Д. В. Алейников, Е. П. Холодова, Ю. Н. Силкович

ЯЗЫК ПРОГРАММИРОВАНИЯ С++

Введение в объектно-ориентированное программирование

Учебно-методическое пособие

Рекомендовано учебно-методическим объединением
по образованию в области управления для студентов
учреждений высшего образования специальности
1-26 03 01 Управление информационными ресурсами
в качестве учебно-методического пособия

Минск
2016

УДК 004.432 : 004.42(075)
ББК 32.973.22я7
А45

Д. В. Алейников, Е. П. Холодова, Ю. Н. Силкович

Алейников, Д. В., Холодова, Е. П., Силкович, Ю. Н.

А45 Язык программирования C++ : введение в объектно-ориентированное программирование : учебно-методическое пособие : рекомендовано учебно-методическим объединением по образованию в области управления для студентов учреждений высшего образования специальности 1-26 03 01 Управление информационными ресурсами в качестве учебно-методического пособия / Д. В. Алейников, Е. П. Холодова, Ю. Н. Силкович. – Минск : Национальная библиотека Беларуси, 2016. – 85, [2] с.

ISBN 978-985-7039-84-5

В учебно-методическом пособии рассматриваются ключевые понятия объектно-ориентированного программирования на языке C++. Излагаемые теоретические сведения разбиты на небольшие по объему разделы и дополняются множеством примеров, которые не только иллюстрируют технологии объектно-ориентированного программирования, но и демонстрируют их применение при решении задач экономической направленности.

Изучение пособия предполагает наличие базовых знаний по языку программирования C++.

Разработано для студентов специальности «Управление информационными ресурсами», изучающих дисциплину «Алгоритмизация и программирование».

УДК 004.432 : 004.42(075)
ББК 32.973.22я7

ISBN 978-985-7039-84-5

© Алейников Д.В., 2016
© Холодова Е.П., 2016
© Силкович Ю.Н., 2016
© Оформление. Государственное учреждение «Национальная библиотека Беларуси», 2016

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ 5

ВВЕДЕНИЕ 6

1. ОСНОВНЫЕ ПОНЯТИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ 7

2. КЛАССЫ В С++ 11

- 2.1. Объявление класса 11
- 2.2. Реализация методов класса 12
- 2.3. Понятие объекта 14
- 2.4. Определение объекта 14
- 2.5. Доступ к элементам класса 15
- 2.6. Закрытые и открытые элементы класса 18
- 2.7. Конструкторы и деструкторы 20
  - 2.7.1. Понятие конструктора 20
  - 2.7.2. Понятие деструктора 22
  - 2.7.3. Параметрические конструкторы 24
- 2.8. Перегрузка методов и конструкторов 25
- 2.9. Массивы объектов 28
- 2.10. Статические элементы класса 34
  - 2.10.1. Статические поля класса 34
  - 2.10.2. Статические методы класса 36
- 2.11. Вложенные классы 37
- 2.12. Структура как простейший класс 40
- 2.13. Использование в качестве полей класса массивов 42
- 2.14. Пример реализации проекта «Банк» 44

3. РАБОТА С ОБЪЕКТАМИ 49

- 3.1. Указатели на объекты 49
- 3.2. Ключевое слово this 56
- 3.3. Ссылки на объекты 57
- 3.4. Передача объектов функциям в качестве аргументов 59
- 3.5. Возвращение объектов функциями и методами 62
- 3.6. Конструктор копий объектов 63

3.7. Дружественные функции 65
3.8. Динамическое выделение памяти под объекты 70
3.9. Динамическое выделение памяти под массивы объектов 72
3.10. Наследование в С++ 76
3.10.1. Простое наследование 77
3.10.2. Множественное наследование 81
3.10.3. Защищенные элементы 84
3.10.4. Построение иерархии классов 85
ЛИТЕРАТУРА 87

ПРЕДИСЛОВИЕ

Настоящее учебно-методическое пособие предназначено для студентов экономических специальностей, изучающих язык программирования C++. Оно представляет собой вторую часть серии учебных пособий под общим названием «Язык программирования C++»[1]. В первой части были изложены базовые сведения структурного программирования на C++. Программирование на языке C++ является достаточно сложным по сравнению с применением других языков и средств программирования.

Цель данного пособия – как можно доступнее изложить основные понятия концепции объектно-ориентированного программирования и его применения на языке C++. В издании даются краткие теоретические сведения, необходимые для освоения рассматриваемых тем, а для лучшего их усвоения приводится большое количество примеров программ с подробными комментариями. Отличительной особенностью данного пособия является то, что в качестве примеров, демонстрирующих изучаемые темы, приводятся решения задач экономического характера. Такой практический подход должен еще в большей степени помочь студентам экономических специальностей сориентироваться в такой сложной области, как программирование, и в частности объектно-ориентированное программирование.

Все примеры, приводимые в учебно-методическом пособии, были разработаны авторами в среде Visual Studio 2012.

Знания, полученные при изучении данного издания, помогут далее успешно осваивать применение языка C++ для разработки приложений под управлением ОС Windows.

При написании учебно-методического пособия авторы опирались на практический опыт преподавания курса «Алгоритмизация и программирование» в Институте бизнеса и менеджмента технологий БГУ на факультете бизнеса для студентов специальности «Управление информационными ресурсами» Ю.Н. Силковича и Е.П. Холодовой, а также опыт разработки коммерческих приложений практикующего программиста Парка высоких технологий Д.В. Алейникова.

---

[1] Первая часть называется «Язык программирования C++: структурное программирование».

ВВЕДЕНИЕ

Объектно-ориентированное программирование (ООП) является стандартом в технологии современного программирования. До его изобретения многие проекты достигали такой сложности, что разрабатывать их с использованием только структурного подхода не представлялось возможным. Поэтому логическим развитием структурного программирования явилось объектно-ориентированное, унаследовавшее лучшие идеи структурного и объединившее их с новыми понятиями.

Идея ООП заключается в описании задачи на уровне объектов, которые называются классами. В классе структуры данных и функции их обработки объединяются. Класс используется только через его интерфейс – детали реализации для пользователя класса несущественны. Идея классов отражает строение объектов реального мира – ведь каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами свойствами и поведением. Программы часто предназначены для моделирования предметов, процессов и явлений реального мира, поэтому в языке программирования удобно иметь адекватный инструмент для представления моделей.

Объектно-ориентированное программирование – это не просто набор новых средств, добавленных в язык (на C++ можно успешно писать и без использования ООП и, наоборот, возможно написать объектную по сути программу на языке, не содержащем специальных средств поддержки объектов). ООП часто называют новой парадигмой программирования. Красивый термин «парадигма» означает набор теорий, стандартов и методов, которые совместно представляют собой способ организации знаний, иными словами способ видения мира. В программировании этот термин используется для определения модели вычислений, то есть способа структурирования информации, организации вычислений и данных. Объектно-ориентированная программа строится в терминах объектов и их взаимосвязей.

1. ОСНОВНЫЕ ПОНЯТИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Часто требуется программным способом реализовать объект реального мира так, чтобы этому объекту соответствовал один программный объект. Так как любой программный объект на языке С++ имеет определенный тип, то такую задачу можно решить, лишь создавая новые, пользовательские *абстрактные типы данных*.

В основу объектного подхода заложено понятие *класса*. Класс представляет собой описание, схему, модель реально существующего объекта. Класс является типом данных, определяемым пользователем. Механизм для объявления нового типа позволяет создавать классы. Класс – это определение нового типа, то есть в результате объявления класса создается новый тип данных.

Класс – это сложный тип данных, в котором объединены элементы данных: поля и методы, обрабатывающие эти данные. *Поле класса* – это элемент класса, описывающий данные. *Метод класса* – это процедура или функция, включенная в описание класса. Метод класса вызывается конкретным экземпляром класса и привязан к описанию и структуре класса. Поля и методы класса являются *элементами класса*.

Класс – это всего лишь логическая абстракция, недоступная для прямого использования в программе. В языке С++ спецификация класса используется для создания объектов, то есть получения доступа к полям и методам класса необходимо создать экземпляр класса, называемый *объектом*. К одному классу может принадлежать одновременно несколько объектов, каждый из которых имеет уникальное имя. Объект характеризуется физическим существованием и является конкретным экземпляром класса.

Конкретные величины типа данных «класс» называются экземплярами класса, или объектами. Объекты взаимодействуют между собой, посылая и получая сообщения. *Объект* (или экземпляр класса) – это представитель класса, построенный по хранящемуся в классе описанию.

Программа работает только с объектами, а классы нужны для задания их внутреннего устройства. Условно говоря, класс – это проектное описание модели (например, автомобиля), а объект – это сам физически существующий автомобиль, построенный по чертежу.

Объекты (экземпляры классов), как и любые другие значения допустимых типов, хранятся в программе в переменных объектных типов.

Поля и методы класса могут быть *закрытыми* или *открытыми*. К закрытым полям и методам класса можно обращаться только внутри класса. Открытые элементы класса доступны за его пределами. Как правило, открытая часть кода обеспечивает управляемое взаимодействие (интерфейс) с закрытыми элементами объекта.

Как с синтаксической, так и с семантической точки зрения объект представляет собой переменную, тип которой определен пользователем. Определяя новый тип объекта, устанавливается новый тип данных. Каждый экземпляр этого типа данных является сложной переменной, которая может содержать не только данные, но и код.

В дополнение к полям часто применяется более абстрактное понятие свойства класса. Оно расширяет понятие поля и подразумевает использование промежуточного исполняемого кода, который предварительно обрабатывает хранимые в поле данные перед их выдачей или записью.

Во многих средах программирования понятие класса также дополнено понятием события. *Событие класса* – это формально определенное внешнее событие, на которое данный класс может реагировать, выполняя определенные действия.

Класс обеспечивает целостность вводимого в программу формализуемого прикладного понятия и не допускает разрыва между статическими и динамическими характеристиками некоторой сущности. На этапе проектирования программы с помощью классов удается формировать достаточно точные модели реальных объектов и строить их с помощью систем визуального моделирования.

Любой объектно-ориентированный язык программирования базируется на трех механизмах, которые называются инкапсуляцией, полиморфизмом и наследованием.

Инкапсуляция – это механизм:

- связывающий воедино программный код и данные, которыми он манипулирует;
- обеспечивающий защиту кода и данных от внешнего вмешательства и неправильного использования. Другим объектам доступен только интерфейс объекта, через который осуществляется все взаимодействие с ним;
- разграничивающий доступ к полям и методам класса.

Базовой единицей инкапсуляции является класс, а конкретный экземпляр класса называется объектом. В объектно-ориентированном языке код и данные можно погружать в «черный ящик», который называется объектом. Иначе говоря, объект – это средство инкапсуляции.

Важнейшая характеристика класса – возможность создания на его основе новых классов с наследованием всех его свойств и методов и добавлением собственных.

Наследование – это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. В С++ каждый класс может иметь сколько угодно потомков и предков. Наследование – это отношение, связывающее классы, один из которых является базовым и называется *родительским*, а другой создается на его основе и называется *наследником*.

Наследование заключается в том, что класс-наследник приобретает все свойства и методы родительского класса и добавляет к ним собственные.

Он имеет большое значение, поскольку поддерживает концепцию классификации. Если подумать, все знания организованы по принципу иерархической классификации. Например, антоновка относится к классу «яблоки», который в свою очередь является частью класса «фрукты», входяшего в класс «пища». Если бы классификации не существовало, мы не смогли бы точно описать свойства объектов. Однако при этом необходимо указывать только уникальные свойства объекта, позволяющие выделить его сре-

ди других объектов данного класса. Именно этот принцип лежит в основе механизма наследования, дающего возможность считать конкретный объект специфическим экземпляром более общей разновидности. Наследование является важным аспектом объектно-ориентированного программирования.

Во многих случаях методы родительского класса должны быть переопределены в классе-наследнике. Все переопределяемые методы по имени совпадают с методами родительского объекта, однако компилятор по типу объекта распознает, какой конкретно метод надо использовать.

Полиморфизм – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать метод, вызываемый объектом, в соответствии с типом данного объекта и с учетом иерархии наследования. Полиморфизм позволяет использовать единый унифицированный интерфейс для выполнения однотипных действий с различными данными. В C++ полиморфизм реализуется через перегрузку функций, методов и операторов.

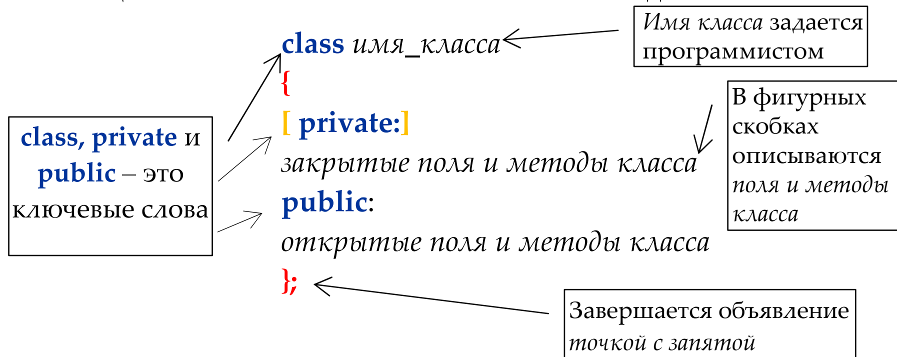
Современные объектно-ориентированные системы в дополнение к трем вышеописанным механизмам объектного программмирования поддерживают концепцию *сообщений*: объекты могут отправлять и получать сообщения, обмениваться ими друг с другом и с внешней средой. На этой концепции построена, в частности, схема работы пользовательского интерфейса.

Программа получает от пользователя и системы Windows сообщения о наступлении тех или иных событий (например, о нажатой кнопке в окне программы) и при необходимости реагирует на них.

