Лекция 15. Полиморфизм

Язык C++ позволяет определять функции с одним и тем же именем, но разным набором параметров. Подобная возможность называется **перегрузкой функций** (**function overloading**). Компилятор же на этапе компиляции на основании параметров выберет нужный тип функции.

Чтобы определить несколько различных версий функции с одним и тем же именем, все эти версии должны отличаться как минимум по одному из следующих признаков:

- имеют разное количество параметров
- соответствующие параметры имеют разный тип

При этом различные версии функции могут также отличаться по возвращаемому типу. Однако компилятор, когда выбирает, какую версию функции использовать, ориентируется именно на количество параметров и их тип.

Рассмотрим простейший пример:

```
1    #include <iostream>
2
3    int sum(int, int);
4    double sum(double, double);
5
6    int main()
7    {
8        int result1 {sum(3, 6)}; // выбирается версия int sum(int, int)
9        std::cout << result1 << std::endl; // 9
10
11
12       double result2 {sum(3.3, 6.6)}; // выбирается версия double sum(double, doubl
13       std::cout << result2 << std::endl; // 9.9
14   }
15   int sum(int a, int b)
16   {
17       return a + b;
18   }
```

```
19  double sum(double a, double b)
20  {
21      return a + b;
22  }
```

Здесь определены две версии функция sum, которая складывает два числа. В одном случае она складывает два числа типа int, в другом - числа типа double. При вызове функций компилятор на основании переданных аргументов определяет, какую версию использовать. Например, при первом вызове передаются числа int:

```
1  int result1 {sum(3, 6)};
```

Соответственно для этого вызова выбирается версия

```
1  int sum(int, int);
```

Во втором вызове в функцию передаются числа с плавающей точкой:

```
1  double result2 {sum(3.3, 6.6)};
```

Поэтому выбирается версия, которая принимает числа double:

```
1  double sum(double, double);
```

Аналогично перегруженные версии функции могут отличаться по количеству параметров:

```
1   #include <iostream>
2
3   int sum(int, int);
4   int sum(int, int, int);
5
6   int main()
7   {
8       int result1 {sum(3, 6)}; // выбирается версия int sum(int, int)
9       std::cout << result1 << std::endl; // 9
10
```

```
11
12      int result2 {sum(3, 6, 2) }; // выбирается версия int sum(int, int, int)
13      std::cout << result2 << std::endl; // 11
14  }
15  int sum(int a, int b)
16  {
17      return a + b;
18  }
19  int sum(int a, int b, int c)
20  {
21      return a + b + c;
22  }
```

Перегрузка функций и параметрами-ссылки

При перегрузке функций с параметрами-ссылками следует учитывать, что параметры типов `data_type` и `data_type&` не различаются при перегрузке. Например, два следующих прототипа:

```
1  void print(int);
2  void print(int&);
```

Не считаются разными версиями функции print.

Перегрузка и параметры-константы

При перегрузке функций константный параметр отличается от неконстантного параметра только для ссылок и указателей. В остальных случаях константый параметр будет идентичен неконстантному параметру. Например, следующие два прототипа при перегрузке различаться НЕ будут:

```
1  void print(int);
2  void print(const int);
```

Во втором прототипе компилятор игнорирует оператор **const**.

Пример перегрузки функции с константными параметрами

```
1   #include <iostream>
```

```
2
3    int square(const int*);
4    int square(int*);
5
6    int main()
7    {
8        const int n1{2};
9        int n2{3};
10       int square_n1 {square(&n1)};
11       int square_n2 {square(&n2)};
12       std::cout << "square(n1): " << square_n1 << "\tn1: " << n1 << std::endl;
13       std::cout << "square(n2): " << square_n2 << "\tn2: " << n2 << std::endl;
14   }
15
16   int square(const int* num)
17   {
18       return *num * *num ;
19   }
20   int square(int* num)
21   {
22       *num = *num * *num; // изменяем значение по адресу в указателе
23       return *num;
24   }
```

Здесь функция square принимает указатель на число и возводит его в квадрат. Но первом случае параметр представляет указатель на константу, а во втором - обычный указатель.

В первом вызове

```
1    int square_n1 {square(&n1)};
```

Компилятор будет использовать версию

```
1    int square(const int*);
```

так как передаваемое число n1 представляет константу. Поэтому переданное в эту функцию число n1 не изменится.

