Лекция 1

Контактная информация

- Email: ksenox94@gmail.com
- Заголовок письма: [OOP_XXXX] <тема письма>
- Не забывайте представляться
- Задавайте конкретные вопросы

Литература

- Р. Лафоре – Объектно-ориентированное программирование в C++
- А. Пол – Объектно-ориентированное программирование на C++
- Э. Гамма и др. – Приемы объектно-ориентированного проектирования
- Б. Страуструп – Язык программирования C++
- М. Фаулер – UML. Основы

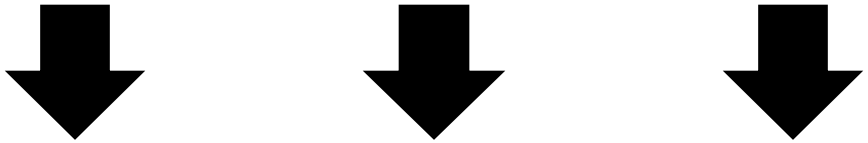
- en.cppreference.com – Документация языка C++

Парадигмы программирования

Парадигма программирования

- Совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию)

Три основных парадигмы



Функциональная

Структурная

Объектно-ориентированная

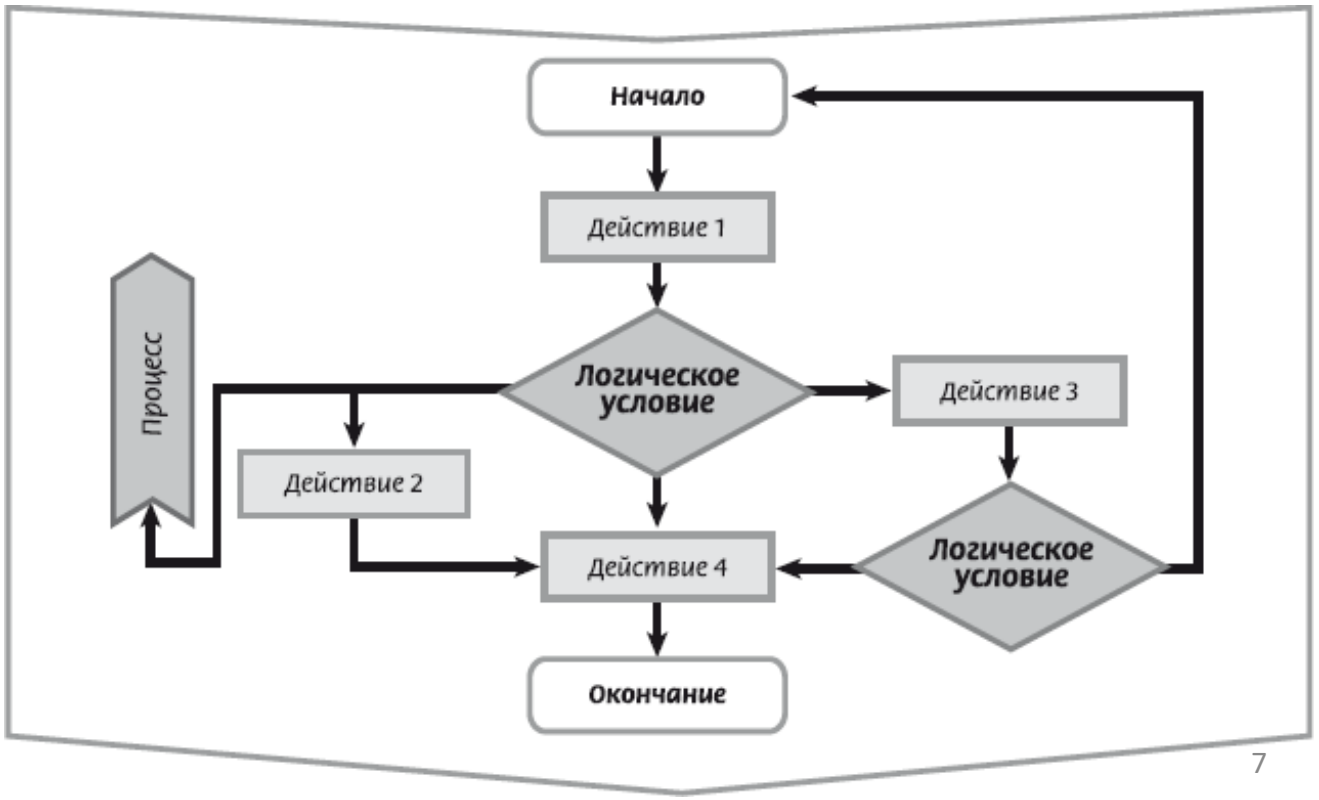
Функциональное программирование 1936г.

- Основывается на λ-исчислении
- Первый язык LISP (изобретатель Джон Маккарти)
- Основное понятие – неизменяемость

```
DEFINE
(((RVRSE,(LAMBDA,(L),(COND,((NULL,L),NIL),
    (T,(CONS,(RVRSE,(CDR,L)),(CONS,(CAR,L),NIL))))))),
(RVDE,(LAMBDA,(L),(REV,L,NIL))),
(REV,(LAMBDA,(J,K),(COND,((NULL,J),K),
    (T,(REV,(CDR,J),(CONS,(CAR,J),K)))))))
()
RVRSE   ((A,B,C,D,E))   ()
RVDE    ((A,B,C,D,E))   ()
```

Структурное программирование 1968г.

- Эдсгер Вибе Дейкстра показал минусы инструкции goto
- Программа должна состоять из:
  - Последовательностей
  - Условий
  - Циклов



ООП 1966г.

- Концепцию предложили Оле-Йохан Даль и Кристен Нюгором
- Появилась из языка ALGOL
  - Сохранение фрейма в динамической памяти
  - Локальные переменные сохранялись после выхода из функции
  - Полиморфизм через указатели на функции





Принципы ООП	
Абстракция	Инкапсуляция
Наследование	Полиморфизм

Абстракция

- Отображение только существенной информации о мире с сокрытием деталей и реализации
- Выделение интерфейса
- Единицей абстракции может быть класс или файл

Инкапсуляция

- Связь кода и данных
- Защита от внешнего воздействия
- Основа инкапсуляции в ООП – класс
- Простое сокрытие данных – не инкапсуляция

Наследование

- Механизм, с помощью которого один объект перенимает свойства другого
- Позволяет добавлять классу характеристики, делающие его уникальным
- Поддержка понятия иерархической классификации
- Уменьшение количества дублирующего кода

Полиморфизм

- Реализация принципа: Один интерфейс – множество реализаций
- Механизм, позволяющий скрыть за интерфейсом общий класс действий
- Виды полиморфизма:
  - Статический
  - Динамический
  - Параметрический (шаблонный)

Указатели и ссылки

Указатели в C++

- Переменная, хранящая адрес некоторой ячейки памяти

```
int value = 0;
int* pointer = 0;
```

- Нулевому указателю не соответствует никакая ячейка памяти
- Для работа с указателем используются операторы:
  1. & - взятие адреса
  2. * - получение значения по адресу (разыменовывание)

```
int value = 0;
int *pointer = &value; // 1 - взятие адреса
*pointer = 42;         // 2 - разыменовывание
```

Передача аргументов

По значению
Работа происходит
с локальными копиями

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
int main()
{
    int k = 10;
    int m = 20;
    swap(k, m);
    std::cout << k << ' ' << m << '\n';
}
```

Через указатель
Работа происходит с адресами

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main()
{
    int k = 10;
    int m = 20;
    swap(&k, &m);
    std::cout << k << ' ' << m << '\n';
}
```

Возврат значения через указатель

```
bool findMaxElement(int *start, int *end, int *max_element)
{
    if (start == end)
        return false;
    *max_element = *start;
    for (; start != end; ++start)
        *max_element = *start > *max_element ? *start : *max_element;
    return true;
}
int main()
{
    int arr[10] = {0, 1, 2, 3, 9, 4, 5, 6, 7, 8};
    int max_element = 0;
    if (findMaxElement(arr, arr + 10, &max_element))
        std::cout << "Maximum = " << max_element << '\n';
    else
        std::cout << "Array is Empty\n";
}
```

Недостатки указателей

- Загрязнение кода операторами * и &
- Отсутствует требование обязательной инициализации
- Допустимость нулевого значения
- Арифметика указателей сильное, но опасное средство

Ссылки в C++

- Исправляют некоторые недостатки указателей
- По факту являются оберткой над указателем
- Уменьшают количество операторов разыменования и взятия адреса

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
int main(){
    int k = 10;
    int m = 20;
    swap(k, m);
    std::cout << k << ' ' << m << '\n';
}
```

Различия ссылок и указателей

- Ссылка не может быть не инициализированной

```
int* pointer;   //* OK
int& link;      //! Ошибка
```

- У ссылки нет нулевого значения

```
int* pointer = 0;   //* OK
int& link = 0;      //! Ошибка
```

- Нельзя создать массивы ссылок

```
int* pointer_array[10];   //* OK
int& link_array[10];      //! Ошибка
```

Различия ссылок и указателей

- Ссылку нельзя переинициализировать

```
int a = 10;
int b = 20;
int* pointer = &a;  // pointer указывает на переменную a
pointer = &b;       // теперь pointer указывает на переменную b
int& link = a;      // link является ссылкой на переменную a
link = b;           // переменной a присваивается значение переменной b
```

- Нельзя получить адрес ссылки или ссылку на ссылку

