Программирование на языке C++

Вводный курс

Александр Морозов
gelu.speculum@gmail.com

ИТМО, весенний семестр 2021

Содержание

Структуры и классы

Enum

Полные и неполные типы

Некоторые классы стандартной библиотеки

Структуры и классы

- задаёт новый тип в программе
- может иметь члены:
  - поля данных
  - методы (функции)
  - типы
  - псевдонимы типов
  - члены вложенных нестрогих `enum`
  - шаблоны
- можно наследовать от других классов

Определение

- *класс имя* { *члены* }
- *класс имя* : *список наследования* { *члены* }

где

- *класс* – это ключевое слово `class` или `struct`
- *имя* – идентификатор
- *члены* – объявление членов класса
- *список наследования* – *base1, ..., baseN*

Тип класса полностью определён *после* закрывающей скобки.

Примеры определений классов

```
1       struct A {};
2
3       struct B {
4         A a;
5         int b;
6         char c;
7       } b1, b2;
8
9       class C {
10        const std::size_t n = 0;
11      public:
12        std::size_t value() const
13        { return n; }
14        using X = B;
15      };
16
17      int main() {
18        A a;
19        C c = C();
20        std::cout << sizeof(a) << ":" << sizeof(b1) << ":" << sizeof(c) <<
              ↪  std::endl;
21        std::cout << C().value() << std::endl;
22        C::X x;
23        std::cout << x.b << std::endl;
24      }
```

Область видимости класса

```
1   struct S {
2     struct T {
3       int z = 10 * n;
4       int baz();
5     };
6     int foo(int x) noexcept(n > 100)
7     { return x * n; }
8     void bar(int y = n);
9     int z = 3 * n;
10    static const int n = 101;
11    char str[n];
12  };
13  int S::T::baz()
14  { return z * 3; }
15  void S::bar(int y)
16  { z *= y; }
```

Область видимости класса

```
1   struct S {
2     struct T {
3       int z = 10 * n;
4       int baz();
5     };
6     int foo(int x) noexcept(n > 100)
7     { return x * n; }
8     void bar(int y = n);
9     int z = 3 * n;
10    static const int n = 101;
11    char str[n];
12  };
13  int S::T::baz()
14  { return z * 3; }
15  void S::bar(int y)
16  { z *= y; }
```

Область видимости класса

```
1   struct S {
2     struct T {
3       int z = 10 * n;
4       int baz();
5     };
6     int foo(int x) noexcept(n > 100)
7     { return x * n; }
8     void bar(int y = n);
9     int z = 3 * n;
10    static const int n = 101;
11    char str[n];
12  };
13  int S::T::baz()
14  { return z * 3; }
15  void S::bar(int y)
16  { z *= y; }
```

Область видимости класса

```
1   struct S {
2     struct T {
3       int z = 10 * n;
4       int baz();
5     };
6     int foo(int x) noexcept(n > 100)
7     { return x * n; }
8     void bar(int y = n);
9     int z = 3 * n;
10    static const int n = 101;
11    char str[n];
12  };
13  int S::T::baz()
14  { return z * 3; }
15  void S::bar(int y)
16  { z *= y; }
```

Область видимости класса

```
1   struct S {
2     struct T {
3       int z = 10 * n;
4       int baz();
5     };
6     int foo(int x) noexcept(n > 100)
7     { return x * n; }
8     void bar(int y = n);
9     int z = 3 * n;
10    static const int n = 101;
11    char str[n];
12  };
13  int S::T::baz()
14  { return z * 3; }
15  void S::bar(int y)
16  { z *= y; }
```

И ещё немного об области видимости класса

```
1       struct S {
2         struct T {
3           int z = 10 * n;
4           int baz();
5         };
6
7         T create() const;
8         T another();
9       };
10
11      //T S::create() const
12      auto S::create() const
13          -> T
14      { return {}; }
15
16      S::T S::another()
17      { return {}; }
```

