

Объектно-ориентированное программирование (ООП)

Язык C++

Загорулько Г.Б.

Язык C++

- Разработан в начале 80-х годов
- **Бьёрном Страуструпом**
- Поддерживает парадигмы программирования:
 - процедурное
 - объектно- ориентированное
 - обобщенное

Три кита ООП

- Инкапсуляция (encapsulation)
- Полиморфизм (polymorphism)
- Наследование (inheritance)

Инкапсуляция



- Объединение данных и функций их обработки
- Скрытие информации, ненужной для использования данных

Полиморфизм



- в биологии - наличие в пределах одного вида резко отличных по облику особей
- в языках программирования -
взаимозаменяемость объектов с одинаковым интерфейсом
«Один интерфейс, множество реализаций».

Наследование



- Возможность создания иерархии классов
- Наследование потомками свойств предков
- Возможность изменения наследуемых свойств и добавления новых

Классы. Инкапсуляция. Полиморфизм.

- Основные понятия:
 - Описание класса
 - Конструкторы и деструкторы
 - Ссылки и указатели. Указатель **this**
 - Функции и операции
 - Перегрузка функций и операторов

Описание класса

Класс – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью.

С точки зрения программирования, **класс** является абстрактным типом данных, определяемым пользователем, который содержит набор данных (полей, атрибутов, членов класса) и функций для работы с ними (методов).

Описание класса

```
class myclass
{
private:           //ключ доступа
    int a;        //члены-данные, свойства
    float b;      //структура в языке C
public:
    void setvalue(int, float); //члены-функции,
    int geta();              //методы,
    float getb();
};
```

Описание класса

- `void myclass::setvalue(int sa, float sb)`
- `{`
- `a=sa;`
- `b=sb; //или this->b=sb;`
- `}`
- `int myclass::geta()`
- `{`
- `return a;`
- `}`
- `float myclass::getb()`
- `{`
- `return b;`
- `}`
- `void main()`
- `{`
- `myclass mc;`
- `cout<<mc.geta()<<"\n"<<mc.getb()<<"\n";`
- `mc.setvalue(31, 3.5);`
- `cout<<mc.geta()<<"\n"<<mc.getb()<<"\n";`
- `}`

Конструкторы и деструкторы класса

```
#include <iostream>
using namespace std;

class myclass
{
private:
    int a;
    float b;
    int *m;
public:
    myclass();                //конструктор по умолчанию
    myclass(int, float);
    myclass(int, float, int*);
    myclass(const myclass &); //конструктор копирования
    ~myclass();               //деструктор
    void print();
};
```

Конструкторы и деструкторы класса

```
myclass::myclass()
{
    a=0;
    b=0.0;
    m = new int[5];
}
```

```
myclass::myclass(int n, float f)
{
    m = new int[5];
    this->a=n;
    this->b=f;
}
```

```
myclass::myclass(int n, float f, int *p)
{
    m = new int[5];
    a=n;
    b=f;
    for (int i=0; i<5; i++)
        m[i]=p[i];
}
```

```
myclass::myclass(const myclass & mc)
{
    if (mc.m)
    {   m= new int[5];
        for (int i=0; i<5; i++) m[i]=mc.m[i];
    }
    else m=0;
    a=mc.a; b=mc.b;
}
myclass::~myclass()
{
    delete [] m;
}
void myclass::print()
{
    cout<<"a="<<a<<"\nb="<<b<<"\nm=";
    for (int i=0; i<5; i++)
        cout<<" "<<m[i];
    cout<<"\n";
}
void main()
{
    int dig[]={1,2,3,4,5};
    myclass mc(12, 25.6, dig);
    mc.print();
}
```

Пример. Телефонная станция

- **Файл tele.h**
- `#include <iostream>`
- `#include <string>`
- `#include <locale>`
- `using namespace std;`
- `#define ssn 5 //количество абонентов`
- `class Exchange;`
- `class Phone`
- `{`
- `private:`
- `string dialtype; //ДИСКОВЫЙ или КНОПОЧНЫЙ`
- `string color;`
- `int price;`
- `string num; //члены-данные, свойства`
- `public:`
- `Phone();`
- `Phone(string, string);`
- `Phone(string, string, string, int);`
- `Phone(string);`
- `void ring(); //звонить`
- `void dial(); //набирать номер`
- `void accept(string); //принимать звонок с номера`
- `void propagate(string, Exchange); //вызывать номер`
- `void info(); //выводить информацию о себе`
- `string getNumber();`
- `};`
- `class Exchange`
- `{`
- `private:`
- `Phone subscribers[ssn];`
- `public:`
- `Exchange(Phone *);`
- `void transferCall(string, string);`
- `};`

