Объектно-ориентированный подход программирования

Литература

- Б. Страуструп – Язык программирования C++
- Р. Лафоре - Объектно-ориентированное программирование в C++
- Г. Макеев - Объектно-ориентированное программирование: с нуля к SOLID и MVC
- М. Вайсфельд - Объектно-ориентированное мышление
- Э. Гамма и др. – Приемы объектно-ориентированного проектирования
- М. Фаулер – UML. Основы

- en.cppreference.com – Документация языка C++

ООП

Различие парадигм программирования

Структурная

- отказ от goto
- можно описать в виде блок-схемы
- содержит только блоки:
  - последовательный
  - ветвление
  - цикл
- блоки могут быть вложены
- один вход - один выход

Функциональная

- используются функции высших порядков
- концепция чистой ф-ции
- иммутабельность
- использование монад
- отсутствие циклов

ООП

- используется моделирование информационных объектов
- иерархическая структура программы
- концепция отправки сообщений
- (обычно) внутри классов используется структурная парадигма

В настоящее время, многие ООП языки также используют концепции из функциональной парадигмы

Принципы ООП

- Абстракция - рассмотрение только значимой информации предметной области и исключение незначимой
- Инкапсуляция - объединение данных и методов, работающих с ними, в классе
- Наследование - свойство системы, позволяющее описывать новый класс на основе существующего
- Полиморфизм - использование объектов с общим интерфейсом без информации о внутренней структуре объекта

Понятия ООП

- Класс - единица абстракции. Универсальный тип данных, являющийся информационной моделью
- Объект - экземпляр класса, хранящийся в памяти
- Поля данных - параметры (свойства) объекта определяющие его состояние. Переменные, объявленные как принадлежащие классу
- Методы - функции связанные с классом. Определяют действия, которые можно выполнить над объектом, и которые объект может выполнить
- Модификаторы доступа - механизм определяющий доступность полей и методов объекта

Структуры

Способ синтаксически и физически сгруппировать логически связанные данные

```
struct Point{
  double x;
  double y;
};
struct Segment{
  Point p1, p2;
};
double length(Point p1, Point p2);
bool intersect(Segment seg1, Segment seg2, Point *p);
```

Поля структуры

- В качестве полей структуры можно указывать любые типы данных кроме типа самой структуры

```
struct Example
{
  std::string std;
  int* pi;
  Example& ref;
  Example* ptr;
  Example self; //Ошибка - Неполный тип
};
```

Методы структуры

- В C++ в структуре также можно объявить и определить функции - методы
- Метод имеет доступ ко всем полям структуры
- Для обращение к полям используется оператор `.`
- Для обращения через указатель используется оператор `->`

```
struct Point{
  double x;
  double y;
};

struct Segment{
  Point p1, p2;
  double length(){} //аргументы берутся из структуры
  Point* intersect(const Segment& seg){} //аргументом принимает другой отрезок
};
```

Структуры и классы

- Единственное различие в модификаторе доступа для полей
- В соответствии с формальным синтаксисом C++, объявление структуры создает тип класса
- Структуры в C++ сохранены для совместимости с C
- Структуры в основном используются только для объединения данных в единое целое
- Классы используются, если необходимо добавить методы для работы с данными

Модификаторы доступа

- **public** – доступ открыт всем, кто видит определение класса
- **protected** – доступ открыт классам, производным от данного
- **private** – доступ открыт самому классу, функциям-друзьям и классам-друзьям

- По умолчанию все поля и методы объявлены закрытыми (**private**)
- Для доступа к закрытым полям используются геттеры и сеттеры
- Применимы для структур (по умолчанию все поля **public**)

Объявление класса

- Используется ключевое слово **class**, вместо **struct**
- Синтаксически подобно объявлению структуры

```
class LinearFunction{
private:
  double a = 0.0;  double b = 0.0;
protected:
  std::string name = "Linear function";
public:
  void setCoefficients(double new_a, double new_b){a = new_a; b = new_b;}
  double calculate(double x){return a*x + b;}
};

int main(){
  LinearFunction lin_func;
  lin_func.setCoefficients(0.5, 1);
  std::cout << lin_func.calculate(2);
  lin_func.name = "Square function"; //Ошибка попытка доступа к закрытому полю
}
```

Определение методов вне класса

- В классе только сигнатура функции
- Имя функции надо указывать с названием класса через оператор ::

```
class LinearFunction{
private:
  double a = 0.0;
  double b = 0.0;
public:
  void setCoefficients(double new_a, double new_b);
  double calculate(double x); //Оставили сигнатуры методов
};
//Реализация должна быть после методов
void LinearFunction::setCoefficients(double new_a, double new_b){
  a = new_a; b = new_b;}
double LinearFunction::calculate(double x){return a*x + b;}
```

