# **Виртуал функциялар**

## **Виртуал функциялардын жалпы түшүнүгү**

Классик программалоодо функцияга объектерди берилиш катары берүүдө, функцияны жазуу учурунда берилүүчү объект тиби дайындалат. Башкача айтканда объектин аракетин байандоо үчүн алар тибин билүү жана байандоо зарыл. ООП нун күчү програм аткарылган учурда объект өзү аныктап алгандай виртуал функцияны жазуу мүмкүнчүлүгү каралганда турат. Башкача сөз менен айтканда виртуал функциялар жардамы менен объект өзү өзүнүн аракетин аныктайт. Виртуал функцияларды колдонуу ыкмасы *полиморфизм*депаталат. Полиморфизм көп калыптар (формалар) ээ болууну билгизет.Програмдагы объект чынында бир класс эмес түрдүү класстар көптүгүн көрсөтүшү мүмкүн, бул учурда алар жалпы база класс менен мурастоо жолу менен байланышкан болуу керек.

Си++ эрежелерине ылайык база классынын көрсөткүчү ушул класстын объектисине шилтениши мүмкүн, ошондой эле база классынын башка каалагандай туунду классынын объектерине шилтенет. А,В жана С класстарынын жөнөкөй иерархын карап көрөлү. А-база класс болсун дейли, В-классы А дан туулган туунду класс, ал эми В дан С классы туулсун дейли. Бул байланыш төмөнкү сүрөттө келтирилген.

Програмда төмөнкүдөй жарыяланат:

A object\_A; // А тибиндеги объект жарыяланат  
B object\_B; // В тибиндеги объект жарыяланат  
C object\_C; // С тибиндеги объект жарыяланат

Ушул эреже бойунча А тибиндеги көрсөткүч бул үч объектин каалаганына шилтене алат. Башкача айтканда

A \*point\_to\_Object; // база классына көрсөткүч жарыяланат  
point\_to\_Object=&object\_C; //С объектинин дареги көрсөткүчкө менчиктелет  
point\_to\_Object=&object\_B; /\*көрсөткүчкө В объектинин дарегин менчиктейбиз\*/

Ал эми төмөнкү жазылыштар ката

В \*point\_to\_Object; // туунду класска көрсөткүч жарыяланды  
point\_to\_Object=&object\_А; /\*база объектиге көрсөткүчтү менчиктөөгө болбойт\*/

point\_to\_Object көрсөткүчү А\* тибине ээ болгону менен ал С(же В) объекттерине шилтене алат. C тибиндеги объект А объектисинин өзгөчө түрү, бирок А объектиси С объектисине өзгөчө түр болуп саналбайт. Бул принцип мурастоо менен байланышкан класстарда виртуал функциялар аныкталганда өзгөчө маанилүү болуп калат. Виртуал функциялар так эле ошондой түргө ээ жана кадимки функциядай эле программаланат. Тек аны жарыялоо *virtual*кызматчы сөзү менен аткарылат. Мисалы биздин А база класс *v\_function()* виртуал функциясын төмөнкүчө жарыялайт:

class AutoCorrect

{  
public:  
virtual void v\_function(void);//функция А классынын кандайдыр бир жүрүш-турушун байандап жазат \*/  
};

Виртуал функция башка функциялар сыйактуу параметр менен дагы жарыяланат жана маани кайтара алат. Класста каалаган сандагы виртуал функцияларды жарыяласа болот. Алар класстын жабык, ачык же корголгон бөлүктөрүнүн каалаганында жайгашышы мүмкүн.

А классынан туулган В классында кандайдыр бир башка жүрүш-турушту байандоо керек болсо, анда виртуал функцияны ошондой эле *v\_function()* аты менен байандаса болот.

class B: public AutoCorrect  
{  
public:  
virtual void v\_function(void);/\*алмаштырган функция В классынын жаңы жүрүш-турушун байандайт \*/  
};

База классындагы виртуал функция менен бирдей аталган В классында виртуал функция аныкталса,  мындай функция *орун басар* деп аталат. В классында *v\_function()* виртуал функциясы ошол эле аттуу А классындагы виртуал функцияны алмаштырат. Чынында бир кыйла татаал жана жөнөкөй эле аттардын дал келиши эмес. Бул жөнүндө кичине кийин айтылат. Азыр болсо эң маанилүү болгон жана В тибиндеги object\_В объектисин көрсөткөн А\* тибиндеги point\_to\_Object көрсөткүчкө кайрылалы. *point\_to\_Object* көрсөткөн объект үчүн v\_function() виртуал функциясын чакырган операторду карасак

A \*point\_to\_Object; // база класскакөрсөткүч жарыялайбыз  
point\_to\_Object=&object\_B; /\*көрсөткүчкө В объектисинин дарегин менчиктейбиз\*/  
point\_to\_Object->v\_function(); // функцияны чакырабыз

*point\_to\_Object* көрсөткүчү А же В тибиндеги объект дарегин сактай алат. Демек *point\_to\_Object->v\_function()*оператору аткарылган учурда анын объектисине шилтенген класс виртуал функциясы аркылуу чакырылат. Эгерде *point\_to\_Object А* тибиндеги объектке шилтенсе, анда А классына таандык функция чакырылат. Эгерде *point\_to\_Object В* тибиндеги объектке шилтенсе, анда ушул класска таандык функция чакырылат. Демек бир эле оператор объект даректелген класс функциясын чакырат. Мына ушул аракет програм аткарылган учурунда аныкталат.