Пример. Телефонная станция

- **Файл tele.cpp**
- `#include "tele.h"`
- `Phone::Phone(){}
Phone::Phone(string, string){}
Phone::Phone(string dt, string c, string n, int p)
{
 dialtype=dt;
 color=c;
 num=n;
 price=p;
}
Phone::Phone(string n)
{
 dialtype="but";
 color="";
 num=n;
 price=100;
}`
- `void Phone::info()
{ cout<<"Я - телефон номер "<<num<<"\n"; }
void Phone::ring()
{ cout<<"Я звоню\n"; }
void Phone::dial()
{ cout<<"Я набираю номер\n"; }
void Phone::accept(string abNum)
{
 info();
 ring();
 cout<<"Меня вызывает абонент номер"<<abNum<<"\n\n";
}
void Phone::propagate(string abNum, Exchange ex)
{
 info();
 dial();
 cout<<"Я вызываю абонента номер "<<abNum<<"\n\n";
 ex.transferCall(num, abNum);
}
string Phone::getNumber()
{ return num;}`

Пример. Телефонная станция

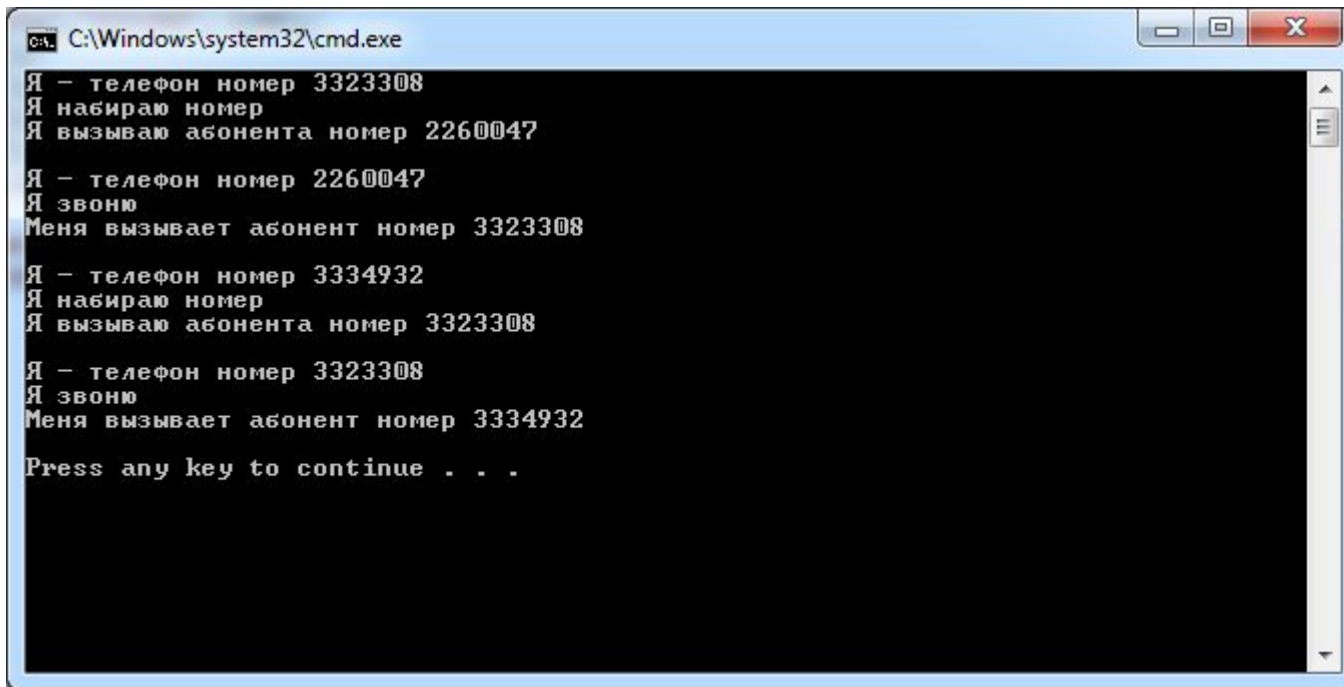
Файл exchange.cpp

```
• #include "tele.h"
• Exchange::Exchange(Phone *sss)
• {
•     int i;
•     for (i=0; i<ssn; i++)
•         subscribers[i]=sss[i];
• }
• void Exchange::transferCall(string from, string to)
• {
•     int i;
•     for (i=0; i<ssn; i++)
•         if(subscribers[i].getNumber()==to)
•             break;
•     if (i==ssn)
•     {
•         cout<<"Набранный вами номер не существует\n";
•         return;
•     }
•     subscribers[i].accept(from);
• }
```

Файл teletest.cpp

```
• #include "telephone.h"
•
• void main()
• {
•     setlocale(LC_ALL,"Russian");
•
•     Phone p1("but","white","3323308", 1200);
•     Phone p2("3328359");
•     Phone p3("3063560");
•     Phone p4("2260047");
•     Phone p5("3334932");
•     Phone sss[ssn]={p1,p2,p3,p4,p5};
•     Exchange ex(sss);
•     p1.propagate("2260047", ex);
•     p5.propagate("3323308",ex);
• }
```

