Лекция по C++

ООП, указатели, ссылки, управление памятью

Указатели

Указатель - переменная, которая хранит в себе адрес ячейки памяти.

```
int main() {
    int some_int;
    int* pointer_to_int;
}
```

Указатели

& - оператор взятия адреса

$*ptr$ - разыменование указателя *ptr*

```
int main() {
    int some_num = 137;
    int* first_ptr = &some_num;
    ++(*first_ptr); // same as ++some_num
    std::cout << *first_ptr; // 138
}
```

Указатели

```
int main() {
    int one = 1;
    int* first_ptr = &one;
    int* second_ptr = &one;
    std::cout << first_ptr << " " << second_ptr;
    // first_ptr == second_ptr
}
```

Ссылки

```
int main() {
    int some_variable = 42;
    int& alias = some_variable;
    // alias - ссылка на переменную some_variable
    ++alias;
    std::cout << some_variable; // 43

    // у них один и тот же адрес
    assert(&alias == &some_variable); // ok
}
```

Ссылки и указатели

В чем разница между ссылкой и указателем?

```
int main() {
    int* b;
    b = nullptr;
    // int& a; - не скомпилируется, ссылку обязательно
    // инициализировать при создании, а указатель может
    // быть ничем не проинициализирован

    int first_int = 1, second_int = 2;
    b = &first_int;
    b = &second_int;
    // указатель может указывать на разные объекты,
    // ссылка не может быть переопределена
}
```

new и delete

Оператор *new* динамически выделяет память на объект в куче

```
int* a = new int(13);
```

Оператор *delete* освобождает память, выделенную с помощью new

```
delete a;
```

Вызов *new* без вызова *delete* приведет к утечке памяти. Не делайте так.

Ссылки и указатели на классы

```
struct A {
    A() = default;
};

struct B : A {
    B() = default;
};

int main() {
    B derived;
    A& parent = derived;

    A* ptr = new B();
    delete ptr;
}
```

Это легально, поскольку класс B наследуется от класса A

Виртуальные фуннкции

```
struct A {
    void Print() { std::cout << "This is A!"; }
};

struct B : A {
    void Print() { std::cout << "This is B!"; }
};

int main() {
    A* sample = new B();
    sample->Print(); // will print "This is A!"
    delete sample;
}
```

Виртуальные фуннкции

```
struct A {
    virtual void Print() { std::cout << "This is A!"; }
};

struct B : A {
    void Print() override { std::cout << "This is B!"; }
};

int main() {
    A* sample = new B();
    sample->Print(); // will print "This is B!"
    delete sample;
}
```

Виртуальные фуннкции

```
struct A {
    virtual void Print() { std::cout << "A!"; }
};

struct B : A {
    void Print(int num) override { std::cout << "B!"; }
};
```

error: non-virtual member function marked 'override' hides virtual member function

...

note: hidden overloaded virtual function 'A::Print' declared here: different number of parameters (0 vs 1)

...

