MASTER OF SOFTWARE ENGINEERING

Лекция 7. Шаблоны.

Илья Макаров

ИТМО JB

19 октября 2021

Санкт-Петербург

Глобальные переменные

Объявление глобальной переменной:

```
extern int global;

void f () {
    ++global;
}
```

Определение глобальной переменной:

```
int global = 10;
```

Проблемы глобальных переменных:

- Масштабируемость.
- Побочные эффекты.
- Порядок инициализации.

Статические глобальные переменные

Статическая глобальная переменная — это глобальная переменная, доступная только в пределах модуля.

Определение:

```
static int global = 10;

void f () {
    ++global;
}
```

Проблемы статических глобальных переменных:

- Масштабируемость.
- Побочные эффекты.

Статические локальные переменные

Статическая локальная переменная — это глобальная переменная, доступная только в пределах функции.

Время жизни такой переменной — от первого вызова функции `next` до конца программы.

```
int next(int start = 0) {
    static int k = start;
    return k++;
}
```

Проблемы статических локальных переменных:

- Масштабируемость.
- Побочные эффекты.

Статические функции

Статическая функция, доступная только в пределах модуля.

Файл 1.cpp:

```
static void test() {
    cout << "A\n";
}
```

Файл 2.cpp:

```
static void test() {
    cout << "B\n";
}
```

Статические глобальные переменные и статические функции проходят *внутреннюю линковку*.

Про линковку

- Нас интересуют 2 типа линковки: `internal` и `external`.
- Внутренняя линковка проходит в рамках одной единицы трансляции (.cpp файла).
- Статические глобальные переменные и статические функции, функции в безымянном `namespace` проходят внутреннюю линковку.

Статические поля класса

Статические поля класса — это глобальные переменные, определённые внутри класса.

Объявление:

```
struct User {
    ...
private:
    static size_t instances_;
};
```

Определение:

```
size_t User::instances_ = 0;
```

Для доступа к статическим полям не нужен объект.

Статические методы

Статические методы — это функции, определённые внутри класса и имеющие доступ к закрытым полям и методам.

Объявление:

```
struct User {
    ...
    static size_t count() { return instances_; }
private:
    static size_t instances_;
};
```

Для вызова статических методов не нужен объект.

```
    cout << User::count();
```

Ключевое слово `inline`

Советует компилятору встроить данную функцию.

```
inline double square(double x) { return x * x; }
```

- В месте вызова `inline`-функции должно быть известно её определение.
- `inline` функции можно определять в заголовочных файлах.
- Все функции, определённые внутри класса, являются `inline`.
- При линковке из всех версий `inline`-функции (т.е. её код из разных единиц трансляции) выбирается только одна.
- Все определения одной и той же `inline`-функции должны быть идентичными.
- `inline` — это совет компилятору, а не указ.

Правило одного определения

Правило одного определения (One Definition Rule, ODR)

- В пределах любой единицы трансляции сущности не могут иметь более одного определения.
- В пределах программы глобальные переменные и не-`inline` функции не могут иметь больше одного определения.
- Классы и `inline` функции могут определяться в более чем одной единице трансляции, но определения обязаны совпадать.

Класс Singleton

```
struct Singleton {
    static Singleton & instance() {
            static Singleton s;
            return s;
    }

    Data & data() { return data_; }

private:
    Singleton() = default;
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    Data data_;
};
```

Использование Singleton-а

```
int main()
{
    // первое обращение
    Singleton & s = Singleton::instance();
    Data d = s.data();

    // аналогично d = s.data();
    d = Singleton::instance().data();
    return 0;
}
```

Метапрограммирование

- **Метапрограммированием** называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.
- Метапрограммы C++ позволяют оперировать типами, шаблонами и compile-time значениями.

Метапрограммирование

- Метапрограммирование в C++ можно применять для широкого круга задач:
  - compile-time вычисления,
  - compile-time проверка ошибок,
  - условная компиляция,
  - генеративное программирование,
  - …
- Для метапрограммирования существуют целые библиотеки, например, MPL и Hana из boost.

Замечание: сложные шаблоны существенно замедляют компиляцию.

Метафункции

Метафункция — это шаблонный класс, который определяет имя типа `type` или целочисленную константу `value`.

