

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекции 1,2
03.09.2021
10.09.2021

Вывод типов шаблонов

```
template <typename T>
void f(const T& param)
```

В коде

```
int x = 0;
f(x);
```

тип Т выводится как int.

Вывод типов шаблонов

Случай 1. Тип параметра — указатель или ссылка:

```
template <typename T>
void f(T& param);
```

Ссылочная часть игнорируется.

```
int x = 1;
const int cx = x;
const int& rx = x;
f(x);
f(cx);
f(rx);
```

Вывод типов шаблонов

Случай 1. Тип параметра — указатель или ссылка:

```
template <typename T>
void f(T& param);
```

Ссылочная часть игнорируется.

```
int x = 1;
const int cx = x;
const int& rx = x;
f(x); // f<int>(int)
f(cx); // f<const int>(const int&)
f(rx); // f<const int>(const int&)
```

Вывод типов шаблонов

Случай 2. Передача по значению:

```
template <typename T>
void f(T param);
```

Ссылочная часть игнорируется. Если после этого остается const, то он тоже игнорируется.

```
int x = 1;
const int cx = x;
const int& rx = x;
f(x); // f<int>(int)
f(cx); // f<int>(int)
f(rx); // f<int>(int)
```

Вывод типов шаблонов

Случай 2. Передача по значению:

```
template <typename T>
void f(T param);
```

Ссылочная часть игнорируется. Если после этого остается const, то он тоже игнорируется.

```
template <typename T>
void f(T param);

const char* const ptr = "abcd";
f(ptr);
```

Вывод типов шаблонов

Случай 2. Передача по значению:

```
template <typename T>
void f(T param);
```

Ссылочная часть игнорируется. Если после этого остается const, то он тоже игнорируется.

```
template <typename T>
void f(T param);

const char* const ptr = "abcd";
f(ptr); // f<const char*>(const char*)
```

Вывод типов шаблонов

Указатели на массивы

```
const char s[] = "ab"; // ??
```

```
template <typename T>
void f(T& param);
```

```
f(s); // f<char const [3]>(char const (&) [3])
```

Вывод типов шаблонов

Указатели на массивы

```
const char s[] = "ab"; // ??
```

```
template <typename T>
void f(T param);
```

```
f(s); // f<const char*>(const char*)
```

Автоматический вывод типа переменной

```
auto x = 0;
auto y = x, *z = &x;
auto& ref = x;
const auto & xref = x;
auto* p = &x;
const auto* cp = p;
const auto pc = &x ;
```

Автоматический вывод типа переменной

```
auto x = 0;
auto y = x, *z = &x; // int * y
auto& ref = x; // int& ref
const auto & xref = x; // const int&
auto* p = &x; // int* p;
const auto* cp = p; // const int * p;
const auto pc = &x ; // int * const pc;

auto предполагает список initializer_list в фигурных скобках, вывод шаблона – нет.
```

Ключевое слово decltype

```
decltype(1 + 2) x = 1; // int
decltype(x) y = x; // int
decltype(1,x) z = x; // int&
```

decltype можно использовать везде, где можно использовать тип.

```
auto cmp = [](char const* p1, char const* p2) {
    return strcmp(p1, p2) < 0;
}
std::map<char const*, unsigned, decltype(cmp)> MyMap(cmp);
```

Круглые и фигурные скобки

До C++11 невозможно было создать контейнер содержащим определенный набор значений.

```
std::vector<int> v {1, 3, 5};
```

Запрет сужающей инициализации:

```
double a, b;  
int c{a + b}; // error!  
int d = a + b; // ok  
int e(a + b); // ok
```

Круглые и фигурные скобки

Что будет напечатано?

```
class T {
public:
    T(int a, bool b)
        { std::cout << "int, bool" << std::endl;}
    T(int a, double b)
        { std::cout << "int, double" << std::endl;}
    T(std::initializer_list<long double> l)
        {std::cout << "init list" << std::endl; }
};

int main() {
    T t1(10, true);
    T t2 {10, true};
    T t3(10, 0.2);
    T t4 {10, 0.2};
    return 0;
}
```

Круглые и фигурные скобки

Что будет напечатано?

```
class T {
public:
    T(int a, bool b)
        { std::cout << "int, bool" << std::endl;}
    T(int a, double b)
        { std::cout << "int, double" << std::endl;}
    T(std::initializer_list<long double> l)
        {std::cout << "init list" << std::endl; }
};

int main() {
    T t1(10, true); // int, bool
    T t2 {10, true}; // init list
    T t3(10, 0.2); // int, double
    T t4 {10, 0.2}; // init list
    return 0;
}
```

range-based циклы

```
for (T var : container) {  
    // ...  
}
```

begin и end запоминаются перед циклом.

```
using TElem = std::pair<int, int>;  
using TCont = std::vector<TElem>;  
TCont container;  
for (auto x : container) {  
    // x - копия элемента в контейнере  
}  
for (auto& x : container) {  
    // x - ссылка на элемент в контейнере  
}
```

Ключевое слово override

Где ошибки?

```
struct A {
    virtual void foo();
    void bar();
};

struct B : A {
    void foo() const override;
    void foo() override;
    void bar() override;
};
```

Ключевое слово override

Где ошибки?

```
struct A {
    virtual void foo();
    void bar();
};

struct B : A {
    void foo() const override; // ошибка, не совпадают сигнатуры
    void foo() override; // ok
    void bar() override; // ошибка, A::bar не виртуальная
};
```

Ключевое слово `override`

Перекрытие функций:

- ▶ Функция базового класса должна быть виртуальной
- ▶ В базовом и производном классе должны совпадать:
 - ▶ имена функций,
 - ▶ типы параметров,
 - ▶ константность,
 - ▶ возвращаемые типы и спецификации исключений.

Ключевое слово final

Препятствие перекрытия

```
struct A {  
    virtual void foo() final; // запрет переопределения функции  
    void bar() final; // ошибка, так как функция не виртуальная  
};  
struct B final : A { // от структуры B нельзя отнаследоваться  
    void foo(); // ошибка: A::foo - final  
};  
struct C : B { // ошибка, B - final  
};
```

Удаленные функции

Некопируемый класс:

```
template <typename T>
class TConfig {
    private:
        TConfig(const TConfig&);
        TConfig& operator=(const TConfig&);
};
```

Удаленные функции

Некопируемый класс:

```
template <typename T>
class TConfig {
public:
    TConfig(const TConfig&) = delete;
    TConfig& operator=(const TConfig&) = delete;
};
```

Удаленные функции

- ▶ Удаленной может быть любая функция (не только функции-члены класса).
- ▶ Полезное применение — предотвращение использования ненужных инстанцирований шаблонов.

Псевдонимы

typedef:

```
typedef std::shared_ptr<std::map<std::string, std::string>>  
TMyPtr;
```

```
typedef bool (*FPtr)(int, int);
```

using (C++11):

```
using TMyPtr =  
std::shared_ptr<std::map<std::string, std::string>>;  
using FPtr = bool (*)(int, int);
```

В чем отличие **typedef** от **using**?

Объявление псевдонимов поддерживает шаблонизацию.

```
template <typename T>  
using MyVec = std::vector<T, std::allocator<T>>;
```

scoped enumerations

```
enum Color {black, white, blue};  
bool white; // error!
```

C++11:

```
enum class Color { red, green = 20, blue };  
Color r = Color::blue;  
switch (r) {  
    case Color::red: // ...  
    case Color::green: // ...  
    case Color::blue: // ...  
}  
int n = r; // ошибка  
int n = static_cast<int>(r);
```

Базовый тип — int.

constexpr

```
const int a = 10;
const int b = std::numeric_limits<int>::max();
const int c = INT_MAX;
```

```
int a;
const int b = a; // ok
constexpr auto s = a; // error
```

```
constexpr int f() {return 1024;}
```

constexpr-функция должна состоять из одного `return` (C++11), возвращать константу или вызывать такую же функцию. Вычисление должно производиться во время компиляции (с аргументами, значения которых известны во время компиляции).

Пример: проверка простоты числа в compile-time

```
constexpr bool is_div(int a, int b) {
    return (b == 1) || (a % b != 0 && is_div(a, b - 1));
}

constexpr bool is_prime(int number) {
    return number != 1 && is_div(number, number / 2);
}

int main() {
    static_assert(is_prime(29) , " 29 is not prime");
    static_assert(is_prime(36) , " 36 is prime");
    return 0;
}
```

Еще про constexpr

Вычисления в момент компиляции:

```
constexpr int fact(int n) {
    return n == 0 ? 1 : n * fact(n - 1);
}

static_assert (fact(7) == 5040);
```

- ▶ только return;
- ▶ только операции над константами и аргументами;
- ▶ можно вызывать constexpr-функции;
- ▶ sizeof, throw;
- ▶ можно рекурсию и тернарный оператор.

Еще про constexpr

C++14:

```
constexpr int fact(int n) {
    int result = 1;
    for (int i = 0; i <= n; ++i) result *= i;
    return result;
}
```

Еще про constexpr

C++17:

```
constexpr auto GetArray() {
    std::array<int, 3> a = {1, 2, 3};
    a[0] = 5;
    return a;
}

int main() {
    auto x = GetArray();
    cout << x[0];
}
```

constexpr if

C++17:

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

Variadic templates

Шаблоны с переменным числом аргументов (C++11).

```
#include <iostream>
void myprintf (const char *str) {
    std::cout << str;
}

template <typename T, typename... Targs>
void myprintf(const char* str, T value, Targs... Fargs) {
    for ( ; *str != '\0' ; ++str) {
        if (*str == '%') {
            std::cout << value;
            myprintf(str + 1, Fargs...);
            return;
        }
        std::cout << *str;
    }
}
```

Variadic templates

Получение i-го значения:

```
#include <iostream>

template <unsigned n, class T, class... Args>
constexpr auto Get(T value, Args... args) {
    if constexpr(n > sizeof...(args)) {
        return;
    } else if constexpr (n > 0) {
        return Get<n - 1>(args...);
    } else {
        return value;
    }
}

int main() {
    std::cout << (Get<2>(1, "abc", 'c'));
}
```

Variadic templates

На этом определен std::tuple.

```
template <typename... Args>
class Tuple;
template<>
class Tuple{};
```

```
template <typename Head, typename... Tail>
class Tuple<Head,Tail...> : Tuple<Tail...> {
    // ...
}
auto t = std::make_tuple(1, 10.0, "abc");
// std::get<0>(t); std::get<1>(t); std::get<2>(t);
```

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 3
17.09.2021

Variadic templates

На variadic templates определен std::tuple.

```
template <typename... Args>
class Tuple;
template<>
class Tuple{};

template <typename Head, typename... Tail>
class Tuple<Head,Tail...> : Tuple<Tail...> {
    // ...
}
```

std::tuple

```
auto t = std::make_tuple(1, 10.0, "abc");
// std::get<0>(t); std::get<1>(t); std::get<2>(t);
```

Если хочется вернуть кортеж из функции, то приходилось писать при помощи `std::tie`:

```
std::tuple<int, std::string, int> func();
```

```
int a, b;
std::string s;
std::tie(a, s, b) = func();
```

C++17:

```
auto [a, s, b] = func();
```

Структурное связывание

```
// C++17
for (const auto &[key, value] : get_pairs()) {
    // do smth with key, value
}
```

Пример: std::tie и сравнение

```
enum class ModelType {
    ADVType = 0,
    SBSCRType = 1,
    RENTType = 2,
};

class TOffer {
    int price;
    ModelType type;
    std::string partnerName;
public:

    bool operator<(const TOffer& other) const {
        return
            std::tie(price, partnerName, type) <
            std::tie(other.price, other.partnerName, other.type);
    }
};
```

Сортировка

```
template <typename T>
void sort(T * begin, T * end) {
    for (T* p1 = begin; p1 != end; ++p1) {
        for (T* p2 = p1 + 1; p2 != end; ++p2) {
            if (*p1 > *p2)
                std::swap(*p1, *p2);
        }
    }
}

int main() {
    int a[] = {3, 5, 2, 8, 8, 1, 0, 15, 12, -1, 3, 4, 7};
    sort(a, a + sizeof(a) / sizeof(a[0]));
    for (const auto &v : a) {
        std::cout << v << " ";
    }
    std::cout << std::endl;
}
```

Сортировка

```
template <typename T>
bool CompareTLess(const T&a, const T&b) { return a < b;}

template <typename T>
bool CompareTGreater(const T&a, const T&b) { return a > b;}

template <typename T>
void sort(T * begin, T * end, bool (*cmp)(const T&, const T&)) {
    for (T* p1 = begin; p1 != end; ++p1) {
        for (T* p2 = p1 + 1; p2 != end; ++p2) {
            if (cmp(*p2, *p1))
                std::swap(*p1, *p2);
        }
    }
}
// ...
sort(a, a + sizeof(a) / sizeof(a[0]), &CompareTLess<int>);
sort(a, a + sizeof(a) / sizeof(a[0]), &CompareTGreater<int>);
```

Функторы

```
template <typename T>
class TComparer {
private:
    bool IsLess;
public:
    TComparer(bool IsLess)
        : IsLess(IsLess)
    {}
    bool operator() (const T&a, const T&b) const {
        return IsLess ? a < b : a > b;
    }
};
```

Функторы

```
template <typename T, typename TComparerType>
void sort(T * begin, T * end, const TComparerType& cmp) {
    for (T* p1 = begin; p1 != end; ++p1) {
        for (T* p2 = p1 + 1; p2 != end; ++p2) {
            if (cmp(*p2, *p1))
                std::swap(*p1, *p2);
        }
    }
}
// ...

sort(a, a + sizeof(a) / sizeof(a[0]), TComparer<int>(true));
sort(a, a + sizeof(a) / sizeof(a[0]), TComparer<int>(false));
```

Функторы

Функции сравнения, конечно же, уже есть — `std::less`,
`std::greater`

```
sort(a, a + sizeof(a) / sizeof(a[0]), std::less<int>());
```

```
sort(a, a + sizeof(a) / sizeof(a[0]), std::greater<int>());
```

Что неудобно, если это не готовый функтор? Функции
описываются вне места применения.

Локальные функции и шаблонные классы запрещены.

Функторы

Можно так:

```
int main() {
    struct TC {
        bool operator() (int a, int b) const {
            return a < b;
        }
    };
    int a[] = {3, 5, 2, 8, 8, 1, 0, 15, 12, -1, 3, 4, 7};

    sort(a, a + sizeof(a) / sizeof(a[0]), TC());
    for (const auto &v : a) {
        std::cout << v << " ";
    }
    std::cout << std::endl;
}
```

Но это некрасиво.

lambda-объекты

```
sort(a, a + sizeof(a) / sizeof(a[0]),
    [](int a, int b) {
        return a < b;
    }
);
```

Лямбда-выражения

Быстрый способ создать такую структуру с оператором ():

```
struct T {  
    bool operator()(int x){};  
};  
Do(T(), ...);
```

- ▶ [] — список переменных, которые захватывает лямбда-выражение;
- ▶ () — входные аргументы функции;
- ▶ {} — тело функции.

Лямбда-выражения

```
[capture] (params) mutable exception_attribute -> ret {body}
[capture] (params) -> ret {body}
[capture] (params) {body}
[capture] {body}
```

Пример:

```
std::vector<int> v = {-1, -2, -3, -4, -5, 1, 2, 3, 4, 5};
std::sort(v.begin(), v.end(), [](int l, int r) {
    return l * l < r * r;
});
```

Лямбда-выражения

- ▶ [] — без захвата переменных
- ▶ [=] — все переменные захватываются по значению
- ▶ [&] — все переменные захватываются по ссылке
- ▶ [x] — захват x по значению
- ▶ [&x] — захват x по ссылке
- ▶ [x, &y] — захват x по значению, у по ссылке
- ▶ [=, &x, &y] — захват всех переменных по значению, но x,y — по ссылке
- ▶ [&, x] — захват всех переменных по ссылке, кроме x
- ▶ [this] — для доступа к переменной класса

return

Вот так не работает:

```
auto cmp = [&data](int a, int b) {
    if (a == 12)
        return -1;
    return data[a] < data[b];
};
```

А вот так всё хорошо:

```
auto cmp = [&data](int a, int b) -> int {
    if (a == 12)
        return -1;
    return data[a] < data[b];
};
```

mutable

Те элементы, которые захвачены по значению, автоматически становятся константами внутри лямбды.

```
auto cmp = [&d, d1](int a, int b) mutable -> int {  
    d[0] = d1[0] = 0;  
    if (a == 12)  
        return -1;  
    return d[a] < d[b];  
};
```

Упражнение

Отсортировать массив, не испортив его — вывести перестановку, сохранив исходные данные.

```
const int a[] = {3, 5, 2, 8, 15, 12, -1, 3, 4, 7}; //  
size_t n = sizeof(a) / sizeof(a[0]);  
std::vector<size_t> idx(n);  
for (int i = 0; i < n; ++i) {  
    idx[i] = i;  
}  
  
// ... ?  
  
for (const auto &i : idx) {  
    std::cout << a[i] << " ";  
}  
std::cout << std::endl;  
}
```

Захват данных класса

```
class T {
private:
    std::vector<int> Data;
public:
    T(const std::vector<int>& data)
        :Data(data)
    {}
    void Do() {
        auto f = [this](int a, int b) {
            return Data[a] < Data[b];
        };
    }
};
```

Опасности захвата по умолчанию

```
using T = std::vector<std::function<bool(int)>>;\n\nvoid AddFunc(T& funcs) {\n    static int x = 2;\n    funcs.emplace_back(\n        [=](int v) { return v % x == 0;}\n    );\n    ++x;\n}\n\nint main() {\n    T funcs;\n    AddFunc(funcs);\n    AddFunc(funcs);\n    funcs[0](5);\n    funcs[1](5);\n}
```

Захватывать явно — заметнее ошибки.

