Лабораторная работа №1

по курсу «Программирование»

«ИЕРАРХИЯ КЛАССОВ И ОБЪЕКТЫ В C++»

Цель:

Получить практические навыки реализации иерархии классов на языке C++.

Основное содержание работы:

Написать программу, в которой создаются и разрушаются объекты, определенного пользователем класса. Выполнить исследование вызовов конструкторов и деструкторов. Создать иерархию классов. Показать использование виртуальных функций.

Порядок выполнения работы:

1. Определить пользовательский класс (базовый) в соответствии с вариантом задания (смотри далее).
2. Определить в классе следующие конструкторы: без параметров, с параметрами, копирования.
3. Определить в классе деструктор.
4. Определить в классе компоненты-функции для просмотра и установки полей данных.
5. Написать демонстрационную программу, в которой создаются и разрушаются объекты пользовательского класса и каждый вызов конструктора и деструктора сопровождается выдачей соответствующего сообщения (какой объект какой конструктор или деструктор вызвал).
6. Определить иерархию классов (в соответствии с вариантом).
7. Реализовать классы.
8. Написать демонстрационную программу, в которой создаются объекты различных классов.
9. Сделать соответствующие методы невиртуальными и посмотреть, что будет.

Содержание отчета:

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи. Следует дать конкретную постановку, т.е. указать, какие классы должны быть реализованы, какие должны быть в них конструкторы, компоненты-функции и т.д.
3. Иерархия классов в виде графа.
4. Определение пользовательских классов с комментариями.
5. Реализация всех видов конструкторов и деструктора.
6. Листинг демонстрационной программы.
7. Листинг основной программы, в котором должно быть указано, в каком месте и какой конструктор или деструктор вызываются.
8. Объяснение необходимости виртуальных функций. Следует показать, какие результаты будут в случае виртуальных и невиртуальных функций.

Краткие теоретические сведения:

Класс

Класс – фундаментальное понятие C++, он лежит в основе многих свойств C++. Класс предоставляет механизм для создания объектов. В классе отражены важнейшие концепции объектно-ориентированного программирования: абстрагирование, инкапсуляция, наследование (иерархия). Одним из мощных средств разработки классов является полиморфизм.

С точки зрения синтаксиса, класс в C++ – это структурированный тип, образованный на основе уже существующих типов.

В этом смысле класс является расширением понятия структуры. В простейшем случае класс можно определить с помощью конструкции:

тип_класса имя_класса { список_членов_класса };

где

тип_класса – одно из служебных слов **class, struct, union**;

имя_класса – идентификатор;

список_членов_класса – определения и описания типизированных данных и принадлежащих классу функций:

функции – это методы класса, определяющие операции над объектом.

данные – это поля объекта, образующие его структуру. Значения полей определяет состояние объекта.

Примеры.

```
struct date                    // дата
{int month,day,year;           // поля: месяц, день, год
  void set(int,int,int);       // метод – установить дату
  void get(int*,int*,int*);    // метод – получить дату
  void next();                 // метод – установить следующую дату
  void print();                // метод – вывести дату
};
struct class complex           // комплексное число
{double re,im;
  double real(){return(re);}
  double imag(){return(im);}
  void set(double x,double y){re = x; im = y;}
  void print(){cout<<"re = "<<re; cout<<"im = "<<im;}
};
```

Для описания объекта класса (экземпляра класса) используется конструкция

имя_класса имя_объекта;

```
date today,my_birthday;
date *point = &today;          // указатель на объект типа date
date clim[30];                 // массив объектов
date &name = my_birthday;      // ссылка на объект
```

В определяемые объекты входят данные, соответствующие членам – данным класса. Функции – члены класса позволяют обрабатывать данные конкретных объектов класса. Обращаться к данным объекта и вызывать функции для объекта можно двумя способами. Первый с помощью “квалифицированных” имен:

имя_объекта.имя_данного
имя_объекта.имя_функции

Например:

```
complex x1,x2;
x1.re = 1.24;
x1.im = 2.3;
x2.set(5.1,1.7);
x1.print();
```

Второй способ доступа использует указатель на объект

указатель_на_объект –> имя_компонента

```
complex *point = &x1;  //  или  point = new complex;
point ->re = 1.24;
point ->im = 2.3;
point ->print();
```

Доступность компонентов класса:

В рассмотренных ранее примерах классов компоненты классов являются общедоступными. В любом месте программы, где “видно” определение класса, можно получить доступ к компонентам объекта класса. Тем самым не выполняется основной принцип абстракции данных – инкапсуляция (сокрытие) данных внутри объекта. Для изменения видимости компонент в определении класса можно использовать спецификаторы доступа: **public, private, protected**.

