

Программирование

Лекция 5. Константность.
Конструктор копирования. Класс
массива. ООП

Константные методы

Ключевое слово `const` можно использовать для нотации методов классов, которые не изменяют полей объектов.

- Методы классов могут быть объявлены как `const`.

```
struct IntArray {  
    size_t size() const;  
};
```

Класс целочисленного массива
Метод возвращает размер массива, не изменяя полей класса

Если в таком методе менять поля класса, то будет

ошибка.

- Такие методы не могут менять поля объекта (тип `this` — указатель на `const`).
- У константных объектов (через указатель или ссылку на константу) можно вызывать только константные методы:

```
IntArray const * p = foo();  
p->resize(); // ошибка resize не является константным методом
```

- Внутри константных методов можно вызывать только константные методы.

Ключевое слово mutable

Иногда возникает необходимость менять поля класса
внутри константных методов

Ключевое слово `mutable` позволяет определять поля, которые
можно изменять внутри константных методов:

```
struct IntArray {  
    size_t size() const {  
        ++counter_  
        return size_  
    }  
  
private:  
    size_t size_;  
    int * data_;  
  
    mutable size_t counter_;  
};
```

Например, необходимо
посчитать, сколько раз
вызывается метод у
определенного экземпляра
класса

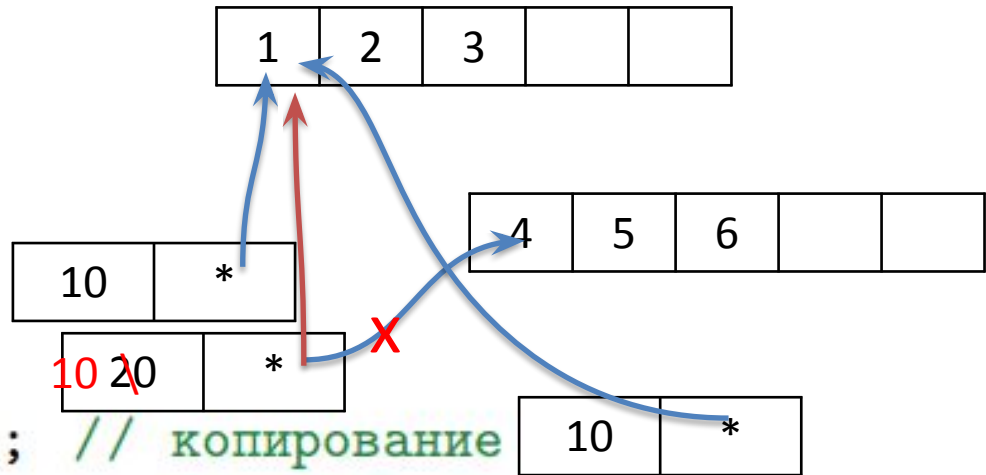
Ключевым словом `mutable` нужно определять
только те поля, которые не являются частью
состояния объекта.

Поле `counter` никак не влияет
на значение массива

Копирование объектов

```
struct IntArray {  
    ...  
private:  
    size_t size_;  
    int * data_;  
};
```

```
int main() {  
    IntArray a1(10);  
    IntArray a2(20);  
    IntArray a3 = a1; // копирование  
    a2 = a1; // присваивание  
  
    return 0;  
}
```



При выходе из функции произойдет вызов деструкторов: `a3` (освобождение памяти динамического массива `[1,2,3,...]`), `a2` (попытка освободить память – ошибка), `a1`. Кроме того, утечка памяти – массив `[4,5,6,...]` остался

Конструктор копирования

Если не определить конструктор копирования, то он сгенерируется компилятором.

```
struct IntArray {  
    IntArray(IntArray const& a)  
        : size_(a.size_), data_(new int[size_])  
    {  
        for (size_t i = 0; i != size_; ++i)  
            data_[i] = a.data_[i];  
    }  
    ...  
private:  
    size_t size_;  
    int * data_;  
};
```

Константная ссылка на объект того же типа

Копируем поле size

Указатель на новый массив

Копируем значения массива

Конструктор копирования вызывается в тех случаях, когда при копировании создается новый объект, н-р, при передаче объекта в функцию по значению.

