Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет математики и информатики

Т. А. Галаган

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. ЯЗЫК C++

Учебное пособие

Благовещенск

2016

ББК 32.973-018.1
Г15

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного
университета

Т.А. Галаган

Объектно-ориентированное программирование. Язык С++ . Учебное пос**о**бие **д**ля бакалавров направления подготовки 38.03.05 – «Бизнес-информатика». – Благовещенск: Амурский гос. ун-т, 2016. – 56 с.

Пособие посвящено основам объектно-ориентированного подхода к построению программ на примере популярного языка программирования С++. Рассмотрены основные элементы языка и примеры простых для понимания программ. Рассмотрены основные принципы объектно-ориентированного программирования и правила их реализации в простых программах.

Рецензенты:

Е.Ф Алутина канд. физ.-мат. наук, доцент, зав. кафедрой информатики и методики преподавания информатики ФГБОУ ВПО «Благовещенский государственный педагогический университет»;

Н.А Чалкина канд. пед. наук, доцент кафедры общей математики и информатики ФГБОУ ВПО «Амурский государственный университет»;

ВВЕДЕНИЕ

Пособие посвящено популярному объектно-ориентированному подходу в программирование, основные принципы которого рассмотрены в примерах на языке С++. Пособие предназначено для студентов второго курса, которые изучили основы построениями синтаксических конструкций языка С++ и принципы разработки приложений на основе структурного и модульного подходов в рамках изучении дисциплины «Программирование». Поэтому материал не содержит непосредственного изложения синтаксических правил построения простейших конструкций языка С++.

Оно предназначено для самостоятельной работы студентов в курсе «Объектно-ориентированный анализ и программирование».

В пособии рассмотрены основные принципы объектно-ориентированного программирования: инкапсуляция, наследование и полиморфизм, а также механизм создания шаблонов, использование контейнерных классов стандартной библиотеки языка С++, принципы построения диаграмм классов на языке UML.

Материал содержит не только теоретические сведения, но и сопровождается примерами программ. Разделы завершаются вопросами самопроверки и упражнениями.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ

Общие положения объектного подхода

Анализ предметной области необходим до начала написания программы. Объектный подход к разработке сложных программных систем предполагает необходимость этапов анализа и моделирования предметной области до непосредственного кодирования программы на каком-либо языке программирования.

Объектный подход применяется на всех основных стадиях жизненного цикла программного обеспечения и включает в себя несколько этапов.

Объектно-ориентированный анализ – методология анализа предметной области, который выполняется с целью выделения объектов и классов в качестве требования к проектируемой системе.

Объектно-ориентированное проектирование – методология проектирования, объединяющая в себе процесс декомпозиции объектов и приемы их представления логической и физической моделью проектируемой системы.

Объектно-ориентированное программирование – методология (парадигма программирования), в основе которой лежит построение программы в форме взаимодействия объектов, каждый из которых является экземпляром определенного класса.

Под объектом понимают некоторую сущность, обладающую определённым состоянием и поведением, имеющую заданные значения свойств (атрибутов) и операций над ними (методов).

Объект определяется через его внешнее отличие от других объектов. Внутренняя особенность объекта (его структура, внутренние характеристики) не влияет на внешнее отличие и для объектного моделирования значения не имеет.

Главная цель объектного анализа – представление предметной области в виде множества объектов со своими свойствами и характеристиками, которые достаточны для их определения и идентификации, а также для задания поведе-

ния объектов в рамках выбранной системы понятий и абстракций. Каждый объект – это уникальный элемент, имеющий, по крайней мере, одно свойство или характеристику и уникальную идентификацию во множестве объектов.

Предметная область может являться самостоятельным объектом или быть объектом в составе другой предметной области.

Класс представляет собой множество объектов, обладающих одинаковыми свойствами и операциями, отношениями и семантикой. Любой объект является экземпляром класса.

Отношения между классами. Диаграммы классов на языке UML

Язык унифицированного моделирования UML (Unified Modeling Language) является визуальным средством представления моделей программ, ставшим уже стандартом. UML широко применяется для визуализации, специфицирования, конструирования и документирования различных программных систем.

Под моделями программ понимается их графическое представление в виде различных диаграмм, отражающих связи между объектами в программном коде. Одной из основных диаграмм UML является диаграмма классов, которая очень удобна для сопоставления различных вариантов проектных решений. Кроме того, она используется для описания шаблонов проектирования, которые в сочетании с объектно-ориентированной парадигмой программирования лежат в основе современного подхода к разработке программного обеспечения.

На диаграмме UML класс изображается в виде прямоугольника, состоящего из трех частей, расположенных вертикально. В верхней части указывается имя класса. В средней части приводится список полей, называемых в диаграмме UML атрибутами. Возможно, указание типов и атрибутов полей после двоеточия. Список методов (операций) приводится в нижней части, возможно, с указанием возвращаемого значения.

Имена абстрактных классов и операций принято обозначать курсивом. Перед именами полей и методов указывается спецификатор доступа с помощью

одного из трех символов: + для public, - для private, # для protected. Для статических элементов класса после спецификатора доступа записывается символ $.

Во второй и третьих частях могут указываться не все элементы класса, а только представляющие интерес на данном уровне абстракции. Эти части могут быть совсем пусты, и в этом случае их можно не указывать.

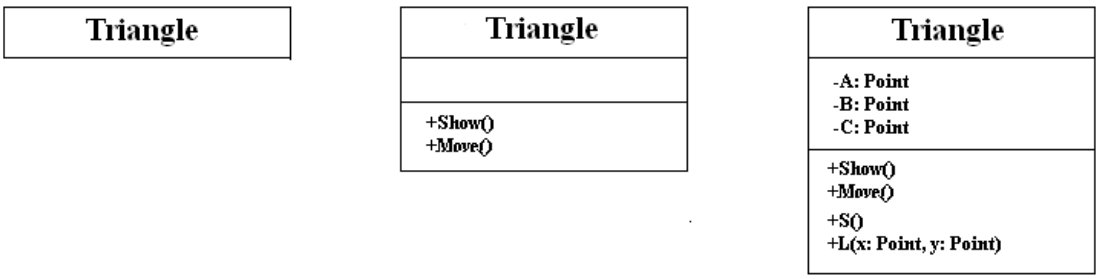

Рисунок 1 – Пример изображения класса на диаграмме UML

Существуют следующие отношения между классами: ассоциация, наследование, агрегация, зависимость.

Если два класса взаимодействуют друг с другом концептуально, то их взаимодействие называется ассоциацией. На диаграмме ассоциация показывается соединительной линией, над которой может быть указана кратность – то есть сколько объектов данного класса может быть связано с одним объектом другого класса. В случае произвольного количества указывается символ звездочка.

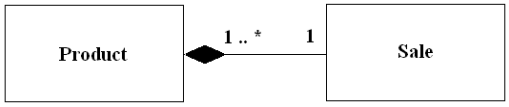

Рисунок 2 – Обозначение отношения «ассоциация» между классами на диаграмме UML

Отношение «ассоциация» выявляется на ранней стадии проектирования, и в дальнейшем, как правило, подлежит конкретизации.

Наследование на диаграмме классов показывается линией со стрелкой в виде не закрашенного треугольника, которая указывает на базовый класс. До-

пускается объединение несколько стрелок в одну для разгрузки диаграммы.

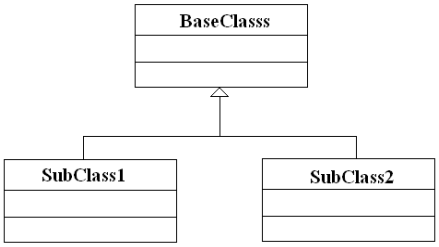

Рисунок 3 – Обозначение наследования между классами

Из диаграммы видно, что классы SubClass1 и SubClass2 являются потомками класса BaseClass.

Отношение «агрегация» подразумевает включение в качестве составной части объектов другого класса (отношение целое/часть). На диаграмме такая связь обозначается стрелкой в виде не закрашенного ромба. При этом стрелка указывает на «целое». В программе для реализации нестрогой агрегации «часть» включается в «целое» по ссылке. Такой компонент может динамически появляться и исчезать.

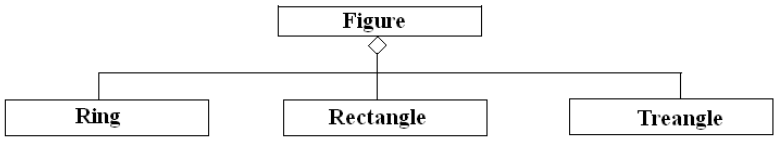

Рисунок 4 – Обозначение отношения «агрегация»

Строгая агрегация имеет название – композиция. Она означает, что компонент не может исчезнуть, пока объект «целое» существует. Композиция обозначается линией со стрелкой в виде закрашенного ромба.

Рисунок 5 – Обозначение отношения «композиция»

Отношение зависимости (использования) между классами показывает, что один класс пользуется некоторыми услугами другого класса. На диаграмме она обозначается пунктирной линией со стрелкой.

