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

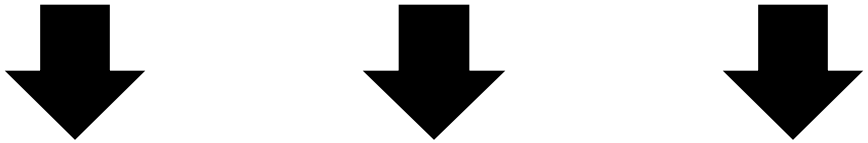
- en.cppreference.com – Документация языка C++

Парадигмы программирования

Парадигма программирования

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

Три основных парадигмы



Функциональная

Структурная

Объектно-ориентированная

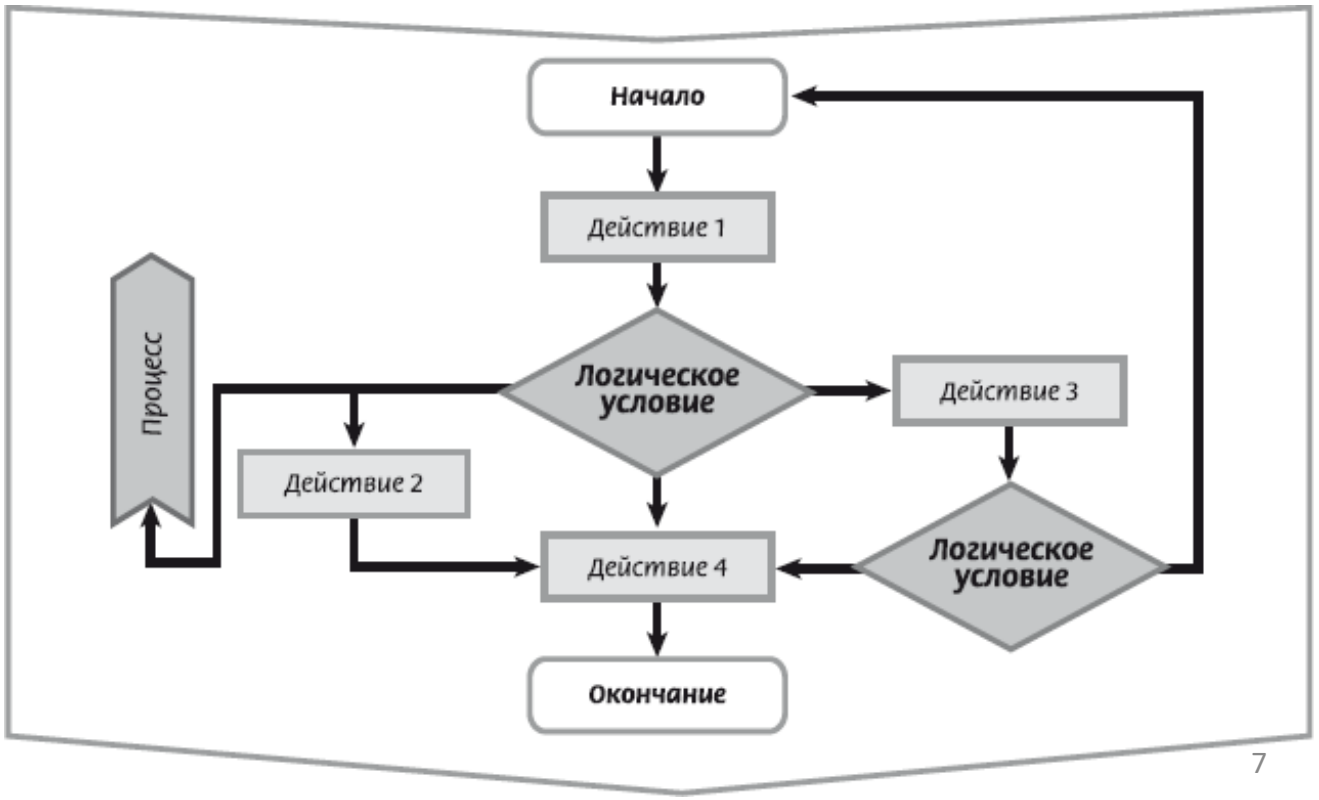
Функциональное программирование 1936г.

- 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