УЧЕБНИК
СИБИРСКОГО
ФЕДЕРАЛЬНОГО
УНИВЕРСИТЕТА

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

УДК 004.438(07)
ББК 32.973.22я73
О-294

Авторы:
И. В. Баранова, С. Н. Баранов, И. В. Баженова
Е. В. Кучунова, С. Г. Толкач

Р е ц е н з е н т ы:

Л. Ф. Ноженкова, д-р техн. наук, профессор, зав. отделом прикладной информатики Института вычислительного моделирования СО РАН;

Е. Н. Бельская, канд. техн. наук, доцент Сибирского государственного университета науки и технологий имени академика М. Ф. Решетнева

О-294 **Объектно-ориентированное программирование на C++** : учебник / И. В. Баранова, С. Н. Баранов, И. В. Баженова [и др.]. – Красноярск : Сиб. федер. ун-т, 2019. – 288 с.

ISBN 978-5-7638-4034-6

Рассмотрены основные концепции объектно-ориентированного, обобщенного и событийного программирования. В качестве языка программирования выбран язык C++. Подробно изложены принципы и механизмы работы с классами и объектами, в том числе наследование, перегрузка функций и операций, виртуальные функции, шаблоны функций и классов. Большое внимание уделено работе со стандартной библиотекой шаблонов, раздельной компиляции программ и применению компонентного и событийного подходов в разработке программных приложений.

Предназначено для студентов бакалавриата, обучающихся по направлениям подготовки 01.03.01 «Математика» и 01.03.02 «Прикладная математика и информатика».

Электронный вариант издания см.:
http://catalog.sfu-kras.ru

УДК 004.438(07)
ББК 32.973.22я73

ISBN 978-5-7638-4034-6

© Сибирский федеральный университет, 2019

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Целью написания данного учебника является систематизация и изложение основных концепций и механизмов объектно-ориентированного, обобщенного, компонентно-ориентированного и событийного программирования.

Изучение учебника позволит студентам сформировать практические навыки разработки программных приложений, создания программ на языке высокого уровня, а также применения изученных инструментов и технологий для решения прикладных задач моделирования и обработки данных.

В качестве языка программирования выбран язык C++, являющийся одним из наиболее распространенных и востребованных языков программирования. Учебник знакомит не только с понятиями, принципами и методологией вышеперечисленных парадигм программирования, но и особенностями их реализации в языке C++.

Изучение основ объектно-ориентированного подхода и связанных с ним обобщенного, компонентного и событийного программирования требует предварительного освоения значительного объема теоретических знаний по конструкциям и операторам языка C++, принципам структурного программирования и алгоритмам работы с массивами, указателями, строками, структурами, файлами и динамическими структурами данных. Также предполагается достаточная квалификация в области разработки программных приложений. Поэтому учебник рекомендуется использовать не для первоначального, а продолжающего этапа обучения программированию.

Данный учебник состоит из девяти глав. В первой главе рассматриваются основные понятия и принципы объектно-ориентированного программирования (ООП), преимущества и недостатки объектно-ориентированного подхода, понятие объектно-ориентированного языка (ООЯ) программирования и требования, предъявляемые к подобным языкам. Описывается структура объектно-ориентированной программы.

Во второй главе подробно излагаются основы работы с классами и объектами, в том числе синтаксис описания класса, спецификаторы доступа к членам класса, конструкторы, деструкторы и пр.

В третьей главе рассматривается один из важнейших принципов объектно-ориентированного подхода – наследование классов. Демонстрируется синтаксис объявления наследуемых классов для ситуации одиночного и множественного наследования, управление правами доступа класса-наследника к данным родительского класса. Также дается описание правил наследования и перекрытия методов в наследуемых классах.

Четвертая глава учебника посвящена описанию механизма перегрузки функций и операций. Приводятся основные сведения о синтаксисе, способах реализации и особенностях данного механизма.

В пятой главе рассматривается один из важнейших инструментов современных языков программирования – виртуальные функции, позволяющие успешно решать задачи, в которых для родственных классов (из одной иерархии) выполняются сходные, но не одинаковые действия.

В шестой главе представлены шаблоны функций и классов – средства языка программирования, предназначенные для создания обобщенных алгоритмов, не зависящих от используемых в них типов данных. Шаблоны относятся к обобщенному программированию.

Седьмая глава учебника посвящена обзору стандартной библиотеки шаблонов – инструмента, которым должен владеть каждый профессиональный программист и который позволяет уменьшить сроки разработки программ и повысить их надежность, переносимость и универсальность. Здесь дано описание средств стандартной библиотеки шаблонов STL: контейнеров, итераторов, потоков, функциональных объектов, адаптеров и стандартных алгоритмов.