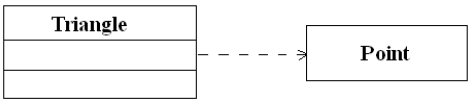

Рисунок 6 – Обозначение зависимости между классами

Современное программирование базируется не только на самих идеях объектно-ориентированного подхода, но и на применении шаблонов проектирования, включающих в себя наиболее удачные решения типичных проблем, возникающих при разработке программ.

Вопросы самоконтроля

1. Дайте определение понятий «объект», «класс»?
2. Какие этапы включает объектно-ориентированного подход?
3. Изучите диаграммы классов, представленные на рисунке 7. Содержит ли представленная диаграмма ошибки? Если да, то в чем они заключаются? Перечислите имена классов, отношения между ними. Каковы отношения между классами?

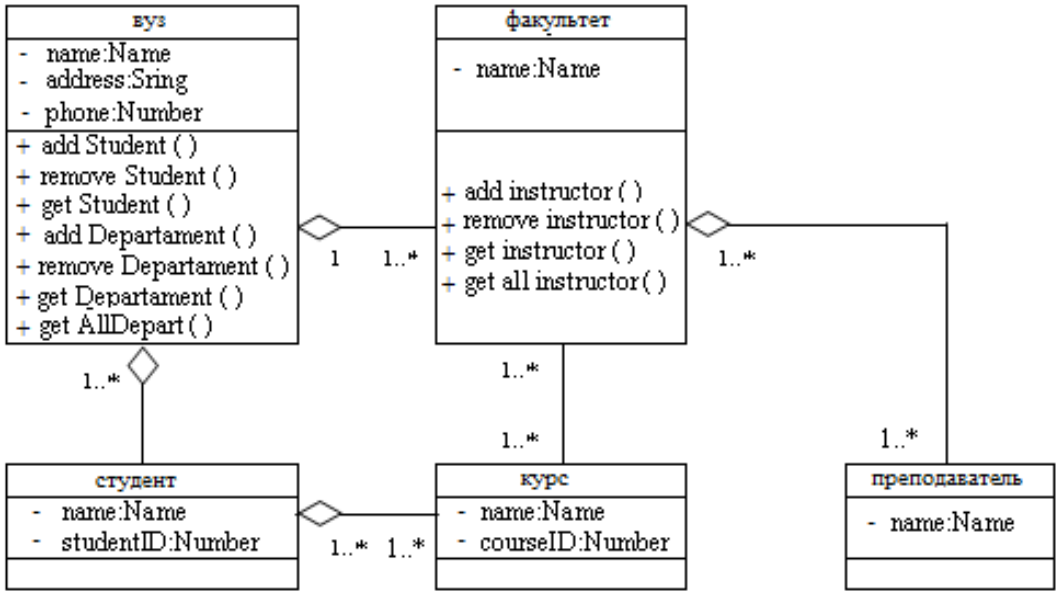

Рисунок 7 – UML-диаграмма классов

4. Изучите диаграмму, представленную на рисунке 8. Ответьте по рисунку на вопросы из предыдущего задания.

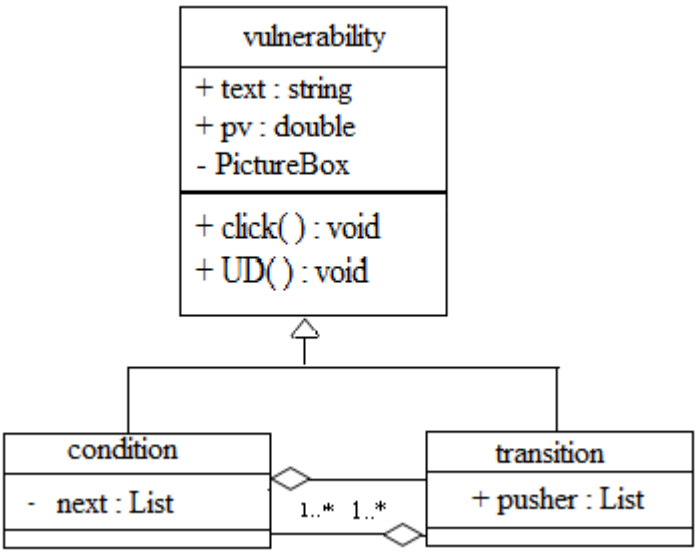

Рисунок 8 – UML-диаграмма классов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В основе объектно-ориентированного программирования (ООП) лежит идея разбиения задачи на группу объектов.

Объект содержит в себе набор данных различных типов, однозначно определяющий некоторый реальный объект, и функции обработки этих данных. Таким образом, создается новый тип, называемый *классом*.

Переменные типа класса, называются объектами (экземплярами класса). На их основе строится вся программа.

Достоинства объектно-ориентированного подхода в программировании заключаются в следующем:

- хорошая структурированность программ, что облегчает понимание их алгоритма;

- возможность разбиения программного продукта на небольшие компоненты и тестирование их по отдельности;

- легкость дальнейшего расширения.

В основе объектно-ориентированное программирование лежат три клю-

чевых принципа: инкапсуляция, наследование, полиморфизм.

Инкапсуляция (сокрытие информации) заключается в определении пользователем новых типов данных. Каждый такой тип (класс) содержит определение набора значений и действий, которые могут быть выполнены над этими значениями. Действия задаются функциями. При этом никакие другие функции, кроме тех, которые определенны внутри класса, не должны иметь доступа к набору значений этого типа.

Наследование – механизм, позволяющий строить иерархию классов. Это предполагает определение базового класса (или прародителя), а затем его использование для построения производных классов (потомков). Причем каждый производный класс наследует все свойства своего прародителя, включая как данные, так и функции. И при этом они могут также иметь свои дополнительные свойства. Таким образом, они не дублируют свойства базовый класс, а только используют его возможности.

Полиморфизм в переводе с греческого означает «много форм». Он заключается в обозначении некоторого общего действия(функции) одним именем, которое используется во всей иерархии типов. Но каждый тип в этой иерархии реализует это действие своим собственным, пригодным только для него способом. В языке C++ полиморфизм имеет две формы:

- перегрузка операций и функций,
- использование виртуальных функций.

Понятие «класс»

Класс является типом данных, определяемым пользователем. Он представляет собой модель реального объекта в виде данных и функций для работы с ними. Класс в языке C++ есть расширение понятие структуры.

Данные, содержащиеся в описании класса, называются его полями, а функции – методами. Поля и методы являются элементами класса. Описание класса выглядит примерно так:

```
class <имя> {
```

```
private:      описание закрытых элементов;
 public:      описание открытых (общедоступных) элементов;
 protected:   описание защищенных элементов;
};
```

Описанию элементов класса предшествуют ключевые слова, являющиеся спецификаторами доступа: private (закрытый), public(открытый) и protected (защищенный).

Закрытые элементы предназначены только для внутреннего использования в классе и обеспечивают принцип инкапсуляции. Как правило, они являются полями. Они доступны только методам класса, в состав которого они входят.

Открытые элементы доступны снаружи класса. Как правило, они отвечают за внешний интерфейс класса и являются методами.

Защищенные элементы доступны только для методов данного класса и классов, производных от него.

При создании класса программист самостоятельно решает, какие из элементов класса будут общедоступными, а какие закрытыми. Однако описание большинства классов имеет типовую схему: закрытая часть содержит данные (поля), а открытая – функции для работы с этими данными (методы). Секция защищенных элементов используется в случаях предположения, что данный класса будет в дальнейшем использоваться в качестве базового класса в наследовании.

В описании класса могут присутствовать многочисленные открытые и закрытые секции сколько угодно раз (в том числе и ни одного) в производном порядке.

Спецификатор доступа не является обязательным. Если он не указан, по умолчанию элемент класса становятся закрытым.

Действие текущего спецификатора в объявлении класса распространяется на все элементы до следующего спецификатора. Объявление элементов

внутри секции соответствует синтаксису объявления переменных и функций языка программирования C++.

Для полей класса справедливы следующие правила:

они могут быть любого типа, кроме типа того же класса (но могут быть ссылками или указателями на класс, в котором объявлены);

поля могут быть объявлены с модификаторами const, но при этом они принимают значение только один раз (с помощью конструктора) и в дальнейшем не могут изменяться;

поля могут быть описаны с модификаторами static, но не auto, extern или register;

не допускается инициализация полей при объявлении;

поля класса доступны всем методам собственного класса напрямую, т.е. передавать их в качестве параметров не требуется.

Рассмотрим объявление класса, созданного для хранения даты и времени.

```
class TimeData {
private:
      unsigned day, month, year, hour, minute;
public:
      void Dispay( );
      void SetTime(unsigned d, unsigned m, unsigned y, unsigned h, unsigned min);
};
```

В рассмотренном примере класс TimeData содержит семь элементов: пять полей – year, math, day, hour, minute и два метода – Dispay( ) и SetTime( ).

