Virtual functions

- In C ++, polymorphism is implemented in two ways: at the compilation stage and at the execution stage. At the compilation stage, polymorphism is realized by overloading functions and operators. This is a static polymorphism. At the execution stage (dynamic polymorphism) the polymorphism is realized through virtual functions.

- Pointers to derived classes are the basis for virtual functions and dynamic polymorphism.

- A pointer to a base class may appear as a pointer to an object of any derived class.

- The following example shows some valid code. The type inconsistency error is not generated when the base class type pointer points to an object of the derived class.

```
class  base
{
……..
};

class derived : public base
{
…….
};

void main()
{
        base ob_base;
        derived ob_derived;
        base *ptr = NULL;

        ptr = &ob_base;       //Pointer ptr points to the base class object
        ptr = &ob_derived;    //Pointer ptr points to the derived class object
}
```

- If the base class type pointer points to an object of the derived class, then:

  - through this pointer we can access only those components of the derived class that inherit from the base class - the base class knows nothing about the components that arose in the derived class.

  - Pointer arithmetic is associated with the data type to which that pointer remained declared. From this, it follows:

```
derived ob_derived[10]
base *ptr = &ob_derived[0];
ptr++;     //does not point to the next object of the derived class!
```

  - The derived class pointer cannot be used to access objects of the base class.

- A virtual function is a member declared in a base class and defined in a derived class.

- To create a virtual function, precede its declarations in the base class with the keyword *virtual*. A derived class that inherits a virtual function from the base class defines it according to its needs.

➢ In the base class, virtual functions define the interface form. Each redefinition of a virtual function in a derived class determines a specific implementation of this function for a given derived class. This is how a specific method is created.

➢ The keyword *virtual* is not needed when defining a virtual function in a derived class.

➢ Virtual functions can be called as ordinary functions - class methods. After all, dynamic polymorphism occurs if virtual functions are called by a pointer to a base class type that points to a derived class object.

➢ When a base class pointer is used to point to an object containing a virtual function, C ++ decides which version of the function to choose based on the type of the object pointed to by that pointer. The version of virtual function is selected at the stage of program execution - the virtual function that belongs to the class object, pointed by the pointer of the base class type, will be called.

Example W34.

➢ Redefining a virtual function in a derived class is different from the function overload:

- the prototype of the function (name, number, types of formal arguments and type of return value) redefined in the derived class must exactly match the prototype in the base class. (With function overload, each realization differs with the list of formal arguments). If the prototype of a virtual function remains changed in a derived class, such a function will be treated as an overloaded function, not a virtual one.

- virtual functions must be non-static members of the class they belong to (overloaded functions - not).

- The choice of what virtual functions is selected during program execution (for overloaded functions - at the compilation stage).

- Constructors cannot be virtual functions, and destructors can be.

Example W35.

- If we allocate memory for an object of derived class D using a pointer of the base class B, and then, we release the memory of the derived class using the pointer of a base type, then destructor of a derived class is not called if it does not declare as a virtual destructor. If the base class does not contain a virtual destructor, then there is no guarantee that the derived class's destructor will be called.

➢ In order to emphasize the differences between virtual functions and overloaded functions, redefining virtual functions is referred to as *overriding*.

➢ Virtual functions can be called by using a base class reference.

Example W36.

➢ A virtual function preserves the virtual attribute when inherited. This means that when the next derived class is inherited in another derived class, the virtual function may also be overridden.

➢ If the virtual function has not been overridden in the derived class, then the objects of this class referring to such a function will use the function defined in the base class.

Abstract functions (pure virtual functions)

➢ It often turns out that in the base class it is impossible to create a useful definition of a virtual function - the base class simply does not have enough information. Then the base class contains only declarations (prototypes) of the virtual functions without its definition - it defines the interface:

virtual return_type function_name (list_of_arguments) = 0;

- The “= 0” syntax tells the compiler that a function definition does not exist in the base class.

- Each derived class should have its own definition of each abstract function - it determines its own realization.

- A class that contains at least one abstract function is called an *abstract class*. Abstract class objects must not be created - such classes are not complete. After all, you can create pointers to abstract classes and references only.

- An important use of abstract classes and virtual functions is in class libraries. For example, the *Figure* abstract class contains common data for various planar geometric objects, but knows nothing about the implementation details - area computation, data writing to disk, reading from disk, displaying data on a monitor. But it is possible to extend the functionality of this class - create a derived class and define in it those virtual method implementations that are needed for each given object.

- Early linking refers to the event that occurs during compilation (the object and the function call are linked at the compile stage). Examples: calling an ordinary function, calling functions and overloaded operators. The main advantage is a high performance (since all information to call the function is determined at the compilation stage). Disadvantage - little flexibility.

