Без темы

М. А. Ложников

21 октября 2021 г.
Ревизия: 0

1 Статические члены классов

Поле в классе можно сделать общим для всех объектов этого класса. Это означает, что если поменять значение этого поля в объекте одного класса, то оно поменяется и во всех остальных объектах данного класса (как тех, которые есть в данный момент, так и в тех, которые будут созданы). Такие поля называют статическими. Методы класса также могут быть статическими, такие методы не привязаны к какому-либо объекту, поэтому они могут обращаться напрямую только к статическим полям и методам этого класса. В статических методах недоступно ключевое слово `this`.

1.1 Пример класса с одним статическим полем

Предположим, что в заголовочном файле `static.hpp` объявлен класс, содержащий одно статическое поле.

```
#ifndef STATIC_HPP
#define STATIC_HPP

/* Класс с одним статическим полем типа int. */
class StaticValue {
 public:
  /* Для объявления статического поля или метода используется ключевое поле static. */
  static int value;
};

#endif // STATIC_HPP
```

Listing 1: Файл static.hpp

Теперь перейдём к примеру использования этого класса. К статическим члена класса можно обращаться через имя класса и два двоеточия или через объект и точку (указатель и стрелку).

```
#include <iostream>
#include "static.hpp"

/* Статическое поле класса обязательно должно быть объявлено ровно в одном файле
   исходного кода в программе. Если этого не сделать, то будет ошибка при линковке.
   В современных стандартах языка C++ из этого правила есть исключения, например,
   если поле является константным. */
int StaticValue::value = 5;

```

```cpp
int main() {
  StaticValue sv1;
  StaticValue sv2;

  /* Обращение к статическому полю через класс. */
  std::cout << "Value = " << StaticValue::value << std::endl; // Выведет 5.
  /* Обращение к статическому полю через объект. */
  std::cout << "sv1.value = " << sv1.value << std::endl;      // Выведет 5.

  StaticValue::value = 8;

  std::cout << "Value = " << StaticValue::value << std::endl; // Выведет 8.
  std::cout << "sv1.value = " << sv1.value << std::endl;      // Выведет 8.

  sv1.value = 10;

  std::cout << "Value = " << StaticValue::value << std::endl; // Выведет 10.
  std::cout << "sv1.value = " << sv1.value << std::endl;      // Выведет 10.
  std::cout << "sv2.value = " << sv2.value << std::endl;      // Выведет 10.

  return 0;
}
```

Listing 2: Файл static.cpp

1.2 Пример класса со статическим методом

```cpp
#include <iostream>

/* Число вида a / b */
class Rational {
private:
  int numerator;
  int denominator;

public:
  double Result() const {
    /* В C++ для приведения numerator к типу double пишут double(numerator).
       Приведение в стиле языка C ((double) numerator) не используют. */
    return double(numerator) / denominator;
  }

  /* Статический метод класса не привязан ни к какому объекту, поэтому он не может
     обращаться напрямую к нестатическим членам класса без указания объекта, которому они
     принадлежат. */
  static Rational FromNumbers(int numeratorIn, int denominatorIn) {
    Rational result;

    result.numerator = numeratorIn;
    result.denominator = denominatorIn;

```

```
    return result;
  }
};

int main() {
  Rational res;

  res = Rational::FromNumbers(1, 2);

  Rational res2;

  res2 = res.FromNumbers(3, 4);

  std::cout << "res = " << res.Result() << std::endl;
  std::cout << "res2 = " << res2.Result() << std::endl;

  return 0;
}
```

Замечание 1.1 *Вместо метода* `Result()` *можно было бы объявить операцию приведения* `Rational` *к типу* `double`.

```
class Rational {

/* Часть пропущена. */

 public:
  operator double() const {
    return double(numerator) / denominator.
  }
};
```

Тогда сработал бы следующий код:

```
  std::cout << "res = " << double(res) << std::endl;
  std::cout << "res2 = " << double(res2) << std::endl;
```