Туунду класста база классынын виртуал функциясына караганда ошол эле аттуу жана ошондой эле параметрлер тобу менен, бирок кайтарылуучу мааниси башка типте болгон функцияны аныктоого болбойт. Бул учурда компилегич ката берет. Эгерде туунду классында база классынын виртуал функциясындай аныкталгандай ат менен жана кайтарылган маани тиби менен функция киргизсек, анда бул туунду класс функция виртуал болбой калат. Мында база класс көрсөткүчүнүн жардамы менен бул көрсөткүчтүн каалагандай маанисинде база классынын функциясына кайрылуу аткарылат. Мында такыр башка виртуал функция болуп калат. Ошентип виртуал функцияларды ордуна койууда типтердин, функция аттарынын жана кайтарылган маанилер тибинин база жана туунду класстарында толук дал келүүсү талап кылынат.

*Мисал.*

#include <iostream.h>  
using namespace std;  
  
class base {  
public: int i;  
base(int x){x=i;}  
virtual void func()  
      {cout<<”База классында ”<<i<<’\n’;}  
};  
  
class derived1:public base  
{public:  
      derived1(int x):base(x){}  
      void func()  
      {cout<<”derived1 классында ”;  
      cout<< i\*i<<’\n’;  
      }  
};  
  
  
class derived2:public base  
{  
      public:  
            derived2(int x):base(x){}  
            void func()  
            {cout<<”derived2 классында”;  
             cout<<i+i<<endl;  
            }  
}  
  
  
int main()  
      {  
            base \*p, ob(10);  
            derived1 ob2(10);  
            derived2 ob2(10);  
      p=&ob;  
      p->func();  
      p=&ob1;  
      p->func();  
      p=&ob2;  
      p->func();  
      return 0;  
}

Програмы аткарылганда

База классында 10  
derived1 классында 100  
derived2 классында 20

жыйынтыгы басылып чыгат.

                Эгерде туунду класста виртуал функция алмаштрылбаса, анда база класста аныкталган виртуал функция колдонот. Жогорку програмда *derived2* классы үчүн төмөнкүдөй

class derived2:public base  
{  
      public:  
      derived2(int x):base(x){}  
};

өзгөртүү киргизсек, анда програм төмөнкү жыйынтыкты берет:

База классында 10  
derived1 классында 100  
derived2 классында 10

Програм иштөө учурунда кокус окуялар чакырылуучу виртуал функциянын версиясына таасир бериши мүмкүн. Төмөнкү мисалда, кокус сандар генератору берген санга жараша, програм эки объектинин бирисин тандайт.

#include <iostream>  
#include <cstdlib>  
using namespace std;  
  
class base{  
    public:  
        int i;  
    base(int x){i=x;}  
    virtual void func()  
    {cout<<"Базалык класста:";  
        cout<<i<<"\n";  
    }  
};  
  
class derived1:public base{  
    public:  
        derived1(int x):base(x){}  
            void func()  
            {cout<<"Derived1 классында:";  
                cout<<i\*i<<"\n";  
            }  
        };  
  
class derived2:public base{  
    public:  
        derived2(int x):base(x){}  
            void func()  
            {cout<<"Derived2 классында:";  
                cout<<i+i<<"\n";  
            }  
        };  
  
 int main()  
    {  
       base \*p;  
       derived1 object1(5);  
       derived2 object2(5);  
       int i,j;  
       for(i=0;i<5;i++)  
       {j=rand();  
        if(j%2) p=&object1;//так болсо object1 колдонот  
              else  
               p=&object2;//жуп болсо object2 колдонот  
              p->func();//тиешелүү функцияны чакыруу  
                }  
        return 0;  
}

Виртуал функцияларды колдонууга реалдуу бир мисал карап көрөлү. Төмөнкү програмда ***area*** базалык классы түзүлөт. Ал класста фигуранын эки өлчөмдөрү сакталат. Ошол эле жерде ***getarea()*** виртуал функциясы жарыяланат. Функция туунду класста алмаштырылып койулганда туунду класста берилген фигуранын аянтын кайтарып берет. Бул учурда базалык класстын ичинде ***getarea()*** функциясы интерфейсти аныктайт, ал эми реалдуу ишке ашырылышы ***area*** базалык класстан мурас алган класстарда жүргүзүлөт. Мисалда үч жана төрт бурчтуктун аянттары эсептелет.