Инкапсуляция сложной инициализации в лямбду

Пример:

```
TSomeType obj;
for (auto i = 2; i <= N; ++i) {
    obj += some_func(i);
}
// далее obj не меняется
```

Вынесем

```
const TSomeType obj = [&]{
    TSomeType val;
    for (auto i = 2; i <= N; ++i) {
        val += some_func(i);
    }
    return val;
}();
```

C++17: constexpr-lambdas

```
constexpr auto add(int y) {
    return [=](int x) { return x + y;};
}
int main() {
    constexpr auto inc = add(5);
    static_assert(inc(3) == 8);
}
```

C++17: if-init expressions

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

// C++17:
if (auto x = get_result(); x == 1) {
```

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 4
24.09.2021

hashset, hashmap

```
template<
    typename Key,
    typename Value,
    typename Hash = std::hash<Key>,
    typename KeyEqual = std::equal_to<Key>,
    typename Allocator =
        std::allocator<std::pair<const Key, Value>>
> class unordered_map;
```

hashmap: случай Key = const char*

```
std::unordered_map<const char*, unsigned> table;
const char* x = "ABC";

std::string s = "ABC";
const char* y = s.c_str();

table[x] = 1;
// strcmp(x, y) == 0
// table.find(y) == table.end()
// std::hash<const char*>()(x) == 13930663984845506963
// std::hash<const char*>()(y) == 12394125337132834388
```

hashmap: случай Key = const char*

Функция для хэширования:

```
struct THashString {
    void hashCombine(size_t& seed, const char v) const {
        seed ^= v + 0x9e3779b9 + (seed << 6) + (seed >> 2);
    }

    size_t operator() (char const* p) const {
        size_t hash = 0;
        for (; *p; ++p) {
            hashCombine(hash, *p);
        }
        return hash;
    }
};
```

hashmap: случай Key = const char*

Сравнение ключей:

```
struct TCompString {  
    bool operator() (const char* p1, const char* p2) const {  
        return strcmp(p1, p2) == 0;  
    }  
};
```

hashmap: случай Key = const char*

```
std::unordered_map<const char*, unsigned,
                    THashString, TCompString> table;
const char* x = "ABC";

std::string s = "ABC";
const char* y = s.c_str();

table[x] = 1;
// strcmp(x, y) == 0
// table.find(y) != table.end()
// THashString()(x) == 11093822720383
// THashString()(y) == 11093822720383
```

C++17: std::optional

Если всё хорошо, то есть значение, иначе — значения нет.

```
std::optional<int> GetCount(const int param) {
    const static std::map<int, int> MAPINT_PARAMS = ....;
    auto it = MAPINT_PARAMS.find(param);
    if (it != MAPINT_PARAMS.end()) {
        return it->second;
    } else {
        // return std::nullopt;
        // return std::optional<int>();
        return {};
    }
}

int main() {
    int param = 3;
    if (auto count = GetCount(param)) {
        std::cout << "res = " << *count << std::endl;
    }
}
```

C++17: std::optional

Существует возможность взять std::hash от std::optional.

```
#include <iostream>
#include <optional>
#include <string>
#include <unordered_set>
int main()
{
    std::unordered_set<std::optional<std::string>> s = {
        "ABC", "DEF", std::nullopt
    };

    for(const auto& item : s) {
        std::cout << item.value_or("nothing") << ' ';
    }
}
```

Отступление: указатели на методы внутри класса

```
class C {  
private:  
    int a, b;  
public:  
    C(int a, int b) : a(a), b(b) {};  
    void f() const { ... }  
    void g() const { ... }  
};
```

Хотим сделать указатель на функцию `f` из класса,

```
void (*ptr) ();           //  
ptr = &C::f;             // так нельзя  
void (C::*ptr)() const;  
ptr = &C::f;              // ok
```

std::function

```
#include <functional>
struct Foo {
    void print(int i) const { std::cout << i << std::endl; }
};

void print_num(int i) {
    std::cout << i << std::endl;
}

int main() {
    std::function<void(int)> f = print_num;
    f(1);

    std::function<void()> g = []() { print_num(2); };
    g();

    std::function<void(const Foo&, int)> h =
        &Foo::print;
    h(Foo(), 1);
}
```

Функторы в <functional>

```
greater
less
greater_equal
less_equal
equal_to
not_equal_to
plus
minus
multiply
divide
negate -
modulus
logical_and
logical_or
logical_not
```

```
sort(a.begin(), a.end(), std::greater<int>());
transform(a.begin(), a.end(), b.begin(),
          ostream_iterator<int> (cout, " "), plus<int>());
```

Связыватели в C++98

```
remove_copy_if (a.begin(),
                a.end(),
                ostream_iterator<int> (cout, " "),
                bind2nd(less<int>(), 3));
```

Связыватели в C++98

Но вот так не работает:

```
class MyCmp {
public:
    bool operator()(int a, int b) const {
        return a > b;
    }
};

remove_copy_if (a.begin(),
                a.end(),
                b.begin(),
                std::bind2nd(MyCmp(), 3));
```

Связыватели в C++98

А так ок:

```
class MyCmp : public std::binary_function<int, int, bool>{
public:
    bool operator()(int a, int b) const {
        return a > b;
    }
};

remove_copy_if (a.begin(),
                a.end(),
                b.begin(),
                std::bind2nd(MyCmp(), 3));
```

C++11: std::bind

```
int f(int a, int b) {
    return a - b;
}

int main() {
    auto g = std::bind(f, 3, std::placeholders::_2);
    std::cout << g(1, 4) << std::endl;
    return 0;
}
-1

std::bind(h, _2, _1, 2, 3, 4) (x, y); // h(y, x, 2, 3, 4)
```

C++11: std::bind

```
int f(int a, int b) {
    return a - b;
}

int g(int x) {
    return x * x;
}

int main() {
    auto f2 = std::bind(
        f,
        std::bind(
            g,
            std::placeholders::_2
        ),
        std::placeholders::_1
    );
    std::cout << f2(3, 7) << std::endl;
}
```

C++17: std::invoke

Вызывает callable-объект с заданными аргументами.

```
class C {
    int a;
public:
    C(int a) : a(a) {}
    void f(int k) const {
        std::cout << a << " " << k << std::endl;
    }
};

void print_int(int i) { std::cout << i << std::endl; }

int main()
{
    std::invoke(print_int, 117);
    std::invoke([]() { print_int(117); });
    const C obj(117);
    std::invoke(&C::f, obj, 42);
}
```

C++17: std::apply

Вызывает callable-объект с аргументами, переданными в виде tuple.

```
int add(int first, int second, int third) {
    return first + second + third;
}

int main()
{
    std::cout << std::apply(add, std::make_tuple(1, 2, 3));
    // 6
}
```

C++20: std::bind_front

Вызов callable-объекта с параметрами, первые из которых берутся из аргументов std::bind_front.

Вызов

```
std::bind_front(f, bound_args...)(call_args...)
```

эквивалентен

```
std::invoke(f, bound_args..., call_args....)
```

Пример:

```
int minus(int a, int b){  
    return a - b;  
}  
int main()  
{  
    auto f = std::bind_front(minus, 117);  
    std::cout << f(1); // 116  
}
```

C++20: std::bind_front

Пример: «провяжем» вызов метода foo класса T с конкретным инстансом x.

Без bind_front:

```
auto f = [x](auto &&... args) -> decltype(auto) {
    return x->foo(std::forward<decltype(args)>(args)...);
}
```

Альтернатива:

```
auto f = std::bind_front(&T::foo, x);
```

mem_fn в C++11

std::mem_fn

Defined in header `<functional>`

<code>template< class M, class T ></code>		(since C++11)
<code>/*unspecified*/ mem_fn(M T::* pm) noexcept;</code>		(until C++20)
<code>template< class M, class T ></code>		
<code>constexpr /*unspecified*/ mem_fn(M T::* pm) noexcept;</code>		(since C++20)

Function template `std::mem_fn` generates wrapper objects for pointers to members, which can store, copy, and invoke a [pointer to member](#). Both references and pointers (including smart pointers) to an object can be used when invoking a `std::mem_fn`.

variadic — может работать с произвольным числом аргументов.

```
C c(1);
```

```
auto g = std::mem_fn(&C::f);
```

```
// the same:
```

```
auto g2 = [] (C& c, int x) { return c.f(x); } ;  
g(c, 10);
```

std::mem_fn

```
class C {
    int a;
public:
    C(int a) : a(a) {}
    void f() const {
        std::cout << a << std::endl;
    }
};

int main() {
    C c [] = {C(1), C(2), C(3), C(4)};
    C* cptr [] = {c, c + 1, c + 2, c + 3};
    std::for_each(c, c + 4, std::mem_fn(&C::f));
    std::for_each(cptr, cptr + 4, std::mem_fn(&C::f));
    std::for_each(c, c + 4,
                  std::function<void(const C&)>(&C::f));
    std::for_each(cptr, cptr + 4,
                  std::function<void(const C*)>(&C::f));
}
```

std::not_fn

Создает функцию с инвертированным результатом.

```
std::vector<int> v1 = { /*....*/ };
auto divisible_by_3 = [] (int i){ return i % 3 == 0; };

int divisible =
    std::count_if(v1.begin(), v1.end(), divisible_by_3);

int not_divisible =
    std::count_if(v1.begin(), v1.end(),
    std::not_fn(divisible_by_3));
```

Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void DoConst(const std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

int main() {
    Do(std::vector<int>(10));
    DoConst(std::vector<int>(10));
}
```

Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void DoConst(const std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

int main() {
    Do(std::vector<int>(10)); // error
    DoConst(std::vector<int>(10));
}
```

Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void Do(const std::vector<int> &v) {
    std::cout << "const v.size() == " << v.size() << std::endl;
}

int main() {
    std::vector<int> v(10);
    Do(v);
    Do(std::vector<int>(10));
}
```

Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void Do(const std::vector<int> &v) {
    std::cout << "const v.size() == " << v.size() << std::endl;
}

int main() {
    std::vector<int> v(10);
    Do(v); // v.size() == 10
    Do(std::vector<int>(10)); // const v.size() == 10
}
```

Типы ссылок

```
#include <iostream>

int rvalue() {
    return 5;
}

int& lvalue() {
    static int tmp = 0;
    return tmp;
}

int main() {
    &lvalue(); // ok
    &rvalue(); // error
    lvalue() = rvalue(); // ok
}
```

move

```
template <typename T>
void Swap(T& a, T& b) {
    T t(a);
    a = b;
    b = t;
}
```

```
template <typename T>
void Swap(T& a, T& b) {
    T t(std::move(a));
    a = std::move(b);
    b = std::move(t);
}
```

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 5
08.10.2021

Типы ссылок

```
#include <iostream>

int rvalue() {
    return 5;
}

int& lvalue() {
    static int tmp = 0;
    return tmp;
}

int main() {
    &lvalue(); // ok
    &rvalue(); // error
    lvalue() = rvalue(); // ok
}
```

move

```
template <typename T>
void Swap(T& a, T& b) {
    T t(a);
    a = b;
    b = t;
}
```

```
template <typename T>
void Swap(T& a, T& b) {
    T t(std::move(a));
    a = std::move(b);
    b = std::move(t);
}
```

Еще пример

```
A& ref = A(); // error  
A&& ref = A(); // ok
```

```
#include <iostream>  
#include <string>  
  
int main() {  
    std::string x = "abc";  
    std::string y = std::move(x);  
    std::cout << x << "-" << y << std::endl;  
    return 0;  
}
```

lvalue и rvalue

```
class T {....};  
  
int x = 5; // 5 - rvalue  
int y = x + 2; // (x + 2) - rvalue  
int* p = &x; // p - lvalue  
int* p1 = &(x + 2); // error  
x + 2 = 5; // error  
T t;  
t = T(); // T() - rvalue
```

Операторы копирования и перемещения

Наиболее частое использование move-семантики – move-операторы перемещения и move конструкторы.

```
T& T::operator=(const T& rhs);
```

```
T& T::operator=(T&& rhs);
```

Move-семантика реализована для STL-контейнеров, что ускоряет их.

lvalue or rvalue

```
void push1(A& a, std::vector<A>& v) {
    v.push_back(a); // copy constructor
}

void push2(A&& a, std::vector<A>& v) {
    v.push_back(a); // ???
}
```

lvalue or rvalue

```
void push1(A& a, std::vector<A>& v) {  
    v.push_back(a); // copy constructor  
}  
  
void push2(A&& a, std::vector<A>& v) {  
    v.push_back(a); // copy constructor  
}
```

lvalue or rvalue

```
void push1(A& a, std::vector<A>& v) {
    v.push_back(a); // copy constructor
}

void push3(A&& a, std::vector<A>& v) {
    v.push_back(std::move(a)); // move constructor
}
```

emplace_back

```
int main() {  
    std::vector<A> as;  
    as.push_back(A(3));  
    as.emplace_back(1);  
}
```

Создаем объект без лишнего копирования.

Типы ссылок

▶ lvalue

- ▶ обычные ссылки, константные ссылки, ...
- ▶ это не временный объект и не тот объект, который вскоре будет уничтожен.

▶ xvalue

- ▶ объект который вот-вот должен быть уничтожен (expired)
- ▶ пример — то, что приходит в оператор перемещения

▶ prvalue

- ▶ pure, настоящее rvalue
- ▶ пример — результат вызова функции, у которой возвращаемое значение – не ссылка.

glvalue — lvalue или xvalue

rvalue — xvalue, или временный объект, или значение, не ассоциированное с объектом.

xvalue: два примера

1.

```
int&& f(){    return 3; }
// ...
f();
```
2.

```
static_cast<int&&>(7);
std::move(7);
```

Типы ссылок: практическое правило

1. Если у выражения можно взять адрес — это **lvalue**
2. Если тип выражения — `T&` или `const T&`, — это **lvalue**
3. Иначе это **rvalue**. Обычно — литералы, результат вызова функций и т. п.

Универсальная ссылка

При инициализации универсальной ссылки определяется, какую ссылку она представляет – rvalue/lvalue.

