Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

А. В. Демаков, А. А. Квасников, С. П. Куксенко

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Томск
2022

УДК 004.42(075.8)
ББК 32.973.2
Д 30

Демаков А. В., Квасников А. А., Куксенко С. П.

Д 30 Объектно-ориентированное программирование : учебное пособие / А. В. Демаков, А. А. Квасников, С. П. Куксенко. – Томск : Эль Контент, ТУСУР, 2022. – 190 с.

ISBN 978-5-4332-0302-0

В пособии представлены материалы для получения базовых знаний по объектно-ориентированному программированию на языке C++.

Для студентов высших учебных заведений, обучающихся по направлению подготовки 11.03.01 «Радиотехника», а также по другим направлениям подготовки технических специальностей.

ISBN 978-5-4332-0302-0

Оглавление

Предисловие

…Ведь я хочу только одного – быть понятым.

И. Кант. Критика практического разума

Вы, профессор, воля ваша, что-то нескладное придумали! Оно, может, и умно, но больно непонятно. Над вами потешаться будут.

М. А. Булгаков. Мастер и Маргарита

Мир вокруг нас состоит из объектов, у каждого из которых есть свои характерные черты, отличающие объект одного вида от объектов другого (вспомните, например, собаку и кошку). При этом параметры объектов одного вида в общем случае различаются, что делает каждый объект уникальным (кошка черная и кошка белая). В этом, по сути, и кроется идея объектно-ориентированного подхода к разработке программного обеспечения, где данные первичны, а методы работы с ними вторичны.

Иная ситуация при использовании процедурного подхода, в котором, как вы знаете, используется разделение программы на функции и процедуры, оперирующие данными. В этом случае данные вторичны, а функции первичны. Однако утверждать, что один подход лучше другого, мы бы не стали, т. к. каждый из них имеет свои преимущества и недостатки. Более того, каждая практическая задача может быть решена с помощью обоих подходов, а какой из них окажется эффективнее, зависит от самой задачи и навыков программиста, реализующего программу для ее решения.

Авторы придерживаются точки зрения, что каждый обучающийся программированию должен владеть навыками как процедурного, так и объектно-ориентированного программирования. Поэтому, когда мы узнали, что обучающиеся уже знакомы с процедурным программированием на языке С, мы с радостью откликнулись на предложение факультета дистанционного обучения ТУСУРа разработать данное пособие. Поскольку это позволило практически

сразу перейти к рассмотрению принципов объектно-ориентированного программирования и не останавливаться на особенностях создания циклов, функций и пр. При этом выбор языка для изложения этих принципов стал очевиден – им оказался C++, т. к. для поддержки объектно-ориентированного подхода он исторически и разрабатывался, став одним из самых популярных языков программирования.

Как показывает практика, разработка программного обеспечения невозможна без декомпозиции предметной области. Это позволяет разбить ее на несколько составных частей, каждая из которых может вестись независимо. При этом разработка должна подчиняться итерационному принципу, в рамках которого каждая новая версия становится более продуманной и законченной. Именно этот подход был использован при подготовке пособия.

Три основных раздела пособия, посвященные объектно-ориентированной парадигме программирования, были написаны авторами раздельно, с последующими обсуждениями по улучшению материала и его модернизацией. Кроме того, мы старались использовать «живой» язык изложения, приводить мнения экспертов и практические рекомендации, а также простые, но наглядные примеры, чтобы материал не было скучно изучать. В результате, мы надеемся, что наш труд не оказался напрасным и изучать представленный материал будет интересно и, что немаловажно, не скучно.

Данное пособие не претендует на полноту изложения материала из-за его ограниченного объема. Для этого есть другие масштабные произведения. При этом в пособии была предпринята попытка дать базовые знания по объектно-ориентированному программированию на языке C++, а также подстегнуть интерес читателя к саморазвитию и расширению его навыков программирования.

Отдельно хотим отметить, что часть материла намеренно вынесена в приложения, чтобы не перегружать основную часть пособия. Так, помимо чисто практических материалов, посвященных установке и настройке интегрированных сред разработки, в приложениях приведены некоторые концепции объектно-ориентированного анализа требований к программному обеспечению и его проектированию с использованием языка UML и шаблоны проектирования, к изучению которых целесообразно перейти после получения навыков объектно-ориентированного проектирования. Это позволит не только закрепить полученные знания и навыки, но и по-новому взглянуть на них и расширить.

Хотя пособие имеет относительно небольшой объем, его разработка потребовала значительных творческих усилий. Авторы выражают признательность

своему коллеге И. А. Онищенко за помощь при подготовке инструкций по установке и настройке интегрированных сред разработки и проектов в них. Эти инструкции сделали пособие более информативным.

Мы будем благодарны за любые замечания и пожелания, направленные на улучшение качества пособия. Их можно высылать по адресу: umo@2i.tusur.ru.

Приятного изучения!

Авторы

Введение

Возможно все. На невозможное просто требуется больше времени.

Дэн Браун

– Ты полетишь!
– Почему вы так думаете?
– Потому что мы верим в тебя!

М/ф «38 попугаев»

Данное пособие предназначено для изучения объектно-ориентированного программирования на языке C++. При этом предполагается, что читающий уже знаком с синтаксисом языка C и умеет на нем программировать. Поэтому в пособии нет информации о встроенных типах данных, циклах, условиях, массивах, пользовательских функциях и пр., т. е. всего того, что лежит в основе изучения синтаксиса языка C и процедурного программирования.

Существует много суждений о том, что язык C++ сложен в изучении, что есть более «удобные» и простые языки и пр. Однако, по мнению авторов данного пособия, если читатель сначала изучит C++, а затем расширит свои навыки программирования, освоив другой язык (например, C#, Python или JavaScript), то ему будет гораздо легче, чем если бы он поступил наоборот.

Кроме того, языки C и C++, имеющие много общего, используются повсеместно, например, при разработке операционных систем, мощных пакетов прикладных программ и игр, в программировании микроконтроллеров и пр. Наконец, что немаловажно, программные коды на C и C++ в процессе компиляции преобразуются в машинный код, повышая их информационную защищенность от дизассемблирования. Это, в совокупности с использованием специализированных библиотек, делает пользовательские программы более защищенными.

Пособие имеет композицию «от простого к сложному» и содержит 5 разделов и 8 приложений.

В разделе 1 приведены краткие описания онлайн-компиляторов и интегрированных сред разработки, рекомендуемых к использованию при изучении материалов пособия.

Раздел 2 предназначен для демонстрации особенностей C++ и его различий с языком C.

Разделы 3–5 посвящены рассмотрению трех главных принципов объектно-ориентированного программирования: инкапсуляции, наследования и полиморфизма.

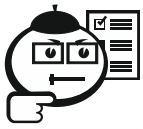
В приложениях приведены инструкции по установке и настройке интегрированных сред разработки Microsoft Visual Studio и Qt Creator и созданию пользовательских проектов в них, рекомендации по именованию элементов программного кода, а также краткие сведения о стандартной библиотеке шаблонов языка STL C++, унифицированном языке моделирования UML и шаблонах проектирования.

Соглашения, принятые в учебном пособии

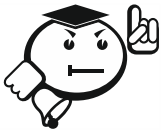
Для улучшения восприятия материала в данном учебном пособии используются пиктограммы и специальное выделение важной информации.

Эта пиктограмма означает определение или новое понятие.

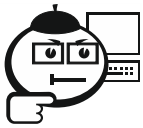
Эта пиктограмма означает «Внимание!». Здесь выделена важная информация, требующая акцента на ней. Автор может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.



Эта пиктограмма означает задание. Здесь автор может дать указания для выполнения самостоятельной работы или упражнений, сослаться на дополнительные материалы.

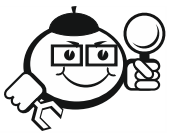


В блоке «На заметку» автор может указать дополнительные сведения или другой взгляд на изучаемый предмет, чтобы помочь читателю лучше понять основные идеи.



Эта пиктограмма означает ссылку на ресурс в сети.

Эта пиктограмма означает совет. В данном блоке можно указать более простые или иные способы выполнения определенной задачи. Совет может касаться практического применения только что изученного, или содержать указания на то, как немного повысить эффективность и значительно упростить выполнение некоторых задач.



Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Выводы

Эта пиктограмма означает выводы. Здесь автор подводит итоги, обобщает изложенный материал или проводит анализ.

Контрольные вопросы по разделу

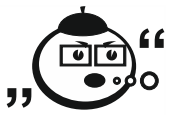
1 Онлайн-компиляторы и интегрированные среды разработки

Мнения автора могут не совпадать
с его точкой зрения.

В. Пелевин. Generation «П»

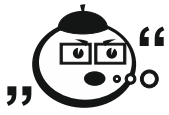
1.1 Вводные замечания

При изучении любого языка, будь то русский, английский, C++ или JavaScript, требуется постоянно практиковаться. Поэтому при изучении материала пособия для понимания принципов и получения навыков объектно-ориентированного программирования (ООП) на языке C++ нужно постоянно писать программы. Для этого рекомендуется использовать онлайн-компиляторы или интегрированные среды разработки. При этом следует помнить, что хотя и существуют универсальные приемы программирования, то, каким будет программный код, полностью зависит от его создателя.



Б. Страуструп пишет: «Программирование – это искусство выражать решения задач так, чтобы компьютер мог их осуществить. Основные усилия программиста направлены на то, чтобы найти и уточнить решение, причем довольно часто полное понимание задачи приходит лишь в ходе программирования ее решения» [1].

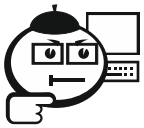
Отдельное внимание заострим на читаемости пользовательского программного кода.



Д. Босуэлл и Т. Флетчер пишут: «Мы работали в успешных компаниях, занимающихся разработкой программного обеспечения, с выдающимися инженерами, но код, с которым мы сталкивались, был далек от совершенства. Встречался нам и ужасный код. Чтение красиво написанного кода вдохновляет. Хороший код позволяет быстро разобраться в том, что он делает. Его приятно использовать, он мотивирует вас на оптимизацию собственного кода» [2].

Отсюда можно сделать вывод о том, что чем больше вы будете практиковаться в написании программ и постоянно пытаться их улучшить, изучая различные возможности языка программирования, тем качественнее они будут у вас получаться, а ваш код будет легко читаемым. Это, в целом, справедливо для изучения любого языка. Так, постоянно практикуясь в общении, например, на английском языке, вы непременно улучшите уровень понимания и свои разговорные навыки. При этом, если вы хотите стать высококвалифицированным ИТ-специалистом, то в дальнейшем от вас потребуется знание нескольких языков. Но пусть вас это не пугает, так как после изучения одного или двух языков освоение других будет происходить намного легче и быстрее.

Изучив данное пособие, вы получите базовые знания о возможностях C++. Язык C++ был выбран из-за его универсальности, быстродействия и применимости для низкоуровневого программирования различных радиоэлектронных устройств. Для подтверждения сказанного обратимся к мнению Евгения Зуева, ведущего программиста исследовательского центра Samsung.



Языки программирования: критерии выбора. Часть 1.
Языки программирования: критерии выбора. Часть 2.

1.2 Онлайн-компиляторы

Онлайн-компилятор – это инструмент, позволяющий пользователю писать свои первые программы, не заботясь об установке и настройке интегрированной среды разработки (IDE). Для его использования нужен только выход в Интернет и браузер. При этом средства, заложенные в современные онлайн-компиляторы, позволяют позиционировать их как браузерные среды разработки (*browser based IDE*). На данный момент разработано несколько универсальных онлайн-компиляторов, поддерживающих разные языки программирования. Однако авторы рекомендуют использовать GDB online Debugger [3] и Replit [4]. Их стартовые страницы приведены на рисунках 1.1 и 1.2 соответственно.

Для работы в GDB online Debugger нужно в правом верхнем углу стартовой страницы (рис. 1.1) выбрать версию стандарта языка C++. Рекомендуется использовать C++20.

Рис. 1.1 – Вид стартовой страницы GDB online Debugger

Чтобы писать программы в Replit, нужно в нижней части стартовой страницы (рис. 1.2, *б*) выбрать язык C++. После этого откроется рабочее окно для написания программ.

а)

languages

Clojure	Unlambda	JavaScript	Swift	Dart	SQLite
Haskell	CoffeeScript	Deno (beta)	Python (with Turtle)	Reason Node.js	Java
Kotlin	Scheme	Golang	Basic (beta)	Tcl	PHP CLI
QBasic	APL	C++	R	Erlang	Pyxel
Forth	Lua	C	Bash	TypeScript	Raku
LOLCODE	Ruby	C#	Crystal	Pygame	Scala (beta)
BrainF	Roy	F#	Julia	Love2D	Nix (beta)
Emoticon	Python	HTML, CSS, JS	Elixir	Emacs Lisp (Elisp)	Kaboom (beta)
Bloop	Node.js	Rust	Nim	PHP Web Server	

б)

Рис. 1.2 – Виды верхней (*а*) и нижней (*б*)
частей стартовой страницы Replit

Отметим, что после прохождения регистрации или активации входа через аккаунты сторонних сервисов эти онлайн-компиляторы позволяют создавать

пользовательские проекты, а также сохранять и делать их доступными другим пользователям. Этот процесс реализован интуитивно понятно, поэтому его описание здесь опущено.

1.3 Интегрированные среды разработки

В отличие от онлайн-компиляторов для работы в интегрированных средах разработки их сначала требуется установить на пользовательский компьютер. При этом существуют как «легкие», так и «тяжелые» среды. Последние позволяют создавать не только компьютерные программы, но и мобильные приложения, веб-сайты, веб-приложения и др. Кроме того, они содержат различные инструментальные средства, облегчающие процесс разработки, такие как отладчик исходного кода, профилировщик кода, конструктор для создания приложений с графическим интерфейсом и пр. При этом обязательными компонентами этих сред разработки являются текстовый редактор, отладчик и компилятор. Отметим, что рассмотрение особенностей создания графического интерфейса пользовательских программ выходит за рамки данного пособия.

Dev-C++

Dev-C++ – свободно распространяемая интегрированная среда разработки с открытым исходным кодом для операционных систем семейства Windows [5].

Microsoft Visual Studio

Microsoft Visual Studio (MVS) – интегрированная среда разработки программного обеспечения для операционных систем семейства Windows. Она содержит большое число инструментальных средств для упрощения процесса разработки, отладки, тестирования и повышения производительности программ [6]. В ней реализуются не только консольные приложения, но и программы со сложным графическим интерфейсом.

Qt Creator

Qt Creator – кросс-платформенная и свободно распространяемая интегрированная среда разработки [7]. Так же как и MVS, она позволяет создавать универсальные программы со сложным графическим интерфейсом.

1.4 Рекомендации по использованию средств разработки

Использовать онлайн-компиляторы или интегрированные среды разработки – это выбор читателя. Отметим, что онлайн-компиляторы прежде всего

удобны при изучении языка программирования. Так, при разработке демонстрационных программ, рассмотренных в данном пособии, авторы в основном пользовались онлайн-компиляторами. Однако они неприменимы при разработке программного обеспечения, у которого должен быть графический интерфейс пользователя, а не только ввод данных с клавиатуры. Так как авторы надеются, что читатель будет программировать не только в ходе освоения дисциплины «Объектно-ориентированное программирование», но и в дальнейшем, они рекомендуют использовать интегрированные среды разработки.

Инструкции по установке Microsoft Visual Studio или Qt Creator приведены в приложениях А и Б соответственно. Установка среды Dev-C++ является интуитивно понятной, поэтому соответствующая инструкция в пособии не приведена.

1.5 Оформление программного кода

В пособии для представления программного кода используются блоки, оформленные в виде нумерованных списков. Общий вид таких блоков:

```
1 | #include <iostream> // строка 1 программного кода
2 | int main(){ // строка 2 кода
3 |    int t; // строка 3 кода
4 |    // остальной код программы
5 | }
```

Комментарии, приведенные в блоках кода, выделены курсивом и зеленым цветом (*// строка 3 кода*). Вертикальная черта использована для отделения строк кода от их нумерации. Такое оформление блоков позволяет копировать из них программный код и помещать его в файлы, созданные в интегрированной среде разработки или онлайн-компиляторе, для его изучения и проверки работоспособности. Кроме того, это упрощает процесс редактирования при изучении кода. Для наглядности в блоках кода, а также основном тексте пособия элементы, относящиеся к программному коду, выделены другим шрифтом. `Такой шрифт применяется для элементов программного кода и элементов UML.` Строки кода в основном тексте пособия (не в блоке кода) приводятся в кавычках: «`a = b + 3;`», а элемент кода без них: `cout`.

Результаты работы используемых демонстрационных программ, как правило, не приводятся. Это связано со стремлением авторов пособия подтолкнуть читателя к анализу программного кода и проверке его работоспособности, а также экспериментальным исследованиям при его модернизации.

