Опис проблема

- Имамо разне геометријске облике.
- Желимо да обављамо неку обраду над структуром таквих елемената – тако да сваки тип елемената има своју верзију те обраде (нпр. draw, perimeter итд.)

Желимо да можемо урадити овако нешто:

- Неке од тема:
- Како складиштити разноврсне елементе у контејнеру?
- Како обављати исту обраду над разноврсним елементима?
- Додавање новог елемента (колико има врста елемената)
- Додавање нове обраде (колико има обрада)

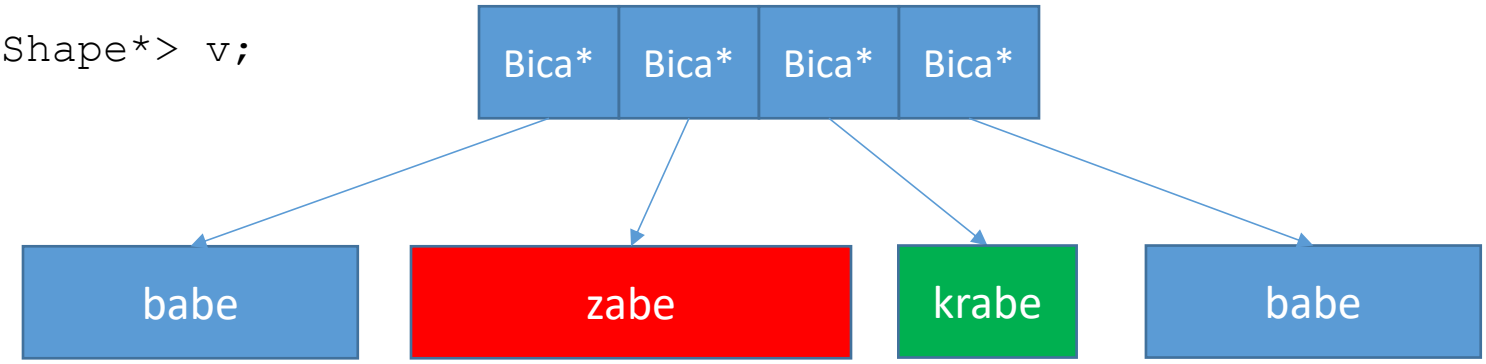
```
Canvas canv;
Rectangle r;
Circle c;

r.draw();
c.draw();

canv.addShape(r); canv.addShape(c);

canv.drawShapes();
```

Решење 1 - ООП

```
struct Shape {
  void draw() {
    // заједнички део кода -- нпр. постави боју и дебљину
    drawLines(); // део специфичан за конкретни облик
  }
protected:
  virtual void drawLines() =0;
  ...
  virtual ~Shape() {}
};

struct Rectangle : Shape {
protected:
  void drawLines() override { /*...исцртај Rectangle...*/ }
};

struct Circle : Shape {
protected:
  void drawLines() override { /*...исцртај Circle...*/ }
};
```

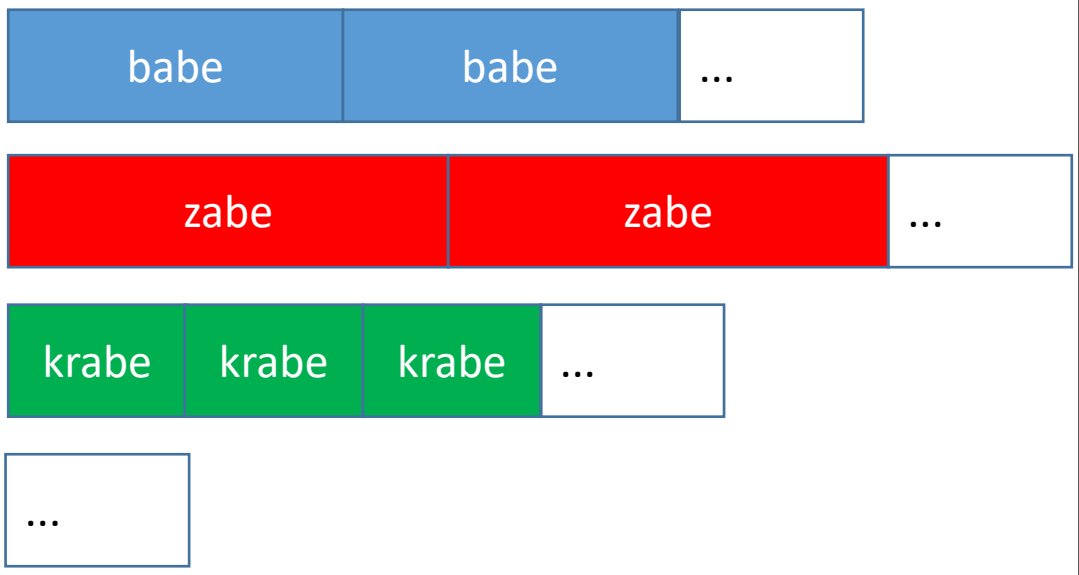
Решење 1 - ООП

```
struct Canvas {
  void addShape(Shape& x) { v.push_back(&x); }
  void drawShapes() {
    for (auto& it : v)
      it->draw();
  }
private:
  std::vector<Shape*> v;
};

Canvas canv;
Rectangle r;
Circle c;

r.draw();
c.draw();

canv.addShape(r); canv.addShape(c);

canv.drawShapes();
```



