Лабораторная работа №8

Полиморфизм

### 

**Цель работы:** Целью лабораторной работы является получение практических навыков при использовании полиморфизма в языке С++.

**Задание:** Построить иерархию классов в соответствии с вариантом задания. Построить диаграммы классов.

1. Создать абстрактный базовый класс **уравнения** с виртуальной функцией *печать корней уравнения*. Создать производные классы: **линейное уравнение** и **квадратное уравнение**. Определить функцию вычисления корней уравнений. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.  
  
2. Создать абстрактный базовый класс **фигура** с виртуальной функцией – *печать объёма фигуры*. Создать производные классы: **параллелепипед**, **пирамида**, **тетраэдр**, **шар** со своими функциями печати объема. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.  
  
3. Создать абстрактный базовый класс **фигура** с виртуальной функцией – *вычисление площади фигуры*. Создать производные классы: **прямоугольник**, **круг**, **прямоугольный треугольник**, **трапеция** со своими функциями вычисления площади. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.  
  
4. Создать абстрактный базовый класс **фигура** с виртуальной функцией – *печать площади поверхности*. Создать производные классы: **параллелепипед**, **тетраэдр**, **шар** со своими функциями печати площади поверхности. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов, а затем печатаются площади всех фигур, входящих в список.   
  
5. Создать абстрактный базовый класс **прогрессия** с виртуальной функцией – *вычисление суммы прогрессии*. Создать производные классы: **арифметическая прогрессия** и **геометрическая прогрессия**. Каждый класс имеет два поля типа double: первый член прогрессии и постоянная разность (для арифметической) и постоянное отношение (для геометрической). Определить функцию вычисления суммы, где параметром является количество элементов прогрессии.  
  
6. Создать абстрактный класс – **млекопитающее**, с чисто виртуальной функцией info(), которая возвращает строку с описанием объекта. Определить производные классы - **животное** и **человек**. У животных определить производные классы **собака** и **корова**. Определить виртуальные функции описания человека, собаки и коровы. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов, а затем печатается информация обо всех млекопитающих из списка.  
  
7. Создать абстрактный класс **кривая** с виртуальной функцией *вычисления координаты* y для некоторой x. Создать производные классы: **прямая**, **эллипс**, **гипербола** со своими функциями вычисления y в зависимости от входного параметра x. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов, а затем печатается информация обо всех конкретных элементах списка.  
  
8. Создать абстрактный класс **структура данных** с виртуальной функцией: *норма*. Создать производные классы: **комплексные числа**, **вектор** из 10 элементов, **матрица** (2х2). Определить функцию нормы - для комплексных чисел - модуль в квадрате, для вектора - корень квадратный из суммы элементов по модулю, для матрицы - максимальное значение по модулю. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов, а затем печатается информация обо всех конкретных элементах списка.  
  
9. Создать абстрактный базовый класс -  **Предок**, с строковым параметром *имя*. определить виртуальную функцию печати имени. Создать производный класс **Ребенок**, у которого функция печати дополнительно к имени предка выводит имя ребёнка. Создать производный класс от последнего класса - **Внук**, у которого есть отчество. Написать для него свою функцию печати. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов, а затем печатается информация обо всех конкретных элементах списка.  
  
10. Создать абстрактный базовый класс - **работник** и производные классы - **служащий с почасовой оплатой**, **служащий в штате** и **служащий с процентной ставкой**. Определить функцию начисления зарплаты. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов, а затем печатается информация обо всех конкретных элементах списка.  
  
11. Создать абстрактный базовый класс **фигура** с виртуальной функцией – *возврат количества углов в фигуре*. Создать производные классы: **параллелепипед**, **пирамида**, **тетраэдр**, **треугольник** со своими функциями. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.  
  
12. Создать абстрактный базовый класс **список**. Реализовать на его базе классы **стек** и **очередь** с виртуальными функциями добавления и удаления элементов.  
  