2 Дружественные функции

Для того, чтобы функция могла обращаться напрямую к приватным полям класса, её можно объявить дружественной к этому классу. Для этого нужно объявить прототип функции внутри класса, поставив перед ним ключевое слово `friend`. Рассмотрим пример.

```
#include <iostream>

class Complex {
 private:
  double re;
  double im;
```

```
 public:
  Complex(double re = 0, double im = 0) :
      re(re),
      im(im)
  { }

  /* Эта функция не является членом класса, но она может обращаться к его
     приватным полям. */
  friend void PrintComplex(const Complex& c);
};

void PrintComplex(const Complex& c) {
  /* Здесь происходит обращение к приватным полям re и im класса Complex. Если бы
     функция PrintComplex() не была бы объявлена дружественной к классу Complex,
     то была бы ошибка компиляции. */
  std::cout << "(" << c.re << ", "
       << c.im << ")" << std::endl;
}

int main() {
  Complex c(10, 3.14);

  PrintComplex(c);

  return 0;
}
```

Замечание 2.1 *Точно также можно сделать дружественной операцию вывода в поток.*

```
class Complex {

  /* Часть пропущена. */

  friend std::ostream& operator<<(std::ostream& out, const Complex& c);
};
```

В этом случае внешняя функция `operator<<()` сможет обращаться к приватным полям класса `Complex` напрямую.

Замечание 2.2 *Дружественным можно объявить не только функцию, но и класс. Например, если внутри объявления класса `A` написано `friend class B;`, то класс `B` сможет обращаться к приватным полям класса `A` напрямую.*

3 Наследование и исключения

Рассмотрим набор классов для описания различных исключений:

```
// Базовый класс.
class BaseException { };

```

```
// Набор производных классов.
class DerivedExceptionOne : public BaseException { };

class DerivedExceptionTwo : public BaseException { };

class DerivedExceptionThree : public BaseException { };
```

Теперь рассмотрим код, который генерирует и отлавливает исключения указанных типов.

```
  for (int i = 0; i < 5; i++) {
    try {
      // Некоторый код, который выбрасывает исключения.

      if (i == 0)
        throw DerivedExceptionOne();
      else if (i == 1)
        throw DerivedExceptionTwo();
      else if (i == 2)
        throw DerivedExceptionThree();
      else if (i == 3)
        throw BaseException();
      else if (i == 4)
        throw 1;

    } catch (DerivedExceptionOne& ) {
      std::cout << "Exception DerivedExceptionOne has been caught!" << std::endl;
    } catch (DerivedExceptionTwo& ) {
      std::cout << "Exception DerivedExceptionTwo has been caught!" << std::endl;
    } catch (BaseException& ) {
      std::cout << "Exception BaseException has been caught!" << std::endl;
    } catch (...) {
      std::cout << "Unknown exception has been caught!" << std::endl;
    }
  }
```

В данном примере исключение типа `DerivedExceptionThree` перехватывается в `catch` блоке для исключения типа `BaseException`. Это происходит по той причине, что `catch` блоки для базовых классов способны перехватывать исключения любых производных классов, для которых не указан обработчик, расположенный выше соответствующего обработчика для данного класса.

Замечание 3.1 *Из этого следует, что обработчики для исключений следует располагать в порядке от частного к общему.*

3.1 Тонкости проброса исключений

Рассмотрим пример из двух классов исключений:

```
// Базовый класс.
class BaseException { };
// Производный класс.
class DerivedException : public BaseException { };
```

Теперь разберём два варианта проброса исключений во внешний обработчик: некорректный и корректный.

```
void Foo(int i) {
  try {
    if (i == 0)
      throw DerivedException();
    else if (i == 1)
      throw BaseException();

  } catch (BaseException& e) {
    // Некорректный проброс исключения. Во внешнем обработчике
    // будет отлавливаться исключение типа BaseException в обоих случаях.
    throw e;
  }
}

void Bar(int i) {
  try {
    if (i == 2)
      throw DerivedException();
    else if (i == 3)
      throw BaseException();

  } catch (BaseException& e) {
    // Корректный проброс исключения.
    throw;
  }
}
```

Основное отличие функций `Foo()` и `Bar()` реализации блока `catch`. В первом случае исключение пробрасывается во внешний обработчик конструкцией `throw e;`, а во втором случае написано `throw;`. Отличие заключается в том, что функция `Foo()` всегда пробрасывает наружу исключение типа `BaseException`, а функция `Bar()` пробросит исключение именно того типа, который был изначально сгенерирован. Убедиться в этом можно на следующем примере:

```
for (int i = 0; i < 4; i++) {
  try {
    Foo(i);
    Bar(i);
  } catch (DerivedException& ) {
    std::cout << "Exception DerivedException has been caught!" << std::endl;
  } catch (BaseException& ) {
    std::cout << "Exception BaseException has been caught!" << std::endl;
  }
}
```