В втором вызове

```
1    int square_n2 {square(&n2)};
```

Компилятор будет использовать версию

```
1    int square(int*);
```

в которой для примера также меняется передаваемое значение. Поэтому переданное в эту функцию число n2 изменит свое значение. Консольный вывод программы

```
square(n1): 4    n1: 2
square(n2): 9    n2: 9
```

С передачей константной ссылки все будет аналогично.

Виртуальные функции и их переопределение

При вызове функции программа должна определять, с какой именно реализацией функции соотносить этот вызов, то есть связать вызов функции с самой функцией. В C++ есть два типа связывания - статическое и динамическое.

Когда вызовы функций фиксируются до выполнения программы на этапе компиляции, это называется статическим связыванием (static binding), либо ранним связыванием (early binding). При этом вызов функции через указатель определяется исключительно типом указателя, а не объектом, на который он указывает. Например:

```
1    #include <iostream>
2
3    class Person
4    {
5    public:
6        Person(std::string name): name{name}
7        { }
8        void print() const
```

```
9         {
10            std::cout << "Name: " << name << std::endl;
11        }
12    private:
13        std::string name;       //  имя
14    };
15    class Employee: public Person
16    {
17    public:
18        Employee(std::string name, std::string company): Person{name}, company{compa
19        { }
20        void print() const
21        {
22            Person::print();
23            std::cout << "Works in " << company << std::endl;
24        }
25    private:
26        std::string company;    // компания
27    };
28
29    int main()
30    {
31        Person tom {"Tom"};
32        Person* person {&tom};
33        person->print();     // Name: Tom
34
35        Employee bob {"Bob", "Microsoft"};
36        person = &bob;
37        person->print();    // Name: Bob
38    }
```

В данном случае класс Employee наследуется от класса Person, но оба этих класса определяют функцию `print()`, которая выводит данные об объекте. В функции main создаем два объекта и поочередно присваиваем их указателю на тип **Person** и вызываем через этот указатель функцию print. Однако даже если этому указателю присваивается адрес объекта Employee, то все равно вызывает реализация функции из класса Person:

```
Employee bob {"Bob", "Microsoft"};
person = &bob;
person->print();    // Name: Bob
```

То есть выбор реализации функции определяется не типом объекта, а типом указателя. Консольный вывод программы:

```
Name: Tom
Name: Bob
```

Динамическое связывание и виртуальные функции

Другой тип связывания представляет динамическое связывание (dynamic binding), еще называют поздним связыванием (late binding), которое позволяет на этапе выполнения решать, функцию какого типа вызвать. Для этого в языке C++ применяют **виртуальные функции**. Для определения виртуальной функции в базовом классе функция определяется с ключевым словом **virtual**. Причем данное ключевое слово можно применить к функции, если она определена внутри класса. А производный класс может **переопределить** ее поведение.

Итак, сделаем функцию print в базовом классе Person виртуальной:

```
#include <iostream>

class Person
{
public:
    Person(std::string name): name{name}
    { }
    virtual void print() const // виртуальная функция
    {
        std::cout << "Name: " << name << std::endl;
    }
```

```
12  private:
13      std::string name;
14  };
15  class Employee: public Person
16  {
17  public:
18      Employee(std::string name, std::string company): Person{name}, company{compa
19      { }
20      void print() const
21      {
22          Person::print();
23          std::cout << "Works in " << company << std::endl;
24      }
25  private:
26      std::string company;
27  };
28
29  int main()
30  {
31      Person tom {"Tom"};
32      Person* person {&tom};
33      person->print();     // Name: Tom
34      Employee bob {"Bob", "Microsoft"};
35      person = &bob;
36      person->print();    // Name: Bob
37                              // Works in Microsoft
38  }
```

Таким образом, базовый класс Person определяет виртуальную функцию print, а производный класс Employee **переопределяет** ее. В первом же примере, где функция `print` не была виртуальной, класс Employee не переопределял, а скрывал ее. Теперь при вызове функции print для объекта Employee через

указатель Person* будет вызываться реализация функции именно класса Employee. Соответственно тепепрь мы получим другой консольный вывод:

```
Name: Tom
Name: Bob
Works in Microsoft
```

В этом и состоит отличие переопределения виртуальных функций от скрытия.

Класс, который определяет или наследует виртуальную функцию, еще назвается **полиморфным** (polymorphic class). То есть в данном случае Person и Employee являются полиморфными классами.

