Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

В. В. Смелов

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Рекомендовано
учебно-методическим объединением
высших учебных заведений Республики Беларусь
по образованию в области информатики и радиоэлектроники
в качестве учебно-методического пособия для студентов учреждений,
обеспечивающих получение высшего образования по направлению
специальности «Информационные системы и технологии
(издательско-полиграфический комплекс)»

Минск 2009

УДК 004.434(075.8)
ББК 22.18я73
С50

Рецензенты:
кафедра управления информационными ресурсами
Академии управления при Президенте Республики Беларусь
(кандидат технических наук, профессор *Н. И. Белодед*);
кандидат технических наук,
доцент кафедры информационных технологий
автоматизированных систем БГУИР *О. В. Герман*

Смелов, В. В.

С50 Введение в объектно-ориентированное программирование на C++ : учеб.-метод. пособие для студентов по направлению специальности «Информационные системы и технологии (издательско-полиграфический комплекс». – Минск : БГТУ, 2009. – 94 с.

ISBN 978-985-434-873-5

Учебно-методическое пособие посвящено практическому введению в объектно-ориентированное программирование на языке C++. Основное внимание уделено применению четырех важнейших механизмов. Это технологии абстракции, инкапсуляции, наследования и полиморфизма. Изложение сопровождается многочисленными схемами и примерами. Заключительная глава пособия представляет собой практикум, который может быть использован для закрепления изученного материала. Предполагается, что читатель имеет начальные навыки программирования на языке C++ и работы с отладчиком Microsoft Visual Studio.

Для студентов высших учебных заведений по направлению специальности «Информационные системы и технологии (издательско-полиграфический комплекс)».

УДК 004.434(075.8)
ББК 22.18я73

ISBN 978-985-434-873-5

ВВЕДЕНИЕ

Почему объектно-ориентированное программирование стало востребованным и завоевало большую популярность среди профессиональных программистов-разработчиков? Как правило, все ответы студенческой аудитории сводились к следующим аргументам: удобно программировать сложные интерфейсы, получаются хорошо читаемые программы, проще сопровождать, проще документировать. Все эти ответы правильные, но они не отражают главное преимущество, которое дает объектно-ориентированный подход при разработке программных систем.

Общеизвестный факт, что основные и самые трудно исправимые ошибки при разработке программного обеспечения возникают на этапе проектирования. Большинство таких ошибок возникает на переходе от заказчика к аналитику, а затем к программисту. Заказчик выдвигает требования к программному продукту, аналитик эти требования воплощает в виде проектной документации, а программист реализует программный продукт, руководствуясь этой документацией. При передаче законченного продукта заказчик иногда выясняет, что аналитик неправильно понял его требования, а программист реализовал совсем не то, что предполагал аналитик. Такой сценарий – довольно частое явление в мире разработчиков программного обеспечения. Это происходит, в общем-то, по простой и всем понятной причине – все три взаимодействующие стороны имеют разное представление о том, что и как надо сделать. Проблема в том, что каждый проблему воспринимает по-своему.

Объектно-ориентированный подход к разработке программной системы основывается на двух простых, понятных всем участникам этого процесса понятиях: объект и система. Любое программное приложение при таком подходе представляется как система, состоящая из взаимодействующих объектов, а каждый объект – это совокупность свойств и методов (так называются процедуры изменения свойств объекта). Свойства объектов могут сами являться объектами, и такая вложенность может быть достаточно глубокой. Применение такой формализации для описания программной системы на основных этапах ее разработки (проектирование, кодирование и тестирование) и является сутью объектно-ориентированного подхода. Основной эффект здесь достигается за счет того, что основные участники этого процесса используют однозначные и всем им понятные средства описания программной системы.

Изучение технологии объектно-ориентированного программирования, по мнению автора, целесообразно разбить на три этапа. На первом этапе следует усвоить основные понятия объектно-ориентированного программирования, на втором – ознакомиться с принципами объектно-ориентированного проектирования, а на третьем – снова вернуться к программированию, но уже для изучения техники программирования на определенном объектно-ориентированном алгоритмическом языке.

Предлагаемое пособие предназначено для применения его на первом этапе обучения. Изучив его, читатель ознакомится с основными понятиями и принципами объектно-ориентированного программирования. В качестве инструмента демонстрации примеров используется язык C++. Поэтому для эффективного усвоения материала читатель должен обладать навыками программирования на этом языке в объеме источника [1] и опытом работы в среде Visual Studio. Последняя глава пособия является практикумом, предназначенным для закрепления изученного материала. Для выполнения практических работ достаточно внимательно изучить приведенные в книге примеры.

На втором этапе изучения рекомендуется книга [4], посвященная объектно-ориентированному проектированию.

И для тех, кто на третьем этапе изучения в качестве инструмента разработки своих приложений выберет язык C++, рекомендуется уже ставшая классической книга создателя этого языка Бьерна Страуструпа [2].

Язык C++ – компилируемый, строго типизируемый язык программирования общего назначения. Он поддерживает разные парадигмы программирования, но наибольшее внимание уделено объектно-ориентированному.

C++ возник в начале 1980-х гг. в фирме Bell Laboratories как усовершенствованный язык программирования C. До начала стандартизации он развивался в основном усилиями сотрудника Bell Laboratories Бьерна Страуструпа, который по праву считается основателем этого языка.

Первый стандарт языка появился в результате работы комитета ANSI-ISO в 1998 г. и назывался ISO/IEC 14882:1998 Programming Language C++. Стандарт состоял из двух основных частей: ядра языка и стандартной библиотеки. Значительная часть стандартной библиотеки до включения ее в стандарт C++ называлась Standard Template Library (STL) и сначала была разработкой фирмы Hewlett Packard,

а затем Silicon Graphic. Кроме того, стандарт имел нормативную ссылку на стандарт языка С 1990 г. и не определял самостоятельно те функции, которые заимствованы у этого языка. Сейчас стандартизацией С++ занимается комитет JTC1/SC22/WG21, входящий в состав Международной стандартизирующей организации (МОК, ISO).

Сегодня последняя действующая версия стандарта С++ датируется 2003 г. (ISO/IEC 14882:2003) и уже объявлено, что в 2009 г. выйдет новая версия стандарта языка под кодовым названием С++09. С процессом стандартизации языка С++ можно ознакомиться на официальном сайте комитета JTC1/SC22/WG21 (http://www.open-std.org/jtc1/sc22/wg21).

Справедливо отмечая, что С++ не является простым языком, в своей книге [1] Бьерн Страуструп приводит следующие доводы в пользу его изучения:

– достаточная ясность для обучения основным концепциям программирования;

– доступность для организаций с различными средами разработки и исполнения;

– содержательность для служения инструментом обучения более сложным концепциям и методам;

– реалистичность, эффективность и гибкость для критичных проектов.

Изучив С++, читатель получит в свои руки мощный инструмент, позволяющий ему эффективно решать самые разнообразные задачи. Кроме того, концепции, положенные в основу языка программирования С++, используются во многих других современных системах программирования и читатель может без больших усилий начать программировать на C# или Java.

Глава 1. КЛАССЫ

1.1. Понятие класса

Каждый идентификатор, используемый в программе, имеет определенный тип. Все типы в C++ делятся на две группы: фундаментальные и определяемые пользователем. Фундаментальные типы – это логический (**bool**), символьный (**char**), целый (**int**), с плавающей точкой (**float**, **double**, **long double**) и синтаксический (**void**). Определяемые пользователем типы – это перечисление (**enum**) и класс (**class**).

Класс – это определяемый пользователем тип, представляющий собой описание объектов этого типа. В общем случае это описание набора объектов различных типов (данные-члены или атрибуты класса), набора функций (функции-члены класса) и правил доступа к содержимому объекта.

Необходимо правильно понимать отличие между понятиями «класс» и «объект класса». Объект – это реализация класса. В этом смысле класс описывает множество объектов, имеющих общую природу. Разница между классом **A** и объектом **a** класса **A** такая же, как и разница между типом **int** и переменной **x** типа **int**.

Наиболее близким понятию «класс» является понятие «структура». Более того, структура в C++ является частным случаем класса, у которого нет ограничений на доступ к членам-функциям и членам-данным.

Реализацию данных-членов в объекте называют ***свойствами***, а реализацию функций-членов в объекте – ***методами***. Свойства и методы объекта иногда называют ***элементами*** объекта. Таким образом, объект в C++ представляет собой набор свойств и методов. Или иначе, объект – это совокупность его элементов. Некоторые элементы объекта могут быть доступны для использования извне объекта, а к другим доступ может быть ограничен или закрыт.

Важно знать, что понятия «класс» и «объект» существуют в приложении на языке C++ только на уровне компилятора. Исполняющая среда приложения ничего «не знает» об объектах и тем более классах.

Определение класса в программе сходно с определением структуры, но используется ключевое слово «**class**». Объявление объекта

в простейшем случае ничем не отличается от объявления переменной в программе.

На рис. 1.1 приведен пример программы, использующей класс с именем **Person**. В начале программы определяется класс **Person** (отмечено комментариями), содержащий три массива типа **char** (данные-члены) и одну функцию с именем **Plus** (функция-член). В теле главной функции создается объект **p1** класса **Persson**. Обратите внимание, что создание объекта осуществляется при его объявлении.

```
#include "stdafx.h"
#include <cstring>                    // memcpy_s
class Person {                        // определение класса
             char FirstName[50];
             char LastName[50];
             char Address[200];
             int  plus(int x1, int x2){return x1+x2;};
             };

int _tmain(int argc, _TCHAR* argv[])
{
   int sizePerson = sizeof(Person);   // = 300
   Person p1;                         // создание объекта типа Person

   memcpy_s(&p1,              50, "Petrovich",  sizeof("Petrovich"));
   memcpy_s((void *)(((int)&p1)+50), 50, "Evgeny", sizeof("Evgeny"));
   memcpy_s((void *)(((int)&p1)+100), 200, "Minsk", sizeof("Minsk"));

   return 0;
}
```

Рис. 1.1. Пример программы с использованием простейшего класса

1.2. Исследование простейшего класса

Продолжим рассматривать пример, приведенный на рис. 1.1. Для доступа к элементам объекта программист обычно использует те же нотации, что и для доступа к элементам структуры Но применение этих нотаций для доступа к элементам объекта **p1** приведет в ошибке компиляции, содержащее сообщение похожее на следующее.

```
...cannot access private member declared in class 'Person'
.........................................
...see declaration of 'Person'
```

В действительности трудно представить, для чего может быть применен класс **Person** в том виде, в котором он представлен в этом

примере. Тем не менее можно заполнить массивы объекта **p1** способом, который представлен на рис. 1.1.

Если установить точку прерывания отладчика на операторе **return** и просмотреть с помощью функции-отладчика **watch** память, занимаемую объектом **p1**, то результат будет, как на рис. 1.2.

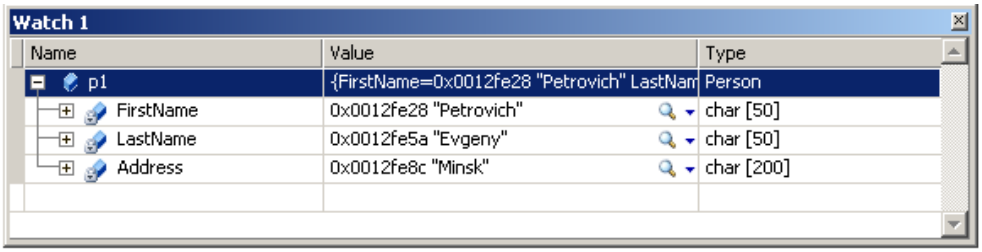

Рис. 1.2. Область памяти, занимаемая объектом

Следует обратить внимание, что для объекта **p1** отведено 300 байт памяти для трех символьных массивов, описанных в классе **Person**, но никак не отражена функция **plus**. Более того, все попытки вызвать эту функцию приводят к ошибкам компиляции.

1.3. Управление доступом

При объявлении членов класса могут использоваться модификаторы **public**, **private**, **protected**. Эти модификаторы определяют порядок доступа к соответствующим элементам объекта.

Модификатор **public** указывает на то, что элементы объекта будут доступны для использования как в методах самого объекта, так и во внешнем по отношению к объекту коде программы.

На рис. 1.3 приведен пример класса **Person**, для членов которого используется модификатор **public**.

Действие модификатора **public** в данном случае распространяется на все члены класса, поэтому все элементы объекта **p1** становятся доступными для использования из внешнего кода. Обратите внимание на то, что теперь появляется возможность использовать символические имена для доступа к свойствам и методам объекта, а функциональность объектов класса **Person** в этом случае ничем не отличается от функциональности подобной структуры.

Модификатор **private** устанавливается по умолчанию и указывает на то, что элементы объекта будут доступны только методам самого объекта и недоступны внешнему по отношению к объекту коду.

Именно поэтому в примере на рис. 1.1 не было возможности использовать символические имена элементов объекта **p1**.

```
#include "stdafx.h"
#include <cstring>                          //  strcpy_s
class Person {
         public:
             char FirstName[50];
             char LastName[50];
             char Address[200];
             int  plus(int x1, int x2){return x1+x2;};
             };

int _tmain(int argc, _TCHAR* argv[])
{
   int sizePerson = sizeof(Person);         // = 300
   int x = 0;
   Person p1;                               // p1  -  объект типа Person

   strcpy_s(p1.FirstName, sizeof(p1.FirstName), "Petrovich");
   strcpy_s(p1.LastName,  sizeof(p1.LastName), "Evgeny");
   strcpy_s(p1.Address,   sizeof(p1.Address), "Minsk");
   x = p1.plus(3, 4);                       // = 7
   return 0;
}
```

Рис. 1.3. Применение модификатора public

На рис. 1.4 представлен класс **Person**, в котором используются модификаторы **private** и **public**.

Класс **Person** содержит три символьных массива (**FirstName**, **LastName** и **Address**) и одну функцию (**toLine**), доступ к которым запрещен (или, говорят, закрыт) за пределами класса. На все перечисленные члены класса распространяется действие модификатора **private**. Если ниже по тексту программы встречается другой модификатор (в нашем случае **public**), то расположенные еще ниже описания подчиняются действию последнего модификатора. Пять функций-членов класса **Person setFirstName**, **setLastName**, **setAddress**, **getLine** и **MaxSizeLine** – описывают методы объектов типа **Person**, доступ к которым будет возможен (открыт) из основной программы.

Следует обратить внимание, что доступ к членам-функциям и данным класса в границах самого класса возможен всегда. Например, функция **setFirstName**, которая подчинена действию модификатора **public**, использует символьный массив **FirstName** из области действия модификатора **private**. Использование же этого массива в тексте программы за пределами класса приведет к ошибке компилятора. Аналогичная ситуация и с вызовом функций.

Обычно описание классов выносят из текста основной программы в отдельные файлы с расширением **h** (от англ. header file – заголовочный файл). В качестве имени файла, как правило, используется имя класса. В нашем случае полное имя файла будет **Person.h**. Для включения заголовочного файла в основной файл программы следует применить директиву препроцессора **#include**.

```
class Person
{
#define MAXSIZELINE
(sizeof(FirstName)+sizeof(LastName)+sizeof(Address)+2)
  private:
          char FirstName[50];
          char LastName[50];
          char Address[200];
          char* toLine(char *l) const
          {
              int n = 0;
              strcpy_s(l, sizeof(FirstName), this->FirstName);
              strcpy_s(l + (n += strlen(this->FirstName)), 2, ",");
              strcpy_s(l + (n += 1),sizeof(LastName), this->LastName);
              strcpy_s(l + (n += strlen(this->LastName)), 2, ",");
              strcpy_s(l + (n += 1), sizeof(Address),this->Address);
              return l;
          };

  public:
          void setFirstName(char fn[])
                  {strcpy_s(FirstName, sizeof(FirstName), fn);};
          void setLastName(char ln[])
                  {strcpy_s(LastName, sizeof(LastName), ln);};
          void setAddress(char as[])
                  {strcpy_s(Address, sizeof(Address), as);};
          char* getLine (char *l) const {return toLine(l);};
          int MaxSizeLine(){return MAXSIZELINE;}
#undef MAXSIZELINE
};
```

Рис. 1.4. Применение модификатора private

На рис. 1.5 приведен пример программы, использующей класс **Person**, описание которого находится в файле **Person.h**.

В программе используются методы **setFirstName**, **setLastName**, **setAddress** объекта **p1** для установки значений его свойств, а помощью метода **getLine** формируется символьная строка, представляющая собой конкатенацию строк **FirstName**, **LastName** и **Address** с запятой в качестве разделителя.

Следует отметить, что такой способ реализации класса, когда его атрибуты закрыты, а доступ к ним осуществляется с помощью специ-

альных открытых методов, является типичным в практике объектно-ориентированного программирования. Открытые функции-члены, предназначенные для установки значений закрытых атрибутов класса, на программистском сленге часто называют «сеттеры» (от англ. set – устанавливать), а открытые функции-члены для чтения значения атрибутов – «геттеры» (от англ. get – получать).

```
#include "stdafx.h"
#include <cstring>     //  strcpy_s
#include "Person.h"    //  класс Person

int _tmain(int argc, _TCHAR* argv[])
{
   int sizePerson = sizeof(Person);
   Person p1;
   char* Line = new char[p1.MaxSizeLine()];
   p1.setFirstName("Petrovich");
   p1.setLastName("Evgeny");
   p1.setAddress("Minsk");
   strcpy_s(Line,p1.MaxSizeLine(),p1.getLine(new char[303]));
   return 0;
}
```

Рис. 1.5. Программа, использующая класс Person

Действие модификатора **protected** похоже на действие модификатора **private**, но возможность доступа к свойствам и методам объекта подчиненных этому модификатору распространяется на методы производных объектов. Понятия «производный класс» и «производный объект» будут пояснены ниже.

На рис. 1.6 приведен пример программы, выполняющей то же, что и программа на рис. 1.5, но с применением указателя на объект класса **Person**. В этом случае объект создается динамически с помощью оператора **new**, а для доступа к элементам применяется оператор → (разыменование указателя на объект).

```
#include "stdafx.h"
#include <cstring>                  //  strcpy_s
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
   Person* pp1 = new Person();      // pp1  - указатель на бъект типа
   char* Line = new char[pp1->MaxSizeLine()];
   pp1->setFirstName("Petrovich");
   pp1->setLastName("Evgeny");
   pp1->setAddress("Minsk");
   strcpy_s(Line,pp1->MaxSizeLine(),pp1->getLine(new char[303]));
   return 0;
}
```

Рис. 1.6. Применение указателей на класс

1.4. Конструкторы класса

В примере реализации класса **Person**, представленном на рис. 1.4, имеется существенный недостаток: сразу после создания объекта этого класса значение свойств объекта не инициализированы. Действительно, если сразу после создания объекта выполнить метод **getLine**, то результат его выполнения будет непредсказуемым. Скорее всего (в конечном счете, это зависит от реализации компилятора C++), это приведет к аварийному завершению программы. Для предотвращения подобных ошибок свойства объектов должны быть проинициализированы до того, как они будут использованы. Первоначальная инициализация свойств объекта выполняется с помощью специального механизма, основанного на применении конструктора класса.

Конструктор – это функция-член класса с именем, совпадающим с именем класса. Эта функция вызывается при создании объекта класса и используется, как правило, для инициализации свойств объекта. Конструктор не может возвращать никакое значение, поэтому единственный способ сообщить об ошибке, произошедшей в процессе выполнения конструктора, – это сгенерировать исключение.

Класс может иметь несколько конструкторов, различающихся набором аргументов. В том случае, если конструктор не указан, то компилятор генерирует конструктор по умолчанию.

Конструктор по умолчанию – это конструктор, не имеющий аргументов. В примере на рис 1.5 конструктор по умолчанию вызывается при объявлении объекта **p1**, а в примере на рис. 1.6 – при выполнении оператора **new.**

На рис. 1.7 приведен фрагмент класса **Person**, в котором явно определен конструктор, инициализирующий символьные массивы.

```
class Person
{
  // ..........................................
  public:
         Person(){FirstName[0]= LastName[0]= Address[0] = '\x00';};
  // ..........................................
};
```

Рис. 1.7. Фрагмент класса с объявлением конструктора по умолчанию

В этом случае при создании объекта класса **Person** начальными значениями массивов **FirstName**, **LastName** и **Address** будет пустая строка, а выполнение метода **getLine** не вызовет ошибку.

На рис. 1.8 приведен фрагмент класса **Person**, использующего два конструктора: конструктор по умолчанию и конструктор с аргументами.

```
class Person
{
// ..........................................
  public:
  void setFirstName(char fn[])
         {strcpy_s(FirstName, sizeof(FirstName), fn);};
  void setLastName(char ln[])
         {strcpy_s(LastName, sizeof(LastName), ln);};
  void setAddress(char as[])
         {strcpy_s(Address, sizeof(Address), as);};
  Person()                                  // конструктор по умолчанию
         {FirstName[0]= LastName[0]= Address[0] = '\x00';};
  Person(char fn[], char ln[], char as[]) // конструктор с аргументами
        {
            setFirstName(fn);
            setLastName(ln);
            setAddress(as);
        };
// ..........................................
};
```

Рис. 1.8. Фрагмент класса с объявлением двух конструкторов

Выбор подходящего конструктора осуществляется компилятором. При этом действуют те же правила, которые действуют при перегрузке функций C++ [3].

На рис. 1.9 демонстрируется применение двух видов конструкторов при создании двух объектов класса **Person**. В первом случае использовался конструктор по умолчанию, во втором – конструктор с аргументами.

Часто при разработке класса используют специальный вид конструктора – копирующий конструктор. Копирующий конструктор применяется для создания клона объекта.

На рис. 1.10 приведен фрагмент класса, использующего три конструктора: по умолчанию, с аргументами и копирующего. В качестве аргумента копирующего конструктора выступает объект, который является образцом для создания копии (клона).