Методы класса объявлены прототипами функций. Предположительно, они выполняют какие-то действия над полями. Их тела должны быть описаны позднее. Но их описанию должны обязательно предшествовать имя класса и операция разрешения видимости (::), сообщающая о принадлежности данного метода определенному классу.

```
        // Определение методов
void TimeData::SetTime(unsigned d, unsigned m, unsigned y, unsigned h, unsigned min)
{
        day = d;        month = m;    year = y;
        minute = min;          hour = h;
};
void TimeData::Display( )
{
        cout<< "Дата:"<< day<< "."<< month<< "."<<year<<endl;
         cout << "Время:"<< hour<< ":"<< minute<<endl;
};
```

В примере методы отвечают за задание значений полям и вывод их на экран.

Методы могут содержать полное свое описание внутри определения класса. В этом случае метод является встроенной (inline) функцией.

Встроенную функцию-элемент класса можно определить и вне тела класса, указав в заголовке ее определения ключевое слово inline.

У методов класса существуют отличия от обычных функций:

1. Внутри метода все операторы имеют прямой доступ к элементам своего класса, и передавать их в качестве параметров не требуется.

2. При использовании прототипов в описании метода его имени предшествует имя класса и оператор разрешения области видимости**.** Это позволяет однозначно определить метода, поэтому в программе у различных классов могут методы с одинаковыми именами.

3. Методы класса могут использоваться только элементами собственного класса.

Объект класса – это переменная типа «класс». Можно объявлять сколько угодно объектов одного класса, причем синтаксис их объявления аналогичен объявлению переменных любого другого типа:

class идентификатор класса идентификатор переменной;

Количество объектов в одном объявлении не ограничивается, их можно перечислять через запятую.

Объекты можно передавать в качестве параметров функций или возвращать их как значение функции, объявлять массивы, состоящие из объектов класса и указатели на класс.

Как и любая другая переменная, объект класса может быть динамическим (т.е. создаваться каждый раз, когда управление достигает его объявления, и уничтожаться, когда управление выходит из данного блока) или статическим (т.е. создаваться один раз и уничтожаться по завершению программы).

С другой стороны, язык C++ обрабатывает классы иначе, чем встроенные типы. Большинство встроенных операций не могут применяться к классам. К объектам нельзя напрямую применять арифметические операции и операции сравнения. Но все эти операции применимы для полей классов.

Для самих объектов справедливы следующие встроенные операции: выбор элемента (.), присваивание (=), получение адреса (&), косвенная адресация (*), и операция указания –>.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операции выбора при обращении к элементу через имя объекта и операция –> при обращении через указатель.

Использование методов ранее объявленного класса TimeData возможно в функции main( ) следующим образом:

```
int main ( )
{
 TimeData today;              //объявление объекта today класса TimeData
 today.SetTime( 13, 2, 2016, 21, 12);     //вызов метода SetTime для объекта today
 today.Display( );                        // вызов метода  Display для объекта today
 return 0;
}
```

Методы класса могут вызывать другие функции-элементы того же класса, используя имя функции. Обычно функции или функции-элементы других клас-

сов могут вызывать элементы класса с помощью операций . и ->, применяемых представителю или указателю на представитель класса.

В программе объявление классов, как правило, являются глобальными, т.е. располагаются вне какой-либо функции. При разработке достаточно больших программ с несколькими классами их описание располагаются в отдельном файле, который затем подключается к основному файлу директивой #include.

В следующем примере программы класс Coord содержит встроенные методы.

```
#include <iostream.h>
#include <math.h>
//описание класса
class Coord {
        float x, y;
public:
        void SetCoord(float _x, float _y) {x = _x; y = _y;}
        void GetCoord(float & _x, float & _y) {_x = x; _y = y;}
        int Length( ){ return sqrt(x*x+y*y); }
    };
int main()
{
Coord org;                          // локальный объект
Coord *orgPtr = &org;
//указатель на объект инициализирован адресом ранее объявленного объекта
org.SetCoord(10, 10);               // вызов метода класса
float col, row;
orgPtr->GetCoord(col, row);         //вызов метода через указатель на объект
cout<<“расстояние от точки (”<<col<< “;”<<row<< “ ) равно”<<org. Length( )<<endl;
return 0;
}
```

В примере демонстрируется возможность обращения к закрытым элементам класса через его методы. А также объявление указателя на объект и вызов методов класса через указатели на объект.

Конструкторы

Язык программирования C++ имеет две встроенных особенности при работе с классами – конструкторы и деструкторы – помогающие не забывать про инициализацию объектов и про очистку памяти после завершения работы с ними. Использование в программе конструкторов и деструкторов необходимо во избежание массы досадных ошибок.

Конструктором называют специальный метод класса, внутри которого размещаются действия по заданию начальных значений полям класса до использования объекта

Конструктор носит тоже имя, что и класс, в котором он содержится. Он не возвращает никакого значения, в том числе и типа void. Поэтому в описании его заголовка имя типа не указывается.

Вызов конструкции осуществляется компилятором автоматически при создании каждого экземпляра (объекта) класса. При вызове конструктора с параметрами, их фактические значения должны располагаться в круглых скобках после имени объекта при его объявлении.

Если конструктор в программе не определен, то компилятор автоматически генерирует конструктор, присваивающий всем полям класса нулевые значения. В рассмотренных ранее примерах конструкторы явным образом не создаются, следовательно, они будут созданы компилятором.

Синтаксис объявления конструктора аналогичен синтаксису объявления любого другого метода класса.

Пример.

```
#include <iostream.h>
//описание класса
class  Number {
```

```
private: int x;
           int Get_x( ) {return x;}
public:   Number(int a ) { x=a;  }   //конструктор
           void PrintNumber( ) { cout<< ”x= ”<<x<<endl; }
           void ModifyX(int b) { x+=b; }
         };
int main( )
{ Number A, B(17);
  A. PrintNumber( ); // x= 0
  A.ModifyX(-5);  A. PrintNumber( );         // x= -5
  B. PrintNumber( );            // x= 17
  return 0;
}
```

В примере для объекта с именем А вызывается конструктор, генерирующийся компилятором, поскольку конструктор без параметров в классе не объявлен, и при объявлении объекта А параметры конструктора не указаны.

Выделяют несколько видов конструкторов:

1. конструктор по умолчанию – конструктор без параметров, именно такой конструктор автоматически формируется компилятором;

2. конструктор с параметрами, на количество и типы параметров такого конструктора не накладывается никаких ограничений;

3. конструктор копии, параметром которого является объект, ссылка или константная ссылка на объект собственного класса;

4. конструктор со списком инициализации, в котором указанный список располагается после заголовка конструктора и отделяется от него двоеточием;

5. конструктор преобразования типов также является конструктором с параметрами, которому передается только один параметр, не совпадающий с типом класса, к которому принадлежит конструктор.

Каждый класс может содержать любое количество конструкторов, что позволяет инициализировать поля объекта различным образом. Но в этом слу-

чае они должны различаться набором параметров (иметь уникальную сигнатуру).

Рассмотрим пример объявление класса с несколькими конструкторами.

```
enum color {white, black, auburn, spotted};
class kitten {
        char *name;
        int age;
        color skin;
public:
        kitten ( );                 // конструктор по умолчанию
        //конструкторы с параметрами
        kitten ( int a, color c);
        kitten ( color sk);
        kitten (char *nam);
        kitten ( kitten &K);  //конструктор копии
        void Print( );
};
        // описания конструкторов
kitten :: kitten (kitten &K)              // конструктор копии
{ age=K.age+1; skin=K.skin; strcpy(name, K.name); }

kitten :: kitten  ( color sk)             // конструктор с параметрами
{ switch (sk) {
                case black :   age=10; skin=black;  strcpy (name, “Черныш” );   break;
                case white :   age=2; skin=white; strcpy (name, “Снежок” ); break;
                case auburn :  age=3;  skin=auburn; strcpy (name, “Рыжик” );  break;
                case spotted : age=5; skin=spotted; strcpy (name, “Васька ” );  break;
                default: cout<< “ такой цвет отсутствует”; age=skin=name=0;
         }
}
```

```
kitten:: kitten ( char *nam)            // конструктор с параметром
{ name = new char [strlen(nam) +1];
strcpy (name, nam);
cout<< “введите возраст:”;          cin>> a;
cout<< “окрас”;        cin>>skin;
}
//конструктор со списком инициализации
kitten :: kitten ( int a, color c) :  age(a), skin(c),  name(“Гав”) { };

//конструктор по умолчанию
kitten::kitten ( )        //конструктор по умолчанию
{ age=3; skin=3; strcpy(name, “Мурка”); }
```

В примере в объявлении конструкторов используются прототипы, поэтому в их описании перед заголовком указаны имя типа и операция разрешения видимости.

Особенности использования деструктора

Деструктор является дополнением конструктора. По завершению работы с объектом он и очищает память, которую занимали поля объекта.