Общедоступные (public) компоненты доступны в любой части программы. Они могут использоваться любой функцией как внутри данного класса, так и вне его. Доступ извне осуществляется через имя объекта:

имя_объекта.имя_члена_класса
ссылка_на_объект.имя_члена_класса
указатель_на_объект -> имя_члена_класса

Собственные (private) компоненты локализованы в классе и не доступны извне. Они могут использоваться функциями – членами данного класса и функциями – “друзьями” того класса, в котором они описаны.

Защищенные (protected) компоненты доступны внутри класса и в производных классах.

Изменить статус доступа к компонентам класса можно и с помощью использования в определении класса ключевого слова **class**. В этом случае все компоненты класса по умолчанию являются собственными.

Пример.

```
class complex
{
  double re, im;                  // private по умолчанию
  public:
  double real(){return re;}
  double imag(){return im;}
  void set(double x,double y){re = x; im = y;}
};
```

Конструктор

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо было вызвать функцию типа set (как для класса complex), либо явным образом присваивать значения данным объекта. Однако для инициализации объектов класса в его определение можно явно включить специальную компонентную функцию, называемую **конструктором.** Формат определения конструктора следующий:

имя_класса (список_форм_параметров){операторы_тела_конструктора }

Имя этой компонентной функции по правилам языка C++ должно совпадать с именем класса. Такая функция автоматически вызывается при определении или размещении в памяти с помощью оператора `new` каждого объекта класса.

Пример.

```
complex(double re1 = 0.0,double im1 = 0.0){re = re1; im = im1;}
```

Конструктор выделяет память для объекта и инициализирует данные – члены класса.

Конструктор имеет ряд особенностей:

- Для конструктора не определяется тип возвращаемого значения. Даже тип void не допустим.
- Указатель на конструктор не может быть определен, и соответственно нельзя получить адрес конструктора.
- Конструкторы не наследуются.
- Конструкторы не могут быть описаны с ключевыми словами *virtual, static, const, mutuable, valatile.*

Конструктор всегда существует для любого класса, причем, если он не определен явно, он создается автоматически. По умолчанию создается *конструктор без параметров* и *конструктор копирования*. Если конструктор описан явно, то конструктор по умолчанию не создается. По умолчанию конструкторы создаются общедоступными (public).

Параметром конструктора не может быть его собственный класс, но может быть ссылка на него (T &). Без явного указания программиста конструктор всегда автоматически вызывается при определении (создании) объекта. В этом случае вызывается конструктор без параметров. Для явного вызова конструктора используются две формы:

имя_класса имя_объекта (фактические_параметры);

имя_класса (фактические_параметры);

Первая форма допускается только при непустом списке фактических параметров. Она предусматривает вызов конструктора при определении нового объекта данного класса:

```
complex ss (5.9,0.15);
```

Вторая форма вызова приводит к созданию объекта без имени:

```
complex ss = complex (5.9,0.15);
```

Существуют два способа инициализации данных объекта с помощью конструктора. Ранее мы рассматривали первый способ, а именно, передача значений параметров в тело конструктора. Второй способ предусматривает применение списка инициализаторов данного класса. Этот список помещается между списком параметров и телом конструктора. Каждый инициализатор списка относится к конкретному компоненту и имеет вид:

имя_данного (выражение)

Примеры.

```
class CLASS_A
{
  int i; float e; char c;
public:
  CLASS_A(int ii,float ee,char cc) : i(8),e( i * ee + ii ),c(cc){}
. . .
};
Класс "символьная строка".
#include <string.h>
#include <iostream.h>
class string
{
```

```
  char *ch; // указатель на текстовую строку
  int len;  // длина текстовой строки
public:
  // конструкторы
  // создает объект – пустая строка
  string(int N = 80): len(0){ch = new char[N+1]; ch[0] = '\0';}
  // создает объект по заданной строке
  string(const char *arch)
      {
        len = strlen(arch);
        ch = new char[len+1];
        strcpy(ch,arch);
      }
  // компоненты-функции
  // возвращает ссылку на длину строки
  int& len_str(void){return len;}
  // возвращает указатель на строку
  char *str(void){return ch;}
. . .};
```

Здесь у класса string два конструктора – перегружаемые функции.

