Lecture #3

Object-oriented programming

based on the C language++
2th semester

- Static elements of the class
- Inheritance

static class elements.

In C ++ is the ability to access all objects created in one particular class variable (field), the contents of which is stored in one place. To do this, declare the variable:

static type name;

This storage class (static) can be used not only to declare static fields (class variables), but also for class methods. The memory is reserved for this purpose at the start of the program up to an explicit object creation. Therefore, it is as if the same for all copies of the class fields. Access to this variable (: :) is possible only after the initialization:

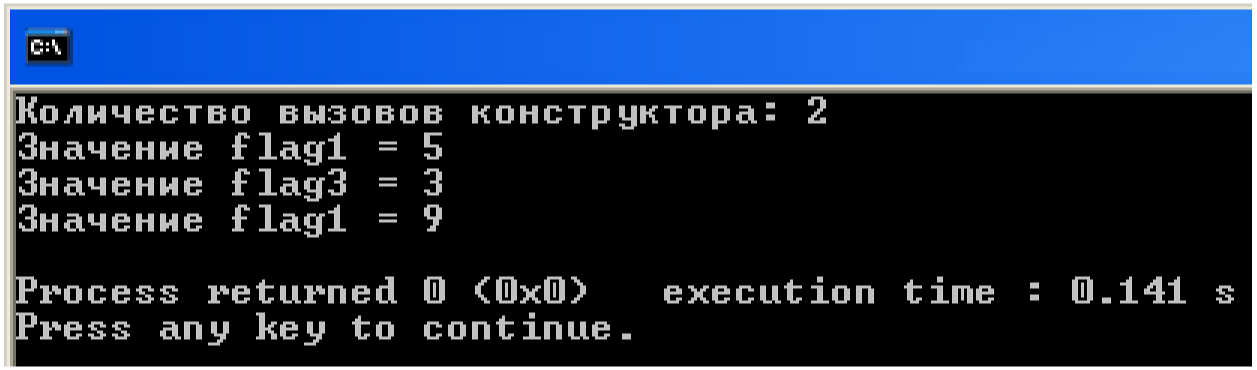
type-name variable-name :: = nach.znachenie;

You can not initialize static class within the class members in the body of the class methods. They are available in the initialization file scope. In this respect, static data members are similar to global variables.

Example 9. *Static class components.*

```
#include <iostream>
using namespace std;
class     Stat
{
private:
     static int flag2;
     static const int flag3;
public :
     static int flag1;
     Stat ()   { flag1 ++; }
     int Ret_flag1()  { return flag1; }
     static int Ret_flag2()  { return flag2; }
     static int Ret_flag3()  { return flag3; }
};
int Stat::flag1=0;
int Stat::flag2;
const int Stat::flag3=3;
```

```
int main ()
{
    Stat S1, S2;
    cout<<"Количество вызовов конструктора: "<<Stat::flag1<<endl;
    S1.flag1=4;
    Stat S3;
    cout<<"Значение flag1 = "<<S3.Ret_flag1()<<endl;
    cout<<"Значение flag3 = "<<S1.Ret_flag3()<<endl;
    S2.flag1=9;
    cout<<"Значение flag1 = "<<S3.Ret_flag1()<<endl;
    return 0;
}
```



```
Количество вызовов конструктора: 2
Значение flag1 = 5
Значение flag3 = 3
Значение flag1 = 9

Process returned 0 (0x0)   execution time : 0.141 s
Press any key to continue.
```

Static fields, you can set the initial values of one (and only one) time in the action file. Access to public static class members via any possible class object class name, or by using a binary operation permission scope.

Static class elements exist even when there are no objects of that class.

To provide access to private or protected class member must be provided for public static method to be called with the addition of his name to the class name and the binary operation permission scope.

Method class may be declared as static, if it does not have access to non-static class members.

Static method has no this pointer, because the static fields and static methods exist independently of any class of objects.

static class fields are created in the single copy regardless of the amount determined in the program objects. All objects (even dynamically created) share a single copy of the static fields.

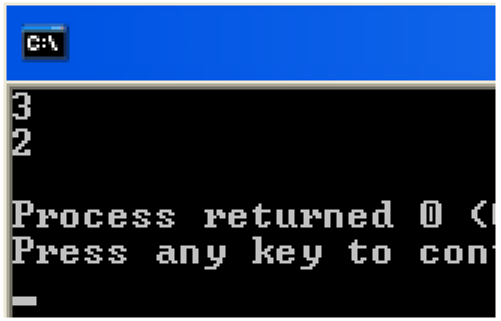
Count class facilities.

The classic application of static fields - count objects. To do this in a class declares a static field of a type that increases in the constructor and destructor decreases.