#include <iostream.h>  
{ double d1,d2;  
class area  
{ double dim1, dim2;  
  void setarea(double &d1, double &d2)  
      {dim1=d1;  
       dim2=d2;}  
  void getdim(double &d1, double &d2)  
      {d1=dim1;  
       d2=dim2;  
      }  
  virtual double getarea()  
      { return 0.0;}  
};  
  
class rectangle: public area {  
      public:  
      double getarea()  
            {double d1,d2;  
             getdim(d1,d2);  
             return d1\*d2;}  
 };  
  
class triangle:public area{  
      public:  
      double getarea()  
      { double d1,d2;  
        getdim(d1,d2);  
        return 0.5\*d1\*d2;  
};

int main()  
{ area \*p;  
  rectangle r;  
  triangle t;  
  r.setarea(3.0,4.0);  
  t.setarea(4.0,5.0);

  p=&r;  
  cout<<”4 бурчтук аянты=”<<p->getarea()<<endl;  
  p=&t;  
  cout<<”3 бурчтук аянты=”<<p->getarea()<<endl;  
  return 0;  
}

## **Абстрак класстар жана колдонуу мисалдары**

### **Виртуал функциялар жөнүндө кошумча маалыматтар**

Кээ бирде виртуал функция базалык класста жарыяланып, маанилүү эч кандай амалдарды аткарбайт. Анын ордуна функциянын ичинде базалык мүчө-функцияларды жана өзгөрмөлөрдү камтытып, туунду класста «жетишпеген» нерселер аныкталат. Базалык класстын виртуал функциясы маанилүү амалдары жок болсо, бул базалык класстын каалаган тунду классында мындай функция кайрадан аныкталыш керек. Бул жобону ишке ашыруу максатында Си++ тилинде таза виртуал функциялар колдонот.

Таза виртуал функциялар базалык класста аныкталбайт. Ал жерге мындай функциялардын прототиптери гана кошулат. Таза виртуал функциялар үчүн төмөнкү негизги форма колдонот:

***virtual тип функция\_ысымы(параметрлер\_тизмеси)=0;***

Бул жарыялоонун эң негизгиси – функцияны нөлгө барабар кылуу. Ал компилегичке базалык класста функциянын денеси жок экендигин билдирет. Эгерде функция таза виртуал функция катары берилсе, ал ар бир туунду класста алмаштырылыш керек экендигин билдирет. Эгерде бул шарт аткарылбаса, анда көмпилөө учурунда ката жөнүндө маалымат алабыз.

Жогорууда келтирилген програмды өзгөртүп, *getarea()* функциясын *area* базалык классында таза виртуал функция катары жарыялайбыз.

#include <iostream>  
using namespace std;  
  
class area{  
    double dim1, dim2;  
    public:  
        void setarea(double d1, double d2)  
    {dim1=d1;  
    dim2=d2;  
    }  
  
        void getdim(double &d1, double &d2)  
        {d1=dim1;  
        d2=dim2;  
        }  
  
        virtual double getarea()=0;//таза виртуал функция  
    };  
  
    class rectangle:public area{  
        public:  
            double getarea()  
        {double d1,d2;  
            getdim (d1,d2);  
            return d1\*d2;  
        }  
    };  
  
    class triangle:public area{  
        public:  
            double getarea()  
        {double d1,d2;  
            getdim (d1,d2);  
            return d1\*d2/2;  
        }  
    };  
  
    int main()  
    { area \*p;  
        rectangle r;  
        triangle t;  
        r.setarea(2.2,3.5);  
        t.setarea(4.2,2.5);  
        p=&r;  
        cout<<"4 бурчтук аянты="<<p->getarea()<<"\n";  
        p=&t;  
        cout<<"3 бурчтук аянты="<<p->getarea()<<"\n";  
        return 0;  
}

Төмөнкү програмда мурастоо учурунда виртуал функциянын жаратылышы кандайча сакталары көрсөтүлгөн.

#include <iostream>  
using namespace std;  
  
class basa{  
    public:  
    virtual void func()  
    {cout<<"Базалык класста аткарылышы\n";}  
};  
  
class tuundu1:public basa{  
    public:  
    void func()  
    {cout<<"Биринчи туунду класста аткарылышы\n";}  
};  
  
class tuundu2:public tuundu1{  
    public:  
    void func()  
    {cout<<"Экинчи туунду класста аткарылышы\n";}  
};  
   
int main()  
{  
    basa \*p;  
    basa object;  
    tuundu1 object1;  
    tuundu2 object2;  
    p=&object;  
    p->func();//базалык класста  
    p=&object1;  
    p->func();//биринчи туунду класста  
    p=&object2;  
    p->func();//экинчи туунду класста  
        return 0;  
}

Бул програмда *func()*виртуал функциясы алды менен *tuundu1* класста мурасталат дагы анын ичинде алмаштырылат. Андан кийин *tuundu2* классы *tuundu1* классынан мурас алат дагы ал класстын ичинде кайрадан алмаштырылат.