Обратите внимание на то, что аргумент копирующего конструктора записан с модификатором **const**, указывающего компилятору, что передаваемый по ссылке объект не будет изменяться в конструкторе. Это привело к необходимости добавить модификатор **const** для аргументов функций, вызываемых в копирующем конструкторе. Применение модификатора **const** в копирующем конструкторе не является обязательным, но эта деталь является отличительной чертой профессионального разработчика.

```
#include "stdafx.h"
#include <cstring>    //  strcpy_s
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
   Person *pp1 = new Person(),  // pp1 - указатель на бъект типа Person
           p2("Kuprynova","Katy","Brest");   // p2 - объект типа Person
   char   *Line1 = new char[pp1->MaxSizeLine()],
          *Line2 = new char[p2.MaxSizeLine()];
   pp1->setFirstName("Petrovich");
   pp1->setLastName("Evgeny");
   pp1->setAddress("Minsk");
   strcpy_s(Line1,
           pp1->MaxSizeLine(),
           pp1->getLine(new char[pp1->MaxSizeLine()]));
   strcpy_s(Line2,
           p2.MaxSizeLine(),
           p2.getLine(new char[p2.MaxSizeLine()]));
   return 0;
}
```

Рис. 1.9. Применение конструктора с аргументами

```
class Person
{
// .............................................
public:
    void setFirstName(const char fn[])
            {strcpy_s(FirstName, sizeof(FirstName), fn);};
    void setLastName(const char ln[])
            {strcpy_s(LastName, sizeof(LastName), ln);};
    void setAddress(const char as[])
            {strcpy_s(Address, sizeof(Address), as);};
    Person(){FirstName[0]= LastName[0]= Address[0] = '\x00';};
    Person(char fn[], char ln[], char as[])
           {
            setFirstName(fn);
            setLastName(ln);
            setAddress(as);
           };
    Person (const Person& p) // копирующий конструктор
           {
            setFirstName(p.FirstName);
            setLastName(p.LastName);
            setAddress(p.Address);
           };
// .............................................
};
```

Рис. 1.10. Фрагмент класса, использующего копирующий конструктор

На рис. 1.11 приведен пример программы, использующей копирующий конструктор для создания копии объекта.

```
#include "stdafx.h"
#include <cstring>          // strcpy_s
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
   Person *pp1 = new Person("Kuprynova","Katy","Brest");
   Person p2(*pp1);         // p2 - объект ипа Person (клон *pp1)
   char *Line1 = new char[pp1->MaxSizeLine()],
        *Line2 = new char[p2.MaxSizeLine()];
   strcpy_s(Line1,
            pp1->MaxSizeLine(),
            pp1->getLine(new char[pp1->MaxSizeLine()]));
   strcpy_s(Line2,
            p2.MaxSizeLine(),
            p2.getLine(new char[p2.MaxSizeLine()]));

   return 0;
}
```

Рис. 1.11. Применение копирующего конструктора

В программе на рис. 1.11 создается два одинаковых объекта класса **Person**. В первом случае при динамическом создании объекта

с помощью оператора **new** используется конструктор с аргументами для инициализации свойств создаваемого объекта. Во втором случае осуществляется объявление объекта с применением копирующего конструктора. В качестве аргумента этому конструктору передается первый объект, который является образцом. Если установить точку останова программы отладчика на операторе **return** и исследовать содержимое памяти двух объектов с помощью watch-окна, то результат будет, как на рис. 1.12.

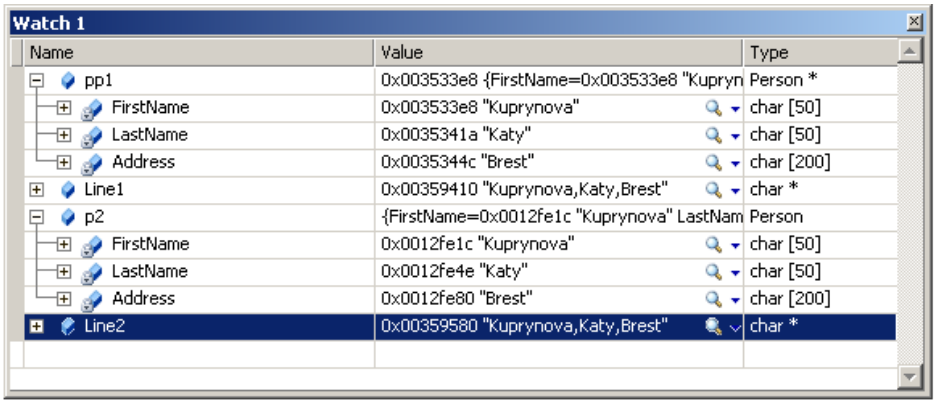

Рис. 1.12. Области памяти, занимаемые объектами-клонами

1.5. Деструктор класса

Деструктор – это функция-член класса, имя которой имеет вид **~ClassName**, где **ClassName** – имя класса. Эта функция не может иметь аргументов и не может возвращать значение. Деструктор автоматически вызывается при уничтожении объекта. Если деструктор не объявленв классе, то компилятор автоматически создает его автоматически.

Обычно деструктор используется для освобождения ресурсов, используемых уничтожаемым объектом. В качестве ресурса могут выступать динамически запрошенная объектом память, открытые файлы, сетевые соединения, подключения к базе данных и т. п.

На рис. 1.13 и 1.14 приведен пример класса **Person**, содержащего деструктор, предназначенный для освобождения памяти, динамически полученной методом **setAutoBy**.

```
#define MAXSIZELINE
(sizeof(FirstName)+sizeof(LastName)+sizeof(Address)+2)
class Person
{
  private:
         char FirstName[50];
         char LastName[50];
         char Address[200];
         char *AutoBy;
         int  SizeAutoBy;
         char* toLine(char* l);
  public:
         void setFirstName(const char fn[]);
         void setLastName(const char ln[]);
         void setAddress(const char as[]);
         void setAutoBy(int size_aby = 0, const void* aby = (void*)0);
         Person();
         Person(char fn[], char ln[], char as[]);
         Person (const Person& p);
         ~Person ();
         char* getLine (char *l){return toLine(l);};
         int MaxSizeLine(){return MAXSIZELINE;};
};
```

Рис. 1.13. Класс Person с явно объявленным деструктором

Обратите внимание, что описание класса **Person** теперь разделено на две части. В первой содержится собственно описание класса, а во второй – исполняемый код. При профессиональной разработке классов поступают именно так, руководствуясь принципом: реализация членов-функций должна быть скрыта от пользователя класса. Имя файла, в котором находится код функций-членов, обычно совпадает с именем класса и имеет расширение **.cpp**. Поэтому описательная часть класса, содержащая описание атрибутов и прототипы функций-членов, в данном примере содержится, как и прежде, в файле **Person.h**, а реализация функций-членов – в файле **Person.cpp**.

```
#include "stdafx.h"
#include <cstring>
#include "Person.h"
char* Person::toLine()
   {
    //.....................................
   };
void Person::setFirstName(const char fn[])
   {
    //.....................................
   };
void Person::setLastName(const char ln[])
   {
    //.....................................
   };
void Person::setAddress(const char as[])
   {
    //.....................................
   };
void Person::setAutoBy(int size_aby, const void* aby )
   {
     if (size_aby != 0 && aby != (void*)0)
     {
      AutoBy = new char[SizeAutoBy = size_aby];
      memcpy(AutoBy, aby, size_aby);
     }
     else
     {
       SizeAutoBy = 0;
       AutoBy = (char*)0;
     }
   };
Person::Person()
   {
    //.....................................
    setAutoBy();
   };
Person::Person(char fn[], char ln[], char as[])
   {
    //.....................................
    setAutoBy();
   };
Person::Person (const Person& p)
  {
   //.....................................
   setAutoBy(p.SizeAutoBy, p.AutoBy);
  };
Person::~Person ()
  {
    if (SizeAutoBy > 0) delete[] AutoBy;
  };
#undef MAXSIZELINE
```

Рис. 1.14. Члены-функции класса Person,
расположенные в отдельном файле Person.cpp

Код членов-функций класса Person на рис. 1.14 приведен не полностью. Здесь отражены лишь необходимые для данного примера изменения, а код, не претерпевший изменений по отношению к предшествующему примеру, обозначен комментариями с многоточием.

На рис. 1.15 приведен текст программы, использующей класс **Person**. В программе вызывается метод **setAutoBy** объекта класса **Person**. Для хранения данных (например, автобиографических сведений), передаваемых во втором параметре, метод **setAutoBy** динамически выделяет память размера, указанного первым параметром с помощью оператора **new** (рис. 1.14).

```
#include "stdafx.h"
#include <iostream>
#include <cstring>    //  strcpy_s
#include <locale.h>
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
   setlocale(LC_ALL, "rus");
   Person *pp1 = new Person("Куприянова",
                            "Екатерина",
                           "Брест"); // pp1 - указатель на бъект Person
   char autoby[] = "Дата рождения: 01.01.1988, "
                   "Место рождения: Брест,"
                "Образование: высшее";
   pp1->setAutoBy(sizeof(autoby), autoby);
   delete pp1;
  return 0;
}
```

Рис. 1.15. Использование класса Person

Вызов деструктора **~Person** объекта (рис. 1.14) будет осуществлен при выполнении оператора **delete**, удаляющего этот объект. Деструктор проверяет количество выделенной памяти и, если оно превышает нулевое значение, то освобождает ее до удаления самого объекта.

Следует отметить, что если бы оператора **delete** не было в программе на рис. 1.15, вызов деструктора все равно бы состоялся, но инициировал бы этот вызов оператор **return**, который приводит к удалению всех созданных в приложении объектов.

1.6. Ссылка на себя

В любом классе **X** неявно определен указатель с именем **this** и имеющий тип **X*** (указатель на объект класса **X**). В процессе создания объекта **x** класса **X** значение **this** автоматически инициализируется значением **&x** (адрес объекта **x**). Указатель **this** может быть использован в любой функции-члене класса, но **this** не является обычной пе-

ременной. Попытка получить ее адрес или присвоить ей какое-нибудь значение приведет к ошибке компиляции.

В предыдущих примерах при использовании данных-членов класса в функциях-членах их имена не квалифицировались точно. Это иногда может привести к двусмысленности. На рис. 1.16 демонстрируется пример применения указателя **this**.

```
//  ----- Person.h -----
class Person
{
  private:
          char FirstName[50];
// ........................................
  public:
          void setFirstName(const char fn[]);
// ........................................
};

// ---- Person.cpp -----
// ........................................
void Person::setFirstName(const char fn[])
{
   char FirstName[sizeof(this->FirstName)] = "<firstname id= \"";

   strcpy_s(FirstName+strlen(FirstName),
            sizeof(this->FirstName)- strlen(FirstName),
            fn);
   strcpy_s(FirstName+strlen(FirstName),
            sizeof(this->FirstName)- strlen(FirstName),
            "\" />");
   strcpy_s(this->FirstName, sizeof(this->FirstName), FirstName);
};
// ........................................
```

Рис. 1.16. Применение указателя this

Обратите внимание, что объявление в функции **setFirstName** переменной с именем **FirstName** «закрывает» область видимости атрибут класса **FirstName**. Применение указателя **this** в примере на рис. 1.16 позволяет точно квалифицировать атрибут класса **FirstName**, что дает возможность разрешить проблему.

Применение указателя **this** дает возможность использовать интересный прием программирования, позволяющий применять цепочки функций-членов для выполнения вычислений.

На рис. 1.17 и 1.18 демонстрируется пример построения цепочки функций для вычисления заработной платы, которая складывается из базовой суммы, персональной надбавки, надбавки за выслугу и премии. Все дополнительные начисления вычисляются как процент от некоторой суммы. Персональная надбавка считается относи-

тельно базовой суммы, надбавка за выслугу относительно суммы базовой величины и персональной надбавки и, наконец, премиальные вычисляются от общей суммы. Функции **setBaseSalary** и **setPercentSalary** используются для установки значений базовой заработной платы (атрибут **BaseSalary**) и дополнительных начислений в процентом выражении (атрибуты **PercentAdd**, **PercentLength** и **PercentBonus**).

```
class Person
{
  private:
   // ........................................
      int BaseSalary,                    // базовая зарплата
          SumSalary ;                    // общая сумма зарплаты
      int PercentAdd,                    // процент надбавки
          PercentLength,                 // процент за выслугу
          PercentBonus;                  // процент премиальных

      Person& SalaryBase();              // =  базовая зарплата
      Person& SalaryAdd();               // + %-надбавки
      Person& SalaryLength();            // + %-выслуга
      Person& SalaryBonus();             // + %-премия
  // ........................................

  public:
   // ........................................

      void setBaseSalary(int base = 0);  // установить базовую зарплату
      void setPercentSalary(int add = 0, // установить проценты
                            int length = 0,
                            int bonus = 0);
      int getSalary();                   // вычислить зарплату

   // ........................................
};
```

Рис. 1.17. Класс Person с дополнительными функциями для вычисления заработной платы

Обратите внимание на возвращаемый тип функций-членов **SalaryBase**, **SalaryAdd**, **SalaryLength** и **SalaryBonus** – это ссылка на объект класса **Person**. Именно такой возвращаемый тип позволяет построить цепочку функций, как это представлено на рис. 1.18 в функции **getSalary**. Обратите внимание, что каждое последующее увеличение суммы заработной платы в цепочке осуществляется в процентах относительно суммы, вычисленной раньше (левее). Каждая из функций в цепочке «ничего не знает» о цепочке для вычислений, а изменение порядка функций в цепочке приведет к изменению базы для начисления надбавок и премиальных.

```
//.........................
Person::Person()
{
  //.........................
  setPercentSalary();
  setBaseSalary();
 };
//... аналогичное дополнение необходимо внести во все конструкторы класса

void Person::setPercentSalary(int add, int length, int bonus)
{
     this->PercentAdd    = add;
     this->PercentLength = length;
     this->PercentBonus  =  bonus;
};
Person& Person::SalaryBase()
{
     this->SumSalary = this->BaseSalary;
     return *this;
};
Person& Person::SalaryAdd()
{
     this->SumSalary += this->SumSalary * this->PercentAdd/100;
     return *this;
};
Person& Person::SalaryLength()
{
     this->SumSalary += this->SumSalary * this->PercentLength/100;
     return *this;
};
Person& Person::SalaryBonus()
{
     this->SumSalary += this->SumSalary * this->PercentBonus/100;
     return *this;
};
int  Person::getSalary()
{
    return
SalaryBase().SalaryAdd().SalaryLength().SalaryBonus().SumSalary;

};
```

Рис. 1.18. Пример построения цепочек функций для вычислений

Запись вычислений в виде цепочек функций позволяет легко перестраивать эти цепочки и в случае необходимости строить новые. Обратите внимание на окончание цепочки – там находится атрибут, который использовался для хранения промежуточных вычислений и конечного результата.

1.7. Статические члены класса

Довольно часто при разработке приложений появляется необходимость иметь общие данные для всех объектов одного класса. В этом случае применяют статические данные-члены. Для объявления статических членов класса используется спецификатор **static**.

Объявление статических данных-членов только в классе не приводит к реальному резервированию памяти, как это происходит в случае с нестатическими данными-членами, а попытка сразу использовать статические атрибуты приведет к ошибке компоновщика, который будет пытаться разрешить внешнюю ссылку (именно так представляет компилятор статические данные). Для реального выделения и инициализации памяти следует сделать еще одно объявление данных отдельно, вне класса, придерживаясь при этом правила одного определения, т. е. такое дополнительное определение должно быть единственным в программе. Обычно это дополнительное определение статических данных делают в cpp-файле данного класса.

Продолжим рассматривать пример, представленный на рис. 1.17 и 1.18. Предположим, что величина базовой зарплаты для всех субъектов, описанных классом **Person**, одинакова, различны только величины надбавок. На рис. 1.19 представлены фрагменты файлов **Person.h** и **Person.cpp**, в которых демонстрируется применение статического члена **BaseSalary** класса **Person**.

Обратите внимание, что объявление статической переменной осуществляется сначала в рамках класса со спецификатором **static** и потом в cpp-файле без спецификатора, но с применением оператора **::** (два двоеточия), указывающего имя класса, которому принадлежит переменная.

Так как статическая переменная **BaseSalary** является общей для всех объектов класса **Person**, то инициализировать эту переменную теперь следует один раз в момент выделения реальной памяти и независимо от создания объектов. Если выполнять инициализацию, как раньше, в конструкторах класса, то это приведет к непредсказуемому результату: каждый вновь созданный объект будет устанавливать свое значение общей для всех объектов переменной **BaseSalary** (в нашем случае это нуль).

На рис. 1.20 приведен пример программы, использующей два объекта класса **Person**, а на рис. 1.21 – результат ее выполнения.

Обратите внимание, что свойство **BaseSalary** устанавливается методом объекта ***pp1**, но после этого значение **BaseSalary** объекта **p2** становится таким же.

```
// --  файл Person.h
//.............................................
class Person
{
  private:
//.............................................

        static int BaseSalary;        // базовая зарплата
        int SumSalary ;               // общая сумма зарплаты
        int PercentAdd,               // процент надбавки
        PercentLength,                // процент за выслугу
        PercentBonus;                 // процент премиальных
//.............................................
};
//-----------------------------------------------------------------
// --   файл  Person.cpp
#include "stdafx.h"
#include <cstring>
#include "Person.h"
int Person::BaseSalary = 0;   // выделение памяти и инициализация static
//.............................................
Person::Person()
{
  FirstName[0]= LastName[0]= Address[0] = '\x00';
  setAutoBy();
  setPercentSalary();  // добавить во все кострукторы,в копирующий с парам.
  // необходимо удалить во всех конструкторах → setBaseSalary();
};
//.............................................
```

Рис. 1.19. Пример применения статического атрибута класса

```
#include "stdafx.h"
#include <iostream>
#include <cstring>    //  strcpy_s
#include <locale.h>
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
setlocale(LC_ALL, "rus");

Person *pp1 = new Person("Куприянова","Екатерина","Брест");
pp1->setBaseSalary(1000);
std::cout<< "Зарплата:$" << pp1->getSalary()<< std::endl;
//Зарплата:$1000

Person p2;
std::cout<< "Зарплата:$" << p2.getSalary() << std::endl;
//Зарплата:$1000

delete pp1;
return 0;
}
```

Рис. 1.20. Пример программы, использующей два объекта класса Person

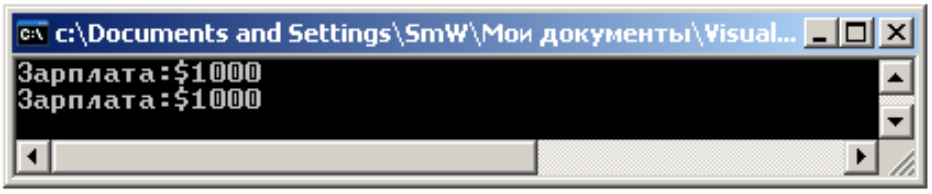

Рис. 1.21. Результат выполнения программы на рис. 1.20

В примере на рис. 1.19–1.21 явно виден существенный недостаток класса **Person**. Действительно, пользователь, применяя метод **setBaseSalary** объекта ***pp1**, интуитивно ожидает, что этот метод устанавливает величину базовой зарплаты для объекта ***pp1**, а не для всех объектов сразу. Такое поведение метода **setBaseSalary** выглядело логично, если бы и он был общим для всех объектов. Для этого необходимо объявить метод **setBaseSalary** статическим.

На рис. 1.22 приведены фрагменты h- и cpp-файлов с описанием класса **Person**, использующего статический метод **setBaseSalary**.

```
// --  файл Person.h
//.................................................
class Person
{
  private:
//.................................................
      static int BaseSalary;            // базовая зарплата
//.................................................
  public:
//.................................................
     static void setBaseSalary(int base = 0);
//.................................................
};
//-------------------------------------------------------------------

// --   файл  Person.cpp
//.................................................
int Person::BaseSalary = 0;   // выделение памяти и инициализация static
//.................................................
void Person::setBaseSalary(int base)
{
   // заменить → this->BaseSalary = base;
   Person::BaseSalary = base;
};
//.................................................
```

Рис 1.22. Пример описания статической функции класса

Для того чтобы функцию-член сделать статической, необходимо указать спецификатор **static** при ее объявлении. Кроме того, статическая функция может работать только со статическими данными класса.

Поэтому ключевое слово **this**, которое использовалось раньше для атрибута **BaseSalary**, вызовет ошибку компиляции. Для доступа к статическим членам следует использовать оператор **::**, как это продемонстрировано в примере на рис. 1.22.

На рис. 1.23 приведен пример программы, использующей статическую функцию-член **setBaseSalary** для установки значения базовой заработной платы для всех объектов класса **Person**.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale.h>
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
setlocale(LC_ALL, "rus");

Person::setBaseSalary(1000);

Person *pp1 = new Person("Куприянова","Екатерина","Брест");
std::cout<< "Зарплата:$" << pp1->getSalary() << std::endl;
//Зарплата:$1000

Person p2;
std::cout<< "Зарплата:$" << p2.getSalary() << std::endl;
//Зарплата:$1000

delete pp1;
return 0;
}
```

Рис. 1.23. Пример применения статической функции класса

Обратите внимание, что статический метод можно применять до создания объектов и для его вызова следует использовать оператор **::** с именем класса.

1.8. Константные члены класса

Практически любой объект содержит в себе данные, которые не должны меняться в процессе его использования. Значение их заранее предопределено разработчиком класса или устанавливается один раз в конструкторе при инициализации объекта. Для объявления таких данных в классе используется квалификатор **const**. Значение атрибутов с таким квалификатором могут быть только проинициализированы,

а попытка изменения установленного при инициализации значения вызовет ошибку компиляции.

На рис. 1.24 представлен фрагмент описания класса **Person**, содержащего константные атрибуты трех типов: статические константы (**SIZE_FIRSTNAME**, **SIZE_LASTNAME**, **SIZE_ADDRESS** и **SIZE_LINE**), нестатическая константа (**Order**) и константы перечисления (**MALE**, **FEMALE** и **NILL**).

```
// --  файл Person.h
//.............................................
class Person
{
  public:
       const static int SIZE_FIRSTNAME  = 50;
       const static int SIZE_LASTNAME   = 50;
       const static int SIZE_ADDRESS    = 200;
       const static int SIZE_LINE       = SIZE_FIRST_NAME
                                      +SIZE_LAST_NAME
                                  +SIZE_ADDRESS+2;
       const int Order;                  // номер созданного объекта
       enum SEX {MALE, FEMALE, NILL};    // тип пола

  private:
       char FirstName[SIZE_FIRSTNAME];  // фамилия
       char LastName[SIZE_LASTNAME];     // имя
       SEX  Sex;                         // пол
       char Address[SIZE_ADDRESS];       // адрес
//.............................................
       static int Number;                // счетчик объектов
       static int Count;                 // текущее количество объектов
//.............................................

  public:
//.............................................
       int MaxSizeLine() {return SIZE_LINE;};
       static int getCount(){return Count;};
//.............................................

};
```

Рис. 1.24. Объявление константных атрибутов в классе

Статические константы не требуют дополнительного объявления, как это было бы необходимо для обычных статических атрибутов, и должны инициализироваться в классе сразу при объявлении. По своим свойствам статические константы напоминают препроцессорные константы, определяемые с помощью директивы **#define**, в рамках объекта для них не резервируется память. Обратите внимание, что значения этих констант потом используется при объявлении размерности атрибутов. Такой прием программирования позволяет

существенно сократить затраты времени на изменение размерности атрибутов в случае возникновения такой необходимости.

Для нестатических констант в отличие от статических выделяется реальная память в рамках каждого объекта класса. Инициализировать эти атрибуты можно только в конструкторе и только специальным образом. На рис. 1.25 приведен пример конструктора, в котором инициализируется константный атрибут **Order**. Обратите внимание, что инициализация осуществляется в заголовочной строке конструктора, оператор инициализации отделяется от заголовка функции двоеточием, а значение инициализации указывается в скобках.

