Још неколико интересантних тема

поравнање, std::optional, strong typing, std::variant

Поравнање

- Сваки тип има своје природно поравнање, што значи да објекти тог типа треба да се налазе у меморији на адреси која је умножак неког целог броја, односно – тачније – неког степена двојке.
- Информацију о поравнању добијамо са **alignof**:

```
struct SomeType{
  char c;
  int n;
};

std::size_t x = alignof(int);
std::size_t y = alignof(SomeType);
```

Поравнање

- Можемо неком објекту одредити и веће поравнање од његовог природног.

```
alignas(16) int data = 0;

alignas(long) int array[57];

struct alignas(2 * alignof(int)) Struct {
  char Data1;
  char Data2;
  char Data3;
}
```

std::optional

- Размотримо следећи пример:

```
bool findResult(MyType x, double& res);

double result;
if (findResult(a, result))
  std::cout << result;
else {
  std::cout << "Invalid";
  // немој користити резултат, јер није валидан
}
```

std::optional

- Размотримо следећи пример:

```
std::tuple<bool, double> findResult(MyType x);

auto result = findResult(a);
if (std::get<0>(result))
  std::cout << std::get<1>(result);
else {
  std::cout << "Invalid";
  // немој користити std::get<1>(result), јер није валидно
}
```

std::optional

- Размотримо следећи пример:

```
std::pair<bool, double> findResult(MyType x);

auto result = findResult(a);
if (result.first)
  std::cout << result.second;
else {
  std::cout << "Invalid";
  // немој користити result.second, јер није валидно
}
```

std::optional

- Размотримо следећи пример:

```
std::optional<double> findResult(MyType x);

auto result = findResult(a);
if (result)
  std::cout << *result; // or result.value()
else {
  std::cout << “Invalid”;
 // немој користити резултат, јер није валидан
}
```

std::optional

- Размотримо следећи пример:

```
std::optional<double> findResult(MyType x);

std::cout << findResult(a).value_or(0.0);
```

Strong typing

- Наше променљиве (објекти) обично представљају нешто из стварног домена примене.
- На пример:

```
// прима вредности у метрима, метрима^2, и враћа у метрима^3
double calculateVolume(double height, double baseArea) {
    return height * baseArea;
}

double h = 1.0; // у стопама
double base = 2.5; // у стопама^2

std::cout << calculateVolume(base, h);
```

Strong typing

- Овако је тај проблем решен за време:

```
void foo(std::chrono::seconds t);


foo(1s);
foo(1min); // десиће се конверзија -> 60s
foo(1); // грешка!
```

Strong typing

- Покушаћемо да урадим слично за дужину, површину и запремину.

```
struct LengthInMeters {
    LengthInMeters(double x) : m_x(x) {}
    double getAmount() const { return m_x; }
protected:
    double m_x = 0.0;
};

struct AreaInMeters {
    AreaInMeters(double x) : m_x(x) {}
    double getAmount() const { return m_x; }
protected:
    double m_x = 0.0;
};

struct VolumeInMeters {
    VolumeInMeters(double x) : m_x(x) {}
    double getAmount() const { return m_x; }
protected:
    double m_x = 0.0;
};
```

Strong typing

- Покушаћемо да урадим слично за дужину, површину и запремину.

```
AreaInMeters operator*(const LengthInMeters& a, const LengthInMeters& b)
{
    return AreaInMeters(a.getAmount() * b.getAmount());
}
VolumeInMeters operator*(const LengthInMeters& a, const AreaInMeters& b)
{
    return VolumeInMeters(a.getAmount() * b.getAmount());
}
VolumeInMeters operator*(const AreaInMeters& a, const LengthInMeters& b)
{
    return VolumeInMeters(a.getAmount() * b.getAmount());
}
```

Strong typing

- Покушаћемо да урадим слично за дужину, површину и запремину.

```
VolumeInMeters calculateVolume(
   LengthInMeters height, AreaInMeters baseArea) {
    return height * baseArea;
}

LengthInMeters h = 1.0; // у стопама <- сада је грешка мало видљивија
AreaInMeters base = 2.5; // у стопама^2

std::cout << calculateVolume(base, h).getAmount();
             // грешка током превођења!!!

std::cout << calculateVolume(h, base).getAmount();
             // сада је OK
```

