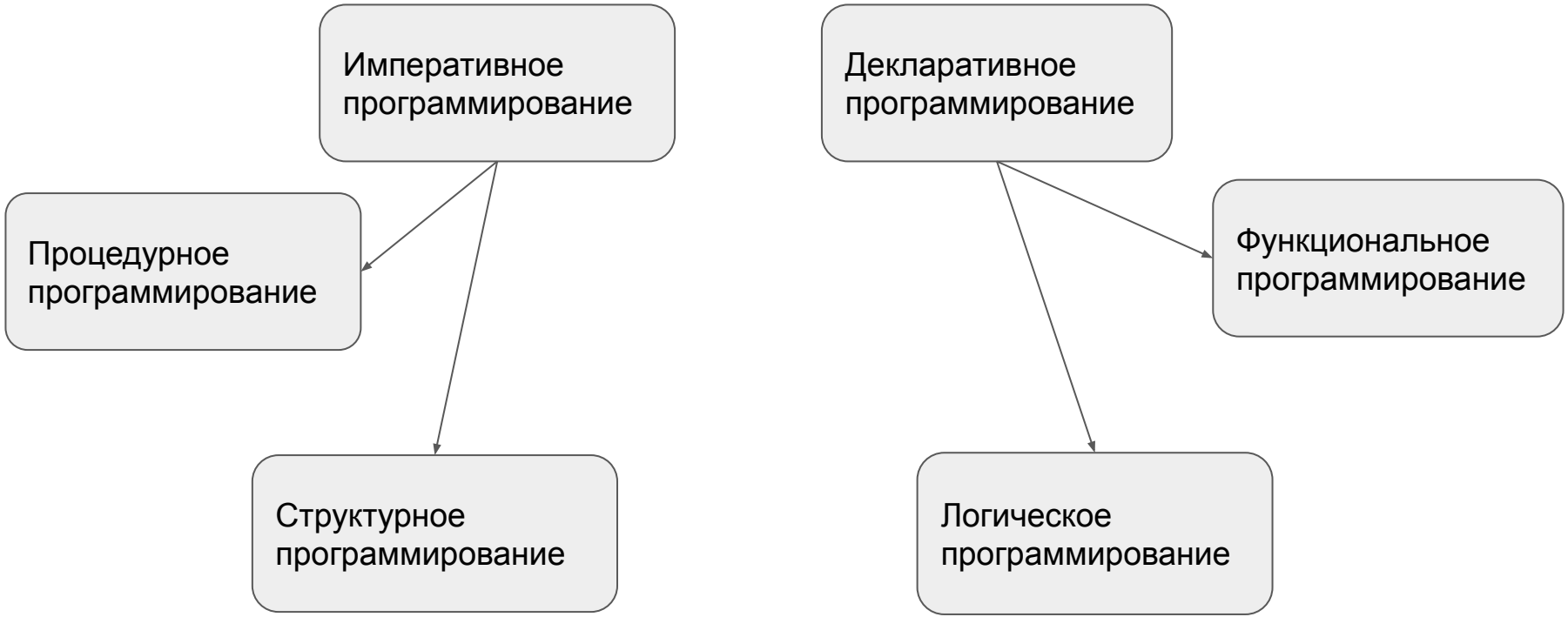
ООП

Парадигмы программирования



Императивное программирование

- Состояние программы и инструкции (команды)
- Исполнение программы - последовательные переходы между состояниями с помощью инструкций
- Структурное программирование - код структурирован блоками с простыми переходами
- Процедурное программирование - объединение последовательностей инструкций в одну, с целью её многократного использование

Декларативное программирование

- Программа - описание задачи
- Исполнение программы - автоматический процесс определенный языком программирования
- Функциональное программирование - описание задачи в терминах системы уравнений. Исполнение программы - решение системы
- Логическое программирование - описание задачи в терминах логики и правил вывода. Исполнение программы - автоматическое построение вывода из аксиом с помощью правил вывода.