Имя деструктора также соответствует имени класса, но перед ним указывается префикс-тильда (~). Деструктор вызывается всякий раз, когда уничтожается представитель класса.

Явное описание деструктора в программах – нечастое явление. Как правило, оно присутствует только при использовании динамической памяти для объекта или его поля. В остальных случаях он создается компилятором.

Вызов деструктора происходит автоматически для всех объектов перед завершением работы программы или блока, в котором объект создается.

```
#include <iostream.h>
class Pair{
```

```
        int first, second;
public:
        Pair ( int one, int two): first(one), second (two)          //конструктор
        {cout<< " объект создан "<< endl;}
        ~Pair ( ) { cout <<" объект удален "<< endl;}        //деструктор
        void out ( ) { cout<< first<< " << second; }
    }
int main( )  {
 Pair num(2,3); num. out( );
Pair num(4,5); num. out( );
return 0; }
```

Выделение динамической памяти одна из важных функций конструктора. Память, выделенная для динамического объекта в конструкторе, освобождается при удалении объекта в деструкторе.

Примером деструктора для рассмотренного класса ранее kitten может являться является следующим:

```
kitten :: ~ kitten( ) {delete [ ] name;}
```

Вызов деструктора никогда не выполняется явным образом. Он всегда осуществляется компилятором автоматически.

Указатель this

Встречаются ситуации, когда внутри метода класса необходимо обратиться к указателю на объект. Это можно осуществить через ключевое слово this, которое содержит в себе адрес объекта, полем которого он является. У каждого объекта указатель this свой.

Указатель this называют неявным указателем, поскольку он автоматически создается компилятором, и тогда каждый объект имеет копию собственных полей. Методы же класса существуют только в единственном экземпляре. При вызове метода ему передается неявный аргумент, который обозначает конкретный объект класса.

Неявный указатель this можно использовать и явным образом, как и любой другой указатель, для работы с полями объекта.

```
class Simple{
                int a;
public:
                Simple( );
                void Greet( ) { cout<< “Hello!”<<endl; }
               };
Simple::Simple( )
{
        this->a=15;
        Greet( );
        (*this).Greet;                  //оба оператора вызывают функцию Greet
};
```

Существует операция присваивания, которая позволяет присваивать объекту значение другого объекта. Операция присваивания является функцией-операцией с именем operator=, который воспринимает единственный аргумент типа ссылки или константной ссылки на собственный класс.

```
Simple& operator = (const Simple &scr);  // прототип
Simple & Coord::operator = (const Simple &scr) //описание операции присваивания
{
  a = scr.a;
   return *this;
}
```

Статические элементы класса

Статическое поле класса существует в единственном экземпляре для всех объектов. При этом любой объект класса может получать и изменять его значение. Статическое поле класса широко применяется для подсчета числа используемых объектов. Например:

```
class race_cars {
private:  static int count;                    //статическое поле
            int number;
            char name[30];
public:    race_cars( ) { count++; }      // при создании объекта count увеличивается
           ~race_cars( ) {count--; }     //при удалении объекта count уменьшается
            int GetCount( ) { return count; }
         };
         int rase_cars :: count = 0;
```

Объявление статического поля производится вне класса. Оно аналогично объявлению глобальной переменной. В классе статическая переменная лишь скрывается от воздействия извне. К статическому полю можно обратиться только через экземпляр класса, хотя оно имеет значение даже тогда, когда ни одного объекта не существует.

Существуют также статические методы. Они не являются полноценными методам, так как могут обращаться только к статическим полям класса. Однако их можно вызывать даже тогда, когда не существует ни одного объекта класса.

```
class A {
private:        static int count;
public:         A( ) { count ++};
                static void display_count( ) { cout<<count;}
};
int A:: count=0;        // объявление статической переменной
int main( )
{
A:: display_count( ); //число объектов до их создания
A a1, a2, a3;
A:: display_count( ); //число объектов после их создания
}
```

Вопросы самопроверки

1. Сколько элементов содержится класс Coord, приведенный в примере? Каковы их спецификаторы доступа?

2. Выберите верную трактовку термина «инкапсуляция»:

а) создание новых классов на базе ранее созданных (базовых), причем новые классы обладают всеми свойствами базовых классов и имеют новые, свойственные только им;

б) создание сложных типов данных, включающих в себя наборы данных и методы для их обработки;

в) обозначение одним именем общего действия во всей иерархии типов, реализуемое в каждом классе собственным способом;

г) определение функций, вызывающих самих себя.

3. Выберите верное утверждение из следующих:

а) закрытые элементы класса традиционно являются функциями;

б) обращение к закрытым элементам класса осуществляется только через методы производных классов;

в) открытые элементы класса доступны только элементам собственного класса и производным от него;

г) спецификатор доступа может обозначаться с помощью ключевых слов private, public, virtual, protected.

4. Что называют методом класса?

а) элемент класса, содержащий данные

б) элемент класса, содержащий функцию

в) функцию, отвечающую за инициализацию данных в классе

г) функцию, отвечающую за уничтожение данных в классе

5. Какой принцип ООП означает обозначение одним именем общего действия во всей иерархии типов?

а) инкапсуляция

б) полиморфизм

в) наследование

г) рекурсия

6. Изучите программный код и сделайте выбор среди приведенных вариантов оценки.

```
class A {
        int x;
public:
        A ( ) {cout<<”x-?”;        cin>>x;}
        void SetX (int  a) { x = a; }
};
int main ( )
{ A *a;
SetX (10);
return 0;}
```

a) для доступа следует писать a -> SetX (10);

b) для доступа следует писать A.SetX (10);

c) для доступа следует писать a.SetX (10);

d) для доступа следует писать A -> SetX (10);

Упражнения

1. Организовать класс *квадратная матрица,* содержащий методы вывода матрицы на экран в общепринятом виде, нахождения транспонированной матрицы и определителя матрицы. Класс должен содержать несколько конструкторов для инициализации полей объекта: заданием значений с клавиатуры, с помощью счетчика случайных чисел и в виде нулевой и единичной матриц. В программе продемонстрировать работу всех методов класса.

2. Организовать класс *дробь* с полями числитель и знаменатель, содержащий методы: вывод дроби в общепринятом виде, выделение целой части, приведение дроби к несократимому виду. Класс должен содержать конструкторы: по умолчанию, с параметром, копии. В программе продемонстрировать работу всех методов класса.

3. Описать класс *число*, содержащий значение числа и вид его представления (десятичное, восьмеричное, двоичное), методы вывода перевода числа из одного представления в другое. Класс должен содержать конструкторы: по умолчанию, с параметром, копии. В программе продемонстрировать работу всех методов класса.

4. Описать класс *вектор* на плоскости, содержащий координаты его начала и конца, а также методы: вывод координат на экран, нахождения длины вектора, конструкторы различного типа. В программе продемонстрировать работу всех методов класса.

5. Описать класс *вектор* на плоскости, содержащий его координаты и методы: вывода координат на экран, определение величины угла, образованного вектором с осью OX, конструкторы различного типа. В программе продемонстрировать работу всех методов класса.

6. Описать класс *многочлен*, с полями: степень, аргумент и коэффициенты. Создать методы: вычисление значения многочлена от аргумента, вывода многочлена в общем виде на экран, конструкторы различного вида. В программе продемонстрировать работу всех методов класса.

7. Создать класс *комплексное число*. Класс должен содержать методы: нахождение аргумента комплексного числа, определение значения модуля комплексного числа, вывода комплексного числа на экран в общепринятом виде, конструкторы различного типа. В программе продемонстрировать работу всех методов класса.

8. Организовать класс *дата,* содержащий данные – число, месяц, год. Создать метод, проверяющий правильность введенной даты, метод вывода даты на экран в формате день.месяц.год, конструкторы различного типа. В программе продемонстрировать работу всех методов класса.

9. Описать класс *множество* с полями элементы множества и количество элементов, позволяющий добавлять и удалять элементы из множества, вывод всей элементов множества на экран, конструкторы различного типа. В программе продемонстрировать работу всех методов класса.

10. Описать класс *одномерный массив*, содержащий его элементы и их количество, а также метод вывода всех значений на экран и нахождения минимального и максимального его элементов, конструкторы различного типа. В программе продемонстрировать работу всех методов класса.

НАСЛЕДОВАНИЕ

Простое наследование

Простое наследование описывает родство между двумя класса. Класс, являющийся родителем, называется *базовым* классом. Класс, созданный на его основе, является *производным* классом (потомком). Производный класс, обладая всеми характеристиками базового класса, может обладать новыми свойствами, характерными только для него.

Класс может стать базовым многократно – на его основе можно создавать множество классов. Производный класс, в свою очередь, сам может стать базовым. Таким способом создается иерархия классов.

Синтаксис механизма наследования следующий:

class новый идентификатор: ключ доступа идентификатор базового класса

{тело класса};

Ключ доступа обозначается теми же ключевыми словами, что и спецификаторы доступа для элементов класса.