13. Создать абстрактный класс **объект мироздания**. Определить наследуемые классы - **лиса**, **кролик** и **трава**. Лиса может “съесть” кролика. Кролик “ест” траву. Лиса и кролик могут “умереть” - определен их возраст. Кроме этого определен класс - **отсутствие жизни**. Если в окрестности имеется больше травы, чем кроликов, то трава остается, иначе трава съедена. Если лис слишком старый он может умереть. Если лис слишком много (больше 5 в окрестности), лисы больше не появляются. Если кроликов меньше лис, то лис ест кролика.   
  
14. Создать абстрактный класс **человек**, и производные от него **девушка** и **юноша**. Определить виртуальную функцию реакции человека на вновь увиденного другого человека. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.   
  
15. Создать абстрактный базовый класс **данные**. Создать производные классы **сигнал**, **результат обработки** и **вспомогательные данные**. Все данные имеют виртуальные функции отображения, сохранения и обработки.  
  
16. Поступающие звонки оператор классифицирует (т.е. в какой город, регион, страну и т.д. поступил звонок), заносит соответствующие данные и добавляет запись к абоненту. Необходимо реализовать унифицированные функции подсчета общего времени и стоимости как для абонента (звонящий) так и для всех абонентов, выдать на экран специфическую информацию о вызовах. Для этого реализовать абстрактный базовый класс *звонок* с виртуальной функцией *стоимость*, и наследники этого класса **входящий звонок** и **исходящий звонок.**  
  
17. Реализовать распознавание объектов из некоторого множества (расширяемого) на основе вопросов и ответов. Для этого реализовать абстрактный базовый класс **объект распознавания** и 2-3 его наследника. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.  
  
18. Реализовать обработку сообщений и информацию о типе в базовом классе. Создать производные классы. При отправке сообщений проверять существование объекта. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.  
  
19. Создать абстрактный базовый класс **data** и производные **целое**, **вещественное**, **строка** для чтения и записи информации, как в текстовом, так и в бинарном формате. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.

**Виртуальные функции. Полиморфизм**

При наследовании классов существует возможность переопределять функции базовых классов – в производном классе допустимо определить функцию с тем же именем и сигнатурой, что и в базовом, уточняя или изменяя выполняемые действия. Рассмотрим пример.

//Листинг 8. Проблема статического связывания функций

class Base

{ public:

int func1(int x) {return x\*x;}

int func2(int x){return func1(x)/2;}

};

class Child: public Base

{public:

int func1(int x) {return x\*x\*x;}

};

main()

{ Child c;

cout<<c.func2(5); //на экран выводится 12

Base \*ptb=new Child;

cout<<ptb->func1(2); //на экран выводится 4

cout<<c.func1(2); // на экран выводится 8

}

В классе *Base* определены две функции *func1* и *func2*, причем вторая функция вызывает первую. В классе *Child* переопределена функция *func1,* а функция *func2* просто наследуется. При вызове функций результаты их работы оказываются для многих неожиданными. Так, вызов c.func2(5) дает результат 12 вместо ожидаемых 62, а вызов ptb->func1(2) дает результат 4, а не 8. Дело в том, что в обоих случаях будет вызвана функция *func1* базового класса, а не переопределенная в производном классе. Такое поведение объектов связано со **статическим (ранним) связыванием** функций при трансляции программы. Когда транслятор в процессе обработки программы встречает вызов какой-либо функции, то на место вызова он подставляет в текст оттранслированной программы адрес вызываемой функции. Таким образом, компилируя тело компонентной функции *func2* класса *Base*, транслятор на место вызова функции *func1* подставит адрес компонентной функции *func1* из класса *Base*, так как только эта функция с подобным именем ему известна (содержимое класса *Child* транслируется позже). В итоге функция *Base*::*func2* всегда будет вызывать функцию *Base*::*func1,* как бы ни был оформлен вызов самого метода *func2.*

Аналогично, компилируя тело функции *main* и встретив вызов ptb->func1(2), транслятор должен подставить на место вызова адрес функции, которой будет передано управление в данной точке программы. К этому моменту транслятору известны две функции с именем *func1*: *Base*::*func1* и *Child*::*func1.* Так как вызов метода осуществляется для указателя на объект *Base*, транслятор подставит на место вызова адрес функции именно этого класса (определить, что в указатель *ptb* записан адрес объекта класса *Child,* и поэтому вызвать метод *Child*::*func1,*  транслятор не может).

