Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Локтионова Оксана Геннадьевна
Должность: проректор по учебной работе
Дата подписания: 01.10.2024 10:01:51
Уникальный программный ключ:
0b817ca911e6668abb13a5d426d39e5f1c11eabbf73e943df4a4851fda56d089

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Юго-Западный государственный университет»
(ЮЗГУ)
Кафедра программной инженерии

УТВЕРЖДАЮ
Проректор по учебной работе
____________ О.Г. Локтионова
«_9_» _04_ __________ 2024 г.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ

Методические указания к выполнению практических заданий для студентов направления подготовки 02.03.03 «Математическое обеспечение и администрирование информационных систем»

Курск 2024

Составитель Л.А. Лисицин

Рецензент
Кандидат технических наук, доцент Ю.А. Халин

Объектно-ориентированный анализ и программирование: методические указания к выполнению практических заданий/ ЮгоЗап. гос. ун-т; сост.: Л.А. Лисицин. Курск, 2024. 29 с. Библиогр.: с. 29.

Содержит методические указания к выполнению практических заданий по дисциплине «Объектно-ориентированный анализ и программирование». Предназначено для студентов направления подготовки 02.03.03 «Математическое обеспечение и администрирование информационных систем».

Текст печатается в авторской редакции

Подписано в печать 9.07. Формат 60x84 1\16.
Усл.печ. л. 2.03. Уч.-изд.л. 1.84. Тираж 100 экз.Заказ 534 Бесплатно.
Юго-Западный государственный университет
305045, г.Курск, ул 50 лет Октября, 94.

Содержание

1. Общие принципы объектного подхода. Объектная декомпозиция.
2. Работа с классами и объектами.
3. Изучение механизма наследования

Практическая работа.
Общие принципы объектного подхода. Объектная декомпозиция

Цель работы: научиться выполнять объектную декомпозицию предметной области для разработки программы на основе объектно-ориентированного подхода.

Объектной декомпозицией называют процесс представления предметной области задачи в виде совокупности функциональных элементов (объектов), обменивающихся в процессе выполнения программы входными воздействиями (сообщениями).

Каждый выделяемый объект предметной области отвечает за выполнение некоторых действий, зависящих от полученных сообщений и параметров самого объекта. Совокупность значений параметров объекта называют его состоянием, а совокупность реакций на получаемые сообщения - поведением.

Параметры состояния и элементы поведения объектов определяются условием задачи. В процессе решения задачи объект, получив некоторое сообщение, выполняет заранее определенные действия, например, может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и, в свою очередь, сформировать сообщения другим объектам. Таким образом, процессом решения задачи управляет последовательность сообщений. Передавая эти сообщения от объекта к объекту, программа выполняет необходимые действия.

Пример

Составить диаграмму объектов программы исследования элементарных функций, которая для функций y=sin x, y=cos x, y=tg x, y=ln x, $y=e^x$ выполняет следующие действия: 1) строит таблицу значений функции на заданном отрезке с заданным шагом;

2) определяет корни функции на заданном отрезке;

3) определяет максимум и минимум функции на заданном отрезке.

В основе объектной декомпозиции также лежит граф состояний интерфейса. Будем считать, что каждое состояние интерфейса – это состояние некоторого функционального элемента системы, т. е. объекта. Состояний интерфейса пять, соответственно получаем пять объектов. Назовем эти объекты следующим образом: Главное меню, Меню операций, Табулятор, Определитель корней, Определитель экстремумов. Эти объекты передают управление друг другу, генерируя сообщение Активизировать. Результат объектной декомпозиции изображают в виде диаграммы объектов (рис. 1).

Кроме этого можно выделить еще один объект Функцию, который должен обеспечивать вычисление выбранной функции по заданному аргументу. Номер функции сообщается данному объекту Главным меню после того, как пользователь осуществит выбор.

Полная характеристика объекта включает идентифицирующее условное имя, а также перечень и описание параметров состояния и аспектов поведения.