Стоит отметить, что вызов виртуальной функции через имя объекта всегда разрешается статически.

```
Employee bob {"Bob", "Microsoft"};
Person p = bob;
p.print();  // Name: Bob - статическое связывание
```

Динамическое связывание возможно только через указатель или ссылку.

```
Employee bob {"Bob", "Microsoft"};
Person &p {bob};    // присвоение ссылке
p.print();  // динамическое связывание

Person *ptr {&bob};    // присвоение адреса указателю
ptr->print();  // динамическое связывание
```

При определении вирутальных функций есть ряд ограничений. Чтобы функция попадала под динамическое связывание, в производном классе она должна иметь тот же самый набор параметров и возвращаемый тип, что и в базовом классе. Например, если в базовом классе виртуальная функция определена как константная, то в производном классе она тоже должна быть константной. Если же функция имеет разный набор параметров или несоответствие по константности, то мы будем иметь дело со скрытием функций, а не переопределением. И тогда будет применяться статическое связывание.

Также статические функции не могут быть виртуальными.

Ключевое слово override

Чтобы явным образом указать, что мы хотим переопредлить функцию, а не скрыть ее, в производном классе после списка параметров функции указывается слово **override**

```
1   #include <iostream>
2
3   class Person
4   {
5   public:
6       Person(std::string name): name{name}
7       { }
8       virtual void print() const // виртуальная функция
9       {
10          std::cout << "Name: " << name << std::endl;
11      }
12  private:
13      std::string name;
14  };
15  class Employee: public Person
16  {
17  public:
18      Employee(std::string name, std::string company): Person{name}, company{compa
19      { }
20      void print() const override // явным образом указываем, что функция переопреде
21      {
22          Person::print();
23          std::cout << "Works in " << company << std::endl;
24      }
25  private:
26      std::string company;
27  };
28
```

```
int main()
{
    Person tom {"Tom"};
    Person* person {&tom};
    person->print();     // Name: Tom
    Employee bob {"Bob", "Microsoft"};
    person = &bob;
    person->print();     // Name: Bob
                         // Works in Microsoft
}
```

То есть здесь выражение

```
void print() const override
```

указывает, что мы явным образом хотим переопределить функцию print. Однако может возникнуть вопрос: в предыдущем примере мы не указывали `override` для вирутальной функции, но переопределение все равно работало, зачем же тогда нужен `override`? Дело в том, что `override` явным образом указывает компилятору, что это переопределяемая функция. И если она не соответствует виртуальной функции в базовом классе по списку параметров, возвращаемому типу, константности, или в базовом классе вообще нет функции с таким именем, то компилятор при компиляции сгенерирует ошибку. И по ошибке мы увидим, что с нашей переопределенной функцией что-то не так. Если же `override` не указать, то компилятор будет считать, что речь идет о скрытии функции, и никаких ошибок не будет генерировать, компиляция пройдет успешно. Поэтмоу при переопределении виртуальной функции в производном классе лучше указывать слово **override**

При этом стоит отметить, что виртуальную функцию можно переопределить по всей иерархии наследования в том числе не в прямых производных классах.

Принцип выполнения виртуальных функций

Стоит отметить, что виртальные функции имеют свою цены - объекты классов с виртуальными функциями требуют немного больше памяти и немного больше времени для выполнения. Поскольку при создании объекта полиморфного класса (который имеет виртуальные функции) в объекте создается специальный указатель. Этот указатель используется для вызова любой виртуальной функции в объекте. Специальный указатель указывает на таблицу указателей функций, которая создается для класса. Эта таблица, называемая виртуальной таблицей или vtable, содержит по одной записи для каждой виртуальной функции в классе.

Когда функция вызывается через указатель на объект базового класса, происходит следующая последовательность событий

1. Указатель на vtable в объекте используется для поиска адреса vtable для класса.
2. Затем в таблице идет поиск указателя на вызываемую виртуальную функцию.
3. Через найденный указатель функции в vtable вызывается сама функция. В итоге вызов виртуальной функции происходит немного медленнее, чем прямой вызов невиртуальной функции, поэтому каждое объявление и вызов виртуальной функции несет некоторые накладные расходы.