Именование элементов программного кода очень важно для повышения его читаемости. Поэтому в приложении В приведены соответствующие рекомендации. (В пособии есть отступления от этих рекомендаций из-за стремления авторов к минимализму при представлении программного кода, ограниченного объема пособия и использования небольших демонстрационных примеров.)

Контрольные вопросы по разделу 1

1. Чем отличается онлайн-компилятор от интегрированной среды разработки?
2. Какие интегрированные среды разработки предназначены для создания программ, работающих под управлением операционных систем семейства Windows?
3. Какие компоненты интегрированной среды разработки являются обязательными?
4. Позволяют ли онлайн-компиляторы создавать графический интерфейс пользователя?

2 Языки C и C++ как братья

– Поехали к шефу!

– В таком виде я не могу. Я должен сначала принять ванну, выпить чашечку кофе.

– Будет тебе там и ванна, будет и кофа, будет и какава с чаем... Поехали!

Х/ф «Бриллиантовая рука»

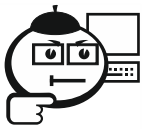
2.1 Вводные замечания

Созданием языка программирования C мировое сообщество обязано Д. Ритчи и Б. Кернигану (1970-е гг.), а разработке его «брата» C++ – Б. Страуструпу (1980-х гг.). Язык C++ первоначально назывался «C с классами» (C with classes) и представлял собой модификацию языка C, разработанную для «собственных нужд» Б. Страуструпа. За основу синтаксиса языка C++ взят синтаксис языка C. Б. Страуструп хотел сохранить совместимость с языком C (программист может использовать только некоторые возможности C++ или не использовать их вовсе), а также реализовать поддержку разных стилей программирования, в частности процедурного и объектно-ориентированного. При этом часто, но не всегда, конструкции, общие для языков C и C++, имеют одинаковую семантику или смысловую нагрузку. Поэтому, по всей видимости, правильно рассматривать язык C как подмножество языка C++.

Отдельно отметим, что цели и история создания языка C++, идеи и принципы, заложенные в него, пояснения, почему он называется именно так, а не иначе, и многое другое можно найти в книге его автора [8].

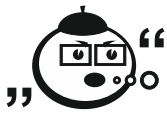
Впоследствии рассматриваемые языки программирования стали развиваться раздельно. Так, у языка C++, как и у языка C, периодически выходят обновления, которые в основном ориентированы на расширение и добавление новых функциональных возможностей языка. На момент выхода учебного пособия C++20 является самой свежей версией – выход его спецификации датируется декабрем 2020 г. В свою очередь последняя версия языка C – C17 (2017 г.).

В качестве визуального начала к разделу предлагается видео, доступное по ссылке ниже.



Бьерн Страуструп: почему я создал C++ (перевод)

Идеи, заложенные при разработке языка C++, позволяют ему до сих пор оставаться одним из востребованных языков программирования как на системном, так и прикладном уровнях, а его освоение значительно упрощается, если изучающий уже имеет навыки программирования на языке C.

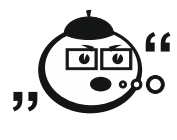


Б. Страуструп пишет: «*Подход к изучению языка C++ после изучения C приводит к ненужной потере времени и приучает студентов к неправильному стилю программирования, вынуждая их решать задачи, имея в своем распоряжении ограниченный набор средств, конструкций и библиотек. Язык C++ предусматривает более строгую проверку типов, чем язык C, а стандартная библиотека лучше соответствует потребностям новичков и позволяет применять исключения для обработки ошибок*» [1].

Авторы пособия, наоборот, убеждены, что знание синтаксиса и приемов языка C позволяет обучающимся быстрее освоить философию языка C++, т. к. они в этом случае уже знают основные типы данных, владеют навыками работы с циклами, массивами и функциями, умеют разрабатывать алгоритмы и писать программы.

Совместимость языков позволяет разрабатывать программы на C++ с использованием процедурного программирования. При этом в настоящее время язык C++ наиболее популярен из-за реализации объектно-ориентированной парадигмы программирования. Поэтому знание синтаксиса языка C («алфавита») позволяет без изучения C++ сразу перейти на новый уровень программирования и к написанию более сложных программ («сочинений»).

Кроме того, по мере изучения C++ обучающийся может выполнять сопоставление конструкций обоих языков и парадигм программирования для понимания того, какую из парадигм следует использовать при решении конкретных задач, т. к. не все программы следует писать с применением только процедурного или объектно-ориентированного программирования. Следует помнить, что «панацея» с точки зрения как парадигмы, так и языка программирования пока не изобретена. Поэтому авторам пособия больше импонирует мнение другого эксперта по C++.



Брюс Эккель в предисловии к своему труду пишет: «Язык C++, как и любой из человеческих языков, предназначен для выражения смысловых концепций. Если правильно использовать это выразительное средство, оно оказывается гораздо более удобным и гибким, чем его альтернативы (особенно с увеличением объема и сложности задач). C++ нельзя рассматривать как механический набор функциональных возможностей, многие из которых сами по себе не имеют смысла. Если речь идет о проектировании, а не о простом кодировании, необходимо использовать всю совокупность этих возможностей. Но чтобы понять C++ с этой точки зрения, нужно представлять себе основные проблемы C и программирования в целом» [9].

Сказанное выше можно расценивать как философское предисловие, т. к. освоить и понять язык можно только через практику. К ней далее и перейдем.

2.2 C и C++: явные различия

Поскольку предполагается, что читатель данного пособия уже знаком с синтаксисом языка C, то для облегчения написания первых программ на C++ сначала рассмотрим основные различия в синтаксисе этих языков, а затем – некоторые основные конструкции и новые функциональные возможности языка C++, тем самым заложив основу для успешного изучения ООП на языке C++.

Первым явным различием между языками C и C++ является считывание данных с клавиатуры и их вывод в консоль, а вторым – выделение памяти из кучи и ее освобождение. Более «сложное» различие заключается в объявлении функций, имеющих одно имя, но разные аргументы.

Сначала сравним особенности стандартного ввода/вывода данных в языках.

// C

```
1  #include <stdio.h>
2  void main(){
3      printf("-TUSUR- \n");
4      int b;
5      scanf("%d", b);
6      printf("\n");
7  }
```

```
                                    // C++
1   #include <iostream>
2   int main(){
3       std::cout << "-TUSUR-" << std::endl;
4       int b;
5       std::cin >> b;
6       std::cout << std::endl;
7       return 0;
8   }
```

Видно, что в программах на языке C++ (аналогично C) обязательной является функция `main()`. Кроме того, для предотвращения конфликтов имен идентификаторов используется пространство имен `std`. Так, все идентификаторы (переменные, функции и пр.), определенные в пределах пространства имен, доступны друг другу без необходимости дополнительного уточнения. Это позволяет объединять логически связанные области программного кода, что, в конечном счете, повышает его читаемость и масштабируемость.

Пространство имен *– это логически объединенная область программного кода, в которой определены разные идентификаторы* (*функции, переменные и пр.*).

Для использования идентификаторов за пределами их пространства имен надо обращаться к ним с указанием полного имени каждого идентификатора. Например, в строке «`std::cout<<"-TUSUR-";`» именно оператор «`::`» позволяет избежать конфликта имен.

Для объявления используемого пространства имен вводится объявление `using` и ключевое слово `namespace`. Так, для объявления использования пространства имен `std` нужно в программный код добавить строку «`using namespace std;`». В результате можно вызывать идентификаторы из пространства имен без указания его имени (`cout << "-TUSUR-";`). В противном случае вызов идентификатора должен содержать его полное имя в пространстве имен (`std::cout<<"-TUSUR-";`).

```
1   #include <iostream>
2   using namespace std; // используемое пространство имен
3   int main(){
4       cout << "-TUSUR-" << endl;
```

```
    int b;
    cin >> b;
    cout << endl;
    return 0;
}
```

Продемонстрируем создание пользовательского пространства имен и реализацию в нем пользовательской функции `sqrt()`, которая извлекает квадратный корень числа. Имя этой функции намеренно совпадает с именем стандартной функции.

```
#include <iostream>
#include <cmath> // для работы стандартной sqrt()
using namespace std;
namespace MY{ // пользовательское пространство имен
    float sqrt(float a){
       return (::sqrt(a)+1); // return (std::sqrt(a)+1);
    }
}
int main(){
    float tmp = 25.0;
    cout << sqrt(tmp) << endl; // стандартная sqrt()
    cout << MY::sqrt(tmp) << endl;
    return 0;
}
```

Здесь для типового извлечения квадратного корня использована стандартная функция `sqrt()` из заголовочного файла `<cmath>`, включающего заголовок `<math.h>` стандартной библиотеки языка C и добавляющего имена ее функций в пространство имен `std`.

Перейдем к демонстрации динамического выделения и освобождения памяти из кучи в обоих языках.

```
                                // C
#include <malloc.h>
void main(){
    int* ptrArr =(int*)malloc(10 * sizeof(int));
    free(ptrArr);
}
```

```
// C++
1  void main(){
2      int* ptrArr = new int [10];
3      delete []ptrArr;
4  }
```

Как видно, в языке C++ упрощена работа с памятью за счет добавления новых операторов (ключевых слов) `new` и `delete`. Записи стали более короткими и интуитивно понятными, не требуется подключение заголовочного файла. Кроме того, блок памяти под объект выделяется (с помощью `new`) в процессе работы программы, а ее освобождение производится «вручную» (`delete`).

Наконец, продемонстрируем возможности языка C++ по использованию функций с одинаковым именем, но разными типами данных ее аргументов и их числом (на примере объявления этих функций). Этот механизм называется *перегрузкой функций* и более подробно рассмотрен в подразд. 5.2.

```
// C
1  int calc(int);
2  float calc(int); // ошибка
3  int calc(float); // ошибка
4  int calc(float,int); // ошибка
5  float calc(int, float); // ошибка
```

```
// C++
1  int calc(int);
2  float calc(int); // ошибка
3  int calc(float); // нет ошибки
4  int calc(float, int); // нет ошибки
5  float calc(int, float); // нет ошибки
```

Различие здесь объясняется тем, что в языках C и C++ по-разному определяется сигнатура функции. В языке C сигнатура функции включает только ее имя, а в языке C++ – имя функции, число и типы данных ее аргументов. Тогда при программировании на языке C требуется изменить имена последующих функций. Например, надо заменить имя второй из них на «`int calcF(float);`», что, в сравнении с функционалом языка C++, не совсем удобно, т. к. в C++ компилятор сам выбирает нужную функцию.

В C++ также добавлена возможность использования значений аргументов функции по умолчанию. При этом если пользователь при вызове функции не укажет значение аргумента, то оно будет подставлено компилятором.

```
1  int calc(int a = 1){
2      return a + 1;
3  }
4  int main(){
5      cout << "1: " << calc(2) << endl;
6      cout << "2: " << calc() << endl;
7      return 0;
8  }
```

Небольшие различия есть и в типах данных. Например, логический тип в C – это `_Bool`, а в C++ – `bool`. Поэтому читателю рекомендуется самостоятельно ознакомиться с различиями во встроенных типах данных языков. Подробное описание с пояснениями различий синтаксисов и конструкций языков C и C++ можно найти в гл. 27 книги Б. Страуструпа [1].

2.3 Особенности синтаксиса C++

2.3.1 Потоковый ввод и вывод

Как было показано выше, в C++ для ввода и вывода информации применяются библиотека `<iostream>` и пространство имен `std`. При этом используется потоковый ввод и вывод информации. Отметим, что эта библиотека реализована с использованием объектно-ориентированной парадигмы программирования. В ней определены 4 стандартных потока: ввода (`cin`), общего вывода (`cout`), выводов об ошибках (`cerr`) и вывода о результатах работы программы (`clog`). Так как библиотека работает с потоками, в ней перегружены операторы поразрядного сдвига для вывода << и ввода >> информации соответственно. (Подробно перегрузка операторов рассмотрена в разд. 3–5.)

Примеры «простого» использования `cin >>` и `cout <<` приведены выше. Продемонстрируем, как их можно использовать для более сложного и настраиваемого ввода и вывода. Для этого используются специальные функции-манипуляторы (методы-манипуляторы), устанавливающие параметры ввода и вывода данных. Наиболее часто используемые из них приведены в таблице 2.1 с соответствующими пояснениями.

Таблица 2.1 – Манипуляторы ввода/вывода данных и их назначение

Манипулятор	**Назначение**
Настройки вывода: точность	
`precision(`*точность*`)`	Метод для задания числа значащих цифр (точность) в выводимом значении
`scientific`	Флаг использования экспоненциального вывода с числом цифр после запятой, определяемым `precision`
`fixed`	Флаг использования вывода с числом цифр после запятой, определяемым `precision` (по умолчанию)
`defaultfloat`	Флаг использования вывода с общим числом цифр, определяемым `precision`
`hexfloat`	Флаг использования шестнадцатеричного форматированного вывода числа с плавающей запятой
`boolalpha/noboolalpha`	Флаги включения/выключения вывода логических величин в текстовом/числовом виде
Настройки вывода: формат поля	
`width(`*ширина*`)`	Метод общей установки символьной ширины (общее число символов) поля вывода
`setw(`*ширина*`)`	Метод потоковой установки символьной ширины (общее число символов) поля вывода (надо `#include <iomanip>`)
`right`	Флаг правого прижатия
`left`	Флаг левого прижатия (по умолчанию)
`fill('`*символ*`')`	Метод общего заполнения пустых знаковых мест в поле вывода указанным символом
`setfill('`*символ*`')`	Метод потокового заполнения пустых знаковых мест в поле вывода указанным символом (надо `#include <iomanip>`)
`showpos/noshowpos`	Флаги включения/выключения добавления знака «+» при выводе неотрицательных чисел
`endl`	Добавление в конец потока символа новой строки `'\n'`
Настройка вывода: система счисления (надо `#include <iomanip>`)	
`setbase`	Метод задания системы счисления для вывода значений
`uppercase/nouppercase`	Флаги включения/отключения верхнего регистра
`dec`	Флаг вывода целочисленного (`int`) значения в 10-й форме (по умолчанию)
`oct`	Флаг вывода целочисленного (`int`) значения в 8-й форме
`hex`	Флаг вывода целочисленного (`int`) значения в 16-й форме
Настройки ввода	
`get(`*указатель*`, `*число*`, '`*символ*`')`	Ожидание и ввод максимального числа символов и их сохранение по указателю. Третий параметр определяет символ окончания потока ввода (по умолчанию – `'\n'`). Задание всех трех параметров может быть опущено (`cin.get()`)
`getline(`*указатель*`, `*число*`, '`*символ*`')`	Метод, аналогичный `get()`, но в `getline()` символ окончания ввода удаляется из потока ввода перед его сохранением
`read(`*указатель*`, `*число*`)`	Метод неформатированного ввода символов и сохранение (по ссылке) их заданного числа в массив
`wigth(`*ширина*`)`	Метод задания ширины (по умолчанию равна нулю) считываемой символьной строки
`clear()`	Метод очистки потока

Продемонстрируем особенности настройки точности вывода.

```
#include <iostream>
using namespace std;
int main(){
    double val = 123.456;
    cout << val << endl; // 123.456
    cout.precision( 4 );
    cout << val << endl; // 123.5
    cout << scientific << val << endl; // 1.2346e+02
    cout << fixed << val << endl; // 123.4560
    cout << defaultfloat << val<< endl; // 123.5
    cout<< hexfloat <<val<<endl; // 0x1.edd2f1a9fbe77p+6
    return 0;
}
```

Далее продемонстрируем особенности настройки формата поля вывода.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
    double val = 123.456; // 7 символов
    cout.width(10); // отступ 10-7 = 3 (символа)
    cout.fill('#');
    cout << val << endl; // ###123.456
    cout << showpos << val << endl; // +123.456
    cout << val << endl; // +123.456
    cout.width(10);
    cout.fill('#');
    cout << val << endl; // ##+123.456
    cout<<setw(9)<<setfill('!')<<val<<endl; // !+123.456
    cout << setw(8) << "val=" <<left<<noshowpos << val<<
setfill('!') << endl; // !!!!val=123.456
    cout<<"val="<<left<<setw(8)<<setfill('!')<<val<<
endl; // val=123.456!
    return 0;
}
```

Наконец, продемонстрируем использование систем счисления.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
int val{1234};
    cout << hex << val << endl; // 4d2
    cout << oct << val << endl; // 2322
    cout << dec << val << endl; // 1234
    cout<< uppercase << hex << val << endl; // 4D2
    cout << nouppercase << setbase(16)<< val<<endl;//4d2
    cout << setbase(8) << val << endl; // 2322
    cout << setbase(10) << val << endl; // 1234
    return 0;
}
```

2.3.2 Ключевые слова

В языке C++ есть много ключевых или зарезервированных слов, которых нет в языке C. Как уже упоминалось, C++20 является текущей версией языка C++. Предыдущими версиями были C++89 (1989 г.), C++03 (2003 г.), C++11 (2011 г.), C++14 (2014 г.) и C++17 (2017 г.). При этом версия C++14 в основном содержит исправленные ошибки и небольшие дополнения к версии C++11. Стоит отметить, что версии C++11 и C++14 использованы в книге [1].

