ЛАБОРАТОРНАЯ РАБОТА №7.

Виртуальные функции.

Цель работы: Знакомство с понятиями динамического полиморфизма и позднего связывания в ООП; изучение механизма виртуальных функций C++; приобретение навыков использования виртуальных функций при проектировании и разработке программ.

Теоретические сведения.

7.1) Указатели и ссылки на базовые и производные классы.

Прежде, чем приступить к изучению собственно виртуальных функций, рассмотрим вопрос, связанный с наследованием и использованием указателей и ссылок на пользовательские типы данных – классы.

Хорошо известно, что в общем случае указатель одного типа не может указывать на объект другого типа. Например, под указателем типа int может размещаться только переменная типа int, но не может размещаться переменная типа float, char, bool и т.д. Это правило справедливо почти всегда, однако не распространяется на классы, находящиеся в отношении «предок - потомок». Это исключение формулируется так: **в языке C++ указатель на базовый класс может указывать на объект производного класса**.

Исключение это связано тем, что потомок наследует все характеристики и поведение своего предка (поля и методы), то есть потомок фактически *является* своим предком. Мы можем утверждать, например, что парнокопытное является млекопитающим, а млекопитающее, в свою очередь, является животным. Аналогичным образом, в C++ между объектами базового и производного класса существует более тесная связь, чем между неродственными объектами.

Рассмотрим следующий пример. Пусть `book` – базовый класс книги, содержащий сведения о названии книги, ее авторах и т.д., а `electronic_book` – производный от него класс электронной книги, содержащий дополнительно информацию о формате книги (fb2, epub, djvu и др.) и объеме файла в килобайтах. Предположим далее, что в программе имеются следующие объявления

```
book *pointer;              // объявляем указатель на объект класса book
book mybook1;               // создаем объект базового класса
electronic_book mybook2;    // создаем объект производного класса
```

Тогда корректными с точки зрения компилятора будут следующие операции

```
pointer = &mybook1;    // здесь указатель pointer устанавливается на объект book
pointer = &mybook2;    // а здесь он указывает на объект electronic_book
```

Правильнее будет считать, что во втором случае `pointer` указывает не на объект весь электронной книги, а только на его внутреннюю часть, унаследованную от `book`. Таким образом, указатель `pointer` может дать нам доступ к полям и методам базового класса, унаследованным всеми его потомками.

Допустим, метод `display()` базового класса `book` выводит на экран значения полей объекта, то есть название книги, ее авторов и год выпуска. Тогда пара операторов

```
pointer = &mybook1;
pointer->display();
```

выведет на экран информацию об объекте базового класса с именем `mybook1`. Применим теперь аналогичную процедуру к объекту производного класса

```
pointer = &mybook2;
pointer->display();
```

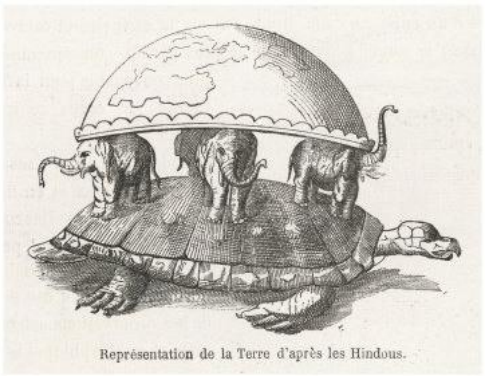
В этом случае на экран будут выведены сведения об электронной книге, содержащиеся в объекте `mybook2`. Однако выведены будут только унаследованные поля, тогда как дополнительные данные (формат и размер файла в Кб) на экран выводиться не будут, так как в данном случае через `pointer` мы можем вызвать только методы базового класса.

Отметим здесь следующее. Мы рассмотрели ситуацию, когда указатель на базовый класс используется в качестве указателя на производный объект. Обратное действие не возможно, то есть указатель на производный класс не может использоваться для доступа к объектам базового типа.

Рассмотренные особенности использования указателей в C++ применимы также к ссылкам. То есть, ссылки на базовый класс могут также использоваться как ссылки на производный тип. Эта техника нередко применяется при работе с функциями. А именно, если какой-либо параметр функции является ссылкой на базовый класс, то функция может принимать ссылки как на объект базового класса, так и на объекты производных классов.