В восьмой главе излагаются основы работы с пользовательскими библиотеками (модулями) и многофайловыми проектами. Рассматриваются особенности реализации раздельной компиляции программ.

Особое внимание уделяется ключевым технологиям разработки программных продуктов, отвечающим современным требованиям качества и надежности – применению компонентного и событийного подходов в разработке программных приложений. Основные понятия и механизмы данных подходов к программированию описываются в девятой главе учебника. Программные продукты, демонстрируемые в данной главе, представляют собой формы под Windows и другие операционные системы.

Теоретический материал каждой главы проиллюстрирован практическими примерами, синтаксис которых соответствует стандарту языка C++. Примеры программ из глав 2–8 являются консольными приложениями (т. е. программами, для которых устройством ввода является клавиатура, а устройством вывода – монитор). Консольные приложения удобны как иллюстрации при рассмотрении общих вопросов программирования.

Данный учебник предназначается студентам института математики и фундаментальной информатики Сибирского федерального университета, обучающимся по программе бакалавриата по направлениям 01.03.01 «Математика» и 01.03.02 «Прикладная математика и информатика» и рекомендуется для изучения дисциплины «Программирование».

1. ОСНОВНЫЕ ПОНЯТИЯ И ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА

1.1. Основные понятия объектно-ориентированного программирования

Объектно-ориентированное программирование является одним из наиболее эффективных и востребованных подходов в современном программировании.

Объектно-ориентированное программирование представляет собой парадигму программирования, в основе которой лежат понятия объектов и классов [1]. Под термином «парадигма программирования» понимается совокупность идей и понятий, определяющих стиль написания программ, методологию разработки программного обеспечения и способ организации вычислений. Концепции, предложенные в рамках данного подхода, стали фундаментальным шагом в методологии программирования. Применение ООП позволило повысить модульность программ, улучшить структурирование кода, а также сделать процесс разработки и сопровождения сложных и объемных программ более ясным, простым и надежным.

ООП сформировалось и приобрело широкую популярность во второй половине 1980-х гг., хотя базовые понятия и идеи этой парадигмы были предложены еще в конце 1960-х – начале 1970-х гг. Основателями ООП считаются норвежские ученые К. Нигаард и О. Й. Даль. Также огромный вклад в развитие методологии ООП внесли А. Кэй, Д. Ингаллс, Г. Буч, А. Голдберг, Б. Страуструп, И. Якобсон, Т. Бадд и многие другие.

Главная идея объектно-ориентированного подхода – представление программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса.

Объект – сущность, которая объединяет в себе данные и действия, производимые над этими данными, и рассматривается как единое целое.

Данные объекта называются *полями* (или *членами-данными*).

Функции объекта, выполняющие действия над полями, называются *методами* (или *членами-функциями*).

Понятие класса является ключевым механизмом объектно-ориентированного подхода. ***Класс*** − абстрактный тип данных, определяемый пользователем (пользовательский тип данных) [2]. В классе задаются свойства и поведение какого-либо предмета или процесса в виде набора данных (аналогично структуре) и функций, выполняющих преобразование этих данных. Идея классов отражает строение объектов реального мира, поскольку каждый предмет или процесс обладает набором отличительных характеристик (свойств) и поведением. Создаваемый тип данных

имеет практически те же свойства, что и стандартные типы, и позволяет задать представление данных в памяти компьютера, множество значений, которые могут принимать величины данного типа, и операции над ними.

Как уже было отмечено выше, ***объект класса*** – это экземпляр класса (т. е. переменная или константа этого типа), который содержит конкретные данные и работает в соответствии с описанными в классе правилами [2].

Класс является своего рода формой, определяющей, какие данные и функции будут включены в объект класса. При объявлении класса не создаются никакие объекты этого класса, по аналогии с тем, что существование типа `int` еще не означает существование переменных этого типа.

Таким образом, класс представляет собой описание каких-то сходных между собой объектов (сходных по их характеристикам и возможным действиям). Поясним это на примере.

Пусть в программе требуется работать с данными о странах. Страна – это абстрактное понятие. У нее есть такие характеристики, как название, количество населения, площадь, столица, флаг и др. Для описания страны будет использоваться класс «Страна» с соответствующими полями данных. Такие страны, как Россия, Франция и Великобритания, относятся к этому классу, т. е. они являются объектами этого класса (конкретными представителями типа «Страна», или *экземплярами класса*). Хотя все три страны обладают схожими характеристиками (название, количество населения, площадь и т. д.), но каждая страна имеет свое уникальное значение для каждой характеристики (свое название, свое количество населения, размер площади и т. д.).

