Лабораторная работа №7.
Наследование. Полиморфизм.

Общие сведения.

Наиболее значимой особенностью ООП является наследование. Это процесс создания новых классов, называемых наследниками или производными, из уже существующих или базовых классов. Наследование заключается в приме в производном классе компонент базового класса. Кроме того, производный класс дополняется своими собственными компонентами.

Объявление производного класса представляется следующим образом:

```
class Имя_класса : [private|protected|public] Имя_базового_класса
{
        Тело класса;
};
```

По определению считается, что все компоненты базового класса являются компонентами производного, за исключением конструкторов, деструкторов и операции присваивания (=). Предполагается, что первое поле производного класса расположено после всех полей, наследованных им от базового класса.

Если базовых полей несколько, в заголовке производного класса они должны быть перечислены все через запятую, каждый со своим ключом доступа.

Очередность активизации конструкторов следующая:

- конструктор базового класса;
- конструктор подобъектов;
- конструктор производного класса.

Соответствующие деструкторы активизируются в обратом порядке.

Ключи доступа.

Перед именем базового класса ставят ключи доступа: private (для классов по умолчанию), protected или public (для структур по умолчанию). Слово private указывает на приватный базовый класс, а public – на обобществленный.

Не следует путать эти ключевые слова с одноименными словами в теле класса, описывающими уровень доступа к компонентам класса. Рассмотрим и разберем смысл этих слов на примерах.

Пример 1. Производный класс с обобществленным базовым классом.

```
class Base
{
        int base;
public:
        Base(){};
        Base(int b)
        {
                base = b;
        }
        void Out()
        {
                cout << " Base: " << base << endl;
        }
};
```

```
class Derived :public Base          //  производный класс с обобществленным
                                    //  базовым классом Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
        void Out()
        {
                cout << " Derived: " << derived << endl;
        }
};
```

В этом случае все компоненты производного класса имеют доступ к элементам базового класса в соответствии с их ключами доступа. Например, следующее обращение в методе производного класса приведет к ошибке:

```
void Out()
        {
                cout << " Base: " <<  base << endl;
                cout << " Derived: " << derived << endl;
        }
```

visual studio 2010\projects\inher_1\inher_1\inher_1.cpp(31): error C2248: base: невозможно обратиться к private члену, объявленному в классе "Base".

Прямой доступ к закрытым компонентам базового класса не допустим. Ситуация изменится, если обратиться через метод базового класса или разместить поле base в области protected базового класса. Эта область, подобно области private, не допускает обращение к полю извне, но делает поле доступным для всех компонентов производного класса. В этом и состоит отличие слов private и protected.

Пример 2. Производный класс с защищенным базовым классом.

```
class Derived : protected Base//производный класс с защищенным базовым классом
Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
        void Out()
        {
                cout << " Base: " << base << endl;
                //  если поле base в области protected
                // или Base::Out(); если поле base в области private
                cout << " Derived: " << derived << endl;
        }
};
```

В этом случае поле private базового класса остается таким же, а поля protected и public, понимаются как protected, то есть защищенными от внешнего воздействия, но доступными из производного класса.

Пример 3. Производный класс с приватным базовым классом.

Если в заголовке базовый класс описывается с ключевым словом private или спецификатор отсутствует, считается, что базовый класс приватный. В этом случае все поля базового класса без исключения считаются по умолчанию private.

```
class Derived : private Base                    //производный класс с закрытым
(приватным) базовым классом Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
        void Out()
        {
                cout << " Base: " << base << endl; // если поле base в области protected
                Base::Out();
                cout << " Derived: " << derived << endl;
        }
};
```

Обратиться к отдельным компонентам можно средствами оператора разрешения области видимости, например, Base::Out();. Обращение к области private по-прежнему допустимо только через методы базового класса.

Конструктор производного класса.

При определении конструктора производного класса необходимо обеспечить инициализацию полей базового класса. Осуществляется это путем использования конструктора со списком инициализации. Другим способом передать параметр и инициализировать базовый класс нельзя. Рассмотрим еще раз конструктор класса Derived:

```
Derived(int b, int d):Base(b)
{
        derived = d;
}
```

Здесь в списке инициализации используется выражение :Base(b), обеспечивающее инициализацию поля базового класса. Если в базовом классе есть конструктор требующий несколько параметров, в конструкторе производного класса необходимо обеспечить их инициализацию через список инициализации. Попытка передачи параметров в теле конструктора, например, Base = b;, приведет к ошибке.