***Example 11.** Counting the object class.*

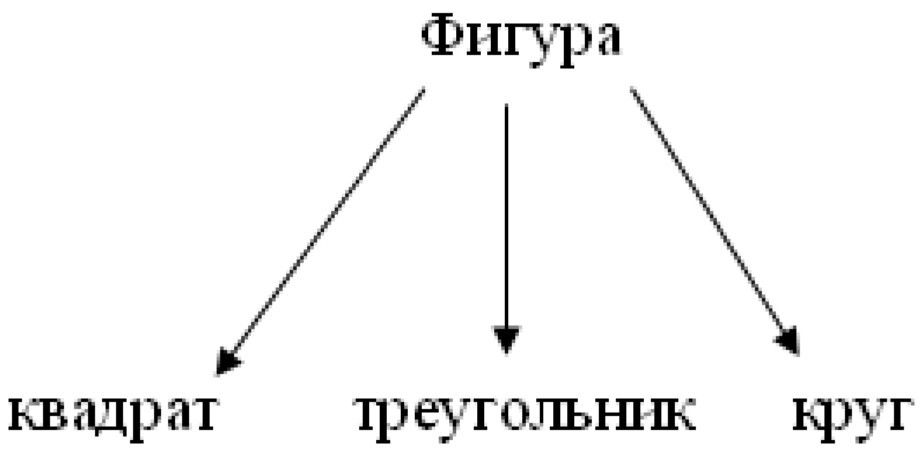
```
#include <iostream>
using namespace std;
class Object
{
        static unsigned int count;
    public:
        Object();
        ~Object();
        static unsigned int Count();
};

unsigned int Object::count = 0;
Object::Object() { ++count; }
Object::~Object() { --count; }
unsigned int Object::Count()
              { return count; }
```

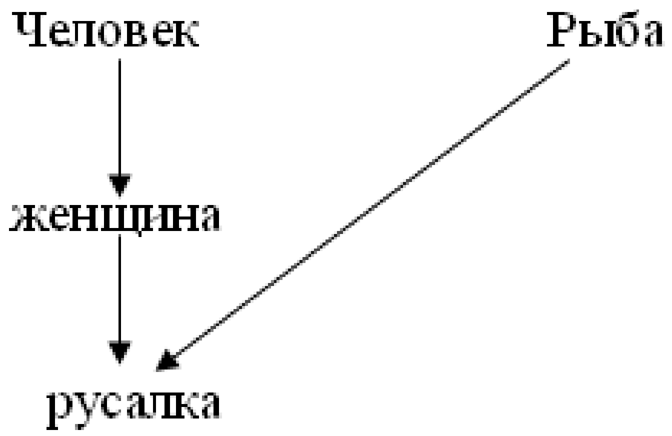
```
int main()
{
    Object a, b, c;
    int i;
    cout<<a.Count()<<endl;
    a.~Object();
    cout<<b.Count()<<endl;
    return 0;
}
```


```
3
2

Process returned 0 (
Press any key to con
_
```

notion inheritance.

Inheritance- *a way to re-use the software in which new classes are created from existing classes by borrowing their attributes and functions and enrichment of these opportunities of new classes. Reusing code saves time when developing programs.*





Each derived class object is also an object corresponding to the base class. However, the converse is not true: the base class object is not an object class generated by this base class.

Types of inheritance: *simple and multiple.*

simple inheritance - each class has only one parent class the next level.

multiple inheritance - descendant class is created using a few basic classes-parents.

Inheritance

Inheritance is a powerful tool PLO and is used for the following inzaimosvyazannyh purposes:

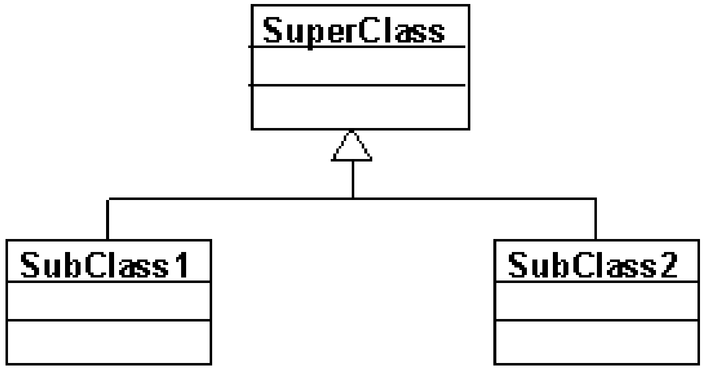
- exclusion from the program repeating code fragments;
- simplify modification program;
- facilitate the creation of new programs on the basis of the existing ones.
- In addition, the inheritance is the only opportunity to use the facilities, source code is not available, but you want to make changes.