Одним из важных понятий, используемых в ООП, является понятие абстракции данных. ***Абстракция*** – выделение в моделируемом предмете только необходимых характеристик и методов, достаточных для описания объекта при решении поставленной задачи. Например, создавая класс для описания студента в программе работы деканата, следует выделить только необходимые его характеристики: фамилию, имя, отчество, номер зачетной книжки, группу, дату рождения, оценки по предметам. Нет смысла добавлять такие поля, как вес, цвет глаз или кличка его кота/собаки и т. д.

1.2. Основные принципы объектно-ориентированного программирования

В основе объектно-ориентированного программирования лежат три фундаментальных принципа, определяющие его методологию: инкапсуляция, наследование и полиморфизм [3].

Инкапсуляция – свойство объектно-ориентированного языка программирования, позволяющее объединить в единое целое данные и методы и скрыть данные и способы их обработки от внешнего воздействия, чтобы защитить их от случайного изменения. Этот принцип реализуется с помощью построения классов – пользовательских типов данных, объединяющих свое содержимое в единый тип и реализующих некоторые операции или методы над ним. Инкапсуляция позволяет изолировать класс от остальных частей программы и сделать его «самодостаточным» для решения конкретной задачи. Осуществляется это путем скрытия деталей реализации класса от его пользователей и предоставления интерфейса для работы с ним. *Интерфейс* – это набор методов класса, доступных для использования другими классами. К интерфейсным методам класса относятся методы, предназначенные для оперирования содержимым (значениями полей и внутренними методами). Методология ООП предполагает, что помимо различных методов, предназначенных для решения поставленной задачи, в классе обязательно должны присутствовать специальные методы, отвечающие за элементарные операции с определенным полем (так называемый интерфейс присваивания и считывания значения), которые осуществляют непосредственный доступ к полю.

Наследование является вторым важнейшим принципом ООП. ***Наследование*** – способ определения новых классов, которые наследуют элементы (данные и методы) одного или нескольких уже существующих классов, модифицируя или расширяя их [4]. Класс, от которого производится наследование, называется *базовым* или *родительским*. Новый создаваемый класс называют *производным* или *дочерним* (или *классом-потомком*). Производный класс автоматически наследует поля и методы своего родителя и может дополнять их новыми. Поскольку каждый производный класс может быть базовым для других создаваемых классов (порождать своих потомком), может быть создана целая *иерархия* классов.

Таким образом, принцип наследования обеспечивает поэтапное создание сложных классов и разработку собственных библиотек классов. Подробнее наследование будет рассмотрено в третьей главе учебника.

Полиморфизм – свойство классов, которое позволяет использовать одно имя для обозначения действий, общих для родственных классов (из одной иерархии) и гибко выбирать требуемое действие во время выполнения программы [3].

В языке программирования С++ существуют три вида полиморфизма:

1) перегрузка функций и операций,

2) виртуальные функции,

3) шаблоны функций и классов.

Перегрузка операций и функций представляет собой возможность переопределить их действия так, чтобы в зависимости от типа передаваемых данных они выполняли разные действия [4]. Так, например, перегрузка функций означает

выбор из нескольких вариантов подходящей функции по соответствию передаваемых ей параметров. Более подробно перегрузка операций и функций будет рассмотрена в четвертой главе.

Другим примером полиморфизма является механизм *виртуальных функций*. Данный механизм означает способность объектов самим определять, какие методы они должны выполнить в зависимости от того, к какому классу они относятся, причем выбор метода для вызова осуществляется во время исполнения программы. Пятая глава учебника посвящена изучению механизма виртуальных функций.

Третий вид полиморфизма – *шаблоны* – иначе называют *обобщенным программированием*. Шаблоны позволяют использовать одни и те же функции или классы для работы с данными разных типов. Так, например, шаблоны функций создают семейства родственных функций, в которых выполняются одни и те же действия (алгоритмы), но только над данными разных типов. Особенности синтаксиса и работы с шаблонами рассмотрены в шестой главе учебника.

Перегрузка функций и операций представляет собой *статистический полиморфизм*, поскольку поддерживается на этапе компиляции. Виртуальные функции и шаблоны функций и классов относятся к *динамическому полиморфизму*, поскольку реализуются при выполнении программы.

1.3. Преимущества и недостатки объектно-ориентированного подхода к программированию

Объектно-ориентированное программирование возникло в результате развития идеологии структурного программирования.

Структурное программирование – методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Данная парадигма программирования была предложена в начале 1970-х Э. Дейкстрой, дополнена Н. Виртом, Б. Мейером, Т. Хоаром и другими авторами. В соответствии с данной парадигмой любая программа строится без использования оператора безусловного перехода `goto` из трех базовых управляющих структур: последовательности, ветвления и цикла; кроме того, используются подпрограммы (функции и процедуры). Разработка программы ведется пошагово, методом «сверху вниз» [5]. Также важными свойствами структурного подхода является возможность описания собственных (пользовательских) типов данных (структур, объединений и т. д.) и группировки функций и связанных с ними данных в отдельные файлы (модули).