2. КЛАССЫ В C++

2.1. Объявление класса

Общий синтаксис объявления класса имеет вид:



Управляют доступностью элементов класса ключевые слова ***private*** и ***public*** (иногда их называют *спецификаторами доступа*), после которых следует знак двоеточия. Сначала обычно перечисляются закрытые поля и методы класса. Ключевое слово ***private*** перед ними можно не указывать, так как все элементы класса по умолчанию считаются закрытыми. Перед объявлением открытых элементов обязательно указывается ключевое слово ***public***.

Можно задавать несколько секций ***private*** и ***public***, порядок их следования значения не имеет. В объявлении ключевое слово степени доступа (public, private и т. д.) применяется ко всем расположенным ниже элементам до тех пор, пока не встретится следующее ключевое слово степени доступа.

✍ Обычно поля класса делают закрытыми, а доступ к ним организуется через открытые методы.

Пример. Объявление класса с именем **SALE** (для хранения и обработки информации о товарах), в состав которого входят пять полей и два метода.

```
class SALE /* Объявление класса SALE. Имя класса – SALE
    становится именем типа, определенного пользователем */
{
  public: // Все элементы класса открытые
  // Далее объявляются поля класса:
     int  IDTov;   // код товара
     char Nazv[15];  // название товара
     double Price;  // цена товара
     int AvBegin;  // наличие на начало месяца
     int AvEnd;   // остаток на конец месяца
  // Далее объявляются методы класса:
     int total(); /* вычисление количества проданного товара */
     int total_cost(); /* вычисление общей стоимости проданного товара */
  }; /* Завершается объявление класса точкой с запятой */
```

🔔 При объявлении класса **SALE** память не резервируется. Это объявление просто сообщает компилятору о существовании класса **SALE**, о том, какие поля он содержит (переменные *IDTov*, *Nazv*, *Price*, *AvBegin*, *AvEnd*) и что он делает (методы *total()*, *total_cost()*). Также объявление сообщает компилятору о размере класса **SALE**, то есть сколько места должен зарезервировать компилятор для каждого объекта класса **SALE**. Методы не требуют выделения памяти, для них пространство в памяти не резервируется.

2.2. Реализация методов класса

Метод – это функция, объявленная в классе. Общий синтаксис объявления метода класса:

тип_результата *имя_метода*(*список_параметров*)**;**

Каждый объявленный метод класса должен быть реализован. *Реализация* – это описание действий метода. Реализация метода может быть определена в самом классе или за пределами класса.

Синтаксис реализации метода в классе:

```
тип_результата имя_метода(список_параметров)
{
операторы метода
}
```

Определение реализации метода вне класса содержит имя класса и имя метода, разделенных с помощью оператора, состоящего из двух последовательных двоеточий (**::**) и называемого *оператором разрешения области видимости;* список параметров метода.

Синтаксис реализации метода класса вне класса:

```
class имя_класса
{
...
//Прототип определяемого метода:
тип_результата  имя_метода(список_параметров);
...
};
//Реализация метода класса:
тип_результата имя_класса :: имя_метода(список_параметров)
{
операторы метода
}
```

Пример. Для класса **SALE** реализация метода *total()* определена в классе, а метода *total_cost()* вне класса.

```
class SALE
{
  public: // Все элементы класса открытые
  // Далее объявляются поля класса:
     int  IDTov;   // код товара
     char Nazv[15];  // название товара
     double Price;  // цена товара
     int AvBegin;  // наличие на начало месяца
     int AvEnd;   // остаток на конец месяца
  // Методы класса:
     /* метод total  - для вычисления количества проданного товара */
     int total()
     {return AvBegin-AvEnd;} /* его реализация определена в классе */
      int total_cost(); /* прототип метода для вычисления общей стоимости
      проданного товара */
  }; /* Завершается объявление класса точкой с запятой */
int SALE :: total_cost()  /* Реализация метода total_cost вне класса */
{    return (AvBegin-AvEnd)*Price;  /* Инструкции  метода заключаются в
фигурные скобки */
}
```

2.3. Понятие объекта

Объект – это конкретная переменная типа «класс». Отдельный объект определенного класса называют экземпляром этого класса, а процесс его создания – созданием экземпляра. Существует различие между абстрактным классом, например **SALE**, и каждым конкретным объектом класса **SALE**, который соответствует определенному товару. Только после объявления класса можно создать объект типа **SALE**.

2.4. Определение объекта

Объект нового типа можно определить так же, как и любую переменную базового типа:

имя_класса *имя_объекта;*

Пример. В этой строке определяется объект **Product** класса (или типа) **SALE**.

```
SALE Product;
```

Также список создаваемых объектов класса можно указать после закрывающих фигурных скобок объявления класса:

class *имя_класса*
{
закрытые поля и методы класса
public:
открытые поля и методы класса
} *список объектов*;

Пример. Два объекта **Product1** и **Product2** создаются при объявлении класса **SALE**.

```
class SALE
{
   public:
   // поля класса
   int IDTov; char Nazv[15]; double Price; int
AvBegin;  int AvEnd;
   // методы класса
   int total(); int total_cost();
} Product1, Product2; /* объекты Product1 и
Product2 класса SALE */
```

2.5. Доступ к элементам класса

После определения объекта класса для доступа к его элементам (как полям, так и методам) используется точечный оператор (.):

имя_объекта.*имя_поля*

или

имя_объекта.*имя_метода*

🔔 В языке С++ нельзя присвоить значение базовому типу данных, оно присваивается только переменной.

Неправильный код:

```
int = 125; /* нельзя присвоить число 5 типу int */
```

Правильный код:

```
int ID; /* определить переменную типа int, например, ID */
ID = 125; /* присвоить переменной ID, имеющей тип int, значение 5 */
```

То же относится и к классам. Следующая строка недопустима:

```
SALE.IDTov =125; // неверно
```

Компилятор пометит ее как ошибочную, поскольку сначала необходимо создать объект класса **SALE**, а потом его переменной IDTov присвоить значение 125.

```
SALE Product;
Product.IDTov = 125;
```

Пример. В программе объявляется класс **SALE**, создается два объекта (Product1 и Product2) типа **SALE**, полям объектов присваиваются значения и выводятся на экран.

```
#include <iostream>
#include <cstring>
#include <conio.h>
using namespace std;
class SALE /* Объявление класса SALE */
{
public: // Открытые элементы класса:
// поля класса:
int IDTov; char Nazv[15]; double Price; int AvBegin;
int AvEnd;
// методы класса:
int total()
{return AvBegin-AvEnd;}
int total_cost();
```

```
};
int SALE :: total_cost()  /* Реализация метода
total_cost вне класса */
{      return (AvBegin-AvEnd)*Price;
}
int main()  {
SALE Product1; /* Создание объекта Product1 класса
SALE: */
// Полям объекта Product1 присваиваются значения:
Product1.IDTov=125;
strcpy(Product1.Nazv, "Tefal  "); /* Функция strcpy
помещает слово Tefal в переменную Nazv объекта
Product1 */
Product1.Price=120000; Product1.AvBegin=20;
Product1.AvEnd=3;
SALE Product2; /* Создание объекта Product2 класса
SALE: */
// Полям объекта Product2 присваиваются значения:
Product2.IDTov=130;
strcpy(Product2.Nazv, "Philips");
Product2.Price=145600; Product2.AvBegin=40;
Product2.AvEnd=5;
// вывод иформации о продуктах на экран
cout<< "Product1 "<<Product1.IDTov << " " <<
Product1.Nazv << " " << Product1.Price <<" "<<
Product1.AvBegin  << " " << Product1.AvEnd << " " <<
Product1.total() << " " << Product1.total_cost() <<
endl;
cout << "Product2 "<<Product2.IDTov << " " <<
Product2.Nazv << " " << Product2.Price <<" "<<
Product2.AvBegin  << " " << Product2.AvEnd << " " <<
Product2.total() << " " << Product2.total_cost();
_getch();
return 0; }
```

Результат выполнения программы (последние два числа для каждого продукта – это значения, возращаемые методами *total()* и *total_cost()*):

```
Product1 125 Tefal   120000 20 3 17 2040000
Product2 130 Philips 145600 40 5 35 5096000
```

2.6. Закрытые и открытые элементы класса

Как уже было сказано, поля и методы класса могут быть открытыми *(public)* и закрытыми *(private)*. По умолчанию все элементы класса являются закрытыми. К закрытым элементам могут обращаться только те методы, которые принадлежат этому классу. Открытые элементы доступны для всех других функций программы.

Согласно общей стратегии использования классов их поля следует оставлять закрытыми. Следовательно, чтобы передавать и возвращать значения закрытых переменных, необходимо создать открытые методы, известные как *методы доступа.* Применение методов доступа позволяет скрыть от пользователя подробности хранения данных в объектах, предоставляя в то же время методы их использования.

Пример. Использование открытых и закрытых элементов класса.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Sale {   //Объявление класса
    //Поля класса закрытые:
  int price;//цена товара
  int number;//количество товара
public:  //Методы  класса открытые:
  int total(int i,int j); /*вычислить и вернуть
  общую стоимость*/
  void show(int t);  //вывести общую стоимость
};
int Sale::total(int i,int j) /* Реализация метода
total */
{  /* метод возвращает произведение цены на
количество */
price=i;  number=j; return price*number;
}
void Sale::show(int t) //Реализация метода show
{
cout << "Цена товара = " << price <<endl;
cout << "Количество = " << number <<endl;
cout << "Общая стоимость = " << t <<endl;
```

```
}
int main() {
setlocale(LC_ALL, "Russian"); /* функция позволяет
использовать русский шрифт */
int t;  //объявление переменной t типа int
Sale dress; //создание объекта dress класса Sale
t=dress.total(850000,5); /* переменной t
присваивается общая стоимость закупки, которая
вычисляется в методе total */
dress.show(t); //вызов метода show для объекта dress
_getch();
return 0;}
```

Результат выполнения программы:

```
Цена товара = 850000
Количество = 5
Общая стоимость = 4250000
```

Пример. В данном примере метод *show()* класса **Sale** уже закрытый и вызывается в методе *total()*.

```
#include<iostream>
#include<conio.h>
#include <locale.h>
using namespace std;
class Sale {
    //Закрытые элементы класса:
int price; int number;
/* Функция  show закрытая */
void show();  //вывести общую стоимость
public:
//Открытые элементы класса:
void total(int i,int j); /*вычислить и вернуть общую
стоимость*/
};
//Реализация методов класса:
void Sale::total(int i,int j)
{price=i;  number=j;
 show();
}
```

```
void Sale::show(){
cout << "Цена товара = " << price <<endl;
cout << "Количество = " << number <<endl;
cout << "Общая стоимость = " << price*number
<<endl;}
int main()
{ setlocale(LC_CTYPE, "Russian");
Sale dress; //создание объекта dress класса Sale
dress.total(120000,3); /*вызов метода total */
_getch();
return 0;}
```

Результат выполнения программы:

```
Цена товара = 120000
Количество = 3
Общая стоимость = 360000
```

2.7. Конструкторы и деструкторы

2.7.1. Понятие конструктора

В C++ существуют два способа определить переменную. Можно сначала объявить переменную, а затем (несколько ниже в программе) присвоить ей значение, например:

```
int Kurs; // объявить целочисленную переменную
Kurs = 2; // присвоить переменной значение
```

Или, определив переменную, сразу же инициализировать ее, например:

```
int Kurs = 2; /*определить переменную Kurs и
инициализировать значением 2 */
```

Инициализация объединяет определение переменной с присвоением ей исходного значения, что, однако, ничуть не запрещает изменить это значение впоследствии. Инициализация гарантирует, что переменная никогда не останется без значения.

В предыдущих примерах переменные объектов класса устанавливались «вручную» с помощью инструкций, наподобие нижеследующих:

```
dress.price=850000; dress.number=5;
```

В профессионально написанных программах такой подход все же применяется редко. Для инициализации полей используется специальный метод, называемый *конструктором*.

Конструктор – это метод класса, который служит для инициализации объекта при его создании. Имя конструктора совпадает с именем самого класса. При необходимости конструктор может получать параметры, но не может возвращать значения, даже типа **void**. Конструктор без параметров называется ***стандартным***.

Общая форма стандартного конструктора, реализация которого определяется в классе:

имя_класса ( )
{
операторы конструктора
}

Если реализация конструктора определяется за пределами класса, то в классе указывается прототип конструктора в формате:

имя_класса ( );
а за пределами класса описывается его реализация:

имя_класса::имя_класса ( )
{
операторы конструктора
}

Обычно конструктор используется для придания переменным экземпляра, определенным в классе, начальных значений, а также для выполнения других начальных процедур, требуемых для создания полностью сформированного объекта.

2.7.2. Понятие деструктора

В некотором роде противоположностью конструктору является ***деструктор***. Во многих случаях объекту при его удалении требуется выполнение некоторого действия или последовательности действий. Локальные объекты создаются при входе в свой блок и уничтожаются при выходе из блока. Глобальные объекты уничтожаются, когда завершается программа. Может быть много причин, приводящих к необходимости иметь деструктор. Например, объекту может понадобиться освободить ранее выделенную память или закрыть открытый им файл. В С++ такого рода операции выполняет деструктор.

Деструктор удаляет из памяти отработавшие объекты и освобождает выделенную для них память. Чтобы придать классу законченность, при объявлении конструктора необходимо объявить и деструктор, даже если ему нечего делать. Деструктору всегда присваивается имя класса с символом тильды (~) вначале. Деструкторы не получают аргументов и не возвращают значений!

Объявление деструктора класса будет выглядеть следующим образом:

~ имя_класса();

🔔 Если конструктор или деструктор не созданы явно, то компилятор создаст их сам. Созданный компилятором конструктор будет стандартным, то есть без параметров. Конструкторы и деструкторы, созданные компилятором, не только не имеют аргументов, но и ничего не делают.

Пример. Демонстрация использования стандартного конструктора и деструктора.

