МОСКОВСКИЙ ИНЖЕНЕРНО-ФИЗИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)

Каферда № 46
"Компьютерные медицинские системы"

К. Г. Финогенов

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

"Основы объектно-ориентированного программирования"

Москва 2008

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

МОСКОВСКИЙ ИНЖЕНЕРНО-ФИЗИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)

К.Г. Финогенов

Основы объектно-ориентированного программирования

Лабораторный практикум

Москва 2008

УДК 004.432.(076.5)
ББК 32.973.202-018я7
Ф60

Финогенов К.Г. Основы объектно-ориентированного программирования. Лабораторный практикум. Учебное пособие. М.: МИФИ, 2008. 92 с.

Пособие посвящено современной идеологии разработки программных продуктов – объектно-ориентированному программированию (ООП). В первой части описываются основные понятия и концепции ООП, рассматривается аппарат ООП, изучается техника составления прикладных объектно-ориентированных программ.

Вторая часть пособия – описание лабораторного практикума по освоению разработки объектно-ориентированных приложений Windows.

Предназначено для обучения студентов кафедры компьютерных медицинских систем факультета автоматики и электроники МИФИ (специальность "Электроника и автоматика физических установок") по курсам "Языки программирования и операционные системы" и "Компьютерный практикум". Пособие может быть также полезно студентам, аспирантам и преподавателям, знакомым с языком C++ и программированием в системе Windows и желающим самостоятельно освоить современные методы разработки программных систем.

Рецензент В.В. Комаров
Рекомендовано редсоветом МИФИ в качестве учебного пособия

ISBN 978-5-7262-0896-1

Редактор Е.Е Шумакова

Подписано в печать 18.02.2008 г. Формат 60 × 84 1/16
Печ. л. 5,75 Уч.-изд. л. 5,75 Тираж 120 экз.
Изд. № 056-1 Заказ N 61

Московский инженерно-физический институт (государственный университет)
Типография МИФИ. 115409, Москва, Каширское шоссе, 31

Содержание

Часть 1
Теоретические сведения

1. КЛАССЫ И ОБЪЕКТЫ

Объектно-ориентированный подход к программированию

Объектно-ориентированное программирование (ООП) представляет собой относительно новую идеологию разработки программных продуктов. Занимаясь обычным, *процедурным* программированием, мы описываем в некотором месте программы требуемые для ее выполнения данные, а затем разрабатываем процедуры, или функции, последовательность выполнения которых определяет суть программы и способ обработки данных. Это *разделение данных и функций* является отличительной чертой процедурного программирования.

Суть объектно-ориентированного подхода заключается в том, что мы *объединяем* некоторые коллекции данных и функций, связанных с этими данными, в субстанции, называемые *классами*. Таким образом, класс – это совокупность данных и функций для их обработки (функции, входящие в состав класса, часто называют *методами*). Например, в класс **Image** может входить массив, предназначенный для хранения данных, получаемых в результате оцифровки изображения некоторого медицинского препарата (опухоли, клетки и др.), а также функции, позволяющие вводить реальный массив данных с диска и выводить его на экран, возможно, с изменением масштабов по осям или с прокруткой получаемого изображения.

Определяя впервые такой класс, мы, разумеется, должны продумать состав его данных и написать все требуемые для их обработки функции. Однако в дальнейшем о деталях содержимого класса можно позабыть. Образовав затем в программе *объект*, со-

ответствующий описанию класса и характеризуемый конкретными значениями данных, т. е. относящийся к конкретному изучаемому изображению, мы сразу же получаем и все данные, характеризующие этот объект, и необходимый набор функций для работы с ним.

В дальнейшем на основе разработанных ранее классов можно создавать новые, *производные*, в которые будут входить все средства базовых классов плюс какие-то дополнения. Например, от описанного выше класса **Image** может быть образован производный класс **Filter**, в котором предусмотрены функции цифровой фильтрации изображения по тем или иным алгоритмам с записью на диск получаемых отфильтрованных данных, а также класс **Retuoch**, позволяющий ретушировать изображения, выводимые на экран, вручную с помощью мыши. Разрабатывая эти производные классы, мы уже не должны заниматься средствами ввода и вывода изображения, поскольку эти функции были разработаны ранее, имеются в базовом классе **Image** и доступны во всех производных от него классах; для класса **Filter** следует только разработать, например, функцию **Gray()** для преобразования цветного изображения в полутоновое и функцию **Sharp()** для более отчетливого выделения границ опухоли, а для класса **Retouch** – функцию **Draw()** графического редактирования изображения на экране.

Данные, входящие в класс, описывают *состояние* объекта этого класса, а функции (методы) – его возможное *поведение*. Разумно сконструированный класс включает в себя все функции, необходимые для самостоятельного существования объектов этого класса. Наделение объектов максимумом самостоятельности (делегирование им максимума полномочий) является краеугольным камнем ООП. Для того чтобы подчеркнуть самостоятельность объектов в выполнении запланированных для них действий, запрос к объекту на реализацию им той или иной линии поведения (фактически выражающийся в вызове соответствующей функции класса) часто трактуют как посылку объекту *сообщения*.

Программирование теперь заключается не в последовательном выполнении некоторых операций над данными, входящими в программу, а в создании объектов определенных заранее классов и посылке этим объектам сообщений, запрашивающих те или иные варианты их поведения.

Можно предложить два основных стиля использования ООП в прикладных задачах. Первый заключается в самостоятельной разработке необходимой иерархии и состава классов, позволяющих реализовать тот или иной программный проект достаточной сложности. Достоинство такого подхода проявляется в том, что после определения состава и поведения объектов разработчик может забыть о деталях конкретных реализаций и сосредоточиться на более высоком уровне взаимодействия объектов друг с другом и с внешним миром. Важно подчеркнуть, что разработанные для решения конкретной задачи классы часто носят абстрактный характер, описывая такие объекты, как списки, деревья, структуры и пр. Это дает возможность использовать разработанный инструментарий для решения совсем иных задач. Например, класс, описывающий поведение связного списка некоторых объектов и включающий полный набор средств обслуживания этого списка (добавление и уничтожение элементов списка, упорядочение списка по различным критериям, вывод содержимого списка на печать и т. д.), с равным успехом можно использовать при разработке карточных игр (со списками – колодами карт), автоматизации отдела кадров предприятия (со списками сотрудников), создания гипертекстового справочника (со списками связанных терминов) и в других, столь же различающихся предметных областях.

Другой стиль реализации идей ООП заключается в использовании *уже разработанных* библиотек классов либо абстрактного характера, либо вполне конкретных. К последним, в частности, относятся библиотеки классов, служащих для разработки приложений Windows. Применение библиотек классов, во-первых, существенно (не на проценты, а во много раз) снижает трудоемкость программирования и уменьшает размер разрабатываемых программ, и, во-вторых, позволяет заметно повысить их качество. Последнее связано с тем, что реализация объектов Windows, включаемых в библиотеки классов, выполняется высоко квалифицированными программистами с использованием эффективных и не всегда очевидных алгоритмов.

Настоящее пособие посвящено началам объектно-ориентированного программирования. В нем будут рассмотрены основные конструкции и наиболее важные методики ООП, без понимания которых невозможно ни разрабатывать собственные классы, ни ис-

пользовать стандартные библиотеки классов. К сожалению, красота и сила объектно-ориентированного подхода достаточно наглядно демонстрируются только при рассмотрении относительно сложных (и законченных) задач, изучение которых в рамках такого пособия, как это, просто невозможно. Примеры же, которые мы сможем привести, будут поневоле носить формальный характер и демонстрировать не столько идеологию и преимущества ООП, сколько просто правила написания и использования наиболее употребительных конструкций языка C++ в той его части, которая описывает средства ООП. Все примеры будут представлять собой программы, предназначенные для выполнения в системе Windows; поэтому их изучение требует знакомства с основными видами и принципами реализации приложений Windows.

Понятия класса и объекта

Класс (определение которого начинается с ключевого слова **class**) является естественным развитием понятия *структуры* и отличается от последней тем, что помимо данных включает еще и функции для их обработки. Вспомним основные правила объявления и обслуживания структур на примере простой структуры с именем **Men**, включающей в себя информацию об имени и возрасте человека:

```
struct Men{
  char* name;//Указатель на имя индивидуума
  int age;//Его возраст
  };//Завершающая скобка и точка с запятой
```

Приведенное выше объявление представляет собой лишь придуманное нами описание нового, удобного для нас, типа данных. Для того чтобы включить в программу информацию о конкретных людях, надо объявить в программе требуемое количество структурных данных типа **Men**. Как известно, объявить структурную переменную можно двумя способами – по имени или посредством указателя. В зависимости от способа объявления структурной переменной изменяются и синтаксические правила обращения к ее членам, именно, при объявлении переменной по имени используется оператор точка, а при объявлении посредством указателя – оператор стрелка:

```
Men m1;//Объявляем структурную переменную типа Men с именем m1
m1.name = "Иванов";//Инициализация члена name
m1.age = 19;//Инициализация члена age

Men* pm2;//Объявляем указатель типа Men*
pm2 = new Men//Выделяем память под структурную переменную
pm2->name = "Петров";//Инициализация члена name
pm2->age = 20;//Инициализация члена age
```

Напомним, что имена переменных-указателей часто начинают с буквы p (от pointer, указатель). Такой стиль обозначения удобен, так как наглядно выдсляет в программс указатели среди прочих переменных, хотя и нсобязателен.

Опрсделим тсперь класс **Men** с теми же данными (данные, входящис в состав класса, называют *данными-членами*) и базовым набором функций (*функций-членов*, или *методов*) для инициализации и чтения данных, принадлежащих этому классу. Заметим, что имя класса принято начинать с прописной буквы, хотя это и необязательно. Часто также это имя начинают с буквы C (от Class) или T (от Type).

```
class Men{
private: //Закрытые данные
  char* name;
  int age;
public: //Открытые функции
  void SetName(char* n){name=n;}//Инициализация имени
  void SetAge(int a){age=a;}      //Инициализация возраста
  char* GetName(){return name;} //Чтение имени
  int GetAge(){return age;}       //Чтение возраста
  };//Завершающая скобка и точка с запятой
```

Данные, включснные в описание класса после описателя **private**, являются *закрытыми*. К ним можно обращаться только в функциях, принадлежащих этому же классу. Из программы или из другого класса к ним обратиться нельзя: компилятор сообщит, что данное с таким именем не существует. Закрытие (сокрытие) данных является важной концепцией ООП, способствующей защите данных от несанкционированного доступа. Вопрос о доступе к данным позже будет рассмотрен подробнее.

Функции объявлены и описаны в *открытой* части класса после описателя **public**. Это делает их доступными из программы, т. е.

из главной функции **WinMain()**, а также и из прикладных функций, вызываемых по ходу выполнения программы. В классе обязательно должны быть открытые функции, иначе к элементам класса просто нельзя будет обратиться.

Как и в случае структур, *объявление* класса не создает реальных данных. Это лишь шаблон, описывающий созданный нами тип коллекции данных, в которую входят не только собственно данные, но и функции для их обработки. Создание *объектов*, или *экземпляров* класса осуществляется точно так же, как и создание структурных переменных:

```
/*Создание экземпляра класса по имени*/
Men m1;
m1.SetName("Храмов");
m1.SetAge(54);
char txt[50];//Символьный массив для формирования вывода
wsprintf(txt,"Имя: %s\nВозраст: %d", m1.GetName(),
                                              m1.GetAge());
MessageBox(NULL,txt,"Индивидуум 1",MB_OK);//Вывод в окно
                                  //сообщения данных-членов класса

/*Создание экземпляра класса с помощью указателя*/
Men* pm2=new Men;
pm2->SetName("Хромов");
pm2->SetAge(23);
char txt[50];//Символьный массив для формирования вывода
wsprintf(txt,"Имя: %s\nВозраст: %d", pm2->GetName(),
                                            pm2->GetAge());
MessageBox(NULL,txt,"Индивидуум 2",MB_OK); //Вывод
                              // в окно сообщения данных-членов класса
```

Обратите внимание на то, что перед именем вызываемой функции необходимо указывать объект, для которого она вызывается (имя объекта или указатель на объект). Нельзя выполнить предложение

```
SetName("Громов");
```

так как неизвестно, к какому объекту оно относится (в приведенном выше примере **m1** или **pm2**). Если объект класса создан по имени, то, как и для структур, имя объекта и имя функции разделяются точкой; если обращение к объекту выполняется с помощью указателя на объект, то указатель и имя функции разделяются оператором **->**.

Доступ к членам класса

Существуют три градации уровня доступа к данным и функциям, входящим в класс, для каждой из которых предусмотрен свой описатель.

Данные или функции, описанные после ключевого слова **public**, являются *открытыми* (общедоступными, общими, публичными). Это значит, что к ним можно обращаться откуда угодно: из функций данного класса, из функций других классов, из программы. Обычно открывают функции общего назначения, чтобы с их помощью можно было посылать сообщения объектам.

Данные или функции, описанные после ключевого слова **private**, являются *закрытыми*. Это значит, что к ним можно обращаться только из функций данного класса (как открытых, так и закрытых). Из программы к закрытым данным обратиться нельзя, такое обращение не пропустит компилятор. Обычно наиболее ответственные данные-члены класса закрывают, чтобы к ним можно было обратиться только с помощью специально предусмотренных для этого функций-методов. Иногда закрывают так же и функции, обслуживающие внутренние задачи класса и не предназначенные для вызова извне класса.

Данные или функции, описанные после ключевого слова **protected**, являются *защищенными*. Такие члены класса доступны только данному классу и классам, производным от него. Таким образом, защищенные данные имеют смысл только при наличии иерархии производных классов. О производных классах речь будет идти ниже; отметим только, что закрытые (**private**) члены класса в производных классах недоступны, и если к каким-то данным надо открыть доступ из производного класса, оставив их закрытыми от внешнего мира, их следует объявить защищенными (**protected**).

Ключевые слова **public**, **private** и **protected** могут встречаться в описании класса в любом порядке и в любом количестве. По умолчанию (при отсутствии ключевого слова-спецификатора доступа) данные-члены считаются закрытыми (**private**). Приведенные ниже описания классов вполне равнозначны:

```
class A{
  int x;//Закрытое по умолчанию данное-член
  int y;//То же
```

```
  void func();//Закрытая по умолчанию функция-член
public:
  void SetX(int);//Открытая функция
  void SetY(int;//То же
};
class A{
public:
  void SetX(int);//Открытая функция
  void SetY(int;//То же
private:
  int x;//Закрытое данное
  int y;//То же
  void func();//Закрытая функция
  };
```

Функции встроенные и невстроенные

В приведенном выше примере описания класса **Men** функции-члены класса описывались непосредственно в теле класса, в месте их объявления. Такой способ удобен для коротких функций. Если же текст функции занимает много места, нагляднее вынести его за пределы описания класса. В этом случае в теле класса должен быть обязательно объявлен прототип функции. Соответствующие строки для нашего класса (с единственным данным-членом **name**) будут выглядеть следующим образом:

```
class Men{
  char* name;//Закрытое по умолчанию данное
public:
  void SetName(char* n){name=n;}//Встроенная функция
  char* GetName();//Прототип невстроенной функции
  };//Конец описания класса
char* Men::GetName (){//Определение объявленной ранее
  return name;          //функции-члена
  }
```

Если в описании класса указан только прототип функции, а сама функция определена за пределами описания класса, то необходимо указать, к какому классу она относится. Имя класса в этом случае помещается перед именем функции, отделяясь от него специальным *оператором разрешения области видимости* "::". Обратите внимание на то, что класс, к которому относится функция, указы-

вается непосредственно перед именем функции, *после* описания типа возвращаемого функцией данного.

Расположение в программе фрагмента с определением функции значения не имеет, он может размещаться перед функцией **WinMain()** или после нее, в конце программы (но не внутри другой функции!). Однако описание класса должно предшествовать определению входящих в него функций-членов.

Функции, описанные *вне* описания класса (в который в этом случае помещается лишь прототип функции, см. функцию **GetName()** в последнем примере), компилируются обычным для подпрограмм образом: тело функции включается в состав загрузочного модуля один раз, а все вызовы этой функции в программе преобразуются в команду процессора **call**. Если функция требует аргументов, то перед выполнением оператора **call** значения аргументов загружаются в стек, откуда они будут извлечены в процессе выполнения функции-подпрограммы.

Функции, описанные *внутри* описания класса, носят специальное наименование *встроенных* (inline). Такие функции обрабатываются компилятором иначе: во всех местах программы, содержащих вызов inline-функции, компилятор просто вставляет тело функции. Операторы **call** в этом случае отсутствуют, так же как и строки загрузки аргументов в стек, что благотворно сказывается на быстродействии программы. С другой стороны, тело функции встраивается в загрузочный модуль при каждом ее вызове, что может привести к чрезмерному увеличению размеров программы, если функция вызывается многократно и имеет заметный размер. Очевидно, что в качестве встроенных целесообразно использовать только короткие функции. Впрочем, компилятор анализирует сложность функций, описываемых внутри описания класса, и в некоторых случаях (например, при включении в тело функции оператора цикла **for**) выдает сообщение о невозможности сделать функцию встроенной.

Статические переменные-члены класса

Если различать переменные по областям их видимости, то мы сталкивались до сих пор с переменными двух видов: глобальными и локальными.

Глобальные переменные объявляются *вне* всех функций программы (обычно перед главной функцией **WinMain()**); значения этих переменных видны во всех точках программы, в том числе во всех ее функциях и блоках. Глобальные переменные используются для передачи некоторых значений из одной функции в другую. Например, в типичном приложении Windows инструменты рисования (кисти, перья, шрифты) создаются обычно в процессе создания окна в функции **OnCreate()**, однако используются эти объекты в другой функции – именно, **OnPaint()**. Для того чтобы передать дескриптор **hPen** созданного в функции **OnCreate()** пера в функцию **OnPaint()**, его следует объявить глобальным.

Локальные переменные объявляются внутри той функции, в которой они будут использоваться; ни в какой другой функции (и, в частности, в функции **WinMain()**) эти переменные не видны. Таким образом, локальные переменные используются для хранения временных значений, которые нужны только в процессе выполнения данной функции. Кстати, в отличие от глобальных переменных, которые после инициализации сохраняют свои значения неограниченно долго, локальные переменные при каждом новом вызове функции, в которой они объявляются, теряют свои предыдущие значения, и их следует инициализировать заново.

Какова область видимости членов классов? *Открытые данные* видны, естественно, всюду в программе или, точнее, всюду в той функции, в которой был создан объект класса (если объект класса объявлен и создан в некоторой функции, то он представляет собой локальную переменную, которая при завершении функции уничтожается и, таким образом, не видна нигде кроме этой функции).

У *закрытых* данных-членов класса область видимости у́же, так как к ним можно обратиться только из функций-членов данного класса. При этом каждое данное-член в каждом объекте имеет свое значение. Таким образом, закрытые данные не могут служить для передачи информации между объектами.

Статическая переменная, объявленная в качестве закрытого данного-члена класса, выступает *для объектов этого класса* в качестве глобальной переменной. Все объекты класса имеют к ней доступ, и для всех объектов она, в отличие от обычных данных-членов, имеет одно и то же значение. Происходит это потому, что компилятор, выделяя для обычных данных каждого объекта свою

отдельную область памяти, статическую переменную размещаст в памяти только один раз (рис. 1.1).

