Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 10
16.11.2018

Шаблоны

Параметр шаблона, в частности, может не быть типом:

```
template <int N>
struct A;
```

Если между классами есть зависимости, то компилятор будет пытаться вычислять зависимости на стадии компиляции, в частности, на стадии компиляции можно заставить компилятор выполнять нетривиальные задачи.

Отсутпление в сторону

В C++17 можно делать так:

```
template<auto n>
void f() {
    // ...
}

int main()
{
    f<3>();    // int
    f<'a'>(); // char
}
```

А как это сделать в C++11?

Шаблонное метапрограммирование

Вычисления на этапе компиляции:

```
#include <iostream>

template <int Base, int N>
struct Power {
  enum {result = Power<Base, N - 1> ::result * Base };
};

template <int Base>
struct Power <Base, 0> {
  enum {result = 1 };
};

int main () {
  std::cout << Power<2,10> :: result;
}
```

Вычисления на шаблонах

Что напечатает программа?

```
#include <iostream>

template <typename T>
struct A {
    static const int res = 1;
};

template <typename T>
struct A<T*> {
    static const int res = 1 + A<T>::res;
};

int main () {
    std::cout << A<int***> :: res << std::endl;
}
```

Задача

Напишите шаблон для вычисления C_n^k на этапе компиляции.

Шаблонное метапрограммирование

В C++11 появляется `std::integral_constant` — обёртка над статическими константами.

```
template<class T, T v>
struct integral_constant {
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant type;
    constexpr operator value_type() const noexcept {
        return value;
    }
    constexpr value_type operator()() const noexcept {
        return value;
    }
};
```

std::integral_constant

И теперь можно написать так:

```
template <typename X, typename Y>
using sum = std::integral_constant<
                        decltype(X::value + Y::value),
                        X::value + Y::value
            >;

using result =
    sum<std::integral_constant<int, 3>,
        std::integral_constant<int, 4>>;
```

User-defined literals

Возможно определить пользовательские литералы:

- целочисленные литералы `1_km`,
- float-литералы `0.5_rad`,
- символьные литералы `'a'_X`
- строковые литералы `"abc"_Q`

Все суффиксы должны начинаться с подчеркивания, суффиксы стандартной библиотеки не начинаются с подчеркивания.

User-defined literals

```
constexpr long double operator"" _deg (long double deg)
{
    return deg * 3.1415 / 180;
}

constexpr unsigned long long operator"" _MB(
    unsigned long long mb
) {
    return static_cast<size_t>(mb) << 20;
}

static const auto MEMORY_LIMIT = 100_MB;
int main() {
    double alpha = 90.0_deg;
}
```

User-defined literals

```
struct A {
    int val;
    A(int x) : val(x) {}
};

A operator"" _A (unsigned long long x) { return A(x); }

int main() {
    A a = 1_A;
}
```

При этом нельзя написать

```
A operator"" _A (int x) { return A(x); }
```

Задача: compile-time strings

Определите

```
template <char ...c> struct TString { };
```

и необходимые операторы так, чтобы компилировался код:

```
constexpr auto hello = "hello"_s + " world"_s;
static_assert(hello == "hello world"_s);
```

Метапрограммирование и макросы

```cpp
#include <iostream>
#include <string>

enum class TCode {
    OK = 200,
    NOT_FOUND = 404
};

std::string ToString(TCode code) {
    switch (code) {
        case TCode::OK: return "OK";
        case TCode::NOT_FOUND: return "NOT FOUND";
        default: return "UNK";
    }
}

int main() {
    std::cout << ToString(TCode::OK) << std::endl;
}
```

X macro

Определим таблицу нужных нам кодов и будем определять XX:

```
#define STATUS_CODES \
    XX(200, OK) \
    XX(404, NOT_FOUND)

#define XX(code, text)\
    text = code,

enum class TCode {
    STATUS_CODES
};

#undef XX
```

X macro

```
#define XX(code, text) \
    case TCode::text: return #text;

std::string ToString(TCode code) {
    switch(code) {
        STATUS_CODES
        default: return "UNKNOWN";
    }
}

#undef XX
```

Обработка ошибок

Основные способы обработки ошибок:

1. Возвращение ошибок
2. Обработка исключений

Возвращение ошибок

Будем возвращать из функции 1 (или `false`), если произошла ошибка.
Чем это плохо?