- Аргументы метафункции — это аргументы шаблона.
- Возвращаемое значение — это `type` или `value`.

Метафункции

Метафункции могут возвращать типы:

```
template<typename T>
struct add_pointer
{
    using type = T *;
};
```

и значения целочисленных типов:

```
template<int N>
struct square
{
    static int const value = N * N;
};
```

Метафункции

Для типов удобно использовать шаблонные алиасы имен:

```
template<typename T>
using add_pointer_t = add_pointer<T>::type;
```

для значений аналогичные константы:

```
template<int N>
constexpr auto square_v = square<N>::value;
```

Метафункции

```
template<typename T>
int foo(T * value) {
    constexpr auto is_pointer = std::is_same_v<
        int *,
        add_pointer_t<T>>;
    if constexpr (is_pointer) { return *value; }
    else { return square_v<10>; }
}

int main() {
    int a = 1;
    float b = 1.f;
    foo(&a); // 1
    foo(&b); // 100
}
```

Вычисления в compile-time

```
template<int N>
struct Fact {
    static int const value
        = N * Fact<N - 1>::value;
};

template<>
struct Fact<0> {
    static int const value = 1;
};

int main()
{
    std::cout << Fact<10>::value << std::endl;
}
```

Вычисления в compile-time

Это вычисление можно реализовать через `constexpr` функцию.

```
constexpr int fact(int N) {
    if (N == 0) { return 1; }
    return N * fact(N - 1);
}

int main()
{
    constexpr auto a = fact(10);
    std::cout << a << std::endl;
}
```

Вычисления в compile-time

Подход без рекурсии.

```
constexpr int fact(int N) {
    int result = 1;
    for (auto i = 1; i <= N; ++i)
    {
        result *= i;
    }
    return result;
}

int main()
{
    constexpr auto a = fact(10);
    std::cout << a << std::endl;
}
```

Определение списка

Шаблоны позволяют определять алгебраические типы данных.

```
// определяем список
template <typename ... Types>
struct TypeList;

// специализация по умолчанию
template <typename H, typename... T>
struct TypeList<H, T...>
{
    using Head = H;
    using Tail = TypeList<T...>;
};

// специализация для пустого списка
template <>
struct TypeList<> { };
```

Длина списка

```
// вычисление длины списка
template<typename TL>
struct Length {
    static int const value = 1 +
        Length<typename TL::Tail>::value;
};

template<>
struct Length<TypeList<>> {
    static int const value = 0;
};

int main() {
    using TL = TypeList<double, float, int, char>;
    std::cout << Length<TL>::value << std::endl;
}
```

Операции со списком

Добавление элемента в начало списка:

```
template<typename H, typename TL>
struct Cons;

template<typename H, typename... Types>
struct Cons<H, TypeList<Types...>> {
    using type = TypeList<H, Types...>;
};
```

Операции со списком

Конкатенация списков:

```
template<typename TL1, typename TL2>
struct Concat;

template<typename... Ts1, typename... Ts2>
struct Concat<TypeList<Ts1...>, TypeList<Ts2...>>
{
    using type = TypeList<Ts1..., Ts2...>;
};
```

Вывод списка

```
// вывод списка в поток os
template<typename TL>
void printTypeList(std::ostream & os) {
    os << typeid(typename TL::Head).name() << '\n';
    printTypeList<typename TL::Tail>(os);
};

// вывод пустого списка
template<>
void printTypeList<TypeList<>>(std::ostream &) {}

int main() {
    using TL = TypeList<double, float, int, char>;
    printTypeList<TL>(std::cout);
}
```

Генерация классов

```
struct A {
    void foo() {std::cout << "struct A\n";}
};
struct B {
    void foo() {std::cout << "struct B\n";}
};
struct C {
    void foo() {std::cout << "struct C\n";}
};

using Bases = TypeList<A, B, C>;
```

Генерация классов

```
template<typename TL>
struct inherit;

template<typename... Types>
struct inherit<TypeList<Types...>> : Types... {};

struct D : inherit<Bases> { };
```

Генерация классов

```
struct D : inherit<Bases> {
    void foo() { foo_impl<Bases>(); }
    template<typename L> void foo_impl();
};
template<typename L>
inline void D::foo_impl() {
    // приводим this к указателю на базу из списка
    static_cast<typename L::Head *>(this)->foo();

    // рекурсивный вызов для хвоста списка
    foo_impl<typename L::Tail>();
}
template<>
inline void D::foo_impl<TypeList<>>() {}
```

SFINAE

SFINAE = Substitution Failure Is Not An Error.