Пример. Телефонная станция



```
C:\Windows\system32\cmd.exe
Я - телефон номер 3323308
Я набираю номер
Я вызываю абонента номер 2260047

Я - телефон номер 2260047
Я звоню
Меня вызывает абонент номер 3323308

Я - телефон номер 3334932
Я набираю номер
Я вызываю абонента номер 3323308

Я - телефон номер 3323308
Я звоню
Меня вызывает абонент номер 3334932

Press any key to continue . . .
```

Указатели и ссылки

Указатель – переменная, значением которой является адрес некоторой области памяти.

```
int *a, n;      *a=10; a=&n;
```

```
float *b;      .....
```

```
char *c;       .....
```

```
void *f;       .....
```

Указатели и ссылки на объект

```
myclass *pmc, mc1, mc2(45, 3.5);
```

При объявлении указателя на объект
выделяется память только для указателя!

```
pmc->a=23; //ошибка-не выделена память под объект
```

```
pmc=&mc1;
```

```
*pmc=mc2; pmc->a=23; (*pmc).b=12.05;
```

Указатели и ссылки

Ссылка – понятие, родственное указателю. Является скрытым указателем. Во всех случаях ее можно использовать как еще одно имя переменной

Ссылку можно:

1. Передавать в функцию
2. Возвращать из функции
3. Использовать как независимую переменную

При использовании ссылки как независимой переменной, она должна быть проинициализирована при объявлении

1. `myclass mc(12, 25.6, dig), &s=mc;`

После этого `s` можно считать еще одним именем переменной `mc`.

Указатели. Передача в функцию

```
void swap(int *a, int *b)
{
    int d;
    d=*a;
    *a=*b;
    *b=d;
}
void main()
{
    int a=10, b=20;
    cout<<"a="<<a<<" b="<<b<<"\n";
    swap(&a,&b);
    cout<<"a="<<a<<" b="<<b<<"\n";
}
```

Ссылки. Передача в функцию

```
void swp(int &a, int &b)
{
    int d;
    d=a;
    a=b;
    b=d;
}
void main()
{
    int a=10, b=20;
    cout<<"a="<<a<<" b="<<b<<"\n";
    swp(a,b);
    cout<<"a="<<a<<" b="<<b<<"\n";
}
```

Указатель **this**

С++ содержит специальный указатель **this**. Он автоматически передается любой функции-члену при ее вызове и указывает на объект, генерирующий вызов.

Перегрузка функций

- **Сигнатурой** функции называют список типов ее параметров и возвращаемого значения.
- В C++ можно определять функции с одним и тем же именем, но разной сигнатурой. Эта возможность называется **перегрузкой** функции.
- Перегрузка функций является проявлением **полиморфизма**.

Операторы

```
class myclass
{
private:
    int a;
    float b;
    int *m;
public:
    myclass();
    myclass(int, float);
    myclass(int, float, int*);
    myclass(const myclass &);
    ~myclass();
    void print();
    myclass & operator=(const myclass &);
};
```

Оператор ***** можно рассматривать как функцию с именем **operator***

Вызов этой функции происходит без операции «.»:

x=y;

или, что менее удобно:

x.operator=(y);

Оператор присваивания

```
myclass & myclass::operator=(const myclass &mc)
{
    m= new int[5];
    for (int i=0; i<5; i++)
        m[i]=mc.m[i];
    a=mc.a; b=mc.b;
    print();
    return *this;
}
```

Задание 1. Строки

Реализовать класс **MyString**

```
class
    MyString
{
private:
    char *data;
    ...
};
```

Класс должен содержать:

1. Конструктор по умолчанию
2. Конструктор с параметром `char*`
3. Конструктор копирования
4. Деструктор
5. Функции для работы со строками
(length, concat, compare, insert, print)
6. Операторы для работы со строками
(=, +, +=, [])

Шаблоны

Шаблоны функций

Шаблоны классов

Шаблон позволяет отделить алгоритмы от конкретных типов данных.

Шаблон может применяться к любым типам данных без переписывания кода.

Шаблоны функций

- Шаблон функции – параметризованная функция, которая помимо обычных параметров имеет еще один – некоторый тип.
- Шаблоны функций чаще всего используются при создании функций, выполняющих одни и те же действия, но с данными различных типов.

Шаблоны функций

```
template <typename T>
T abs(T a)
{
    return (a >= 0) ? a : -a;
}
```

```
int main()
{
    int a=3, b=0;
    float f=-5.5, g=0.00;
    cout<<abs<int>(a)<<"\n"<<abs(b)<<"\n";
    cout<<abs(f)<<"\n"<<abs<float>(g)<<"\n";
    return 0;
}
```

При вызове функции `abs<T>()` указывать явно параметр `<T>` необязательно.