Оператор присваивания

- Когда копирование происходит в уже существующий объект, вызывается оператор присваивания.

Если не определить оператор присваивания, то он тоже сгенерируется компилятором.

```
struct IntArray {  
    IntArray & operator=(IntArray const& a)  
    {  
        if (this != &a) {  
            delete [] data_;  
            size_ = a.size_;  
            data_ = new int[size_];  
            for (size_t i = 0; i != size_; ++i)  
                data_[i] = a.data_[i];  
        }  
        return *this;  
    }  
    ...  
};
```

Ссылка на зн-е текущего объекта

Проверка, не присваиваем ли зн-е самого себя

Сначала удаляем данные, к-рые уже есть в объекте

Выделяем новый дин.массив

Разыменовываем, т.о. получаем, ссылку

Метод swap

- Реализацию оператора присваивания можно упростить методом swap

```
struct IntArray {  
    void swap(IntArray & a) {  
        size_t const t1 = size_  
        size_ = a.size_  
        a.size_ = t1;  
  
        int * const t2 = data_  
        data_ = a.data_  
        a.data_ = t2;  
    }  
    ...  
private:  
    size_t size_  
    int * data_  
};
```

Ссылка на объект того же типа

Обмениваем зн-я поля size

Обмениваем зн-я поля data

После вызова метода swap, данные массивов обменяются местами

Метод swap

- Реализацию метода swap можно упростить за счет использования библиотечной функции swap.

Можно использовать функцию `std::swap` и файла `algorithm`.

```
#include <algorithm>

struct IntArray {
    void swap(IntArray & a) {
        std::swap(size_, a.size_);
        std::swap(data_, a.data_);
    }
    ...
private:
    size_t size_;
    int * data_;
};
```

Переставляет
местами 2 зн-я

Реализация оператора = при помощи swap

- Используя конструктор копирования и метод swap, можно реализовать оператор

```
struct IntArray {
    IntArray(IntArray const& a)           Конструктор
        : size_(a.size_), data_(new int[size_]) { копирования
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    IntArray & operator=(IntArray const& a) Оператор
        if (this != &a) присваивания
            IntArray(a).swap(*this);
        return *this;
}
...
private:
    size_t size_;
    int * data_;
};
```

- 1) IntArray t(a)
- 2) t.swap(*this)
- 3) Временное зн-е t будет уничтожено

Запрет копирования объектов

- Иногда необходимо запретить копирование объектов

Для того, чтобы запретить копирование, нужно объявить конструктор копирования и оператор присваивания как `private` и не определять их.

```
struct IntArray {  
    ...  
private:  
    IntArray(IntArray const& a);  
    IntArray & operator=(IntArray const& a);  
  
    size_t size_;  
    int * data_;  
};
```

Конструктор копирования

Произойдет ошибка компиляции при копировании или присваивании переменной данного типа

Методы, генерируемые компилятором

Компилятор генерирует четыре метода:

1. конструктор по умолчанию,
2. конструктор копирования,
3. оператор присваивания,
4. деструктор.

Если потребовалось переопределить конструктор копирования, оператор присваивания или деструктор, то нужно переопределить и остальные методы из этого списка.