По умолчанию создается также *конструктор копирования* вида T::T(const T&), где T – имя класса. Конструктор копирования вызывается всякий раз, когда выполняется копирование объектов, принадлежащих классу. В частности он вызывается:

а) когда объект передается функции по значению;

б) при построении временного объекта как возвращаемого значения функции;

в) при использовании объекта для инициализации другого объекта.

Если класс не содержит явным образом определенного конструктора копирования, то при возникновении одной из этих трех ситуаций производится побитовое копирование объекта. Побитовое копирование не во всех случаях является адекватным. Именно для таких случаев и необходимо определить собственный конструктор копирования. Например, в классе string:

```
string(const string& st)
{
      len=strlen(st.len);
      ch=new char[len+1];
      strcpy(ch,st.ch);
}
```

Можно создавать массив объектов, однако при этом соответствующий класс должен иметь конструктор по умолчанию (без параметров).

Массив объектов может инициализироваться либо автоматически конструктором по умолчанию, либо явным присваиванием значений каждому элементу массива.

```
class demo{
 int x;
 public:
  demo(){x=0;}
  demo(int i){x=i;}
};
void main()
{
 class demo a[20]; //вызов конструктора без параметров(по умолчанию)
 class demo b[2]={demo(10),demo(100)};//явное присваивание
}
```

Деструктор

Динамическое выделение памяти для объекта создает необходимость освобождения этой памяти при уничтожении объекта. Например, если объект формируется как локальный

внутри блока, то целесообразно, чтобы при выходе из блока, когда уже объект перестает существовать, выделенная для него память была возвращена. Желательно, чтобы освобождение памяти происходило автоматически. Такую возможность обеспечивает специальный компонент класса – **деструктор** класса. Его формат:

~имя_класса() { операторы_тела_деструктора }

Имя деструктора совпадает с именем его класса, но предваряется символом "~" (тильда).

Деструктор не имеет параметров и возвращаемого значения. Вызов деструктора выполняется неявно (автоматически), как только объект класса уничтожается.

Например, при выходе за область определения или при вызове оператора **delete** для указателя на объект.

```
string *p=new string "строка");
delete p;
```

Если в классе деструктор не определен явно, то компилятор генерирует деструктор по умолчанию, который просто освобождает память, занятую данными объекта. В тех случаях, когда требуется выполнить освобождение и других объектов памяти, например область, на которую указывает ch в объекте string, необходимо определить деструктор явно: `~string(){delete []ch;}`

Так же, как и для конструктора, не может быть определен указатель на деструктор.

Указатель this

Когда функция-член класса вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет имя **this** и неявно определен в каждой функции класса следующим образом:

*имя_класса *const this = адрес_объекта*

Указатель **this** является дополнительным скрытым параметром каждой нестатической компонентной функции. При входе в тело принадлежащей классу функции **this** инициализируется значением адреса того объекта, для которого вызвана функция. В результате этого объект становится доступным внутри этой функции.

В большинстве случаев использование **this** является неявным. В частности, каждое обращение к нестатической функции-члену класса неявно использует this для доступа к члену соответствующего объекта.

Примером широко распространенного явного использования **this** являются операции со связанными списками.

Наследование

Наследование – это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.

Существующие классы называются ***базовыми***, а новые – ***производными***. Производный класс наследует описание базового класса; затем он может быть изменен добавлением новых членов, изменением существующих функций-членов и изменением прав доступа. С помощью наследования может быть создана иерархия классов, которые совместно используют код и интерфейсы.

Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах.

В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов (***public, protected***).

Допускается множественное наследование – возможность для некоторого класса наследовать компоненты нескольких никак не связанных между собой базовых классов. В иерархии классов соглашение относительно доступности компонентов класса следующее:

private – член класса может использоваться только функциями – членами данного класса и функциями – "друзьями" своего класса. В производном классе он недоступен.

protected – то же, что и ***private,*** но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и функциями – "друзьями" классов, производных от данного.

public – член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к ***public*** - членам возможен доступ извне через имя объекта.

Следует иметь в виду, что объявление **friend** не является атрибутом доступа и не наследуется.