Решење 2 - Више контејнера

```
struct Canvas {
  void addShape(Rectangle x) { vr.push_back(x); }
  void addShape(Circle x) { vc.push_back(x); }
  void drawShapes() {
    for (auto& it : vr) it.draw();
    for (auto& it : vc) it.draw();
  }
private:
  std::vector<Rectangle> vr;
  std::vector<Circle> vc;
};

Canvas canv;
Rectangle r;
Circle c;

r.draw();
c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```



Решење 2 - Више контејнера

Заједнички део се понавља :-/

```
struct Rectangle {
  void draw() {
    // заједнички део кода -- нпр. постави боју и дебљину
    drawLines();
  }
protected:
  void drawLines() { /*...исцртај Rectangle...*/ }
};

struct Circle{
  void draw() {
    // заједнички део кода -- нпр. постави боју и дебљину
    drawLines();
  }
protected:
  void drawLines() { /*...исцртај Circle...*/ }
};
```

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:

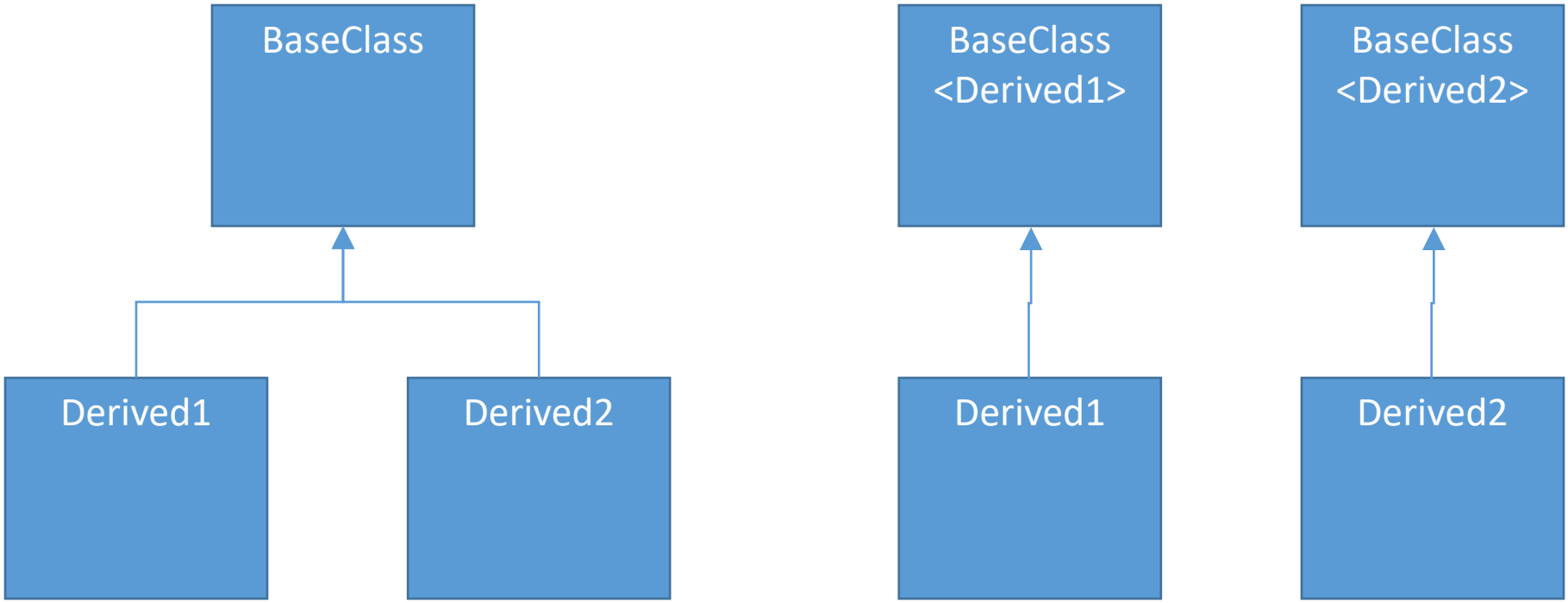
```
template<typename T>
struct BaseClass {
  void function() {
    std::cout << "Some general code";
    static_cast<T*>(this)->specific();
  }
};