Таким образом, можно сформулировать проблему: необходимо обеспечить полиморфное поведение некоторой функции, определенной в базовом классе и переопределенной в производных. Под полиморфным поведением понимается реализация функции таким образом, чтобы всякий раз при вызове функции вызывалась именно та ее реализация, которая отражает поведение того объекта, для которого осуществлялся вызов. Полиморфное поведение функций обеспечивается **поздним связыванием** функций, когда определение того экземпляра одноименных функций, которому будет передано управление, определяется на этапе выполнения программы.

Для того, чтобы компонентная функция обеспечивала полиморфное поведение, ее необходимо объявить **виртуальной**. Для этого в определении функции в классе необходимо указать ключевое слово **virtual**.

*virtual тип имя\_функции* (*список\_формальных параметров*)

{*тело функции* }

Если мы изменим определение функции *func1* , объявив ее виртуальной, поведение объектов программы изменится.

//Листинг 9. Использование виртуальных функций

class Base

{ …

virtual int func1(int x) {return x\*x;}

};

class Child: public Base

{…

virtual int func1(int x) {return x\*x\*x;}

};

main()

{ Child c;

cout<<c.func2(5); //на экран выводится 62

Base \*ptb=new Child;

cout<<ptb->func1(2); //на экран выводится 8

}

В языке С++ позднее связывание реализуется путем поддержки для каждого объекта **таблицы виртуальных функций**. Таблица виртуальных функций представляет собой массив указателей на реализации виртуальных функций, доступные для данного объекта. Структура объекта при использовании им виртуальных функций, изображена на рис.6.

|  |
| --- |
| Указатель на таблицу виртуальных методов (vtbl) |
| Компонентные данные объекта |
| Адрес виртуальной функции 1 (&func1) |
| Адрес виртуальной функции 2 (&func2) |
| … |
| Адрес виртуальной функции n (&funcN)  Рисунок 6. Структура объекта, использующего виртуальные функции |

Вызов виртуальной функции в тексте программы транслятор преобразует в обращение к соответствующей строке таблицы виртуальных функций. Предположим, что *pObj* – это указатель, в который записан адрес объекта, структура которого отображена на рисунке 6. В таком случае вызов виртуальной функции *func2* для этого объекта вида

pObj->func2()

преобразуется транслятором в следующий вызов:

(\*(pObj->vptr[1])) (pObj)

При таком вызове функции нет жесткой привязки вызова к какой-то конкретной реализации компонентной функции в одном из классов, как это происходило при статическом связывании. Теперь будет вызываться та функция, адрес которой записан в элемент с индексом *1* таблицы виртуальных функций объекта. Позднее связывание обеспечивается тем, что заполнение таблицы виртуальных функций объекта происходит уже на этапе выполнения программы. Это делает конструктор, занося в каждую строку таблицы адрес той реализации виртуального метода, который правильно описывает поведение объекта. Указатель на таблицу виртуальных функций обязательно включается в самый "верхний" базовый фрагмент объекта производного класса. В таблицу указателей включаются адреса функций-членов фрагмента самого "нижнего" уровня, содержащего объявления этой функции. Такая дополнительная функциональность конструктора обеспечивается транслятором.

Использование позднего связывания не отрицает возможности вызова из производного класса экземпляра виртуальной функции базового. Просто для подобного использования необходимо указывать при вызове полное квалифицированное имя функции. Пример для программы из листинга 9:

main()

{

Base \* ptb=new Child;

cout<<ptb->Base::func1(2); //на экран выводится 4

}

При определении виртуальных функций существует ряд синтаксических особенностей. Если какая-то функция определена в базовом классе как виртуальная, а в производном классе переопределена с тем же прототипом (то есть в производном классе в точности совпадают тип, имя и список параметров компонентной функции), то в производном классе ключевое слово *virtual* можно опустить – функция останется виртуальной по умолчанию. Если сигнатура (список формальных параметров) функции в производном классе изменится, то полиморфное поведение функций базового и производного классов станет невозможным. Если же в производном классе будет определена компонентная функция с тем же названием и сигнатурой, но с другим типом возвращаемого значения, то транслятор выдаст сообщение об ошибке. Виртуальной функцией может быть только нестатическая компонентная функция класса.