7.2) Динамический полиморфизм и позднее связывание.

Динамический полиморфизм является одной из трех ключевых парадигм объектно-ориентированного программирования, наряду с инкапсуляцией и наследованием (см. рис. слева). Говоря кратко, динамический полиморфизм – это способность объекта вызывать методы производного класса, который может *даже не существовать* на момент создания базового класса.


Représentation de la Terre d'après les Hindous.

Что это означает в реальности? Предположим, что программист разрабатывает класс `Животное`, который должен выступить в качестве прародителя для производных классов, таких как `Млекопитающее`, `Птица`, `Земноводное`, `Рыба`, и т.д. Одна из характерных особенностей животного – это его способность передвигаться, поэтому логичным было бы определить специальный метод `move()` для класса `Животное`. Этот метод будет унаследован производными классами, то есть по умолчанию млекопитающее, птица, рыба и земноводное будут перемещаться в пространстве одинаково. Вместе с тем мы знаем, что различные виды животных используют разные способы передвижения – млекопитающие бегают, птицы летают, рыбы плавают и т.д. Для программиста C++ это означает, что метод `move()` необходимо переопределить в большинстве производных классов.

Предположим теперь, что в программе имеется указатель `pointer` на объект класса `Животное`. В соответствии с правилами, рассмотренными в предыдущем пункте, `pointer` может указывать не только на животное, но и на экземпляр любого производного класса, то есть на млекопитающее, птицу, рыбу и т.д. Более того, под указателем `pointer` можно

разместить объект нового класса, который даже не был разработан на момент создания класса животного. Вызывая метод `move()` следующим образом

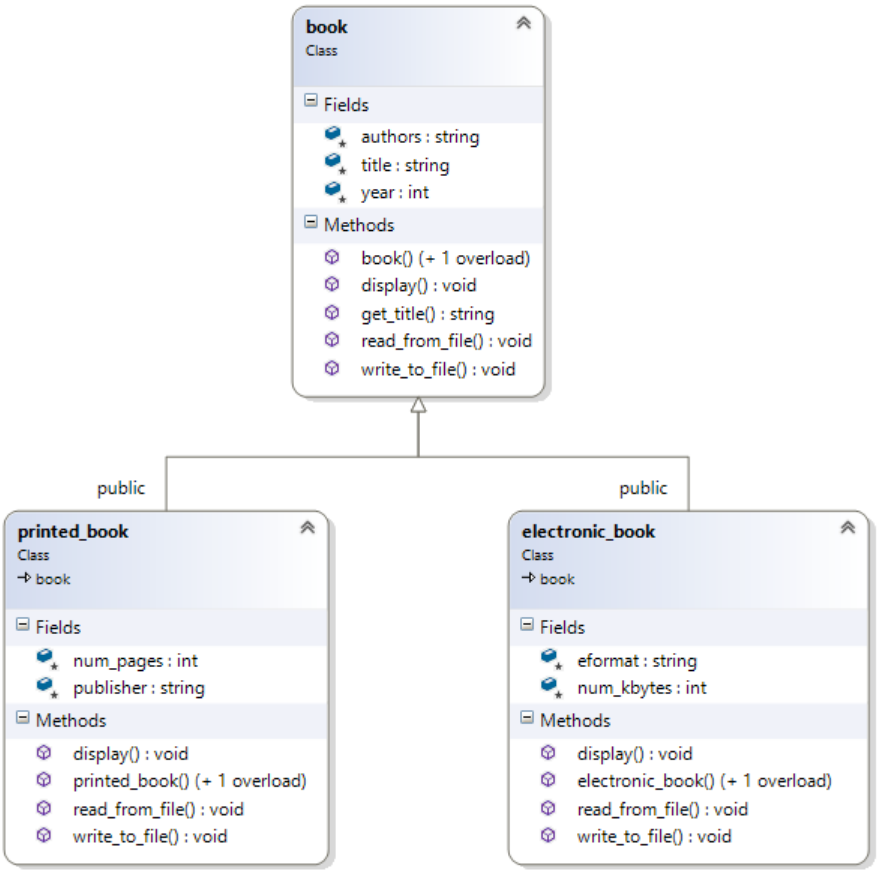
```
pointer->move();
```

программист, вообще говоря, не знает, какой именно способ перемещения здесь будет задействован – бег, полет, плавание и т.д.