Задание №1.

Объявить базовый и производный классы, моделирующие слово (базовый класс) и строку (производный класс). В базовом классе предусмотреть поле для хранения одного слова произвольной длины (можно использовать тип string), а в производном, кроме

наследованного слова, необходимо объявить поле, содержащее количество слов в строке. Для классов определить методы, обеспечивающие ввод слов и строк и вывод их на экран. Проанализировать случаи приватного, защищенного и обобществленного наследования.

Задание №2.

В конструкторах базового и производного классов обеспечить вывод в стандартный поток сообщения, идентифицирующего принадлежность объекта тому или иному классу. Проанализировать последовательность активизации конструкторов при объявлении объектов производного класса. Дополнить классы деструкторами, оценить порядок их активации.

Простое наследование.

Наследование называется простым, если у производного класса один базовый класс. Рассмотрим примеры наследования составных частей базового класса при простом наследовании.

Наследование статических компонентов базового класса.

```
class Base
{
protected:
	int base;
public:
	Base(){};
	Base(int b)
	{
		base = b;
	}
	static int st_base;
	static  int Get()
	{
		return st_base;
	}
	friend ostream &operator <<(ostream &, const Base &);
};

int Base::st_base = 64;

ostream &operator <<(ostream &out, const Base &b)
{
	out << b.base << endl;
	return out;
}
class Derived : public Base
{
	int derived;
public:
	Derived(){};
	Derived(int b, int d):Base(b)
	{
		derived = d;
		// можно и здесь  st_base = 33;
	}
```

```
        /*

        static int Get()
        {
                return st_base;
                // можно переопределить, а можно наследовать из базового
        }
        */
        static void SetStat()
        {
                st_base = 33;
        }
};
```

В теле основной функции наберите следующий код и проанализируйте результаты.

```
Base bas(2000);
Derived der(33,64);
cout << " Статическое поле базового класса: " << bas.Get() << endl;
der.SetStat();
cout << " Статическое поле базового класса: " << der.Get() << endl;
```

Доступ к статическому полю базового класса возможен непосредственно в методе производного класса, если оно объявлено в открытой области или через метод базового класса, если объявлено в закрытой. Наследование статических переменных имеет практический смысл, так как в иерархии возможны элементы, доступные для любого объекта данной иерархии. Жизненным примером может служить наследование потомками фамилии родителей (предков). Необходимо заметить, что объект производного класса может не только воспользоваться значением статического поля, но и изменить его при необходимости. В приведенном примере в производном классе описан метод SetStat(), позволяющий менять статическое поле.

Метод static int Get() в производном классе переопределен, но закомментирован, в этом случае будет вызываться метод базового класса. Для вызова своего метода, с него необходимо снять комментарии.

Задание №3.

Дополните объявление класса слово статическим полем, приведите пример его использования объектами и методами производного класса строка.

Самостоятельно разобрать и привести пример наследования константных и ссылочных полей.

Наследование дружественных функций.

Рассмотрим возможность использования функции дружественной базовому классу объектами производного класса. Это допустимо, поскольку объект производного класса по умолчанию считается объектом базового класса. Необходимо заметить, что функция дружественная базовому классу, имеющая неограниченный доступ к его полям, не имеет доступа к полям производного класса. Для устранения этого недостатка необходимо переопределить аналогичную функцию, дружественную производному классу, которая будет дружественной и базовому классу. В качестве примера рассмотрим дружественную функцию-операцию вывода в стандартный поток.

```
class Base
{
protected:
	int base;
public:
	Base(){};
	Base(int b)
	{
		base = b;
	}
	friend ostream &operator <<(ostream &, const Base &);
};

ostream &operator <<(ostream &out, const Base &b)
{
	out << "  Функция-операция дружественная классу Base: " << b.base  << endl;
	return out;
}

class Derived : public Base
{
	int derived;
public:
	Derived(){};
	Derived(int b, int d):Base(b)
	{
		derived = d;
	}
};

int main()
{
	setlocale(0,"RUS");
	Base bas(2000);
	Derived der(33,64);
	cout << bas;
	cout << der;
	system("pause");
	return 0;
}
```

В этом примере объект производного класса пользуется услугами операций вывода: cout << der;, объявленной в базовом классе. Разберите этот пример и проанализируйте результаты.

Задание №4.