Объявления и определения методов по файлам

Как и для обычных функций, определение и объявление методов можно разделять по файлам

```
//LinearFunction.h
class LinearFunction{
private:
  double a = 0.0;
  double b = 0.0;
public:
  void setCoefficients(double, double);
  double calculate(double);
};
```

```
//LinearFunction.cpp
#include "LinearFunction.h"
void LinearFunction::setCoefficients(
     double new_a, double new_b){
  a = new_a;
  b = new_b;
}
double LinearFunction::calculate(
     double x){
  returna*x + b;
}
```

Неявный указатель **this**

- Неявный параметр является указателем типа класса и имеет имя **this**, и у каждого объекта содержит указатель на самого себя
- Методы реализованы как обычные функции, имеющие дополнительный параметр
- Можно считать, что настоящая сигнатура методов следующая:

```
class Point2D{
  double x,y;
public:
  void shift(/*Point2D* this, */ int x, int y){
    this->x += x;
    this->y += y;}
  Point2D* self(/*Point2D* this*/){
    return this;}
};
```

- Позволяет избежать перекрытия имен
- Объект может возвращать указать на самого себя

Константные методы

- Методы классов и структур могут быть помечены модификатором **const**
- Такие методы не могут изменять поля объекта
- Указатель **this** является **Type const * this** в константных методах
- У константных объектов можно вызывать только константные методы
- Является частью сигнатуры метода

```
class Example{
  void method() const {
      // ...
  }
};
```

Константная перегрузка

- Перегрузка через **const** позволяет делать версии для константных и не константных объектов

```
class IntArray{
  int data[10] = {1,2,3,4,5,6,7,8,9,0};
public:
  int& get(size_t index){
    return data[index];
  }
  int get(size_t index) const {
    return data[index];
  }
  void print() const{
    for(size_t i = 0; i < 10; ++i)
      std::cout << data[i] << ' ';
    std::cout << '\n';
  }
};
```

```
void change_func(const IntArray& a){
  a.get(4) = 42; //ошибка в любом случае
}

int main(){
  IntArray a1;
  a1.get(3) = 333;
  a1.print();
  const IntArray a2;
  a2.get(3) = 333; //ошибка
  a1.get(3) = a2.get(7);
  a1.print();
  return 0;
}
```

Ключевое слово static (поля класса)

- Модификатор **static** создает поле класса, разделяемое между всеми объектами класса
- **static** поле инициализируется во время запуска программы
- Изменение **static** поля в одном объекте класса видно во всех остальных объектах класса

```
struct Common{
  static int value;
};
//static поле инициализируется отдельно в том же файле где объявлена
int Common::value = 333;

int main(){
  std::cout << Common::value << '\n'; //Получение значение через класс
  Common o1, o2;
  o1.value = 1010; //Изменение через объект
  std::cout << o2.value; //Изменения видны в другом объекте
}
```

Ключевое слово static (методы)

- **static** метод класса «не привязан» к объектам класса
- **static** метод класса может обращаться только к **static** полям класса
- **static** метод не имеет доступа к идентификатору **this**
- **static** методы “нарушают” принципы ООП

```
class Counter{
  static size_t counter;
public:
  static size_t calls_num(){
    return counter;
  }
  void call(){
    //обычный метод может менять static поле
    counter++;
  }
};
size_t Counter::counter = 0;
```

```
int main(){
  //вызов через идентификатор класса
  std::cout << Counter::calls_num() << '\n';
  Counter c;
  c.call();
  c.call();
  //вызов через объект класса
  std::cout << c.calls_num << '\n';
  return 0;
}
```

Создание и уничтожение объектов

Конструктор

- Специальная функция объявленная в классе
- Имя функции совпадает с именем класса
- Не имеет возвращаемого значения
- Предназначена для инициализации объектов

```
class LinearFunction{
private:
  double a = 0.0;  double b = 0.0;
public:
  LinearFunction(double a, double b){ this→a = a; this→b = b; }
  double calculate(double x){return a*x + b;}
};

int main(){
  LinearFunction lin_func(0.5, 1);
  std::cout << lin_func.calculate(2);
}
```

