

Язык C++

ООП. Наследование, Полиморфизм

Наследование

```
class CPerson {
public:
    CPerson(const std::string& name, unsigned yearOfBirth)
        : yearOfBirth_(yearOfBirth)
        , name_(name)    {}

    unsigned age() const {
        const std::chrono::time_point now{std::chrono::system_clock::now()};
        const std::chrono::year_month_day ymd{std::chrono::floor<std::chrono::days>(now)};

        return static_cast<int>(ymd.year()) - yearOfBirth_;
    }
    const std::string& name() const {
        return name_;
    }
private:
    std::string name_;
    unsigned yearOfBirth_;
};
```

Наследование

- позволяет описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.
- **полиморфизм подтипов, is-a relationship**
- обеспечивает повторное использование кода (следствие но не причина)
- множественное наследование

Наследование

```
class CStudent : public CPerson {  
public:  
    CStudent(const std::string& name, unsigned age, const std::string& university)  
        : CPerson(name, age)  
        , university_(university)  
    {}  
private:  
    std::string university_;  
};
```

Наследование

Наследник:

- Хранит в себе родителя
- Сохраняет методы родителя*
- Приведение к базовому классу (slicing)
- Модификаторы доступа

Constructor\Destructor order

Наследование

Specifiers	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes
			Sitesbay.com

Наследование

```
class CStudent : public CPerson {
public:
    CStudent(const std::string& name, unsigned yearOfBirth, const std::string& university)
        : CPerson(name, yearOfBirth)
        , university_(university)
    {}

    const std::string& university() const {
        return university_;
    }

    void Hello() const {
        std::cout << "Hello. I'am " << name() << " I'am from " << university_ << std::endl;
    }

private:
    std::string university_;
};
```


Наследование

```
// CBudgetStudent is a CStudent. CStudent is a CPerson

class CBudgetStudent : public CStudent {
public:
    CBudgetStudent(const std::string& name, unsigned yearOfBirth,
                   const std::string& university, unsigned sallary)
        : CStudent(name, yearOfBirth, university)
        , sallary_(sallary)
    {}

private:
    unsigned sallary_;
};
```

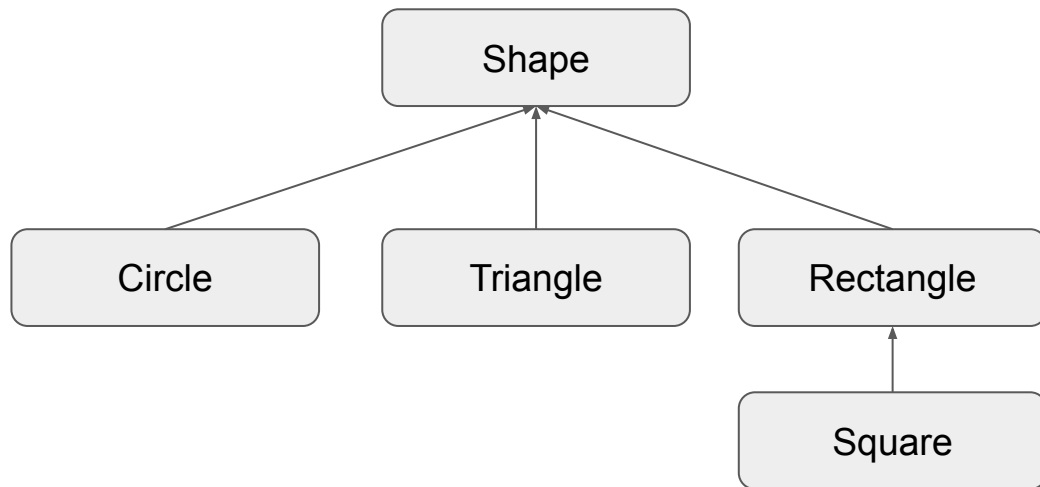
is-a relationship

```
void Hello(const CPerson& p) {  
    p.Hello();  
}  
  
int main() {  
    CBudgetStudent st = {"Ivan Ivanov", 2002, "ITMO", 20000};  
    Hello(st);  
    return 0;  
}
```