- Чтобы понять, в чем ошибка, нужно совершить дополнительные действия.
- Ошибки могут произойти в каждой строчке кода.
- Код обработки ошибок обычно плохо тестируется.
- Сложно обработать ошибки на большой глубине стека вызовов.

std::optional (C++17)

Шаблонный класс для хранения опционального значения.

```
template<class T>
class optional;
```

- `operator*` или метод `value` — доступ к хранимому значению.
- Метод `has_value` — проверка того, что есть значение.
- Метод `value_or` возвращает хранимое значение или переданный параметр, если значения нет.
- Пустое значение `std::nullopt`.

std::optional (C++17)

Удобно возвращать объекты этого класса в функциях, где возможна какая-то ошибка:

```
std::optional<int> GetCount(const TParam& param) {
    auto it = Params2Int.find(param);
    if (it != Params2Int.end()) {
        return it->second;
    } else {
        return {};
        // or:
        // return std::nullopt;
        // return std::optional<int>();
    }
}

int main() {
    // ...
    if (auto count = GetCount(param)) {
        std::cout << "res = " << *count << std::endl;
    }
}
```

Исключения: напоминание

Механизм исключений — наиболее предпочтительный механизм передачи сообщений об ошибке в C++.
Строки кода, в которых может сгенерироваться ошибка, заключаются в обертку

```
try {
    /*here may be error*/
} catch (/*what*/) {
    /*what to do*/
} catch (/*what*/) {
    /*what to do*/
} ...
```

Когда происходит исключение, генерируется некоторый объект исключения. Этот объект любой природы.
Среди всех `catch` будет выбрана лучшая функция, соответствующая нашему аргументу. Если текущий блок не может обработать исключение, то оно «проваливается» дальше на уровень выше.

Примеры исключений

- Исключение `std::bad_alloc` при нехватке динамической памяти.
- Исключение `std::length_error` при попытке сделать размер вектора или строки больше, чем `max_size()`.
- Исключение `std::out_of_range` при попытке функцией `at()` обратиться к контейнеру по некорректному индексу.

Исключения: catch

- `catch (MyException& ex) { ex.s = "error"; throw; }`

  У этого единственного объекта ex было изменено поле s, его кидаем дальше и он изменен.

- `catch (MyException& ex) { ex.s = "error"; throw ex; }`

  Генерируем новое исключение с объектом ex, вызывается конструктор копирования, а старый уничтожится.

- `catch (MyException ex) { ex.s = "error"; throw ex; }`

  Изменили копию, копию скопировали и бросили.

- `catch (MyException ex) { ex.s = "error"; throw; }`

  Изменили копию, а кинули то же исключение (исходный неизмененный объект).

Исключения и retry

```
for (int i = 1; ; ++i) {
    try {
        auto params = GetParams();
        auto result = GetResult(params);
        return;
    } catch (const std::exception& e) {
        if (i == MaxRetryCount) {
            std::cerr << "Error" << std::endl;
            throw;
        }
        std::cerr << "Error on " << i << std::endl;
    } catch (...) {
        std::cerr << "Fatal error" << std::endl;
        std::terminate();
    }
}
```

Исключения и retry

```
template <class TFunc>
auto Retry(TFunc func, int maxAttemptsCount) {
    for (int i = 1; ; ++i) {
        try {
            return f();
        } catch (const std::exception& e) {
            std::cerr << "Error on " << i << std::endl;
            if (i == maxAttemptsCount) {
                std::cerr
                    << "max attempts has been reached"
                    << std::endl;
                throw;
            }
        }
    }
}
```

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;
```