### **Полиморфизмди колдонуу**

Бир нече жолу белгилеп кеткендей, полиморфизм жалпы интерфейс аркылуу бир нече окшош жагдайларда колдонуучу процесс, б.а. «бир интерфейс, көп метод» философиясын ишке ашырат. Полиморфизм өтө маанилүү, себеби ал татаал тутумдарды жөнөкөйлөткөнгө жардам берет. Жакшы аныкталган интерфейс ар кандай, бирок мааниси бойунча байланышта турган амалдарды аткарууга жарайт.

Си++ тилинде жана ОБП кеңири колдонгон эки термин бар: ***эрте байланыштыруу*** жана ***кеч байланыштыруу***. Эрте байланыштыруу компилөө учурунда билинген окуяларга таандык, айрыкча компилөө учурунда функциялардын чакырылышын курууда. Эрте байланыштыруу функцияларга кадымки функциялар, кайрадан жүктөлүүчү, виртуал эмес мүчө-функциялар жана шериктеш функциялар кирет. Бул типтеги функцияларды көмпилөө учурунда алардын чакыруу даректери белгилүү болот. Эрте байланыштыруунун артыкчылыгы програмдын ылдам иштөөсүндө турат. Ал эми кемчилик катары ийкемлүүлүктүн жоголушун белгилеп кетсе болот.

Кеч байланыштыруу програм аткарылып жаткан учурдагы окуяларга таандык. Кеч байланыштыруу функцияны чакырууга програм аткарылганга чейин чакырылып жаткан функциянын дареги белгисиз болгон учурду караса болот. Си++ тилинде виртуал функция кеч байланыштыруу объектиси болот. Эгерде виртуал функцияга кайрылуу базалык класстын көрсөткүчү аркылуу жүргүзүлсө, анда иштөө процессинде програм кайсы типтеги объектиге шилтеме жүргүзүлүп жаткандыгын аныктап, кайсы версиясын аткарыш керектигин тандаш керек. Кеч байланыштыруунун артыкчылыгы програмдын иштөө учурунда ийкемдүүлүгүндө турат. Програм каалагандай кокус окуяларга таасирдениш керек. Кемчилик катары функцияны чакыруу үчүн көбүүрөк амалдарды жүргүзүүдө турат.

Ар кандай жагдайларга жараша колдонуучу эрте же кеч байланыштыруу формасынын бирин өзү тандап алыш керек.

Төмөндө «бир интерфейс, көп метод» принцибин көрсөткөн програм келтирилген. Анда бүтүн типтеги тизмени камтыган класс аныкталган. Тизменин интерфейси *sakta()*жана *tanda()* таза виртуал функциялар аркылуу аныкталат. Маанини тизмеге сактоо үчүн *sakta()* функциясы чакырылат, ал эми *tanda()* функциясы тизмеден маанини алуу үчүн колдонот. Базалык *tizme* классында бул амалдарды аткаруу үчүн кыстарылган эч кандай ыкма берилбейт. Анын ордуна ар бир туунду класста кандай типтеги тизме колдоноору айкын аныкталат. Програмда тизменин эки тиби ишке ашырылган: кезек жана стек. Мындай типтеги тизмелер менен иштөө ыкмалары ар кандай болгону менен, ар бирине жетүү үчүн бир интерфейс колдонот.

#include <iostream>  
#include <cstdlib>  
#include <cctype>  
using namespace std;  
  
class tizme{  
    public:  
    tizme \*bashi;//тизменин башына шилтеме  
    tizme \*ayagi;//тизменин айагына шилтеме  
    tizme \*kiyinki;//кийинки элементке шилтеме  
    int san;//сакталуучу сан  
    tizme(){bashi=ayagi=kiyinki=NULL;}  
    virtual void sakta(int i)=0;  
    virtual int tanda()=0;  
};  
  
//кезек тибиндеги тизмени түзүү  
class kezek:public tizme{  
    public:  
    void sakta(int i);  
    int tanda();  
};  
  
void kezek::sakta(int i)  
{  
    tizme \*maani;  
    maani=new kezek;  
    if(!maani)  
    {cout<<"Эс бөлүүдө ката\n";  
        exit(1);}  
        maani->san=i;  
        //элементти тизменин айагына кошуу  
        if(ayagi) ayagi->kiyinki=maani;  
            ayagi=maani;  
        maani->kiyinki=NULL;  
        if(!bashi) bashi=maani;  
        }  
  
        int kezek::tanda()  
        {  
            int i;  
            tizme \*p;  
            if(!bashi){cout<<"Тизме бош\n";  
                return 0;}  
        //Элементти тизменин башынан жойу  
        i=bashi->san;  
                p=bashi;  
                bashi=bashi->kiyinki;  
                delete p;  
                return i;  
            }  
  