Разница между использованием различных ключей доступа при объявлении наследования представлена в таблице 1.

Таким образом, закрытые элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним в производном классе возможно и осуществляется только через методы базового класса.

Защищенные элементы базового класса при наследовании с ключом доступа private становятся в производном классе закрытыми, а в остальных случаях доступ к ним не изменяется.

Доступ к открытым элементам при наследовании становится соответст-

венным ключу доступа.

Таблица 1

Ключ доступа	Спецификатор доступа элемента базового класса	Доступ к этому элементу в производном классе
private	private	доступ отсутствует
	protected	private
	public	private
protected	private	доступ отсутствует
	protected	protected
	public	protected
public	private	доступ отсутствует
	protected	protected
	public	public

Указание ключа доступа не является обязательным, по умолчанию принимается private.

Производный класс наследует из базового класса поля и методы, а также деструктор, но не конструкторы и операции присваивания, их нужно заново определять.

Считается хорошим стилем программирования вызывать конструктор базового класса в конструкторе производного класса, что уменьшает повтор фрагментов кода, повышает логичность построения программы и упрощает дальнейшую модификацию.

Пример.

```
class Base {
private:
    int count;                    // закрытое поле класса
public:
    Base( ) { count = 0; }              // конструктор
    void SetCount(int n) { count = n; }
```

```
int GetCount( ) { return count; }
};
```

Если в дальнейшем требуется класс, имеющий все свойства Base, но, дополнительно, обладающий способностью изменения значения поля объекта на заданную величину, можно объявить производный класс следующего вида:

```
class Derived: public Base {
public:
    Derived( ):Base( ){  };
    void ChangeCount(int n) { SetCount( GetCount( ) + n ) };
    };
```

Порядок вызова конструктора в производном классе определяется следующими правилами. Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть без параметров). Если конструктор базового класса требует указания параметров, он должен быть вызван явным образом в конструкторе производного класса в списке инициализации.

Для иерархии классов, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После выполняются конструкторы элементов классов, которые являются объектами, в порядке их объявления в классе.

В случаях нескольких базовых классов их конструкторы вызываются в порядке объявления классов.

Вызов методов базового класса также предпочтительнее копирования фрагментов кода из базового класса в производный.

Вызов деструкторов в производном классе также имеет свои особенности:

1. если в производном классе деструктор не объявлен, он формируется по умолчанию и вызывает деструкторы всех базовых классов.

2. отличие от конструкторов, при создании деструктора в производном классе не требуется явного вызова деструкторов базовых классов.

3. для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем – деструкторы элементов класса, а потом деструктор базового класса.

Пример.

```
#include<iostream.h>
class angle {
protected:      int degree, minute;
public:         angle ( ){      cout<< “введите величину угла”<<endl;
                                cout<< “градусы:”; cin>>degree;
                                cout<< “минуты:”; cin>>minute;
                            }
                angle ( int d, int m ) { degree=d; minute=m; }
                 void modify_angle (int  additives)
                { additives+= 60*degree + minute;

        degree= additives/60;
        minute= additives%60;
         }
       void display( )
        { cout<< “градусы:”<<degree<<endl<< “минуты:”<<minute<<endl;
        }
};
class exact_angle: public angle {
private: int second;
public:
        exact_angle( ): angle ( ) {     cout<< “секунды:”; cin>>second;  }
        exact_angle(int d, int m, int s ): angle ( int d, int m ) { second=s; }
        void modify_angle (int  add)
        { add+= 3600*degree + 60*minute + second;
```

```
        degree= add/3600;
        second= add%60;
        minute= (add – second)%3600;
         }
       void display( )
        { angle::display(  );
        cout<< “секунды:”<<second<<endl;
        }
};
int main( ) {
angle A(45, 16);
A. modify(48);   A. display();
exact_angle E(181, 36, 12);   E. modify(27);   E. display();
return 0;
}
```

Если элемент производного класса имеет то же имя, что и элемент базового класса, для объекта производного класса используется представитель производного класса, а для объекта базового класса – метод базового. Этот механизм называется *замещением.* Если для объекта производного класса необходимо вызвать одноименный метод базового класса следует использовать операцию разрешения видимости, уточняющую какой из экземпляра метода нужно применять.

Множественное наследование

Множественное наследование позволяет создавать новый класс из нескольких базовых. При этом в объявлении производного класса в заголовке после двоеточия следует перечислить через запятую имена всех базовых классов с соответствующими ключами доступа. Например:

```
class D: public A, public B, public C {… };
```

Множественное наследование отличается от простого лишь тем, что про-

изводный класс наследует все свойства всех перечисленных классов.

Также, как и в простом наследовании, во множественном – необходимо инициализировать все поля базовых классов. Для этого в конструкторе производного класса вызываются конструкторы всех базовых классов. Это реализуется перечислением их вызовов в заголовке конструктора производного класса, после двоеточия.

D( ): A( ), B( ), C( ){. . .};

В случае конфликта имен элементов базовых классов следует использовать оператор разрешения видимости.

Вопросы самопроверки

1. Для чего используют конструкторы и деструкторы?
2. В чем отличия конструктора от обычных методов?
3. Каковы правила объявления конструктора?
4. В чем отличие простого и множественного наследования?
5. Выберите правильный вариант результата работы программы:

```
class A {
        int a;
public:  A( ) {cout<< “construct A ”;}
        ~A( ) {cout<< “destruct A ”;}
};
class B : class A {
public:   B( ) {cout<< “construct B ”;}
          ~B( ) {cout<< “destruct B ”;}
};
int  main ( ) {
B b;
return 0; }
```

а) ничего не выведется

б) construct B destruct B

в) construct A destruct A construct B destruct B

г) construct A construct B destruct B destruct A

д) construct B construct A destruct A destruct B

6. Выберите верное утверждение:

а) закрытые элементы базового класса в производном классе недоступны;

б) обращение к закрытым элементам класса осуществляется только через методы производных классов;

в) открытые элементы класса доступны только производным классам;

г) спецификатор доступа может обозначаться с помощью ключевых слов private, public, virtual, protected.

7. Какое из определений соответствует термину «множественное наследование»?

a) создание новых классов на основе одного базового класса

b) создание в одной программе нескольких новых классов на основе множества классов

c) создание нового класса на основе нескольких базовых классов

d) создание нескольких классов в одной программе

Упражнения

1. Измените программу, выполненную в упражнения 1 из предыдущего раздела – организуйте на основе класса квадратная матрица производный класс. Он дополнительно должен содержать содержащий методы н сложения матриц и произведения матриц.

2. Измените программу, выполненную в упражнения 2 из предыдущего раздела – организуйте на основе класса дробь производный класс, содержащий дополнительно методы вычисления сложения, умножения и деления дробей.

3. Измените программу, выполненную в упражнения 3 из предыдущего раздела – организуйте на основе класса число производный класс, дополни-

тельно содержащий методы сложения и вычитания чисел.

4. Измените программу, выполненную в упражнения 4 из предыдущего раздела – организуйте на основе класса вектор производный класс, дополнительно содержащий методы умножения вектора на число, вычитания и сложения двух векторов.

5. Измените программу, выполненную в упражнения 5 из предыдущего раздела – организуйте на основе класса вектор производный класс, дополнительно содержащий методы умножения вектора на число, вычитания и сложения двух векторов.

6. Измените программу, выполненную в упражнения 6 из предыдущего раздела – опишите на основе многочлен производный класс, позволяющий умножать многочлен на число, складывать и вычитать многочлены.

7. Измените программу, выполненную в упражнения 7 из предыдущего раздела – опишите на основе класса комплексное число производный класс, содержащий методы умножения, деления, сложения и вычитания комплексных чисел.

8. Измените программу, выполненную в упражнения 8 из предыдущего раздела – организуйте на основе класса дата производный класс, дополнительно включив в дату день недели. Опишите функцию, добавления к указанной дате указанного количества дней, переопределите функцию вывода даты на экран с указанием дня недели.

9. Измените программу, выполненную в упражнения 9 из предыдущего раздела – организуйте на основе класса множество производный класс, дополнительно методы объединения и пересечения множеств.

10. Измените программу, выполненную в упражнения 10 из предыдущего раздела – организуйте на основе класса одномерный массив производный класс, дополнительно содержащий методы умножения элементов на число, нахождения минимального и максимального значений в массиве.

РЕАЛИЗАЦИЯ ПРИНЦИПА ПОЛИМОРФИЗМА В ЯЗЫКЕ С++

Виртуальные функции

Принцип полиморфизма позволяет родственным объектам могут вести себя по-разному. Для этого один и тот же метод базового класса переопределяют в каждом классе потомке различными способами.

Такие функции принято объявлять в родительском классе и классе потомке с атрибутом virtual. Он размещается перед заголовками функции.