```
#include <iostream>
#include <cstring>
#include <conio.h>
using namespace std;
class SALE /* Объявление класса SALE */
{
 public: // Открытые члены класса:
 char Nazv[15];  // название товара
```

```
 double Price;  // цена товара
 int Number;  // количество товара
         SALE(); /* объявление стандартного
конструктора */
/* Конструктор объявлен после описателя public, так
как конструктор будет вызываться из кода вне его
класса */
      ~SALE(); // объявление деструктора
/* прототип метод total_cost для расчета общей
стоимости  */
 int total_cost();
 };
SALE::SALE()   /* реализация стандартного
конструктора */
{  strcpy(Nazv, "Untitled"); /* полю Nazv
присваивается значение «Untitled» */
    Price=0; Number=0;  }
SALE::~SALE()    // реализация деструктора
{cout << "Deleting an object...";} /* Деструктор в
программе выводит сообщение, но в реальности еще
освобождает ресурсы (например, память), используемую
объектом */
int SALE :: total_cost()  /* реализация метода
total_cost */
{      return Number*Price;   }
 int main()  {
SALE clothing; /* Создание объекта clothing класса
SALE.   Когда   создается   объект,   стандартный
конструктор  вызывается  по  умолчанию,  присваивая
переменным Price и Number объекта clothing значение
«0», а переменной Nazv  значение «Untitled» */
// вывод сведений об объекте
cout << clothing.Nazv << " " << clothing.Number << "
" << clothing.Price << " " << clothing.total_cost();
cout << endl;
_getch();
return 0; }
```

Результат выполнения программы:

```
Untitled 0 0 0
Deleting an object...
```

2.7.3. Параметрические конструкторы

Параметры добавляются в конструктор так же, как они добавляются в метод: необходимо объявить их внутри круглых скобок после имени конструктора.

Общая форма объявления параметрического конструктора:

имя_класса ***имя_объекта***(*список параметров*)**;**

Пример. Создание параметрического конструктора для класса **SALE**.

```
#include <iostream>
#include <cstring>
#include <conio.h>
using namespace std;
class SALE /* Объявление класса SALE */
{
 public: // Открытые элементы класса:
 char Nazv[15];  double Price;  int Number;
 SALE(char p_Nazv[15], double p_Price, int
p_Number); /* объявление параметрического
конструктора */
          ~SALE(); // объявление деструктора
 /* прототип метод total_cost для расчета общей
стоимости  */
 int total_cost();
 };
/*Конструктор определяет три параметра: p_Nazv,
p_Price и p_Number, которые используются для
инициализации переменных экземпляра*/
SALE::SALE(char p_Nazv[15], double p_Price, int
p_Number) /* реализация  параметрического
конструктора */
{  strcpy(Nazv, p_Nazv);   Price=p_Price;
Number=p_Number;
    }
SALE::~SALE()    // реализация деструктора
{cout << "Deleting an object..." << endl;}
int SALE :: total_cost()  /* реализация метода
total_cost */
{      return Number*Price;   }
```

```
 int main()  {
SALE furniture("bag",23000,15); /* создание объекта
clothing класса SALE с помощью параметрического
конструктора. При его вызове параметрам конструктора
p_Nazv, p_Price и p_Number передаются соответственно
аргументы «bag», 23000 и 15  */
// вывод сведений об объекте
cout << furniture.Nazv << " " << furniture.Number <<
" " << furniture.Price << " " <<
furniture.total_cost();
cout << endl;
_getch();
return 0; }
```

Результат выполнения программы:

```
Untitled 0 0 0
bag 15 23000 345000
Deleting an object...
Deleting an object...
```

2.8. Перегрузка методов и конструкторов

Как и обычные функции, методы классов можно перегружать. В этом случае создается несколько вариантов одного и того же метода, но с разными прототипами. Отличие может быть связано с разным типом и количеством параметров.

Формально перегруженные методы могут быть (с точки зрения функциональности) абсолютно разными. Но хорошим стилем считается, если разные варианты перегруженного метода объединены общей идеей. Такой подход позволяет использовать единый интерфейс и при этом учесть особенности вызова соответствующего метода с разным набором аргументов.

Данная концепция получила название *полиморфизма*, одного из трех фундаментальных механизмов, на которых базируется объектно-ориентированное программирование.

Конструкторы также можно перегружать. Для перегрузки конструктора класса необходимо объявить разные его формы.

Пример. Реализация трех перегруженных конструкторов для класса **Sale**.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Sale { /*Объявление класса*/
public:
int price; int number;
Sale(); //перегруженный стандартный конструктор
Sale(int i);  /* перегруженный параметрический
конструктор с одним параметром */
Sale(int i_p, int i_n); /* перегруженный
параметрический конструктор с двумя парметрами */
~Sale(); //объявление деструктора
};
Sale::Sale()  /* Реализация 1-го конструктора: оба
поля класса инициализирутся нулями */
{price=0;number=0;}
Sale::Sale(int i) /* Реализация 2-го конструктора:
оба поля класса инициализирутся одинаково -
значением передаваемого аргумента i */
{price=i;number=i;}
Sale::Sale(int i_p, int i_n) /* Реализация 3-го
конструктора: поля получают значения i_p и i_n  */
{price=i_p;number=i_n;}
Sale::~Sale()//Реализация деструктора
{cout << "DELETE...";}
int main()
{
Sale clothing1;/* создание объекта clothing1 (по
умолчанию вызывается стандартный конструктор) */
cout << clothing1.number << " "<<clothing1.price <<
endl;
Sale clothing2(50);/* создание объекта clothing2
(используется конструктор Sale(int i)) */
cout << clothing2.number << " "<<clothing2.price <<
endl;
Sale clothing3(30,5);/* создание объекта
(используется конструктор Sale(int i_p, int i_n)) */
cout << clothing3.number << " "<<clothing3.price <<
endl;
```

```
_getch();
return 0;}
```

Результат выполнения программы:

```
0 0
50 50
5 30
DELETE...DELETE...DELETE...
```

Пример. Перегрузка методов класса **SALE**.

```
#include<iostream>
#include<conio.h>
using namespace std;
class SALE {
public:
double price;//цена товара
int number;//количество товара
SALE(double Price, int Number);
~SALE();
double total(); /* 1-й перегруженный метод:
вычислить и вернуть общую стоимость */
double total(double skidka); /* 2-й перегруженный
метод: вычислить общую стоимость с учетом процента
скидки */
};
SALE::SALE(double Price, int Number) /*реализация
параметрического конструктора*/
{  price=Price; number=Number; }
SALE::~SALE()     // реализация деструктора
{cout << "Deleting an object..." << endl;}
double SALE::total()/*Реализация 1-го метода total()
класса */
{return price*number;}
double SALE::total(double skidka) /* Реализация 2-го
метода total() класса */
{return price*number*(1-skidka);}
int main()
{ setlocale(LC_ALL, "Russian");
```

```
SALE clothing(1500,4); /* создание объекта clothing
класса SALE */
cout << " Без скидки " << clothing.total() << endl;
cout << " Со скидкой " << clothing.total(0.10) <<
endl;
_getch();
return 0;}
```

Результат выполнения программы:

```
Без скидки 6000
Со скидкой 5400
```

2.9. Массивы объектов

Массивы объектов можно создавать так же, как и массивы базовых типов данных. В качестве типа элементов массива указывается имя класса. Синтаксис создания массива объектов:

имя_класса имя_массива[размерность];

Обращение к объектам, являющимся элементами массива, осуществляется обычным образом с помощью индексов, например:

имя_массива[индекс]. *имя_элемента_класса*

Пример. Создание массива объектов **equipment** класса **Sale**.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Sale {
public:/* Открытые элементы класса: название товара,
цена товара и количество */
char *names; int price; int number;
int total(); //вычислить и вернуть общую стоимость
};
int Sale::total()//Реализация метода класса:
{return price*number;}
```

```
int main() {
setlocale(LC_ALL, "Russian");
Sale equipment[2]; /* создание массива объектов
класса Sale */
// Присваивание значений полям 1-го объекта массива
equipment[0].names="Планшет  ";
equipment[0].price=405000;equipment[0].number=4;
// Присваивание значений полям 2-го объекта массива
equipment[1].names="Смартфон ";
equipment[1].price=155000;equipment[1].number=6;
// Вывод значений полей объектов на экран
cout << equipment[0].names<<"
"<<equipment[0].price<<" ";
cout<<equipment[0].number<<"
"<<equipment[0].total()<<endl;
cout << equipment[1].names<<"
"<<equipment[1].price<<" ";
cout<<equipment[1].number<<"
"<<equipment[1].total()<<endl;
_getch();
return 0;}
```

Результат выполнения программы:

```
Планшет   405000 4 1620000
Смартфон  155000 6 930000
```

🕭 Если в классе описан конструктор с обязательной передачей аргументов, то при объявлении массива его необходимо инициализировать. Для инициализации в фигурных скобках перечисляются вызовы конструкторов с нужными аргументами.

Пример. В программе объявляется массив объектов класса **SALE** с его одновременной инициализацией.

```
#include<iostream>
#include<conio.h>
using namespace std;
class SALE {
public:
char names[15]; // название товара
```

```
double price;//цена товара
int number;//количество товара
SALE(char Nazv[15], double Price, int Number); /*
объявление параметрического конструктора */
~SALE(); // объявление деструктора
int total(); /* метод: вычислить и вернуть общую
стоимость */
};
SALE::SALE(char Nazv[15], double Price, int Number)
/* реализация параметрического конструктора */
{  strcpy(names, Nazv); price=Price; number=Number;
}
SALE::~SALE()    // реализация деструктора
{cout << "Deleting an object..." << endl;}
int SALE::total() //реализация метода класса
{return price*number;}
int main()  {
setlocale(LC_ALL, "Russian");
/* создание массива equipment объектов класса SALE с
его инициализацией */
SALE equipment[3] = {
     SALE("Планшет", 405000,4),
     SALE("Смартфон",155000,6),
     SALE("Модем",265000,8)
    };
for (int i=0; i<3; i++) /* вывод сведений об
объектах */
{
cout << equipment[i].names<<"
"<<equipment[i].price<<" ";
cout << equipment[i].number<<"
"<<equipment[i].total()<<endl;
}
_getch();
return 0;}
```

Результат выполнения программы:

```
Планшет 405000 4 1620000
Смартфон 155000 6 930000
Модем 265000 8 2120000
```

Пример. В данном примере объявляется класс APTEKA для хранения информации о лекарствах. В основной программе создается массив объектов типа APTEKA. Значения для полей объектов считываются из текстового файла Apteka.txt, содержимое которого представлено на рис. 1.

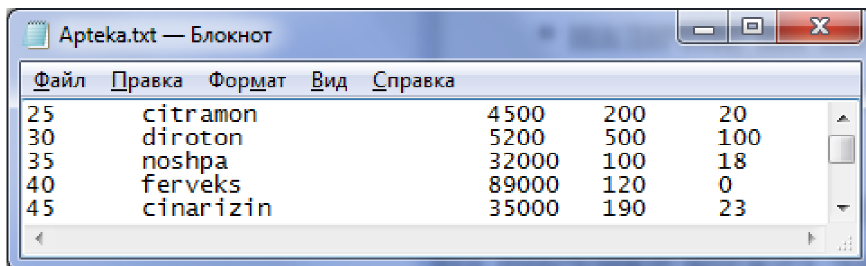


Рисунок 1 – Содержимое текстового файла Apteka.txt

```
/* Демонстрация создания и использования класса с
именем APTEKA */
#include <fstream>
#include <iostream>
#include <cstring>
#include <conio.h>
#include <iomanip>
using namespace std;
class APTEKA  // Объявление класса APTEKA
{
    int  Ras; /* закрытое поле - расход лекарства за
месяц */
    void Rashod(); /* Прототип закрытого метода
Rashod */
public:
int IDmed;  // открытое поле - код лекарства
char Nazv[15];  // открытое поле - название
double Price;  // открытое поле - цена
int AvBegin;  /* открытое поле - наличие на начало
месяца */
int AvEnd;   // открытое поле - остаток на конец
месяца
int Rashod_out(); /* Прототип отрытого метода
Rashod_out */
```

```
APTEKA(); //объявление стандартного конструктора
// далее объявляется параметрический конструктор
APTEKA(int  p_IDmed, char p_Nazv[15], double
p_Price, int p_AvBegin, int p_AvEnd);
~APTEKA(); //объявление деструктора
};
//Реализация стандартного конструктора
APTEKA::APTEKA()
{
    IDmed=0; strcpy(Nazv, ""); Price=0; AvBegin=0;
AvEnd=0;
    }
//Реализация параметрического конструктора
APTEKA::APTEKA(int  p_IDmed, char p_Nazv[15], double
p_Price, int p_AvBegin, int p_AvEnd)
{
    IDmed=p_IDmed; strcpy(Nazv, p_Nazv);
Price=p_Price; AvBegin=p_AvBegin; AvEnd=p_AvEnd;
    }
APTEKA::~APTEKA()
{cout << "DELETE...";} //Реализация деструктора

void APTEKA::Rashod() /* Прототип закрытого метода
Rashod */
    { Ras = AvBegin - AvEnd;}  /* Реализация :
вычисление расхода лекарства за месяц*/
int APTEKA::Rashod_out() /*Прототип отрытого метода
Rashod_out*/
    { Rashod(); /*Реализация: вызывается метод
Rashod для расчета поля Ras*/
return Ras; /* и возвращается значение закрытого
поля Ras */
}

int main() {
setlocale(LC_ALL,"Russian");
char zagl[4][10] = {"Код", "Название", "Цена",
"Продажа"};
APTEKA mas[5];  /* Объявление массива mas для
хранения 5 объектов APTEKA*/
```