Следование принципам структурного программирования позволило улучшить структурированность программ, сделать тексты программ нормально читаемыми и упростить процесс разработки и сопровождения программ. Но в непрекращающемся процессе роста и усложнения программ постепенно стали выявляться недостатки структурного программирования. Перечислим основные проблемы этой методологии:

1. Данные и подпрограммы их обработки формально не связаны.

2. Нет возможности для поддержания правильного значения переменных программы, так как часто разные переменные программы хранят связанные значения, и за поддержание этой логической связности несет ответственность программист.

3. Программы получаются большие и сложные, причем структура взаимосвязей функций между собой имеет достаточно непростой вид.

4. Возникают трудности в модификации программы, так как изменение в глобальных данных может привести к изменению всех функций, работающих с ними.

5. Отделение данных от функций не позволяет описывать объекты реального мира.

Идеи структурного программирования получили свое дальнейшее развитие в ООП, причем использование объектно-ориентированного подхода позволило объединить данные и методы их обработки, упростить структуру программ, повысить модульность, добиться управляемости крупных программных продуктов и решить все перечисленные выше проблемы. Отметим основные *преимущества ООП*:

1. Предоставляет возможность описывать объекты реального мира.

2. Позволяет объединять данные и операции, которые могут выполняться над этими данными, в единое целое, а также защищает доступ к данным объекта (с помощью принципа инкапсуляции).

3. Минимизирует затраты на перепрограммирование при модифицировании существующих функций (использование наследования и полиморфизма).

4. Реализует повторное использование кода (при написании наследуемых классов и шаблонов), что уменьшает объем программ и сокращает время на написание исходного кода.

5. Снижает количество межмодульных вызовов и объем информации, передаваемой между модулями.

6. Выполняет «естественное» разбиение программного кода на отдельные модули, которое существенно облегчает его разработку.

7. Использует стандартные компоненты (обобщенные классы), что позволяет унифицировать структуру и пользовательский интерфейс программы, а также облегчает ее понимание и упрощает ее применение.

Конечно, объектно-ориентированный подход не лишен своих *недостатков*. Перечислим их:

• требуется достаточная квалификация в области программирования и разработки программных продуктов;

• необходимо освоить большой объем информации по синтаксису и механизмам объектно-ориентированного подхода;

• при разработке программы значительно увеличивается время на предварительный анализ предметной области и проектирование системы;

• плохо разработанная иерархия классов приводит к созданию сложных и запутанных программ;

• инкапсуляция снижает скорость доступа к данным;

• созданные программы часто неэффективны в отношении памяти;

• использование стандартных компонентов и алгоритмов может потребовать от программиста изучения объемных библиотек классов.

Необходимо отметить, что концепция ООП включает в себя классическое процедурное и структурное программирование, т. е. использует все основные достижения и принципы этих двух методологий: базовые конструкции программирования (последовательное выполнение операторов, ветвление и цикл); подпрограммы (функции и процедуры), модули, технологию пошаговой разработки программы и т. д.

1.4. Объектно-ориентированные языки программирования

Объектно-ориентированные языки программирования – языки, построенные на принципах объектно-ориентированного программирования [6].

На данный момент большинство существующих языков программирования являются объектно-ориентированными. Перечислим некоторые из известных языков программирования, реализующих ООП-парадигму: C++, Java, C#, Delphi, Python, Ruby, Objective-C, Perl, ActionScript, PHP, Scala, JavaScript, Lua, Visual Basic, Simula, Smalltalk, Objective Pascal, Eiffel, Ada, D и многие другие.

Впервые основные понятия и идеи ООП (объекты, классы, виртуальные методы и др.) были предложены в языке *Simula* (1967 г.). Несмотря на наличие синтаксических конструкций, используемых в современных языках ООП, язык Simula традиционно не считается объектно-ориентированным языком в каноническом смысле этого слова.

Первым широко распространенным объектно-ориентированным языком программирования стал *Smalltalk*, разработанный А. Кэйем, Д. Ингаллсом и дру-

гими в 1970-х гг. Здесь понятие класса стало основообразующей идеей для всех остальных конструкций языка (класс в Smalltalk является примитивом, с помощью которого описываются более сложные конструкции).

Наиболее распространенные объектно-ориентированные языки – *C++*, *Delphi*, *C#*, *Java*, *PHP* и др. – воплощают объектную модель Simula. Примерами языков, опирающихся на модель языка Smalltalk, являются *Python*, *Ruby*, *Objective-C*.

