bytes we wanted to write.

Example:

```
unsigned long nBytesToWrite = 1024, nWrittenBytes = 0;
unsigned char *pBuffer = new unsigned char[nBytesToWrite];
…………………………………………………………………….
HANDLE hFile;
int Result = WriteFile(hFile, pBuffer, nBytesToWrite, &nWrittenBytes, NULL);
if (!Result)
     cout << "Data not written, error " <<  GetLastError() << endl;
else if (nBytesToWrite != nWrittenBytes)
    cout << "Only " << nWrittenBytes << " bytes instead of " << nBytesToWrite
            << " was written" << endl;
else
    cout << nWrittenBytes << " bytes was written" << endl;
```

The complete specification of function *WriteFile* is on website
https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-writefile

Asynchronous I/O in Windows (5)

The synchronous I/O operations with disk files in most cases do not lead to problems. For Windows, however, any data that is not in the memory region set for the current application is considered to be in a file. For example, if our application controls an external device connected to computer through serial port COM1, then we need to read data from and write data to file \\.*COM1*. This external device may be temporarily or perpetually (is switched off?) not able to send data to us or retrieve data that we want to send to it. In that case the synchronous I/O operations block their thread and it is impossible to unblock it – in other words the application hangs.

For asynchronous I/O we need Windows events:

```
HANDLE handle_name = CreateEventA (
                attributes, // outside the course scope, set to NULL
                reset_mode,  // manual reset TRUE, auto reset FALSE
                initial_state,  // not signaled FALSE, signaled TRUE
                name); // outside the course scope, set to NULL
```

Other standard functions for events:

```
SetEvent(handle_name); // state to signaled
ResetEvent(handle_name); // state to non-signaled
CloseHandle(handle_name);
```

See more on website:

https://docs.microsoft.com/en-us/windows/desktop/Sync/synchronization-functions#event-functions

Asynchronous I/O in Windows (6)

The events are to block and unblock a thread:

```
int result_name = WaitForSingleObject(event_handle_name, timeout_in ms);
```

If the event is non-signaled, *WaitForSingleObject* function does not return and thus blocks its thread. To unblock, another thread or an asynchronous I/O operation must set the event to signaled. The function returns also when the timeout interval elapses. If the timeout is set to *INFINITE*, time is not measured.

The return value may be:

- *WAIT_OBJECT_0* – the event was set to signaled
- *WAIT_TIMEOUT* – the event is still non-signaled, but time has elapsed
- *WAIT_FAILED* – system problems

If the event is defined as auto-reset, *WaitForSingleObject* function also turns it back to not signaled. If the event is defined as manual reset, the user has to apply function *ResetEvent*.

The other blocking / unblocking function is

```
int result_name = WaitForMultipleObjects(number_of_handles,
                                pointer_to_array_of_handles, all_or_one, timeout_in ms);
```

It checks the state of several events stored into array. If the third parameter *all_or_one* is *TRUE*, function *WaitForMultipleObjects* returns only when all the specified events are signaled. If *all_or_one* is *FALSE*, it returns when at least one of them is signaled. If just the array's *i*-th event has become signaled, the return value is *WAIT_OBJECT_0 + i*. If several or all the events are signaled, *i* is the smallest index from the signaled events subset.

Asynchronous I/O in Windows (7)

For asynchronous I/O we need a struct of type *OVERLAPPED*:

```
OVERLAPPED Overlapped;
memset(&Overlapped, 0, sizeof Overlapped); // fill with zeroes
```

Next we have to create an initially non-signaled event and store its handle in *Overlapped*:

```
Overlapped.hEvent = CreateEventA(NULL, FALSE, FALSE, NULL);
```

We need at least one more non-signaled event that is triggered from another thread. For example, let us suppose that this event with handle *hExitEvent* will be set to signaled when the user has decided to exit the application.