```
/* В переменную name_of_file сохраняется
месторасположение текстового файла Apteka.txt  */
const char* name_of_file ="D:/Apteka.txt";
ifstream fin;
fin.open(name_of_file); // Открытие файла Apteka.txt
if(!fin.is_open())
{
cout << "Проблемы с открытием файла!" << endl; /*
Сообщение выдается, если файл не открылся*/
}
else /* блок else работает, если файл успешно
открылся */
{ // начало блока else
for(int i=0;i<5;i++) /* Присваивание значений,
считываемых из файла полям объектов массива*/
{// начало for
 fin >> mas[i].IDmed;/* Считывание кода лек-ва в
поле Nazv объекта mas[i] */
 fin >> mas[i].Nazv; /* Считывание назв-я лек-ва в
поле Nazv объекта mas[i] */
 fin >> mas[i].Price;
 fin >> mas[i].AvBegin;
 fin >> mas[i].AvEnd;
}  // конец for
if(!fin.good())
{ cout << "Ошибка считывания данных!" << endl; }
fin.close(); // закрытие текстового файла
} // конец блока else
for (int i=0; i<4; i++) cout << setw(15)<< zagl[i];
/* Вывод заголовков столбцов таблицы */
cout << endl; cout << endl;

for(int i=0;i<5;i++) /* Вывод значений полей
объектов */
{
 cout << setw(15)<< mas[i].IDmed; //Вывод кода лек-ва
cout << setw(15)<< mas[i].Nazv; /* Вывод
наименования лекарства */
cout << setw(15)<< mas[i].Price; //Вывод цены лек-ва
cout << setw(15) << mas[i].Rashod_out(); /* Вывод
поля Ras с помощью метода Rashod_out */
cout << endl;
```

```
}
getch();return 0;}
```

Результат выполнения программы:

Код	Название	Цена	Продажа
25	citramon	4500	180
30	diroton	5200	400
35	noshpa	32000	82
40	ferveks	89000	120
45	cinarizin	35000	167

2.10. Статические элементы класса

Хотя каждый объект класса имеет одинаковый набор полей и методов, значения полей у каждого объекта свои, а результат вызова методов в общем случае зависит от того, из какого объекта метод вызывается. На самом деле это не всегда так. Существуют особые элементы класса, которые называют статическими.

🕭 Статический элемент является общим для всех объектов этого класса.

2.10.1. Статические поля класса

В отличие от обычного поля класса статическое поле не исчезает при удалении объекта. В некотором смысле статические переменные напоминают глобальные переменные программы. Однако статические переменные, в отличие от глобальных переменных, полностью согласуются с принципами объектно-ориентированного программирования, в частности с принципом инкапсуляции. Статические элементы объявляются, как и обычные, но перед статическим элементом указывается ключевое слово **static**.

Пример. Демонстрация использования статического поля.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Sale {
public:
/*статическое поле amount класса  Sale */
static int amount; /* при объявлении статической
переменной класса память под нее не выделяется */
// поля класса
double price; //цена товара
int number; //количество товара
} Product1, Product2; /* создание объектов Product1
и Product2 при объявлении класса */
int Sale::amount; /* Повторное объявление
статической переменной. Оно выполняется для
выделения места в памяти под эту переменную */
int main() {
setlocale(LC_ALL, "Russian");
cout << "Количество  = " << Sale::amount << endl; /*
Для обращения к статическому полю используется имя
класса, а не имя объекта. На экран будет выведен 0,
т.к.  статическому полю типа int при объявлении
присваивается ноль */
Product1.price=25.5; Product1.number=5;
Product1.amount++; /* обращение к статическому полю
возможно через объект */
cout<< "Количество  = " << Sale::amount << endl;
Product2.price=36.9; Product2.number=7; /* значения
обычных полей у каждого из объектов свои */
Product2.amount ++; /*  статическое поле одно для
всех объектов */
cout<< "Количество  = " << Sale::amount << endl;
Sale::amount++; /* значение статического поля одно
для всех объектов, поэтому следующая команда выведет
три одинаковых значения */
cout<< "Количество  = " << Product1.amount<< " " <<
Product2.amount << " " << Sale::amount <<endl;
_getch();
return 0;
}
```

Результат выполнения программы:

```
Количество  = 0
Количество  = 1
Количество  = 2
Количество  = 3 3 3
```

2.10.2. Статические методы класса

Подобно статическим полям объявляются и статические методы. Однако по сравнению с обычными методами класса на статические накладываются существенные ограничения. Так, например, статические методы напрямую (то есть не через аргументы) могут ссылаться на статические поля класса и имеют ряд других особенностей.

Пример. Демонстрация использования статического метода.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Sale {
public:
/*статическое поле amount класса Sale */
static int amount; /* при объявлении статической
переменной класса память под нее не выделяется */
//нестатические поля:
double price; //цена товара
int number; //количество товара
//Статический метод:
static double total(double skidka);
} Product1, Product2; /*создание объектов при
объявлении класса*/
//Описание статического метода:
double Sale::total(double skidka)
{return amount+skidka;}
int Sale::amount; /*Повторное объявление статической
переменной. Оно выполняется для выделения места в
памяти под эту переменную */
int main() {
```

```
cout << "Amount  = " <<Sale::amount << endl; /* Для
обращения к статическому полю используется имя
класса, а не имя объекта. На экран будет выведен 0,
т.к.  статическому полю при объявлении присваивается
ноль */
Product1.price=25; Product1.number=5;
/*обращение к статическим полям и методам возможно
через объект*/
Product1.amount++;
cout<< "Amount  = " << Product1.amount << " Total =
" << Product1.total(0.05) << endl;
Product2.price=12; Product2.number=10; /* значения
обычных полей у каждого из объектов свои*/
Product2.amount ++; /* статическое поле одно для
всех объектов */
cout<< "Amount  = " << Product1.amount << " Total =
" << Product2.total(0.03) << endl;
Sale::amount++; /*  статические поля и методы одни
для всех объектов */
cout << "Amount  = " <<Sale::amount << endl;
cout<< "TOTAL  = " << Product1.total(0.01) << " " <<
Product2.total(0.02)<< " " << Sale::total(0.03)
<<endl;
_getch();
return 0;
}
```

Результат выполнения программы:

```
Amount  = 0
Amount  = 1 Total = 1.05
Amount  = 2 Total = 2.03
Amount  = 3
TOTAL  = 3.01 3.02 3.03
```

2.11. Вложенные классы

Полями класса могут быть не только переменные базовых типов, но и объекты других классов.

Пример. Использование объектов в качестве полей класса.

```
#include <iostream>
#include <conio.h>
using namespace std;
/* Класс Skidka будет использоваться в качестве поля
класса Tovar*/
class Skidka {
public:
double pr_sk; // Поле: процент скидки
Skidka(){  /* Конструктор без аргументов для класса
Skidka */
pr_sk=0;
/* Следующее сообщение выводится при создании
объекта класса Skidka */
cout<<"Создание объекта Skidka = "<<pr_sk<<endl;}
~Skidka(){ //Деструктор:
/* Следующее сообщение выводится при удалении
объекта класса Skidka*/
cout<<"Удаление объекта Skidka = "<<pr_sk<<endl;}
};
class Tovar { /* Класс Tovar */
public:
double price; // Поле (цена товара) типа double
Skidka Procent;//Поле-объект типа Skidka (скидка)
Tovar(double pr){/*Перегруженный конструктор с одним
параметром */
price=pr;  /* цена созаваемого товара не
предусматривает скидку */
/* Следующее сообщение выводится при создании
объекта класса Tovar*/
cout<<"Создание товара с ценой (без учета скидки) =
"<<price<<endl;}
Tovar(double pr,double sk){/*Перегруженный
конструктор с двумя параметрами */
Procent.pr_sk=sk;
price=pr;
cout<<"Создание товара с ценой = "<<price<<endl;
price=pr*(1-Procent.pr_sk); /* Расчет цены товара с
учетом скидки */
}
//Деструктор:
~Tovar(){
```

```
/* Следующее сообщение выводится при удалении
объекта класса Tovar*/
cout<<"Удаление товара с ценой = "<<price<<endl;}
};
int main(){
setlocale(LC_ALL,"Russian");
Tovar Product1(56700); /* Создание объекта класса
Tovar с помощью конструктора с одним параметром. В
качестве аргумента в этот конструктор передается
значение 56700*/
cout<<"Товар1:цена = "<<Product1.price<<"
Товар1:скидка = "<<Product1.Procent.pr_sk<<endl; /*
Вывод значений полей объектов */
cout << endl;
Tovar Product2(70000,0.03); /* Создание объекта
класса Tovar с помощью конструктора с двумя
параметрами. В качестве аргументов в этот
конструктор передаются значения 70000 и 0.03*/
cout<<"Товар2: скидка = "<<Product2.Procent.pr_sk <<
"  Товар2: новая цена  = "<<Product2.price<<endl;/*
Вывод значений полей объектов */
cout << endl;
_getch();
return 0;}
```

Результат выполнения программы:

```
Создание объекта Skidka = 0
Создание товара с ценой (без учета скидки) = 56700
Товар1:цена = 56700  Товар1:скидка = 0

Создание объекта Skidka = 0
Создание товара с ценой = 70000
Товар2: скидка = 0.03  Товар2: новая цена  = 67900

Удаление товара с ценой = 67900
Удаление объекта Skidka = 0.03
Удаление товара с ценой = 56700
Удаление объекта Skidka = 0
```

При создании объектов класса **Tovar** перед сообщением о создании данного объекта выводится сообщение о создании объекта **Procent** класса **Skidka**, который является полем класса **Tovar**.

2.12. Структура как простейший класс

Структуры объявляются с помощью ключевого слова **struct**. Описатель **struct** синтаксически схож с **class**, оба создают классовый тип. В языке С **struct** может содержать только поля, но в С++ это ограничение снято. В С++ **struct** – это, в сущности, просто другой способ задать класс. Фактически в С++ единственное различие между **class** и **struct** заключается в том, что по умолчанию все члены в **struct** открыты, а в **class** закрыты. Во всех остальных отношениях структуры и классы эквивалентны.

Пример. В программе определяется структурный тип **Tarif**, в котором есть одно закрытое поле и три открытых метода.

```
#include <iostream>
#include <conio.h>
using namespace std;
/*Объявление структуры */
struct Tarif {
/* Открытые методы */
void begin_price(double bp) { price=bp;} /* Метод
задает значение для поля price: цену тарифа
(например мобильного оператора)*/
double end_price(double pr) { return price*(1+pr); }
/* Метод возвращает стоимость оплаты за тариф с
учетом НДС */
double show() {return price;}/* Метод возвращает
значение поля price */
private:
double price; //Закрытое поле
};
int main()
{ Tarif Smart500; /*Создание объекта Smart500 класса
Tarif*/
Smart500.begin_price(70000); /* Полю price объекта
Smart500 передается значение 70000*/
cout <<"Tarif = " << Smart500.show() << endl; /*
Вывод значения поля*/
cout <<"K oplate = " << Smart500.end_price(0.2)<<
endl; /*Вывод суммы к оплате с учетом НДС*/
```

```
_getch();
return 0;
}
```

Пример. В программе реализуется та же задача, что и в предыдущем примере, но уже при помощи класса **Tarif**.

```
#include <iostream>
#include <conio.h>
using namespace std;
/*Объявление класса */
class Tarif {
double price; //Закрытое поле
public:/* Открытые методы */
void begin_price(double bp) { price=bp;} /* Метод
задает значение для поля price: цену тарифа
(например мобильного оператора)*/
double end_price(double pr) { return price*(1+pr); }
/* Метод возвращает стоимость оплаты за тариф с
учетом НДС */
double show() {return price;}/* Метод возвращает
значение поля price */
};
int main()
{ Tarif Smart500; /*Создание объекта Smart500 класса
Tarif*/
Smart500.begin_price(70000); /* Полю price объекта
Smart500 передается значение 70000*/
cout <<"Tarif = " << Smart500.show() << endl; /*
Вывод значения поля*/
cout <<"K oplate = " << Smart500.end_price(0.2)<<
endl; /*Вывод суммы к оплате с учетом НДС*/
_getch();
return 0;
}
```

Результат выполнения обеих программ одинаков:

```
Tarif = 70000
K oplate = 84000
```

2.13. Использование в качестве полей класса массивов

Пример. В программе определяется класс **Tarif**, в котором есть два закрытых поля и три открытых метода, а также класс **Abonent**, одним из полей которого является символьный массив FIO.