Запрет переопределения

С помощью спецификатора **final** мы можем запретить определение в производных классах функций, которые имеют то же самое имя, возвращаемый тип и список параметров, что и виртуальная функция в базовом классе. Например:

```
1    class Person
2    {
3    public:
4        virtual void print() const final
5        {
6
7        }
8    };
9    class Employee : public Person
10   {
11   public:
12       void print() const override     // Ошибка!!!
13       {
14
15       }
16   };
```

Также можно переопределить функцию базового класса, но запретить ее переопределение в дальнейших производных классах:

```
1    class Person
```

```
{
public:
    virtual void print() const // переопределение разрешено
    {

    }
};
class Employee : public Person
{
public:
    void print() const override final   // в классах, производных от Employee пере
    {

    }
};
```

Иногда возникает необходимость определить класс, который не предполагает создания конкретных объектов. Например, класс фигуры. В реальности есть конкретные фигуры: квадрат, прямоугольник, треугольник, круг и так далее. Однако абстрактной фигуры самой по себе не существует. В то же время может потребоваться определить для всех фигур какой-то общий класс, который будет содержать общую для всех функциональность. И для описания подобных сущностей используются абстрактные классы.

Абстрактные классы - это классы, которые содержат или наследуют без переопределения хотя бы одну чистую виртуальную функцию. Абстрактный класс определяет интерфейс для переопределения производными классами.

Что такое **чистые виртуальные функции** (pure virtual functions)? Это функции, которые не имеют определения. Цель подобных функций - просто определить функционал без реализации, а реализацию определят производные классы. Чтобы определить виртуальную функцию как чистую, ее объявление завершается значением "=0". Например, определим абстрактный класс, который представляет геометрическую фигуру:

```
class Shape
{
public:
    virtual double getSquare() const = 0;      // площадь фигуры
    virtual double getPerimeter() const = 0;   // периметр фигуры
};
```

Класс Shape является абстрактным, потому что он содержит как минимум одну чистую виртуальную функцию. А в данном случае даже две таких функции - для вычисления площади и периметра фигуры. И ни одна из функций не имеет никакой реализации. В данном случае обе функции являются константными, но это необязательно. Гловно, чтобы любой производный класс от Shape должен будет предоставить для этих функций свою реализацию.

При этом мы не можем создать объект абстрактного класса:

```
Shape shape{};
```

Для применения абстрактного класса определим следующую программу:

```
#include <iostream>

class Shape
{
public:
    virtual double getSquare() const = 0;      // площадь фигуры
    virtual double getPerimeter() const = 0;   // периметр фигуры
};
class Rectangle : public Shape   // класс прямоугольника
{
public:
    Rectangle(double w, double h) : width(w), height(h)
    { }
    double getSquare() const override
    {
        return width * height;
    }
    double getPerimeter() const override
    {
        return width * 2 + height * 2;
    }
private:
    double width;    // ширина
    double height;   // высота
};
class Circle : public Shape      // круг
{
public:
    Circle(double r) : radius(r)
    { }
    double getSquare() const override
    {
        return radius * radius * 3.14;
    }
```

```
35      double getPerimeter() const override
36      {
37          return 2 * 3.14 * radius;
38      }
39  private:
40      double radius;  // радиус круга
41  };
42
43  int main()
44  {
45      Rectangle rect{30, 50};
46      Circle circle{30};
47
48      std::cout << "Rectangle square: " << rect.getSquare() << std::endl;
49      std::cout << "Rectangle perimeter: " << rect.getPerimeter() << std::endl;
50      std::cout << "Circle square: " << circle.getSquare() << std::endl;
51      std::cout << "Circle perimeter: " << circle.getPerimeter() << std::endl;
52  }
```

Здесь определены два класса-наследника от абстрактного класса Shape - Rectangle (прямоугольник) и Circle (круг). При создании классов-наследников все они должны либо определить для чистых виртуальных функций конкретную реализацию, либо повторить объявление чистой виртуальной функции. Во втором случае производные классы также будут абстрактными.

В данном же случае и Circle, и Rectangle являются конкретными классами и реализуют все виртуальные функции.

Консольный вывод программы:

```
Rectangle square: 1500
Rectangle perimeter: 160
Circle square: 2826
Circle perimeter: 188.4
```