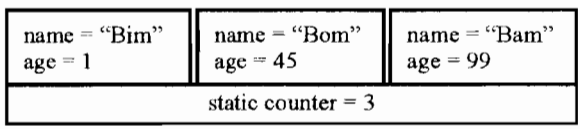

Рис. 1.1 Несколько объектов класса **Men** *со статической переменной* **counter**

Любой объект класса может обращаться к статической переменной, читая или модифицируя ее значение; после такой модификации для всех объектов данного класса статическая переменная будет видна с этим новым значением.

Статическая переменная объявляется в составе закрытых данных класса со спецификатором класса памяти **static** и *обязательно* с указанием начального значения (например, 0):

```
class Men{
  static int counter = 0;
  ...
  };
```

2. КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

Конструкторы

Класс может содержать любое количество функций-членов самого разнообразного назначения, но два типа функций занимают особое положение. Эти функции называются *конструктором* и *деструктором*.

Функция-конструктор формально отличается от всех остальных функций тем, что имеет имя, совпадающее с именем класса. Например, для класса **Men** конструктор должен имеет вид

Men (*параметры*) {*тело конструктора*}

Конструктор может иметь произвольное число параметров или не иметь их совсем, хотя часто в конструкторе предусматривается столько параметров, сколько данных-членов класса необходимо

инициализировать при создании нового объекта. Конструктор не может возвращать никакого значения (даже **void**).

Конструктор вызывается автоматически при создании объекта, и его назначение – выполнение необходимых инициализирующих действий. Прежде всего это, конечно, инициализация данных-членов. Часто в конструкторе выполняется создание и инициализация объектов вспомогательных классов, которые будут использоваться функциями данного класса, однако инициализирующие действия могут и не иметь прямого отношения к классу, например, это может быть открытие файлов с данными или вывод на экран какого-либо сообщения.

Класс может включать несколько конструкторов с различным составом параметров (более того, обычно именно так и бывает), но может не иметь ни одного. Действительно, если ни переменные класса, ни другие элементы системы не требуют инициализации в момент создания объекта, то и в конструкторе нет необходимости; при создании экземпляров класса компилятор автоматически выделит под них память, хотя в этом случае данные-члены класса, естественно, не инициализируются и будут содержать мусор.

Поскольку назначение конструкторов – создание экземпляров класса (объектов), они, как правило, объявляются в открытой части класса, чтобы иметь к ним доступ из программы или из других классов.

Вернемся к нашему примеру с классом **Men** и включим в его состав конструктор, который, как ему и положено, будет служить для инициализации данных-членов класса задаваемыми в программе значениями (именем и возрастом индивидуума):

```
class Men{
  char* name;
  int age;
public:
  Men(char* n,int a){//Конструктор
    name=n;            //с двумя
    age=a;             //параметрами
    }
  char* GetName(){return name;}//Встроенные
  int GetAge(){return age;}     // функции-члены
  };
```

Поскольку инициализация членов класса теперь осуществляется конструктором, мы удалили из класса функции **SetName()** и **SetAge()**, которые выполняли ту же операцию.

Конструктор класса **Men** имеет два параметра и, следовательно, при создании объектов класса необходимо указывать два аргумента. Создать объект класса без указания начальных значений имени и возраста не удастся, так как в данном классе не предусмотрен конструктор без параметров. Его можно было объявить (наряду с конструктором с двумя параметрами); такая методика (перегрузка конструкторов) будет рассмотрена в последующих разделах.

Создание объектов класса Men при наличии конструктора будет выглядеть следующим образом:

```
Men m1("Freston",17);
Men* pm2=new Men("Young",23);
```

Конструктор с инициализацией членов класса по умолчанию

Важнейшей разновидностью конструктора является конструктор с инициализацией членов класса по умолчанию, который позволяет при его вызове *с аргументами* инициализировать данные создаваемого объекта указанными значениями аргументов, а при вызове *без аргументов* – инициализировать их некоторыми определенными заранее значениями по умолчанию. Для нашего примера конструктор с инициализацией по умолчанию может выглядеть следующим образом:

```
Men(char n="***",int a=0){name=n;age=a;}
```

Если такой конструктор вызвать без аргументов, то данные-члены создаваемого объекта будут инициализированы указанными в конструкторе значениями (три звездочки на месте имени и возраст, равный 0). В то же время, вызов конструктора с конкретными значениями аргументов позволит по-прежнему инициализировать объект требуемым образом:

```
Men m1;//name="***", age=0
Men m2("Собакин",41);//name="Собакин", age=41
Men m3("Львов");//name="Львов", age=0 (по умолчанию)
Men m4(32); //Недопустимый вызов
```

Как видно из приведенных примеров, при наличии в конструкторе с инициализацией по умолчанию нескольких параметров (у

нас – двух), допустимо вызывать конструктор вообще без аргументов, со всеми аргументами, или только с начальными аргументами (если бы в конструкторе было четыре параметра, можно было бы вызвать его с указанием первого, двух первых, трех первых или всех четырех аргументов). Последний же вызов недопустим, так как в нем число 32 стоит на месте первого аргумента, который должен быть указателем на символьную строку. Недопустима также конструкция

```
Mem(,32);
```

Необязательно иметь умолчание для всех переменных, инициализируемых с помощью конструктора. Однако при этом все параметры, не подлежащие инициализации, должны располагаться в начале списка параметров, а инициализирусмые параметры – в конце этого списка. Так, можно образовать конструктор

```
Men(char n,int a=0){name=n;age=a;}//По умолчанию задается
                                  //только значение возраста
```

однако недопустима конструкция

```
Men(char n="***",int a){name=n;age=a;}
```

Компилятор, встретив такое предложение, сообщит об ошибке отсутствия умолчания после параметра **a**.

Деструкторы

Деструктор, как и конструктор, имеет имя, совпадающее с именем класса, но перед ним ставится знак "~". Для класса **A** деструктор должен иметь вид

```
~A(){тело деструктора}
```

Деструктор не может иметь ни параметров, ни возвращаемого значения. Он автоматически вызывается при уничтожении объекта класса.

Основное назначение деструктора – выполнение завершающих действий (например, закрытие файлов или уничтожение таймеров). Однако чаще всего деструктор используется для освобождения памяти, динамически выделенной в процессе создания или использования объекта. Если, например, в конструкторе предусмотрено выделение памяти под массив данных с помощью оператора **new** (что

может потребоваться, если размер этого массива заранее неизвестен и передается в конструктор через один из его параметров), то в деструкторе необходимо эту память освободить с помощью оператора **delete[]**:

```
class A{
  char* array;
public:
  A(int size){
    array=new char[size];
    }
  ~A(){
    delete[] array;
    }
  };
```

При таком определении класса **A** объект **a1**, созданный предложением

```
A a1(10);
```

будет содержать в себе символьный массив длиной 10 байтов, а предложение

```
A a2(1000);
```

создаст объект **a2** с массивом длиной 1000 байтов.

При создании объекта класса **A** автоматически вызываемый конструктор выделит память под массив; при уничтожении объекта (оператором **delete** или при завершении программы) автоматически вызываемый деструктор освободит память массива, после чего будет уничтожена оставшаяся часть объекта.

Очень часто в завершающих действиях нет необходимости, и тогда надобность в явном определении деструктора отпадает. Таким образом, деструктор включается в состав класса, только если он нужен; в простых случаях можно обойтись без него.

3. ПЕРЕГРУЗКА

Под перегрузкой понимается использование одного и того же идентификатора для обозначения двух или большего числа различных операций. Перегрузка является одним из видов *полиморфизма*, под которым понимают придание различных значений или свойств одному и тому же обозначению (имени функции, оператору) в за-

висимости от типа обрабатываемых данных. Перегружены могут быть как функции, так и операторы. При перегрузке функций в программе объявляется несколько одноименных функций, выполняющих различающиеся (возможно, даже совершенно различные) действия. Перегрузка операторов, например, знака плюс или знака равенства позволяет придавать этим операторам в различающихся контекстах разный смысл.

Перегрузка не является атрибутом именно объектно-ориентированного программирования и может использоваться и в традиционных программах. Однако для ООП перегрузка является одним из краеугольных камней; многие фундаментальные методики ООП целиком основаны на этом понятии.

Перегрузка функций

C++ позволяет определять в программе произвольное количество функций с одним именем при условии, что все они имеют разный состав параметров (разное количество или разные типы параметров). Состав параметров функции иногда называют ее *сигнатурой*. При вызове перегруженной функции в программе компилятор анализирует ее сигнатуру и выбирает из списка одноименных функций ту, сигнатура которой соответствует вызванной. Например, функции с прототипами

```
void Func(int);
void Func(int,int);
void Func(char*);
```

являются перегруженными; при вызове функции **Func()** с одним целочисленным аргументом будет вызвана первая функция, при вызове функции с тем же именем, но с двумя целочисленными аргументами – вторая, а при указании в качестве аргумента адреса символьной строки – третья. При этом *возвращаемый* функцией тип роли не играет; функции

```
void Func(int);
int Func(int);
```

не являются перегруженными, компилятор их не различает, и, следовательно, такая пара функций не имеет права на существование.

Каким образом компилятор отслеживает в тексте программы перегруженные функции? Для этого он использует *кодирование* или *декорирование* имен функций, которое заключается в том, что

функциям даются новые, дополнительные имена, в которые входят как имя класса, в котором объявлена функция (если функция объявлена в классе), так и список ее параметров. Если, например, приведенные выше три функции **Func()** объявлены в классе **MyClass**, то компилятор даст им следующие имена:

```
void Func(int);          @MyClass@Func$qi
void Func(int,int);      @MyClass@Func$qii
void Func(char*);        @MyClass@Func$qpzc
```

В результате функции, имеющие, с нашей точки зрения, одно и то же имя, для компилятора представляются различными.

Рассмотрим пример перегрузки функций-членов класса, воспользовавшись нашим простым классом **Men**:

```
class Men{
  char* name;
  int age;
public:
  Men(char n,int a){name=n;age=a;}//Конструктор
  void Age(int a){age=a;}//Запись возраста
  int Age(){return age;}//Чтение возраста
  };
```

Теперь и для инициализации, и для чтения переменной **age** мы используем один идентификатор **Age()**, хотя, в зависимости от состава используемых аргументов, он обозначает разные функции:

```
Men* pm = new Men("Орлов",18);//Вызов конструктора
pm->Age(19);// Коррекция возраста
int a = pm->Age();//Чтение возраста
```

Во втором предложении этого фрагмента функция **Age()** вызывается с целочисленным аргументом (типа **int**), и компилятор обращается к функции для записи возраста; в третьем предложении функция **Age()** вызывается без аргумента, и компилятор подставляет в программу вызов функции для чтения возраста.

Перегрузка конструкторов

Перегрузка конструкторов приобретает особое значение из-за того, что конструктору нельзя назначить произвольное имя – оно всегда должно совпадать с именем класса. В то же время в классе, как правило, приходится определять несколько конструкторов с

разными сигнатурами. Наличие нескольких функций-конструкторов, имеющих одно имя, но разный состав параметров, возможно благодаря реализации в языке средства перегрузки функций.

Рассмотрим пример класса с перегруженными конструкторами. Пусть класс с именем **Rect** описывает прямоугольник, для которого можно задавать координаты левого верхнего (*x1* и *y1*) и правого нижнего (*x2* и *y2*) углов. Таким классом можно воспользоваться в приложении, формирующем на экране блок-схему программы или процесса. Координаты можно задавать парами целых чисел, однако в некоторых случаях может оказаться удобнее использовать структурные переменные типа **POINT**. Для реализации этой возможности в класс следует включить перегруженные конструкторы:

```
class Rect{
int x1,y1,x2,y2;//Координаты
public:
  Rect(int a,int b,int c,int d){//4 аргумента типа int
    x1=a; y1=b; x2=c; y2=d;
    }
  Rect(POINT pt1,POINT pt2){//2 аргумента типа POINT
    x1=pt1.x; y1=pt1.y; x2=pt2.x; y2=pt2.y;
    }
  };
```

Первый конструктор требует четыре целочисленных аргумента; при вызове второго конструктора необходимо указать две структурных переменных типа **POINT**:

```
Rect r1(20,10,220,110);//x1=20, y1=10, x2=220, y2=110
POINT p1={250,50};
POINT p2={255,55};
Rect r2(p1,p2);//x1=250, y1=50, x2=255, y2=55
```

Довольно часто встречается ситуация, когда при создании объекта неизвестны значения его данных и, соответственно, нельзя воспользоваться конструктором-инициализатором. Например, некоторый объект должен быть заполнен данными с диска; в этом случае необходимо сначала создать "пустой" объект, а затем прочитать в него значения данных, записанные на диске. В таких случаях в составе класса среди прочих конструкторов предусматривают перегруженный "конструктор по умолчанию", который выделяет память под объект, но больше ничего не делает:

```
class Men{
char name[20];//Символьная строка для имени
int age;//Возраст
public:
  Men(char*n,int a){          //Обычный конструктор-
    strcpy(name,n); age=a;}//инициализатор с параметрами
  Men(){}//Конструктор по умолчанию, создающий "пустой" объект
  void SetName(char*n) {name=n;}//Инициализирующая функция
  void SetAge(int a) {age=a;}    //Инициализирующая функция
  };
```

В прежних примерах класса Men для имени индивидуума предусматривалась переменная-указатель типа **char***, в которую можно было легко записать адрес строки с фактическим именем простым оператором присваивания:

```
m.name="Showman";
```

Однако такой объект нельзя сохранить на диске. Записать-то его на диск можно, но храниться на диске будет не само имя, а лишь адрес строки с именем. При последующем чтении объекта с диска (программой, предназначенной для чтения объектов и их обработки) в программу попадет адрес, который в этой программе будет указывать на случайную ячейку памяти, в которой, естественно, никакого имени не будет. Поэтому в настоящем примере для имени предусмотрена пустая символьная строка достаточной длины (20 байтов), в которую имя придется переносить операцией копирования (см. вызов функции **strcpy()** в конструкторе с параметрами). Теперь на диск будет записываться сама строка с именем.

Заметьте, что при наличии в классе конструктора по умолчанию должна быть предусмотрена возможность инициализировать данные уже после создания объекта. В приведенном выше примере для этого в класс включены функции **SetName()** и **SetAge()**. Теперь создавать объекты класса можно двояко:

```
Men m1("Громыко",25);//Объект создается и сразу же
                     //инициализируется конструктором-инициализатором
Men m2;//Пустой объект (создается конструктором по умолчанию)
m2.SetName("Крымов");//Инициализация пустого объекта
m2.SetAge(38);//вызовом соответствующих функций
```

Следует заметить, что реальные классы (как прикладные, так библиотечные) практически всегда содержат несколько, а иногда и много (до 10 – 15) перегруженных конструкторов.

Перегрузка операторов

Перегрузку операторов приходится применять в тех случаях, когда смысл привычных для нас операций, таких как сложение, умножение или присваивание, оказывается для объектов конкретного класса отличным от общепринятого. Например, сложение комплексных чисел заключается в почленном сложении как действительных, так и мнимых составляющих чисел-слагаемых. Однако обычный (глобальный) оператор "**+**" умеет складывать только два числа, а не две пары чисел. Таким образом, для класса комплексных чисел этому оператору следует придать другой смысл или, лучше сказать, другую реализацию, для чего и используется перегрузка. Часто перегрузка изменяет реализацию оператора, но его интуитивный смысл остается без изменения (как в приведенном примере с оператором "+"); в других случаях перегрузка совершенно изменяет назначение оператора. Например, глобальные операторы языка C++ **<<** и **>>** служат для сдвига операнда влево или вправо на заданное число битов; однако при использовании с потоковыми объектами эти операторы приобретают смысл "вставить в поток" и "извлечь из потока" (см. раздел "Потоки ввода-вывода").

Перегрузить можно почти любой оператор языка (**=**, **+**, *****, **^**, **<**, **<<** и т. д.); как видно из этого перечня, перегружаются как одноместные операторы, применяемые к одному объекту (например, **++** или **!**), так и двуместные, требующие двух операндов (операторы сложения или умножения, логических операций и другие).

Перегрузка операторов осуществляется с помощью специальной своеобразной функции, имеющей имя **operator** (ее называют *операторной функцией*). Рассмотрим ее применение на примере перегрузки нескольких операторов для класса **Point3D**, описывающего координаты точки в трехмерном пространстве.

Будем считать, что равенство двух точек обозначает равенство их соответствующих координат, а сложение заключается в образовании точки, у которой каждая координата вычисляется как сумма соответствующих координат точек-операндов. В простейшем случае класс **Point3D** выглядит следующим образом:

```
class Point3D{
int x,y,z;//Данные-члены (координаты точки)
public:
  Point3D(int xx,int yy,int zz){//Конструктор-инициализатор
    x=xx; y=yy; z=zz; }
```

```
Point3D(){};//Конструктор для создания "пустого" объекта
bool operator == (Point3D&);//Прототип функции перегрузки ==
Point3D operator+ (Point3D&);//Прототип функции перегрузки +
Point3D operator= (Point3D&);//Прототип функции перегрузки =
};
```

Как видно из приведенного фрагмента, все функции **operator** принимают в качестве аргумента объект класса; при этом функции перегрузки операторов + и =, которые должны образовывать новые объекты, возвращают объекты класса, а функция перегрузки оператора сравнения возвращает булев результат сравнения.

Замстим, что обычно аргумент перегружаемого оператора передается в него по ссылке (как в нашем примере). Передача по ссылке уменьшает объем копирования данных при преобразовании аргумента в локальный параметр функции и поэтому предпочтительнее. Однако передать аргумент можно и по значению:

```
bool operator == (Point3D);//Аргумент передается по значению
```

Определение функции **operator** для операции сравнения на равенство выглядит следующим образом:

```
bool Point3D::operator == (Point3D obj&){
  if((x==obj.x)&&(y==obj.y)&&(z==obj.z)) return true;
  else return false;
  }
```

Для выяснения того, как работает эта функция, рассмотрим сначала строки программы, в которых используется оператор сравнения:

```
Point3D p1(5,10,15), p2(5,10,15);//Два равных объекта
if(p1==p2)...
```

Оператор == сравнивает объекты **p1** и **p2** (в данном случае равные). Как эти два объекта поступают в функцию **operator**?

Общее правило здесь таково: оператор == (или любой другой двуместный оператор) вызывается для *первого* операнда. *Второй* операнд поступает в функцию в качестве ее аргумента. Другими словами, функция **operator** вызывается таким образом:

```
p1.operator == (p2);
```

Обозначение операторной функции, вызываемой для объекта **p1**

Аргумент операторной функции (объект **p2**)

Функция **operator==** является членом нашего класса и ей, естественно, доступны все данные объекта **p1**, для которого она вызывается (конкретно **x**, **y** и **z**). Объект **p2** поступает в операторную функцию в качестве аргумента и внутри нее принимает имя **obj**. Таким образом, **x**, **y** и **z** в теле операторной функции являются данными объекта **p1**, а обозначения **obj.x**, **obj.y** и **obj.z** соответствуют данным **x**, **y** и **z** объекта **p2**.

Из текста рассматриваемой функции видно, что если значения соответствующих данных-членов двух объектов-операндов совпадают, функция возвращает **true**; в противном случае – **false**.

