Семестр 2. Лекция 9. STL C++11. Threads

Евгений Линский

4 Мая 2018

unordered_set, unordered_map

```
template<class Key,
         class Hash = std::hash<Key>,
         class KeyEqual = std::equal_to<Key>,
         class Allocator = std::allocator<Key>>
    class unordered_set;
```

- Типовая реализация: карманы и списки
- Для своего класса нужно реализовать сравнение на равенство и хэш функцию
- *find* is O(1) but O(n) worst case (все элементы оказались в одном кармане)
- *insert* is O(1) but O(n) worst case (может вызвать рехэширование — “увеличение числа карманов”)

unordered_set, unordered_map

```
class Point {
private:
    int x, y;
public:
    bool operator == (const Point& rhs) const{
        return x == rhs.x && y == rhs.y;
    }
};
namespace std {
    template <>
    struct hash<Point> {
        size_t operator()(Point const & p) const {
            return (std::hash<int>()(p.getX()) * 51
                + std::hash<int>()(p.getY()));
        }
    };
}
std::unordered_set<Point> points;
//Можно реализовать функторы PointComparator, PointHasher
std::unordered_set<Point, PointComparator, PointHasher> ps;
```

unique_ptr, shared_ptr

```
struct Foo {
    ...
    void bar() { std::cout << "Foo::bar\n";  }
};

void f(const Foo &) { std::cout << "f(const Foo&)\n"; }

int main() {
    std::unique_ptr<Foo> p1(new Foo);  // p1 owns Foo
    if (p1) p1->bar();

    // now p2 owns Foo
    std::unique_ptr<Foo> p2(std::move(p1));
    f(*p2);

    p1 = std::move(p2);  // ownership returns to p1
    if (p1) p1->bar();

    // p1 = p2; // error, assign operator is =delete
}
```

other

- контейнер (т.е. есть итераторы) template<class T, std::size_t N> struct array;

```
std::array<int, 3> a2 = {1, 2, 3};
```

- std::tuple — "как pair", но только с произвольным числом полей (реализован на variadic templates)

```
auto t = std::make_tuple("String", 5.2, 1);
std::cout << std::get<0>(t) << ' '
          << std::get<1>(t) << ' '
          << std::get<2>(t) << '\n';
```

- regex, regex_search, regex_match

```
regex reg("[a-zA-Z_][a-zA-Z_0-9]*\\.[a-zA-Z0-9]+");
regex_search("Print readme.txt", reg);
```

std::function

std::function — "шаблонная обертка" для всего, что можно вызвать (callable): функция, функтор, лямбда.

```
void execute(const vector<function<void ()>>& fs){
    for (auto& f : fs) f();
}
void plain_old_func() {
    cout << "old plain function" << endl;
}
struct functor {
    void operator()() const{
        cout << "functor" << endl;
    }
};
int main() {
    vector<function<void ()>> x;
    x.push_back(plain_old_func);
    functor functor_instance;
    x.push_back(functor_instance);
    x.push_back([] () { cout << "lambda" << endl; });
    execute(x);
}
```

std::bind

std::bind — позволяет создать “обертку” над функцией, уменьшив число ее параметров (“удаленным” параметрам задаются конкретные значения).

```
void show_text(const string& t) {
    cout << "TEXT: " << t << endl;
}

int main() {
    vector<function<void ()>> x;
    // void f() { show_text("Hello"); }
    function <void ()> f = bind(show_text, "Hello");
    x.push_back(f);
    execute(x);
}
```

std::bind, placeholder

```
using namespace std::placeholders;
int multiply(int a, int b) { return a * b; }
int main() {
  //_1 -- placeholder;
  // 1ый параметр функции f подставляется
  // на второе место в multiply
  // int f(int _1) { return multiply(5, _1); }
  auto f = bind(multiply, 5, _1);
  cout << out: << f(6); // out:  30
}
```

Неформальные воспоминания про потоки 1

- Процессы (программы)
  - “несколько программ запущены одновременно”
  - независимые адресные пространства
- Потоки (функции в программе)
  - ”несколько функций внутри одной программы запущены одновременно в разных потоках”
  - общее адресное пространство (могут иметь доступ к общим переменным)

std::thread

```
void f1(int n){
    std::cout << "f1: " << n << std::endl;
}

void f2(int& n){
    n++;
}

int main(){
    int m = 45;
    std::thread t1(f1, m);
    // std::ref нужен, чтобы у функции, выполняемой
    // потоком была тип void f(int &x)
    std::thread t2(f2, std::ref(m));
    t1.join();
    t2.join();
    cout << m;
}
```

std::thread

Пример с лямбдой.

```
std::vector<std::thread> threads;

for(int i = 0; i < 5; ++i){
    threads.push_back(std::thread([](){
        std::cout << std::this_thread::get_id()
        << std::endl;
    }));
}

for(auto& thread : threads){
    thread.join();
}
```

А можно передать фунтор и сохранить в нем результат исполнения.

