Семинар #3: Динамический полиморфизм

Часть 1: Полиморфизм

Полиморфизм – это способность функций обрабатывать данные разных типов.

Полиморфизм существует в большинстве языков программирования в том или ином виде. Например, рассмотрим следующую программу на языке Python:

```
def plus(a, b):
    return (a + b)

print(plus(10, 20))        # Напечатает 30
print(plus("Cat", "Dog"))  # Напечатает CatDog
```

Здесь функция `plus` является полиморфной, так как может работать с данными разных типов.

Статический полиморфизм

Полиморфизм в языке C++ мы уже частично прошли. Ведь такие возможности языка как перегрузка функций и шаблоны позволяют писать функции, которые работают с разными типами:

```
#include <iostream>
#include <string>
using namespace std::string_literals;

template<typename T>
T plus(T x, T y)
{
    return x + y;
}

int main()
{
    std::cout << plus(10, 20) << std::endl;          // Напечатает 30
    std::cout << plus("Cat"s, "Dog"s) << std::endl;  // Напечатает CatDog
}
```

Однако, в этом случае, то, какая функция будет вызвана, определяется на этапе компиляции: при раскрытии шаблона или при выборе перегрузки. Вид полиморфизма, при котором вызываемая функция определяется на этапе компиляции, называется статическим полиморфизмом.

Динамический полиморфизм

Вид полиморфизма, при котором вызываемая функция определяется на этапе выполнения, называется динамическим полиморфизмом. При динамическом полиморфизме даже неизвестен тип аргумента, приходящего на вход функции на этапе компиляции, он становится известен только на этапе выполнения. Таким образом в языке должны существовать объекты, которые могут "менять" свой тип.

В языке Python динамический полиморфизм просто встроен в язык и любые объекты могут менять свой тип:

```
a, b = 10, 20
print(plus(a, b))     # Напечатает 30, тип объектов a и b - int
a, b = "Cat", "Dog"
print(plus(a, b))     # Напечатает CatDog, тип объектов a и b - строка
```

В языке C++ у всех объектов фиксированный тип, а динамический полиморфизм достигается с помощью наследования и механизма виртуальных функций.

Часть 2: Виртуальные функции

Виртуальные функции (также известные как виртуальные методы) – это механизм, который позволяет осуществлять динамический полиморфизм в языке C++. Виртуальные функции позволяют динамически определять, какая функция будет вызвана в зависимости от типа объекта, а не типа указателя или ссылки, указывающего на этот объект. Виртуальные функции не следует путать с виртуальным наследованием, это разные вещи.

Чтобы указать, что какой-либо метод является виртуальным, нужно использовать ключевое слово `virtual` в объявлении метода в базовом классе. После этого этот метод, а также все методы с такой же сигнатурой во всех производных классах станут виртуальными. Использовать `virtual` в объявлениях методов производных классов необязательно. Виртуальный метод отличается от обычного следующим:

Если в программе возникнет ситуация когда указатель (или ссылка) на базовый класс будет указывать на объект производного класса, то при вызове виртуального метода через этот указатель (или ссылку) будет вызываться метод производного класса, а не метод базового класса.

Рассмотрим пример класса `Bob`, который наследуется от класса `Alice` и у этих классов есть виртуальные методы под названием `say`. Метод `say` в классе `Alice` стал виртуальным, так как мы пометили его ключевым словом `virtual`. Метод `say` в классе `Bob` также будет виртуальным, потому что он переопределяет виртуальный метод в родительском классе. Таким образом, независимо от того пометим мы метод `Bob::say` словом `virtual` или нет, он будет виртуальным.

```
struct Alice
{
    virtual void say()
    {
        std::cout << "Alice" << std::endl;
    }
};

struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};
```

Виртуальные методы отличаются от обычных методов тем, что если мы вызываем этот метод через указатель (или ссылку) на объект, то будет вызываться метод класса объекта на который указывает этот указатель.

```
Bob b;
Alice* pa = &b;
pa->say();  // Вызовется метод Bob::say, так как pa указывает на объект типа Bob
            // Если бы метод say не был бы виртуальным, то вызвался бы метод Alice::say,
            // так как указатель имеет тип Alice*
```

Обратите внимание, что виртуальные функции работают в случае когда есть указатель (или ссылка) на базовый класс, который указывает на объект производного класса, но не работают когда, например, объект базового класса инициализирован объектом производного класса.