Отличие переопределенных функций от виртуальных заключается в *стратегии динамического связывания*. Она применяется компилятором ко всем виртуальным методам и означает, что решение, какой из экземпляров виртуального метода должен быть вызван, принимается не на этапе компиляции, а на этапе выполнения, когда уже известен тип объекта.

Для реализации стратегии динамического связывания виртуальная функция должна вызывается только через указатель или ссылку на базовый класс.

Виртуальные методы наследуются, поэтому переопределять их в производном классе требуется только в случае задания различающихся действий.

Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ и к варианту метода базового класса с помощью операции разрешения видимости.

Виртуальный метод не может быть статическим, но может быть дружественным.

Пример.

```
#include <iostream.h>
class Base {
protected:      int x;
public:         Base (int y) {x=y;}
                void PrintX( ) { cout<< “x =”<<x;}
                virtual ModifyX( ) {x *= 2;}
};
```

```
class Derived : public Base {
public: Derived ( int f) :  Base( f ) { }
virtual ModifyX( ) {x/=2; }
};
int main ( )
{
Base b(10); Derived d(10);
b.PrintX( ); d.PrintX( );
Base *pB;              //указатель на базовый класс
pB=&b;                 // связывание указателя с объектом базового класса
pB-> ModifyX( );       // вызов виртуального метода для объекта базового класса
pB-> PrintX( );
pB=&d;                 // связывание указателя с объектом производного класса
pB-> ModifyX( );       // вызов виртуального метода для объекта производного класса
pB-> PrintX( );
return 0; }
```

Программа выведет на экран монитора: x=10 x=10 x=20 x=5

Виртуальная функция, объявленная в базовом классе иерархии порождения, может совсем не использоваться для объектов этого класса, и даже не иметь определения. Для выделения этой особенности ее приравнивают к нулю. В этом случае функция называется *чисто виртуальной*.

Класс, содержащий одну или более чисто виртуальных функций, называется *абстрактным*. Абстрактный класс может использоваться только в роли базового класса для порождений потомков. Создавать объекты абстрактного класса и использовать его в качестве типа значения, возвращаемого функцией, и типов параметров функции нельзя.

Пример

```
#include <iostream.h>
#include <math.h>
const float Pi=3.1435;
```

```
class Shape {
protected:      float radius, height;
public:
                Shape (float r, float h) { radius=r; height=h;}
                virtual float S( ) = 0;
                virtual float V( ) = 0;
};
class Cylinder : public Shapes {
public:         Cylinder(float r, float h) : Shapes( r, h) { }
                virtual float S( ) {return 2 * Pi * r *(r +h); }
                virtual float V( ) {return Pi * r * r * h; }
};
class Cone : public Shapes {
public:         Cone(float r, float h) : Shapes(r, h) { }
                virtual float S( ) {return Pi *r*(r +sqrt(r*r+h*h)); }
                virtual float V( ) {return Pi *r*r*h/3;}
};
```

На основе приведенных описаний невозможно создать объекта класса Shapes, поскольку он содержит чисто виртуальные функции. Объявлять же объекты классов Cylinder и Cone возможно, так как виртуальные функции S( ) и V ( ) в них переопределены.

Если базовый класс содержит хотя бы один виртуальный метод, то рекомендуется снабжать этот класс виртуальным деструктором, даже если он ничего не делает. Это предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс.

Перегрузка операций

В языке C++ допускается использование перегруженных функций. Под перегрузкой функций (не обязательно методов класса) понимается создание нескольких прототипов функции, имеющих одинаковое имя. Компилятор

различает их по набору аргументов. Перегруженные функции оказываются весьма полезными, когда одну и ту же операцию необходимо выполнить над аргументами различных типов. Какую именно функцию требуется вызвать, компилятор определяет по типу фактических параметров. Этот процесс называется разрешением перегрузки. Тип возвращаемого значения в разрешении перегрузки не участвует.

Разрешается осуществлять не только перегрузку функций, но и операций. В большинстве языков программирования концепция перегрузки операций, реализована в неявном виде. Так, например, оператор суммирования позволяет складывать значения разных типов.

В создаваемом классе можно изменить свойства большинства стандартных операторов, таких как +, -, *, /, заставив их работать не только с данными базовых типов, но и также с объектами.

Перегружать явным образом можно большинство операций, за исключением: . ?: # ## sizeof.

Перегрузка осуществляется с помощью методов специального типа (функций-операций). Синтаксис функции-операции:

тип operator операция (список параметров) {тело функции}

Например,

```
#include <iostream.h>
class angle {
    int degree, minite, second;
public: angle  (int d, int m, int s): degree(d), minite(m), second(s) { }
    bool  operator > ( angle &a)          // перегруженный оператор >
    { if (3600*degree + 60*minite + second>3600*a.degree + 60*a.minite + a.second)
            return 1;
                else return 0;
     }
    angle operator + ( angle &a)
    { angle result;
```

```
        result. second=3600*(degree+a.degree)+60*(minite + a.minite)+ second+a.second;
         result.degree= result. second /3600;
        result.minute= (result. second – 60* result.degree)/60;
        result. second% = 60;
        return result;
         }
        angle operator –= ( angle &a)
        {
        second=3600*(degree–a.degree) + 60*(minite – a.minite)+ second – a.second;
         degree= second /3600;
        minute= (second – 60* degree)/60;
        second% = 60;
        return this;
         }
        void display( )
        { cout<< “градусы:”<<degree<<endl<< “минуты:”<<minute<<endl;
        }
        };
int main ( )
{
        angle A(45, 18, 39), B(184, 34, 0), C(56, 10, 7), D;
        if (B>C) D= C + A; else D= B + A;
        D. display( );
        B –=A;
        B. display( );
        return 0;
}
```

На перегрузку операторов накладываются следующие ограничения:

они должны сохраняться количество аргументов, приоритет оператора и порядок группировки его операндов, используемые в стандартных типах дан-

ных;

невозможно изменить синтаксис вызова оператора;

нельзя переопределять смысл стандартного оператора применительно к базовым классам переопределять.

Вопросы самопроверки

1. Какой класс называют абстрактным?

а) класс, не содержащий встроенные методы

б) класс, содержащий встроенные методы

в) класс, имеющий чистые виртуальные функции

г) класс, не имеющий виртуальные функции

2. Можно ли создать объект абстрактного класса? Почему?

3. Какие операции нельзя перегружать?

4. Во фрагменте программы приводится описание класса угол. Заполните пропуски.

```
class angle {
    int degree, minite, second;
public:     angle ( ) { degree= minute= second=0; }
            angle ( int d, int m, int s) { degree= d; minute=m;  second=s; }
            angle operator _____ ( _______a) {          // перегруженный оператор
second =3600*(degree + a.degree)+ 60*(minute+a.minite) +second+ a.second;
degree=second/3600; minute= (second – 60* degree)/60; second%= second;
return *this; }
};
```

5. В чем заключается различия между переопределением методов в производных классах и определением виртуальных функций?

Упражнения

1. Создать абстрактный класс *средство передвижения*. На его основе реализовать классы *самолет*, *машина*, *корабль*. Все классы должны хранить па-

раметры средств передвижения: скорость, расход топлива, наименование производителя, год выпуска, метод вывода на экран всех данных, определения срока службы. Индивидуально для самолета указать высоту и максимальную дальность полета, для машины – объем двигателя, для самолета и корабля – количество посадочных мест, для корабля – водоизмещение.

2. Создать абстрактный класс *линия второго порядка* с полями – коэффициенты уравнения второго порядка. На его основе создать классы окружность, парабола (с методом нахождения директрисы), гипербола, эллипс (с методом нахождения эксцентриситета). Предусмотреть виртуальные методы нахождения центра (вершин или фокусов) линий и функции вывода данных на экран.

3. Описать абстрактный класс *фигура на плоскости*. На его базе создать классы круг, треугольник, прямоугольник. Предусмотреть виртуальные методы создания объектов, вычисление площади фигур, периметра для треугольника и прямоугольника, длины окружности – для круга.

4. Создать абстрактный класс *правильный многогранник* с полями длина ребра и число ребер. На его основе создать классы тетраэдр, куб, октаэдр (восьмигранник). Предусмотреть виртуальные методы создания объектов, вычисления их площади поверхности и объема.

5. Создать абстрактный класс *вектор*. На его основе создать классы вектор на плоскости, в трехмерном пространстве, в пятимерном пространстве. Предусмотреть виртуальные методы создания объектов, вычисления их длины, вывода на экран их координат.

6. Организовать класс *треугольник*, с полями: длины сторон и методами нахождения периметра треугольника и виртуального метода нахождения площади (по формуле Герона), конструктором. Создать производный класс, переопределив функцию нахождения площади по высоте и основанию.

7. Организовать класс *вектор на плоскости*, с полями - координаты и методами нахождения длины вектора и виртуального метода скалярного произведения векторов, конструктором. Создать производный класс, переопределив

функцию нахождения скалярного произведения по формуле через длины векторов.

8. Измените программу упражнения 2 из предыдущего раздела, определив производном классе перегруженные операции больше, меньше, равно.