Так состояние объекта Функция характеризуется единственным параметром: номером функции, который передает ему Главное меню. Поведение же включает реакции на два типа сообщений: получив номер функции, объект должен сохранить его, изменив таким образом свое состояние, а получив запрос на вычисление значения функции, сопровождающийся определенным значением аргумента, - вернуть значение функции в заданной точке.

Таким образом, при выполнении объектной декомпозиции определяют и описывают множество объектов предметной области и множество сообщений, которое формирует и получает каждый объект.

Рис. 1

Задание: выполнить объектную декомпозицию заданной предметной области в виде диаграммы объектов и связей между ними.

Варианты заданий:

1. Графический редактор, который выполняет рисование следующих типов геометрических фигур: линия, треугольник, параллелограмм, прямоугольник, окружность, эллипс, ромб. Для каждой фигуры должна быть предусмотрена возможность выбора цвета линий, заливки, масштабирования, удаления, а также сохранения рисунка в файл.

2. Текстовый редактор, позволяющий набирать текст и выполнять с ним стандартные операции форматирования, а также сохранять его в файл.

3. Калькулятор (в соответствии со стандартной программой Windows «Калькулятор» типа «Обычный»).

4. Музыкальный проигрыватель стандартных аудиофайлов.

5. Архиватор, позволяющий сжимать файлы в архив и извлекать их, просматривать файлы в архиве, задавать пароли при архивировании, удалять файлы из архива, тестировать архив на наличие ошибок.

6. Почтовый клиент, позволяющий получать, отправлять, удалять письма электронной почты, хранить адреса контактов, выполнять массовую рассылку писем.

7. Интернет-браузер, содержащий строку адреса сайта, закладки, домашнюю страницу, загружаемую по умолчанию, историю посещённых страниц, средство печати страниц.

8. Менеджер паролей, который хранит данные учётных записей пользователя: логин, пароль и сайт, для которого предназначены учётные данные. Менеджер должен иметь мастер-пароль для доступа к хранимым данным, позволять добавлять, изменять и удалять сведения об учётных записях. Предусмотреть возможность наличия более одной учётной записи для одного и того же сайта, копирование пароля в буфер обмена.

9. Программа-коммуникатор, позволяющая обмениваться текстовыми сообщениями по сети индивидуально или в совместном чате, а также проводить видеоконференции, звонить на стационарные и мобильные телефоны.

10. Программа-часы, позволяющая ставить будильник с выбираемым пользователем звуковым сигналом, задавать текущее время, выбирать часовой пояс, ставить таймер обратного отсчёта и секундомер.

Контрольные вопросы

1. Дайте определение понятию «Предметная область».
2. Сформулируйте принципы правильной декомпозиции системы.
3. Какие основные подходы выделяют при разработке ПО? Особенности и достоинства каждого подхода.
4. Какие диаграммы наиболее часто применяют при структурном подходе?
5. Опишите применение различных диаграмм структурного подхода на стадиях формирования требований и проектирования.
6. С учетом каких принципов строится объектная модель в объектно-ориентированном подходе?
7. Основные понятия объектно-ориентированного подхода: объект, класс, наследование и полиморфизм.

8. Назовите преимущества и недостатки объектно-ориентированного подхода.

Практическая работа.

Работа с классами и объектами

Цель. Получить практические навыки реализации классов программирования на на C#.

<u>Основное содержание работы.</u>

Написать программу, в которой создаются и разрушаются объекты, определенного пользователем класса. Выполнить исследование вызовов конструкторов и деструкторов.

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: инкапсуляция, наследование, 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 – установитьдату void
get(int*,int*,int*); // метод – получитьдату void next(); //
метод – установить следующую дату void print(); // метод
– вывести дату}; struct class complex // комплексноечисло
{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