Как правило, объектно-ориентированный язык содержит следующие элементы:

- объявление классов с полями и методами;
- механизм наследования – порождение нового класса от существующего класса-предка (большинство ООЯ поддерживают только одинарное наследование);
- полиморфное поведение объектов за счет использования виртуальных методов (в некоторых ООЯ все методы классов являются виртуальными).

Есть языки, содержащие дополнительные средства:

- конструкторы, деструкторы;
- скрытие данных с помощью средств управления видимостью компонентов классов (модификаторы доступа: `public`, `private`, `protected`, интерфейсы и др.);
- перегрузку операций и функций;
- шаблоны функций и классов.

Хотя класс является ключевым понятием в ООП, существуют бесклассовые объектно-ориентированные языки, например, *Lua*, *Self*, относящиеся к прототипному программированию.

Прототипное программирование – это стиль объектно-ориентированного программирования, при котором отсутствует понятие класса, а наследование свойств производится путем клонирования существующего экземпляра объекта – *прототипа*.

В рамках данного учебного пособия будем изучать фундаментальные принципы, методы и механизмы объектно-ориентированного программирования на основе языка программирования C++.

Язык C++ является одним из наиболее популярных языков объектно-ориентированного программирования и позволяет создавать приложения, эффективные по объему кода и скорости выполнения. C++ был разработан Бьерном Страуструпом в начале 80-х гг. XX в. на основе процедурно-ориентированного языка C, созданного в 1972 г. Деннисом Ритчи во время работы над операционной системой Unix.

Язык C++ с небольшими изменениями сохраняет основные конструкции и элементы языка C [2]. Его главное отличие от родоначальника заключается в дополнении возможностей C механизмами объектно-ориентированного про-

граммирования. C++ поддерживает такие парадигмы программирования, как процедурное, структурное, объектно-ориентированное и обобщенное программирование [7].

Язык включает большое число операций и типов данных, средства управления вычислительными процессами, механизмы модификации типов данных и методы их обработки. C++ имеет богатую стандартную библиотеку, содержащую в себе распространенные контейнеры и алгоритмы, ввод-вывод, регулярные выражения, поддержку многопоточности и другие возможности [3]. C++ широко используется для разработки программного обеспечения, являясь одним из самых востребованных и популярных языков программирования. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также развлекательных приложений (игр).

1.5. Структура ООП-программы и парадигмы программирования, связанные с объектно-ориентированным подходом

Объектно-ориентированное программирование не связано с ходом выполнения программы, оно связано с ее организацией [4, 7].

В ***ООП-программе*** производится описание различных классов, затем создаются объекты. Программа представляет собой набор объектов, имеющих некоторое состояние (данные) и поведение (их методы). Данные каждого объекта скрыты от остальной программы.

Работа ООП-программы заключается в том, что входящие в нее объекты взаимодействуют между собой с помощью сообщений. В самых первых объектно-ориентированных языках (например, в Smalltalk) взаимодействие объектов представлялось как «настоящий» обмен сообщениями, т. е. пересылка от одного объекта другому специального объекта-сообщения. Данный подход реализован в языках программирования Smalltalk, Ruby, Objective-C, Python.

Однако данный механизм обмена сообщениями требует дополнительных затрат памяти, что не всегда приемлемо. Поэтому в большинстве современных объектно-ориентированных языков программирования под «отправкой сообщения» понимается вызов метода объекта с необходимыми параметрами. В рамках этой концепции объекты всех классов должны иметь доступные извне методы (интерфейсные методы), с помощью вызова которых и происходит взаимодействие объектов. Данный подход реализован в огромном количестве языков программирования, в том числе C++, Java, Delphi, C#, Oberon-2. Некоторые языки

используют смешанное представление этих двух подходов, например Python, CLOS.

С объектно-ориентированным программированием тесно связаны две парадигмы программирования, играющие важную роль в современной индустрии разработки программного обеспечения: компонентно-ориентированное программирование и событийно-ориентированное программирование. Компонентное программирование представляет собой следующий этап развития ООП.

Компонентно-ориентированное программирование – парадигма программирования, представляющая собой своеобразную «надстройку» над ООП и содержащая набор правил и ограничений, направленных на создание крупных развивающихся программных систем с большим временем жизни. Программная система в этой методологии – набор компонентов с хорошо определенными интерфейсами.

Компонент – независимый модуль исходного кода программы, предназначенный для повторного использования и развертывания и реализующийся в виде множества языковых конструкций (в объектно-ориентированных языках программирования – в виде классов), объединенных по общему признаку и организованных в соответствии с определенными правилами и ограничениями.

Изменения в существующую систему вносятся путем создания новых компонентов в дополнение или в качестве замены ранее существующих. Подробнее компонентное программирование рассмотрено в девятой главе.

С помощью ООП легко реализуется так называемая событийно-управляемая модель программы.