Списки инициализации

- Предназначены для инициализации полей при создании объекта
- Инициализация происходит в порядке объявления полей в классе
- Позволяет инициализировать константные поля

```
class LinearFunction{
private:
  const double a = 0.0;
  const double b = 0.0;
public:
  LinearFunction(double a, double b){
    this→a = a; //Попытка изменения const
    this→b = b; //Попытка изменения const
  }
};
```

```
class LinearFunction{
private:
  const double a = 0.0;
  const double b = 0.0;
public:
  LinearFunction(double a, double b)
    :a(a),b(b){ //Перекрытия имен не будет
  }
};
```

Конструктор по умолчанию

- Создается компилятором, только если в классе отсутствует конструктор
- Не имеет аргументов

```
class Segment{
  Point p1;
  Point p2;
public:
  Segment(Point p1, Point p2)
    :p1{p1},p2{p2}{}
};


int main(){
  Segment s1; //ОШИБКА - нет конструктора
  Segment s2({1,2},{3,4});
}
```

```
class Segment{
  Point p1;
  Point p2;
public:
  Segment(Point p1, Point p2)
    :p1{p1},p2{p2}{}
  Segment(){}
};

int main(){
  Segment s1; //OK - есть конструктор
  Segment s2({1,2},{3,4});
}
```

Ключевое слово **default**

Позволяет явно задать конструктор по умолчанию

```
class Segment{
  Point p1;
  Point p2;
public:
  Segment() = default;
  Segment(Point p1, Point p2):p1{p1},p2{p2}{}
};

int main(){
  Segment s1; //OK - есть конструктор
  Segment s2({1,2},{3,4});
}
```

Делегирующий конструктор

- Позволяет вызывать конструктор из конструктора того же класса
- Сокращает дублирование кода

```
class Point{
  int x;
  int y;
public:
  Point(int x, int y)
    :x(x),y(y){
    std::cout << x << ' ' << y << '\n';
  }
  Point(double y)
    :x(0),y(int(y)){
    std::cout << x << ' ' << y << '\n';
  }
};
```

```
class Point{
  int x;
  int y;
public:
  Point(int x, int y)
    :x(x),y(y){
    std::cout << x << ' ' << y << '\n';
  }
  Point(double y)
    :Point(0, int(y)){ //Вызов конструктора
  }
};
```

Деструктор

- Специальный метод, объявленная в классе
- Имя функции совпадает с именем класса, плюс знак ~ в начале
- Не имеет возвращаемого значения и аргументов
- Предназначен для освобождения используемых ресурсов
- Вызывается автоматически при удалении экземпляра класса / структуры

```
class SecretValue{
  int* value;
public:
  SecretValue(int x):value(new int(x)){};
  ~SecretValue(){
    delete value;
  }
};
```

Копирование и перемещение

- Копирование - создание объекта идентичного второму объекту. Второй объект остается неизмененным.
- Перемещение - создание объекта с “получением” всех характеристик второго объекта. Состояние второго объекта может быть неопределено.

- По умолчанию - производится поверхностное копирование.
- Для глубокой копии или если есть захватываемые ресурсы, то необходимо самостоятельно определять процесс копирования и перемещения.