        //Стек тибиндеги тизмени түзүү  
            class stek:public tizme{  
                public:  
                void sakta(int i);  
                int tanda();  
            };  
  
            void stek::sakta(int i)  
            { tizme \*maani;  
                maani=new stek;  
                if(!maani)  
                {  
                    cout<<"Эс бөлүүдө ката\n";  
                    exit(1);  
                }  
                maani->san=i;  
        //Элементти тизменин башына кошуу  
                if(bashi) maani->kiyinki=bashi;  
                    bashi=maani;  
                if(!ayagi) ayagi=bashi;  
                }  
  
                int stek::tanda()  
                {  
                    int i;  
                    tizme \*p;  
                    if(!bashi)  
                    {cout<<"Тизме бош\n";  
                        return 0;  
                    }  
        //Элементти тизменин башынан жойу  
                    i=bashi->san;  
                    p=bashi;  
                    bashi=bashi->kiyinki;  
                    delete p;  
                    return i;  
                }  
  
                int main()  
                {tizme \*p;  
            //Кезек менен иштөө  
                    kezek object\_k;  
                    p=&object\_k;  
                    p->sakta(1);  
                    p->sakta(2);  
                    p->sakta(3);  
                    cout<<"Кезек :";  
                    cout<<p->tanda();  
                    cout<<p->tanda();  
                    cout<<p->tanda()<<"\n";  
            //Стек менен иштөө  
                    stek object\_s;  
                    p=&object\_s;  
                    p->sakta(1);  
                    p->sakta(2);  
                    p->sakta(3);  
                    cout<<"Стек :";  
                    cout<<p->tanda();  
                    cout<<p->tanda();  
                    cout<<p->tanda()<<"\n";  
                    return 0;  
                }

Бул програм төмөнкү жыйынтыкты берет:

Кезек :123  
Стек :321

Бул програмда *main()* функциясы класстардын иштөөсүн көрсөттү. Динамикалык полиморфизмди изилдөө үчүн бул програмда төмөнкү *main()* функциясын колдонолу.

  int main()  
          {tizme \*p;  
           kezek object\_k;  
           stek object\_s;  
           char belgi;  
           int i;  
            for(i=0;i<5;i++)  
             {cout<<"Кезек же стек (К/С)?:\n";  
              cin>>belgi;  
              belgi=tolower(belgi);  
              if(belgi=='k') p=&object\_k;  
                   else  
                       p=&object\_s;  
                    p-> sakta(i);  
                     }  
cout<<"Ишти айактоо учун Q белгисин киргиз\n";  
        for(;;)  
           {cout<<"Кезек же стектен алалыбы (К/С)?:\n";  
            cin>>belgi;  
            belgi=tolower(belgi);  
            if(belgi=='q') break;  
            if(belgi=='k') p=&object\_k;  
                     else  
                         p=&object\_s;  
                        cout<<p->tanda()<<"\n";}  
                         
       return 0;  
        }

Мында *main()* функциясы програмда виртуал функцияларды жана динамикалык полиморфизмди колдонууда кокус окуялар кандайча иштетилерин көрсөтүп жатат. Програмда цикл ичинде тизменин стек же кезек тибин тандоого сунуш берилет. Жоопко жараша базалык ***р*** көрсөткүчү тандалган объектиге орнотулат жана ал маани сакталат. Цикл айактагандан кийин башка цикл башталып элементти кайсы тизмеден алуу керектигин сурайт. Жоопко жараша элемент кезек же стектен алынат.

Таза виртуал функцияга мисал катары

class AutoCorrect  
{ public:  
virtual void v\_function(void)=0;//таза виртуал функция   
};

Мында болгон айырма көрсөтмөдө «=0» пайда болгону жана ал «таза спецификатор» деп аталат. Таза виртуал функция эч нерсе аткарбайт жана аны чакыруу мүмкүн эмес. Ал туунду класстарда функцияны алмаштыруу үчүн гана кызмат кылат. Жок дегенде бир таза виртуал функцияны камтыган класс *абстрак класс* деп аталат. Абстрак деп аталганынын себеби мындай класстан өз алдынча объект түзүү мүмкүн эмес. Бул аракет болгону башка класстар үчүн даярдык.

Мисалы, бардык айбандар өздөрүнүн жүрүш –турушунда «тамак жеген», «суу ичкен», «уктаган» жана «добуш чыгарган» функцияларды аткарат. Мына ушул бардык функцияларды камтыган база классын аныктоо максатка ылайык. Бул класстан андан ары конкреттүү айбандарды байандаган класстарды чыгарабыз. Мындагы база класс чынында абстрактуу болот. Чынында анда конкреттүү айбан байандалбайт. Бул балык дагы, куш дагы болушу мүмкүн. Бардык класс сыйактуу абстрак класс айкын аныкталган конструкторго ээ. Конструктордон класстын методдорун чакырса болот. Бирок конструктордон таза виртуал функцияларга кайрылуу програм аткарылып жатканда катага алып келиши мүмкүн. Кадимки класстарга караганда абстрак класстар чектелген укуктарга ээ. Жогоруда айтылгандай абстрак класстын объектисин түзүүгө болбойт. Абстрак классты функция параметрлер тибин берүү үчүн же кайтаруулучу маанинин тиби катары колдонууга болбойт. Аны типтерди айкын которууда колдонууга болбойт. Бирок абстрак класска шилтөө жана көрсөтүүнү аныктоого болот.