Как уже было отмечено выше, основной смысл полиморфного определения функций – переопределить в производном классе их поведение в соответствии с его особенностями. Однако, такое определение зачастую приводит к тому, что действие, соответствующее полиморфной функции, в базовом классе неопределенно в предметной области решаемой задачи. Например, создадим иерархию классов для геометрических фигур:

//Листинг 10. Иерархия классов для геометрических фигур

class Point

{ public:

int x,y;

int color;

};

class Shape

{ protected:

// base - точка привязки фигуры

// на плоскости

Point base;

int color;

public:

virtual void show()

{ }

virtual void hide()

{ }

void move(int xn,int yn)

{ hide()

base.x+=xn; base.y+=yn;

show();

}

};

class Circle:public Shape

{ int radius;

public:

void show()

{/\*рисуем окружность c центром в точке base\*/}

void hide()

{/\*скрываем окружность\*/}

};

class Rectangle:public Shape

{//высота и ширина прямоуг-ка

int width,height;

public:

void show()

{/\*рисуем прямоугольник с левым верхним углом в base\*/}

void hide()

{/\*скрываем прямоугольник\*/}

};

Классы *Окружность* (*Circle*) и *Прямоугольник* (*Rectangle*) унаследованы от класса *Фигура* (*Shape*). В классе *Shape* обобщены все общие характеристики и поведение геометрических фигур (цвет, положение базовой точки на плоскости, метод перемещения фигур). В частности, для того, чтобы переместить фигуру, необходимо методом *hide* скрыть ее в текущей позиции, переместить координаты базовой точки на указанные смещения и отобразить фигуру в новой позиции методом *show*. Именно это и происходит в методе *move* класса *Shape*. Теперь, если вызвать метод *move* для экземпляра класса *Circle*, то, учитывая полиморфность объявления методов *hide* и *show*, окружность будет корректно переноситься и отображаться:

Circle c(100, 100, 10); //задаем в конструкторе координаты

//центра (базовой точки) и радиуса

c.move(-10, 10);

В структуре класса *Shape* обращает на себя внимание присутствие методов *hide* и *show*. Сам объект *Shape* является программной абстракцией, позволяющей сократить объем описания производных классов, для него не требуется операций отображения и скрытия, тем более, что как объекты этого класса в программе создаваться не будут. Однако, обойтись вообще без методов *hide* и *show* в этом классе нельзя – к ним обращается метод *move*. Здесь возникает противоречие между семантикой и синтаксисом определения класса – методы должны присутствовать в классе, но для них нет полезного наполнения, реально будут вызываться только их полиморфные переопределения в производных классах. В листинге 10 определения функций *hide* и *show* упрощены до пустого тела. Для того, чтобы определение класса *Shape* полностью отвечало его семантике, эти функции лучшее определить как *чистые виртуальные*.

Необходимо также отметить, что для исключения возможных утечек памяти деструкторы классов, использующих хотя бы одну виртуальную функцию также необходимо объявлять виртуальными.

Чистая виртуальная функция определяется следующим образом:

*virtual тип имя\_функции*( *список\_формальных\_параметров* )=0;

Чистая виртуальная функция не имеет реализации, ее нельзя вызвать в программе, она служит лишь как основа для дальнейшего полиморфного переопределения в производном классе. Если в классе определена хотя бы одна чистая виртуальная функция, он становится *абстрактным*. Главное отличие абстрактных классов – на их основе невозможность создавать объекты, они могут служить только основой для наследования. Класс *Shape* по замыслу является абстрактным, поэтому его можно переопределить следующим образом:

class Shape

{ protected:

Point base;

int color;

public:

virtual void show()=0;

virtual void hide()=0;

void move(int xn,int yn)

{ hide()

base.x+=xn; base.y+=yn;

show();

} };