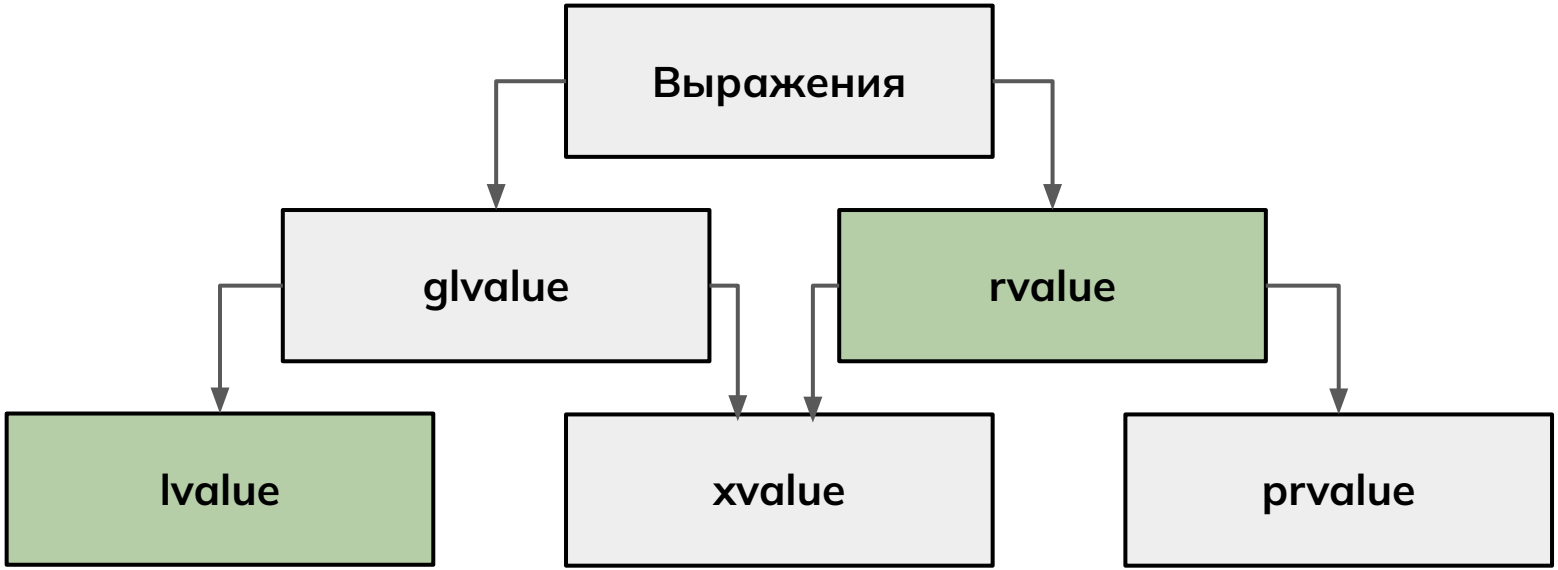
Проблема поверхностного копирования

```
class Secret{
  int* value;
public:
  Secret(int x):value(new int(x)){};
  ~Secret(){
    delete value;
  }
  int& what(){
    return *value;
  }
};
```

```
int main(){
  Secret my_secret(10);
  //Создаем путем копирования
  Secret other_secret = my_secret;
  //Меняем значение у копии
  other_secret.what() = -1;
  //Изменения будут видны
  std::cout << my_secret.what();
  //В конце будет двойна
  //Очистка памяти
}
```

Категории выражений

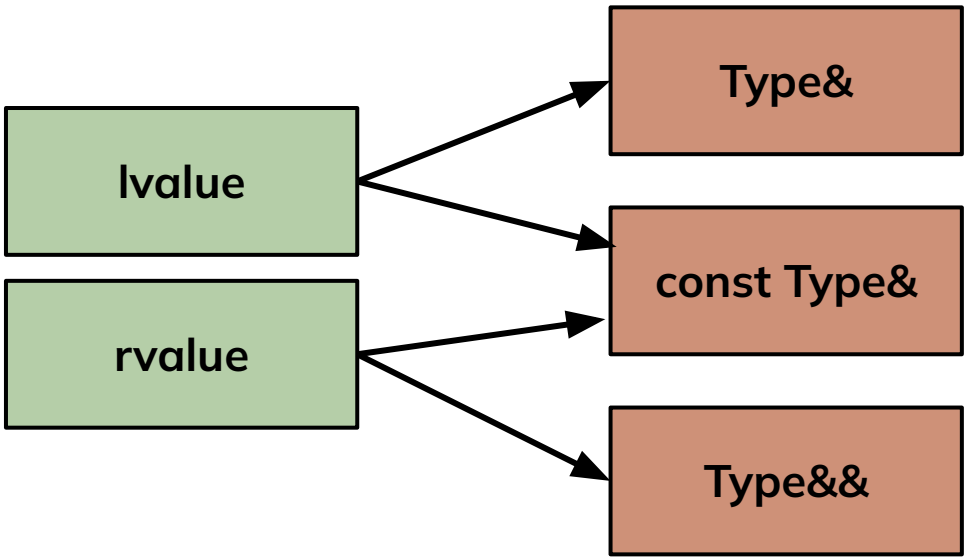
Выражение - оператор с операндами, литерал, имя переменной.
Определяется типом и категорией



lvalue (не **const**) может быть слева в операторе =, а **rvalue** не может

lvalue и rvalue ссылки

- Ссылка на **lvalue** имеет вид: **[const] type &**
- Ссылка на **rvalue** имеет вид: **type &&**



Конструктор копирования

- Позволяет определить, каким образом будет происходить копирование объекта класса
- Правило: если определен конструктор копирования, то необходимо реализовать оператор присваивания с копированием
- **Конструктор вызывается при создании объекта.**
  Оператор присваивания если объект уже существует.