В классе слово определить дружественную функцию, выводящую отдельное слово в стандартный поток. Перегрузите аналогичную функцию в классе строка. Приведите примеры использования дружественных функций.

Переменная this в условиях наследования.

У каждой нестатической составной функции класса есть доступ к переменной this, указывающую на ту часть объекта, которая является производной этого класса. Рассмотрим следующий пример иерархии классов.

```
class One
{
        int one_1, one_2;
public:
        One(int o_1, int o_2):one_1(o_1), one_2(o_2){};
        int Get()
        {
                cout << " Адрес объекта: " << this << ' ';
                return one_1;
        }
};

class Two_1 :public One
{
        long two_1;
public:
        Two_1(int o_1, int o_2):One(o_1, o_2){};
};

class Two_2 :public One
{
        char two_2;
public:
        Two_2(int o_1, int o_2):One(o_1, o_2){};
};

class All :public Two_1, public Two_2
{
public:
        All(int o_1, int o_2, int o_3, int o_4):
          Two_1(o_1, o_2), Two_2(o_3, o_4){};
};

All  Var(1,2,3,4);
int main()
{
        setlocale(0,"RUS");
        cout << " Значение поля: " << Var.Two_1::Get() << endl;
        cout << " Значение поля: " << Var.Two_2::Get() << endl;
        system("pause");
        return 0;
}
```

В каждом из двух обращений переменная this указывает на различные части объекта Var. Выполнение программы покажет различные адреса и значения частей объекта.

Задание №5. *

Создать иерархию классов, показать пример наследования переменной this объектами производных классов.

Преобразование указателей.

Важной особенностью производного класса с обобществленным базовым классом является возможность преобразования указателей объекта этого класса в указатель объекта базового класса.

Разрешается также преобразование ссылки на производный класс в ссылку на объект базового класса, в том числе связывание параметра ссылки на базовый класс с выражением, идентифицирующим объект производного класса.

Оба преобразования относятся к стандартным и не требуют явного использования операции преобразования.

Допускается также присвоение объекту базового класса объекта его производного класса.

Пример 1. Преобразование указателя объекта производного класса на указатель объекта базового класса.

```
class Base
{
protected:
	int base;
public:
	Base(){};
	Base(int b)
	{
		base = b;
	}
	int Out()
	{
		return base;
	}
};

class Derived : public Base
{
	int derived;
public:
	Derived(){};
	Derived(int b, int d):Base(b)
	{
		derived = d;
	}


};

Derived der(33,64);
int main()
{
	setlocale(0,"RUS");
	Base *ptr_Base = (Base *)&der;
	cout << ptr_Base->Out() << endl;
```

```
	system("pause");
	return 0;
}
```

В этой программе объявление Base *ptr_Base = &der; равносильное Base *ptr_Base = (Base *)&der;, относится к стандартным и явного преобразования можно не делать. В случае, если базовый класс описывается с ключом private или protected, преобразования нужно задавать явно.

Пример 2. Присвоение объекту базового класса объекта производного класса.

```
class Base
{
protected:
	int base;
public:
	Base(){};
	Base(int b)
	{
		base = b;
	}
	int Out()
	{
		return base;
	}
};

class Derived :  public Base
{
	int derived;
public:
	Derived(){};
	Derived(int b, int d):Base(b)
	{
		derived = d;
	}


};

Derived der(33,64);
int main()
{
	setlocale(0,"RUS");
	Base bas;
	bas = *(Base *)&der;
	cout << bas.Out() << endl;
	system("pause");
	return 0;
}
```

В этом примере оператор bas = *(Base *)&der; можно заменить его эквивалентом bas = &der;, так как подобное преобразование компилятор выполнит автоматически. Проверьте работоспособность примера. Замените ключ доступа в объявлении производного класса и вновь проверьте. Проанализируйте сообщения компилятора.

Виртуальные функции.

Виртуальной (полиморфной) функцией является каждая нестатическая составная функция класса, объявленная со спецификатором virtual, а также каждая функция такого же типа, содержащаяся в произвольной последовательности производных классов, происходящих от этого класса. Использование в объявлении класса хотя бы одной полиморфной функции приводит к расширению полей класса на скрытое идентификационное поле, в котором хранится информация о типе объекта данного класса. Полиморфным может быть деструктор, несмотря на то, что в разных классах у него разные имена.