```
int value = 10;
int* pointer = &value;       //pointer указывает на переменную value
int** p_pointer = &pointer; //p_pointer указывает на переменную pointer
int& link = value;           //link ссылается на переменную value
int* p_link = &link;         //p_link указывает на переменную value
int&& l_link = link;         //ошибка
```

std::ref и std::reference_wrapper

- ref создает объект типа reference_wrapper
- reference_wrapper является оболочкой над ссылкой
- reference_wrapper можно копировать и присваивать
- Позволяют хранить ссылки в массиве и контейнерах
- Функция cref создает константую ссылку

std::ref пример

```cpp
#include <functional>

void print(int value)
{
    std::cout << value << '\n';
}
int main()
{
    int i = 10;
    auto f1 = std::bind(print, i);           //закрепили i = 10
    auto f2 = std::bind(print, std::ref(i)); //закрепили ссылку на i
    i = 20;
    f1(); //Вывод: 10
    f2(); //Вывод: 20
}
```

std::reference_wrapper пример

```
//инициализация списка
std::list<int> l(10);
std::iota(l.begin(), l.end(), 0);
//вывод списка
for (auto x : l) std::cout << x << ' ';
std::cout << " - list\n";
//создаем вектор с ссылками на элементы списка
std::vector<std::reference_wrapper<int>> v(l.begin(), l.end());
//перемешиваем элементы вектора (список перемешать нельзя)
std::shuffle(v.begin(), v.end(), std::mt19937{std::random_device{}()});
for (auto x : v)
    std::cout << x << ' ';
std::cout << " - vector (randomed list)\n";
//меняем первый элемент списка (0), изменения будут видны в векторе
l.front() = -42;
for (auto x : v) std::cout << x << ' ';
std::cout << " - vector after list change\n";
```

Указатели и const

- Указатель на константу

```
int a = 10;
const int* first_const_ptr = &a;
int const* second_const_ptr = &a;
*first_const_ptr = 10; //Ошибка
second_const_ptr = nullptr;
```

- Константный указатель

```
int * const const_ptr = &a;
*const_ptr = 10;
const_ptr = nullptr;  //Ошибка
```

- Константный указатель на константу

```
const int * const const_ptr = &a;
*const_ptr = 10;       //Ошибка
const_ptr = nullptr;  //Ошибка
```

Ограничения преобразования констант

- Разрешены неявные преобразования T* к T const *
- Запрещены неявные преобразования T** к T const **

```
const int value = 1;
int* pointer = nullptr;
pointer = &value // Ошибка. Т.к. преобразование int const* к int*
int const ** p_pointer = &pointer //Ошибка. Запрещено преобразование
                          //int** к int const **
*p_pointer = &value; //ОК. Т.к. *p_pointer  имеет тип int const*
```

Константные ссылки

- Ссылка сама по себе является неизменяемой

```
int a = 10;
int& const link = a;
int const& const_link = a;
const_link = -10;                    //Ошибка
```

- Позволяет избежать копирования объектов при передаче в функцию и запретить их изменение внутри функции

```
Point2D midPoint(Segment const& seg)
```

Пользовательские типы

Перечисления (enum)

- Контекст для описания диапазона значения
- Переход к категориальным значениям
- Нумерация с нуля, по возрастанию
- Могут быть неявно преобразованы в целочисленные типы, но не наоборот

```
enum {RED, GREEN, BLUE};          //неименованное перечисление
std::cout << GREEN << '\n';       //Вывод: 1
enum color{R, B, G};              //именованное перечисление
color red = R;                    //Можно создать переменную
std::cout << red << '\n';         //Вывод: 0
```

Порядок в перечислении

- Можно задать целое значение для каждого элемента
- Если явно не задано значение, то будет взято предыдущее + 1
- По умолчанию значение первого элемента равно 0

```
enum A{aone, atwo, athree, afour};
std::cout << A::aone << A::atwo << A::athree << A::afour << '\n';
//0123
enum B{bone, btwo = 2, bthree, bfour};
std::cout << B::bone << B::btwo << B::bthree << B::bfour << '\n';
//0234
enum C{cone, ctwo = 2, cthree = 1, cfour};
std::cout << C::cone << C::ctwo << C::cthree << C::cfour << '\n';
//0212
```

Необходимость группировки данных

- Сигнатура функции для подсчета длины отрезка

```
double length(double x1, double y1, double x2, double y2);
```

- Сигнатура функции для нахождения точки пересечения отрезков

```
bool intersect(double x11, double y11, double x12, double y12,
               double x21, double y21, double x22, double y22,
               double *xi, double *yi);
```

- Логически связанные данные: координаты точки и точки отрезка

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);
```

Определение структуры

- Группа связанных переменных
- Составной тип данных
- Имя структуры – спецификатор пользовательского типа
- Член структуры – переменная, которая является частью структуры

```
struct <Имя структуры>
{
    <Тип данных> <Название поля 1>;
    <Тип данных> <Название поля 2>;
};
```

Доступ к элементам структур

- Для обращения к полям используется оператор `.`

```
double length(Segment seg){
    double dx = seg.p1.x - seg.p2.x;
    double dy = seg.p1.y - seg.p2.y;
    return std::sqrt(dx * dx + dy + dy);
}
```

- Для обращения к полям через указатель используется оператор `->`

```
double length(Segment* seg){
    double dx = seg->p1.x - seg->p2.x;
    double dy = seg->p1.y - seg->p2.y;
    return std::sqrt(dx * dx + dy + dy);
}
```

Класс

- Пользовательский тип данных, который задает формат группы объектов
- Связывает данные с кодом
- Функции и переменные, входящие в класс называются его членами:
  - Член данных (поле, атрибут)
  - Функция-член (метод)

Объявление класса

- Используется ключевое слово class
- Синтаксически подобно определению структуры

```
class Human{
    int age;
    std::string name;

public:
    int getAge();
    std::string getName();
};
int main(){
    Human h;
    std::cout << sizeof(Human) << '\n';
    std::cout << sizeof(h) << '\n';
}
```

Модификаторы доступа

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

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

Структуры и классы

- Единственное различие в модификаторе доступа для полей
- В соответствием с формальным синтаксисом C++ объявление структуры создает тип класса
- Структуры в C++ сохранены для совместимости с C

Лекция 2

Методы класса

Методы

- Функции, определенные внутри структуры
- Отличие заключается в прямом доступе к полям структуры
- Обращение к методу аналогично обращению к полям

```
struct Segment{
    Point2D start;
    Point2D end;
    double length(){
        double dx = start.x - end.x;
        double dy = start.y - end.y;
        return sqrt(dx * dx + dy * dy);}
};
int main(){
    Segment segment = {{0.4, 1.4}, {1.2, 6.3}};
    std::cout << segment.length() << '\n';}
```

Реализация метода вне класса

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

```
struct Segment{
    Point2D start;
    Point2D end;
    double length();
};

double Segment::length(){
    double dx = start.x - end.x;
    double dy = start.y - end.y;
    return sqrt(dx * dx + dy * dy);}

int main(){
    Segment segment = {{0.4, 1.4}, {1.2, 6.3}};
    std::cout << segment.length() << '\n';}
```

Объявление и определение методов

- Как и для обычных функций, можно разделять объявление и определение

```
//point.h
struct Point2D {
    double x;
    double y;
    void shift(double x, double y);};
```

```
#include "point.h"
void Point2D::shift(double x, double y){
    this->x += x;
    this->y += y;
}
```

Неявный указатель this

- Методы реализованы как обычные функции, имеющие дополнительный параметр
- Неявный параметр является указателем типа класса и имеет имя this
- Можно считать, что настоящая сигнатура методов следующая:

```
struct Point2D {
    double x;
    double y;
    void shift(/* Point2D *this,*/ double x, double y){
        this->x += x;
        this->y += y; }
};
```

- Позволяет обратиться к полям объекта при перекрытии имен

Перегрузка функций

- Определение нескольких функций с одинаковым именем, но отличающимся списком параметров (типами и/или количеством)

```
class Point2D {
    int x;
    int y;
public:
    void move(int dx, int dy);
    void move(Point2D vector);
} zero;
int main() {
    Point2D point;
    point.move(10, 20);
    zero.move(point);
}
```

Инвариант класса

- Публичный интерфейс – список методов, доступный внешним пользователям класса

- Инвариант класса – набор утверждений, которые должны быть истинны применительно к любому объекту данного класса в любой момент времени, за исключением переходных процессов в методах объекта

- Для сохранения инварианта класса:
  - Все поля должны быть закрытыми
  - Публичные методы должны сохранять инвариант

Определение констант

- Ключевое слово const позволяет определять типизированные константы
- Попытка изменений таких значений пресекается компилятором
- Попытка изменить константные данные приводит к неопределённому поведению

Константные методы

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

Константные методы

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

```
class IntArray {
    int size;
    int *data;
public:
    int get(int index) const {
        return data[index];
    }
    int &get(int index) {
        return data[index];
    }
};
```

Синтаксическая и логическая константность

- **Синтаксическая** – константные методы не могут модифицировать поля (обеспечивается компилятором)
- **Логическая** – запрещено изменение данных, определяющих состояние объекта в константных методах (обеспечивается разработчиком)

```
class IntArray {
    mutable int size; //Нарушение логической константности
    int *data;
public:
    void method() const {
        data[10] = 1; //Нарушение логической константности
    }
};
```

Ключевое слово mutable

- Позволяет определять поля, доступные для изменения внутри константных методов
- Можно использовать только с полями, не являющимися частью состояния объекта

```
class IntArray {
    int size;
    int *data;
    mutable int counter;
public:
    int size() const {
        ++counter;
        return size;
    }
};
```

```
struct Example {
    int n1;
    mutable int n2;
};
int main(){
    const Example a;
    a.n1 = 2; //ошибка
    a.n2 = 2;
}
```