```
// --  файл Person.cpp
#include "stdafx.h"
#include <cstring>
#include "Person.h"
int Person::BaseSalary = 0;
int Person::Number = 0;
int Person::Count = 0;
//.................................................
void Person::setSex(SEX mf)
{
    this->Sex = mf;
};
//.................................................
Person::Person()  :Order(++Number)
{
    ++Person::Count;
    //........Аналогично необходимо сделать во
    //........всех кострукторах
    setSex(SEX::NILL);
 };
Person::~Person ()
{
    --Person::Count;
    if (SizeAutoBy > 0) delete[] AutoBy;
};
//.................................................
```

Рис. 1.25. Пример инициализации нестатических константных атрибутов класса

Константы перечисления объявляются с помощью оператора **enum**. В примере на рис. 1.24 с помощью перечисления объявлены три константы. По своим функциональным возможностям они ничем не отличаются от статических констант, но обладают одним дополнительным свойством – их можно использовать для инициализации атрибутов этого типа перечисления. На рис. 1.24 атрибут **Sex** имеет тип

перечисления **SEX**. Как и для обычного перечисления, компилятор будет контролировать корректность присвоения этому атрибуту значений.

На рис. 1.26 приведена программа, использующая константные атрибуты класса. Обратите внимание на способ доступа к константам класса.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale.h>
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
   setlocale(LC_ALL, "rus");

   char fn[Person::SIZE_FIRSTNAME]="Чечко";
   char ln[Person::SIZE_LASTNAME]="Анжела";
   char as[Person::SIZE_ADDRESS]="Минск, ул. Свердлова, 13";
   char line[Person::SIZE_LINE];

   Person::setBaseSalary(1000);

   Person *pp1 = new Person(fn,ln,as);
   pp1->setSex(Person::SEX::FEMALE);

   Person p2(*pp1),
         *pp3 = new Person("Шамашова", "Анастасия", "Гомель"),
          p4;

   delete pp3;

   std::cout<< "Счетчик = " << Person::getCount()<< std::endl;
   std::cout<< "*pp1:Order = " << pp1->Order << std::endl
            << "  p2:Order = " << p2.Order   << std::endl
            << "  p2:Order = " << p4.Order   << std::endl;

   delete pp1;
   return 0;
}
```

Рис. 1.26. Пример программы, использующей константные атрибуты класса

Все создаваемые объекты в примере на рис. 1.25 нумеруются в порядке их создания, начиная с первого. Для хранения номера каждого объекта используется константный атрибут **Order**. Учет общего количества созданных объектов осуществляется в статическом атрибуте **Number**. Для подсчета активных (созданных и неудаленных) объектов класса используется статический атрибут **Count**, значение которого увеличивается при каждом создании объекта (в конструкторах) и уменьшается при каждом удалении (в деструкторе). На рис. 1.27 представлен результат работы программы на рис. 1.26.

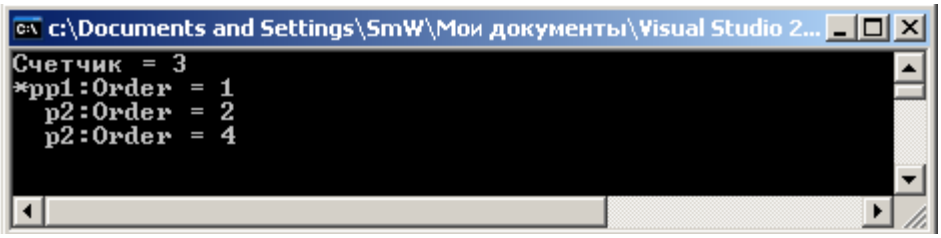

Рис. 1.27. Результат выполнения программы,
представленной на рис. 1.26

```
// --  файл Person.h
//.................................................
class Person
{
  public:
        const static int SIZE_FIRST_NAME = 50;
        const static int SIZE_LAST_NAME  = 50;
        const static int SIZE_ADDRESS    = 200;
        const static int SIZE_LINE       = SIZE_FIRST_NAME
                                          +SIZE_LAST_NAME
                                    +SIZE_ADDRESS+2;
 //.................................................
        const int Order;
        enum SEX {MALE, FEMALE, NILL};
 //.................................................
 private:
 //.................................................
        char* toLine(char *l) const;
 //.................................................
 public:
 //.................................................
        char* getLine (char *l) const {return toLine(l);} ;
        int MaxSizeLine()const{return SIZE_LINE;} ;
 //.................................................
};

// --  файл Person.cpp
//.................................................
char* Person::toLine(char *l) const
{     int n = 0;
      strcpy_s(l, sizeof(FirstName), this->FirstName);
      strcpy_s(l + (n += strlen(this->FirstName)), 2, ",");
      strcpy_s(l + (n += 1),sizeof(LastName), this->LastName);
      strcpy_s(l + (n += strlen(this->LastName)), 2, ",");
      strcpy_s(l + (n += 1), sizeof(Address),this->Address);
    return l;
};
//.................................................
```

Рис. 1.28. Применение константных функций-членов

Помимо константных данных класс может содержать константные функции-члены (рис. 1.28). Константные функции-члены – это такие функции класса, которые не изменяют внутреннее состояние

(атрибуты класса) объекта класса. Для их объявления тоже используется квалификатор **const**, но записывается он в заголовке или прототипе функции после формальных параметров, как на рис. 1.28.

В примере на рис. 1.28 объявлены три константные функции: **toLine**, **getLine** и **MaxSizeLine**. Первая функция является закрытой, две другие – открытыми. Константная функция не может вызывать функцию, которая может изменить состояние объекта. Если, например, из определения функции **toLine** убрать квалификатор **const**, то это приведет к ошибке компиляции.

Заметим, что функция **MaxSizeLine**, в принципе, является лишней, т. к. получить максимальную длину строки для функции **getLine** можно через отрытое свойство **SIZE_LINE**. При разработке классов ситуация, когда в классе существует несколько методов или свойств, обладающих одинаковой функциональностью, возникает достаточно часто. Дело в том, что процесс разработки, использования и совершенствования класса может длиться долго, и для совместимости с предыдущими его версиями разработчики стараются сохранить старый интерфейс, ведь он может быть уже задействован во многих программах. Такой подход позволяет свести установку новой версии класса к простой перекомпиляции.

Глава 2. ИНКАПСУЛЯЦИЯ

2.1. Основные понятия

Инкапсуляция (encapsulation) **–** свойство языка программирования, позволяющее объединить данные и код и скрыть реализацию объекта от пользователя. Пользователю предоставляется только интерфейс класса, с помощью которого он может взаимодействовать с объектом.

Сокрытие деталей реализации класса обусловлено главным образом тем, что при разработке классов заранее предполагается, что их реализация будет в дальнейшем претерпевать изменения, которые могут вынудить пользователей переписывать код своих программ. В хорошо инкапсулированном классе интерфейс практически не зависит от реализации, поэтому изменение реализации не потребует изменения пользовательских программ или сведет этот процесс к минимуму.

Вторая цель, которая достигается инкапсуляцией, – это абстракция данных. ***Абстракция данных*** – это подход к обработке данных по принципу «черного ящика», который в нашем случае заключается в том, что пользователь не должен задумываться о внутренних свойствах объектов, но должен иметь возможность манипулировать объектами в соответствии с их функциональным предназначением. В идеальном случае используемый пользователем объект должен обладать свойствами, похожими на переменные простых встроенных типов C++, таких как, например, **int** и **char**.

Неявно инкапсуляция уже демонстрировалась в примерах первой главы. Применение модификаторов доступа, статических и константных членов, разделение текста класса на две части (h-файл и cpp-файл) позволяют создавать объекты, обладающие таким свойством.

Другой стороной инкапсуляции и абстракции объектов является необходимость тщательно продумывать интерфейс класса, т. к. он должен обладать свойством ***полноты***. Говорят, что интерфейс является полным, если пользователь обладает достаточным набором открытых свойств и методов объекта для удобной работы с ним.

Следует сказать, что для достижения сокрытия реализации и абстракции объектов применяют еще один механизм объектно-ориентированного программирования – наследование классов, который будет рассматриваться в следующей главе.

Разработанный на C++ класс обычно поставляется пользователю в виде двух файлов: заголовочного файла (файла с расширением **h**) и библиотеки. Библиотека содержит откомпилированные функции, которые раньше находились в cpp-файле класса. Она может быть статической (расширение **lib**) или динамически загружаемой (расширение **dll**). Вместе с динамически загружаемой библиотекой обычно поставляется специальная библиотека экспорта функций. Информация пользователя о применяемом классе ограничена h-файлом, который, по сути, является спецификацией интерфейса объектов этого класса.

Самый простой способ построить библиотеку функций – использовать соответствующий визард (от англ. wizard – волшебник) Microsoft Visual Studio. Процесс создания статических и динамических библиотек функций C++ подробно описывается в источнике [6].

2.2. Вложенные объекты и классы

Вложенным объектом будем назвать объект некоторого класса, объявленного в качестве атрибута другого класса.

Вложенным классом называется класс, определенный внутри другого класса.

Существует два подхода относительно применения таких классов. Некоторые считают, что класс должен быть компонентом, максимально изолированным от других классов, и применяют вложенные объекты. Другие допускают применение вложенных классов, если это согласуется с его логикой. В этом разделе буду рассматриваться оба случая.

В примерах предыдущей главы использовался класс **Person**, последняя версия которого предназначалась для хранения информации о некотором индивидууме (фамилия, имя, пол, адрес, величина заработной платы). Несколько усовершенствуем атрибут с именем **Address**, превратив его во вложенный объект класса **Address** с именем **address**. На рис. 2.1 предлагается пример описания класса **Address** (файл **Address.h**).

Для хранения основной информации в классе **Address** используются 4 атрибута: **Country**, **Postcode**, **Region**, **Info1** и **Info2**. Так же, как и в примерах предыдущей главы, здесь применяются статические константы для хранения размерностей атрибутов класса. С помощью перечисления **GET** объявлены четыре константы, имена которых совпадают с именами атрибутов. Класс имеет три конструктора (по умолчанию, с аргументами и копирующий) и деструктор. Кроме того, в классе объявлено три открытых функции: **setAddress** (установка значений

атрибутов), **get** (чтение значений атрибутов) и **toXMLLine** (формирование xml-строки). Последняя функция является константной.

```
#pragma once    // исключть повтороный include
class Address
{
 public:
 const static int SIZE_COUNTRY  = 3;
 const static int SIZE_POSTCODE = 7;
 const static int SIZE_REGION   = 11;
 const static int SIZE_INFO     = 101;
 const static int SIZE_LINE     = SIZE_COUNTRY + SIZE_POSTCODE
                                  + SIZE_REGION + 2*SIZE_INFO;
 const static int SIZE_XMLLINE  = SIZE_LINE+101;

 enum GET{COUNTRY, POSTCODE, REGION, INFO1, INFO2};
private:
 char Country[SIZE_COUNTRY];    //  ISO-код
 char Postcode[SIZE_POSTCODE];  //  почтовый индекс
 char Region[SIZE_REGION];      //  область
 char Info1[SIZE_INFO];         //  произвольная информация
 char Info2[SIZE_INFO];         //  произвольная информация
 void xml(char* l,              //  вспомогательная функция
          const char s[], int lv,
          const char v[], const char e[])const;
public:
 Address(void){set();};
 Address(char Country[SIZE_COUNTRY],
        char Postcode[SIZE_POSTCODE],
          char Region[SIZE_REGION],
          char Info1[SIZE_INFO],
        char Info2[SIZE_INFO])
      {set(Country, Postcode,Region, Info1, Info2);};
 Address(const Address& as)
      {set(as.Country, as.Postcode,as.Region, as.Info1, as.Info2);};
 ~Address(void) {};
 void set(const char Country[SIZE_COUNTRY] = "",
         const char Postcode[SIZE_POSTCODE] = "",
          const char Region[SIZE_REGION] = "",
          const char Info1[SIZE_INFO] = "",
         const char Info2[SIZE_INFO] = "");
 char* get(GET t, char*);
 char* toXMLLine(char l[SIZE_XMLLINE]) const;  //  XML-строка адреса
};
```

Рис. 2.1. Пример класса для хранения информации об адресе

На рис. 2.2 приведен пример файла **Address.cpp**, содержащего реализацию функций-членов класса **Address**.

Обратите внимание, что в функции **toXMLLine** вызывается вспомогательная функция с именем **xml**. Вызывающая функция **toXMLLine** является константной, т. е. компилятор должен исключить любые операции в этой функции, которые могут изменить состояние класса. При вызове функции **xml** в четвертом параметре передается

адрес атрибута класса. Если бы при описании этого параметра функции **xml** не был установлен квалификатор **const**, компилятор выдал бы ошибку, полагая, что отсутствие квалификатора **const** предполагает возможность изменения атрибута в функции **xml**.

```
#include "StdAfx.h"
#include <cstring>
#include "Address.h"
void Address::set (const char Country[SIZE_COUNTRY],
                   const char Postcode[SIZE_POSTCODE],
                   const char Region[SIZE_REGION],
                   const char Info1[SIZE_INFO],
                   const char Info2[SIZE_INFO])
{
  strcpy_s(this->Country, SIZE_COUNTRY, Country);
  strcpy_s(this->Postcode,SIZE_POSTCODE,Postcode);
  strcpy_s(this->Region,SIZE_REGION,Region);
  strcpy_s(this->Info1,SIZE_INFO,Info1);
  strcpy_s(this->Info2,SIZE_INFO,Info2);
}
char* Address::get(GET t, char* b)
switch(t)
 {
 case COUNTRY  :strcpy_s(b, SIZE_COUNTRY,  this->Country);  break;
 case POSTCODE :strcpy_s(b, SIZE_POSTCODE, this->Postcode); break;
 case REGION   :strcpy_s(b, SIZE_REGION,   this->Region);   break;
 case INFO1    :strcpy_s(b, SIZE_INFO,     this->Info1);    break;
 case INFO2    :strcpy_s(b, SIZE_INFO,     this->Info2);    break;
 default       :strcpy_s(b,1, '\0');
 }
return b;
};
void Person::xml(char* l,
                   const char s[], int lv,
                   const char v[], const char e[]) const
{
strcpy_s(l+strlen(l), strlen(s)+1, s);
strcpy_s(l+strlen(l), lv, v);
strcpy_s(l+strlen(l), strlen(e)+1, e);
}
char* Address::toXMLLine(char l[SIZE_XMLLINE]) const // XML-строка адреса
{
 strcpy_s(l,10,"<address>");
 xml(l,"<country>",SIZE_COUNTRY,this->Country,"</country>" );
 xml(l,"<postcode>",SIZE_POSTCODE,this->Postcode,"</postcode>" );
 xml(l,"<region>",SIZE_POSTCODE,this->Region,"</region>" );
 xml(l,"<info1>",SIZE_INFO,this->Info1,"</info1>" );
 xml(l,"<info2>",SIZE_INFO,this->Info2,"</info2>" );
 strcpy_s(l+strlen(l),11,"</address>");
return l;
}
```

Рис. 2.2. Реализация функций-членов класса Address

На рис. 2.3 приведена программа, использующая класс **Address**, а на рис. 2.4 – результат ее выполнения.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale.h>
#include "Address.h"

int _tmain(int argc, _TCHAR* argv[])
{
   setlocale(LC_ALL, "rus");
   Address a1("BY","222000","Минск","ул. Сверлова 13", "");
   Address a2(a1);

   std::cout<< "XML = "
            << a1.toXMLLine(new char[Address::SIZE_XMLLINE])
            <<std::endl;
   std::cout<< "XML = "
            << a2.toXMLLine(new char[Address::SIZE_XMLLINE])
            <<std::endl;

   std::cout<< "Country = "
            << a1.get(Address::COUNTRY, new char[Address::SIZE_COUNTRY])
            <<std::endl;
   std::cout<< "Region = "
             << a2.get(Address::REGION,  new char[Address::SIZE_REGION])
            <<std::endl;
  return 0;
}
```

Рис. 2.3. Пример программы, использующей класс Address

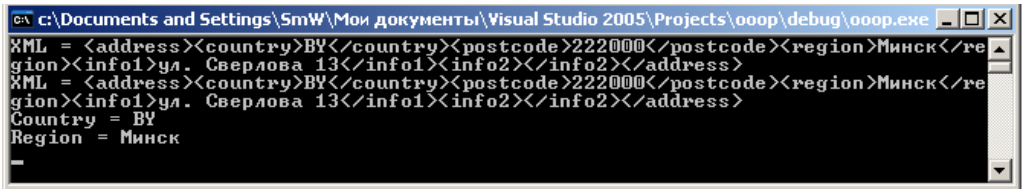

Рис. 2.4. Результат выполнения программы на рис. 2.3

Внесем изменения в класс **Person**, разработанный в примерах предыдущей главы: изменим тип атрибута **Address**, изменим имя атрибута на **address** и проведем соответствующие изменения в функциях-членах. При этом постараемся сделать так, чтобы для приложений, уже использующих данный класс, не требовалось переписывать текст программ, а все свелось к простой замене файлов и перекомпиляции. Иначе говорят, что новая версия класса обеспечивает совместимость со старой по принципу «сверху вниз», т. е. все старые интерфейсы класса (нижняя версия) в новой версии (верхней) останутся работоспособными.

На рис. 2.5 приведен фрагмент класса **Person**, использующего объект класса **Address**. Как всегда, в листинге программы приводятся лишь те фрагменты, которые претерпели изменения. Обратите

внимание, что все открытые члены-данные (кроме атрибута a**ddress**) и все прототипы членов функций остались без изменений.

```
// --  файл Person.h
#pragma once
#include "Address.h"
class Person
{
 public
 //.................................................
    const static int SIZE_XMLLINE   = SIZE_LINE +101;
    Address  address;                    // адрес
 //.................................................

 private:
 //.................................................
    void xml(char* l,                    // вспомогательная функция
           const char s[], int lv,
           const char v[], const char e[])const;
 //.................................................
public:
 //.................................................
char* toXMLLine(char l[SIZE_XMLLINE]) const;  //  XML-строка
}
```

Рис. 2.5. Фрагмент класса Person, использующего объект класса Address

В класс **Person** добавлены: открытый константный статический атрибут **SIZE_XMLLINE** (максимальная длина xml-строки), закрытая функция с именем **xml** (вспомогательная функция) и окрытая фунция **toXMLLine** (формирование xml-строки). Кроме того, претерпел изменение атрибут **Address**: ранее он был закрыт и имел строковый тип, а теперь это открытый атрибут **address** типа **Address**.

Обратите внимание, что функция **xml**, которая применяется для вспомогательных операций при формировании xml-строки, используются и в классе **Address**, и в классе **Person**. Строго говоря, для компилятора это две различные функции, имеющие имена **Address:xml** и **Person:xml**. В нашем случае эти две функции абсолютно одинаковы, но используются в разных классах. Конечно, следует стремиться к тому чтобы класс был самодостатчным и независимым, но это приводит к появлению одинковых функций в разных классах. Дублирование функций усложняет сопровождение классов, потому что трудно отследить соответсвие версий одинаковых функций в разных классах (если, конечно, теребуется, чтобы эти функции работали одинаково). Позже будет предложено несколько способов решения этой проблемы.

На рис. 2.6 приведен фрагменты файла **Person.cpp**, содержащего реализацию функций-членов класса **Person**.

```
// --  файл Person.cpp
//.................................................
char* Person::toLine(char *l) const
{    //.....................................
     strcpy_s(l + (n += strlen(this->LastName)), 2, ",");
     this->address.get(Address::INFO1, l + (n += 1));
     return l;
};
//.................................................
void Person::setAddress(const char as[])
{
    this->address.set("","","",as,"");
};
//.................................................
Person::Person()
  :Order(++Number), address()
{     //.......... Нет изменений ..............
};
Person::Person(char fn[], char ln[], char as[])
     :Order(++Number), address("","","",as,"")
{     //.......... Нет изменений ..............
};
Person::Person (const Person& p)
:Order(++Number),address(p.Address)
{    //.......... Нет изменений ..............
};
void Person::xml(char* l,
                 const char s[], int lv,
                 const char v[], const char e[])const
{     //.....Аналогично Address::xml .........
};
 char* Person::toXMLLine(char l[SIZE_XMLLINE] ) const
 {
 strcpy_s(l,10,"<person>");
 xml(l,"<firstname>", Person::SIZE_FIRSTNAME,
                   this->FirstName,"</firstname>");
 xml(l,"<lastname>",  SIZE_LASTNAME,this->LastName,  "</lastname>");
 xml(l,"<sex>", 7,   (this->Sex == Person::MALE?"male":
                         this->Sex == Person::FEMALE?"female":
                                                "nill"), "</sex>");
 this->address.toXMLLine(l+strlen(l));
 strcpy_s(l+strlen(l),11,"</person>");
 return l;
 };
```

Рис. 2.6. Фрагмент cpp-файла класса Person

Прежде всего, следует обратить внимание, что инициализация нового атрибута **adderess** осуществляется в заголовочных строках конструкторов класса. Такой способ инициализации уже демонстрировался ранее для константных атрибутов. В разных конструкторах класса **Person** используются соответствующие конструкторы класса **Addrress**: в конструкторе по умолчанию класса **Person** – конструктор по умолчанию класса **Address**, в копирующем конструкторе класса **Person** – копирующий конструктор класса **Address** и т. д.

Такое соответствие, конечно, не контролируется компилятором, но правильно спроектированный класс должен придерживаться именно такой логики. Если бы в заголовочных строках конструкторов не была прописана инициализация, то компилятором все равно будет генерироваться вызов конструктора по умолчанию класса **Address**.

Кроме того, заметьте, что прототипы конструкторов не изменились, а это значит, что все приложения, использовавшие класс **Person** до его изменения, могут и далее его использовать.

Другая важная деталь, позволяющая скрыть устройство класса **Address**, – это применение в функциях-членах **toLine** и **toXMLLine** класса **Person**, функций-членов класса **Address**. Такой подход позволяет сделать независимыми реализации этих двух классов, т. е. изменение в будущем класса **Address** оставит без изменений (или сведет этот процесс к минимуму) код функций класса **Person**.

На рис. 2.7 приведен пример программы, использующей класс **Person**, а на рис. 2.8 – результат ее выполнения.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale.h>
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
   setlocale(LC_ALL, "rus");

   Person p1,p2("Прищепа", "Виктор","Минск"), *pp3 = new Person(p2);

   p1.setFirstName("Солодкевич");
   p1.setLastName("Никита");
   p1.setSex(Person::MALE);
   p1.address.set("BY","222000","Минск", "Свердлова 13", "303-1");

   char* xx = p1.toXMLLine(new char[Person::SIZE_XMLLINE]);
   std::cout<< "XML = "
            << p1.toXMLLine(new
char[Person::SIZE_XMLLINE])<<std::endl;
  return 0;
}
```

Рис. 2.7. Пример программы, использующей новый класс Person

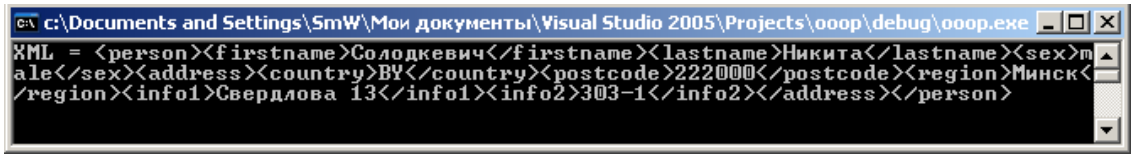

Рис. 2.8. Результат работы программы на рис. 2.7

Открытый интерфейс этого класса расширился: в примере используется его новая функция-член **Person::address.set**, используемая для установки свойства a**ddress** объекта. Вместе с тем, для совместимости с предыдущей версией сохранена и функция-член **Person::setAddress**, которую и теперь могут использовать разработанные ранее приложения.