Поля и конструкторы

- Обобщим знания о классе для массива.

```
struct IntArray {
    explicit IntArray(size_t size) {
        : size_(size), data_(new int[size]) {
            for (size_t i = 0; i != size_; ++i)
                data_[i] = 0;
        }
    }
    IntArray(IntArray const& a) {
        : size_(a.size_), data_(new int[size_]) {
            for (size_t i = 0; i != size_; ++i)
                data_[i] = a.data_[i];
        }
    }
    ...
private:
    size_t size_;
    int * data_;
};
```

Обязательно нужен 1 аргумент

Запрет неявного преобразования

Инициализация элементов нулями

Конструктор копирования

2 поля

Деструктор, оператор присваивания и swap

```
~IntArray() {  
    delete [] data_;  
}
```

Удаляем динамический массив

```
IntArray & operator=(IntArray const& a) {  
    if (this != &a)  
        IntArray(a).swap(*this);  
    return *this;  
}
```

```
void swap(IntArray & a) {  
    std::swap(size_, a.size_);  
    std::swap(data_, a.data_);  
}
```

Библиотечная функция swap из стандартной библиотеки

Наследование

Наследование — это механизм, позволяющий создавать производные классы, расширяя уже существующие.

```
struct Person {
    string name() const { return name_; } «getter»
    int age() const { return age_; }
private:
    string name_;
    int age_;
};

struct Student : Person {
    string university() const { return uni_; }
private:
    string uni_;
};
```

Класс, описывающий человека

Класс Student является производным класса Person

Класс Student будет иметь также все поля и методы класса Person

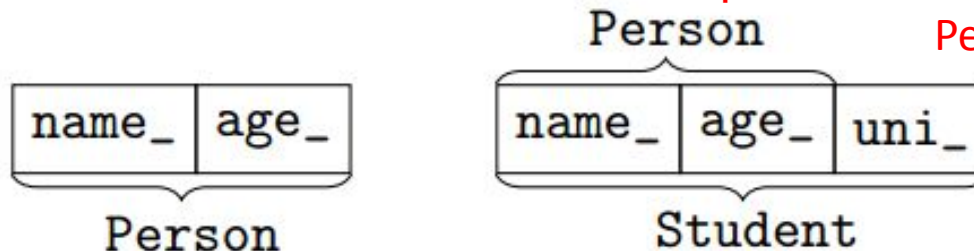
Класс-наследник

У объектов класса-наследника можно вызывать публичные методы родительского класса.

```
Student s;  
cout << s.name() << endl  
      << s.age() << endl  
      << s.university() << endl;
```

Внутри объекта класса-наследника хранится экземпляр родительского класса.

Вначале объекта типа Student хранится экземпляр класса



Создание/удаление объекта производного класса

При создании объекта производного класса сначала вызывается конструктор родительского класса.

```
struct Person {
    Person(string name, int age)
        : name_(name), age_(age)
    {}
    ...
};
struct Student : Person {
    Student(string name, int age, string uni)
        : Person(name, age), uni_(uni)
    {}
    ...
};
```

Конструктор от 2х параметров

При создании производного класса нужно вызвать конструктор родителя

После деструктора Student вызывается деструктор Person.

Приведения

- Производные классы связаны со своими базовыми (или родительскими) классами при помощи приведений

Для производных классов определены следующие приведения:

```
Student s("Alex", 21, "Oxford");  
Person & l = s; // Student & -> Person &  
Person * r = &s; // Student * -> Person *
```

Создаем ссылку и указатель базового класса на производный тип

Поэтому объекты класса-наследника могут присваиваться объектам родительского класса:

```
Student s("Alex", 21, "Oxford");  
Person p = s; // Person("Alex", 21);
```

Скопированы соответствующие поля

При этом копируются только поля класса-родителя (срезка).
(Т.е. в данном случае вызывается конструктор копирования `Person(Person const& p)`, который не знает про `uni_.`)

Модификатор доступа protected

- Класс-наследник не имеет доступа к private-членам родительского класса. (к полям и методам)
- Для определения закрытых членов класса доступных наследникам используется модификатор protected.

```
struct Person {  
    ...  
protected:  
    string name_  
    int age_  
};  
  
struct Student : Person {  
    ... // можно менять поля name_ и age_  
};
```

Перегрузка функций

- Для того, чтобы продолжить изучать наследование и говорить о переопределении методов, вспомним, что такое перегрузка.
- Перегрузка – это возможность определить несколько функций с одним и тем же именем, но различными типами

