**Теория в конце задание**

**КЛАССЫ И ОБЪЕКТЫ В С++**

**Основное содержание работы.**

Написать программу, в которой создаются и уничтожаются объекты,

определенного пользователем класса. Выполнить исследование вызовов

конструкторов и деструкторов.

**Краткие теоретические сведения.**

**Класс.**

Класс − фундаментальное понятие С++, он лежит в основе многих

свойств С++. Класс предоставляет механизм для создания объектов. В

классе отражены важнейшие концепции объектно-ориентированного про-

граммирования: инкапсуляция, наследование, полиморфизм.

С точки зрения синтаксиса, класс в С++ − 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 “ква-

лифицированных” 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 дос-

тупны 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 ав-

томатической инициализации создаваемых объектов. Для каждого вновь

создаваемого объекта необходимо было вызвать функцию типа set (как для

класса complex) либо явным образом присваивать значения данным объек-

та. Однако для инициализации объектов класса в его определение можно

явно включить специальную компонентную функцию, называемую **конст-**

**руктором** . Формат определения конструктора следующий:

***имя\_класса(список\_форм\_параметров){операторы\_тела\_конструктора}***

Имя этой компонентной функции по правилам языка С++ должно

совпадать с именем класса. Такая функция автоматически вызывается при

определении или размещении в памяти с помощью оператора **new** каждого

объекта класса .

**Пример .**

с omplex(double re1 = 0.0,double im1 = 0.0){re = re1; im = im1;}

Конструктор выделяет память для объекта и инициализирует дан-

ные − 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 соответст-

венно 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 кон-

структор всегда автоматически вызывается при определении (создании)

объекта. В этом случае вызывается конструктор без параметров. Для явно-

го вызова конструктора используются две формы:

*имя\_класса имя\_объекта (фактические\_параметры);*

*имя\_класса (фактические\_параметры);*

Первая форма допускается только при не пустом 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 ), с ( сс ){}

. . .

};

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 Т – 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 произво-

дится побитовое копирование объекта. Побитовое копирование не во всех

случаях является адекватным. Именно для таких случаев и необходимо оп-

ределить 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 соответствую-

щий класс должен иметь конструктор по умолчанию (без параметров).

Массив объектов может инициализироваться либо автоматически

конструктором по умолчанию, либо явным присваиванием значений каж-

дому 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 умолча-

нию)

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 не явно (автоматически), как только объект клас-

са уничтожается.

Например, при выходе за область определения или при вызове опе-

ратора delete для указателя на объект.

string \*p=new string “ строка ”);

delete p;

Если в классе деструктор не определен явно, то компилятор генери-

рует деструктор по умолчанию, который просто освобождает память, заня-

тую данными объекта. В тех случаях, когда требуется выполнить освобож-

дение 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.

**Указатели на компоненты - функции .**

Можно определить указатель на компоненты-функции.

***тип\_возвр\_значения(имя\_класса::\*имя\_указателя\_на\_функцию)***

***(специф\_параметров\_функции);***

**Пример.**

double(complex : :\*ptcom)(); // Определение указателя

ptcom = &complex : : real; // Настройка указателя

// Теперь для объекта А можно вызвать его функцию

complex A(5.2,2.7);

cout<<(A.\*ptcom)();

Можно определить также тип указателя на функцию

typedef double&(complex::\*PF)();

а затем определить и сам указатель

PF ptcom=&complex::real;

**Порядок выполнения**

1. Определить пользовательский класс в соответствии с заданием.

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. Определить указатель на компоненту-функцию.

6. Определить указатель на экземпляр класса.

7. 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). 8. Показать в программе использование указателя на объект и указателя на компоненту-функцию.

**Методические указания:**

1. 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 поле name типа указатель:

char\* name.

Однако в этом случае реализация компонентов-функций усложняется.

2. Пример реализации конструктора с выдачей сообщения.

STUDENT::STUDENT(char\*NAME,int AGE,float GRADE)

{ strcpy(name,NAME); age=AGE; grade=GRADE;

cout<< \n"Constructor with parameters was called"

<<this<<endl; }

3. 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);

4. В программе необходимо предусмотреть размещение объектов как в статической, так и в динамической памяти, а также создание массивов объектов.

Примеры.

а) массив студентов размещается в статической памяти

STUDENT gruppa[3];

gruppa[0].Set(“Иванов”,19,50);

и т.д.

или

STUDENT gruppa[3]={STUDENT(“Иванов”,19,50),

STUDENT(“Петрова”,18,25.5),

STUDENT(“Сидоров”,18,45.5)};

б) массив студентов размещается в динамической памяти

STUDENT \*p;

p=new STUDENT [3];

p-> Set(“Иванов”,19,50); и т.д.

5. Пример использования указателя на компонентную функцию

void (STUDENT::\*pf)();

pf=&STUDENT::Show;

 (p[1].\*pf)();

**6. Программа использует три файла:**

• заголовочный h-файл с определением класса,

• cpp-файл с реализацией класса,

• сpp-файл демонстрационной программы.

Для предотвращения многократного включения файла-заголовка

следует использовать директивы препроцессора

#ifndef STUDENTH

#define STUDENTH

// модуль STUDENT.H

#endif

ЗАДАНИЕ:

ЦЕХ

имя – char\*

начальник – char\*

количество  работающих – int