Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 12
04.12.2019

Исключения стандартной библиотеки

Стандартная библиотека предоставляет иерархию классов исключений. Базовый класс `std::exception`.

- `logic_error`
  - `invalid_argument`
  - `domain_error`
  - `length_error`
  - ...
- `runtime_error`
  - `overflow_error`
  - `range_error`
  - ...
- `bad_weak_ptr`
- `bad_cast`
- ...

std::exception_ptr (C++11)

- Хранит в себе исключения и имеет семантику указателя.
- Сохранить исключение можно через `std::current_exception`.
- Информация о типе исключения не теряется.
- Повторно сгенерировать исключение из этого объекта можно через `std::rethrow_exception`.

```
int main() {
    std::exception_ptr eptr;
    try {
        throw std::out_of_range("some info");
    } catch (...) {
        eptr = std::current_exception();
    }
    //...
    if (eptr) {
        std::rethrow_exception(eptr);
    }
}
```

Value Or Exception

`std::optional` не работает с исключениями. Создадим класс, объекты которого хранят в себе либо значение, либо `std::exception_ptr` сгенерированного исключения.

Как создать тип, объекты которого хранят значения одного из заданных типов?

- `union`
- `std::variant` (C++17)

std::variant (C++17)

Это такой union, который знает, какой именно тип он хранит.

```
std::variant<int, char, double> v;
v = 5;
std::cout << std::get<int>(v);

// std::cout << std::get<double>(v);
// исключение std::bad_variant_access

// std::cout << std::get<float>(v);
// не компилируется

auto p = std::get_if<char>(&v); // nullptr
```

Value Or Exception

```
template <typename T>
class TValueOrError {
    std::variant<T, std::exception_ptr> valueOrError;
  public:
    TValueOrError(std::exception_ptr eptr) : valueOrError(eptr) {}
    TValueOrError(T&& val) : valueOrError(std::move(val)) {}
    TValueOrError(const T& val) : valueOrError(val) {}

    bool IsValue() const { return valueOrError.index() == 0; }
    bool IsError() const { return !IsValue(); }

    const T& GetValueOrThrow() const {
        if (IsValue()) {
            return std::get<0>(valueOrError);
        }
        std::rethrow_exception(std::get<1>(valueOrError));
    }
};
```

Value Or Exception

Схема использования:

```
TValueOrError<int> GetResult(int param) {
    try {
        if (param == 0) {
            throw std::logic_error("param cant be zero");
        } else {
            return param / 2;
        }
    } catch (...) { return std::current_exception(); }
}

int main() {
    try {
        auto result = GetResult(0);
        std::cout << result.IsError() << std::endl;
        auto r = result.GetValueOrThrow();
        // use it ...
        std::cout << "result = " << r << std::endl;
    } catch (std::logic_error& e) {
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

Value Or Exception

Что делать, если нужно прям по месту, не кидая исключения, записать исключение?

```
try {
    throw std::logic_error("logic error");
} catch (...) {
    TValueOrError<int> error = std::current_exception();
    std::cout << error.IsError() << std::endl;
}
```

Value Or Exception

Что делать, если нужно прям по месту, не кидая исключения, записать исключение?

```
try {
    throw std::logic_error("logic error");
} catch (...) {
    TValueOrError<int> error = std::current_exception();
    std::cout << error.IsError() << std::endl;
}
```

Чтобы так не делать, существует `std::make_exception_ptr`:

```
TValueOrError<int> error =
    std::make_exception_ptr(std::logic_error("logic error"));
std::cout << error.IsError() << std::endl;
```

noexcept

```
void f(int param) noexcept;
```

- Модификатор означает, что функция гарантированно не генерирует исключений.
- От этого может зависеть эффективность вызывающего кода.
- Допускается вызов из noexcept-функции других функций, которые noexcept не являются.
- В C++11 все функции освобождения памяти и все деструкторы неявно являются noexcept.

Гарантии безопасности исключений

1. Гарантия отсутствия исключений.
2. Строгая гарантия (исключения могут происходить, но все объекты остаются в согласованном и предсказуемом состоянии).
3. Базовая гарантия (исключения могут происходить, объекты остаются в согласованном состоянии, но не обязательно в предсказуемом).

Гарантии безопасности исключений

1. Гарантия отсутствия исключений.
2. Строгая гарантия (исключения могут происходить, но все объекты остаются в согласованном и предсказуемом состоянии).
3. Базовая гарантия (исключения могут происходить, объекты остаются в согласованном состоянии, но не обязательно в предсказуемом).

Пример: стек и операция `pop`.

- Согласованность означает соответствие size() и числа элементов при исключении.
- Предсказуемость означает, что при исключении число элементов в стеке не уменьшилось.

Пример

Как можно изменить код класса, чтобы избавиться от вызовов конструктора копирования без изменения клиентского кода?