```
template<typename T>
void f(T&& x);
```

```
T a;
f(a); // lvalue-ссылка
f(std::move(a)); // rvalue-ссылка
```

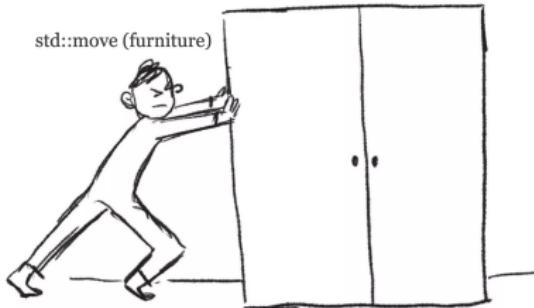
Типы ссылок

```
template<typename T>
void f(std::vector<T>&& x);
```

```
std::vector<int> v;
f(v); // error
```

```
template<typename T>
void f(const T&& x); // rvalue-ссылка
```

std::move



```
template< class T >
constexpr std::remove_reference_t<T>&& move( T&& t ) noexcept;
```

Возвращаемое значение:

```
static_cast<typename std::remove_reference<T>::type&&>(t)
```

std::move ничего не перемещает и не делает никаких действий в runtime.

std::move

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

class TMyType {
public:
    TMyType(const TSuperClass val) : field(std::move(val));
private:
    TSuperClass field;
}
```

В чем тут проблема?

std::forward

- ▶ std::move выполняет безусловное приведение своего аргумента к rvalue
- ▶ std::forward выполняет приведение только при соблюдении определенных условий.

std::forward

```
class A{};  
void Do(const A& x) {  
    std::cout << "call Do lvalue" << std::endl;  
}  
void Do(A&& x) {  
    std::cout << "call Do rvalue" << std::endl;  
}  
template <typename T>  
void call(T&& obj) {  
    Do(obj);  
}  
int main() {  
    A x;  
    call(x);  
    call(std::move(x));  
}  
  
call Do lvalue  
call Do lvalue
```

std::forward

```
class A{};  
void Do(const A& x) {  
    std::cout << "call Do lvalue" << std::endl;  
}  
void Do(A&& x) {  
    std::cout << "call Do rvalue" << std::endl;  
}  
template <typename T>  
void call(T&& obj) {  
    Do(std::forward<T>(obj));  
}  
int main() {  
    A x;  
    call(x);  
    call(std::move(x));  
}  
  
call Do lvalue  
call Do rvalue
```

Как работает std::forward

Шаблон с универсальной ссылкой

```
template <typename T>
void call(T&& obj);
```

- ▶ Если в качестве аргумента передается lvalue, то T выводится как lvalue-ссылка.
- ▶ Если в качестве аргумента передается rvalue, то T не является ссылкой.

```
int x;
call(x);
call(std::move(x));
```

Как работает std::forward

Шаблон с универсальной ссылкой

```
template <typename T>
void call(T&& obj);
```

- ▶ Если в качестве аргумента передается lvalue, то T выводится как lvalue-ссылка.
- ▶ Если в качестве аргумента передается rvalue, то T не является ссылкой.

```
int x;
call(x);    // T - int&
call(std::move(x)); // T - int
```

Свертывание ссылок

Стандарт определяет следующие правила свертки ссылок, применимые для определений `typedef` и `decltype`, а также параметров шаблонов:

- ▶ `A& &` становится `A&`
- ▶ `A& &&` становится `A&`
- ▶ `A&& &` становится `A&`
- ▶ `A&& &&` становится `A&&`

Свертывание ссылок

```
template <typename T>
struct A {
    typedef T&& TRef;
};

// ...
A<int&> x;

typedef int& && TRef; → typedef int& TRef;
```

Свертывание ссылок

Свертывание ссылок применяется при:

- ▶ инстанцировании шаблонов,
- ▶ генерации типа `auto`,
- ▶ `typedef` и `using`,
- ▶ `decltype`.

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 6
15.10.2021

Универсальные ссылки и rvalue-ссылки

```
class A {
public:
    template <typename T>
    void set(T&& x) {
        text = std::move(x);
    }
private:
    std::string text;
};

int main() {
    A obj;
    std::string text = "123";
    obj.set(text); // text теперь пусто
}
```

Универсальные ссылки и rvalue-ссылки

Тогда так:

```
class A {
public:
    void set(const std::string& x) {
        text = x;
    }
    void set(std::string&& x) {
        text = std::move(x);
    }

private:
    std::string text;
};
```

Универсальные ссылки и rvalue-ссылки

Воспользуемся std::forward:

```
class A {
public:
    template <typename T>
    void set(T&& x) {
        text = std::forward<T>(x);
    }
private:
    std::string text;
};
```

Перегрузка

```
void Do(std::set<std::string>& strings,
        const std::string& str) {
    std::cout << str << std::endl;
    strings.emplace(str);
}

int main() {
    std::set<std::string> strings;
    std::string s1("text");
    Do(strings, s1);
    Do(strings, "some");
    Do(strings, std::string("string"));
    return 0;
}
```

Перегрузка

Перепишем на универсальную ссылку:

```
template <typename T>
void Do(std::set<std::string>& strings, T&& str) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}

int main() {
    std::set<std::string> strings;
    std::string s1("text");
    Do(strings, s1);
    Do(strings, "some");
    Do(strings, std::string("string"));
    return 0;
}
```

Перегрузка

```
void Do(std::set<std::string>& strings, int x) {  
    strings.emplace(std::to_string(x));  
}
```

И внезапно ломается код:

```
short x = 2;  
Do(strings, x);
```

Функции с универсальными ссылками могут выполнить инстанцирование с точным соответствием практически любому типу.

Перегрузка: еще пример

```
class A {
private:
    std::string text;
public:
    template <typename T>
    explicit A(T&& str) : text(std::forward<T>(str)) {}

    explicit A(int x) : text(std::to_string(x)) {}

};

int main() {
    A x("123");
    auto copyX(x);
}
```

Перегрузка: еще пример

Класс после инстанцирования

```
class A {
private:
    std::string text;
public:
    explicit A(A& str) : text(std::forward<A&>(str)) {}
    A(const A& rhs); // сгенерировано компилятором
    explicit A(int x) : text(std::to_string(x)) {}
};
```

Прямая передача

```
template <typename T>
void fwd(T&& x) {
    f(std::forward<T>(x));
}
```

Целевая функция `f` должна получить в точности те же объекты, которые переданы функции `fwd`.

```
void f(const std::vector<int>& v);
f({0, 1, 0, 1});      // ok
fwd({0, 1, 0, 1});   // error
auto x = {0, 1, 0, 1}; // std::initializer_list<int>
fwd(x);              // ok
```

Перемещающие операции

Перемещающий конструктор и перемещающий оператор присваивания:

- ▶ генерируются только при необходимости;
- ▶ выполняют «почленное перемещение»;
- ▶ не генерируются при явном объявлении;
- ▶ не являются независимыми;
- ▶ не генерируются при явном объявлении копирующих операций или деструктора.

Перемещающие операции

Если все-таки нужно сгенерировать?

```
class A {  
public:  
    A(A&&) = default;  
    A& operator(A&&) = default;  
    virtual ~A() { ... }  
};
```

Некоторые выводы

- ▶ Перемещение — новая ключевая идея C++ — обычно используется для оптимизации копирования.
- ▶ std::move ничего не перемещает, std::forward ничего не передает.
- ▶ Не объявляйте объекты константными, если нужно выполнять перемещение из них.
- ▶ Применяйте std::move к rvalue-ссылкам, а std::forward к универсальным ссылкам.
- ▶ Перегрузка для универсальных ссылок может привести к неприятным эффектам (конструкторы с прямой передачей соответствуют неконстантным lvalue обычно лучше копирующих конструкторов)
- ▶ Большинство стандартных типов в C++11 перемещаемы, например, контейнеры STL.
- ▶ Некоторые типы только перемещаемы, например, объекты потоков, std::thread, std::unique_ptr.

Операторы new и delete

Зачем они нужны?

```
#include <iostream>
class C {
public:
    int arr[100];
    C(int a) { /*...*/ }
};

int main() {
    C * c = new C(123);
    // ...
    delete c;
}
```

- ▶ Создание и удаление динамических объектов

Проблема: Временем жизни таких объектов приходится управлять вручную.

Проблемы new и delete

1. Можно забыть написать delete.
2. Можно написать лишний delete.
3. Утечки памяти при исключениях и т.п.
4. delete / delete[].

Как решать?

- ▶ Оставить delete умным указателям.
- ▶ Оставить new make-функциям.

Но всё-таки про new/delete

Оператор **new** состоит из двух частей:

1. Выделение сырой (свободный кусок динамической) памяти. Может возникнуть исключение.
2. Конструирование объекта в сырой памяти.

Оператор **new** гарантирует, что если в конструкторе произошло исключение, то выделенная динамическая память автоматически очистится.

Оператор **delete** делает все наоборот:

1. Вызывается деструктор.
2. Освобождается память.

Размещающий оператор new

Как было раньше:

```
int * p = (int*)(malloc(sizeof(int));  
// ...  
free(p);
```

Два способа нельзя смешивать (malloc + delete, new + free)

Способ с new предпочтительнее, и его реализацию можно перегружать:

```
void *p = malloc(sizeof(C));  
C * c = new (p) C(123); // placement new;  
// ...  
c -> ~C();  
free(p);
```

Размещающий оператор new

Как-то надо бороться с исключениями:

```
void * p = malloc(sizeof(C));
if (!p) return 1;
C * c;
try {
    c = new (p) C(123);
} catch (...) {
    free(p);
    throw;
}
try {
    // ...
} catch (...) {
    c -> ~C();
    free(p);
    throw;
}
c -> ~C();
free(p);
```

operator new, operator delete

```
//new C(x)
void *p = operator new(sizeof(C));
C * c;
try {
    c = new(p) C(x);
} catch (...) {
    operator delete(p);
    throw;
}
//delete p
if (p!=NULL) {
    p->~C();
    operator delete(p);
}
```

Перегрузка

```
void * operator new (size_t sz) {
    std::cout << "operator new with " << sz << std::endl;
    void * p = malloc(sz);
    if (!p) throw std::bad_alloc();
    return p;
}

void operator delete(void * p) {
    std::cout << "operator delete" << std::endl;
    free(p);
}
```

Перегрузка

Можно перегрузить так:

```
void*operator new (size_t sz, double a, int x) {  
    // ...  
    return ::operator new(sz);  
}
```

Но тогда нужно написать парный ему

```
void operator delete(void * p, double a, int x) {  
    // ...  
    ::operator delete(p);  
}
```

Как тогда их вызвать?

```
C* p = new(1.23, 123) C(111);  
delete p;
```

Перегрузка

Оператор new/delete внутри класса обязан быть статическим.
static можно не писать.

```
class A {
    int param;
public:
    A(int a): param(a) {
        cout << "A::A(" << a << ")" << std::endl;
    }
    virtual ~A() { cout << "A::~A()" << std::endl; }
    static void* operator new(size_t sz) {
        cout << "A::operator new" << std::endl;
        return ::operator new(sz);
    }
    static void operator delete(void* ptr) {
        cout << "A::operator delete" << std::endl;
        ::operator delete(ptr);
    }
};
```

Аллокаторы

Класс, который выделяет память каким-то специальным образом.

```
map<int, string> m;  
// map<int, string, less<int>, allocator<pair<int, string>> > m;
```

Обязательно определяются:

- ▶ `value_type` — тип, для которого работает аллокатор
- ▶ `allocate` — выделение памяти под n объектов, но не конструирование
- ▶ `deallocate` — освобождение памяти

Необязательно:

- ▶ `construct` — инициализация заданной памяти заданным значением
- ▶ `destroy` — уничтожение памяти без освобождения

Указатели



Чем плохи обычные встроенные указатели?

Smart pointers

Чем плохи обычные встроенные указатели?

- ▶ Указывают на массив или на объект?
- ▶ Владеет ли указатель тем, на что указывает?
- ▶ Трудно обеспечить уничтожение ровно один раз.
- ▶ Обычно сложно определить, является ли указатель висячим.
- ▶ Нельзя предоставить информацию компилятору о том, могут ли два указателя указывать на одну область памяти.

«Умный» («интеллектуальный») указатель притворяется обычным указателем с дополнительными функциями.
Обертка над обычными указателями.

Smart pointers

Хочется что-то вроде такого

```
SmartPointer sp(new C);
```

и дальше пользоваться как обычным указателем, не задумываясь об удалении.

А что делать тут?

```
SmartPointer sp2 = sp;
```

Всегда можно обмануть умный указатель:

```
C * ptr = new C;
```

```
SmartPointer sp(ptr);
```

```
SmartPointer sp2(ptr);
```

Smart pointers: стратегии

- ▶ Запрет копирования и присваивания.
- ▶ Глубокое копирование.
- ▶ Подсчет ссылок в специальных счетчиках.
- ▶ Список ссылок.
- ▶ Передача владения.

Стратегия передачи владения

Если кто-то пытается скопировать указатель, то ему и передается владение, и исходный умный указатель не указывает больше на объект. Такой указатель не нужно класть в контейнер .

Умные указатели в C++ 11 / 14

```
std::auto_ptr<> // deprecated  
std::unique_ptr<>  
std::shared_ptr<>  
std::weak_ptr<>
```

std::unique_ptr

- ▶ Реализует семантику исключительного владения
- ▶ Перемещение передает владение от исходного указателя целевому, целевой при этом обнуляется.
- ▶ Копирование не разрешается.
- ▶ При деструкции освобождает ресурс, которым владеет.

Обычное использование – возвращаемый тип фабричных функций для объектов иерархии:

```
template <typename T>
std::unique_ptr<Base> makeObject(T&& params);
```

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 7
22.10.2021

std::unique_ptr

- ▶ Реализует семантику исключительного владения
- ▶ Перемещение передает владение от исходного указателя целевому, целевой при этом обнуляется.
- ▶ Копирование не разрешается.
- ▶ При деструкции освобождает ресурс, которым владеет.

Обычное использование – возвращаемый тип фабричных функций для объектов иерархии:

```
template <typename T>
std::unique_ptr<Base> makeObject(T&& params);
```

Некоторые методы std::unique_ptr

- ▶ `reset` — заменяет объект;
- ▶ `release` — освобождает владение;
- ▶ `get` — возвращает указатель на объект, которым владеет;

Запрещены неявные преобразования обычного указателя в умный:

```
std::unique_ptr<Base> p;  
p = new A();
```

std::unique_ptr

Две разновидности для индивидуальных объектов и для массивов:

```
std::unique_ptr<T> // *, ->  
std::unique_ptr<T[]> // []
```

std::unique_ptr можно присваивать в std::shared_ptr
(можно не задумываться над тем, как будет использован
возвращаемый указатель).

Как избавиться от new?

Написать обертку!

```
template <typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params) {
    return std::unique_ptr<T>(
        new T(std::forward<Ts>(params)...))
}
```

Чего не хватает? Массивов, пользовательских удалителей.
Функция уже есть — std::make_unique в C++14.

```
std::unique_ptr<Base> p(new Base); // дважды пишем Base
auto p1(std::make_unique<Base>()); // make
```

Пользовательские удалители

```
auto Deleter = [](Base* p) {
    std::cout << "delete" << std::endl;
    delete p;
};

using TPtr = std::unique_ptr<Base, decltype(Deleter)>

TPtr BuildObject(int param) {
    TPtr p(nullptr, Deleter);
    if (param == 1) {
        p.reset(new A());
    } else if (param == 2) {
        p.reset(new B());
    }
    return p;
}
```

Пользовательские удалители

- ▶ Тип удалителя является параметром шаблона
- ▶ Пользовательские удалители могут вообще говоря увеличить размер `std::unique_ptr`

std::shared_ptr

- ▶ Реализует семантику совместного владения.
- ▶ Использует метод подсчета ссылок. Счетчик ссылок хранится в динамически выделяемой памяти. Объект про счетчик ссылок ничего не знает.
- ▶ Тип удалителя не является частью типа указателя.
- ▶ Может работать только для указателей на объекты.

Что происходит здесь?

```
sp1 = sp2;
```

Некоторые методы std::shared_ptr

- ▶ `use_count` — количество shared ptr'ов, ссылающихся на этот же объект;
- ▶ `reset` — заменяет объект;
- ▶ `unique` — проверяет, что объект контролируется единственным указателем shared ptr;
- ▶ `get` — возвращает указатель на объект, которым владеет;

std::shared_ptr

- ▶ Перемещение быстрее копирования.
- ▶ Счетчик ссылок хранится в динамически выделяемой памяти.
- ▶ Пользовательский удалитель не является частью типа указателя.

Управляющий блок

- ▶ Функция `std::make_shared` всегда создает управляющий блок.
- ▶ Управляющий блок создается, когда указатель `std::shared_ptr` создается из указателя с исключительным владением
- ▶ Когда конструктор `std::shared_ptr` вызывается с обычным указателем — он создает управляющий блок.

Типичная ошибка

```
A* p = new A;  
std::shared_ptr<A> sptr1(p);  
std::shared_ptr<A> sptr2(p);
```

Пользовательские удалители

```
auto d1 = [](A* p) {
    std::cout << "delete 1" << std::endl;
    delete p;
};

auto d2 = [](A* p) {
    std::cout << "delete 2" << std::endl;
    delete p;
};

int main() {
    std::shared_ptr<A> sptr1(new A, d1);
    std::shared_ptr<A> sptr2(new A, d2);
    sptr1 = sptr2;
    return 0;
}
```

- ▶ Тип удалителя не является частью типа указателя.
- ▶ Пользовательский удалитель не влияет на размер указателя.

std::weak_ptr

Дополнение функциональности std::shared_ptr.