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 {
  inline static int objCounter = 0;

public:
  MyClass1() { ++objCounter; }
  MyClass1(const MyClass1& x) { ++objCounter; } // плус остали код за конструкцију
  ~MyClass1() { --objCounter; }
  int getObjectNum() { return objCounter; }
};
```

- 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 {
  inline static int objCounter = 0;

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

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

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> {
};
```

Решење 2 - Више контејнера

Заједнички део се понавља :-/

```
struct Rectangle {
  void draw() {
    // заједнички део кода -- нпр. постави боју и дебљину
    drawLines();
  }
protected:
  void drawLines() { /*...исцртај Rectangle...*/ }
};

struct Circle{
  void draw() {
    // заједнички део кода -- нпр. постави боју и дебљину
    drawLines();
  }
protected:
  void drawLines() { /*...исцртај Circle...*/ }
};
```

Решење 2 - Више контејнера

```
template<typename T>
struct Shape {
  void draw() {
    // заједнички део кода -- нпр. постави боју и дебљину
    static_cast<T*>(this)->drawLines();
  }
protected:
  ...
};

struct Rectangle : Shape<Rectangle> {
protected:
  void drawLines() { /*...исцртај Rectangle...*/ }
};

struct Circle : Shape<Circle> {
protected:
  void drawLines() { /*...исцртај Circle...*/ }
};
```

Решење 2 - Више контејнера

```
template<typename T>
struct Shape {
  void draw() {
    // заједнички део кода -- нпр. постави боју и дебљину
    static_cast<T*>(this)->drawLines();
  }
protected:
  ...
};

struct Rectangle : Shape<Rectangle> {
  friend Shape<Rectangle>; // постави претка за пријатеља
protected:
  void drawLines() { /*...исцртај Rectangle...*/ }
};

struct Circle : Shape<Circle> { // или уклони protected/private
  void drawLines() { /*...исцртај Circle...*/ }
};
```

Шта ако...

- Имамо типове разних геометријских облика (и сви они наслеђују Shape)
- Желимо да обављамо већи број разних обрада над структуром таквих елемената – тако да сваки тип елемената има своју верзију те обраде (нпр. draw_shape)

Проблеми са досадашњим решењима

- У свим класама елемената (изведеним из Shape) морамо имплементирати одговарајућу виртуелну функцију која обавља жељену обраду... а то је мукотрпно ако имамо много таквих обрада, па морамо да „скачемо“ по класама.
- Такође, ако имам много обрада, онда претрпавамо типове елемената виртуелним методама.

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct C {
  void foo(myClass1& x); // 1
  void foo(myClass2& x); // 2
  void foo(myClass3& x); // 3
};


// ...
C a;
// ...
a.foo(y); // 1, 2 ili 3? Од чега зависи? Када ће се знати?
```

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct C {
  void foo(myClass1& x); // 1
  void foo(myClass2& x); // 2
  void foo(myClass3& x); // 3
};


// ...
C a;
// ...
a.foo(y); // 1, 2 ili 3? Од чега зависи? Када ће се знати?
```

Статичко разрешење

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct B {
  virtual void bar(myClass& x) = 0;
};

struct D1 : B {
  void bar(myClass& x) override { ... }
};

struct D2 : B {
  void bar(myClass& x) override { ... }
};



B* p;
myClass y;
// ...
p->bar(y); // B::bar, D1::bar ili D2::bar? Од чега зависи?
           // Када ће се знати?
```

Разрешење позива функције

- Како се и када одређује шта ће бити позвано?