inheritance syntax

access keys

class name: [Private | protected | public] bazovyy_klass

{Class} body;



```
class A {...};
class B {...};
class C {...};
class D: A, Protected B, public C
{...};
```

- The successor can be describe new fields and methods and overrideexisting methods. Override methods in several ways.
- If any method in the child to work in a completely different way than in the ancestor, the method described in the child again. However, he may have a different set of arguments.
- If required add adding ancestor method, a method corresponding child along with a description of additional operations performed ancestor method call via access operations scope.
- If the program is planned to run simultaneously with different types of object hierarchies or plan to add new objects in the hierarchy, the method is declared as virtual with the keyword virtual. All virtual methods of the hierarchy with the same name must have the same list of arguments.

heirship

access key	Specifier in the base class	Access to the derived class
private	private	not
	protected	private
	public	private
protected	private	not
	protected	protected
	public	protected
public	private	not
	protected	protected
	public	public

In other words:

- privatethe basic elements of the class in a derived class is not available, regardless of the key. Appeal to them can only be done through the methods of the base class.
- elements protected inheritance with the key private are in a derived class privateIn other cases the right of access to it are not changed.
- Access to the elements public inheritance becomes relevant access key.

If a base class is inherited from the key private, you can selectively make certain elements available in the derived class:

```
    class Base {
     ...
     public: void f ();
    };
    class Derived: private Base {
     ...
     public: Base :: void f ();
  };
```

simple inheritance

```
class daemon: public monstr{
 int brain;
 public:
 // ------------- constructors:
 daemon (int br = 10) {brain = br};
 daemon (color sk) : monstr (sk) {Brain = 10;}
 daemon (char * nam) : monstr (nam) {brain = 10;}
 daemon (daemon & M) :monstr (M) {brain =M.brain;}
```

If the base class constructor requires a parameter, it must be explicitly invoked in the constructor of the derived class in the initialization list

The order of constructor calls

constructors not inherited, So the derived class must have its own designers. The order of constructor calls:

- If the constructor descendant of an explicit call to the constructor is absent parent, *automatically called const-ruktor default ancestor*.
- For hierarchy consisting of several levels of constructors are called ancestors starting from the top level. This is followed by the designers of the class members, which are the objects in the order they are declared in the class, and then marks the class constructor.
- In the case of several of their ancestors constructors are called in the order of declaration.

assignment operation

```
const daemon & operator = (daemon & M) {
 if (& M == this) return * this;
 brain = M.brain;
 monstr:: operator = (M);
 return *this; }
```

Fields inherited from class monstr, Available functions of the derived class, as defined in the base class as a private.

A derived class can not only complement but also to adjust the behavior of the base class. only virtual methods recommended override in a derived class

inheritance destructors

Destructors are not inherited. If a destructor in a derived class is not described, it is formedautomatically and calls destructors of base classes.

The destructor of the derived class does not need to explicitly call the base class destructor, it will be done automatically.

For hierarchy consisting of several layers, destructors are invoked in a manner strictly callback constructors: first calls the destructor, then - the elements of the class destructor, then the base class destructor.

1. The concept of a virtual function.

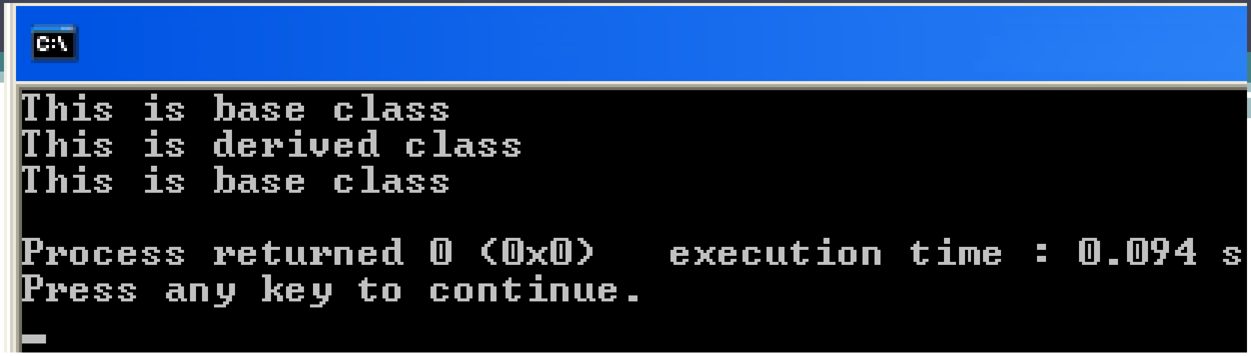
Virtual function (virtual function) *represents function base class that is overridden in a derived class.*

virtual type functionname (parameters);

Example 19. *Virtual functions.*

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

class base
{
    public: void print ()
     { cout<<"This is base class\n";}
};

class derived: public base
{
    public : void print ( )
     { cout<<"This is derived class\n";}
};
```