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
- static поля противоречат концепции инкапсуляции

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

Создание объектов

Конструкторы

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

```
class Date {
    int year;
    int month;
    int day;
public:
    void init(int day, int month, int year);
    void setYear(int year);
    void setMonth(int month);
    void setDay(int day);
};
```

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day, int month, int year);
    void setYear(int year);
    void setMonth(int month);
    void setDay(int day);
};
```

Перегрузка конструкторов

- Может быть объявлено несколько конструкторов
- Должны иметь разное количество или тип параметров

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day, int month, int year);
    Date(int day, int month);
    Date(int day);
    Date();
    Date(char const *date);
};
```

Списки инициализации

- Предназначены для инициализации полей
- Инициализация происходит в порядке объявления полей

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day, int month, int year):year(year),day(day),month(month){}
};
```

Значения по умолчанию

- Конструкторы, как и другие функции, могут иметь значения по умолчанию

- Значения параметров по умолчанию необходимо указывать при объявлении

Значения по умолчанию

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day = 1, int month = 1, int year = 1970)
        :year(year), month(month), day(day) {}
    Date() {} // ОШИБКА
};

int main() {
    Date zero;
    Date days (10);
    Date daysAndMonths (10, 2);
}
```

Конструктор от одного аргумента

- Задает неявное преобразование от значения аргумента к значению класса/структуры

```
class Segment {
    Point first;
    Point second;
public:
    Segment() : first(0, 0), second(0, 0) {}
    Segment(int length) : first(0, 0), second(length, 0) {}
};

int main() {
    Segment first;
    Segment second(10);
    Segment third = 20;
}
```

Ключевое слово explicit

- Запрещает неявное преобразование

```
class Segment {
    Point first;
    Point second;
public:
    Segment() : first(0, 0), second(0, 0) {}
    explicit Segment(int length) : first(0, 0), second(length, 0) {}
};

int main() {
    Segment first;
    Segment second(10);
    Segment third = 20; // Compile error
}
```

Конструктор по умолчанию

- Создается компилятором
- Только если в классе отсутствует конструктор
- Не имеет параметров

```
class Segment {
    Point first;
    Point second;
public:
    Segment(Point first, Point second) : first(first), second(second) {}
};

int main() {
    Segment first; //Compile Error
    Segment second(Point(), Point(1, 2));
}
```

Ключевое слово default

- Позволяет явно задать конструктор по умолчанию

```
class Segment {
    Point first;
    Point second;
public:
    Segment() = default;
    Segment(Point first, Point second) : first(first), second(second) {}
};

int main() {
    Segment first;
    Segment second(Point(), Point(1, 2));
}
```

Делегирующий конструктор

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

```
class Point {
    int x;
    int y;
public:
    explicit Point(int x = 0, int y = 0)
        :x(x), y(y){
        cout << x << " " << y << endl;
    }
    explicit Point(double y)
        :x(0), y(int(y)){
        cout << x << " " << y << endl;
    }
};
```

```
class Point {
    int x;
    int y;
public:
    explicit Point(int x = 0, int y = 0)
        :x(x), y(y){
        cout << x << " " << y << endl;
    }
    explicit Point(double y)
        :Point(0, int(y)) {}
};
```

Деструктор

- Специальная функция, объявляемая в классе
- Имя функции совпадает с именем класса, плюс знак ~ в начале
- Не имеет возвращаемого значения и аргументов
- Предназначены для освобождения используемых ресурсов
- Вызывается автоматически при удалении экземпляра класса / структуры

```
class IntArray {
    int _size;
    int *data;
public:
    explicit IntArray(int size)
        :_size(size), data(new int[_size]){}
    ~IntArray(){
        delete []data;
    }
};
```

Методы, генерируемые компилятором

- Конструктор по умолчанию
- Конструктор копирования
- Конструктор перемещения
- Операторы присваивания
- Деструктор

- Запретить создание можно спецификатором delete

```
class SomeClass{
public:
    SomeClass() = delete;
};
```

Копирование и перемещение объектов

lvalue и rvalue

- lvalue (locator value) представляет собой объект, который занимает идентифицируемое место в памяти (имеет имя и адрес)
- rvalue – всё, что не является lvalue

```
int main(){
    int a = 10; int b = 20;                          //1
    int arr[10] = {1, 2, 3, 4, 5, 5, 4, 3, 2, 1};    //2
    int &link1 = a;                                  //3
    int &link2 = a + b;                              //4
    int &link3 = *(arr + a / 2);                     //5
    int &link4 = *(arr + a / 2) + 1;                 //6
    int &link5 = (a + b > 10) ? a : b;               //7
}
```

Примеры

- Выражения в C++ делятся на:
  1. lvalue - выражения, значения которых являются ссылкой
  2. rvalue - временные значения, не соответствующие какой-то переменной / элементу массива
- Указатели и ссылки могут использоваться только с lvalue

Корректно

```
int var;
var = 4;
```

Не корректно

```
4 = var;          // Ошибка
(var + 1) = 4;    // Ошибка
```

Примеры

```
int foo() {
    return 2;
}

int main() {
    foo() = 2;
    return 0;
}
```

```
int globalvar = 20;

int& foo() {
    return globalvar;
}

int main() {
    foo() = 10;
    return 0;
}
```

```
int& foo() {
    return 2;
}
```

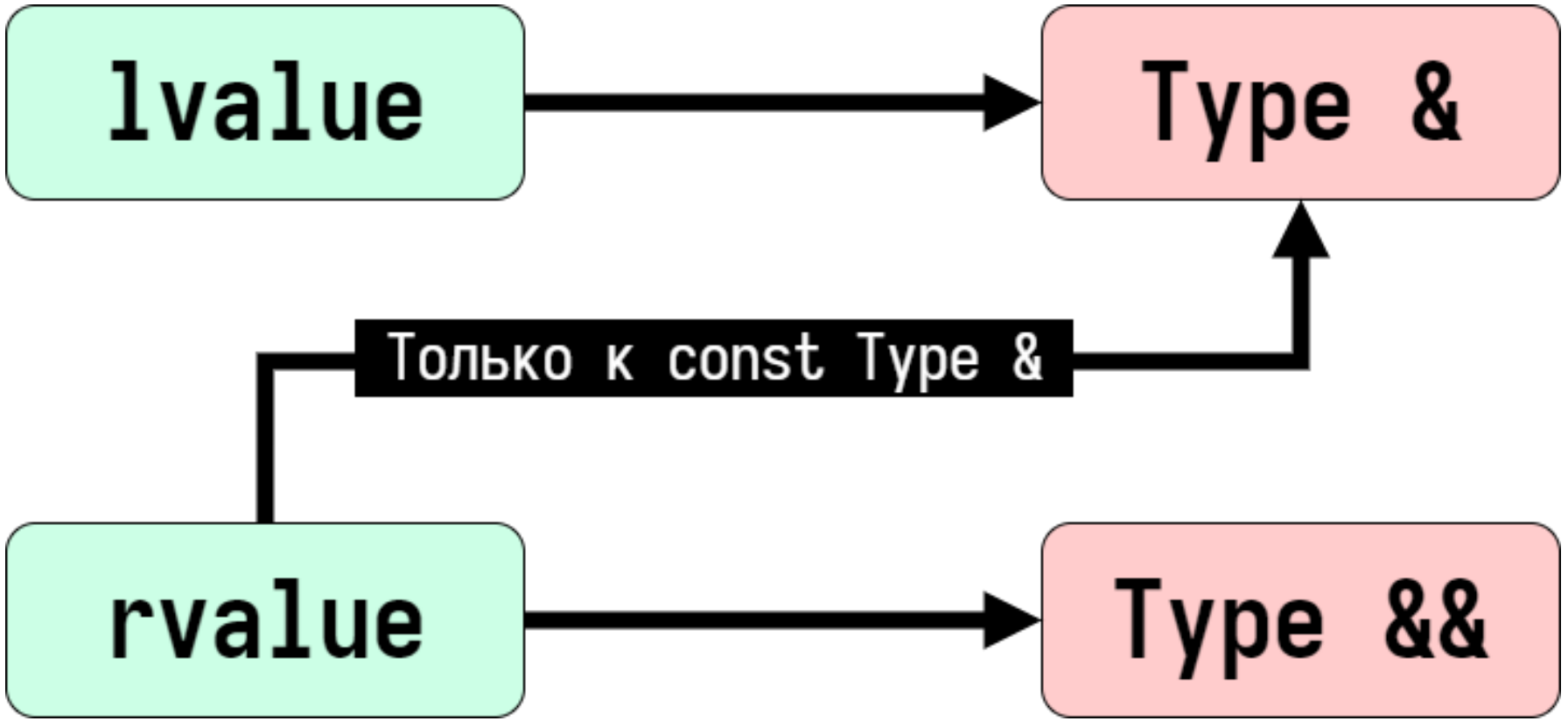
Преобразования между lvalue и rvalue

- Все операции со значениями требуют rvalue в качестве аргументов

```
int a = 1;  // a - lvalue int
b = 2;      // b - lvalue int
c = a + b;  // '+' требует rvalue
```

- В rvalue могут быть преобразованы все lvalue, которые не являются массивом, функцией и не имеют неполный тип
- Оператор * принимает rvalue и возвращает lvalue
- Оператор & принимает lvalue и возвращает rvalue
- Const ссылка может ссылаться на rvalue. Время жизни rvalue расширяется

lvalue и rvalue ссылки



Конструктор копирования

- Позволяет определить, каким образом будет происходить копирование объекта класса
- Правило, если реализован конструктор копирования, то необходимо реализовать оператор присваивания с копированием

```
class SomeClass{
public:
    SomeClass(const SomeClass& obj){
        //...
    }
};
```

