C++

Fall 2020

compscicenter.ru

Башарин Егор

eaniconer@gmail.com
https://t.me/egorbasharin

Лекция V

Enumerations, Classes

Enumerations

Перечисления

Motivation

```
const int RED = 0;     // we could use #define instead of
const int GREEN = 1;   // to get the same behavior
const int BLUE = 2;

uint32_t uintColor(int color);
```

нетипобезопасно: `uintColor` может принять любой объект, который неявно преобразуется к `int`

Что хотим получить:

```
uintColor(1);         // Error
uintColor(BLUE);      // OK
```

Enumeration

```
enum Color {
    RED,
    GREEN,
    BLUE
};

uint32_t uintColor(Color color);

uintColor(1);     // Error: No matching function for call
uintColor(RED);   // OK
```

Enumeration

```
enum Color { RED, GREEN, BLUE };
```

Syntax:

```
enum-key [enum-name] [enum-base] { [enumerator-list] };
```

Перечисление:

- Задает отдельный тип
- Значения ограничены некоторым диапазоном
- Может содержать именованные константы, значения который имеют целочисленный тип. Этот тип известен как базовый тип (`underlying type`) перечисления

Enumeration

Syntax:

```
enum-key [enum-name] [enum-base] { [enumerator-list] };
```

- `enum-key`
  - unscoped: `enum`
  - scoped: `enum class`, `enum struct`
- `enum-name` — имя перечисления; может отсутствовать у unscoped-перечислений
- `enum-base` — двоеточие с целочисленным типом; указывает на базовый тип перечисления

Enumeration

Syntax:

```
enum-key [enum-name] [enum-base] { [enumerator-list] };
```

- `enumerator-list` — список идентификаторов (enumerators), перечисленных через запятую.

Enumerator syntax:

```
identifier [ = constexpr]
```

Unscoped enumeration

Form:

```
enum name { [enumerator-list] }         (1)
enum name : type { [enumerator-list] }  (2)
```

(1) Базовый тип не фиксирован и зависит от реализации:

- все элементы перечисления представимы в этом типе
- не более `int`/`unsigned int`, если соответсвующего типа достаточно

(2) Базовый тип фиксирован

Unscoped enumeration

Элементы перечисления становятся именованными константами

```
enum ShapeType {
  TRIANGLE,
  SQUARE,
};

ShapeType getShapeType();

int main() {
    switch (getShapeType()) {
    case TRIANGLE: std::cout << "Triangle\n"; break;
    case SQUARE:   std::cout << "Square\n";   break;
    }
}
```

Unscoped enumeration

Enumerator values

- Значения элементов перечисления имеют тип совпадающий с базовым
- Значение явно задается инициализатором
- Если инициализатора нет, то значение равняется значению предыдущего элемента плюс 1
- Если у первого элемента нет инициализатора, то его значение 0

Unscoped enumeration

Значения неявно приводятся к целочисленным типам

```
enum ShapeType {
  TRIANGLE,
  SQUARE,
};

int main() {
    int t = TRIANGLE; // implicitly-convertible
}
```

Unscoped enumeration

Значение целочисленного типа, типа с плавающей точкой или типа перечисления может быть преобразовано к любому типу перечисления с помощью `static_cast`

- unfixed underlying type:

```
enum AccessType { read = 1, write = 2 }; // range: 0..3

int main() {
    // OK, value in range
    AccessType ac1 = static_cast<AccessType>(3);

    // UB (since c++17), value out of range
    AccessType ac2 = static_cast<AccessType>(4);
}
```

Unscoped enumeration

Значение целочисленного типа, типа с плавающей точкой или типа перечисления может быть преобразовано к любому типу перечисления с помощью `static_cast`

- fixed underlying type:

```
// range: all values of int
enum AccessType : int { read = 1, write = 2 };

int main() {
    AccessType ac1 = static_cast<AccessType>(3); // OK
    AccessType ac2 = static_cast<AccessType>(4); // OK
}
```

Unscoped enumerations

Проблемы

- Если **базовый тип** не указан явно, то он **не фиксирован стандартом**. Поэтому знаковость и размеры типа могут меняться от компилятора к компилятору, что ведет к проблемам с **переносимостью**.
- **Неявное преобразование** к целочисленным типам.
- **Область видимости**. Имена констант из разных перечислений могут конфликтовать.

