Курс-интенсив

Программирование на C++

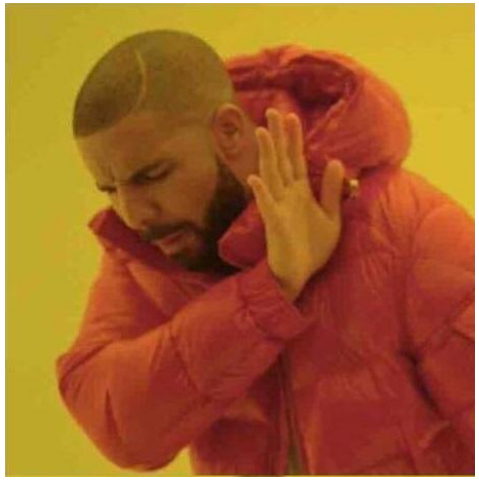
Шаблоны C++

academy.rubius.com
konstantin.dobrychev@rubius.com
Константин Добрычев

Полиморфные классы?

```
class IntegerList
{
public:
    IntegerList();
    ~IntegerList();

    void add(int value);
    void remove(size_t index);
    int get(size_t index) const;

private:
    size_t size_;
    int* array_;
};
```

```
class StringList
{
public:
    StringList();
    ~StringList();

    void add(std::string value);
    void remove(size_t index);
    std::string get(size_t index) const;

private:
    size_t size_;
    std::string* array_;
};
```



```
class DoubleList;
class BooleanList;
class DateTimeList;
...
class AnythingElseList;
```

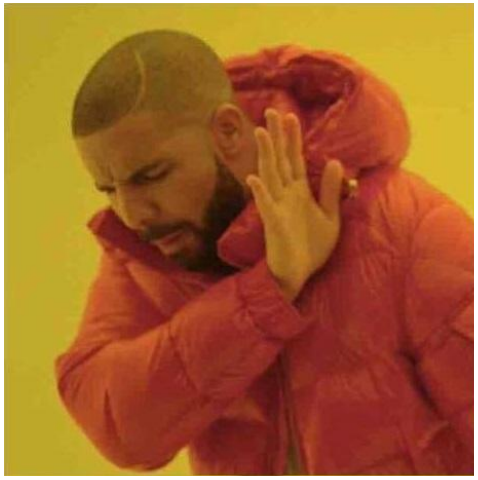
Полиморфные классы?

Один базовый тип:

```
Object
void*
std::any
...
```

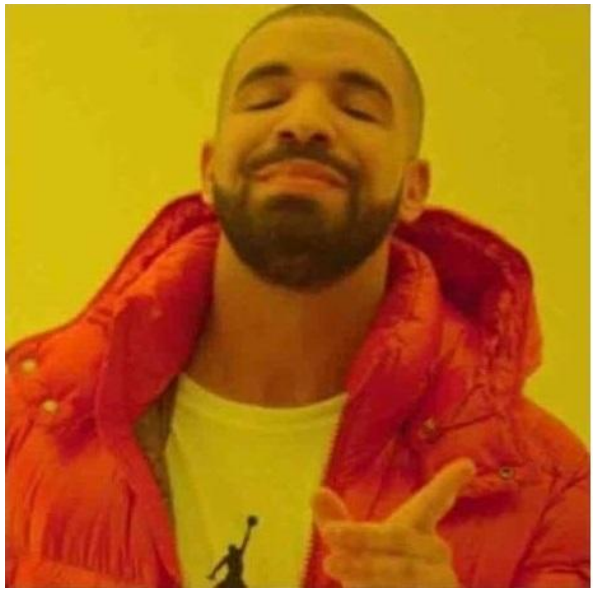
```
class List
{
public:
    List();
    ~List();

    void add(std::any value);
    void remove(size_t index);
    std::any get(size_t index) const;

private:
    size_t size_;
    std::any* array_;
};
```

- Нужно приводить типы
- Нужно следить за содержимым
- Лишний код/работа/сущности



Шаблоны

```
template<typename T>
class List
{
public:
    List();
    ~List();

    void add(T value);
    void remove(size_t index);
    T get(size_t index) const;

private:
    size_t size_;
    T* array_;
};
```



Нужно стирание типов?

https://www.boost.org/doc/libs/1_70_0/doc/html/boost_typeerasure.html

Немного терминов

Шаблон — это параметризованное описание семейства программных сущностей (функций/классов/методов/переменных).

Процесс создание сущности путём подстановки конкретных параметром шаблона называется **инстанцированием шаблона**.

Сущности, появляющиеся в результате инстанцирования, называются **специализациями шаблона**.