Синтаксис определения производного класса:

```
class имя_класса : список_базовых_классов {
      список_компонентов_класса
};
```

В производном классе унаследованные компоненты получают статус доступа **private**, если новый класс определен с помощью ключевого слова **class**, и статус **public**, если с помощью **struct**.

Явно изменить умалчиваемый статус доступа при наследовании можно с помощью атрибутов доступа – ***private***, ***protected*** *и* ***public***, которые указываются непосредственно перед именами базовых классов.

Конструкторы и деструкторы производных классов

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Параметры конструктора базового класса указываются в определении конструктора производного класса. Таким образом, происходит передача аргументов от конструктора производного класса конструктору базового класса.

Например:

```
class Basis {
      int a,b;
public:
      Basis(int x, int y) { a = x; b = y; }
};

class Inherit : public Basis {
      int sum;
public:
      Inherit(int x, int y, int s) : Basis(x, y) { sum = s; }
};
```

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный класс. Таким образом, объект производного класса содержит в качестве подобъекта объект базового класса.

Уничтожаются объекты в обратном порядке: сначала производный, потом его компоненты-объекты, а потом базовый объект.

Таким образом, порядок уничтожения объекта противоположен по отношению к порядку его конструирования.

Виртуальные функции

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы, включающие такие функции, называются **полиморфными** и играют особую роль в ООП.

Виртуальные функции предоставляют механизм **позднего** (**отложенного**) или **динамического связывания.** Любая нестатическая функция базового класса может быть сделана виртуальной, для чего используется ключевое слово **virtual**.

Пример.

```
class base {
public:
      virtual void print() { cout << "\nbase"; }
      . . .
};

class dir : public base {
public:
      void print() { cout << "\ndir"; }
};

void main() {
      base B, *bp = &B;
      dir D, *dp = &D;
      base *p = &D;
      bp -> print();    // base
      dp -> print();    // dir
      p  -> print();    // dir
}
```

Таким образом, интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, т.е. от типа объекта, для которого выполняется вызов.

Выбор того, какую виртуальную функцию вызвать, будет зависеть от типа объекта, на который фактически (в момент выполнения программы) направлен указатель, а не от типа указателя.

Виртуальными могут быть только нестатические функции-члены.

Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор **virtual** может не использоваться.

Конструкторы не могут быть виртуальными, в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.

Абстрактные классы

Абстрактным называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция.

Чистой виртуальной функцией называется компонентная функция, которая имеет следующее определение:

```
virtual тип имя_функции (список_формальных_параметров) = 0;
```

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Абстрактный класс может использоваться только в качестве базового для производных классов.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. При этом построение иерархии классов выполняется по следующей схеме. Во главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизировать и реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые, по сути, есть **абстрактные методы.**

Пример:

```
class Base {
public:
      Base();                          // конструктор по умолчанию
      Base(const Base&);               // конструктор копирования
      virtual ~Base();                 // виртуальный деструктор
      virtual void Show()=0;           // чистая виртуальная функция
                                       // другие чистые виртуальные функции
protected:                             // защищенные члены класса
      ...
private:
      ...                              // часто остается пустым, иначе будет
                                          мешать будущим разработкам
};

class Derived: virtual public Base {
public:
      Derived();                       // конструктор по умолчанию
      Derived(const Derived&);         // конструктор копирования
      Derived(параметры);              // конструктор с параметрами
      virtual ~Derived();              // виртуальный деструктор
      void Show();                     // переопределенная виртуальная
                                          функция
      ...                              // другие переопределенные виртуальные
                                          функции
                                       // другие перегруженные операции
protected:
                                       // используется вместо private,
                                          если ожидается наследование
private:
                                       // используется для деталей реализации
};
```

Объект абстрактного класса не может быть формальным параметром функции, однако формальным параметром может быть указатель на абстрактный класс. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс. Таким образом, мы получаем **полиморфные объекты**.

Абстрактный метод может рассматриваться как обобщение *переопределения.* В обоих случаях поведение родительского класса изменяется для потомка. Для абстрактного метода, однако, поведение просто не определено. Любое поведение задается в производном классе.