- Late linking applies to functions whose calling is determined at program execution. Example - virtual functions, which are called by a pointer or references to the base class, and the decisions about which implementation will be called, will be made at program execution based on the type of the pointed object. The main advantage - flexibility, which makes it possible to react to various events that occur only during the execution of the program. It may enlarge the execution time (for the function call is not determined until it is executed).

Runtime Type Identification: RTTI

- C ++ implements polymorphism through class hierarchies, virtual functions, and base class pointers. Base class pointers can be used to point to base class objects as well as any objects derived from the base class.

- Sooner or later, it becomes necessary to solve the inverse problem - to find out what type of object the pointer of the base class type points to. This is done at the stage of program execution. C++ presents to us the operators *typeid* and *dynamic_cast* to solve the mentioned above problem.

- Operator *typeid* (add <*typeinfo*> header):

typeid(*object*) typeid(*reference_to_object*)

object – name of object, the type of which we have to define – it is a built-in type or a user-created type.

- Returns a reference of t*ype_info* to the argument object.

- The following public components are defined in the *type_info* class.

- bool operator == (const type_info &ob); //is used to compare the types
- bool operator != (const type_info &ob); //is used to compare the types
- bool before(const type_info &ob); //for internal use in sorting.
- const char *name(); //returns a pointer to a type name

Example 1.

```
class A
{
};

class B
{
};

void main()
{
    int i;
    A a;
    B b;
    cout << typeid(i).name() << endl;
    cout << typeid(a).name() << endl;
    cout << typeid(b).name() << endl;
    system("pause");
}
```

```
int
class A
class B
Для продолжения нажмите любую клавишу . . .
```

Example 2:

```
class A
{
};

class B
{
};

//template <class T, class K> bool funtypcompare(T &x, K &y)  //typeid(referencje_do_obiektu)
template <class T, class K> bool funtypcompare(T x, K y)
{
	cout << typeid(x).name() << " i " << typeid(y).name() << " ? \n";
	return (typeid(x) == typeid(y));
}

void main()
{
		int i = 0, j = 0;
		A a;
		B b;
		cout << funtypcompare(a, b) << endl;
		cout << funtypcompare(i, j) << endl;
		system("pause");
}
```

```
class A i class B ?
0
int i int ?
1
Press any key to return  . . .
```

➢ Example 3. Polymorphic classes. A base class pointer can point to a base class object as well as a derived class. To determine the type of an object, we use the indirect operator * *ptr.* If the pointer is null, operator *typeid* (* *ptr*) throws *bad_typeid* exception. For polymorphic classes, the operator *typeid* (* *ptr*) automatically returns the real type to which the pointer *ptr* points.

```
class B
{
public:
    virtual void f() { cout << “I am B\n"; }
};

class D1 : public B
{
public:
    void f() { cout << “I am D1\n"; }
};
```

```
void main()
{
	B b, *pb, *pnull = NULL;
	D1 d1;

	pb = &b;
	pb->f();
	cout << "pb points to the object of type " << typeid(*pb).name() << endl;

	pb = &d1;
	pb->f();
	cout << "pb points to the object of type " << typeid(*pb).name() << endl;

	try
	{
		cout << typeid(*pnull).name() << endl;
	}
	catch(bad_typeid aa)
	{
		cout << "bad pointer\n";
	}
	system("pause");
}
```

```
I am  B
pb points to the object of type class B
I am D1
pb points to the object of type class D1
bad pointer
Press any key to continue...
```

➢ The non-polymorphic classes - code is the same as before, only the virtual keyword has been commented out.

```
class B
{
public:
        /*virtual*/ void f() { cout << “I am B\n"; }
};

class D1 : public B
{
public:
        void f() { cout << “I am D1\n"; }
};

void main()
{
    B b, *pb;
    D1 d1;

    pb = &b;
    pb->f();
    cout << " pb points to the object of type " << typeid(*pb).name() << endl;

    pb = &d1;
    pb->f();
    cout << " pb points to the object of type " << typeid(*pb).name() << endl;
}
```

```
I am B
pb points to the object of type B
I am B
pb points to the object of type B
Press any key to continue...
```

- The second form of the *typeid* operator is used for type comparison (does an object have a given type?) :

typeid(*type_name*)

```
class B
{
public:
        virtual void f() { cout << "Jestem B\n"; }
};

class D1 : public B
{
public:
        void f() { cout << "Jestem D1\n"; }
};

void main()
{
        B b, *pb;
        D1 d1;

        pb = &d1;
        if(typeid(*pb) == typeid(D1))
                cout << " pb points to the object of type D1\n";

        system("pause");
}
```

```
pb points to the object of type D1
Press any key to continue...
```

- The operator typeid can be used for class templates.