Список ключевых слов языка C++ пополняется с выходом его новых версий (рис. 2.1). Кроме того, в C++ используются специальные, но не зарезервированные идентификаторы, такие как `override`, `final`, `import` и `module`, которые имеют разное значение в зависимости от контекста.

Отметим, что ключевые слова C++, которых нет в языке C, могут быть использованы в программах на языке C как пользовательские идентификаторы (имена переменных и функций, макросы и пр.). Очевидно, что если эти программы должны быть совместимы с C++, то их нельзя использовать в качестве идентификаторов.

В данном пособии в основном использована версия C++17 и лишь некоторые нововведения C++20, ведь чтобы понять их смысл, надо хорошо владеть базовыми принципами, заложенными в предыдущих версиях.

alignas	**const_cast**	int	static_assert
alignof	continue	long	**static_cast**
and	co_await	mutable	struct
and_eq	co_return	**namespace**	switch
asm	co_yield	**new**	**template**
auto	decltype	noexcept	**this**
bitand	default	not	thread_local
bitor	**delete**	not_eq	**throw**
bool	do	**nullptr**	true
break	double	**operator**	**try**
case	**dynamic_cast**	or	typedef
catch	else	or_eq	typeid
char	enum	**private**	**typename**
char8_t	**explicit**	**protected**	union
char16_t	export	**public**	unsigned
char32_t	extern	register	**using**
class	false	**reinterpret_cast**	**virtual**
compl	float	requires	void
concept	for	return	volatile
const	friend	short	wchar_t
consteval	goto	signed	while
constexpr	if	sizeof	xor
constinit	inline	static	xor_eq

Рис. 2.1 – 92 ключевых слова языка C++20

Первая группа ключевых слов включает известные ключевые слова из языка C (на рисунке 2.1 они подчеркнуты), вторая группа (`delete`, `namespace`, `new` и `using`) была использована выше, а третья – рассмотрена далее в пособии (на рисунке они выделены жирным). Наконец, ознакомление с четвертой группой ключевых слов отводится на самостоятельное изучение.

В языке C++ реализована функциональность, позволяющая явно не указывать тип переменной. Это реализуется с помощью ключевого слова `auto`. Так, компилятор, встречая это слово, сам определяет требуемый тип переменной на

основе ее значения. При этом `auto` также можно использовать вместо типа возвращаемого значения функции, тогда оно указывает компилятору, что он сам должен определить тип данных возвращаемого значения. Однако нельзя использовать `auto` вместо явного указания типа данных аргумента функции. Кроме того, его нельзя использовать без инициализации.

```
1  #include <iostream>
2  using namespace std;
3  auto intSum(int a, int b){
4      return a + b;
5  }
6  auto intSum(int a, double b){
7      return a + b;
8  }
9  double intSum(int a, auto b){ // ошибка из-за auto b
10     return a + b;
11 }
12 int main(){
13     int a = intSum(2, 4);
14     cout << a << endl; // 6
15     int b = intSum(2, 4.1);
16     cout << b << endl; // 6
17     double c = intSum(2, 4.1);
18     cout << c << endl; // 6.1
19     auto d = intSum(2, 4.1);
20     cout << d << endl; // 6.1
21     auto f; // ошибка, нельзя без инициализации
22     auto t = "TUSUR";
23     auto *pa = &a;
24     cout << *pa << endl;
25     return 0;
26 }
```

Использовать `auto` надо очень осторожно, т. к. в двусмысленных ситуациях результат может оказаться неожиданным или даже непредсказуемым. При этом главная область применения ключевого слова `auto` – это, пожалуй, определение типа переменных циклов `for`, в частности цикла `for` для диапазона (*range-*

based for). Эта версия цикла `for` автоматически распознает массивы и ориентирована в основном на работу с элементами стандартной библиотеки C++ (прил. Г). Ее синтаксис: «`for(тип и имя переменной: выражение);`».

Представим, как работает такой цикл по сравнению со стандартным циклом `for`.

```
1  #include <iostream>
2  using namespace std;
3  int main(){
4     int x[4] = { 11, 21, 13, 14 };
5     for(int i = 0; i < 4; i++) // стандартный for
6        cout << x[i] << " ";
7     cout << endl;
8     for(auto i = 0; i < 4; i++) // стандартный for с auto
9        cout << x[i] << " ";
10    cout << endl;
11    for( int i : x ) // for для диапазона
12       cout << i << " ";
13    cout << endl;
14    for( auto i : x ) // for для диапазона с auto
15       cout << i << " ";
16    cout << endl;
17    return 0;
18 }
```

Здесь показано четыре примера вывода на печать элементов массива. Первые два – это использование стандартного цикла `for`, а два остальных – `for` для диапазона. Результаты работы всех четырех циклов будут совпадать. Так, использование цикла `for` для диапазона в сочетании с ключевым словом `auto` позволяет не заботиться ни о возможном выходе за границы массива, ни о задании типа данных для счетчика цикла.

2.3.3 Инициализация переменных

В языке C++ помимо копирующей инициализации переменных («`int val = 13;`») может быть использована прямая инициализация с использованием круглых скобок.

```
1  int val(13);
```

Как показала практика, эти инициализации работают по-разному с одними и теми же типами данных. Поэтому впоследствии для унификации работы со всеми типами данных добавлен еще один способ инициализации с использованием фигурных скобок.

```
1  int val{13};
2  float arr[3]{1.1, 3.2, 13.3}; // arr[2] = 13.3
```

При этом если оставить фигурные скобки пустыми, то переменная инициализируется нулем, что очень удобно.

```
1  int val{}; // val = 0
2  float arr[3]{1.1, 3.2}; // arr[2] = 0.000000000
```

При такой инициализации также происходит проверка типов данных переменной и значения, которое ей присваивается. Если они не совпадают, то при компиляции кода возникает соответствующая ошибка.

2.3.4 Явное приведение типов

Напомним, что в языке C возможно как явное, так и неявное приведение `(cast)` типов. При этом как в первом, так и во втором случае всю работу за пользователя делает компилятор. Однако для явного приведения пользователь должен указать тип данных, к которому это приведение должно привести. Для явного приведения используется запись вида `(тип) выражение` или `тип (выражение)`.

В языке C++ такой подход иногда называют «приведение в стиле языка C» (C-style).

```
1  float valF = 13.13; // valF = 13.13
2  int val = (int)valF; // val = 13
3  valF = (float)val; // valF = 13
4  int* ptr = (int*)10;
```

Приведение типов чревато программными ошибками, например, при разыменовании указателя из строки 4. При этом такой синтаксис приведения типов слабо заметен в программном коде. Поэтому в языке C++ для явного приведения используются операторы: `static_cast`, `reinterpret_cast`, `const_cast` и `dynamic_cast`.

Первый предназначен для явного приведения типов данных, включая указатели одного типа в указатели другого типа. Как отмечает создатель языка C++,

«отвратительное» имя `static_cast` выбрано им сознательно для выделения «опасного» оператора, который следует использовать только в исключительных случаях.

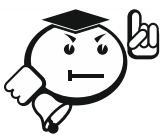
Второй оператор, имеющий, по всей видимости, еще более «отвратительное» имя `reinterpret_cast`, преобразует никак не связанные между собой типы данных, например `int` в `double*`.

Третий (`const_cast`) позволяет при приведении удалять или добавлять классификатор `const`.

Наконец, `dinamic_cast` используется только с указателями и ссылками на классы или с `void*`.

Синтаксис использования операторов приведения имеет общий вид: `оператор <тип> (выражение)`. Далее рассмотрено несколько примеров использования первых двух операторов.

```
1  float val = 13.3
2  cout << static_cast<int>(val) << endl; // 13
3  cout << "'T'="<< static_cast<int>('T')<<endl; // 'T'=84
4  float f { static_cast<float>(val / 2.) }
5  cout << "f=" << ff << endl; // f=6.65
6  int* j= reinterpret_cast<int*>(123);// число в указатель
7  cout << "j="<< j << endl;
```



При необходимости приведения типов в любом случае используйте явное приведение типов вместо неявного. При этом прежде чем использовать приведение типов подумайте, можно ли написать программный код как-то иначе – так, чтобы оно не потребовалось?

2.3.5 Условный оператор с инициализатором

Условные конструкции из операторов `if – else if – else` позволяют разветвлять программный код и часто используются на практике в обоих языках. Начиная с версии C++17 стало возможным инициализировать переменную внутри поля условия (в скобках) оператора `if`, которая может быть использована только в пределах действия оператора.

```
1  int d;
2  cin >> d;
```

```
3  if(int a = 5; d / a > 3)
4      cout << " 1 " << endl;
5  else if (d / a == 3)
6     cout << " 2 " << endl;
7  else
8     cout <<" 3 " << endl;
9  cout << a; // ошибка, a – не объявлена в этой области
10 if(float a = 3.1, b = d; a == b){
11    // требуемый программный код
12 }
```

2.3.6 Ссылки

Использование указателей (переменных, хранящих адреса других переменных) в языке С является одной из главных сложностей, поэтому в С++ реализован механизм *ссылка*, упрощающий работу программиста.

Ссылка *– это указатель, который неявно разыменовывается при доступе к значению, на которое он указывает.*

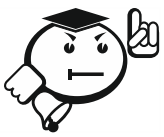
Продемонстрируем работу через указатели и ссылки на классическом примере функции, меняющей значение переменных местами (`swap`). (Напомним, что при работе с указателями для доступа к содержимому участка памяти, адрес которого хранится в указателе, используется оператор разыменования «*».)

```
1  #include <iostream>
2  using namespace std;
3  void swap_ptr(int* p1, int* p2){ // указатели
4      int tmp = *p1;
5      *p1 = *p2;
6      *p2 = tmp;
7  }
8  void swap_ref (int& r1, int& r2){ // ссылки
9      int tmp = r1;
10     r1 = r2;
11     r2 = tmp;
12 }
13 int main(){
14     int x, z;
```

```
15      x = z = 2;
16      int y, v;
17      y = v = 13;
18      cout << "before x = " << x << " y = " << y << endl;
19      swap_ptr(&x, &y); // указатели
20      cout << "after x = " << x << " y = " << y << endl;
21      cout << "before z = " << z << " v = " << v << endl;
22      swap_ref (z, v); // ссылки
23      cout << "after z = " << z << " v = " << v << endl;
24      return 0;
25  }
```

Важно помнить, что ссылки ограничены в функциональности по сравнению с указателями. Так, в отличие от указателей, ссылки должны быть инициализированы корректными объектами, которые не могут быть нулевыми и далее не могут быть изменены.

```
1   #include <iostream>
2   int main(){
3   // ссылки
4       int value = 13;
5       int& ref; // ошибка, неинициализированная ссылка
6       int& ref_ = value; // нет ошибки, ref = 13
7       &ref_ = 14; // ошибка, ссылка не может быть изменена
8       ref_ = 14; // нет ошибки, ref = 14
9   // указатели
10      int* pointer; // правильнее int* pointer = nullptr;
11      *pointer = 111; // нет ошибки
12      pointer = &value; // нет ошибки
13      return 0;
14  }
```



Обязательно инициализируйте используемые указатели, как минимум нулевым указателем (`nullptr`). Иначе они будут хранить случайные адреса памяти, по которым можно поместить некие данные, что, в свою очередь, может привести к нежелательным эффектам.

Если надо гарантировать, чтобы функция не меняла значение ее аргументов или не изменяла значения переменных в ходе выполнения программы, то используется ключевое слово `const`. Это ключевое слово также используется для объявления константных переменных и ссылок.

```
1  float func(const float val){
2      return val + = 2; // ошибка, из-за const у val
3  }
4  const int value = 13;
5  value += 1; // ошибка, изменение константы запрещено
6  int funcPow(const float& val){
7      return val * val; // нет ошибки
8  }
```

Если задача может быть решена с помощью указателей или ссылок, то лучше (проще) использовать последние. Указатели обязательно следует использовать, когда ссылки недостаточно эффективны, например при динамическом выделении памяти.

2.3.7 Псевдонимы типов данных

Напомним, что в языке С для объявления вместо сложных составных типов данных может быть использована директива `typedef`, позволяющая, например, вместо `unsigned char` использовать более короткую и информативную запись `byte_t` («`typedef unsigned char byte_t;`»). При этом «новый» тип данных `byte_t` становится синонимом, а не заменой `unsigned char`. Поэтому они оба могут равноценно использоваться в пользовательской программе.

В языке C++ в дополнение к `typedef` реализована возможность применения более наглядной записи с использованием ключевого слова (объявления) `using`. Такой синтаксис называется псевдонимом типа (type alias). Так, в C++ запись «`typedef unsigned char byte_t;`» является синонимом записи «`using byte_t = unsigned char;`».

```
1  #include <iostream>
2  using namespace std;
3  typedef unsigned char byte_t;
4  using byte_u = unsigned char;
5  int main(){
```

```
6       byte_t x = 'x';
7       byte_u y = 'y';
8       cout << x << endl << y << endl;
9       x=y;
10      cout << x << endl << y << endl;
11      return 0;
12  }
```

Запись «`using byte_t = unsigned char;`» считается более структурированной и понятной. Кроме того, использование `using` позволяет получить более понятные записи, чем `typedef`, в частности, при использовании указателей на функции. Указатели на функции используются как в языке C, так и в языке C++. Поэтому с помощью следующего примера напомним, как работает этот функционал.

```
1   #include <iostream>
2   using namespace std;
3   float funcMul(float z, float v){
4      return z * v;
5   }
6   float funcDiv(float z, float v){
7      return z / v;
8   }
9   int main(){
10     float (*func)(float , float); // указатель на
    функцию, имеющую 2 аргумента и возвращаемый тип - float
11     func = funcMul;
12     float result = func(2.1, 2.);
13     cout << "result mul=" << result << endl; // 4.2
14     func = funcDiv;
15     result = func(2.1, 2.);
16     cout << "result div=" << result << endl; // 1.05
17     return 0;
18  }
```

Здесь использован указатель на функцию, возвращающую значение и имеющую два аргумента. Ему последовательно присваиваются адреса двух функций, которые имеют такие же параметры. В результате использование указателя позволяет работать с однотипными функциями. При этом строку 10

«`float (*func)(float, float);`» можно убрать из функции `main()`, определив ее псевдоним типа или использовав директиву `typedef`.

```
using func = float (*)(float, float);
typedef float (*func)(float, float);
```

Обе эти строки по раздельности дают один и тот же результат, но считается, что первая имеет более понятный синтаксис. Поэтому в программах, написанных на C++, для повышения их читаемости рекомендуется использовать псевдонимы типа (`using`), а не `typedef`.

2.3.8 Шаблоны функций

Очевидно, что чем меньше повторяющегося кода в программе, тем легче он модифицируется и в нем проще найти возможные ошибки. Кроме того, например, при написании некой функции может быть еще точно не известно, какие типы данных будут иметь ее аргументы. Так как язык C++ (как и C) – это строго типизированный язык (все переменные должны иметь конкретные типы данных), то для решения поставленной задачи программист вынужден реализовывать однотипные функции, различающиеся лишь используемым типом данных их параметров и переменных (частично показано в подразд. 2.2). Продемонстрируем сказанное на примере рассмотренной выше функции `swap_reference`, меняющей значения двух переменных местами. Для наглядности используем три типа данных: `int`, `float` и `char`.

```
#include <iostream>
using namespace std;
void swap_ref (int& r1, int& r2){ // аргументы – ссылки
    int tmp = r1;
    r1 = r2; r2 = tmp;
}
void swap_ref (float& r1, float& r2){ // ссылки
    float tmp = r1;
    r1 = r2; r2 = tmp;
}
void swap_ref (char& r1, char& r2){ // ссылки
    char tmp = r1;
    r1 = r2; r2 = tmp;
}
int main(){
```

```
16      int ia=1, ib=2;
17      float fa=1.1, fb=2.1;
18      char ca='a',cb='b';
19      cout<< "before: ia=" << ia << " ib=" << ib << endl;
20      cout<< "before: fa=" << fa << " fb=" << fb << endl;
21      cout<< "before: ca=" << ca << " cb=" << cb << endl;
22      swap_ref(ia,ib);
23      swap_ref(fa,fb);
24      swap_ref(ca,cb);
25      cout<< "after: ia=" << ia << " ib=" << ib << endl;
26      cout<< "after: fa=" << fa << " fb=" << fb << endl;
27      cout<< "after: ca=" << ca << " cb=" << cb << endl;
28      return 0;
29  }
```

Даже в этом небольшом примере реализованные функции имеют много повторяющихся строк кода (строки 3–17). При этом было использовано только три типа данных. Очевидно, что их увеличение приведет к еще большему числу повторов в коде. Поэтому логичным выглядит решение использовать некий универсальный механизм, который сам будет вызывать ту или иную версию функции в зависимости от требуемого типа данных. Этот механизм в языке C++ называется *шаблоном*.