### **Жөнөкөй програм**

Мисалды жөнөкөйлөтүү максатында ар айбанды байандоодо, анын ысмы жана чыгарган добуш тиби менен чектелебиз. Програмдын иштөө негизинде айбандын ысмы жана ал чыгарган добушу экранга чыгаруу болот.

#include <iostream.h>

//абстрак база классы

class Animal  
{  
public:  
char \*Title; //айбан ысмы  
Animal(char \*t) {Title=t;} //жөнөкөй конструктор  
virtual void speak(void)=0; //таза виртуал функция  
};  
  
//бакалар классы  
class Frog: public Animal  
{  
public:  
Frog(char \*Title): Animal(Title) { };  
virtual void speak(void) { cout<<Title<<" ква- ква "<<" деп сүйлөйт "<<endl; };  
};  
  
//иттер классы  
class Dog: public Animal  
{  
public:  
Dog(char \*Title): Animal(Title) { };  
virtual void speak(void) { cout<<Title<<" 'гав-гав' "<<" деп сүйлөйт"<<endl;};  
};  
  
//мышыктар классы  
class Cat: public Animal  
{  
public:  
Cat(char \*Title): Animal(Title) { };  
virtual void speak(void) { cout<<Title<<"мяу-мяу "<<"деп сүйлөйт'"<<endl;};  
};  
  
//арстандар классы  
class Lion: public Cat  
{  
public:  
Lion(char \*Title): Cat(Title) { };  
virtual void speak(void) { cout<<Title<<" ррр-ррр "<<"деп сүйлөйт"<<endl;};  
// virtual int speak(void) { cout<<Title<<" ррр-ррр "<<" деп сүйлөйт "<<endl;  
return 0;};  
// virtual void speak(int When) { cout<<Title<<" ооа-ооу "<<  
"деп сүйлөйт"<<endl; };  
};  
  
  
int main ()  
{  
// Animal база классына көрсөткүчтөр массивин жарыялабыз  
//жана аны объекти түзүү менен эстеп калалы  
Animal \*animals[4] = { new Dog("Балтек"),  
new Cat("Мурка"),  
new Frog("Кермит"),  
new Lion("Кинг")}; // айбандар тизмеси  
  
for(int k=0; k<4; k++) animals[k]->speak();  
return 0;  
}

База класс катары *Animal* абстрак классын түздүк. Ал айбан ысмын байандаган жалгыз *Title* берилиш-мүчөгө ээ. Айбанга ысым менчиктеген айкын аныкталган конструктору бар. Жалгыз *speak()* аттуу таза виртуал функциясы айбан чыгарган добушту байандайт. Бул класстан «арстан» классынан башка класстар жаралды. Арстан классы «мышык» классынан жаралды. Мындай аракет виртуал функциялардын кылдат мамилелерин көрсөтүү үчүн болуп жатат. Бардык туунду класстарда *speak()* өздүк алмаштыруучу  виртуал функция байандалды, ал экранга айбандар кандай добуш бере турганын басып чыгарат. Програм денесинде *Animal\** тибиндеги *animals[4]* массив көрсөткүчү жарыяланды. Ошону менен катар эле класстардын динамик объекттери түзүлдү жана массив көрсөткүчтөрү толтурулду.Ал эми *for()* циклында көрсөткүч бойунча *speak()* виртуал функциясы чакырылат. Програм иштешинде

Балтек «гав-гав» деп сүйлөйт  
Мурка «мяу-мяу» деп сүйлөйт  
Кермит «ква-ква» деп сүйлөйт  
Кинг «ррр-ррр» деп сүйлөйт

чыгарылат. Виртуал функциялар аракеттенди. Эми *Lion* (арстан) классынын байандалашына токтолсок. Анда жалгыз *speak()* виртуал функциясы ордунда үч функция камтылган. Алардын экөө түшүндүрмөгө (комментарийге) киргизилген. Эгерде биринчисин  комментарийге кошуп экинчи ачып койсок, анда туура эмес жыйынтыкка алып келип

Error: animals.cpp(42,25):Virtual function 'Lion::speak()' conflicts with base class 'Cat'

кабарын берет. Анда виртуал функция база классындагыдай эле аргументке ээ, бирок башка тип кайтарат. Мындай болбойт деген кабарын берет.

Эми үчүнчү функцияны комментарийден чыгарып, биринчи экөөнү комментарийге кошсок, компилегич :

«Мында алмаштырылуучу виртуал функция ошол эле типтүү кайтарылуучу маани, бирок параметрлердин башка тобу жарыяланган учуруна жолугабыз» кабарын чыгарат.