Объект базового класса, инициализированный объектом производного

```
#include <iostream>
struct Alice
{
    void say()
    {
        std::cout << "Alice" << std::endl;
    }
};
struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Bob b;
    Alice a = b;
    a.say();  // Напечатает Alice
}
```

```
#include <iostream>
struct Alice
{
    virtual void say()
    {
        std::cout << "Alice" << std::endl;
    }
};
struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Bob b;
    Alice a = b;
    a.say();  // Всё равно напечатает Alice
}
```

Указатели на базовый класс, хранящие адрес объекта производного класса

```
#include <iostream>
struct Alice
{
    void say()
    {
        std::cout << "Alice" << std::endl;
    }
};
struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Bob b;
    Alice* pa = &b;
    pa->say();  // Напечатает Alice
}
```

```
#include <iostream>
struct Alice
{
    virtual void say()
    {
        std::cout << "Alice" << std::endl;
    }
};
struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Bob b;
    Alice* pa = &b;
    pa->say();  // Напечатает Bob
}
```

Вызов виртуальных функций из методов класса

Ещё один случай когда работают виртуальные функции – если мы вызываем виртуальный метод из другого метода (не важно виртуального или не виртуального).

```cpp
#include <iostream>
struct Alice
{
    void say()
    {
        std::cout << "Alice" << std::endl;
    }

    void func()
    {
        say();
    }
};

struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Bob b;
    b.func();  // Напечатает Alice
}
```

```cpp
#include <iostream>
struct Alice
{
    virtual void say()
    {
        std::cout << "Alice" << std::endl;
    }

    void func()
    {
        say();
    }
};

struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Bob b;
    b.func();  // Напечатает Bob
}
```

Как виртуальные функции приводят к динамическому полиморфизму

Указатель типа `Alice*` может указывать как на объект типа `Alice` или на объект типа `Bob` (или на объект любого другого наследника `Alice`). В зависимости от того куда указывает этот указатель будет вызываться та или иная виртуальная функция. То, на какой объект будет указывать `Alice`, часто будет известно только на этапе выполнения. Таким образом метод `say` может работать либо с объектом типа `Alice` либо с объектом типа `Bob` и то, с каким объектом будет работать данный метод, зависит от значения указателя `Alice*` и будет известно только на стадии выполнения.

```cpp
Alice a; Bob b;

Alice* p;
int x;
std::cin >> x;
if (x == 0) p = &a;
else        p = &b;

p->say();  // Напечатает Alice или Bob, в зависимости от того, какое число было
           // введено на этапе выполнения
```

Виртуальные функции в конструкторах и деструкторах

Если производный класс вызывает конструктор базового класса (это происходит автоматически в перед вызовом конструктора производного класса) и в конструкторе базового класса вызывается виртуальный метод, то вызовется метод базового класса. То есть в этом случае виртуальность метода "не работает". Это происходит просто потому что объект производного класса не может вызывать свои методы, так как он ещё не готов к этому.

```
#include <iostream>
struct Alice
{
    Alice()     {say();}
    void func() {say();}
    virtual void say()
    {
        std::cout << "Alice" << std::endl;
    }
};

struct Bob : public Alice
{
    void say()
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Bob b;     // Вызовет конструктор Alice(), который вызовет Alice::func и напечатает Alice
    b.func();  // Вызовет Bob::func и напечатает Bob
}
```

Похожая ситуация будет и при вызове виртуального метода в деструкторе:

```
#include <iostream>
struct Alice
{
    virtual void say() {std::cout << "Alice" << std::endl;}
    void func() {say();}
    virtual ~Alice()    {say();}
};

struct Bob : public Alice
{
    void say() {std::cout << "Bob" << std::endl;};
    ~Bob() {}
};

int main()
{
    Bob b;
    b.func();  // Напечатает Bob
               // Напечатает Alice при уничтожении объекта
}
```

Виртуальный деструктор

Одна из самых распространённых ошибок при работе с виртуальными функциями – это забыть сделать деструктор базового класса виртуальным. Это может привести к тому, что при удалении объекта производного класса, через указатель на базовый класс, не будет вызываться деструктор производного класса. Это может привести к ошибкам, например в примере ниже это приводит к утечке памяти.

```
#include <iostream>

struct Alice
{
    virtual void say()
    {
        std::cout << "Alice" << std::endl;
    }
};

struct Bob : public Alice
{
    char* data;

    Bob()
    {
        std::cout << "Allocating Memory" << std::endl;
        data = new char[100];
    }

    void say()
    {
        std::cout << "Bob" << std::endl;
    }

    ~Bob()
    {
        std::cout << "Freeing Memory" << std::endl;
        delete[] data;
    }
};

int main()
{
    Alice* p = new Bob;  // Напечатает Allocating Memory
    p->say();            // Напечатает Bob, так как функция say - виртуальная
    delete p;            // НЕ напечатает Freeing Memory, так как деструктор НЕ виртуальный
}
```

Решение проблемы – нужно просто пометить деструктор базового класса словом `virtual`, то есть добавить в класс `Alice` строку:

```
virtual ~Alice() {}
```

`override` и `final`

Полиморфизм и умные указатели

Как программа понимает, какой виртуальный метод вызывать

Размер объектов полиморфных типов. Скрытый указатель на таблицу виртуальных функций.