Шаблон функции *– это конструкция, создающая при компиляции конкретную функцию на основе заданных пользователем параметров или аргументов.*

Другими словами, шаблон позволяет создавать универсальную программную конструкцию, не зависящую от определенных типов данных. При этом требуемые типы определяются на этапе компиляции программы. Для объявления шаблона используется запись вида: `template <typename Type>`, где `template` – это ключевое слово, поясняющее компилятору, что он должен использовать нужный тип данных, соответствующий контексту, `typename` – еще одно ключевое слово (префикс), означающее «использовать для всех типов данных Type», предназначенное для «облегчения» работы компилятора. Соответственно `Type` – это общее имя типов данных (параметр шаблона), которое может принимать «значения», такие как `int`, `float`, `double`, `char` и пр. Это имя может

быть произвольным, но чаще используют `Type` или `T`. Дополнительно отметим, что символ «`;`» в конце строки с объявлением шаблона не ставится.

Далее приведен шаблон функции `swap_reference`, имеющей один шаблонный параметр (аргумент).

```
template <typename Type> // или template <class Type>
void swap_ref (Type& r1, Type& r2){ // ссылки
    Type tmp = r1;
    r1 = r2; r2 = tmp;
}
```

При использовании (вызове) шаблонной функции можно явно указывать требуемый тип данных (в угловых скобках) или не указывать его, полностью положившись на компилятор.

```
swap_ref<int>(ia,ib);// или swap_ref(ia,ib);
swap_ref<float>(fa,fb); // swap_ref(fa,fb);
swap_ref<char>(ca,cb); // swap_ref(ca,cb);
```

Отметим, что если ошибочно при вызове этой функции указать аргументы, имеющие разный тип данных (например, «`swap_ref(fa,ib);`»), то это приведет к ошибке компиляции.

В приведенном выше шаблоне функции использован один параметр (аргумент) – `Type`. Однако механизм шаблонов позволяет использовать несколько параметров (аргументов) с разными именами. Например, разработаем шаблонную функцию, которая не только меняет местами значения переменных, но и возводит их в требуемую степень.

```
template <typename Type1, typename Type2 >
void swap_ref_pow(Type1& r1, Type1& r2, Type2 n){
    Type1 tmp = pow(r1,n);
    r1 = pow(r2,n);
    r2 = tmp;
}
```

Здесь для возведения в степень использована стандартная функция `pow()` из заголовочного файла `cmath`. Вызов этой шаблонной функции в целом аналогичен приведенной выше, за тем лишь различием, что при ее использовании надо указывать уже два типа данных.

```
1  swap_ref_pow<int, float>(ia,ib,3.1);
   // или swap_ref_pow(ia,ib,3.1);
2  swap_ref_pow<float, int>(fa,fb,5.1);
   // или swap_ref_pow(fa,fb,5.1);
3  swap_ref_pow<char, int>(ca,cb,2);
   // или swap_ref_pow(ca,cb,2);
```

2.3.9 Обработка исключений

Для проверки правильности работы функции или программы, написанных на языке С с использованием процедурного подхода, предполагается анализ значений, возвращаемых функцией (программой) с помощью оператора `return`. По полученному значению судят о правильности работы функции (программы), т. е. о том, отработала ли она корректно или нет (например, если получено значение «`0`», то корректно, а если «`1`», то нет). Это хоть и удобно, но не информативно, т. к. непонятно, в какой части функции или программы возникла ошибка. При этом многие вещи в функции или программе могут пойти не так, поэтому надо увеличивать число кодов возврата и постоянно их проверять, чтобы понять, где произошла ошибка. Это приводит к тому, что растет сложность самой программы и ухудшается читаемость ее кода из-за необходимости реализации большого числа проверок. В связи с этим в C++ разработан специальный механизм обработки исключений, позволяющий отделить обработку ошибок или других исключительных обстоятельств от общего потока выполнения кода. Это предоставляет больше свободы в конкретных ситуациях и повышает эффективность проверки работы программного кода и его читаемость.

Оператор `throw` используется для обнаружения исключения (ошибки). После оператора указывается значение любого типа данных, которое можно использовать для сигнализации. Как правило, этим значением является описание проблемы или код ошибки. Однако исключение затем надо обработать. Для этого применяется конструкция из двух блоков. Код, в котором могут возникнуть ошибки, помещается в блок `try{}`, за которым следует блок `catch(){}`, в который помещается код для обработки исключительной ситуации. В результате блок `try{}` обнаруживает любые исключения и направляет их в соответствующий блок `catch(){}` для обработки. При этом блок `try{}` должен иметь по крайней мере один блок `catch(){}`.

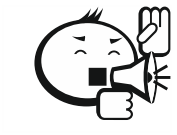
Продемонстрируем сказанное на примере проверки правильности работы пользовательской функции деления двух чисел.

```
1  #include <iostream>
2  using namespace std;
3  float myDiv(float a, float b){
4      string errStr = "Division by zero in myDiv()";
5      if(a == 0)
6          throw -1; // или throw "Numerator is = 0"; – (*)
7      if (b == 0)
8          throw errStr;
9      return a / b;
10 }
11 int main(){
12     float firstNum, secondNum;
13     cout << "Enter 2 numbers"<<endl;
14     cin >> firstNum; // ввод числа 1
15     cin >> secondNum; // ввод числа 2
16     try{
17         float resultDiv = myDiv(firstNum, secondNum);
18         cout << "Result = "<< resultDiv << endl;
19     }
20     catch(string msg){
21         cout << msg << endl;
22     }
23     catch(int err){//или catch(const char* err){ для (*)
24         cout << err << endl;
25     }
26     return 0;
27 }
```

Здесь для наглядности, хотя и избыточно, использовано два блока `catch(){}`, обрабатывающих разные исключения. Так, первый блок выводит на печать код ошибки, а второй – сообщение об ошибке. Иначе, если нет ошибок (исключений), то на печать выводится результат работы функции. Обратим внимание на строки `6` и `23`, с помощью которых показана возможность вывода на печать двух сообщений об ошибках вместо одного сообщения и одного кода ошибки.

Как только исключение было «поймано» блоком `try{}` и направлено в блок `catch(){}`, оно считается обработанным, после чего выполнение программы возобновляется. При этом допустимо, что блок `catch(){}` может быть пустым. Однако очевидно, что это не лучшее решение проблемы, т. к. обработки

исключения по сути нет. Кроме того, следует учитывать, что блок `catch(){}` не находится внутри блока `try{}`. Тогда, если в нем сгенерируется новое исключение, то обрабатывать его будет следующий блок `try{}`, который в этом случае надо предусмотреть.



Что произойдет, если в функции генерируется исключение, но оно никак не обрабатывается (отсутствуют блоки `try{}` и `catch(){}`)?

Ответ на поставленный вопрос зависит от используемых средств компиляции программного кода. Так, при использовании интегрированной среды разработки, если функция `main()` завершает свое выполнение с необработанным исключением, операционная система чаще всего выводит сообщение об ошибке во всплывающем окне. Однако также может произойти программный сбой. Использование онлайн-компиляторов обычно приводит к прекращению выполнения пользовательской программы с выдачей сообщения об ошибке.

В рассмотренном примере использовано два обработчика исключений, различающихся типом данных их аргументов (`string` и `int` в строках `20` и `23` соответственно). Если необходимо добавить еще несколько вариантов обработки исключений, следует для каждого из этих вариантов дополнительно реализовывать свои блоки `catch(){}`. Тогда, в случае возникновения одного из исключений, поиск некорректной работы программы может значительно упроститься.

При написании программного кода не всегда удается учесть все возможные варианты возникновения исключений. Поэтому дополнительно реализован механизм, позволяющий обрабатывать каждый тип исключений, которых нет в предыдущих блоках `catch(){}`. Для этого используется блок `catch(...){}` с оператором «`...`» (многоточие), обрабатывающий каждый тип исключений, которых нет в предыдущих блоках `catch(){}`. Следует помнить, что этот блок всегда должен быть задан последним в цепочке блоков `catch(){}`. Кроме того, его использование позволяет сделать одно из исключений «основным», а остальные «второстепенными» или вообще сделать один обработчик для всех исключений. Однако так делать не рекомендуется.

```
1  try{
2      float resultDiv = myDiv(firstNum, secondNum);
3      cout << "Result = " << resultDiv << endl;
```

```
}
catch(string msg){
    cout << msg << endl;
}
catch(int err){
    cout << err << endl;
}
catch(...){ // catch-all
    cout << "the catch-all worked" << endl;
}
```

2.3.10 Встраиваемые функции

Помимо обычных функций в языке C++ используются так называемые *встраиваемые функции*. Так, функция, объявленная как `inline`, называется встроенной. При ее вызове происходит подстановка ее содержимого (тела функции) в место вызова. Это призвано ускорить процесс выполнения программы за счет устранения нагрузки на вызов этой функции. Важно отметить, что решение – выполнять подстановку встраиваемой функции или нет – в конечном счете все же зависит от компилятора. Например, если функция слишком большая, то компилятор проигнорирует подстановку. Поэтому объявлять встраиваемыми следует только небольшие функции.

При работе с обычной функцией после компиляции программный код функции помещается в ячейку памяти. Тогда при ее вызове происходит переход по этому адресу и отработка функции. Использование `inline` позволяет отказаться от этого (переход в функцию, вычисление и возврат из функции) и тем самым ускорить процесс работы программы. Встраивание функции происходит по всей программе, где есть ее вызовы.

Рассмотрим использование пользовательской функции `max()`, которая сравнивает два значения, переданные по ссылкам, и возвращает по ссылке большее из них.

```
#include <iostream>
using namespace std;
inline const double& max(const double& a, const
double& b){
   return (a > b)? a : b;
}
int main(){
```

```
7       cout<< max(10.001,10.1) << endl;
8       cout<< max(-1.1,1.1) << endl;
9       cout<< max(20.,-20.) << endl;
10      return 0;
11  }
```

Здесь параметры функции и возвращаемая ссылка объявлены константными, а для сравнения значений внутри функции использован тернарный оператор. Эта программа эквивалентна следующей, в которой содержимое функции встроено напрямую в программный код.

```
1   #include <iostream>
2   using namespace std;
3   int main(){
4       cout<< ((10.001 > 10.1)? 10.001 : 10.1) << endl;
5       cout<< ((-1.1 > 1.1)? -1.1 : 1.1) << endl;
6       cout<< ((20. > -20.)? 20. : -20.) << endl;
7       return 0;
8   }
```

Используйте объявление функций встраиваемыми (`inline`) только при конечной настройке программного кода, т. е. когда программа полностью работает так, как вам надо. (Помните: «Преждевременная оптимизация – корень всех зол».)

2.4 Разделение программы на файлы

2.4.1 Файлы заголовков и реализации

Материал данного параграфа важен для освоения дисциплины и приведен для напоминания. Знакомый с материалом читатель может изучить его бегло или пропустить.

Если программа содержит много строк кода, то возможно два варианта повышения его читаемости. Первый – это объявить до функции `main()` все функции, классы и пр., а их реализацию привести ниже этой функции. Однако более оптимальным является разнесение отдельных частей кода по разным файлам. Так, часть функций может храниться в одном файле исходного кода, а вторая часть – во втором файле.

Например, уже не раз была использована запись вида «`cout << `». При этом нигде явно не было определено, что такое `cout`. Поэтому может возникнуть вопрос: откуда компилятор знает, что это такое? Дело в том, что `cout` объявлен в заголовочном файле `iostream`. Запись «`#include <iostream>`» означает, что все содержимое файла `<iostream>` копируется в файл программы и, соответственно, становится доступным для использования.

В заголовочных файлах (файлах заголовков), как правило, записываются только объявления функций, классов и пр., а их определения – в исполняемых файлах (файлах реализации). При этом если подключается заголовочный файл стандартной библиотеки C++, то расширение файла не указывается, а его имя помещается в угловые скобки. Если же подключается пользовательский заголовочный файл, то его имя с указанием расширения помещается в кавычки (например, `"my_pow.h"`).

Продемонстрируем сказанное на примере небольшого проекта (`Lecture`), состоящего из одного заголовочного и двух исполняемых файлов (рис. 2.2). Инструкции по созданию проектов в MVS и Qt Creator приведены в приложениях Д и Е соответственно.

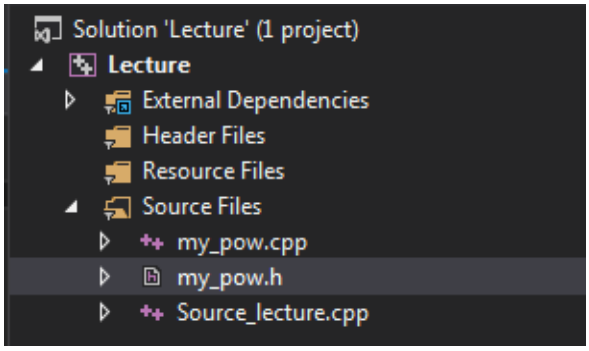


Рис. 2.2 – Структура проекта в MVS с разделением
на файлы заголовков и реализации

В примере на рисунке 2.2 объявление функции `my_pow()` помещено в файл `my_pow.h`, а ее реализация – `my_pow.cpp`. Использоваться же она будет из функции `main()`, помещенной в файл `Source_lecture.cpp`.

```
                        // файл my_pow.h
1   double my_pow(double, int); // объявление
                      // файл my_pow.cpp
1   #include "my_pow.h"
2   double my_pow(double val, int num){ // реализация
3       double result 1;
```

```
    for(int i = 1; i <= num; i++)
        result *= val;
    return val;
}
```

```
// файл Source_lecture.cpp
#include <iostream>
#include "my_pow.h"
using namespace std;
int main(){
    double x = 5;
    double result = my_pow(x, 4); // использование
    cout<< "result = " << result << endl;
    return 0;
}
```

Созданные вами заголовочные файлы (`*.h`) должны содержать только объявления функций, классов и пр. Их определения должны находиться в файлах реализации (`*.cpp`), которые также должны содержать директиву включения (`include`) соответствующих им созданных заголовочных файлов. *Исключением являются шаблоны функций, объявление и реализацию которых надо помещать только в заголовочные файлы.* (Отметим, что есть альтернативные решения, но здесь они не рассматриваются.)

2.4.2 Защита от повторного включения файлов заголовков

Как было показано в п. 2.4.1, разделение на заголовочные файлы и файлы реализации позволяет значительно повысить читаемость кода, однако могут возникнуть ситуации, когда код не компилируется, даже для небольших проектов. Это связано с тем, что нарушается принцип одноразового определения переменных, функций и прочих элементов программного кода. Очевидно, что этот принцип позволяет избежать неоднозначности при использовании этих элементов. Так, например, следующий код не может быть скомпилирован из-за повторного определения одной и той же функции.

```
#include <iostream>
int calc(int a){
```

```
3       return a + 1;
4   }
5   int calc(int a){
6       return a + 1;
7   }
8   int main(){
9       int x = 5;
10      x = calc(x);
11      return 0;
12  }
```

Аналогичная ситуация может возникнуть, когда один или несколько заголовочных файлов включают другие заголовочные файлы. Тогда при компиляции возникнет такая же ситуация, как при повторном определении функции. Поэтому при разработке программ требуется избегать повторного включения заголовочных файлов (`*.h`) в один и тот же файл реализации (`*.cpp`).

Для этого разработан специальный защитный механизм включения (*include guard*), который повсеместно использован в стандартной библиотеке C++ (прил. Г). Для «включения» этого механизма используется конструкция, основанная на сочетании директив `#ifndef`, `#define` и `#endif`. Аргументом первой директивы является уникальный идентификатор – часто это имя заголовочного файла, записанное в обобщенном виде. Например, для файла `Circuit.h` это имя – `CIRCUIT_H`.

```
1   #ifndef CIRCUIT_H // или #if !defined CIRCUIT_H
2       #define CIRCUIT_H
3       // Блок кода, который добавляется только один раз
4   #endif // CIRCUIT_H
```

Директива `#ifndef` проверяет, определен ли идентификатор с именем `CIRCUIT_H`. Если он уже определен, то управление передается директиве `#endif`. Иначе с помощью директивы `#define` производится определение идентификатора, а также кода, находящегося после этой директивы.

С учетом сказанного одной из возможных корректировок кода, приведенного выше, будет использование охранного механизма. При этом в качестве прототипа имени идентификатора может быть использовано имя функции (`CALC_H`).

```
1   #include <iostream>
2   #ifndef CALC_H
3   #define CALC_H
```

```
4  int calc(int a){
5      return a + 1;
6  }
7  #endif // CALC_H
8  #ifndef CALC_H
9  #define CALC_H
10 int calc(int a){
11     return a + 1;
12 }
13 #endif // CALC_H
14 int main(){
15     int x = 5;
16     x = calc(x);
17     return 0;
18 }
```

Эстетическое удовольствие от полученного программного кода, конечно, получить сложно, но эта версия программы скомпилируется без ошибок. Кроме того, такая реализация позволяет наглядно продемонстрировать работу охранного механизма.