 Програм иштөө жыйынтыгы төмөнкүнү берет:

Балтек гав-гав деп сүйлөйт  
Мурка мяу-мяу деп сүйлөйт  
Кермит ква-ква деп сүйлөйт  
Кинг мяу-мяу деп сүйлөйт

Арстан башка добуш бергенине көбө болуудабыз. Мында компилегич функция аргументи тууралуу кабарлаган. Ага карабастан туура аныкталган виртуал функция болбогондуктан көрсөткүч бойунча А база классынын*speak()* виртуал функциясы чакырылат. Биздин учурда *Lion* классы үчүн *Cat* классы база класс болуп саналат, ошондуктан *Cat* классынын функциясы чакырылды.

## **Виртуал деструкторлор**

Виртуал деструкторлор качан керек? Алгач мисал катары кандайдыр бир класс түзөлү, анда сап маанини эстей алсын жана биз үчүн база класс болсун дейли. Ошентип

class Base  
{  
private:  
char \*sp1;  
public:  
Base(const char \*S) { sp1=strdup(S); } //конструктор  
~Base() { delete sp1; } //деструктор  
};

Мындагы конструктор *strdup* китепкана функциясына кайрылуу аркылуу сап үчүн эс ыйгарат жана *sp1* көрсөткүчүндө жаңы сап дарегин сактайт. Класс деструктору Base класс объектисинин көрүү чөйрөсүнөн чыкканда бул эсти бошотот. Андан кийин база классынан жаңы класс бөлүп чыгарабыз, ал төмөнкү түргө ээ

class Derived: public Base  
{  
private:  
char \*sp2;  
public:  
Derived(const char \*S1, const char \*S2): Base(S1)  
{ sp2=strdup(S2); } //бул конструктор  
~Derived() { delete sp2; } //ал эми бул болсо деструктор  
};

Бул класс *sp2* көрсөткүчү шилтенген экинчи сапты сактайт. Жаңы конструктор база класс конструкторун чакыруу менен база классына сапты берет, ошондой эле экинчи сап үчүн эс ыйгарат жана жаңы саптын дарегин *sp2* көрсөткүчүнө сактайт. Бул класс деструктору ушул эсти бошотот.

Програмда ушундай класс объектисин түзөбүз:

Derived MyStrings(“string 1”, “string 2”);

Ушул объект көрүү чөйрөсүнөн чыкканда, алгач *Derived* классынын деструктору, андан кийин Base классынын деструктору чакырылат. Муну менен бардык эс бошотулат.

Эми башка вариантын карап көрөлү. *Base* база классына көрсөткүч жарыялап, ага *Derived* классынын объектисинин дарегин менчиктедик дейли. Мындай аракет аткарууга болот. Програмда бул аракет

Base \*pBase; //базалык класска көрсөткүч  
pBase=new Derived(“string 1”, “string 2”);

түрүндө жазылат. Эгерде pBase көрсөткүчү көрсөткөн объект програмдан алып салынса эмне болот?

delete pBase;

Компилегич *pBase* көрсөткүчү *Base* классынын объектисине шилтенүүгө тийиш экендигин көрөт. Ошондуктан програмда база класс деструкторун гана чакырат жана ал жалгыз бир сапты алып салат, бирок башкасын эсте калтырат. *Derived* классынын деструктору чакырылган эмес да. Ошентип виртуал деструктор ушул учурда керек болот.

Бул абалды оңдоо үчүн бардык класстардагы деструкторлорду *virtual* кызматчы сөзү менен жарыялоо керек. Ошентип деструкторлор төмөнкүчө жарыяланат:

virtual ~Base() { delete sp1; } //деструктор  
virtual ~Derived() { delete sp2; } //деструктор

Бул учурда эмне аткарылат? Деструкторлор виртуал болуп жарыялангандыктан аларды чакыруу програм аткарылып жаткан учурда аткарылат. Башкача айтканда кайсы деструктор чакырылаарын объектер өзүлөрү аныктайт. Биздин pBase көрсөткүч чынында Derived класс объектисине шилтенгендиктен бул класстын деструктору база класс деструктору менен катар чакырылат. База класс деструктору туунду класс деструкторунан кийин автоматтык түрдө аткарылат.

Эми акырында айтарыбыз: айрым учурда класста таза виртуал деструкторду аныктоо ыңгайлуу болушу мүмкүн.

class Something //виртуал функциясыз абстракт классы  
{  
public:  
virtual ~Something()=0; //таза виртуал деструктору  
};

Бул класс таза виртуал функцияны (деструкторду) камтыгандыктан ал абстрактуу. Деструктор виртуалдуу болгондуктан кийин деструкторду чакыруу бойунча проблем болбойт. Бул деструктор

Something::~Something() {};

түрүндө аныкталат. Муну аткаргандыктын зарылдыгы деструктор алгач туунду класс үчүн чакырылып, андан кийин жогору жаткан мурастоо чынжырындагы класс деструкторлор аткарылып акырында база класс деструктору аткарылат. Мында компилегич ~*Something()* чакыруусун класс абстрактуу болсо дагы аткарат, ошондуктан функция денесин сөзсүз аныктоо керек. Эгер муну жасабаса, компилегич символ жок тууралуу катаны берет.

