Лабораторная работа №2

Наследование и динамический полиморфизм

Класс, который **наследуется** другим классом, называется **базовым** или **родительским**. Класс, выполняющий наследование, называется **производным** или **дочерним.** При этом как у одного базового класса может быть несколько родительских (множественное наследование), так и у одного родительского несколько базовых.

Спецификаторы доступа при наследовании

В C++ члены класса классифицируются в соответствии с правами доступа на следующие три категории: открытые (public), закрытые (private) и защищенные (protected).

От того, с каким **спецификатором доступа** объявляется наследование базового класса, зависит статус доступа к членам производного класса. Общая форма наследования классов имеет следующий вид:

```
class имя_производного_класса: доступ имя_базового_класса
{
      // код
};
```

Здесь **доступ** определяет, каким способом наследуется базовый класс. Спецификатор доступа может принимать одно из трех значений: **private**, **protected**, **public**. Если он не указан, то: подразумевается **public** для производных классов объявленных через **struct**, и **private** – для объявленных через **class.**

Если спецификатор принимает значение **public**, то все открытые и защищенные члены базового класса становятся соответственно открытыми и защищенными членами производного класса.

Если указан **protected**, то все открытые и защищенные члены базового класса становятся защищенными членами производного класса.

Если указан **private**, то все открытые и защищенные члены базового класса становятся закрытыми членами производного класса.

Деструкторы производных классов

В деструкторе производного класса компилятор автоматически генерирует вызовы деструкторов его базовых классов.

Для удаления объекта производного класса следует сделать деструктор в базовых классах виртуальным с целью корректного вызова деструкторов при использовании оператора **delete.**

Передача параметров в базовый класс

Когда базовый класс имеет конструктор с аргументами, производные классы должны передавать базовому классу необходимые аргументы. Для этого используется расширенная форма конструкторов:

```
порожденный_конструктор(список_аргументов):
      базовый1(список_аргументов1), …, базовыйN(список_аргументовN)
{
      // код
}
```

Здесь под базовый1, …, базовыйN обозначены имена базовых классов. С помощью двоеточия конструктор производного класса отделяется от списка конструкторов базового класса. Списки аргументов, передаваемых в конструкторы базовых классов, могут состоять из констант, глобальных переменных или параметров конструктора производного класса.

Если конструкторы производных классов не вызываются в явном виде (как в примере выше), вместо них из базовых классов автоматически вызываются конструкторы без параметров. Кроме того, конструкторы без параметров автоматически вызываются при инициализации **вместо** указанных конструкторов с параметрами в случае, если имеет место виртуальное наследование (см. ниже).

Указатели и ссылки на производные классы

В общем случае указатель одного типа не может указывать на объект другого типа, однако в C++ указатель на базовый класс может указывать на объект производного класса. Ссылки на базовый класс могут быть использованы для ссылок на объект производного типа.

Виртуальные функции

Виртуальная функция – это функция, объявленная с ключевым словом **virtual** в базовом классе и **переопределенная** в производных классах.

Это означает, что при использовании объекта производного класса **с помощью указателя или ссылки на него как на объект базового класса,** виртуальная функция вызывается из того класса, посредством которого этот объект был **создан**. То есть, в одном и том же участке кода для разных объектов могут вызываться разные версии одной и той же виртуальной функции. Класс, содержащий хотя бы одну виртуальную функцию, называется **полиморфным** классом.

Для виртуальных функций существуют следующие правила:

- виртуальную функцию нельзя объявлять как **static**;
- спецификатор **virtual** необязателен (но желателен) при **переопределении** функции в производном классе.