Конструктор перемещения

- Позволяет избегать излишнего копирования
- Основывается на move-семантике (std::move и std::swap)

```
class SomeClass{
public:
    SomeClass(SomeClass&& obj){
        //...
    }
};
```

RAII

Идиома программирования

- Устойчивый способ выражения некоторой составной конструкции в языках программирования
- Шаблон решения задачи, алгоритма или структуры данных путем комбинирования встроенных элементов языка
- Может выглядеть по-разному в разных языках, либо в ней может не быть надобности в некоторых из языков

Идиома RAII

- Resource Acquisition Is Initialization - получение ресурса есть инициализация
- Идиома объектно-ориентированного программирования
- Основная идея - с помощью механизмов языка получение ресурса неразрывно совмещается с инициализацией , а освобождение – с уничтожением объекта
- Типичный способ реализации - получение доступа в конструкторе, а освобождение в деструкторе

- Применяется для:
  - Выделения памяти
  - Открытия файлов / устройств / каналов
  - Мьютексов / критических секций / других механизмов блокировки

Пример реализации RAII на C++

```
class File {
    std::FILE *file;
public:
    //Захват ресурса
    File(const char *filename) : file(std::fopen(filename, "w+")) {
        if (!file)
            throw std::runtime_error("file open failure");
    }
    //Освобождение ресурса
    ~File() {
        std::fclose(file);
    }
    //Взаимодействие с ресурсом
    void write(const char *data) {
        if (std::fputs(data, file) == EOF)
            throw std::runtime_error("file write failure");
    }
};
```

Пример реализации RAII на C++

```
int f(){
    try{
        File file("log.txt");
        file.write("information");
    }
    catch(std::runtime_error &e){
        e.what();
    }
}
```

Отношения классов

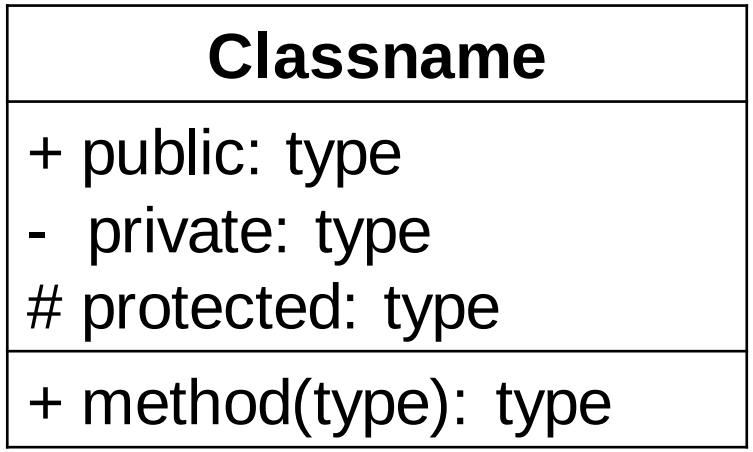
UML

https://www.uml-diagrams.org/

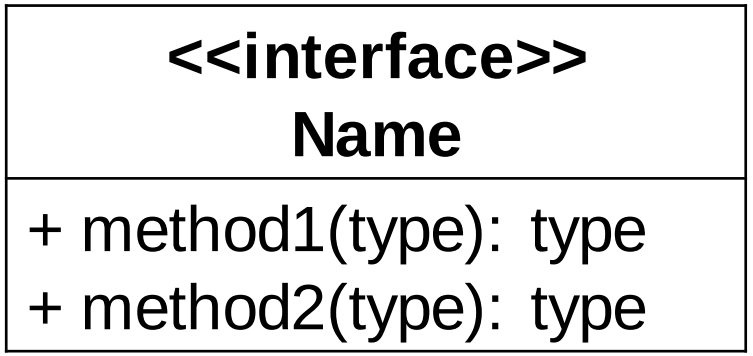
- UML – унифицированный язык моделирования (Unified Modeling Language)
- Включает в себя такие диаграммы как:
  - Диаграмма классов
  - Диаграмма пакетов
  - Диаграмма объектов
  - Диаграмма развертывания
  - Use-case диаграмма
  - Диаграмма состояний
  - Диаграмма последовательностей

Структурные сущности

Класс

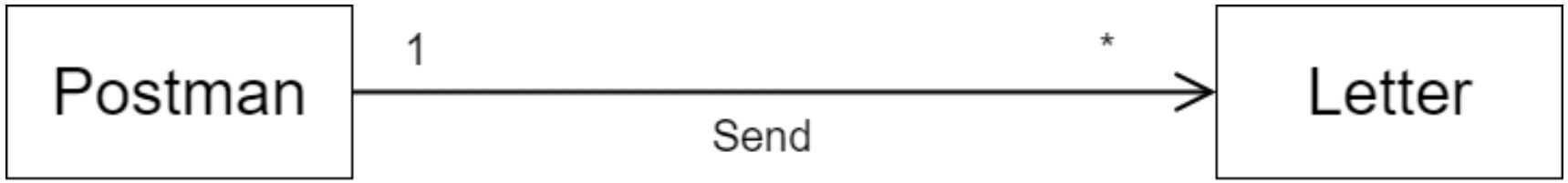


Интерфейс



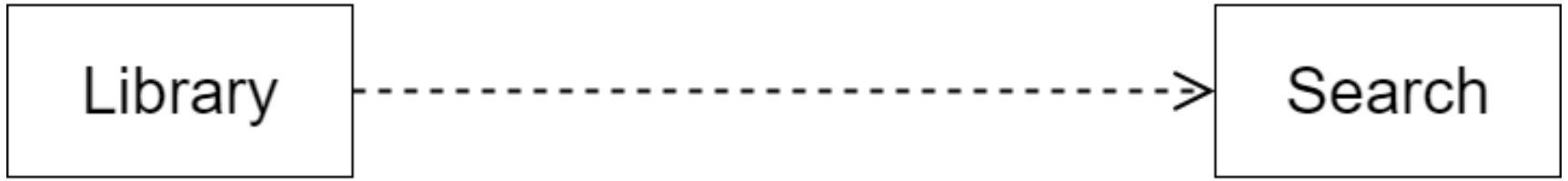
Ассоциация

- Означает, что классы связаны физически или логически
- Самая слабая связь



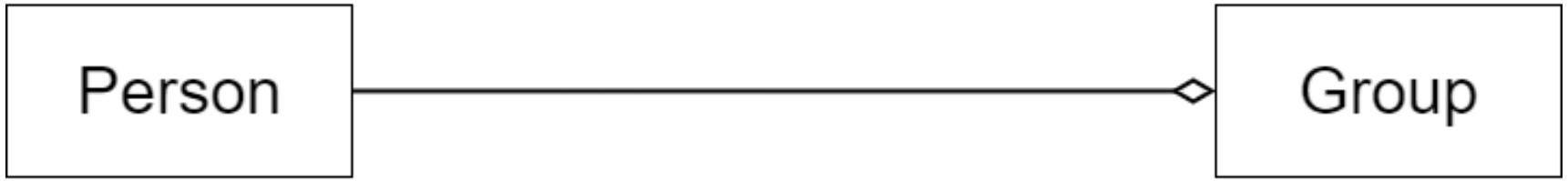
Зависимость

- Означает, что классы связаны напрямую физически
- Зависимость отображает то, что класс хранит другой класс, либо принимает его в качестве аргументов методов



Агрегация

- Определяет отношение HAS A – то есть отношение владения
- Описывает целое и составные части, которые в него в ходят
- Целое НЕ является владельцем части и НЕ управляет временем её жизни

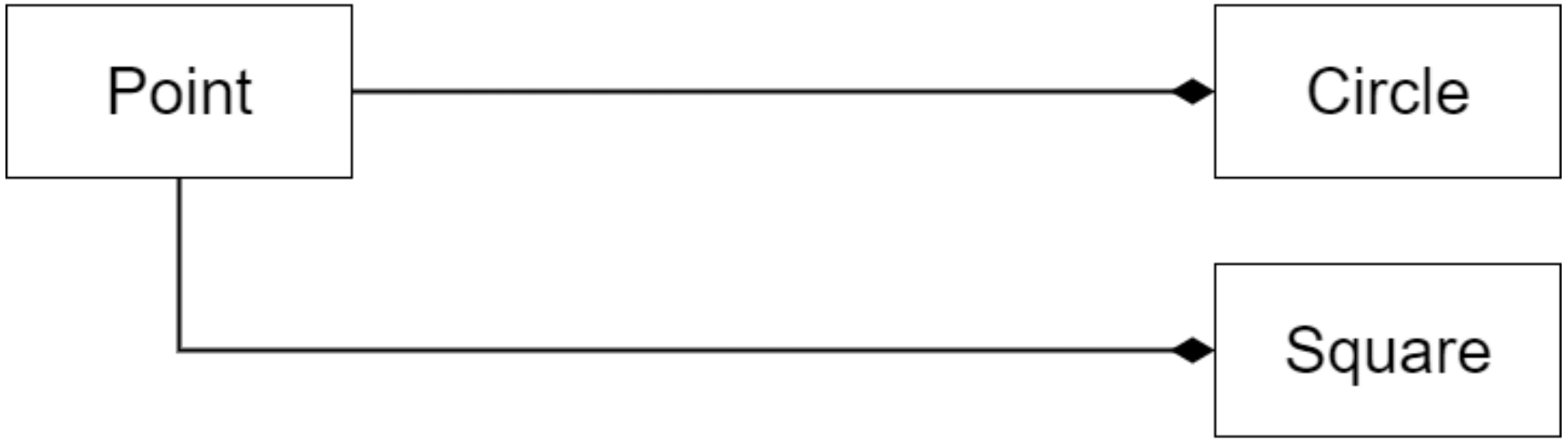


Пример агрегации

```
class Person {
    //...
};

class Group {
    Person *members;
public:
    void addMember(Person *member) {
        // Добавление участника
    }
    void removeMember(Person *member) {
        // Исключение участника
    }
};
```