Акырында эгерде класс виртуал функция камтыса, анда ал үчүн виртуал деструктор түзүү керек.

## **КӨНҮГҮҮЛӨР**

1. Аянт виртуал функциясы бар абстрак базалык класс түзгүлө. Төрт бурчтук, айлана, тик бурчтуу үч бурчтук жана трапециянын аянттарын эсептеген туунду класстарды түзгүлө.

2.Норма виртуал функциясын камтыган абстрак классын түзгүлө. Комлекс сандар, 10 элементтен турган вектор, 2х2 матрица туунду класстарды түзгүлө. Норма функциясын төмөнкүчө аныктагыла: комплекс сандар үчүн - модулдун квадраты, вектор үчүн – модул боюнча элементтердин суммасынын квадраттык тамыры, матрица үчүн – модул боюнча эң чоң мааниси.

3.Каалагандай ***х*** үчүн ***у*** координатасын эсептеген ийри сызык абстракт классын түзгүлө. ***х*** параметрине жараша ***у*** маанисин тапкан түз сызык, эллипс жана гипербола туунду класстарды түзгүлө. Түз сызыктын теңдемеси: *y*=*ax*+*b* , эллипстики: *x*2/*a*2+*y*2/*b*2=1, гиперболаныкы: *x*2/*a*2-*y*2/*b*2=1

4.Прогрессиянын суммасы виртуал функциясын камтыган базалык класс түзгүлө. Арифметикалык прогрессия жана геометриялык прогрессия туунду класстарын түзгүлө. Ар бир класс double тибиндеги эки талаага ээ болот. Биринчи талаа прогрессиянын биринчи мүчөсү, экинчиси – турактуу айырма (арифметикалык үчүн) жана турактуу катыш (геометриялык үчүн). Прогрессиянын суммасын эсептеген функцияны аныктагыла (параметр прогрессиянын элементтеринин санын берет).

        Арифметикалык прогрессия *aj*=a0+*jd*, *j*=0,1,2,…  
Арифметикалык прогрессиянын суммасы: *sn*=(*n*+1)(*a*0+*an*)/2  
Геометриялык прогрессия: *aj=a0rj*, *j*=0,1,2,…  
Геометриялык прогрессиянын суммасы: *sn*=(*a*0-*anr*)/(1-*r*)

5. Тизме базалык классын түзгүлө. Тизменин негизинде стек жана кезекти ишке ашыргыла. Алар кыстаруу жана сууруп алуу виртуал функцияларын камтыш керек.

6. Фигура базалык жана тегерек, төрт бурчтук, трапеция туунду класстарын түзгүлө. Аянт, периметр жана жыйынтыкты басып чыгаруу виртуал функцияларды аныктагыла.

7.Кызматкер базалык жана толук ставкада, жарым ставкада жана саат акысында иштеген кызматкерлер туунду класстарын түзгүлө. Айлыкты эсептеген функцияны аныктагыла.

8. Беттин аянтын эсептеген виртуал функциясын камтыган абстрак база классын аныктагыла. Өзүнүн аянтын эсептеген параллелепипед, тетраэдр жана шар туунду класстарын түзгүлө. Абстрак классына шилтеген массивди түзүп, ага ар кайсы объектилердин даректерин менчиктегиле.

       Параллелепипеддин бетинин аянты: S=6*xy*, шардыгы: *S*=4π*r*2, тетраэдрдики: *S*=*a*2√3

9.Көлөм виртуал функциясын камтыган абстрак база классын түзгүлө. Көлөмдөрдү эсептеген параллелепипед, пирамида, тетраэдр жана шар туунду класстарын аныктагыла. Абстрак классына шилтеген массивди түзүп, ага ар кайсы объектилердин даректерин менчиктегиле.

      Параллелепипеддин көлөмү - *V*=*xyz ,* пирамиданыкы: *V*=*xyh,* тетраэдрдики: V=*a*32/12, шардыкы: *V*=4π*r*3/3.

10.Жаныбарлар абстрак классын түзгүлө. Адам жана айбан туунду класстарын аныктагыла. Айбан классында ит жана уй туунду класстарын аныктагыла. Адам, ит жана уйду байандаган виртуал функцияларды түзгүлө.

11.Ата базалык классын түзгүлө. Анын ысымы бар. Ысымды экранга басып чыгарган виртуал функцияны аныктагыла. Атасынын жана өзүнүн атын басып чыгарган Бала туунду классын түзгүлө. Бул класстан Небере туунду классын түзгүлө. Бул класс үчүн толук аты-жөнүн экранга чыгарган функцияны түзгүлө.

12.Теңдемененин тамырлары виртуал функциясын камтыган абстрак база класс түзгүлө. Сызыктуу жана квадраттык теңдемелер туунду класстарын аныктагыла. Теңдеменин тамырларын тапкан функцияны түзгүлө.