Напредно генеричко програмирање 2

библиотека за својства типова, специјализација шаблона, шаблони променљивих и шаблони алијаса, смернице за закључивање типова, статички полиморфизам, CRTP

Својства типова

- Библиотека <type_traits> нам нуди низ (око 70) предиката који се тичу типова:

```
std::is_void<T>::value
std::is_pointer<T>::value
std::is_floating_point<T>::value
std::is_const<T>::value
std::is_abstract<T>::value
std::is_polymorphic<T>::value
std::is_copy_assignable<T>::value

std::is_convertible<From, To>::value
std::is_base<Base, Derived>::value
```

- Као и неколико (око 25) трансформација типова:

```
std::remove_cv<T>::type // ово је тип, исти као T, само без const
std::add_pointer<T>::type

add_pointer_t<T> // од Це++14
```

Својства типова

- Сада можемо, на пример, направити шаблон функције који ради само ако тип параметара може да се копира. Компајлер ће лепо пријавити грешку уколико је шаблон инстанциран са лошим параметрима.

```
template<typename T>
void foo(T x) {
  static_assert(std::is_copy_assignable<T>::value
    && std::is_copy_constructible<T>::value,
    “foo expect copy assignable and copy constructible classes”);
}
```

Специјализација шаблона класа

- Ако желимо, слично као код функција, да имамо класу (тип) која има свој општи облик (представљен шаблоном), али посебну дефиницију за неки специфичан стварни параметар шаблона, можемо то урадити овако:

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


template<>
class MyClass<bool> { // само за bool тип
  //...
};
```

- Ово називано **потпуном специјализацијом** шаблона класа, јер...

Специјализација шаблона класа

- ...постоји и **делимична (парцијална) специјализација**.

```
template<typename T>
class Foo;

template<typename T>
class Foo<T*> { // само за чисте показиваче
  // ...
};

template<typename T>
class Bar;

template<typename T, typename... Args>
class Bar<T(Args...)> { // само за функције
  // ...
};
```

Специјализација шаблона класа

- Како имплементирати ово?

```
is_void<T>::value
```

- Да вредност овога буде true ако T јесте void, а false у супротном.

Специјализација шаблона класа

- Како имплементирати ово?

```
is_void<T>::value
```

- Да вредност овога буде true ако T јесте void, а false у супротном.

```
struct is_void {
  static bool value; // !!!
};
```

Специјализација шаблона класа

- Како имплементирати ово?

```
is_void<T>::value
```

- Да вредност овога буде true ако T јесте void, а false у супротном.

```
struct is_void {
  static constexpr bool value;
};
```

Специјализација шаблона класа

- Како имплементирати ово?

```
is_void<T>::value
```

- Да вредност овога буде true ако T јесте void, а false у супротном.

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

Специјализација шаблона класа

- Како имплементирати ово?

```
is_void<T>::value
```

- Да вредност овога буде true ако T јесте void, а false у супротном.

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

template<>
struct is_void<void> {
  static constexpr bool value = true;
};
```

Специјализација шаблона класа

- Како имплементирати ово?

```
is_void<T>::value
```

- Нека буде true и за const void, volatile void и const volatile void.

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

template<>
struct is_void<void> { static constexpr bool value = true; };

template<>
struct is_void<const void> { static constexpr bool value = true; };

template<>
struct is_void<volatile void> { static constexpr bool value = true; };

template<>
struct is_void<const volatile void> { static constexpr bool value = true; };
```

Специјализација шаблона класа

- Како имплементирати ово?