Перепишем эту программу в рекомендуемом виде. Для этого создадим два заголовочных файла: `calc.h` и `test.h`. Во втором выполним подключение первого, а также реализуем в них охранные механизмы. Кроме того, создадим файл реализации и выполним в нем подключение этих двух заголовочных файлов. Тогда программа успешно скомпилируется и повторного подключения файла `calc.h` не произойдет.

```
                        // файл calc.h
1  #ifndef CALC_H
2  #define CALC_H
3  int calc(int a){
4      return a+1;
5  }
6  #endif // CALC_H
                        // файл test.h
1  #ifndef TEST_H
2  #define TEST_H
3  #include "calc.h"
4  #endif // TEST_H
```

```
                    // файл calc.cpp
1  #include <iostream>
2  #include "calc.h"
3  #include "test.h"
4  using namespace std;
5  int main(){
6      int x = 5;
7      x = calc(x);
8      cout << x << endl;
9      return 0;
10 }
```

Как уже говорилось выше, идентификатор должен иметь уникальное имя, но это может создать ситуацию, когда в программных проектах, особенно больших, подключается несколько заголовочных файлов с одинаковыми именами. В результате из-за совпадения имен файлов будут совпадать и их идентификаторы, что, скорее всего, вызовет ошибку компиляции. Поэтому при задании идентификаторов лучше использовать более сложные имена, например, включать в них информацию о дате создания файла, названии проекта, имени автора и пр. (`NUMERICAL_ELECTRIC_FIELD_CALC_2022_H`).

Большинство компиляторов поддерживает альтернативную и более короткую форму объявления защитного механизма от повторного включения заголовочных файлов, но не все, т. к. этот механизм не стандартизован (поэтому для совместимости ваших программных кодов рекомендуется придерживаться «классического» механизма защиты *include guard*). Для этого используется директива `#pragma once`. При этом она может быть применима только в заголовочных файлах. После ее задания весь последующий код заголовочного файла будет использован только один раз. Тогда файлы `calc.h` и `test.h` из последней программы примут более компактный вид.

```
                    // файл calc.h
1  #pragma once
2  int calc(int a){
3      return a+1;
4  }
                    // файл test.h
1  #pragma once
2  #include "calc.h"
```

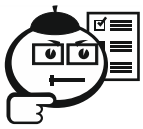
Рекомендации по созданию и использованию заголовочных файлов (`*.h`) и файлов реализации (`*.cpp`):

1. Помещайте определения в заголовочные файлы (`*.h`), а реализацию – в (`*.cpp`).
2. Следует избегать переопределения переменных в заголовочных файлах (`*.h`), если они не являются константами (например, «`const float pi = 3.14;`»).
3. Каждый заголовочный файл (`*.h`) должен выполнять свою определенную работу и быть как можно более независимым. Например, можно поместить все объявления, связанные с файлом `Calc.cpp`, в файл `Calc.h`, а все объявления, связанные с `Data.cpp`, – в файл `Data.h`. Тогда, например, если нужно работать только с `Calc.cpp`, будет достаточным подключить только `Calc.h`.
4. Используйте имена файлов реализации (`*.cpp`) в качестве имен заголовочных файлов (`*.h`). Например, `factorial.h` работает в связке с `factorial.cpp`.
5. Следует избегать подключения одних заголовочных файлов (`*.h`) из других заголовочных файлов. Иначе надо использовать защитные механизмы от повторного включения `#ifndef`, `#define`, `#endif` или `#pragma once`.
6. Нельзя подключать в проекты файлы реализации (`#include *.cpp`).
7. Следует размещать объявления и реализации шаблонов функций только в заголовочных файлах (`*.h`).

Контрольные вопросы по разделу 2

1. Каково назначение конструкций `cin >>` и `cout <<`?
2. Для чего нужны операторы `new` и `delete`?
3. Можно ли использовать ключевые слова языка C++ в качестве имен переменных и функций?
4. Для чего используется ключевое слово `auto`?
5. В чем преимущество цикла `for` для диапазона по сравнению со стандартным циклом `for`?
6. Сколько способов инициализации переменных в языке C++?

7. Каково назначение оператора `static_cast`?
8. Что позволяет делать оператор `if` с инициализатором?
9. Чем ссылка отличается от указателя?
10. Что означает ключевое слово `nullptr`?
11. Для чего используется ключевое слово `const`?
12. Что позволяет объявление псевдонима типа данных?
13. Каково назначение шаблонов функций?
14. Для чего используется оператор `throw`?
15. Может ли блок `try{}` быть вложенным в блок `catch(){}`?
16. Для чего функции объявляются встраиваемыми `(inline)`?
17. Зачем следует помещать объявление и реализацию функций в разные файлы?
18. Как работает защитный механизм, основанный на совместном использовании директив `#ifndef`, `#define`, `#endif`?
19. Для чего используется `pragma once`?



Задания для самостоятельного выполнения

1. Разработать программу для вычисления площади треугольника. Входными данными, которые вводит пользователь, являются координаты трех точек в пространстве. Выходными данными является площадь треугольника, построенного на точках, координаты которого введены пользователем. Программную реализацию выполнить с использованием шаблонной функции. Реализацию шаблонной функции выполнить в заголовочном файле.
2. В разработанную по предыдущему пункту программу включить механизм обработки исключений. Исключение должно возникать, например, при введении одинаковых координат всех трех точек (площадь треугольника равна нулю).

3 Объектно-ориентированное программирование: инкапсуляция

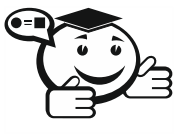
> – Мы, управление дома, пришли к вам после общего собрания жильцов нашего дома, на котором стоял вопрос об уплотнении квартир дома.
>
> – Кто на ком стоял? Потрудитесь излагать ваши мысли яснее.
>
> *М. А. Булгаков. Собачье сердце*

3.1 Вводные замечания

Сможете ли вы, дорогой читатель, различить стул и телевизор? Думаем, что ответ будет положительным. По сути, весь окружающий нас мир состоит из объектов (экземпляров) того или иного назначения. По их характерным (существенным) свойствам мы понимаем, что, например, перед нами стул, а не телевизор, т. е. мы разделяем объекты одного вида от объектов другого вида.

Стул, являющийся мебельным изделием и предназначенный для сидения одного человека, состоит из спинки, сиденья и, как правило, опирается на четыре ножки. При этом, в зависимости от производителя, в конструкцию могут быть включены подлокотники, удалена одна из ножек и пр. Однако, отметим, что основное назначение стула будет неизменным и его пользователь знает, как его использовать по прямому назначению.

Более сложный объект другого вида – это современный телевизор, управляемый с помощью кнопок, расположенных на поверхности его корпуса и (или) пульта. Пользователь телевизора знает, что для начала просмотра любимой передачи ему надо нажать определенную комбинацию кнопок. При этом, в подавляющем большинстве случаев, пользователь имеет лишь поверхностные знания об устройстве телевизора. Тем не менее, это не мешает ему успешно пользоваться этой разработкой радиоэлектронной промышленности. Получается, что телевизор – это некий «черный ящик», реализация которого скрыта от пользователя, а кнопки управления – это интерфейс, с помощью которого пользователь управляет работой этого «ящика». Надеемся, что этот пример пояснил главные идеи абстракции.



Абстракция *– процесс выявления главных характеристик объекта, отличающих его от других объектов, для их последующего раздельного анализа и инкапсуляции* (*сокрытие реализации*).

Применительно к программам можно заключить, что абстракция данных предназначена для отделения незначительных деталей их реализации от значительных деталей их использования. Другими словами, абстракция разделяет программный код на код, реализующий интерфейс, т. е. то, как можно использовать функциональность, и код, реализующий ее. При этом это деление может быть сделано не только на уровне всей программы (телевизор), но и на уровне ее составных частей (дисплей, пульт) и элементов (печатная плата пульта). Так, вместо того чтобы сосредоточиться на написании функций, теперь сконцентрируем внимание сначала на определении объектов, которые имеют четкий набор поведений или характеристик, а уже потом на их программном описании.

Эта парадигма называется объектно-ориентированной. Ее использование выражается в создании новых пользовательских (абстрактных) типов данных – *классов*, которые могут обобщать разнородные данные и методы (функции) работы с ними. Это обеспечивает более интуитивный способ работы с данными, позволяя программисту определить, как он будет взаимодействовать с объектами и как эти объекты будут взаимодействовать с другими объектами. Другими словами, класс – это тип, описывающий характеристики и поведение объекта.

Считается, что ООП не заменяет традиционные методы программирования, а дополняет их, позволяя снизить сложность разработки и последующей модификации программ. Если читателю из прочитанного до конца еще не все стало понятно или он окончательно «запутался», то не беда, т. к. далее с использованием демонстрационных примеров суть сказанного должна проясниться. Затем рекомендуем перечитать эти вводные строки, и все должно встать на свои места.

3.2 Структуры

В процедурном (не объектно-ориентированном) программировании тоже есть возможность использования пользовательских типов данных, таких как перечисления (`enum`) и структуры (`struct`). Структуры наиболее важны для дальнейшего изложения, поэтому остановимся на них подробнее. Так, например, структура для хранения комплексного числа может иметь следующий вид.

```
struct Complex{
    double real_d; // реальная часть
    double image_d; // мнимая часть
};
```

Переменные, «находящиеся внутри» структуры, принято назвать полями (свойствами). Для наглядности рассмотрим еще одну структуру – `Student`, содержащую минимальную информацию, а именно: поля фамилии, имени, отчества, возраста и номера учебной группы.

```
struct Student{
    string surname_s; // фамилия
    string name_s; // имя
    string patronymic_s; // отчество
    unsigned int age_ui; // возраст
    int group_i; // номер учебной группы
};
```

Как видно, поля структуры могут иметь разные типы данных, включая другие структуры. Например, структуру `Student` можно организовать на основе структуры `Full name` (фамилия, имя, отчество).

```
struct FullName{
    string surname_s; // фамилия
    string name_s; // имя
    string patronymic_s; // отчество
};
struct Student{
    FullName name_str; // ФИО
    unsigned int age_ui; // возраст
    int group_i; // номер учебной группы
};
```

При этом использование полей структуры, в целом, аналогично использованию переменных стандартных типов данных. Так, создадим и инициализируем три объекта (экземпляра) типа `Complex`, а затем будем оперировать ими.

```
Complex a_c; a_C.real_d = 5.5; a_C.image_d = -10.1;
Complex b_c; b_C.real_d = 4.5; b_C.image_d = 0.2;
Complex c_c; c_C.real_d = 0.0; c_C.image_d = 0.0;
a_C.real_d  += b_C.real_d;
a_C.image_d  = b_C.real_d;
```

Видно, что при работе с полями структуры использован оператор «.» (точка). Этот оператор используется для «прямого» обращения к полям структуры (или класса). Если обращение к полям осуществляется через указатель на структуру (или класс), то вместо оператора «.» (точка) следует использовать оператор «->» (стрелка).

3.3 Классы

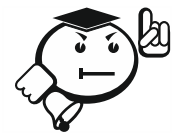
На первый взгляд классы мало чем отличаются от структур, поскольку у классов также есть поля. Так, классы `Complex` и `Student` могут иметь следующий вид:

```
1  class Complex{
2      double real_d; // реальная часть
3      double image_d; // мнимая часть
4  };
5  class Student{
6      string surname_s; // фамилия
7      string name_s; // имя
8      string patronymic_s; // отчество
9      unsigned int age_ui; // возраст
10     int group_i; // номер учебной группы
11 };
```

Видно, что принципиально ничего не изменилось. Так, вместо одного ключевого слова `struct` использовано другое – `class`. (Напомним, что ключевые слова – зарезервированные идентификаторы, которые нельзя использовать в качестве пользовательских идентификаторов: переменных, функций и пр.) Однако после этой замены и попытки запуска программы читателя ждет «неприятный сюрприз» в виде неуспешной компиляции кода. Причины этого будут пояснены в п. 3.3.6. Отметим лишь, что это различие обусловлено той самой абстракцией данных, о которой говорилось в начале раздела.

3.3.1 Конструкторы

Структуры и классы могут содержать внутренние функции (действия, выполняемые их объектами), которые принято называть методами. Они имеют доступ к любым данным объекта класса, членом которого являются.



В целом *структуры* и *классы* – это идентичные конструкции, с помощью которых пользователь может определять собственные типы данных и способы работы с ними.

Первым рассмотрим метод, который называется *конструктор*.

Конструктор *– это специальный метод класса, предназначенный для инициализации его полей при создании объектов класса. Его имя совпадает с именем самого класса.*

Использование конструктора позволяет при создании объекта (экземпляра) структуры/класса автоматически выполнять требуемую инициализацию полей. При этом тип данных возвращаемого значения не указывается, а имя конструктора совпадает с именем самой структуры/класса.

```
1  Complex(double x, double y){
2      real_d = x; image_d = y;
3  }
```

Очень важно, что конструкторов может быть несколько. Так, для рассматриваемого примера создадим еще два конструктора с отличающимися списками аргументов, последний из которых, не имеющий аргументов, называется конструктором по умолчанию. (Если его нет в классе, то компилятор автоматически создаст его, поэтому так делать не рекомендуется.)

```
1  Complex(double x){
2      real_d = x; image_d = 0;
3  }
4  Complex(){ // конструктор по умолчанию
5      real_d = 0; image_d = 0;
6  }
```

Как было показано в подразд. 2.2, в функциях и методах можно использовать значения их аргументов по умолчанию. Тогда вместо конструктора по умолчанию можно использовать конструктор, имитирующий его работу, с заданными значениями аргументов.

```
Complex(double x = 0, double y = 0){
    real_d = x; image_d = y;
}
```

Структуры/классы могут содержать и другие методы. Например, пусть имеется некий метод, возводящий реальную и мнимую части комплексного числа в квадрат.

```
Complex povComplex(){
    real_d = real_d * real_d;
    image_d = image_d * image_d;
}
```

Отметим, что все методы, включая конструкторы, должны размещаться (как минимум объявление) в теле самой структуры или класса.

```
struct Complex{
    double real_d;
    double image_d;
    Complex(double x, double y){ // конструктор 1
        real_d = x; image_d = y;
    }
    Complex(double x){ // конструктор 2
        real_d = x; image_d = 0;
    }
    Complex(){ // конструктор по умолчанию
        real_d = 0; image_d = 0;
    }
    void povComplex(){
        real_d = real_d * real_d;
        image_d = image_d * image_d;
    }
};
```

Конструкторы по умолчанию, как правило, не имеют параметров (`Complex(){}`). Однако они могут инициализировать поля, используя значения по умолчанию, как показано выше. При этом можно явно запретить компилятору использовать конструктор по умолчанию. Для этого при определении надо указать его как удаленный – с помощью оператора «=» и ключевого слова `delete` (важно, что при этом фигурные скобки не ставятся).

```
Complex() = delete;
```

Для задания конструкторов можно использовать короткую запись с использованием списка инициализаторов полей, что делает код легко читаемым.

```
1  Complex(double x, double y): real_d(x), image_d(y) { }
   // конструктор 1
2  Complex(double x): real_d(x), image_d(0) { }
   // конструктор 2
3  Complex(): real_d(0), image_d(0) { } // конструктор
   по умолчанию
```

Для демонстрации работы конструкторов создадим еще несколько объектов и используем для них метод `povComplex()`.

```
1  Complex d; // d = (0, 0)
2  Complex w(-1, 10); // w = (-1, 10)
3  Complex h(2); //  h = (2, 0)
4  h.povComplex(); // (4, 0)
```

Выводы

Переменные класса называются полями, а его функции – методами. Можно определить несколько методов-конструкторов для одного класса, которые не имеют возвращаемых значений, а их имена совпадают с именем самого класса. При объявлении объектов класса (конкретных переменных этого типа данных, т. е. класса) компилятор сам выбирает подходящие конструкторы на основании числа заданных аргументов. Использование конструкторов является эффективным способом инициализации полей класса.