В отличие от C в C++ можно определить несколько функций с одинаковым именем, но разными параметрами.

```
double square(double d) { return d * d; }  
  
int square(int i) { return i * i; }
```

При вызове функции по имени будет произведен поиск наиболее подходящей функции:

```
int a = square(4); // square(int)  
double b = square(3.14); // square(double)  
double c = square(5); // square(int) Возвращ.зн-е будет  
int d = square(b); // square(double) преобразовано  
float e = square(2.71f); // square(double)
```

Не double и не int

Перегрузка методов

- Аналогично перегрузке функций, в C++ существует перегрузка методов.

```
struct Vector2D { Класс, описывающий вектор на пл-ти
    Vector2D(double x, double y) : x(x), y(y) {}

    Vector2D mult(double d) const Умн-е вектора на число
        { return Vector2D(x * d, y * d); }

    double mult(Vector2D const& p) const Умн-е вектора на
        { return x * p.x + y * p.y; } вектор
                                           (скалярное)

    double x, y;
};
```

```
Vector2D p(1, 2); Определяем
Vector2D q = p.mult(10); // (10, 20) Умн-е на число
double r = p.mult(q); // 50 Умн-е p на q
```

Перегрузка при наследовании

```
struct File {  
    void write(char const * s);  
    ...  
};
```

Класс для работы с файлами
Метод для записи строки в файл

```
struct FormattedFile : File {  
    void write(int i);  
    void write(double d);  
    using File::write;  
    ...  
};
```

Производный класс
Объявлены перегрузки – методы для записи целых и вещест. чисел
Данные методы «перекрывают» методы базового класса
Данная строка позволяет использовать метод из базового класса

```
FormattedFile f;  
f.write(4);  
f.write("Hello");
```

Не скомпилируется, если нет using File::write...

Правила перегрузки

- Как компилятор выбирает правильную функцию при перегрузке?
 1. Если есть точное совпадение, то используется оно.
 2. Если нет функции, которая могла бы подойти с учётом преобразований, выдаётся ошибка.
 3. Есть функции, подходящие с учётом преобразований:
 - 3.1 Расширение типов.
`char, signed char, short → int`
`unsigned char, unsigned short → int/unsigned int`
`float → double`
 - 3.2 Стандартные преобразования (числа, указатели).
 - 3.3 Пользовательские преобразования.  `Н-р, double в int`

В случае нескольких параметров нужно, чтобы выбранная функция была *строго лучше* остальных. **Иначе – ошибка – вызов**

NB: перегрузка выполняется на этапе компиляции. **неоднозначен, т.е. неск-ко функций одинаково**
лат. Nota bene **(не на этапе выполнения)**

«обрати внимание»

Задача

Есть три версии функции foo:

```
void foo(char) { std::cout << "char" << std::endl; }  
void foo(signed char) { std::cout << "signed char" << std::endl; }  
void foo(unsigned char) { std::cout << "unsigned char" << std::endl; }
```

Отметьте все верные утверждения относительно вызова функции foo.

- в результате вызова foo('a') будет выведено unsigned char
- в результате вызова foo('a') будет выведено char
- в результате вызова foo('a') будет выведено signed char
- в результате вызова foo(97) будет выведено signed char
- в результате вызова foo(97) будет выведено char
- вызов foo('a') приведет к ошибке компиляции
- вызов foo(97) приведет к ошибке компиляции

Задача 2

В программе определены две функции:

```
float square(float value) { return value * value; }  
double square(float value) { return (double)value * value; }
```

Далее в программе есть вызов:

```
double sq = square(2.0);
```

Отметьте все верные утверждения из списка.

- программа не скомпилируется, потому что такая перегрузка функции square не допустима
- программа не скомпилируется, потому что вызов double sq = square(2.0) неоднозначен
- программа скомпилируется, будет вызвана функция float square(float value)
- программа скомпилируется, будет вызвана функция double square(float value)

Задача 3

```
void promotion(char &) { std::cout << "char" << std::endl; }  
void promotion(int &) { std::cout << "int" << std::endl; }  
void promotion(long &) { std::cout << "long" << std::endl; }
```

Кроме того в программе есть вызов:

```
short sh = 10;  
promotion(sh);
```

Отметьте все верные утверждения.