Событийно-ориентированное программирование – парадигма программирования, в которой выполнение программы определяется событиями: действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, а также событиями операционной системы. В рамках данного подхода для любой программы реализуется проверка происходящих событий и вызов соответствующих функций, в которых описана реакция на произошедшее событие (т. е. действия, которые нужно выполнить в случае, если данное событие произошло). Применение ООП в событийно-ориентированном программировании будет также рассмотрено в девятой главе.

Следует отметить, что не существует единственного самого лучшего способа создания программ. Для решения задач разного рода и уровня сложности требуется применять разные технологии программирования. В простейших случаях достаточно использовать структурный подход к программированию, особенно в ситуации, когда нет необходимости создания иерархии классов. Для реализации же сложных крупномасштабных проектов требуется существенно улучшить структурирование кода программы, повысить управляемость процессом модели-

рования, модульность и производительность программ, а также упростить все стадии разработки программы. Поэтому для решения задач такого уровня сложности следует применять более универсальные и функциональные инструменты, составляющие ООП: классы, наследование, полиморфизм, шаблоны, компоненты и т. д. Также объектно-ориентированный подход представляет собой наиболее адекватный способ написания программ, предназначенных для моделирования предметов, процессов и явлений. Он показывает отличные результаты в условиях изменения требований заказчика, обладает гибкостью в обслуживании и возможностью повторного использования кода.

Контрольные вопросы

1. Какая идея лежит в основе объектно-ориентированного подхода?
2. Перечислите основные принципы объектно-ориентированного программирования. В чем заключается их назначение?
3. Дайте определения объекта и класса. Какая между ними существует связь?
4. Что представляют собой поля и методы класса? Как они связаны между собой?
5. Приведите пример класса и его объектов.
6. Какой из принципов объектно-ориентированного подхода позволяет выполнять защиту данных класса от несанкционированного доступа другими функциями?
7. Для чего предназначен механизм наследования классов?
8. Что представляют собой объектно-ориентированные языки программирования?
9. Какие элементы входят в состав объектно-ориентированных языков программирования?
10. В чем заключаются преимущества объектно-ориентированного подхода к программированию в сравнении со структурным подходом?

2. РАБОТА С КЛАССАМИ И ОБЪЕКТАМИ

2.1. Описание класса

Напомним, что в языке C++ класс – это тип данных, определяемый пользователем (т. е. программистом). Описать класс можно с помощью ключевого слова `class` следующим образом:

```
class имя_класса {список_членов_класса};
```

Если рассматривать термин «класс» в более широком смысле слова, то вместо ключевого слова `class` могут использоваться ключевые слова `struct` и `union`.

Имя класса – это синтаксически правильный идентификатор. Довольно часто при именовании классов используют соглашение о том, что имя начинается с заглавной буквы, а также в качестве первой буквы имени используют букву `C`. В соответствии с таким соглашением классу, описывающему точку на плоскости, можно дать имя `CPoint`.

Список членов класса представляет собой объявление данных и функций, которые становятся членами этого класса. ***Поля класса*** (или ***данные-члены***) описывают атрибуты, характеризующие свойства класса, ***методы класса*** (или ***функции-члены***) описывают поведение класса и возможные действия, производимые с данными класса.

2.1.1. Спецификаторы доступа

При описании класса члены класса группируются в зависимости от их видимости [4]. Рассмотрим следующую синтаксическую конструкцию:

```
class имя_класса{
    private:
        закрытые члены класса
    public:
        открытые члены класса
    protected:
        защищенные члены класса
};
```

Здесь ключевые слова `private`, `public`, `protected`, так называемые ***спецификаторы доступа***, показывают статус доступности членов класса.

Члены класса, объявленные со спецификатором `private`, доступны только для членов этого класса и закрыты (невидимы) вне класса. Открытые члены класса (спецификатор доступа `public`) могут использоваться как другими членами класса, так и вне класса, в любом месте программы. Защищенные члены класса, объявленные со спецификатором доступа `protected`, доступны внутри класса и в производном классе (подробнее будет рассмотрено при изучении наследования). По умолчанию все члены класса являются закрытыми, т. е. они доступны только внутри класса. Последовательность использования спецификаторов доступа при описании класса не имеет значения: члены класса указываются в любом порядке.

В закрытую часть класса обычно помещают данные. Методы класса, осуществляющие интерфейс с объектами класса для пользователя, располагают в открытой части класса.

Отметим, что тип данных «структура» является частным случаем класса, все члены которого открытые, т. е. по умолчанию имеют спецификатор доступа `public`.