Таким образом, на момент редактирования и даже компиляции кода C++ вызов метода `move()` еще не связан с конкретной функцией. Эта связь появляется только на этапе выполнения программы, когда становится известно, какой именно объект располагается под указателем `pointer`. Этот тип связывания называется *поздним*, в отличие от *раннего* связывания, когда выбор конкретной функции происходит на этапе компиляции программы.

Позднее связывание дает программисту огромное преимущество – он может выполнять над объектами операции, даже не предполагая о том, как эти операции реализованы (в том числе, будут реализованы в будущем).

Вернемся теперь к приложению «Книжный магазин». Перепроектируем систему классов следующим образом. Пусть `book` – базовый класс книги, а `printed_book` и `electronic_book` – производные классы, описывающие печатную книгу и электронную книгу, соответственно. Каждый из производных классов задает несколько специфических полей: для печатной книги указывается издательство и число страниц, для электронной – формат файла и его объем в килобайтах. Получаем следующую иерархию классов



Как видно из диаграммы, производные классы переопределяют метод `display()` базового класса (вывод на экран содержимого полей), а также методы `read_from_file()` и `write_to_file()` (чтение из файла и запись в файл).

Нашей следующей задачей будет создание единой коллекции книг, включающей разнотипные издания (электронные, печатные, или другие издания, являющиеся наследниками от класса `book`). Объявим массив указателей на объекты базового класса и выделим память под несколько элементов

```
book** p_books;              // массив указателей на объекты book
p_books = new book*[5];      // выделяем память под 5 указателей
```

Заполним массив, используя указатели как на объекты `book`, так и на объекты производных классов (см. п. 7.1)

```
p_books[0] = new printed_book("Дети капитана Гранта", "Верн Ж.", 2003, "Эксмо", 800);
p_books[1] = new electronic_book("Эффективный C++", "Мейерс С.", 2014, "djvu", 35);
p_books[2] = new book("Алиса в Зазеркалье", "Кэрролл Л.", 1871);
p_books[3] = new printed_book("Алгебра", "Виноградов И.М.", 1977, "Сов. энц.", 412);
p_books[4] = new electronic_book("Бесы", "Достоевский Ф.М.", 1866, "epub", 21);
```

Теперь массив содержит объекты разных классов. Посмотрим далее, сможем ли мы обрабатывать эти разнородные данные, используя общий интерфейс. Например, выведем на экран содержимое всей коллекции, вызывая последовательно метод `display()` для всех элементов массива

```
for (int i = 0; i < 5; i++)
        p_books[i]->display();
```

Результаты работы цикла, скорее всего, нас разочаруют: на экран будут выведены только внутренние (унаследованные) поля, то есть название книги (title), авторы (authors) и год публикации (year). Дополнительная информация из производных классов отображена не будет. Фактически вызов функции `display()` будет транслирован в вызов функции `book::display()`. Причиной является то, что к настоящему моменту мы не задействовали механизма позднего связывания.

7.3) Виртуальные функции.

Динамический полиморфизм (полиморфизм времени исполнения) в языке C++ реализуется через использование виртуальных функций. Виртуальная функция – это функция, объявленная в базовом классе с ключевым словом virtual, и затем переопределенная в одном или нескольких производных классах.

Вызов виртуальной функции обрабатывается особым способом. В процессе вызова виртуальной функции через указатель или ссылку на объект базового класса вначале происходит определение фактического типа объекта (к какому из производных классов относится объект), а затем происходит вызов соответствующего варианта функции. В частности, в рассмотренном нами примере для объектов класса `printed_book` будет вызвана функция `printed_book::display()`, а для объектов класса `electronic_book` – функция `electronic_book::display()`.

Общий синтаксис объявления виртуальной функции следующий

```
class имя_класса
{
public:
    virtual тип имя_функции(параметры);
};
```

При переопределении виртуальной функции в производном классе ключевое слово virtual можно не указывать – компилятор автоматически сделает ее виртуальной.