Шаблоны классов

- Шаблон класса – параметризованный класс, которому тип инкапсулированных в нем данных передается в качестве параметра.
- Чаще всего шаблоны используются при создании контейнерных классов

Шаблоны классов. Односвязный список

Непараметризованный класс

```
class LIST
{
private :
    class Node
    {
    public:
        int dat;
        Node * next;
        Node (int d=0)
        {
            dat=d; next=0;
        }
    };
    Node * head, *tail;
public:
    LIST (){head=0; tail=0}
    ~LIST ();
    void insert_beg (int);
    void insert_end (int);
    void del (int);
    int find(int);
    void display();
};
```

Параметризованный класс

```
template <class T> class LIST
{
private:
    class Node
    {
    public:
        T dat;
        Node * next;
        Node (T d=0)
        {
            dat=d;next=0;
        }
    };
    Node * head, *tail;
    friend class LIST;
public:
    LIST (){head=0; tail=0}
    ~LIST ();
    void insert_beg (T);
    void insert_end (T);
    void del (T);
    int find(T);
    void display();
};
```

Шаблоны классов.

Шаблоны функций

```
void LIST::insert_beg (int data)
{
    Node * nel=new Node(data);
    nel->next=head;
    head=nel;
}
```

```
template <class T>
void LIST <T>::insert_beg (T data)
{
    Node * nel=new Node(data);
    nel->next=head;
    head=nel;
}
```

Шаблоны классов. Использование

```
void main()
{
    LIST <char> lst;
    char i;
    do
    {
        cin>>i;
        if (i!=48)
            lst.insert_beg(i);
    } while (i!=48);
    lst.display();
}
```

Итераторы. Что это такое и зачем они нужны

Итератор — объект, предоставляющий доступ к элементам некоторого контейнера (массива, списка, стека) и позволяющий перебирать его элементы.

Главное предназначение итераторов — предоставление пользователю возможности обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или списком.

Проектирование класса итератора обычно тесно связано с соответствующим классом контейнера. Обычно контейнер предоставляет методы создания итераторов.

Итераторы. Что это такое и зачем они нужны

Итератор является обобщением понятия «указатель».

Как и указатель он указывает на отдельный элемент коллекции объектов (предоставляет доступ к элементу) и содержит функции для перехода к другому элементу списка (следующему или предыдущему).

Контейнер, который реализует поддержку итераторов, должен предоставлять первый элемент списка, а также возможность проверить, перебраны ли все элементы контейнера (является ли итератор конечным).

В зависимости от используемого языка и цели, итераторы могут поддерживать дополнительные операции или определять различные варианты поведения.

Итераторы. Что это такое и зачем они нужны

```
class Iterator {  
    Node *cur;  
public:  
    Iterator(Node *);  
    Iterator(const Iterator &);  
    Iterator & operator++();  
    int operator!=(Node *);  
    T& operator*();  
    T* operator->();  
};
```

```
void main() {  
    LIST <int> lst;  
    ...  
    LIST <int>::Iterator it(lst.begin());  
    while (it != lst.end())  
    {  
        cout << *it << " ";  
        ++it;  
    }  
    //выводятся на экран элементы списка  
    ...  
}
```

Задание **2.** Шаблоны классов

- Реализовать шаблон класса List (методы, объявленные в классе).
- Реализовать конструктор копирования и оператор присваивания для класса List.
- Реализовать итератор для класса List.

Интерфейс класса LIST

```
template <typename T>
class LIST {

//-----<Node-----
    class Node {
    private:
        T dat;
        Node *next;
        Node(T)
        friend class LIST;
    };
//-----Node>-----

    Node *head;
    public:

//-----<Iterator-----
    class Iterator {
        Node *cur, *prev;
    public:
        Iterator(Node *);
        Iterator(Iterator &);
        Iterator & operator++();
        Iterator & operator=(const Iterator &);
        int operator!=(Node *);
        int operator!=(const Iterator &);
        T& operator*(); //применяется для простых типов T
        T* operator->(); //применяется для структурных
        //ТИПОВ T
    }
};
//-----Iterator>-----

LIST();
LIST(const LIST &);
~LIST();

void insert_beg(T);
void insert_end(T);

void del(T);
void del(Iterator);
void display();

LIST<T> & operator=(LIST<T> & l);
Iterator & begin();
Iterator end();
Iterator & find(T);
};
//-----LIST>-----
```

Наследование

Наследование – механизм, поддерживающий

- построение иерархии классов
- Полиморфизм

```
class имя_произв_кл: ключ_доступа имя_баз_кл
{
    ....
};
```

Наследование

class Phone

- {
- protected:
- string dialtype; //ДИСКОВЫЙ, КНОПОЧНЫЙ, СЕНСОРНЫЙ
- string color;
- string num; //ЧЛЕНЫ-ДАННЫЕ, СВОЙСТВА
- int price;
- public:
- Phone();
- Phone(string, string);
- Phone(string, string, string, int);
- Phone(string);
- void accept(string); //ПРИНИМАТЬ ЗВОНОК С НОМЕРА
- void propagate(string, Exchange); //ВЫЗЫВАТЬ НОМЕР
- void info(); //ВЫВОДИТЬ ИНФОРМАЦИЮ О СЕБЕ
- string getNumber();
- };

Class CellPhone : public Phone

```
{
protected:
    string mobileOperator;
    string producer;
    string bodyStyle;
public:
    CellPhone();
    CellPhone(sring, string, string, string, string, string);
    void info(); //ВЫВОДИТЬ ИНФОРМАЦИЮ О СЕБЕ
};
```

Наследование. Ключевые понятия

- Ключи доступа
- Простое наследование. Конструкторы и деструкторы.
- Раннее и позднее связывание
- Виртуальные методы. Абстрактные классы
- Множественное наследование

Ключи доступа

Ключ доступа	Спецификатор в баз. классе	Доступ в произв. классе
private	private protected public	нет private private
protected	private protected public	нет protected protected
public	private protected public	нет protected public

Конструкторы и деструкторы

- Конструкторы (и операторы присваивания) не наследуются. В производном классе (ПК) **должен быть** собственный конструктор.