В приведенном примере продемонстрирована инкапсуляция объекта класса **Address** в объекте класса **Person**, которая проявляется в двух основных моментах: 1) вложенности (инкапсуляции) данных; 2) инкапсуляции методов.

Обычно разработчик класса разрабатывает несколько конструкторов, позволяющих пользователю удобным для него способом создавать объекты этого класса. Как правило, среди конструкторов должен присутствовать хотя бы один, позволяющий проинициализировать все свойства создаваемого объекта в момент его создания (рис. 2.9).

```
// --  файл Person.cpp
//.............................................
class Person
{//.............................................
Person(const char fn[],              // фамилия
     const char ln[],                // имя
     const Address* as,              // адрес
     SEX   sex,                      // пол М,Ж
       int size_aby,                 // размер автобиографии в байтах
     const char aby[],               // автобиография
     int  add_p,                     // процент надбавки
     int  length_p,                  // процент за выслугу
     int  bonus_p );                 // процент премиальных
//.............................................
};
// --  файл Person.cpp
//.............................................
Person::Person(const char fn[],      // фамилия
             const char ln[],        // имя
             const Address* as,      // адрес
             SEX   sex,              // пол М,Ж
               int   size_aby,       // размер автобиографии в байтах
             const char aby[],       // автобиография
             int  add_p,             // процент надбавки
             int  length_p,          // процент за выслугу
             int  bonus_p)           // процент премиальных
:Order(++Number), address(*as), Sex(sex),
 PercentAdd(add_p), PercentLength(length_p), PercentBonus(bonus_p)
{
 setFirstName(fn);
 setLastName(ln);
 setAutoBy(size_aby, aby);
};
//.............................................
```

Рис. 2.9. Конструктор класса Person, инициализирующий все свойства объекта

Обратите внимание, что в приведенном на рис. 2.9 примере конструктора все атрибуты, кроме строковых, инициализируются в заголовочной строке конструктора.

Выше уже указывалось на наличие одинаковой функции с именем **xml** в классах **Person** и **Address**, а также обсуждались достоинства и недостатки подобного решения. Существует простой технический прием, позволяющий избежать дублирования одинаковых функций.

Для этого следует удалить из файлов с классов **Person** и **Address** прототипы (h-файл) и коды функций (cpp-файл) **xml**, а также включить в проект два дополнительных файла, которые в примере на рис. 2.10 названы **Common.h** и **Common.cpp**.

```
// --  файл Common.h
#pragma once          // исключть повтороный include
namespace Common      // пространство имен
{
  void xml(char* l,
           const char s[], int lv,  const char v[], const char e[]);
}
//---------------------------------------------------------------------
// --  файл Common.cpp
//.............................................
#include "stdafx.h"
#include <cstring>
namespace Common
{
  void xml(char* l,
           const char s[], int lv,  const char v[], const char e[])
  {
      strcpy_s(l+strlen(l), strlen(s)+1, s);
      strcpy_s(l+strlen(l), lv, v);
      strcpy_s(l+strlen(l), strlen(e)+1, e);
  };
};
```

Рис. 2.10. Содержимое файлов Common.h и Common.cpp

Файл **Common.h** предназначен для описания прототипов общих для классов **Person** и **Address** функций. Для исключения совпадения имен функций с помощью ключевого слова **namespace** [3] здесь определяется пространство имен **Common**. В приведенном примере в этом пространстве содержится только прототип функции с именем **xml**.

Файл **Common.cpp** содержит код функции **xml**. Обратите внимание на то, что код функции тоже находится в пространстве имен **Common**.

На рис. 2.11 и 2.12 приведены фрагменты файлов **Address.cpp** и **Person.cpp**, содержащие код функций, вызывающих общую функцию с именем **Common:xml**.

```
// --  файл Address.cpp
#include "StdAfx.h"
#include <cstring>
#include "Common.h"
#include "Address.h"
//.................................................
char* Address::toXMLLine(char l[SIZE_XMLLINE]) const  //  XML-строка
адреса
{
 strcpy_s(l,10,"<address>");
 Common::xml(l,"<country>",SIZE_COUNTRY,this->Country,"</country>" );
 Common::xml(l,"<postcode>",SIZE_POSTCODE,this->Postcode,"</postcode>"
);
 Common::xml(l,"<region>",SIZE_REGION,this->Region,"</region>" );
 Common::xml(l,"<info1>",SIZE_INFO,this->Info1,"</info1>" );
 Common::xml(l,"<info2>",SIZE_INFO,this->Info2,"</info2>" );
 strcpy_s(l+strlen(l),11,"</address>");
return l;
}
//.................................................
```

Рис. 2.11. Применение функции Common::xml
в функции класса Address

```
// --  файл Person.cpp
#include "stdafx.h"
#include <cstring>
#include "Common.h"
#include "Person.h"
//.................................................

char* Person::toXMLLine(char l[SIZE_XMLLINE] ) const
 {
 strcpy_s(l,10,"<person>");
 Common::xml(l,
    "<firstname>", Person::SIZE_FIRSTNAME,this-
>FirstName,"</firstname>");
 Common::xml(l,
    "<lastname>",  SIZE_LASTNAME,this->LastName,  "</lastname>");
 Common::xml(l,
    "<sex>", 7,   (this->Sex == Person::MALE?"male":
                        this->Sex == Person::FEMALE?"female":
                                                "nill"),  "</sex>");
 this->address.toXMLLine(l+strlen(l));
 strcpy_s(l+strlen(l),11,"</person>");
 return l;
 }
//.................................................
```

Рис. 2.12. Применение функции Common::xml
в функции класса Person

Обратите внимание, что для использования общей функции **Common:xml** необходимо с помощью директивы препроцессора **include** включить в файлы **Person.cpp** и **Address.cpp** файл **Common.h**.

Является ли решение, использующее единый экземпляр функции **Common:xml** в двух классах, лучше предыдущего, где использовались две одинаковые функции **Person:xml** и **Address:xml**?

Поддерживать одну копию функции значительно проще, и это, конечно, является положительной стороной такого подхода.

С другой стороны, теперь вместо двух файлов каждого класса необходимо иметь четыре. Более того, если количество классов увеличивается и они взаимодействуют (инкапсулируют, наследуют и т. п.) между собой, то содержимое файла **Common** начинает стремительно расти. Это приводит к тому, что, если потом вдруг потребуется использовать только один небольшой класс, например все тот же **Address**, то он «потянет» за собой большие общие файлы **Common**, хотя использует только одну несложную функцию **Common:xml**. Если программист «в качестве исключения и только один раз» продублирует эту функцию в классе, то можно быть уверенным, что рано или поздно он вернется к первому случаю.

В предложенной схеме взаимодействия объекта класса **Address** с объектом класса **Person** предполагается, что объекты класса **Address** могут существовать независимо от объектов класса **Person**. Обратное утверждение несправедливо – любой объект класса **Person** содержит объект класса **Address** и поэтому является зависимым. Действительно, любое изменение класса **Address** неминуемо отразится и на классе **Person**. В процессе развития класса **Address** у него могут появиться такие свойства, которые будут противоречить логике (и даже мешать работе) класса **Person**. Естественный способ избежать таких проблем – скрыть от пользователя объект класса **Address**, т. е. инкапсулировать его.

Для того чтобы инкапсулировать в классе **Person** объект **address**, необходимо, прежде всего, сделать его закрытым. Но в этом случае пользователю станут недоступными функции **Person:address.set** и **Person::address.get**, позволяющие записать или считать данные адреса. Для решения этой проблемы придется создать аналогичные открытые функции-члены класса **Person**, которые инкапсулируют закрытые для пользователя, но доступные «изнутри» класса **Person** методы объекта **address**.

На рис. 2.13 приведен фрагмент доработанного класса **Person**, инкапсулирующего объект **address** класса **Address**. Объявление объекта **address** перенесено в область действия модификатора **private**, что делает его свойства и методы закрытыми для доступа за пределами класса. Появились новые константы, объявленные с помощью перечисления **GETADDRESS**, а также новые методы **setAddress** и **getAddress**.

```
// --  файл Person.h
//.............................................
class Person
{
 //.............................................
 public:
    enum GETADDRESS{COUNTRY  = Address::COUNTRY ,
                    POSTCODE = Address::POSTCODE,
                    REGION   = Address::REGION,
                    INFO1    = Address::INFO1,
                    INFO2    = Address::INFO2};
 //.............................................
 private:
    Address  address;                 // адрес
 //.............................................
 public:
    void setAddress(const char as[]);
    void setAddress(const char Country[Address::SIZE_COUNTRY],
                    const char Postcode[Address::SIZE_POSTCODE],
                    const char Region[Address::SIZE_REGION],
                    const char Info1[Address::SIZE_INFO],
                    const char Info2[Address::SIZE_INFO]);
    char* getAddress(GETADDRESS t, char*) const;
//.............................................
}
```

Рис. 2.13. Фрагмент класса Person, инкапсулирующего методы объекта класса Address

Обратите внимание на способ инкапсуляции констант, определяющих имена свойств объекта **address**. Теперь пользователю класса **Person** для поименного доступа к свойствам адреса предоставлен набор констант класса **Person** (а не класса **Address**, как было ранее). Если новая версия класса **Address** предусмотрит расширение своего списка данных-членов, то это изменение останется «незамеченным» объектами класса **Person**.

Для инкапсуляции функции **Address::set** в классе **Person** предназначена открытая функция-член **setAddress**, имеющая четыре параметра. Обратите внимание, что новая функция перегружает уже существующую функцию **setAddress** с одним параметром, сохраненную для совместимости с предыдущими версиями класса. В главе 1 уже обсуждалась перегрузка конструкторов класса. Относительно остальных функций-членов класса (за исключением деструктора), действуют те же правила, т. е. правила перегрузки функций C++ [3].

Для инкапсуляции функции **Address::get** в классе **Person** предназначена отрытая функция-член **getAddress**. Она имеет тот же набор параметров, что и **Address::get**.

На рис. 2.14 представлен фрагмент файла **Person.cpp**, демонстрирующий реализацию инкапсулирующих функций **setAddress** и **getAddress**.

```
// --  файл Person.cpp
//.............................................
void Person::setAddress(const char as[])
{
    this->address.set("","","",as,"");
};

void Person::setAddress(const char Country[Address::SIZE_COUNTRY],
                        const char Postcode[Address::SIZE_POSTCODE],
                        const char Region[Address::SIZE_REGION],
                        const char Info1[Address::SIZE_INFO],
                        const char Info2[Address::SIZE_INFO])
{
     this->address.set(Country,Postcode,Region,Info1,Info2);
};

char* Person::getAddress(GETADDRESS t, char* as) const
 {
    return this->address.get((Address::GET)t, as);
 };
//.............................................
```

Рис. 2.14. Реализация инкапсулирующих функций setAddress и getAddress

Работа инкапсулирующих функций, как правило, сводится просто к вызову функций закрытого инкапсулируемого объекта. Обратите внимание: в функции **getAddress** при передаче параметров потребовалось явное преобразование типа **Person:**:**GETADDRESS** к типу **Address::GET**.

Очевидно, что объекты класса **Address** найдут применение не только в рамках класса **Person**, предназначенного для хранения информации об индивидууме. Действительно, объекты этого класса могут быть использованы при описании самых различных классов. Например, класс, описывающий здание, класс, описывающий учреждение и т. д.

Часто при определении класса удобно использовать вложенный класс, т. е. класс, определенный внутри определения другого класса. Как правило, внутренний класс неразрывно связан с внешним и его отдельное использование представляется маловероятным. Наиболее подходящими атрибутами класса **Person** для превращения их в атрибуты вложенного в **Person** класса являются **FirstName** и **LastName**.

На рис. 2.15 представлен фрагмент новой версии класса **Person** с вложенным классом **Name**. Описание класса **Name** располагается внутри фигурных скобок, ограничивающих описание внешнего клас-

са **Person**. Использованная здесь конструкция **class Name {…} name** позволяет одновременно определить класс **Name** и атрибут **name** этого типа. Кроме того, здесь представлен прототип нового конструктора класса **Person** с полным набором параметров, включая параметр типа **Name**.

```
// --  файл Person.h
//.............................................
class Person
{
 public:
 class Name  // вложенный класс
   {
    public:
      const static int SIZE_FIRSTNAME  = 50;
      const static int SIZE_LASTNAME   = 50;
      const static int SIZE_PATRNAME   = 50;
      const static int SIZE_LINE       = SIZE_FIRSTNAME
                                      + SIZE_LASTNAME
                                + SIZE_PATRNAME +3;
      const static int SIZE_XMLLINE =  SIZE_LINE +101;
    private:
      char FirstName[SIZE_FIRSTNAME];  // имя
      char LastName[SIZE_LASTNAME];    // фамилия
      char PatrName[SIZE_PATRNAME];    // отчество
    public:
      Name(){FirstName[0]=LastName[0]=PatrName[0]='\0';};
      Name(const char FirstName[SIZE_FIRSTNAME],
           const char LastName[SIZE_LASTNAME],
           const char PatrName[SIZE_PATRNAME]);
      Name(const Name& name);
      void setFirst(const char FirstName[SIZE_FIRSTNAME]);
      void setLast(const char  LastName[SIZE_FIRSTNAME]);
      void setPatr(const char  LastName[SIZE_PATRNAME]);
      char* getFirst(char buf[SIZE_FIRSTNAME]) const;
      char* getLast(char  buf[SIZE_LASTNAME])  const;
      char* getPatr(char buf[SIZE_PATRNAME])   const;
      char* toLine(char *l) const;
      char* toXMLLine(char l[SIZE_XMLLINE]) const;
    }   name;                        // открытый атрибут
   const static int SIZE_LINE  = Name::SIZE_LINE
                              +Address::SIZE_LINE+2;
   const static int SIZE_XMLLINE   = SIZE_LINE +101;
   //.............................................
   Person(const Name* name,     // фамилия, имя, отчество
          const Address* as,    // адрес
          SEX   sex,            // пол М,Ж
          int size_aby,         // размер автобиографии в байтах
          const char aby[],     // автобиография
          int  add_p,           // процент надбавки
          int  length_p,        // процент за выслугу
          int  bonus_p);        // процент премиальных
   //.............................................
}
```

Рис. 2.15. Фрагмент класса Person с вложенным классом Name

```
// --  файл Person.cpp
//.............................................
void Person::Name::setFirst(const char FirstName[SIZE_FIRSTNAME])
{
   strcpy_s(this->FirstName, SIZE_FIRSTNAME,FirstName);
};
void Person::Name::setLast(const char LastName[SIZE_LASTNAME])
{
   strcpy_s(this->LastName, SIZE_LASTNAME, LastName);
};
void Person::Name::setPatr(const char PatrName[SIZE_PATRNAME])
{
   strcpy_s(this->PatrName, SIZE_PATRNAME, PatrName);
};
Person::Name::Name(const char FirstName[SIZE_FIRSTNAME],
                const char LastName[SIZE_LASTNAME],
                const char PatrName[SIZE_PATRNAME])
{
  this->setFirst(FirstName);
  this->setLast(LastName);
  this->setPatr(PatrName);
};
Person::Name::Name(const Name& name)
{
   this->setFirst(name.FirstName);
   this->setLast(name.LastName);
   this->setPatr(name.PatrName);
};
char* Person::Name::toLine(char *l) const
{
   strcpy_s(l, SIZE_FIRSTNAME, this->FirstName);
   strcpy_s(l + strlen(l), 2, ",");
   strcpy_s(l + strlen(l),SIZE_LASTNAME, this->LastName);
   strcpy_s(l + strlen(l), 2, ",");
   strcpy_s(l + strlen(l),SIZE_PATRNAME, this->PatrName);
   return l;
}
char* Person::Name::toXMLLine(char l[SIZE_XMLLINE]) const
{
   strcpy_s(l,7,"<name>");
   Common::xml(l,"<firstname>",SIZE_FIRSTNAME,
                          this->FirstName,"</firstname>");
   Common::xml(l,"<lastname>", SIZE_LASTNAME,
                          this->LastName,  "</lastname>");
   strcpy_s(l+strlen(l),8,"</name>");
   return l;
};
//.............................................
```

Рис. 2.16. Реализация функций-членов класса Name

Построение класса **Name** выполнено по тому же принципу, что и для классов **Address** и **Person**: сначала объявляются открытые константы, значение которых указывает размерность строковых атрибутов класса (**SIZE_FIRSTNAME**, **SIZE_LASTNAME**, **SIZE_PATRNAME**), максимальную длин строк (**SIZE_LINE, SIZE_XMLLINE**), формируемые

функциями **Name::toLine** и **Name::toXMLLine**. Для хранения данных используется три закрытых атрибута (**FirstName**, **LastName** и **PatrName**). Далее следует описание открытых функций-членов, конструкторов и деструктора. Реализация функций-членов вложенного класса **Name** представлена на рис. 2.16.

```
// --  файл Person.cpp
//.............................................
char* Person::toLine(char *l) const
{   strcpy_s(l, Name::SIZE_LINE, name.toLine(l));
    strcpy_s(l + strlen(l), 2, ",");
    address.get(Address::INFO1, l + strlen(l));
    return l;  };
void Person::setFirstName(const char fn[])
 {  this->name.setFirst(fn); };

void Person::setLastName(const char ln[])
 {  this->name.setLast(ln);  };

Person::Person(char fn[], char ln[], char as[])
  :Order(++Number), name(fn, ln, ""), address("","","",as,""),
Sex(NILL),
  PercentAdd(0), PercentLength(0), PercentBonus(0)
 {  ++Person::Count;
   setAutoBy();};

Person::Person(const Name* n,
            const Address* as,    // адрес
            SEX   sex,            // пол М,Ж
            int size_aby,         // размер автобиографии в байтах
            const char aby[],     // автобиография
            int  add_p,           // процент надбавки
            int  length_p,        // процент за выслугу
            int  bonus_p)         // процент премиальных
  :Order(++Number), name(*n),  address(*as), Sex(sex),
   PercentAdd(add_p), PercentLength(length_p), PercentBonus(bonus_p)
 {  ++Person::Count;
   setAutoBy(size_aby, aby);};

Person::Person (const Person& p)
  :Order(++Number), name(p.name),  address(p.address), Sex(p.Sex),
  PercentAdd(p.PercentAdd),
  PercentLength(p.PercentLength),PercentBonus(p.PercentBonus)
{ ++Person::Count;
  setAutoBy(p.SizeAutoBy, p.AutoBy);};

char* Person::toXMLLine(char l[SIZE_XMLLINE] ) const
{  strcpy_s(l,10,"<person>");
  this->name.toXMLLine(l+strlen(l));
  this->address.toXMLLine(l+strlen(l));
  strcpy_s(l+strlen(l),11,"</person>");
  return l;};
//.............................................
```

Рис. 2.17. Функции-члены класса Person, изменившиеся в результате замены атрибутов FirstName и LastName объектом name вложенного класса Name

Функции-члены вложенных классов обычно размещают в том же файле, что и функции члены внешнего класса (в нашем случае – это файл **Person.cpp**). Для указания вложенности членов класса **Name** используется нотация **Person::Name::**.

На рис. 2.17 представлены функции-члены класса **Person**, претерпевшие изменения в результате замены атрибутов **FirstName** и **LastName** объектом **name** вложенного класса **Name**.

Обратите внимание, что класс **Person** теперь фактически превратился в оболочку (капсулу) для объектов классов **Address** и **Name**. Типичным для функций классов, осуществляющих инкапсуляцию, является вид функции **Person::toXMLLine**, которая предназначена для генерации xml-строки. Обычно работа функций инкапсулирующего класса сводится к вызову функций инкапсулируемого класса.