```
is_void<T>::value
```

- Може се мало поједноставити:

```
struct false_type { static constexpr bool value = false; };

struct true_type { static constexpr bool value = true; };

template<typename T>
struct is_void : false_type;

template<>
struct is_void<void> : true_type;

template<>
struct is_void<const void> : true_type;

template<>
struct is_void<volatile void> : true_type;

template<>
struct is_void<const volatile void> : true_type;
```

Специјализација шаблона класа

- Како имплементирати ово?

`is_void<T>::value`

- А постоји и std::true_type и std::false_type (погледати како су они тачно дефинисани)

```
template<typename T>
struct is_void : std::false_type;

template<>
struct is_void<void> : std::true_type;

template<>
struct is_void<const void> : std::true_type;

template<>
struct is_void<volatile void> : std::true_type;

template<>
struct is_void<const volatile void> : std::true_type;
```

Специјализација шаблона променљивих

- Употреба је сада оваква:

```
if (is_void<T>::value) ...
```

- Али, сада постоје и шаблони променљиве, па можемо направити и ово:

```
template<typename T>
constexpr bool is_void_v = false;

template<>
constexpr bool is_void_v<void> = true;

template<>
constexpr bool is_void_v<const void> = true;

template<>
constexpr bool is_void_v<volatile void> = true;

template<>
constexpr bool is_void_v<const volatile void> = true;
```

Мада је ово уобичајена имплементација:

```
template<typename T>
constexpr bool is_void_v
  = is_void<T>::value;
```

- Па употреба може бити оваква:

```
if (is_void_v<T>) ...
```

Специјализација шаблона класа

- Још мало да поједноставимо, коришћењем парцијалне специјализације:

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

template<typename T> struct remove_volatile { using type = T; };
template<typename T> struct remove_volatile<volatile T> { using type = T; };

template<typename T>
struct remove_cv {
    using type = typename remove_volatile<typename remove_const<T>::type>::type;
};
```

- **typename** мора да претходи зависним именима.
- Сада is_void можемо да формулишемо овако:

```
template<typename T> struct is_void_helper : std::false_type;
template<> struct is_void_helper<void> : std::true_type;

template<typename T>
struct is_void : is_void_helper<typename remove_cv<T>::type>;
```

Шаблон алијаса и њихова специјализација

- Слично може бити дефинисано и is_pointer:

```
namespace detail {
    template<typename T> struct is_pointer_helper : std::false_type;
    template<typename T> struct is_pointer_helper<T*> : std::true_type;
}

template<typename T>
struct is_pointer : detail::is_pointer_helper<typename remove_cv<T>::type>;
```

- А могу да мало помогну и шаблони алијаса.

```
template<typename T>
using remove_cv_t = typename remove_cv<T>::type;

template<typename T>
struct is_pointer : detail::is_pointer_helper<remove_cv_t<T>>;
```

Мала дигресија

- Како можемо имплементирати **is_same**?

```
template<typename T1, typename T2>
struct is_same : std::false_type;

template<???>
struct is_same<???> : std::true_type;

template<typename T1, typename T2>
constexpr bool is_same_v = is_same<T1, T2>::value;
```

- Или is_function.

```
template<typename T>
struct is_function : std::false_type;

template<???>
struct is_function<???> : std::true_type;

template<typename T>
constexpr bool is_function_v = is_function<T>::value;
```

Мала дигресија

- Како можемо имплементирати **is_same**?

```
template<typename T1, typename T2>
struct is_same : std::false_type;

template<typename T>
struct is_same<T, T> : std::true_type;

template<typename T1, typename T2>
constexpr bool is_same_v = is_same<T1, T2>::value;
```

- Или is_function.

```
template<typename T>
struct is_function : std::false_type;

template<typename FT, typename... Args>
struct is_function<FT(Args...)> : std::true_type;

template<typename T>
constexpr bool is_function_v = is_function<T>::value;
```

Специјализација шаблона функција

- Сличан је синтакса као и за класе:

```
template<typename T>
void foo(T x) { ... }

template<>
void foo<bool>(bool x) { ... }
```

- Али, то јако ретко радимо.
- Боље је ово:

```
template<typename T>
void foo(T x) { ... }

void foo(bool x) { ... }
```

- Или овако нешто:

```
template<typename T>
void foo(T x) {
    if constexpr(std::is_same_v<bool, T>) { ...
    } else { ...
    }
}
```

Специјализација шаблона функција

- За шаблоне функција нема парцијалне специјализације:

```
template<typename T>
void foo(T x) { ... }

template<typename T>
void foo<T*>(T* x) { ... } // ово не може!
```