9. Измените программу упражнения 3 из предыдущего раздела, определив производном классе перегруженные операции сложения и вычитания чисел.

10. Измените программу упражнения 6 из предыдущего раздела, определив производном классе перегруженные операции сложения, вычитания и составное присваивание со сложением.

СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА С++

Потоковые классы С ++

Библиотека С++ содержит три класса с помощью которых можно управлять файловым вводом-выводом:

ifstream подключает к программе файл, предназначенный для ввода данных (входной файловый поток)

ofstream подключает к программе файл, предназначенный для вывода данных (выходной файловый поток)

fstream подключает к программе файл, предназначенный как для ввода, так и для вывода

Для подключения данных классов необходимо подключить файл ifstream.h. Для создания объекта класса ifstream и связи с ним файла, находящегося в текущем каталоге, требуется записать следующее:

```
ifstream f (“text.txt”, ios::in);
```

Итак, создан объект с именем ifsin класса ifstream. С ним связан файл text.txt. Если файл, с которым связан объект, находится не в текущем каталоге необходимо полностью указать путь его расположения. Вторым аргументом указывается один из следующих флагов:

Флаг	*Назначение*
ios :: in	Файл открывается для чтения, его содержимое не открывается
ios :: out	Файл открывается для записи
ios :: ate	После создания объекта маркер текущей позиции устанавливается в конец файла
ios :: app	Все выводимые данные добавляются в конец файла
ios :: trunc	Если файл существует, его содержимое очищается автоматически

Для объектов рассмотренных классов определено множество методов. Самыми часто используемыми из них являются:

get (c) – получение символа из потока и запись его в c;

put(c) – выводит символ в поток;

seekg(pos) – устанавливает текущую позицию чтения в значение pos;

tellg( ) – возвращает текущую позицию чтения (записи) в поток;

close( ) – закрытие потока.

Пример

```
#include <fstream. h>
#include <iostream. h>
int main ( )
{
 char ch;
ifstream fin (“text.txt”, ios :: in);
if   (!fin) cout<<” Невозможно открыть файл”<<endl;
ofstream fout (“text1.txt”, ios :: out);
if   (!fout ) cout<<” Невозможно открыть файл”<<endl;
while (fout && fin.get(ch) );
fout.put(ch);
fin. close( );
fout. close( );
return 0;}
```

В программе демонстрируется создание потоков ifstream и ofstream для

обмена данными между файлами.

Шаблоны классов

Шаблоны классов поддерживают парадигму обобщенного программирования, т.е. программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении класса или функции.

В дальнейшем шаблон класса может быть использован для создания классов. Процесс генерации компилятором определения конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием (актуализацией) шаблона.

Определение шаблона класса имеет следующий синтаксис:

template <параметры шаблона> class имя класса {тело} ;

Параметры шаблона перечисляются через запятую. В их качестве могут использоваться не только типы и переменные, но и шаблоны.

Типы, используемые в шаблонах, могут быть как встроенными, так и определенные пользователем. Внутри шаблона параметр типа может применяться в любом месте, где допустимо в дальнейшем использовать спецификацию типа.

Пример. Для представления точки на плоскости разработан класс Point, в котором координаты задаются двумя числами типа double. А в другом приложении требуется задать точки для целочисленной системы координат (тип int). Поэтому сначала объявлен шаблон класса.

```
template <class T> class Point {
private:    T x, y;
public:     Point (T a, T b) : x(a), y(b) {}
            void show( ) {cout<< “(“<<x<<”,”<<y<<”)”<<endl; }
};
```

Префикс template <class T> означает, что объявлен шаблон класса, в котором T – некоторый абстрактный тип. T – параметр шаблона. Вместо T может использоваться любое имя типа. После своего объявления T используется внут-

ри шаблона, аналогично именам обычных типов.

Вместо template <class T> class Point можно писать конструкции вида template <typename T> class Point, но первый вариант считается более распространенным.

При создании шаблона класса в программе генерация классов не происходит немедленно. Для этого нужно создать экземпляр шаблонного класса, который создается либо объявлением объекта Либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией new.

Point <int> anyPoint(-13, 5);

Point <double> otherPoint=new Point<double>(-13.56789, 5.65478);

Возможно вынесение определений шаблона в отдельный файл, а затем его подключение к программе директивой препроцессора.

Для создания шаблона для массивов из n элементов возможна следующая конструкция.

template <class T, int n> class Array {…}

Актуализация данного шаблона:

Array<Point, 20> array; // массив из 20 элементов

В этом случае параметры class T, int n могут рассматриваться как формальные параметры шаблона, на место которых при компиляции встанут конкретные значения.

Методы шаблона автоматически становятся шаблонами функций. Если метод описывается вне шаблона, его заголовок должен иметь следующие элементы:

template <описание параметров шаблона>

тип имя класса <параметры шаблона>:: имя функции (параметры)

Правила описания шаблонов:

Локальные классы не могут содержать шаблоны в качестве своих элементов;

шаблоны методов не могут быть виртуальными;

шаблоны классов не могут содержать статические элементы, дружественные функции и классы;

шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также, в свою очередь являться базовыми для шаблонов и обычных классов;

внутри шаблонов нельзя объявлять дружественные шаблоны.

Шаблоны являются мощным и эффективным средством обращения с различными типами данных. К недостатком использования шаблонов можно отнести то, что программа с шаблонами должна содержать полный код для каждого порождаемого типа, что значительно увеличивает размер исполняемого файла.

Стандартная библиотека C++ содержит достаточно большой набор шаблонов.

Контейнерные классы

Контейнерные классы – это классы, предназначенные для хранения данных, организованных определенным образом. К ним относятся массивы, линейные списки, стеки и другие. Для каждого типа контейнера определены методы работы с его элементами, независимо типа элементов данных, хранимых в контейнере. Поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована стандартной библиотекой шаблонов (STL – Standard Template Library) языка C++.

Использование контейнеров позволяет повысить надежность, универсальность, переносимость программ, уменьшить время их разработки, но за это приходится расплачиваться снижением их быстродействия.

Все контейнеры делятся на два класса: последовательные и ассоциативные. Последовательные обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности. К ним относятся векторы (vector), двусторонние очереди (deque), списки (list), стеки (stack) и очереди с приоритетами (priority_queue). Шаблоны указанных контейнеров хранятся в одноименных библиотеках языка C++.

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Они построены на основе сбалансированных деревьев. Это словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset), битовые множества (bitset).

На основе контейнеров стандартной библиотеки можно создавать собственные контейнерные классы.

Контейнерные классы обеспечивают стандартизированный интерфейс их применения. Смысл одноименных операций для различных контейнеров одинаков. Стандарт определяет только интерфейс контейнеров, поэтому их реализации могут отличаться.

Практически в любом контейнерном классе определены следующие поля указанных типов:

value_type	тип элемента контейнера
size_type	тип индексов, счетчиков элементов и т.д.
iterator	итератор
const_iterator	константный итератор
reverse_iterator	обратный итератор
const_reverse_iterator	константный обратный итератор
reference	ссылка на элемент
const_reference	константная ссылка на элемент
key_type	тип ключа (для ассоциативных контейнеров)
key_compare	тип критерия сравнения (для ассоциативных контейнеров)

Термин «итератор» является аналогом указателя на элемент. Он может применяться для прохода по элементам контейнера в прямом и обратном направлениях. Когда значения элементов контейнера не предполагается изменять применяют константные итераторы.

В помощь для работы с итераторами определено несколько методов, представленных в таблице 2.

В каждом контейнере эти типы и методы определяются способом, зависящим от их реализации.

Таблица 2

iterator begin( ) const_iterator begin( ) const	указывают на первый элемент
iterator end( ) const_iterator end( ) const	указывают на элемент за последним
reverse_iterator rbegin( ) const_reverse_iterator rbegin() const	указывают на первый элемент в обратной последовательности
reverse_iterator rend( ) const_reverse_iterator rend( ) const	указывают на элемент, следующий за последним в обратной последовательности

Во всех контейнерах определены методы, позволяющие получить сведения о размере контейнеров:

size( ) – число элементов,

max_size( ) – максимальный размер контейнера,

empty( ) – булевская функция, отвечающая на вопрос: пуст ли контейнер.

Последовательные контейнеры

Вектором называют структуру, которая позволяет обеспечить эффективный произвольный доступ к своим элементам, добавление и удаление из конца структуры.

Двусторонняя очередь в дополнение к вектору разрешает удаление и добавление с обеих ее сторон.

Список эффективно реализует вставку и удаление элементов в произвольное место своей структуры, но не обеспечивает доступа к этим элементам. Операции последовательных контейнеров представлены в таблице 3.

Обращение к встроенным методам выполняется аналогично обращению к обычным методам класса – операцией доступа (точкой).