<u>Например:</u> 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. <u>Пример.</u>

```
class complex
{
double re, im; // private поумолчанию
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_к онструктора)

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:

1. Для конструктора не определяется тип возвращаемого значения. Даже тип void не допустим.
2. Указатель на конструктор не может быть определен, и соответственно нельзя получить адрес конструктора.
3. Конструкторы не наследуются.
4. 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 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 ),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’;}
// создаетобъектпозаданнойстроке 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, вклассе 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 не явно (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.

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 A 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;

<u>Порядок выполнения работы.</u>

1. Определить пользовательский класс в соответствии с вариантом задания (смотри приложение).
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.

Примеры

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&); // конструкторкопирования

~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.

2. 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;

}

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); ит.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,
   1. cpp-файл с реализацией класса,
   2. cpp-файл демонстрационной программы.

Для предотвращения многократного включения файлазаголовка следует использовать директивы препроцессора

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

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, какой класс должен быть реализован, какие должны быть в нем конструкторы, компоненты-функции и т.д.

3. Определение пользовательского класса с комментариями.

4. Реализация конструкторов и деструктора.

5. Фрагмент программы, показывающий использование указателя на объект и указателя на функцию с объяснением.

6. Листинг основной программы, в котором должно быть указано, в каком месте и какой конструктор или деструктор вызываются.

Контрольные вопросы.

1. Для чего служит конструктор?
2. Может ли в классе быть несколько конструкторов?
3. Чем должны отличаться различные конструкторы одного и того же класса?
4. Для чего служит деструктор класса?
5. Имеет ли деструктор параметры?
6. В каком случае в тело деструктора включается оператор delete?
7. Какие сообщения и в какой последовательности будут выведены на монитор в следующем примере?

```
class Alpha {
public: int x, y;
Alpha(){cout<<"Constructor #1"<<endl;}
Alpha(int _m){cout<<"Constructor #2"<<endl; x=y=_m; }
~Alpha(){cout<<"Destructor "<<endl;}
}; void main() {
Alpha a1; a1.x=1;
Alpha *a2;
a2=new Alpha;
Alpha a3[2];
Alpha a4(4);
Alpha   a5=a1;   Alpha
a6(a1);
cout<<a5.x<<endl<<a6.x<<endl;
}
```

8. Определен следующий класс.

```
class Alhpa { public: int abc; };
```

Запишите обращения к компоненте abc с использованием точки и стрелки.

9. Что такое агрегация классов и как она изображается на диаграммах?

10. Чем отличается строгая агрегация от нестрогой?

Практическая работа.
Изучение механизма наследования

Цель работы: ознакомление с отношением наследования, виртуальными функциями, абстрактными классами.

Основные сведения Базовый и производные классы

Наследование – механизм языка, позволяющий написать новый класс на основе уже существующего (базового, родительского) класса. Формат определения производного класса следующий:

тип_класса имя_производного_класса : список[
модификатор_доступа имя_базового_класса]
{определение_производного_класса};

В объекте производного класса наряду с собственными полями создаются поля, имя и тип которых определены в базовом классе, а сам объект производного класса получает доступ к методам базового. В этом, собственно, и заключается реализация механизма наследования.

Использование наследования позволяет строить иерархии классов.

При создании объектов производного класса сначала автоматически вызываются конструкторы базовых классов согласно списку базовых классов в объявлении производного класса, а затем конструктор производного класса. Объекты разрушаются в порядке, обратном их созданию, т.е. вначале вызывается деструктор производного класса, а затем базового.

Один класс может быть базовым для нескольких производных, производный класс может быть, в свою очередь, базовым для какого-либо класса и т.д. Если производный класс имеет несколько базовых, то такое наследование называется множественным.

Если доступ к собственным компонентам производных классов определяется обычным образом, то на доступ к наследуемым компонентам влияет, во-первых, атрибут доступа в базовом классе и, во–вторых, модификатор доступа (public / private), указанный перед именем базового класса в конструкции определения производного класса. Общепринятое правило - поля, базового класса определяются как защищенные (protected).

На диаграммах отношение наследования изображается сплошной линией, начинающейся у производного класса и заканчивающейся стрелкой-пустым треугольником у базового класса.

При создании иерархии классов, включающей базовый и производные классы, следует придерживаться принципа "это есть" (is a), выделяя общие черты классов и инкапсулируя их в базовом. Положим, например, что нам надо создать классы Car (автомобиль) и Lorry (грузовик). Что общего у этих классов и чем они разнятся? На самом деле у них много общих черт и много различий, но для упрощения примем следующие формулировки:

- грузовик есть средство передвижения, имеющее цену и год выпуска, а также характеризующееся грузоподъемностью;
- автомобиль есть средство передвижения, имеющее цену и год выпуска, а также характеризующееся скоростью.

Создадим базовый класс Vehicle. В нем определим 2 целых поля для цены и года и методы для установки и возвращения значений этих полей. В производных классах Car (автомобиль) и Lorry (грузовик) определим по одному полю и соответствующие методы.

Листинг 2.1

