Стандарт C++ 11/14: безопасность и многопоточность

Александр Смаль

CS центр
6 апреля 2016
Санкт-Петербург

Ключевое слово `noexcept`

- Используется в двух значениях:
  1. Спецификатор функции, которая не бросает исключение.
  2. Оператор, проверяющий во время компиляции, что выражение специфицированно как небросающее исключение.
- Если функцию со спецификацией `noexcept` покинет исключение, то стек не обязательно будет свёрнут, перед тем как программа завершится.
  (В отличие от аналогичной ситуации с `throw()`.)
- Использование спецификации `noexcept` позволяет делать функции более оптимизируемыми, т.к. компилятору не нужно заботиться о сворачивании стека.

Использование noexcept

```
void may_throw();
void no_throw() noexcept;

struct T { ~T(){} /* copy ctor is noexcept */ };
struct U {
  ~U(){} /* copy ctor is noexcept(false) */
  std::vector<int> v;
};
struct V {  std::vector<int> v; };
```

```
T t; U u; V v;

noexcept(may_throw())             == false;
noexcept(no_throw())              == true;
noexcept(std::declval<T>().~T())  == true;
noexcept(T(std::declval<T>()))    == true;
noexcept(T(t))                    == true;
noexcept(U(std::declval<U>()))    == false;
noexcept(U(u))                    == false;
noexcept(V(std::declval<V>()))    == true;
noexcept(V(v))                    == false;
```

Условный noexcept

В спецификации noexcept можно указывать условные выражения времени компиляции.

```
template <class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;

    void swap(pair & p) noexcept(noexcept(swap(first, p.first)) &&
                                 noexcept(swap(second, p.second)));
    ...
};
```

Зависимость от noexcept

Проверка noexcept используется в стандартной библиотеке для обеспечения строгой гарантии исключений с помощью std::move_if_noexcept (например, vector::push_back).

```
struct Bad {
    Bad() {}
    Bad(Bad&&);  // may throw
    Bad(const Bad&); // may throw as well
};
struct Good {
    Good() {}
    Good(Good&&) noexcept; // will NOT throw
    Good(const Good&) noexcept; // will NOT throw
};
int main() {
    Good g;
    Bad b;
    Good g2 = std::move_if_noexcept(g); // move
    Bad b2 = std::move_if_noexcept(b);  // copy
}
```

Потоки

В стандартной библиотеке есть два способа выполнения задач асинхронно:

- создавая поток вручную `std::thread`
- используя `std::async`.

```
int doAsyncWork();

// прямое создание потока
std::thread t(doAsyncWork);

// использование std::async
std::future<int> fut = std::async(doAsyncWork);
int res = fut.get();
```

Использование `std::async` позволяет в некоторых случаях (зависит от планировщика) отложить выполнение задачи до вызова `get` или `wait`.

std::async

- Имеет две стратегии выполнения: асинхронное выполнение и отложенное (синхронное) выполнение.
  1. `std::launch::async`
  2. `std::launch::deferred`
- По умолчанию имеет стратегию:
  `std::launch::async | std::launch::deferred`
- Отложенная задача может никогда не выполнится, если не будет вызвано `get` или `wait`.
- Возвращает `std::future<T>`, который позволяет получить доступ к возвращаемому значению.

std::thread

- Если в конструктор передать функцию, то она сразу же начинает выполняться.
- Метод `t.join()` позволяет заблокировать текущий поток, пока выполнение потока `t` не завершится.
- Метод `t.detach()` позволяет отключить поток от объекта, т.е. разорвать связь между объектом и созданным потоком.
- При вызове деструктора неподключаемого потока программа завершается, т.е. мы обязаны вызвать либо `join`, либо `detach`.
- Позволяет получить платформенно-зависимый дескриптор: `t.native_handle()`.

Синхронизация

Для синхронизации применяются объекты `std::mutex`.

```
std::mutex mtx;           // mutex for critical section

void print_block (int n, char c) {
  // critical section (exclusive access to std::cout signaled by locking mtx):
  mtx.lock();
  for (int i=0; i<n; ++i) { std::cout << c; }
  std::cout << '\n';
  mtx.unlock();
}

int main ()
{
  std::thread th1 (print_block,50,'*');
  std::thread th2 (print_block,50,'-');

  th1.join();
  th2.join();
}
```

Синхронизация: std::lock_guard

Для std::mutex определена RAII обёртка std::lock_guard.

```
std::mutex mtx;           // mutex for critical section

void print_block (int n, char c) {
  // critical section (exclusive access to std::cout signaled by locking mtx):
  std::lock_guard<std::mutex> lck (mtx);
  for (int i=0; i<n; ++i) { std::cout << c; }
  std::cout << '\n';
}

int main ()
{
  std::thread th1 (print_block,50,'*');
  std::thread th2 (print_block,50,'-');

  th1.join();
  th2.join();
}
```

std::atomic

- Шаблон `std::atomic` позволяет определить переменную, операции с которой будут атомарны.
- Определён только для целочисленных встроенных типов и указателей.

```
template<class T>
struct shared_ptr_data {
    void addref() {
        ++counter; // atomic increment
    }

    T * ptr;
    std::atomic<size_t> counter;
};
```

Общие советы

- Делайте константные функции-члены безопасными в смысле потоков (например, при кешировании).
- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения.
- Используйте `std::atomic` вместо мьютекса в тех случаях, когда нужно синхронизация нужна только для одной целочисленной переменной.
- `volatile` — это не про многопоточность.