При обработке виртуальных методов компилятор формирует таблицу виртуальных методов (vtbl), в которой для каждого виртуального метода записывается его адрес в памяти. Каждый объект данной иерархии содержит дополнительной скрытое поле ссылки на vtbl, называемое vptr. Это поле заполняется конструктором класса при создании объекта. При компиляции программы ссылки на виртуальные методы замещаются на обращения к таблице виртуальных методов (vtbl) через указатель (vptr) объекта. На этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов, выполняется через дополнительный этап получения адреса функции из таблицы. Описанный механизм носит название механизма позднего или динамического связывания. Через виртуальные методы в C++ реализуется полиморфизм.

Рассмотрим пример полиморфной функции.

```
class First
{
        int first;
public:
        First(){};
        First(int f):first(f){};
        virtual void Out()
        {
                cout << " First: " << first << endl;
        }
};

class Second :public First
{
        int second;
public:
        Second(){};
        Second(int f, int s):First(f)
                { second = s;}
        void Out()
        { cout << " Second: " << second << endl; }
};

class Third :public Second
{
```

```
        int third;
public:
        Third(int f, int s, int t): Second(f,s)
        {
                third = t;
        }
        void Out()
        { cout << "Third: " << third << endl;}
};



int main()
{

        First *ptr_first = new First;
        Second sec(100,200);
        ptr_first = &sec;
        ptr_first->Out();               // (1)
        Third trd(300,400,500);
        ptr_first = &trd;               // (2)
        ptr_first->Out();
        return 0;
}
```

Функция Out() в данной иерархии классов является виртуальной. Вызов функции ptr_first->Out(); приведет к появлению различных результатов, так как в первом случае будет работать функция Out() класса Second, а во втором - класса Third.

Механизм виртуальных функций работает только через указатели или ссылки на объекты. Объект, определенный через указатель, называется полиморфным, что означает выполнение различных действий, в зависимости, на какой фактический объект ссылается указатель.

Задание №6.

Разработать иерархию классов «человек, служащий, студент», в которой класс человек имеет поля имя, фамилия, возраст. Класс служащий дополняет его полем специальность, а класс студент – полями группа и средний балл. Предусмотреть полиморфные методы, позволяющие получить информацию о субъекте в зависимости от его типа.

Абстрактный класс.

Если первые полиморфные функции данной иерархии являются избыточными, их можно определить как чисто виртуальные. Функция является чисто виртуальной, если вместо тела будет размещена запись =0, например, void Out()=0;. Класс, содержащий, по меньшей мере, одну чисто виртуальную функцию, называется абстрактным классом. Запрещается объявлять объекты его типа, но допускаются указатели. Абстрактный класс обычно выступает как прародитель большой иерархии, содержащий в себе наиболее общие свойства, характерные для всей иерархии.

Задание №7.

В задании №6 класс человек определить как абстрактный. Привести примеры использования абстрактного класса.

Операция dynamic_cast.

Эта операция применяется для преобразования указателей родственных классов, в основном указателя базового класса в указатель на производный класс. Формат операции следующий: dynamic_cast<тип *>(выражение);.

Выражение – это указатель или ссылка на класс, тип – производный или базовый для данной иерархии. Операция dynamic_cast предусматривает проверку допустимости преобразования. В случае успешного выполнения операции формирует результат заданного тип, в противном случае в качестве результата возвращает 0 для указателя и bad_cast для ссылки.

Преобразование производного класса в базовый класс называется *повышающим*, относится к стандартным преобразованиям. Явное преобразование можно опустить. Приведение базового класса в производный носит название *понижающего* преобразования, требуется явный вызов оператора dynamic_cast. Возможно также преобразование между базовыми классами для одного производного класса, а также между производными от одного базового класса. Это *перекрестное* преобразование.

Пример 1. Повышающее преобразование.

```
class Base
{
protected:
	int base;
public:
	Base(){};
	Base(int b):base(b){};
	void Out()
	{
		cout << " Base: " << base << endl;
	}
};

class Derived :public Base
{
	int derived;
public:
	Derived(){};
	Derived(int b, int d):Base(b), derived(d){};
	void Out()
	{
		cout << " Derived: " << derived << endl;
	}
};
int main()
{
	setlocale(0,"RUS");
	Base *ptr_Base = new Base(100);
	Derived *ptr_Derived = new Derived(222,333);
	ptr_Base =  dynamic_cast<Base *>(ptr_Derived);
	//  или  ptr_Base = ptr_Derived;
	ptr_Base->Out();
	system("pause");
```

```
	return 0;
}
```

Здесь выражение ptr_Base = dynamic_cast<Base *>(ptr_Derived); преобразует указатель ptr_Derived на класс Derived к указателю класса Base. Результатом работы будет число 222, так как относится к базовой части объекта производного типа. При удалении этого оператора из программы, результат изменится на 100.