3.3.2 Определение и реализация

Методы должны быть обязательно определены внутри структуры/класса, а их реализация может быть выполнена отдельно. Тогда, чтобы компилятору было понятно, к какой именно структуре/классу относятся эти методы, следует добавлять к именам методов имена их структур/классов, а также оператор разрешения области видимости «`::`» (используется для идентификации и устранения неоднозначности идентификаторов). Так, приведенная выше реализация структуры `Complex` может быть представлена так, как показано ниже. При этом анало-

гично функциям объявление методов структуры/класса принято помещать в заголовочный файл с расширением «`*.h`» (строки 1–8), а их реализацию – в исполняемый файл с расширением «`*.cpp`» (строки 9–21).

```
1  struct Complex{
2      double real_d;
3      double image_d;
4      Complex(double, double); // объявление констр. к1
5      Complex(double); // объявление констр. к2
6      Complex(); // объявление конструктора по умолчанию
7      void povComplex(); // объявление метода
8  };
9  Complex::Complex(double x, double y){ // реализация к1
10     real_d = x; image_d = y;
11 }
12 Complex::Complex(double x){ // реализация к2
13     real_d = x; image_d = 0;
14 }
15 Complex::Complex(){ // реализация констр. по умолчанию
16     real_d = 0; image_d = 0;
17 }
18 void Complex::povComplex(){ // реализация метода
19     real_d = real_d * real_d;
20     image_d = image_d * image_d;
21 }
```

Как видно, у каждой структуры/класса есть свое пространство имен (подразд. 2.2). В данном случае это `Complex`.

3.3.3 Деструкторы

Очевидно, что если есть метод, который, по сути, создает объект, то должен быть метод-антипод, который разрушает объект (освобождает память). Этот метод называется деструктор.

Деструктор *– это метод класса, автоматически вызываемый при выходе его объекта из области видимости* (*например, при завершении программы или его удалении*).

Деструктор аналогично конструктору имеет то же имя, что и структура/класс, но ему предшествует тильда «~»: `~Complex()`. В отличие от конструктора, деструктор не может иметь аргументов и возвращать никаких значений, в том числе и `void`. При этом деструктор в классе может быть только один. Разрабатывать деструктор для простых структур/классов не обязательно. В этом случае компилятор предоставит деструктор по умолчанию.

Когда структура/класс оперирует системными ресурсами или указателями, следует обязательно определять деструктор. Тогда при разрушении экземпляра структуры/класса задействованная машинная память должна быть освобождена. Для этого и предназначен деструктор. При этом вызывать этот метод в программе необязательно, т. к. компилятор сделает это самостоятельно.

3.3.4 Арифметические операции

При рассмотрении структуры `Complex` у читателя может возникнуть вопрос: как складывать эти переменные? Для этого можно сделать, например, следующим образом:

```
1  c.real_d = a.real_d + b.real_d;
2  c.image_d = a.image_d + b.image_d;
```

Не совсем удобно, не так ли? Поэтому модернизируем код, добавив «внешнюю» функцию сложения двух комплексных чисел (не через метод структуры).

```
1  Complex addComplex (Complex c1, Complex c2){
2      Complex result;
3      result.real_d = c1.real_d + c2.real_d;
4      result.image_d = c1.image_d + c2.image_d;
5      return result;
6  }
7  c = addComplex(a, b); // реализация записи c = a + b
```

Аналогично могут быть созданы функции разницы (`subComplex`), умножения (`mulComplex`), деления (`divComplex`) и т. д. Отметим, что приведенный пример создания обычных («внешних») функций не является единственным способом для оперирования с полями рассматриваемой структуры. Так, рассмотрим использование метода структуры («внутренней» функции) и указателя `this`.

```
1  void addComplex(Complex c1){
2      this->real_d += c1.real_d;
3      this->image_d += c1.image_d;
```

```
4  }
5  c.addComplex(b); // вместо c = c + b
```

Здесь в качестве примера рассмотрено увеличение значения переменной `c` на величину переменной `b` с использованием указателя `this`, который указывает на объект, для которого вызван метод `addComplex()` (т. к. метод можно вызвать только для объекта). В этом примере это переменная `c` (строка 5). Также видно, что для доступа к полям объекта через указатель используется оператор «`->`» (стрелка), а не оператор «`.`» (точка).

3.3.5 Спецификаторы доступа

Вернемся к пояснению различий между структурами и классами, обозначенными в подразд. 3.3. Для этого используем объявление структуры `Complex`.

```
1  struct Complex{
2      double real_d;
3      double image_d;
4      Complex(double, double);
5      Complex(double);
6      Complex();
7      void povComplex();
8  };
```

При создании объекта (экземпляра) этой структуры, например «`Complex tC;`», содержимое его полей может быть модифицировано в методе `povComplex()` или вообще напрямую из любого места программы, например «`tC.real_d = -100;`», и даже извне (из внешнего кода).

Если обратиться к примеру с телевизором, то получается, что любой пользователь может беспрепятственно менять любые его настройки, перепаивать радиокомпоненты, удалять «ненужные» элементы и пр., а потом удивляться, что телевизор не работает. Чтобы такого не происходило, производитель телевизора реализует возможности изменения некоторых его настроек через специальные интерфейсы (вспомните различные меню вашего телевизора) и тем самым препятствует бесконтрольному доступу к внутреннему устройству телевизора (его реализации).

Если приведенный пример все же не убедил читателя в том, что надо разделять интерфейс структуры и ее реализацию, а также скрывать последнюю, то можно вспомнить про банковский счет, который тоже можно рассматривать как

структуру, хранящую в банковской компьютерной программе сведения о счетах пользователя. Для работы с этим счетом у его пользователя есть интерфейсы (банковская карта, мобильное приложение, онлайн-банк и пр.). При этом логично, что любой другой пользователь не имеет доступа к этому счету (реализация скрыта), не так ли? Поэтому использование глобальных переменных нарушает принцип инкапсуляции.

Вернемся к рассмотрению структуры `Complex`. Так, ее метод `povComplex()` – это как раз и есть тот самый интерфейс. Однако, как уже говорилось, поля структуры открыты для изменения, что противоречит принципу сокрытия данных (инкапсуляции), наглядно показанного на примере банковского счета. В этой возможности «открытой» модификации полей структуры, по сути, и кроется различие между структурами и классами.

Так, по умолчанию модификация полей разрешена в структуре, но запрещена в классе. Именно поэтому в подразд. 3.3 упоминалось, что замена ключевого слова `struct` на `class` не позволит получить желаемый результат в виде успешной компиляции программы. Для того чтобы использовать класс `Complex` вместо структуры `Complex` (в том виде как было до этого), надо разрешить доступ к его полям и методам с помощью спецификатора доступа `public`. (Далее приведено только объявление методов класса. Их реализация, описанная ранее, опущена для экономии места.)

```
1  class Complex{
2      public:
3          double real_d;
4          double image_d;
5          Complex(double, double);
6          Complex(double);
7          Complex();
8          void povComplex();
9  };
```

Спецификатор `public` предназначен для определения элементов класса (полей и методов), которые будут доступны извне объектов этого класса в рамках программы. Такая реализация класса позволяет получить полностью эквивалентную функциональность программы, как при использовании структуры.

Спецификатор `private` определяет, какие элементы класса (поля и методы) закрыты для доступа извне. При этом если при описании класса не использовать спецификаторы, то все его элементы считаются закрытыми (т. е. в классах

по умолчанию используется `private`). Тогда нельзя будет создавать объекты данного класса, т. к. доступ к конструкторам тоже закрыт. В результате по умолчанию для классов используется спецификатор `private`, а для структур, наоборот, `public`. Поэтому если убрать строку 2 в реализации класса `Complex`, то программный код не скомпилируется.

При описании классов можно чередовать объявление спецификаторов доступа. Действие одного спецификатора распространяется на программный код, расположенный вплоть до объявления другого спецификатора.

```
1  class Complex{
2      double image_d; // доступ извне класса закрыт
3      public:
4          Complex(double, double); // доступ открыт
5          Complex(double); // доступ открыт
6      public: // здесь можно убрать, ничего не изменится
7          Complex();// доступ открыт
8          void povComplex();// доступ открыт
9      private:
10         double real_d; // доступ закрыт
11 };
```

Стоит отметить, что существует третий спецификатор доступа – `protected`, который используется в случаях, когда классы образуют иерархию наследования. Так, если в классе объявлен элемент (метод или поле) с этим спецификатором, то он будет доступен из внутренних методов как базового, так и производного (унаследованного) классов. Особенности использования данного спецификатора подробнее рассмотрены в разделе 4, посвященном второму принципу ООП – наследованию.

3.3.6 Сеттеры и геттеры

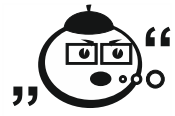
Как показано выше, указывать спецификаторы доступа можно для каждого элемента класса (или структуры) или для их групп. При этом в целях сокрытия данных для полей классов принято использовать спецификатор `private`, а для их методов (интерфейсов) – `public`. *Это сделано для того, чтобы модифицировать данные класса могли только его методы*. Поэтому для первоначальной инициализации полей при создании объектов класса используются открытые методы – конструкторы.

Для возможности дальнейшей модификации значений полей класса в рамках программы используются дополнительные методы, которые принято называть *сеттеры* (от англ. *set* – класть), а для считывания этих значений противоположные методы – *геттеры* (от англ. *get* – брать). С учетом сказанного реализация класса `Complex` примет следующий вид (объявления конструкторов и их реализации опущены):

```
1  class Complex{
2      public:
3          void povComplex();
4          void setValue(double, double); //объявление
5          Complex getValue(); // объявление
6      private:
7          double real_d;
8          double image_d;
9  };
10 void Complex::setValue(double re, double im){ // сеттер
11     real_d = re;
12     image_d = im;
13  }
14 Complex getValue(){ // геттер
15     return Complex(real_d, image_d);
16 }
```

Такая реализация класса позволяет обеспечить сокрытие данных (инкапсуляцию) за счет разделения интерфейса (набора открытых методов) класса и его реализации. При этом приведенный способ реализации сеттеров и геттеров не является единственным. Так, можно создать дополнительные геттеры и сеттеры, работающие раздельно с реальной и мнимой частями комплексного числа (показано в п. 3.3.8). Важно, что сначала принято описывать открытые члены класса, а затем закрытые. Делать наоборот допустимо, но не рекомендуется.

После пояснения назначения спецификаторов доступа вернемся к «самому интересному». Так, у читателя может возникнуть резонный вопрос: зачем нам классы, если есть структуры? Для ответа обратимся к мнению Г. Шилдта, автора популярных книг по программированию [10]:



«Тот факт, что и структуры, и классы обладают практически идентичными возможностями, создает впечатление избыточности. Многие новички в программировании на C++ недоумевают,

почему в нем существует такое очевидное дублирование. Нередко приходится слышать предложения отказаться от ненужного ключевого слова (`class` *или* `struct`) *и оставить только одно из них.*

Ответ на эту цепь рассуждений лежит в происхождении языка C++ от C и намерении сохранить C++ совместимым снизу вверх с C. В соответствии с современным определением C++ стандартная C-структура одновременно является совершенно законной C++-структурой. В языке C, который не содержит ключевых слов `public` *или* `private`*, все члены структуры являются открытыми. Вот почему и члены C++-структур по умолчанию являются открытыми* (*а не закрытыми*)*. Поскольку конструкция типа* `class` *специально разработана для поддержки инкапсуляции, есть определенный смысл в том, чтобы по умолчанию ее члены были закрытыми. Следовательно, чтобы избежать несовместимости с языком C в этом вопросе, аспекты доступа, действующие по умолчанию, менять было нельзя, поэтому и решено было добавить новое ключевое слово. Но в перспективе можно говорить о более веской причине для отделения структур от классов. Поскольку тип* `class` *синтаксически отделен от типа* `struct`*, определение класса вполне открыто для эволюционных изменений, которые синтаксически могут оказаться несовместимыми с C-подобными структурами. Поскольку мы имеем дело с двумя отдельными типами, будущее направление развития языка C++ не обременяется «моральными обязательствами», связанными с совместимостью с C-структурами».*

……

Надеемся, ознакомление с приведенными примерами и мнением эксперта укрепило понимание того, чем различаются структуры и классы. Так, если при написании программы используется объектно-ориентированный подход, рекомендуется применять классы (совокупность методов и полей), а если процедурный – структуры (объединение разнотипных переменных под одним именем).

Отметим, что при разработке классов программисты часто используют структуры для хранения составных переменных. Пожалуй, одним из ярких примеров этого является структура-пара `std::pair<>`, позволяющая обрабатывать два объекта, которые могут иметь разные типы данных, как один объект. Их главное назначение – возврат двух значений из функции и в качестве элементов контейнеров стандартной библиотеки шаблонов C++ (прил. Г). Наличие угловых

скобок у `std::pair<>` сигнализирует о том, что эта структура является шаблонной (аналогично шаблонным функциям из п. 2.3.8).

```
#include <iostream>
using namespace std;
pair<int, double> testFunc(){
    return pair<int, double> (1, 1e+2);
}
pair<int, double> testFunc1(){
    return make_pair(1, 1e+2);
}
int main( ){
   pair<int, double> p = testFunc();
   pair<int, double> p1 = testFunc1();
   cout << p.first <<" " << p.second << endl;
   cout << p1.first <<" " << p1.second << endl;
   return 0;
}
```

Здесь может возникнуть резонный вопрос: можно ли возвращать из функции три значения? Да, можно. При этом не только три, но и больше. Для этого используется кортеж `std::tuple<>`.

```
#include <iostream>
#include <tuple>
using namespace std;
tuple<int, double, string> testFunc(int x){
    return make_tuple(x, 10.2, "tuple");
}
tuple<int, double, string> testFunc1(int x, double y){
    return make_tuple(x+1, y+1, "tuple1");
}
int main( ){
    int a = 13;
    double b = 1.1;
    string str;
    tie(a, b, str) = testFunc(13);
    cout << a << " " << b << " " << str << endl;
    tuple<int, double, string> result;
    result = testFunc1(a, b);
```

```
18      cout << get<0>(result)<< " " << get<1>(result)<< " "
    << get<2>(result)<< endl;
19      return 0;
20  }
```

Аналогичную задачу по возвращению из функции нескольких значений можно решить путем передачи в нее параметров по ссылкам или указателям. Однако использование пары или кортежа является более наглядным.

3.3.7 Перегрузка операторов

Поскольку пособие посвящено ООП, то далее будем оперировать классами. Как трансформировать структуру в класс, было показано ранее. Поэтому, без потери общности, вернемся к арифметическим операциям с объектами классов.

Функция и метод `addComplex()`, приведенные в п. 3.3.5 и оперирующие с полями класса `Complex`, не очень удобны в использовании, т. к. более привычной выглядит запись «`c = a + b;`». Для возможности такого сложения комплексных чисел надо перезагрузить стандартный `operator+`.

Обратим внимание, что перегрузка операторов относится к третьему принципу ООП – полиморфизму, который подробно рассмотрен в разделе 5. Здесь лишь отметим, что для классов существует три способа перегрузки операторов: через методы класса; через обычные (внешние) функции; через дружественные функции, имеющие доступ к закрытым членам (полям и методам) класса. Более детальному рассмотрению дружественных функций, а также дружественных классов посвящен подразд. 4.4. Здесь дружественные функции рассматриваются лишь поверхностно.

Рассмотрим *использование внешней функции* для перегрузки `operator+`. Прямого доступа к полям класса у внешней функции нет, т. к. поля имеют спецификатор `private`. Тогда для доступа к полям следует использовать раздельные геттеры для реальной и мнимой частей комплексного числа соответственно. В результате получим два сеттера и два геттера. (Здесь для экономии места их реализация приведена непосредственно в теле класса.)

```
1   void setValueRe(double re){ // сеттер реальной части
2       real_d = re;
3   }
4   double getValueRe(){ // геттер реальной части
5       return real_d;
6   }
```

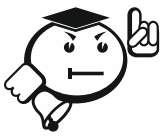
```
7  void setValueIm(double im){ // сеттер мнимой части
8      image_d = im;
9  }
10 double Complex::getValueIm(){ // геттер мнимой части
11     return image_d;
12 }
```

После того как выполнена реализация геттеров и сеттеров, их можно использовать во внешней функции для перегрузки `operator+`.

```
1  Complex operator+ (Complex c1, Complex c2){
2      Complex result;
3      result.setValueRe(c1.getValueRe()+c2.getValueRe());
4      result.setValueIm(c1.getValueIm()+c2.getValueIm());
5      return result;
6  }
```

Далее рассмотрим, как перегрузить `operator+` *через метод класса*. Для этого, как и ранее, используется указатель `this`.

```
1  Complex operator+ (Complex c1){
2      Complex result;
3      result.real_d = this->real_d + c1.real_d;
4      result.image_d = this->image_d + c1.image_d;
5      return result;
6  }
```



Указатель `this` всегда хранит адрес текущего объекта класса. При этом обращаться через него к любым членам класса можно только внутри класса.