Композиция

- Определяет отношение HAS A – то есть ношение владения
- Описывает целое и составные части, которые в него входят
- Конкретный экземпляр части может принадлежать только одному владельцу
- Целое управляет временем жизни входящих в него частей



Пример композиции

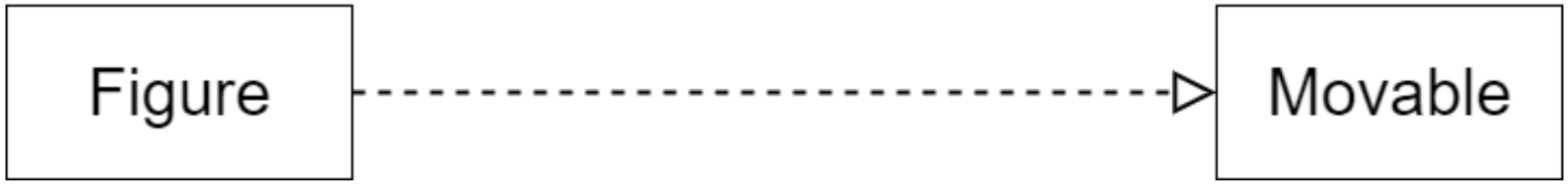
```
class Point {
    int x;
    int y;
public:
    Point(int x, int y): x(x), y(y) {}
};

class Circle {
    Point *center;
    int radius;
public:
    Circle(int x, int y, int radius)
        :center(new Point(x, y)), radius(radius) {};
    ~Circle() { delete center; }
};
```

Обобщение (наследование)

- Базовый принцип ООП
- Определяет отношение IS A - то есть "является"
- Позволяет дочернему получить функционал родительского

Реализация

- Наследование от интерфейса
- Интерфейс – класс, содержащий только чистые виртуальные методы и не обладающий состоянием

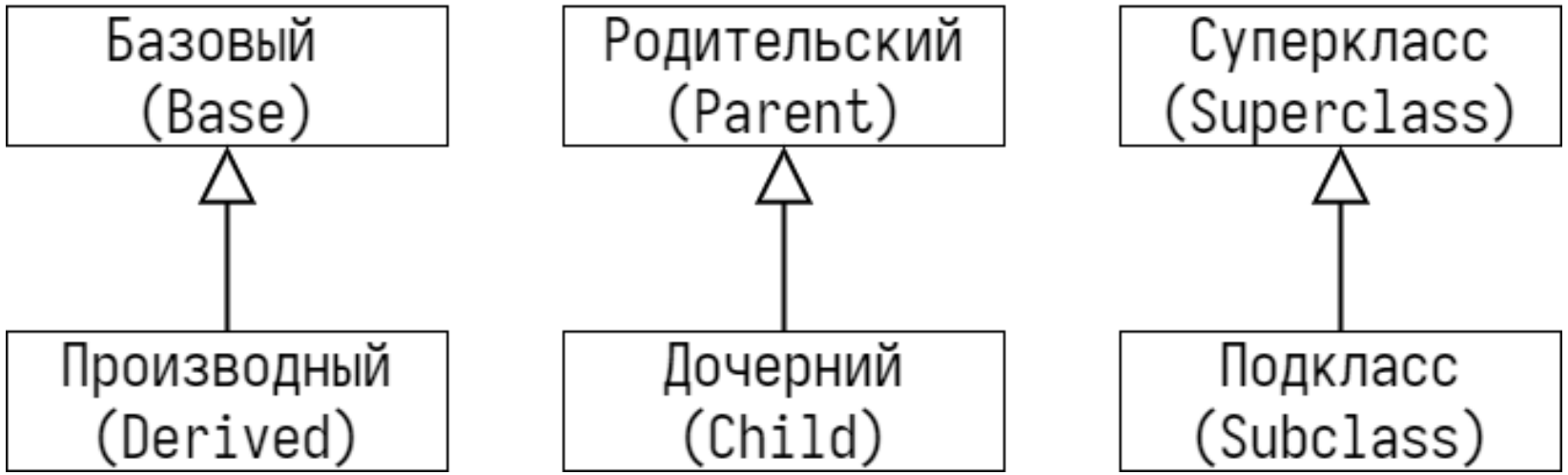


Лекция 3

Наследование

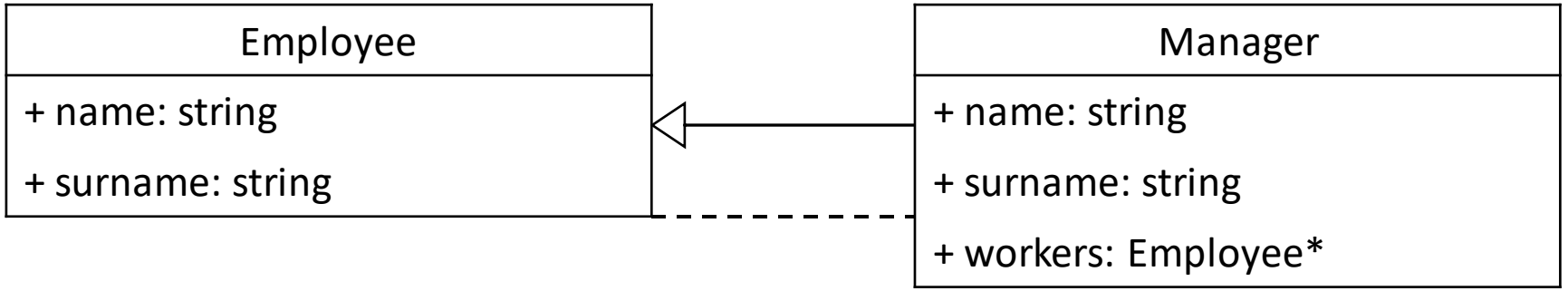
Наследование

- Один из основных механизмов ООП
- Позволяет создавать классы на основе существующих
- Изменяют или расширяют функционал тех классов, на основе которых создаются
- В C++ допускается множественное наследование



Расположение в памяти

- Дополнительная память только для новых полей
- Внутренний объект родительского класса располагается в начале дочернего объекта
- Ссылку или указатель на объект дочернего класса можно использовать везде, где допустимо использование ссылки или указателя на объект родительского класса



Порядок определения

- Использование класса в качестве базового эквивалентно созданию неименованного поля в дочернем объекте
- Для использования в качестве базового, класс должен быть объявлен перед этим
- Наследование может отличаться модификатором доступа

```
class Base; //только объявление без определения

class Deriv: public Base{
    //Определение
};
```

Приведение по ссылке или указателю

- Это возможно за счет одинакового расположения полей

```
class Base{

};

class Deriv: public Base{

};

int main(){
    Base* obj1 = new Deriv();
    Deriv obj2;
    Base& obj3 = obj2;
}
```

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

- Конструкторы и деструкторы не наследуются

Доступ к полям при наследовании

- Происходит понижение к самому ограничивающему модификатору доступа

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

Ограничение доступа при protected

```
class Base{
private:
    void f(){std::cout << "Private";}
protected:
    void g(){std::cout << "Protected";}
public:
    void h(){std::cout << "Public";}
};

class Deriv: protected Base{

};

int main(){
    Deriv obj;
    obj.f();
    obj.g();
    obj.h();
}
```

Ограничение доступа при public

```
class Base{
private:
    void f(){std::cout << "Private";}
protected:
    void g(){std::cout << "Protected";}
public:
    void h(){std::cout << "Public";}
};

class Deriv: public Base{

};

int main(){
    Deriv obj;
    obj.f();
    obj.g();
    obj.h();
}
```

Порядок конструирования объектов

- Объекты создаются «снизу-вверх» - от базовых к производным
- Порядок вызовов конструкторов:
  1. Конструкторы виртуальных базовых классов
  2. Конструкторы прямых базовых классов
  3. Конструкторы полей
  4. Конструктор класса

- Деструкторы вызываются в обратном порядке

Переопределение функций

- Переопределение функций – создание функции в дочернем классе с сигнатурой, совпадающей с функцией в родительском классе

- Для определения вызываемой функции компилятор сначала ищет ее 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 <<  "Base\n";}
};

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

int main(){
    Base b;
    b.print(); // "Base"
    Deriv d;
    d.print(); // "Deriv"
    Base& b_ref = d;
    b_ref.print(); // "Base"
}
```

Вызов конструкторов базового класса (1)

- В данном случае ошибка, так как конструктор 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 obj(4);
}
```

Вызов конструкторов базового класса (2)

- Необходимо получать аргументы и явно вызывать базовый конструктор

```
class Base{
public:
    Base(int a){}
};

class Deriv: public Base{
public:
    Deriv(int a):Base(a){}
};

int main(){
    Deriv obj(4);
}
```

Важность деструкторов при наследовании

- Так как деструкторы не наследуются, возможна утечка памяти

```
class Base{
    int* data;
public:
    Base():data(new int[10]){}
    ~Base(){delete [] data;}
};

class Deriv: public Base{
public:
    Deriv():Base(){}
};

int main(){
    Deriv obj();
}
```