Порядок вызова конструкторов:

- Если в конструкторе ПК нет явного вызова конструктора базового класса (БК), то вызывается конструктор БК по умолчанию.
- Для иерархии, состоящей из нескольких уровней, конструкторы БК вызываются, *начиная с самого верхнего уровня*, а затем выполняется конструктор класса.

Конструкторы и деструкторы

Рассмотрим разные варианты работы конструкторов на примере двух простых классов.

```
#include <iostream>
using namespace std;

class Base {
protected:
    int x, y;
public:
    Base() {
        cout << "Base_default\n";
    }
    Base(int a) {
        cout << "Base_a\n";
    }
    Base(int a, int b) {
        cout << "Base_a_b\n";
    }
};
```

```
class Derived : Base {
protected:
    int x, z;
public:
    Derived() {
        cout << "Derived_default\n";
    }
    Derived(int a) {
        cout << "Derived_a\n";
    }
    Derived(int a, int b):Base(a,b){
        cout << "Derived_a_b\n";
    }
    Derived(int a, int b, int c) {
        cout << "Derived_a_b_c\n";
    }
};
```

Конструкторы просто выводят сообщение со своим именем, чтобы было понятно, какой именно конструктор отработал

Конструкторы и деструкторы

Создадим несколько объектов производного класса и посмотрим, какие конструкторы и в каком порядке будут отработывать

```
int main(){  
    Derived d;  
    return 0;  
}
```

Вызывается конструктор базового класса (КБК) по умолчанию и конструктор производного класса (КПК) по умолчанию

```
int main(){  
    Derived d(2);  
    return 0;  
}
```

Вызывается КБК по умолчанию (хотя КБК с одним параметром определен, он не наследуется производным классом) и КПК с одним параметром

```
int main(){  
    Derived d(1,6);  
    return 0;  
}
```

Вызываются КБК и КПК с двумя параметрами. В КПК с двумя параметрами стоит явный вызов соответствующего КБК. КПК может явно вызывать любой другой КБК.

```
int main(){  
    Derived d(2,3,4);  
    return 0;  
}
```

Вызывается КБК по умолчанию и КПК с тремя параметрами. В базовом классе соответствующий конструктор не определен.

Конструкторы и деструкторы

- Деструкторы не наследуются. Если в производном классе (ПК) деструктор не определен, то он формируется по умолчанию и вызывает деструкторы всех БК.

Порядок вызова деструкторов:

- Деструкторы БК вызываются из деструктора ПК автоматически.
- Для иерархии, состоящей из нескольких уровней, деструкторы *вызываются в порядке, строго обратном вызову конструкторов.*

Виртуальные методы

- Указателю на БК можно присвоить значение адреса объекта любого ПК.

```
class Base_Class {  
    void f();  
    ....  
};  
class Derived_Class : public Base_Class {  
    void f();  
    ...  
};  
.....  
Base_Class *bc;  
bc= new Derived_Class; // указатель ссылается на объект ПК.  
bc->f(); //вызывается метод Base_Class– механизм раннего связывания
```

Виртуальные методы

- Наряду с ранним связыванием, в C++ реализован механизм **позднего связывания**. Этот механизм реализован с помощью **виртуальных методов**.

```
class Base_Class {  
    virtual void f();  
    ....  
};  
class Derived_Class : public Base_Class {  
    void f();  
    ...  
};  
.....  
Base_Class *bc;  
bc= new Derived_Class; // указатель ссылается на объект ПК.  
bc->f(); //вызывается метод Derived_Class
```

Виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы.

Чистые виртуальные методы. Абстрактные классы

- В базовом классе виртуальный метод может не иметь реализации. Такой метод называется **чистым виртуальным**. Он будет определяться в производных классах.

```
class Base_Class {  
    virtual void f()=0; //чистый виртуальный метод  
    ....  
};  
  
class Derived_Class : public Base_Class {  
    void f(){  
        .....  
    }  
    ...  
};
```

Класс, имеющий хотя бы один чистый виртуальный метод называется **абстрактным**.

~~Base_Class bc;~~

Нельзя создать объект абстрактного класса!

Множественное наследование

Класс может быть наследником нескольких базовых классов. Такое наследование называется **множественным**.

```
class B1 {...};  
class B2 {...};  
class B3 {...};  
class D : public B1, public B2, public B3{...};
```

Класс D наследует элементы трех базовых классов.