Специјализација шаблона функција

- За шаблоне функција нема парцијалне специјализације:

```
template<typename T>
void foo(T x) { ... }

template<typename T>
void foo<T*>(T* x) { ... } // ово не може!
```

- Али функтори су класе, и они могу бити парцијално специјализовани:

```
template<typename T>
struct Foo {
    void operator()(T x) { ... }
};

template<typename T>
struct Foo<T*> {
    void operator()(T* x) { ... }
};
```

- Али онда нема закључивања стварних параметара шаблона:

```
Foo<int>()(5);
Foo<int*>()(&t);
```

Специјализација шаблона функција

- За шаблоне функција нема парцијалне специјализације:

```
template<typename T>
void foo(T x) { ... }

template<typename T>
void foo<T*>(T* x) { ... } // ово не може!
```

- Али функтори су класе, и они могу бити парцијално специјализовани:

```
template<typename T>
struct Foo {
    void operator()(T x) { ... }
};

template<typename T>
struct Foo<T*> {
    void operator()(T* x) { ... }
};
```

- Али онда нема закључивања стварних параметара шаблона:

```
Foo<int>()(5);
Foo<int*>()(&t);
```

Осим ако то не замотамо у функцију:

```
template<typename T>
void foo(T x) { Foo<T>()(x); }
```

Закључивање шаблонских параметара код инстанцирања шаблона класа???

- Зашто механизам који се користи код шаблона функција не може да се примени код шаблона класа?
- Функција има фиктивне и стварне параметре, па се то може искористити, а класа нема.
- Али, класа има специјалну функцију – конструктор, зар се не би то могло искористити?
- Па, питање је у ком тренутку је то „има“. Класа прво мора да се направи на основу шаблона и тек онда постоје њени конструктори... а тада је већ касно за закључивање параметара.

```
template<typename T>
struct Tad {
    Tad(T x) : m_v(x) {}
    T m_v;
};
// у овом тренутку компајлер зна само за шаблон класе под именом Tad, и мора овако:
Tad<int> a{5};
// не може овако:
Tad a{5};
```

Закључивање шаблонских параметара код инстанцирања шаблона класа???

- Једно решење (које смо већ видели) јесте да додамо помоћну функцију која ће служити само за закључивање параметара.

```
template<typename T>
struct Tad {
    Tad(T x) : m_v(x) {}
    T m_v;
};

template<typename T>
auto makeTad(T x) {
    return Tad<T>{x};
};

auto a = makeTad(5);
```

- И то је чест приступ, али има неке недостатке.

Смернице за закључивање

- У Це++17 стандарду препозната је посебност ове ситуације и понуђено је решење.
- Сада постоје „смернице за закључивање“ (енгл. deduction guides). Смернице су једноставније и јасније од помоћних функција, директно изражавају намеру, а могу се користити само у овом контексту.

```
template<typename T>
struct Tad {
    Tad(T x) : m_v(x) {}
    T m_v;
};

template<typename T>
auto makeTad(T x) {
    return Tad<T>{x};
};

auto a = makeTad(5);
```

```
template<typename T>
auto makeTad(T x) -> Tad<T> {
    return Tad<T>{x};
};
```

```
template<typename T>
Tad(T x) -> Tad<T>;

Tad a{5};
auto a = Tad{5};
```

Смернице за закључивање

- Ситуације је још боља: за овако једноставне случајеве смернице имплицитно постоје и компајлер их прати.
- Дакле, могуће је ово без икаквог писања смерница од стране програмера.

```
template<typename T>
struct Tad {
    Tad(T x) : m_v(x) {}
    T m_v;
};

Tad a{5};
auto a = Tad{5};
```

Смернице за закључивање

- Ситуације је још боља: за овако једноставне случајеве смернице имплицитно постоје и компајлер их прати.
- Дакле, могуће је ово без икаквог писања смерница од стране програмера.

```
template<typename T>
struct Tad {
    Tad(T x) : m_v(x) {}
    Tad(T* x) : m_v(*x) {}
    T m_v;
};


Tad a{6};
Tad b{&a};
```