Проблема переопределения функций

```
class Base{
public:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};

class Deriv: public Base{
public:
    void print(std::string a){std::cout << "Deriv - " << a << '\n';}
};

int main(){
    Deriv obj;
    obj.print("a");
    obj.print('a'); //Ошибка
    obj.print(47);  //Ошибка
}
```

Ключевое слово using

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

using с конструкторами

- Позволяет использовать все базовые конструкторы

```
class Base{
public:
    Base(int a){}
};
class Deriv: public Base{
public:
    using Base::Base;
};
int main(){
  Deriv obj(4);
}
```

using с методами

```
class Base{
public:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};
class Deriv: public Base{
public:
    using Base::print;
    void print(std::string a){std::cout << "Deriv - " << a << '\n';}
};
int main(){
    Deriv obj;
    obj.print("a");
    obj.print('a');
    obj.print(47);
}
```

Изменение видимости доступа (1)

- Запрещаем использовать public методы из базового класса

```
class Base{
public:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};
class Deriv: public Base{
private:
    using Base::print;
};
int main(){
    Deriv obj;
    obj.print('a'); //Ошибка
    obj.print(47);  //Ошибка
}
```

Изменение видимости доступа (2)

- Разрешаем использовать protected методы из базового класса

```
class Base{
protected:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};
class Deriv: public Base{
public:
    using Base::print;
};
int main(){
    Deriv obj;
    obj.print('a');
    obj.print(47);
}
```

Виртуальные функции

Виртуальные функции

- Функция-член класса, которую предполагается переопределить в производных классах
- Вызывает функцию производного класса даже через ссылку или указатель на базовый класс
- Модификатор **virtual** располагается перед типом возвращаемого значения
- Должна быть определена в месте первого объявления
- Может быть переопределена в дочерних классах

Использование виртуальных функций

- Определение конкретного типа объекта не требуется

- При вызове будет использован полиморфизм и реализация будет выбрана в зависимости от типа объекта

```
void printList(const vector<Employee*>& employees) {
    for (Employee* current : employees) {
        current->print();
    }
}
```

Полиморфный класс

- Любой класс, содержащий по крайней мере одну виртуальную функцию, является полиморфным
- Каждый объект такого класса содержит таблицу виртуальных функций **(vtable)**
- При использовании ссылки / указателя разрешение методов происходит динамически в момент вызова

Таблица виртуальных функций

- Координирующая таблица (vtable)
- Указатель на vtable хранится в каждом объекте
- Содержит адреса динамически связанных методов объекта
- Выбор реализации метода при вызове осуществляется определением адреса требуемого из таблицы виртуальных методов

Таблица виртуальных функций в памяти

```
struct Person {
    virtual ~Person() {}
    string name() const;
    virtual string position() const = 0;
};

struct Student : Person {
    string position() const;
    virtual int group();
};
```

Person		
0	~Person	0xAB22
1	position	0x0000

Student		
0	~Student	0xAB46
1	position	0xAB68
2	group	0xAB8A

Таблица виртуальных функций в памяти

```
struct Person {
    virtual ~Person() {}
    virtual string position() const = 0;
};

struct Teacher : Person {
    string position() const;
    virtual string course();
};
struct Professor : Teacher {
    string position() const;
    virtual string thesis();
};
```

Person		
0	~Person	0xAB20
1	position	0x0000

Teacher		
0	~Teacher	0xAB48
1	position	0xAB60
2	course	0xAB84

Professor		
0	~Professor	0xABA8
1	position	0xABB4
2	course	0xAB84
3	thesis	0xABC8

Виртуальные ф-ии и полиморфизм

- Виртуальные функции являются основным механизмом динамического полиморфизма

```
class Base{
public:
    virtual void  print() const{std::cout << "Base\n";}
};
class Deriv: public Base{
public:
    void print() const {std::cout << "Derived\n";}
};
void f(const Base& obj){obj.print();}
int main(){
    f(Base());
    f(Deriv());
}
```

Ключевое слово final

- Для виртуальных функций может обозначать, что это конечное переопределение
- Не может применяться к не виртуальным функциям
- Для класса определяет, что от него нельзя дальше наследоваться
- Для определения финального класса может использовать `std::is_final`

Ключевое слово final

```
struct Base{
    virtual void foo();
};

struct A : Base{
    void foo() final; // Base::foo переопределен и A::foo последнее переопределение
    void bar() final; // Ошибка: bar не может быть финальным
};

struct B final : A{ // struct B финальный класс
    void foo() override; // Ошибка: foo не может быть переопределена
};

struct C : B{ // Ошибка: B финальный класс
};
```

Определение типа объекта

Определение типа объекта

- Для определения типа объекта, располагающегося в указателе на базовый класс, можно:
  1. Убедиться, что указатель может ссылаться только на объект базового класса (спецификатор final)
  2. Использовать специальное поле для хранения информации о типе объекта
  3. Использовать механизм виртуальных функций
  4. Использовать dynamic_cast
  5. Использование оператора typeid из модуля typeinfo

- dynamic_cast и typeid реализуют механизм RTTI

Использование поля для хранения типа

```
class Employee {
public:
    enum EmployeeType {MANAGER, EMPLOYEE};
    Employee() : type(EMPLOYEE) {}
    EmployeeType getType() const {
        return type;
    }
protected:
    Employee(EmployeeType type) : type(type) {}
private:
    EmployeeType type;
};

class Manager : public Employee {
public:
    Manager() : Employee(MANAGER), level(0) {}
    int getLevel() const {
        return level;
    }
private:
    int level;
};
```

Определение типа через поле

```
void printEmployee (const Employee *employee) {
    switch (employee->getType()) {
        case Employee::MANAGER:
            const Manager *manager = (const Manager*)employee;
            cout << manager->getLevel() << endl;
        case Employee::EMPLOYEE:
            cout << employee->getName() << endl;
    }
}

void printList(const vector<Employee*>& employees) {
    for (Employee* current : employees) {
        printEmployee(current);
    }
}
```

Функции приведения типов

- static_cast – производится на этапе компиляции
  - TYPE static_cast<TYPE> (object);
- dynamic_cast – производится во время работы программы
  - TYPE& dynamic_cast<TYPE&> (object);
  - TYPE* dynamic_cast<TYPE*> (object);
- dynamic_cast – работает только с полиморфными классами

Приведение типов

- Использование ссылки производного класса допустимо везде, где предполагается ссылка базового класса

```
Manager manager(“Name”, “Surname”, “Sales”);
Employee &ref = manager; // Manager& -> Employee&
Employee *ptr = &manager; // Manager* -> Employee*
```

- Допустимо присвоение переменной базового класса объекта производного
- При этом используется конструктор копирования родительского класса

```
Manager manager(“Name”, “Surname”, “Sales”);
Employee employee = manager; // Employee(“Name”, “Surname”);
```

Приведение типов с модификаторами доступа

- При использовании **public** наследования: использование ссылки на базовый класс допустимо везде

```
class Class {};
class PublicChild : public Class{};
class ProtectedChild : protected Class{};
class PrivateChild : private Class{};
```

- При наследовании с модификатором **protected**: о том что *Class* является базовым для *ProtectedChild* знают сам класс и его наследники
- При использовании модификатора **private**: приведение ссылки к базовому классу допустимо только внутри *PrivateChild*

Интерфейсы

Чистая виртуальная функция

- Функция, которая объявляется в базовом классе, но не имеет в нем определения

- Всякий производный класс обязан иметь свою собственную версию

- Для объявления чистой виртуальной функции следует:
  1. Использовать ключевое слово `virtual`, расположив его перед типом возвращаемого значения
  2. Указать `= 0;` после списка аргументов
  3. Исключить тело функции - оставить её без реализации

Абстрактный класс

- Любой класс, содержащий по крайней мере одну чистую виртуальную функцию, 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, содержащий только **public** секцию и только чистые виртуальные методы
- Интерфейс не должен содержать поля
- Каждый интерфейс является абстрактным классом, но не каждый абстрактный класс интерфейс
- При использовании интерфейс реализуют, абстрактный класс – наследуют

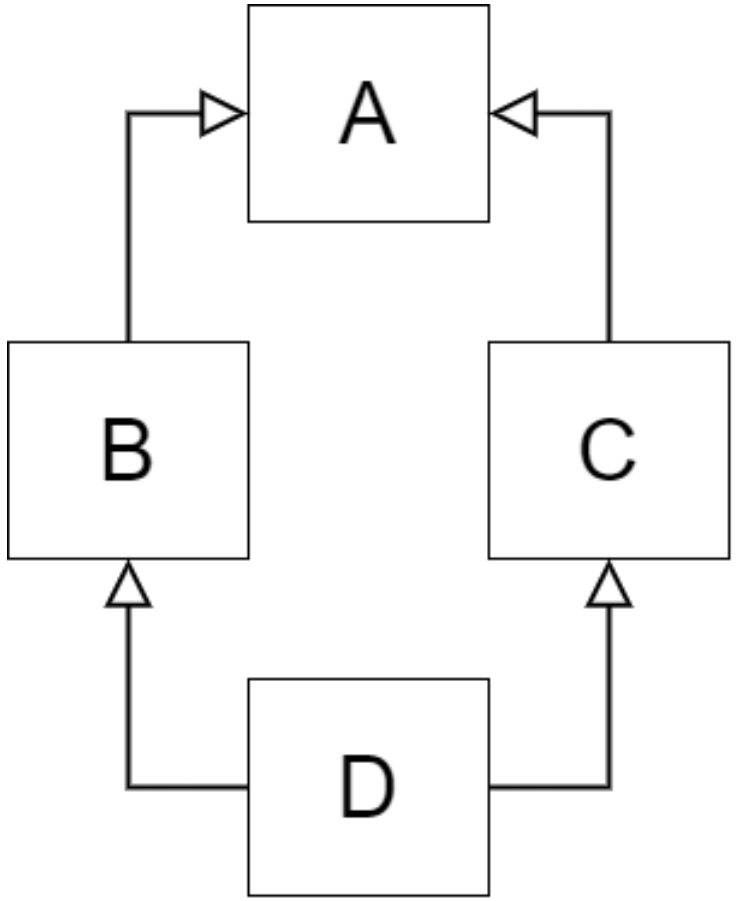
Использование чистых виртуальных функций

```
class Base{
public:
    virtual void  print() = 0;
};
class Deriv: public Base{
public:
    void print() const {std::cout << "Derived\n";}
};
void f(const Base* obj){obj->print();}
int main(){
    f(new Base()); //Ошибка
    f(new Deriv());
}
```