```
class A {
  public:
    A() { /*...*/ }
    A(const A&) { /*...*/ }
    A(A&&) { /*...*/ }
};

void Get(size_t, A&) { /*...*/ }
int main() {
    std::vector<A> v;
    A a;
    for (size_t i = 0; i < 100; ++i) {
        Get(i, a);
        v.push_back(a);
    }
}
```

Пример

Метод `resize` у вектора в случае реаллокации предоставляет строгую гарантию безопасности исключений.

```
class A {
  public:
    A() { /*...*/ }
    A(const A&) { /*...*/ }
    A(A&&) noexcept { /*...*/ }
};

void Get(size_t, A&) { /*...*/ }
int main() {
    std::vector<A> v;
    A a;
    for (size_t i = 0; i < 100; ++i) {
        Get(i, a);
        v.push_back(a);
    }
}
```

Assert

Проверяем предположения о данных объекта в программе, проверяем инварианты.

Assert

Проверяем предположения о данных объекта в программе, проверяем инварианты.

- assert времени компиляции

Программа не компилируется, если условие не выполняется.

```
static_assert(
    std::is_pointer<decltype(TCalcerPtr)>::value,
    "must be pointer"
);
```

Assert

- assert времени выполнения

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /*implementation defined*/
#endif
```

А полезно ли это?

Итераторы

Iterators describe types that can be used to identify and traverse the elements of a container.

- Итераторы ввода. Поддерживают только две операции – разыменование для чтения и сдвиг вперед (++ в двух вариантах). Типичный пример – считывание из потока ввода.
- Итераторы вывода. Тоже две, только разыменование для записи. Аналогия с потоком вывода.
- Однонаправленный итератор. Объединение первых двух. Можно разыменовывать для чтения и записи, движение только вправо. Возникает в односвязном списке.
- Двунаправленный итератор. Может быть перемещен в обоих направлениях. Возникает в двусвязном списке. Можно разыменовывать, читать и писать, сдвигать на одну позицию в любом направлении.
- Итератор произвольного доступа. Это вектор, дек, строка и обыкновенный массив.
- C++17: ContiguousIterator.

std::iterator_traits

Iterators describe types that can be used to identify and traverse the elements of a container.

- `difference_type` – a signed integer type that can be used to identify distance between iterators
- `value_type` – the type of the values that can be obtained by dereferencing the iterator. This type is void for output iterators.
- `pointer` – defines a pointer to the type iterated over (value_type)
- `reference` – defines a reference to the type iterated over (value_type)
- `iterator_category` – the category of the iterator. Must be one of iterator category tags.

std::iterator_traits

```
#include <iterator>

template<class BidirIt>
void f(BidirIt first, BidirIt last)
{
    typename std::iterator_traits<BidirIt>::difference_type n =
        std::distance(first, last);
    if (n > 0) {
        typename std::iterator_traits<BidirIt>::value_type t =
            *first;
        // ...
    }
}
```

Диспетчеризация категориями итераторов

```
template< class BDIter >
void f(BDIter, BDIter, std::bidirectional_iterator_tag)
{
    std::cout << "f() called for bidirectional iterator\n";
}

template <class RAIter>
void f(RAIter, RAIter, std::random_access_iterator_tag)
{
    std::cout << "f() called for random-access iterator\n";
}

template< class Iter >
void f(Iter first, Iter last)
{
    f(first, last,
      typename std::iterator_traits<Iter>::iterator_category());
}
```

std::iterator

Как написать свой итератор? Использовать std::iterator.

```
template<
    typename Category,
    typename T,
    typename Distance = std::ptrdiff_t,
    typename Pointer = T*,
    typename Reference = T&
> struct iterator
{
    using iterator_category = Category;
    using value_type = T;
    using difference_type = Distance;
    using pointer = Pointer;
    using reference = Reference;
};
```

std::iterator

Как написать свой итератор? Использовать std::iterator.

```
template<
    typename Category,
    typename T,
    typename Distance = std::ptrdiff_t,
    typename Pointer = T*,
    typename Reference = T&
> struct iterator
{
    using iterator_category = Category;
    using value_type = T;
    using difference_type = Distance;
    using pointer = Pointer;
    using reference = Reference;
};
```

Deprecated in C++17.

std::iterator

На самом деле устаревшим объявлено наследование от std::iterator.

```
class TMyIterator : public
    iterator<forward_iterator_tag, int, int, int*, int&>
```

vs

```
class TMyIterator {
  public:
    using iterator_category = forward_iterator_tag;
    using value_type = int;
    using difference_type = int;
    using pointer = int*;
    using reference = int&;
    // ...
```

std::iterator

Что важно:

- В определении std::iterator было явно зафиксировано три значения по умолчанию. Их придется писать явно.

```
class TMyIterator
    : public std::iterator<std::forward_iterator_tag, int>
```

- Если последние должны быть void, то их можно пропустить.

```
class TOutputIterator
: public std::iterator<std::output_iterator_tag, void, void,
                                                 void, void>

class TOutputIterator {
  public:
    using iterator_category = std::output_iterator_tag;
    // ...
```