Наследование, устройство в памяти

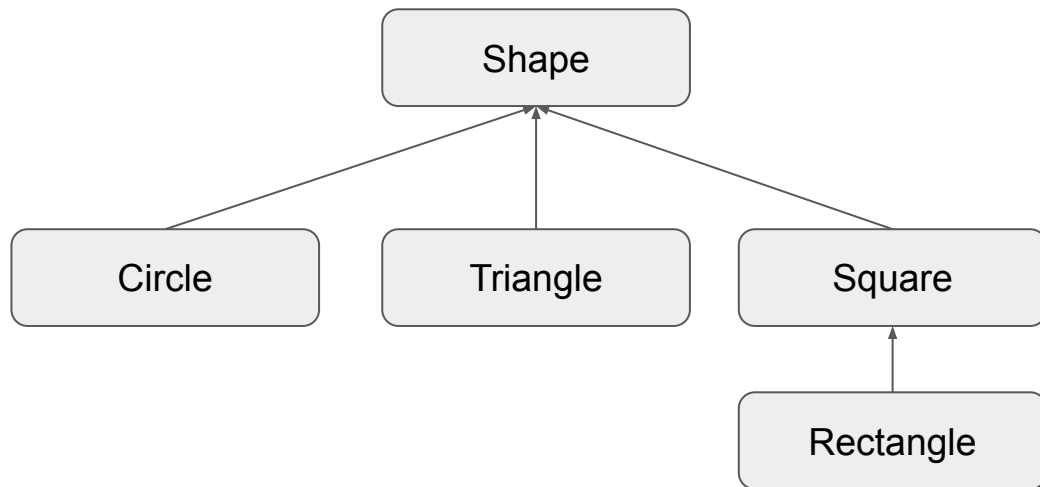
```
int main(int, char**) {  
    std::cout << "sizeof(CPerson): " << sizeof(CPerson) << std::endl;  
    std::cout << "sizeof(CStudent): " << sizeof(CStudent) << std::endl;  
    std::cout << "sizeof(CBudgetStudent): " << sizeof(CBudgetStudent) <<  
std::endl;  
}
```

Иерархия геометрических фигур



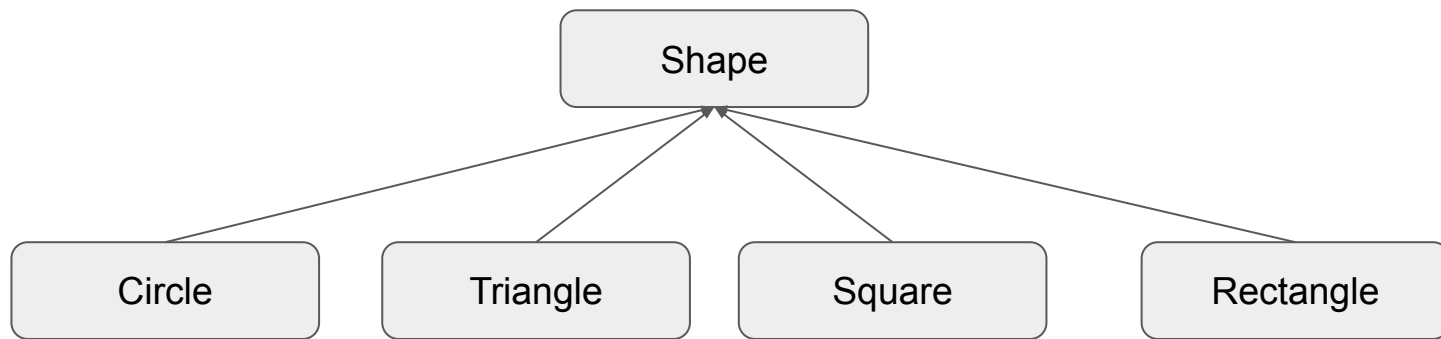
```
void double_width(Rectangle& r)
{
    r.setWidth(r.width*2);
}
```