Рассмотрим теперь перегрузку оператора **+**. Функция **operator +** для нашего класса будет иметь следующий вид:

```
Point3D Point3D::operator + (Point3D obj){
  Point3D temp;//Вспомогательный временный объект
  temp.x=x+obj.x;//Образуем во временном
  temp.y=y+obj.y;//объекте temp суммы членов
  temp.z=z+obj.z;//объектов-операндов
  return temp;//Вернем объект temp с суммами членов
  }
```

Сложение объектов класса **Point3D**:

```
Point3D p3 = p1 + p2;
```

Как и в предыдущем случае, функция **operator+** вызывается для первого операнда-слагаемого **p1**, и обозначения **x**, **y** и **z** в операторной функции относятся к членам объекта **p1**; второй операнд **p2** поступает в операторную функцию в качестве аргумента и принимает в ней имя **obj**.

Прежде всего в функции создается временный объект нашего класса (**temp**). Каждое данное-член этого временного объекта принимает значение суммы соответствующих данных первого и второго операндов, после чего объект **temp** используется в качестве возвращаемого значения.

Третий перегруженный оператор, оператор присваивания, как и оператор сравнения, не требует временного объекта **temp**:

```
Point3D Point3D::operator = (Point3D obj){
  x=obj.x;
  y=obj.y;
  z=obj.z;
```

```
return *this;
}
```

Данным-членам первого операнда (стоящего до знака равенства) присваиваются значения членов второго операнда (поступающего в функцию через параметр **obj**). Вообще говоря, на этом операция присваивания завершается, и данная функция могла бы не возвращать никакого значения. Однако возврат текущего объекта (***this**), в котором уже образовалась копия второго операнда, нужен для того, чтобы объекты класса могли участвовать в операциях множественного присваивания:

```
Point3D p(10,20,30);//Объект со значениями
Point3D p1,p2;//"Пустые" пока объекты p1 и p2
p1 = p2 = p;//И p1, и p2 приравниваются объекту p
```

Здесь объекту **p1** присваивается значение выражения, стоящего справа от него, т. е. выражения **(p2 = p)**. Чтобы эта операция присваивания могла выполняться, надо не только придать объекту **p2** значение объекта **p**, но и сделать так, чтобы выражение **p2 = p** образовывало объект. Другими словами, оператор присваивания должен возвращать объект – результат выполненной им операции присваивания.

Поясним смысл выражения ***this**. Каждый раз, когда вызывается функция-член класса, она автоматически получает указатель с именем **this** на объект, для которого она вызвана. Указатель **this** является неявным параметром всех функций-членов классов, и его можно использовать в любой функции. Например, в операторной функции перегрузки оператора сложения предложение

```
temp.x=x+obj.x;
```

можно было бы записать таким образом:

```
temp.x=this->x+obj.x;
```

хотя в этом и нет необходимости. Однако функция перегрузки оператора присваивания, как уже отмечалось, должна возвращать объект, для которого она вызывается. Указатель на этот объект поступает в функцию в качестве неявного параметра и принимает имя **this**. Для того чтобы, имея указатель, получить сам объект, надо снять ссылку с указателя, что и выполняется операцией ***this**.

4. ПРОИЗВОДНЫЕ КЛАССЫ И НАСЛЕДОВАНИЕ

Наследование относится к числу наиболее фундаментальных концепций ООП. Суть этого понятия заключается в том, что, имея разработанные заранее классы, пользователь может создавать своеобразные копии этих классов – *производные классы*, к которым переходят (*наследуются*) все возможности родительских, базовых классов. При этом пользователь может произвольно изменять свойства создаваемых производных классов, добавляя в них новые функции или замещая имеющиеся.

Таким образом, наследование является основным механизмом, позволяющим использовать созданные заранее классы или целые библиотеки классов при разработке новых программных продуктов. Более того, коммерческие библиотеки классов и сами фактически строятся на базе этого механизма, представляя собой разветвленные иерархические структуры, в которых из каждого базового класса, обеспечивающего наиболее общие черты поведения всей иерархии, порождаются многочисленные потомки, каждый из которых, с одной стороны, использует общие для всей иерархии возможности родительского класса, а с другой – вносит свои специфические детали, обеспечивая функционирование конкретных объектов.

Объявление производного класса

Рассмотрим сначала формальную сторону вопроса об объявлении производного класса и его возможностях. Воспользуемся в качестве базового класса вариантом нашего класса **Men**:

```
class Men{
  char* name;
  int age;
public:
  void Set(char* n, int a){name=n; age=a;}
  };
```

Функция **Set()** позволяет присваивать членам класса требуемые значения, т. е. фактически выполняет задачи конструктора и, кроме того, позволяет по ходу программы модифицировать значения данных, образующих объект класса.

Образуем от класса **Men** производный класс **MenEx**, введя в него новое данное-член **wages** для хранения зарплаты. Для того чтобы сделать класс **MenEx** производным от **Men**, используется следующая конструкция:

```
class MenEx:public Men{//Класс Men указан как базовый для MenEx
...//Описание класса MenEx
  };
```

Ключевое слово **public** перед именем базового класса объявляет все *открытые и защищенные* члены базового класса такими же (открытыми и защищенными) для производного класса, и, следовательно, доступными в нем. Однако *закрытые* члены базового класса недоступны производному. Так, в нашем случае данные **name** и **age**, хотя и будут входить в состав объектов класса **MenEx**, недоступны в них для прямого обращения. Доступ к закрытым данным базового класса осуществляется только через открытые функции базового класса, в нашем примере через функцию **Set()**.

В целом определение класса **MenEx** может выглядеть следующим образом:

```
class MenEx:public Men{
  int wages;
public:
  void Set(char*n,int a,int w){//Заместим функцию Men::Set()
    wages=w;
    Men::Set(n,a);//Вызовем исходную (замещенную) функцию Set()
    };
```

Открытая функция **Set()** класса **Men** доступна в классе **MenEx**, однако с ее помощью можно придать значения только данным **name** и **age**, а нам надо еще инициализировать переменную **wages**. Потому функцию **Set()** придется *заместить*, т. е. описать в производном классе функцию с тем же именем, но с другим списком параметров и другим содержимым.

Функция **Set()** класса **MenEx** будет принимать три аргумента по числу данных в обоих классах. В переменную **wages** легко передать значение одного из аргументов (последнего в нашем примере):

```
wages=w;
```

Однако к переменным **name** и **age** обратиться напрямую нельзя. Для их инициализации придется вызывать функцию **Set()** базового класса:

```
Men::Set(n,a);
```

Спецификатор области видимости **Men::** позволяет указать, что вызывается функция именно класса **Men**, а не класса **MenEx**.

Замещение функций базовых классов используется в ООП чрезвычайно широко. Действительно, создавая производный класс, нетрудно включить в него *дополнительные* функции-члены для обработки данных производного класса, а также и открытых или защищенных данных базового. Однако во многих случаях нас может не устраивать реализация тех или иных функций базового класса. Замещая функции базового класса, мы можем придать им новые функциональные возможности.

Для замещения какой-либо функции базового класса нужно в производном классе объявить функцию с таким же именем. Новая функция может иметь тот же список параметров и возвращаемое значение, что и замещаемая, однако это не обязательно. Главное, чтобы у них совпадали имена. Замещение приводит к тому, что исходная функция, продолжая существовать, становится невидимой. Однако, как было показано выше, ее все же можно вызвать, если использовать оператор разрешения видимости "::".

Состав производного класса

Как уже отмечалось, производный класс наследует все содержимое базового, хотя и с некоторыми ограничениями (закрытые члены базового класса непосредственно недоступны в производном). Таким образом, *производный класс шире базового* (рис. 4.1).

Как создается такой составной объект? При создании объекта производного класса сначала автоматически вызывается конструктор базового класса, конструирующий ту часть объекта, которая переходит в производный класс из базового. После этого вызывается конструктор производного класса, дополняющий объект составляющими производного класса. В нашем примитивном примере в классах нет конструкторов, однако принято считать, что в таком случае компилятор создает конструкторы по умолчанию, которые, разумеется, ничего не инициализируют, а просто выделяют

память под создаваемые в программе объекты. В реальных классах конструкторы всегда присутствуют, и полезно понимать, в каком порядке они вызываются.

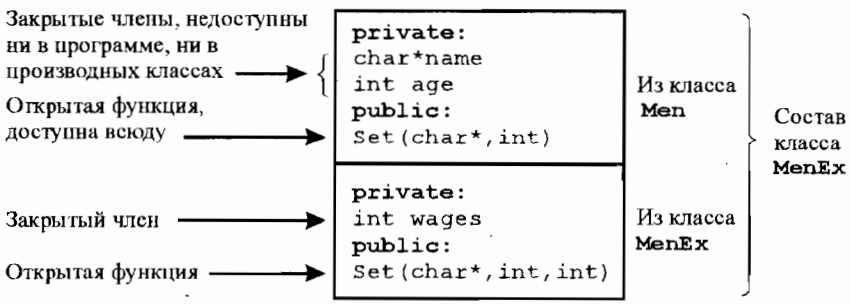

Рис. 4.1. Состав объекта производного класса

Точно так же при отсутствии в классах явно описанных деструкторов, они создаются компилятором по умолчанию. При уничтожении объекта производного класса сначала вызывается деструктор производного класса, уничтожающий более специализированную часть объекта, затем деструктор базового класса, уничтожающий общую часть.

Защищенные члены класса

В последнем примере данные-члены обоих классов объявлены (по умолчанию) закрытыми, как это обычно и делается. Однако к закрытым данным базового класса нельзя обратиться из производного. Решить эту проблему можно двумя способами: предусмотреть в базовом классе открытые функции для доступа к данным или объявить данные базового класса *защищенными* (**protected**). Защищенные данные по-прежнему закрыты для программы, которая не может их случайно разрушить, однако из функций производного класса к ним доступ есть. Модифицируем последний пример, объявив данные базового класса защищенными и введя в производный класс функцию **Print()** для их распечатки:

```
class Men{
protected:
  char* name; int age;
```

```
public:
  void Set(char* n, int a){name=n; age=a;}
  };
class MenEx:public Men{
  int wages;
public:
  void Set(char*n,int a,int w){//Заместим функцию Men::Set()
    name=n; age=a; wages=w;
    }
  void Print(){
    char txt[50];
    wsprintf(txt,"Имя: %s\nВозраст: %d\nЗарплата: %d",
          name, age, wages);
    MessageBox(NULL,txt,"MenEx",MB_OK);
    }
  };
```

Поскольку защищенные данные базового класса непосредственно доступны в производном, отпала необходимость вызова функции **Set()** базового класса в замещающей функции **Set()** класса **MenEx**; в ней можно инициализировать данные обоих классов. В новой функции **Print()** с помощью вызова **wsprintf()** формируется символьная строка с содержимым объекта, выводимая затем в окно сообщения. Ниже приведен фрагмент программы, в которой создается, инициализируется и выводится на экран объект класса **MenEx** (в предположении, что оба класса описаны ранее, например, в подсоединенном к программе заголовочном файле).

```
MenEx* pm=new MenEx();
pm->Set("Хромов",22,5000);
pm->Print();
```

Вывод программы показан на рис. 4.2.

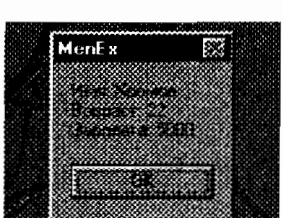

Рис. 4.2. Вывод в окно сообщения содержимого класса

Конструкторы и производные классы

Обычно при создании объекта производного класса требуется инициализировать данные не только производного, но и базового классов. Для этого в конструкторе производного класса надо явно вызвать конструктор базового класса и передать ему необходимые параметры. Рассмотрим этот вопрос на примере классов **Men** и **MenEx**, из которых уберем функции **Set()** и **Print()** и добавим конструкторы-инициализаторы:

```
class Men{
  char* name;
  int age;
public:
  Men(char* n, int a){name=n; age=a;}
  };
class MenEx:public Men{
  int wages;
public:
  MenEx(char* n,int a,int w):Men(n,a){
    wages=w;
    }
  };
```

В конструкторе производного класса перечисляются в качестве параметров все переменные как производного, так и базового классов. В соответствии с правилами написания заголовка функции все параметры предваряются их типами. После двоеточия указывается имя конструктора базового класса с перечнем *его* параметров. Здесь типы переменных уже не указываются. В теле конструктора производного класса осуществляется обычная инициализация *его* данных. При вызове конструктора производного класса (с целью создания комбинированного объекта, включающего члены обоих классов) необходимо, в соответствии с его описанием, передать ему три аргумента:

```
MenEx m("Краснов", 44, 12000);
```

Первые два аргумента (**"Краснов"** и **44**) преобразуются в параметры **n** и **a** и будут переданы конструктору класса **Men**, в котором они инициализируют данные-члены класса **name** и **age**. Третий аргумент (**12000**) поступит в параметр **w** и будет использован

в конструкторе производного класса для присвоения значения переменной **wages**, входящей в класс **MenEx**.

Как известно, параметры любых функций в языке C++ позиционные. Вызывая функцию, мы должны передавать ей фактические аргументы в том порядке, в каком они указаны в ее определении. Поэтому и в конструкторе класса **MenEx** параметры класса **Men** следует перечислять в том порядке, в каком они указаны в конструкторе (уже имеющемся) класса **Men**. Таким образом, при разработке производного класса необходимо знать, как выглядит прототип конструктора базового класса.

Мы рассмотрели формат *встроенного* конструктора производного класса, когда и вызов базового конструктора, и тело производного включаются в его объявление. Для *невстроенного* конструктора вызов конструктора базового класса осуществляется не в прототипе, а в определении конструктора производного класса:

```
MenEx(char*,int,int);//Прототип конструктора
MenEx(char* n,int a,int w):Men(n,a){wages=w;}//Определение
```

5. ВИРТУАЛЬНЫЕ ФУНКЦИИ

Понятие виртуальной функции

В разделе 3 мы уже столкнулись с понятием полиморфизма, когда описывали перегруженные функции, т. е. функции с одним именем, но с различающимися списками параметров. Настоящий раздел будет посвящен другой форме полиморфизма, реализуемой в *виртуальных* функциях. Виртуальные функции тоже имеют одно имя, но различаются они не по сигнатурам, как в случае перегруженных функций, а по тому, в каком именно классе (из некоторой иерархии классов) они определены. Виртуальные функции замечательны тем, что выбор требуемой из множества виртуальных функций с одним именем осуществляется не во время компиляции программы, а динамически, в процессе выполнения программы, по конкретному значению указателя, посредством которого эта функция вызывается.

Виртуальная функция объявляется в базовом классе с помощью ключевого слова **virtual** и затем обязательно замещается по

крайней мере в одном производном классе. При этом не только имя, но и типы параметров, а также тип возвращаемого значения для замещаемых функций должны быть такими же, как для "исходной" функции, объявленной в базовом классе. В производных классах объявления этих замещенных функций могут для наглядности содержать описатель **virtual**, но это не обязательно; если функция объявлена виртуальной в базовом классе, то все замещающие ее функции в производных классах автоматически становятся виртуальными.

Виртуальные функции проявляют свойство виртуальности только в том случае, если они вызываются через указатель или ссылку на *базовый* класс. Как будет показано ниже, указатель на базовый класс может принимать конкретное значение указателя на производный класс; если этот указатель к моменту вызова функции фактически указывает на объект базового класса, вызывается вариант функции из базового класса; если же указатель фактически указывает на некоторый объект производного класса, вызывается вариант функции из этого производного класса.

Обслуживание множества объектов класса

Во многих случаях класс описывает физический объект, существующий в одном экземпляре. Например, класс может быть предназначен для управления устройством ввода изображения с микроскопа, с помощью которого анализируются гистологические или цитологические препараты. Программа управления установкой создает один экземпляр этого класса и, вызывая те или иные его функции, осуществляет ввод изображения, его анализ и отображение на экране.

Однако чаще класс предназначен для описания множества однородных объектов: тех же гистологических срезов или томографических проекций, медицинских карт или историй болезней пациентов, анкет сотрудников предприятия и т. д. Как в этих случаях осуществляется обработка объектов?

Рассмотрим в качестве примера класс **Image**, описывающий набор полученных с помощью микроскопа и телекамеры увеличенных изображений медицинских препаратов – мазков крови группы пациентов. Пусть эти изображения хранятся не в формате битовых

матриц, содержащем кроме собственно изображения еще и дополнительную управляющую информацию, а в виде однородных числовых массивов, заполненных кодами цветов последовательных точек изображения. Такой формат упрощает математическую обработку данных, хотя и несколько усложняет их вывод на экран. В случае 24-битовых цветных изображений каждый цветной пиксель будет описываться в файле тройкой байтов, содержащих интенсивности красного, зеленого и синего компонентов цвета данной точки экрана (рис. 5.1).

Рис. 5.1. Структура числового файла с цветным изображением

Для обращения к файлу с данными удобнее всего воспользоваться методом отображения файла на память, для чего в составе класса следует предусмотреть указатель на проекцию файла в памяти. Индексируя этот указатель, программа сможет прочитать любой байт файла.

Используя данные из файла, программа должна сформировать из них цветное изображение препарата; для этого в состав объекта должен входить небольшой блок совместимой памяти, размер которого соответствует размеру изображения препарата. В класс следует включить указатель на эту память, а также указатель на ее контекст устройства.

Для сравнения изображений препаратов друг с другом и визуального анализа результатов их обработки желательно выводить на экран сразу группу изображений (разумеется, в программе могут быть предусмотрены также и средства их вывода по отдельности в увеличенном масштабе). Этот общий кадр можно сформировать в большом глобальном блоке памяти (размером с окно приложения), который, после переноса в него отдельных изображений в требуемом порядке, будет копироваться в окно приложения.

Поскольку каждый создаваемый объект класса привязан к определенному файлу с изображением препарата, в составе класса должна иметься переменная, в которую будет передаваться имя этого файла. Наконец, в состав класса полезно включить целочисленную переменную для хранения номера изображения; эта переменная позволит управлять расположением отдельных изображений на экране.

Такой класс имеет минимальные функциональные возможности, и его разумно использовать в качестве базового в программном комплексе, обеспечивающем обработку изображений:

```
class Image{
protected:
  BYTE* ptr;//Указатель на проекцию файла в памяти
  HBITMAP hBmp;//Совместимая память для хранения изображения
  HDCP hdcMem;//Совместимый контекст этой памяти
  char* name;//Имя файла с данными
  int nmb;//Номер загруженного изображения
public:
  Image(int nmb,char* name);//Конструктор класса
  void View();//Функция вывода изображения
  };
```

В открытую секцию класса включены объявления конструктора класса и функции **View()**, формирующей изображение конкретного объекта.

В программе объявляется массив из требуемого числа указателей на объекты типа **Image**:

```
Image* pImg[5];
```

Далее в цикле из 5 шагов создаются объекты класса **Image**; конструктору класса передаются в качестве аргументов номер изображения и имя содержащего его файла (предполагается, что в программе имеется массив **char* names[]** имен этих файлов):

```
for(int i=0;i<5;i++)
  Img[i]=new Image(i,names[i]);
```

Конструктор класса, автоматически вызываемый при создании очередного объекта, выполняет отображение указанного файла с данными на память и создает блок совместимой памяти объекта.