Для того чтобы вывести на печать полученные результаты, надо реализовать соответствующий метод класса, внешнюю или дружественную функции. Еще один вариант – перегрузить оператор потокового вывода (`operator<<`). Сначала используем метод класса `showComplex()`. Так как это тоже интерфейс, то для его реализации надо использовать спецификатор `public`.

```
1  public:
2      void showComplex (){
```

```
3         cout <<"(" <<real_d <<"," <<image_d <<")" <<endl;
4     }
```

Если для перегрузки используется внешняя функция, то для ее реализации надо пользоваться геттерами.

```
1  void showComplex (Complex a){
2      cout <<"("<<a.getValueRe()<<","<<a.getValueIm()<<")"
   << endl;
3  }
```

При использовании дружественной функции (ключевое слово `friend`), т. е. функции, имеющей доступ к полям и методам класса, пользоваться геттерами не нужно. Однако внутри класса надо ее объявить.

```
1  public:
2      friend void showFriComplex (Complex);
```

Затем надо выполнить ее реализацию вне класса. При этом ключевое слово `friend` не используется.

```
1  void showFriComplex (Complex a){
2      cout <<"(" <<a.real_d <<"," <<a.image_d <<")"<<endl;
3  }
```

Тогда для сложения комплексных чисел может быть использована «привычная» запись, а результат может быть выведен на печать.

```
1  Complex a_c(1,-2);
2  Complex b_c(4,5);
3  Complex c_c;
4  c_c = a_c + b_c; // (5,3)
5  c_c.showComplex(); // использование метода класса
6  showComplex(c_c); // использование внешней функции
7  showFriComplex(c_c); // использование дружественной
   функции
```

Как уже было отмечено ранее, перегрузка оператора вывода (`operator<<`) и других операторов рассмотрена в разделе 5, посвященном полиморфизму.

Выводы

Члены класса (поля и методы), объявленные публичными (спецификатор `public`), могут быть напрямую использованы любой функцией из всех частей

программы. Приватные члены класса (спецификатор `private`) могут использоваться только методами класса и друзьями (классами или функциями) класса. По умолчанию (в классе отсутствует спецификатор `public`) все члены класса являются закрытыми.

Для наглядности покажем, как выглядят аналогичные внешние функции для структуры `Complex`. Так как поля структуры открыты для модификации, внешняя функция `operator+()` может оперировать с ее полями напрямую, без необходимости использования геттеров. Аналогичное справедливо и для функции вывода на печать. По этой же причине использование дружественной функции при реализации структуры по умолчанию или со спецификатором `public` не имеет смысла. Это еще раз наглядно демонстрирует различие между структурами и классами.

```
1  Complex operator+ (Complex c1, Complex c2){
2      Complex result;
3      result.real_d = c2.real_d + c1.real_d;
4      result.image_d = c2.image_d + c1.image_d;
5      return result;
6  }
7  void showComplex (Complex a_s){
8      cout << "(" << a_s.real_d << "," << a_s.image_d
   <<")" << endl;
9  }
```

Рекомендации по перегрузке операторов приведены в разделе 5. Здесь лишь отметим, что если стоит выбор между перегрузкой некого оператора через внешнюю или дружественную функции, то стоит использовать первую из них, если в классе уже есть геттеры. Так как, чем меньше функций касается внутренних элементов вашего класса, тем лучше.

Однако не следует добавлять дополнительный геттер только для того, чтобы перегрузить оператор через обычную функцию. Если геттера нет по умолчанию или он не используется вообще (в нем нет необходимости), то стоит использовать перегрузку через дружественную функцию.

Выводы

Главное отличие структур от классов заключается в том, что по умолчанию первые из них открыты для доступа, а вторые, наоборот, закрыты.

3.3.8 Массивы объектов

Напомним, что массив – это последовательность объектов, имеющих один и тот же тип данных и занимающих непрерывную область памяти. Поскольку класс является пользовательским типом данных, то можно создать массивы из его объектов. Например, рассмотрим, как создать массив из объектов класса `Complex` и оперировать ими.

```
1  Complex arrCompl[10];
2  arrCompl[0].setValue(1,-1); // сеттер
3  arrCompl[1].setValue(100,-100); // сеттер
4  arrCompl[0].showComplex();
5  arrCompl[1].showComplex();
```

Здесь сначала указывается элемент массива через его индекс (например, `arrCompl[0]`), а затем вызывается требуемый метод. Как видно, нет принципиального различия между работой с элементом массива объектов класса и отдельным объектом класса.

3.3.9 Пользовательский класс Complex: часть 1

Сведем воедино весь разработанный класс `Complex`. Для работы с его полями использованы только методы класса. При этом, согласно рекомендациям из п. 2.3.10, объявление и реализация элементов класса разделены на два файла: `Complex.h` и `Complex.cpp`. Пример работы с его объектом помещен в файл `main.cpp`. (Надеемся, что пытливый читатель проверит работоспособность всех методов класса.)

```
                        // файл Complex.h
1  #ifndef COMPLEX_H
2  #define COMPLEX_H
3  #include <iostream>
4  class Complex{
5      public: // объявление открытых членов класса
```

```
6         Complex(double, double); // конструктор 1
7         Complex(double); // конструктор 2
8         Complex(); // конструктор по умолчанию
9         void povComplex();
10        void setValue(double, double); // сеттер
11        Complex getValue(); // геттер
12        void setValueRe(double); // сеттер реальной части
13        double getValueRe(); // геттер реальной части
14        void setValueIm(double); // сеттер мнимой части
15        double getValueIm(); // геттер мнимой части
16        Complex operator+ (Complex);
17        void showComplex ();
18    private: // объявление закрытых членов класса
19        double real_d; // реальная часть
20        double image_d; // мнимая часть
21 };
22 #endif // COMPLEX_H
```

// файл Complex.cpp

```
1  #include "Complex.h"
2  using namespace std;
3  Complex::Complex(double x, double y): real_d(x),
   image_d(y) { }
4  Complex::Complex(double x): real_d(x), image_d(0) { }
5  Complex::Complex(): real_d(0), image_d(0) { }
6  void Complex::povComplex(){
7      real_d = real_d * real_d;
8      image_d = image_d * image_d;
9  }
10 void Complex::setValue(double re, double im){ // сеттер
11     real_d = re; image_d = im;
12  }
13 Complex Complex::getValue(){ // геттер
14     return Complex(real_d, image_d);
15 }
16 void Complex::setValueRe(double re){ // сеттер
17     real_d = re;
18 }
19 double Complex::getValueRe(){ // геттер
20     return real_d;
```

```
}
void Complex::setValueIm(double im){ // сеттер
    image_d = im;
}
double Complex::getValueIm(){ // геттер
    return image_d;
}
Complex Complex::operator+ (Complex c1){ // перегрузка
    Complex result;
    result.real_d = this->real_d + c1.real_d;
    result.image_d = this->image_d + c1.image_d;
    return result;
}
void Complex::showComplex (){
    cout << "(" <<real_d << "," <<image_d << ")"<< endl;
}
```

// файл main.cpp

```
#include "Complex.h"
int main(){
    Complex a(0,1);
    a.setValueRe(100);
    a.showComplex();
    // остальной код программы
 return 0;
}
```

Важно отметить, что поля класса хоть и могут быть открытыми, но так делать нельзя, т. к. это нарушает принцип инкапсуляции (сокрытия данных).

Выводы

С полностью инкапсулированным классом нужно знать, какие методы являются доступными (интерфейсными) для использования, какие аргументы они принимают и какие значения возвращают, и не нужно знать, как он реализован внутри. Это значительно снижает сложность разрабатываемых программ, а также уменьшает количество возможных ошибок. Это ключевое преимущество инкапсуляции. Кроме того, классы с открытыми полями имеют ту же проблему

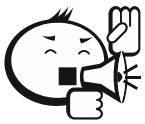
(в меньших масштабах), что и глобальные переменные, которые «опасны», поскольку нет строгого контроля за тем, кто имеет к ним доступ и как он их использует.

3.3.10 Динамическое создание объектов

При создании объекта класса (и структуры тоже) предпочтительным является выделение памяти в куче под его хранение с помощью указателя. Для этого используется оператор (ключевое слово) `new`. Когда работа с объектом будет закончена, следует освободить выделенную память с помощью оператора (ключевое слово) `delete` (подразд. 2.2).

```
1  Complex a(1,2); // статическое создание объекта
2  a.showComplex();
3  Complex *c = new Complex(1,12); //динамическое создание
4  c->showComplex();
5  delete c; // освобождение динамически выделенной памяти
```

Отметим, что для доступа к полям статически созданного объекта используется оператор «.» (точка), а динамически созданного – оператор «->» (стрелка).



Сколько памяти занимает переменная, например, типа `int`, известно априори. А сколько памяти требуется для хранения объектов класса `Complex`, созданных статически и динамически?

В целом ответ на поставленный вопрос очевиден. Но мы же программисты, поэтому заставим программу ответить на этот вопрос.

```
1  #include <iostream>
2  using namespace std;
3  class ComplexTest{
4      public:
5          ComplexTest(double x, double y): real_d(x),
   image_d(y) { }
6      private:
7          double real_d;
8          double image_d;
9  };
```

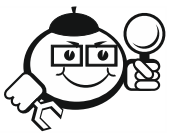
```
10 int main(){
11     double x = 5;
12     ComplexTest sC(1,-1);
13     ComplexTest *dC = new ComplexTest(1,-1);
14     cout << sizeof(x) << endl; // 8 (байт)
15     cout << sizeof(double) << endl; // 8(байт)
16     cout << sizeof(sC) << endl; // 16 (байт)
17     cout << sizeof(*dC) << endl; // 16 (байт)
18     cout << sizeof(ComplexTest) << endl; // 16 (байт)
19     cout << sizeof(4 + 5) << endl; // 4 (байта)
20     cout << sizeof(4 + 5.01) << endl; // 8(байт)
21     cout << sizeof(" -TUSUR- ") << endl; // 6 (байт)
22     delete []dC;
23     return 0;
24 }
```

Здесь использован унарный оператор (ключевое слово) `sizeof()`. (Напомним, что этот оператор также есть и в языке С.) Этот оператор возвращает затрачиваемый объем памяти для хранения объекта, указанного в скобках оператора. Отметим, что объектом может быть тип данных (строки 15 и 18), имя переменной (строка 14) или объекта класса (строки 16 и 17) и выражение (строки 19 и 20) и пр. (например, строка 21).

Так как класс – это пользовательский тип данных, то требуемый объем машинной памяти для хранения объекта класса определяется суммарными затратами на хранение всех полей класса, на основе их типов данных. В используемом классе `ComplexTest` есть два поля с типом данных `double`, поэтому суммарные затраты памяти для хранения его объекта составляют 16 байт. (Если класс пустой, т. е. не содержит полей, то затраты памяти составляют 1 байт для хранения адреса класса.)

3.3.11 Шаблоны классов: пользовательский класс Vector

Как показано в п. 2.3.8, шаблоны позволяют разрабатывать обобщенные функции, что существенно упрощает написание кода. Поэтому в C++ есть возможность реализовывать даже шаблоны классов. На первый взгляд, это может показаться ненужным, однако это не так. Следующий пример призван показать, что шаблоны классов – это крайне важный механизм языка C++, который лежит в основе стандартной библиотеки шаблонов, содержащей классы, реализующие эффективные алгоритмы и структуры данных (прил. Г).



Пример

Задача. Разработать класс `Vector` (с минимальным функционалом), предназначенный для работы с одномерными массивами произвольного размера. Для тестирования класса, при задании типов данных объектов этого класса, использовать разработанный ранее класс `Complex` и тип данных `double`.

Решение. Сначала выполним проектирование класса `Vector`, а затем его реализацию. Так, в п. 2.4.1 указано, что шаблоны должны помещаться только в заголовочных файлах, поэтому для реализации шаблона создадим файл `Vector.h`. При его подключении используем защитный механизм. Закрытым полем класса будет размер массива. Для инициализации объектов класса используем конструктор. Так как размер массива заранее не известен, то используем динамическое выделение памяти (используемый указатель должен быть закрытым), а для заполнения массива при его создании – стандартный генератор случайных чисел. Из-за использования динамического выделения памяти также должен быть реализован деструктор. Для визуального контроля работы программы понадобится метод вывода в консоль, назовем его `Print()`. С учетом сказанного разработанный шаблон будет иметь следующий вид:

```
                    // файл Vector.h
1  #ifndef VECTOR_H
2  #define VECTOR_H
3  #include "Complex.h"
4  using namespace std;
5  template <class Type>
6  class Vector{
7     public:
8        Vector(int n){
9           N=n;
10          ptrArr = new Type [N];
11          for (auto count = 0; count < N; count ++)
12             ptrArr[count] = rand();
13       }
14       void Print(){
15          if constexpr (is_same<Type, Complex>::value)
16             for (auto count = 0; count < N; count ++)
17                ptrArr[count].showComplex();
```

```
            else
               for (auto count = 0; count < N; count ++)
                  cout<< ptrArr[count] << endl;
      }
      ~Vector(){
         delete [] ptrArr;
      }
   private:
      int N;
      Type* ptrArr = nullptr;
};
#endif // VECTOR_H
```

Выполним тестирование разработанного шаблона класса `Vector`.

```
                  // файл main.cpp
#include "Complex.h"
#include "Vector.h"
int main(){
   Vector<Complex> mC(4);
   mC.Print();
   Vector<double> mD(4);
   mD.Print();
   return 0;
}
```

Результатат работы программы – следующие строки (будут меняться при каждом запуске, т. к. для заполнения использован генератор случайных чисел):

```
(1.80429e+09,0)
(8.46931e+08,0)
(1.68169e+09,0)
(1.71464e+09,0)
1.95775e+09
4.24238e+08
7.19885e+08
1.64976e+09
```

Как видно, шаблон спроектирован корректно.

Выполним анализ программной реализации шаблона класса `Vector` из рассмотренного примера. Наибольшее внимание привлекает метод `Print()`, а

точнее его содержимое. Поскольку в используемой реализации класса `Complex` не перегружен `operator<<`, то строка «`cout << ptrArr[count] << endl;`» подходит только для типа данных `double`, а для типа данных `Complex` использован имеющийся в классе метод `showComplex()`. При этом потребовалось внести в метод `Print()` конструкцию `if constexpr – else`, позволяющую разделить его работу в зависимости от используемого типа данных. Так, использовано ключевое слово `constexpr`, уточняющее оператор `if` (так называемый статический `if`).

Может возникнуть резонный вопрос: почему не использован стандартный вариант оператора `if`? Ответ прост: код не скомпилируется. Это легко проверить, убрав в строке 15 слово `constexpr` и выполнив компиляцию кода. В результате получим ошибку: компилятор не может «справиться» с типом `Complex` в строке 20. Связано это с тем, что компилятор, разбирая обе ветки ветвления, не может создать экземпляры класса по аргументу шаблона здесь для `Complex`. Главная идея `if constexpr` заключается в том, чтобы использовать или игнорировать участки кода при компиляции в зависимости от значения константы времени компиляции[1], выражаемого условием оператора `if`. В данном случае это условие является проверкой типа данных. Для этого в строке 15 использовано сравнение типов данных: шаблонного типа `Type` и `Complex` (`is_same<Type, Complex>::value`). Если типы данных совпадают, то выполняются строки 16 и 17, иначе – 19 и 20.

Стоит отметить, что при необходимости также может быть использована конструкция `if constexpr - else if constexpr – else`. Для демонстрации дополнительно сделаем исключение для типа данных `char`.

```
1  void Print(){
2     if constexpr (is_same<Type, Complex> :: value)
3        for (auto count = 0; count < N; count ++)
4           ptrArr[count].showComplex();
5        else if constexpr (is_same<Type, char> :: value)
6           for (auto count = 0; count < N; count ++)
```

[1]В C++ используется два вида констант: 1) константы времени выполнения – их значения вычисляются только во время работы программы (например, введенное пользователем значение); 2) константы времени компиляции – значения, которые вычисляются во время компиляции программы (например, скорость света в вакууме «`const double c = 299 792 458`»). При этом использование вторых позволяет компилятору выполнить оптимизацию кода, недоступную для первых.

```
7                cout << " bad value " << endl;
8          Else
9             for (auto count = 0; count < N; count ++)
10               cout << ptrArr[count] << endl;
11   }
```

Как видно, в этой версии метода `Print()` использовано уже три частных случая: `Complex`, `char` и остальные типы данных.