Шаблоны функций

```
int sum(int a, int b)
{
    return a + b;
}

template<typename T>
T sum(T a, T b)
{
    return a + b;
}

template<typename T1, typename T2, typename RT = double>
RT sum(T1 a, T2 b)
{
    return a + b;
}
```

```
sum(1, 2);
sum("Hello "s, "World"s);
sum<std::string>("Hello ", "World");
sum<double, long, int>(4.1, 3.6);
```

Шаблоны функций

```
template<typename T1, typename T2, typename RT = ???>
RT sum(T1 a, T2 b)
{
    return a + b;
}
```

```
RT = T1
RT = T2
RT = long double
RT = decltype(std::declval<T1>() + std::declval<T2>())
RT = std::common_type_t<T1, T2>
```

```
template<typename T1, typename T2>
auto sum(T1 a, T2 b)
{
    return a + b;
}
```

Шаблоны классов

```
template<typename T>
class List
{
public:
    List();
    List(std::initializer_list<T> ilist);
    ~List();

    void add(T value);
    void remove(size_t index);
    T get(size_t index) const;

private:
    size_t size_;
    T* array_;
};
```

```
List<std::string> list;
List list<int> = {1, 2, 3, 4};
List list = {1, 2, 3, 4};
```

Шаблоны классов

```
template<typename T>
void List<T>::remove(size_t index)
{
    T* dst = array_ + index;
    std::memmove(dst, dst + 1, (size_ - index));
    --size_;
}

template<typename T>
template<typename U>
List<T>& List<T>::extend(const List<U>& other) {
    for (size_t i = 0; i < other.size_; ++i) {
        add(other.get(i));
    }

    return *this;
}
```

Шаблоны классов

Полная специализация

```
template<>
class List<User>
{
    ...
    void add(const User& value);
    ...
};
```

Частичная специализация

```
template<typename T>
class List<T*>
{
    ...
    void remove(size_t index);
    ...
};
```

Параметры шаблонов

Типовые

```
template<typename T>
class List;
```

Нетиповые

```
template<typename T, size_t MaxSize>
class List;
```

Шаблонные

```
template<
  typename T,
  template<typename> class Container
>
class List
```

Ещё немного шаблонов

```
template<typename T>
constexpr T PI = 3.14;

auto a = PI<float>;
auto b = PI<double>;
```

```
template<typename T>
using InvokeResult = Acquired<ServiceFuture<T>>;
```

Универсальные ссылки

```
template<typename T>
void doSomething(T&& value);
```

```
int a = 42;
doSomething(a); // T -> int&

int& b = a;
doSomething(b); // T - > int&

int&& c = std::move(a);
doSomething(c); // T -> int&&
```

Прямая передача

```
template<typename U, typename V>
class Pair {
public:
    Pair(U&& first, V&& second)
        : first_(std::forward<U>(first))
        , second_(std::forward<V>(second))
    {}

private:
    U first_;
    V second_;
};
```

Вариативные шаблоны

```
template<typename... Ts>
auto sum(Ts... args);

template<typename T>
auto sum(T a) { return a; }

template<typename T, typename... Ts>
auto sum(T head, Ts... tail)
{
    return head + sum(tail...);
}

sum(1);
sum(2, 3);
sum(2.0, 3, 45.98f);
```

```
template<typename... Ts>
class Tuple;


Tuple<> t0;
Tuple<std::string> t1;
Tuple<float> t2;
Tuple<int, long, Point> t3;
```

Выражения свёртки

```
// унарная правая
template<typename... Ts>
auto sum(Ts... args)
{
    return (args + ...);
}
```

```
// унарная левая
template<typename... Ts>
auto sum(Ts... args)
{
    return (... + args);
}
```

```
// бинарная правая
template<typename... Ts>
auto sum(Ts... args)
{
    return (args + ... + 0);
}
```

```
// бинарная левая
template<typename... Ts>
auto sum(Ts... args)
{
    return (0 + ... + args);
}
```

Мета-программирование

Функция

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

factorial(5);
```

Мета-функция

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

template<>
struct Factorial<0> {
    static constexpr int value = 1;
};

Factorial<5>::value
```

Мета-программирование

```
template<typename T>
struct IsReference {
    static constexpr bool value = false;
};

template<typename T>
struct IsReference<T&> {
    static constexpr bool value = true;
};

template<typename T>
struct IsReference<T&&> {
    static constexpr bool value = true;
};
```

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

template<typename T>
struct RemoveReference<T&> {
    using type = T;
};

template<typename T>
struct RemoveReference<T&&> {
    using type = T;
};
```

```
std::cout << IsReference<const std::string&>::value << "\n";
std::cout << IsReference<int>::value << "\n";

RemoveReference<std::string&&>::type text = "Text";
std::cout << IsReference<decltype(text)>::value << "\n";
```

Свойства типов

https://en.cppreference.com/w/cpp/header/type_traits