Ошибка при подстановке шаблонных параметров не является сама по себе ошибкой.

```
// ожидает, что у типа T определён
// вложенный тип value_type
template<class T>
void foo(typename T::value_type * v);

// работает с любым типом
template<class T>
void foo(T t);
```

```
// при инстанциировании первой перегрузки
// происходит ошибка (у int нет value_type),
// но это не приводит к ошибке компиляции
foo<int>(0);
```

Полная специализация шаблонов: классы

```
template<class T>
struct Array { ... };

template<>
struct Array<bool> {
    static unsigned const BITS = 8 * sizeof(unsigned);
    explicit Array(size_t size)
        : size_(size)
        , data_(new unsigned[size_ / BITS + 1])
    {}
    bool operator[](size_t i) const {
        return data_[i / BITS] & (1 << (i % BITS));
    }
private:
    size_t  size_;
    unsigned *  data_;
};
```

Полная специализация шаблонов: функции

```
template<class T>
void swap(T & a, T & b) {
    T tmp(a);
    a = b;
    b = tmp;
}

template<>
void swap<Database>(Database & a, Database & b) {
    a.swap(b);
}

template<class T>
void swap(Array<T> & a, Array<T> & b) {
    a.swap(b);
}
```

Специализация шаблонов и перегрузка

```
template<class T>
void foo(T, T) { std::cout << "same"; }

template<class T, class V>
void foo(T, V) { std::cout << "different"; }

template<>
void foo<int, int>(int, int) { std::cout << "both int"; }

int main() {
    foo(3, 4); // same (not both int)
    foo(3., 4); // different
    return 0;
}
```

Частичная специализация шаблонов

```
template<class T>
struct Array {
    T & operator[](size_t i) { return data_[i]; }
    ...
};

template<class T>
struct Array<T *> {
    explicit Array(size_t size)
        : size_(size)
        , data_(new T *[size_])
    {}
    T & operator[](size_t i) { return *data_[i]; }
private:
    size_t  size_;
    T **    data_;
};
```

Как проверить наличие родственных связей?

```
using YES = char;
struct NO { YES m[2]; };

template<class B, class D>
struct is_base_of {
    static YES test(B * );
    static NO  test(...);

    static bool const value =
        sizeof(YES) == sizeof(test((D *)0));
};

template<class D>
struct is_base_of<D, D> {
    static bool const value = false;
};
```

Как проверить наличие родственных связей?

```
namespace details {
    template <typename B>
    std::true_type test(B*);
    template <typename>
    std::false_type test(void*);

    template <typename, typename>
    auto is_base_of(...) -> std::true_type;
    template <typename B, typename D>
    auto is_base_of(int) -> decltype(
        test<B>(static_cast<D*>(nullptr)));
}
```

Как проверить наличие родственных связей?

```
template <typename B, typename D>
struct is_base_of
    : std::integral_constant<
        bool,
        std::is_class_v<B> && std::is_class_v<D> &&
        decltype(details::is_base_of<B, D>(0))::value
    >{};

class A {};
class B : A {};

int main() {
    std::cout << std::is_base_of<A, B>::value; // 1
    std::cout << std::is_base_of<B, A>::value; // 2
}
```

Как определить наличие метода?

```
struct A {  void foo() { std::cout << "struct A\n"; } };
struct B {  }; // нет метода foo()
struct C {  void foo() { std::cout << "struct C\n"; } };
```

```
template<typename L>
inline void D::foo_impl()
{
    // приводим this к указателю на базу из списка
    static_cast<typename L::Head *>(this)->foo();

    // рекурсивный вызов для хвоста списка
    foo_impl<typename L::Tail>();
}
```

Используем SFINAE

```
template<class T>
struct is_foo_defined
{
    // обёртка, которая позволит проверить
    // наличие метода foo с заданой сигнатурой
    template<class Z, void (Z::*)() = &Z::foo>
    struct wrapper {};

    template<class C>
    static std::true_type check(wrapper<C> * p);
    template<class C>
    static std::false_type check(...);

    static bool const value = std::is_same_v<
        std::true_type,
        decltype(check<T>(0))>;
};
```