Функция **View()**, входящая в состав класса **Image**, выводит цветное изображение, хранящееся в объекте, в совместимую па-

мять объекта и копирует затем эту память в соответствующее место совместимой памяти окна (в данном примере изображения препаратов имеют размер 100 × 100 пикселей):

```
void Image::View(){
  for(int i=0;i<100;i++){//Цикл по столбцам изображения
    for(int j=0;j<300;j+=3){//Цикл по строкам изображения
      BYTE byte0=ptr[i*300+j];//Получим красный компонент
      BYTE byte1=ptr[i*300+j+1];//Получим зеленый компонент
      BYTE byte2=ptr[i*300+j+2];//Получм синий компонент
//Выведем цветную точку в совместимую память объекта
      SetPixel(hdcMem,j/3,i,RGB(byte0,byte1,byte2));
      }
    }
//Скопируем память объекта в память кадра
  BitBlt(hdcBigMem,5+nmb*105,5,100,100,hdcMem,0,0,SRCCOPY);
  SetBkMode(hdcBigMem,TRANSPARENT);//Прозрачный фон символов
//Выведем под изображением имя файла
  TextOut(hdcBigMem,5+nmb*105,108,name,strlen(name));
  }
```

Нетрудно далее организовать (например, посредством меню) вызов функции **View()** в цикле для всех объектов **Image**:

```
  for(int i=0;i<5;i++){//Цикл по всем объектам
    pImg[i]->View();//Вызов View() для каждого объекта
    }
  }
```

Результат работы программы показан на рис. 5.2 (в действительности на экран выводятся цветные изображения).

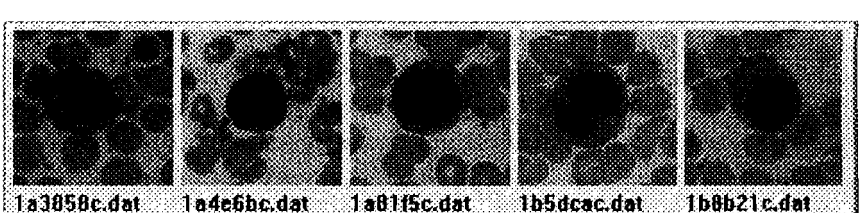

Рис. 5.2. Исходные изображения 5 препаратов

Рассмотрим теперь случай, когда от базового класса **Image** образован производный класс с именем, например, **ImageGray**,

предназначенный для обработки изображений препаратов, имеющих лишь несколько градаций серого цвета. Такие "упрощенные" исходные данные могут быть полезны при отладке программ цифровой обработки изображений. Заметим кстати, что файл с цветными изображениями при нашем методе кодирования будет содержать 30000 байтов (100 × 100 пикселей по три байта на пиксель), а файл с серым изображением будет в три раза короче, так как каждый пиксель будет описываться одним байтом.

Включим в класс конструктор и замещающую функцию **View()**:

```
ImageGray:public Image{
public:
  ImageGray (int n,cahr* f):Image(n,f){}
  void View();
  };
```

Новая функция **View()** будет отличаться от аналогичной функции базового класса алгоритмом заполнения совместимой памяти объекта; кроме того, она будет копировать блоки памяти объектов в память кадра на 130 пикселей ниже, чтобы можно было одновременно наблюдать изображения обоих классов:

```
void ImageGray::View(){
  for(int i=0;i<100;i++){//Цикл по столбцам изображения
    for(int j=0;j<100;j++){//Цикл по строкам изображения
      BYTE byte=ptr[i*100+j];//Получим значение серого
//Выведем точку серого цвета в совместимую память объекта
      SetPixel(hdcMem,j,i,RGB(byte,byte,byte));
      }
    }
//Скопируем память объекта в память кадра
  BitBlt(hdcBigMem,5+nmb*105,5+130,100,100, hdcMem,
        0,0,SRCCOPY);
  SetBkMode(hdcBigMem,TRANSPARENT);//Прозрачный фон символов
//Выведем под изображениями имена файлов
  TextOut(hdcBigMem,5+nmb*105,108+130,name,strlen(name));
  }
```

Предположим теперь, что у нас имеются пять файлов с исходными полноцветными изображениями, из которых с помощью отдельной программы образованы пять файлов с серыми изображениями. Программа, использующая класс **ImageGray**, должна объ-

явить пять указателей на базовый класс и столько же указателей на производный:

```
Image* pImg[5];
ImageGray* pImgGray[5];
```

после чего создать все 10 объектов:

```
for(int i=0;i<5;i++)
   pImg[i] = new Image(i,names[i]);
for(int i=0;i<5;i++)
   pImgGray[i] = new ImageGray(i,namesGray[i]);
```

Очевидно, что любые циклические действия с массивами изображений придется осуществлять *отдельно* для массивов двух классов:

```
for(int i=0;i<5;i++)
   pImg[i]->View();
for(int i=0;i<5;i++)
   pImgGray[i]->View();
```

Результат выполнения этих двух циклов приведен на рис. 5.3.

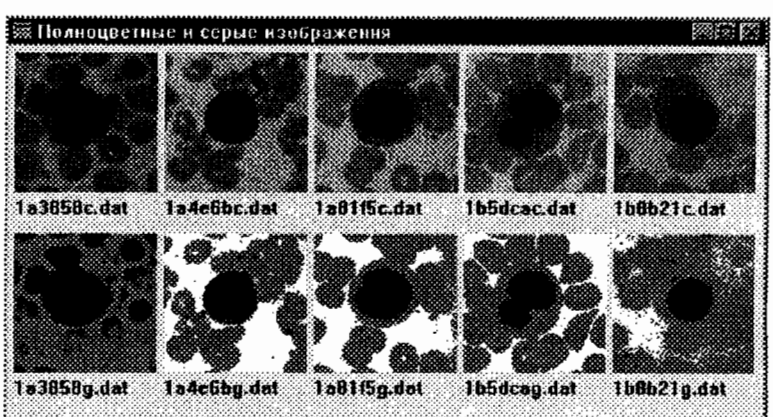

Рис. 5.3. Исходные полноцветные изображения и полученные из них изображения с несколькими градациями серого цвета

Если производных классов несколько, как это часто бывает на практике, то для каждого класса придется предусматривать отдельный цикл. Использование виртуальных функций, как это будет по-

казано в следующем подразделе, позволяет обрабатывать объекты базового и производных классов в одном цикле, если обращаться к ним с помощью указателя на базовый класс.

Использование виртуальных функций

Воспользуемся примером предыдущего раздела и объявим функцию **View()** базового класса **Image** виртуальной:

```
virtual void View();
```

Такое объявление сделает эту функцию виртуальной и во всех производных классах.

Объявим теперь *один* массив указателей на *базовый* класс, увеличив, естественно, число элементов в нем:

```
Image* pImg[10];
```

Создадим требуемое количество объектов-изображений обоих классов посредством объявленных нами указателей на базовый класс:

```
for(int i=0;i<5;i++)//Создаем объекты класса Image
  pImg[i]=new Image(i,names[i]);
for(int i=5;i<10;i++)//Создаем объекты класса ImageGray
  pImg[i]=new ImageGray(i,names[i]);
```

Порядок создания объектов не имеет значения; можно было, например, создать нечетные объекты базового класса, а четные – производного.

Теперь объекты обоих классов можно вывести на экран в одном цикле:

```
for(int i=0;i<10;i++)
  pImg[i]->View();
```

Функция **View()** вызывается посредством указателей на базовый класс **pImg[i]**. Поскольку она виртуальная, в процессе выполнения происходит автоматический выбор варианта этой функции: если некоторому указателю **pImg[i]** присвоен адрес объекта базового класса, вызывается вариант **View()** базового класса; если же указателю **pImg[i]** присвоен адрес объекта производного класса, вызывается вариант **View()** производного класса.

Абстрактные базовые классы

В приведенном выше примере базовый класс **Image** вполне мог использоваться самостоятельно (без производного класса), и, соответственно, его функция **View()** была не только объявлена, но и определена. Производный класс расширял возможности базового добавлением новых функций и, в нашем случае, замещением функции **View()** с изменением ее алгоритма (рис. 5.4).

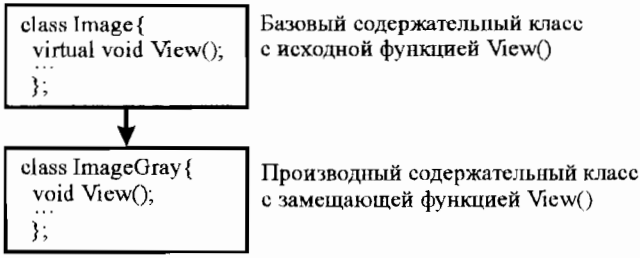

Рис. 5.4. Иерархическая структура из двух содержательных классов

Однако часто базовый класс создается лишь как шаблон для выведения из него (возможно многих) производных классов. Такой базовый класс определяет состав функций, которые должны быть реализованы в производных классах, но сам не предоставляет реализаций этих функций. Другими словами, в базовом классе объявлены лишь прототипы функций, но нет их определений.

Разумеется, в каждом производном классе, образованном от такого базового, все функции должны быть определены, иначе нельзя будет создать объект производного класса. Для таких случаев мы должны иметь какой-то способ, который позволит нам убедиться, что производный класс действительно переопределил все необходимые функции. Решение этой проблемы в C++ заключается в использовании чисто виртуальных функций.

Чисто виртуальная функция – это виртуальная функция, объявленная в базовом классе, но не имеющая в этом классе своего определения. Определяются эти функции (обязательно) в производных классах. Для объявления чисто виртуальной функции ее прототип в базовом классе приравнивается нулю:

```
virtual void View() = 0;
```

Класс, в котором имеется хотя бы одна чисто виртуальная функция, называется *абстрактным классом*. Для такого класса нельзя создать объект. Поэтому абстрактные классы применяются в таких иерархических структурах, где содержательными являются только производные классы; а базовый класс содержит лишь объявления виртуальных функций (рис. 5.5).

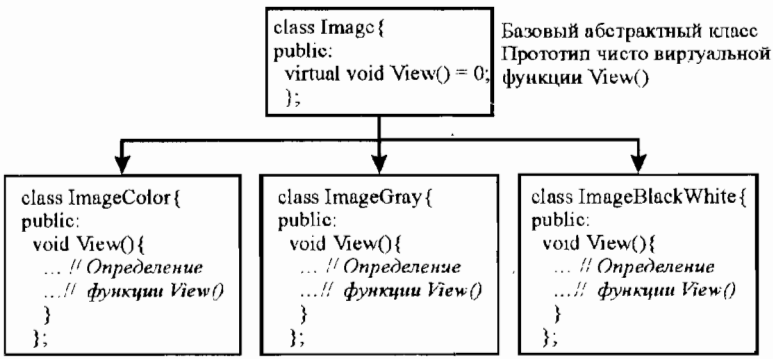

Рис. 5.5. Абстрактный базовый класс и производные от него содержательные классы

Применительно к рассмотренному ранее примеру при такой организации иерархии классов класс **Image** сам по себе никакой работы выполнять не может; класс **ImageColor** предназначен для обработки цветных изображений, класс **ImageGray** – для обработки серых, а класс **ImageBlackWhite** – черно-белых. Пример использования класса **ImageBlackWhite** показан на рис. 5.6.

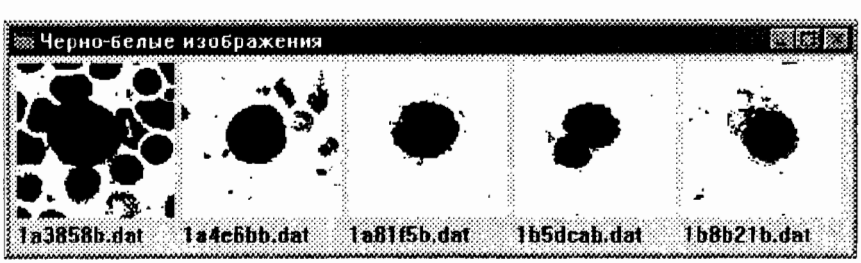

Рис. 5.6. Черно-белые изображения тех же препаратов

6. ПОТОКИ ВВОДА-ВЫВОДА

Иерархия потоковых классов ввода-вывода в C++

Для операций с файлами можно использовать предусмотренные для этого функции API **CreateFile()**, **ReadFile()**, **WriteFile()** и др. Однако в языке C++ предусмотрена и другая возможность более обобщенного подхода к передаче данных, которая реализована в объектно-ориентированной библиотеке ввода-вывода, использующей базовое понятие *потока*. Поток является абстрактным объектом, который либо создает, либо поглощает данные. Система ввода-вывода C++ позволяет связать поток с конкретным устройством, после чего операции с потоком приводят к физической передаче байтов данных из конкретного устройства или на него. При этом все функции и операторы ввода-вывода в принципе можно использовать со всеми типами устройств: клавиатурой, экраном, принтером, файлами и пр. Последнее, впрочем, справедливо лишь для текстовой операционной системы типа MS-DOS или для консольных приложений Windows; в обычных оконных приложениях Windows ввод с клавиатуры и вывод на экран осуществляется более сложными методами, а потоки используются главным образом при работе с файлами.

Библиотека ввода-вывода содержит весьма сложную и разветвленную систему классов, реализующих различные режимы и возможности ввода-вывода. На рис. 6.1 приведена часть иерархии классов ввода-вывода, имеющая отношение к интересующему нас вопросу потоковых операций над файлами.

Класс **ios** служит базовым для всех потоковых классов, предоставляя лишь самые общие данные и функции обслуживания любых потоков; классы **istream** и **ostream** обеспечивают базовые средства обслуживания потоков ввода и вывода соответственно, а класс **iostream** поддерживает потоки, через которые можно выполнять обе операции – и ввода, и вывода. Реализация функций, обеспечивающих работу *файловых* потоков, содержится в производных классах **ifstream**, **ofstream** и **fstream**, первый из которых предназначен для операций ввода из файлов, второй – для операций вывода в файлы, а третий позволяет через один поток

выполнять и ввод, и вывод. Именно эти классы используются в прикладных программах для создания потоковых объектов, обеспечивающих интерфейс программ с файлами на дисках.

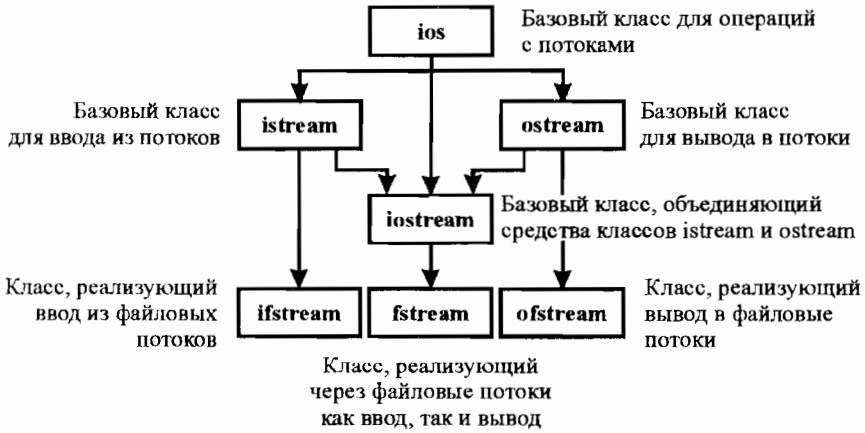

Рис. 6.1. Часть иерархии классов потоков ввода-вывода

Базовые операции с файловыми потоками

Наиболее общими операциями при работе с файлами являются операции открытия или создания файла, чтения или записи, установки указателя и закрытия файла. Все эти операции поддерживаются потоковыми классами. Для создания потока и связи его с конкретным создаваемым или открываемым файлом можно использовать конструктор соответствующего класса, в качестве аргументов которого указывается имя открываемого файла и, при необходимости, режим открытия, определяемый с помощью констант, описанных в классе **ios**. К наиболее употребительным относятся следующие режимы открытия:

ios::binary – файл открывается в двоичном режиме. В этом случае все байты данных пересылаются в файл или из файла без всяких изменений. Следует иметь в виду, что по умолчанию все файлы открываются в текстовом режиме, в котором могут иметь место неявные преобразования байтов. Например, после кода 13 (который при выводе текстовых строк на экран или принтер обозначает возврат каретки) автоматически вставляется код 10 (пере-

вод строки); могут происходить и потери байтов. Разумеется, такая ситуация при работе с числовыми данными недопустима, поэтому файлы с расчетными или экспериментальным данными всегда следует открывать в двоичном режиме.

ios::app – весь вывод дописывается в конец файла. Этот режим удобен в том случае, когда полученными из какого-либо источника данными следует дополнить существующий файл.

ios::ate – при открытии файла указатель в нем устанавливается, как и в режиме **ios::app**, на конец файла, однако далее вызовом функции **seekp()** указатель можно переместить на требуемую позицию и осуществить запись или чтение, начиная с любого байта файла (режим **ios::app** не позволяет произвольно перемещать указатель).

ios::trunc – открываемый файл усекается до нулевой длины, что приводит к потере всех содержащихся в нем данных (если указанный файл не существует, он создается заново). Для потоков вывода этот режим действует по умолчанию.

При необходимости можно указывать сразу несколько режимов, объединяя соответствующие константы с помощью операции побитового ИЛИ (|).

Приведем несколько примеров создания файловых потоков.

```
ofstream outpt1 ("a.dat",ios::binary);//Создается поток
      //outpt1 для вывода в файл с именем A.DAT в двоичном режиме;
      //существующий файл с таким именем затирается

ofstream outpt2 ("b.dat",ios::binary|ios::ate);//Создается
      //поток outpt2 для вывода в конец существующего файла B.DAT
      //в двоичном режиме с возможностью последующей установки
      //указателя на любой байт файла

ifstream inpt ("a.dat",ios::binary);//Создается поток inpt
                                    //для ввода из файла A.DAT в двоичном режиме
```

Для записи в файл используется функция **write()** базового класса **ostream**, для чтения – функция **read()** базового класса **istream**. Как и всегда при работе с файлами, каждая операция записи или чтения перемещает указатель в файле в конец соответствующей порции данных. Таким образом, последовательные вызовы, например, функции **write()** будут дополнять файл новыми данными.

Функции записи и чтения требуют в качестве первого аргумента адрес записываемых данных, приведенный к типу **char***, а в качестве второго – число записываемых байтов:

```
int mas[100];//Массив данных (100 целых чисел по 4 байта)
short d;//Переменная с данным
ofstream strmw ("1.dat",ios::binary);//Создаем поток
                    //strmw для вывода в файл 1.DAT в двоичном режиме
strmw.write((char*)mas,400)//Записываем в 1.DAT 400 байтов
strmw.write((char*)&d,2)//Дописываем в 1.DAT еще 2 байта из d

int masin[100];//Массив для приема данных типа int
short din;//Переменная для приема данного типа short
ifstream strmr ("1.dat",ios::binary);//Создаем поток strmr
                    //для ввода из файла 1.DAT в двоичном режиме
strmr.read((char*)masin,400);//Вводим первые 400 байтов в массив
strmr.read((char*)&din,2);//Вводим еще2 байта в din
```

Установка указателя в файле осуществляется с помощью функций **seekp()** (при записи) или **seekg()** (при чтении):

```
int dd[2];//Массив для приема части данных
ifstream strm ("1.dat",ios::binary);//Создаем поток strm для
                    //ввода из файла 1.DAT в двоичном режиме
strm.seekg (100);//Устанавливаем указатель на 26-е данное
                    //(пропустив 25 * 4 = 100 байтов)
strm.read((char*)dd,8);//Вводим в массив dd 26-е и 27-е данные
```

После окончания работы с потоком и файлом их следует закрыть функцией **close()**:

```
strm.close();//Закрывается поток strm, связанный с файлом 1.DAT
```

Описанными здесь функциями и режимами не исчерпывается обширный аппарат потоковых классов для работы с файлами.