Пример. Параллельное сложение векторов

Что с производительностью на одном процессоре (с одним ядром)?

```
typedef vector<int> ivec;
void sum_vec(const ivec& v1, size_t start, size_t end,
    const ivec& v2, ivec& res){
    for(int i = start; i < end; i++) { res[i] = v1[i] + v2[
}
}
void parallel_sum_vec(const ivec& v1, const ivec& v2,
    ivec& res) {
    thread t1(sum_vec, (size_t)0, v1.size() / 2,
      cref(v1), cref(v2), ref(res));

    thread t2(sum_vec, v1.size() / 2, v1.size(),
      cref(v1), cref(v2), ref(res));
    t1.join(); t2.join();
}
parallel_sum_vec(v1, v2, res);
```

- std::cref — const reference
- g++ –std=c++11 vec_sum.cpp -o vs -lpthread

Неформальные воспоминания про потоки 2

- Переключением между процессами/потоками занимается планировщик ОС
- При переключении с потока 1 на поток 2 необходимо все регистры, используемые потоком 1 сохранить в память, а все регистры, используемые потоком 2 восстановить из памяти (переключение контекста). Но:
  - Бывает программа не использует процессор (ввод-вывод, ожидание).
  - А еще бывает несколько процессоров. =)
- NB (след. слайд): С точки зрения процесса/потока переключение происходит в произвольный момент времени!

Вывод из потоков.

```
void hello(){
    std::cout << "Hello from thread " <<
      std::this_thread::get_id() << std::endl;
}

int main(){
    std::vector<std::thread> threads;

    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread(hello));
    }

    for(auto& thread : threads){
        thread.join();
    }

    return 0;
}
```

Гонки (Race conditions).

Может так:

```
Hello from thread 140276650997504
Hello from thread 140276667782912
Hello from thread 140276659390208
Hello from thread 140276642604800
Hello from thread 140276676175616
```

А может и так:

```
Hello from thread Hello from thread Hello from
thread 139810974787328Hello
from thread 139810983180032Hello from thread
139810966394624
139810991572736
139810958001920
```

Гонки (Race conditions). Еще примеры.

- Примитивные типы (int, ...) неатомарны (атомарный — операция не может быть прервана)
- Связный список: поток 1 вставляет элементы в середину, поток 2 выводит элементы на экран. “Контекст переключился” с 1 на 2, когда не все указатели next были проставлены.
- Еще классический пример

```
1 int x = 0;
2 // Thread 1:
3 while (!stop) {
4   x++;
5 }
6 // Thread 2:
7 while (!stop) {
8   if (x%2 == 0) cout<< x;
9 }
```

На экране может быть нечетное число. Почему?

Гонки (Race conditions). Проблема.

```
struct Counter {
    int value;
    Counter() : value(0){}
    void increment(){
        ++value;
    }
};
```

```
Counter counter;
vector<thread> threads;
for(int i = 0; i < 5; ++i){
    threads.push_back(thread([&counter](){
        for(int i = 0; i < 100; ++i){
            counter.increment();
        }
    }));
}
for(auto& thread : threads){ thread.join();}
cout << counter.value << endl;
```

Гонки (Race conditions). Вывод.

```
442
500
477
400
422
487
```

Причина (одна из):
Thread 1 : read the value, get 0, add 1, so value = 1
Thread 2 : read the value, get 0, add 1, so value = 1
Thread 1 : write 1 to the field value and return 1
Thread 2 : write 1 to the field value and return 1

Мутексы.

```
struct Counter {
    mutex mutex;
    int value;

    Counter() : value(0) {}

    void increment(){
        mutex.lock();
        ++value;
        mutex.unlock();
    }
};
```

- Внутри блока std::mutex.lock(); ... std::mutex.unlock(); может находиться только один поток, остальные будут ожидать.
- NB: для синхронизации одной переменной примитивного типа хватило бы std::atomic, а вот для линейного списка mutex бы подошел.
- std::lock_guard — RAII обертка для mutex (типа scoped_ptr)