Рассмотрим пример использования виртуальных функций:

```
#include <iostream>
#include <string>
#include <exception>
#include <clocale>
#include <typeinfo>

using namespace std;

// класс, описывающий существ
class creature
{
private:
	string title;
	double mass;  //kg
public:
	// конструктор без параметров
	creature() : mass(0.0) {}

	// конструктор копирования
	creature(const creature& obj)
		: title(obj.title), mass(obj.mass) {}
```

```
	// конструктор с параметрами
	creature(const string& _title, double _mass)
		: title(_title), mass(_mass) {}

	// деструктор (объявлен как виртуальный)
	virtual ~creature()
	{
		cout << "creature deleted" << endl << endl;
	}
protected:
	// защищенная виртуальная функция для вывода информации об объекте
	// не должна использоваться вне данного класса и его производных
	virtual void _print() const
	{
		cout << "title: " << title
		     << ", mass: " << mass;
	}
public:
	// открытая функция для вывода информации об объекте
	void print() const
	{
		// выводит название класса, к которому принадлежит
		// объект, для которого она вызвана,
		cout << typeid(*this).name() << ": (";

		// и вызывает виртуальную функцию
		// т.к. функция _print виртуальная, вызываться она
		// будет не только из текущего класса, но и из производных,
		// в зависимости от того, для какого объекта
		// осуществляется вызов
		_print();

		cout << ")" << endl;
	}
};

// класс animal наследуется от класса creature
class animal : public creature
{
private:
	double speed; //m/s
public:
	// конструкторы класса animal осуществляют вызовы конструкторов
	// базового класса (creature)
	animal() : creature() {}

	animal(const animal& obj)
		: creature(obj), speed(obj.speed) {}

	animal(const string& _title, double _mass, double _speed)
		: creature(_title, _mass), speed(_speed) {}
	~animal()
	{
		cout << "animal deleted" << endl;
	}
protected:
	// виртуальная функция _print переопределяется в производном классе
	void _print() const
	{
		creature::_print();
		cout << ", speed: " << speed;
	}
};

// класс bird наследуется от класса animal
class bird : public animal
{
```

```
private:
	double topfly; //km
public:
	bird() : animal() {}

	bird(const bird& obj)
		: animal(obj), topfly(obj.topfly) {}

	bird(const string& _title, double _mass
		, double _speed, double _topfly)
		: animal(_title, _mass, _speed), topfly(_topfly) {}
	~bird()
	{
		cout << "bird deleted" << endl;
	}
protected:
	// виртуальная функция _print переопределяется
	// теперь уже в классе bird
	void _print() const
	{
		animal::_print();
		cout << ", topfly: " << topfly;
	}
};

// класс fish наследуется от класса animal
class fish : public animal
{
private:
	double maxdeep; //km
public:
	fish() : animal() {}

	fish(const fish& obj)
		: animal(obj), maxdeep(obj.maxdeep) {}

	fish(const string& _title, double _mass
		, double _speed, double _maxdeep)
		: animal(_title, _mass, _speed), maxdeep(_maxdeep) {}
	~fish()
	{
		cout << "fish deleted" << endl;
	}
protected:
	// виртуальная функция _print переопределяется
	// в классе fish
	void _print() const
	{
		animal::_print();
		cout << ", maxdeep: " << maxdeep;
	}
};

int main()
{
	setlocale(0, "Rus");

	// создание объектов и вывод информации о них
	animal("Паук", 0.003, 0.05).print();
	bird("Ворона", 0.3, 10, 0.1).print();
	fish("Рыба Молот", 150, 5, 0.5).print();

	cout << endl;

	return 0;
}
```

Чисто виртуальные функции и абстрактные классы

Заметим, что не в любом случае для базового класса можно определить поведение той или иной функции. Примеры этого можно найти в реальном мире, в частности, все животные в том или ином виде могут перемещаться: рыбы плавают, звери бегает, змеи ползают, птицы летают (упрощено для примера). Однако как именно перемещается некое абстрактное животное, сказать нельзя.

В подобных случаях C++ дает возможность не определять в базовом классе код виртуальной функции, такие функции определяются нулевым значением и называются **чисто виртуальными**, например:

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

Класс, в котором определена хотя бы одна чисто виртуальная функция, называется **абстрактным**. Следует отметить, что создать нельзя объект от абстрактного класса непосредственно. Объекты можно создавать только от тех его производных классов, в которых все чисто виртуальные функции переопределены.