ООП

- Декларативные идеи в императивных языках
- Программа - описание набора сущностей(объектов)
- Объекты - объединение данных и способов работы с ними
- Взаимодействие между собой объекты определяют сами
- Безопасность данных
- Универсальность кода - применение одного кода для разных объектов

Основные идеи ООП

- Инкапсуляция - объединение данных и способов работы с ними в неделимое целое(черный ящик). Зачастую вместе с сокрытием самих данных.
- Полиморфизм - использование объектов с одинаковым интерфейсом, без информации о внутреннем устройстве объектов. Обобщенное программирование
- Наследование - описание новых классов на основе существующих с использованием и/или доопределением функциональности старых

Инкапсуляция

Инкапсуляция в C++

- Объединение данных(поля) и функционала(методы) в целое(объект) с помощью описания общей структуры объекта(класс)
- Настройка доступа к полям и методам с помощью модификаторов доступа:
  - public - доступ открыт всем, кто видит описание данного класса
  - private - доступ открыт методам этого класса, а также дружественным(friend) функциям и классам. По-умолчанию поля и методы класса - private
  - protected - доступ открыт производным от данного классам
- Интерфейс класса - поля и методы доступные вне объектов класса

Пример

```
class Rational{
        int numerator;                  //Числитель
        unsigned int denominator;       //Знаменатель
public:
        int Numerator() {               // метод объявленный и
                return numerator;// реализованный в классе
        }           // возвращает значение поля

        unsigned int Denominator();     /* открытый метод
        объявленный в классе с внешней реализацией */
private:
        void Reduce();                  /* закрытый метод
        объявленный в классе с внешней реализацией */
};
```

```
unsigned int Rational::Denominator(){
        return Rational::denominator;
}

/* реализация public-метода снаружи класса для обращения к
методам и полям требуется указание класса, к которому эти
методы принадлежат */

void Rational::Reduce(){
        int tmp=GCD(Rational::numerator,Rational::denominator);
        Rational::numerator /= tmp;
        Rational::denominator /= tmp;
}

/* реализация private-метода снаружи класса для обращения к
методам и полям требуется указание класса, к которому эти
методы принадлежат. При этом поля остаются доступные,
поскольку это реализация метода - члена класса*/
```

Стандартные методы классов. Конструктор

- Конструктор - метод инициализирующий объект класса. Выделяет память под поля объекта и выполняет действия, которые мы укажем необходимые для инициализации. Имеет название совпадающее с названием класса
- Конструктор не имеет типа
- Конструктор может иметь неограниченное количество аргументов и многократно определяться (см. перегрузка в разделе Полиморфизм)
- Если конструктор не задан - создается конструктор по-умолчанию (только выделяет память под поля)
- Возможно сокращение кода, с использованием списков инициализации
- Конструктор вызывается автоматически при создании нового объекта:
    - MyClass A(arg1,arg2) - на стеке
    - MyClass* =new MyClass(arg1,arg2) - в куче

Пример. Тривиальный конструктор

```
class Rational{
        int numerator;                  //Числитель
        unsigned int denominator;       //Знаменатель
public:
        Rational(const int&  a, const unsigned int&  b){
                numerator = a;
                denominator = b;
        }
};
// пример использования использования:
Rational R(12,9);           // (12,9)
```

```
// более короткая версия через список инициализации:
class Rational{
        int numerator;                  //Числитель
        unsigned int denominator;       //Знаменатель
public:
        Rational(const int&  a, const unsigned int&  b):
                        numerator(a), denominator(b) {}
};
// пример использования использования:
Rational R(12,9);           // (12,9)
```

```
/* можно задавать параметры по умолчанию*/
class Rational{
        int numerator;                  //Числитель
        unsigned int denominator;       //Знаменатель
public:
        Rational(const int&  a = 1, const unsigned int&  b = 1):
                        numerator(a), denominator(b) {}
};
// пример использования использования:
Rational Q(12,9);           // (12,9)
Rational R(5);                      // (5,1)
Rational T;                         // (1,1)

//при этом в методах отрабатывает приведение типов:
Rational S(1,-1);                   // (1,4294967295)
```