- вызов `promotion(sh)` не скомпилируется, так как есть несколько подходящих функций для вызова
- вызов `promotion(sh)` не скомпилируется, так как нет ни одной подходящей функции для вызова
- при вызове `promotion(sh)` произойдет преобразование типа и будет вызвана функция `promotion(long &)`
- при вызове `promotion(sh)` произойдет преобразование типа и будет вызвана функция `promotion(int &)`
- при вызове `promotion(sh)` произойдет преобразование типа и будет вызвана функция `promotion(char &)`

Переопределение методов (overriding)

- Перегрузка – определение функции с тем же именем, но другой сигнатурой.
- Переопределение – определение метода с тем же именем и сигнатурой, как у базового класса.

```
struct Person { Базовый класс
    string name() const { return name_; }
    ...
};
struct Professor : Person {
    string name() const {
        return "Prof. " + Person::name();
    }
    ...
};
```

**Метод из базового
класса**

```
Professor pr("Stroustrup");
cout << pr.name() << endl; // Prof. Stroustrup
Person * p = &pr;
cout << p->name() << endl; // Stroustrup
```

Виртуальные методы

```
struct Person {  
    virtual string name() const { return name_; }  
    ...  
};  
struct Professor : Person {  
    string name() const {  
        return "Prof. " + Person::name();  
    }  
    ...  
};
```

Тогда при вызове метода через указатель на базовый класс, какой метод будет вызван, будет зависеть не от типа указателя, а от объекта, на который он ссылается

```
Professor pr("Stroustrup");  
cout << pr.name() << endl; // Prof. Stroustrup  
Person * p = &pr;  
cout << p->name() << endl; // Prof. Stroustrup
```

Чистые виртуальные (абстрактные)

МЕТОДЫ

- Особый вид виртуальных методов – чистые вирт. методы. Это виртуальные методы, у которых отсутствует реализация.

```
struct Person {  
    virtual string occupation() const = 0;  
    ...  
};  
struct Student : Person {  
    string occupation() const {return "student";}  
    ...  
};  
struct Professor : Person {  
    string occupation() const {return "professor";}  
    ...  
};
```

Нет реализации

Сам класс становится абстрактным, т.е. нельзя создать экземпляр класса

Но можно создавать производные класса

```
Person * p = next_person();  
cout << p->occupation();
```

Но можно создавать указатели на Student или Professor

Будет вызвана соответствующая реализация, основываясь на типе

продолжение

Виртуальный деструктор

К чему приведёт такой код?

```
struct Person {
    ...
};
struct Student : Person {
    ...
private:
    string uni_; Название университета
};

int main() {
    Person * p = new Student("Alex", 21, "Oxford");
    ...
    delete p; Деструктор класса Person, то есть поле "uni"
              останется висеть в памяти
}
```

Виртуальный деструктор

Правильная реализация:

```
struct Person {  
    ...  
    virtual ~Person() {}  
};  
struct Student : Person {  
    ...  
private:  
    string uni_;  
};  
  
int main() {  
    Person * p = new Student("Alex", 21, "Oxford");  
    ...  
    delete p; При удалении будет вызван ~Student  
}
```

Нужно указать, что
деструктор базового
класса является
виртуальным

Полиморфизм

Полиморфизм

Возможность единообразно обрабатывать разные типы данных.

Перегрузка функций Первый механизм полиморфизма

Выбор функции происходит в момент компиляции на основе типов аргументов функции, *статический полиморфизм*.

Виртуальные методы Второй механизм полиморфизма

Выбор метода происходит в момент выполнения на основе типа объекта, у которого вызывается виртуальный метод, *динамический полиморфизм*.