- Сигнатура конструктора копирования:
  Classname(const Classname&)
- Сигнатура оператора присваивания с копированием:
  Classname& operator = (const Classname&)

Конструктор копирования пример

```
class Secret{
  int* value;
public:
  Secret(int x):value(new int(x)){};
  Secret(const Secret& object){
    //Выделяем новую память и записываем копию значения
    value = new int(*object.value);
  }
  Secret& operator = (const Secret& object){
    if(this != &object){ //проверка на самоприсваивание
      delete value; //освобождаем старые ресурсы
      value = new int(*object.value);
    }
    return *this;
  }
};
```

Конструктор перемещения

- Позволяет избегать излишнего копирования
- Основывается на move-семантике (**std::move** и **std::swap**)
- Правило: если определен конструктор перемещения, то необходимо реализовать оператор присваивания с перемещением
- swap - меняет местами значения. move - перемещает значения откуда-то

- Сигнатура конструктора перемещения:
  Classname(Classname&&)
- Сигнатура оператора присваивания с перемещением:
  Classname& operator = (Classname&&)

Конструктор перемещения пример

```
class Secret{
  int* value;
public:
  Secret(int x):value(new int(x)){};
  Secret(Secret&& object):value(nullptr){
    std::swap(value, object.value);
  }
  Secret& operator = (Secret&& object){
    std::swap(value, object.value);
    return *this;
  }
};
```

Методы генерируемые компилятором

- Конструктор по умолчанию - если нет других конструкторов
- Конструктор копирования и оператор присваивания с копированием
- Конструктор перемещения и оператор присваивания с перемещением
- Деструктор

```
class Secret{
  int* value;
public:
  Secret():Secret(0){};
  Secret(int x):value(new int(x)){};
};
int main(){
  Secret a; //Создание по умолчанию
  Secret b = a; //Создание через копирование
  a = b; //Копирование
  Secret c = std::move(b); //Создание через перемещение
}
```

Запрет на копирование и перемещение

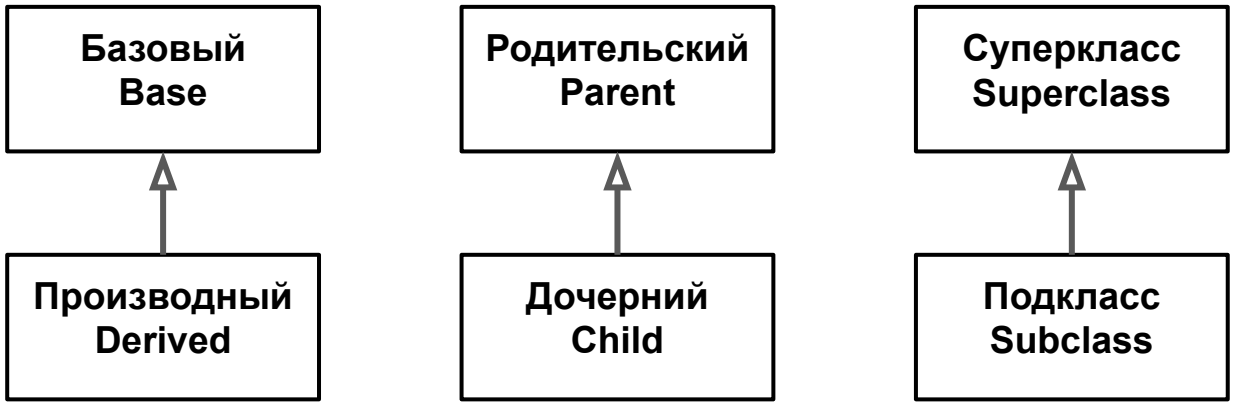
- Можно удалить конструкторы указав **= delete**
- Также можно удалить конструктор по умолчанию

```
class Printer{
  std::string last_str;
public:
  Printer():last_str("---"){}
  Printer(const Printer&) = delete;
  Printer(Printer&&) = delete;
};
```

Наследование

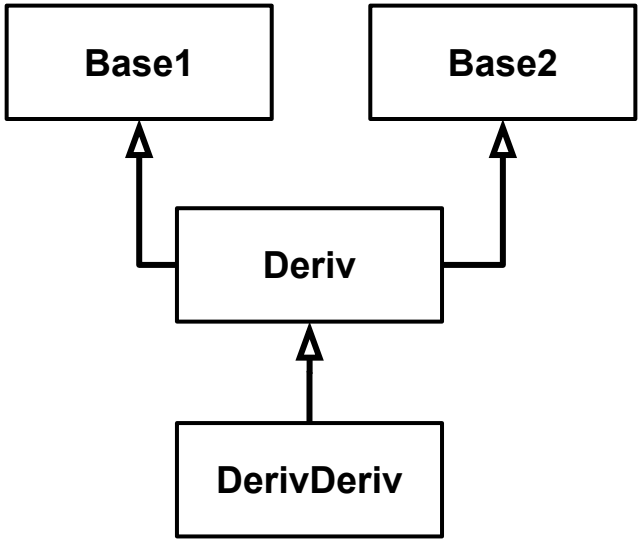
Наследование

- Один из основных механизмов ООП
- Позволяет создавать классы на основе существующих
- Изменяет и/или **расширяет** функционал тех классов, на основе которых создаются другие классы
- В C++ допускается множественное наследование