```
//Базовый класс
classVehicle{ protected:
intprice;//цена intyear;
//год выпуска public:
//Устанавливаем цену и год выпуска void
setVehicle(int _price, int _year)
{price=_price; year=_year;} int
getPrice()const{return price;} int
getYear()const{return year;}
};
};
//Производныйкласс class
Lorry:public Vehicle{ intcapacity; //
грузоподъемность public:
//Устанавливаем и возвращаем грузоподъемность
void setCapacity(int _capacity) {capacity=_capacity;}
int getCapacity()const {return capacity;} };
// Производныйкласс class Car:
public Vehicle{ intspeed; //
скорость public:
//устанавливаем и возвращаем скорость void
setSpeed(int _speed){speed=_speed;} int
getSpeed()const {return capacity;} };
```

Собственно, все. Теперь в main можно создавать одиночные объекты Lorry и Car, массивы и задавать их параметры, например, так:

```
Lorry lo; lo. setVehicle(10000,
2012); lo. setCapacity(5000); Car
car1[4];
car1[0]. setVehicle(15000, 2011); car1[0]. setSpeed(200);
car1[1]. setVehicle(....); car1[1]. setSpeed(...); Lorry* lor=new
Lorry[3];
. . .
```

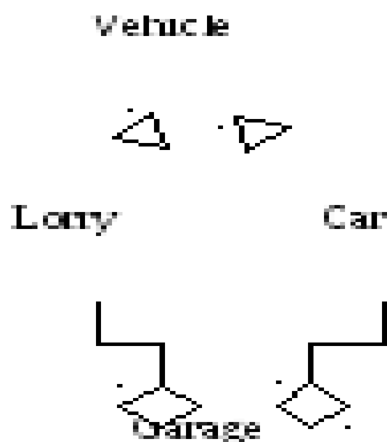
Предположим теперь, что нам надо добавить к программе автобусы. Все просто: автобус есть средство передвижения для перевозки пассажиров – добавляем класс Bus с полем, например, "количество пассажиров". При этом уже имеющиеся классы не трогаем; их можно даже не перекомпилировать. Таким образом, расширение программы происходит гораздо проще, чем без использования наследования. Это обстоятельство является одной из главных причин применения наследования.

Положим, что в при использовании разработанных классов мы всегда будем иметь дело с массивами их объектов, которые надо организовать и обрабатывать. Чтобы не заставлять клиента (в данном случае main) заниматься этим, введем класс Garage, включив в него разработанные нами типы, сформировав массивы и определив методы доступа к ним.

Вот как он может выглядеть (продолжаем программу): class

```
Garage{ int n, m; // Количество грузовиков и автомобилей в гараже
// Указатели на соответствующие классы
Lorry *lor; Car
*car; public:
//Конструкторы
Garage(int _n, int _m)// конструктор для создания обоих видов транспорта
{ n=_n; lor=new Lorry[n];
m=_m; car=new Car[m];
}
// Ввод данных для автомобилей
void setCars()
{ int _price, _year, _speed; for(int i=0;
i<m; i++)
{ cout<<"Input values price, year, speed for "<<i+1<< " Car"
<<endl; cin>>_price>>_year>>_speed;
car[i].setVehicle(_price,_year);
car[i].setSpeed(_speed); }
//Вводданныхдлягрузовиков          void
setLorries()
{ int _price, _year, _capacity; for(int i=0;
i<n; i++)
{ cout<<"Price, year, capacity for "<<i+1<< " Lorry"<<endl;
cin>>_price>>_year>>_capacity; lor[i].setVehicle(_price,_year);
lor[i].setCapacity(_capacity);
}
}
//Выводпараметровгрузовиков          void
getLorries()
```