Стоит отметить, что абстрактный класс может определять и обычные функции и переменные, может иметь несколько конструкторов, но при этом нельзя создавать объекты этого абстрактного класса. Например:

```
1   #include <iostream>
2
3   class Shape
4   {
5   public:
6       Shape(int x, int y): x{x}, y{y}
7       {}
```

```
8         virtual double getSquare() const = 0;      // площадь фигуры
9         virtual double getPerimeter() const = 0;   // периметр фигуры
10        void printCoords() const
11        {
12            std::cout << "X: " << x << "\tY: " << y << std::endl;
13        }
14    private:
15        int x;
16        int y;
17    };
18    class Rectangle : public Shape   // класс прямоугольника
19    {
20    public:
21        Rectangle(int x, int y, double w, double h) : Shape{x, y}, width(w), height(h)
22        { }
23        double getSquare() const override
24        {
25            return width * height;
26        }
27        double getPerimeter() const override
28        {
29            return width * 2 + height * 2;
30        }
31    private:
32        double width;    // ширина
33        double height;   // высота
34    };
35    class Circle : public Shape      // круг
36    {
37    public:
38        Circle(int x, int y, double r) : Shape{x, y}, radius(r)
39        { }
40        double getSquare() const override
41        {
42            return radius * radius * 3.14;
43        }
44        double getPerimeter() const override
45        {
46            return 2 * 3.14 * radius;
47        }
48    private:
49        double radius;   // радиус круга
50    };
51
52    int main()
53    {
```

```
54      Rectangle rect{0, 0, 30, 50};
55      rect.printCoords();     // X: 0    Y: 0
56
57      Circle circle{10, 20, 30};
58      circle.printCoords();   // X: 10   Y: 20
59  }
```

В данном случае класс Shape также имеет две переменных, конструктор, который устанавливает их значения, и невиртуальную функцию, которая выводит их значения. В производных классах также необходимо вызвать этот конструктор. Однако объект абстрактного класса с помощью его конструктора мы создать не можем.

Полиморфизм

Полиморфизм предоставляет возможность объединять различные виды поведения при помощи общей записи, что как правило означает использование функций для обработки данных различных типов; считается одним из трёх столпов объектно-ориентированного программирования. По особенностям реализации полиморфизм можно разделить на ограниченный и неограниченный, динамический и статический.

Понятие ограниченный (bounded) означает, что интерфейсы полностью определены заранее, например, конструкцией базового класса. Неограниченный полиморфизм же не накладывает ограничений на тип, а лишь требует реализацию определённого синтаксиса.

Статический означает, что связывание интерфейсов происходит на этапе компиляции, динамический – на этапе выполнения.

Язык программирования C++ предоставляет *ограниченный динамический* полиморфизм при использовании наследования и виртуальных функций и *неограниченный статический* – при использовании шаблонов. Поэтому в рамках данной статьи данные понятия будут именоваться просто *статический* и *динамический* полиморфизм. Однако, вообще говоря, различные средства в различных ЯП могут предоставлять различные комбинации типов полиморфизма.

Динамический полиморфизм

Динамический полиморфизм – наиболее частое воплощение полиморфизма в целом. В C++ данная возможность реализуется при помощи объявления общих возможностей с использованием функционала виртуальных функций. При этом в объекте класса хранится указатель на таблицу виртуальных методов (vtable), а вызов метода осуществляется путём разыменования указателя и вызова метода, соответствующего типу, с которым был создан объект. Таким образом можно

управлять этими объектами при помощи ссылок или указателей на базовый класс (однако нельзя использовать копирование или перемещение).

Рассмотрим следующий простой пример: пусть есть абстрактный класс Property, который описывает облагаемую налогом собственность с единственным чисто виртуальным методом `getTax`, и полем `worth`, содержащим стоимость; и три класса: `CountryHouse`, `Car`, `Apartment`, которые реализуют данный метод, определяя различную налоговую ставку:
Пример

```
class Property
{
protected:
    double worth;
public:
    Property(double worth) : worth(worth) {}
    virtual double getTax() const = 0;
};
class CountryHouse :
    public Property
{
public:
    CountryHouse(double worth) : Property(worth) {}
    double getTax() const override { return this->worth / 500; }
};

class Car :
    public Property
{
public:
    Car(double worth) : Property(worth) {}
    double getTax() const override { return this->worth / 200; }
};

class Apartment :
    public Property
{
public:
    Apartment(double worth) : Property(worth) {}
    double getTax() const override { return this->worth / 1000; }
};


void printTax(Property const& p)
{
    std::cout << p.getTax() << "\n";
}

// Или так

void printTax(Property const* p)
{
    std::cout << p->getTax() << "\n";
}

int main()
{
    Property* properties[3];
    properties[0] = new Apartment(1'000'000);
```

```
    properties[1] = new Car(400'000);
    properties[2] = new CountryHouse(750'000);

    for (int i = 0; i < 3; i++)
    {
        printTax(properties[i]);
        delete properties[i];
    }

    return 0;
}
```