Перегрузка операторов вставки и извлечения

Выполняя запись и чтение с помощью потоков ввода-вывода, удобно вместо относительно громоздких предложений с потоковыми функциями **write()** и **read()** использовать операторы вставки в поток (**<<**) и извлечения из потока (**>>**). В языке C++ эти операторы служат для побитового сдвига данных влево или вправо на заданное число двоичных разрядов. Однако в потоковых классах **ostream** и **istream** (а также и в производных от них) они пере-

гружены и обозначают совсем другие операции: пересылку указанного данного в поток (оператор **<<**) и, наоборот, получение из потока очередного данного с сохранением его в указанной переменной (оператор **<<**). Операторы **<<** и **>>** всегда указываются справа от переменной, обозначающей объект потока:

```
streamout << data1;//Вывод данного из переменной data1 в поток
                   //вывода streamout
streamin >> data2; //Извлечение данного из потока ввода streamin
                   //в переменную data2
```

Большим достоинством операторов вставки и извлечения является возможность их объединения в цепочки:

```
streamout << data1 << data2 << data3;//Вывод в поток
                   //streamout последовательно трех данных
```

Операторы **<<** и **>>** широко использовались в приложениях MS-DOS и все еще используются в консольных приложениях Windows, однако их непосредственное применение в современных прикладных программах затруднительно, так как они осуществляют автоматическое преобразование числовых данных в символьные строки и наоборот. Так, например, фрагмент

```
int data = 125;
stream << data
```

выполнит запись в файл, связанный с потоком **stream**, не четырех байтов, содержащих число 125 в двоичной форме, а трех байтов с кодами 0x31, 0x32 и 0x35, представляющих собой символьные представления десятичных цифр 1, 2 и 5.

Для того чтобы с помощью операторов вставки и извлечения осуществлять запись и чтение числовых данных в машинном формате (что обычно и требуется в прикладных программах обработки результатов экспериментов или моделирования), необходимо перегрузить эти операторы применительно к составу прикладного класса. Сущность перегрузки заключается в том, что в операторную функцию, перегружающую соответствующий оператор, включается необходимое число вызовов потоковых функций **read()** или **write()**, каждый из которых обеспечивает передачу порции однородных данных.

Рассмотрим пример перегрузки операторов вставки и извлечения применительно к простому классу с именем **Point2D**, в состав

которого входят два целых числа (представляющие собой, например, координаты точки на экране).

```
/*Файл POINT.H*/
class Point2D{
public:
  int x,y;
  Point2D(int xx,int yy){//Конструктор-инициализатор
    x=xx;y=yy;
    }
  Point2D(){}//Конструктор по умолчанию (без параметров)
//Включим в класс прототипы операторных функций для перегрузки
  friend ofstream& operator << (ofstream&, Point2D&);
  friend ifstream& operator >> (ifstream&, Point2D&);
  };

/*Файл POINT.CPP*/
#include <windows.h>
#include <fstream.h>//Для работы с потоками ввода-вывода
//Определим операторную функцию для перегрузки оператора <<
ofstream& operator << (ofstream& strm, Point2D& obj){
  strm.write((char*)&obj.x,4);
  strm.write((char*)&obj.y,4);
  return strm;
  }
//Определим операторную функцию для перегрузки оператора >>
ifstream& operator >> (ifstream& strm, Point2D& obj){
  strm.read((char*)&obj.x,4);
  strm.read((char*)&obj.y,4);
  return strm;
  }
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  Point2D pt1(100,200);//Создадим экземпляр класса Point2D
  Point2D pt2;//Создадим пустой объект для чтения в него
  ofstream outpt("test.dat",ios::binary);//Создадим поток
                          //вывода, связанный с файлом TEST.DAT
  outpt<<pt1;//Вывод объекта pt1 в файл TEST.DAT
  outpt.close();//Закроем поток и файл
  ifstream inpt("test.dat",ios::binary); //Создадим поток
                          //ввода, связанный с файлом TEST.DAT
  inpt>>pt2; //Чтение данных из файла TEST.DAT в объект pt2
  inpt.close();//Закроем поток и файл
  char txt[200];
  wsprintf(txt,"%d  %d",pt2.x,pt2.y);//Преобразуем в строку
```

```
MessageBox(NULL,txt,"Из файла",MB_OK);//и выведем в окно
return 0;
}
```

Прикладной класс **Point2D** определен, как это обычно и делается, в заголовочном файле **POINT.H**. Данные-члены класса **x** и **y** для простоты помещены в открытую секцию, чтобы к ним можно было обратиться из программы (в функции **wsprintf()**). В состав класса, кроме данных, входят два перегруженных конструктора (конструктор-инициализатор и конструктор по умолчанию), а также прототипы операторных функций **operator<<()** и **operator>>()**, с помощью которых мы перегрузим операторы вставки и извлечения.

Из прототипа функции перегрузки оператора вставки в поток

```
friend ofstream& operator << (ofstream&, Point2D&);
```

видно, что она принимает в качестве параметров ссылку на объект класса **ofstream**, в который она будет пересылать данные, а также ссылку на объект прикладного класса (**Point2D** в данном случае), который будет пересылаться в поток. Напомним, что при использовании перегруженного оператора

```
outpt << pt1;
```

оператор **<<** вызывается для объекта, стоящего слева от него, а объект, стоящий справа, поступает в операторную функцию в качестве аргумента. Оба параметра операторная функция должна получать по ссылке (обозначения **ofstream&** и **Point2D&**).

Для того чтобы оператор **<<** мог использоваться в цепочечных операциях, функция возвращает ссылку на объект потока **ofstream**.

Описатель **friend** определяет, что наша функция является *дружественной* для класса, в котором она описана, т. е. для класса **Point2D**. Дружественные функции замечательны тем, что, сами не будучи членами класса, они имеют доступ к закрытым членам класса. Правда, в нашем классе **Point2D** все члены открыты, однако сокрытие данных является важной концепцией объектно-ориентированного программирования, и вынужденное открытие данных является серьезным нарушением его принципов.

Но зачем нужен описатель **friend**? Почему нельзя объявить перегруженный оператор просто членом класса? Ведь в этом случае он также будет иметь доступ к закрытым членам класса. Оказывается, перегруженные операторы **>>** и **<<** не могут быть членами класса. Это объясняется тем, что у любого перегруженного оператора-члена класса левым операндом должен быть объект этого класса. Изменить это требование нельзя. В нашем же случае левым операндом оператора **<<** является поток (**outpt** в приведенном выше примере), а объект класса, для которого осуществляется операция вывода, поступает в операторную функцию через правый параметр. Такой оператор не может быть членом класса, однако его можно объявить другом класса. В этом случае отмеченное выше ограничение на тип левого операнда отменяется, и, в то же время, оператор-друг класса имеет доступ ко всем его закрытым членам.

Все сказанное относится и к оператору извлечения из потока **>>**.

Операторные функции определены в файле POINT.CPP. Еще раз можно отметить, что хотя прототипы этих функций объявлены в составе класса **Point2D**, сами они не являются членами этого класса (в заголовке каждой функции перед ее именем не указана принадлежность к классу **Point2D::**).

Каждая операторная функция включает в себя последовательность вызовов функций **read()** или **write()**, вызываемых для левого операнда (потока) с передачей им по очереди всех членов класса, которые требуется поместить в поток или извлечь из него. В нашем примере это переменные **x** и **y**, принадлежащие объекту, являющемуся правым операндом перегружаемых операторов (и, соответственно, вторым аргументом операторных функций). Особенности использования функций **read()** и **write()** были описаны выше.

В главной функции **WinMain()** создаются два объекта класса **Point2D** (один – инициализированный для записи его на диск, а другой – "пустой" для чтения в него данных с диска), открывается поток, связанный с файлом POINT.DAT, и одним предложением

```
outpt << pt1;//Вывод объекта pt1 в файл TEST.DAT
```

осуществляется вывод в файл всех данных, входящих в объект. Далее файл закрывается, открывается снова для чтения, и предложением

```
inpt >> pt2; //Чтение данных из файла TEST.DAT в объект pt2
```

все записанные на диск данные читаются в объект **pt2**. В заключительных предложениях прочитанные данные преобразуются в символьную форму и выводятся в окно сообщения (рис. 6.2).

Рис. 6.2. Контрольный вывод в окно сообщения данных из файла

7. "ЖИВУЧЕСТЬ" ОБЪЕКТОВ

Проблемы хранения объектов на диске

Вернемся к классу **Men**, с которого мы начали изучение ООП, и рассмотрим возможности работы с его данными:

```
class Men{
  char* name;
  int age;
  };
```

Легко сообразить, что хотя с объектами этого класса можно выполнять любые требуемые преобразования в памяти, однако их нельзя записать на диск. Действительно, после создания объекта и его инициализации переменная **name** будет содержать адрес символьной строки с именем индивидуума; сама строка, введенная в программу, например, с клавиатуры, будет находится где-то в памяти. Если записать такой объект на диск, а затем прочитать его (другой программой), в переменную **name** поступит тот же адрес, но в памяти по этому адресу уже никакого имени не будет, и фактически переменная **name** будет указывать на случайное место в памяти.

Для того, чтобы символьная строка входила в состав объекта, переменную **name** можно объявить как символьный массив:

```
char name[20];
```

Однако с символьными массивами, содержащими фамилии людей, работать неудобно. Размер массива придется определять с за-

пасом, и в большей части объектов в массивах окажется много неиспользуемого места; инициализацию такой переменной нельзя выполнить простой операцией присваивания

```
name="Переплетчиков";
```

(имени массива **name** нельзя присвоить никакого значения!). Строку придется копировать:

```
strcpy(name,"Переплетчиков");
```

Таким образом, при необходимости сохранения на диске объектов, содержащих символьные строки, возникают определенные сложности. Эти сложности можно преодолеть, используя библиотечный класс **string**, который содержит целый ряд средств для работы с символьными строками и, в частности, позволяет повысить мобильность объектов, содержащих строки текста.

Библиотечный класс string

Класс **string** (его определение содержится в заголовочном файле CSTRING.H) предназначен для хранения символьных строк и выполнения с ними разнообразных операций. В класс входят 15 конструкторов, около пяти десятков функций-членов и большое количество перегруженных операторов, обеспечивающих практически все действия, которые приходится выполнять над символьными строками. Приведем несколько примеров.

Конструкторы

string() – конструктор по умолчанию; создает объект со строкой нулевой длины (в некоторых случаях требуется сначала создать "пустой" объект, и лишь затем заполнить его данными, например, с помощью операций копирования или чтения с диска).

string(char* str) – создание объекта класса **string** из символьной строки **str**. Другими словами, этот конструктор преобразует обычную символьную строку в объект класса **string**.

string (char c) – создание объекта класса **string**, содержащего единственный символ **c**.

Функции-члены

char* c_str() – функция возвращает символьную строку, содержащуюся в объекте класса **string**. Фактически эта функция

выполняет операцию преобразования объекта класса **string** в обычную символьную строку.

unsigned int length() – функция возвращает длину строки, содержащейся в объекте класса **string**.

string& append(string& s) – функция сцепляет объект-аргумент класса **string** с объектом (тоже класса **string**), для которого вызывается эта функция.

Перегруженные операторы

Перегруженный оператор присваивания "=" позволяет назначить объекту класса **string** значение другого объекта того же класса.

Перегруженный оператор сложения "+" выполняет операцию сцепления (конкатенации) строк, хранящихся в виде объектов класса **string**.

Группа перегруженных операторов "**==**", "**!=**", "**<=**", "**>=**", "**<**", "**>**" позволяет выполнять операции сравнения символьных строк, хранящихся в виде объектов класса **string**.

Рассмотрим несколько примеров использования объектов класса **string**.

Вывод в окно приложения обычной символьной строки (типа **char***) выполняется следующим образом:

```
char txt[]="Программа №15";
TextOut(hdc,10,10,txt,strlen(txt));
```

При использовании класса **string** та же операция будет выглядеть так:

```
string txt("Программа №15");//Создание объекта класса string
TextOut(hdc,10,10,txt.c_str(),txt.length());
```

Функция Windows **TextOut()** требует в качестве четвертого аргумента адрес символьной строки, а в качестве пятого – ее длину. Имея объект **txt** класса **string** (который мы в этом примере создали с помощью второго из приведенных выше конструкторов класса **string**), адрес символьной строки мы получаем с помощью функции-члена **c_str()**, а длину строки – с помощью функции-члена **length()**.

При использовании обычных символьных строк образование составной строки требует таких операций:

```
char s1[]="Имя: ";//1-я исходная строка
char s2[]="Петров";//2-я исходная строка
char s3[100];//Строка-приемник достаточной длины
strcpy(s3,s1);//Копируем в приемник строку s1
strcat(s3,s2);//Подцепляем строку s2
```

При использовании класса **string** тот же пример будет выглядеть заметно изящнее:

```
string s1("Имя: ");//1-я исходная строка, она же приемник
string s2("Петров");//2-я исходная строка
s1.append(s2);//Операция сцепления строк, в s1 - результат
```

В последнем предложении мы вызываем функцию **append()** для объекта **s1**, указывая в качестве аргумента функции подцепляемую строку. Обратите внимание на эффективность класса **string**: объекты этого класса могут изменять свою длину, и отпадает необходимость предусматривать строки-приемники с длиной, заведомо большей, чем это будет нужно в действительности.

Вернемся к нашему классу **Men**. Заменим указатель на символьную строку объектом типа **string**:

```
#include <cstring>
class Men{
  string name;//Данное-член – объект класса string
  int age;
public:
  Men(string n, int a){
    name=n; age=a;
    }
  };
```

При таком варианте описания класса конструктор класса должен иметь аргумент типа **string**, и это необходимо учитывать при создании экземпляров класса:

```
Men m1(string("Курбатова"), 18);
Men* pm2=new(string("Richter"),55);
```

Теперь уже нельзя создать объект, указав в качестве аргумента конструктора символьную строку:

```
Men m1("Курбатова"), 18);
```

Необходимо преобразовать символьную строку в объект класса **string**, что выполняется вызовом конструктора класса **string**:

```
string("Курбатова")
```

"Живучие" объекты

"Живучими" (устойчивыми, перманентными, персистентными) называются такие объекты, которые хранятся на диске, могут загружаться в память и в случае модификации автоматически записываются на диск в новом, модифицированном варианте, так что при следующем считывании с диска мы получаем последний вариант объекта.

Рассмотрим вопрос о придании объектам свойства "живучести" на основе нашего простого класса **Men**. Приведем сначала полный текст программы, в которой создаются и записываются на диск три объекта класса **Men**.

```
/*Файл PERSIST.H*/
#include <classlib\objstrm.h>//Для операций с потоками
class Men:public TStreamableBase{
  string name;
  int age;
public:
  char txt[200];//Для формирования строки с содержимым объекта
  Men(string,int);//Прототип конструктора-инициализитора
  void Print();//Формирует строку с содержимым объектом
  DECLARE_STREAMABLE(,Men,1);//Класс Men объявляется потоковым
   };
/*Файл PERSIST.CPP*/
#include <windows.h>
#include "pers.h"
IMPLEMENT_STREAMABLE(Men);
Men::Men(string n,int a){//Конструктор-инициализатор
  name=n; age=a;
  }
void Men::Print(){//Формирование строки с объектом
  wsprintf (txt,"Имя: %s\nВозраст: %#x\n\n",name.c_str(),age);
  }
/*Функция-член Read из вложенного класса Streamer*/
LPVOID Men::Streamer::Read(ipstream& val, uint32) const{
   val >> GetObject()->name>>GetObject()->age;
   return GetObject();
   }
/*Функция-член Write из вложенного класса Streamer*/.
void Men::Streamer::Write(opstream& val) const{
   val << GetObject()->name<<GetObject()->age;
   }
```

```
/*Глобальные переменные*/
char report[600];//Для формирования полного отчета
Men* pMen[3];//Массив из трех указателей на класс Men
/*Главная функция WinMain()*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  ZeroMemory(report,strlen(report));//Обнулим report
  pMen[0]=new Men(string("White"),0x11);//Создадим и
  pMen[1]=new Men(string("Gray"),0x22);//инициализируем
  pMen[2]=new Men(string("Black"),0x33);//объекты
  for(int i=0;i<3;i++)//В цикле
    pMen[i]->Print();//формируем строки с данными объектов
    strcat(report,pMen[i]->txt);//и сцепляем их в report
    }
  MessageBox(NULL,report,"Men",MB_OK);//Выводим в окно
  ofpstream obj("1.dat",ios::binary);//Создаем объект obj
                  //класса ofpstream, связанный с файлом 1.DAT
  for(int i=0;i<3;i++)//В цикле отправляем в obj
    obj<<pMen[i];       //последовательно объекты Men
  obj.close();//Закрываем поток и файл
  return 0;//Завершение WinMain()
  }
```

Содержательная часть класса **Men** состоит из двух данных-членов **name** и **age**, обычного инициализирующего конструктора **Men()** и функции-члена **Print()** для контрольного вывода содержимого создаваемых объектов в окно сообщения. Кроме этого в состав класса включен пустой символьный массив **txt** для формирования строки сообщения и макрос **DECLARE_STREAMABLE**, о котором речь будет идти позже.

Дисковые операции с "живучими" объектами осуществляются с помощью технологии *потоковых* (streamable) классов, реализуемой средствами библиотеки классов ввода-вывода C++. Для того чтобы сделать прикладной класс потоковым, его следует объявить производным от базового потокового класса **TStreamableBase**:

```
class Men:public TStreamableBase{...}
```

Для упрощения процедуры создания потоковых объектов в библиотеке ввода-вывода C++ предусмотрен ряд макросов, из которых важнейшими являются два: **DECLARE_STREAMABLE** и **IMPLEMENT_STREAMABLE**. Включение их в соответствующие места программы добавляют к нашему прикладному классу все необходимые для функционирования потоковых объектов функции.

Макрос **DECLARE_STREAMABLE** включается непосредственно в состав создаваемого потокового класса. Он принимает три аргумента: первый аргумент нужен при работе с DLL-библиотеками и в данном случае не используется, вторым является имя нашего класса, а третий представляет собой номер версии объектов. Внутри этого макроса содержатся спецификаторы доступа (**public** и **private**), поэтому его лучше размещать в самом конце определения класса.

Макрос **IMPLEMENT_STREAMABLE** включается в текст программы (до функции **WinMain()**). В качестве аргумента ему передается имя нашего класса.

Макрос **DECLARE_STREAMABLE** добавляет в наш класс пару операторов вставки в поток **<<** и пару операторов извлечения из потока **>>**. Из каждой пары один оператор предназначен для работы с именами объектов, а второй – с указателями на них. Как уже отмечалось выше, в языке C++ эти операторы обозначают побитовый сдвиг на заданное число битов; однако будучи перегружены в классах **opstream** (класс, организующий запись потоковых объектов) и **ipstream** (класс чтения потоковых объектов), они выполняют совсем другие действия – помещение в поток и извлечение из него объектов прикладных классов.

Классы **opstream** и **ipstream** входят в иерархию потоковых классов, обеспечивающих операции с файлами (рис. 7.1).