- Као да постоје ове смернице:

```
template<typename T>
Tad(T) -> Tad<T>;

template<typename T>
Tad(T*) -> Tad<T>;
```

Смернице за закључивање

- Али некада је неопходно писати смернице.
- Два типична примера су ови:

```
template <typename T>
struct vector {
    //...
    template <typename Iter> vector(Iter start, Iter end);

    T* ptr;
};

template<typename Iter>
vector(Iter b, Iter e) -> vector<typename std::iterator_traits<Iter>::value_type>;
```

```
template<typename T>
class Something {
    ...
public:
    Something(const T* x);
};

Something(const char*) -> Something<std::string>;
```

Смернице за закључивање

- Још један интересантан пример:

```
template<typename T>
struct Tad {
    Tad(T&& x) : m_v(std::forward<T>(x)) {}

    T m_v;
};

Tad<int> a(6);
Tad<int> b(i);
Tad<int> c(ci);

template<typename T>
Tad(T&&) -> Tad<T>;

Tad a(6);
Tad b(i);
Tad c(ci);
```

- Шта је T у ова три случаја?

Полиморфизам

- Ово је релативно честа конструкција:

```
struct BaseClass {
  virtual void specific() =0;

  void function() {
    std::cout << “Some general code”;
    specific();
  }
};

struct Derived1 : BaseClass {
  void specific() override { std::cout << “Derived1”; }
};

struct Derived2 : BaseClass {
  void specific() override { std::cout << “Derived2”; }
};

Derived1 x;
Derived2 y;
x.function();
y.function();
```

Полиморфизам

- Али, можемо урадити и овако:

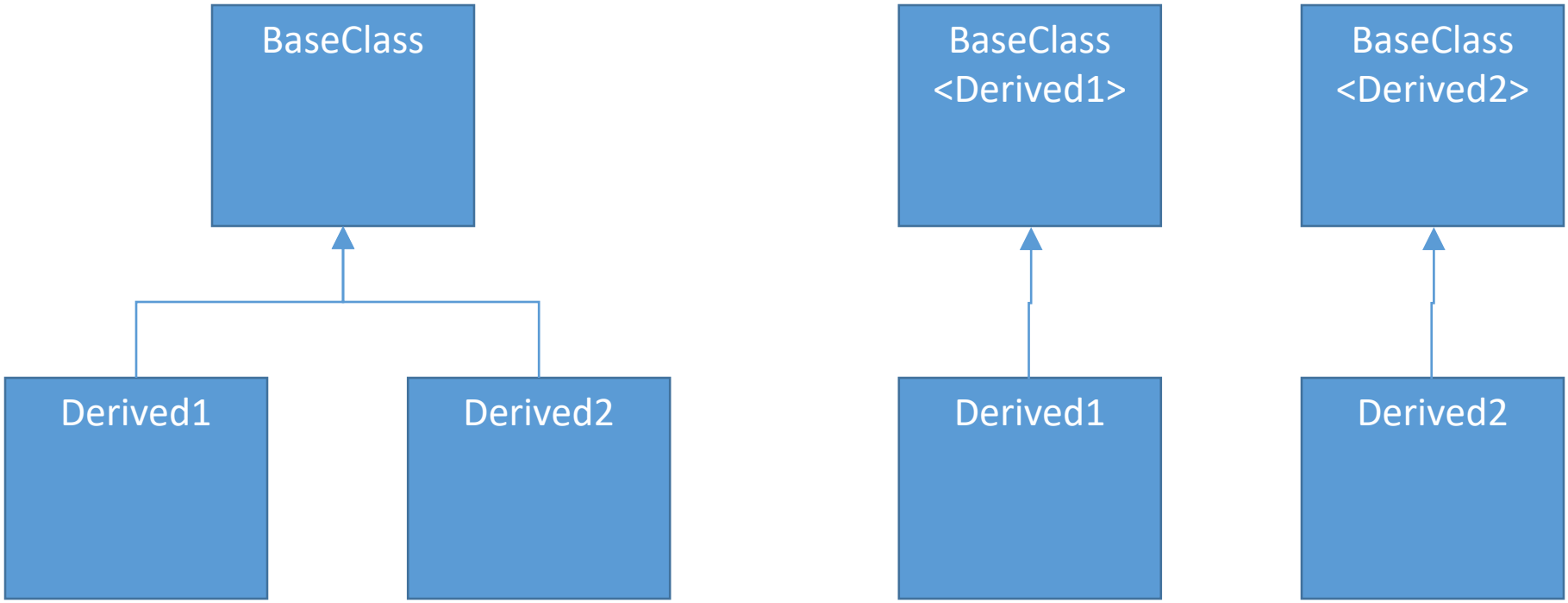
```
template<typename T>
struct BaseClass {
  void function() {
    std::cout << "Some general code";
    static_cast<T*>(this)->specific();
  }
};

struct Derived1 : BaseClass<Derived1> {
  void specific() { std::cout << "Derived1"; }
};

struct Derived2 : BaseClass<Derived2> {
  void specific() { std::cout << "Derived2"; }
};

Derived1 x;
Derived2 y;
x.function();
y.function();
```

