ПРАКТИЧЕСКАЯ РАБОТА №6
РЕШЕНИЕ ЗАДАЧ НА НАСЛЕДОВАНИЕ
В ЯЗЫКЕ С++

ЦЕЛЬ ПРАКТИЧЕСКОЙРАБОТЫ:

Целью данной практической работы является приобретение практических навыков использования принципа ООП – наследования для разработки программ на языке программирования С++.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:

В объектно-ориентированном программировании часто возникает необходимость создавать новые классы, структура и поведение которых расширяют структуру и поведение уже имеющегося класса.

Для решения таких задач в объектно-ориентированном программировании существует механизм наследования. При создании нового класса, расширяющего старый, в определении нового класса указывается, что он наследует от старого. Это означает, что в новый класс автоматически попадают поля и методы старого класса, и в теле нового класса нам нужно указывать только новые поля и методы, которых не было в старом классе. Эта возможность повторного использования программного кода весьма удобна и упрощает разработку программ.

Обычно, в связи с наследованием, которое представляет собой отношение между классами, используется следующая терминология: если класс D наследует от класса B, то говорят, что класс B — базовый для D, а D — производный от B.

Кроме повторного использования фрагментов программы, механизм наследования в объектно-ориентированных языках программирования традиционно предоставляет программисту еще одну возможность упростить процесс программирования. Эта возможность называется «полиморфизм» и позволяет производить одинаковые по смыслу действия над объектами разных классов при помощи одних и тех же выражений.

Для иллюстрации наследования, в качестве примера, рассмотрим задачу создания двух классов, первый из которых описывает автомобиль, а второй — автобус.

Автомобиль имеет координаты своего положения и угол, описывающий направление движения. Он может быть изначально поставлен в любую точку с любым направлением (конструктор), может проехать в выбранном направлении определенное расстояние и может повернуть (сменить текущее направление на любое другое). Текст класса «Автомобиль» приведен ниже.

```
1) class Car { double x, y, dir;
2)     Car(double xx, double yy, double d);
3)     void move(double len);
4)     void setDir(double nd);
5) };
6) Car::Car(double xx, double yy, double d):
7)     x(xx), y(yy), dir(d) {}
8) void Car::move(double len)
9) {
10)      x += len*cos(dir);
11)      y += len*sin(dir);
12)  }
13)  void Car::setDir(double nd);
14)  { dir = nd; }
```

Следующий класс будет описывать автобус. Кроме того, что имеется у автомобиля, у автобуса должны быть поля, содержащие число пассажиров и количество полученных денег, изначально равные нулю. Также, должны быть методы «войти» и «выйти», изменяющие число пассажиров. Наконец, метод move должен быть изменен (что допустимо), чтобы увеличивать количество денег в соответствии с количеством пассажиров и проезжаемым расстоянием.

```
1) class  Bus : Car{ int people;
2)     double money;
3)     Bus(double xx, double yy, double d);
4)     void move(double len);
5)     void enter(int n);
```

```
6)      void exit(int n); };
7) Bus::Bus(double xx, double yy, double d):
8)      Car(xx,yy,d), people(0), money(0) {}
9) void Bus::move(double len)
10) {
11)     Car::move(len);
12)     money += people*len;
13) }
14) void Bus::enter(int n)
15) {
16)      if(n>=0) people += n;
17) }
18) void Bus::exit(int n)
19) {
20)      if(n>=0)
21)          if(n>=people)
22) }
23)          people = 0;
24)      else
25)          people -= n;
```

Здесь в конструкторе используется инициализатор для базового класса. Он выглядит как имя_базового_класса(параметры) и позволяет инициализировать поля, унаследованные от базового класса. Конечно, в данном случае мы могли бы просто присвоить им значения в теле конструктора, однако если поля базового класса определены в нем как private, то использование инициализатора — единственный способ инициализировать поля базового класса. Конструкторы базовых классов вызываются до конструкторов полей, в том порядке, в котором базовые классы перечислены в определении производного класса.

В методе move производного класса используется вызов того же метода, но относящегося к базовому классу. Такой синтаксис тоже часто используется, поскольку часто бывает так, что метод базового класса выполняет почти всю необходимую работу, к которой производный класс должен добавить лишь небольшую часть. В этом случае, чтобы не выписывать заново весь текст метода базового класса, можно сослаться на него.

Теперь обратимся более пристально к вопросу о полиморфизме. Полиморфизм в C++ может обеспечиваться разными

механизмами, здесь мы рассмотрим тот вариант полиморфизма, который обеспечивается наследованием. Этот вариант полиморфизма связан со следующими обстоятельствами: во-первых, указатель, тип которого в соответствии с его описанием является указателем на базовый класс, вполне может указывать в действительности на объект производного класса (правда, только при public-наследовании). Это возможно потому, что объект производного класса так же, как и объект базового, имеет все те открытые поля и методы, которые имеет базовый класс. Во-вторых, в производном классе можно переопределять, т. е. заново, по-другому, реализовывать некоторые из методов базового класса. И если указатель на базовый класс указывает на объект производного класса, и через него вызывается метод, переопределенный в производном классе, то есть возможность сделать так, чтобы вызвана была именно та версия метода, которая находится в производном классе, т. е. та версия, которая соответствует реальному объекту, на который указывает указатель.