Для доступа к членам порожденного класса, унаследованного от нескольких базовых классов, используются те же правила, что и при порождении из одного базового класса.

Проблемы множественного наследование

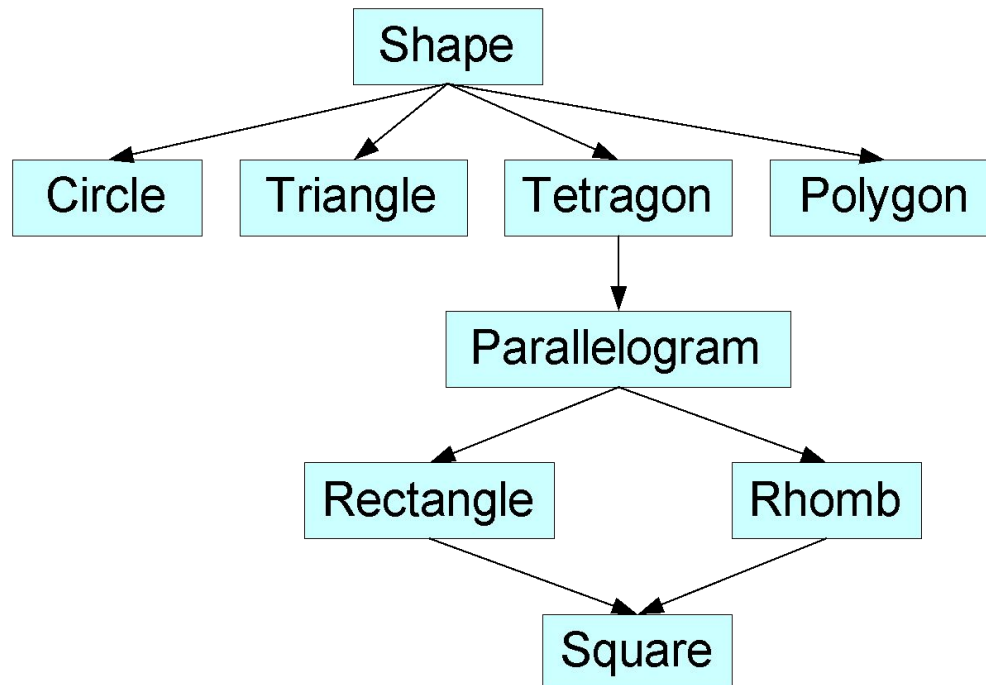
Проблемы могут возникнуть, если в порожденном классе используется член с таким же именем, как в одном из базовых классов, и/или в нескольких базовых классах определены члены с одинаковыми именами.

В этих случаях необходимо использовать оператор разрешения контекста для уточнения класса, к которому принадлежит нужный элемент.

```
class B1 {int a;...};
class B2 {int a;...};
class B3 {int a;...};
class D : public B1, public B2, public B3{
    int a;
    ...
    void f(){
        a=B1::a+B2::a+B3::a;
        ...
    }
};
```

Задание 3.

- Реализовать иерархию классов геометрических объектов



Задание 3.

Класс Shape должен содержать такие свойства и методы:

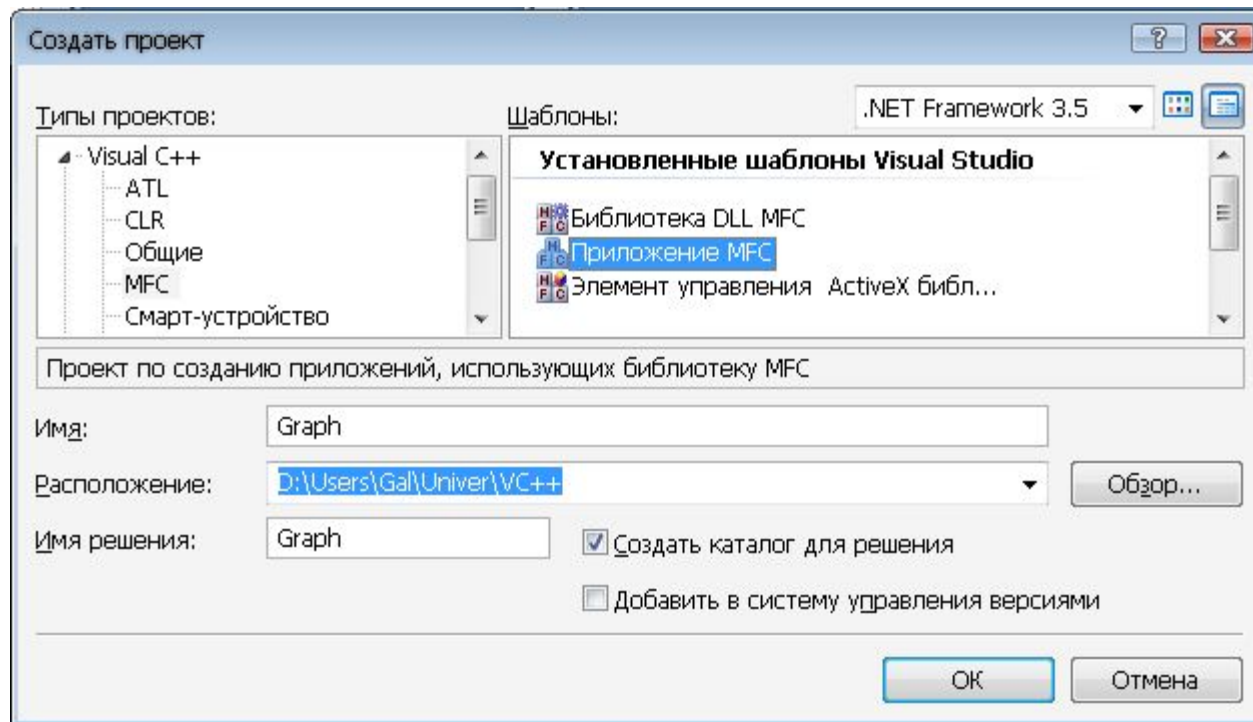
- Периметр и площадь фигуры;
- Параллельный перенос фигуры;
- Поворот фигуры;
- Печать информации о фигуре;
- Определение класса фигуры;