Наследование - синтаксис

- Для указания базовых классов необходимо после названия класса перечислить все базовые классы




```
class Base1{};

class Base2{};

class Deriv: public Base1, public Base2{};

class DerivDeriv: public Deriv{};
```

Приведение к базовому типу

- Это возможно за счет одинакового расположения полей

```
struct Base{
  int base_value;
};
struct Deriv: public Base{
  int deriv_value;
};

int main(){
  Base b1 = Deriv{333,42}; //OK, но так делать плохо
  Deriv d2{333,42};
  Base& b2 = d2; //OK
  Base* b3 = new Deriv(); //OK
  Deriv* d4 = new Base(); //Ошибка
}
```

Типы наследования

- Базовый класс может быть объявлен с одним из следующих модификаторов доступа:
  - **private**
  - **protected**
  - **public**

- Приватные члены базового класса недоступны в дочернем ни при каком типе наследования
- Накладывает ограничение по приведению к типу базового класса
- Конструкторы и деструкторы не наследуются явно

Доступ к полям при наследовании

- Происходит понижение к самому ограничивающему модификатору доступа:
  - **public** наследование ничего не меняет
  - **protected** наследование делает из **public** полей **protected**
  - **private** наследование делает все поля **private**

Тип наследования	Область видимости базового класса		
	`public`	`protected`	`private`
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	`private`	`private`	`private`

Порядок конструирования объектов

- Объекты создаются «снизу-вверх» - от базовых к производным

- Порядок вызовов конструкторов:
  a. Конструкторы виртуальных базовых классов
  b. Конструкторы прямых базовых классов
  c. Конструкторы полей
  d. Конструктор класса

- Деструкторы вызываются в обратном порядке

```
class Base{
public:
  Base(){std::cout << "Base\n";}
  ~Base(){std::cout << "~Base\n";}
};
class Field{
public:
  Field(){std::cout << "Field\n";}
  ~Field(){std::cout << "~Field\n";}
};
class Deriv: public Base{
  Field f;
public:
  Deriv(){std::cout << "Deriv\n";}
  ~Deriv(){std::cout << "~Deriv\n";}
};
int main(){Deriv obj;}
```

Вызов конструкторов базового класса - проблема

- В данном случае ошибка, так как конструктор явно не наследуется, и нельзя вызвать конструктор производного класса с параметром

```
class Base{
public:
  Base(int a){}
};
class Deriv: public Base{
public:
  Deriv(){}
};
int main(){
  Deriv obj1(4); //Нет подходящего конструктора
  Deriv obj2{}; //Не может вызвать конструктор базового класса без аругментов
  return 0;
}
```

Вызов конструкторов базового класса - решение

- Необходимо явно вызывать конструктор базового класса в списке инициализации

```
class Base{
public:
  Base(int a){}
};
class Deriv: public Base{
public:
  Deriv():Base(333){}
  Deriv(int b):Base(b){}
};
int main(){
  Deriv obj1(4); //OK
  Deriv obj2{}; //OK
  return 0;
}
```

Вызов методов при наследовании

- Модификаторы доступа работают для методов также как для полей:
  - **private** - нельзя использовать
  - **protected** - наследник может использовать
  - **public** - можно использовать вне класса наследника

- Оператор :: может явно указать вызов метода базового класса

```
class Base{
private:
  void baseF1(){}
protected:
  void baseF2(){}
public:
  void baseF3(){}
};
class Deriv: public Base{
public:
  void derivF(){
    baseF1(); // Ошибка
    baseF2(); // OK
  }
};
int main(){
  Deriv obj;
  obj.derivF(); //OK
  obj.baseF3(); //OK
  obj.baseF2(); //Ошибка
  obj.Base::baseF3(); // Можно указать какую ф-цию вызывать
}
```

Переопределение метода

- Переопределение метода – создание метода в дочернем классе с сигнатурой, совпадающей с методом в родительском классе

- Для определения вызываемого метода компилятор сначала ищет его в дочернем классе, если не находит, то ищет в базовых классах по цепочке наследования

- Переопределение функций позволяет изменять поведение базового класса. Является статическим полиморфизмом.