To finish the preparations create an array of events:

```
HANDLE hEvents[] = { Overlapped.hEvent, hExitEvent };
```

In calls to *ReadFile* and *WriteFile* functions the last parameter must be the pointer to *Overlapped*, for example:

```
unsigned long nBytesToRead = 1024, nReadBytes = 0;
unsigned char *pBuffer = new unsigned char[nBytesToRead];
HANDLE hFile;
int Result = ReadFile(hFile, pBuffer, nBytesToRead, &nReadBytes, &Overlapped);
```

If the output value is not *FALSE*, the I/O operation succeeded without any delays and we may check does the number of read or written bytes match the number of bytes we wanted to read or write (see slides *Asynchronous I/O in Windows (3)* and *(4)*).

Asynchronous I/O in Windows (8)

If the output value is *FALSE*, we must call *GetLastError()*:

```
int error = GetLastError();
```

If the error code is *ERROR_IO_PENDING*, the I/O operation failed now but may succeed later and we have to wait:

```
int WaitResult = WaitForMultipleObjects(2, hEvents, FALSE, timeout);
```

The waiting stops when:

- At last the I/O operation was completed, *WaitResult* is *WAIT_OBJECT_0*.
- The user wants to exit the application and triggers *hExitEvent*, *WaitResult* is *WAIT_OBJECT_0 + 1*. Thus the application is always under the user's control.
- Timeout occurred (only if *timeout != INFINITE*), *WaitResult* is *WAIT_TIMEOUT.*
- Some system errors, *WaitResult* has other values

If the I/O operation was completed, we may get the number of bytes that was actually read or written:

```
GetOverlappedResult(hFile, &Overlapped, &nReadBytes, FALSE);
```

or

```
GetOverlappedResult(hFile, &Overlapped, &nWrittenBytes, FALSE);
```

If the error code was not *IO_ERROR_PENDING*, the I/O operation has totally failed.

See also *WindowsAsyncIO.cpp* from *ICS0025 Examples.zip*.

Dynamic link libraries (1)

Large application consist of more that one file: *.exe + several *.dll. Why the DLLs are necessary:

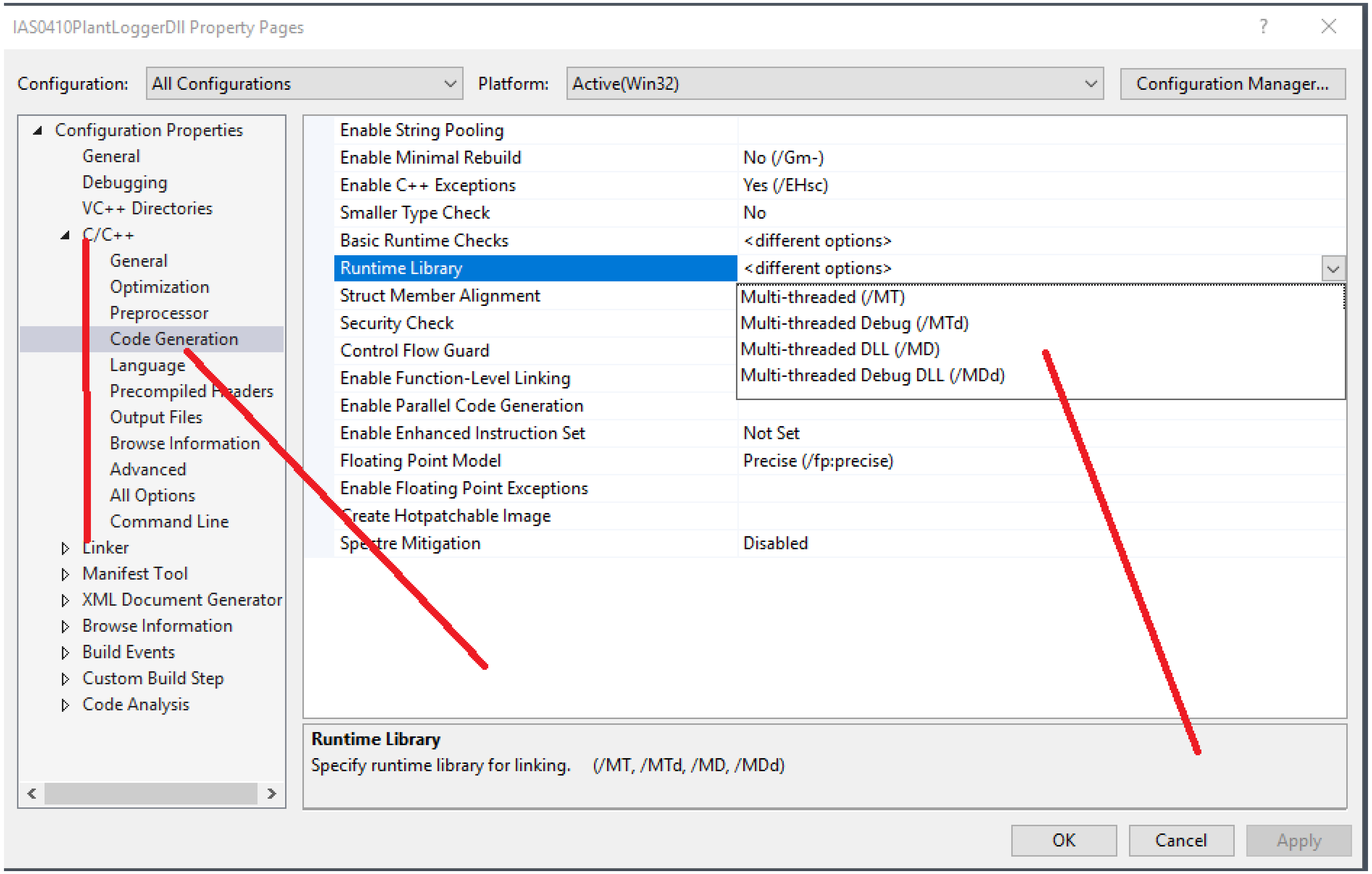
1. Very large executive files can be divided into smaller modules.
2. In development: each programmer (or group of programmers or company) can link his part of code (a software module) as a DLL and thus work without disturbing the other participants.
3. In maintenance: the developer changes one of the DLLs and sends it to the customer.
4. Industrial software development: one DLLs may be used in many different applications.

