

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 14

15.12.2017

Многопоточность: non-joinable

1. Объекты `std::thread`, сконструированные конструктором по умолчанию.
2. Объекты `std::thread`, для которых выполнена функция `join`.
3. Объекты `std::thread`, для которых выполнена функция `detach`.
4. Объекты `std::thread`, из которых выполнено перемещение.

Многопоточность: joinable + деструктор

При вызове деструктора для joinable объекта программа завершает работу (terminate).

А что она могла бы сделать?

1. Неявный join.
2. Неявный detach.

Последствия плохие, поэтому запрещено.

Многопоточность: joinable + деструктор

При вызове деструктора для joinable объекта программа завершает работу (terminate).

А что она могла бы сделать?

1. Неявный join.
2. Неявный detach.

Последствия плохие, поэтому запрещено.

Но можно написать обертку.

Обертка для std::thread

```
class TMyThread {
public:
    enum class TFinishAction { join, detach };
private:
    TFinishAction finishAction;
    std::thread t;
public:
    TMyThread(std::thread&& t, TFinishAction a)
        : finishAction(a)
        , t(std::move(t))
    {}
    ~TMyThread() {
        if (t.joinable()) {
            if (finishAction == TFinishAction::join) {
                t.join();
            } else {
                t.detach();
            }
        }
    }
    std::thread& Get() {return t;}
};
```

Задача

Функция `run` выполняет задачу в потоке.

```
run([]{  
    A a = f();  
    B b = g();  
    h(a, b);  
});
```

Используя функцию `run`, переписать код так, чтобы `f` и `g` выполнялись параллельно.

const-методы

Ожидается, что константные методы класса (так как они представляют операцию чтения) будут потокобезопасны.

const-методы

Ожидается, что константные методы класса (так как они представляют операцию чтения) будут потокобезопасны.

```
class A {
    public:
        double GetScore() const {
            if (!done) {
                score = Do();
                done = True;
            }
            return score;
        }
    private:
        mutable bool done {false};
        mutable double score {0.0};
};
```


const-методы

```
class A {  
    public:  
        double GetScore() const {  
            std::lock_guard<std::mutex> l(m);  
            if (!done) {  
                score = Do();  
                done = True;  
            }  
            return score;  
        }  
    private:  
        mutable bool done {false}; // кэширующий флаг  
        mutable double score {0.0};  
        mutable std::mutex m;  
};
```

const-методы

```
class A {
    public:
        double GetScore() const {
            std::lock_guard<std::mutex> l(m);
            if (!done) {
                score = Do();
                done = True;
            }
            return score;
        }
    private:
        mutable bool done {false}; // кэширующий флаг
        mutable double score {0.0};
        mutable std::mutex m;
};
```

Замечание: `std::mutex` не является ни копируемым ни перемещаемым.

C++14: новые возможности языка

Некоторые особенности новых стандартов.

C++14: новые возможности языка

Некоторые особенности новых стандартов.

- ▶ auto в объявлениях параметров функции

Правила вывода типа такие же, как и в случае шаблонов.

```
auto f = [&v](const auto& nv) {v = nv;}  
f({1, 2, 3}); // error
```

Лямбды и std::bind

```
class A {  
    public:  
        template <typename T>  
        void operator() (const T& x) const;  
};  
// ...
```

```
A a;  
auto boundA = std::bind(a, _1);
```

Теперь можно вызвать с разными типами аргументов:

```
boundA("123");  
boundA(123);
```

Через lambda-выражения не получится в C++11.

Лямбды и std::bind

```
class A {  
    public:  
        template <typename T>  
        void operator() (const T& x) const;  
};  
// ...
```

```
A a;  
auto boundA = std::bind(a, _1);
```

Теперь можно вызвать с разными типами аргументов:

```
boundA("123");  
boundA(123);
```

Через lambda-выражения не получится в C++11.

Но получится в C++14:

```
auto boundA = [a](const auto& x) {a(x);};
```

C++11: вывод возвращаемого типа

C++11: auto означает, что возвращаемый тип функции будет объявлен после списка параметров.

```
template <typename T, typename I>
auto
Get(T&& a, I i)
-> decltype(std::forward<T>(a)[i]) {
    // ...
    return std::forward<T>(a)[i];
}
```

C++14: вывод возвращаемого типа

C++14: auto означает, что имеет место вывод типа.

```
template <typename T, typename I>
auto Get(T&& a, I i) {
    // ...
    return std::forward<T>(a)[i];
}
```

Для функции с auto в возвращаемом значении применяется вывод типа шаблона, где игнорируется, ссылочность.

А нам нужно возвращать в точности тот же тип, что и `a[i]`.

Поэтому:

C++14: вывод возвращаемого типа

C++14: auto означает, что имеет место вывод типа.

```
template <typename T, typename I>
auto Get(T&& a, I i) {
    // ...
    return std::forward<T>(a)[i];
}
```

Для функции с auto в возвращаемом значении применяется вывод типа шаблона, где игнорируется, ссылочность.