Одно из преимуществ абстрактного метода является чисто концептуальным: программист может мысленно наделить нужным действием абстракцию сколь угодно высокого уровня. Например, для геометрических фигур мы можем определить метод *Draw*, который их рисует: треугольник *TTriangle*, окружность *TCircle*, квадрат *TSquare.* Мы определим аналогичный метод и для абстрактного родительского класса *TGraphObject*. Однако такой метод не может выполнять полезную работу, поскольку в классе *TGraphObject* просто нет достаточной информации для рисования чего бы то ни было. Тем не менее присутствие метода *Draw* позволяет связать функциональность (рисование) только

один раз с классом *TGraphObject,* а не вводить три независимые концепции для подклассов *TTriangle, TCircle, TSquare.*

Имеется и вторая, более актуальная причина использования абстрактного метода. В объектно-ориентированных языках программирования со статическими типами данных, к которым относится и C++, программист может вызвать метод класса, только если компилятор может определить, что класс действительно имеет такой метод. Предположим, что программист хочет определить полиморфную переменную типа *TGraphObject,* которая будет в различные моменты времени содержать фигуры различного типа. Это допустимо для полиморфных объектов. Тем не менее, компилятор разрешит использовать метод *Draw* для переменной, только если он сможет гарантировать, что в классе переменной имеется этот метод. Присоединение метода *Draw* к классу *TGraphObject* обеспечивает такую гарантию, даже если метод *Draw* для класса *TGraphObject* никогда не выполняется. Естественно, для того чтобы каждая фигура рисовалась по-своему, метод *Draw* должен быть *виртуальным.*

Методические указания:

1. Программа использует три файла:
   - заголовочный h-файл с определением класса,
   - cpp-файл с реализацией класса,
   - cpp-файл демонстрационной программы.

Для предотвращения многократного включения файла-заголовка следует использовать директивы препроцессора

```
#ifndef STUDENTH
#define STUDENTH
// модуль STUDENT.H
...
#endif
```

2. Пример определения класса.

```
const int LNAME=25;
class STUDENT{
 char name[LNAME]; // имя
 int age;          // возраст
 float grade;      // рейтинг
 public:
  STUDENT();         // конструктор без параметров
  STUDENT(char*,int,float); // конструктор с параметрами
  STUDENT(const STUDENT&);  // конструктор копирования
  ~STUDENT();
  char * GetName() ;
  int GetAge() const;
  float GetGrade() const;
  void SetName(char*);
  void SetAge(int);
  void SetGrade(float);
  void Set(char*,int,float);
  void Show();
};
```

Более профессионально определение поля **name** типа указатель: char* name. Однако в этом случае реализация компонентов-функций усложняется.

3. Пример реализации конструктора с выдачей сообщения.

```
STUDENT::STUDENT(char*NAME,int AGE,float GRADE)
{
strcpy(name,NAME); age=AGE; grade=GRADE;
cout<< \nКонструктор с параметрами вызван для объекта  <<this<<endl;
}
```

4. Следует предусмотреть в программе все возможные способы вызова конструктора копирования. Напоминаем, что конструктор копирования вызывается:

а) при использовании объекта для инициализации другого объекта

Пример.

```
STUDENT a("Иванов",19,50), b=a;
```

б) когда объект передается функции по значению

Пример.

```
void View(STUDENT a){a.Show;}
```

в) при построении временного объекта как возвращаемого значения функции

Пример.

```
STUDENT NoName(STUDENT & student)
{STUDENT temp(student);
temp.SetName("NoName");
return temp;}

STUDENT c=NoName(a);
```

5. Для определения иерархии классов связать отношением наследования классы, приведенные в приложении (для заданного варианта). Из перечисленных классов выбрать один, который будет стоять во главе иерархии. Это абстрактный класс.
6. Определить в классах все необходимые конструкторы и деструктор.
7. Компонентные данные класса специфицировать как **protected**.

Варианты заданий:

1. студент, преподаватель, персона, заведующий кафедрой;
2. служащий, персона, рабочий, инженер;
3. рабочий, кадры, инженер, администрация;
4. деталь, механизм, изделие, узел;
5. организация, страховая компания, судостроительная компания, завод;
6. журнал, книга, печатное издание, учебник;
7. тест, экзамен, выпускной экзамен, испытание;
8. место, область, город, мегаполис;
9. игрушка, продукт, товар, молочный продукт;
10. квитанция, накладная, документ, чек;
11. автомобиль, поезд, транспортное средство, экспресс;
12. двигатель, двигатель внутреннего сгорания, дизель, турбореактивный двигатель;
13. республика, монархия, королевство, государство;
14. млекопитающие, парнокопытные, птицы, животное;
15. корабль, пароход, парусник, корвет.