```
struct B {
  virtual void bar(myClass& x) = 0;
};

struct D1 : B {
  void bar(myClass& x) override { ... }
};

struct D2 : B {
  void bar(myClass& x) override { ... }
};
```

Динамичко разрешење

```
B* p;
myClass y;
// ...
p->bar(y); // B::bar, D1::bar ili D2::bar? Од чега зависи?
           // Када ће се знати?
```

Посетилац (engl. Visitor)

- Образац Посетиоца комбинује ова два механизма:

```
struct B {
  virtual void accept(Visitor& v) = 0;
};
struct D1 : B {
  void accept(Visitor& v) override { v.visit(*this); }
};
struct D2 : B {
  void accept(Visitor& v) override { v.visit(*this); }
};
// accept мора постојати за сваку класу у хијерархији

struct Visitor {
  virtual void visit(D1& x) = 0;
  virtual void visit(D2& x) = 0; //по једна метода за сваку класу
};

struct VisForFunc1 : Visitor { // по класа за сваку обраду
  void visit(D1& x) override { ... }
  void visit(D2& x) override { ... }
};
```

```
B* p; VisForFunc1 vis;
p->accept(vis);
```

Решење 3 - са варијантама

```
template<typename... Ts>
struct Canvas {
  using ShapesVar = std::variant<Ts...>;
  void addShape(ShapesVar x) { v.push_back(x); }
  void drawShapes() {
    for (auto& it : v)
      it->draw();
  }
private:
  std::vector<ShapesVar> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

r.draw(); c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```

```
struct V {
  void operator()(Rectangle x) { x.draw(); }
  void operator()(Circle x) { x.draw(); }
};
```

Решење 3 - са варијантама

```
template<typename... Ts>
struct Canvas {
  using ShapesVar = std::variant<Ts...>;
  void addShape(ShapesVar x) { v.push_back(x); }
  void drawShapes() {
    for (auto& it : v)
      std::visit(V{}, it);
  }
private:
  std::vector<ShapesVar> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

r.draw(); c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```

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

Решење 3 - са варијантама

```
template<typename... Ts>
struct Canvas {
  using ShapesVar = std::variant<Ts...>;
  void addShape(ShapesVar x) { v.push_back(x); }
  void drawShapes() {
    for (auto& it : v)
      std::visit(V{}, it);
  }
private:
  std::vector<ShapesVar> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

r.draw(); c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```

Решење 3 - са варијантама

```
template<typename... Ts>
struct Canvas {
  using ShapesVar = std::variant<Ts...>;
  void addShape(ShapesVar x) { v.push_back(x); }
  void drawShapes() {
    for (auto& it : v)
      std::visit([](auto x){ x.draw(); }, it);
  }
private:
  std::vector<ShapesVar> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

r.draw(); c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```

Решење 4 - са торкама

```
template<typename... Ts>
struct Canvas {
  template<typename T>
  void addShape(T x) { std::get<std::vector<T>>(v).push_back(x); }

  void drawShapes() {
    ?????????
    ?????????
  }
private:
  std::tuple<std::vector<Ts>...> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

r.draw(); c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```

Решење 4 - са торкама

```
template<typename... Ts>
struct Canvas {
  template<typename T>
  void addShape(T x) { std::get<std::vector<T>>(v).push_back(x); }

  void drawShapes() {
    auto f = [](auto x){ for (auto it : x) it.draw(); };
    ?????????
  }
private:
  std::tuple<std::vector<Ts>...> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

r.draw(); c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```

Решење 4 - са торкама

```
template<typename... Ts>
struct Canvas {
  template<typename T>
  void addShape(T x) { std::get<std::vector<T>>(v).push_back(x); }

  void drawShapes() {
    auto f = [](auto x){ for (auto it : x) it.draw(); };
    ( f(std::get<std::vector<Ts>>(v)), ... );
  }
private:
  std::tuple<std::vector<Ts>...> v;
};

Canvas<Rectangle, Circle> canv;
Rectangle r;
Circle c;

r.draw(); c.draw();

canv.addShape(r); canv.addShape(c);
canv.drawShapes();
```