Вторжение в область видимости класса

```
1   int x = 0;
2
3   struct S
4   {
5     char data[x]; // error
6
7     constexpr std::size_t size() const
8     {
9       return x; // OK
10    }
11
12    static constexpr std::size_t x = 55;
13  };
```

Использование имён членов вне области видимости класса

- область видимости наследников
- оператор `.` : `expr.name`
  где *expr* имеет тип:
  - класса
  - потомка
- оператор `->` : `expr->name`
  где *expr* имеет тип:
  - указатель на класс
  - указатель на потомок
- оператор `::` : `Class::member`
  где *Class*:
  - имя класса
  - имя потомка

Поля и методы

- статические
- нестатические

```
1   struct S
2   {
3     static int a;
4     int b;
5
6     static int get_a();
7     int get_b();
8   };
9
10  int main()
11  {
12    std::cout << S::a;
13    std::cout << S::get_a();
14    S s;
15    std::cout << s.b;
16    std::cout << s.get_b();
17    std::cout << s.a;
18    std::cout << s.get_a();
19  }
```

Статические поля и методы

```
1   struct S
2   {
3     static int a;
4     static const int b, c = 5;
5     static constexpr int d = 1; // inline
6     static S s; // incomplete type
7     static thread_local S ss;
8     static const auto e = sizeof(int); // declaration only
9     static const auto f = 111;
10
11    static S & instance(); // declaration
12  };
13  int S::a;
14  const int S::b = -1, S::c;
15  S S::s;
16  thread_local S S::ss;
17  const int S::f;
18
19  // definition
20  S & S::instance()
21  { return ss; }
22
23  char xx[S::e];
24  // const void * pp = &S::e;
25  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 нестатических полей и методов

```
 1  struct S
 2  {
 3    const int a = 10;
 4    int b = 5;
 5    char str[6] = "Hello";
 6
 7    void plus_one()
 8    { ++b; }
 9    void alt_plus_one()
10    { ++this->b; }
11    constexpr std::size_t length() const
12    { return sizeof(str) / sizeof(char); }
13  };
14
15  int main()
16  {
17    S s1, s2;
18    std::cout << "1:␣" << s1.a << ",␣" << s1.b << ",␣" << s1.str << std::endl;
19    std::cout << "2:␣" << s2.a << ",␣" << s2.b << ",␣" << s2.str << std::endl;
20    s1.plus_one();
21    s1.plus_one();
22    --s2.b;
23    s2.str[1] = 'X';
24    std::cout << "1:␣" << s1.a << ",␣" << s1.b << ",␣" << s1.str << std::endl;
25    std::cout << "2:␣" << s2.a << ",␣" << s2.b << ",␣" << s2.str << std::endl;
26    constexpr S s3;
27    std::cout << s3.length() << std::endl;
28    // s1.a++;
29    // s3.plus_one();
30  }
```

Операторы классов

```
1   class C
2   {
3   public:
4     C & operator++ ()
5     {
6       // some custom increment logic
7     }
8     C & operator &= (const C & rhs)
9     {
10      return *this;
11    }
12  };
13
14  C c, cc;
15  ++c;
16  c &= ++cc;
```

Отображение синтаксиса операторов

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

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

Вложенные классы

```
1   struct S {
2     class C {
3       int f();
4     };
5     class CC;
6   };
7   int S::C::f() {}
8
9   S::C c;
10  int n = c.f();
11
12  class S::CC {};
```

- область видимости включает имена окружающего класса
- как член класса имеет полные права доступа к другим его членам

Наследование

```
1   struct A {};
2   struct B
3   {
4     int a, b;
5   };
6   struct C : A, B
7   {
8     double a, c;
9   };
10
11  C c;
12  int x = c.b;
13  double y = c.c;
14  auto z = c.a;
15
16  static_assert(sizeof(C) >= sizeof(A) + sizeof(B));
```

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