- ▶ не участвует в совместном владении,
- ▶ позволяет понять, не является ли указатель висячим,
- ▶ нельзя ни разыменовать, ни проверить на `nullptr`.

```
auto sp = std::make_shared<A>();
std::weak_ptr<A> wp(sp);
```

```
// sp.use_count() == 1, wp.expired() == false;
// нельзя написать *wp, можно написать *sp
```

```
sp = nullptr;
// sp.use_count() == 0, wp.expired() == true;
```

std::weak_ptr

Разыменование происходит через преобразование к std::shared_ptr<>:

- ▶ Через функцию lock() – удаленный объект соответствует nullptr.
- ▶ Прямая конвертация – удаленный объект вызывает исключение.

```
auto sp2 = wp.lock();
// sp2 нулевой, если wp expired
```

```
std::shared_ptr<A> sp3(wp);
// exception std::bad_weak_ptr, если wp expired
```

std::weak_ptr: зачем?

Пусть есть фабрика объектов:

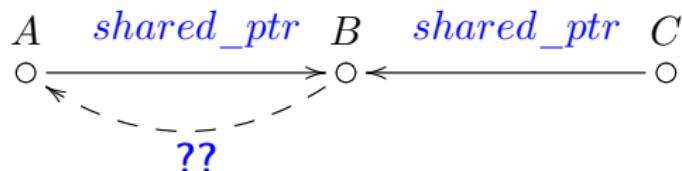
```
std::unique_ptr<const T0bject> BuildObject(int param);
```

Кэш с удалением неиспользованных кэшированных значений.

```
std::shared_ptr<const T0bject> FastBuildObject(int param) {
    static std::unordered_map<int,
                           std::weak_ptr<const T0bject>>
        cache;
    auto objPtr = cache[param].lock();
    if (!objPtr) {
        objPtr = BuildObject(param);
        cache[param] = objPtr;
    }
    return objPtr;
}
```

Предупреждение циклов std::shared_ptr

Рассмотрим такую структуру совместного владения:



Каким должен быть указатель?

- ▶ raw pointer
- ▶ shared_ptr
- ▶ weak_ptr

Счетчик слабых ссылок

Время между уничтожением последнего `shared_ptr` и `weak_ptr` значительно.

```
auto sp = std::make_shared<A>();  
// создание shared_ptr, weak_ptr  
// ..  
// удаление всех shared_ptr  
// ... (!)  
// удаление последнего weak_ptr
```

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 8
29.10.2021

std::make_shared

Где тут потенциальная проблема?

```
Do(std::shared_ptr<A>(new A), getItem());
```

Исправляем:

```
Do(std::make_shared<A>(), getItem());
```

+ единовременное выделение памяти под объект и счетчик ссылок

Но в `std::make_shared` нельзя использовать custom-удалитель:

```
std::shared_ptr<A> sp(new A, customDeleter);
```

Как тогда исправить

```
Do(std::shared_ptr<A>(new A, customDeleter), getItem());
```

чтобы было безопасно?

```
std::shared_ptr<A> sp(new A, customDeleter);
void Do(sp, getItem());
```

Чего не хватает теперь для оптимальности?

Другие умные указатели

- ▶ `intrusive_ptr` — облегченная версия `shared_ptr` для классов, имеющих встроенные механизмы подсчёта ссылок.
- ▶ `scoped_ptr` — аналог `const auto_ptr` с запрещенными конструктором копирования и оператором присваивания.
- ▶ `copy_ptr`
- ▶ `linked_ptr`
- ▶ ...

intrusive_ptr

```
#include "intrusive_ptr.h"
class TDoc: public TRefCounter<TDoc> {
    // ...
}

// ...

TIntrusivePtr<TDoc> ptr = nullptr;
ptr = MakeIntrusive<TDoc>();
```

Владение памятью

Что такое правильно спроектированное владение памятью?

- ▶ в каждой точке программы понятно, кто каким объектом владеет?
- ▶ в каждой точке программы понятно, кто владеет данным объектом или что владение не меняется?

Владение памятью

Что такое правильно спроектированное владение памятью?

Ф.Пикус определяет так:

- ▶ Если некоторая функция или класс никак не изменяет владение памятью, то это понятно каждому клиенту и автору
- ▶ Если некоторая функция или класс принимает уникальное владение объектами, то это должно быть понятно клиенту
- ▶ Если некоторая функция или класс разделяет владение объектом, то это должно быть понятно клиенту
- ▶ Для любого созданного объекта в любой точке использования понятно, должен код удалить объект или нет

Владение памятью

Что такое неправильно спроектированное владение памятью?

- ▶ Нужна дополнительная информация. Например, кто владеет возвращаемым объектом в этом коде:

```
TObjectFactory factory;  
TObject* p = factory.MakeObject();
```

- ▶ А что с владением здесь:

```
TObject* p1 = Process(p);
```

- ▶ Функция принимает владение вектором, а зачем:

```
void Double(std::shared_ptr<std::vector<int>> v) {  
    if (!v) return;  
    for (auto& x: *v) x *= 2;  
}  
  
...  
std::shared_ptr<std::vector<int>> sv(...);  
Double(sv);
```

Владение памятью

В терминах синтаксиса умных указателей можно говорить про разные стратегии владения, передачи владения, преобразование владения.

Какой тип владения самый распространенный?

Владение памятью: невладение

```
void Process(TObject *p); // я не буду удалять p  
void Do(TObject& p); // я тоже
```

Невладеющие указатели, ссылки.

not null

У указателей есть nullptr. И это значение всегда нужно проверять. CppCoreGuidelines:

I.12: Declare a pointer that must not be null as not_null

Reason To help avoid dereferencing nullptr errors. To improve performance by avoiding redundant checks for nullptr.

Example

```
int length(const char* p);           // it is not clear whether length(nullptr) is valid  
length(nullptr);                  // OK?  
  
int length(not_null<const char*> p); // better: we can assume that p cannot be nullptr  
  
int length(const char* p);           // we must assume that p can be nullptr
```

By stating the intent in source, implementers and tools can provide better diagnostics, such as finding some classes of errors through static analysis, and perform optimizations, such as removing branches and null tests.

not null

- ▶ CompileTime: код не компилируется, если объекту `not_null<T*>` присвоить `nullptr`.
- ▶ RunTime: поведение программы переопределяется (исключение, игнор, `terminate`, ...)
- ▶ Это расширение языка

Владение памятью: уникальное владение

```
void f() {  
    T0bject o;  
    Do(o);  
    Process(o);  
}
```

- ▶ Локальные переменные
- ▶ std::unique_ptr

Владение памятью: уникальное владение

Обычное использование — создание объектов фабриками:

```
std::unique_ptr<TObject> p(ObjetFactory());
```

Что должна возвращать ObjectFactory?

Мы хотим, чтобы клиент этой фабрики был вынужден использовать её так, чтобы принимать владение.

```
std::unique_ptr<TObject> ObjectFactory() {  
    return std::unique_ptr<TObject>(...);  
}
```

```
void Process(TObject* p);
```

```
// ...
```

```
std::unique_ptr<TObject> p(ObjetFactory()); // ok
```

```
Process(ObjetFactory()); // compile error
```

```
Process(p.get()); // ok
```

Владение памятью: разделяемое владение

- ▶ Несколько сущностей владеют объектом на равных основаниях
- ▶ `std::shared_ptr`
- ▶ Совместное владение быть избыточно
- ▶ Если функция принимает `std::shared_ptr` в качестве параметра, она извещает клиента о том, что намеревается принять частичное владение
- ▶ Монопольное владение предпочтительнее — его проще отследить и оно более эффективно.

Где проблемы?

```
class A;  
using VecPtr = std::vector<std::shared_ptr<A>>;  
  
class A {  
public:  
    void process(VecPtr& done) {  
        done.emplace_back(this);  
    }  
};  
  
int main() {  
    std::shared_ptr<A> p(new A);  
    VecPtr done;  
    p->process(done);  
    return 0;  
}
```

Конструируемый shared_ptr создает новый управляющий блок.

Задача

```
class A;
using VecPtr = std::vector<std::shared_ptr<A>>;

class A : public std::enable_shared_from_this<A>{
public:
    void process(VecPtr& done) {
        done.emplace_back(shared_from_this());
    }
};

int main() {
    std::shared_ptr<A> p(new A);
    VecPtr done;
    p->process(done);
    return 0;
}
```

intrusive_ptr

```
#include "intrusive_ptr.h"
class TDoc: public TRefCounter<TDoc> {
    // ...
}

// ...

TIntrusivePtr<TDoc> ptr = nullptr;
ptr = MakeIntrusive<TDoc>();
```

Идиома CRTP

The curiously recurring template pattern. Класс отнаследован от шаблонного класса, в котором наследник – аргумент шаблона:

```
template <typename Derived>
class CuriousBase {
    // ...
};

class Curious : public CuriousBase<Curious> {
    // ..
};
```

Coplien, James O. (February 1995).

Пример: enable_shared_from_this.

Идиома CRTP

```
template <typename Derived>
class CuriousBase {
public:
    void f(int x) { static_cast<Derived*>(this)->f(x);}
protected:
    int y;
};

class Curious : public CuriousBase<Curious> {
public:
    void f(int x) { y += x; }
};

// ...
CuriousBase<Curious>* b = ...;
b->f(10); // без механизма виртуальных функций!
```

Идиома CRTP

Для любого класса, использующего оператор равенства, сделать автоматически оператор неравенства (как инверсию первого):

```
template <typename D>
struct not_eq {
    bool operator != (const D& rhs) const {
        return !static_cast<const D*>(this)->operator==(rhs);
    }
};

class C: public not_eq<C> {
public:
    bool operator == (const C& rhs) const {
        // ...
    }
};
```

Идиома CRTP

Ограничеваем число объектов класса.

```
#include <stdexcept>
template <typename T, size_t maxN>
class LimitedInstances {
    static size_t counter;
protected:
    LimitedInstances() {
        if (counter >= maxN) {
            throw std::logic_error("too many instances");
        }
        ++counter;
    }
    ~LimitedInstances() {
        --counter;
    }
};

template <typename T, size_t maxN>
size_t LimitedInstances<T, maxN>::counter(0);
```

Идиома CRTP

```
class oneInst: public LimitedInstances<oneInst, 1> {};
class twoInst: public LimitedInstances<twoInst, 2> {};

int main() {
    oneInst obj;
    try {
        oneInst();
    } catch (std::logic_error &e) {
        std::cerr << "Caught: " << e.what() << std::endl;
    }
}

twoInst obj1;
twoInst obj2;
try {
    twoInst();
} catch (std::logic_error &e) {
    std::cerr << "Caught: " << e.what() << std::endl;
}
};

};
```

Идиома CRTP

- ▶ Часто заменяют динамический полиморфизм через статический: в базовом классе вызываем методы класса, которым он параметризован при инстанцировании
- ▶ Техника реализации, при которой общая функциональность предоставляется нескольким производным базовым классам, и каждый расширяет и настраивает интерфейс шаблона базового класса.

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 9
12.11.2021

Идиома PImpl

Pointer to implementation.

Метод, при котором члены-данные класса заменяются указателем на класс реализации с этими данными.

a.h:

```
#include "myitems.h"
class A {
    TMyItem item1, item2;
public:
    A();
    // ...
};
```

Break compilation dependencies!

a.h:

```
class A {  
    struct Impl;  
    Impl *pImpl;  
public:  
    A();  
    // ...  
};
```

a.cpp:

```
#include "a.h"  
#include "myitems.h"  
struct A::Impl {  
    TMyItem item1, item2;  
};  
  
A::A() : pImpl(new Impl) {}  
A::~A() {delete pImpl;}
```

Идиома PImpl: C++11

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
public:
    A();
    // ...
};
```

a.cpp:

```
#include "a.h"
#include "myitems.h"
struct A::Impl {
    TMyItem item1, item2;
};
```

```
A::A() : pImpl(std::make_unique<A::Impl>()) {}
```

main.cpp:

```
#include "a.h"
A a; // !error
```

Идиома PImpl

Нужно обеспечить полноту в точке уничтожения
std::unique_ptr<A::Impl>.

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
public:
    A();
    ~A();
    // ...
};
```

a.cpp:

```
~A::A() = default;
```

Идиома PImpl

Нужны перемещающие функции:

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
public:
    A();
    ~A();
    A(A&& other) = default;
    A& operator=(A&& other) = default;
    // ...
};
```

И снова та же проблема!

Идиома PImpl

Объявляем в заголовочном файле, реализуем в файле реализации:

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
public:
    A();
    ~A();
    A(A&& other);
    A& operator=(A&& other);
    // ...
};
```

a.cpp:

```
A::A(A&& other) = default;
A& A::operator=(A&& other) = default;
```

Идиома PImpl

Потребуются копирующие операции:

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
public:
    A();
    ~A();
    A(A&& other);
    A& operator=(A&& other);
    A(const A& other);
    A& operator=(const A& other);
    // ...
};
```

Идиома PImpl

a.cpp:

```
A::A(const A& other) : pImpl(nullptr) {
    if (other.pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    }
}