Новая версия класса **Person**, позволяет установить значения фамилии, имени и отчества с помощью функций **Person::Name::setFirst**, **Person::Name::setLast** и **Person::Name::setPatr**. Для совместимости с предыдущей версией необходимо переписать функции-члены **setFirstName** и **setFirstName**, а также некоторые конструкторы, не приведенные на рис. 2.17.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale.h>
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
 setlocale(LC_ALL, "rus");

 Person p1(new Person::Name("Закрута", "Илья", "Андреевич"),
         new Address("BY","222000","Витебск","Орша,ул.О.Бендера,14",""),
           Person::FEMALE,
         19,"Образование:высшее",
         10,20,30);
 std::cout<< "XML = "
       << p1.toXMLLine(new char[Person::SIZE_XMLLINE])<<std::endl;
 std::cout<< "LINE = "
       << p1.getLine(new char[Person::SIZE_LINE])<<std::endl;
  return 0;
}
```

Рис. 2.18. Пример программы, использующей
объекты классов Person, Address и Person::Name

Применять вложенные классы можно также, как и классы первого уровня, но для их обозначения используется специальная нотация. На рис. 2.18 приведен пример программы, использующей объекты классов **Person**, **Address** и **Name**, а на рис. 2.19 – результаты ее работы. Обратите внимание на нотацию, используемую при создании объекта **Person::Name**.

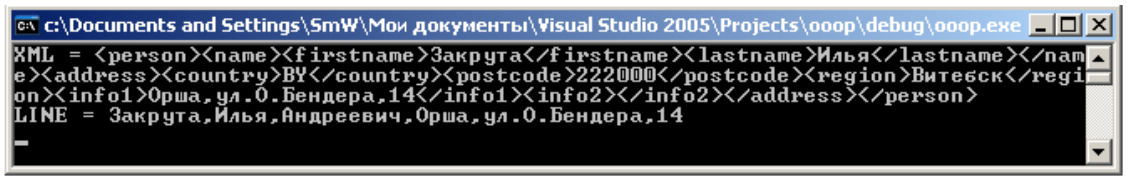

Рис. 2.19. Результат работы программы на рис. 2.18

Хорошо спроектированная система классов представляет собой набор сходных по своей структуре классов. Взаимодействие в этой системе объектов классов-капсул и объектов инкапсулируемых классов осуществляется по принципу матрешки: функции-члены класса-капсулы вызывают похожие функции внутренних объектов.

2.3. Обработка исключений

Генерация исключений в методах объекта применяется, как правило, при обнаружении ошибки во внешних по отношению к объекту данных. Если ошибка не может быть обработана средствами самого объекта, то генерируется исключение, которое информирует внешнюю среду о возникшей ошибке и предлагает ее обработать.

В некоторых случаях другого способа обработки ошибки просто не существует. Конструкторы и деструктор класса по правилам языка C++ не могут возвращать значений. Единственный способ остановить работу этих функций и сообщить о возникшей ошибке – сгенерировать исключение.

Механизм исключений в C++ основан на использовании трех инструкций языка: **try**, **catch** и **throw** [2, 3].

На рис. 2.20 представлен фрагмент новой версии класса **Addrres**, функции-члены которого будут генерировать исключения в случае обнаружения некоторых ошибок.

Новая версия описательной части класса **Address** (файл **Address.h**) отличается от предыдущей лишь наличием вложенного класса с именем **Exception**, предназначенного для хранения информации о возникшей ошибке. Объект этого класса будет создаваться в момент генерации исключения (инструкция **throw**) будет содержать в себе код ошибки (атрибут **code**) текстовое сообщение (**message**), указатель на объект, сгенерировавший исключение (**paddress**) и указатель на объект, который явился причиной генерации исключения (**lastexcept**). Последний атрибут не будет использоваться при генерации исключений в объектах класса **Address**.

```
// --  файл Address.h
#define ADDRESS_ERROR    100   // серия кодов исключений
#define  MSG_ERROR_COUNTRY   "код страны"
#define  MSG_ERROR_POSTCODE  "почтовый индекс"
#define  MSG_ERROR_REGION    "область"
#define  MSG_ERROR_INFO1     "строка 1 детализации адреса"
#define  MSG_ERROR_INFO2     "строка 2 детализации адреса"
class Address
{
 public:
 class Exception
 {
  public:
    const static int ERROR_COUNTRY  = ADDRESS_ERROR + 1;
    const static int ERROR_POSTCODE = ADDRESS_ERROR + 2;
    const static int ERROR_REGION   = ADDRESS_ERROR + 3;
    const static int ERROR_INFO1    = ADDRESS_ERROR + 4;
    const static int ERROR_INFO2    = ADDRESS_ERROR + 5;
    const static int MAX_MSG        = 100;
    Exception(int code,                       // код ошибки
              char msg[MAX_MSG],              // сообщение
              const Address* as,              // объект с ошибкой
              const void* le = (void*)0);     // объект с ошибкой
    int code;                                 // код ошибки
    char message[MAX_MSG];                    // сообщение
    const void   *lastexcept;                 // объект с ошибкой
    const Address *paddress;                  // объект с ошибкой

 };
 //.............................................
};
```

Рис. 2.20. Фрагмент класса Address с вложенным классом Exception

С помощью константных статических атрибутов с именами **ERROR_xxx** перечислены все возможные коды ошибок, а с помощью препроцессорных констант с именами **MSG_ERROR_xxx** – все возможные текстовые сообщения.

Препроцессорная константа **ADDRES_ERROR** определяет серию кодов. Такой подход позволяет просто изменить коды ошибок, если вдруг существующие коды будут перекрываться с кодами ошибок, используемыми в других классах.

На рис. 2.21 представлены пример реализации функции-члена **set** и конструктора вложенного класса **Exception.**

```
// --  файл Address.cpp
//.............................................
void Address::set(const char Country[SIZE_COUNTRY],
                  const char Postcode[SIZE_POSTCODE],
                    const char Region[SIZE_REGION],
                    const char Info1[SIZE_INFO],
                 const char Info2[SIZE_INFO])
{
 if        (!Common::isstralpha(Country))
      throw Exception(Exception::ERROR_COUNTRY, MSG_ERROR_COUNTRY, this);
 else  if  (!Common::isstrprint(Postcode))
      throw Exception(Exception::ERROR_POSTCODE, MSG_ERROR_POSTCODE, this);
 else  if  (!Common::isstralpha(Region))
      throw Exception(Exception::ERROR_REGION, MSG_ERROR_REGION, this);
 else  if  (!Common::isstrprint(Info1))
      throw Exception(Exception::ERROR_INFO1, MSG_ERROR_INFO1, this);
 else  if  (!Common::isstrprint(Info2))
      throw Exception(Exception::ERROR_INFO2, MSG_ERROR_INFO2, this);
 else {strcpy_s(this->Country, SIZE_COUNTRY, Country);
       strcpy_s(this->Postcode,SIZE_POSTCODE,Postcode);
       strcpy_s(this->Region,SIZE_REGION,Region);
       strcpy_s(this->Info1,SIZE_INFO,Info1);
       strcpy_s(this->Info2,SIZE_INFO,Info2);};
};
Address::Exception::Exception(int code, char msg[MAX_MSG],
                              const Address* as, const void* le)
:code(code), paddress(as), lastexcept(le)
{
       strcpy_s(this->message, MAX_MSG, msg);
};
//.............................................
```

Рис. 2.21. Функции-члены класса Address, генерирующие исключение

Для простоты изложения будем предполагать, что все конструкторы и функции-члены класса **Address** используют для установки значений атрибутов этого класса только функцию **set.** Новая функция **set** выполняет проверку параметров на корректность с помощью двух функции **isstralpha** и **isstrprint**, расположенных в пространстве имен **Common.** Пример реализации этих функций приводится на рис. 2.22. В случае обнаружения ошибки (функции **isstralpha** или **isstrprint** возвращают значение **false**) генерируется исключение типа **Exception** с соответствующими параметрами: кодом ошибки, сообщением и указателем на объект.

Исключение создает объект типа **Address:: Exception** и запускает конструктор, который просто инициализирует свойства этого объекта.

Функция **isstralpha**, представленная на рис. 2.22, предназначена для проверки содержимого строки. В том случае, если строка состоит только из символов-букв алфавита текущей локали [3], то функция возвращает значение **true**, иначе возвращается **false**. Функция **isstrprint** работает по тому же принципу, что и **isstralpha**, но корректными считает символы, имеющие изображение (печатаемые символы).

```
// --  файл Common.h
namespace Common
{
//.................................................
bool isstrprint(const char s[]); // только печатаемые символы или 0x00?
bool isstralpha(const char s[]); // только буквы или 0x00?
}

// --  файл Common.cpp
//.................................................
#include <locale>
namespace Common
{
bool isstrprint(const char s[])
{
   int i = -1, ls = strlen(s);
   bool rc = true;
   if (s[0] != 0x00)
   {
     while ( i <= ls &&  isprint((int)((unsigned char) s[++i])) > 0 );
     rc = (i == ls)?true:false;
   };
   return rc;
};
bool isstralpha(const char s[])  ?
{
   int i = -1, ls = strlen(s);
   bool rc = true;;
   if (s[0] != 0x00)
   {
     while ( i <= ls &&  isalpha((int)((unsigned char) s[++i])) > 0 );
     rc = (i == ls)?true:false;
   };
   return rc;
};
//.................................................
```

Рис. 2.22. Пример реализации функций для проверки корректности строковых значений

На рис. 2.23 приведен пример программы, обрабатывающей исключения, сгенерированные объектом класса **Address**.

Обратите внимание на ошибку, которая допущена в значении первого параметра конструктора объекта **a2**. Она будет обнаружена с помощью функции **Common::isstralpha** (рис. 2.22) в методе **a2.set** (рис. 2.21), вызываемого в конструкторе класса **Address**. При обнаружении этой ошибки в методе **a2.set** с помощью инструкции **throw** генерируется исключение типа **Address:Exception** с кодом **Address::Exception::ERROR_COUNTRY** и сообщением **MSG_ERROR_COUNTRY**. Объект типа **Address::Exception**, созданный инструкцией **throw**, после того, как выполнение конструктора было прервано генерированным исключением, становится доступным в блоке **catch** (рис. 2.23) в виде переменной **e**. Результат обработки ошибки представлен на рис. 2.24.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale>
#include "Common.h"
#include "Person.h"

int _tmain(int argc, _TCHAR* argv[])
{
 setlocale(LC_ALL, "rus");
 try
 {
  Address  a1;
  Address  a2("B3", "222000",
              "Гродно","ул. К.Воробянинова, 3а, 14", "этаж 6");
 }
 catch (Address::Exception e)
 {std::cout<< "Ошибка в адресе:" << e.code << ":" <<e.message
<<std::endl;}
 return 0;
}
```

Рис. 2.23. Пример программы, обрабатывающей исключение, сгенерированное объектом класса Address

Рис. 2.24. Результат работы программы, представленной на рис. 2.23

Выполним для класса **Person::Name** доработку, аналогичную предыдущему случаю. На рис. 2.25 изображен фрагмент файла **Person.h**, содержащий измененную часть вложенного класса **Name**. Вложенный в **Name** класс **Exception** по своей структуре подобен

классу **Address:Exception**, поэтому в дополнительном пояснении не нуждается.

Для простоты реализации в класс **Person::Name** добавлена функция **Name::set**, которая потом будет применяться при реализации других функций этого класса.

```
// --  файл Person.h
//.............................................
#define PERSON_ERROR_NAME       300            // серия кодов исключений
#define  MSG_ERROR_NAME_FIRST  "Фамилия"
#define  MSG_ERROR_NAME_LAST   "Имя"
#define  MSG_ERROR_NAME_PATR   "Отчество"
class Person
{
 public:
   class Name
   {
   public:
       class Exception
       {
        public:
        const static int ERROR_FIRST     = PERSON_ERROR_NAME + 1;
        const static int ERROR_LAST      = PERSON_ERROR_NAME + 2;
        const static int ERROR_PATR      = PERSON_ERROR_NAME + 3;
       const static int MAX_MSG         = 100;
       Exception(){code = 0; message[0]=0x00;};
       Exception(int code,
                  char msg[MAX_MSG],
                  const Name* n,
                  const void* le = (void*)0);
        int code;
        char message[MAX_MSG];
       const void   *lastexcept;
       const Name *name;
       };
   //.............................................
   void set(const Name& n)
     {setFirst(n.FirstName); setLast(n.LastName);
setPatr(n.PatrName);};
   //.............................................
  };
//.............................................
}
```

Рис. 2.25. Фрагмент класса Person::Name
с вложенным классом Exception

На рис. 2.26 представлен пример реализации некоторых функций-членов класса **Person::Name**. Конструктор вложенного в **Name** класса **Exception** практически идентичен конструктору класса **Address::Exception**. Функции **setFirst**, **SetLast** и **setPatr** перед установкой значения атрибута осуществляют проверку параметров с помощью все тех же функций **Common::isstralpha** и **Common::isstrprint**

и в случае обнаружения ошибки генерируют исключения с соответствующими параметрами точно также, как это выполнялось в функциях-членах класса **Address**. Здесь же приведены примеры двух конструкторов класса **Person::Name**, которые в итоге вызывают функции, выполняющие проверку на корректность параметров.

```
// --  файл Person.cpp
//.............................................
Person::Name::Exception::Exception(int code, char msg[MAX_MSG],
                                   const Name* n, const void* le )
 :code(code), name(n), lastexcept(le)
{ strcpy_s(this->message, MAX_MSG, msg);};
void Person::Name::setFirst(const char FirstName[SIZE_FIRSTNAME])
{
if (!Common::isstralpha(FirstName))
 throw Exception(Name::Exception::ERROR_FIRST, MSG_ERROR_NAME_FIRST,
this);
else  strcpy_s(this->FirstName, SIZE_FIRSTNAME,FirstName);
};
void Person::Name::setLast(const char LastName[SIZE_LASTNAME])
{
if  (!Common::isstralpha(LastName))
 throw Exception(Name::Exception::ERROR_LAST, MSG_ERROR_NAME_LAST,
this);
 else  strcpy_s(this->LastName, SIZE_LASTNAME, LastName);
};
void Person::Name::setPatr(const char PatrName[SIZE_PATRNAME])
{
if  (!Common::isstralpha(PatrName))
 throw Exception(Name::Exception::ERROR_PATR, MSG_ERROR_NAME_PATR,
this);
else  strcpy_s(this->PatrName, SIZE_PATRNAME,PatrName);
};
Person::Name::Name(const char FirstName[SIZE_FIRSTNAME],
                const char LastName[SIZE_LASTNAME],
                const char PatrName[SIZE_PATRNAME])
{
this->setFirst(FirstName);
this->setLast(LastName);
this->setPatr(PatrName);
};
Person::Name::Name(const Name& name) { this->set(name);};
//.............................................
```

Рис. 2.26. Функции-члены класса Person::Name,
генерирующие исключения

Если оставить класс **Person** в том виде, в котором он существует на текущей момент, то в результатами работы объектов этого класса могут быть сгенерированные исключения типа **Address::Exception** и **Person::Name::Exception**. Ясно, что это нарушает принцип инкапсуляции: капсула «выпускает наружу» свойства других внутренних объектов. Кроме того, по всей видимости, будет необходимо делать

проверки на корректность данных и в собственных методах объекта класса **Person**, которые тоже могут сгенерировать исключения. В этом случае добавится еще один тип исключения, который может быть выработан объектом. Все это приводит к непредсказуемости поведения объекта типа **Person** и трудному восприятию его пользователями. Очевидно, что наиболее приемлемым решением будет свести все исключения к одному типу, например к **Person::Exception**, но в этом случае исключения, выработанные инкапсулированными объектами, должны быть «перехвачены» (далее будем использовать это слово без кавычек) в методах объекта класса **Person** и сгенерированы исключения типа **Person::Exception**.

Построим вложенный класс **Person::Exception** по тому же принципу, что и в двух предыдущих случаях. На рис. 2.27 снова представлен фрагмент файла **Person.h**, но с описанием класса **Person::Exception**. Легко видеть, что структура класса **Person::Exception** ничем не отличается от структуры классов **Address::Exception** или **Person::Name:Exception**. Обратим внимание только на то, что здесь предполагается только два типа кода, сопровождающих исключение **Person::ERROR_NAME** и **Person::ERROR_ADDRESS**.

```
// --  файл Person.h
//.................................................
#define PERSON_ERROR            200             // серия кодов исключений
#define  MSG_ERROR_NAME         "Имя"
#define  MSG_ERROR_ADDRESS      "Адрес"
//.................................................
class Person
{
 public:
   class Name
   {
   public:
      class Exception
       {//.....................................................};
   //.................................................
   };
   class Exception
   {
    public:
    const static int ERROR_NAME     = PERSON_ERROR + 1;
    const static int ERROR_ADDRESS  = PERSON_ERROR + 2;
    const static int MAX_MSG        = Person::Name::Exception::MAX_MSG
                                     +Address::Exception::MAX_MSG +100;
    Exception(){code = 0; message[0]=0x00;};
    Exception(int code,
              char msg[MAX_MSG],
              const Person* as,
              const void* le = (void*)0);
    int code;
    char message[MAX_MSG];
    const void   *lastexcept;
    const Person *person;
   };
 //.................................................
};
```

Рис. 2.27. Фрагмент класса Person с вложенным классом Exception

Код **Person::ERROR_NAME** и сообщение **MSG_ERROR_NAME** будут сопровождать исключения, выработанные в результате перехвата исключений типа **Person::Name::Exeption**, а код **Person::ERROR_ADDRESS** и соощение **ERROR_ADDRESS –** при перехвате исключений типа **Address::Exception**.

На рис. 2.28 представлен пример реализации некоторых функций-членов класса **Person**, которые перехватывают исключения типа **Address::Exception** и **Person::Name::Exeption** и генерируют собственные кличения типа **Person::Exeption**.

```
// --  файл Person.cpp
//.............................................
void Person::setAddress(const char Country[Address::SIZE_COUNTRY],
                        const char Postcode[Address::SIZE_POSTCODE],
                        const char Region[Address::SIZE_REGION],
                        const char Info1[Address::SIZE_INFO],
                        const char Info2[Address::SIZE_INFO])
{
  try{this->address.set(Country,Postcode,Region,Info1,Info2);}
  catch (Address::Exception e)
 {throw Exception(Exception::ERROR_ADDRESS, MSG_ERROR_ADDRESS, this, &e);
};
Person::Person(const Name* n,
            const Address* as,     // адрес
            SEX   sex,             // пол М,Ж
            int size_aby,          // размер автобиографии в байтах
            const char aby[],      // автобиография
            int  add_p,            // процент надбавки
            int  length_p,         // процент за выслугу
            int  bonus_p )         // процент премиальных
 :Order(++Number),   Sex(sex),
  PercentAdd(add_p), PercentLength(length_p), PercentBonus(bonus_p)
{
 ++Person::Count;
 try
 {
  this->address.set(*as);
  this->name.set(*n);
 }
 catch (Address::Exception e)
 {throw Exception(Exception::ERROR_ADDRESS, MSG_ERROR_ADDRESS, this, &e);}
 catch (Name::Exception e)
 {throw Exception(Exception::ERROR_NAME, MSG_ERROR_NAME, this, &e);};
  setAutoBy(size_aby, aby);
};
Person::Exception::Exception(int code, char msg[MAX_MSG],
                             const Person* as,  const void* le)
:code(code), person(as), lastexcept(le)
{ strcpy_s(this->message, MAX_MSG, msg);};
//.............................................
```

Рис. 2.28. Пример реализации некоторых функций-членов класса Person, выполняющих обработку и генерацию исключений

Обратите внимание, что при вызове конструктора **Person::Exception** в инструкциях **throw** используется четвертый параметр. В предыдущих случая его значение устанавливалось по умолчанию в ноль. С помощью этого параметра объект класса **Person::Exception** сохраняет указатель на объект, сопровождающий перехваченное исключение.

На рис. 2.29 приведен пример программы, обрабатывающей исключения, сгенерированные объектами классов **Address**, **Person::Name** и **Person**.

```
//.............................................
int _tmain(int argc, _TCHAR* argv[])
{
   setlocale(LC_ALL, "rus");
 try
 {
  Person p1(new Person::Name("Закрута","Илья","Андреевич"),
         New Address("BY","222000","Витебск","Орша,ул.О.Бендера,14",""),
         Person::FEMALE,
         19,"Образование:высшее",
         10,20,30);

   // p1.name.setFirst("Розенбкевич");
   p1.setAddress("B3", "222000",
                 "Гродно","ул.К.Воробянинова, 3а, 14", "этаж 6");
 }
 catch (Person::Name::Exception e)
 {std::cout<< "Имя:" << e.code << ":" <<e.message <<std::endl;}
 catch (Address::Exception e)
 {std::cout<< "Адрес:" << e.code << ":" <<e.message <<std::endl;}
 catch (Person::Exception e)
 {
    std::cout<< "Персональные данные:" << e.code << ":" <<e.message;
    if (e.code == Person::Exception::ERROR_ADDRESS)
    std::cout<<":"<<((Address::Exception*)e.lastexcept)->code<<":"
             <<((Address::Exception*)e.lastexcept)->message<<std::endl;
    if (e.code == Person::Exception::ERROR_NAME)
    std::cout<<":"<<((Person::Name::Exception*)e.lastexcept)->code<<":"
             <<((Person::Name::Exception*)e.lastexcept)->message<<std::endl;
 }
 return 0;
}
```

Рис. 2.29. Пример программы, обрабатывающей исключения, сгенерированные объектами классов Adderss, Person::Name и Person

При создании объекта **p1** в качестве параметров конструктора используются динамически создаваемые объекты классов **Address** и **Person::Name**, которые могут генерировать исключения. Для обработки этих исключений предусмотрены соответствующие catch-блоки.

Обратите внимание, что в первом параметре вызываемого в try-блоке метода **setAddress** сделана ошибка, которая приведет к генерации исключения объектом класса **Address**, которое будет перехвачено в методе **setAddress** (рис. 2.28) и сгенерировано новое исключение типа **Perspon::Exception**. Обработка этого исключение будет осуществляться в catch-блоке программы, представленной на рис. 2.29.

Обработка исключения **Perspon::Exception** выполняется в два этапа. Вначале обрабатывается объект класса **Person::Exception**, а затем в зависимости от кода, сопровождающего исключение, обрабатывается либо объект класса **Person::Name::Exception**, либо объект класса

Address::Exception (как это происходит в нашем случае). Результат выполнения программы демонстрируется на рис. 2.30.

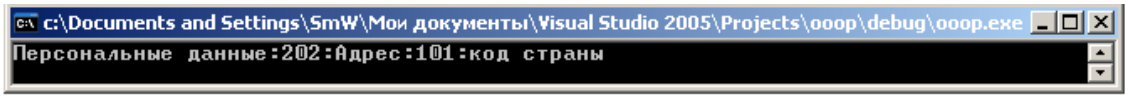

Рис. 2.30. Результат выполнения программы на рис. 2.29

К сожалению, выполненные выше построения не позволяют перехватить все исключения, генерируемые объектом класса **Person**. Если в программе на рис. 2.29 снять комментарии внутри try-блока, то ошибка, сделанная в параметре метода **p1.name.setFirst**, приведет к результату, как на рис. 2.31.

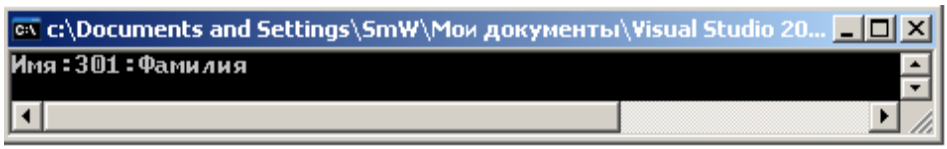

Рис. 2.31. Результат выполнения программы на рис. 2.29
после корректировки try-блока

Это происходит из-за того, что при разработке (и доработке) класса **Person** не был скрыт (инкапсулирован) его атрибут **name**. Для того чтобы исключения не «прорывались», необходимо полностью инкапсулировать все атрибуты класса. Очевидно, в этом конкретном случае необходимо переместить атрибут **name** в область действия модификатора **private**, а доступ к свойствам и методам объекта **name** следует организовать через методы объекта **Person**, которые и должны помимо всего прочего перехватывать генерируемые объектом **name** исключения типа **Person::Name::Exception** и генерировать собственные исключения типа **Person::Exception**.