```
{ cout<<"Lorries"<<endl; for(int i=0;
i<n; i++)
cout<< lor[i].getPrice()<<" "<<lor[i].getYear()<<" "
<<lor[i].getCapacity()<<endl; }
//Вывод для авто аналогичен
// Деструктор
~Garage(){ delete[] car;
delete[] lor;
}
};
//Теперь наш клиент приобретает совсем простой вид
int _tmain(int argc, _TCHAR* argv[]) {
Garage g1(2,3); //Объект - массив из 2-х грузовиков и 3-х авто g1.
setLorries(); // Вводданных
g1.setCars();
return 0; }
```



В классе Garage можно инкапсулировать все требуемые методы работы с массивами объектов: сортировку, ввод-вывод в файл и т.п.

Упрощение клиентской части программы за счет введения некоторого дополнительного класса является содержанием паттерна Фасад (Façade)[2].

Разобранный выше пример иллюстрирует обычный подход к использованию наследования – в каждом производном классе определяются свои собственные методы, работающие с объектами производных классов наряду с методами базового.

Другой подход предполагает использование в иерархии классов виртуальных функций с целью получения общего интерфейса иерархии.

Виртуальной называется функция, определенная с атрибутом virtual в базовом классе и имеющая в каждом производном классе ту же самую сигнатуру - тип, имя и список параметров. При этом реализация функции (ее тело) в конкретном производном классе может отличаться от ее реализации в базовом и других производных классах.

```
class Based {
/* ...*/ public: virtual int
fb(){return 3;}
/* ...*/
};
class Derived1:public Based {
/* ...*/
int fb() {return 5;}
/* ...*/
};
class Derived2:public Based {
int fb() {return 7;}
/* . . . */
};
```

Говорят, что если виртуальная функция вызывается из производного класса, то она перекрывает базовую версию.

Виртуальная функция класса может вызываться обычным образом - как член производного или базового класса через его объект. Но тогда никаких преимуществ по сравнению с обычной функцией ее применение не приносит.

Правильный вызов виртуальной функции заключается в последовательном применении следующих 2-х инструкций:

указатель_ на_базовый_ класс=указатель_на__объект_производного_ класса; указатель_на базовый_класс вызов_виртуальной_функции;

(Инструкции в тексте могут быть разделены. Заметьте, что в первой инструкции никакого преобразования типа указателей не нужно.)

Для нашего примера вызов функции fb из производного класса:

```
Based* ptr=new Derived1; ptr->fb(); //
возвращает 5
. . .
ptr=new Derived2; ptr->fb();//
возвращает 7
```

Виртуальная функция в базовом классе может быть равна 0
(чистая функция). В нашем случае

```
virtual int fb( )=0;
```

Класс с чистой виртуальной функцией называется абстрактным; объект такого класса создать нельзя, только указатель на объект. Мало того, если в каком-либо производном классе эта функция отсутствует, то от такого класса тоже нельзя создавать объекты.

При наличии виртуальных функций деструктор в базовом классе должен быть виртуальным.

Ниже приводится программа, аналогичная листингу 2.1, но с некоторыми упрощениями. Листинг 2.2 class Vehicle{ protected:

```
int price;//цена intyear;
//год выпуска public:
//Чистые виртуальные функции virtual
void setVehicle()=0; virtual void
getVehicle()=0;
//Виртуальныйдеструктор virtual ~
Vehicle(){}
};
class Lorry: public Vehicle{
intcapacity; public:
//Для упрощения делаем параметры любого грузовика одними и теми же
void setVehicle(){cout<<"setLorry"<<endl;price=40000;
year=5;capacity=6;}
void getVehicle(){cout<<"Lorry"<<endl<<price<<" "<<year<<"
"<<capacity<<endl;}
~Lorry(){}
};
class Car:public Vehicle{
intspeed; public:
//То же для авто
void setVehicle(){cout<<"setCar"<<endl; price=10000;
year=2;speed=3;} void getVehicle(){cout<<"Car"<<endl<<price<<"
"<<year<<" "<<speed<<endl;}
~Car(){}
}; classGarage
{ public:
/*Для упрощения количество грузовиков и авто в гараже
одинаковы и заданы константой */ staticconstintn=2;
//Массивы указателей на производные классы
Lorry*lor[n];
Car*car[n];
//Указатель на базовый класс
Vehicle*vec; public: Garage()
{
//Инициализация указателей на производные классы for (int
i=0; i<n; i++) { car[i]=new Car; lor[i]= new Lorry;} }
// Ввод и вывод данных для машин с вызовом виртуальных функций
void setVehicle(){ for(int i=0;
i<n; i++) { vec=car[i];
```