Дополним предыдущий пример:

```
#include <iostream>
#include <string>
#include <exception>
#include <clocale>
#include <typeinfo>

using namespace std;

// класс creature
class creature
{
private:
	string title;
	double mass;  // kg
protected:
	// конструкторы вынесены в protected, т.к. необходимости создавать
	// объекты непосредственно этого класса нет, однако, эти
	// конструкторы должны быть доступы из производных классов
	// аналогично вынесен деструктор
	creature() : mass(0.0) {}

	creature(const creature& obj)
		: title(obj.title), mass(obj.mass) {}

	creature(const string& _title, double _mass)
		: title(_title), mass(_mass) {}

	virtual ~creature()
	{
		cout << "creature deleted" << endl << endl;
	}

	virtual void _print() const
	{
		cout << "title: " << title
			<< ", mass: " << mass;
	}
public:
	void print() const
	{
		cout << typeid(*this).name()
```

```
                  << ": ("; _print(); cout << ")" << endl;
      }

      // добавлена функция получения массы существа
      double get_mass() const
      {
            return mass;
      }
};

// класс animal
class animal : public creature
{
private:
      double speed; // m/s
protected:
      animal() : creature() {}

      animal(const animal& obj)
            : creature(obj), speed(obj.speed) {}

      animal(const string& _title, double _mass, double _speed)
            : creature(_title, _mass), speed(_speed) {}
      ~animal()
      {
            cout << "animal deleted" << endl;
      }
      void _print() const
      {
            creature::_print();
            cout << ", speed: " << speed;
      }
public:
      double get_speed() const
      {
            return speed;
      }
};

// класс bird
class bird : public animal
{
private:
      double topfly; // km
public:
      bird() : animal() {}

      bird(const bird& obj)
            : animal(obj), topfly(obj.topfly) {}

      bird(const string& _title, double _mass
            , double _speed, double _topfly)
            : animal(_title, _mass, _speed), topfly(_topfly) {}
      ~bird()
      {
            cout << "bird deleted" << endl;
      }
protected:
      void _print() const
      {
            animal::_print();
            cout << ", topfly: " << topfly;
      }
};
```

```
// класс fish
class fish : public animal
{
private:
	double maxdeep; // km
public:
	fish() : animal() {}

	fish(const fish& obj)
		: animal(obj), maxdeep(obj.maxdeep) {}

	fish(const string& _title, double _mass
		, double _speed, double _maxdeep)
		: animal(_title, _mass, _speed), maxdeep(_maxdeep) {}
	~fish()
	{
		cout << "fish deleted" << endl;
	}
protected:
	void _print() const
	{
		animal::_print();
		cout << ", maxdeep: " << maxdeep;
	}
};

// объявление нового класса - хищник (predator),
// класс predator наследуется от animal
class predator : public animal
{
protected:
	predator() {}
public:
	~predator() {}

	// чисто виртуальная функция hunt
	// будет определять посредством производных классов,
	// удастся ли хищнику поохотиться на жертву (obj)
	virtual bool hunt(const animal& obj) = 0;
	// т.к. hunt в данном классе - чисто виртуальная,
	// класс predator является абстрактным
};

// класс орел (eagle) наследуется от bird и predator
// (множественное наследование), т.к. орел - и птица, и хищник
class eagle : public bird, public predator
{
public:
	eagle() : bird() {}

	eagle(const eagle& obj)
		: bird(obj) {}
	eagle(double _mass
		, double _speed, double _topfly)
		: bird("Орел", _mass, _speed, _topfly) {}

	// определение тела функции hunt
	// т.к. hunt определена, класс eagle - НЕабстракный
	bool hunt(const animal& obj)
	{
		// т.к. функция get_mass может наследоваться классом eagle
		// из класса animal и через класс bird, и через класс predator
		// укажем, что она наследуется через bird
		return obj.get_mass()<bird::get_mass()
			&& obj.get_speed()<bird::get_speed();
	}
};
```

```
// проверка
int main()
{
	setlocale(0, "Rus");

	bird raven("Ворона", 0.3, 10, 0.1);
	eagle eagle1(1, 100, 1);
	fish hammerhead("Рыба-молот", 150, 5, 0.5);

	raven.print();
	hammerhead.print();

	cout << "Eagle vs raven: " << eagle1.hunt(raven) << endl;
	cout << "Eagle vs hammerhead: " << eagle1.hunt(hammerhead) << endl;

	cout << endl;

	return 0;
}
```

Также, в данном примере функцию **main** можно переписать, вызывая **hunt** через ссылку на **predator**:

```
int main()
{
	setlocale(0, "Rus");

	bird raven("Ворона", 0.3, 10, 0.1);
	fish hammerhead("Рыба-молот", 150, 5, 0.5);

	raven.print();
	hammerhead.print();

	// динамическое создание объекта класса eagle,
	// объект запоминается как ссылка на объект класса predator,
	// подобное возможно, т.к. eagle является производным для predator
	predator &eagle1 = *new eagle(1, 100, 1);

	cout << "Eagle vs raven: " << eagle1.hunt(raven) << endl;
	cout << "Eagle vs hammerhead: " << eagle1.hunt(hammerhead) << endl;

	cout << endl;

	// не забываем очистить динамически выделенную память
	delete &eagle1;

	return 0;
}
```