A& A::operator=(const A& other) {
    if (!other.pImpl) {
        pImpl.reset();
    } else if (!pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    } else {
        *pImpl = *other.pImpl;
    }
    return *this;
}
```

В случае `std::shared_ptr` всё проще! Тип удалителя не является частью типа указателя и указываемые типы не обязательно должны быть полными.

Паттерны проектирования

- ▶ Способ построения кода для решения часто встречающихся проблем проектирования
- ▶ successful stories
- ▶ Готовые абстракции для решения классов проблем + унификация деталей и названий
- ▶ Не нужно их употреблять везде, не нужно им строго следовать

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования.

[contains a lot of «ancient» C++ code]

Ф. Пикус Идиомы и паттерны проектирования в современном C++.

Пример: паттерн Bridge

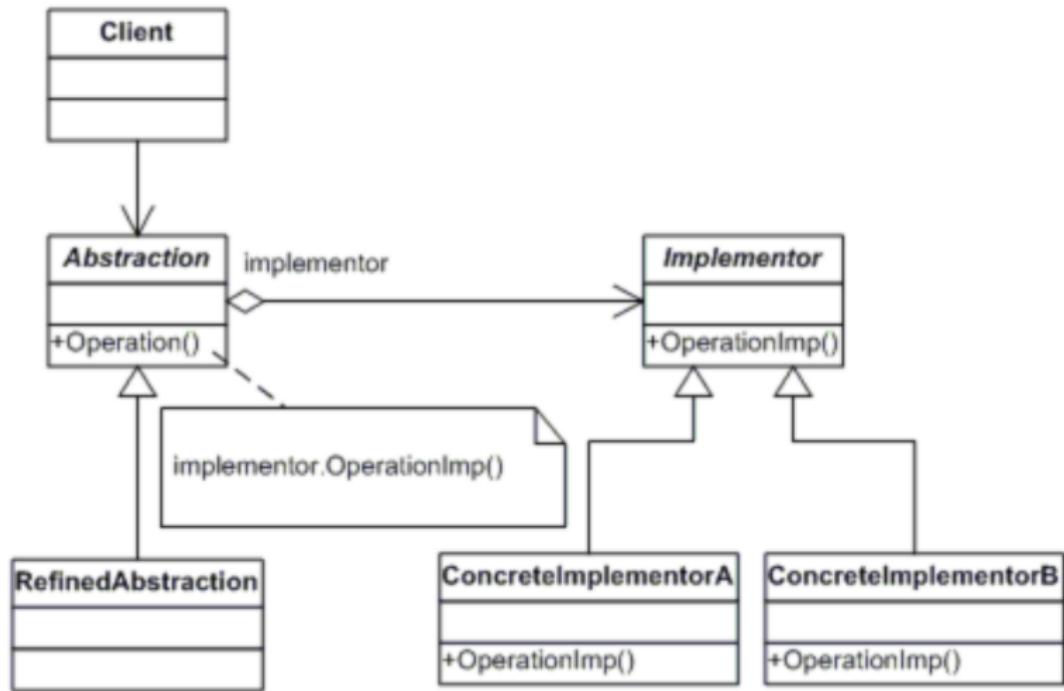
Цель: разделить абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

Почему не наследование: наследование жестко привязывает реализацию к абстракции. Это затрудняет расширение и повторное использование абстракции и ее реализаций.

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Основной класс содержит указатель на реализацию `impl`, который используется для перенаправления пользовательских запросов в неё.

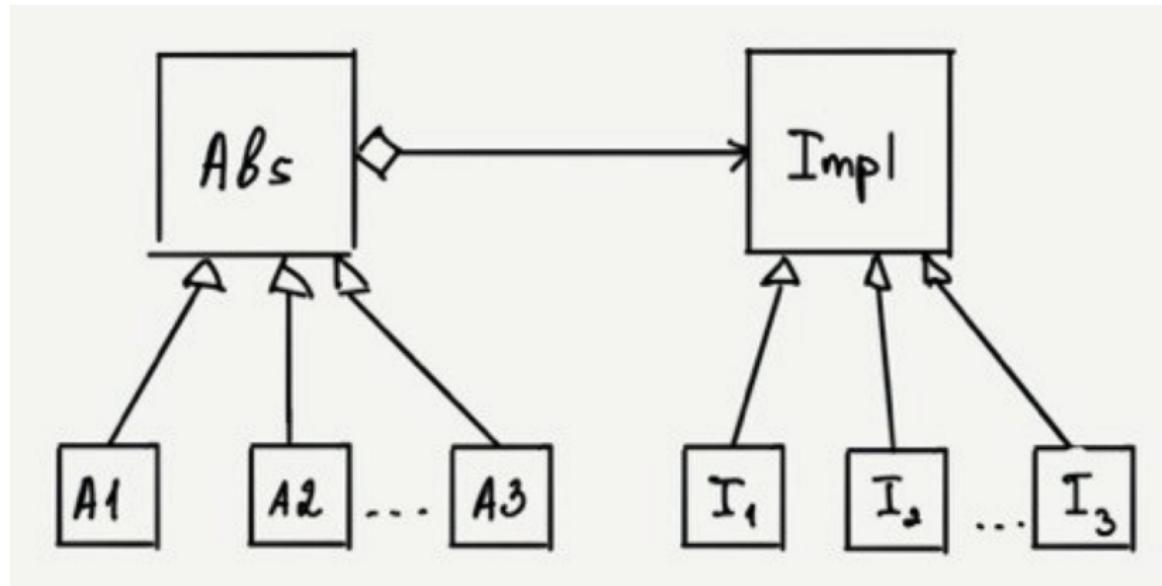
Все детали реализации, связанные с какими-либо особенностями находятся во *второй иерархии*.

Пример: паттерн Bridge

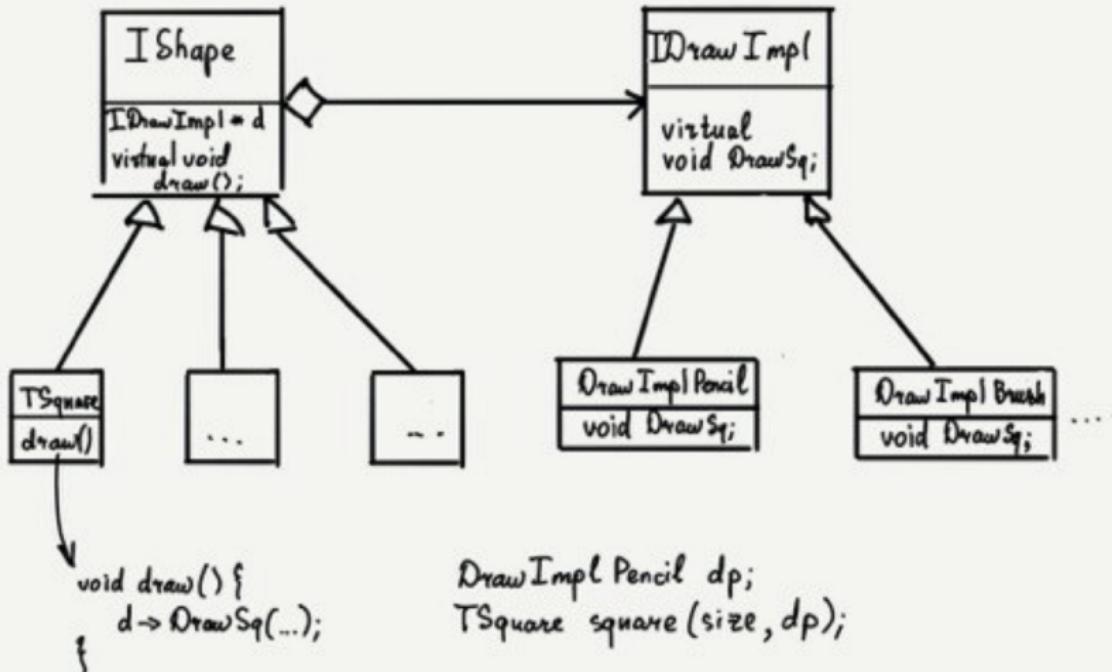


Abstraction перенаправляет объекту Implementation запросы клиента.

Пример: паттерн Bridge



Пример: паттерн Bridge



Пример: паттерн Bridge

Когда: когда нужно часто изменять реализацию какого-нибудь метода с сохранением API

Когда: когда используется постоянно изменяющаяся внешняя библиотека

Когда: когда нужно добиться разделения ответственности между классами

Пример: паттерн Bridge

В чем отличие от PIMPL?

- ▶ PIMPL — способ скрыть реализацию, в основном для того, чтобы убрать зависимости
- ▶ Bridge — поддержка множественных реализаций, а в PIMPL обычно не изменяется реализация, это отдельно компилируемый класс
- ▶ PIMPL — идиома проектирования на уровне файлов с кодом, Bridge — паттерн объектно-ориентированного проектирования.

Пример: паттерн Command

Инкапсулирует запрос в виде объекта, делая возможной параметризацию клиентских объектов с другими запросами, организацию очереди или регистрацию запросов, а также поддержку отмены операций.

Пример: паттерн Command

```
struct talk {
    void operator()(){
        std::cout << "croak!" << std::endl;
    }
};

struct walk {
    void operator()() {
        std::cout << "im walking" << std::endl;
    }
};

struct jump {
    void operator()() {
        std::cout << "jump" << std::endl;
    }
};
```

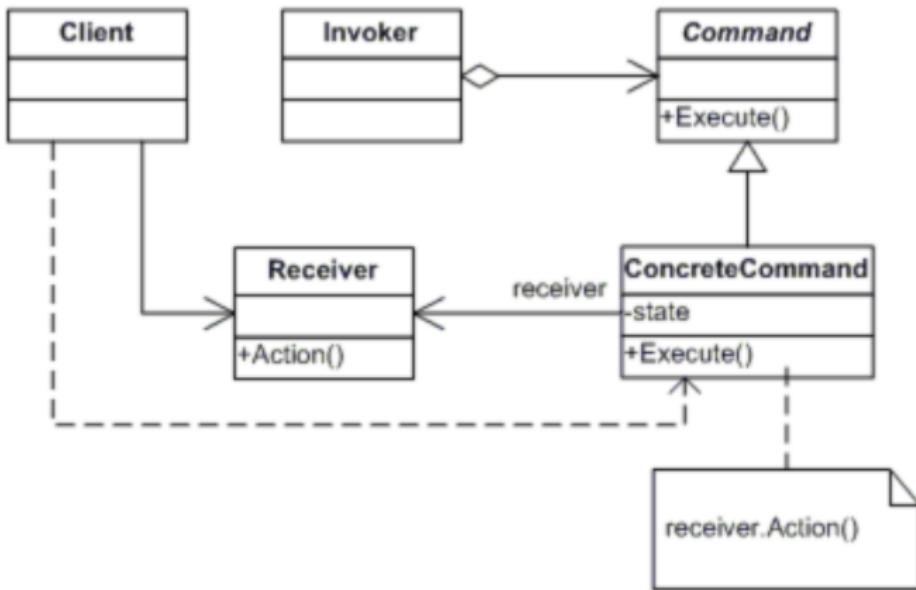
Пример: паттерн Command

```
void dofunc(std::function<void()> f) {
    f();
}

int main() {
    dofunc(talk{});
    dofunc(walk{});
    auto f = jump();
    dofunc(f);
}
```

Если все действия пользователя в программе реализованы в виде командных объектов, программа может сохранить стек последних выполненных команд.

Пример: паттерн Command



Client — среда генерации команд; **Receiver** — знает, как провести операцию, связанную с командной; **Command** — инкапсуляция действия; **Invoker** — последующие действия с командой или пулом команд.

Пример: паттерн Command

- ▶ Undo-Redo
- ▶ Организация очереди при многопоточной обработке;
- ▶ Транзакционные алгоритмы - регистрация событий и восстановление после сбоя;

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 10
19.11.2021

Пример: паттерн Singleton

Глобальные переменные — это некоторое зло.

a.cpp:

```
std::vector<int> va;  
//...
```

b.cpp:

```
extern std::vector<int> va;  
struct TInit {  
    TInit() { va.push_back(1);}  
};  
TInit Init;
```

Порядок инициализации?

Глобальные объекты → local static объекты:

```
std::vector& GetVal() {  
    static std::vector<int> va;  
    return va;  
}
```

Пример: паттерн Singleton

Singleton — класс, у которого в любой момент времени существует не более одного объекта.

Когда можно использовать:

- ▶ Физическая причина для единственного объекта
(физическая величина, автомобиль для программы им управляемой, ...)
- ▶ Глобальные объекты из проектных соображений
(диспетчеры ресурсов, ...)

Реализация через local static объекты потокобезопасная.

Пример: паттерн Singleton

```
class Singleton {  
private:  
    Singleton(){}  
    static Singleton* instance;  
public:  
    // data  
    // ...  
    Singleton(const Singleton&) = delete;  
    static Singleton* Instance() {  
        if (instance == nullptr) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
};  
  
Singleton* Singleton::instance = nullptr;
```

Утечки памяти!

Пример: паттерн Singleton

Синглтон С. Мейерса.

```
class Singleton {  
protected:  
    Singleton(){ /*...*/}  
    ~Singleton(){ /*...*/}  
public:  
    // data  
    // ...  
    Singleton(const Singleton&) = delete;  
    Singleton(Singleton&&) = delete;  
    Singleton& operator=(Singleton const&) = delete;  
    Singleton& operator=(Singleton &&) = delete;  
    static Singleton& Instance() {  
        static Singleton instance;  
        return instance;  
    }  
};
```

Единственный способ получить доступ — обратиться к
`Singleton::Instance()`.

Singleton + PImpl

a.h:

```
struct SingletonImpl;  
class Singleton {  
public:  
    // api  
private:  
    static SingletonImpl& impl();  
};
```

a.cpp:

```
struct SingletonImpl{  
    SingletonImpl() {...}  
};  
SingletonImpl& Singleton::impl() {  
    static SingletonImpl instance;  
    return instance;  
}
```

Пример: паттерн Singleton

Почему это плохой паттерн?

- ▶ Это скрытие глобальной переменной — в обход всего к ней можно получить доступ.
- ▶ Сложно работать с наследованием.
- ▶ Невозможно простым способом развернуть код в несколько функций с разными объектами-синглтонами.

Пример: паттерн Strategy

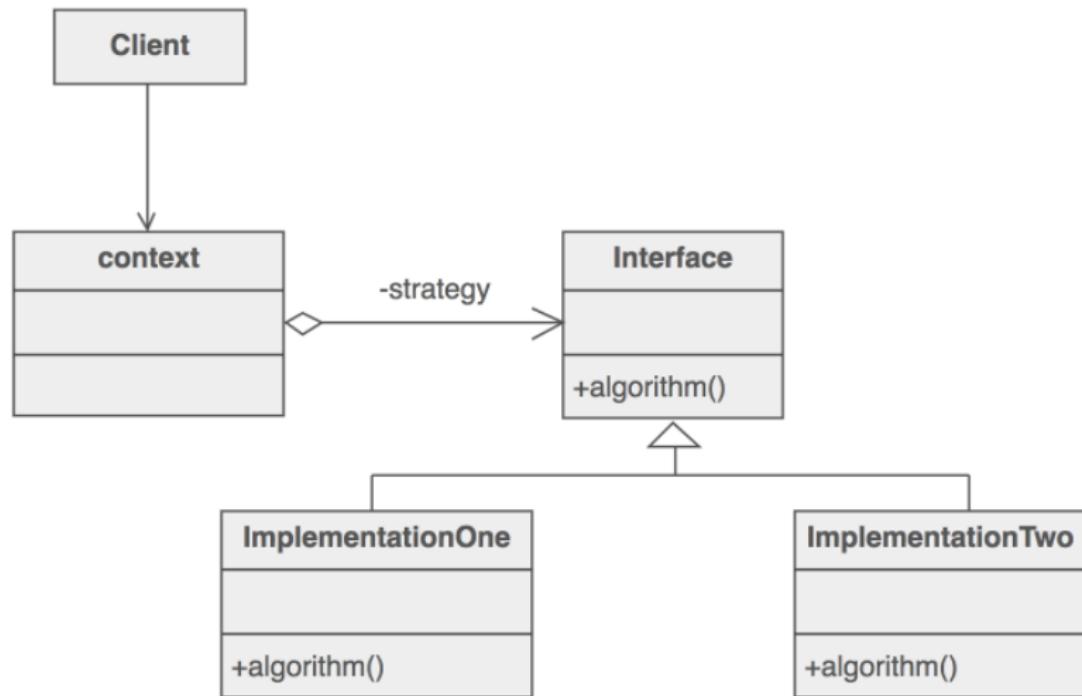
Паттерн, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.

- ▶ Инкапсуляция алгоритма,
- ▶ увеличение модульности и проверяемости кода,
- ▶ дешевое масштабирование кода,
- ▶ выбор алгоритма, основываясь на данных (в процессе исполнения кода можно это изменить).

Когда?

- ▶ Нужны разные варианты алгоритма или поведения,
- ▶ нужно изменять поведение объектов в runtime,
- ▶ нужны разные алгоритмы в зависимости от состояния.

Пример: паттерн Strategy



Пример: паттерн Strategy и кофе-машина

```
class Recipe {
public:
    virtual double GetAmountOfWater() const = 0;
    virtual void Make() = 0;
};

class HotBeverage {
    void BoilWater(double amount) {
        std::cout << "boiling " << amount << " ml of water..." 
        << std::endl;
    }
    void Pour() {
        std::cout << "pouring in cup" << std::endl;
    }
    std::shared_ptr<Recipe> recipe;
public:
    HotBeverage(std::shared_ptr<Recipe> r) : recipe(r) {}
    void prepare() {
        BoilWater(recipe->GetAmountOfWater());
        recipe->Make();
        Pour();
    }
};
```

Пример: паттерн Strategy и кофе-машина

```
class Coffee: public Recipe {
    double AmountOfWater;
    int StrongLevel;
public:
    Coffee(double amountOfWater, int level)
        : AmountOfWater(amountOfWater)
        , StrongLevel(level)
    { }
    virtual double GetAmountOfWater() const { return AmountOfWater; }
    virtual void Make() { std::cout << "brewing coffee..."; }
};

class HotChocolate : public Recipe {
    double AmountOfWater;
public:
    HotChocolate(double amountOfWater)
        : AmountOfWater(amountOfWater)
    { }
    virtual double GetAmountOfWater() const { return AmountOfWater; }
    virtual void Make() { std::cout << "making hot chocolate..."; }
};
```

Пример: паттерн Strategy и кофе-машина

```
int main() {
    auto coffee = std::make_shared<Coffee>(200, 3);
    auto hotChocolate = std::make_shared<HotChocolate>(100);
    std::vector<HotBeverage> beverages = {
        HotBeverage(coffee),
        HotBeverage(hotChocolate)
    };
    for (auto&x : beverages) x.prepare();
}
```

Пример: паттерн Strategy и кофе-машина через лямбды

```
class HotBeverage {
    void BoilWater(double amount) {
        std::cout << "boiling " << amount << " ml of water...";
    }
    void Pour() {
        std::cout << "pouring in cup" << std::endl;
    }
    std::function<double()> GetAmountOfWater;
    std::function<void()> Make;
public:
    HotBeverage(std::function<double()> getAmountOfWater,
                std::function<void()> make)
        : GetAmountOfWater(getAmountOfWater)
        , Make(make) {}
    void prepare() {
        BoilWater(GetAmountOfWater());
        Make();
        Pour();
    }
};
```

Пример: паттерн Strategy и кофе-машина через лямбды