Ещё раз об ООП

Объектно-ориентированное программирование — концепция программирования, основанная на понятиях объектов и классов.

Это экземпляры класса

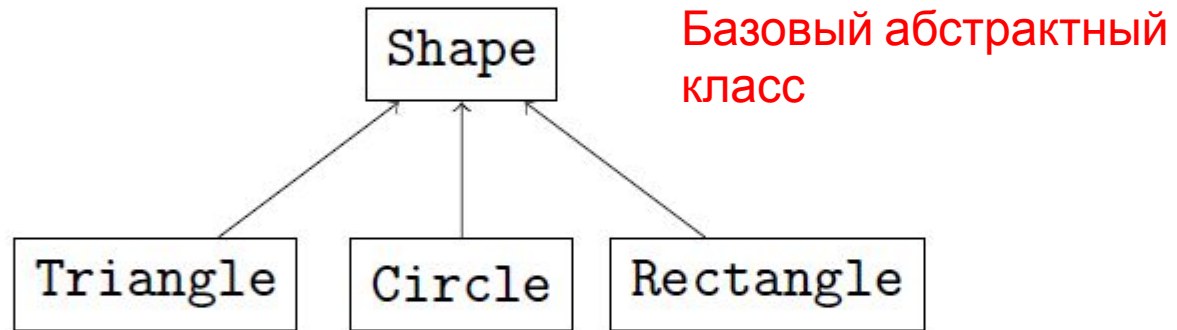
Основные принципы:

- инкапсуляция, Объединение логически связанных данных с методами работы с этими данными
- наследование, Возможность создавать производные класса
- полиморфизм, Работа с подклассами через ссылку на базовый класс
- абстракция. Возможность скрыть реализацию класса при помощи модификаторов доступа

Подробнее о принципах проектирования ООП-программ можно узнать по ключевым слову „шаблоны проектирования”.

Как правильно построить иерархию?

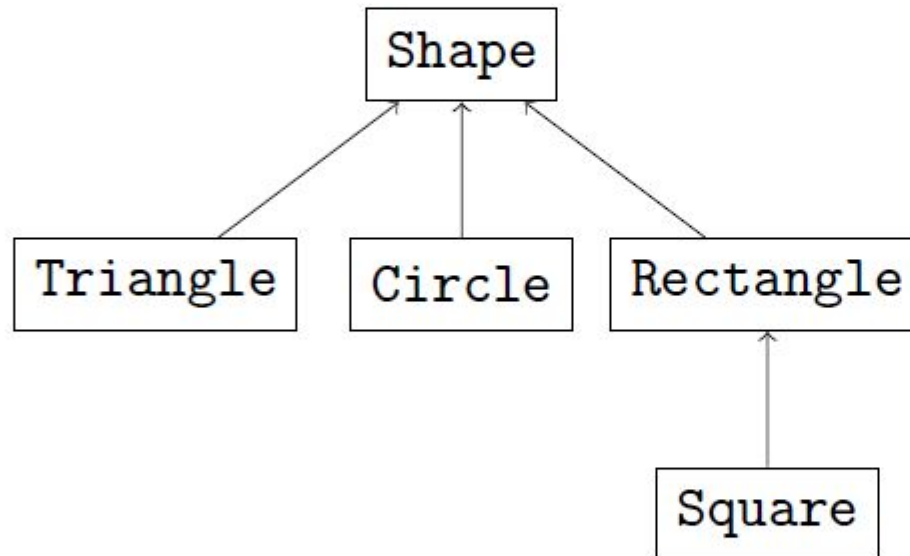
Иерархия геометрических фигур:



Куда добавить класс Square?

Как правильно построить иерархию?

Квадрат — это прямоугольник, у которого все стороны равны.



