# Наследование

# Общий вид перегруженных операторов ввода - вывода

```
class Sample
{
public:
    friend std::ostream& operator<<(std::ostream& stream, Sample const& out)
    {
        // ...
        return stream;
    }

    friend std::istream& operator>>(std::istream& stream, Sample& in)
    {
        // ...
        return stream;
    }
};
```

# Перегрузка операторов вывода - вывода

```
class Float
{
    float value;
public:
    Float() : value(0) {}
    Float(float val) : value(val) {}

    Float operator+(Float const&) const;
    Float operator-(Float const&) const;
    Float operator*(Float const&) const;
    Float operator/(Float const&) const;

    friend std::ostream& operator<<(std::ostream&, Float const&);
    friend std::istream& operator>>(std::istream&, Float&);
};
```

```
Float Float::operator*(Float const& other) const
{
    return Float(value * other.value);
}

Float Float::operator+(Float const& other) const
{
    return Float(value + other.value);
}

Float Float::operator-(Float const& other) const
{
    return Float(value - other.value);
}

Float Float::operator/(Float const& other) const
{
    return Float(value / other.value);
}
```

```cpp
std::ostream& operator<<(std::ostream& stream, Float const& out)
{
    stream << out.value;
    return stream;
}

std::istream& operator>>(std::istream& stream, Float& in)
{
    stream >> in.value;
    return stream;
}

int main()
{
    Float PI = 3.14F;
    Float radius;

    std::cout << "Enter radius: ";
    std::cin >> radius;

    std::cout << "Length = " << PI * 2 * radius << std::endl;
}
```

# Паттерн «Singleton»

```
class Singleton
{
private:
    Singleton() {}
public:
    static Singleton& getInstance()
    {
        static Singleton instance;
        return instance;
    }
};

void f()
{
    Singleton& singleton = Singleton::getInstance();
}
```

# Перечисления

Перечисление – это тип данных, который может содержать значения, указанные программистом. Объявляется с помощью ключевого слова enum.

Существует два типа перечислений:

- Scoped
- Unscoped

Общий вид:

```
enum [class] имяПеречисления [: type] { ЗНАЧ_1, ЗНАЧ_2, ЗНАЧ_3, ... };
```

# Unscoped enum

- Значения поддерживают автоматическое преобразование к int.
- Значения видны во всей области видимости, в которой объявлен enum.
- Поддерживаются switch как базовый тип.
- Лежащий в основе тип всегда int.

# Unscoped enum

```
enum Color { BLACK = 0x000000, WHITE = 0xFFFFFF };

int main()
{
    Color color = RED;
    std::cout << std::hex;

    switch (color)
    {
        case BLACK:
            std::cout << "Black color value = " << BLACK << std::endl;
            break;
        case WHITE:
            std::cout << "White color value = " << WHITE << std::endl;
            break;
    }
}
```

# Scoped enum

- Указывется с помощью ключевого слова class.
- Значения не поддерживают автоматическое преобразование к int.
- Значения не видны во всей области видимости, в которой объявлен enum, их область видимости ограничена enum.
- Поддерживаются switch как базовый тип.
- Лежащий в основе тип можно выбрать самостоятельно.

# Scoped enum

```
class Screen
{
public:
    enum class Resolution : char { LOW, MID, HIGH };

    void resetResoulution(Resolution resolution)
    {
        res = resolution;
    }

    void showScreenInfo();
private:
    Resolution res;

    int width;
    int height;
};
```

# Scoped enum

```
void Screen::showScreenInfo()
{
    std::cout << "w: " << width << std::endl;
    std::cout << "h: " << height << std::endl;

    switch (res)
    {
        case Resolution::HIGH:
            std::cout << "high res";
            break;
        case Resolution::LOW:
            std::cout << "low res";
            break;
        case Resolution::MID:
            std::cout << "mid res";
            break;
    }
}
```

# Scoped enum

```
int main()
{
    Screen::Resolution res = Screen::Resolution::HIGH;

    Screen screen;

    screen.resetResoulution(res);
}
```