А нам нужно возвращать в точности тот же тип, что и a[i].

Поэтому:

```
template <typename T, typename I>
decltype(auto) Get(T&& a, I i) {
    // ...
    return std::forward<T>(a)[i];
}
```

C++14: decltype(auto)

```
decltype(auto) f() {  
    int x;  
    // ...  
    return x;      // -> int  
}
```

```
decltype(auto) g() {  
    int x;  
    // ...  
    return (x);    // -> int&  
}
```

C++17: НОВЫЕ ВОЗМОЖНОСТИ

Уже упоминалось:

- ▶ `template <auto>`
- ▶ `std::optional`
- ▶ `std::variant`

Что ещё?

C++17: `emplace_back`

Было:

```
v.emplace_back(10);  
auto& val = v.back();
```

Стало:

```
auto& val = v.emplace_back();
```

C++17: std::string_view

Хранит размер строки и указатель на её начало.

```
void process(std::string_view sv);
```

```
process("some_string");
```

```
process(std_string);
```

C++17: std::string_view

Хранит размер строки и указатель на её начало.

```
void process(std::string_view sv);

process("some_string");
process(std_string);

std::map <std::string_view, int> m;

void add(const std::string &str, int val) {
    m[str] = val;    // плохо
}
```

C++17: std::string_view

Хранит размер строки и указатель на её начало.

```
void process(std::string_view sv);

process("some_string");
process(std_string);

std::map <std::string_view, int> m;

void add(const std::string &str, int val) {
    m[str] = val;    // плохо
}
```

std::string_view в std::string явно не конвертируется.

C++17: вложенные пространства имён

```
namespace Namespace :: Nested {  
}
```


C++17: вложенные пространства имён

```
namespace Namespace :: Nested {  
}
```

А ещё может быть так:

```
namespace Namespace :: Nested {  
    namespace NestedDetailed {  
        // (a)  
    }  
    // (b)  
}
```

ВМЕСТО

```
namespace Namespace {  
    namespace Nested {  
        namespace NestedDetailed {  
            // (a)  
        }  
        // (b)  
    }  
}
```

C++17: указание правил вывода типов

Вывод типов шаблонных параметров для классов:

```
template <typename T, typename U>
struct Pair {
    Pair(T t, U u)
        : first (std::move(t))
        , second(std::move(u))
    {}
    T first;
    U second;
};

int main() {
    Pair p(1, 2.3);
}
```

C++17: указание правил вывода типов

Вывод типов шаблонных параметров для классов:

```
template <typename T, typename U>
struct Pair {
    Pair(T t, U u)
        : first (std::move(t))
        , second(std::move(u))
    {}
    T first;
    U second;
};
```

```
int main() {
    Pair p(1, 2.3);
}
```

Если конструктора нет, можно явно указать:

```
template <typename T, typename U>
Pair(const T& t, const U& u) -> Pair<T, U>;
```

C++17: указание правил вывода типов

А если так:

```
template <typename T, typename U>
struct Pair {
    template <typename A, typename B>
    Pair(A&& t, B&& u)
        : first (std::forward<A>(t))
          , second(std::forward<B>(u))
    {}
    T first;
    U second;
};
```

Тогда:

```
template <typename T, typename U>
Pair(T&& t, U&& u) -> Pair<std::decay_t<T>, std::decay_t<U>>;
```

C++17: if constexpr

(метапрограммирование)

```
if constexpr (/*constant expression */) {  
    // if true this block is compiled  
} else {  
    // if false this block is compiled  
}
```

C++17: noexcept

```
void f(void(*fptr)() noexcept);  
void g();
```

`f(&g);` // ошибка в C++17, но не в C++14

Спецификация исключений из функций окончательно удалена, остается лишь noexcept.

C++17: if-init выражения

Было:

```
auto x = get_result();  
if (x == 1) {  
}
```

Стало:

```
if (auto x = get_result(); x == 1) {  
}
```

Читаемость?..

C++17: структурное связывание

Было:

```
for (const auto &v : get_pairs()) {  
    // do smth with v.first, v.second  
}
```

Стало:

```
for (const auto &[key, value] : get_pairs()) {  
    // do smth with key, value  
}
```


Вопрос

Что лучше?

```
std::string f() {  
    auto s = get_result();  
    return s;  
}
```

или

```
std::string f() {  
    auto s = get_result;  
    return std::move(s);  
}
```

Вопрос

Что лучше?

```
std::string f() {  
    auto pair = get_result();  
    return pair.first;  
}
```

или

```
std::string f() {  
    auto pair = get_result;  
    return std::move(pair.first);  
}
```

Вопрос

Что лучше?

```
std::string f() {  
    auto [s, v] = get_result();  
    return s;  
}
```

или

```
std::string f() {  
    auto [s, v] = get_result;  
    return std::move(s);  
}
```