Scoped enumerations

Syntax:

```
enum struct|class name { [enumerator-list] }         (1)
enum struct|class name : type { [enumerator-list] }  (2)
```

- (1) — перечисление с базовым типом `int`
- `struct` и `class` эквивалентны
- элементы перечисления — именованные константы, которые доступны, только с указанием имени перечисления
- отсутствует неявное преобразование к интегральным типам (можно преобрзовать используя `static_cast`)

Scoped enumerations

Syntax:

```
enum struct|class name { [enumerator-list] }        (1)
enum struct|class name : type { [enumerator-list] }  (2)
```

```cpp
enum class Color { red, green, blue };

int main() {
    int red = Color::red; // error
    int blue = static_cast<int>(Color::blue); // OK
}
```

Classes

Классы

Class

Задает пользовательский тип

```
#include <iostream>

struct MyType {};

int main() {
    MyType myObject;
    std::cout
        << "sizeof(MyType): "
        << sizeof(MyType)
        << std::endl;
}
```

Syntax

```
class-key class-head-name { member-specification }
```

`class-key` — это `class` или `struct`; влиет на дефолтный модификатор доступа к членам класса

`class-head-name` — имя класса

`member-specification` содержит спецификаторы доступа, поля, псевдонимы типов, методы…

Non-static data members

Поля

```
struct MyType {
  int field;
};

int main() {
    MyType obj{10};
    std::cout << obj.field << std::endl;
}
```

Non-static data members

Поля

```
struct MyType {
  int field; // data member
  int& ref;  // data member of reference type
  int* ptr;  // data member of pointer type

  int arr[2]; // array
};

int main() {
    int i = 0;
    MyType obj{10, i, &i, {1,2}};
}
```

Non-static member functions

Определение и объявление методов

```
struct Vec {
  double x;
  double y;
  double norm2() { return std::sqrt(x*x + y*y); }
  double norm1();
};

double Vec::norm1() // not inline
{
    return std::abs(x) + std::abs(y);
}

int main() {
    Vec vec{3.0, 4.0};
    assert(vec.norm2() == 5.0);
    assert(vec.norm1() == 7.0);
}
```

- Методы, определенные в теле класса, считаются `inline`

Non-static member functions

Константность

```
struct Vec {
  double x;
  double y;
  double norm2() { return std::sqrt(x*x + y*y); }
};

int main() {
    const Vec vec{3.0, 4.0}; // const object
    assert(vec.norm2() == 5.0); // error: function not m
}
```

Non-static member functions

Константность

```
struct Vec {
  double x;
  double y;
  double norm2() const { return std::sqrt(x*x + y*y); }
};

int main() {
    const Vec vec{3.0, 4.0}; // const object
    assert(vec.norm2() == 5.0); // OK
}
```

Non-static member functions

Пример 1

```
struct ValueHolder {
  int value_;
  ...
};

int main() {
    ValueHolder holder{10};
    holder.value() = 11;                // OK
    assert(holder.value() == 11);
    assert(holder.value_ == 11);

    const ValueHolder& holderRef = holder;
    assert(holderRef.value() == 11);
    // holderRef.value() = 12;                // Error;
}
```

Non-static member functions

Пример 1

```
struct ValueHolder {
  int value_;
  int& value() { return value_; }
  int value() const { return value_; }
};

int main() {
    ValueHolder holder{10};
    holder.value() = 11;              // OK
    assert(holder.value() == 11);
    assert(holder.value_ == 11);

    const ValueHolder& holderRef = holder;
    assert(holderRef.value() == 11);
    // holderRef.value() = 12;              // Error;
}
```

Non-static member functions

Что еще дает const specifier

```
struct ValueHolder {
  int value_;
  int value() const {
      // Error: cannot assign within const member func
      value_ = 1;

      return value_;
  }
};

int main() {
    ValueHolder holder{10};
    assert(holder.value() == 11);

    const ValueHolder& holderRef = holder;
    assert(holderRef.value() == 11);
}
```

Non-static member functions

Константные методы

- Используюся для константных объектов
- Используются для неконстантных объектов, если неконстантная перегрузка метода отсутствует
- Защищают поля класса от изменений

Спецификаторы доступа

```
struct S { int i; }; // public by default
class C { int i; };  // private by default

int main() {
    S s;
    s.i = 1; // OK: access to public member
    C c;
    c.i = 1; // error: no access to private member
}
```

Спецификаторы доступа

public, private

```
class C {
public:
    int getI() const { return i_; } // public
    int getJ() const { return j_; } // public
private:
    int i_ = 10;  // private
    int j_ = 11;  // private
};

int main() {
    C c;
    c.getI(); // OK
    c.getJ(); // OK
    c.i_; // error
    c.j_; // error
}
```

static members

Статические члены (*не связаны с объектом класса*)

Syntax:

```
static data_member;
static member_function;
```

```
struct T {
  static int n;       // declaration
  static int getN(); // declaration
};

int T::n = 10;                  // definition
int T::getN() { return n; }  // definition

int main()
{
    // forms of access:
    int n1 = T::getN(); // form1
    T t;
    int n2= t.getN();   // form2
}
```

Constant static members

```
enum class Color { RED, BLUE };
struct T {
  const static Color color = Color::RED; // OK
  const static int n = 33;                // OK
  const static double d = 10.0;           // Error
};
```

Для целочисленных типов и типов перечисления можно использовать инициализаторы (константные выражения) прямо в теле класса.