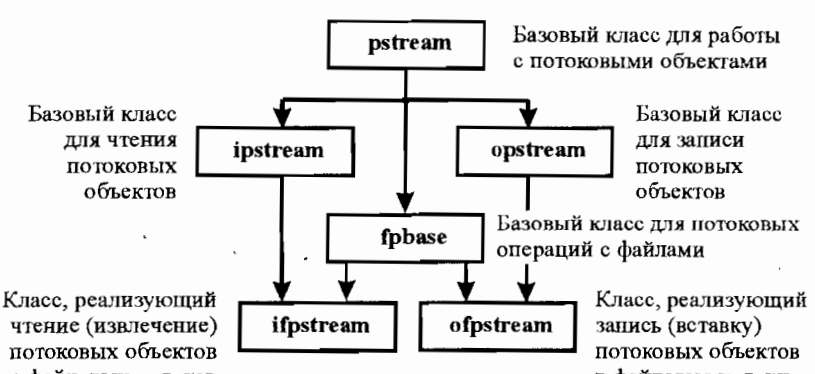

Рис. 7.1. Часть иерархии потоковых классов для работы в файлами

Для выполнения потоковых операций с некоторым файлом программа должна создать объекты потоков вывода и ввода, связанные с этим файлом:

```
ofpstream outstrm("Myfile.dat,ios::binary");//Объект вывода
                                        //в файл в двоичном режиме
ifpstream instrm("Myfile.dat,ios::binary ");// Объект ввода
                                        //из файла в двоичном режиме
```

Поскольку операторы вставки в поток и извлечения из потока перегружены для классов **opstream** и **ipstream**, они наследуются производными от них классами **ofpstream** и **ifpstream** и, таким образом, могут быть применены к нашим объектам **outstrm** и **instrm**:

```
Men m("Row",10);
outstrm << m;//В поток outstrm передается объект
             // нашего класса с указанием имени объекта
Men* pm = new Men("Dow",12);
outstrm << pm;//Тот же синтаксис (!), хотя теперь объект
              //передается в поток посредством указателя
```

То же для операторов извлечения из потока:

```
Men m;//Создается пустой объект нашего класса по имени
instrm >> m;//Из потока instrm объект передается в переменную m
Men* pm=new Men;//Создается пустой объект посредством указателя
instrm >> pm;//Из потока объект передается через указатель pm
```

В качестве аргументов операторов **<<** и **>>** используются имена или указатели объектов прикладного класса. Однако в объекте прикладного класса может быть сколько угодно данных, и совсем не обязательно мы захотим записывать в файл все эти данные. Кроме того, неясно, *в каком порядке* будут записываться данные в файл, а этот порядок важен для последующего правильного чтения данных из файла. Все эти вопросы решаются следующим образом.

Функции перегрузки операторов **<<** и **>>**, определенные в классах **opstream** и **ipstream**, включают в себя вызов целого ряда служебных функций (которые, в частности, записывают в файл и читают из него, кроме собственно данных, еще и служебную информацию – префикс и суффикс записи), которые в итоге вызывают функции **Write()** и **Read()** класса **Streamer**. В этих функциях, содержимое которых в известной степени определяется про-

граммистом, как раз и определяется, какие именно данные из объекта нашего прикладного класса и в каком порядке будут записываться в файл и затем читаться из него. Каким образом функции **Streamer::Write()** и **Streamer::Read()** включаются в нашу программу?

Выше указывалось, что перегруженные операторы вставки и извлечения добавляются в наш прикладной класс макросом **DECLARE_STREAMABLE**. Второе важное действие, выполняемое этим макросом, состоит в том, что он включает в наш класс вложенный класс с именем **Streamer**. Этот класс наследует от своего базового класса **TStreamer** две упомянутые выше чисто виртуальные функции **Read()** для чтения из потока и **Write()** для записи в поток с такими прототипами:

```
virtual void* Read( ipstream&, uint32 ) = 0;
virtual void Write( opstream&) = 0;
```

Поскольку эти функции объявлены в классе **Streamer** как чисто виртуальные, их *необходимо* определить в нашем классе, даже если мы, например, собираемся только записывать в файл, но не читать из него (или наоборот). Хотя функции **Read()** и **Write()** формально являются прикладными, их определение подчиняется жестким правилам. Функция записи в поток **Write()** определяется для рассматриваемого примера следующим образом:

```
void Men::Streamer::Write(opstream& outstrm) const{
   outstrm << GetObject()->name << GetObject()->age;
   }
```

Здесь **outstrm** – произвольное имя потока, который связан с конкретным файлом, и через который данные будут записываться в этот файл. **GetObject()** – функция класса **Streamer**, возвращающая адрес объекта нашего класса, который направляется в поток **outstrm**, а **name** и **age** – имена членов-данных этого объекта. Поскольку возвращаемым значением функции **GetObject()** является указатель на объект, конструкция

```
GetObject()->data
```

представляет конкретное данное-член **data** этого объекта.

Однако как действует в данном случае оператор **<<**? Выше было показано, что если в качестве аргумента этого оператора (т. е. пе-

ременной, стоящей *справа* от него) используется объект прикладного класса, то компилятор подставляет перегруженный оператор класса **opstream**. В функции же **Write()** аргументом оператора вставки выступают не объекты, а конкретные данные-члены самых разных типов (в нашем примере – **string** и **int**). Как известно, при наличии нескольких перегруженных функций компилятор определяет, какую именно функцию следует использовать, по списку аргументов ее вызова. Таким образом, операторная функция вставки с аргументом-объектом и такая же функция с аргументом-данным представляют собой совершенно различные функции, которые могут выполнять различающиеся действия.

В самом деле, в состав класса **opstream** включено большое количество перегруженных операторов вставки **<<**. Один из них, требующий в качестве аргумента имя объекта или указатель на объект, действует так, как было описано выше – вызывает целый ряд служебных функций и, в конце концов, функцию **Streamer::Write()**. Остальные перегруженные операторы вставки, по одному на каждый тип данных (**char**, **unsigned char**, **int**, **unsigned int**, **short**, **float**, **string** и т. д.), просто записывают в поток соответствующее данное. В функции **Write()** используются как раз эти "элементарные" (хотя и перегруженные!) операторы вставки.

Необходимо иметь в виду, что порядок перечисления членов нашего класса (**name** и **age** в данном случае), в принципе произвольный, определяет порядок записи данных в файл. В дальнейшем при чтении из файла данные следует принимать в программу в том же порядке.

В приведенном примере в файл направляются два члена класса **name** и **age**. Если в классе данных-членов больше, то цепочка операторов вставки в поток **<<** вместе с вызовами функции **GetObject()** становится длиннее:

```
outstrm<<GetObject()->d1<<GetObject()->d2<<GetObject()->d3...;
```

Здесь **d1**, **d2**, **d3** и т. д. – имена членов класса, выводимых в поток.

Функция чтения из потока **Read()** определяется чуть сложнее:

```
LPVOID Men::Streamer::Read(ipstream& instrm, uint32) const{
   instrm >> GetObject()->name >> GetObject()->age;
```

```
return GetObject();
}
```

Здесь обозначения имеют тот же смысл, только поток **instrm** направляет данные из файла, с которым он связан, в указанные данные объекта класса. Все приведенные выше рассуждения относительно перегруженных операторов вставки справедливы и для перегруженных операторов извлечения **>>**. Если в качестве аргумента этого оператора выступает объект, автоматически вызывается функция **Streamer::Read()**; в самой функции **Read()** используются другие варианты операторов извлечения с аргументами-данными, выполняющими просто перенос данных из потока, связанного с файлом, в данные-члены прикладного класса.

В приведенном выше тексте программы выполнялась только запись объектов класса **Men** в файл. Для чтения их из файла следует, закрыв поток вывода в файл, открыть поток чтения из него и в цикле выполнить операции извлечения из файла записанных в него объектов:

```
ZeroMemory(report,strlen(report));//Обнулим report
ifpstream instrm("Myfile.dat,ios::binary");//Создаем
                    //объект instrm класса ifpstream для ввода
                    //из файла MYFILE.DAT в двоичном режиме
Men* pMan=new Men;//Создаем один "пустой" объект для чтения
for(int i=0;i<3;i++){
  instrm>>pMan;//Читаем из файла объект за объектом
  pMan->Print();//Формируем строку вывода
  strcat(report,pMan->txt);//и переносим ее в report
  }
MessageBox(NULL,report,"Чтение",MB_OK);//Вывод в окно
instrm.close();//Закроем поток ввода и файл
```

Обратите внимание на важное обстоятельство: для того чтобы можно было создать "пустой" объект класса для чтения в него данных с диска, в классе необходимо иметь конструктор по умолчанию. В определение класса **Men** в файле PERSIST.H такой конструктор включен не был. Таким образом, чтобы приведенный выше фрагмент мог выполняться, класс **Men** необходимо дополнить конструктором по умолчанию, т. е. конструктором без параметров, не выполняющим никаких действий:

```
Men(){}//Конструктор по умолчанию для создания "пустого" объекта
```

Часть 2
Лабораторный практикум

Работы лабораторного практикума

Работа №1. Понятия класса и объекта (индивидуальное задание А)

Опишите в заголовочном файле (с расширением .H) класс, состоящий из указанных в индивидуальном задании А закрытых данных-членов и соответствующего количества открытых функций-членов **Setxxx()** для инициализации данных. Кроме этого, включите в состав класса открытую функцию **Print()** для распечатки в окне сообщения значений объектов класса в наглядном формате с соответствующими комментариями (например: "**Возраст: 22**" или "**Банк: Гута-банк**"). В описание класса включите не только прототипы функций **Setxxx()**, но и их реализацию, т. е. сделайте эти функции встроенными.

В невстроенной функции **Print()** объявите достаточно длинный пустой символьный массив **txt** для формирования выводимой строки и включите вызовы функций Windows **wsprintf()** для формирования строки и **MessageBox()** для ее вывода. Предусмотрите в функции **Print()** анализ значения булевой переменной-члена и вывод этого значения в виде разумной строки текста (например: "**Наличие: есть**" или "**Наличие: нет**").

Составьте программу (файл .CPP), в которой создаются два объекта описанного класса, один – по имени, а другой – с помощью указателя. Вызовом функций **Setxxx()** инициализируйте данные обоих объектов конкретными значениями, а функцией **Print()** выведите в последовательные окна сообщений содержимое обоих объектов.

Работа №2. Встроенные и невстроенные функции-члены

Воспользуйтесь текстом программы из работы **№1**. Замените объявление и описание одной из функций **Setxxx()**, сделав ее невстроенной, для чего в объявлении класса (в файле .H) укажите

только прототип функции, а ее определение приведите отдельно, разместив его в файле .CPP вне функции **WinMain()**. Остальные функции **Setxxx()** оставьте встроенными.

Запустив программу под управлением отладчика, остановите ее выполнение в точке перед вызовом первой из функций **Setxxx()** и откройте кадр CPU, в котором виден результат компиляции программы. Рассмотрите строки программы, относящиеся к вызову функций-членов. Ответьте на следующие вопросы:

а. Сколько команд процесссора занимает выполнение встроенной функции?

б. Сколько команд процессора занимает выполнение невстроенной функции? Обратите внимание на участки кода, выполняющие подготовительную работу (настройку стека), вызов функции (команда процессора **CALL**) и завершающие действия (восстановление стека).

в. Сколько раз повторяется тело встроенной функции при ее повторных вызовах? А невстроенной?

Работа №3. Конструкторы

Модифицируйте программу предыдущей работы. Удалите из определения класса все функции для инициализации данных (**Setxxx()**), заменив их невстроенным конструктором. Прототип конструктора определите в заголовочном файле, его текст – в файле программы до функции **WinMain()**. Создайте два-три объекта класса вызовом конструктора с указанием конкретных значений аргументов. Выведя с помощью функции **Print()** содержимое объектов, проконтролируйте результат.

Работа №4. Деструкторы

Модифицируйте программу предыдущей работы. Дополните класс описанием деструктора, в котором выведите с помощью функций **wsprintf()** и **MessageBox()** значение *адреса* уничтожаемого объекта. Адрес объекта содержится в указателе **this**, неявно поступающем в любую функцию-член при ее вызове, в том числе и в деструктор.

Кроме этого, модифицируйте текст функции-члена **Print()**, включив в нее вывод в окно сообщения, помимо содержимого объекта, еще и значение его адреса.

В главной функции приложения создайте два объекта класса, один по имени, а другой – посредством указателя на объект. Выведите (как и в предыдущих программах) содержимое этих объектов вызовом функции-члена **Print()**. Учтите, что при выделении памяти под объект с помощью оператора **new** ее полагается освобождать (после того, как необходимость в ней отпала) оператором **delete**, поэтому предусмотрите в соответствующем месте функции **WinMain()** предложение с этим оператором.

Запустите программу, проанализируйте выводимую ею информацию. Убедитесь, что при уничтожении объектов для каждого из них автоматически вызывается деструктор. Ответьте на вопрос: "Когда выделяется и когда освобождается память для каждого из ваших объектов?"

Работа №5. Конструктор с инициализацией по умолчанию

Вернитесь к тексту программы из работы **№3**. Модифицируйте объявление конструктора в файле .H, включив в него инициализацию всех данных-членов класса какими-либо значениями по умолчанию. Создайте столько объектов класса, сколько может быть вариантов вызова конструктора с инициализацией по умолчанию. Выведя с помощью функции **Print()** содержимое всех созданных объектов, проконтролируйте результат.

Работа №6. Статическая переменная в составе класса

Воспользовавшись программой из предыдущей работы, введите в состав закрытых членов класса две дополнительных переменных с именами, например, **number** и **counter**, в первую из которых будет заноситься номер создаваемого объекта, а во вторую – полное число созданных объектов. Поскольку переменная **counter**, хотя она и входит в состав каждого объекта, должна в любой момент иметь *единственное* доступное всем объектам значение, ее следует объявить статической, предварив ее объявление спецификатором класса памяти **static** и указав начальное значение (в данном случае 0). В конструктор объекта включите предложения инкремента переменной **counter** и сохранения ее нового значения в переменной **number**. В функции **Print()** предусмотрите вывод в окно сообщения, помимо содержимого объекта, еще и значений двух новых переменных.

Включите в состав класса деструктор, в котором выполняйте декремент переменной **counter** (поскольку при уничтожении объекта общее число объектов уменьшается на 1).

В главной функции **WinMain()** создайте первый объект вашего класса *посредством указателя* и выведите в окно сообщения значения всех его членов. Создайте также второй объект, выведите его содержимое, после чего снова выведите содержимое *первого* объекта. Создайте третий объект и еще раз выведите содержимое как третьего, так и первого объектов. Уничтожьте вызовом оператора **delete** второй или третий объект и снова выведите содержимое первого. Удостоверьтесь в том, что в каждом конкретном объекте значение переменной **counter** описывает общее число созданных на данный момент времени объектов, а в переменной **number** хранится номер данного объекта.

Работа №7. Перегрузка функций

Воспользуйтесь текстом работы **№3** (класс с конструктором) в качестве заготовки. Добавьте в определение класса пару перегруженных функций, имена которых совпадают с именем одного из членов-данных класса, но начинаются с прописной буквы. *Сделайте их невстроенными.* Первая функция должна устанавливать значение соответствующего данного, а вторая (с тем же именем) – возвращать значение этого данного. Фактически вы создадите функции, которые в современном программировании носят название "свойств" (properties) – одноименных перегруженных функций, позволяющих как задавать, так и получать значения переменных. Модифицируйте функцию **Print()**, использовав в ней для получения значения данного перегруженную функцию.

В главной функции **WinMain()** создайте объект вашего класса вызовом конструктора с некоторым набором аргументов. Воспользуйтесь перегруженной функцией для присвоения соответствующему данному класса нового значения. Вызовом функции **Print()** выведите исходное содержимое объекта, а также его содержимое после модификации. Ответьте на следующие вопросы:

а. В каком месте программы вызывается каждая из перегруженных функций?

б. Каким образом компилятор определяет, какую из перегруженных функций следует вызывать в том или ином месте?

в. Откройте объектный файл вашей программы (с расширением .OBJ). Найдите в нем имена ваших перегруженных функций. Расшифруйте декорированные имена перегруженных функций для вашего конкретного случая.

Работа №8. Перегрузка конструкторов (индивидуальное задание B)

Составьте обычное приложение Windows с главным окном и обработкой сообщений **WM_PAINT** и **WM_DESTROY**. Задайте для главного окна достаточно большой размер и цветной фон. Включите в заголовочный файл приложения определение класса в соответствии с индивидуальным заданием **B**. Класс описывает некоторые характеристики определенной фигуры (прямоугольника, эллипса, линии и др.), несколько экземпляров которой будут выводиться в окно приложения. В число задаваемых характеристик фигуры могут входить ее координаты, размеры, цвет заливки и др.

Определите в классе несколько перегруженных конструкторов с разным числом параметров для создания различных вариантов экземпляров фигуры, как это указано в индивидуальном задании.

Включите в состав открытых членов класса функцию **Draw()**, которая, если ее вызывать в функции **OnPaint()**, будет выводить в окно приложения создаваемые в приложении фигуры. Поскольку все функции рисования требуют в качестве первого параметра дескриптор контекста устройства, функция **Draw()** должна принимать этот дескриптор в качестве своего единственного параметра.

В функции **OnPaint()** создайте несколько объектов фигур по числу определенных в классе конструкторов и выведите эти объекты в окно приложения вызовом для них функции **Draw()**. Убедитесь в том, что вид выводимых фигур соответствует определению конкретных объектов.

Работа №9. Перегрузка операторов (индивидуальное задание C)

Воспользуйтесь в качестве заготовки текстом предыдущей работы. Включите в заголовочный файл приложения определение класса в соответствии с индивидуальным заданием **C**. Класс описывает некоторую фигуру (линию, прямоугольник и др.), несколько экземпляров которой будут выводиться в окно приложения. Если фигура является замкнутой, она должна быть прозрачной. В число за-

даваемых характеристик фигуры могут входить ее координаты, размеры, цвет образующей и др.

Определите в классе конструктор для создания объектов с заданными координатами, а также конструктор без параметров для создания "пустого" (неинициализированного) объекта.

Включите в состав открытых членов класс *две* функции рисования **Draw1()** и **Draw2()**. Пусть **Draw1()** выводит фигуру толстым синим пером, а **Draw2()** – красным.

Включите в состав открытых членов класса перегруженный оператор +, выполняющий над объектами класса действия, указанные в индивидуальном задании.

Определите в функции **OnPaint()** два объекта вашего класса, а также объект, получаемый как "сумма" этих двух объектов. Выведите в окно приложения два исходных объекта с помощью функции **Draw1()**, а "объект-сумму" с помощью функции **Draw2()**. Убедитесь в том, что вид выводимых фигур соответствует определению конкретных объектов, а операция суммирования осуществляется указанным в индивидуальном задании образом.

Работа №10. Базовые и производные классы

а. Формирование сводного отчета по всем объектам класса. Воспользуйтесь текстом работы **№3** в качестве заготовки. Включите в состав *открытых* членов класса, помимо конструктора, пустой символьный массив **txt** достаточной длины, удалив его из функции **Print()**. Это даст возможность обращаться к нему не только из функций класса, но и из программы. В функции **Print()** удалите также вызов функции **MessageBox()**, оставив только формирование строки с содержимым объекта в массиве **txt**.

В тексте программы (файл .CPP) предусмотрите символьный массив **report** для формирования сводной таблицы объектов (не забудьте обнулить этот массив). Объявите массив из 3-4 указателей на исходный класс. Вызовом конструктора с различными значениями аргументов создайте соответствующее количество объектов, сохранив их указатели в элементах массива указателей. Организуйте цикл формирования сводного отчета, в котором сначала вызовом функции **Print()** для каждого объекта заполняется строка **txt**, а затем с помощью функции C++ **strcat()** эти строки объединя-

ются в символьном массиве **report**. Наконец, с помощью функции **MessageBox()** выведите полученный сводный отчет в окно сообщения. Проконтролируйте результат.