Пример. Конструктор с доп.действиями

```
class Rational{
        int numerator;                  //Числитель
        unsigned int denominator;       //Знаменатель
public:
        Rational(const int&  a = 1, const unsigned int&  b = 1):
                        numerator(a), denominator(b) {
                Reduce();
        }
        void Reduce(){
                int  tmp = GCD(numerator, denominator);
                numerator /= tmp;
                denominator /=tmp;
        }
};
// пример использования использования:
Rational Q(12,9);           // (4,3)
Rational R(5);                      // (5,1)
Rational T;                         // (1,1)
```

```
class int_vector{
        std::size_t size;           //Знаменатель
        int* array;                 //Знаменатель
public:
        int_vector(std::size_t  s = 1) : size(s), array(new int[s]) {
        for(int i=0; i<s;i++)
                array [ i ] = 0;
        }
        // не забываем потом освобождать
};

// пример использования использования:
int_vector A(9);            // массив длины 9, заполненный нулями
int_vector A;               // массив длины 1, заполненный нулями
```

Стандартные методы классов. Деструктор

- Деструктор - метод удаляющий объекты класса, освобождающий занятую память
- Деструктор не имеет типа
- Деструктор не имеет аргументов
- Если деструктор не задан - создается деструктор по-умолчанию (только освобождает память выделенную под поля)
- Деструктор вызывается автоматически при создании нового объекта:
  - при выходе из области видимости, в которой был создан объект

```
{
MyClass A(arg1,arg2);
…
}                    // при выходе вызовется деструктор
```

  - при освобождении выделенной на объект памяти на куче

```
MyClass* A = new MyClass(arg1,arg2);
…
delete A;//при освобождении памяти под A вызовется деструктор
```

Пример. Нетривиальный деструктор

```
class int_vector{
        std::size_ size;
        int* array;
public:
        int_vector(std::size_t  s = 1) : size(s), array(new int[s]) {
        for(int i=0; i<s;i++)
                array [ i ] = 0;
        }
        ~int_vector(){
                delete[] array;
        }
};

/*В деструкторе освобождаем выделенную в конструкторе
память*/
```

```
struct Node{
        Node* left    =  NULL;
        Node* right =  NULL;
        int data;
};
void DelTree(Node* a){
        if( a == NULL) return;
        DelTree(a->left);
        DelTree(a->left);
        delete a;
}
class BinaryTree{
        Node* root=NULL;
public:
        ~BinaryTree(){
                DelTree(root);
        }
};
/*В деструкторе освобождаем выделенную в конструкторе и
методах память, причем сделать деструктор рекурсивным -
нельзя в связи с отсутствием у него аргументов*/
```

Стандартные методы. Конструктор копирования

- Конструктор Копирования - метод создающий объекты-копии класса по ссылке на другой объект
- Конструктор копирования - конструктор(со всеми вытекающими свойствами)
- Аргумент конструктора копирования - ссылка на объект этого же класса
- Если конструктора копирования не задан - создается конструктора копирования по-умолчанию (создает объект с скопированными значениями полей)
- Конструктор копирования вызывается автоматически
    - `MyClass k2(k1);` // для явной создании копии
    - `MyClass k3 = k2;` // при вызове оператора присвоения
    - `MyClass k4 = MyClass(k3);` // при подобной инициализации объекта
    - `MyClass * ptr = new MyClass(k4);` // при выделении памяти с созданием копии
    - `void function(MyClass A);` // при прямой(не по ссылке) передаче объекта в функцию/метод

Пример. Нетривиальный Конструктор Копий

```
class int_vector{
        std::size_t size;
        int* array;
public:
        int_vector(std::size_t  s = 1) : size(s), array(new int[s]) {
                for(int i=0; i<s;i++)
                        array [ i ] = 0;
        }
        ~int_vector(){
                delete[] array;
        }
        size_t  get_size(){ return size; }
        ...operator[]...        // пока не говорим, как его перегрузить
        int_vector(const int_vector & v): size(v.get_size()),
                                        array(new int[ v.get_size()]){
                for(int i=0; i<s;i++)
                        array [i] = v[i];
        }

};
/*В конструкторе копирования поэлементно копируем данные из
оригинала в копию. Если не определить конструктор копий - в
копии будет лежать указатель на те же данные*/
```

```
struct Node{
        Node* next =  NULL;
        int data;
};
class List{
public:
        Node* head=NULL;
        List(const List& T){
            head=new Node;
            head->data=T.head->data;
            Node* ptr=head;
            for(Node* p=T.head->next; tmp!=NULL; tmp=tmp->next){
                Node* tmp=new Node;
                tmp->data=p->data;
                ptr->next=tmp;
                ptr=ptr->next;
        }
};
/*Поэлементным проходом по оригиналу воссоздаем
необходимую структуру*/
```

Стандартные методы. Конструктор Перемещений

- Конструктор Перемещений - метод заменяющий объект копией по ссылке на другой объект этого класса
- Конструктор Перемещений - извлекает данные из аргумента и передает в объект
- Аргумент конструктора перемещений - ссылка на объект этого же класса
- Погуглите, что это такое.