```
//Автомобили vec-
>setVehicle(); vec=lor[i];
//Грузовики vec-
>setVehicle();
} } void getVehicle(){
for(int i=0; i<n; i++) {
vec=car[i]; vec-
>getVehicle();
vec=lor[i]; vec-
>getVecicle();
}
}
~Garage()
{ for (int i=0; i<n;i++)
{delete car[i]; delete
lor[i];
}
} };
int _tmain(int argc, _TCHAR* argv[])
{ Garage g;
g.setVehicle();
g.getVehicle(); }
```

Что мы получили, используя виртуальные функции? Возможность доступа к производным классам с использованием одних и тех же функций, т.е. получили общий интерфейс иерархии. Естественно, что при добавлении третьего производного класса интерфейс должен остаться тем же.

Класс, содержащей виртуальный метод, называют полиморфным классом. В нашем случае это означает, что функции setVehicle, getVehicle в разных точках программы действуют поразному.

Механизм виртуальных функций правильно работает только для указателей и ссылок на объекты. Полиморфизм проявляется только тогда, когда объект производного класса адресуется косвенно, через указатель или ссылку на базовый. Использование самого объекта базового класса не сохраняет идентификацию типа производного класса.

Заметим, что функции Garage::getVehicle, Garage::setVehicle находятся вне иерархии и не имеют никакого отношения к виртуальным. Их можно было назвать по-другому.

Существенное замечание. Наряду с общими для всех классов виртуальными функциями в производном классе могут быть определены собственные функции (и поля). Но помните, что к ним невозможно обратиться через указатель на базовый класс, ибо последний работает только с теми компонентами иерархии, которые определены в базовом классе.

Варианты задания

Следующие варианты предполагают необходимость создания фасадного класса, в который инкапсулированы массивы объектов в динамической памяти и методы ввода и вывода параметров, и другие методы согласно заданию.

Базовый класс иерархии может и не быть абстрактным, но должен содержать хотя бы одну виртуальную функцию. Если есть необходимость, то в производном классе можно объявлять дополнительные компоненты.

1. Создать базовый класс CList (линейный однонаправленный список) с полями: указатели на следующий элемент; информационная часть – целое число. В производных классах – CQueue (очередь) и CStack (стек) – должны быть определены методы вставки и удаления узла в соответствии с дисциплиной обслуживания соответствующего класса.

2. Описать класс CString (Строка) и производные CStringBit (Битовая_строка) и CStringHex

   (Шестнадцетиричная_строка). (Описание класса см. вариант 9 в 1-й работе.) Строки первого подкласса могут содержать только двоичные символы, второго только шестнадцетиричные. Разработать методы ввода данных с проверкой допустимых символов. Содержимое строк рассматривается как двоичное или шестнадцетиричное число без знака. Разработать операции сравнения двух строк, сложения и вычитания. Фасадный класс - Text.

3. Создать класс ColoredPoint. На его основе создать производные классы Line и PolyLine (многоугольник). Все классы должны иметь методы установки и получения значений всех координат, а также изменения цвета и получения текущего цвета. Фасадный класс - Picture.

4. Создать класс Figure. На его основе реализовать классы Rectangle (прямоугольник), Circle (круг) и Trapeze (трапеция) с возможностью вычисления площади, центра тяжести и периметра. Фасадный класс - Picture.

5. Создать класс Number с виртуальными методами, реализующими арифметические операции. На его основе реализовать классы Integer и Real.Операции в классах не должны быть одинаковыми. Фасадный класс - Calculus.

6. Создать класс Body. На его основе реализовать классы Parallelepiped (прямоугольный параллелепипед), Cone (конус) и Ball (шар) с возможностью вычисления площади поверхности и объема. Фасадный класс - Series.

7. Создать класс Currency для работы с денежными суммами. Определить в нем методы перевода в рубли. На его основе реализовать классы Dollar, Euro и Pound (фунт стерлингов) с возможностью пересчета в центы и пенсы соответственно. Фасадный класс – Purse.

8. Создать класс Triangle (треугольник), задав в нем длину двух сторон, угол между ними, методы вычисления площади и периметра. На его основе создать классы, описывающие равносторонний, равнобедренный и прямоугольный треугольники со своими методами вычисления площади и периметра. Фасадный класс - Picture.

9. Создать класс Solution (решение) с методами вычисления корней уравнения и вывода на экран. На его основе реализовать классы Linear (линейное уравнение) и Square (квадратное уравнение). Фасадный класс - Series.

10. Создать класс Function (функция) с виртуальными методами вычисления значения функции y = f(x) в заданной точке x и вывода результата на консоль. На его основе определить классы Ellipse, Hiperbola и Parabola, в которых реализуются соответствующие математические зависимости. В фасадном классе Series создаются массивы для хранения нескольких последовательных значений y.

11. Создать класс Triad (тройка) с виртуальными методами увеличения на 1. На его основе реализовать классы Date (дата) и Time (время). В фасадном классе Memories, создать массив пар (дата-время) объектов этих классов в динамической памяти. Предусмотреть возможность выборки самого раннего и самого позднего событий.

12. Описать класс Element (элемент логической схемы), задав в нем числовой идентификатор, количество входов, идентификаторы присоединенных к нему элементов (до 10) и двоичные значения на входах и выходе. На его основе реализовать классы AND и OR – двоичные вентили, которые могут иметь различное количество входов и один выход и реализуют логическое умножение и сложение соответственно. В фасадном классе Scheme создать массивы вентилей.

13. Описать класс Element (элемент логической схемы) задав в нем символьный идентификатор, количество входов, идентификаторы присоединенных к нему элементов (до 10) и двоичные значения на входах и выходе. На его основе реализовать классы AND_NOT и OR_NOT — двоичные вентили, которые могут иметь различное количество входов и один выход и реализуют логическое умножение с отрицанием и сложение с отрицанием соответственно. В фасадном классе Scheme создать массивы вентилей.

14. Описать класс Trigger (триггер), задав в нем идентификатор и двоичные значения на входах и выходах. На его основе реализовать классы RS и JK, представляющие собой триггеры соответствующего типа. В фасадном классе Register предусмотреть общий сброс и установку значений произвольного триггера по заданным значениям входов.

15. Создать класс Progression (прогрессия) с виртуальными методами вычисления заданного элемента и суммы прогрессии. На его основе

реализовать классы Linear (арифметическая) и Exponential (геометрическая). Фасадный класс - Series.

16. Создать класс Pair (пара значений) с виртуальными методами, реализующими арифметические операции сложения и вычитания. На его основе реализовать классы Fractional (дробное) и LongLong (длинное целое).В классе Fractional вещественное число представляется в виде двух целых, в которых хранятся целая и дробная часть числа соответственно. В классе LongLong длинное целое число хранится в двух целых полях в виде старшей и младшей части. Фасадный класс - Series.

17. Создать класс Integer (целое) с символьным идентификатором, виртуальными методами, реализующими арифметические операции, и методом вывода на экран. На его основе реализовать классы Decimal (десятичное) и Binary (двоичное). Число представить в виде массива цифр. В фасадном классе Series предусмотреть возможность вывода значений и идентификаторов всех объектов списка и вывода общей суммы всех десятичных значений.

18. Создать класс Sorting (сортировка) с массивом, задаваемым с помощью new, и виртуальными методами ввода элементов, сортировки и вывода на экран. На его основе реализовать классы Choice (метод выбора) и Quick (быстрая сортировка). Фасадный класс - Series.

19. Создать класс Pair (пара значений) с виртуальными методами, реализующими арифметические операции, и методом вывода на экран. На его основе реализовать классы Money (деньги) и Complex (комплексное число). В классе Money денежная сумма представляется в виде двух целых, в которых хранятся рубли и копейки соответственно. При выводе части числа снабжаются словами «руб.» и «коп.». В классе Complex предусмотреть при выводе символ мнимой части (i). Фасадный класс - Series.

20. Создать класс Worker с полями, описывающими должность, фамилию работника и его обязанности, а также фамилию его руководителя. На его основе реализовать классы Manager (руководитель проекта), Developer (разработчик) и Coder программист) с соответствующими методами. Фасадный класс

Group, в котором предусмотрена возможность выборки по фамилии с выводом всего дерева подчиненных. (Роль руководителя проекта заключается в полной ответственности за успешное планирование, выполнение и завершение проекта. Разработчик - поддержка существующего продукта, разработка новых функциональных возможностей и новых компонентов, выбор программных средств.)

Контрольные вопросы

1. В чём заключается наследование одного класса другому?

2. В чем разница в организации наследования полей и методов?

3. Определены 2 класса:

```
class Based{public: int x;};
class Derived :public Based{};
```

Какое значение выводится на консоль при применении следующего кода: `Based b1; b1.x=3; Derived d1; d1.x=4; cout<<b1.x;`

3. Удачной ли является иерархия классов, при которой некоторый класс X является производным от большого количества классов с большим числом полей в каждом (A↓B ↓ C ↓…↓ X)? Какая существует альтернатива наследованию?

4. Каким образом производится управление доступом к унаследованным компонентам производных классов? Есть ли в представленном фрагменте программы ошибки и какое сообщение выдаст компилятор? Как исправить ошибку?

```
class Base {
int x; public:
int y;
void setX(int n){x=n;}
void showX() const{ cout<<x<<endl;}
};
class Derived: public Base {
public:    void    setY(int
n){y=n;}
void show_sum() const{ cout<<x+y<<endl; }
};
int _tmain(int argc, _TCHAR* argv[])
{
Derived d1;
d1.show_sum();
return 0; }
```

5. Есть ли ошибки в нижеследующих объявлениях

```
// класс Shape абстрактный
Shape sh;
Shape *psh;
Shape *psh1=new Shape();
```

6. Что из себя представляет виртуальная функция и как она должна вызываться? 7. Как создать общий интерфейс иерархии классов?

Список литературы

1. Лафоре, Р. Объектно-ориентированное программирование в С++ [Текст] / Р. Лафоре. - 4-е изд. - СПб. [и др.] : Питер, 2012. – 928 с.
2. Зыков, С. В. Введение в теорию программирования : курс лекций / С. В. Зыков. - 2-е изд., испр. - Москва : Национальный Открытый Университет «ИНТУИТ», 2016. - 189 с. - URL: http://biblioclub.ru/index.php?page=book&id=429073 (дата обращения 13.05.2024) . - Режим доступа: по подписке. - Текст : электронный.
3. Буч, Г. Объектно-ориентированный анализ и программирование и проектирование с примерами и приложениями на С++ / пер. с англ. - 2-е изд. - СПб. : Бином ; СПб. : Невский диалект, 2001. - 560 с. - Текст : непосредственный.
4. Сорокин, А.А. Объектно-ориентированное программирование [Электронный ресурс] : учебное пособие (курс лекций) / А.А. Сорокин. – Ставрополь : Изд-во СКФУ, 2014. – 174 с. Режим доступа / https://biblioclub.ru/index.php?page=book_view_red&book_id=457696