```cpp
#include <iostream>
#include <type_traits>

struct foo
{
    void m() { std::cout << "Non-cv\n"; }
    void m() const { std::cout << "Const\n"; }
    void m() volatile { std::cout << "Volatile\n"; }
    void m() const volatile { std::cout << "Const-volatile\n"; }
};

int main()
{
    foo{}.m();
    std::add_const<foo>::type{}.m();
    std::add_volatile<foo>::type{}.m();
    std::add_cv<foo>::type{}.m();
}
```

Output:

```
Non-cv
Const
Volatile
Const-volatile
```

constexpr

```
constexpr int TIMEOUT = 10 * 1000;

constexpr int factorial(int n)
{
    int result = 1;

    for (int i = 1; i <= n; ++i) {
      result *= i;
    }

    return result;
}
```

```
class Point {
public:
    constexpr Point(int x = 0, int y = 0)
      : x_(x)
      , y_(y)
    {}

    constexpr int x() const { return x_; }
    constexpr int y() const { return y_; }

private:
    int x_;
    int y_;
};
```

```
constexpr Point point(-11, factorial(12));
```

SFINAE

SFINAE - Substitution Failure Is Not An Error

```
template<typename C, typename = typename C::iterator>
void processAll(const C& container) {
    for (auto value : container) {
        // do something ...
    }
}

template<typename C>
decltype(begin(std::declval<C>()), end(std::declval<C>()), void)
processAll(const C& container) {
    for (auto value : container) {
        // do something ...
    }
}
```

SFINAE

```
template<typename C>
std::enable_if_t<std::is_pointer_v<typename C::value_type>, void>
deleteAll(const C& container)
```

```
template<
    typename C,
    typename = std::enable_if_t<std::is_pointer_v<typename C::value_type>>
>
void deleteAll(const C& container)
```

```
#define REQUIRES(...) typename = std::enable_if_t<__VA_ARGS__>
```

```
template<typename C, REQUIRES(std::is_pointer_v<typename C::value_type>)>
void deleteAll(const C& container)
```

SFINAE

```
template<typename C>
std::enable_if_t<std::is_pointer<typename C::value_type>::value>
clearContainerImpl(C &container)
{
    deleteAll(container);
    container.clear();
}

template<typename C>
std::enable_if_t<!std::is_pointer<typename C::value_type>::value>
clearContainerImpl(C &container)
{
    container.clear();
}

template<typename C>
void clearContainer(C &container)
{
    clearContainerImpl(container);
}
```

constexpr if

```
template<typename C>
void clearContainer(C& container)
{
    if constexpr (std::is_pointer_v<typename C::value_type>) {
        deleteAll(container);
        container.clear();
    } else {
        container.clear();
    }
}
```

std::void_t

```
template<typename... Ts>
using void_t = void;
```

```
template<typename T, typename = void>
struct IsReversableSequence : std::false_type {};

template<typename T>
struct IsReversableSequence<
   T, std::void_t<
       decltype(std::declval<T>().begin()),
       decltype(std::declval<T>().end()),
       decltype(std::declval<T>().rbegin()),
       decltype(std::declval<T>().rend())
   >
> : std::true_type {};
```

```
static_assert(IsReversableSequence<std::list<int>>::value);
static_assert(!IsReversableSequence<std::forward_list<int>>::value);
```

CRTP

Curiously Recurring Template Pattern

```
template<typename T>
class Base
{
    // ...
};

class Derived : public Base<Derived>
{
    // ...
};
```

Концепты

https://en.cppreference.com/w/cpp/language/constraints

```
template<typename T>
concept bool Animal()
{
  return requires(T a) {
    { a.name() } -> std::string;
    { a.voice() } -> void
  };
}
```

```
template<typename T>
concept bool EqualityComparable()
{
  return requires(T a, T b) {
    { a == b } -> bool;
    { a != b } -> bool;
  };
}
```

```
template<typename T>
concept bool Array = std::is_array_v<T>;
```

```
template<typename T, typename U>
concept Derived = std::is_base_of<U, T>::value;
```

Концепты

```
template<typename T>
struct Point {
    T x;
    T y;
};

template<EqualityComparable T>
bool operator==(const Point<T>& lhs, const Point<T>& rhs)
{
    return (lhs.x == rhs.x) && (lhs.y == rhs.y);
}

template<typename T>
requires EqualityComparable<T>() && Swappable<T>()
void doSomething(const T& value) { … }
```

Куда копать дальше?

Курс-интенсив

Программирование на C++

Шаблоны C++

academy.rubius.com
konstantin.dobrychev@rubius.com
Константин Добрычев