

Программирование на языке C++

Шаблов Анатолий
anatoliishablov@gmail.com

ИТМО, весенний семестр 2024

Структуры и классы

Структуры и классы

- Определяет новый тип данных в программе
- Может включать:
 - поля данных
 - методы (функции)
 - типы
 - псевдонимы типов
 - члены вложенных `enum`
 - шаблоны
- Может быть наследован от других классов

Определение

класс имя [: список наследования] { члены }

- класс – ключевое слово *class* или *struct*
- имя – идентификатор
- члены – объявление членов класса
- список наследования – [*public/protected/private*] *base1*, ...

P.S. Класс полностью определен после ‘}’

P.P.S. Класс может быть объявлен через *класс имя*;

Примеры определений классов

```
struct A {};
```

```
struct B {  
    A a;  
    int b;  
    char c;  
} b1, b2;
```

```
class C {  
    const std::size_t n = 0;  
public:  
    std::size_t value() const { return n; }  
    using X = B;  
};
```

Область видимости класса

```
struct S {  
    struct T {  
        int z = 10 * n;  
        int baz();  
    };  
    int foo(int x) noexcept(n > 100) {  
        return x * n;  
    }  
    void bar(int y = n);  
    int z = 3 * n;  
    static const int n = 101;  
    char str[n];  
};
```

```
int S::T::baz() {  
    return z * 3;  
}  
  
void S::bar(int y) {  
    z *= y;  
}
```

Область видимости класса

```
struct S {  
    struct T {  
        int z = 10 * n;  
        int baz();  
    };  
    int foo(int x) noexcept(n > 100) {  
        return x * n;  
    }  
    void bar(int y = n);  
    int z = 3 * n;  
    static const int n = 101;  
    char str[n];  
};
```

```
int S::T::baz() {  
    return z * 3;  
}
```

```
void S::bar(int y) {  
    z *= y;  
}
```

Область видимости класса

```
struct S {  
    struct T {  
        int z = 10 * n;  
        int baz();  
    };  
    int foo(int x) noexcept(n > 100) {  
        return x * n;  
    }  
    void bar(int y = n);  
    int z = 3 * n;  
    static const int n = 101;  
    char str[n];  
};
```

```
int S::T::baz() {  
    return z * 3;  
}  
  
void S::bar(int y) {  
    z *= y;  
}
```

Область видимости класса

```
struct S {  
    struct T {  
        int z = 10 * n;  
        int baz();  
    };  
    int foo(int x) noexcept(n > 100) {  
        return x * n;  
    }  
    void bar(int y = n);  
    int z = 3 * n;  
    static const int n = 101;  
    char str[n];  
};
```

```
int S::T::baz() {  
    return z * 3;  
}  
  
void S::bar(int y) {  
    z *= y;  
}
```

Область видимости класса

```
struct S {  
    struct T {  
        int z = 10 * n;  
        int baz();  
    };  
    int foo(int x) noexcept(n > 100) {  
        return x * n;  
    }  
    void bar(int y = n);  
    int z = 3 * n;  
    static const int n = 101;  
    char str[n];  
};
```

```
int S::T::baz() {  
    return z * 3;  
}  
  
void S::bar(int y) {  
    z *= y;  
}
```

И еще...

```
struct S {  
    struct T {  
        int z = 10;  
        int baz();  
    };  
  
    T create() const;  
    T another();  
};
```

```
// T S::create() const  
auto S::create() const -> T {  
    return {};  
}  
  
S::T S::another() {  
    return {};  
}
```

И еще...

```
int x = 0;
```

```
struct S {  
    char data[x];    // error
```

```
constexpr std::size_t size() const {  
    return x;    // OK  
}
```

```
static constexpr std::size_t x = 55;  
};
```

Поля и методы

- Область видимости наследников
- Оператор `.` `expr.name`
- Оператор `->` `expr->name`
- Оператор `::` `Class::member`

Поля и методы

- Статические
- Нестатические

```
struct S {  
    static int a;  
    int b;  
  
    static int get_a();  
    int get_b();  
};
```

```
int main() {  
    std::cout << S::a;  
    std::cout << S::get_a();  
    S s;  
    std::cout << s.b;  
    std::cout << s.get_b();  
    std::cout << s.a;  
    std::cout << s.get_a();  
}
```

Статические поля

```
struct S {  
    static int a;  
    static const int b, c = 5;  
    static constexpr int d = 1; // inline  
    static S s;                  // incomplete type  
    static thread_local S ss;  
    static const auto e = sizeof(int); // declaration only  
    static const auto f = 111;  
  
    static S& instance(); // declaration  
};
```