Реализация механизма виртуальных функций

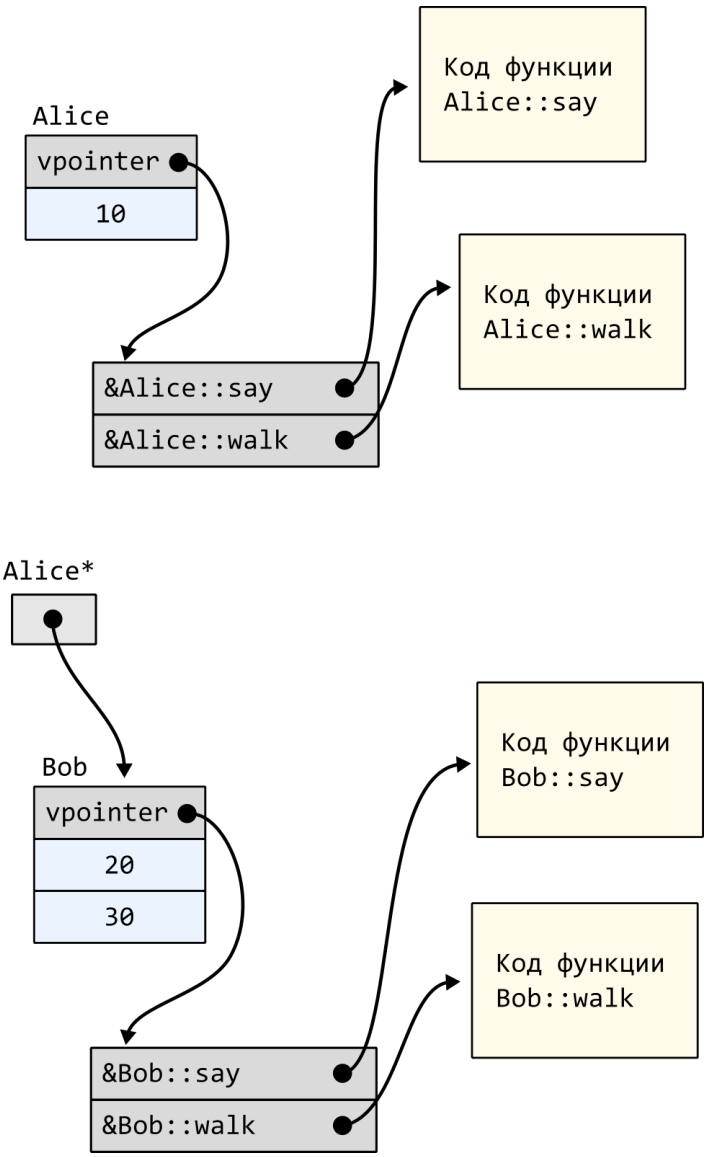
```cpp
#include <iostream>
struct Alice
{
    int x;
    virtual void say()
    {
        std::cout << "Alice Say\n";
    }
    virtual void walk()
    {
        std::cout << "Alice Walk\n";
    }
};

struct Bob : public Alice
{
    int y;
    void say() override
    {
        std::cout << "Bob Say\n";
    }
    void walk() override
    {
        std::cout << "Bob Walk\n";
    }
};

int main()
{
    Alice a {10};
    Bob b {20, 30};

    Alice* p = &a;
    p->say();  // Напечатает Alice Say

    p = &b;
    p->say();  // Напечатает Bob Say
}
```