Пример. Нетривиальный Конструктор Копий

```
class int_vector{
        std::size_t size;
        int* array;
public:
        int_vector(std::size_t  s = 1) : size(s), array(new int[s]) {
                for(int i=0; i<s;i++)
                        array [ i ] = 0;
        }
        ~int_vector(){
                delete[] array;
        }
        size_t  get_size(){ return size; }
        ...operator[]...        // пока не говорим, как его перегрузить
        int_vector(const int_vector & v): size(v.get_size()),
                                        array(new int[ v.get_size()]){
                for(int i=0; i<s;i++)
                        array [i] = v[i];
        }

};
/*В конструкторе копирования поэлементно копируем данные из
оригинала в копию. Если не определить конструктор копий - в
копии будет лежать указатель на те же данные*/
```

```
struct Node{
        Node* next =  NULL;
        int data;
};
class List{
public:
        Node* head=NULL;
        List(const List& T){
            head=new Node;
            head->data=T.head->data;
            Node* ptr=head;
            for(Node* p=T.head->next; tmp!=NULL; tmp=tmp->next){
                Node* tmp=new Node;
                tmp->data=p->data;
                ptr->next=tmp;
                ptr=ptr->next;
        }
};
/*Поэлементным проходом по оригиналу воссоздаем
необходимую структуру*/
```

Стандартные методы. Оператор присваивания

- Оператор присваивания - оператор “=”
- Служит для инициализации, копирования и присваивания(как для стандартных типов)
- Примеры использования
  - `MyClass k2 = k1;` // для инициализации копией - copy assignment
  - `k3 = k2;` // для изменения значения - move assignment
- Вызывает соответствующие конструкторы
- Пример… см. “Финальный пример”

Правило трёх

- При написании кода принято, в случае определения одного из трех методов - явным(не “по умолчанию”), остальные - также делать явными. Эти три метода:
  - Деструктор
  - Конструктор копий
  - Оператор присваивания копированием(copy assignment)
- Стандарты языка до C++11

Правило пяти

- При написании кода принято, в случае определения одного из пяти методов - явным(не “по умолчанию”), остальные - также делать явными. Эти пять методов:
  - Деструктор
  - Конструктор копий
  - Оператор присваивания копированием(copy assignment)
  - Конструктор перемещений
  - Оператор присваивания перемещением(move assignment)
- Стандарты языка от C++11
- Для улучшения качества кода также используется идеология copy-and-swap.

Финальный пример

```
class int_vector{
protected:          // открываем поля другим объектам класса
        std::size_t size;
        int* array;
public:
        int_vector(std::size_t  s = 1) : size(s), array(new int[s]) {
                for(int i=0; i<s;i++)
                        array [ i ] = 0;
        }               // конструктор
        ~int_vector(){
                delete[] array;
        }               // деструктор
        int_vector(const int_vector& v): size(v.size()),
                                array(new int[ v.size()]){
                for(int i=0; i<s;i++)
                        array [i] = v[i];
        }               // конструктор копирования
        int_vector(const int_vector&& v):size(v.size()),array(v.array){
                v.array=nullptr;
        }               //конструктор перемещения
        friend void swap(int_vector& a, int_vector& b){
                using std::swap;
                swap(a.size,b.size);
                swap(a.array,b.array);
        }           // swap - меняем местами объекты
        int_vector& operator=(const int_vector& v){
                int_vector tmp(v);
                swap(*this, tmp);
                return *this;
        }           // оператор присваивания копированием
        int_vector& operator=(const int_vector&& v){
                swap(*this, tmp);
                return *this;
        }           // оператор присваивания копированием
};
```

/*Пример сильно затрагивает следующую тему, вернитесь к нему позднее*/