Пример переопределения

```
class Base{
public:
  void print(){std::cout << "Hello, this is Base\n";}
  int calculate(int a, int b){return a + b;}
};
class Deriv: public Base{
public:
  void print(){
    std::cout << "Hello, this is Deriv and value is ";
    std::cout << calculate(3,111) << '\n'; //используется метод Deriv
  }
protected:
  int calculate(int a, int b){return a * b;} //переопределили закрытым
};
```

Пример переопределения - продолжение

```
int main(){
  Base b;
  b.print();
  std::cout << b.calculate(21,21) << '\n';
  Deriv d;
  d.print();
  std::cout << d.calculate(11,11) << '\n'; //Ошибка, видит метод закрытым
  std::cout << d.Base::calculate(11,11) << '\n'; //OK
  Base& b_ref = d;
  b_ref.print(); //Используется метод Base
}
```

Проблема переопределения методов

```
class Base{
public:
  void print(int a){std::cout << a << '\n';}
  void print(char a){std::cout << a << '\n';}
};
class Deriv: public Base{
public:
  void print(std::string_view a){std::cout << a << '\n';}
};
int main(){
  Deriv obj;
  obj.print(333); //ОШИБКА
  obj.print('R'); //ОШИБКА
  obj.print("TEXT");
}
```

Ключевое слово **using**

- Позволяет не переопределять каждую перегруженную функцию базового класса
- Позволяет задать использование базового конструктора в качестве дочернего
- Позволяет изменять спецификатор доступа функций

```
class Base{
public:
  Base(int a = 0){}
};
class Deriv: public Base{
public:
  using Base::Base;
};
int main(){
  Deriv obj1(4); //OK
  Deriv obj2{}; //OK
  return 0;
}
```

Деструкторы при наследовании

Так как базовый класс не знает про производный, то нельзя просто так вызвать метод производного класса через идентификатор базового

```
class Base{
public:
  Base(){std::cout << "B\n";}
};
class Deriv: public Base{
    int* data;
public:
  Deriv():Base(),data(new int[10]){std::cout << "D\n";}
   ~Deriv(){delete [] data; std::cout << "DEL\n";}
};
int main(){
  Base* obj = new Deriv();
  delete obj; //деструктор Deriv не будет вызван
  return 0;
}
```

Виртуальные методы

Виртуальные методы

- Методы класса, которые предполагается переопределить в производных классах
- Вызывает метод производного класса даже через ссылку или указатель на базовый класс
- Модификатор **virtual** располагается перед типом возвращаемого значения
- Метод должен быть определен виртуальным в месте первого объявления
- Основной механизм динамического полиморфизма

Виртуальные методы

- Определение конкретного типа объекта не требуется
- Вызов необходимого метода определяется в **runtime** - динамический полиморфизм

```
class Base{
public:
  virtual void print(){std::cout << "BASE\n";}
};
class Deriv: public Base{
public:
  void print(){std::cout << "DERIV\n";}
};
int main(){
  Base* obj1 = new Base;
  obj1→print(); //Вывод - "BASE"
  Base* obj2 = new Deriv;
  obj2→print(); //Вывод - "DERIV"
}
```

Полиморфный класс

- Любой класс, содержащий по крайней мере один виртуальный метод, является полиморфным

- Каждый объект такого класса знает про таблицу виртуальных методов (**vtable**)

- При использовании ссылки / указателя разрешение методов происходит динамически в момент вызова

Аргументы по умолчанию виртуальных методов

Виртуальные методы могут иметь аргументы по умолчанию, каждый для своего класса, но при вызове будет подставлен аргумент по умолчанию вызывающего класса.

```
class Base{
public:
  virtual void print(int i = 42){std::cout << "Base " << i << '\n';}
};
class Deriv: public Base{
public:
  void print(int i = 333){std::cout << "Deriv " << i << '\n';}
};
int main(){
  Base* obj = new Deriv;
  obj→print(); //Deriv 42
}
```

Чистый виртуальный метод

- Метод, который объявляется в базовом классе, но не имеет в нем определения

- Всякий производный класс обязан иметь свою собственную версию

- Для объявления чистого виртуального метода следует:
    a. Использовать ключевое слово **virtual**
    b. Указать = 0; после списка аргументов
    c. Исключить тело функции - оставить её без реализации