Методы в классе Shape виртуальные. Они должны определяться в конкретных классах.

Создание пользовательских интерфейсов средствами **MFC**

- Пакет **Microsoft Foundation Classes** (MFC) — библиотека на языке C++, разработанная Microsoft и призванная облегчить разработку GUI-приложений (**Graphical User Interface**) для Microsoft Windows путем использования богатого набора библиотечных классов.

Создание проекта. Шаг 1



Создание проекта

В простейшем случае программа, написанная с помощью библиотеки MFC, содержит два класса, порождаемые от классов иерархии библиотеки: класс, предназначенный для создания приложения, и класс, предназначенный для создания окна.

```
class CGraphApp : public CWinApp
{
...
};
```

```
class CGraphDlg : public CDialogEx
{
};
```

Создание проекта. Шаг 2

Мастер приложений MFC - Graph

Тип приложения

Обзор

Тип приложения

Поддержка составных документов

Строки шаблона документов

Поддержка базы данных

Свойства интерфейса пользователя

Дополнительные параметры

Созданные классы

Тип приложения:

☐ Один документ

☐ Несколько документов

☒ На базе диалоговых окон

☐ Диалоговое окно HTML

☐ Несколько документов верхнего уровня

☒ Поддержка архитектуры Document/View

Стиль проекта:

☐ Проводник Windows

☒ Стандарт MFC

Использование MFC:

☐ Использовать MFC в общей DLL

☒ Использовать MFC в статической библиотеке

Связь с MFC DLL.

Язык ресурсов:

Русский

☒ Использовать библиотеки с поддержкой Юникода

< Назад

Далее >

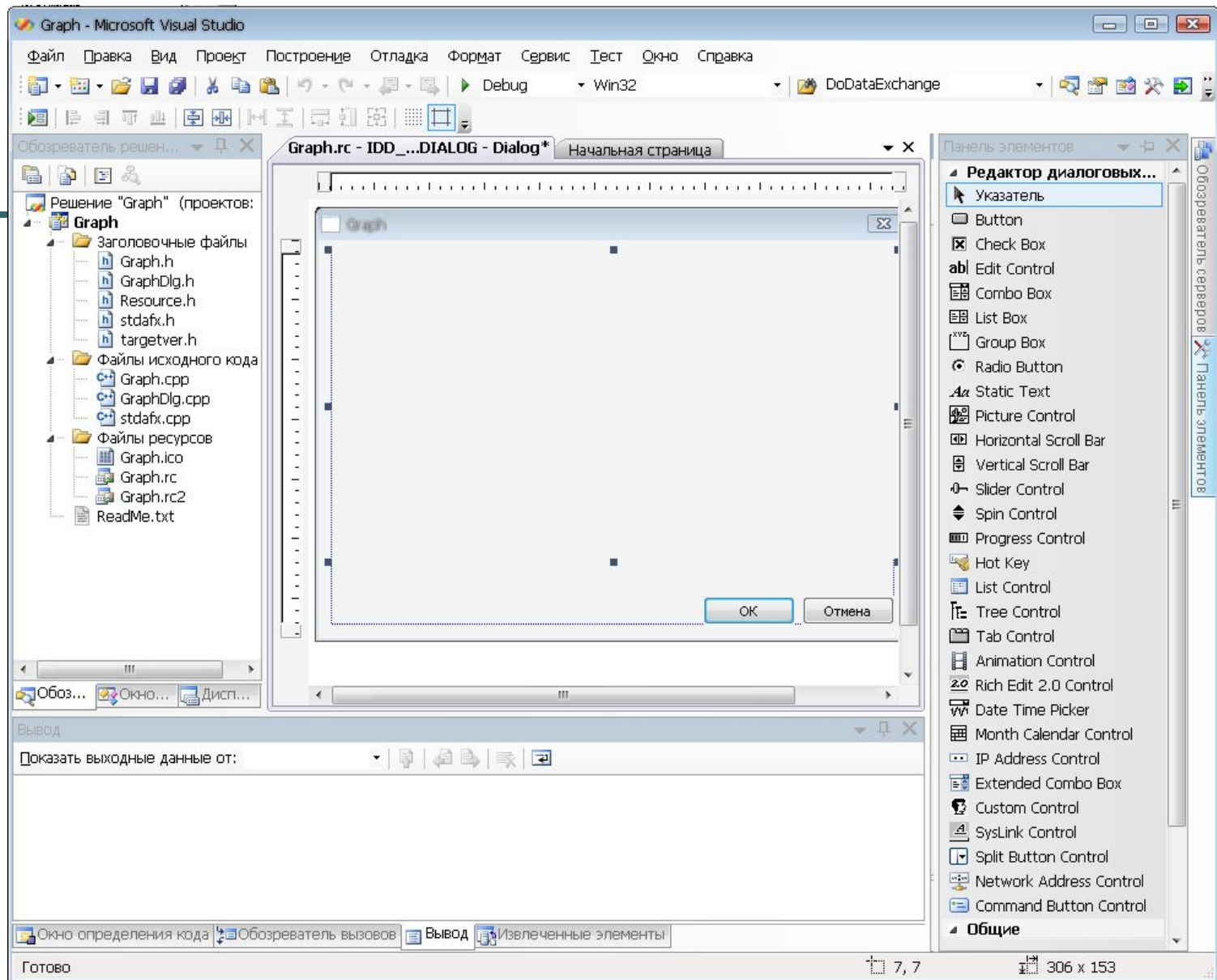
Готово

Отмена

Создание проекта. Шаг 3

- Помещаем на диалог элемент, в котором будет рисоваться график (н-р Static Text)
- В окне свойств задаем ему уникальный ID **IDC_GRAPH**
- Добавляем в класс **IDC_GRAPH** переменную типа CStatic **m_DrawArea**;
- Связываем переменную **m_DrawArea** и элемент **IDC_GRAPH**:

DDX_Control(pDX, IDC_GRAPH, m_DrawArea);
в методе DoDataExchange



Создание проекта. Шаг 4

Мастер добавления переменной-члена - Graph

Добро пожаловать в мастер добавления переменной-члена

Доступ: public

Тип переменной: CStatic

Имя переменной: m_DrawArea

☒ Переменная элемента управления

Идентификатор элемента управления: IDC_Graph

Тип элемента управления: LTEXT

Минимальное значение:

Максимальное количество знаков:

Максимальное значение:

Файл .h:

Файл .cpp:

Комментарий (нотация // не требуется):

Готово Отмена

Создание проекта. Шаг 5

- Добавляем на диалоговое окно кнопку, при нажатии на которую будет происходить отрисовка графика
- Двойным щелчком по кнопке создаем соответствующий метод

Контекст устройств

Графический ввод-вывод в Windows унифицирован для работы с различными физическими устройствами. Для этого предусмотрен специальный объект, называемый контекстом устройства (Device context). Рисование на некотором абстрактном DC. Если DC связать с окном на экране, то рисование будет происходить в окне; если связать его с принтером – то на принтере; если с файлом – то, соответственно, в файл.

Класс **CClientDC** – разновидность контекстов устройств; позволяет выводить графику в рабочей области окна.

Для рисования в некоторой функции (н-р, обработчике события нажатия кнопки), нужно получить контекст устройства. Это делается так: `CClientDC dc(this);`

Отрисовка графика

- ```
void CGraphDlg::OnBnClickedDraw()
{
 // TODO: добавьте свой код обработчика уведомлений
 //Создаем контекст, в котором будем рисовать
 CClientDC dc(&m_DrawArea);

 //Узнаем размеры прямоугольника
 CRect rc; //Графический объект
 m_DrawArea.GetClientRect(&rc);
 int w = rc.Width();
 int h = rc.Height();

 int x_start = 10;
 int y_start = h-10;
```

*//Отрисовка ...*

*CPen pnPenBlack(PS\_SOLID,1,RGB(0,0,0)); //Графический  
//объект. Устанавливаем гр. объект в контекст устройства*

```
CPen * pOldPen = dc.SelectObject(&pnPenBlack);
dc.FillSolidRect(rc,RGB(255,255,255));
dc.MoveTo(x_start - 5,y_start);
dc.LineTo(x_start + w-15, y_start);
dc.MoveTo(x_start,y_start+5);
dc.LineTo(x_start, y_start-h+15);
CPen pnPenRed(PS_SOLID,1,RGB(255,0,0));
dc.SelectObject(&pnPenRed);
dc.MoveTo(x_start, y_start);
for(int i = 3; i < w-x_start-2; i+=3)
{
 dc.LineTo(x_start + i, y_start - int(h/3*(1 - sin((float)i))));
}
dc.SelectObject(pOldPen);
}
```