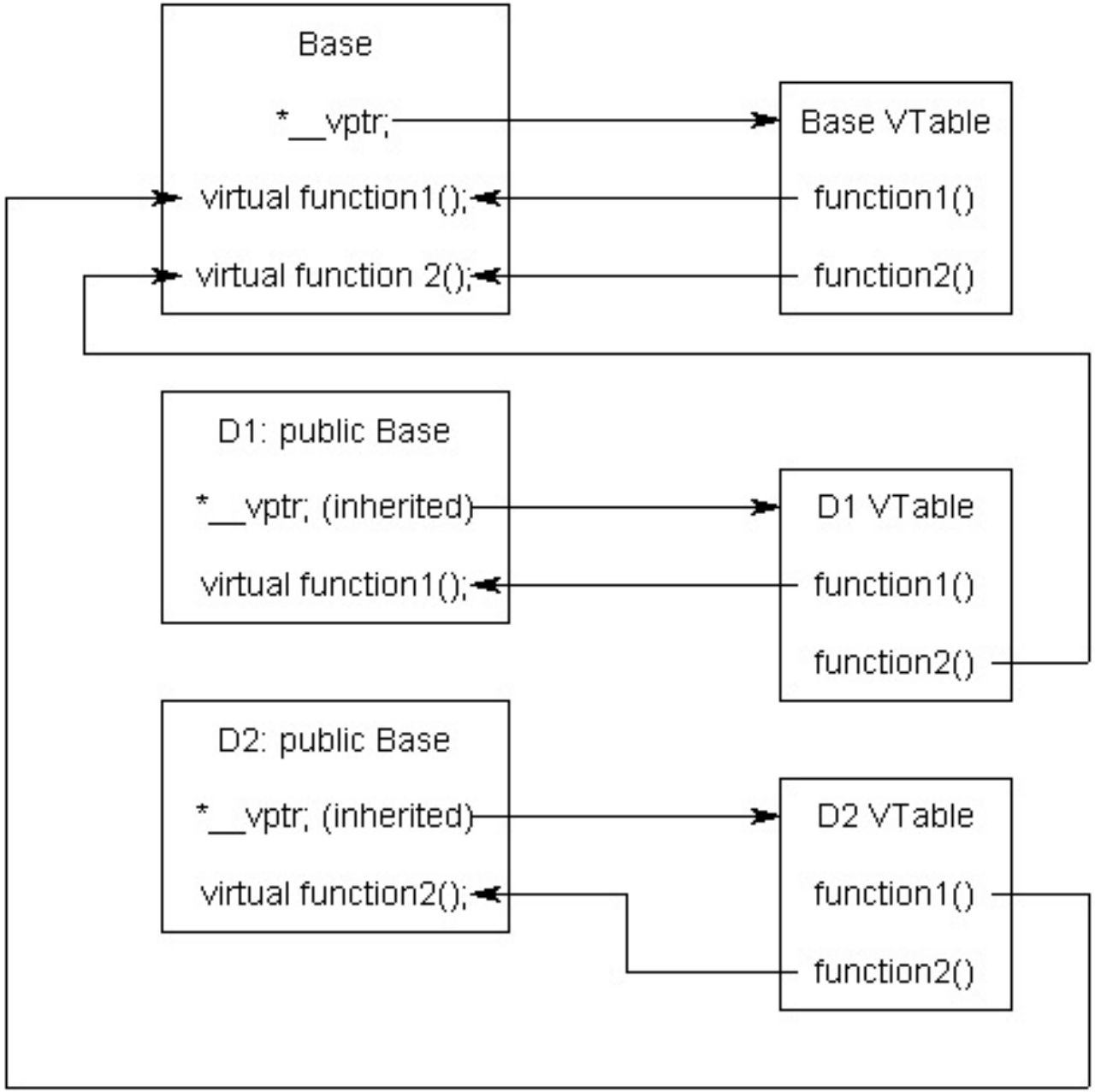
Таблица виртуальных фуннкций

```
struct Base {
    virtual void function1() {};
    virtual void function2() {};
};

struct D1: Base {
    void function1() override {};
};

struct D2: Base {
    void function2() override {};
};
```

Таблица виртуальных фуннкций



Утечка памяти

```
struct A {
    ~A() { std::cout << "A destructor called"; };
};
struct B : A {
    ~B() { std::cout << "B destructor called"; };
    Data some_really_heavy_data_;
};

int main() {
    A* test = new B();
    delete test; // "A destructor called"
}
```

Деструктор B никогда не вызовется, что приведет к утечке памяти.

Виртуальные деструкторы

```
struct A {
   virtual ~A() { std::cout << "A destructor called"; };
};
struct B : A {
    ~B() { std::cout << "B destructor called"; };
    Data some_really_heavy_data_;
};

int main() {
    A* test = new B();
    delete test;
}
```

Виртуальные функции

Не вызывайте виртуальные методы в конструкторах!

```
struct A {
    A() { Foo(); }
    virtual void Foo() { n_ = 1; }
    int n_;
};

struct B : A {
    B() : A() {}
    void Foo() override { n_ = 2; }
};

int main(){
    B b;
    std::cout << b.n_; // 1 ?
}
```

Статический полиморфизм

```
struct A {
    void Print(int a) {
        std::cout << a << std::endl;
    }
    void Print(const std::string& a) {
        std::cout << a << std::endl;
    }
};

int main() {
    A test;
    test.Print(12);
    test.Print("asdjls");
}
```

Виртуальное наследование

```
struct Animal {
    virtual void eat() { std::cout << "eat!"; };
};
struct Mammal : Animal {};
struct WingedAnimal : Animal {};
struct Bat : Mammal, WingedAnimal {};

int main() {
    Bat bat;
    bat.eat();
}
```

error: non-static member 'eat' found in multiple base-class subobjects of type 'Animal':
structBat− > structMammal− > structAnimal
structBat− > structWingedAnimal− > structAnimal