Иерархия геометрических фигур



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

Иерархия геометрических фигур



Множественное наследование

```
class CEmployee : public CPerson {  
public:  
    CEmployee(const std::string& name, int yearOfBirth, unsigned salary)  
        : CPerson(name, yearOfBirth)  
        , salary_(salary)  
    {}  
  
private:  
    unsigned salary_ ;  
};
```

Множественное наследование

```
class CIntern : public CEmployee, public CBudgetStudent {
public:
    CIntern (
        const std::string& name,
        int yearOfBirth,
        const std::string& university,
        unsigned universSallary,
        unsigned workSallary
    )
        : CEmployee (name, yearOfBirth, workSallary)
        , CBudgetStudent (name, yearOfBirth, university, universeSallary)
    {}
};
```


Множественное наследование

```
int main(int, char**) {  
    std::cout << "sizeof(CPerson): " << sizeof(CPerson) << std::endl;  
    std::cout << "sizeof(CStudent): " << sizeof(CStudent) << std::endl;  
    std::cout << "sizeof(CBudgetStudent): " << sizeof(CBudgetStudent) << std::endl;  
  
    std::cout << "sizeof(CEmployee): " << sizeof(CEmployee) << std::endl;  
    std::cout << "sizeof(CIntern): " << sizeof(CIntern) << std::endl;  
}
```

Diamond Problem

```
int main(int, char**) {  
    CIntern intern("Ivan Ivanov", 2002, "ITMO", 20000, 50000);  
  
    intern.Hello();  
  
    // std::cout << intern.name();  compile-time error  
  
    std::cout << intern.CEmployee::name() << std::endl;  
    std::cout << intern.CBudgetStudent::name() << std::endl;  
    return 0;  
}
```

Проблемы множественного наследования

```
class CEmployee : public CPerson {
public:
    void IncreaseSalary() {
        salary_ += 1000;
    }
protected:
    unsigned salary_ ;
};

class CBudgetStudent : public CStudent {
public:
    void IncreaseSalary() {
        salary_ += 1000;
    }
protected:
    unsigned salary_ ;
};
```

```
class CIntern : public CEmployee, public CBudgetStudent {
public:
    using CEmployee::IncreaseSalary;

    unsigned Salary() const {
        return CEmployee::salary_ + CBudgetStudent::salary_;
    }
};

int main(int, char**) {
    CIntern intern("Ivan Ivanov", 2002, "ITMO", 20000, 50000);

    intern.IncreaseSalary();
    std::cout << intern.Salary() << std::endl;

    return 0;
}
```

Полиморфизм

- *свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.*

Динамический полиморфизм

- Позднее и раннее связывание
- Виртуальные функции

Виртуальные функции

```
class CConsoleLogger : public ILogger {  
public:  
    void Log(const char* message) {  
        std::cout << message << std::endl;  
    }  
};
```