```
#include <iostream>
#include <conio.h>
#include <cstring>
using namespace std;
/*Объявление класса */
class Tarif { /* Внутренний класс */
double price; //Закрытое поле
char *nazv; //Закрытое поле
public:/* Открытые методы */
void naz_tarif(char* nz) { strcpy(nazv, nz);} /*
Метод задает значение для поля nazv*/
void begin_price(double bp) { price=bp;} /* Метод
задает значение для поля price: цену тарифа
(например мобильного оператора)*/
double end_price(double pr) { return price*(1+pr); }
/* Метод возвращает стоимость оплаты за тариф с
учетом НДС */
double show() {return price;}/* Метод возвращает
значение поля price */
void show2() {cout << " Название тарифа " << nazv <<
endl;}
Tarif() {nazv=new char[10]; price=0;}
~Tarif() { delete nazv;}
};
class Abonent
{
public:
char *FIO[3];
Tarif tarif_ab;
Abonent(char *Fam, char *Ima, char *Otch, double
Cena, char *Nazvanie );
~Abonent();
};
Abonent::Abonent(char *Fam, char *Ima, char *Otch,
double Cena, char *Nazvanie)
{
```

```
FIO[0]=new char[15];  FIO[1]=new char[15];
FIO[2]=new char[15];

strcpy(FIO[0], Fam);
strcpy(FIO[1], Ima);
strcpy(FIO[2], Otch);

//cout << *FIO[0] << " " << *FIO[1] << " " <<
FIO[2]<< endl;
tarif_ab.begin_price(Cena);/* Полю price объекта
tarif_ab передается значение Cena*/
tarif_ab.naz_tarif(Nazvanie);
}
Abonent::~Abonent()
{delete [] FIO;
}
int main()
{
setlocale(LC_ALL, "Russian");

Abonent AB456("Холодова", "Елена", "Петровна",
109000, "Smart1"); /*Создание объекта AB456 класса
Abonent*/
cout << " ФИО абонента " << AB456.FIO[0] << " " <<
AB456.FIO[1] << " " << AB456.FIO[2] << endl;
AB456.tarif_ab.show2(); /* Вывод значения поля*/
cout << " Стоимость тарифа " <<
AB456.tarif_ab.show() << endl; /* Вывод значения
поля*/
cout << " К оплате " <<
AB456.tarif_ab.end_price(0.2)<< endl; /*Вывод суммы
к оплате с учетом НДС*/
_getch();
return 0;
}
```

Результат выполнения программы:

```
ФИО абонента Холодова Елена Петровна
Название тарифа Smart1
Стоимость тарифа 109000
К оплате 130800
```

2.14. Пример реализации проекта «Банк»

Рассмотрим класс, который представляет банковский счет. Для начала требуется определиться, что представляет банковский счет. Например, он должен содержать:

- имя банка, где этот счет открыт;
- сумму вклада;
- номер счета.

Теперь рассмотрим перечень операций, которые можно выполнять с банковским счетом:

- снять деньги со счета;
- увеличить сумму вклада;
- просмотреть сумму вклада;
- произвести аутентификацию клиента.

Листинг программного кода

```
#include <iostream>
#include <conio.h>
#include <cstring>
using namespace std;
class account // Объявление класса account
{
private: //закрытые поля и методы
 char *name_of_bank; // имя банка
 int number;  // номер счета
 double sum; // сумма на счете
 double decrease_sum (double get); /*метод для
снятия денег со счета*/
double increase_sum (double add);/*метод для
пополнения счета*/
 double get_sum()const; /*метод для просмотра
текущей суммы счета*/
public: // открытые методы
void getpincode(); /* метод для аутентификации
клиента */
/* Далее - объявление трех перегруженных
конструкторов. */
 account(); /* Первый – стандартный, без
параметров.*/
```

```
 account(const char* name, int n, double s); /*
Второй конструктор используется для инициализации
всех полей класса при создании объекта. */
explicit account(double s); /* Третий конструктор
инициализирует только одно поле, а остальные поля,
как и в случае с конструктором по умолчанию,
инициализируются при создании объекта значениями по
умолчанию. */

 ~account(); // Деструктор класса

};
// Реализация конструкторов и методов
account::account() // Конструктор по умолчанию
 :name_of_bank(0),number(0),sum(0.0) /* Все поля
первоначально инициализируются нулевыми значениями
*/
{
 cout << "Создание объекта с использованием
конструктора по умолчанию!" << endl;
 name_of_bank=new char[strlen("Belarusbank")+1];
 strcpy(name_of_bank, "Belarusbank");/* Полю
name_of_bank присваивается значение «Belarusbank» */
}
/* Второй конструктор используется для инициализации
всех полей класса при создании объекта. После
круглых скобок перед телом конструктора через
двоеточие идет список инициализации полей объекта */
account::account(const char* name, int n, double s)
 :name_of_bank(0),number(n),sum(s)
{
 cout << "Создание объекта с использованием второго
конструктора!" << endl;
 name_of_bank=new char[strlen(name)+1];
 strcpy(name_of_bank, name); /* Полю name_of_bank
присваивается значение параметра  name */
}
/* Третий конструктор инициализирует только одно
поле */
account::account(double s)
 :name_of_bank(0),number(111),sum(s)
{
```

```
 cout << "Создание объекта с использованием третьего
конструктора!" << endl;
 name_of_bank=new
char[std::strlen("Belarusbank")+1];
 strcpy(name_of_bank, "Belarusbank");
}
account::~account()
{
 delete [] name_of_bank;
}
/* Метод get_sum возвращает текущую сумму счета. Эта
функция не изменяет значения полей класса. Такие
методы называются константными методами класса и для
их определения желательно использовать ключевое
слово const. Если определить метод double get_sum()
и второй метод double get_sum()const, то данные
методы будут считаться разными! По спецификатору
const можно перегружать функции-методы */
double account::get_sum( )const
{return sum;}
/* Методы increase_sum и decrease_sum соответственно
увеличивают и уменьшают значение поля sum и
возвращают окончательную сумму счета. Поле sum
является закрытым, однако оно доступно методам
класса. Кроме того, выше упомянутые методы также
являются закрытыми, поскольку объявлены под ключевым
словом private. Они не могут быть вызваны напрямую,
а только внутри других методов класса, как,
например, в функциях increase_sum и getpincode */
double account::increase_sum (double add)
{
 sum += add;
 return get_sum();
}
double account::decrease_sum (double get)
{
 sum -= get;
 return sum;
}
/* Метод getpincode требует введения кода, который
совпадает с номером счета. Если введенное число не
совпадает с номером счета, выдается сообщение Access
```

```
denied. В случае соответствия пользователю
предлагается выбрать то или иное действие, пока не
будет введен нуль */

void account::getpincode ()
{
 int k;
 cout << "Введите пароль" << endl;
 cin >> k;
 if (k == number)
 {
 int choice;
 do
 {
 cout << "Для просмотра суммы нажмите 1" << endl;
 cout << "Для снятия денег нажмите 2" << endl;
 cout << "Для пополнения счета нажмите 3" << endl;
 cout << "Для выхода нажмите 0" << endl;
 cin >> choice;
 switch (choice)
 {
 case 1:
 cout << "На вашем счете  " << get_sum() <<endl;
 break;
 case 2:
 cout << "Введите сумму: " << endl;
 double m;
 cin >> m;
 sum = decrease_sum (m);
 break;
 case 3:
 cout << "Введите сумму: " << endl;
 double s;
 cin >> s;
 sum = increase_sum(s);
 break;
 default:
 break;
 }
 continue;
 } while (choice != 0);
 }
```

```
 else
 {
 cout << "Доступ запрещен! " << endl;
 }}
// Главная программа
int main()
{
    setlocale(LC_ALL,"Russian");
account myAccount; /* Создание объекта myAccount
класса account */
myAccount.getpincode(); /* Вызов метода getpincode()
объекта*/
account *hisAccount = new account("Минский
банк",123,234.56); /* Создание объекта в
динамической памяти через указатель с использованием
второго конструктора */
hisAccount->getpincode();/* Вызов метода
getpincode() объекта*/
delete hisAccount; /* Удаление объекта */
hisAccount = 0; /* Обнуление указателя */
account herAccount(321.45); /* Создание объекта
herAccount с использованием третьего конструктора */
herAccount.getpincode();/* Вызов метода getpincode()
объекта*/
_getch();
return 0;
}
```

3. РАБОТА С ОБЪЕКТАМИ

В этом разделе учебно-методического пособия рассматриваются более сложные приемы, используемые при работе с объектами в рамках концепции объектно-ориентированного программирования.

3.1. Указатели на объекты

Для объектов могут создаваться указатели, как и для переменных базовых типов. Значением указателя является адрес первой ячейки области памяти, выделенной под объект.

Синтаксис объявления указателя на объект:

имя_класса **иимя_указателя*;

Чтобы получить адрес памяти, по которому записан объект, перед именем объекта указывают оператор &. Если перед именем переменной-указателя на объект указать оператор *, получим значение переменной, на которую ссылается указатель, то есть получаем доступ непосредственно к объекту.

Можно обратиться к объекту либо непосредственно (как это делалось во всех предыдущих примерах), либо посредством указателя на этот объект.

При обращении к конкретному элементу объекта посредством указателя на объект вы должны использовать оператор-стрелку: –> (ввести знак «минус», а затем знак «больше»). После указателя через стрелку указывается имя поля или метода.

Пример. Демонстрация создания и использования указателей на объекты.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Sale /*Объявление класса*/
{
public: /*Открытые элементы класса*/
```

```
int price;/*цена товара */
int number;/*количество товара*/
void total(int i_p, int i_n); /*Прототип метода*/
};
void Sale::total(int i_p, int i_n)/* Реализация
метода класса */
{price=i_p; number=i_n;}
int main()
{
Sale clothing; /* создание объекта clothing класса
Sale */
Sale *p_tov; /* создание указателя p_tov на объект
типа Sale */
clothing.total(56000,10); /* вызов метода через
clothing */
cout << clothing.price << " " << clothing.number <<
endl;
p_tov=&clothing;/*присвоить указ-лю p_tov адрес
объекта clothing*/
     /* вызов метода посредством указателя на
clothing */
p_tov->total(32000,10);
/* или можно так: (*p_tov).total(32000,10); */
cout << p_tov->price << " " << (*p_tov).number <<
endl;
_getch();
return 0;}
```

Результат выполнения программы:

```
56000 10
32000 10
```

В программе объявляется класс **Sale**. В функции **main()** создается объект **clothing** класса **Sale**, а также указатель **od** на объект этого класса (команда `Sale *od;`).

Значение указателя определяется командой `od=&clothing;`, в силу чего переменной-указателю **od** в качестве значения присваивается адрес памяти для объекта **clothing**. Доступ к членам объекта возможен в форматах:

od-> price *или* (*od). price *или* clothing.price
od-> number *или* (*od).number *или* clothing.number
od-> total *или* (*od).total *или* clothing.total

Доступ к объекту напрямую

Доступ к объекту посредством указателя

Когда указатель инкрементируется или декрементируется, он фактически увеличивается или уменьшается таким образом, чтобы указывать на следующий элемент его базового типа. То же самое происходит при операциях инкремента или декремента указателя на объект: указатель начинает указывать на следующий объект. Для иллюстрации этого правила изменим предыдущую программу так, чтобы **clothing** стал трехэлементным массивом типа **Sale**. Для обращения к элементам массива над указателем выполняются операции инкремента и декремента.

Пример. Использование операций инкремента и декремента для указателей на объекты.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Sale /*Объявление класса*/
{
int price;/*цена товара */
int number;/*количество товара*/
public: /*Открытые элементы класса*/
Sale(int i_p, int i_n); /* Прототип конструктора */
void Display(); /* Прототип метода Display */
};//Конец объявления класса
Sale::Sale(int i_p, int i_n)/* Реализация
конструктора класса */
{price=i_p; number=i_n;}
void Sale::Display() /* Реализация метода Display
класса */
// Вывод закрытых полей и их произведения
{cout << price << " " << number << " " <<
price*number << endl; }
```

```
int main() {
Sale
clothing[3]={Sale(100,10),Sale(200,20),Sale(300,30)}
;/*создание и инициализация массива clothing класса
Sale */
Sale *od; /* создание указателя od на объект типа
Sale */
od=&clothing[0]; /* получить указатель на первый
объект массива */
od->Display(); /* вывести значение clothing[0] через
указатель */
od++; /* переход к следующему объекту массива (к
clothing[1]) */
od->Display();/* вывести значение clothing[1] через
указатель */
od++; /* переход к следующему объекту массива (к
clothing[2]) */
od->Display(); /* вывести значение clothing[2] через
указатель */
od--; /* вернуться к предыдущему объекту массива (к
clothing[1]) */
od->Display(); /* вывести значение clothing[1] через
указатель */
_getch();
return 0; }
```

Результат выполнения:

```
100 10 1000
200 20 4000
300 30 9000
200 20 4000
```

Указатели на объекты играют центральную роль в одной из наиболее важных концепций C++ – полиморфизме.

Пример. В главной программе создается массив из 5 объектов с одновременной инициалиализацией с использованием параметризованного конструктора. Данные выводятся на экран и в текстовый файл. Доступ к полям объектов осуществляется различными способами.

