C++ 0x aka C++11

- Jako jiné jazyky, např. Free/Object Pascal, se C++ ve standardu ++0x dočkal podpory vláken
- Výhodou je, že standardní knihovna je platformě nezávislá na úrovni zdrojového kódu

- Základním kamenem je třída std::thread

```
#include <thread>
#include <iostream>

void my_thread_func() {
    std::cout<<"hello"<<std::endl;
}

void write_sum(int x,int y){
    std::cout<<x<<" + "<<y<<" = "
             <<(x+y)<<std::endl;
}


int main() {
    std::thread t1(my_thread_func);
    std::thread t2(write_sum,123,456);
    t1.join();
    t2.join();
}
```

- Třída bere funkci, i např. přetížený operátor () objektu, jako první argument
    - A pak následují další argumenty
    - Argumenty se kopírují do interního úložiště threadu

- Je-li cílem spustit jinou funkci než funkční operátor ()
  - předá se pointer na funkci
  - pointer, který se použije jako hodnota this
  - a případné argumenty

```
#include <thread>
#include <iostream>

class CSayHello {
public:
    void greeting(std::string const& message) const {
        std::cout<<message<<std::endl;
    }
};

int main() {
    CSayHello x;
    std::thread t(&CSayHello::greeting,&x,"goodbye");
    t.join();
}
```

  - pochopitelně se musí zajistit, že objekt přežije vlákno, např. lze takto:

```
int main() {
    std::shared_ptr<CSayHello> p(new CSayHello);
    std::thread t(&CSayHello::greeting,p,"goodbye");
    t.join();
}
```

- Je-li cílem předat referenci na existující objekt
  - Tj. chceme-li zavolat () operátor přímo na konkrétním objektu, ne na jeho kopii
  - A chceme-li čitelný kód

```
#include <thread>
#include <iostream>
#include <functional>

class CPrintThis {
public:
    void operator()() const     {
        std::cout<<"this="<<this<<std::endl;
    }
};

int main() {
    CPrintThis x;
    x();
    std::thread t(std::ref(x));
    t.join();
}
```

  - std::ref se dá použít i pro argumenty

- Synchronizace
  - std:mutex
    - má lock, try_lock a unlock
    - není rekurzivní
  - std::recursive_mutex
  - timed_mutex a recursive_timed_mutex umožňují u trylock specifikovat dobu čekání
    - try_lock_for a try_lock_until
  - důležitá je šablona std::lock_guard
    - zajistí, že daný blok kódu bude hlídán mutexem
    - a že bude kritická sekce opuštěna vždy, když vykonávaný kód opustí daný blok kódu
      - a to včetně zpracování vyjímek
      - => nemusí se tedy na konec try a do catch dávat unlock

```
std::mutex m;
unsigned counter=0;

unsigned increment() {
    std::lock_guard<std::mutex> lk(m);
    return ++counter;
}

unsigned query(){
    std::lock_guard<std::mutex> lk(m);
    return counter;
}
```

- Šablona std::unique_lock
  - Umožňuje vytvořit zámek, který se zamkne, až mu řekneme

```
#include <mutex>
#include <thread>
#include <chrono>

struct Box {
    explicit Box(int num) : num_things{num} { }

    int num_things;
    std::mutex m;
};

void transfer(Box &from, Box &to, int num){
    // don't actually take the locks yet
    std::unique_lock<std::mutex> lock1(from.m,
                                       std::defer_lock);
    std::unique_lock<std::mutex> lock2(to.m,
                                       std::defer_lock);

    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    lock1.unlock();
    lock2.unlock();
}

int main() {
    Box acc1(100);
    Box acc2(50);
    std::thread t1(transfer, std::ref(acc1),
                             std::ref(acc2), 10);
    std::thread t2(transfer, std::ref(acc2),
                             std::ref(acc1), 5);

    t1.join();
    t2.join();
}
```

- Podmínkové proměnné
  - Condition_variable
  - notify/All a wait/for/until
  - wait bere jako parametr zámek
    - zámek musí být zamčený v době volání wait

```
std::condition_variable cv;
std::mutex cv_m;
int i = 0;

void waits() {
    std::unique_lock<std::mutex> lk(cv_m);
    std::cerr << "Waiting... \n";
    cv.wait(lk, [](){return i == 1;});
    std::cerr << "...finished waiting. i == 1\n";
}

void signals() {

std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cerr << "Notifying...\n";
    cv.notify_all();

std::this_thread::sleep_for(std::chrono::seconds(1));
    i = 1;
    std::cerr << "Notifying again...\n";
    cv.notify_all();
}
```

- Atomické operace a inicializace
  - call_once zavolá funkci jenom jednou, i kdyby byla volána z více vláken
  - viz šablony std::atomic*

- Vyhnutí se deadlocku
  - std::lock – viz příklad výše, viz priority vidliček večeřících filozofů

- Asynchronní volání
  - std::async vykoná kód funkce nezávisle na vláknu, ze kterého byla std::async zavolána
    - nezaručuje, že se vykoná v jiném vláknu
  - std::future vrací hodnotu std::async výpočtu

```
#include <future>
#include <iostream>

int calculate_the_answer_to_LtUaE();
void do_stuff();

int main()
{
    std::future<int>
the_answer=std::async(calculate_the_answer_to_LtUaE);
    do_stuff();
    std::cout<<"The answer to life, the universe and
                everything is "
             <<the_answer.get()<<std::endl;
}
```

- Komplikovaněji, ale zato s větší kontrolou nad vlákny, je možnost použití std::promise
  - Např. pro I/O operace
  - Promise je svázán s future
    - Jeden thread pomocí set_value zapíše výsledek
    - A druhý thread si ho pak vyzvedne
    - Promise je v podstatě úložiště hodnoty, dokud si ji někdo nevyzvedne přes svázanou future
      - Producent vloží hodnotu do promise
      - Konzument si vyzvedne hodnotu z future

```
typedef int (*calculate)(void);
void func2promise(calculate f, promise<int> &p)
{
  p.set_value(f());
}

int main(int argc, char *argv[])  {
  getUserData();
  promise<int> p1, p2;
  future<int> f1 = p1.get_future(),
              f2 = p2.get_future();

  thread t1(&func2promise, calculateB,
                           std::ref(p1)),
         t2(&func2promise, calculateC,
                           std::ref(p2));

  c = (calculateA() + f1.get()) * f2.get();
  t1.join();
  t2.join();

  showResult();
}
```

- Další čtení a příklady převzaty z:
  - http://en.cppreference.com/w/cpp/thread
  - http://en.cppreference.com/w/cpp/atomic
  - http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.html
  - http://msdn.microsoft.com/en-us/magazine/hh852594.aspx