Замечание 3.2 *Исключения, пойманные во внутреннем обработчике, можно пробрасывать наружу только конструкцией `throw;`.*

4 Особенности выделения памяти

Обсудим различия в средствах выделения памяти языков C и C++. С одной стороны, в языке C++ доступны все те же средства выделения памяти, которые были доступны в языке C. Например, функции `malloc()`, `free()`, `realloc()` и т. д. доступны при подключении заголовочного файла `<cstdlib>`. С другой стороны, этими средствами не всегда можно пользоваться. Рассмотрим несколько примеров. Для этого введём класс `Logger`:

```
class Logger {
 public:
  Logger() :
   id() {
    std::cout << "Logger '" << id << "' created!" << std::endl;
  }

  Logger(const std::string& id) :
   id(id) {
    std::cout << "Logger '" << id << "' created!" << std::endl;
  }

  void Print() {
    std::cout << "Logger '" << id << "' printed something!" << std::endl;
  }

  ~Logger() {
    std::cout << "Logger '" << id << "' destroyed!" << std::endl;
  }
 public:
  std::string id;
};
```

Если требуется создать динамически объект данного типа в куче, то правильно использовать следующие конструкции:

```
  // Выделение памяти в куче и создание объекта при помощи конструктора по-умолчанию.
  Logger* logger = new Logger;
  // Выделение памяти в куче и создание объекта при помощи специального конструктора.
  Logger* specialLogger = new Logger("some id");

  logger->Print();
  specialLogger->Print();

  // После использования память необходимо освободить. Оператор delete вызывает деструктор
  // и освобождает память.
  delete logger;
  delete specialLogger;
```

В случае создания нескольких объектов следует использовать операторы `new[]`/`delete[]`.

```
  const int N = 3;
  // Выделение памяти в куче под N объектов и создание N объектов при помощи
```

```
  // конструктора по-умолчанию.
  Logger* loggers = new Logger[N];

  for (int i = 0; i < N; i++)
    loggers[i].Print();

  // После использования память необходимо освободить. Оператор delete[] вызывает
  // деструкторы каждого созданного объекта и освобождает память.
  delete[] loggers;
```

Замечание 4.1 *Операторы `new`/`delete` и операторы `new[]`/`delete[]` не совместимы друг с другом. Если был использован оператор `new`, то соответствующую память следует освобождать оператором `delete`. Наоборот, если же был использован оператор `new[]`, то соответствующую память следует освобождать оператором `delete[]`.*

Функции языка C `malloc()`/`free()` точно так же как и операторы `new[]`/`delete[]` выделяют и освобождают память в куче. Однако, функции языка C **не умеют** вызывать конструктор. Таким образом, они не создают сами объекты. Без вызова конструктора поле `id` класса `Logger` осталось бы неинициализированным. В некоторых случаях выделение памяти без вызова конструктора вообще может привести к ошибке при работе с памятью, например, при работе с полиморфными типами.

```
class Base {
 public:
  virtual void Print() const {
    std::cout << "Base" << std::endl;
  }

  virtual ~Base() { }
};

class Derived : public Base {
 public:
  void Print() const override {
    std::cout << "Derived" << std::endl;
  }
};
```

В данном примере использование функций языка C для выделения памяти приведёт к ошибке, поскольку полиморфные объекты содержат внутри себя так называемую таблицу виртуальных функций, которая требуется для корректного полиморфного поведения виртуальных функций. Эта таблица инициализируется оператором `new`/`new[]`.

Замечание 4.2 *Поскольку средства выделения/освобождения памяти в языке C не умеют вызывать конструкторы/деструкторы, то их следует использовать только для элементарных типов данных или для простых структур данных, которые не требуют вызова конструктора. Иными словами, поля этих структур могут быть либо элементарными типами данных либо такими же простыми структурами.*

Замечание 4.3 *В языке C++ есть вариант операратора `new`, называемый **placement new**, который позволяет создать объект в заранее выделенной сырой памяти.*

Список литературы

[1] *Бьерн Страуструп* Язык программирования C++. М:Бином. 2011.

[2] `https://en.cppreference.com/w/cpp/language/new`