Абстрактный класс

- Любой класс, содержащий по крайней мере один чистый виртуальный метод, является абстрактным
- Предназначен для хранения общей реализации и поведения некоторого множества дочерних классов
- Объекты абстрактного класса создать нельзя
- Рекомендуется добавлять чисто виртуальный деструктор

Использование чистых виртуальных функций

- В C++ отсутствует специальная синтаксическая конструкция для определения интерфейса

- Интерфейсом является класс:
    a. все члены которого объявлены как **public**
    b. содержит только чистые виртуальные методы
    c. не содержит никаких полей-данных

- Каждый интерфейс является абстрактным классом, но не каждый абстрактный класс интерфейс

- При использовании интерфейс реализуют, абстрактный класс наследуют

Реализация интерфейса

```
class IPrinter{
public:
  virtual void print(int value) = 0;
};
class TerminalPrinter: public IPrinter{
public:
  void print(int value){
    std::cout << value << '\n';
  }
};
class FancyPrinter: public IPrinter{
  std::string prefix;
public:
  FancyPrinter(std::sting str):prefix(str){}
  void print(int value){
    std::cout << prefix << ' ' << value << '\n';
  }
};
```

Использование интерфейса

```
void client(IPrinter* printer){
  int x,y;
  std::cin >> x >> y;
  printer→print(x+y);
}

int main(){
  TerminalPrinter simple_printer;
  FancyPrinter fancy_printer("Sum is");
  client(&simple_printer);
  client(&fancy_printer);
}
```

Внедрение зависимости

Dependency Injection (DI)

Внедрение зависимости (DI)

- Техника, при которой объект настраивается внешними сущностями, а не “самонастройкой”

- Позволяет:
    a. Сделать код масштабируемым
    b. Проще контролировать время жизни объекта
    c. Упрощенное тестирование кода

Внедрение зависимости - пример (1)

```
class Calculator{
public:
  void sumPrint(int x, int y){
    std::cout << x + y << '\n';
  }
  void prodPrint(int x, int y){
    std::cout << x * y << '\n';
  }
};

int main(){
  Calculator calc;
  calc.sumPrint(3,4);
  calc.prodPrint(3,4);
}
```

Внедрение зависимости - пример (2)

```
class Operation{
public:
  virtual int calculate(int x, int y) = 0;
};
class SumOperation: public Operation{
public:
  int calculate(int x, int y){return x + y;}
};
class ProdOperation: public Operation{
public:
  int calculate(int x, int y){return x * y;}
};
```

Внедрение зависимости - пример (2.5)

```
class Calculator{
public:
  void calcAndPrint(int x, int y, Operation* op){
    std::cout << op→calculate(x,y) << '\n';
  }
};
void client(Calculator& calc){
  calc.calcAndPrint(3,4,new SumOperation());
  calc.calcAndPrint(3,4,new ProdOperation());
}
int main(){
  Calculator calc;
  client(calc);
}
```

Внедрение зависимости - пример (4)

```
class Calculator{
  Operation* op;
public:
  Calculator(bool is_sum){
    if(is_sum){
      op = new SumOperation();
    }
    else{
      op = new ProdOperation();
    }
  };
  void calcAndPrint(int x, int y){
    std::cout << op→calculate(x,y) << '\n';
  }
};
```

Внедрение зависимости - пример (4.5)

```
void client(Calculator& calc){
  calc.calcAndPrint(3,4);
}

int main(){
  Calculator calc_sum(true);
  Calculator calc_prod(false);
  client(calc_sum);
  client(calc_prod);
}
```

Внедрение зависимости - пример (5)

```
class Calculator{
  Operation* op;
public:
  Calculator(Operation* op):op(op){};
  void calcAndPrint(int x, int y){
    std::cout << op→calculate(x,y) << '\n';
  }
};

void client(Calculator& calc){
  calc.calcAndPrint(3,4);
}
```

Внедрение зависимости - пример (5.5)

- Применение техники DI:
  - Можно легко добавлять новые операции реализуя интерфейс
  - Клиент знает только о данных для вычисления
  - Время жизни объектов согласовано и никакой динам. памяти
  - По аналогии можно реализовать выбор потока для печати

```
int main(){
  SumOperation sum_op;
  ProdOperation prod_op;
  Calculator calc_sum(&sum_op);
  Calculator calc_prod(&prod_op);
  client(calc_sum);
  client(calc_prod);
}
```

Что изучать дальше?

- Идиома RAII
- Паттерны (принципы) GRASP
- Принципы SOLID
- Паттерны проектирования GoF
- Шаблоны в C++
- Лямбда-функции