Проверяем наличие метода

```
template<class L>
void foo_impl()
{
    using Head = typename L::Head;
    constexpr bool has_foo =
        is_foo_defined<Head>::value;
    if constexpr (has_foo) {
        // call foo
    }
    foo_impl<typename L::Tail>();
}
```

Проверяем наличие метода

C++20 все сильно упрощает.

```
template<class T>
std::string optionalToString(T* obj)
{
    constexpr bool has_str = requires(const T& t) {
        t.toString();
    };

    if constexpr (has_str)
        return obj->toString();
    else
        return "toString not defined";
}
```

std::enable_if

```
namespace std {
    template<bool B, class T = void>
    struct enable_if {};

    template<class T>
    struct enable_if<true, T> { using type = T; };
}
```

```
template<class T>
typename std::enable_if_t<std::is_integral_v<T>, T>
    div2(T t) { return t >> 1; }

template<class T>
typename std::enable_if_t<std::is_floating_point_v<T>, T>
    div2(T t) { return t / 2.0; }
```

std::enable_if

```
template<class T>
T div2(T t, typename std::enable_if_t<
    std::is_integral_v<T>, T> * = 0)
{ return t >> 1; }

template<class T, class E = typename std::enable_if_t<
    std::is_floating_point_v<T>::value, T>>
T div2(T t)
{ return t / 2.0; }
```

```
template<class T, class E = void>
class A;

template<class T>
class A<T, typename std::enable_if_t<
    std::is_integral_v<T>>>
{};
```

Нетиповые шаблонные параметры

Параметрами шаблона могут быть целочисленные значения.

```
template<class T, size_t N, size_t M>
struct Matrix {
    ...
    T & operator()(size_t i, size_t j)
    { return data_[M * j + i]; }
private:
    T data_[N * M];
};

template<class T, size_t N, size_t M, size_t K>
Matrix<T, N, K> operator*(Matrix<T, N, M> const& a,
                          Matrix<T, M, K> const& b);
```

Нетиповые шаблонные параметры

Параметрами шаблона могут быть указатели/ссылки на значения с внешней линковкой.

```
// log - это глобальная переменная
template<ofstream & log>
struct FileLogger { ... };
```

Шаблонные параметры — шаблоны

Параметрами шаблона могут быть шаблоны.

```
// int -> string
string toString( int i );

// работает только с Array<>
Array<string> toStrings( Array<int> const& ar ) {
    Array<string> result(ar.size());
    for (size_t i = 0; i != ar.size(); ++i)
        result.get(i) = toString(ar.get(i));
    return result;
}
```

Шаблонные параметры — шаблоны

```
// от контейнера требуются:
// - конструктор от size
// - методы size() и get()
template<template <class> class Container>
auto toStrings(Container<int> const& c) {
    Container<string> result(c.size());
    for (size_t i = 0; i != c.size(); ++i)
        result.get(i) = toString(c.get(i));
    return result;
}
```

Компиляция шаблонов

- Шаблон независимо компилируется для каждого значения шаблонных параметров.
- Компиляция (*инстанциирование*) шаблона происходит в точке первого использования — *точке инстанциирования шаблона*.
- Компиляция шаблонов классов — ленивая, компилируются только те методы, которые используются.
- В точке инстанциирования шаблон должен быть полностью определён.
- Шаблоны следует определять в заголовочных файлах.
- Все шаблонные функции (свободные функции и методы) являются `inline`.
- В разных единицах трансляции инстанциирование происходит независимо.

Резюме про шаблоны

- Большие шаблонные классы следует разделять на два заголовочных файла: объявление (`array.hpp`) и определение (`array_impl.hpp`).
- Частичная специализация и шаблонные параметры по умолчанию есть только у шаблонов классов.
- Вывод шаблонных параметров есть только у шаблонов функций.
- Предпочтительно использовать перегрузку шаблонных функций вместо их полной специализации.
- Полная специализация функций — это обычные функции.
- Виртуальные методы, конструктор по умолчанию, конструктор копирования, оператор присваивания и деструктор не могут быть шаблонными.
- Используйте `typedef` или `using` для длинных шаблонных имён.