```
static void MakeCofee() { std::cout << "brewing coffee..."; }
static void MakeHotChocolate() { std::cout << "making chocolate..."; }
static double GetAmountOfWater(double amount) { return amount; }

int main() {
    auto coffee = HotBeverage(
        [] { return GetAmountOfWater(200); },
        MakeCofee
    );
    auto hotChocolate = HotBeverage(
        [] { return GetAmountOfWater(100); },
        MakeHotChocolate
    );
    std::vector<HotBeverage> beverages = {
        coffee, hotChocolate
    };
    for (auto&x : beverages) x.prepare();
}
```

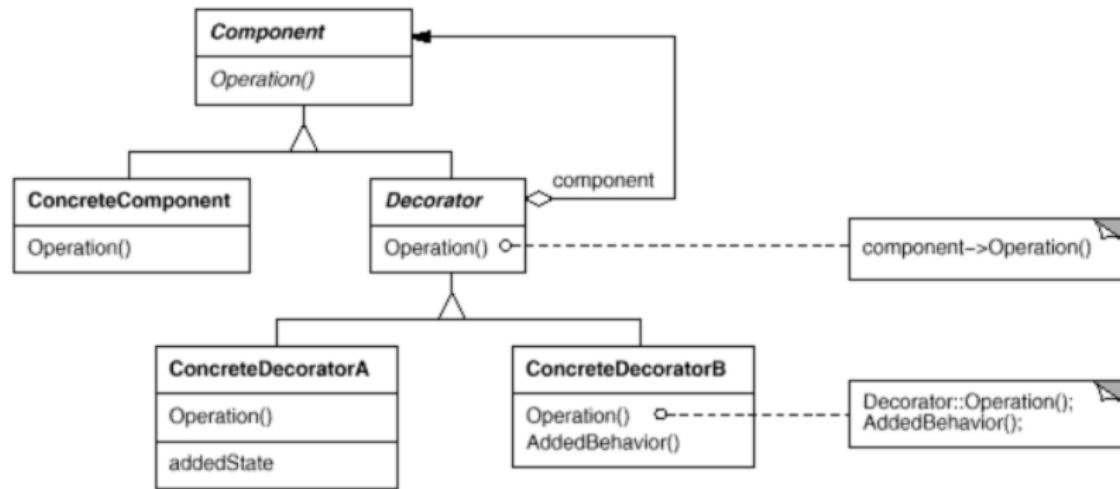
Пример: паттерн Decorator

Динамически добавляет дополнительное поведение объекту.

Декоратор создает список объектов-оберток над другими объектами. Они наследуются от того же самого интерфейса.

Перегрузкой методов можно либо использовать исходные варианты, либо добавлять свою функциональность.

Пример: паттерн Decorator



- ▶ Декоратор имеет тот же интерфейс, что и Component (использование декоратора).
- ▶ Декоратор содержит указатель на конкретный Component (реализация декоратора).

Пример: паттерн Decorator

```
class TWriterInterface {
public:
    virtual ~TWriterInterface() = default;
    virtual void Write(const std::string& s) = 0;
};

class TStandardWriter : public TWriterInterface {
public:
    virtual ~TStandardWriter() = default;
    virtual void Write(const std::string& s) { std::cout << s
                                                << std::endl; }
};
using TWriterInterfacePtr = std::unique_ptr<TWriterInterface>;

class Decorator : public TWriterInterface {
    TWriterInterfacePtr Interface;
public:
    Decorator(TWriterInterfacePtr ptr) { Interface = std::move(ptr); }
    virtual void Write(const std::string& s) override {
        Interface->Write(s);
    }
};
```

Пример: паттерн Decorator

```
class DecoratorWithBorder : public Decorator {
    std::string Name;
public:
    DecoratorWithBorder(TWriterInterfacePtr ptr, const std::string& n)
        : Decorator(std::move(ptr))
        , Name(n)    {}
    virtual void Write(const std::string& s) override {
        std::cout << "==== " << Name << " ===" << std::endl;
        Decorator::Write(s);
        std::cout << "====" << std::string(Name.size(), '=')
              << "====" << std::endl;
    }
};

class DecoratorWithExclamation : public Decorator {
public:
    DecoratorWithExclamation(TWriterInterfacePtr ptr)
        : Decorator(std::move(ptr)) {}
    virtual void Write(const std::string& s) override {
        std::cout << "ATTENTION!!!" << std::endl;
        Decorator::Write(s);
    }
};
```

Пример: паттерн Decorator

```
int main() {
    TWriterInterfacePtr writer = std::make_unique<TStandardWriter>();
    writer->Write("some information");
}
```

```
some information
```

Пример: паттерн Decorator

```
int main() {
    TWriterInterfacePtr writer = std::make_unique<TStandardWriter>();
    TWriterInterfacePtr writer2 =
        std::make_unique<DecoratorWithBorder>(
            std::move(writer), "Magic");
    writer2->Write("some information again");
}
```

```
==== Magic ===
some information again
=====
```

Пример: паттерн Decorator

```
int main() {
    TWriterInterfacePtr writer = std::make_unique<TStandardWriter>();
    writer->Write("some information");
    TWriterInterfacePtr writer2 =
        std::make_unique<DecoratorWithBorder>(
            std::move(writer), "Magic");
    TWriterInterfacePtr writer3 =
        std::make_unique<DecoratorWithExclamation>(std::move(writer2));
    writer3->Write("some information again and again");
}
```

ATTENTION!!!

==== Magic ===

some information again and again

=====

Пример: паттерн Decorator

Feature: возможность кастомизации и конфигурации ожидаемого поведения. Работа начинается с пустым объектом, который имеет базовую функциональность. Затем происходит выбор декораторов, обрабатывающих и обогащающих базовый объект.

Наследование или Декоратор?

- ▶ В случае декоратора проще изменять объекты в run-time.
- ▶ Проще создавать множественные изменения поведений.
- ▶ Если динамически менять поведение объекта не нужно — не нужен и декоратор, наследование может быть проще.

Стратегия? Декоратор?

- ▶ Декораторы обрабатывают объект снаружи, стратегии же вставляются в него внутрь по неким интерфейсам.
- ▶ Недостаток стратегии: класс должен быть спроектирован с возможностью вставки стратегий.
- ▶ Недостаток декоратора: не всегда желательное смешение публичного интерфейса и интерфейса кастомизации.

Пример: паттерн Observer

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

- ▶ субъекты (объекты, которые могут изменяться)
- ▶ наблюдатели (объекты, уведомляемые при изменении состоянии)

Субъекты не заинтересованы в управлении временем жизни своих наблюдателей, но заинтересованы в том, чтобы если наблюдатель был уничтожен, субъекты не пытались к нему обратиться. Тогда так: каждый субъект хранит контейнер указателей `????_ptr` на своих наблюдателей.

Пример: паттерн Observer

```
class Observer {  
    std::string name;  
public:  
    Observer(const std::string& s) : name(s) {}  
    void Notify(const std::string& source) {/*...*/}  
};  
  
class Observable {  
    std::string name;  
public:  
    void Subscribe(std::shared_ptr<Observer> observer);  
    void Unsubscribe(std::shared_ptr<Observer> observer);  
    void Notify();  
    Observable(const std::string& s) : name(s) {}  
private:  
    std::vector<std::weak_ptr<Observer>> observers;  
};
```

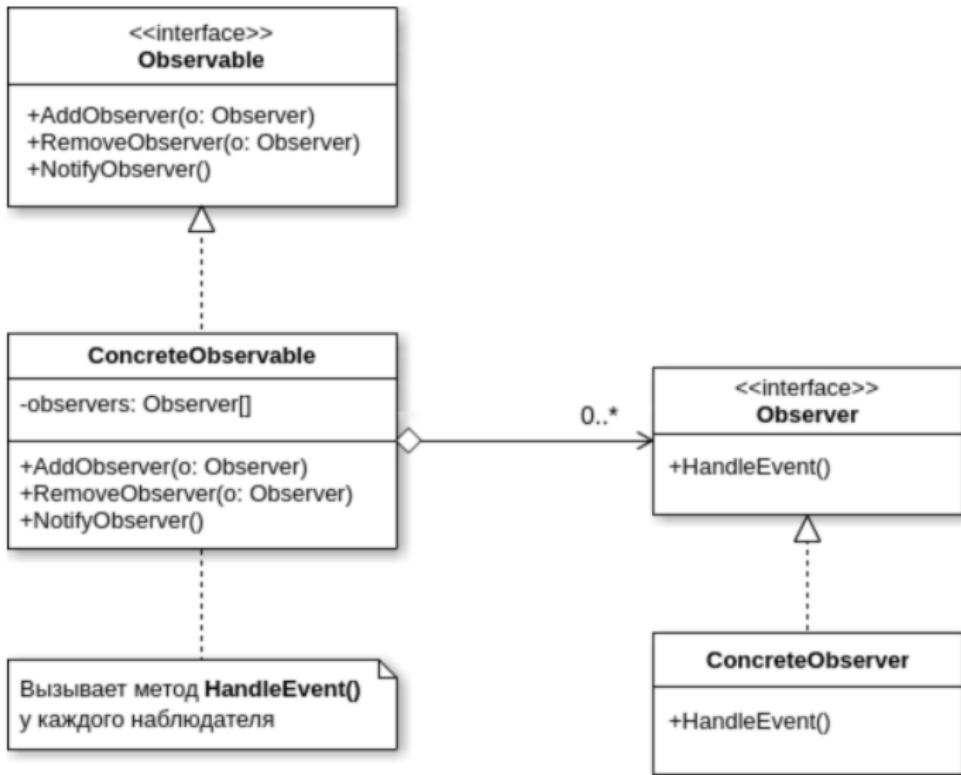
Пример: паттерн Observer

```
void Observable::Subscribe (std::shared_ptr<Observer> observer) {
    observers.push_back(observer);
}

void Observable::Notify() {
    for (auto wptr: observers) {
        if (!wptr.expired()) {
            auto observer = wptr.lock();
            observer->Notify(this->name);
        }
    }
}

void Observable::Unsubscribe(std::shared_ptr<Observer> observer) {
    observers.erase(
        std::remove_if(
            observers.begin(),
            observers.end(),
            [&](const std::weak_ptr<Observer>& wptr) {
                return wptr.expired() || wptr.lock() == observer;
            }
        ),
        observers.end());
}
```

Пример: паттерн Observer



Пример: Factory method

Паттерн проектирования, основывающийся на наследовании, при котором создание объекта делегируется подклассам, которые реализуют фабричный метод для создания объекта. Создание объектов по внешним параметрам.

Абстрактная фабрика — порождающий паттерн проектирования, позволяющий создавать группы взаимосвязанных или взаимозависимых объектов без указаний их конкретных классов. Во многих реализациях использует шаблон Фабричный метод.

Фабрика

Иерархия объектов:

```
// objects.h

class IObject {
public:
    virtual void Do() const = 0;
};

class TCustomObject : public IObject {
public:
    virtual void Do() const override {
        std::cout << "TCustomObject DO " << std::endl;
    }
};

class TSuperObject : public IObject {
public:
    virtual void Do() const override {
        std::cout << "TSuperObject DO " << std::endl;
    }
};
```

Фабрика

Как можно было бы сделать:

```
class TFactory {
public:
    virtual IObject* Create(const std::string& name) {
        if (name == "custom object") {
            return new TCustomObject();
        } else if (name == "...")
        ....
        else return nullptr;
    }
};
```

Фабрика

```
//factory.h

class TFactory {
    class TImpl;
    std::unique_ptr<const TImpl> Impl;

public:
    TFactory();
    ~TFactory();
    std::unique_ptr<IObject> CreateObject(
        const std::string& name
        /*, const TOptions opts*/) const;
    std::vector<std::string> GetAvailableObjects() const;
};
```

Фабрика

```
// factory.cpp

#include "factory.h"
class TFactory::TImpl {

    class ICreator {
        public:
            virtual ~ICreator(){}
            virtual std::unique_ptr<IOBJECT> Create() const = 0;
    };
    using TCreatorPtr = std::shared_ptr<ICreator>;
    using TRegisteredCreators =
        std::map<std::string, TCreatorPtr>;
    TRegisteredCreators RegisteredCreators;
// ...
}
```

Фабрика

```
class TFactory::TImpl {
// ...
public:
    template <class TCurrentObject>
    class TCreator : public ICreator{
        std::unique_ptr<IOBJECT> Create() const override{
            return std::make_unique<TCurrentObject>();
        }
    };
// ...
}
```

Фабрика

```
class TFactory::TImpl {
// ...
public:
    TImpl() { RegisterAll(); }

    template <typename T>
    void RegisterCreator(const std::string& name) {
        RegisteredCreators[name] = std::make_shared<TCreator<T>>();
    }

    void RegisterAll() {
        RegisterCreator<TCustomObject>("custom object");
        RegisterCreator<TSuperObject>("super object");
    }

    std::unique_ptr<IOBJECT> CreateObject(const std::string& n) const {
        auto creator = RegisteredCreators.find(n);
        if (creator == RegisteredCreators.end()) {
            return nullptr;
        }
        return creator->second->Create();
    }
}
```

Фабрика

```
class TFactory::TImpl {
// ...
public:
    std::vector<std::string> GetAvailableObjects () const {
        std::vector<std::string> result;
        for (const auto& creatorPair : RegisteredCreators) {
            result.push_back(creatorPair.first);
        }
        return result;
    }
};

std::unique_ptr<IOBJECT> TFactory::CreateObject(const std::string& n)
{
    return Impl->CreateObject(n);
}

TFactory::TFactory() : Impl(std::make_unique<TFactory::TImpl>()) {}
TFactory::~TFactory(){}

std::vector<std::string> TFactory::GetAvailableObjects() const {
    return Impl->GetAvailableObjects();
}
```

Фабрика

```
#include "factory.h"

int main() {
    TFactory factory;
    auto objects = factory.GetAvailableObjects();
    for (const auto& obj : objects) {
        std::cout << obj << std::endl;
    }

    for (const auto& objName : {"super object", "custom object"}) {
        factory.CreateObject(objName)->Do();
    }
    return 0;
}
```

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 11
26.11.2021

offtop: аллокаторы

- ▶ По умолчанию все запросы на выделение памяти проходят через `::operator new()`, а на освобождение – через `::operator delete`.
- ▶ Для конкретных классов переопределять эти операторы может быть выгодно.
- ▶ Аллокатор – шаблонный класс, который управляет памятью. Делает три вещи:
 - ▶ получает память от диспетчера памяти,
 - ▶ возвращает память диспетчеру памяти,
 - ▶ конструирует копированием сам себя из связанного аллокатора.
- ▶ `std::allocator` – обёртка над `::operator new`.

Аллокаторы

- ▶ Классы стандартной библиотеки `container<T>` обычно имеют внутренний тип `container_item<T>`.
- ▶ Например, для списка это такой `TNode` со значением типа `T` и двумя указателями на `TNode`.
- ▶ Если для них писать свой аллокатор, то хочется писать аллокатор для `T` а не для `container_item<T>`.
- ▶ Часто при переопределении диспетчера памяти для какого-нибудь вектора не требуется это делать для всех векторов в программе.

Аллокаторы

- ▶ Аллокаторы *без состояния* – это аллокаторы, которые не имеют нестатических членов-данных. `std::allocator` без состояния.
- ▶ Два экземпляра такого класса неразличимы – память, выделенная одним аллокатором, может быть освобождена другим.
- ▶ Аллокаторы без состояния выделяют память от одного и того же диспетчера памяти. Это глобальный ресурс.
- ▶ Аллокатор с *внутренним состоянием* более сложно создавать. Однако, проще создавать несколько кусков памяти для различных целей. Если необходимо, чтобы все запросы не обязательно проходили через единый глобальный диспетчер памяти.

Аллокатор в C++11

```
template <class T>
struct custom_allocator {
    using value_type = T;

    custom_allocator() = default;

    template <class U>
    custom_allocator (const custom_allocator<U>&) = default;

    T* allocate (size_t n, void const* = 0) {
        return reinterpret_cast<T*>(
            ::operator new(n*sizeof(T))
        );
    }

    void deallocate (T* p, size_t n) {
        ::operator delete(p);
    }
};
```

Аллокатор в C++11

Обязательно определяются:

- ▶ Конструктор
- ▶ Конструктор копирования
- ▶ `value_type` — тип, для которого работает аллокатор
- ▶ `allocate` — выделение памяти под n объектов, но не конструирование
- ▶ `deallocate` — освобождение памяти

Аллокатор в C++11

До C++11 обязательно было необходимо определять `rebind`:

```
class Allocator{  
    ...  
  
    template<class Other>  
        struct rebind{  
            typedef Allocator<Other> other;  
        };  
    ...  
}; //Allocator
```

Начиная с C++11 работает `std::allocator_traits`. Внутри есть алиас `rebind_alloc`, определяемый для аллокатора `Alloc` как:

```
Alloc::rebind::other if present,  
otherwise Alloc<T, Args> if this Alloc is Alloc<U, Args>
```

Аллокатор

Обычно запросы памяти для одного класса требуют одинаковое число байтов.