После описания класса возможно объявление объекта класса в качестве экземпляра этого класса. Для создания объекта используется имя класса как спецификатор типа данных:

```
имя_класса имя_объекта;
```

Например, если был определен класс `CPoint` – точка на плоскости, то можно сделать объявления:

```
CPoint point1, points[10], &point2=point1, *pPoint;
pPoint=new CPoint[5];
```

В представленном фрагменте были объявлены объект класса `CPoint` `point1`, массив объектов из 10 элементов `points`, ссылка с именем `point2` на объект `point1`, динамический массив объектов из пяти элементов с помощью указателя `pPoint`.

2.1.2. Методы класса

Методы класса, так же как и обычные функции, должны быть определены. Это можно сделать внутри описания класса, т. е. такой метод будет ***встроенной*** (`inline`) ***функцией***. Вызов `inline`-функции компилятор заменяет ее кодом. Обычно так поступают с методами, имеющими короткую длину кода (в одну строку). Другим способом определения является внешнее описание: нужно указать в описании класса прототип метода, а его определение расположить

вне класса в глобальной области видимости. В этом случае определение метода класса происходит с помощью *операции расширения области видимости* «**::**», которая позволяет связать имя класса с его методом. Синтаксис определения метода класса следующий:

```
тип_функции имя_класса::имя_функции (список_параметров)
{   тело_функции}
```

Когда объект класса создан, можно обращаться к открытым членам класса через имя объекта с использованием операций «.» (*доступ к члену класса*) и «–>» (*доступ по указателю*). Варианты обращения:

```
имя_объекта.имя_члена_класса
ссылка_на_объект.имя_члена_класса
указатель_на_объект.имя_члена_класса
```

Необходимо обратить внимание на то, что вызов методов класса происходит через объект в отличие от вызова обычной функции. Допустим, в классе `CPoint` был определен некоторый метод `method()`. Тогда

```
point1. method(); // корректный вызов
method(); // некорректный вызов
```

С учетом того что данные класса рекомендуется помещать в закрытую часть класса, возникает необходимость в методах, позволяющих получить доступ к данным, а именно установить (изменить) и получить значения полей. Как уже упоминалось в предыдущей главе, такие методы принято называть *интерфейсными*.

Рассмотрим пример простейшего класса, описывающего точку с целочисленными координатами на плоскости. Например, такой класс можно использовать для графического изображения точки на плоскости с целью рисования геометрических фигур.

Пример 2.1. Класс «Целочисленная точка»

Требуется реализовать класс «Целочисленная точка». Класс должен содержать два поля (координаты точки x и y), методы, выполняющие доступ к полям (получение и изменение значений).

Листинг программы:

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

```
class CPoint
{int x,y;  // данные класса находятся в закрытой части класса
(private по умолчанию)
 public: //открытая часть класса
   int GetX() {return x;};//метод возвращает координату x
   int GetY() {return y;};//метод возвращает координату y
   void SetX(int a){x=a;};//метод устанавливает значение x
   void SetY(int b){y=b;};//метод устанавливает значение y
};

void main()
{
   setlocale(0,"");
   char tmp[80];
   CharToOem("класс Целочисленная точка", tmp);
   SetConsoleTitle(tmp);// Заголовок окна консольного приложения
   CPoint point1, point2;// Объявление двух объектов класса
CPoint
   int a, b;
   cout<<"Введите координаты первой точки:\n ";
   cin>>a>>b;
   point1.SetX(a);// устанавливаем значение x для первой точки
   point1.SetY(b);// устанавливаем значение y для первой точки
   cout<<"Введите координаты второй точки: \n ";
   cin>>a>>b;
   point2.SetX(a);// устанавливаем значение x для второй точки
   point2.SetY(b);// устанавливаем значение y для второй точки
   cout<<"Созданы объекты-точки с координатами:\n";
   cout<<"("<<point1.GetX()<<","<<point1.GetY()<<") и
("<<point2.GetX()<<","<<point2.GetY()<<")\n";
   system("pause");
}
```

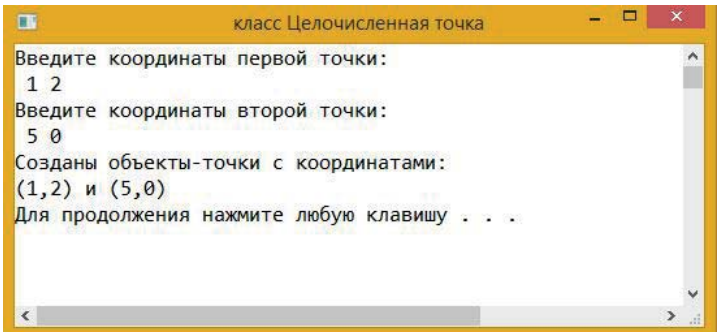

Рис. 2.1. Окно консольного приложения
«Класс “Целочисленная точка”»

Результатом работы программы будет вывод на экран, представленный на рис. 2.1.