```
int main ()
{
    base B, *bp=&B;
    derived D, *dp=&D;
    base *p=&D;
    bp ->print();
    dp -> print();
    p -> print();
    return 0;
}
```



```
This is base class
This is derived class
This is base class

Process returned 0 (0x0)   execution time : 0.094 s
Press any key to continue.
```

If you do not use virtual functions, then when you create a derived class object and call for him any overriding function occurs a function call is generated by the class, not the base. If the overriding function call to use a pointer or reference to an object of the base class, there will be a function call is the base class.

Select the desired function is performed at the stage of program compilation and defined pointer type, rather than its value.

Overridden function is called in accordance with the class pointer. it<u>early or static binding</u>.

It gives the most flexibility method <u>late or dynamic binding</u> (Via the virtual function) implemented at step program execution.

When declaring a virtual function in the base class before its name indicates keyword ***virtual****. In the derived class virtual function is overridden. Each such overriding (overriding) the virtual function in the derived class is the creation of a specific method. When you override a virtual function in a derived class virtual keyword is not specified.*

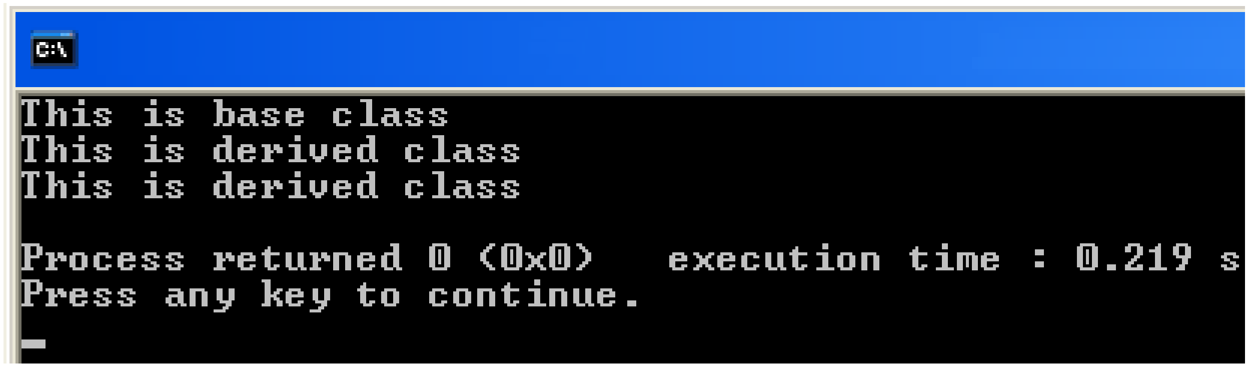
Virtual function can be called like any other component function. However, to support dynamic polymorphism virtual function call through base class pointer, used as a reference to an object of a derived class.

If the derived class object contains a virtual function and virtual function is called with this pointer, then when the compilation is called the function which corresponds to the type of object that is referenced by a pointer.

If there are multiple classes derived from base class contains a virtual function, then the base class link pointer on various objects of these derived classes are executed various versions of virtual functions.

Redefines the virtual function must have the same type, the number of parameters and return type. Virtual function must be a component class.

If in the example determine print () method as a virtual, that is, add one word virtual, the work program will be the result