Такое поведение (вызов правильной версии метода) называется диспетчеризацией. Чтобы ее использовать, нужно объявить соответствующий метод базового класса виртуальным, поставив перед его заголовком ключевое слово virtual. Может возникнуть вопрос: а зачем вообще нужны невиртуальные методы? Ответ дает один из принципов построения языка C++: не пользуешься — не плати. Дело в том, что наличие у класса виртуальных методов приводит к увеличению, правда, незначительному, размера объекта соответствующего класса. Так что если у класса есть хотя бы один виртуальный метод, размер объекта этого класса будет больше, чем если бы их не было.

В C++ виртуальными бывают также деструкторы; это нужно для того, чтобы динамический объект производного класса можно было правильно уничтожить при помощи операции delete, в которую подставлен указатель на базовый класс, на самом деле указывающий на объект производного класса.

Диспетчеризация позволяет иметь так называемые абстрактные базовые классы, у которых некоторые методы вовсе не реализованы. Такие методы должны иметь вместо тела текст = 0;. Они называются чисто виртуальными. Объект такого класса не может быть определен в программе, однако иметь указатели на такой класс можно; они будут

указывать всегда на объекты производных классов, у которых все методы, в том числе и унаследованные от базовых классов, реализованы.

До сих пор речь шла об указателях, но все то же самое справедливо и для ссылок, поскольку внутри любой ссылки на самом деле спрятан указатель на ту переменную, синонимом которой ссылка является. А вот без ссылок или указателей такой полиморфизм, увы, не работает. Обычная переменная (не указатель и не ссылка), если она имеет в качестве типа базовый класс, всегда относится только к базовому классу.

В качестве примера использования полиморфизма рассмотрим следующую задачу.

Написать иерархию классов, описывающих служащих в компании. Она должна состоять из абстрактного базового класса Employee и производных от него классов Manager, Agent и Worker.

Базовый класс должен иметь чисто виртуальный метод рассчета заработной платы, переопределенный в каждом из производных классов.

Заработная плата Manager постоянна и равна 13000, заработная плата Agent определяется как оклад 5000 + 5% объема продаж, который хранится в специальном поле класса Agent, и заработная плата Worker определяется как 100×число отработанных часов, которое также хранится в отдельном поле (классы Agent и Worker должны иметь конструкторы по вещественному числу, устанавливающие значение своего поля).

В основной программе завести массив из 9 указателей на Employee и заполнить его указателями на объекты производных классов (первые 3 — Manager, следующие 3 — Agent и последние 3 — Worker). Вывести на экран величину заработной платы всех 9 служащих.

Для начала опишем класс Employee.

```
1) class Employee
2) {
3)     virtual ~Employee() {}
4)     virtual double salary() = 0;
5) };
```

Теперь опишем класс Manager. Поскольку зарплата Manager постоянна, никакие поля здесь не нужны.

```
1) class Manager : Employee
2) {
3)     double salary()
4)     {
5)         return 13000;
6)     }
7) };
```

Затем опишем класс Agent. Поскольку для вычисления его зарплаты требуется знать объем продаж, что является характеристикой Agent, то уместно завести соответствующее поле в классе Agent.

```
1) class Agent : Employee
2) {
3)     double amount;
4)     Agent(double a): amount(a) {}
5)     double salary()
6)     {
7)         return 5000+0.05*amount;
8)     }
9) };
```

Наконец, опишем класс Worker. Он будет иметь поле time, содержащее число отработанных часов, на основании значения которого вычисляется его зарплата.

```
1) class Worker : Employee
2) {
3)     double time;
4)     Worker(double t): time(t) {}
5)     double salary()
6)     {
7)         return 100*time;
8)     }
9) };
```

После всех необходимых нам классов напишем основную программу. Вспомним, что она представляет собой функцию main.

```
1) class Worker : Employee
2) {
3)     double time;
4)     Worker(double t): time(t) {}
5)     double salary()
6)     {
7)         return 100*time;
8)     }
9) };
```

Полиморфизм здесь проявляется в двух последних строках тела программы, где одно и то же выражение используется для вычисления зарплаты всех сотрудников, независимо от их должности (Manager, Agent или Worker), а также динамические переменные уничтожаются при помощи одинаковых вызовов delete.

Полиморфизм оказывается очень удобен для программирования графического интерфейса пользователя.

ВАРИАНТЫ ЗАДАНИЙ

1. Написать иерархию классов, описывающих имущество налогоплательщиков. Она должна состоять из абстракт- ного базового класса Property и производных от него классов Appartment, Car и CountryHouse. Базовый класс должен иметь поле worth (стоимость), конструктор с одним параметром, заполняющий это поле, и чисто виртуальный метод рассчета налога, переопределенный в каждом из производных классов. Налог на квартиру вычисляется как 1/1000 ее стоимости, на машину — 1/200, на дачу — 1/500. Также, каждый производный класс должен иметь конструктор с одним параметром, передающий свой параметр конструктору базового класса. В функции main завести массив из 7 указателей на Property и заполнить его указателями на динамические объекты производных классов (первые 3 —

Appartment, следующие 2 — Car и последние 2 — CountryHouse). Вывести на экран величину налога для всех 7 объектов. Не забудьте также уничтожить динамические объекты перед завершением программы.

2. Написать набор классов, представляющий выражения. В этом наборе должен быть один абстрактный базо- вый тип, а также набор производных от него типов по видам выражений (константа, переменная, сумма, разность, произведение, частное, sin, cos, exp, ln). У каждого из классов должны быть следующие виртуальные функции: напе- чатать выражение (без параметров), вычислить выражение (параметр — значение переменной, результат — значение выражения), вернуть производную выражения (без параметров), создать копию выражения (тоже без параметров).