```
/*Демонстрация доступа к полям объекта посредством
указателя */
#include <iostream>
#include <fstream> //для работы с потоками
#include <cstring> /* для работы со строковыми
функциями */
#include <conio.h>
using namespace std;
class APTEKA { // Объявление класса APTEKA
public:
int  IDmed;  //  код лекарства
char Nazv[15];  // название
double Price;  // цена
int AvBegin;  // наличие на начало месяца
int AvEnd;   // остаток на конец месяца
//объявление конструктора
APTEKA(int  p_IDmed, char p_Nazv[15], double
p_Price,int p_AvBegin, int p_AvEnd);
~APTEKA(); //объявление деструктора
int Rashod(); /* прототип метода без параметров
Rashod */
double Nalog(double Pr); /* прототип метода с
параметрами  Nalog  */
};
//Реализация параметрического конструктора
APTEKA::APTEKA(int  p_IDmed, char p_Nazv[15], double
p_Price, int p_AvBegin, int p_AvEnd)
{ IDmed=p_IDmed; strcpy(Nazv, p_Nazv);
Price=p_Price;
  AvBegin=p_AvBegin; AvEnd=p_AvEnd;
}
APTEKA::~APTEKA()
{cout << "DELETE..."; } //Реализация деструктора
int APTEKA::Rashod() //Реализация метода Rashod
    { return AvBegin - AvEnd;}
double APTEKA::Nalog(double Pr) /* Реализация метода
Nalog */
    { return (AvBegin - AvEnd)*Price*Pr;}
int main()
{
/* Объявление и инициализация массива Preparat для
хранения 5 объектов класса APTEKA */
```

```
APTEKA Preparat[5]={
    APTEKA ( 25, "citramon", 4500, 20, 0),
    APTEKA ( 35, "fenkarol", 2500, 39, 2),
    APTEKA ( 45, "antigrip", 3500, 59, 4),
    APTEKA ( 55, "teraflex",1500, 27, 15),
    APTEKA ( 65, "vitaminC", 6500, 47, 30)
                    };
APTEKA *p_Prep; /* Создание указателя на объект типа
APTEKA */
p_Prep=&Preparat[0];/*Присвоить указателю p_Prep
адрес первого объекта массива Preparat */
ofstream fout;// Открытие файла для вывода
fout.open("text.txt",ios_base::out|ios_base::binary)
;
 if(!fout.is_open())  { cout << "Can't open file!"
<< endl;  }
 else
 {  cout << "File has been opened!" << endl;
 for (int i=0; i<=4; i++) /* Форматированный вывод в
файл */
{  /* доступ к полям осуществляется с помощью
указателя */
fout << p_Prep->IDmed << "\t"; /* "\t" - знак
табуляции */
fout << p_Prep->Nazv << "\t";
fout << p_Prep->Price << "\t";
fout << p_Prep->AvBegin << "\t";
fout << p_Prep->AvEnd << "\t";
fout << p_Prep->Rashod() << "\t";
fout << p_Prep->Nalog(0.25) << "\t";
p_Prep++; // инкрементирование указателя
fout << "\r\n"; //перевод курсора на новую строку
}
 }// Конец вывода в файл
fout.close(); // закрытие текстового файла
p_Prep=&Preparat[0];/*Присвоить указателю p_Prep
адрес первого объекта массива Preparat */
// Вывод значений полей  объекта на экран
cout << "Preparats: dostup STRELKA" << endl;
for (int i=0; i<=4; i++)
{
cout << " Kod- " << p_Prep->IDmed << endl;
```

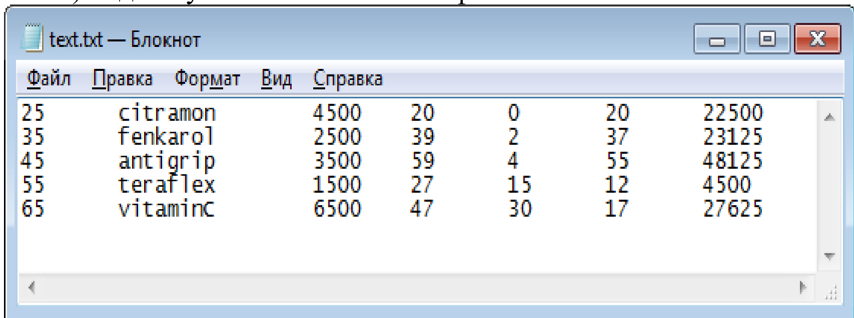
```
cout << " Nazvanie- " << p_Prep->Nazv << endl;
cout << " Cena-  " << p_Prep->Price << endl;
cout << " Begin- " << p_Prep->AvBegin << endl;
cout << " End- " << p_Prep->AvEnd << endl;
cout << " Rashod- " << p_Prep->Rashod() << endl;
cout << " Nalog- " << p_Prep->Nalog(0.25)<< endl;
cout <<endl;
p_Prep++; }
getch(); return 0; }
```

Результаты выполнения программы:

а) вид консольного окна:

```
File has been opened!
Preparats: dostup STRELKA
 Kod- 25
 Nazvanie- citramon
 Cena-  4500
 Begin- 20
 End- 0
 Rashod- 20
 Nalog- 22500

 Kod- 35
 Nazvanie- fenkarol
 Cena-  2500
 Begin- 39
 End- 2
 Rashod- 37
 Nalog- 23125

 Kod- 45
 Nazvanie- antigrip
 Cena-  3500
 Begin- 59
 End- 4
 Rashod- 55
 Nalog- 48125

 Kod- 55
 Nazvanie- teraflex
 Cena-  1500
 Begin- 27
 End- 15
 Rashod- 12
 Nalog- 4500

 Kod- 65
 Nazvanie- vitaminC
 Cena-  6500
 Begin- 47
 End- 30
 Rashod- 17
 Nalog- 27625
```

б) вид полученного текстового файла:



25	citramon	4500	20	0	20	22500
35	fenkarol	2500	39	2	37	23125
45	antigrip	3500	59	4	55	48125
55	teraflex	1500	27	15	12	4500
65	vitaminC	6500	47	30	17	27625

3.2. Ключевое слово this

Существует один особый указатель, который неявно передается каждому методу класса. Это указатель на объект, из которого вызывается метод. Чтобы получить значение указателя на вызывающий метод объект, используют ключевое слово **this**.

Каждый раз, когда активизируется метод класса, он автоматически получает указатель с именем **this** на объект, для которого он вызван. Указатель **this** является неявным параметром всех методов. Таким образом, внутри методов **this** может быть использован для обращения к данному объекту.

Метод может непосредственно обращаться к закрытым данным своего класса. Например, если описан такой класс:

```
class Test {
int i;
void f() { ... };
} ;
```

то внутри функции `f()` для присваивания переменной `i` значения может быть использовано такое предложение:

```
i = 10;
```

Фактически, однако, это предложение является сокращенным вариантом следующего:

```
this->i = 10;
```

Пример. Использование указателя **this**.

```
#include <iostream>
#include <conio.h>
using namespace std;
class Sale{
public:
int price;
int number;
void show(){
cout<<"price = "<<this->price<<"  ";
cout<<"number = "<<this->number<<endl;}
void set(int p,int n){
this->price=p;
this->number=n;}
};
int main(){
Sale a,b;
a.set(69,20);
b.set(88,10);
a.show();
b.show();
_getch();
return 0;
}
```

Результат выполнения программы:

```
price = 69  number = 20
price = 88  number = 10
```

3.3. Ссылки на объекты

На объекты, как и на переменные базовых типов, можно выполнять ссылки. **Ссылка** – это фактически псевдоним объекта или переменной. Существует несколько способов использования ссылок.

Независимая ссылка на объект позволяет использовать для одного и того же объекта разные названия. Ссылка при объяв-

лении сразу инициализируется. В качестве значения ссылки указывается объект, для которого создается ссылка. Перед именем ссылки в объявлении указывается оператор **&**. В качестве типа ссылки указывают имя класса, на объект которого выполняется ссылка. После создания ссылки к объекту можно обращаться через его имя или через имя ссылки на этот объект.

Синтаксис:

имя_класса *имя_объекта*; // Создание объекта класса

имя_класса &имя_ссылки=имя_объекта; /* Создание независимой ссылки на объект */

Пример. Использование независимой ссылки на объект.

```
#include <iostream>
#include <conio.h>
using namespace std;
class Sale  //Объявление класса Sale
{
public:
    double price; // цена
    int number; // количество
};
int main(){

Sale prod;//Создание объекта prod класса Sale

Sale &ref=prod; /*Создание независимой ссылки ref
 на объект prod. Теперь к объекту можно обращаться
 либо через имя prod, либо через имя ref */

/* Обращение к полям объекта по его имени и через
ссылку: */
prod.price=23.5;
cout<<"price = "<<ref.price<<endl;
ref.number=10;
cout<<"number = "<<prod.number<<endl;
_getch();
return 0;
}
```

Результат выполнения программы:

```
price = 23.5
number = 10
```

3.4. Передача объектов функциям в качестве аргументов

Объекты могут передаваться в качестве аргументов функциям и методам, как и переменные базовых типов. В этом случае в качестве типа передаваемого объекта указывается имя класса, к которому принадлежит соответствующий объект.

Как и с базовыми типами, объекты в аргументах функции можно передавать по значению и по ссылке. По умолчанию объекты передаются по значению. Это означает, что в функцию передается не сам объект, а его *копия*. Следовательно, изменения объекта внутри функции не отражаются на объекте, использованном в качестве аргумента функции.

Пример. Передача методу класса и внешней функции объектов по значению (при передаче аргумента по значению в функции обрабатывается копия реального аргумента).

```
#include <iostream>
#include <conio.h>
#include <iomanip>
using namespace std;
class Tovar {     // Первый класс Товар
public:
double price; // Цена единицы товара
int number; // Количество товара
double cost; // Стоимость товара
double calc_cost() //Вычисление стоимости
{ cost = price*number;
return 0;}
 };
class Skidka {     //Второй класс Скидка
public: /* Переменные S1, S2, S3 – проценты скидок*/
double S1; double S2; double S3;
```

```
/* Методу sum_sk класса Skidka должен передаваться
объект типа Tovar: */
double sum_sk(Tovar tov)
{/* Метод вычисляет сумму предоставляемой скидки. В
зависимости от цены и количества товара
предоставляется разный процент скидки */
/* В функции поле cost объекта (типа класса Tovar)
изменяется и возвращается в качестве результата
функции */

if ((tov.price > 50000) && (tov.number > 50))
{tov.cost*=(1-S1); return tov.cost;}
    else if ((tov.price > 40000) && (tov.number >
40)) {tov.cost*=(1-S2); return tov.cost;}
else {tov.cost*=(1-S3); return tov.cost;}
    }
};
void show(Skidka sk); /* Прототип внешней функции,
которой в качестве аргумента должен передаваться
объект класса Skidka*/
int main(){
double sk_cost; /* Стоимость со скидкой */
setlocale(LC_ALL, "Russian"); /* Функция, которая
позволяет выводить текст на русском */
Tovar Dress; Skidka skDress; //Создание объектов
cin >> Dress.number; // Ввод количества товара
cin >> Dress.price;// Ввод цены товара
Dress.calc_cost(); /* Вычисляется стоимость товара,
то есть значение переменной cost объекта Dress
класса Tovar */
skDress.S1=0.20; skDress.S2=0.15; skDress.S3=0.10;
show(skDress); /* Внешней функции show передается
объект skDress класса Skidka */
cout << "Количество товара: " << Dress.number <<
endl;
cout << "Цена товара: " << Dress.price << endl;
cout << "Стоимость: ";
printf("%.2f",Dress.cost); cout << endl;
cout << "Со скидкой: ";
sk_cost=skDress.sum_sk(Dress); /* Переменной sk_cost
присваивается измененное значение cost, возвращаемое
```

```
методом sum_sk объекта skDress. В качестве аргумента
метод принимает объект Dress класса Tovar */
printf("%.2f",sk_cost);cout << endl; /* Выводится
стоимость со скидкой */
/* Изменение значения cost внутри функции sum_sk не
влияет на значение cost объекта Tovar внутри main */
cout << "Значение стоимости в классе Tovar не
изменилось: ";
printf("%.2f",Dress.cost); cout << endl;
_getch();
return 0; }
void show(Skidka sk){ /* Функция выводит на экран
варианты предоставляемых скидок */
cout << "Скидка1 = " << setw(4) << sk.S1 << " - при
покупке >50 товаров по цене >50000 р." << endl;
cout << "Скидка2 = " << setw(4) << sk.S2 << " - при
покупке >40 товаров по цене >40000 р." << endl;
cout << "Скидка3 = " << setw(4) << sk.S3 << " - в
остальных случаях" << endl;
}
```

Результат выполнения программы:

```
45
35000
Скидка1 =  0.2 - при покупке >50 товаров по цене >50000 р.
Скидка2 = 0.15 - при покупке >40 товаров по цене >40000 р.
Скидка3 =  0.1 - в остальных случаях
Количество товара: 45
Цена товара: 35000
Стоимость: 1575000,00
Со скидкой: 1417500,00
Значение стоимости в классе Tovar не изменилось: 1575000,00
```

При передаче аргумента по ссылке, поле объекта, переданного аргументом функции, меняется. Если в предыдущей программе в функцию **sum_sk** объект передавать по ссылке, то есть

```
double sum_sk(Tovar &tov)   ,
```

то результат выполнения программы будет выглядеть следующим образом:

```
45
35000
Скидка1 =  0.2 - при покупке >50 товаров по цене >50000 р.
Скидка2 = 0.15 - при покупке >40 товаров по цене >40000 р.
Скидка3 =  0.1 - в остальных случаях
Количество товара: 45
Цена товара: 35000
Стоимость: 1575000,00
Со скидкой: 1417500,00
Значение стоимости в классе Tovar не изменилось: 1417500,00
```

Передача объектов аргументами в методы и функции – подход достаточно эффективный, поскольку позволяет обрабатывать целиком объекты, а не передавать каждое поле отдельно.

3.5. Возвращение объектов функциями и методами

Объекты могут не только передаваться в функции, но и возвращаться из них. В качестве типа результата функции указывается имя класса, объект которого возвращается в качестве результата функции или метода. Поскольку возвращается объект, то в функции или методе необходимо создать временный, локальный объект такого же типа, как результат.

Пример. Возвращение в качестве результата объекта.

```
#include <iostream>
#include <conio.h>
using namespace std;
class Tovar {     // Первый класс Товар
public:
double price; // цена товара
 };
class Margin {     //Второй класс Наценка
public:
double margin; // процент наценки на товар
};
Tovar Retail_Price(Tovar tov, Margin mar) /* Функция
Retail_Price принимает два аргумента: объекты
классов Tovar и Margin*/
```

```
{
Tovar tmp; // создание временного объекта
tmp.price=tov.price+tov.price*mar.margin;
return tmp; /* Функция Retail_Price возвращает
объект класса Tovar с измененным значением price */
}
int main(){
Tovar Dress; // создание объекта Dress класса Tovar
Margin marginDress; /* создание объекта marginDress
класса Margin */
Dress.price=45000;marginDress.margin=0.3; /*
присваивание значений полям объектов*/
cout<< "Wholesale price dresses=  " << Dress.price
<< endl; /* вывод закупочной цены товара */
Dress=Retail_Price(Dress, marginDress); /* объекту
Dress присваивается результат вызова функции
Retail_Price */
cout<<  "Retail price dresses= " <<  Dress.price
<<endl; /* вывод розничной цены товара */
_getch();
return 0; }
```

Результат выполнения программы:

```
Wholesale price dresses=  45000
Retail price dresses= 58500
```

3.6. Конструктор копий объектов

Когда объект передается в функцию в качестве аргумента или возвращается из функции, по умолчанию создается копия этого объекта. Конструктор копий (или конструктор копирования) представляет собой специальный тип перегруженного конструктора, который активизируется автоматически, при потребности в копии объекта. Можно также определить и собственный конструктор копий для класса.

Общий синтаксис конструктора копий:

имя_класса (**const** имя_класса &объект) {
// инструкции конструктора
}

Пример. Демонстрация переопределения конструктора создания копии объекта.