Pointers to members

Функции

```
struct T
{
  void f() const { std::cout << "f()\n"; }
  static void g() { std::cout << "g()\n"; }
};

int main()
{
   void (T::*ptrToMethod)() const = &T::f;
   void (*ptrToStaticFunc)() = T::g;

   T t;
   (t.*ptrToMethod)();
   ptrToStaticFunc();
}
```

Pointers to members

Поля

```
struct T
{
  static int s;
  int n;
};
int T::s = 10;

int main()
{
    int* ptrToStaticMember = &T::s;
    int T::* ptrToMember = &T::n;

    T t { 11 };
    std::cout << *ptrToStaticMember << "\n"
        << (t.*ptrToMember);
}
```

Pointer to class

```
struct Pair {
  int first;
  int second;
};

int main()
{
    Pair pair{1, 2};
    Pair* ptr = &pair;
    std::cout << ptr->first + ptr->second;
}
```

Для доступа к членам класса используется →

this pointer

- в теле нестатической функции указывает на объект, для которого этот метод был вызван

```
struct T {
  void f() const {
      std::cout << "f: " << this << std::endl;
  }
};

int main()
{
    T t;
    std::cout << "main: " << &t << std::endl;
    t.f();
}
```

this pointer

использование → для доступа к членам класса

```
struct T {
  int f() { return this->g() + this->i; }
private:
  int g() { return 42; }
  int i;
};

int main()
{
    T t;
    std::cout << t.f() << std::endl;
}
```

Constructor

Конструктор — специальный метод, используемые для инициализации объекта

Declaration syntax:

```
class-name ([parameter-list])
```

Constructor

Example

```
class T {
public:
    T() { std::cout << "T()\n"; }
    T(int) { std::cout << "T(i)\n"; }
    T(int, int) { std::cout << "T(i,i)\n"; }
};

int main()
{
   T t1();
   T t2(1);
   T t3(1, 1);
}
```

Constructor

Example

```
class T {
public:
    T() { std::cout << "T()\n"; }
    T(int) { std::cout << "T(i)\n"; }
    T(int, int) { std::cout << "T(i,i)\n"; }
};

int main()
{
   T t1(); // function declaration
   T t2(1);
   T t3(1, 1);
}
```

Data member initialization

Поля класса инициализируются до выполнения тела конструктора.

Два способа инициализации:

1. member initializer list (*список инициализации*)
2. default member initializer

Data member initialization

```
class T {
public:
    T(int i, int j) : i_(i), j_(j) { } // member initial

private:
    int i_;
    int j_ = 3; // default member initializer
    int k_ = 4; // default member initializer
};
```

Data member initialization

Порядок инициализации

Поля инициализируются в порядке появлениях их объявлений в теле класса.

Изменения порядке инициализаторов в списке инициализации не имеют никакого эффекта.

Data member initialization

Порядок инициализации. Пример.

```
struct T {
    T(int val) : j(2*val), i(j) { }
    int i;
    int j;
};

int main()
{
   T t(3);
   std::cout << t.i;
}
```

Default constructor

Конструктор, который может быть вызван без аргументов.

```
struct T {
    T() : i(0) {}

    int i;
};
```

Default constructor

Implicitly-declared default constructor

Если пользователь не задал других конструкторов, то компилятор может создать неявно конструктор по умолчанию.

При наличии других конструкторов, можно явно дать указание компилятору сгенерировать конструктор по умолчанию:

```
struct T {
    T() = default;
    T(int) { }
};
```

Copy constructor

Конструктор, у которого:

- первый параметр имеет тип `T&` или `const T&`
- либо больше параметров нет, либо все они имеют значение по умолчанию

Этот конструктор вызывается всякий раз, когда нужно инициализировать новый объект, используя объект того же типа.

Компилятор может сгенерировать конструктор копирования, если отсутствуют пользовательские конструкторы копирования.

Destructor

Специальный метод, который вызываетя, когда время жизни объекта подходит к концу.

```
struct T {
    ~T() { }
};
```

explicit specifier

```
struct T {
    T(int) { }
};

void f(T t) {}

int main()
{
    f(10); // implicit conversion
}
```

explicit specifier

```
struct T {
    explicit T(int) { }
};

void f(T t) {}

int main()
{
    f(10); // error
}
```