Виртуальное наследование

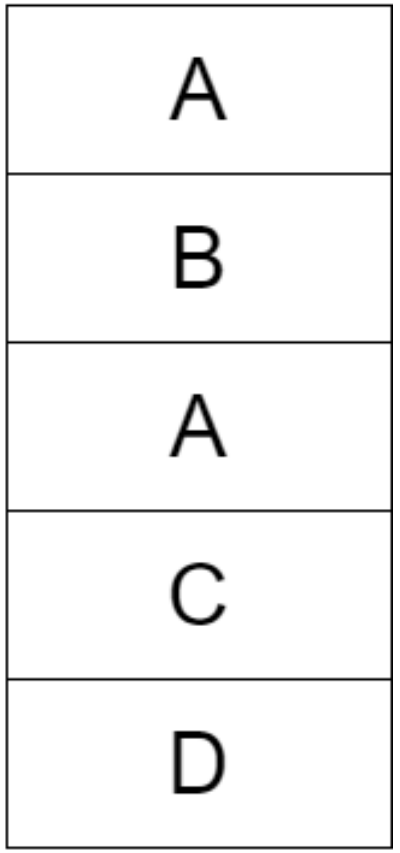
Проблема «ромбовидного» наследования

```
class A{
public:
    void foo();
};

class B: public A{
};

class C: public A{
};

class D: public B, public C{
};
```



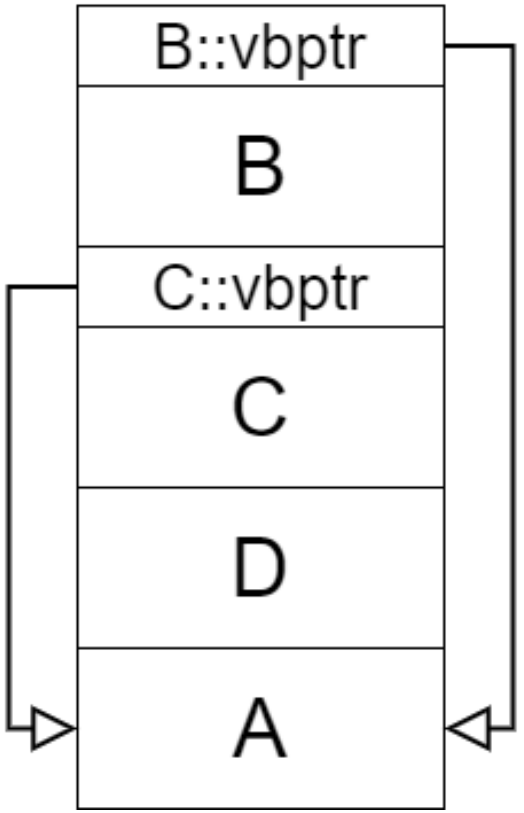
Проблема «ромбовидного» наследования

```
class A{
public:
    void foo();
};

class B: public A{
};

class C: public A{
};

class D: public B, public C{
};
```



Проблема «ромбовидного» наследования

```
class A{
public:
    void foo();
};

class B: virtual public A{
};

class C: virtual public A{
};

class D: public B, public C{
};
```



Виртуальное наследование

- Необходимо явно вызывать конструкторы всех виртуально унаследованных классов
- Необходимо стараться избегать множественного наследования не от интерфейсов
- Опасность вызова функции из класса «брата»

Дружественность

Дружественные функции

- Дружественные функции:
  - Имеют доступ к **private** и **protected** полям класса
  - Не являются методом класса
  - Дружественность объявляется внутри класса спецификатором **friend**

- Дружественность не наследуется

Дружественные классы

- Дружественный класс имеет доступ к private и protected полям класса

- Все методы дружественного класса становятся дружественными

Лекция 4

Константные выражения

constexpr

- constexpr сообщает, что выражение должно быть рассчитано на этапе компиляции
- Может быть применено к переменным и функциям

```
constexpr int sum(int a, int b){
    return a + b;
}

int main(){
    constexpr int val1 = 14;
    constexpr int val2 = sum(3,4);
    int n = 10;
    constexpr int val3 = sum(n, 4); //Ошибка
    constexpr int val4 = sum(val1, val2);
    int val5 = sum(n, val1);
}
```

constexpr класс

```
class ConstExprClass{
    int value;
public:
    constexpr ConstExprClass(int value):value(value){}
    constexpr int getValue() const{
        return value > 0 ? value : 0;}
    void increase(){
        value++;}
};

int main(){
    constexpr ConstExprClass obj{6};
    std::cout << obj.getValue() << '\n';
    ConstExprClass obj2{7};
    obj2.increase();
    std::cout << obj2.getValue() << '\n';
    return 0;
}
```

constexpr ограничения переменных

- Могут быть переменные, для которых соблюдаются условия :
  1. Скалярные типы
  2. Указатели
  3. Массив скалярных типов
  4. Класс, в котором:
     - Деструктор по умолчанию
     - Все нестатические поля – литеральные типы данных
     - Хотя бы один constexpr конструктор или их отсутствие

constexpr ограничения функций

- Могут быть функции, для которых соблюдаются условия:
  1. Должны быть не virtual
  2. Должны возвращать литеральный тип
  3. Все аргументы должны иметь литеральный тип
  4. Тело функции может содержать только:
     - static_assert
     - typedef и using
     - Ровно один return, который содержит constexpr выражение

Перегрузка операторов

Оператор

- Оператор – функция, обозначенная специальным символом
- Сигнатура оператора такая же, как у функций, но с ключевым словом **operator #**, где # знак оператора
- Существуют унарные и бинарные операторы

Таблица операторов

Оператор	Тип по смыслу	По кол-ву аргументов
+ - * / %	Арифметические	Бинарные
+= -= *= /= %=		Унарные
+a -a	Смена знака	Унарные
++a --a	Префиксный инкремент	Унарные
a++ a--	Постфиксный инкремент	Унарные
&& || !	**Логические**	**Бинарные**
& | ^ << >>	Битовые	Бинарные
~		Унарные
=	Присваивание	Бинарный
== != < > <= >=	Сравнение	Бинарные
<< >>	Вывод в поток	Бинарные

Специальные операторы

- ***a*->** – доступ к полям по указателям – перегружать не рекомендуется
- ***a*.** – доступ к полям – перегружать нельзя
- ? : – тернарный оператор – перегружать нельзя
- :: – доступ к полю – перегружать нельзя
- () – вызов функции
- (type) – приведение к типу
- new – выделение памяти – перегружать не рекомендуется
- delete – освобождение памяти – перегружать не рекомендуется

Оператор присваивания

- Возвращает ссылку на объект класса
- Внутри оператора возвращается *this

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    MyInt& operator = (const MyInt& val){
        if(this != &val)
            this->i = val.i;
        return *this;
    }
};
```

Унарные арифметические операторы

- Принимают один аргумент
- Видоизменяют существующий объект
- Принято возвращать ссылку на текущий объект

```
MyInt& operator +=(const MyInt& val){
        this->i += val.i;
        return *this;
    }
```

Бинарные арифметические операторы

- Объявляются с модификатором friend
- Принимают 2 аргумента – левый и первый операнд
- Создают новый объект
- Возвращают результат по значению

```
friend MyInt operator + (const MyInt& val1, const MyInt& val2){
    return MyInt(val1.i + val2.i);
}
friend MyInt operator + (const MyInt& val1, const int& val2){
    return MyInt(val1.i + val2);
}
```

Унарные операторы

- Не принимают аргумента
- Возвращают ссылку на текущий объект

```
MyInt& operator -(){
    this->i = - this->i;
    return *this;
}
MyInt& operator +(){
    return *this;
}
```

Операторы инкремента и декремента

- Префиксный оператор не принимает аргументов и сначала меняет объект, потом возвращает ссылку на него
- Постфиксный инкремент принимает фиктивный аргумент, создает копию объекта, меняет текущий объект и возвращает копию по значению

```
MyInt& operator ++(){
        this->i += 1;
        return *this;
    }
MyInt operator ++(int){
    MyInt temp(this->i);
    this->i += 1;
    return temp;
}
```

Логические операторы