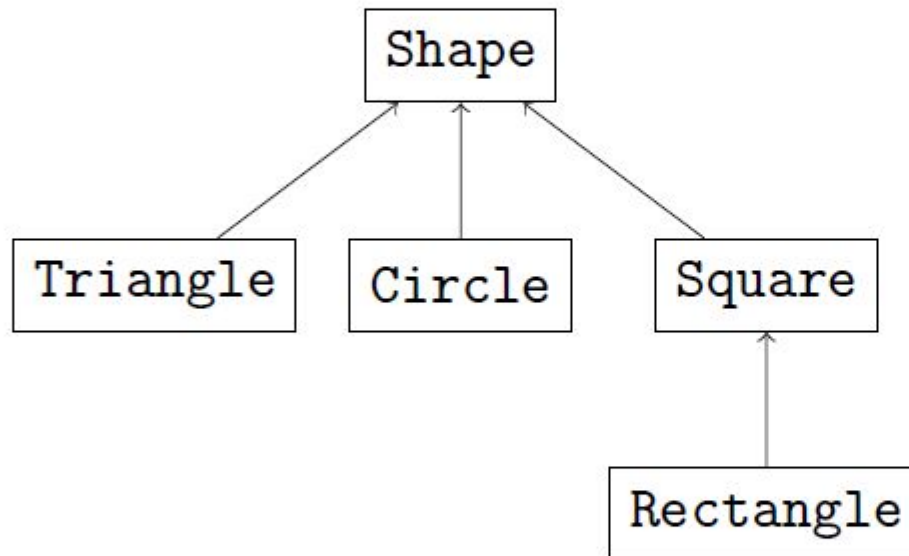
```
void double_width(Rectangle & r) {
    r.set_width(r.width() * 2);
}
```

Функция получает прямоугольник по ссылке и увеличивает его ширину в 2 раза

Но если в эту функцию передадим квадрат, то будет некорректная работа программы

Как правильно построить иерархию?

Прямоугольник задаётся двумя сторонами, а квадрат — одной.



Расширили класс квадрата, добавив еще 1 поле

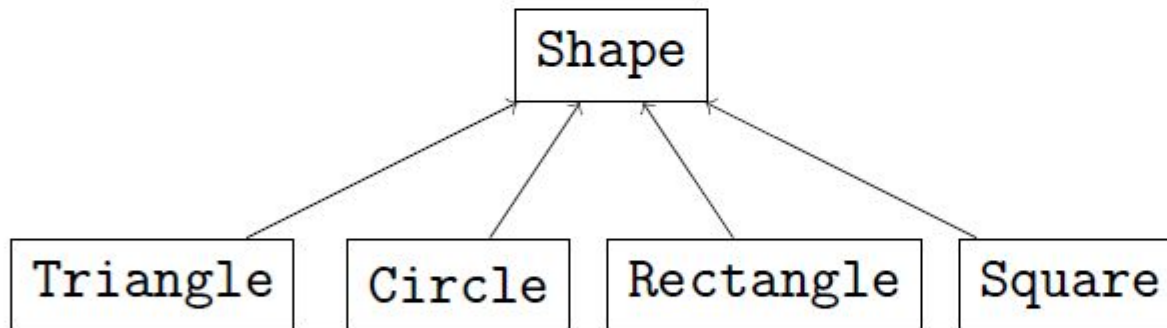
```
double area(Square const& s) {
    return s.width() * s.width();
}
```

Метод вычисляет площадь

Метод будет работать некорректно для прямоугольника

Как правильно построить иерархию?

Правильное решение — сделать эти классы независимыми:



Но в некоторых частных случаях, квадрат можно унаследовать от прямоугольника и наоборот, в зависимости от того, какие методы и функции понадобятся для работы с этими классами.

Агрегирование vs наследование

- Иногда вместо наследования используют агрегирование.
- *Агрегирование* — это включение объекта одного класса в качестве поля в другой. Н-р, класс компьютер (содержит клавиатуру, мышь и т.д.)
- Наследование устанавливает более сильные связи между классами, нежели агрегирование:
 - приведение между объектами,
 - доступ к `protected` членам.Тогда класс комп-р агрегирует в себя объекты классов клавиатуры, мыши и т.д.
- Если наследование можно заменить легко на агрегирование, то это нужно сделать. Чем меньше зависимость, тем меньше нужно менять при изменении компонентов.