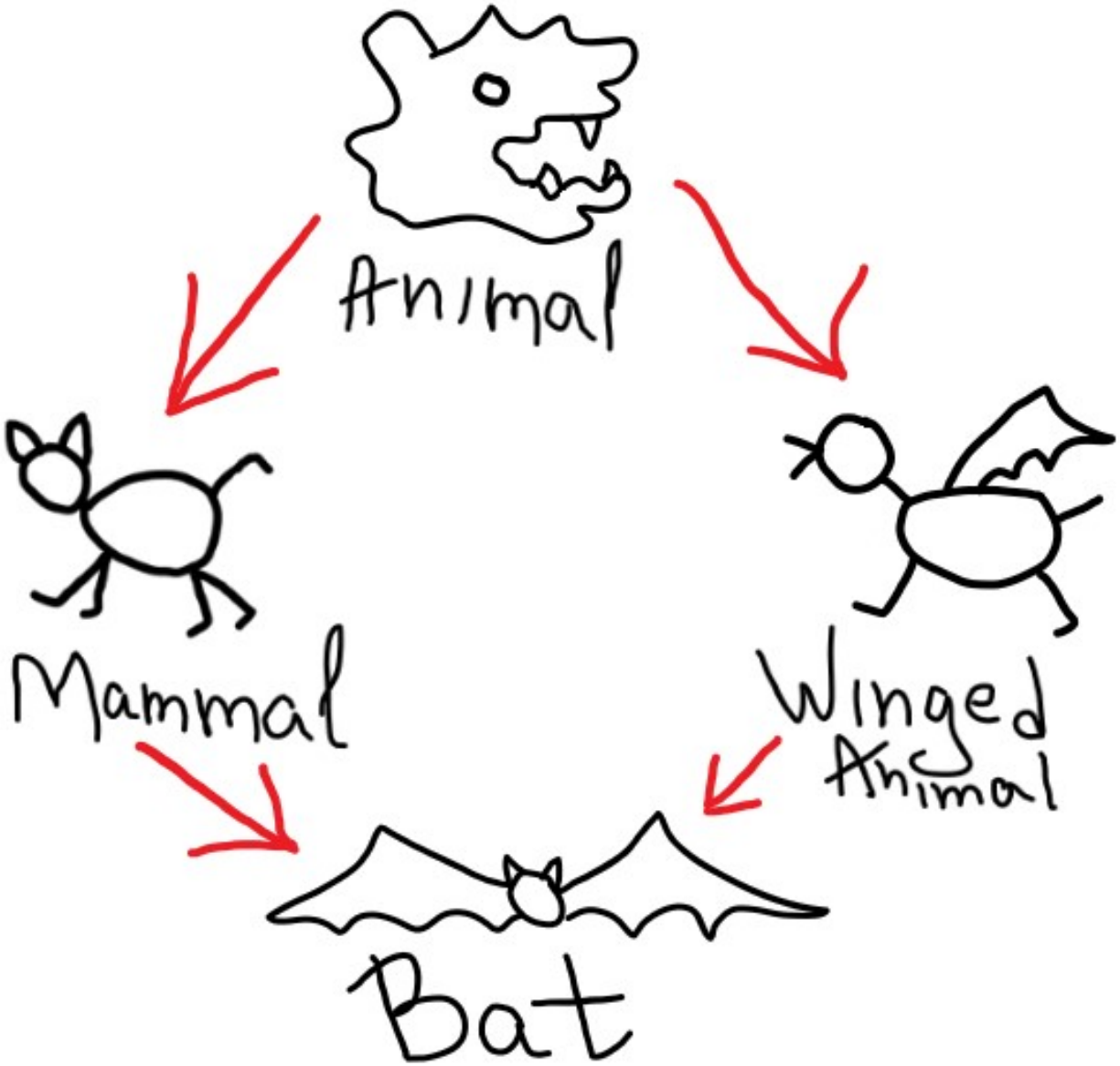
Виртуальное наследование

```
struct Animal {
    virtual void eat() { std::cout << "eat!"; };
};
struct Mammal : Animal {};
struct WingedAnimal : Animal {};
struct Bat : Mammal, WingedAnimal {};

int main() {
    Bat bat;
    bat.eat();
}
```

error: non-static member 'eat' found in multiple base-class subobjects of type 'Animal':
structBat− > structMammal− > structAnimal
structBat− > structWingedAnimal− > structAnimal

Виртуальное наследование



Виртуальное наследование

```
struct Animal {
    virtual void eat() { std::cout << "eat!"; };
};
struct Mammal : virtual Animal {};
struct WingedAnimal : virtual Animal {};
struct Bat : Mammal, WingedAnimal {};

int main() {
    Bat bat;
    bat.eat();
}
```