```
friend bool operator == (const MyInt& val1, const MyInt& val2){
    return (val1.i == val2.i); //Необходимо реализовать
}
friend bool operator != (const MyInt& val1, const MyInt& val2){
    return !(val1 == val2);
}
friend bool operator < (const MyInt& val1, const MyInt& val2){
    return (val1.i < val2.i); //Необходимо реализовать
}
friend bool operator > (const MyInt& val1, const MyInt& val2){
    return (val2.i < val1.i);
}
friend bool operator <= (const MyInt& val1, const MyInt& val2){
    return !(val1.i > val2.i);
}
friend bool operator >= (const MyInt& val1, const MyInt& val2){
    return !(val1.i < val2.i);
}
```

Оператор приведения к типу

- Особенность – не имеет возвращаемого типа
- Можно приводить к любому типу, известному в месте объявления оператора

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    operator int() const{
        return this->i;
    }
};
```

Перегрузка оператора ввода

- Левым операндом является ссылка на istream
- Возвращает по ссылке поток, из которого происходило чтение
- Правый операнд должен быть не константным

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    friend std::istream& operator>>(std::istream& in, MyInt& obj){
        in >> obj.i;
        return in;
    }
};
```

Перегрузка вывода в поток

- Левым операндом является ссылка на ostream
- Возвращает по ссылке поток, в который происходила запись

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    friend std::ostream& operator<<(std::ostream& out, const MyInt& obj){
        out << obj.i;
        return out;
    }

};
```

Операторы new и delete

- Увеличение производительности за счёт кеширования
- Выделение памяти сразу под несколько объектов
- Реализация собственного сборщика мусора
- Вывод логов выделения и освобождения памяти
- Реализация своего placement new
- Выделение памяти без исключений
- Собственные формы new

Рекомендации по перегрузке

Оператор	**Рекомендуемая форма**
Все унарные операторы	Член класса
+= -= /= *= ^= &= и т.д	Член класса
= () [] -> ->*	Обязательно член класса
Остальные бинарные операторы	Не член класса

Пользовательские литералы

Пользовательские литералы

- Пользовательские литералы определяются как
  operator "" suffix_identifier
- Позволяют переводить какой-либо литерал к пользовательскому типу

```
constexpr long double operator "" _deg (long double deg){
    return deg * 3.14159265358979323846264L / 180;
}
```

Ограничение на пользовательские литералы

- Оператор можно применять к типам:
  - unsigned long long int
  - long double
  - char, wchar_t, char8_t (c++20), char16_t, char32_t
  - const char *
  - (const char * , std::size_t)

Функторы

Оператор вызова функции

- Класс, для которого определен оператор вызова функции, называется функтором

```
class MyInt{
    int i;
public:
    MyInt(int i = 0):i(i){}
    MyInt& operator ()(int val){
        this->i *= val;
        return *this;
    }
    void print(){
        std::cout << i << '\n';
    }
};
```

Пример функтора 1

```
class SQR{
    double a;
    double b;
    double c;
public:
    SQR(double a = 1, double b = 0, double c = 0):a(a),b(b),c(c){}
    double operator()(double x = 0){
        return a*x*x + b*x + c;
    }
};

int main(){
    SQR f(1,-2,1);
    std::cout << f(5) << " " << f(1) << '\n';
    return 0;
}
```

Пример функтора 2

```
class F{
    int val;
public:
    F(int val):val(val){}
    F& operator()(int x){
        this->val += x;
        return *this;
    }
    operator int(){
        return this->val;
    }
};

int main(){
    std::cout << F(6)(4)(3)(-1);
}
```

Пример функтора 3

```
class Yield{
    std::vector<double> m_result;
public:
    double operator()(double val){
        double e_val = exp(val);
        m_result.push_back(e_val);
        return e_val;
    }
    std::vector<double> result() const{
        return m_result;
    }
    double operator[](size_t i){
        return m_result.at(i);
    }
};
```

Функторы вместо lambda

- Функторы также могут использоваться вместо lambda-выражений

```
Functor func_obj;
std::vector<int> v = {-1,1,2};
std::for_each(v.begin(), v.end(), [](const int& val){std::cout << val << ' ';});
std::for_each(v.begin(), v.end(), func_obj);
```

std::mem_fn

- mem_fn позволяет преобразовать метод класс в функцию вне класса

```
struct Foo {
    void display_greeting() {
        std::cout << "Hello, world.\n";
    }
    void display_number(int i) {
        std::cout << "number: " << i << '\n';
    }
    int data = 7;
};
int main() {
    Foo f;
    auto greet = std::mem_fn(&Foo::display_greeting);
    greet(f);
    auto print_num = std::mem_fn(&Foo::display_number);
    print_num(f, 42);
    auto access_data = std::mem_fn(&Foo::data);
    std::cout << "data: " << access_data(f) << '\n';
}
```

std::function

- Полиморфная оболочка для функций

```
void proceedOperation(double val1, double val2,
       std::function<double(double,double)> f){
    std::cout << "Result is " << f(val1, val2) << '\n';
}
```

std::function

```
class Functor{
    double value;
public:
    Functor(double val):value(val){}
    double operator()(double val1, double val2){
        return val1 + val2 + value;
    }};

double sum(double val1, double val2){
    return val1 + val2;}

void proceedOperation(double val1, double val2, std::function<double(double,double)> f){
    std::cout << "Result is " << f(val1, val2) << '\n';}

int main(){
    double a = 0.4, b = -1.6;
    Functor f{3.4};
    proceedOperation(a, b, f);
    proceedOperation(a, b, sum);
    proceedOperation(a, b, [](double a, double b){return a - b;});
    std::function<double(double,double)> saved_func = sum;
}
```

Лямбда-функции

Лямбда-функции

- Является анонимной функцией
- Можно хранить в переменных
- Обычно используются для однострочных и редко используемых функций

```
int main(){
    [](){}; //Пустая лямбда
    [](){std::cout << "Hello, World!\n";}(); //Объявляем и сразу вызываем
    //Соханяем в переменную
    auto f = [](){std::cout << "Goodbye, World!\n";};
    f(); //Вызываем из переменной
    return 0;
}
```

Передача аргументов

- Аргументы можно передавать как и в обычной функции

```
int main(){
    int a = 10, b = 7;
    auto f = [](int a, int b){return a + b;};
    std::cout << f(a,b) << '\n';
    std::cout << ([](int a, int b){return a - b;}(a,b)) << '\n';
    return 0;
}
```

Захват переменных

- Можно производить захват переменных из области видимости, где объявляется лямбда

```
int main(){
    int a = 10, b = 7;
    auto f = [=]{return a + b;}; //Захват по значению (read-only)
    std::cout << f() << '\n';
    std::cout << ([&]{return a - ++b;}()) << '\n'; //Захват по ссылке
    std::cout << a << ' ' << b << '\n';
    return 0;
}
```

Захват переменных

- Можно указывать, какие переменные и как захватывать

```
int main(){
    int a = 10, b = 7;
    auto f = [a](int b){return a + b;}; //Захват по значению (read-only)
    std::cout << f(3) << '\n';
    //Захват по ссылке
    std::cout << ([&b](int a){return a - ++b;}(22)) << '\n';
    std::cout << a << ' ' << b << '\n';
    return 0;
}
```

Захват переменных в классе

- Лямбды не знают про поля класса
- Можно захватить this и дать доступ к полям класса

```
class A{
    int x;
public:
    void func(){
        double z;
        [this]{x *= 2; /*z *= 2;*/}();
    }
};
```

Возвращаемый тип

- По умолчанию тип выводится из результата операции
- Можно явно задать тип после списка аргументов

```
int main(){
    double a = 10.5, b = 7.2;
    auto f = [=]()->int {return a + b;};
    std::cout << f() << '\n';
    std::cout << ([=]()->int {return a - b;}()) << '\n';
    std::cout << a << ' ' << b << '\n';
    return 0;
}
```

Пример использования (1)

```
class SumResult{
    int value;
public:
    SumResult():value(0){}
    auto getFunc(){
        return [this](int add_value){value += add_value;};
    }
    int getResult(){
        return value;
    }
};

int main(){
    SumResult res;
    std::vector<int> vec = {1,2,-3,5,6};
    std::for_each(vec.begin(), vec.end(),res.getFunc());
    std::cout << res.getResult();
    return 0;
}
```

Пример использования (2)

```
class SumResult{
    int value;
public:
    SumResult():value(0){}
    void apply(std::function<void(int&)> f){
        f(value);}
    int getResult(){
        return value;
    }};

int main(){
    SumResult res;
    int a = 10;
    int b = 4;
    res.apply([](int& value){value += 2;});
    res.apply([=](int& value){value *= a;});
    res.apply([&](int& value){value -= b++;});
    std::cout << b << ' ' << res.getResult();
}
```

Сложный пример (1)

```
#include<iostream>
#include<functional>

typedef std::function<void(int& )> log_func;

class Logger{
public:
    virtual log_func getLogFunc() = 0;
};

class ConsoleLogger: public Logger{
public:
    log_func getLogFunc(){
        return [](int& value){std::cout << value << '\n';};
    }
};
```

Сложный пример (2)

```
class Operator{
    int value;
    std::vector<log_func> funcs_to_do;
    void proceedFuncs(){
        for(auto f: funcs_to_do){
            f(value);
        }
    }
public:
    Operator(int value = 0):value(value){}
    void addFunc(log_func f){
        funcs_to_do.push_back(f);
    }
    void add(int value){
        this->value += value;
        this->proceedFuncs();
    }
    void mul(int value){
        this->value *= value;
        this->proceedFuncs();
    }
};
```

Сложный пример (3)

```
int main(){
    Operator op(1);
    Logger* logger = new ConsoleLogger();
    op.addFunc(logger->getLogFunc());
    op.addFunc([](int& value){value /= 2;});
    op.add(3);
    op.add(5);
    op.mul(4);
    delete logger;
    return 0;
}
```