2.1.3. Конструкторы

На практике использовать методы вида `SetX()` и `SetY()` из рассмотренного примера для того, чтобы инициализировать значения полей класса, достаточно неудобно и чревато ошибками (допустим, можно забыть инициализировать какие-то поля класса). Для этой цели существуют специальные методы, называемые конструкторами.

Конструктор – метод класса, вызываемый при создании нового объекта класса, задача которого – инициализировать объект.

Имя конструктора совпадает с именем класса, т. е. объявить конструктор в определении класса можно следующим образом:

```
имя_ класса(список_параметров);//прототип конструктора
```

Для класса можно определить несколько конструкторов. Несмотря на то что они будут иметь одинаковые имена, компилятор вызывает нужный экземпляр конструктора по *сигнатуре* (списку параметров). Также следует знать, что конструкторы не возвращают значение, но при этом они не объявляются с типом `void`, как обычные функции. Еще одно свойство конструкторов – невозможность получить адрес этой функции.

Существует три типа конструкторов:

- конструктор по умолчанию;
- конструктор с параметрами;
- конструктор копирования.

Если ни один конструктор явно не определен в классе, он создается автоматически. Такой конструктор не имеет параметров и называется *конструктором по умолчанию* (стандартным конструктором). Конструктор по умолчанию вызывается при объявлении объекта, например:

```
CPoint point1;
```

Однако конструктор по умолчанию может быть определен явно, например так:

```
CPoint::CPoint(){}// внешнее определение конструктора по
умолчанию без инициализации
```

Другой вариант:

```
CPoint::CPoint(){x=0; y=0;}//конструктор по умолчанию с
инициализацией точки в начале координат
```

Вызов конструктора может происходить явным и неявным образом, что соответствует двум формам записи:

```
имя_класса имя_объекта= имя_конструктора(список_аргументов);
имя_конструктора имя_объекта (список_аргументов);
```

Например, синтаксически правильные вызовы конструкторов:

```
CPoint point1=CPoint();
CPoint point1; // компактная форма вызова
```

Если в определении конструктора используется список параметров, то такой тип конструктора так и называется: *конструктор с параметрами*. Его непосредственное назначение – инициализация объекта определенными значениями (введенными с клавиатуры, считанными из файла, константами и т. д.). Например, в определение класса `CPoint` можно включить следующий конструктор:

```
CPoint::CPoint(int a, int b){
x=a; y=b; }
```

Такой конструктор при вызове получает в качестве аргументов значения некоторых переменных `a` и `b` и присваивает их полям (данным-членам).

При определении конструктора с параметрами можно воспользоваться параметрами по умолчанию (напомним, что такая возможность существует и для обычных функций). Допустим, в определение класса включим следующее определение конструктора:

```
CPoint::CPoint(int a=0, int b=0){
x=a; y=b;}
```

В этом случае при создании объекта без инициализирующих значений данный конструктор присвоит данным-членам нулевые значения, а при наличии значений параметров `a` и `b` объект будет инициализирован этими значениями.

Пример объявления объектов:

```
CPoint point1, point2(1,2);
```

В этом примере объект `point1` получит координаты (0, 0), а объект `point2` – координаты (1, 2). В обоих случаях будет вызываться конструктор с параметрами по умолчанию, определенный выше. Необходимо обратить внимание на то, что попытка добавить в определение класса конструктор без параметров вызовет ошибку компиляции, так как оба конструктора будут рассматриваться компилятором как конструкторы по умолчанию.

Еще одна синтаксическая возможность определения конструктора с параметрами – использование *списка инициализации*, или *функциональной формы инициализации параметров*, которая считается более предпочтительной в практике программирования на C++. В этом случае определение конструктора выглядит так:

```
имя_класса::имя_конструктора(параметр_1,параметр_N):
поле_1(параметр_1), поле_N(параметр_N){}
```

Как видим, при такой форме записи инициализирующие значения записаны в круглых скобках после имени соответствующего поля и дополнительно появляется элемент синтаксиса «:». Например, для класса `CPoint` конструктор со списком инициализации будет выглядеть следующим образом:

```
CPoint::CPoint(int a, int b): x(a), y(b) {}
```

Важным видом конструктора является *конструктор копирования*, который вызывается каждый раз, когда происходит копирование объектов данного класса. Если конструктор копирования явно не определен, он генерируется автоматически в виде конструктора побитового копирования. Конструктор копирования будет вызван в следующих случаях:

1. При использовании объекта для инициализации другого объекта этого же класса. Например, рассмотрим фрагмент кода:

```
CPoint point1(1,2);
CPoint point2(point1);
```

В первом случае был вызван конструктор с параметрами и создан объект `point1`. Во втором случае была создана копия `point1` и с ее помощью инициализирован объект `point2`.