```
This is base class
This is derived class
This is derived class

Process returned 0 (0x0)   execution time : 0.219 s
Press any key to continue.
_
```

polymorphic class *- a class in which virtual functions are defined (at least one).*

Virtual keyword can only write in the base class in the function declaration, the overridden function will be considered virtual.

Rules of the description and use of virtual functions, methods:

1. *A virtual function can only be a class method.*

2. *Any method can be overloaded class virtual make such an assignment operation or conversion operation type.*

3. *Virtual function inherited.*

4. *A virtual function can be constant.*

5. If the base class virtual function is defined, the derived class method with the same name and the prototype (including the return type, and the constancy of the method) is automatically and replaces the virtual method in the base class.

6. Static methods can not be virtual.

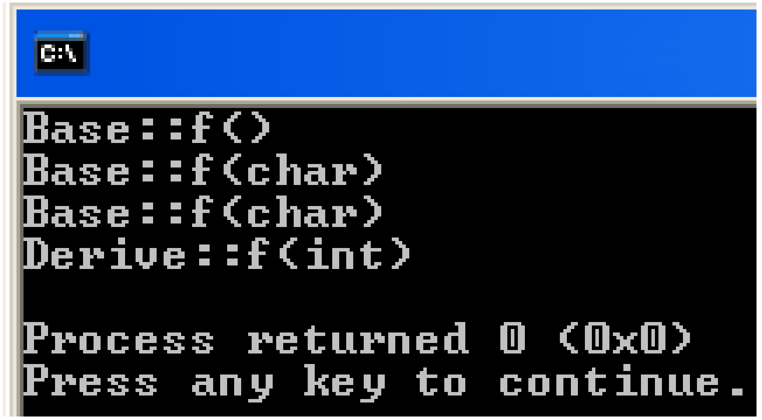
7. Constructors can not be virtual.

8. Destructors can (usually - must) be virtual - it ensures the correct deallocation through base class pointer.

9. Const method is considered different from a non-const method with the same prototype.

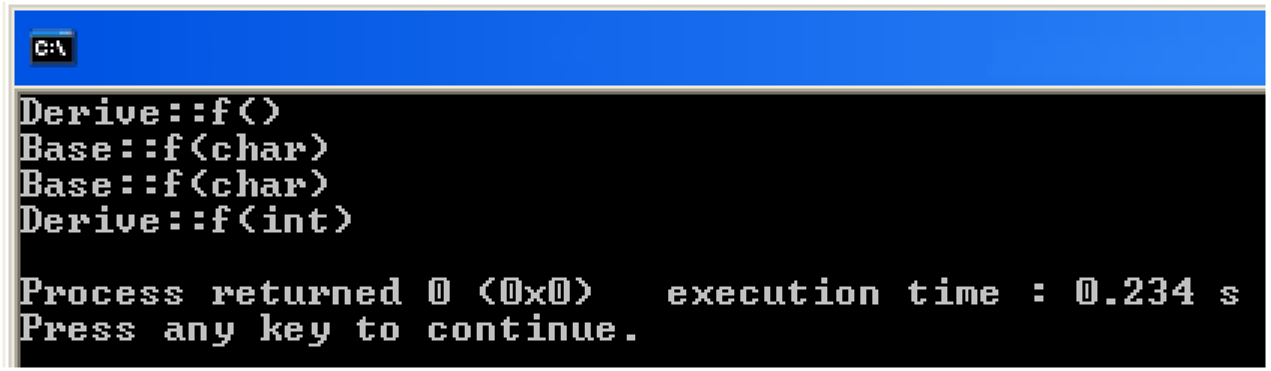
Example. *Virtual functions.*

```
#include <iostream>
using namespace std;
class Base
{
public:
    virtual int f() const
    {
        cout<<"Base::f()"<<endl;
        return 0;
    }
    virtual void f(const char &s) const
    {
        cout<<"Base::f(char)"<<endl;
    }
};

class Derive: public Base
{
public:
    virtual int f(int) const
    {
        cout<<"Derive::f(int)"<<endl;
        return 0;
    }
//  virtual int f() const
//  {
//      cout<<"Derive::f()"<<endl;
//      return 0;
//  }
};

int main()
{
    Base b, *pb;
    Derive d, *pd = &d;
    pb = &d;
    pb->f();        pb->f('n');
    pb->f(10);      pd->f(10);
    return 0;
}
```



```
Base::f()
Base::f(char)
Base::f(char)
Derive::f(int)

Process returned 0 (0x0)
Press any key to continue.
```

If you uncomment the commented-out part of the code that will be on the console as a result of the program?



```
Derive::f()
Base::f(char)
Base::f(char)
Derive::f(int)

Process returned 0 (0x0)   execution time : 0.234 s
Press any key to continue.
```

Through base class pointer can not cause new virtual methods that are defined only in a derived class.

<u>pointer type an explicit cast:</u>

((Derive *) pb) -> f (2);

Virtual function can be called non-virtual, if you specify a qualifier class:

```
Base *cl = new Derive();
cl->print();          // вызов метода-наследника
cl->Base::print();    // явно вызывается базовый метод
```

Such a static call is needed when in the base class implements common actions that should be performed in all derived classes.

When at least one virtual function of class size borderless is 4 - is the size of the pointer. The number of virtual functions is not important - the class size is equal to 4.