Исправление этого недостатка класса **Person** будет предложено выполнить читателю самостоятельно в одном из практических заданий практикума в гл. 5.

Глава 3. НАСЛЕДОВАНИЕ

3.1. Основные понятия

Одной из важнейших концепций объектно-ориентированного программирования является наследование. ***Наследование*** – это форма многократного использования программных средств, при котором

классы создаются поглощением существующих данных и поведений класса с приданием им новых возможностей.

При создании класса вместо написания абсолютно новых переменных и функций программист может обозначить факт того, что новый класс должен наследовать переменные и функции другого класса. Определенный ранее класс называется ***базовым***, а новый – ***производным***. После создания производный класс, в свою очередь, может стать базовым для другого класса. ***Прямой базовый класс*** – это базовый класс, из которого совершает явное наследование производный класс. ***Косвенный базовый класс*** наследуется из двух или более уровней выше. Если существует несколько уровней наследования, то говорят об ***иерархии классов***.

Производный класс, которому добавляется новая функциональность (а значит, и новые члены), по размеру больше своего базового класса. Он более специфичен, чем базовый класс, и представляет более специализированную группу объектов. Члены производного класса могут использовать открытые (**public**) и защищенные (**protected**) члены всех базовых классов. Члены производного класса могут иметь такие же имена, что и члены базовых классов. В этом случае для явного указания используется оператор разрешения области видимости **::** (двойное двоеточие) с именем базового класса.

В языке C++ допускается наследование от многих базовых классов. В этом случае говорят ***множественном наследовании***. При множественном наследовании объект производного класса содержит подобъекты каждого из базовых классов.

При проектировании системы классов возникает проблема выбора формы взаимодействия между классами: включить класс в качестве члена (как это делалось в предыдущей главе) или сделать его базовым. Ключевым отношением между классами, которым следует руководствоваться при выборе наследования, надо считать «является», а для включения – отношения «содержит» или «имеет». Например, «студент является индивидуумом» – типичное отношение между базовым классом, описывающим индивидуума и производным классом, описывающим студента. Другой пример – «индивидуум имеет имя» подразумевает включения объекта класса «имя» в качества атрибута класса «индивидуум», как это было сделано в примерах предыдущей главы с объектом класса **Name**, который являлся членом класса **Person**.

3.2. Базовые и производные классы

Рассмотрим производный класс с именем **Employee** (от англ. employee – служащий), для которого в качестве базового выберем класс **Person**, разработанный в двух предшествующих главах. Прежде всего, подчеркнем, что справедливость утверждения «служащий является индивидуумом» дает нам основание считать этот проект правильным. Как и прежде, для описания класса **Employee** будем использовать два файла: **Employee.h** и **Employee.cpp**.

На рис. 3.1 представлен пример простейшего класса **Employee**, наследующего класс **Person**. Собственно класс **Employee** содержит только два члена: конструктор по умолчанию и деструктор. Причем обе эти функции-члены пустые (ничего не исполняют).

```
// --  файл Employee.h
#pragma once
#include "Person.h"
class Employee:public Person // производный_класс:открытый _базовый класс
{
 public:
    Employee (void){};
   ~Employee (void){};
};
```

Рис. 3.1. Пример простейшего класса Employee, наследующего класс Person

Заголовок **class Employee: public Person** указывает на то, что класс **Employee** наследует класс **Person** или, по-другому, класс **Person** является базовым для производного класса **Employee**.

Перед именем базового класса **Person** в заголовке указан модификатор **public**, указывающий на то, что класс **Person** является открытым базовым классом. Члены производного класса могут использовать открытые (**public**) и защищенные (**protected**) члены базового класса. Более того, все открытые члены открытого базового класса становятся равноправными открытыми членами производного класса и могут применяться пользователем наравне с членами самого производного класса. Если класс **Employee** сам выступит в качестве базового, то для всех его членов, включая унаследованные, будет действовать то же правило, что в случае с классами **Employee** и **Person**. Такое наследование называется ***открытым***.

На рис. 3.2 приведен пример программы, в которой создается и используется объект класса **Employee**, а на рис. 3.3 – результат ее работы.

```
include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale>
#include "Employee.h"

int _tmain(int argc, _TCHAR* argv[])
{
 setlocale(LC_ALL, "rus");
 try
 {
  Employee s1;
  s1.setFirstName("Цветкова");
  s1.setLastName("Юлия");
  s1.setPatrName("Владимир");
  s1.setAddress("RU","333000","Красноярск",
                          "Богучаны, ул. Ерамака, 12","");
  // s1.setLastName("Ю#ия");
  // s1.setAddress("RU","33\x03","", "","");
  std::cout<< "XML = "
          << s1.toXMLLine(new c   har[Person::SIZE_XMLLINE])<<std::endl;
 }
  catch (Person::Exception e)
 {
   std::cout<< "Персональные данные:" << e.code << ":" <<e.message;
  if (e.code == Person::Exception::ERROR_ADDRESS)
     std::cout<<":"<<((Address::Exception*)e.lastexcept)->code<<":"
              <<((Address::Exception*)e.lastexcept)->message<<std::endl;
  if (e.code == Person::Exception::ERROR_NAME)
     std::cout<<":"<<((Address::Exception*)e.lastexcept)->code<<":"
           <<((Address::Exception*)e.lastexcept)->message<<std::endl;
 }
 return 0;
}
```

Рис. 3.2. Пример программы, применяющей объект класса Employee

Рис. 3.3. Результат выполнения программы на рис. 3.2

Для установки значений атрибутов объекта класса **Employee** применяются методы **setFirstName**, **setLastName**, **setPatrName** и **set-Address**, принадлежащие базовому классу **Person**, но ставшие по правилу наследования полноправными открытыми методами объекта класса **Employee**. Как и прежде, для обработки ошибок применяется механизм исключений. В try-блоке закомментирован код, вызывающий исключение типа **Person:Exception** (конечно, при условии, что инкапсулированы исключения, генерируемые членами классов

Address и **Person::Name**, см. гл. 2.3). Если убрать комментарии с этих строк, то результат работы программы будет примерно такой, как на рис. 3.4.

Рис. 3.4. Результат обработки ошибки в программе на рис. 3.2

Выполнение оператором объявления объекта **s1** класса **Employee** в рассматриваемом примере автоматически влечет выполнение конструктора класса по умолчанию. При запуске конструктора производного класса обязательно должен быть выполнен конструктор базового класса. Конструктор базового класса может быть вызван в заголовке конструктора производного класса также, как это делалось для инициализации членов класса в примерах гл. 2. В том случае, если конструктор базового класса не указан в заголовке конструктора, то компилятор сгенерирует вызов конструктора по умолчанию. При удалении производного класса вызывается деструктор, который (это делается автоматически компилятором) выполняет деструкторы всех базовых классов в обратном порядке запуска их конструкторов. На рис. 3.5 демонстрируются конструкторы, в заголовке которых указан вызов конструктора базового класса. В первом случае вызывается конструктор по умолчанию (его можно было бы не указывать), во втором – конструктор с аргументами, в третьем – копирующий конструктор.

Если в заголовке класса перед именем базового класса установлен модификатор **protected** (защищенный), то это приведет к тому, что все открытые (**public**) и защищенные (**protected**) члены базового класса будут доступны только членам производного класса и членам производного от производного класса, но недоступны для прямого вызова из программы, как это было в случае открытого наследования. Возвращаясь к примеру на рис. 3.1 и 3.2, отметим, что если бы там применялось ***защищенное наследование***, то компиляция программы на рис. 3.1 завершилась бы с ошибками, т. к. здесь используются унаследованные, но защищенные методы класса **Person.**

```
pragma once
#include "Person.h"
class Employee: public Person
{
public:
 Employee(void):Person(){};
 Employee(const Person::Name* name,    // имя
          const Address* as,           // адрес
          SEX            sex,          // пол М,Ж
          int            size_aby,     // размер автобиографии в байтах
          const char     aby[],        // автобиография
          int            add_p,        // процент надбавки
          int            length_p,     // процент за выслугу
          int            bonus_p       // процент премиальных
        ): Person(name, as, sex, size_aby, aby, add_p, length_p,
bonus_p)
          {};
Employee(const Employee& emp): Person(emp) {};
   ~Employee(void){};
}
```

Рис. 3.5. Конструкторы, вызывающие конструкторы базовых классов

Обратите внимание на копирующий конструктор класса **Epmloyee**. В нем осуществляется вызов аналогичного конструктора класса **Person**, но в качестве параметра ему передается объект класса **Epmloyee**, при том, что в спецификации копирующего конструктора класса **Person** параметр должен иметь тип **Person**. Дело в том, что в языке C++ возможно специальное преобразование производного класса к базовому, но об этом будет подробно рассказано в следующей главе.

В случае ***закрытого наследования*** (перед именем базового класса указывается модификатор **private**) действует такое же правило, что и при защищенном наследовании, но если такой производный класс выступит в качестве базового, то для нового производного класса не будут доступны член косвенного базового класса, даже если новое наследование было отрытым.

По всей видимости, защищенное и особенно закрытое наследование следует примять, когда производный класс стремится полностью инкапсулировать базовый класс.

Обратите внимание, что ни для какого вида наследования членам производного класса недоступны закрытые члены базового класса. В противном случае сама концепция закрытых членов теряет смысл.

3.3. Множественное наследование

Рассмотрим класс с именем **Performer** (от англ. performer – исполнитель), который вместе с **Person** будем использовать в качестве

базового для класса **Employee**. Этот класс будет содержать дополнительные данные о служащем некоторого учреждения (должность, даты начала и окончания работы, номер контракта, процент надбавки к базовой зарплате). Как и прежде, будем использовать отношение «является» для проверки верности этого решения.

```
// -- Performer.h
#define  MSG_ERROR_D   "день месяца"
#define  MSG_ERROR_M   "месяц"
#define  MSG_ERROR_Y   "год"
class Performer
{
public:
  const static int SIZE_JOBTITLE = 51;
  const static int SIZE_CONTRACT = 31;
  const static int SIZE_XMLLINE  = SIZE_JOBTITLE+SIZE_CONTRACT+100;
  struct Date
  {
   int dd; int mm; int yyyy;
   Date(int dd = 0, int mm = 0, int yyyy = 0)
    {this->dd = dd; this->mm = mm; this->yyyy = yyyy;};
   Date(const Date& d)
    {this->dd = d.dd; this->mm = d.mm; this->yyyy = d.yyyy;};
  };
  class Exception {//.....................};
private:
  Date Firstdate;                // первый день работы
  Date Lastdate;                 // последний день работы
  char Jobtitle[SIZE_JOBTITLE];  // должность
  char Contract[SIZE_CONTRACT];  // № контракта
  int  PercentAdd;               // % надбавки к базовой зарплате
protected:
  void setFirstdate(const Date d);
  void setLastdate(const Date d);
  void setContract(const char c[SIZE_CONTRACT]= "");
  void setJobtitle(const char t[SIZE_JOBTITLE]= "");
  void setAdd(int p);
public:
  void set(const char jt[SIZE_JOBTITLE],  const char ct[SIZE_CONTRACT],
           const Date& fd, const Date& ld, int pa);
  Performer(void);
  Performer(const char jt[SIZE_JOBTITLE],  const char ct[SIZE_CONTRACT],
            const Date& fd, const Date& ld, int pa);
  Performer(const Performer& p);
  ~Performer(void){};
  char* toXMLLine(char l[SIZE_XMLLINE]) const;
};
```

Рис. 3.6. Пример реализации класса Performer

Для класса, который будет здесь спроектирован, проверочным высказыванием будет «служащий является индивидуумом и исполнителем контракта».

На рис. 3.6 представлен класс **Performer**, который в дальнейшем планируется использовать в качестве второго базового класса для **Employee**. Класс имеет уже знакомую по предыдущим примерам структуру: вначале определяются препроцессорные и статические константы, структура **Date** (будет использоваться для представления дат), вложенный класс **Exception**, закрытые, защищенные и открытые члены, три конструктора (по умолчанию с аргументами и копирующий). Ввиду этого, обработка исключений подробно рассматривалась ранее, реализация класса **Exception** здесь не приводится. Для хранения данных в классе **Performer** используются закрытые атрибуты, назначение которых пояснено в комментариях к ним. Здесь впервые используются защищенные (**protected**) члены класса. Напомним, что защищенные члены недоступны для пользователя класса (как закрытые члены), но доступны членам производного класса. Обратите внимание, что в этом классе хранится значение процента надбавки (атрибут **PercentAdd**).

В классе **Person** тоже есть атрибут **PercentAdd**, который имеет такое же назначение. Дублирование одних и тех же данных в разных классах чаще всего говорит об ошибке проектирования. Позже мы исправим эту ошибку.

На рис. 3.7 представлен фрагмент файла **Performer.cpp**, содержащий реализацию основных функций класса **Performer.** Текст некоторых функций-членов, принцип работы которых очевиден, здесь не приводится. Обратите внимание на функцию с именем **validdate,** предназначенную для проверки корректности даты. При обнаружении ошибки в этой функции генерируется исключение.

На рис. 3.8 представлена новая версия класса **Employee,** который наследует два базовых класса – **Person** и **Performer**, причем для последнего используется защищенное наследование. Из этого следует, что все отрытые члены класса **Person** будут доступны пользователю класса **Employee**, а открытые члены класса **Performer** – нет. Вся реализация класса выполнена в одном файле **Employee.h**. Класс имеет только один собственный атрибут с именем **ID,** который предназначен для хранения идентификатора служащего. В связи с тем, что этот атрибут является закрытым, для установки его значения предназначена функция-член **setID**. Для работы с членами базового класса **Performer** в классе **Employee** имеется несколько открытых функций, работа которых сводится к вызову функций базового класса.

```
// -- Performer.cpp
#include "StdAfx.h"
#include "Performer.h"
Performer::Date  validdate(const Performer::Date d, const Performer* p)
     {
      if      (d.dd < 0 || d.dd >31)          throw
       Performer::Exception(Performer::Exception::ERROR_D, MSG_ERROR_D, p);
      else if (d.mm < 0 || d.dd > 12)         throw
       Performer::Exception(Performer::Exception::ERROR_M, MSG_ERROR_M, p);
      else  if (d.yyyy < 1900 && d.yyyy != 0) throw
       Performer::Exception(Performer::Exception::ERROR_Y, MSG_ERROR_Y, p);
      return d;
      };
void Performer::setFirstdate(const Date d)
         {this->Firstdate = validdate(d, this);};

// .... void Performer::setLastdate........

void Performer::setJobtitle(const char t[SIZE_JOBTITLE])
       {strcpy_s(this->Jobtitle,SIZE_JOBTITLE, t);};

// .... void Performer::setContract ..........

void Performer::setAdd(int p){this->PercentAdd = p;};

void Performer::set(const char jt[SIZE_JOBTITLE],
                    const char  ct[SIZE_CONTRACT],
                    const Date& fd, const Date& ld, int pa)
       {
      setJobtitle(jt);  setContract(ct); setFirstdate(fd);
        setLastdate(fd);  setAdd(pa);
       } ;
Performer::Performer(void):Firstdate(),Lastdate(),PercentAdd(0)
       {setJobtitle(); setContract();};
Performer::Performer(const char jt[SIZE_JOBTITLE],
                     const char   ct[SIZE_CONTRACT],
                     const Date& fd, const Date& ld, int pa)
   :Firstdate(fd),Lastdate(ld), PercentAdd(pa)
       {setJobtitle(jt); setContract(jt);}
Performer::Performer(const Performer& p)

:Firstdate(p.Firstdate),Lastdate(p.Lastdate),PercentAdd(p.PercentAdd)
       {setContract(p.Contract); setJobtitle(p.Jobtitle);};

 // ..... Performer::Exception::Exception......
 // ..... char* Performer::toXMLLine.....
```

Рис. 3.7. Фрагмент файла Performer.cpp

Обратите внимание, что во всех конструкторах класса **Employee** осуществляется вызов конструкторов двух базовых классов. Если этого не сделать, то компилятор все равно вызовет конструктор, но это будет конструктор по умолчанию (не имеющий параметров). Как и прежде, в копирующем конструкторе в качестве параметра копирующих конструкторов базовых классов можно указать объект производного класса.

```
// -- Employee.cpp
#include "Performer.h"
#include "Person.h"
class Employee: public Person, protected Performer
{
public:
 static const  int SIZE_XMLLINE = Person::SIZE_XMLLINE
                                + Performer::SIZE_XMLLINE+100;
 Employee(void)throw (Person::Exception, Performer::Exception)
     :ID(0),Person(), Performer() {};
 Employee(int id,
         const Person::Name* name,    // имя
         const Address* as,           // адрес
         SEX            sex,          // пол М,Ж
         int            size_aby,     // размер автобиографии в байтах
         const char     aby[],        // автобиография
         int            add_p,        // процент надбавки
         int            length_p,     // процент за выслугу
         int            bonus_p,      // процент премиальных
         const char jt[Performer::SIZE_JOBTITLE],  // должность
         const char ct[Performer::SIZE_CONTRACT],  // № контракта
         const Performer::Date& fd,   // первый день работы
         const Performer::Date& ld)   // последний день работы
         throw (Person::Exception, Performer::Exception)
   :ID(id),Person(name, as, sex, size_aby, aby, add_p, length_p, bonus_p),
    Performer(jt,ct,fd,ld,add_p) {};
Employee(const Employee& emp)throw
(Person::Exception,Performer::Exception)
  : ID(emp.ID),  Person(emp), Performer(emp){};
~Employee(void){};
void setID(int id){this->ID;};
void setFirstdate(const Date d)throw (Performer::Exception)
     {Performer::setFirstdate(d);};
void setLastdate(const Date d) throw (Performer::Exception)
     {Performer::setLastdate(d);};
void setContract(const char c[SIZE_CONTRACT]= "")
     {Performer::setContract(c);};
void setJobtitle(const char t[SIZE_JOBTITLE]= "")
     {Performer::setJobtitle(t);};
oid setAdd(int p){Performer::setAdd(p); Person::PercentAdd = p;};
void set(int id, const char jt[SIZE_JOBTITLE],
         const char ct[SIZE_CONTRACT], const Date& fd, const Date& ld,
         int pa) throw (Performer::Exception)
     {this->ID =id; Performer::set(jt,ct,fd,ld,pa);
      Person::PercentAdd = pa;};
char* toXMLLine(char l[SIZE_XMLLINE]) const  //  XML-строка адреса
     {
      strcpy_s(l,11,"<employee>");
      this->Person::toXMLLine(l+strlen(l));
      this->Performer::toXMLLine(l+strlen(l));
      strcpy_s(l+strlen(l),12,"</employee>");
      return l;
     };
private:
     int ID;                     //личный номер
}
```

Рис. 3.8. Класс Employee, наследующий два базовых класса

Инкапсуляция исключений базовых классов является желательной, но это часто приводит к большому объему дополнительных работ. Альтернативой инкапсуляции в некоторой мере может служить указание спецификации исключений для функций, в которых они могут быть сгенерированы. Обратите внимание, что в заголовках функций примера на рис. 3.8 указывается список исключений. Такая запись позволит пользователю сразу, не вникая в текст функций, разобраться с механизмом обработки ошибок для данного класса. Более подробно о спецификациях исключений для функций можно прочесть в источниках [2, 3].

Ошибка проектирования, о которой уже упоминалось выше, заставляет поддерживать два одинаковых атрибута в разных объектах. Для того чтобы некоторые переменные класса **Person** стали доступными в классе **Emoloyee**, изменим для них модификатор доступа (рис. 3.9), сделав его **protected**.

```
// -- Person.h
//.................................
class Person
{
//.................................
protected:
             int PercentAdd,                  // процент надбавки
                 PercentLength,               // процент за выслугу
                 PercentBonus;                // процент
//.................................
}
```

Рис. 3.9. Изменение модификатора доступа для атрибутов класса Person

Такое изменение делает возможным в функциях-членах **Employee::setAdd** и **Employee:set** устанавливать одновременно значение атрибутов **PercentAdd** в двух базовых классах. Но у пользователя все равно остается открытые члены класса **Person**, с помощью которых он может установить значения этого атрибута только в объекте класса **Person**, что, в конечном счете, приводит к противоречивости информации в классе **Employee**. О том, как избежать подобных ошибок, будет рассказано позже.

На рис. 3.10 демонстрируется программа, использующая класс **Employee**. В ней создается объект с именем **s1** класса **Employee**, затем с помощью отрытых методов (в том числе унаследованных от класса **Person**) устанавливаются значения его свойств. При создании нового объекта с именем **s2** применяется копирующий конструктор. На рис. 3.11 показан результат выполнения этой программы.

```
#include <cstring>
#include <locale>
#include "Performer.h"
#include "Employee.h"
int _tmain(int argc, _TCHAR* argv[])
{
 setlocale(LC_ALL, "rus");
 try
 {
  Employee s1;
  s1.setFirstName("Цветкова");
  s1.setLastName("Юлия");
  s1.setPatrName("Владимир");
  s1.setAddress("RU","333000","Красноярск",
                "Богучаны, ул. Ерамакаю 12","");
  s1.set(77,"Программист","224/88A",
         *(new Performer::Date(01,01,2005)),
         *(new Performer::Date(01,01,2009)),
         25);

  //  s1.setFirstdate(*(new Performer::Date(32, 12, 2009)));
  // s1.setAddress("RU","33\x03","Красноярск",
  //                "Богучаны, ул. Ерамака, 12","");
  Employee s2(s1);
  std::cout<< "XML = "
           << s2.toXMLLine(new
char[Employee::SIZE_XMLLINE])<<std::endl;
 }
  catch (Person::Exception e)
 {
   std::cout<< "Персональные данные:" << e.code << ":" <<e.message;
   if (e.code == Person::Exception::ERROR_ADDRESS)
     std::cout<<":"
              <<((Address::Exception*)e.lastexcept)->code<<":"
             <<((Address::Exception*)e.lastexcept)->message<<std::endl;
   if (e.code == Person::Exception::ERROR_NAME)
     std::cout<<":"<<((Address::Exception*)e.lastexcept)->code<<":"
              <<((Address::Exception*)e.lastexcept)->message<<std::endl;
 }
 catch (Performer::Exception e)
 {   std::cout<< "Исполнитель:" << e.code << ":" <<e.message; }
 return 0;
};
```

Рис. 3.10. Пример программы, использующей объекты класса Employee

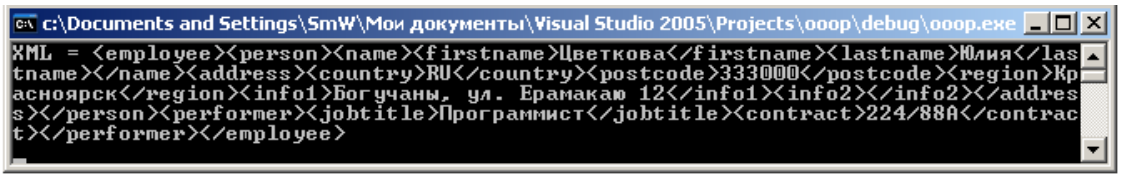

Рис. 3.11. Результат выполнения программы на рис. 3.10

Как и раньше, комментариями в try-блоке демонстрируются вызовы методов, приводящие к генерации исключений. Если снять

комментарий с первой из этих строк, то результат обработки исключения будет, как на рис. 3.12.