```
1   class C
2   {
3   private:
4     int f(int)
5     { return 10; }
6
7   public:
8     int f(double)
9     { return -10; }
10  };
11
12  int main()
13  {
14    C c;
15    return c.f(1); // error
16  }
```

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

```
1   class C
2   {
3     public: // inside of class definition
4   };
5
6   class CC : protected C // in inheritance list
7   {
8   };
```

Пример назначения прав доступа

```
1   #include <iostream>
2
3   class A
4   {
5   public:
6     std::size_t size() const
7     { return sizeof(n); }
8   protected:
9     int next()
10    { return ++n; }
11  private:
12    int n = 0;
13  };
14
15  class B : public A
16  {
17  public:
18    void next()
19    { A::next(); }
20  };
21
22  int main()
23  {
24    A a;
25    B b;
26    std::cout << a.size() << std::endl;
27    // std::cout << a.next() << std::endl;
28    // std::cout << a.n << std::endl;
29    b.next();
30    std::cout << b.size() << std::endl;
31  }
```

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

```
1   class Outer
2   {
3   public:
4     class Inner
5     {
6     public:
7       static int get_a()
8       { return a; }
9       int get_b(const Outer & outer) const
10      { return outer.b; }
11    private:
12      int c = 22;
13    };
14
15    Inner get_inner() const
16    {
17      Inner i;
18      // b = i.c;
19      return i;
20    }
21  private:
22    static const int a = 10;
23    int b = -5;
24  };
```

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

```
1  struct S : A // public inheritance implied
2  {
3    int x; // public access implied
4  };
5
6  class C : B // private inheritance implied
7  {
8    int y; // private access implied
9  };
```

Агрегатная инициализация

```
1    T obj = { a, b, c, ... };
2    T obj{a, b, c, ... };
3    {a, b, c, ...}; // temporary, type must be implied by
         ↪ context
```

Агрегатная инициализация возможно 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.

Пример использования агрегатной инициализации

```
1   struct A {
2     int a_a;
3     int a_b;
4   };
5
6   struct B : A {
7     char b_a;
8     char b_b;
9   };
10
11  struct C : A, B {
12    std::string c_a;
13  };
14
15  C c { 20, 30, -20, -30, '\0', 'X', "Hello" };
```

Содержание

Структуры и классы

Enum

Полные и неполные типы

Некоторые классы стандартной библиотеки

enum

```
1   enum Side {
2     Buy,
3     Sell
4   };
5
6   char encode_side(const Side side) {
7     switch (side) {
8       case Buy: return '1';
9       case Sell: return '2';
10    }
11  }
```

Классические открытые (unscoped) enum

- `enum` [*имя*] { *enumerator* [= *const expr*], ... }
- `enum` *имя* : *тип* { *enumerator* [= *const expr*], ... }
- `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*) $\to \mathbb{I}$

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`
- *const expr* – 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**

```
1   enum E1 { A = 5, B, C };
2   enum class E2 { A, B };
3
4   E1 e1 = static_cast<E1>(1000);
5   E2 e2 = static_cast<E2>(B);
6
7   int main()
8   {
9     // return E2::B;
10    return B;
    }
```

Преимущества строгих enum над классическими

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

Невозможно случайно получить неявное преобразование к совершенно другому типу.

```
1   enum Side { Buy, Sell };
2   enum TimeInForce { Day, GTD, IOC };
3
4   void foo(const Side side, const TimeInForce tif) {
5     if (side == Day) { // obviously, an error, but
          ↪ compiles happily
6       ...
7     }
8   }
```

Содержание

Структуры и классы

Enum

Полные и неполные типы

Некоторые классы стандартной библиотеки

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 объявляемого класса
- ...

Примеры неполных типов

```
1       class C;
2
3       void f(const C *); // OK
4       C g(); // OK
5
6       auto c = g(); // error
7       auto x = C::get(); // error
8       C cc; // error
9
10      struct S
11      {
12        C & c; // OK;
13        C cc; // error
14      };
15
16      enum E {
17        A,
18        B = sizeof(E), // error
19        C
20      };
21
22      enum class EE {
23        A,
24        B = sizeof(EE), // OK
25        C
26      };
```

Opaque enum declaration