Статические поля

```
int S::a;  
const int S::b = -1, S::c;  
S S::s;  
thread_local S S::ss;  
const int S::f;
```

```
// definition  
S& S::instance() {  
    return ss;  
}
```

```
char xx[S::e];  
// const void * pp = &S::e;  
const void* pp = &S::f;
```

Нестатические поля и методы

Поля:

- Являются подобъектами объекта класса
- Тип должен быть полностью определён в точке объявления
- Не могут быть *auto*, *extern*, *thread_local*
- Размер ≥ 1
- Неотделимы от объекта класса
- Инициализация – в объявлении или в конструкторе

Методы:

- При вызове имеют доступ к полям конкретного объекта класса
- Объект класса доступен через *this*
- Могут иметь квалификаторы вызова
- Специальные методы – конструкторы, деструкторы, операторы присваивания

Примеры

```
struct S {
    const int a = 10;
    int b      = 5;
    char str[6] = "Hello";

    void plus_one() { ++b; }
    void alt_plus_one() { ++this->b; }
    constexpr std::size_t length() const { return sizeof(str) / sizeof(char); }
};

int main() {
    S s1, s2;
    std::cout << "1: " << s1.a << ", " << s1.b << ", " << s1.str << std::endl;
    std::cout << "2: " << s2.a << ", " << s2.b << ", " << s2.str << std::endl;
    s1.plus_one();
    s1.plus_one();
    --s2.b;
    s2.str[1] = 'X';
    std::cout << "1: " << s1.a << ", " << s1.b << ", " << s1.str << std::endl;
    std::cout << "2: " << s2.a << ", " << s2.b << ", " << s2.str << std::endl;
    constexpr S s3;
    std::cout << s3.length() << std::endl;
    // s1.a++;
    // s3.plus_one();
}
```

Отображение синтаксиса операторов

Определение операторов, как членов класса – это частный случай перегрузки операторов.

Синтаксис оператора	Синтаксис перегруженной функции
$@a$	$(a).operator@ \quad ()$
$a@$	$(a).operator@ \quad (0)$
$a@b$	$(a).operator@ \quad (b)$
$a(b\dots)$	$(a).operator() \quad (b\dots)$
$a[b]$	$(a).operator[] \quad (b)$
$a->$	$(a).operator-> \quad ()$

Операторы классов

```
class C {  
public:  
    C& operator++() {  
        // some custom increment logic  
    }  
    C& operator&=(const C& rhs) { return *this; }  
};
```

```
C c, cc;  
++c;  
c &= ++cc;
```

Вложенные классы

- Область видимости включает имена окружающего класса
- Как член класса имеет полные права доступа к другим его членам

```
struct S {  
    public:  
        class X {  
            public:  
                int f() { return S::f; }  
        };  
  
    private:  
        static constexpr int f = 0;  
};  
  
int main() {  
    S::X x;  
    return x.f();  
}
```

Наследование

```
struct A {};  
struct B {  
    int a, b;  
};  
struct C: A, B {  
    double a, c;  
};
```

```
C c;  
int x    = c.b;  
double y = c.c;  
auto z   = c.a;  
  
auto z2  = c.B::a;
```

Права доступа к членам класса

- *public* – публичный доступ, нет ограничений
- *private* – закрытый доступ, только другие члены этого класса (или его друзья) имеют доступ к этому имени
- *protected* – защищенный доступ, подобно закрытому, но права доступа есть также у наследников класса

Права доступа к членам класса

```
class C {  
    private:  
        int f(int) { return 10; }  
  
    public:  
        int f(double) { return -10; }  
};  
  
int main() {  
    C c;  
    return c.f(1.0); // Ok  
    // return c.f(1); // error  
}
```

Назначение прав доступа

- Явно в теле класса – действует на все последующие члены до следующего явного указания спецификатора доступа
- При наследовании – действует на все члены родительского класса

```
class C {  
public:    // inside of class definition  
};
```

```
class CC: protected C    // in inheritance list  
{  
};
```

Права доступа и наследование

- *public* – спецификаторы доступа родительского класса наследуются без изменений
- *protected* – *public* и *protected* члены родительского класса считаются *protected* у наследника, *private* не меняется
- *private* – *public* и *protected* члены родительского класса наследуются, как *private*

Права доступа и вложенные классы

- Вложенный класс имеет такой же доступ, как и любой член класса
- Внешний класс не имеет доступа к приватным полям вложенного