```
cout<<sizeof(Base)<<endl;
cout<<sizeof(Derive)<<endl;
```

3. Early and late binding.

static polymorphism *implemented through an overload of functions and operations.*

dynamic polymorphism *- through the use of virtual functions.*

Static and dynamic versions of the polymorphism corresponds to the notion of early and late binding.

early binding *As for the events Compile the program, such as call*

- <u>common functions</u>.

- <u>overloaded functions</u>. - <u>component non-virtual functions</u>.

- <u>friendly features</u>.

When you call these functions all the necessary address information is known at compile time.

The advantage of early binding is a high speed produced executables.

The disadvantage of early binding is to reduce the flexibility of the program.

Late binding*As for the events that occur during the execution of the program. When a function call using late binding address of the called function before the start of the program is unknown. In particular, an object of late binding are virtual functions.*

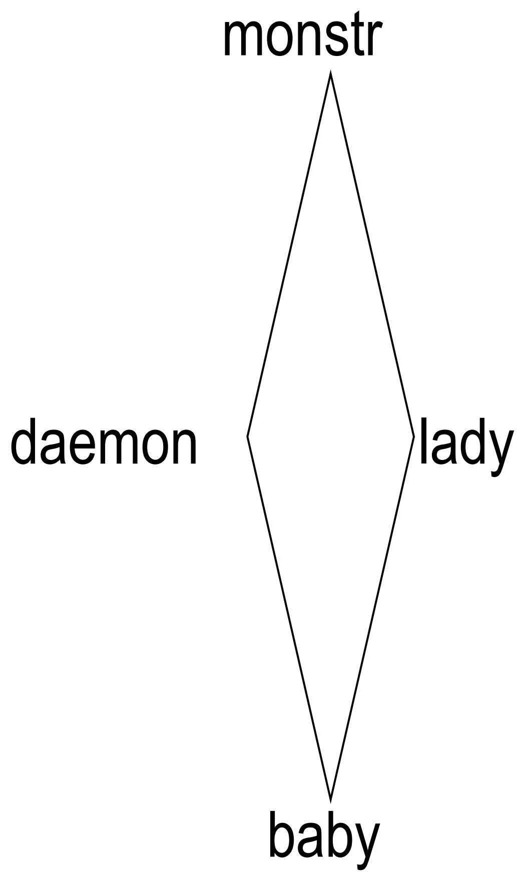
When accessing the virtual function pointer through the base class at runtime is determined by the type of the linked object and the selected version of the virtual function for the call.

The advantage of late binding is performed by the high flexibility of the program, the ability to respond to events.
A disadvantage is the relatively low speed of the program.

multiple inheritance

```
class monstr{
 public: int get_health(); ...
};
class hero {
 public: int get_health(); ...
};
class ostrich: public monstr, Public hero {... };

int main () {
 ostrich A;
 cout << A.monstr::get_health();
 cout << A.hero::get_health();
}
```