```
extern  
fixed_block_memory_manager<fixed_arena_controller>  
list_memory_manager;  
  
template <class T>  
class StatelessListAllocator {  
public:  
    ...  
    T* allocate (size_t n) {  
        return reinterpret_cast<T*>(list_memory_manager.allocate(n * sizeof(T)));  
    }  
  
    void deallocate (T* p) {  
        list_memory_manager.deallocate(p);  
    }  
};
```

Visitor: предпосылки

Зададимся простой иерархией...

```
class TAnimal {  
public:  
    virtual ~TAnimal() {}  
    virtual void Talk() const = 0;  
    virtual void Move() const = 0;  
};
```

Visitor: предпосылки

```
using TAnimalPtr = std::shared_ptr<TAnimal>;\n\n\nclass TCat : public TAnimal {\npublic:\n    virtual void Talk() const override{\n        std::cout << "meow" << std::endl;\n    }\n    virtual void Move() const override{\n        std::cout << "cat jumps" << std::endl;\n    }\n};\n\n\nclass TDog : public TAnimal {\npublic:\n    virtual void Talk() const override{\n        std::cout << "woof" << std::endl;\n    }\n    virtual void Move() const override{\n        std::cout << "dog moves" << std::endl;\n    }\n};
```

Visitor: предпосылки

Простая фабрика по созданию объектов:

```
TAnimalPtr CreateAnimal() {
    std::string subj;
    std::cin >> subj;
    if (subj == "cat") {
        return static_cast<TAnimalPtr>(new TCat);
    } else if (subj == "dog") {
        return static_cast<TAnimalPtr>(new TDog);
    }
    return nullptr;
}
```

Обычный динамический полиморфизм:

```
int main() {
    TAnimalPtr animal = CreateAnimal();
    animal->Talk();
    animal->Move();
    return 0;
}
```

Visitor: операции

Вынесем операции:

```
class TOperation {
public:
    virtual ~TOperation() {}
    virtual void Apply(const TAnimal &animal) const = 0;
};

using TOperationPtr = std::shared_ptr<TOperation>;

class TTalkOperation : public TOperation {
public:
    virtual void Apddy(const TAnimal &animal) const override {
    }
};

class TMoveOperation : public TOperation {
public:
    virtual void Apply(const TAnimal &animal) const override{
    }
};
```

Visitor: операции

Фабрика операций:

```
TOperationPtr CreateOperation() {
    std::string op;
    std::cin >> op;
    if (op == "talk") {
        return static_cast<TOperationPtr>(new TTalkOperation);
    } else if (op == "move") {
        return static_cast<TOperationPtr>(new TMoveOperation);
    }
}
```

А теперь хочется делать так:

```
TAnimalPtr animal = CreateAnimal();
TOperationPtr operation = CreateOperation();
// (animal, operation) -> Apply(); ???
```

Visitor: двойная диспетчеризация

Можно сделать так:

```
class TTalkOperation : public TOperation {
public:
    virtual void Apply(const TAnimal &animal) const {
        if (const TCat* cat = dynamic_cast<const TCat*>(&animal)) {
            std::cout << "meow" << std::endl;
        } else if (const TDog* dog = dynamic_cast<const TDog*>(&animal))
            std::cout << "woof" << std::endl;
    }
};

};
```

Но:

- ▶ много дублирования,
- ▶ важен порядок условий.

Visitor

- ▶ В основной иерархии только виртуальный метод Accept ;
Переопределяется в наследниках

```
TCat::Accept(op) { op.Visit(*this); }
```

- ▶ В Operation(Visitor) определяется набор методов Visit ,
они все виртуальные и их можно переопределять
(Visit(Animal), Visit(Cat), Visit(Dog)).

Visitor

```
class TOperation {
public:
    virtual ~TOperation() {}
    virtual void Visit(const TAnimal &animal) const = 0;
    virtual void Visit(const TCat &cat) const;
    virtual void Visit(const TDog &dog) const;
};

void TOperation::Visit(const TCat &cat) const {
    Visit(static_cast<const TAnimal&>(cat));
}

void TOperation::Visit(const TDog &dog) const {
    Visit(static_cast<const TAnimal&>(dog));
}
```

Visitor

```
class TTalkOperation : public TOperation {
public:
    virtual void Visit(const TAnimal &animal) const override {
        std::cout << "unknown operation" << std::endl;
    }
    virtual void Visit(const TCat &cat) const override{
        std::cout << "meow" << std::endl;
    }
    virtual void Visit(const TDog &dog) const override{
        std::cout << "woof" << std::endl;
    }
};

class TMoveOperation : public TOperation {
public:
    virtual void Visit(const TAnimal &animal) const override {
        std::cout << "unknown operation" << std::endl;
    }
    // ...
};
```

Visitor

```
class TCat : public TAnimal,
    public std::enable_shared_from_this<TCat> {
public:
    virtual void Accept(const TOperation& operation) const override{
        std::shared_ptr<TCat> p{shared_from_this()};
        operation.Visit(p);
    }
};

class TDog : public TAnimal,
    public std::enable_shared_from_this<TDog> {
public:
    virtual void Accept(const TOperation& operation) const override{
        std::shared_ptr<TDog> p{shared_from_this()};
        operation.Visit(p);
    }
};
```

Visitor

Использование:

```
int main() {
    TAnimalPtr animal = CreateAnimal();
    TOperationPtr operation = CreateOperation();
    animal->Accept(*operation);
    return 0;
}
```

Двойная диспетчеризация: при вызове `animal->Accept` находится правильный класс `TAnimal` (механизм виртуальных функций), а затем при вызове `operation->visit(*this)` управление передаетсяциальному `visitor'у`.

Visitor

Дублирование в Accept наследниках можно устраниТЬ:

```
template <typename T>
class TFinalAnimal : public T {
public:
    virtual void Accept(const T0peration& operation) const {
        operation.Visit(*this);
    }
};
```

Идиома Type Erasure

Есть класс, сохраняющий объекты произвольного типа:

```
template <typename T>
class TValue {
    T v;
public:
    T Get() const {};
    template <typename U>
    void Set(U&& val) {v = std::forward<U>(val);}
};
```

И мы хотим «подписаться» на изменения Set:

```
TValue<int> v;
v.DoOnChange(
    [](int newVal) {std::cout << "set " << newVal << std::endl;});
v.Set(1);
```

Идиома Type Erasure

Абстрактный интерфейс вызова:

```
template <typename T>
class IFunctor {
public:
    virtual ~IFunctor() = default;
    virtual void Call(const T&t) = 0;
};
```

Класс для вызова:

```
template <typename T, typename F>
class TFunctor: public IFunctor<T> {
private:
    std::decay_t<F> f;
public:
    TFunctor(F f) : f(std::move(f)){}
    void Call(const T& t) override { f(t); }
};
```

Идиома TypeErasur

Создание объекта, который будет вызывать нужную функцию:

```
template <typename T, typename F>
std::shared_ptr<IFunctor<T>> CreateFunctor(F&& f) {
    return std::make_shared<TFunctor<T, F>> (std::forward<F>(f));
}
```

Идиома TypeErasur

```
template <typename T>
class TValue {
    T v;
    std::shared_ptr<IFunctor<T>> invPtr;
public:
    T Get() const {};
    template <typename U>
    void Set(U&& val) {
        if (val != v) {
            v = std::forward<U>(val);
            invPtr->Call(v);
        }
    }
    template <typename F>
    void DoOnChange(F&& f) {
        invPtr = CreateFunctor<T>(std::forward<F>(f));
    }
};
```

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 12
03.12.2021

Идиома TypeErasur

Пример: `shared_ptr`. Какой тип стирается?

```
template <typename T>
class TSharedPtr {
    struct TDeleteBase {
        virtual void apply(void*) = 0;
        virtual ~TDeleteBase() {}
    };

    template <typename D>
    struct Deleter: public TDeleteBase {
        Deleter(D d) : deleter(d) {}
        virtual void apply(void *p) {deleter(static_cast<T*>(p));}
        D deleter;
    };
    ...
}
```

Идиома TypeErasure

```
template <typename T>
class TSharedPtr {
    ...
public:
    template <typename D>
    TSharedPtr(T* ptr, D dlt): p(ptr), d(new Deleteer<D>(dlt)) {}

    ~TSharedPtr() {d->apply(p); delete d;}

    T* operator->() {return p;}
    const T* operator->() const {return p;}
private:
    T* p;
    TDeleteBase* d;
};

};
```

Проблема перегрузки и универсальных ссылок

Параметр-универсальная ссылка обычно обеспечивает точное соответствие для всего, что бы ни было передано:

```
using TStringSet = std::set<std::string>;
template <typename T>
void Do(TStringSet& strings, T&& str) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}
void Do(TStringSet& strings, int x) {
    std::cout << x << std::endl;
    strings.emplace(std::to_string(x));
}
```

Ломается код:

```
short x = 2;
Do(strings, x);
```

Проблема перегрузки и универсальных ссылок

Добавим новую функцию, которая вызывает две другие:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
    DoImpl(
        strings,
        std::forward<T>(x),
        std::is_integral<T>()
    )
}
```

Какая проблема?

`std::is_integral<int&>` имеет ложное значение.

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
    DoImpl(
        strings,
        std::forward<T>(x),
        std::is_integral<typename std::remove_reference<T>::type>()
    )
}
```

Проблема перегрузки и универсальных ссылок

Меньше символов с C++14:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
    DoImpl(
        strings,
        std::forward<T>(x),
        std::is_integral<std::remove_reference_t<T>>()
    )
}
```

Диспетчеризация дескрипторов: вызов перегруженных функций «диспетчеризует» передачу работы правильной функции путем создания нужного объекта дескриптора.

Проблема перегрузки и универсальных ссылок

Две перегрузки для DoImpl:

```
template <typename T>
void DoImpl(TStringSet& strings, T&& str, std::false_type /*f*/) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}

void DoImpl(TStringSet& strings, int x, std::true_type /*t*/) {
    std::cout << x << std::endl;
    strings.emplace(std::to_string(x));
}
```

Диспетчеризация дескрипторов

- ▶ Решаем проблемы перегрузки и оставляем универсальные ссылки,
- ▶ неперегружаемая функция диспетчеризации `Do` принимает параметр, являющийся универсальной ссылкой,
- ▶ перегружаемая `impl`-функция имеет параметр дескриптора, который спроектирован так, что не существует более одной перегрузки,
- ▶ вызов нужной функции определяется дескриптором.

Вопрос

Что напечатает программа?

```
void f(unsigned i) {
    std::cout << "f(int)" << std::endl;
}

template <typename T>
void f(const T& i) {
    std::cout << "template f" << std::endl;
}

int main() {
    f(3);
}

template f
```

SFINAE: Substitution Failure Is Not An Error

А если так?

```
unsigned f(unsigned i) {
    std::cout << "f(int)" << std::endl;
    return i + 1;
}

template <typename T>
typename T::value_type f(const T& i) {
    std::cout << "template f" << std::endl;
    return i + 1;
}

int main() {
    f(3);
}
f(int)
```

Подстановка int::value_type f(const int i) невалидна.

std::enable_if

std::enable_if: если передано true, то в структуре присутствует тип type (второй параметр), если передано false, то никакого type нет.

```
template<bool B, class T = void >
struct enable_if;
```

C++14 добавляет алиас:

```
template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

std::enable_if: пример

Использование в возвращаемом значении:

```
template <typename T>
std::enable_if_t<std::is_floating_point<T>::value, T> Do(const T& x) {
    std::cout << "float type" << std::endl;
    return x;
}

template <typename T>
std::enable_if_t<std::is_integral<T>::value, T> Do(const T& x) {
    std::cout << "integral type" << std::endl;
    return x;
}
```

`std::enable_if`: пример

В классах:

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

```
template<class T>
class A<T, std::enable_if_t<std::is_integral<T>::value>> {
};
```

- ▶ `std::enable_if_t<std::is_integral<int>::value>` → void
- ▶ `std::enable_if_t<std::is_integral<float>::value>` →
ошибка компиляции

Перегрузка и универсальные ссылки – 2

```
#include <string>

class A {
private:
    std::string text;
public:
    template <typename T>
    explicit A(T&& str) : text(std::forward<T>(str)) {}
    explicit A(int x) : text(std::to_string(x)) {}
};

int main() {
    A x("123");
    auto copyX(x); // error!
}
```

Перегрузка и универсальные ссылки – 2

Хочется написать так в шаблонный конструктор:

```
template <typename T, typename = std::enable_if_t<CONDITION>>
explicit A(T&& str) : text(std::forward<T>(str)) {}
```

CONDITION — тип Т не является классом А.



```
!std::is_same<A, T>::value
```

Тип Т может быть выведен как lvalue-ссылка A&, а это не А.



```
!std::is_same<A, std::decay_t<T>>::value
```

Перегрузка и универсальные ссылки – 2

Всё ок, но есть проблема:

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

Воспользуемся этим:

```
std::is_base_of<T1,T2>::value; // истинно, если  
                                // T2 - производный от T1  
  
std::is_base_of<int,int>::value; // false  
std::is_base_of<A,A>::value;   // true
```

Перегрузка и универсальные ссылки – 2

```
template <
    typename T,
    typename = std::enable_if_t<
        !std::is_base_of<A, std::decay_t<T>>::value
    >
>
explicit A(T&& str) : text(std::forward<T>(str)) {}
```

Но есть еще вторая перегрузка, которую нам нужно вызывать для целочисленных типов:

```
explicit A(int x) : text(std::to_string(x)) {}
```

Перегрузка и универсальные ссылки – 2

Отключаем шаблонный конструктор для обработки целочисленных аргументов:

```
class A {
private:
    std::string text;
public:
    template <
        typename T,
        typename = std::enable_if_t<
            !std::is_base_of<A, std::decay_t<T>>::value
            &&
            !std::is_integral<std::remove_reference_t<T>>::value
        >
    explicit A(T&& str) : text(std::forward<T>(str)) {}
    explicit A(int x) : text(std::to_string(x)) {}
};
```

Замечание

`is_arithmetic<T>`: если T является арифметическим типом (целочисленным или в формате с плавающей запятой), то имеется константа-член `value`, которая будет равна `true`. Для всех остальных типов `value` будет равна `false`.

```
std::is_arithmetic<int*>::value;           // false
std::is_arithmetic<int const>::value;        // true
std::is_arithmetic<int&>::value;            // false
```

Есть ли в классе метод?

```
template<class...> using dummy = void;

template <class T, typename = void>
struct does_have_super_func : public std::false_type {};

template <class T>
struct does_have_super_func<
    T,
    dummy<decltype(std::declval<T>().super_func())>
> : public std::true_type {};

struct A { int super_func(); };
struct B { int func(); };

int main() {
    std::cout << does_have_super_func<A>::value << std::endl;
    std::cout << does_have_super_func<B>::value << std::endl;
    std::cout << does_have_super_func<int>::value << std::endl;
}
```

C++17: constexpr if вместо SFINAE

```
template<class TIt>
void Sort(TIt first, TIt last) {
    using T = std::iterator_traits<TIt>::value_type;
    if constexpr (std::is_integral_v<T>) {
        RadixSort(first, last);
    } else {
        QuickSort(first, last);
    }
}
```

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)

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 func();
        } 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);
    }
}
```

std::exception_ptr (C++11)

- ▶ Можно сравнивать с nullptr_t.
- ▶ Два объекта std::exception_ptr равны только если они оба null или оба указывают на один и тот же объект исключения.
- ▶ Можно сконвертировать только в bool.
- ▶ Тип владения: shared.

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 13
10.12.2021

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;
}
```

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).

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 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?

Вопрос

В чем разница между вызовами для вектора:

- ▶ operator []
- ▶ at()

для несуществующего индекса?

Вопрос

Что произойдет в следующих случаях?

- ▶ Разыменование нулевого указателя
- ▶ Повторный вызов `delete`
- ▶ Переполнение знакового целого

Всё, что угодно. А почему так?

А переполнение беззнакового целого?

undefined behavior

```
void f(std::vector<int>& v) {
    v.reserve(v.size() * 2);
    for (const auto& x: v) {
        v.emplace_back(x + 5);
    }
}
```

UBsan

`-fsanitize=undefined` добавляет специфичные для UB проверки в run-time.