```
1       enum A : int;
2
3       enum class B;
4
5       enum class C : unsigned;
6
7       A get_a(unsigned long x)
8       {
9         return static_cast<A>(x); // potential UB
10      }
11
12      B a_to_b(const A a)
13      {
14        return static_cast<B>(a); // OK
15      }
```

Содержание

Структуры и классы

Enum

Полные и неполные типы

Некоторые классы стандартной библиотеки

std::string

https://en.cppreference.com/w/cpp/string/basic_string

```
1   #include <string>
2
3   std::string s1, s2 = "Hello", s3 = s2;
4
5   void foo(const std::string & str) {
6     if (!str.empty()) {
7       const char * s = str.c_str();
8     }
9     if (str.size() >= 5) {
10      char c = str[4];
11    }
12    if (std::size_t i = str.find("ll"); i != str.npos) {
13      const auto ss = str.substr(i, 3);
14    }
15  }
```

std::array

https://en.cppreference.com/w/cpp/container/array

```
1   #include <array>
2
3   std::array<int, 5> x = {1, 2, 3, 4, 5};
4   std::array<int, 10> y = x; // compilation error
5
6   static_assert(!x.empty());
7   static_assert(x.size() == 5);
8
9   x[2] = 22;
```

std::vector

https://en.cppreference.com/w/cpp/container/vector

```
1  #include <vector>
2
3  std::vector<unsigned> x(10, 0xAE), y(100, 0x11);
4  y = x;
5
6  std::vector<int> z(5, -1);
7  z = x; // compilation error
8
9  x.push_back(1111);
```

Некоторые варианты использования `std::vector`

```
1   std::vector<std::string> strings;
2   strings.reserve(100);
3   for (std::string line; std::getline(std::cin, line); ) {
4     strings.push_back(line);
5   }
6
7   std::cout << "Read␣" << strings.size() << "␣input␣lines"
        ↪  << std::endl;
8
9   strings.clear();
10  assert(strings.size() == 0);
11  assert(strings.size() != strings.capacity());
12
13  strings.shrink_to_fit();
14  assert(strings.size() == strings.capacity()); // maybe
```

std::pair

https://en.cppreference.com/w/cpp/utility/pair

```
1   #include <utility>
2
3   std::pair<int, std::string> x(-10, "Cat");
4   std::cout << x.first << "␣:␣" << x.second << std::endl;
5
6   const auto y = std::make_pair(99, "Dog");
7
8   const auto z = std::make_pair<unsigned, std::string>(99,
        ↪ "Explicit␣dog");
9
10  std::pair<A, B> foo() {
11    A a;
12    B b;
13    return {a, b};
14  }
```

std::tuple

https://en.cppreference.com/w/cpp/utility/tuple

```
1   #include <tuple>
2
3   auto many()
4   {
5     return std::make_tuple(1, 2, 'a', std::string{"str"});
6   }
7
8   std::tuple<int, char, double> alt()
9   {
10    return {1, 'a', 0.1};
11  }
12
13  const auto x = many();
14  std::cout << std::get<0>(x)
15        << ",␣" << std::get<1>(x)
16        << ",␣" << std::get<2>(x)
17        << ",␣" << std::get<3>(x) << std::endl;
18
19  const auto y = std::tuple_cat(x, x, x);
20  static_assert(std::tuple_size_v<decltype(y)> == 12);
```

Structured bindings

```
1   std::tuple<A, B, C> foo();
2
3   auto [a, b, c] = foo();
4   a = A();
5   const auto x = b;
6
7   void bar(std::vector<std::pair<int, std::string>> & v) {
8     for (auto & [n, s] : v) {
9       n = s.size();
10      s += '\n';
11    }
12  }
```