Виртуальное наследование

В описанном выше примере при вызове функций get_mass и get_speed в функции hunt класса eagle можно было не указывать область видимости класса bird, а вызвать get_mass и get_speed напрямую из animal, применив **виртуальное наследование**. Виртуальное наследование является механизмом автоматического разрешения противоречий в случае возникновения неоднозначностей при вызове функций, в случаях, когда класс является **наследником нескольких наследников общего базового класса**.

Для организации виртуального наследования нужно при наследовании перед спецификатором доступа базового класса (или после) указать слово **virtual**.

Покажем изменения в программе:

```
// наследники класса bird будут являться виртуальными наследниками animal
class bird : virtual public animal
{
private:
	double topfly; // km
public:
	bird() : animal() {}

	bird(const bird& obj)
	// данный вызов конструктора класса animal для наследников класса
	// bird проводиться не будет, вместо него будет вызываться
	// конструктор без параметров
		: animal(obj), topfly(obj.topfly) {}

	bird(const string& _title, double _mass
		, double _speed, double _topfly)
	// аналогично
		: animal(_title, _mass, _speed), topfly(_topfly) {}
	~bird()
	{
		cout << "bird deleted" << endl;
	}
protected:
	void _print() const
	{
		animal::_print();
		cout << ", topfly: " << topfly;
	}
};

// наследники класса predator будут являться
// виртуальными наследниками animal
class predator : virtual public animal
{
protected:
	predator() {}
public:
	~predator() {}
	virtual bool hunt(const animal& obj) = 0;
};

class eagle : public bird, public predator
{
public:
	eagle() : bird() {}

	eagle(const eagle& obj)
		: bird(obj),
		// из-за виртуального наследования, для передачи параметров
		// в конструктор класса animal, его необходимо вызвать явно
		  animal(obj) {}

	eagle(double _mass
		, double _speed, double _topfly)
		: bird("", 0, 0, _topfly),
		// аналогично
		  animal("Орел", _mass, _speed) {}

	bool hunt(const animal& obj)
	{
		return obj.get_mass()<get_mass()
			&& obj.get_speed()<get_speed();
	}
};
```

Лабораторное задание

Для выполнения лабораторной работы необходимо:

- самостоятельно выбрать предметную область и составить ее программную модель посредством иерархии классов (если подходящая предметная область не придумывается, можно придумать на основе указанных ниже вариантов (модификации приветствуются));
- составить согласно своей предметной области (или варианту задания):

Минимум:

- программу, реализующую и демонстрирующую работу иерархии классов.

Максимум:

- программу, реализующую и демонстрирующую работу иерархии классов с применением механизмов **динамического полиморфизма** (виртуальных функций) и **абстракции** (чисто виртуальных функций), а также виртуального наследования.

Примечания: все неабстрактные классы помимо этого должны содержать:

- конструктор без параметров;
- конструктор копирования;
- параметризированный(е) конструктор(ы);
- деструктор.

Варианты

№ ПК	Задание
1, 9, 17, 25	Боеприпасы, патроны, артиллеристские орудия, винтовки.
2, 10, 18, 26	Живые существа, клетки живых существ, одноклеточные и многоклеточные существа.
3, 11, 19, 27	Пассажирский транспорт, грузовой транспорт, автомобиль, автобус, грузовик, танкер.
4, 12, 20, 28	Товар, продовольственный товар, промышленный товар, бытовая техника, промышленная техника.
5, 13, 21, 29	Прибор, потребитель электроэнергии, холодильник, электродвигатель, аккумулятор электроэнергии.
6, 14, 22, 30	Звуковая аппаратура, звуковоспроизводящая аппаратура, звуковоспроизводящая аппаратура, кассетный магнитофон, проигрыватель компакт-дисков.
7, 15, 23, 31	Корабль, грузовой корабль, пассажирский корабль, парусник, пароход.
8, 16, 24, 32	Объект компьютерной игры, мифическое животное, человекоподобный персонаж, волшебный персонаж, волшебный предмет.

В данной разработке частично использованы материалы из лабораторных работ И.В. Ашариной.