Примеры некорректного наследования

- Класс `Circle` унаследовать от класса `Point`. Рас-е м/у 2 двумя точками
- Класс `LinearSystem` унаследовать от класса `Matrix`. Транспонирование матриц не имеет смысла для системы линейных уравнений

Принцип подстановки Барбары Лисков

Liskov Substitution Principle (LSP)

Функции, работающие с базовым классом, должны иметь возможность работать с подклассами не зная об этом.

Этот принцип является важнейшим критерием при построении иерархий наследования.

Другие формулировки

- Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом.
- Подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс

Модификаторы при наследовании

При наследовании можно использовать модификаторы доступа:

```
struct A {};
```

B1 может вызывать все методы класса A

```
struct B1 : public A {};
```

Только внутри класса B2 можно вызывать методы класса A

```
struct B2 : private A {};
```

protected = private, только методы будут доступны также в производных классах B3

```
struct B3 : protected A {};
```

Для классов, объявленных как `struct`, по-умолчанию используется `public`, для объявленных как `class` — `private`.

Важно: отношение наследования (в терминах ООП) задаётся только `public`-наследованием.

Использование `private`- и `protected`-наследований целесообразно, если необходимо не только агрегировать другой класс, но и переопределить его виртуальные методы.

Переопределение private виртуальных методов

```
struct NetworkDevice { // сетевое
    void send(void * data, size_t size) { // Публичный метод
        log("start sending");           send – нельзя
        send_impl(data, size);         переопределить у
        log("stop sending");          наследников
    }
    ...
private:
    virtual void send_impl(void * data, size_t size)
    {...}
};

struct Router : NetworkDevice {
private:
    void send_impl(void * data, size_t size) {...}
};
```

Это шаблон template method

Интерфейсы

Интерфейс — это абстрактный класс, у которого отсутствуют поля, а все методы являются чистыми виртуальными.

Отсутствует
реализация

```
struct IConvertibleToString {  
    virtual ~IConvertibleToString() {}  
    virtual string toString() const = 0;  
};
```

Только у деструктора

должна быть

реализация

```
struct IClonable {  
    virtual ~IClonable() {}  
    virtual IClonable * clone() const = 0;  
};
```

```
struct Person : IClonable {  
    Person * clone() {return new Person(*this);}  
};
```

Создаем копию
человека

Множественное наследование

В C++ разрешено множественное наследование.

```
struct Person {};  
struct Student : Person {};  
struct Worker : Person {};  
struct WorkingStudent : Student, Worker {};
```

Базовые классы через запятую (будет 2 экземпляра Person)

Стоит избегать *наследования реализаций* более чем от одного класса, вместо этого использовать интерфейсы.

```
struct IWorker {};  
struct Worker : Person, IWorker {};  
struct Student : Person {};  
struct WorkingStudent : Student, IWorker {}
```

Множественное наследование — это отдельная большая тема.

Использование интерфейсов позволит избежать дублирования данных и неоднозначности

Вопросы

1. Что такое константный метод?
2. Для чего применяется ключевое слово `mutable`?
3. Перечислите методы, генерируемые компилятором.
4. Что такое наследование?
5. Приведите пример наследования на C++

Задача на закрепление материала по модификатору `const`

- Объявите переменную с именем m , в которой хранится указатель на двумерный массив целых чисел (`int`), выделенный в динамической памяти, так чтобы содержимое массива нельзя было поменять, т. е. компилятор должен выдавать ошибку при попытке произвести над m любое действие, которое изменит значение $m[i][j]$ для любых i и j .
- Требования к выполнению задания: в задании требуется только объявить переменную, инициализировать ее не нужно. Например, объявление может выглядеть так:
- `int **m;`

ОТВЕТ

- *int const * const * m;*