Виртуальные функции

```
class CFileLogger : public ILogger {
public:
    CFileLogger(const char* filename)
        : stream_(filename)
    {}

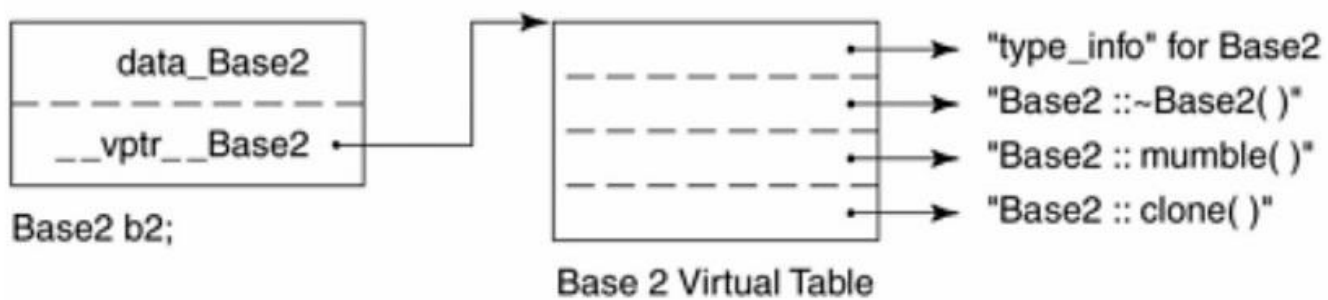
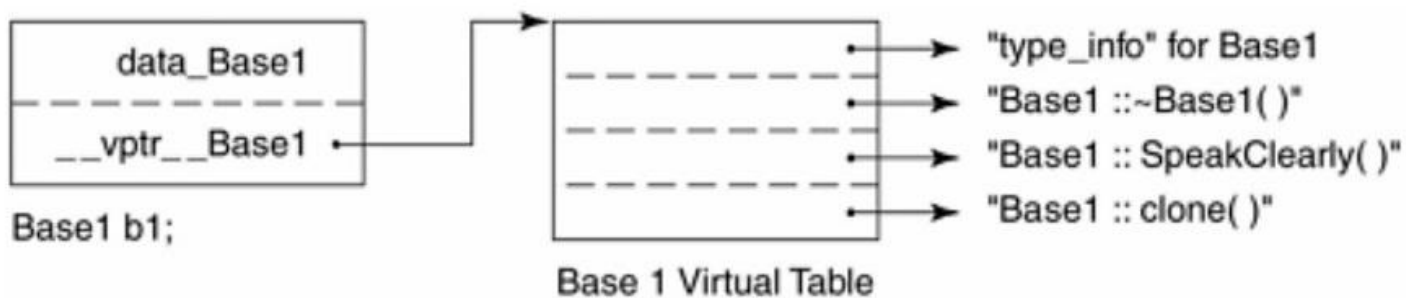
    ~CFileLogger() {
        stream_.close();
    }

    CFileLogger(const CFileLogger&) = delete;
    CFileLogger& operator=(const CFileLogger&) = delete;

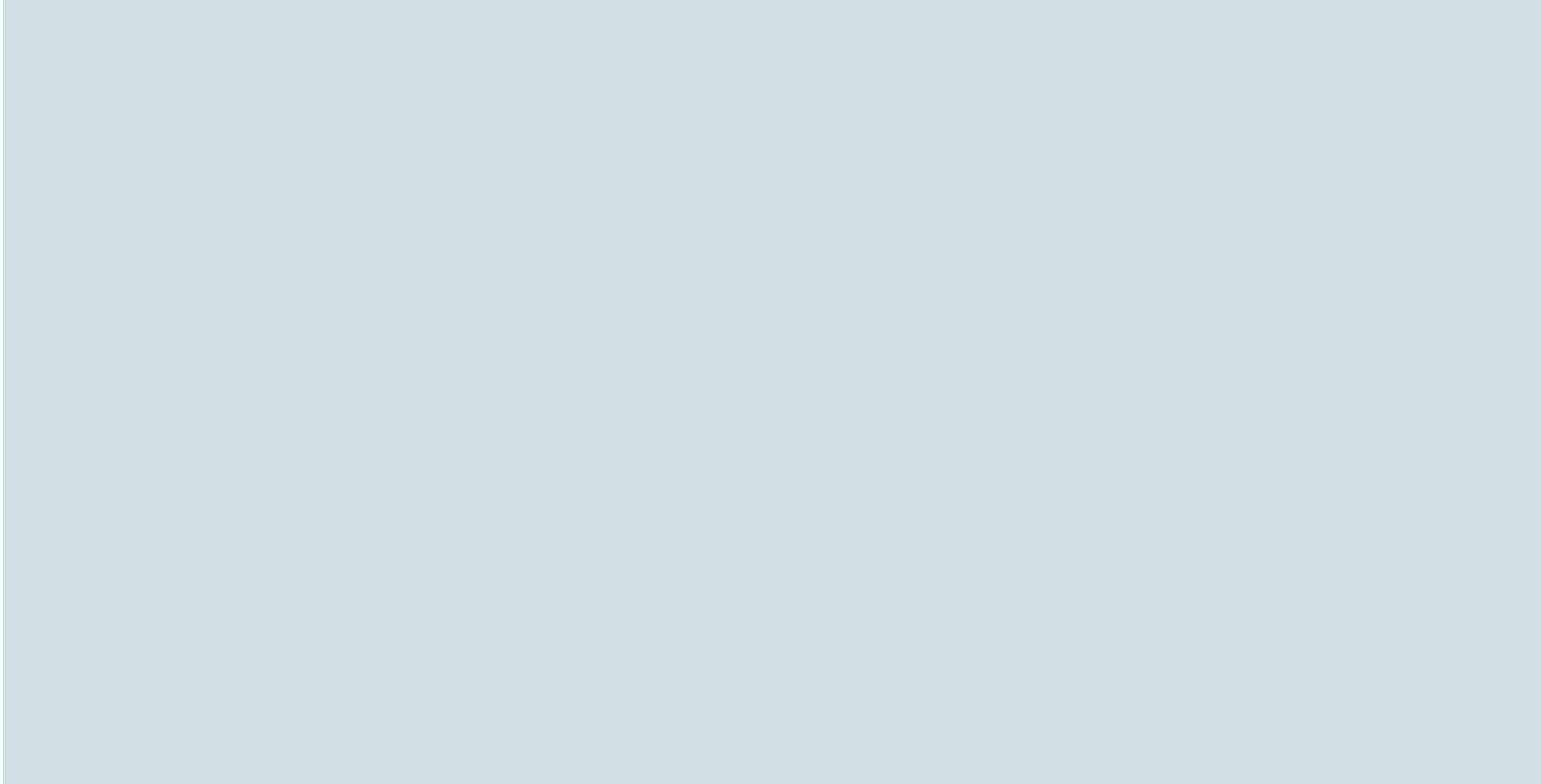
    void Log(const char* message) {
        stream_ << message << std::endl;
    }
private:
    std::ofstream stream_;
}
```