# Общий вид наследования классов

```
class ИмяУнаследованногоКласса : [модификатор] ИмяБазовогоКласса
{

}
```

# Пример наследования

```
struct Date
{
    int day;
    int month;
    int year;
};

class Employee
{
    char* name;
    const double salary;
    const Date hireDay;
public:
    Employee(const char* name, double salary, Date hireDay)
        : salary(salary), hireDay(hireDay)
    {
        name = new char[strlen(name) + 1];
        strcpy(this -> name, name);
    }
```

```
    char const* getName() const
    {
        return name;
    }

    double getSalary() const
    {
        return salary;
    }

    Date getHireDay() const
    {
        return hireDay;
    }
};
```

```
class Manager : public Employee
{
    double bonus;
public:
    Manager(const char* name, double salary, Date hireDay, double bonus)
        : Employee(name, salary, hireDay), bonus(bonus)
    {

    }

    double getSalary() const
    {
        double baseSalary = Employee::getSalary();
        return baseSalary + bonus;
    }

    double getBounus() const
    {
        return bonus;
    }
};
```

```
int main()
{
    Manager boss("Carl Cracker", 80000, { 7, 12, 1982 }, 5000);
    Employee harry("Harry Hacker", 50000, { 6, 4, 1990 });
    Employee tony("Tony Tester", 40000, { 15, 3, 1989 });

    std::cout << boss.getName() << ": " << boss.getSalary() << std::endl;
    std::cout << harry.getName() << ": " << harry.getSalary() << std::endl;
    std::cout << tony.getName() << ": " << tony.getSalary() << std::endl;
}
```

# Перекрытие методов

```
class X
{
public:
    void doSomething(int);
};

class Y : public X
{
public:
    void doSomething(long);
};

int main()
{
    Y object;
    object.doSomething(13);
}
```

# Решение проблемы перекрытия

```
class X
{
public:
    void doSomething(int);
};

class Y : public X
{
public:
    using X::doSomething;
    void doSomething(long);
};

int main()
{
    Y object;
    object.doSomething(13);
}
```

- Добавляет метод из X

# Полиморфизм

Позволяет «подменять» объекты базовых классов на объекты производных во время выполнения программы.

В C++ полиморфизм реализуется с помощью указателей (ссылок) и виртуальных методов.

# Виртуальный метод

- Для объявления виртуальной функции используется ключевое слово virtual.
- Функция-член класса может быть объявлена как виртуальная, если класс, содержащий виртуальную функцию, базовый в иерархии порождения.
- Реализация функции зависит от класса и будет различной в каждом порожденном классе.

# Чисто виртуальные методы

- Чисто виртуальный метод – это метод, реализацию которого программист возлагает на тех, кто будет наследоваться от его класса.
- Классы, содержащие чисто виртуальные методы, называются абстрактными.

# Пример использования полиморфизма и абстрактного класса

```
class Shape
{
public:
    virtual double square() const = 0;
    virtual double perimeter() const = 0;
};
```

```
class Rectangle : public Shape
{
    double a;
    double b;
public:
    Rectangle(double a, double b) : a(a), b(b)
    {

    }

    double square() const override
    {
        return a * b;
    }

    double perimeter() const override
    {
        return 2 * (a + b);
    }
};
```

```
class Triangle : public Shape
{
    double a;
    double b;
    double c;
public:
    Triangle(double a, double b, double c) : a(a), b(b), c(c)
    {

    }

    double square() const override
    {
        double halfP = perimeter() / 2;

        return halfP * sqrt((halfP - a) * (halfP - b) * (halfP - c));
    }

    double perimeter() const override
    {
        return a + b + c;
    }
};
```

```
int main()
{
    Shape* shapes[] = { new Rectangle(2, 4), new Triangle(3, 4, 5) };

    for (auto eachShape : shapes)
    {
        std::cout << eachShape -> square() << ' '
        << eachShape -> perimeter() << std::endl;
    }
}
```