Casting operators

- *dynamic_cast* - used to cast a pointer to a different type pointer or a reference to a different type reference at runtime and verify the correctness of the cast.

dynamic_cast < *target_type* > (*expression*)

- *target_type* - cast target type, *expression* - expression that is cast to the new type.

- *expression* - must expand to a pointer or reference.

- Success result - *dynamic_cast* returns a valid pointer or reference; failed - *dynamic_cast* returns NULL if *expression* is a pointer or throws *bad_cast* exception if *expression* is a reference.

- The *dynamic_cast* operator is used to perform a cast for polymorphic types.

- We have two polymorphic classes: B - base class, D - derived class.

1. You can always cast the pointer D * to the pointer B * (because the pointer of the base class can always point to the object of the derived class).

2. The pointer B * can be cast onto the pointer D * only if the pointed object is really an object of class D.

- The generalization. The use of *dynamic_cast* will be successful if the pointer (or references) being cast has the target type or points to an object derived from the target type. Otherwise, failure.

```
B b, *pb;
D1 d1, *pd1;

pd1 = &d1;
pb = dynamic_cast<B *> (pd1);   //OK, base class pointer always may point to an object of the derived class.
pb = &d1;
pd1 = dynamic_cast <D1 *> (pb); //OK, base class pointer points to a derived class object.

pb = &b;
pd1 = dynamic_cast<D1 *> (pb);  //!OK, base class pointer does not point to an object of type D1
```

Example W42.

- The *dynamic_cast* <> () operator can be used instead of the *typeid* () operator.

We suppose that we have polymorphic classes B, D (base, derived).

```
B *pb;
D *pder;
//……….
if(typeid(*pb) == typeid(D))  //does it point to a derived class object?
        pder = (D *)pb;     //if so, then we cast the bp pointer on the type D
else
        pder = NULL;
```

- Exactly the same can be done in one line of code:

```
pder = dynamic_cast<D *> (pb);
```

- It is an error !!!

```
pder = (D *)pb;
```

- The compiler won't report anything, but if *pb* doesn't point to an object of class D, the program is not expected to run. Such regular errors are very hard to detect in extensive code, because they may not appear every time the program is started.

Example W43.

➢ If the base class pointer points to an object of the derived class of type Typ, then and only then the cast p_derived = *dynamic_cast* <Typ >(p_base) will be successful. So the condition:

```
if(p_derived = dynamic_cast<Typ *> (p_base) )
{
        // here it is guaranteed that the p_base pointer points to an object of polymorphic
        //derive class of type Typ.
}
else
        //p_base does not point to an object of the polymorphic derive class of type Typ.
```

Operator *const_cast*

const_cast<typ> (*expression*)

➢ overrides *const* and / or *volatile* when cast. The target type must be the same as the source type. Most popular usage - removing the influence of the *const* attribute.

- *typ* – target type
- *expression* – an expression, casted to on new type.

```
void fun(const double *ptr)
  {
      double *p;
      p = const_cast<double *> (ptr);  //remove the const modifier, otherwise the compiler will report
                                       //type inconsistency (double and const double)
      //(*ptr)++;                      //error! Modifier const saves an array ptr from changes.
      (*p)++;                          //OK
      cout << *p << endl;
  }

  void main()
  {
      double a[2] = {1.0, 4.0};
      fun(a);
      system("pause");
  }

  2
  Press any key to continue . . .
```

Operator static_cast<> ()

static_cast<*typ*> (*expression*)

- Performs a non-polymorphic cast of the coherent types (for instance, *double* → *int*). No checks are performed in the cast step. This replaces the original cast operator.

  - *typ* – target type
  - *expression* – an expression casted on a new type

```
int i = 10;
double  a;
a = static_cast<double> (i);
//the same:  a = (double)i;
```

Operator reinterpret_cast<> ()

reinterpret_cast<*typ*> (*expression*);

- Is used to convert the given type to a completely different type.

```
void main()
{
    FILE *pf = fopen("myfile.dat", "w+");  //open file for read and write
    if(!pf) { //error handling }
    double arr[] = {1.0, -2.0, 3.141592};
    const char *str = reinterpret_cast<const char *>(arr);
    for(size_t it=0; it<sizeof(arr); ++it)
    {
        if(fputc(str[it], pf) == EOF) { // error handling }  //byte-by-byte writing to file
    }
    fseek(pf, 0, SEEK_SET); //move the file pointer to the beginning of file
    double brr[3];
    char *strs = reinterpret_cast<char *>(brr);
    char ch = 'a';    size_t it = 0;
    while(ch != EOF)
    {
        if((ch = fgetc(pf))  != EOF) {  // reading from file byte-by-byte until we meet EOF
                strs[it++] = ch;
        }
    }
    fclose(pf);  pf = NULL;  //close file
    remove("myfile.dat");    //remove file
}
```