```
class monstr {
 ...
};
class daemon: virtual public monstr {
 ...
};
class lady: virtual public monstr {
 ...
};
class baby: public daemon, public lady {
 ...
};
```



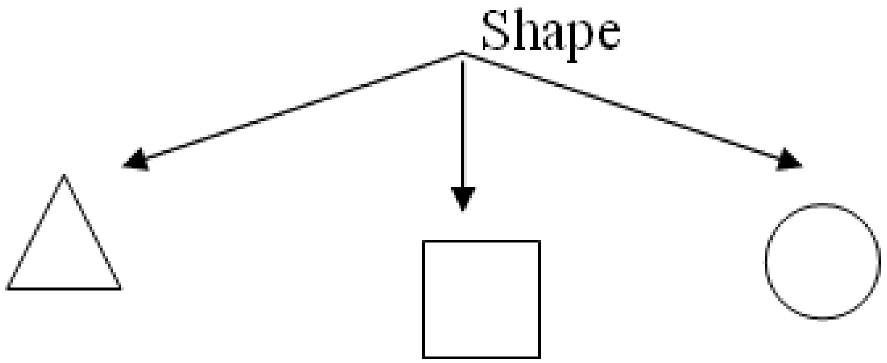
abstract classes.

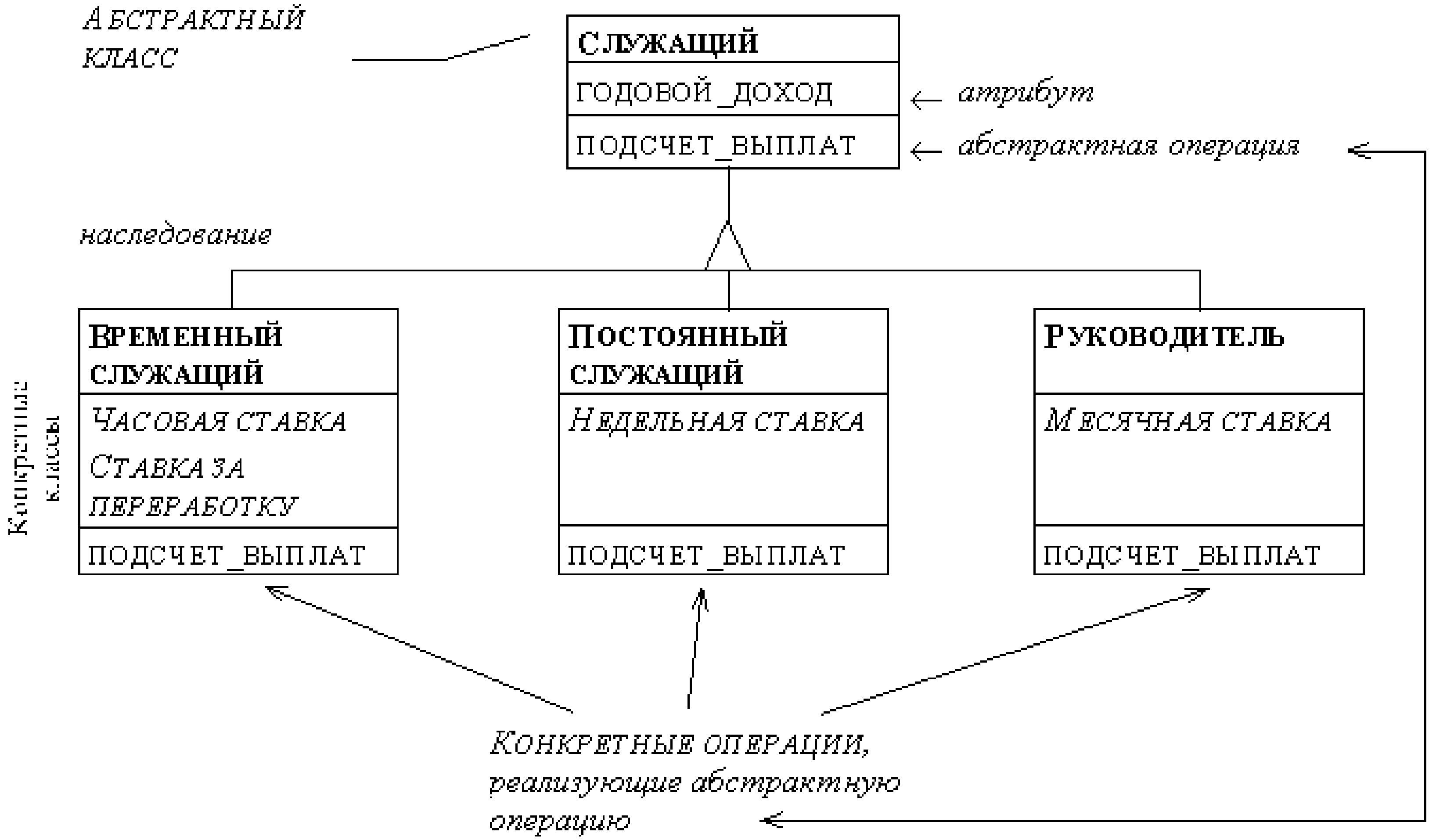
Plan:

1. *The concept of an abstract class.*
2. *Pure virtual functions and abstract classes.*
3. *Virtual destructors.*

notion abstract class.

***An abstract class**- a class that expresses a kind of overall concept, which reflects the basic idea for the use of derived classes. Abstract class is created only in order to create on its basis the other classes. You instantiate objects of such classes can not, so they are called abstract.*




Абстрактный класс
Служащий
годовой_доход
подсчет_выплат
атрибут
абстрактная операция
наследование
Временный служащий
Часовая ставка
Ставка за переработку
подсчет_выплат
Постоянный служащий
Недельная ставка
подсчет_выплат
Руководитель
Месячная ставка
подсчет_выплат
Конкретные классы
Конкретные операции, реализующие абстрактную операцию

To make an abstract class, you must declare one or more of its functions purely virtual.

virtual tip_vozvr_znacheniya functionname (parameters) = Ø;

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

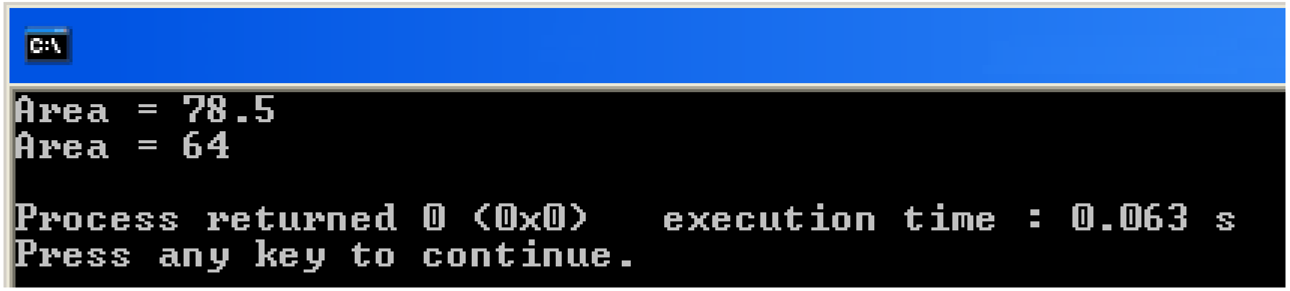
Classes with at least one pure virtual function call abstractAnd can not be used to create objects.

Example.

An abstract class.

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

class Shape
{
public:
    virtual double area () = 0 ;
};

class  Circle : public Shape
{
    double radius;
public:
    Circle (double q) {  radius = q; }
    double area ()  {  return 3.14*radius*radius;   }
};

class  Triangle : public Shape
{
    double height, length;
public:
    Triangle (double a, double b)  { height = a;  length = b; }
    double area ()  {  return height*length*0.5;   }
};
```

```
class  Rectangle : public Shape
{
    double height, length;
public:
    Rectangle (double a, double b)  {   height = a; length = b;   }
    double area ()    {  return height*length;    }
    void print_area (Shape*ptr)    { cout<<"Area = "<<ptr->area()<<endl; }
};

int main()
{
    Circle c(5);
    Triangle t(4, 32);
    Rectangle r(2,7);
    r.print_area (&c);
    r.print_area (&t);
    return 0;
}
```



```
Area = 78.5
Area = 64

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

2. Pure virtual functions and abstract classes.

Virtual function that has no definition of the body is called ***pure (pure)*** *and declared as follows:*