Задействуем динамический полиморфизм в примере с книжной коллекцией. Для этого объявим метод `display()` базового класса `book` виртуальным

```
class book
{
protected:
    string title;
    string authors;
    int year;

public:
    ...
    virtual void display();
    ...
};
```

и реализуем этот метод стандартным способом

```
void book::display()
{
    cout << "Название: " << title << endl;
    cout << "Автор(ы): " << authors << endl;
    cout << "Год     : " << year << endl;
}
```

При объявлении производного класса `printed_book` переопределим метод `display()` и также сделаем его виртуальным

```
class printed_book : public book
{
protected:
    string publisher;
    int num_pages;

public:
    ...
    virtual void display();
    ...
};
```

Обратите внимание на то, как в приведенном ниже фрагменте кода унаследованная функция `book::display()` задействуется для реализации функции `printed_book::display()`

```
void printed_book::display()
{
    book::display();
    cout << "Издатель: " << publisher << endl;
    cout << "Страниц : " << num_pages << endl;
}
```

Этот код показывает, что для печатной книги мы сначала выводим на экран общую информацию (название, авторы, год), а затем дополнительную (издательство и число страниц). Аналогичным образом может быть переопределена виртуальная функция `display()` для производного класса `printed_book` и других производных классов в системе.

7.4) Фрагменты приложения Visual C++.

Создадим в рамках программного проекта «Книжный магазин» единую коллекцию разнотипных книг, включая печатные и электронные издания. Реализуем единый интерфейс работы с этими книгами. В частности, используем метод `display()` для вывода информации на экран, методы `read_from_file(ifstream&)` и `write_to_file(ofstream&)` – для чтения и записи в файл. Задействуем класс-контейнер `bookstore` для хранения книжной коллекции.

Ниже приведены фрагменты кода, расположенные в заголовочных файлах модулей books.h и bookstore.h. Функции классов реализуются стандартным способом, поэтому их код ниже не приводится.

```
======================== файл books.h ========================
```

```
#ifndef BOOKS_H
#define BOOKS_H

#include <string>
#include <iostream>
#include <fstream>

using namespace std;

class book
{
protected:
	string title;
	string authors;
	int year;

public:
	book();
	book(string tit, string aut, int yea);
	string get_title();

	virtual void display();
	virtual void read_from_file(ifstream& stream);
	virtual void write_to_file(ofstream& stream);
};

class printed_book : public book
{
protected:
	string publisher;
	int num_pages;

public:
	printed_book();
	printed_book(string tit, string aut, int yea, string pub, int pag);

	virtual void display();
	virtual void read_from_file(ifstream& stream);
	virtual void write_to_file(ofstream& stream);
};
```

```
class electronic_book : public book
{
protected:
	string eformat;
	int num_kbytes;

public:
	electronic_book();
	electronic_book(string tit, string aut, int yea, string eft, int kbs);

	virtual void display();
	virtual void read_from_file(ifstream& stream);
	virtual void write_to_file(ofstream& stream);
};

#endif
```

====================== файл bookstore.h ======================

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H

#include <string>
#include <fstream>
#include "books.h"

class bookstore
{
private:
	int max_num_books;
	int num_books;
	book **p_books;

public:
	bookstore(int mb);
	~bookstore();

	void operator +=(book *abook);

	void display();
	void read_from_file(string filename);
	void write_to_file(string filename);
	void find_book(string atitle);
};

#endif
```

==========

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №7.

1) Разработайте систему производных классов (как минимум 2), используя механизмы наследования.
2) Реализуйте хранилище объектов различных классов в контейнере через массив указателей базового класса.
3) Разработайте общий интерфейс доступа к объектам разного типа в хранилище. Задействуйте для этого механизм виртуальных функций. Реализуйте следующие функции:
   а) вывода содержимого объекта на экран,
   б) чтение данных из файла,

в) запись данных в файл,

г) поиск элемента в хранилище,

д) свою собственную функцию (по деланию студента).

4) Продемонстрируйте работу виртуальных функций в своей программе.

Содержание отчета по лабораторной работе №7.

1) Стандартная «шапка» отчета

2) Цель: формулировка цели работы

3) Теория: краткие сведения о динамическом полиморфизме в C++ и виртуальных функциях с примерами (объем 2-3 стр.)

4) Программный код

- объявление базового класса с несколькими виртуальными функциями (целиком),
- реализация одной из виртуальных функций базового класса,
- объявление производного класса с виртуальными функциями (целиком),
- реализация переопределенной виртуальной функции в производном классе,
- код основной программы (целиком).