Таблица виртуальных функций

- Таблица заводится для любого класса с виртуальной функцией
- Вызов виртуального метода — это вызов метода по адресу из таблицы
-
- Стандарт не определяет механизм реализации виртуальных функций, однако большинство компиляторов реализуют именно таблицу виртуальных функций



final / override



Virtual destructor

```
class Base {
public:
    Base() { std::cout << "Base\n"; }
    virtual ~Base() { std::cout << "~Base\n"; }
};

class Derived : public Base {
public:
    Derived() { std::cout << "Derived\n"; }
    ~Derived() { std::cout << "~Derived\n"; }
};

int main(int, char**) {
    Base * d = new Derived;
    delete d;

    return 0;
}
```

Абстрактный класс

- класс экземпляр которого не может быть создан
- обычно используется в качестве базового класса
- содержит хотя бы 1 ***pure virtual function*** (чисто виртуальную функцию)

```
class ILogger {
public:
    virtual void Log(const std::string& msg) = 0; // pure virtual function
    virtual ~ILogger() = default;
};

int main(int, char**) {
    ILogger log; // Error : variable type 'ILogger' is an abstract class
    return 0;
}
```

Коллекции полиморфных объектов

Virtual Friend Function Idiom

```
class Base {
public:
    virtual ~Base() = default;

    friend std::ostream& operator<<(std::ostream& stream, const Base& value);
protected:
    virtual void printImpl(std::ostream& stream) const {
        stream << "Base\n";
    }
};

std::ostream& operator<<(std::ostream& stream, const Base& value) {
    value.printImpl(stream);
    return stream;
}
```

Virtual Friend Function Idiom

```
class Derived : public Base {  
protected:  
    void printImpl (std::ostream& stream) const override {  
        stream << "Derived\n";  
    }  
};
```

Стоимость виртуальных функций

- Лишнее обращение к таблице вместо явного адреса
- Не возможно сделать inline optimization
- Для коллекций объектов - они всегда в куче
- Порядок объектов также может влиять на скорость

ООП

- Абстракция
- Инкапсуляция
- Наследование
- Полиформизм