There are two options for connecting *.exe and *.dll:

1. Implicit linking: the DLLs are connected to the application when Windows is loading the application into memory.
2. Explicit linking: a DLL is connected only when the application needs and calls the *LoadLibrary()* function.

Important for Visual Studio users: Project properties ➔C/C++ ➔ Code generation ➔ Runtime library has two options: multithreaded /MT (runtime support libraries are linked to the application, the total amount of *.exe is large), or multithreaded DLL /MD (runtime support libraries are applied as DLLs, the amount of *.exe is smaller but when the customer's PC does not have all the necessary libraries, the application crashes).

Dynamic link libraries (2)



IAS0410PlantLoggerDll Property Pages

Configuration: All Configurations Platform: Active(Win32) Configuration Manager...

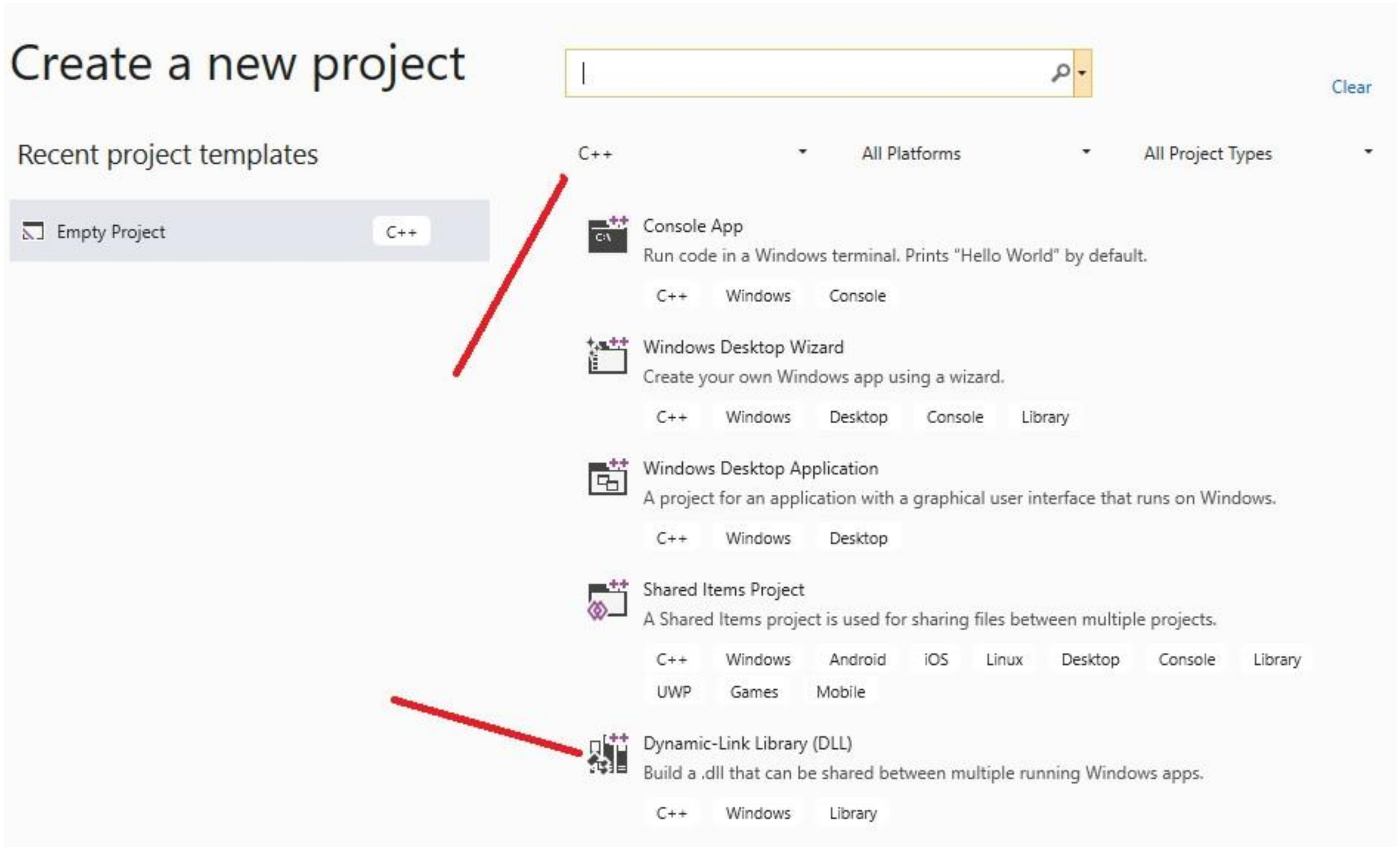
Enable String Pooling	
Enable Minimal Rebuild	No (/Gm-)
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	<different options>
Runtime Library	<different options>
Struct Member Alignment	
Security Check	
Control Flow Guard	
Enable Function-Level Linking	
Enable Parallel Code Generation	
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	
Create Hotpatchable Image	
Spectre Mitigation	Disabled

Multi-threaded (/MT)
Multi-threaded Debug (/MTd)
Multi-threaded DLL (/MD)
Multi-threaded Debug DLL (/MDd)

Runtime Library