Таблица 3

Операция	Метод	vector	deque	list
Вставка в начало	push_front	-	+	+
Удаление из начала	pop_front	-	+	+
Вставка в конец	push_back	+	+	+
Удаление из конца	pop_back	+	+	+
Вставка в произвольное место	insert	+	+	+
Удаление из произвольного места	erase	+	+	+
Произвольный доступ к элементу	[ ], at	+	+	-

Пример. Программа считывает и выводит на экран ряд целых чисел, хранящийся в файле.

```
#include <fstream.h>
#include <vector>
int main( ) {
ifstream in (“primer.txt”);
vector <int> v;
int x;
while (in.eof( ) )
in>>x;
v.push_back(x);
for (vector<int>::iterator i=v.begin(); i!=v.end(); ++i)  cout<<*i<< “ “;
}
```

В примере для создания вектора используется конструктор по умолчанию. Можно пользоваться и другими конструкторами:

```
explicit vector(); // конструктор по умолчанию
explicit vector(size_type n, const T& value=T()); //создается вектор длины n и заполняется одинаковыми элементами – копиями значения value
vector (const vector<T>& x); //конструктор копирования
```

Ключевое слово explicit используется, чтобы при создании объекта ис-

ключить выполнение неявного преобразования при присваивании значения другого типа. Например,

vector <int> v2(10, 1); // вектор из 10 элементов равных единице

vector <int> v4 (v2); // создается вектор v4 равный v2

Другой пример обработки векторов.

```
#include <fstream>
#include <vector>
using namespase std;
int main() {
double arr[]= {1.1, 2.2, 3.3, 4.4 };
int n=sizeof(arr)/sizeof(doule);
vector<double> v1(arr, arr+n);
vector<double> v2;    //пустой вектор
v1.swap(v2);          //обменять содержимым v1 и v2
while (!v2.empty() )
        {
        cout<<v2.back()<< ' '; //вывод последнего элемента
        v2.pop_back();      //удаление элемента
        }
return 0; }
```

Двусторонние очереди и списки

Двусторонняя очередь (deque) является последовательным контейнером, который наряду с вектором, поддерживает произвольный доступ к элементам и обеспечивает вставку и удаление из обоих концов очереди. Указанные операции выполняются за время, пропорциональное количеству перемещаемых элементов, что обеспечивается эффективным способом построения очереди.

Для обеспечения произвольного доступа к элементам очередь разбивается на блоки, доступ к каждому из которых осуществляется через указатель.

Для инициализации двусторонней очереди используются конструкторы, аналогичные конструкторам вектора.

deque <int> D1 (20, 1);

//создается очередь из 20элементов, каждый из которых равеных единице

deque <int> D2 (D1); //вызывается конструктор копии

В шаблоне deque определены: операция присваивания, функция копирования, итераторы, операция сравнения, операции и функции доступа к элементам и изменения объектов.

Дополнительно определены функции добавления и выбораэлемента из начала очереди:

void push_front()(const T& value);

void pop_front();

При выборке элемент удаляется из очереди.

Для очереди не доступны функции capacity и reserve, но можно применять функции resize и size.

В списке не поддерживается произвольного доступа к своим элементам, но выполняются их вставка и удаление. Каждый узел списка содержит ссылки на последующий и предыдущий его элементы. Контейнер список поддерживает конструкторы, операцию присваивания, методы копирования, операции сравнения и итераторы

Доступ к элементам списков ограничивается методами: referencefront(); const_reference front() const; reference back(); const_reference back() const;

Для списков определено несколько собственных методов. Первый из них, splice – соединение списков, служит для перемещения элементов из одного списка в другой без перераспределения памяти, только за счет указателей:

void splice(iterator position, list<T>& x);

void splice(iterator position, list<T>& x, iterator i);

void splice(iterator position, list<T>& x, iterator first, iterator last);

Оба списка должны иметь элементы одного типа. В первой форме метода вставка в вызывающий список осуществляется перед элементом, позиция ко-

торого указана первым параметром, всех элементов списка, указанного в качестве вторым параметра. Элементы второго списка при этом обнуляются.

Не разрешается дублировать список, вставкой в себя.

Вторая форма переносит элемент, заданный третьим параметром, из списка x в вызывающий список.

Третья форма функции переносит из списка в список несколько элементов, диапазон которых задается третьим и четвертым параметрами.

```
#include <list>
using namespase std;
int main() {
list<int> SPISOK;
list<int>::iterator i, j, k;
for(int i=0; i<5; i++) SPISOK.push_back(i+1);
for(int i=12; i<14; i++) SPISOK.push_back(i);
cout<< «Исходный список»;
for(i=SPISOK.begin( ); i!=SPISOK.end( ); ++i) cout<<*i<< “  “;
cout<<endl;
i=SPISOK.begin( ); i++;
k=SPISOK.end( );
j=--k; k++; j--;
SPISOK.splice(i, L1, j, k);
cout<< “Список после сцепки”;
for(i=L1.begin(); i!=L1.end(); ++i) cout<<*i<< “  “;
}
```

К остальным специфическим методам контейнера список относятся:

remove ( ) – удаление элемента по его значению:

```
void remove(const T& value);
```

sort( ) – упорядочивание элементов списка по возрастанию:

```
void sort();
```

unique( ) – сохранение в списке только первого элемента из каждой серии иду-

щих подряд одинаковых элементов.

merge( ) – слияние списка в упорядоченном виде:

void merge(list<T>& x);

Вопросы самопроверки

1. Соотнесите названия библиотек с их содержимым:

а) ifstream 1) класс входных файловых потоков

б) ofstream 2) класс двунаправленных файловых потоков

в) fstream 3) класс выходных файловых потоков

2. Какие контейнеры обеспечивают быстрый доступ к данным по ключу, поскольку они построены на основе сбалансированных деревьев.

3. Соотнесите название метода и действие, им реализуемое

а) вставка в начало 1) pop_back

б) удаление из начала 2) push_back

в) вставка в конец 3) pop_front

г) удаление из конца 4) push_front

4. Какое ключевое слово используется при объявлении шаблонов?

5. Перечислите методы, доступные в списках.

6. Определите результат выполнения программы:

```
#include <fstream>
#include <vector>
using namespase std;
int main() {
float  arr[]= {1.8, 2.38, -37, 0.88, -19.4 };
int n;
n=sizeof(arr)/sizeof(float);
vector<float > v1(arr, arr+n);
vector< float > v2;
v1.swap(v2);
while (!v2.empty() ) {
```

```
cout<<v2.back()/10<< ‘ ‘;
v2.pop_back();
}
return 0;}
```

Упражнения

1. Определить количество вопросительных предложений в заданном файле; вывести в дополнительный файл третье найденное предложение.

2. Определить количество слов в тексте; вывести в дополнительно организованный файл каждое десятое слово.

3. Определить количество слов в тексте, начинающихся с гласной буквы; вывести на экран шестое слово в тексте.

4. Определить количество слов в тексте, у которых первый и последний символы совпадают; найденные слова разместить во вновь созданном файле в алфавитном порядке.

5. Определят количество предложений, начинающихся с гласной буквы; разместить их в алфавитном порядке во вновь созданном файле. Каждое предложение размещать с новой строки.

6. В массиве, содержащем 14 элементов целого типа, определить произведение положительных элементов, расположенных до максимального элемента этого массива и отсортировать по возрастанию элементы, расположенные после максимального.

7. В массиве, содержащем 18 элементов вещественного типа, определить сумму положительных элементов, расположенных между минимальным и максимальным элементами данного массива и их отсортировать по возрастанию.

8. Если положительных элементов массива больше чем отрицательных, то отсортированные по убыванию положительные элементы расположить в начале массива. Иначе в начале массива расположить отсортированные отрицательные элементы.

9. Если номер минимального элемента массива, состоящего из 28 целочисленных элементов меньше 18, отсортировать по возрастанию элементы, расположенные после него, иначе отсортировать элементы с 8 по 18 номер.

10. Если номер максимального элемента вещественного массива, содержащего 25 элементов больше 15, то отсортировать по убыванию элементы массива, расположенные до максимального элемента, иначе отсортировать элементы данного массива, расположенные за максимальным элементом.

ЛИТЕРАТУРА

1. Бабб Т. Объектно-ориентированное программирование в действии. СПб.: Питер, 1997. – 464 с.

2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.

3. Дейл Н., Уимз Ч., Хедингтон М. Программирование на С++. М.: ДМК. 2000. – 672 с.

4. Павловская Т.А. С/С++. Программирование на языке высокого уровня: для магистров и бакалавров. СПб.: Питер, 2012. – 460 с.

5. Павловская Т.А., Щупак Ю.А. С/С++. Структурное и объектно-ориентированное программирование. Практикум. СПб.: Питер, 2011. – 352 с.

СОДЕРЖАНИЕ

Татьяна Алексеевна Галаган,
доцент кафедры ИиУС АмГУ

Объектно-ориентированное программирование. Язык С++.

Изд-во АмГУ. Подписано к печати Формат 60х84/16. Усл. печ. л.
Тираж Заказ