Контейнер указателей на базовый класс, хранящих адреса объектов наследников

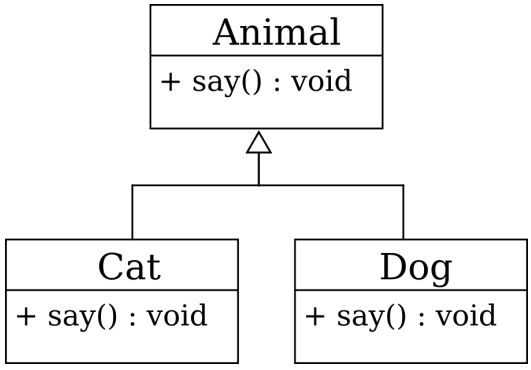
Одна из самых полезных возможностей, которую даёт динамический полиморфизм – это возможность единообразно хранить и обрабатывать объекты разных типов. Для того, чтобы это сделать, создадим контейнер, который будет хранить объекты типа 'указатель на базовый класс'. Как мы знаем, такие указатели могут указывать как на объекты базового класса, так и на объекты производных классов. При вызове виртуального метода через такой указатель, будет вызываться метод того класса, на который указатель указывает. Таким образом, если мы пройдём по массиву и вызовем виртуальный метод, то будут вызываться разные методы в зависимости от того куда указывает конкретный указатель.

```
#include <iostream>
#include <vector>

struct Animal
{
    virtual void say() const
    {
        std::cout << "Hello" << std::endl;
    }
    virtual ~Animal() {};
};

struct Cat : public Animal
{
    void say() const override
    {
        std::cout << "Meow" << std::endl;
    }
};

struct Dog : public Animal
{
    void say() const override
    {
        std::cout << "Woof" << std::endl;
    }
};
```

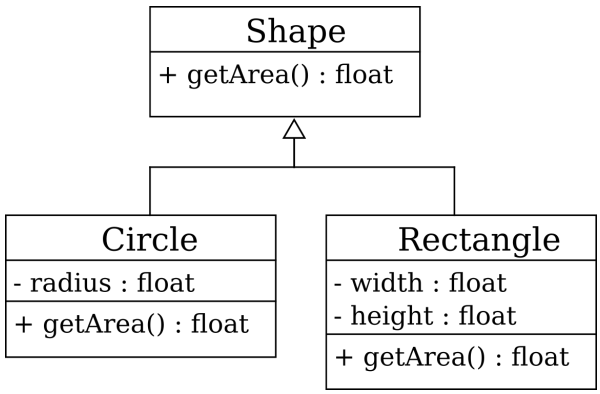



```
int main()
{
    std::vector<Animal*> animals = {new Cat, new Dog, new Dog, new Animal, new Cat};

    // В цикле напечатается Meow Woof Woof Hello Meow
    for (auto p : animals)
        p->say();

    for (auto p : animals)
        delete p;
}
```

Пример использования полиморфизма: Фигуры в 2-х измерениях




```cpp
#include <iostream>
#include <numbers>
#include <vector>

class Shape
{
public:
    virtual float getArea() const {return 0;}
    virtual ~Shape() {}
};

class Circle : public Shape
{
private:
    float radius;
public:
    Circle(float radius) : radius(radius) {}

    float getArea() const override
    {
        return std::numbers::pi * radius * radius;
    }
};

class Rectangle : public Shape
{
private:
    float width;
    float height;
public:
    Rectangle(float width, float height) : width(width), height(height) {}

    float getArea() const override
    {
        return width * height;
    }
};

int main()
{
    std::vector<Shape*> shapes = {new Rectangle{2, 5}, new Circle(1), new Rectangle{10, 1}};

    float areaSum = 0;
    for (auto p : shapes)
        areaSum += p->getArea();
    std::cout << "Sum of areas of all shapes = " << areaSum << std::endl; // 23.1416

    for (auto p : shapes)
        delete p;
}
```

Часть 3: Абстрактные классы и интерфейсы

Чистые виртуальные функции

Чистая виртуальная функция (англ. *pure virtual function*) – это виртуальная функция, в объявлении которой прописывается, то что она "равна нулю", например вот так:

```
virtual void say() = 0;
```

У таких функций, как правило, нет определения и они не предназначены для того, чтобы их вызывали. Единственная их цель заключается в том, чтобы эти функции были переопределены в классах наследниках.

Абстрактные классы

Абстрактный класс – это класс у которого есть хотя бы одна чистая виртуальная функция (либо своя, либо отнаследованная). Объект абстрактного класса нельзя создать. Такие классы предзазначены только для того, чтобы от них наследоваться.

```
#include <iostream>
struct Alice
{
    virtual void say() = 0;
    virtual ~Alice() {}
};

struct Bob : public Alice
{
    void say() override
    {
        std::cout << "Bob" << std::endl;
    }
};

int main()
{
    Alice a;              // Ошибка: нельзя создать объект класса Alice
    Alice* p = new Alice; // Ошибка: нельзя создать объект класса Alice

    Alice* q = new Bob;   // OK
    q->say();             // Напечатает Bob
    delete q;
}
```

Интерфейс

Интерфейс – это абстрактный класс, у которого нет полей, в все методы (за исключением, быть может, деструктора) – чистые виртуальные методы.

pure virtual call и определение чистых виртуальных методов

RTTI и dynamic_cast

Полиморфный тип

dynamic_cast

dynamic_cast от родителя к ребёнку

dynamic_cast в бок

Оператор typeid и класс std::type_info