Specify runtime library for linking. (/MT, /MTd, /MD, /MDd)

Dynamic link libraries (3)

To start with DLL project inform the wizard that your target is C++ Dynamic-link Library.



Create a new project

Recent project templates

Empty Project — C++

C++ · All Platforms · All Project Types

Clear

Console App
Run code in a Windows terminal. Prints "Hello World" by default.
C++ · Windows · Console

Windows Desktop Wizard
Create your own Windows app using a wizard.
C++ · Windows · Desktop · Console · Library

Windows Desktop Application
A project for an application with a graphical user interface that runs on Windows.
C++ · Windows · Desktop

Shared Items Project
A Shared Items project is used for sharing files between multiple projects.
C++ · Windows · Android · iOS · Linux · Desktop · Console · Library · UWP · Games · Mobile

Dynamic-Link Library (DLL)
Build a .dll that can be shared between multiple running Windows apps.
C++ · Windows · Library

Dynamic link libraries (4)

Suppose that our project name is *Example*. The wizard creates several files, among them file *dllmain.cpp* (analyzed on the next slide). First create a **.cpp* file and a **.h* file (by tradition their names must match the DLL name, so in our case *Example.cpp* and *Example.h)*. In the header file write code:

```
#ifdef EXAMPLE_EXPORTS
#define LIBSPEC _declspec(dllexport)
#else
#define LIBSPEC _declspec(dllimport)
#endif
```