```
runtime error: store to null pointer of type 'int'
```

Неопределенное поведение

```
// A:  
int avg(int a, int b) {  
    return (a + b) / 2;  
}
```

```
// B:  
int neg(int a) {  
    return -a;  
}
```

```
// C:  
bool isEven(int a) {  
    return a % 2 == 0;  
}
```

```
// D:  
int getFirst(vector<int>& v) {  
    return *v.begin();  
}
```

Неопределенное поведение

JF Bastien
@jfbastien

Probably the best Undefined Behavior, ever.

`is_computer_on()`
Returns 1 if the computer is on. If the computer isn't on, the value returned by this function is undefined.

Slava Pestov @slava_pestov · Nov 14
BeOS had all sorts of useful APIs. We need to go back

`is_computer_on()`
`int32 is_computer_on(void)`
Returns 1 if the computer is on. If the computer isn't on, the value returned by this function is undefined.

Readability, Correctness, Efficiency

- ▶ Хорошо ли написан код?
- ▶ Корректно ли написан код?
- ▶ Эффективно ли работает код?
- ▶ Тестирование
- ▶ Формальная верификация
(автоматическая/полу-автоматическая)
- ▶ ...

Тестирование

- ▶ Цель тестирования кода?
- ▶ Как искать баги в коде и упростить себе жизнь?
 - ▶ Хороший дизайн кода и code review
 - ▶ unit-тестирование
 - ▶ Интеграционное тестирование
 - ▶ UI-тесты
 - ▶ Мониторинги
 - ▶ Acceptance тестирование

Тестирование

Unit-тесты (модульное тестирование):

- ▶ Быстрые
- ▶ Полные для своего компонента

Интеграционные тесты:

- ▶ Медленные
- ▶ Проверить всё невозможно
- ▶ Могут чаще флапать

Unit-тестирование

Задача: добавляем private-метод в класс с 100500 методами и полями. Нужно протестировать этот метод.

Behavior and State Based Testing

- ▶ Тесты, проверяющие что метод или функция отработали корректно, проверяют состояния объекта после вызова.
- ▶ Тесты на корректность самого взаимодействия объектов с другими объектами (а не результата).

gtest/gmock

Google C++ Testing Framework.

- ▶ Открытая библиотека для unit-тестирования.
- ▶ Ключевое понятие — assert
- ▶ Результат выполнения:
 - ▶ success
 - ▶ nonfatal failure
 - ▶ fatal failure
- ▶ Тест — набор assert'ов
- ▶ Набор тестов — тестовая программа или testing suite

gtest

```
ASSERT_TRUE(condition);
ASSERT_FALSE(condition);
ASSERT_EQ(val1, val2);
ASSERT_NE(val1, val2);
ASSERT_LT(val1, val2);
ASSERT_LE(val1, val2);
ASSERT_GT(val1, val2);
ASSERT_GE(val1, val2);
ASSERT_FLOAT_EQ(val1, val2);
ASSERT_DOUBLE_EQ(val1, val2);
ASSERT_NEAR(val1, val2, abs_error);
```

```
// strings
```

```
ASSERT_STREQ(str1, str2);
ASSERT_STRNE(str1, str2);
ASSERT_STRCASEEQ(str1, str2);
ASSERT_STRCASENE(str1, str2);
```

gtest

```
// exceptions
ASSERT_THROW(statement, exception_type);
ASSERT_ANY_THROW(statement);
ASSERT_NO_THROW(statement);
```

Можно добавлять message:

```
ASSERT_EQ(1, 2) << "i dont know why, but 1 != 2";
```

- ▶ TEST — макрос для определения теста
- ▶ RUN_ALL_TESTS() — запуск всех тестов

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 14
17.12.2021

Behavior and State Based Testing

- ▶ Тесты, проверяющие что метод или функция отработали корректно, проверяют состояния объекта после вызова.
- ▶ Тесты на корректность самого взаимодействия объектов с другими объектами (а не результата).

gtest/gmock

Google C++ Testing Framework.

- ▶ Открытая библиотека для unit-тестирования.
- ▶ Ключевое понятие — assert
- ▶ Результат выполнения:
 - ▶ success
 - ▶ nonfatal failure
 - ▶ fatal failure
- ▶ Тест — набор assert'ов
- ▶ Набор тестов — тестовая программа или testing suite

gtest

```
ASSERT_TRUE(condition);
ASSERT_FALSE(condition);
ASSERT_EQ(val1, val2);
ASSERT_NE(val1, val2);
ASSERT_LT(val1, val2);
ASSERT_LE(val1, val2);
ASSERT_GT(val1, val2);
ASSERT_GE(val1, val2);
ASSERT_FLOAT_EQ(val1, val2);
ASSERT_DOUBLE_EQ(val1, val2);
ASSERT_NEAR(val1, val2, abs_error);
```

```
// strings
```

```
ASSERT_STREQ(str1, str2);
ASSERT_STRNE(str1, str2);
ASSERT_STRCASEEQ(str1, str2);
ASSERT_STRCASENE(str1, str2);
```

gtest

```
// exceptions
ASSERT_THROW(statement, exception_type);
ASSERT_ANY_THROW(statement);
ASSERT_NO_THROW(statement);
```

Можно добавлять message:

```
ASSERT_EQ(1, 2) << "i dont know why, but 1 != 2";
```

- ▶ TEST — макрос для определения теста
- ▶ RUN_ALL_TESTS() — запуск всех тестов

Тестирование и зависимости

- ▶ Зависимости классов
- ▶ Как обеспечить изоляцию?

Основные подходы:

- ▶ **Dummy**. Передаются в тестируемые классы/методы/функции в виде параметра без поведения, внутри как правило с ним ничего не происходит
- ▶ **Stub**. Подменяется внешняя зависимость, игнорируются все данные, входящие в stub из тестируемого объекта.
- ▶ **Fake**. Замена легковесной реализацией.
- ▶ **Mock**. Объекты, которые имитируют поведение реальных. Полезно, когда настоящие объекты непрактично/невозможно вставлять в юнит-тест, но поведение нужно сохранить.

EXPECT_CALL: пример

```
EXPECT_CALL(db, func("string", _))
    .WillOnce(DoAll(SetArgReferee<1>("abc"), Return(false)));
```

- ▶ Метод func должен быть вызван 1 раз
- ▶ Ему будет передан аргумент "string"
- ▶ Что передадут в качестве второго аргумента — не важно
- ▶ Метод сделает второй аргумент равным "abc"
- ▶ Метод вернет false

Тестирование

1. Не стоит использовать EXPECT_TRUE
2. Добавляйте больше контекста
3. Код тестов — это тоже код

Тестирование

Тест упал, что делать дальше?

Основные подходы:

1. tracing и logging

```
cerr << "loading data: started" << endl;
```

2. strace и похожие утилиты
3. gdb и похожие отладчики
4. asan и другие санитайзеры

Debug

Пусть мы хотим залогировать все создания объекта класса A.

```
class A {  
    A() { std::cerr << "lol" << std::endl; }  
};
```

Но неудобно — есть класс B.

```
class B {  
    B() { std::cerr << "kek" << std::endl; }  
};
```

Информативность сообщений...

Debug

Пусть мы хотим залогировать все создания объекта класса A.

```
class A {  
    A() { std::cerr << "A::A()" << std::endl; }  
};
```

Класс B.

```
class B {  
    B() { std::cerr << "B::B()" << std::endl; }  
};
```

Некрасиво.

Debug: еще варианты

```
std::cerr << __FILE__ << " " << __LINE__ << std::endl;
```

Макросы, которые в момент компиляции раскрываются в имя файла и номер строчки.

strace

ОС — средство взаимодействия программы с внешним миром
(файлы, соединения, ...)

strace is a diagnostic, debugging and instructional userspace utility for Linux. It is used to monitor and tamper with interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state.

`strace ./you_binary`

Получаем все системные вызовы, предоставленные операционной системой

GDB

- ▶ **run** – start the debugged program
- ▶ **list** – list specified function or line
- ▶ **break** – set breakpoint
- ▶ **catch** – set catchpoint (exception breakpoint)
- ▶ **info** – show information about the debugged program
- ▶ **step** – step program, steps into functions
- ▶ **next** – step program, steps over function calls
- ▶ **stepli, nexti** – step by instructions, not lines of code
- ▶ **print** – evaluate expression
- ▶ **examine** – display contents of memory address
- ▶ **continue** – continue running (after breakpoint)
- ▶ **kill** – stop execution of the program
- ▶ **backtrace** – print backtrace of stack frames
- ▶ **up, down, frame, select-frame** – select stack frame
- ▶ ...

Отладочные макроопределения

Можно включить специальный отладочный режим стандартной библиотеки C++, в котором контейнеры и алгоритмы делают всевозможные проверки.

-D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC

```
vector<int> v{5};  
v.reserve(2);  
std::cout << v[1] << std::endl;  
  
/usr/include/c++/4.8/debug/vector:346:error: attempt  
to subscript container with out-of-bounds index 1, but  
container only holds 1 elements.
```

Оптимизация программ

- ▶ оптимальные алгоритмы и структуры данных;
- ▶ отсутствие лишних копирований;
- ▶ эффективный параллелизм;
- ▶ компиляторы и оптимизаторы;
- ▶ ...

«*Premature optimization is the root of all evil*» (c)

Все любят «отмазываться» этой цитатой.

Оптимизация программ на C++

Особенности:

- ▶ Порядок инструкций в коде \neq порядок их исполнения;
- ▶ Потоки выполнения, средства синхронизации,
`std::atomic`.
- ▶ STL – универсальная библиотека, часто вручную
закодированный алгоритм работает быстрее.

Эвристическое правило 90/10, bottleneck

90% времени работы программы тратит на исполнение 10% кода.

В программе есть «горячие» точки, где наиболее всего уместна оптимизация.

Если оптимизацией пренебречь в большом проекте, то впоследствии сложно будет что либо ускорить относительно всего дизайна!

Закон Амдала

$$S_T = \frac{1}{(1 - P) + \frac{P}{S_P}}$$

- ▶ S_T — ускорение в целом (улучшение времени выполнения программы в целом в результате оптимизации)
- ▶ P — доля под оптимизацию (сколько времени от общего оптимизируем)
- ▶ S_P — ускорение в оптимизированной части (насколько ускоряем в этой части)

Profiler

- ▶ Программа, которая генерирует статистические данные о том, как и на что программа тратит свое время работы.
- ▶ Может выдать частоты выполнения каждой инструкции или функции и суммарное время выполнения каждой функции.
- ▶ Влияние времени работы самого профайлера на время выполнения работы всей программы не велико и не имеет важного значения.
- ▶ Наиболее важное: увидеть самые проблемные места в коде
- ▶ Профилирование отладочной версии программы дает почти такой же результат, но нагляднее, поскольку ничего не скрывает.

Профайлер не в состоянии посоветовать более эффективный алгоритм решения задачи.

Строки в C++

Класс `std::string`: тысячи возможностей делают его эффективную реализацию невозможной. Существует множество компиляторов, в которых реализация строк не соответствует стандарту.

- ▶ используют динамическое выделение памяти
- ▶ ведут себя как значения в выражениях
- ▶ требуют много копирований

Идиома COW

Copy On Write: идиома для объектов с дорогим копированием.

- ▶ Одна динамическая память может использоваться несколькими значениями.
- ▶ Любая операция, которая изменяет значение строки, сначала убеждается что существует только один указатель на эту память.
- ▶ Если нет — выделяем новую и копируем.
- ▶ C++11 запрещает cow-строки. Можно реализовать с использованием `shared_ptr`.

Строки в C++

- ▶ С-строки.
- ▶ `std::string_view`: содержит указатель, которым не владеет, на строковые данные и их длину. Подстрока и отсечение суффиксов/префиксов более эффективны, чем в `std::string`. Полезен, когда хочется избежать ненужных копирований.
- ▶ Можно сделать другие реализации, но:
 - ▶ они должны давать профит везде, не только в узком кейсе;
 - ▶ замена требует большой работы;
 - ▶ найти среди множества альтернатив весьма трудно.

Пример 1

Как оптимизировать код?

```
for (auto& str: cont) {  
    std::string s;  
    ParseInput(str, s);  
    Process(s);  
}
```

```
std::string s;  
for (auto& str: cont) {  
    s.clear();  
    ParseInput(str, s);  
    Process(s);  
}
```

Пример 2

Как оптимизировать код?

```
char s[] = ".....";
for (size_t i = 0; i < strlen(s); ++i) {
    if (s[i] == ' ') s[i] = '*';
}
```

A:

```
for (size_t i = 0, len = strlen(s); i < len; ++i) {
    if (s[i] == ' ') s[i] = '*';
}
```

B:

```
for (int i = (int)strlen(s) - 1; i >= 0; --i) {
    if (s[i] == ' ') s[i] = '*';
}
```

Пример 3

Как оптимизировать код?

```
for (size_t i = 0; i < v.size(); ++i) {
    double x = v[i].x, y = v[i].y;
    v[i].x = cos(theta) * x - sin(theta) * y;
    v[i].y = sin(theta) * x + cos(theta) * y;
}
```

Выносим инварианты:

```
cos_theta = cos(theta);
sin_theta = sin(theta);
for (size_t i = 0; i < v.size(); ++i) {
    double x = v[i].x, y = v[i].y;
    v[i].x = cos_theta * x - sin_theta * y;
    v[i].y = sin_theta * x + cos_theta * y;
}
```

Пример 4

Как оптимизировать код?

```
class A {  
    private:  
        int x, y;  
    public:  
        // ...  
        std::string get_name () const {  
            return "A";  
        }  
};
```

static! Статические функции не должны вычислять неявный
указатель this.

Пример 5

Как оптимизировать код?

$y = a * x * x * x + b * x * x + c * x + d;$

$y = (((a * x + b) * x) + c) * x + d;$

(схема Горнера)

Пример 6

Как оптимизировать код?

```
int f() {  
    // ...  
    seconds = 24 * days * 60 * 60;  
    // ...  
}
```

```
constexpr float SecondsInDay = 24 * 60 * 60;  
int f() {  
    // ...  
    seconds = days * SecondsInDay;  
    // ...  
}
```

Пример 7

Как оптимизировать код?

```
y = x * 9;
```

```
y = (x << 3) + x;
```

Оптимизация алгоритмов

- ▶ **Предвычисления.** Тут помогает `constexpr` и вычисления компилятором, подстановка шаблонных параметров. Компилятор сам оптимизирует.
- ▶ **Отложенные вычисления.** Как COW, когда копирование откладывается до тех пор, пока кто-то не захочет изменить данные.
- ▶ **Пакетирование.** Сбор элементов и их совместная обработка. Пример — буферизованный вывод.
- ▶ **Кеширование.** Повторное использование уже вычисленных результатов. Пример — не вычисляем длину строки каждый раз при вызове `size()`.
- ▶ **Специализация.** Не делаем те вычисления, которые в конкретном частном случае могут быть не нужны. Пример — `std::swap` использует семантику перемещения, если аргументы перемещаемы.

Оптимизация алгоритмов

- ▶ **Группировка.** Сокращение числа итераций повторов.
Пример — запрос больших данных от ОС, чтобы вызывать функций ядра меньшее количество раз.
- ▶ **Подсказки.** Пример — необязательный аргумент `hint` у `std::map`, — итератор, с которого стоит начать поиск места для вставки.
- ▶ **Оптимизация ожидаемого пути.** Если есть много `else if`, то первыми лучше сделать те, которые встречаются чаще.
- ▶ **Двойная проверка.** Для исключения некоторых случаев используется недорогая проверка, а затем при необходимости используется дорогостоящая. Например, сравнение `std::string` можно проводить, сравнивая сначала длины, а потом уже посимвольно.
- ▶ **Хеширование.** Превращение объекта в целочисленное значение (хеш). Если хеши разные, то и объекты разные.

Функции и инструкции

- ▶ $++i$ в общем случае быстрее, чем $i++$? Компиляторы справляются.
- ▶ Встраиваемые функции. Заменить вызов функции ее телом. Функции – расходы на удобство программирования.
- ▶ Дешевый вариант для вызова:

```
void f();
```

- ▶ Дорогой вариант для вызова: виртуальная функция производного класса, где базовый класс не имеет виртуальных функций и при этом наследование множественное.