б. Обслуживание производного класса. Дополните состав заголовочного файла задания **а** описанием производного класса. Включите в состав открытой секции этого класса две функции-члена: конструктор, который должен вызывать конструктор производного класса с передачей ему необходимых аргументов, и функцию **Print()**, замещающую одноименную функцию из базового класса. Текст новой функции **Print()** оставьте той же, лишь незначительно изменив выводимый ею комментарий (например, вместо слова "**Название:**" выводите "**Новое название:**". Для того чтобы в функциях производного класса можно было обращаться к данным базового, назначьте данным базового класса атрибут **protected**.

В тексте программы (файл .CPP) оставьте строки создания массива объектов базового класса и формирования сводного отчета по ним и введите практически такие же строки создания второго массива объектов, но уже производного класса, с добавлением их данных в сводный отчет. Функцией **MessageBox()** выведите сводный итоговой отчет в окно сообщения. Обратите внимание на то, что массивы объектов базового и производного классов приходится обрабатывать по отдельности, двумя последовательными циклами. Ответьте на следующие вопросы:

а. Сколько циклов обработки данных придется предусмотреть в программе, если от базового класса образовано 5 различных производных классов?

б. Можно ли при такой организации классов получить сводные данные по объектам всех классов (например, суммарную стоимость всех товаров или количество товаров, цена которых превышает некоторую величину)?

Работа №11. Виртуальные функции

Выполните модификацию предыдущей работы (**№10b**). Объявите функции **Print()** обоих классов виртуальными. В тексте программы создайте теперь уже *один* массив из 4-5 указателей на базовый класс. Некоторым из этих указателей назначьте адреса объектов базового класса (вызовом соответствующего конструктора со

списком необходимых аргументов); другим указателям назначьте адреса объектов производного класса. В едином цикле сформируйте общий сводный отчет по объектам обоих классов. Изучив вывод программы, убедитесь в том, что хотя все указатели являются указателями на базовый класс, но для объектов каждого класса (базового или производного) вызывается соответствующий вариант виртуальной функции **Print()**, что дает возможность обрабатывать *в одном цикле* массивы объектов как базового, так и любого количества различных производных классов.

Работа №12. Потоки ввода-вывода

Воспользуйтесь в качестве заготовки программой из работы **№9** (индивидуальное задание **С**). Удалите из класса описание перегруженного оператора сложения. Модифицируйте функции **Draw1()** и **Draw2()** так, чтобы первая выводила в окно заданную фигуру толстым цветным пером, а вторая – тонким черным. Объявите в качестве глобальных переменных четыре указателя на объекты вашего класса. В части **а** работы будет использоваться одна пара этих указателей; в части **б** – другая.

Включите в приложение обработку сообщения **WM_CREATE**. В функции **OnCreate()** создайте два объекта класса (используя объявленные ранее указатели): один – вызовом конструктора с некоторым разумным набором аргументов, а второй – "пустой" (с помощью конструктора по умолчанию). В функции **OnPaint()** последовательным вызовом для этого объекта обсих функций **Draw()** выведите в окно два изображения объекта. Убедитесь в том, что изображения геометрически совпадают.

а. Запись объекта на диск и чтение с диска традиционными файловыми функциями Windows. Включите в функцию **OnCreate()** предложения создания файла, записи в него содержимого вашего объекта, сброса указателя в файле в начало файла и чтения данных с диска в созданный заранее второй (неинициализированный) объект. В функции **OnPaint()** выведите (разными функциями **Draw()**) изображения обоих объектов – исходного и заполненного данными с диска. Убедитесь в совпадении изображений.

б. Запись объекта на диск и чтение с диска с помощью потоков ввода-вывода. Подключите к программе заголовочный файл **<fstream.h>**. Создайте в функции **OnCreate()** третий объекта вашего класса с отличным набором аргументов и четвертый – "пустой". Создайте поток файлового вывода – объект класса **ofstream**, указав в аргументах конструктора новое имя файла. Вызовом функции **write()** для объекта потока запишите в файл содержимое объекта. В качестве аргументов этой функции указывается адрес объекта, приведенный к типу **char***, и размер объекта в байтах. Закройте поток вывода.

Создайте поток файлового ввода – объект класса **ifstream**. Вызовом функции **read()** для этого объекта прочитайте данные с диска в четвертый (неинициализированный) объект. В функции **OnPaint()** выведите в окно изображения обоих объектов. Убедитесь в совпадении изображений.

Работа №13. Перегрузка в прикладном классе операторов вставки и извлечения

Выполняя запись и чтение с помощью потоков ввода-вывода, удобно вместо относительно громоздких предложений с потоковыми функциями **write()** и **read()** использовать операторы вставки в поток (**<<**) и извлечения из потока (**>>**). Однако эти операторы C++, перегруженные для потоков, по умолчанию осуществляют преобразование числовых данных в символьные и наоборот. Для того чтобы осуществлять запись и чтение числовых данных (что обычно и требуется в прикладных программах обработки результатов экспериментов или моделирования), необходимо перегрузить операторы вставки и извлечения применительно к составу прикладного класса.

Воспользуйтесь в качестве заготовки текстом предыдущей работы. Включите в состав вашего класса перегруженные операторы << и >>, объявив их дружественными операторными функциями.

В файле с исходным текстом программы определите содержимое перегруженных операторных функций. Каждая из них должна посредством функций **write()** и **read()** посылать в поток или извлекать из потока последовательно все члены вашего класса. Не забудьте, что в качестве аргументов этих функций указываются ад-

реса пересылаемых данных, приведенные к типу **char***, и их размер в байтах.

Как и в предыдущей работе, создайте в функции **OnCreate()** объект вашего класса с некоторым набором аргументов и второй, "пустой" объект. Создайте поток файлового вывода – объект класса **ofstream**, указав в аргументах конструктора новое имя файла. Перешлите содержимое объекта вашего класса в файл, используя оператор вставки в поток (<<). Закройте поток вывода.

Создайте поток файлового ввода – объект класса **ifstream**. С помощью оператора извлечения из потока (>>) прочитайте данные с диска в неинициализированный объект вашего класса. В функции **OnPaint()** выведите в окно изображения обоих объектов. Убедитесь в совпадении изображений.

Работа №14. Библиотечный класс string

Если в классе, помимо числовых данных, имеются указатели на символьные строки, то такой объект нельзя записать в файл, поскольку в файл следует записывать не указатели, а сами символьные строки. Для включения в объекты символьных строк неопределенной длины удобно воспользоваться классом **string** из библиотеки классов C++. Преобразуйте свой класс так, чтобы в объектах класса вместо указателей на символьные строки содержались сами строки в виде объектов класса **string**.

Воспользуйтесь в качестве заготовки текстом работы **№10а** (класс, позволяющий формировать сводный отчет по всем объектам). Подключите к программе заголовочный файл **<cstring.h>**.

Замените в определении вашего класса указатели на символьные строки (типа **char***) переменными типа **string**. Измените соответственно определение конструктора. Передача переменных типа **string** в конструктор класса может осуществляться как по имени (тип параметра **string**), так и по ссылке (тип параметра **string&**).

Модифицируйте список параметров функции **wsprintf()**, формирующей строки отчета, чтобы она могла воспринимать данные типа **string**. Для извлечения из объекта типа **string** самой строки (типа **char***) следует использовать функцию **c_str()** класса **string**.

Для облегчения отладки программы замените все русские слова, используемые в качестве данных класса, на английские. Заменить надо только члены-данные; комментарии, помещаемые в выводимые на экран строки, изменять не следует. Полезно также указать все числовые данные в шестнадцатеричной системе, чтобы их легче было найти в дампе создаваемого в дальнейшем файла.

Убедитесь, что программа по-прежнему выводит в окно сообщения сводный отчет по всем созданным объектам.

Работа №15. Создание "живучих" объектов и запись их на диск

Замена в классе указателей на символьные строки объектами класса **string** позволяет придать объектам этого класса свойство "живучести", т. е. возможности автоматического сохранения на диске состояния объектов после завершения программы.

Руководствуясь образцом, приведенным в разделе 7 пособия, составьте программу, позволяющую записывать на диск содержимое объектов класса из предыдущей работы.

Подключите к программе файл **<classlib\objstrm.h>**.

Объявите ваш класс производным от **TStreamableBase** (с атрибутом **public**).

Включите в соответствующие места программы макросы **DECLARE_STREAMABLE()** и **IMPLEMENT_STREAMABLE()**.

Определите (до функции **WinMain()**) содержимое функций **Read()** и **Write()**, объявленных во вложенном в ваш класс классе **Streamer**.

В функции **WinMain()** после создания нескольких объектов и вывода их содержимого в сводный отчет (как в предыдущей работе) запишите содержимое объектов на диск. С этой целю создайте объект потока (класса **ofpstream**), указав в его конструкторе произвольное имя файла.

В цикле перешлите созданные вами объекты в поток, воспользовавшись для этого оператором вставки <<.

Закройте поток и файл, вызвав для объекта потока функцию **close()**.

Рассмотрите содержимое созданного вами файла. Найдите в нем данные, представляющие собой содержимое созданных вами объ-

ектов. Убедитесь, что, во-первых, в файл записаны *все* созданные вами объекты, и, во-вторых, что для каждого объекта записаны *все* содержащиеся в нем данные-члены.

Работа №16. Чтение с диска "живучих" объектов

На основе предыдущей работы напишите программу чтения объектов из созданного вами файла. Изменению подвергнется только функция **`WinMain()`**.

Удалите из программы строки, связанные с созданием новых объектов, формированием сводного отчета и записью объектов на диск.

Объявите скалярную переменную-указатель на ваш класс и создайте один "пустой" объект этого класса вызовом конструктора без параметров.

Создайте объект файлового потока класса **`ifpstream`**, указав в качестве аргумента имя созданного в предыдущей работе файла.

Организуйте цикл, в котором выполняются три операции: чтение объекта (оператором извлечения **>>**) из файлового потока в "пустой" объект в программе, формирование символьного сообщения вызовом функции **`Print()`** и вывод сформированной строки в окно сообщения. Завершив цикл чтения объектов, закройте поток и файл.

Убедитесь в том, что прочитанные с диска объекты совпадают с теми, которые вы создали в предыдущей программе.

Индивидуальные задания лабораторного практикума

Задание А1. Состав класса **Stock** (склад): наименование товара, его стоимость, наличие товара на складе.

Задание В1. Класс описывает цветной прямоугольник с прозрачным контуром и содержит следующие данные-члены: координаты левого верхнего угла фигуры, ее ширину и высоту, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

с двумя параметрами, именно, структурой типа **RECT** для задания координат фигуры и переменной типа **COLORREF** для задания цвета заливки;

конструктор без параметров, формирующий красный квадрат размером 100 пикселей и расположенный в левом верхнем углу главного окна.

Задание С1. Класс описывает прямую линию, положение которой задается координатами начала и конца линии. Операция сложения заключается в формировании линии, соединяющей концы линий-операндов.

Задание А2. Состав класса **Student** (студент): фамилия, возраст, наличие задолженностей.

Задание В2. Класс описывает прозрачный эллипс с цветным контуром и содержит следующие данные-члены: координаты левого верхнего угла образующего прямоугольника, его ширину и высоту, толщину и цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из шести параметров;

с четырьмя параметрами, именно, двумя структурами типа **POINT** для задания координат левого верхнего и правого нижнего углов образующего прямоугольника, а также переменными типа **int** для задания толщины контура и типа **COLORREF** для задания его цвета;

конструктор без параметров, формирующий эллипс размером 200 × 100 пикселей, нарисованный тонкой красной линией и расположенный в левом верхнем углу главного окна.

Задание С2. Класс описывает прямоугольник, положение которого задается координатами левой верхней и правой нижней вершин. Операция сложения заключается в формировании прямоугольника, описанного вокруг двух прямоугольников-операндов.

Задание А3. Состав класса **Recipe** (рецепт): название блюда, наличие в нем перца, цена.

Задание В3. Класс описывает левую четверть (сектор) цветного эллипса с прозрачным контуром, образованную диагоналями прямоугольника. Класс содержит следующие данные-члены: координаты левого верхнего угла образующего прямоугольника, диаметры эллипса, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

с двумя параметрами, именно, структурой типа **RECT** для задания координат и диаметров фигуры и переменной типа **COLORREF** для задания цвета заливки;

конструктор без параметров, формирующий сегмент красного цвета и высотой 100 пикселей в прямоугольнике размером 300 × 350 пикселей, расположенном в левом верхнем углу главного окна.

Задание С3. Класс описывает окружность фиксированного диаметра (30 – 50 пикселей), положение которой задается координатами центра. Операция сложения заключается в формировании окружности того же диаметра, расположенной посередине между окружностями-операндами.

Задание А4. Состав класса **Music** (музыкальное произведение): название произведения, количество проданных копий, продажа на CD-ROM или DVD.

Задание В4. Класс описывает сектор, составляющий верхнюю четверть прозрачного круга с цветным контуром, и содержит следующие данные-члены: координаты левого верхнего угла образующего квадрата, длину стороны этого квадрата, толщину и цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров, среди которых цвет задается в виде параметра типа **COLORREF**;

модификацию предыдущего конструктора, в которой цвет контура задается в виде трех компонентов основных цветов;

конструктор без параметров, формирующий сектор, нарисованный тонкой красной линией в квадрате размером 300 пикселей, расположенном в левом верхнем углу главного окна.

Задание С4. Класс описывает окружность, положение которой задается координатами центра и радиусом. Операция сложения заключается в формировании окружности, радиус которой равен сумме радиусов окружностей-операндов, а центр совпадает с центром первого операнда.

Задание А5. Состав класса **Account** (банковский счет): номер счета, фамилия вкладчика, возможность снимать проценты.

Задание В5. Класс описывает цветной квадрат с цветным контуром и содержит следующие данные-члены: координаты левого верхнего угла фигуры, длину стороны, цвет и толщину контура, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из шести параметров, в которых цвета задаются в виде переменных типа **COLORREF**;

модификацию предыдущего конструктора, в которой координаты начала фигуры задаются посредством структуры типа **POINT**;

конструктор без параметров, формирующий квадрат размером 50 пикселей, нарисованный толстой красной линией, залитый розовым цветом и расположенный в левом верхнем углу главного окна.

Задание С5. Класс описывает квадрат, положение которого задается координатами центра и длиной стороны. Операция сложения заключается в формировании квадрата, длина стороны которого равна сумме длин сторон квадратов-операндов, а центр совпадает с центром второго операнда.

Задание А6. Состав класса **Product** (продукт): название продукта, наличие в магазине, цена.

Задание В6. Класс описывает левую половину прозрачного круга с цветным контуром и содержит следующие данные-члены: координаты левого верхнего угла образующего квадрата, длину его стороны, цвет и толщину контура. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой координаты начала фигуры задаются посредством структуры типа **POINT**, а цвет контура – в виде трех компонентов основных цветов;

конструктор без параметров, формирующий полукруг диаметром 150 пикселей, нарисованный тонкой красной линией и расположенный в левом верхнем углу главного окна.

Задание С6. Класс описывает окружность, положение которой задается координатами центра и радиусом. Операция сложения заключается в формировании окружности, площадь которой равна сумме площадей окружностей-операндов, а центр расположен посередине между центрами первого и второго операнда.

Задание А7. Состав класса **Drugstore** (аптека): фамилия владельца, наличие лицензии, месячная прибыль.

Задание В7. Класс описывает цветную прямую линию и содержит следующие данные-члены: структуру типа **RECT** для задания координат начала и конца линии, толщину и цвет линии. Определите в классе три перегруженных конструктора:

с полным набором из трех параметров;

модификацию предыдущего конструктора, в которой координаты начала и конца линии задаются посредством двух структур типа **POINT**;

конструктор без параметров, формирующий красную линию толщиной 10 пикселей, проходящую от верхнего левого до нижнего правого угла главного окна.

Задание С7. Класс описывает окружность фиксированного диаметра (30 – 50 пикселей), положение которой задается координатами центра. Операция сложения заключается в формировании окружности того же диаметра, координаты центра которой определяются как сумма соответствующих координат центров окружностей-операндов.

Задание А8. Состав класса **Book** (книга): название, цена, наличие иллюстраций.

Задание В8. Класс описывает цветную дугу толщиной 10 пикселей и содержит следующие данные-члены: координаты левого верхнего и правого нижнего углов образующего прямоугольника; координаты начального и конечного радиусов, ограничивающих дугу; цвет дуги. Определите в классе три перегруженных конструктора:

с полным набором из девяти параметров;

модификацию предыдущего конструктора, в которой координаты образующего прямоугольника и радиусов задаются в виде двух структур типа **RECT**;

конструктор без параметров, формирующий верхнюю половину красной окружности диаметром 200 пикселей, расположенной в левом верхнем углу главного окна.

Задание С8. Класс описывает прямоугольник, положение которого задается координатами левой верхней вершины и длинами двух сторон. Операция сложения заключается в формировании прямоугольника, линейные размеры которого равны сумме размеров прямоугольников-операндов, а левая верхняя вершина накладывается на соответствующую точку первого операнда.

Задание А9. Состав класса **Race** (гонки): дата соревнования в виде символьной строки (год, месяц и день), количество участников, наличие приза.

Задание В9. Класс описывает прозрачный треугольник, нарисованный цветной линией заданной толщины, и содержит следующие данные-члены: три пары координат трех вершин треугольника, толщину линии и ее цвет. Определите в классе три перегруженных конструктора:

с полным набором из восьми параметров;

модификацию предыдущего конструктора, в которой координаты вершин треугольника задаются тремя структурными переменными типа **POINT**;

конструктор без параметров, формирующий большой прямоугольный треугольник, нарисованный тонкой красной линией и расположенный в левом верхнем углу главного окна.

Задание С9. Класс описывает прямую линию, положение которой задается координатами начала и конца линии. Операция сложения заключается в формировании линии, соединяющей начальные точки линий-операндов.

Задание А10. Состав класса **Ward** (больничная палата): количество мест, фамилия врача, наличие свободных коек.

Задание В10. Класс описывает прозрачный квадрат с цветным контуром и содержит следующие данные-члены: координаты левого верхнего угла квадрата, длину его стороны, толщину и цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой координаты начала фигуры описываются структурной переменной типа **POINT**;

конструктор без параметров, формирующий большой квадрат с красным контуром толщиной 10 пикселей и расположенный в левом верхнем углу главного окна.

Задание С10. Класс описывает прямоугольник, положение которого задается координатами левой верхней и правой нижней вершин. Операция сложения заключается в формировании прямоугольника, проведенного через правую нижнюю вершину первого прямоугольника-операнда и левую верхнюю вершину второго прямоугольника-операнда.

Задание А11. Состав класса **Computer** (компьютер): название процессора, тактовая частота, наличие сетевой платы.

Задание В11. Класс описывает правую половину прозрачного круга с цветным контуром и содержит следующие данные-члены: структуру типа **POINT**, содержащую координаты начала образующего квадрата, длину его стороны, цвет и толщину контура. Определите в классе три перегруженных конструктора:

с полным набором из четырех параметров;

модификацию предыдущего конструктора, в которой координаты левого верхнего угла образующего квадрата задаются посредством двух целочисленных переменных, а цвет контура – в виде трех компонентов основных цветов;

конструктор без параметров, формирующий полукруг диаметром 150 пикселей, нарисованный тонкой красной линией и расположенный в левом верхнем углу главного окна.