```
#include <conio.h>
#include <iostream>
using namespace std;
class Sizes_washer { //объявление класса
public:  //поля класса
int height;
int width;
int depth;
Sizes_washer(int h,int w, int d) /*конструктор с параметрами */
{ height=h; width=w; depth=d;}

Sizes_washer(Sizes_washer &washer) /* конструктор создания копий с параметром типа Sizes_washer */
{height=washer.height+1;/* полю height создаваемого объекта присваивается значение поля height объекта-аргумента, увеличенное на 1 */
width=washer.width+2;/* полю width создаваемого объекта присваивается значение поля width объекта-аргумента, увеличенное на 2 */
depth=washer.depth+3;}

void show(){ //метод для вывода полей класса
cout<<"height= "<<height<<endl;
cout<<"width = "<<width<<endl;
cout<<"depth = "<<depth<<endl;
cout << endl;}
};//конец объявления класса
int main() {
int i;
Sizes_washer SamsungA(60,50,44);/* создание объекта SamsungA класса Sizes_washer */
Sizes_washer SamsungB=SamsungA; /* создание объекта SamsungB на основе SamsungA с использованием конструктора копий */
SamsungA.show();
SamsungB.show();
for(i=1;i<=3;i++){
```

```
    SamsungA=Sizes_washer(SamsungA); /* вызывается
конструктор создания копии */
/* Каждый раз поля объекта увеличиваются на 1, 2 и 3
соответственно*/
    SamsungA.show();}
    _getch();
    return 0;
}
```

Результат выполнения программы:

```
height= 60
width = 50
depth = 44

height= 61
width = 52
depth = 47

height= 61
width = 52
depth = 47

height= 62
width = 54
depth = 50

height= 63
width = 56
depth = 53
```

3.7. Дружественные функции

Как отмечалось ранее, члены класса могут быть закрытыми и открытыми. К закрытым членам класса доступ существует только внутри класса. Однако нередко случается необходимость, чтобы

член класса оставался закрытым, но некоторые внешние функции или методы имели к нему доступ. Такие внешние функции, которые имеют доступ к закрытым членам класса, называются дружественными функциями.

Чтобы задекларировать функцию как дружественную для класса, необходимо указать прототип этой функции в открытой (public) секции описания класса, предварив его ключевым словом **friend**.

Пример. Использование дружественной функции.

```
#include <iostream>
#include <conio.h>
using namespace std;
//Класс с закрытым полем:
class Year{
double year;
public:
Year(double y)
{year=y;}
//Дружественная функция:
friend void Display(Year god);
};
//Реализация дружественной функции:
void Display(Year god){
cout<<"Year = "<<god.year<<endl;}
int main(){
Year Year1(2014);
/* Дружественная функция имеет доступ к закрытым
членам: */
Display(Year1);
_getch();
return 0; }
```

Результат выполнения программы:

```
Year = 2014
```

Концепция дружественных функций может показаться несколько искусственной, но это не так. Но ситуация, когда есть два класса с закрытыми полями и необходимо получить доступ к полям обоих классов, один из тех случаев, когда целесообразным является применение дружественных функций.

Пример. Использование дружественной функции.

```
#include <iostream>
#include <conio.h>
using namespace std;
//Анонс класса:
class NDS;
//Класс с закрытым полем:
class COST{
double x;
public:
COST(double z){x=z;}
//Дружественная функция с двумя аргументами:
friend double summa(COST a, NDS b);
}a(3.5);
//Класс с закрытым полем:
class NDS{
double y;
public:
NDS(double z){y=z;}
//Дружественная функция с двумя аргументами:
friend double summa(COST a, NDS b);
}b(2.3);
double summa(COST a, NDS b){
return a.x+b.y;
}
int main(){
//Вызов дружественной функции:
cout<<"Total is "<<summa(a,b)<<endl;
_getch();
return 0;
}
```

Результат выполнения программы:

```
Total is 5.8
```

Дружественными по отношению к классу могут быть отдельные методы другого класса или целый класс. В последнем случае все методы дружественного класса имеют доступ к закрытым полям и методам исходного класса.

Пример. Дружественные методы.

```
#include <iostream>
#include <conio.h>
using namespace std;
//Анонс класса:
class B;
//Класс с закрытым полем и методом:
class A{
double x;
public:
A(double z){x=z;}
double summa(B b);
}a(3.5);
//Класс с закрытым полем и дружественным методом:
class B{
double y;
public:
B(double z){y=z;}
//Дружественный метод:
friend double A::summa(B b);
}b(2.3);
int main(){
//Вызов дружественного метода:
cout<<"Total is "<<a.summa(b)<<endl;
_getch();
return 0;
}
double A::summa(B b){
return x+b.y;}
```

Результат выполнения программы:

```
Total is 5.8
```

В следующем примере программный код несколько изменен так, чтобы один класс был дружественным к другому классу. При этом в дружественный класс добавлен еще один метод **product()** для вычисления произведения закрытых полей классов.

Пример. Дружественные классы.

```
#include <iostream>
#include <conio.h>
using namespace std;
class NDS;
class COST{
double x;
public:
COST(double z){x=z;}
double summa(NDS b);
double product(NDS b);
}a(3.5);
class NDS{
double y;
public:
NDS(double z){y=z;}
friend class COST;
}b(2.3);
int main(){
cout<<"Total is "<<a.summa(b)<<endl;
cout<<"Product is "<<a.product(b)<<endl;
_getch();
return 0;
}
double COST::summa(NDS b){
return x+b.y;}
double COST::product(NDS b){
return x*b.y;}
```

Результат выполнения программы:

```
Total is 5.8
Product is 8.05
```

Следует отметить, что дружественные классы используются не очень часто – существуют более эффективные способы реализации доступа методов одного класса к членам другого, главным среди которых является механизм *наследования*.

3.8. Динамическое выделение памяти под объекты

Для динамического выделения памяти под объекты класса используется оператор **new**. Сначала объявляется указатель на объект соответствующего класса, после чего под объект выделяется место в памяти и адрес передается указателю:

имя_класса *имя_указателя;

имя_указателя = **new** имя_класса;

Например:

```
Sale *goods;
goods = new Sale;
```

Если при создании объекта конструктору необходимо передать аргументы, они указываются в круглых скобках после имени класса.

Например:

```
Sale *goods;
goods = new Sale(25, "Paper", 23000);
```

Для удаления объекта из памяти используется оператор **delete**, после которого следует указатель на удаляемый объект:

delete имя_указателя;

Например:

```
delete goods;
```

Пример. Динамическое выделение памяти под объект.

```
#include <iostream>
#include <cstring>
#include <conio.h>
using namespace std;
```

```
class Tovar { // объявление класса Tovar
public:
char *Nazv; // название товара
double Price; // цена товара
void show_tovar(){ // вывод названия и цены товара
        cout<<"Название товара = "<< Nazv << " цена
" << Price << endl << endl;}
void show_cr() { /* вывод сведений о добавленном
товаре */
        cout << "Товар  " << Nazv << " с ценой " <<
Price << " добавлен.";
        cout << endl;}
Tovar() { // стандартный конструктор
        Nazv=new char[15];/* динамическое выделение
памяти для переменной*/
        strcpy(Nazv,"Без названия"); /* присваивание
значения переменной*/
        Price=0;
        show_cr();
        }
Tovar (char *N_T, double P_T) { /* конструктор с
параметрами */
        Nazv=new char[15];
        strcpy(Nazv,N_T);
        Price=P_T;
        show_cr(); // вызов метода show_cr
}
~Tovar(){ // деструктор
cout<< "Товар  " << Nazv << " с ценой " << Price <<
" удален!" << endl;
delete [] Nazv;  // освобождение памяти
cout << endl;}
}; // завершение объявления класса
int main(){
setlocale(LC_ALL,"Russian");
Tovar *Product; /* создание указателя Product1 на
объект класса Tovar*/
Product=new Tovar; /*динамическое выделение памяти
под объект. При создании объекта вызывается
конструктор без параметров */
Product->show_tovar(); /* вызов метода show_tovar с
помощью указателя на объект*/
```

```
delete Product; /* освобождение памяти, выделенной
под объект */
Product=new Tovar("Бумага", 3600);/*динамическое
выделение памяти под объект. При создании объекта
вызывается конструктор с параметрами */
Product->show_tovar();/* вызов метода show_tovar с
помощью указателя на объект*/
delete Product; /*освобождение памяти, выделенной
под объект*/
_getch();
return 0;
}
```

Результат выполнения программы:

```
Товар  Без названия с ценой 0 добавлен.
Название товара = Без названия цена 0

Товар  Без названия с ценой 0 удален!

Товар  Бумага с ценой 3600 добавлен.
Название товара = Бумага цена 3600

Товар  Бумага с ценой 3600 удален!
```

3.9. Динамическое выделение памяти под массивы объектов

При динамическом выделении памяти под массив объектов нельзя указать инициализатор, поэтому в классе обязательно должен быть конструктор без параметров. Если такого конструктора нет, то программа компилироваться не будет.

Пример. Динамическое выделение памяти под массив объектов.

```
#include <iostream>
#include <cstring>
#include <conio.h>
```

```
const int NAME_LENGH = 15; /*максимальный размер
названия товара*/
using namespace std;
class Tovar { // объявление класса Tovar
public:
        char *Nazv; // название товара
        double Price; // цена товара

Tovar() // стандартный конструктор
{
    Nazv= new char[NAME_LENGH];
    strcpy(Nazv," ");
    Price=0;}
void Assign_Tovar (char *N_T, double P_T)
{  /* функция, которая присваивает значения полям
класса */
    strncpy(Nazv,N_T,NAME_LENGH);
    Price=P_T;
}
void show_tovar()// вывод названия и цены товара
    {
    cout<<"Название товара = "<< Nazv << " цена " <<
Price << endl << endl;
    }
~Tovar(){ // деструктор
    cout<< "Товар  " << Nazv << " с ценой " << Price
<< " удален!" << endl;
    delete [] Nazv;
    cout << endl;}
}; // завершение объявления класса

int main()
{setlocale(LC_ALL,"Russian");
Tovar *Product; /* создание указателя Product на
объект класса Tovar */

Product=new Tovar[2]; /*динамическое выделение
памяти под массив из 2-х объектов.
При создании объектов вызывается конструктор без
параметров */
```

```
Product->Assign_Tovar("Бумага", 53600);/* вызов
метода для 1-го объекта массива через указатель */

Product->show_tovar(); /* вызов метода show_tovar с
помощью указателя на объект*/

(Product+1)->Assign_Tovar("Ручка", 3200);/* вызов
метода для 2-го объекта массива через указатель */

Product[1].show_tovar(); /* вызов метода show_tovar
с помощью указателя на объект*/

delete [] Product; /*освобождение памяти, выделенной
под массив объектов*/
getch();
return 0;
}
```

Результат выполнения программы:

```
Название товара = Бумага цена 53600

Название товара = Ручка цена 3200

Товар  Ручка с ценой 3200 удален!

Товар  Бумага с ценой 53600 удален!
```

Пример. Демонстрация реализации динамического массива объектов. Элементы динамического списка – объекты класса, одно из полей которого является указателем на объект того же класса.

```
/* Формирование списка реализуется через цепочку
последовательных ссылок. Для каждого объекта из
списка память выделяется динамически */
#include <iostream>
#include <conio.h>
#include <iomanip>
using namespace std;
//Класс для объектов-элементов списка:
class Numbers{
```

```
public:
int n;
Numbers *p;
};
//Функция для создания списка:
Numbers *make(int N){
Numbers *p1,*p2;
int i;
p1=new Numbers;
p1->n=1;
p1->p=NULL;
for(i=2;i<=N;i++){
   p2=new Numbers;
   p2->n=i;
   p2->p=p1;
   p1=p2;}
return p1;}
//Функция для отображения элементов списка:
void showAll(Numbers *q){
do{
   cout<<q->n<<" : "<<q<<endl;
   q=q->p;
}while(q!=NULL);
}
//Функция для удаления списка из памяти:
void deleteAll(Numbers *q){
Numbers *q1;
do{
   q1=q;
   cout<<"deleted: "<<q<<endl;
   q=q1->p;
   delete q1;
}while(q!=NULL);
}
int main(){
int n;
Numbers *q;
cout<<"Enter n = ";
cin>>n;
q=make(n);
showAll(q);
deleteAll(q);
```

```
_getch();
return 0; }
```

Результат выполнения программы:

```
Enter n = 3
3 : 00438700
2 : 004386B8
1 : 00438670
deleted: 00438700
deleted: 004386B8
deleted: 00438670
```

3.10. Наследование в С++

Наследование является фундаментальной концепцией объектно-ориентированного программирования, цель которого состоит в повторном использовании созданных классов. Наследование представляет собой способность производить новый класс (производный класс) из существующего (базового) класса. Производный класс наследует элементы базового класса. Наследование в С++ реализовано таким образом, что наследуемые компоненты не перемещаются в производный класс, а остаются в базовом классе. Производный класс может переопределять и доопределять функции-элементы базовых классов.

Класс может быть порожден от одного *(простое наследование)* или более классов *(множественное наследование)*.

Основные концепции наследования классов в С++:

•Для инициализации элементов производного класса программа должна вызвать конструкторы базового и производного классов.

•Используя оператор точку, программы могут легко обращаться к элементам базового и производного классов.

•Для разрешения конфликта имен между элементами базового и производного классов программа может использовать оператор глобального разрешения, указывая перед ним имя базового или производного класса.

Возможность доступа к элементам базового класса определяется с помощью соответствующего спецификатора доступа:

private – доступные элементы базового класса могут использоваться только внутри производного класса;

protected – доступные элементы базового класса могут использоваться только внутри производного класса и классов, для которых данный производный класс является базовым;

public – доступные элементы базового класса могут использоваться любыми объектами, имеющими доступ к данному классу.

Спецификатор доступа **protected** аналогичен **private**, только он позволяет получить доступ к защищённым данным из производного класса.

Производный класс может обращаться к общим элементам базового класса, как будто они определены в производном классе.

Чтобы обеспечить производным классам прямой доступ к определенным элементам базового класса, в то же время защищая эти элементы от оставшейся части программы, C++ обеспечивает защищенные (protected) элементы класса. Производный класс может обращаться к защищенным элементам базового класса, как будто они являются общими. Однако для оставшейся части программы защищенные элементы эквивалентны частным.

Внутри конструктора производного класса программа должна вызвать конструктор базового класса, указывая двоеточие, имя конструктора базового класса и соответствующие параметры сразу же после заголовка конструктора производного класса.

Если функциям производного класса необходимо обратиться к элементу базового класса, то используется оператор глобального разрешения.

3.10.1. Простое наследование

Предположим, что есть базовый класс **firma**. Предположим, что программе требуется класс **manager**, который добавляет новые элементы данных в класс **firma**.

В данном случае можно выбрать два варианта:

во-первых, в программе можно создать новый класс **manager**, который дублирует многие элементы класса **firma**;

во-вторых, программа может породить класс типа **manager** из базового класса **firma**.

Порождая класс **manager** из существующего класса **firma**, снижается объем требуемого программирования и исключается дублирование кода внутри программы.

Для определения этого класса необходимо указать ключевое слово **class**, имя **manager**, следующее за ним двоеточие и имя **firma**:

```
class manager : public firma
{
элементы
};
```

Когда порождается класс из базового класса, частные элементы базового класса доступны производному классу только через интерфейсные функции базового класса. Таким образом, производный класс не может напрямую обратиться к закрытым элементам базового класса, используя оператор точку.

Когда порождается класс из базового класса, конструктор производного класса должен вызвать конструктор базового класса.

Пример. Программа иллюстрирует использование наследования в C++, создавая класс **manager** из базового класса **firma**. Программа определяет базовый класс **firma**, а затем определяет производный класс **manager**.

```
#include <iostream>
#include <cstring>
#include <conio.h>
using namespace std;
/* Описание базового класса firma */
class firma // Базовый класс
{
public:
   firma(char *, char *, float);
   void show_firma(void);
private:
   char name [30]; // имя
   char position[30]; // должность
```

```
   float salary; // оклад
}; /* ------- */
/*Реализация параметрического конструктора firma
базового класса*/
firma::firma(char *name, char *position,
                   float salary)
{  strcpy(firma::name, name);
   strcpy(firma::position, position);
   firma::salary = salary;
}/* ------- */

/*Реализация открытого метода show_firma базового
класса для вывода данных объекта базового класса*/
void firma::show_firma(void)
{
   cout << "ФИО: " << name << endl;
   cout << "Должность: " << position << endl;
   cout << "Оклад: " << salary << endl;
} /*   ---------   */

/* производный класс manager из базового класса
firma */
/*Ключевое слово public, которое предваряет имя
класса firma, указывает, что общие (public) элементы
класса firma также являются общими и в классе
manager. */
class manager : public firma
{
public:
// конструктор класса
   manager(char *, char *, char *, float, float,
int);
   void show_manager(void); //метод класса
private:
   float annual_bonus; // ежегодная премия
   char company_car[30];// машина компании
   int stock_options; // уставной фонд
};/* ------- */

/*Чтобы вызвать конструктор базового класса,
поместите двоеточие сразу же после конструктора
производного класса, а затем укажите имя
```

```
конструктора базового класса с требуемыми
параметрами: */
/*Реализация параметрического конструктора
производного класса*/
manager::manager(char *name, char *position,
char *company_car, float salary, float bonus,
int stock_options) : firma(name, position, salary)
/*Конструктор базового класса */
{
   strcpy(company_car, company_car) ;
   annual_bonus = bonus ;
   stock_options = stock_options;
}/* ------- */

/*Реализация метода show_manager производного
класса*/
void manager::show_manager(void)
{  show_firma();
   cout << "Машина компании: " << company_car <<
endl;
   cout << "Премия: " << annual_bonus << endl;
   cout << "Фондовый опцион: " << stock_options <<
endl;}
/* ------- */
/*Обратите внимание, что функция show_manager
вызывает функцию show_firma, которая является
элементом класса firma. Поскольку класс manager
является производным класса firma, класс manager
может обращаться к общим элементам класса firma, как
если бы все эти элементы были определены внутри
класса manager.*/
int main()
{  setlocale(LC_ALL,"Russian");
   firma worker("Дмитрий", "Программист", 35000);
   manager boss("Юрий", "Директор ", "Lexus",
                 50000.0, 5000, 1000);
   worker.show_firma() ;
   boss.show_manager();
   _getch();
   return 0; }
```

Результат выполнения программы:

```
ФИО: Дмитрий
Должность: Программист
Оклад: 35000
ФИО: Юрий
Должность: Директор
Оклад: 50000
Машина компании: Lexus
Премия: 5000
Фондовый опцион: 1000
```

3.10.2. Множественное наследование

Множественное наследование представляет собой возможность порождать класс из нескольких базовых классов. При использовании множественного наследования для порождения класса конструктор производного класса должен вызвать конструкторы всех базовых классов, передавая им необходимые параметры. При порождении класса из производного класса создается иерархия наследования (иерархия классов). Для порождения класса из нескольких базовых после имени нового класса и двоеточия указываются через запятую имена базовых классов. При порождении классов может случиться так, что используемый базовый класс реально порожден из других базовых классов. Если так, то программа создает иерархию классов. Если вызывается конструктор производного класса, то вызываются также и конструкторы наследуемых классов (последовательно).

Пример. Программа порождает класс **computer**, используя базовые классы **computer_screen** и **mother_board**.

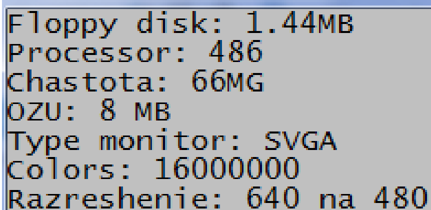
```
#include <iostream>
#include <cstring>
#include <conio.h>
using namespace std;
class computer_screen //1-й базовый класс
```

```
{
public:
   computer_screen(char *, long, int, int);
   void show_screen(void);
private:
   char type[32];
   long colors;
   int x_resolution;
   int y_resolution;
};
computer_screen::computer_screen(char *type, long
colors, int x_res, int y_res)
{
   strcpy(computer_screen::type, type);
   colors = colors;
   x_resolution = x_res;
   y_resolution = y_res;
}
void computer_screen::show_screen(void)
{
   cout << "Type monitor: " << type << endl;
   cout << "Colors: " << colors << endl;
   cout << "Razreshenie: " << x_resolution << " na "
<< y_resolution << endl;
}
class mother_board  //2-й базовый класс
{
public:
   mother_board(int, int, int);
   void show_mother_board(void);
private:
   int processor;
   int speed;
   int RAM;
};
mother_board::mother_board(int processor, int speed,
int ram)
{
   processor = processor;
   speed = speed;
   RAM = ram;
}
```

```
void mother_board::show_mother_board(void)
{
   cout << "Processor: " << processor << endl;
   cout << "Chastota: " << speed << "MG" << endl;
   cout << "OZU: " << RAM << " MB" << endl;
}
/* Класс указывает свои базовые классы сразу после
двоеточия, следующего за именем класса computer */
class computer : public computer_screen, public
mother_board
{
public:
   computer(char *, int, float, char *, long, int,
int, int, int, int);
   void show_computer(void);
private:
   char name [64];
   int hard_disk;
   float floppy;
};
/*конструктор класса computer вызывает конструкторы
классов mother_board и computer_screen */
computer::computer(char *name, int hard_disk, float
floppy, char *screen, long colors, int x_res, int
y_res, int processor, int speed, int ram) :
computer_screen(screen, colors, x_res, y_res),
mother_board(processor, speed, ram)
{
   strcpy(computer::name, name);
   computer::hard_disk = hard_disk;
   computer::floppy = floppy;
}
void computer::show_computer(void)
{
   cout << "Type: " << name << endl;
   cout << "Hard disk: " << hard_disk << "MB" <<
endl;
   cout << "Floppy disk: " << floppy << "MB" <<
endl;
   show_mother_board();
   show_screen();
}
```

```
int main()
{
   computer my_pc("Compaq", 212, 1.44, "SVGA",
16000000, 640, 480, 486, 66, 8);
   my_pc.show_computer();
_getch();
   return 0;
}
```

Результат выполнения программы:


```
Floppy disk: 1.44MB
Processor: 486
Chastota: 66MG
OZU: 8 MB
Type monitor: SVGA
Colors: 16000000
Razreshenie: 640 na 480
```

3.10.3. Защищенные элементы

Производный класс может обращаться к защищенным элементам базового класса напрямую, используя оператор точку. Однако оставшаяся часть программы может обращаться к защищенным элементам только с помощью интерфейсных функций этого класса. Защищенные элементы базового класса занимают промежуточное положение между общими (доступными всей программе) и частными (доступными только самому классу) элементами. Если элемент является защищенным, объекты производного класса могут обращаться к нему, как будто он является общим. Для оставшейся части программы защищенные элементы являются как бы частными. Единственный способ, с помощью которого программы могут обращаться к защищенным элементам, состоит в использовании интерфейсных функций. Следующее определение класса *book* использует метку **protected**, чтобы позволить классам, производным от класса *book*, обращаться к элементам *title*, *author* и *pages* напрямую, используя оператор точку:

```
class book
{
public:
   book(char *, char *, int) ;
   void show_book(void) ;
protected:
   char title [64];
   char author[64];
   int pages;
};
```

3.10.4. Построение иерархии классов

При использовании наследования в C++ для порождения одного класса из другого возможны ситуации, когда порождается класс из класса, который уже, в свою очередь, является производным от некоторого базового класса. Например, предположим, вам необходимо использовать класс *computer* базовый для порождения класса *workstation*, как показано ниже:

```
class work_station : public computer
{
public:
   work_station (char *operating_system, char
*name, int hard_disk, float floppy, char
*screen, long colors, int x_res, int
y_res, int processor, int speed, int RAM);
   void show_work_station(void);
private:
   char operating_system[64];
       };
```

Конструктор класса *workstation* просто вызывает конструктор класса *computer*, который, в свою очередь, вызывает конструкторы классов *computer_screen* и *mother_board*:

```
work_station::work_station( char *operating_system,
char *name, int hard_disk, float floppy, char
```

```
*screen, long colors, int    x_res, int y_res, int
processor, int speed, int RAM) : computer (name,
hard_disk, floppy, screen, colors, x_res, y_res,
processor, speed, RAM)
{
   strcpy(operating_system, operating_system);
}
```

В данном случае класс *computer* выступает в роли базового класса. Класс *computer* был порожден из классов *computer_screen* и *mother_board*. В результате класс *work_station* наследует характеристики всех трех классов.

Иерархия классов, а следовательно и количество наследуемых элементов может быть довольно длинной.

ЛИТЕРАТУРА

1. **Алейников, Д. В., Холодова, Е. П.** Язык программирования С++ : структурное программирование : учебно-методическое пособие : рекомендовано учебно-методическим объединением по образованию в области управления для студентов учреждений высшего образования специальности 1-26 03 01 Управление информационными ресурсами в качестве учебно-методического пособия / Д. В. Алейников, Е. П. Холодова. – Минск : Национальная библиотека Беларуси, 2014. – 114 с.
2. **Васильев, А. Н.** Самоучитель С++ с примерами и задачами / А. Н. Васильев. – СПб. : Наука и техника, 2015. – 480 с.
3. **Лафоре, Р.** Объектно-ориентированное программирование в С++. Классика Computer Science / Р. Лафоре. – 4-е изд. – СПб. : Питер, 2014. – 928 с. : ил.
4. **Прата, С.** Язык программирования С++ : лекции и упражнения / С. Прата. – М. : Вильямс, 2012. – 1248 с.
5. **Страуструп, Б.** Язык программирования С++ : специальное издание / Б. Страуструп. – М. : Бином, 2012. – 1136 с.
6. **Шилдт, Г.** Полный справочник по С++ / Г. Шилдт. – М. : Вильямс, 2014. – 800 с.
7. **Шилдт, Г.** С++ : базовый курс / Г. Шилдт. – М. : Вильямс, 2015. – 624 с.

Учебно-методическое пособие

ЯЗЫК ПРОГРАММИРОВАНИЯ C++

Введение в объектно-ориентированное программирование

Алейников Дмитрий Вячеславович
Холодова Елена Петровна
Силкович Юрий Николаевич

Компьютерная верстка М. И. Александровой
Дизайн обложки Т. В. Пешиной

Подписано в печать 20.11.2015. Формат 60x84$^1/_{16}$. Гарнитура Таймс.
Цифровая печать. Усл. печ. л. 7,70. Уч.-изд. л. 2,64. Тираж 50 экз. Заказ 617.

Полиграфическое исполнение:
государственное учреждение «Национальная библиотека Беларуси».

Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий 1/398 от 02.07.2014.

Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий 2/157 от 02.07.2014.

Пр. Независимости, 116, 220114, Минск.
Тел. (+375 17) 293 27 68. Факс (+375 17) 266 37 23. E-mail: edit@nlb.by.