Статички полиморфизам - CRTP

- То што је показано на претходном примеру представљају статички полиморфизам, а он је постигнут коришћењем нечега што се назива CRTP (Curiously Recurring Template Pattern).
- Статички полиморфизам може бити користан у многим случајевима када немамо показиваче и референце на базну класу, тј. када током превођења имамо информације о типу објеката.
- CRTP може бити коришћен за разне ствари.

Статички полиморфизам - CRTP

- Приметите да је хијерархија класа различита у ова два примера:



CRTP

- CRTP може бити коришћен да дода функционалност типу.
- Ево класе која има функционалност бројања објеката:

```
class MyClass1 {
  static int objCounter;

public:
  MyClass1() { ++objCounter; }
  MyClass1(const MyClass1& x) { ++objCounter; } // plus what has to be done
  ~MyClass1() { --objCounter; }
  int getObjectNum() { return objCounter; }
};

inline static int MyClass1::objCounter = 0;
```

- Ствари се компликују када имамо више конструктора и више различитих класа које треба да имају исту ту функционалност.

CRTP

- Са CRTP обрасцем („идиомом“) можемо направити шаблон базне класе која пружа ту могућност свим изведеним класама.

```
template<typename T>
class EnableObjCount {
  static int objCounter;

protected:
  EnableObjCount() { ++objCounter; }
  ~EnableObjCount() { --objCounter; }

public:
  int getObjectNum() { return objCounter; }
};

template<typename T> inline int EnableObjCount<T>::objCounter = 0;


class MyClass1 : public EnableObjCount<MyClass1> {
  //...
};

class MyClass2 : public EnableObjCount<MyClass2> {
  //...
};
```

CRTP

- На сличан начин, CRTP може бити употребљен и у хијерархији класа са динамичким полиморфизмом.

```
struct BaseClass {
  virtual ~BaseClass(){};
  virtual BaseClass* clone() const =0;
};

struct Derived1 : BaseClass {
  BaseClass* clone() const override {
    return new Derived1(*this);
  }
};

struct Derived2 : BaseClass {
  BaseClass* clone() const override {
    return new Derived2(*this);
  }
};
```

CRTP

- На сличан начин, CRTP може бити употребљен и у хијерархији класа са динамичким полиморфизмом.

```
struct BaseClass {
  virtual ~BaseClass(){};
  virtual BaseClass* clone() const =0;
};

template<typename T>
struct BaseClassCRTP : BaseClass {
  BaseClass* clone() const override {
    return new T(*static_cast<const T*>(this));
  }
};

struct Derived1 : BaseClassCRTP<Derived1> {
};

struct Derived2 : BaseClassCRTP<Derived2> {
};
```

CRTP

- Један (једини?) пример CRTP-а у стандардној библиотеци:

```
struct MyClass : std::enable_shared_from_this<MyClass> {
  //...
};

void foo(MyClass& x);

std::shared_ptr<MyClass> p1 = std::make_shared<MyClass>();

foo(*p1);

void foo(MyClass& x) {
  //...
  std::shared_ptr<MyClass> p2 = x.shared_from_this();
}
```