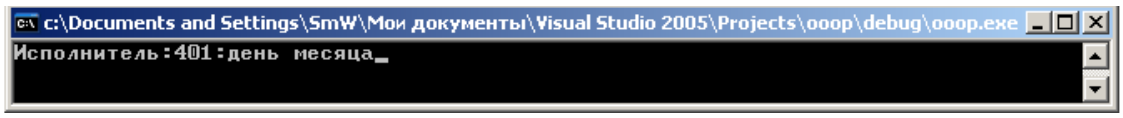

Рис. 3.12. Результат выполнения программы на рис. 3.10
в случае обработки ошибки

3.4. Принципы проектирования классов с наследованием

Объектно-ориентированное проектирование является самостоятельной темой, которая требует отдельного рассмотрения. Для ее изучения рекомендуется источник [4]. В этом разделе будет рассмотрено несколько полезных и простых правил, которыми нужно руководствоваться на первом этапе изучения объектно-ориентированного подхода к программированию. Эти правила помогут избежать сложностей, возникающих при реализации системы классов, связанных наследованием.

Первое правило уже несколько раз применялось при разработке системы классов **Employee**, **Person** и **Performer**, когда мы использовали высказывание «служащий является индивидуумом и исполнителем контракта» для проверки правильности разделения классов на базовые и производные. По-другому это правило можно сформулировать так: базовый класс по своей семантике должен обобщать производные классы.

Нарушение второго правила привело к тому, что в рассматриваемой системе классов появились атрибуты-дубликаты, поддерживать согласованные значения которых бывает сложно, а порой и невозможно. Поэтому при проектировании следует придерживаться следующего правила: в системе классов не должно быть разных атрибутов (даже если они одинаково поименованы и находятся в разных классах), функционально зависимых или имеющих одинаковую семантику. Под функционально зависимыми подразумеваются атрибуты, значение которых могут быть вычислены на основе значений других атрибутов. Все функционально зависимые атрибуты должны быть заменены на функции члены, которые вычисляют их значения.

Третье и последнее правило требует семантически правильного распределения атрибутов по иерархии классов. Придерживаться его

наиболее сложно, так как для распределения атрибутов следует хорошо понимать предметную область, которая моделируется проектируемой системой классов. Это правило тоже нарушено в рассмотренной выше системе классов.

Попытаемся на примере продемонстрировать, как следовало бы рассуждать при распределении атрибутов **BaseSalary** (базовая зарплата), **PercentAdd** (процент надбавки за должность), **PercentLength** (процент за стаж), **PercentBonus** (премиальные) в системе классов.

Пусть в компании, моделью которой будет проектируемая система классов, существуют следующие принципы вознаграждения служащих.

1. Зарплаты всех служащих рассчитываются как сумма базовой зарплаты, надбавки за должность, набавки за стаж работы в компании и премиальных. Все надбавки и премии выражаются в процентном отношении к базовой зарплате.

2. Базовая зарплата для всех служащих одинакова и служит основой для расчета всех остальных надбавок и премий.

3. Набавка за стаж – это процент от суммы базовой зарплаты, которая добавляется к вознаграждению за каждый год работы в компании. Например, если базовая зарплата равна 200 единиц, а процент надбавки за стаж равен 1%, то через 7 лет надбавка будет составлять 14 единиц. Этот процент является общим для всех служащих компании.

4. Надбавка за должность зависит только от занимаемой должности и рассчитывается как процент от суммы базовой зарплаты и надбавки за стаж.

5. Премиальные зависят только от личного вклада служащего и начисляются как процент от суммы всех других вознаграждений.

Очевидно, что два атрибута **BaseSalary** (базовая зарплата) **PercentLength** (процент надбавки за стаж) являются общими для всех служащих компании и, если значения этих атрибутов будут изменяться, то изменится зарплата всех служащих компании. Поэтому целесообразно выделить эти атрибуты в отдельный класс и сделать их статическими. На рис. 3.13 приведен пример такого класса с именем **Stimulus**. Класс содержит четыре статических метода для установки и считывания закрытых атрибутов, а также защищенный метод, позволяющий рассчитать сумму базовой зарплаты и надбавки.

Атрибут **PercentAdd** зависит от занимаемой должности и по всей видимости должен быть отражен в классе **Performer.** На рис. 3.14 представлен фрагмент файла **Performer.h.**

```
// -- Stimulus.h
class Stimulus
{
public:
                              Stimulus(void){};
  ~Stimulus(void){};
  static void Stimulus::setBaseSalary(int bs) {Stimulus::BaseSalary = bs;};
  static int  Stimulus::getBaseSalary(){return Stimulus::BaseSalary;};
  static void Stimulus::setPercentLength(int pl)
         {Stimulus::PercentLength = pl;};
  static int  Stimulus::getPercentLength()
         {return Stimulus::PercentLength; };
protected:
  int getSalary(int len)const
   {
    float s = ((float) Stimulus::BaseSalary *
               (1+ (float)(len*PercentLength)/100));
    return (int) s;
  };
 private:
  static int BaseSalary;            // базовая зарплата
  static int  PercentLength;        // % надбавки за год работы
};
// -- Stimulus.cpp
#include "StdAfx.h"
#include "Stimulus.h"
int Stimulus::BaseSalary = 0;       // базовая зарплата
int Stimulus::PercentLength;        // % надбавки за год работы
```

Рис. 3.13. Пример реализации класса Stimulus

```
// -- Stimulus.h
//..............................
#include "Stimulus.h"
class Performer: virtual protected  Stimulus
{
private:
  //..............................
  int  PercentAdd;
  //..............................
protected:
  //..............................
  void setAdd(int p){this->PercentAdd = p;};
  int getAdd()const {return this->PercentAdd;};
  int getSalary(int len) const
     {
        {float s = (float)Stimulus::getSalary(len)*
                   (1+ ((float)PercentAdd/100);
       return (int)s;
     };
  //..............................
};
```

Рис. 3.14. Пример реализации класса Performer

Обратите внимание, что теперь класс **Performer** наследует класс **Stimulus** и здесь установлено ключевое слово **virtual** перед именем базового класса. Установка **virtual** гарантирует наличие только одного экземпляра класса **Stimulus** в том случае, если класс **Performer** будет выступать базовым вместе с другими классами, которые, может быть, тоже наследуют **Stimulus**. Кроме того, в классе **Performer** присутствует функция-член **getSalary**, имеющая одинаковую сигнатуру с одноименной функцией класса **Stimulus.** Для точного определения вызываемой функции следует использовать уточненную нотацию с применением оператора разрешения пространства имен.

Последний атрибут **PercentBonus** относится к конкретному служащему и, очевидно, должен располагаться в класса **Employee** (рис. 3.15). Информация о стаже работы в компании тоже относится к конкретному работнику, поэтому атрибут **Stagelength** (величина стажа) целесообразно разместить тоже здесь.

```
// -- Employee.h
//..............................
#include "Performer.h"
#include "Person.h"
 class Employee: public Person, protected Performer
{
//..............................
private:
      int ID;                        //личный номер
      int PercentBonus;              // % премии
        int Stagelength;             // стаж полных лет
//..............................
public:
//..............................
 void setBonus(int p){this->PercentBonus = p;};
 int getBonus()const {return this->PercentBonus;};
 int getSalary() const
   {
     float  s =(float) Performer::getSalary(this->Stagelength)
                * 1+ (float) PercentBonus/100);
     return (int)s;
   };
 void setStagelength(int len ){this->Stagelength = len;};
 int getStagelength() const {return this->Stagelength;};

//..............................
}
```

Рис. 3.15. Пример реализации класса Employee

Окончательно вычислить вознаграждение служащего можно с помощью открытой функции **getSalary** класса **Employee**. На рис. 3.16 демострируется программа, использующая объект системы взаимо-

связанных классов для расчета вознаграждения служащего. На рис. 3.17 приведен результат работы этой программы.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale>
#include "Performer.h"
#include "Employee.h"
int _tmain(int argc, _TCHAR* argv[])
{
setlocale(LC_ALL, "rus");
Stimulus::setBaseSalary(200);  // базовая зарплата
Stimulus::setPercentLength(2); //  процент за стаж
try
 {
   Employee s1;
   s1.setFirstName("Никитенко");
   s1.setLastName("Елена");
   s1.setPatrName("Петровна");
   s1.setAddress("BY","222000","Минск", "ул. Свердлова 13","");
   s1.set(77,"Зав. лабораторией","224/88A",
         *(new Performer::Date(01,01,2005)),
         *(new Performer::Date(01,01,2009)),
         15);                                  // % за должность
  s1.setStagelength(7;                         // стаж
  s1.setBonus(25);                             // премия
  std::cout<< "Вознаграждение: " << s1.getSalary()<<std::endl;
 }
catch (Person::Exception e)
 {// ................................ }
catch (Performer::Exception e)
 {// ................................}
 return 0;
};
```

Рис. 3.16. Пример расчета вознаграждения служащего

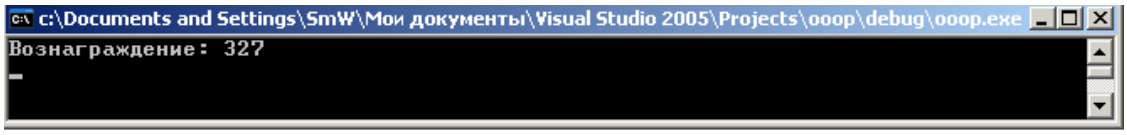

Рис. 3.17. Результат работы программы, приведенной на рис. 3.16

Глава 4. ПОЛИМОРФИЗМ

4.1. Основные понятия

Полиморфизм (от греч. «многообразность») является одним из четырех важнейших механизмов объектно-ориентированного программирования наряду с абстракцией, инкапсуляцией и наследованием.

Кратко смысл полиморфизма можно выразить фразой «один интерфейс, множество методов». С некоторыми проявлениями полиморфизма мы уже встречались в примерах предыдущих глав.

Одним из проявлений полиморфизма является возможность перегрузки функций-членов класса. Многообразие функций с одинаковым именем, но различной сигнатурой дает возможность разработчику придавать методам объекта различную функциональность в зависимости от формы вызова.

Другое проявление полиморфизма было отмечено в примерах, демонстрирующих наследование классов. В копирующих конструкторах производного класса в соответствии с общим правилом вызываются копирующие конструкторы базовых классов. Копирующий конструктор всегда имеет один параметр с тем же типом, что и класс, который он представляет. В примере класса **Employee** (рис. 3.8) копирующий конструктор принимает параметр типа **Employee** и вызывает копирующие конструкторы своих базовых классов **Person** и **Performer**. В качестве параметра им передаются не объекты типа **Person** или **Performer**, как это описано в сигнатуре конструкторов, а производный тип **Employee**, который принимается конструкторами и обрабатывается так, как будто это объекты, соответствующие сигнатуре. Говорят, что объекты класса **Employee** были приведены к типам объектов базовых классов.

4.2. Преобразование типов

На рис. 4.1 приведен фрагмент программы, использующей класс **Employee**, разработанный в предыдущих главах. Вначале программы создается и инициализируется с помощью сеттеров объект **s1** класса **Employee**.

Далее демонстрируется создание нового объекта класса **Person** из объекта класса **Employee** с помощью копирующего конструктора

(комментарий с меткой 1). Объект **pers1** – это объект класса **Person**, который получен как копия базового объекта класса **Person**, входящего в состав производного объекта **s1** класса **Employee**.

```
#include "stdafx.h"
#include <iostream>
#include <cstring>
#include <locale>
#include "Performer.h"
#include "Employee.h"
int _tmain(int argc, _TCHAR* argv[])
{
  setlocale(LC_ALL, "rus");
try
 {
  Employee s1;
  s1.setFirstName("Никитенко");
  s1.setLastName("Елена");
  s1.setPatrName("Петровна");
  s1.setAddress("BY","222000","Минск", "ул. Свердлова 13","");
  s1.set(77,"Зав. лабораторией","224/88А",
        *(new Performer::Date(01,01,2005)),
        *(new Performer::Date(01,01,2009)),
        15);

  Person    pers1(s1);                    //  1
  Person  *ppers2 =  &s1;                 //  2 возможно
  ppers2->setFirstName("Комарова");
  Employee *ps2 = (Employee*)  ppers2;    //  3  возможно
  Employee *ps3 = (Employee*)  &pers1;    //  4  допускается, но опасно
  // Performer perf1(s1);                 //  ошибка компиляции
  // Performer *pperf2 = &s1;             //  ошибка компиляции
  std::cout<<pers1.toXMLLine(new
char[Person::SIZE_XMLLINE])<<std::endl;
  std::cout<<"---------------------------------------"<<std::endl;;
  std::cout<<ppers2->toXMLLine(new
char[Person::SIZE_XMLLINE])<<std::endl;
  std::cout<<"---------------------------------------"<<std::endl;;
  std::cout<<ps2->toXMLLine(new
char[Employee::SIZE_XMLLINE])<<std::endl;
}
// ..............................................
```

Рис. 4.1. Приведение типа объекта производного класса Employee к объекту базового класса Person

Комментарием с меткой 2 отмечен оператор инициализации указателя **ppers2** на объект класса **Person**. Справа от оператора присваивания стоит указатель на адрес **Employee**. В результате преобразования значением указателя **ppers2** является адрес объекта класса **Person**, встроенного в объект **s1** класса **Employee**.

Строки, имеющие комментарии с метками 3 и 4, демонстрируют обратное преобразование из указателя на класс **Person** в указатель

на класс **Employee**. В случае с меткой 3 преобразование будет естественным, т. к. объект **s1** существовал раньше и после выполнения преобразования указатель **ps2** ничем не будет отличаться от адреса **&s1**. В случае с меткой 4 все не так однозначно, т. к. **pers1** – это новый объект класса **Person**, «расширение» которого до **Employee** может привести к ошибкам исполнения, если не предпринять никаких мер. На рис. 4.2 представлен результат выполнения программы на рис. 4.1.

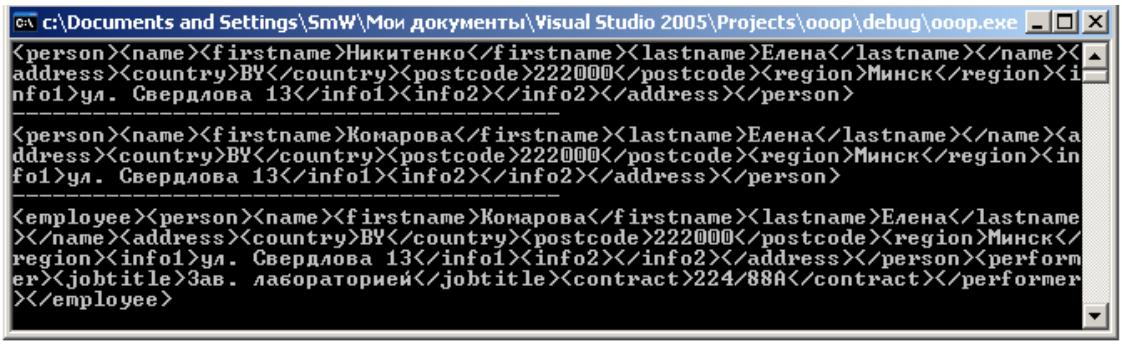

Рис. 4.2. Результат выполнения программы, представленной на рис. 4.1

Если снять комментарий с хотя бы одной из двух строк внутри try-блока, это вызовет ошибку компиляции. Подобные преобразования за пределами класса **Employee** возможны только для открытых (**public**) базовых классов.

В случае защищенного наследования (**protected**) подобные преобразования возможны в членах производного класса и в членах его наследников.

Для закрытого наследования (**private**) такие преобразования разрешены только в членах производного класса.

4.3. Виртуальные функции

Поясним концепцию виртуальных функций на примере. Для этого сначала выполним небольшую доработку класса **Stimulus**, добавив новую открытую функцию-член **toXMLLine** (рис. 4.3), а затем разработаем новый класс с именем **Antiq**, представленный на рис. 4.4. Класс **Antiq** имеет один конструктор по умолчанию, пустой деструктор и одну открытую функцию-член с именем **toXMLLine**, формирующую простую строку. И, наконец, внесем небольшие изменения в классы **Employee**, **Person**, **Performer** и **Stimulus** таким образом, как это показано на рис. 4.5.

```
//-- Stimulus.h
#pragma once
// ..............................................
#include <cstring>
class Stimulus
{
// ..............................................
public:
   char* toXMLLine(char l[SIZE_XMLLINE]) const;
// ..............................................
};

//---------------------------------------------------------------------
//-- Stimulus.cpp

include "stdafx.h"
#include <iostream>
#include "Common.h"
#include "Stimulus.h"
int Stimulus::BaseSalary = 0;
int Stimulus::PercentLength;
char* Stimulus::toXMLLine(char l[SIZE_XMLLINE]) const
{
 char buf[20];
 strcpy_s(l,11,"<stimulus>");
 Common::xml(l,"<basesalary>",
          10,_itoa(this->BaseSalary,buf,10), "</basesalary>" );
 Common::xml(l,"<percentlength>",
          10,_itoa(this->PercentLength,buf,10), "</percentlength>" );
 strcpy_s(l+strlen(l),12,"</stimulus>");
 return l;
};
```

Рис. 4.3. Новая открытая функция в классе Stimulus

```
// -- Antiq.h
#include <cstring>

class Antiq
{
    public:
      const static int SIZE_XMLLINE = 101;
    Antiq(void){};
     ~Antiq(void){};
  char* toXMLLine(char l[SIZE_XMLLINE]) const
    {
     l[0]= '\x00';
     strcpy_s(l,8,"<antiq>");
     strcpy_s(l+strlen(l),9,"</antiq>");
     return l;
    };
};
```

Рис. 4.4. Класс Antiq

```
//-- Employee.h
// ....................................
class Employee: public Person, public Performer
  {
    //....................................
  };
//-- Person.h
//....................................
class Person:  virtual public Antiq
  {
    //....................................
  };
//-- Performer.h
//....................................
class Performer:  public  Stimulus
  {
    //....................................
  };
//-- Stimulus.h
class Stimulus: virtual public Antiq
  {
     //....................................
  };
// ....................................
```

Рис. 4.5. Изменения в системе классов

Изменения в классах **Employee**, **Person**, **Performer** и **Stimulus** заключаются в том, что все наследование стало открытым, а также добавился еще один уровень иерархии наследования для классов **Person** и **Stimulus**. Эти два класса теперь наследуют новый класс **Antiq**. Обратите внимание, что в описании производных классов **Person** и **Stimulus** перед именем базового класса указано ключевое слово **virtual**. Как уже отмечалось, такая запись гарантирует наличие одного экземпляра объекта класса **Antiq** в общей системе объектов.

На рис. 4.6 предлагается фрагмент программы, демонстрирующей преобразования (проявление полиморфизма) объекта класса **Employee** к объектам базовых классов с ожидаемым и естественным результатом на рис. 4.7.

В этой программе преобразуется указатель на объект **Employee** поочередно в указатели на объекты **Person**, **Performer**, **Stimulus** и **Antiq**. После этого они используются для вызова метода с именем **toXMLLine**, присутствующего в каждом из перечисленных объектов. Результат работы программы (рис. 4.7) должен убедить, что при каждом вызове этого метода вызывается именно тот его экземпляр, который соответствует типу указанного объекта.

Сделаем функцию-член **toXMLLine** класса **Performer** виртуальной. Для этого следует просто добавить в объявление этой функции ключевое слово **virtual**, как это продемонстрировано на рис. 4.8.