Пример №2. Понижающее преобразование.

Операция dynamic_cast чаще всего используется для понижающего преобразования. Производные классы могут иметь методы, которых нет в базовых классах. Для их вызова через указатель на базовый класс необходимо иметь уверенность в том, что указатель действительно ссылается на объект производного класса. Такая проверка осуществляется в момент выполнения приведения типа с использованием RTTI (run time type information) – информации о типе во время исполнения. Для допустимости такой проверки, аргумент операции dynamic_cast должен быть полиморфного типа. Для использования RTTI необходимо подключить заголовочный файл <typinfo>. Рассмотрим модифицированный случай предыдущего примера.

```
class Base
{
protected:
	int base;
public:
	Base(){};
	Base(int b):base(b){};
	virtual void Out()
	{
		cout << " Base: " << base << endl;
	}
};
class Derived :public Base
{
	int derived;
public:
	Derived(){};
	Derived(int b, int d):Base(b), derived(d){};
	void Out()
	{
		cout << " Derived: " << derived << endl;
	}
};
int main()
{
	setlocale(0,"RUS");
	Base *ptr_Base = new Base;
	Derived *ptr_Derived = new Derived(20, 30);
	ptr_Base = new Derived(222,333);
	dynamic_cast<Derived *>(ptr_Base)->Out();
	system("pause");
	return 0;
}
```

В этом примере в качестве результата следует ожидать число 333. Оператор dynamic_cast<Derived *>(ptr_Base)->Out(); осуществляет приведение указателя базового класса к указателю на производный и поэтому работает метод производного класса. Заметим еще раз, что функция Out() в базовом классе должна быть виртуальной, иначе преобразование не состоится. В этом примере не осуществляется контроля преобразования, что может привести к ошибочным ситуациям.

Контроль преобразования лучше осуществить в какой-либо функции (желательно обрабатывающую исключительную ситуацию). Простой пример приведен ниже. Классы заимствованы из предыдущего примера.

```
void fun(Base *pBase)
{
        Derived *pDerived = dynamic_cast<Derived *>(pBase);
        if(pDerived) pDerived->Out();
        else cout << " Передан объект не производного класса" << endl;
}
```

В теле основной функции выполните следующую последовательность операторов и оцените результаты.

```
Base *ptr_Base = new Base(100);
Derived *ptr_Derived = new Derived(20, 30);
fun(ptr_Base);
fun(ptr_Derived);
```

Задание №7.

В иерархии «Человек, служащий, студент» преобразовать указатель на класс человек в указатель на класс служащий и студент. Показать результаты работы.

Пример №3. Преобразование ссылок.

Преобразование ссылок производных классов в ссылку на базовый класс относятся к стандартным и не рассматриваться не будет. Понижающее преобразование ссылок имеет несколько другой смысл, чем преобразование указателей. Поскольку ссылка всегда указывает на конкретный объект, операция dynamic_cast должна выполнять преобразование именно к типу этого объекта. Корректность преобразования проверяется автоматически, в случае невозможности преобразования порождается исключение bad_cast. Пример возможного понижающего преобразования без обработки исключений рассмотрен ниже.

```
class A
{
public:
        virtual void f1()
        { cout << " Class A " << endl;}
};

class B :public A
{
public:
        virtual void f2()
        { cout << " Class B " << endl;  }
};
```

```
int main()
{
	B b;
	B &b_ref = b;
	A &a_ref = b_ref;
	dynamic_cast<B &>(a_ref).f2();

	system("pause");
	return 0;
}
```

Выражение dynamic_cast<B &>(a_ref).f2(); преобразует ссылку a_ref к ссылке на класс A. Важно наличие оператора A &a_ref = b_ref;, который обеспечивает понижающее преобразование. При его отсутствии система сгенерирует исключение.

Этот пример демонстрационный, для его полноты необходимо отследить тип передаваемого объекта.

Задание №8.

В соответствие с заданием №7 обеспечить понижающее преобразование ссылок класса человек в ссылку на класс служащий, ссылки на класс служащий в ссылку на класс студент.