Если заглянуть “под капот”, то можно увидеть, что компилятор (в моём случае это gcc) неявно добавляет в начало класса Property указатель на vtable, а в конструктор – инициализацию этого указателя в соответствии с нужным типом. А вот так в дизассемблированном коде выглядит фрагмент с вызовом метода getTax():

```
mov     rbp, QWORD PTR [rbx]; В регистр rbp помещаем указатель на объект
mov     rax, QWORD PTR [rbp+0]; В регистр rax помещаем указатель на vtable
call    [QWORD PTR [rax]]; Вызываем функцию, адрес которой лежит по адресу, лежащему
в rax (первое разыменование даёт vtable, второе – адрес функции.
```

Статический полиморфизм

Перейдём, наконец, к самому интересному. В C++ средством статического полиморфизма являются шаблоны. Впрочем, это весьма мощный инструмент, который имеет огромное количество применений, и их подробное рассмотрение и изучение потребует полноценного учебного курса, поэтому в рамках данной статьи мы ограничимся лишь поверхностным рассмотрением.

Перепишем предыдущий пример с использованием шаблонов, заодно в целях демонстрации воспользуемся тем, на этот раз мы используем неограниченный полиморфизм.
Пример

```
class CountryHouse
{
private:
    double worth;
public:
    CountryHouse(double worth) : worth(worth) {}
    double getTax() const { return this->worth / 500; }
};

class Car
{
private:
    double worth;
public:
    Car(double worth) : worth(worth) {}
    double getTax() const { return this->worth / 200; }
};
```

```
class Apartment
{
private:
    unsigned worth;
public:
    Apartment(unsigned worth) : worth(worth) {}
    unsigned getTax() const { return this->worth / 1000; }
};

template <class T>
void printTax(T const& p)
{
    std::cout << p.getTax() << "\n";
}

int main()
{
    Apartment a(1'000'000);
    Car c(400'000);
    CountryHouse ch(750'000);

    printTax(a);
    printTax(c);
    printTax(ch);
    return 0;
}
```

Здесь я заменил возвращаемый тип `Apartment::GetTax()`. Так как, благодаря перегрузке оператора >>, синтаксис (и, в данном случае, семантика) остался корректным, то данный код вполне успешно компилируется, в то время, как аппарат виртуальных функций нам бы такой вольности не простил.

В данном случае, как и положено при использовании шаблонов, компилятор инстанцировал (то есть создал из шаблона путём подстановки параметров) три различных функции и подставил нужную на этапе компиляции – поэтому полиморфизм на основе шаблонов и является статическим.

Как я уже отмечал во введении, хорошим примером использования статического полиморфизма может послужить STL. Так, например, выглядит простая реализация функции `std::for_each`:

```
template<class InputIt, class UnaryFunc>
constexpr UnaryFunc for_each(InputIt first, InputIt last, UnaryFunc f)
{
    for (; first != last; ++first)
        f(*first);

    return f;
}
```

При вызове функции нам необходимо лишь предоставить объекты, для которых будет корректен синтаксис имеющихся в теле функции операций (плюс, в виду того, что параметры передаются, а результат возвращается, по значению для них должен быть определён конструктор копирования (перемещения)). Однако

следует понимать, что шаблон лишь задаёт синтаксис, поэтому несоответствия между принятым синтаксисом и семантикой могут привести к неожиданному результату. Так, например, естественно предположить, *first не изменяет first, хотя синтаксически никаких ограничений на это нет.

Концепты

Несколько усилить требования к подставляемым типам могут помочь введённый в стандарт относительно недавно (начиная с C++20) аппарат концептов. В принципе, подобного эффекта можно было достичь и раньше с использованием принципа SFINAE (substitution failure is not an error – неудачная подстановка не является ошибкой) и производных инструментов, таких, как std::enable_if, однако их синтаксис является достаточно громоздкий, и полученный код становится читать не очень приятно. Использование концептов, в частности, позволяет получать куда более прозрачное сообщение об ошибке при попытке использования неподходящего типа.

На нашем простом примере концепт мог бы выглядеть, например, так:

```
template <class T>
concept Property = requires (T const& p) { p.getTax(); };
```

А объявление printTax:

```
template <Property T>
void printTax(T const& p);
```

Теперь, если мы попытаемся передать в качестве параметра int, мы получим весьма точный вывод сообщения об ошибке:

```
<source>:46:13: error: no matching function for call to 'printTax(int)'

   46 |     printTax(5);

      |     ~~~~~~~~^~~

<source>:34:6: note: candidate: 'template<class T>  requires  Property<T> void
printTax(const T&)'

   34 | void printTax(T const& p)

      |      ^~~~~~~~

<source>:34:6: note:   template argument deduction/substitution failed:
```

```
<source>:34:6: note: constraints not satisfied
```