```
//.............................................................
int _tmain(int argc, _TCHAR* argv[])
{
 setlocale(LC_ALL, "rus");
 Stimulus::setBaseSalary(200);  // базовая зарплата
 Stimulus::setPercentLength(2); //  процент за стаж
try
 {
  Employee s1;

  s1.setFirstName("Никитенко");
  s1.setLastName("Елена");
  s1.setPatrName("Петровна");
  s1.setAddress("BY","222000","Минск", "ул. Свердлова 13","");
  s1.set(77,"Зав. лабораторией","224/88A",
         *(new Performer::Date(01,01,2005)),
         *(new Performer::Date(01,01,2009)),
         15);                                  // % за должность
  s1.setStagelength(7);                        // стаж
  s1.setBonus(25);                             // премия
  Person    p1(s1);
  Person    *pperson1   =  &s1;
  Performer *pperformer =  &s1;
  Stimulus  *pstimulus =   &s1;
  Antiq     *pantiq    =   &s1;
  std::cout<<s1.toXMLLine(new
char[Employee::SIZE_XMLLINE])<<std::endl;
  std::cout<<"----------------------------------------"<<std::endl;;
  std::cout<<pperson1->toXMLLine(new char[Person::SIZE_XMLLINE])
           <<std::endl;
  std::cout<<"----------------------------------------"<<std::endl;;
  std::cout<<pperformer->toXMLLine(new char[Performer::SIZE_XMLLINE])
           <<std::endl;
  std::cout<<"----------------------------------------"<<std::endl;;
  std::cout<<pstimulus->toXMLLine(new char[Stimulus::SIZE_XMLLINE])
           <<std::endl;
  std::cout<<"----------------------------------------"<<std::endl;;
  std::cout<<pantiq->toXMLLine(new
char[Antiq::SIZE_XMLLINE])<<std::endl;
}
//.............................................................
```

Рис. 4.6. Пребразование объекта класса Employee к объектам базовых классов

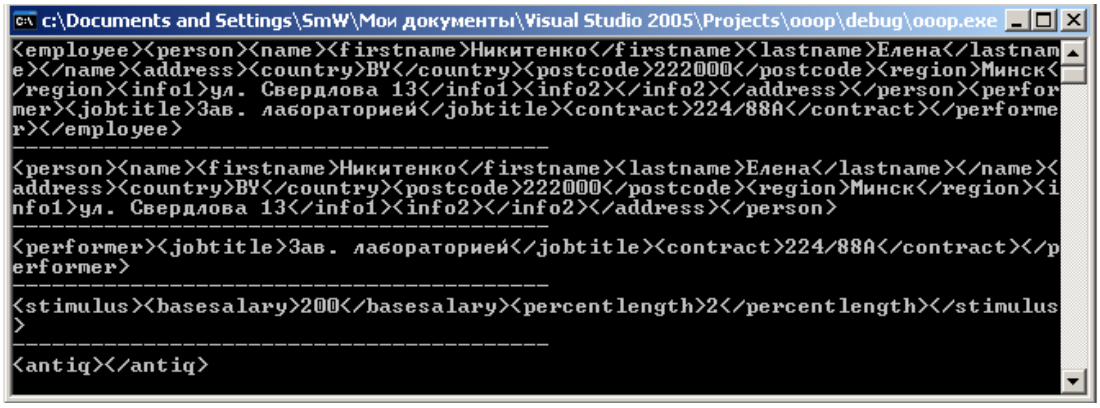

Рис. 4.7. Результат работы программы на рис. 4.7

```
// -- Performer.h
//.....................................................................
class Performer:  public  Stimulus
{
  //...................................................................
public:
  //...................................................................
  virtual char* toXMLLine(char l[SIZE_XMLLINE]) const;
  //...................................................................
};
```

Рис. 4.8. Объявление виртуальной функции toXMLLine

Если теперь выполнить программу на рис. 4.6, то результат изменится и будет таким, как на рис. 4.9.

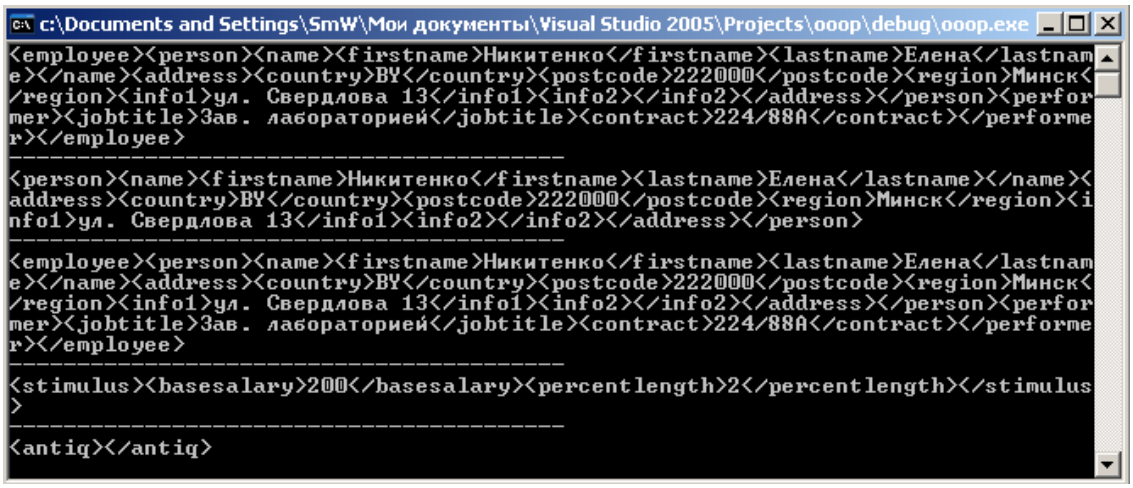

Рис. 4.9. Результат работы программы на рис. 4.7
(виртуальная функция Performer::toXMLLine)

Проанализировав результат на рис. 4.9, несложно заметить, что вместо функции **Performer::toXMLLine** выполнилась функция **Employee::toXMLLine**. Иначе говоря, произошло замещение одной функции на другую.

Продолжим исследование и сделаем виртуальной функцию **Stimulus:: toXMLLine**. Выполнив программу на рис. 4.7, получим результат, как на рис. 4.10.

Теперь замещенными оказались две функции: **Performer::toXMLLine** и **Stimulus::toXMLLine**. Причем если функцию **Performer::toXMLLine** сделать не виртуальной, результат все равно будет таким же, как на рис. 4.9. Это происходит потому, что функция **Performer::toXMLLine** теперь наследует виртуальность от функции **Stimulus::toXMLLine** базового класса.

Теперь не сложно предсказать результат, который получится в результате превращения функции **Antiq::toXMLLine** в виртуальную функцию. Действительно, все классы в итоге наследуют класс **Antiq**, а значит, и все функции **toXMLLine** будут замещены одноименной функцией производного класса, находящегося на самом верху иерархии классов (рис. 4.11).

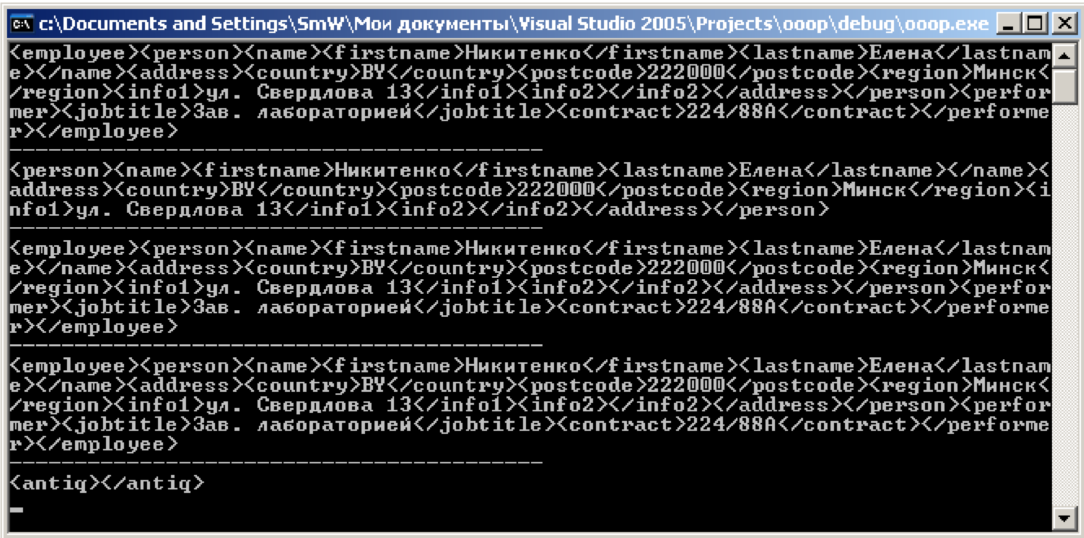

Рис. 4.10. Результат работы программы на рис. 4.7
(виртуальная функция Stimulus::toXMLLine)

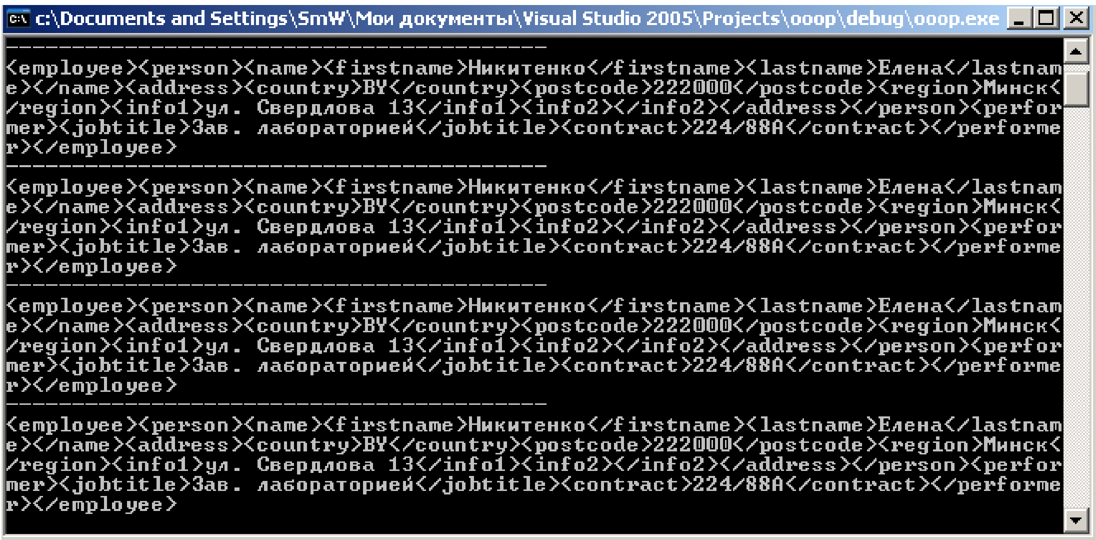

Рис. 4.11. Результат работы программы на рис. 4.7
(виртуальная функция Antiq::toXMLLine)

Механизм виртуальных функций необходим в тех случаях, когда некоторые функции-члены базовых классов следует изменить

на другие, определенные в производном классе. При этом остается возможность вызвать определенный метод базового класса, но для этого необходимо точно квалифицировать имя вызываемой функции, как это сделано на рис. 4.12.

```
//.....................................................
Employee s1;
//.....................................................
Person    p1(s1);
Person    *pperson1   =  &s1;
Performer *pperformer =  &s1;
Stimulus  *pstimulus =   &s1;
Antiq     *pantiq   =    &s1;

std::cout<<s1.Person::toXMLLine(new   char[Person::SIZE_XMLLINE])
          <<std::endl;
std::cout<<"---------------------------------------"<<std::endl;;
std::cout<<pperson1->Person::toXMLLine(new char[Person::SIZE_XMLLINE])
         <<std::endl;
//.....................................................
```

Рис. 4.12. Точная квалификация имени функций-членов

Независимо от того, каким образом в системе классов расставлены виртуальные функции, в этой программе всегда будут вызываться функции **Person::toXMLLine**.

4.4. Абстрактные классы

Часто разработчику бывает необходимо придать группе классов одинаковые свойства. Для решения этой задачи применяют абстрактные классы. ***Абстрактный класс*** – это класс, обладающий хотя бы одной чисто виртуальной функцией.

Для примера превратим класс **Aniq**, используемый в предыдущем разделе, в абстрактный класс и объясним, как его можно применить. На рис. 4.13 приведен преобразованный в абстрактный класс **Aniq**.

Обратите внимание, что в объявлении двух функций **toXMLLine** и **getMaxXMLSize** используется инициализатор **=0**. Он и является признаком чисто виртуальных функций, которая не имеет тела. И весь ее смысл заключается в определении сигнатуры функций производного класса.

Абстрактный класс можно использовать только как определение интерфейса и в качестве базы для других классов. Дело в том, что если абстрактный класс указан в качестве базового для какого-нибудь

класса, то компилятор «заставит» программиста создать для производного класса функции-члены, в точности соответствующие сигнатуре чисто виртуальных функций. Все классы, которые наследуют абстрактный класс, обладают общим набором функций-членов, т. е. они имеют одинаковый интерфейс, спецификацией которого является абстрактный класс.

```
//-- Antiq.h
class Antiq
{
public:
 Antiq(void){};
 ~Antiq(void){};
 virtual char* toXMLLine(char l[]) const = 0;
 virtual int   getMaxXMLSize() const = 0;
};
```

Рис. 4.12. Абстрактный класс Antiq

В связи с тем, что класс **Antiq** наследуют классы **Person**, **Performer Stimulus** и **Employee**, все эти классы будут обладать одинаковым интерфейсом, описанным абстрактным классом **Antiq**.

Глава 5. ПРАКТИКУМ

5.1. Предисловие

Выполнение предлагаемых в этом практикуме заданий полностью основывается на материале данного пособия. Более того, для удобства поиска материала, необходимого для выполнения работ, структура практикума в основном совпадает со структурой пособия. Выполнение некоторых заданий опирается на результат, полученный в предыдущих заданиях. Поэтому последовательность их выполнения является важной.

Для эффективного выполнения заданий потребуется компьютер с установленной операционной системой Windows XP SP2 и среда Microsoft Visual Studio не старше 2005 г. Предполагается, что читатель имеет начальные навыки программирования на языке C++ и применения отладчика Microsoft Visual Studio.

5.2. Практическая работа № 1. Свойства классов

5.2.1. Разработка простейшего класса

Задание 1. Запустите Visual Studio и создайте новый проект типа Win32 Console Application с именем **oop**. Добавьте к проекту два файла с именами: **Person.h** и **Person.cpp**. В пустой файл **Person.cpp** добавьте следующую директиву препроцессора.

```
#include "stdafx.h"

class Student:public Person // производный_класс:открытый _базовый класс
{
 public:
    Student(void){};
   ~Student(void){};
};
```

Откомпилируйте проект и убедитесь в отсутствие ошибок компиляции.

Задание 2. Разработайте класс с именем **Person**, содержащий три строковых атрибута с именами **Firstname** (имя), **Surname** (фамилия), **Patrname** (отчество) без указания модификаторов доступа. Поместите текст описания класса в файл **Person.h**. В файл **oop.cpp** включите следующую директиву препроцессора.

```
#include "Person.h"

class Student:public Person // производный_класс:открытый _базовый класс
{
 public:
    Student(void){};
   ~Student(void){};
};
```

Откомпилируйте проект и убедитесь в отсутствие ошибок компиляции.

Задание 3. Создайте объект типа **Person** с именем **p1** с помощью объявления и новый объект класса **Person** с помощью оператора **new**, поместив указатель на объект в переменную с именем **pp2**. Откомпилируйте проект и убедитесь в отсутствии ошибок компиляции. Запустите программу **oop** на выполнение в режиме **Debug** и с помощью отладчика Visual Studio исследуйте память, занимаемую двумя объектами, определите ее размер. Добавьте в программу оператор **sizeof** для определения размера памяти, занимаемой этими объектами.

Задание 4. Применяя оператор **.** (точка) и оператор → (стрелка), с помощью стандартной функции **strcpy_s** попытайтесь заполнить массивы **Firstname**, **Surname**, **Patrname** своими данными. Откомпилируйте проект, выпишите ошибки и объясните полученный результат.

5.2.2. Управление доступом

Задание 5. Установите модификаторы **public**, **protected** и **private** для разных атрибутов класса **Person**. Попытайтесь снова откомпилировать проект. Объясните полученные сообщения компилятора.

Задание 6. Установите модификаторы **private** для всех атрибутов класса **Person** и разработайте открытые геттереры и сеттеры для этих атрибутов.

5.2.3. Разработка конструкторов и деструктора класса

Задание 7. Разработайте конструктор по умолчанию и деструктор. Проследите с помощью отладчика момент вызова конструктора и деструктора.

Задание 8. Разработайте конструктор с аргументами, позволяющий проинициализировать все свойства объекта. Убедитесь в работоспособности проекта.

Задание 9. Разработайте копирующий конструктор. Убедитесь в работоспособности проекта.

5.2.4. Применение ключевого слова this

Задание 10. Назовите все параметры конструктора с аргументами теми же именами, что соответствующие атрибуты. Используйте ключевое слово **this**, чтобы избежать двусмысленности в работе конструктора.

5.2.5. Применение константных членов класса

Задание 11. Сделайте константными те члены класса **Person**, для которых это возможно. Объясните выбор и проверьте работоспособность приложения.

5.2.6. Применение статических членов класса

Задание 12. Разработайте систему статических переменных и констант, позволяющую подсчитать количество существующих на данный момент объектов типа **Person**.

Задание 13. Разработайте систему статических переменных и констант, позволяющую пронумеровать все объекты класса **Person**. Причем номер должен храниться как константный атрибут этого класса.

5.3. Практическая работа № 2. Технология инкапсуляции

5.3.1. Применение вложенных объектов

Задание 14. Разработайте самостоятельно класс с именем **Address**. За основу можно взять одноименный класс, разработанный в разд. 2. Класс должен быть внешним по отношению к классу **Person**. Создайте в классе **Person** закрытый атрибут **address** типа **Address**. Обеспечьте его инициализацию в конструкторах класса **Person** и доступ к нему с помощью открытых функций-членов класса **Person**. Проверьте работоспособность приложения.

5.3.2. Применение вложенных классов

Задание 15. Разработайте самостоятельно класс с именем **Name**. За основу можно взять одноименный класс, разработанный в разд. 2. Класс должен быть внутренним по отношению к классу **Person**. Создайте в классе **Person** закрытый атрибут **name** типа **Name**. Обеспечьте его инициализацию в конструкторах класса **Person** и доступ к нему с помощью открытых функций-членов класса **Person**. Проверьте работоспособность приложения.

5.3.3. Применение механизма исключений

Задание 16. Разработайте функции контроля вводимых данных для атрибутов классов **Address** и **Name**. Обеспечьте генерацию исключений в случае обнаружения некорректных данных для инициализации или установки атрибутов этих классов. Проверьте работоспособность приложения.

Задание 17. Инкапсулируйте исключения, генерируемые функциями классов **Address** и **Name**, обеспечив только один тип исключений, которые могут быть сгенерированы объектами класса **Person**.

5.4. Практическая работа № 3. Технология наследования

5.4.1. Базовые и производные классы

Задание 18. Разработайте класс с именем **Employee** (за основу возьмите одноименный класс из примеров разд. 3), защищено наследующий класс **Person**. Исследуйте свойства объектов класса **Employee**. Сформулируйте смысл защищенного наследования.

Задание 19. Откорректируйте класс **Employee** таким образом, чтобы он закрыто наследовал класс **Person**. Исследуйте свойства объектов класса **Employee**. Сформулируйте смысл закрытого наследования.

Задание 20. Откорректируйте класс **Employee** таким образом, чтобы он открыто наследовал класс **Person**. Исследуйте свойства объектов класса **Employee**. Сформулируйте смысл открытого наследования.

5.4.2. Множественное наследование

Задание 21. Разработайте класс с именем **Performer** (за основу возьмите одноименный класс из примеров разд. 3). Откорректируйте класс **Employee** таким образом, чтобы он наследовал классы **Person** и **Performer**.

Исследуйте свойства объекта класса **Employee** при закрытом, защищенном и открытом наследовании.

Задание 22. Разработайте класс с именем **Stimulus** (за основу возьмите одноименный класс из примеров разд. 3). Сделайте его базовым для класса **Performer**. Напишите программу, демонстрирующую разницу между защищенным и закрытым наследованием на примере системы классов **Employee**, **Performer** и **Stimulus**. Объясните смысл защищенного наследования.

5.5. Практическая работа № 4. Полиморфизм объектов

5.5.1. Преобразование типов

Задание 23. Разработайте программу, демонстрирующую преобразование объекта **Employee** к объектам базовых классов. Исследуйте свойства этих преобразований и объясните их смысл.

5.5.2. Виртуальные функции

Задание 23. Разработайте класс с именем **Antiq** (за основу возьмите одноименный класс из примеров разд. 4). Сделайте его базовым для классов **Person** и **Stimulus**.

Задание 24. Разработайте программу, демонстрирующую принцип применения виртуальных функций. Объясните, в каких случаях удобно применять эту технологию.

5.5.3. Абстрактные классы

Задание 25. Преобразуйте класс **Antiq** в абстрактный. Разработайте программу, которая бы демонстрировала принцип использования общих интерфейсов, определяемых абстрактным классом.

ЛИТЕРАТУРА

1. Демидович, Е. М. Основы алгоритмизации и программирования. Язык СИ: учеб. пособие / Е. М. Демидович. – СПб.: БХВ-Петербург, 2005. – 440 с.

2. Страуструп, Б. Язык программирования С++. Специальное издание / Б. Страуструп. – М.: Бином-Пресс, 2005. – 1104 с.

3. Лишнер, Р. С++: справочник / Р. Лишнер. – СПб.: Питер, 2005. – 907 с.

4. Майер, Б. Объектно-ориентированное конструирование программных систем / Б. Майер. – М.: Русская Редакция, 2005. – 1232 с.

5. Гецци, К. Основы инженерии программного обеспечения / К. Гецци, М. Джайзайери, Д. Мандиролли. – СПб.: БХВ-Петербург, 2005. – 832 с.

6. Арчер, Т. Visual C++. NET: библия пользователя / Т. Арчер, Э. Уайтчепел. – М.: Вильямс, 2003. – 1216 с.

ОГЛАВЛЕНИЕ

Введение 3

Глава 1. Классы 6
1.1. Понятие класса 6
1.2. Исследование простейшего класса 7
1.3. Управление доступом 8
1.4. Конструкторы класса 12
1.5. Деструктор класса 16
1.6. Ссылка на себя 18
1.7. Статические члены класса 22
1.8. Константные члены класса 25

Глава 2. Инкапсуляция 31
2.1. Основные понятия 31
2.2. Вложенные объекты и классы 32
2.3. Обработка исключений 49

Глава 3. Наследование 60
3.1. Основные понятия 60
3.2. Базовые и производные классы 61
3.3. Множественное наследование 65
3.4. Принципы проектирования классов с наследованием 71

Глава 4. Полиморфизм 76
4.1. Основные понятия 76
4.2. Преобразование типов 76
4.3. Виртуальные функции 78
4.4. Абстрактные классы 84

Глава 5. Практикум 86
5.1 Предисловие к практикуму 86
5.2 Практическая работа № 1. Свойства классов 86
5.2.1. Разработка простейшего класса 86
5.2.2. Управление доступом 87
5.2.3. Разработка конструкторов и деструктора класса 87
5.2.4. Применение ключевого слова this 88
5.2.5. Применение константных членов класса 88

5.2.6. Применение статических членов класса 88
5.3. Практическая работа № 2. Технология инкапсуляции..... 88
5.3.1. Применение вложенных объектов..... 88
5.3.2. Применение вложенных классов..... 88
5.3.3. Применение механизма исключений 89
5.4. Практическая работа № 3. Технология наследования 89
5.4.1. Базовые и производные классы 89
5.4.2. Множественное наследование..... 89
5.5. Практическая работа № 4. Полиморфизм объектов..... 90
5.5.1. Преобразование типов..... 90
5.5.2. Виртуальные функции 90
5.5.3. Абстрактные классы..... 90

Литература 91

Учебное издание

Смелов Владимир Владиславович

ВВЕДЕНИЕ
В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ НА C++

Учебно-методическое пособие

Редактор *Л. Г. Кишко*
Компьютерная верстка *Д. С. Семижен*

Подписано в печать 01.04.2009. Формат 60×84$^{1}/_{16}$.
Бумага офсетная. Гарнитура Таймс. Печать офсетная.
Усл. печ. л. 5,5. Уч.-изд. л. 5,6.
Тираж 60 экз. Заказ .

Учреждение образования
«Белорусский государственный технологический университет».
220006. Минск, Свердлова, 13а.
ЛИ № 02330/0133255 от 30.04.2004.

Отпечатано в лаборатории полиграфии учреждения образования
«Белорусский государственный технологический университет».
220006. Минск, Свердлова, 13.
ЛП № 02330/0150477 от 16.01.2009.