Strong typing

- Можемо мало боље.

```
struct LengthInMeters {
    explicit LengthInMeters(double x) : m_x(x) {}
    double getAmount() const { return m_x; }
protected:
    double m_x = 0.0;
};

LengthInMeters operator""_m(long double x) {
    return LengthInMeters(static_cast<double>(x));
}


// а можемо слично и за Area

AreaInMeters operator""_m2(long double x) {
    return AreaInMeters(static_cast<double>(x));
}
```

Strong typing

- Можемо мало боље.

```
VolumeInMeters calculateVolume(
   LengthInMeters height, AreaInMeters baseArea) {
    return height * baseArea;
}

LengthInMeters h = 1.0_m; // сада је потенцијална грешка још видљивија
LengthInMeters h1 = 1.0; // ово је сада грешка током превођења!
AreaInMeters base = 2.5_m2;

std::cout << calculateVolume(base, h).getAmount();
             // грешка током превођења!!!

std::cout << calculateVolume(h, base).getAmount();
             // сада је OK
```

Strong typing

- А можемо и да поједностављујемо прављење јаких типова:

```
template<typename Name>
struct NamedType {
    explicit NamedType(double x) : m_x(x) {}
    double getAmount() const { return m_x; }
protected:
    double m_x = 0.0;
};

using LengthInMeters = NamedType<struct _LengthInMeters>;
using AreaInMeters = NamedType<struct _AreaInMeters>;
using VolumeInMeters = NamedType<struct _VolumeInMeters>;
```

std::variant

- std::variant је, на неки начин, супротно од std::tuple.
- Оно што је std::tuple наспрам структуре struct, то је std::variant наспрам уније.
- У теорији типова, прво је производ типова, а друго – сума/унија.

```
union X {
    int a1;
    float a2;
};

union Y {
    double a1;
    float a2;
    long a3;
};
```

```
std::variant<
    int,
    float> x;

std::variant<
    double,
    float,
    long> y;
```

std::variant

- За приступ пољима варијанте, користи се get функција.

```
std::variant<int, float> x;

std::cout << std::get<1>(x);
std::get<1>(x) = 5.7f;
```

- Уколико x тренутно садржи алтернативу са индексом 1, биће враћена референца на вредност смештену у x. У супротном, биће бачен овај изузетак: std::bad_variant_access

```
std::variant<int, float> x;

std::cout << std::get<float>(x); // добавља & или баца изузетак
std::get<int>(x) = 5.7f; // грешка током превођења
```

std::variant

- Још корисних функција:

```
std::variant<int, float> x;

x.index(); // индекс тренутне алтернативе

std::holds_alternative<float>(x);
    // true уколико је float тренутна алтернатива


auto p = std::get_if<1>(x);
if (p)
    std::cout << *p;

auto p = std::get_if<float>(x);
if (p)
    std::cout << *p;
```

std::variant

- Занимљив пример:

```
struct Shape {
  void draw() {
    // постави боју и дебљину
    drawLines();
  }
protected:
  virtual void drawLines() =0;
  Color color = Color::RED;
  int lineThickness = 1;
};

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

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

std::variant

- Занимљив пример:

```
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();
```

std::variant

- Другачији приступ:

```
template<typename T>
struct Shape {
  void draw() {
    // постави боју и дебљину
    static_cast<T*>(this)->drawLines();
  }
protected:
  Color color = Color::RED;
  int lineThickness = 1;
};

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

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

std::variant

- Другачији приступ:

```
template<typename T>
struct Shape {
  void draw() {
    // постави боју и дебљину
    static_cast<T*>(this)->drawLines();
  }
protected:
  Color color = Color::RED;
  int lineThickness = 1;
};

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...*/ }
};
```

std::variant

- Ово ради:

```
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();
```

std::variant

- A std::variant може помоћи за остало:

```
using ShapesVar = std::variant<Rectangle, Circle>;

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

Canvas canv;
Rectangle r;
Circle c;

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

canv.drawShapes();
```

std::variant

- A std::variant може помоћи за остало:

```
using ShapesVar = std::variant<Rectangle, Circle>;

struct Canvas {
  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 canv;
Rectangle r;
Circle c;

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

canv.drawShapes();
```

std::variant

- Али, можемо још и боље са std::tuple:

```
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;

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

canv.drawShapes();
```

Да проверимо...

quick-bench.com