virtual type name (parameter) = 0;

It is expected that this feature will be implemented in derived classes.

The class in which there is at least one pure virtual function is called an abstract class. EInstancess abstract class to create prohibited.

When inheriting abstraction remains the same: if the derived class does not implement inherited pure virtual function, it is also abstract.

Net (rire) virtual functions are used in the case where the base class virtual function is not a significant effect. In each derived class from a given class of a function must always be overridden.

virtual destructors.

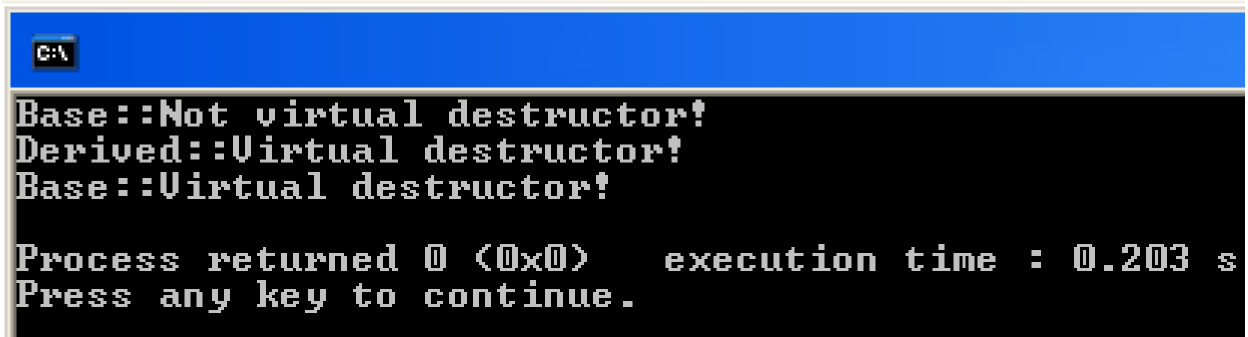
Class destructor can be declared virtual. When the base class destructor is virtual, the destructor of all the heirs and the same. Destructor virtual must declare if access to the dynamic object is executed by the derived class pointer of the base class.

In this case, when an object is destroyed via the pointer base class destructor is called a derived class, and he calls the destructor of the base class. If the base class destructor is not virtual, then the destruction of the derived class object through a pointer to the base class destructor is called only the base class.

Example. *Virtual destructors.*

```
#include <iostream>
using namespace std;
class Base
{
public:
    ~Base()
    {   cout<<"Base::Not virtual destructor!"<<endl; }
};

class Derived : public Base
{
public:
    ~Derived()
    {   cout<<"Derived::Not virtual destructor!"<<endl; }
};
```

```
class VBase
{
public:
    virtual ~VBase()
    {   cout<<"Base::Virtual destructor!"<<endl;    }
};

class VDerived : public VBase
{
public:
    ~VDerived()
    {   cout<<"Derived::Virtual destructor!"<< endl;
    }
};

int main()
{
    Base *bp = new Derived();
    delete bp;
    VBase *vbp = new VDerived();
    delete vbp;
    return 0;
}
```



```
Base::Not virtual destructor!
Derived::Virtual destructor!
Base::Virtual destructor!

Process returned 0 (0x0)   execution time : 0.203 s
Press any key to continue.
```

Destructor can be declared as a pure virtual:

virtual ~ VBase () = 0;

The class, which defines a pure virtual destructor, is abstract, and create objects of this class is prohibited. However, the derived class is not an abstract class, because the destructor is not inherited, and an heir in the absence of clearly defined destructor, it is created automatically. When you declare a pure virtual destructor to write and its definition.