Различие между структурами и классами

- *struct* – по умолчанию, *public* права доступа к членам, *public* наследование
- *class* – по умолчанию, *private* права доступа к членам, *private* наследование

Агрегатная инициализация

Агрегатная инициализация возможна для агрегатов:

- Массив
- Класс с только публичными нестатическими полями, без конструкторов, без виртуальных методов, все базовые классы должны удовлетворять тем же требованиям

Эффект агрегатной инициализации – каждый подобъект объекта-агрегата инициализируется копией значения из списка инициализации.

```
C obj = {a, b, c, ...};
```

```
C obj{a, b, c, ...};
```

```
{a, b, c, ...}; // temporary, type must be implied by context
```

Пример

```
struct A {  
    int a_a;  
    int a_b;  
};
```

```
struct B: A {  
    char b_a;  
    char b_b;  
};
```

```
struct C: A, B {  
    std::string c_a;  
};
```

```
C c{{20, 30}, {{-20, -30}, '\0', 'X'}, "Hello"};
```

Enum

Классические открытые (unscoped) *enum*

- *enum* [имя] { enumerator [= constexpr], ... }
 - *enum* имя : тип { enumerator [= constexpr], ... }
 - *enum* имя : тип;
-
- имя – НОВЫЙ ТИП
 - тип – базовый тип, по умолчанию – не шире int
 - enumerator – константа в той же области видимости
 - const expr – значение, которым инициализируется константа
 - неявное приведение type(enumerator) → I

Строгие (scoped) *enum*

- *enum class* | *struct* имя { enumerator [= constexpr], ... }
 - *enum class* | *struct* имя : тип { enumerator [= constexpr], ... }
 - *enum class* | *struct* имя ;
 - *enum class* | *struct* имя : тип;
-
- имя – НОВЫЙ ТИП
 - тип – базовый тип, по умолчанию – int
 - enumerator – константа в области видимости enum
 - constexpr – значение, которым инициализируется константа

enum и приведения типов

- Неявное приведение открытых *enum* к интегральным типам
- Явное приведение интегральных, floating типов и других *enum* к любому *enum*
- Если базовый тип не задан, а преобразуемое значение не представимо в выбранном по умолчанию – UB

```
enum E1 { A = 5, B, C };  
enum class E2 { A, B };
```

```
E1 e1 = static_cast<E1>(1000);  
E2 e2 = static_cast<E2>(B);
```

```
int main() {  
    // return E2::B;  
    return B;  
}
```

Преимущества строгих епит над классическими

Имена элементов строгого епит не засоряют область видимости – важно для епит, объявляемых в пространстве имен.

Невозможно случайно получить неявное преобразование к совершенно другому типу.

Полные и неполные ТИПЫ

Полные и неполные типы

Полный тип – определение известно.

Неполный тип:

- Предварительное объявление класса
- Определение класса до закрывающей скобки
- *enum* до момента определения его типа реализации
- *void*
- ...

Необходимость полного типа

Требуется знать полный тип T к моменту:

- Вызов функции, возвращающей T
- Объявление переменной типа T
- Объявление нестатического поля типа T
- Явное или неявное преобразование к T
- Обращение к членам класса типа T
- Использование T как родителя определяемого класса
- ...

Примеры

```
class C;

void f(const C *); // OK
C g();             // OK

auto c = g();      // error
auto x = C::get(); // error
C cc;              // error

struct S {
    C &c; // OK;
    C cc; // error
};

enum E {
    A,
    B = sizeof(E), // error
    C
};

enum class EE {
    A,
    B = sizeof(EE), // OK
    C
};
```

Opaque enum declaration

```
enum A : int;
```

```
enum class B;
```

```
enum class C : unsigned;
```

```
A get_a(unsigned long x) {  
    return static_cast<A>(x); // potential UB  
}
```

```
B a_to_b(const A a) {  
    return static_cast<B>(a); // OK  
}
```

Некоторые классы стандартной библиотеки

Некоторые классы стандартной библиотеки

- `std::string`
- `std::array`
- `std::vector`
- `std::pair`
- `std::tuple`
- `std::map`
- `std::unordered_map`
- `std::list`
- ...

<https://en.cppreference.com/w/cpp/container>

Structured bindings

```
std::tuple<A, B, C> foo();
```

```
auto [a, b, c] = foo();
```

```
a      = A();
```

```
const auto x  = b;
```

```
void bar(std::vector<std::pair<int, std::string>>& v) {
```

```
    for (auto& [n, s] : v) {
```

```
        n = s.size();
```

```
        s += '\n';
```

```
    }
```

```
}
```