Задание С11. Класс описывает окружность фиксированного диаметра (30 – 50 пикселей), положение которой задается координатами центра. Операция сложения заключается в формировании окружности того же диаметра, расположенной на одной вертикали с первой окружностью-операндом и на одной горизонтали со второй окружностью-операндом.

Задание А12. Состав класса **Tribe** (племя): название, численность, знакомство с огнем.

Задание B12. Класс описывает цветную пунктирную линию толщиной в 1 пиксель и содержит следующие данные-члены: координаты начала и конца линии, цвет линии. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой координаты начала и конца линии задаются посредством двух структур типа **POINT**;

конструктор без параметров, формирующий горизонтальную красную линию, проходящую по середине главного окна.

Проследите за тем, чтобы сквозь просветы линии просвечивал фон окна.

Задание C12. Класс описывает окружность, положение которой задается координатами центра и радиусом. Операция сложения заключается в формировании окружности, радиус которой равен сумме радиусов окружностей-операндов, а центр совпадает с центром второго операнда.

Задание A13. Состав класса **Patient** (пациент): фамилия, пол, возраст.

Задание B13. Класс описывает цветной круг с прозрачным контуром и содержит следующие данные-члены: координаты левого верхнего угла образующего квадрата, радиус круга, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из четырех параметров;

с тремя параметрами, именно, структурой типа **POINT** для задания координат левого верхнего угла образующего квадрата, целочисленной переменной, задающей радиус круга, и переменной типа **COLORREF** для задания цвета заливки;

конструктор без параметров, формирующий красный круг диаметром 100 пикселей и расположенный приблизительно в левом верхнем углу главного окна.

Задание C13. Класс описывает квадрат, положение которого задается координатами центра и длиной стороны. Операция сложения заключается в формировании квадрата, длина стороны которого равна сумме длин сторон квадратов-операндов, а центр совпадает с центром первого операнда.

Задание A14. Состав класса **PostGraduate** (аспирант): фамилия, год обучения (1, 2 или 3), наличие задолженностей.

Задание B14. Класс описывает цветной круг с цветным контуром и содержит следующие данные-члены: координаты центра круга, его диаметр, цвет и толщину контура, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из шести параметров;

модификацию предыдущего конструктора, в которой координаты центра круга задаются посредством структуры типа **POINT**;

конструктор без параметров, формирующий круг диаметром 50 пикселей, нарисованный красной линией толщиной 5 пикселей, залитый розовым цветом и расположенный в левом верхнем углу главного окна.

Задание C14. Класс описывает прямоугольный равнобедренный треугольник фиксированного размера (длина катета равна 30 – 50 пикселям), стороны которого параллельны осям координат, а положение задается координатами вершины прямого угла. Операция сложения заключается в формировании треугольника, расположенного посередине между двумя треугольниками-операндами.

Задание A15. Состав класса **Magazine** (журнал): название, периодичность в год, поступление в свободную продажу.

Задание B15. Класс описывает прозрачный прямоугольник с цветным контуром заданной толщины и содержит следующие данные-члены: координаты левого верхнего и правого нижнего углов фигуры, толщину и цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из шести параметров;

модификацию предыдущего конструктора, в которой координаты прямоугольника задаются посредством структуры типа **RECT**;

конструктор без параметров, формирующий красный прямоугольник размером 200 × 100 пикселей, расположенный в левом верхнем углу главного окна.

Задание C15. Класс описывает окружность, положение которой задается координатами центра и радиусом. Операция сложения заключается в формировании окружности, центр которой совпадает с центром первой окружности-операнда, а площадь равна сумме площадей обеих окружностей-операндов.

Задание A16. Состав класса **Employee** (служащий): фамилия, зарплата, наличие детей.

Задание B16. Класс описывает цветной квадрат с прозрачным контуром и содержит следующие данные-члены: координаты центра квадрата, задаваемые структурной переменной типа **POINT**, длину стороны квадрата, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из трех параметров;

модификацию предыдущего конструктора, в которой местоположение квадрата задается двумя координатами его центра;

конструктор без параметров, формирующий красный квадрат размером 100 пикселей и расположенный в левом верхнем углу главного окна.

Задание C16. Класс описывает прямоугольный равнобедренный треугольник, стороны которого параллельны осям координат, а положение задается координатами вершины прямого угла и длиной катета. Операция сложения заключается в формировании треугольника, вершина которого совпадает с вершиной первого треугольника-операнда, а длина катета равна сумме длин катетов прямоугольников-операндов.

Задание A17. Состав класса **Bank** (банк): название, уставной капитал, наличие банкомата.

Задание B17. Класс описывает цветной треугольник с прозрачным контуром и содержит следующие данные-члены: координаты трех вершин треугольника, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из семи параметров;

модификацию предыдущего конструктора, в которой координаты вершин треугольника задаются массивом из трех структурных переменных типа **POINT**;

конструктор без параметров, формирующий маленький красный прямоугольный треугольник, расположенный в левом верхнем углу главного окна.

Задание C17. Класс описывает прямую линию, положение которой задается координатами начала и конца линии. Операция сложения заключается в формировании линии, координаты конечных точек которой определяются как средние арифметические координат конечных точек линий-операндов.

Задание A18. Состав класса **Manager** (управляющий): фамилия, возраст, справляется ли с обязанностями.

Задание B18. Класс описывает цветной ромб, у которого ширина в два раза больше высоты. Класс содержит следующие данные-члены: координаты центра фигуры, ширина и цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из четырех параметров;

модификацию предыдущего конструктора, в которой координаты центра ромба задаются структурной переменной типа **POINT**;

конструктор без параметров, формирующий ромб шириной 100 пикселей, покрашенный в красный цвет и расположенный в левом верхнем углу главного окна.

Задание C18. Класс описывает прямоугольник, положение которого задается координатами левой верхней и правой нижней вершин. Операция сложения заключается в формировании прямоугольника, вершины которого находятся в центрах прямоугольников-операндов.

Задание A19. Состав класса **Icecream** (мороженое): название, наличие шоколада, процент жирности.

Задание B19. Класс описывает дугу, образующую почти замкнутую окружность (напоминающую фигуру, используемую окулистом для проверки остроты зрения). В состав членов класса входят следующие данные-члены: структурная переменная типа **POINT** для задания местоположения образующего квадрата, длина стороны этого квадрата, толщина и цвет линии, которой рисуется дуга. Величина разрыва дуги вычисляется функцией рисования как пятая часть длины стороны образующего квадрата, а разрыв в дуге направлен вниз. Определите в классе три перегруженных конструктора:

с полным набором из четырех параметров;

модификацию предыдущего конструктора, в которой местоположение образующего квадрата задается двумя целочисленными переменными;

конструктор без параметров, формирующий красную дугу толщиной 10 пикселей и диаметром 200 пикселей, расположенную в левом верхнем углу главного окна.

Задание С19. Класс описывает окружность фиксированного диаметра (30 – 50 пикселей), положение которой задается координатами центра. Операция сложения заключается в формировании окружности того же диаметра, расположенной на одной горизонтали с первой окружностью-операндом и на одной вертикали со второй окружностью-операндом.

Задание А20. Состав класса **Ship** (корабль): название, водоизмещение, наличие пассажирских кают.

Задание В20. Класс описывает цветной круг с прозрачным контуром и содержит следующие данные-члены: координаты центра круга, задавасмые с помощью структурной переменной типа **POINT**, радиус круга, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из трех параметров;

модификацию предыдущего конструктора, в которой координаты центра круга задаются с помощью двух целых чисел;

конструктор без параметров, формирующий красный круг диамстром 100 пикселей и расположенный в левом верхнем углу главного окна.

Задание С20. Класс описывает прямоугольный равнобедренный треугольник фиксированного размера (длина катета равна 80 – 100 пикселям), стороны которого параллельны осям координат, а положение задается координатами вершины прямого угла. Операция сложения заключается в формировании треугольника, расположенного на одной горизонтали с первым прямоугольником-операндом и на одной вертикали со вторым прямоугольником-операндом.

Задание А21. Состав класса **Person** (индивидуум): фамилия, возраст, пол.

Задание В21. Класс описывает цветную прозрачную окружность, у которой с правой стороны вырезан сектор с раствором 45°, так что в целом фигура напоминает стилизованное изображение открытого рта. Класс содержит следующие данные-члены: координаты левого верхнего угла образующего квадрата, диаметр окружности, толщину и цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой начальные координаты фигуры задаются структурной переменной типа **POINT**;

конструктор без параметров, формирующий фигуру диаметром 200 пикселей, нарисованную красной линией толщиной 5 пикселей и расположенную в левом верхнем углу главного окна.

Задание С21. Класс описывает квадрат, положение которого задается координатами левой верхней вершины и длиной стороны. Операция сложения заключается в формировании квадрата, левая верхняя вершина которого совпадает с соответствующей точкой первого квадрата-операнда, а площадь равна сумме площадей обоих квадратов-операндов.

Задание А22. Состав класса **Tumor** (опухоль): размер, злокачественность, вероятность лечения в процентах.

Задание В22. Класс описывает прозрачный треугольник, нарисованный цветной линией заданной толщины, и содержит следующие данные-члены: три структурные переменных типа **POINT** для задания координат трех вершин прямоугольника, толщину линии и ее цвет. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой координаты вершин треугольника задаются тремя парами целых чисел;

конструктор без параметров, формирующий маленький прямоугольный треугольник, нарисованный толстой красной линией и расположенный в левом верхнем углу главного окна.

Задание С22. Класс описывает небольшую окружность фиксированного диаметра (20 – 30 пикселей), центр которой расположен на 100 пикселей ниже верхнего края окна; расстояние центра от левого края окна задается данным-членом класса. Операция сложения заключается в формировании окружности того же диаметра, расстояние центра которой от левого края окна равно сумме соответствующих расстояний окружностей-операндов.

Задание А23. Состав класса **Butter** (масло): название, наличие растительных добавок, цена.

Задание В23. Класс описывает цветную линию и содержит следующие данные-члены: две пары целых чисел, задающих коорди-

наты начала и конца линии, толщину и цвет линии. Определите в классе три перегруженных конструктора:

с полным набором из шести параметров;

модификацию предыдущего конструктора, в которой координаты начала и конца линии задаются посредством структурной переменной типа **RECT**;

конструктор без параметров, формирующий вертикальную красную линию толщиной 5 пикселей, расположенную вблизи левой стороны главного окна.

Задание С23. Класс описывает прямую линию, начало которой совпадает с началом координат, а координаты конечной точки задаются с помощью данных-членов класса. Операция сложения заключается в формировании линии, координаты конечной точки которой равны суммам соответствующих координат конечных точек линий-операндов.

Задание А24. Состав класса **Drill** (дрель): фирма, число оборотов в минуту, наличие перфоратора.

Задание В24. Класс описывает дугу, образующую почти замкнутую окружность (напоминающую фигуру, используемую окулистом для проверки остроты зрения). В состав членов класса входят следующие данные-члены: две координаты левого верхнего угла образующего квадрата, длина стороны этого квадрата, толщина и цвет линии, которой рисуется дуга. Величина разрыва дуги вычисляется функцией рисования как пятая часть длины стороны образующего квадрата, а разрыв в дуге направлен вверх. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой местоположение образующего квадрата задается структурной переменной типа **POINT**;

конструктор без параметров, формирующий красную дугу толщиной 15 пикселей и диаметром 200 пикселей, расположенную в левом верхнем углу главного окна.

Задание С24. Класс описывает прямоугольный равнобедренный треугольник, стороны которого параллельны осям координат, а положение задается координатами вершины прямого угла и длиной катета. Операция сложения заключается в формировании треуголь-

ника, вершина которого совпадает с вершиной первого треугольника-операнда, а длина катета равна сумме длин катетов прямоугольников-операндов.

Задание A25. Состав класса **City** (город): название, население, площадь.

Задание B25. Класс описывает сектор, составляющий правую четверть цветного вытянутого по горизонтали эллипса с прозрачным контуром и содержит следующие данные-члены: координаты левого верхнего угла образующего эллипса, вертикальный диаметр образующего эллипса, цвет заливки. Горизонтальный диаметр эллипса задайте в два раза больше вертикального. Определите в классе три перегруженных конструктора:

с полным набором из четырех параметров, среди которых цвет заливки задается в виде параметра типа **COLORREF**;

модификацию предыдущего конструктора, в которой цвет заливки задается в виде трех компонентов основных цветов;

конструктор без параметров, формирующий сектор красного цвета, принадлежащий эллипсу с вертикальным диметром 200 пикселей и расположенный в левом верхнем углу главного окна.

Задание C25. Класс описывает прямую линию, положение которой задается координатами начала и конца линии. Операция сложения заключается в формировании линии, соединяющей середины линий-операндов.

Задание A26. Состав класса **Dictionary** (словарь): язык, количество слов, наличие транскрипции.

Задание B26. Класс описывает цветной квадрат с прозрачным контуром и содержит следующие данные-члены: координаты центра квадрата, длину стороны квадрата, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из четырех параметров;

модификацию предыдущего конструктора, в которой местоположение квадрата задается структурной переменной типа **POINT**;

конструктор без параметров, формирующий красный квадрат размером 200 пикселей, расположенный в левом верхнем углу главного окна.

Задание С26. Класс описывает небольшую окружность фиксированного диаметра (20 – 30 пикселей), центр которой расположен на 100 пикселей правее левого края окна; расстояние центра от верхнего края окна задается данным-членом класса. Операция сложения заключается в формировании окружности того же диаметра, расстояние центра которой от верхнего края окна равно сумме соответствующих расстояний окружностей-операндов.

Задание А27. Состав класса **Milk** (молоко): название, жирность в процентах, наличие на складе.

Задание В27. Класс описывает цветную прозрачную окружность, у которой с левой стороны вырезан сектор с раствором 45°, так что в целом фигура напоминает стилизованное изображение открытого рта. Класс содержит следующие данные-члены: координаты центра образующего квадрата, диаметр окружности, толщину и цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой координаты фигуры задаются структурной переменной типа **POINT**;

конструктор без параметров, формирующий фигуру диаметром 200 пикселей, нарисованную красной линией толщиной 15 пикселей и расположенную в левом верхнем углу главного окна.

Задание С27. Класс описывает прямоугольный неравнобедренный треугольник, стороны которого параллельны осям координат, а положение задается координатами вершины прямого угла и длиной катетов. Операция сложения заключается в формировании треугольника, вершина которого совпадает с вершиной второго треугольника-операнда, а длины сторон равны сумме длин катетов прямоугольников-операндов.

Задание А28. Состав класса **Animal** (животное): наименование, количество ног, травоядность.

Задание В28. Класс описывает цветной неправильный четырехугольник, содержащий следующие данные-члены: четыре пары координат вершин фигуры и цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из девяти параметров;

модификацию предыдущего конструктора, в которой координаты вершин четырехугольника задаются четырьмя структурными переменными типа **POINT**;

конструктор без параметров, формирующий трапецию произвольного размера, покрашенную в красный цвет и расположенную в левом верхнем углу главного окна.

Задание С28. Класс описывает прямоугольный равнобедренный треугольник фиксированного размера (длина катета равна 80 – 100 пикселям), стороны которого параллельны осям координат, а положение задается координатами вершины прямого угла. Операция сложения заключается в формировании треугольника, расположенного на одной вертикали с первым прямоугольником-операндом и на одной горизонтали со вторым прямоугольником-операндом.

Задание А29. Состав класса **Poem** (поэма): автор, число строк, наличие рифмы.

Задание В29. Класс описывает сектор, составляющий левую четверть прозрачного круга с цветным контуром, и содержит следующие данные-члены: координаты центра образующего квадрата, длину стороны этого квадрата, толщину и цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из пяти параметров;

модификацию предыдущего конструктора, в которой координаты центра задаются с помощью структурной переменной типа **POINT**;

конструктор без параметров, формирующий сектор круга диаметром 300 пикселей, нарисованный тонкой красной линией и расположенный в левом верхнем углу главного окна.

Задание С29. Класс описывает квадрат, положение которого задается координатами левой верхней вершины и длиной стороны. Операция сложения заключается в формировании квадрата, левая верхняя вершина которого совпадает с соответствующей точкой второго квадрата-операнда, а площадь равна сумме площадей обоих квадратов-операндов.

Задание А30. Состав класса **Doctor** (врач): фамилия, наличие лицензии, количество пациентов.

Задание В30. Класс описывает прозрачный неправильный четырехугольник, нарисованный цветной линией заданной толщины и содержащий следующие данные-члены: четыре пары координат вершин фигуры, толщину контура и его цвет. Определите в классе три перегруженных конструктора:

с полным набором из десяти параметров;

модификацию предыдущего конструктора, в которой координаты вершин четырехугольника задаются четырьмя структурными переменными типа **POINT**;

конструктор без параметров, формирующий трапецию произвольного размера, нарисованную тонкой красной линией и расположенную в левом верхнем углу главного окна.

Задание С30. Класс описывает квадрат, положение которого задается координатами центра и размером стороны. Операция сложения заключается в формировании квадрата, площадь которого равна сумме площадей квадратов-операндов, а центр расположен посередине между центрами первого и второго операнда.

Задание А31. Состав класса **Child** (ребенок): имя, пол, возраст.

Задание В31. Класс описывает цветной четырехугольник с прозрачным контуром и содержит следующие данные-члены: координаты четырех вершин фигуры, цвет заливки. Определите в классе три перегруженных конструктора:

с полным набором из девяти параметров;

модификацию предыдущего конструктора, в которой координаты вершин четырехугольника задаются массивом из четырех структурных переменных типа **POINT**;

конструктор без параметров, формирующий большой красный ромб, расположенный в левом верхнем углу главного окна.

Задание С31. Класс описывает окружность, положение которой задается координатами центра и радиусом. Операция сложения заключается в формировании окружности, центр которой совпадает с центром второй окружности-операнда, а площадь равна сумме площадей обеих окружностей-операндов.

Задание А32. Состав класса **Girl** (девушка): имя, возраст, наличие ухажеров.

Задание B32. Класс описывает прозрачный прямоугольник с цветным контуром заданной толщины и содержит следующие данные-члены: координаты левого верхнего и правого нижнего углов фигуры, задаваемые в виде двух структурных переменных типа **POINT**; толщину контура; цвет контура. Определите в классе три перегруженных конструктора:

с полным набором из четырех параметров;

модификацию предыдущего конструктора, в которой координаты фигуры задаются посредством структуры типа **RECT**;

конструктор без параметров, формирующий прямоугольник размером 200 × 100 пикселей, нарисованный толстой красной линией и расположенный в левом верхнем углу главного окна.

Задание C32. Класс описывает прямоугольный равнобедренный треугольник, стороны которого параллельны осям координат, а положение задается координатами вершины прямого угла и длиной катета. Операция сложения заключается в формировании треугольника, вершина которого совпадает с вершиной второго треугольника-операнда, а длина катета равна сумме длин катетов прямоугольников-операндов.

Список литературы

1. Финогенов К.Г. Прикладное программирование для Windows на Borland C++. – Обнинск: «Принтер», 1999.

1. Финогенов К.Г. Лабораторный практикум "Основы программирования на языке C++ ". – М.: МИФИ, 2004.

2. Финогенов К.Г. Лабораторный практикум "Основы разработки приложений Windows". Кн. 1-2. – М.: МИФИ, 2004 - 2005.