Constant *EXAMPLE_EXPORTS* (generally dllname_EXPORTS) is created by wizard and used when the DLL code is compiled. It is not created for applications that use DLLs.
All the prototypes and definitions of functions that the DLL exports (i.e. they will be called by applications using this DLL) must start with LIBSPEC, for example in *Example.h*:

```
LIBSPEC int Sum(int, int);
```

in *Example.cpp*:

```
#include "pch.h" // if the wizard created this file
#include "Example.h"
LIBSPEC int Sum(int x1, int x2)
{
    return x1 + x2;
}
```

Dynamic link libraries (5)

The wizard-created function *DllMain()* from *dllmain.cpp* is the entry point function.

```
#include "pch.h"
BOOL APIENTRY DllMain( HMODULE hModule,   DWORD  ul_reason_for_call,
                                           LPVOID lpReserved)
{// DllMain() is called automatically each time when the process or one of its threads
 //  attaches or detaches the DLL:
 // 1. In case of implicit linking when the process starts and terminates
 // 2. In case of explicit linking when the application calls LoadLibrary() and
 //  FreeLibrary() functions
   switch (ul_reason_for_call)
   {
   case DLL_PROCESS_ATTACH:
   case DLL_THREAD_ATTACH:
      // TODO: add code for initialization operations
      break;
   case DLL_THREAD_DETACH:
   case DLL_PROCESS_DETACH:
      // TODO: add code for clean-up operations
       break;
   }
   return TRUE;
}
```

Dynamic link libraries (6)

Visual Studio builds two files: *Example.dll* and Example.lib. If the application using our DLL applies explicit linking, it needs only *Example.dll*. In case of implicit linking it needs 3 files: *Example.dll*, *Example.lib* and *Example.h*.

In application with implicit linking calls to functions exported by DLL do not differ from calls to standard functions. The prototypes during compiling are read from *Example.h* that must be copied into the folder where the other source code files are located. However, to link the application we need the object modules (**.obj*) of DLL functions. To play a trick on linker we need *Example.lib* that contains stubs – short empty functions replacing the actual DLL functions. When the application is running, instead of stubs the functions from DLL are called. You may copy the **.lib* into folder containing source codes.

The easest way is to put DLL into the folder where the *.exe is located.

In case of explicit linking the application must load the DLL:
HMODULE handle_name LoadLibraryA(dll_filename_as_C_string);
Zero handle value means that the DLL was not loaded. To know the reason call *GetLastError()*. To detach the DLL:
FreeLibrary(handle_name);

To call a function we need pointer to it:
FARPROC pointer_name = GetProcAddress(handle_name, function_name_as_C_string);
Zero result value means that the function was not found. To know the reason call *GetLastError()*.

Dynamic link libraries (7)

Explicit linking example:

```
#include "Windows.h"
HMODULE hDLL = LoadLibraryA("DLLExample.dll");
if (!hDLL)
{
        cout << "DLLExample.dll not found, error " << GetLastError() << endl;
        return;
}
FARPROC pSum = GetProcAddress(hDLL, "Sum");
if (pSum == NULL)
{
      FreeLibrary(hDLL);
      cout << "Function Sum() not found, error " <<   GetLastError() << endl;
      return;
}
 int result = ((int(*)(int, int))pSum)(5, 6); // cast pointer to actual type
 FreeLibrary(hDLL);
 return;
```

Dynamic link libraries (8)

Turn attention that C++ compiler performs so called name mangling. For example, a function with prototype

```
unsigned char *Run(unsigned char *);
```

in DLL is named as *?Run@@YAPEAEPEAE@Z.* To see the new names you can analyse the DLL with *Dependency Walker* standard utility.

The C compiler keeps the original names. To force the C++ compilers to follow C naming conventions use extern "C" linking:

```
extern "C"
{
     LIBSPEC int Sum(int, int);
}
extern "C"
{
    LIBSPEC int Sum(int x1, int x2)
    {
        return x1 + x2;
    }
}
```