Concurrency in C++11

Arash

Threads

```
#include <iostream>
#include <thread>
using namespace std;

void func(int x, int y, int& z) {
    this_thread.sleep_for(chrono::seconds{5})
    cout << "Inside thread " << x << “, “ << y << “, “ << z << endl;
    x = 100; y = 200; z = 300;
}

int main() {
    int x = 2, y = 12; z = 102
    thread th{&func, x, ref(y), ref(z)}; // The first parameter can also be functor or lambda
    x++; y++; z++;
    cout << "Before join " << x << “, “ << y << “, “ << z<< endl;
    th.join();
    cout << "After join " << x << “, “ << y << “, “ << z << endl;
    return 0;
}
```

std::this_thread::yield

This function shall be called when a thread waits for other threads to advance without blocking.

```
// this_thread::yield example
#include <iostream>       // std::cout
#include <thread>         // std::thread, std::this_thread::yield
#include <atomic>         // std::atomic
std::atomic<bool> ready (false);
void count1m(int id) {
  while (!ready) {              // wait until main() sets ready...
    std::this_thread::yield();
  }
  for (volatile int i=0; i<1000000; ++i) {}
  std::cout << id;
}
int main ()
{
  std::thread threads[10];
  std::cout << "race of 10 threads that count to 1 million:\n";
  for (int i=0; i<10; ++i) threads[i]=std::thread(count1m,i);
  ready = true;                 // go!
  for (auto& th : threads) th.join();
  std::cout << '\n';
  return 0;
}
```

Mutex

```
template <typename T>
class container {
    mutex lock;
    vector<T> elements;
public:
    void add(T element){
      lock.lock();
      elements.push_back(element);
      lock.unlock();
    }
    template<typename... Args>
    void add(T first, Args... args) {
        add(first);
        add(args...);
    }
    void dump() {
        lock_guard<mutex> locker(_lock);
        for(auto e : _elements)
            cout << e << "  ";
        cout << endl;
    }
};
```

```
void func(container<int>& cont)
{
    cont.add(rand()%10, rand()%10, rand()
%10);
}

int main()
{
    srand((unsigned int)time(0));
    container<int> cont;
    std::thread t1(func, ref(cont));
    std::thread t2(func, ref(cont));
    std::thread t3(func, ref(cont));
    t1.join();
    t2.join();
    t3.join();
    cont.dump();
    return 0;
}
```

Propagate exception between threads

```
mutex g_mutex;
vector<exception_ptr> g_exceptions;

void throw_function() {
   throw exception();
}

void func() {
   try {
      throw_function();
   }
   catch(...) {
      lock_guard<mutex> lock(g_mutex);
      g_exceptions.push_back(current_exception());
   }
}

int main() {
   g_exceptions.clear();
   thread t(func);
   t.join();
   for(auto& e : g_exceptions) {
      try {
         if(e != nullptr) {
            rethrow_exception(e);
         }
      }
      catch(const exception& e) {
         cout << e.what() << endl;
      }
   }
   return 0;
}
```

Condition Variable

```
const int size = 10;
mutex mutex;
condition_variable condvar;

struct Data {
    int d;
    int error;
};

queue<Data> messageQ;

void Producer()
{
    int i = 0;
    while(++i<=size) {
        int d = rand()%10;
        int error = (i==size)?1:0;
        Data data {d, error};
        this_thread::sleep_for(milliseconds(rand()%500));
        lock_guard<std::mutex> guard(mutex);
        cout << "Producing message: " << data.d << endl;
        messageQ.push(data);
        condvar.notify_one();
    }
}
void Consumer()
{
    while(1) {
        this_thread::sleep_for(milliseconds(rand()%500));
        unique_lock<std::mutex> ulock(mutex);
        condvar.wait(ulock, [] {return !messageQ.empty(); });

        Data data = messageQ.front();
        cout << "Consuming message: " << data.d  << endl;
        messageQ.pop();
        if (data.error)
            break;
    }
}
```

```
int main() {
   std::thread t1 {Producer};

   std::thread t2 {Consumer};

   t1.join();

   t2.join();
}
```

Async

```
// async example
#include <iostream>       // std::cout
#include <future>         // std::async, std::future
// a non-optimized way of checking for prime numbers:

bool is_prime (int x) {
  std::cout << "Calculating. Please, wait...\n";
  for (int i=2; i<x; ++i) if (x%i==0) return false;
  return true;
}

int main ()
{
  // call is_prime(313222313) asynchronously:
  std::future<bool> fut = std::async (is_prime,313222313);
  std::cout << "Checking whether 313222313 is prime.\n";
  // ...
  bool ret = fut.get();      // waits for is_prime to return
  if (ret) std::cout << "It is prime!\n";
  else std::cout << "It is not prime.\n";
  return 0;
}
```

Parallel for_each using async

```
template<typename Iterator,typename Func>

void parallel_for_each(Iterator first,Iterator last,Func f)
{
    ptrdiff_t const range_length=last-first;

    if(!range_length)
        Return;

    if(range_length==1)
    {
        f(*first);
        return;
    }

    Iterator const mid=first+(range_length/2);
    std::future<void> bgtask=std::async(&parallel_for_each<Iterator,Func>,
                                        first,mid,f);

    try
    {
        parallel_for_each(mid,last,f);
    }
    catch(...)
    {
        bgtask.wait();
        throw;
    }

    bgtask.get();
}
```

Future and Promises

Another transmission mechanism of the data resulting from the operations performed by different threads is to use a std::promise/std::future. A std::promise object provides a mechanism in order to set a type T value, which then can be read by a std::future object.

```
void execute(std::promise& promise) {
     std::string str("processed data");
     promise.set_value(std::move(str)); //-- (3)
}
void main() {
     std::promise promise; //-- (1)
     std::thread thread(execute, std::ref(promise)); //-- (2)
     std::future result(promise.get_future()); //-- (4)
     std::cout << "result: " << result.get() << std::endl; //-- (5)
}
```

packaged_task

A std::package object connects a function and a callable object. When the std::package <> object is called, this calls in turn the associated function or the callable object and prepares the future object in ready state, with the value returned by the performed operation as associated value.

```
int execute(int x, int y) {
     return std::pow(x,y);
}
void main() {
     std::packaged_task task(std::bind(execute, 2, 10));
     std::future result = task.get_future(); //-- (1)
     task(); //-- (2)
     std::cout << "task_bind: " << result.get() << " "; //-- (4)
}
```

Exercise

- Write a parallel find.

Resources

- [1] http://www.codeproject.com/Articles/598695/Cplusplus-threads-locks-and-condition-variables
- [2] http://www.cplusplus.com/reference/future/async/
- [3] http://www.todaysoftmag.com/article/753/multithreading-in-c-11-standard-ii