Абстрактные классы

Абстрактный класс - класс, содержащий хотя бы один чистый виртуальный метод. Нельзя создать экземпляр абстрактного класса.

```
struct A {
    // чистый виртуальный метод
    virtual void Print() = 0;
};

struct B : public A {
    void Print() override { std::cout << "B!"; }
};

int main() {
    A test;
}
```

error: variable type 'A' is an abstract class

Const

```
int a = 2;
const int cs = 3; // константа - неизменяемая переменная
++cs; // error: cannot assign to variable 'cs'
// with const-qualified type 'const int'

const int& rf = a; // константная ссылка
++rf; // error: cannot assign to variable 'rf'
// with const-qualified type 'const int &'
```

Const

```
int a = 2, b = 3;
const int* ptr = &a; // указатель на константу
ptr = &b; // ok
++(*ptr); // error: read-only variable is not assignable

int* const cptr = &a; // константный указатель
cptr = &b; // error: expression is not assignable
++(*cptr); // ok

const int* const cc = &a; // константный
// указатель на константу
```

Const

```
const int a = 2;

const int& b = a; // ok
int& b = a; //  error: binding value of type 'const int'
//to reference to type 'int' drops 'const' qualifier

const int* ptr = &a; // ok
int* ptr = &a; // error: : cannot initialize a variable
// of type 'int *' with an rvalue of type 'const int *'
```

Константные методы

Константные методы не меняют состояние класса

Всегда делайте метод константным, если это возможно!

```
struct A {
    int Foo() const {
        return 4;
    }
};

int main() {
    A test;
    int val = test.Foo();
}
```

Константные методы

Из константных методов нельзя вызывать неконстантные:

```
struct A { // ok
    int Foo() const { return 4; }
    int Get4() { return Foo(); }
};

struct A { // error
    int Foo() { return 4; }
    int Get4() const { return Foo(); }
};
```

Константные методы

```
struct A {
    int Foo() const {
        val_ = 13;
        return val_;
    }
    int val_;
};
```

error: cannot assign to non-static data member within const member function 'Foo'

Но что если очень-очень нужно поменять val_ ?

Константные методы, mutable

```
struct A {
    int Foo() const {
        val_ = 13;
        return val_;
    }
    mutable int val_;
};
```

Но лучше так никогда не делать!

Static

```
struct Rnd {
    static int Random() {
        return 4;
    }
};

int main() {
    Rnd r;
    std::cout << r.Random() << std::endl;
    std::cout << Rnd::Random() << std::endl;
}
```

Static методы общие для всех экземпляров класса.
Их можно вызывать, не создавая экземпляр класса.

Local Static Object

```
int NextNum() {
    static int num = 1;
    return num++;
}

int main() {
    std::cout << NextNum() << std::endl; // 1
    std::cout << NextNum() << std::endl; // 2
    std::cout << NextNum() << std::endl; // 3
}
```

Передача аргументов в функцию

```
void Inc(int a) { // по значению
    ++a;
}

void Inc_(int& a) { // по ссылке
    ++a;
}

int main() {
    int b = 1;
    Inc(b);
    std::cout << b << std::endl; // 1

    Inc_(b);
    std::cout << b << std::endl; // 2
}
```

Передача аргументов в функцию

В случае, если вы не собираетесь менять объект, передавайте его по константной ссылке:

```
struct Pair {
    Pair(int l, int r) : l(l), r(r) {}
    int l, r;
};

int Sum(const Pair& val) {
    return val.l + val.r;
}

int main() {
    Pair p(1, 2);
    std::cout << Sum(p);
}
```

Передача аргументов в функцию

В случае, если вы собираетесь менять объект, передавайте его по указателю. Почему?

```
int Foo(const Data& d1, Data& d2, Data d3, Data* d4) {
    ...
}

int main() {
    Data v1, v2, v3, v4;
    Foo(v1, v2, v3, &v4); // может поменять v2 и v4
}
```

Перегрузка операторов

```
struct Pair {
    Pair(int l, int r) : l(l), r(r) {}

    bool operator<(const Pair& rhs) const {
        if (l == rhs.l) {
            return r < rhs.r;
        }
        return l < rhs.l;
    }
  private:
    int l, r;
};

int main() {
    Pair a(1, 4), b(2, 3);
    std::cout << (a < b);
}
```

Friend

```
class Y {
    int data; // private member
    // the non-member function operator<< will
    // have access to Y's private members
    friend std::ostream& operator<<(std::ostream& out,
        const Y& o);
    friend char* X::foo(int);
    // members of other classes can be friends too
    friend X::X(char), X::~X();
    // constructors and destructors can be friends
};

std::ostream& operator<<(std::ostream& out, const Y& y) {
    return out << y.data;
}
```