Вторым вариантом уточнения работы метода `Print()` для некоторых типов данных, без учета варианта перегрузки `operator<<`, является создание его отдельных версий. Это так называемая частичная специализация шаблона. Для этого *за пределами шаблона класса* надо поместить его экземпляр с указанием требуемого типа данных (например, сразу после описания шаблона класса в файле `Vector.h`). Тогда при необходимости будет использована одна из «уточненных» версий метода `Print()`. Так как тип данных известен, он перестает быть шаблонным типом, поэтому шаблон определяется как `template<>`.

```
1   template<>
2   void Vector<Complex> :: Print(){
3      for (auto count = 0; count < N; count ++)
4         ptrArr[count].showComplex();
5   }
6   template<>
7   void Vector<char> :: Print(){
8    for (auto count = 0; count < N; count ++)
9           cout << " bad value " << endl;
10  }
```

При работе с остальными типами данных метод `Print()` имеет «стандартный» вид.

```
1   void Print(){
2      for (auto count = 0; count < N; count ++)
3         cout << ptrArr[count] << endl;
4   }
```

В п. 2.3.7 было показано, что использование ключевого слова `using` позволяет улучшить читаемость кода за счет создания псевдонима типа для «сокращения» сложных типов данных. При этом использование `using` не

ограничивается стандартными типами данных и применимо для создания псевдонимов любых составных типов. Так, вместо `Vector<Complex>` можно использовать более наглядную и короткую запись, например `vectC`.

```
1  #include "Complex.h"
2  #include "Vector.h"
3  using vectC = Vector<Complex>;
4  using vectD = Vector<double>;
5  int main(){
6     vectC mC(4); // Vector<Complex> mC(4);
7     mC.Print();
8     vectC mD(4); // Vector<double> mD(4);
9     mD.Print();
10    return 0;
11 }
```

3.3.12 Лямбда-выражения

На практике часто возникают ситуации, когда параметр функции (метода) определяется результатом выполнения другой функции (метода). Тогда приходится последовательно вызывать эти две функции (методы). Для удобства в C++ внедрен механизм лямбда-выражений (лямбда-функции), который позволяет определять функцию (метод) внутри другой функции (метода).

Назвать синтаксис лямбда-выражения интуитивно понятным сложно, потребуется привыкнуть к нему. Так, эти выражения не имеют явных имен. Поэтому их еще иногда называют анонимными функциями. Общий вид лямбда-выражения:

```
[переменные] (параметры) -> тип_возвращаемого_значения {
   // требуемый программный код
}
```

Поля лямбда-выражения могут быть заданы или оказаться пустыми. Если у выражения нет параметров, то круглые скобки можно дополнительно опустить.

```
1  [](){}; // или [](){}; - λ1
2  [](int a){ // λ2
3     return (i % 3);
4  }
5  [](float a, float b) -> float{ // λ3
```

```
6       return (a / b);
7   }
```

Приведенные примеры демонстрируют, что подразумевается под параметрами лямбда-выражения. При этом очень важно, что параметром может быть и указатель. С типом возвращаемого значения все интуитивно понятно. Однако, если этот тип явно не указан, то он определяется с помощью ключевого слова `auto`. Осталось только определиться, как их использовать. Для этого рассмотрим несколько примеров.

```
1   #include <iostream>
2   using namespace std;
3   int main(){
4   // пример 1
5       []{ cout<<"lambda-define"<<endl;}; // объявление
6       []{ cout<<"lambda-using"<<endl;}(); // использование
7   // пример 2
8       auto divFloatNumbers{ // объявление
9           [](float a, float b) -> float{ // λ
10              return (a / b);
11          }
12      };
13      cout<< divFloatNumbers(13, 2)<< endl; //использование
14  // пример 3
15      float a = 13.1, b = 3.6;
16      auto result = abs(
17        [](float x, float y) {return (x + y); }(a, b) // λ
18      );
19      cout << result << endl;
20      return 0;
21  }
```

При использовании (вызове) лямбда-выражения в конце строки надо указать возвращаемое значение (пример 2). В последнем примере объявляется и сразу же вызывается лямбда-выражение для переменных `a` и `b`. Здесь лямбда-выражение использовано в качестве аргумента стандартной функции нахождения абсолютного значения числа `abs()`.

Лямбда-выражения также можно вкладывать одно в другое. Например, для примера 3 можно использовать следующую запись, дающую тот же результат. При этом требуется указать захватываемую (по значению) переменную.

```
auto result = abs(
   [b](float x){
      return x + [](float y){
         return y;
      }(b);
   }(a)
);
```

Поясним смысл захватываемых переменных. Внутри лямбда-выражения действует своя область видимости, поэтому внешние переменные ей неизвестны. Для того чтобы переменная была известна внутри области видимости лямбда-выражения, как раз и используется захват переменных. Как уже было показано выше, если поле захвата оставить пустым, то захвата не происходит и лямбда-выражение работает только со своими локальными переменными. В примере 3 использован захват по значению переменной `b`. При этом аналогично обычным функциям можно захватывать (передавать) переменные по ссылке, а также указывать целые списки переменных с указанием типа их захвата:

- `[]` – все внешние переменные не захватываются;
- `[=]` – все переменные захватываются по значениям;
- `[&]` – все переменные захватываются по ссылкам;
- `[&a, b]` – переменная `a` захватывается по ссылке, а `b` – по значению;
- `[=, &b]` – все переменные захватываются по значениям, а `b` – по ссылке.

Рассмотрим простые примеры использования различных типов захвата переменных. Отметим, что лямбда-выражения можно также сохранять в объекты для последующего использования.

```
#include <iostream>
using namespace std;
int main(){
   int a = 2;
   [a]{ cout << "a=" << a << endl; }();
   [a]{ a++; }(); // ошибка изменения значения
   int b = 12;
   [&b]{ b++; }();
   cout << "b=" << b << endl;
   int c = 13;
   [a,c]{ cout << "ac_1=" << ac= << endl; }();
```

```
12    [](int  x,  int  y){cout<<"ac2="<<x*y<<endl;  }(a,b);
   // аналог строки 11
13    [a, &c]{ c = c / a; }();
14    cout << "c=" << c << endl;
15    [&](){ a++, b += 100; c--; }();
16    cout << "a_new= " << a << endl;
17    cout << "b_new= " << b << endl;
18    cout << "c_new=" << c << endl;
19     [=, &c](){ c = a * b; }();
20    cout << "c_new2= " << c << endl;
21     return 0;
22 }
```

На примере метода `Print()` класса `Vector`, разработанного ранее, продемонстрируем, как можно использовать лямбда-выражения в методах класса. В таком случае нужен захват указателя `this`, т. к. лямбда-выражение не имеет доступа к полям класса (`N` и `ptrArr`). Иначе код не скомпилируется.

```
1  void Print(){
2     [this](){ // захват указателя this
3        for (auto count = 0; count < N; count ++)
4           cout << ptrArr[count] << endl;
5     }();
6  }
```

Отметим, что чаще всего лямбда-выражения применяются при работе с алгоритмами и шаблонами из стандартной библиотеки C++.

3.4 Итоговые замечания

Вернемся к классу `Complex` из п. 3.3.10. Сначала рассмотрим некоторые аспекты эффективности его реализации, а затем – как сделать его шаблонным.

В разработанной версии класса `Complex` использовано три конструктора: по умолчанию `Complex()`, с одним `Complex(double)` и двумя `Complex(double, double)` параметрами. Заострим внимание на втором конструкторе.

```
1  Complex :: Complex(double x) : real_d(x), image_d(0) { }
```

При создании объекта, например «`Complex(3.);`», мнимая часть комплексного числа равна нулю, т. е. получим 3 + *i*0. Однако такой конструктор, по сути, осуществляет неявное приведение типа `Complex` к `double`. Аналогичный

результат получим, если используем конструктор с двумя параметрами, задав второй из них по умолчанию равным нулю.

```
Complex :: Complex(double x, double y = 0) : real_d(x),
image_d(y) { }
```

Тогда можно отказаться от конструктора с одним параметром, а его вариант будет отрабатывать конструктор с двумя параметрами. Однако в этом случае также возможно приведение `Complex` к `double`. В рассматриваемой ситуации это вполне разумно, но в некоторых случаях может быть недопустимо. Поэтому рекомендуется объявлять конструкторы как `explicit`, что запрещает неявное приведение типов. Кроме того, требуется объявление и определение этого конструктора разместить только в заголовочном файле `Complex.h`, чтобы «пояснить» компилятору все варианты использования этого конструктора.

```
explicit Complex(double x, double y = 0) : real_d(x),
image_d(y) { }
```

Может возникнуть резонный вопрос, если он не возник ранее: нельзя ли использовать один конструктор «на все случаи жизни»? Ответ будет положительным.

```
explicit Complex(double x = 0, double y = 0): real_d(x),
image_d(y) { }
```

Этот конструктор отрабатывает за все три разработанных ранее. Продемонстрируем это на примере (файл `main.cpp`).

```
#include "Complex.h"
int main(){
    Complex a;
    a.showComplex(); // (0,0)
    Complex b(3);
    b.showComplex(); // (3,0)
    Complex c(3,3);
    c.showComplex(); // (3,3)
    Complex arr[]{Complex(-3,-1),Complex(), Complex(0)};
    arr[0].showComplex(); // (-3,-1)
    arr[1].showComplex(); // (0,0)
    arr[2].showComplex(); // (0,0)
    return 0;
}
```

Далее рассмотрим перегрузку operator+. В разработанной версии класса использована перегрузка через метод класса «Complex operator+ (Complex);».

Во-первых, здесь указан тип возвращаемого значения Complex. Однако более правильно указывать тип как const Complex, т. к. это гарантирует невозможность присваивания «a + b = c;» (ошибка компиляции). Иначе оно возможно (нет ошибки компиляции).

Во-вторых, operator+ не должен менять значение передаваемого параметра, поэтому он должен быть константным.

Наконец, в-третьих, для большей эффективности перегрузки параметр надо передавать не по значению, а по ссылке.

Отдельно отметим, без приведения кода, что модернизация перегрузки operator+ на этом вообще-то не заканчивается, т. к. для перегрузки бинарных операторов рекомендуется использовать внешние или дружественные функции, а не метод класса, как сделано в рассматриваемом случае. При этом для повышения качества перегрузки operator+ сначала надо через метод класса перегрузить operator+=, а затем с его помощью перегрузить operator+ через дружественную или внешнюю функцию.

Теперь рассмотрим, как сделать класс Complex шаблонным. Очевидно, что его разработанная версия является «усеченной», т. к. она работает только с типом данных double. Для создания универсальной версии его надо сделать шаблонным, по аналогии с классом Vector. Тогда определение и реализация шаблона класса Complex должны быть помещены в заголовочный файл.

```
                    // файл Complex.h
1   #ifndef COMPLEX_H
2   #define COMPLEX_H
3   #include <iostream>
4   using namespace std;
5   template <class T>
6   class Complex{
7      public:
8         explicit Complex(T x=0,T y=0): real(x), image(y){}
9         void povComplex(){
10           real = real * real; image = image * image;
11        }
12        void setValue(T re, T im){real = re; image = im;}
```

```
13        Complex getValue(){return Complex(real, image);}
14        void setValueRe(T re){ real = re;}
15        T getValueRe(){ return real;}
16        void setValueIm(T im){ image = im;}
17        T getValueIm(){ return image;}
18        void showComplex (){
19           cout<< "(" <<real << "," <<image << ")"<< endl;
20        }
21        const Complex operator+ ( const Complex &c1){
22           Complex result;
23           result.real = this->real + c1.real;
24           result.image = this->image + c1.image;
25           return result;
26        }
27     private:
28        T real;
29        T image;
30  };
31  #endif // COMPLEX_H
```

Наконец, рассмотрим, как изменится шаблонный класс `Vector<>`. Так, из-за того, что класс `Complex` стал шаблонным (`Complex<>`), для создания объекта шаблонного класса `Vector<>` надо использовать запись вида `Vector<Complex<int>>`. Здесь для `Complex<>` шаблонным типом данных является `int`, а для `Vector<>` – `Complex<int>`. Однако для каждого используемого шаблонного типа класса `Complex<>` (из-за отсутствия перегрузки `operator <<` для объектов этого класса) потребуется частичная специализация шаблона класса `Vector<>`. Так, если требуемыми типами данных являются, например, `Complex<int>`, `Complex<float>` и `Complex<double>`, то потребуется три варианта специализации шаблонного класса `Vector<>`.

// файл Vector.h

```
1  #ifndef VECTOR_H
2  #define VECTOR_H
3  #include "Complex.h"
4  using namespace std;
5  template <class Type>
6  class Vector{
7     public:
```

```
8         explicit Vector(int n){
9            N=n;
10           ptrArr = new Type [N];
11            for (auto count = 0; count < N; count ++)
12               ptrArr[count] = static_cast<Type>(rand());
13        }
14        void Print(){
15           for (auto count = 0; count < N; count ++)
16               cout << ptrArr[count] << endl;
17        }
18        ~Vector(){ delete [] ptrArr; }
19     private:
20        int N;
21        Type* ptrArr = nullptr;
22 };
23 template <>
24 void Vector<Complex<double>> :: Print(){
25  for (auto count = 0; count < N; count ++)
26         ptrArr[count].showComplex();
27 }
28 template <>
29 void Vector<Complex<int>> :: Print(){
30  for (auto count = 0; count < N; count ++)
31         ptrArr[count].showComplex();
32 }
33 template <>
34 void Vector<Complex<float>> :: Print(){
35  for (auto count = 0; count < N; count ++)
36         ptrArr[count].showComplex();
37 }
38 template <>
39 void Vector<char> :: Print(){
40  for (auto count = 0; count < N; count ++)
41         cout << "bad value" << endl;
42 }
43 #endif // VECTOR_H
```

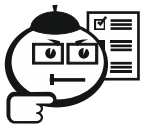
Стоит обратить внимание на строку 12, где использовано явное приведение типов. Теперь шаблонные классы `Vector<>` и `Complex<>` можно использовать совместно.

```
#include "Complex.h"
#include "Vector.h"
int main(){
    Vector<Complex<int>> vecCI(2);
    vecCI.Print();
    Vector<Complex<double>> vecCD(2);
    vecCD.Print();
    Vector<Complex<float>> vecCF(2);
    vecCF.Print();
    Vector<char> vecC(2);
    vecC.Print();
    Vector<int> vecI(2);
    vecI.Print();
    return 0;
}
```

Контрольные вопросы по разделу 3

1. Что такое абстракция данных?
2. Для чего используется инкапсуляция?
3. Чем спецификатор `public` отличается от спецификатора `private`?
4. В чем различие между структурой и классом?
5. Что такое объект класса?
6. Какие есть способы для перегрузки арифметических операторов?
7. Можно ли использовать структуры вместо классов?
8. Как принято называть переменные класса?
9. Что такое метод класса?
10. Что такое интерфейс класса?
11. Можно ли формировать массивы из объектов класса?
12. Что такое конструктор класса?
13. Сколько конструкторов может быть у класса?
14. Зачем нужны деструкторы?
15. Когда при работе с объектами класса используются операторы «.» (точка) и «->» (стрелка)?
16. Для чего используются шаблоны классов?
17. Может ли один класс использовать приватные методы другого класса?

18. Чем `if` отличается от статического `if`?
19. Зачем нужны лямбда-выражения?



Задания для самостоятельного выполнения

1. Используя принцип инкапсуляции, разработать класс `Book`. Класс должен содержать информацию (поля) о названии книги, Ф. И. О. автора, названии издательства, годе издания, числе страниц, тираже (числе экземпляров), стоимости одного экземпляра. Требуется реализовать конструктор(ы) и методы (функции), позволяющие заполнять (сеттеры) и извлекать (геттеры) эти данные с использованием пользовательского ввода и вывода в консоль. Дружественные и внешние функции не использовать.
2. Доработать разработанный класс `Book`, реализовав проверку входных данных перед их сохранением в соответствующие поля объекта класса. Так, например, поля «год издания» и «ФИО» не могут иметь значения «`A`» и «`123456`» соответственно.
3. Разработать пользовательский способ сложения объектов класса `Book` (например, по числу страниц и общей стоимости тиражей). Для этого выполнить доработку класса `Book`, прегрузив `operator+` с использованием внешней функции.
4. Разработать шаблон класса `Library`, позволяющий использовать класс `Book` в качестве его шаблонного типа данных.

4 Объектно-ориентированное программирование: наследование

Я понимаю КАК; не понимаю ЗАЧЕМ.

Дж. Оруэлл. 1984

4.1 Вводные замечания

В предыдущем разделе показано, как создавать раздельные классы и как ими совместно оперировать. При этом новые классы (нет ограничения на их число) могут быть производными от ранее созданных. Этот принцип в ООП называется наследованием.

*Класс, используемый для наследования, называется **базовым классом**, а класс, который использует его программный код, – **производным**.*

Идея этого принципа ООП, основанного на разделении, заключается в том, что каждый производный класс обладает всеми свойствами, присущими базовому классу, а также новыми и уникальными для него свойствами.

Допустим, требуется разработать программу для университета, которая должна выполнять набор специфических функций в зависимости от роли пользователя. Такими ролями могут быть: новый пользователь, студент и преподаватель. Нового пользователя, как и любого человека, можно описать именем собственным. При этом студент и преподаватель дополнительно к этому имеют отличительные особенности (свойства), например такие, как номер группы и преподаваемый предмет. Для описания данных ролей создадим три класса: `Human`, `Student` и `Lecturer`. При этом, для ясности изложения, сначала все их поля намеренно сделаем публичными (спецификатор `public`).

```
1  class Human{
2     public:
3        string name;
4  };
5  class Student{
6     public:
```

```
7       string name;
8       string group;
9   };
10  class Lecturer{
11     public:
12      string name;
13      string subject;
14  };
```

Здесь поле `name` встречается во всех трех классах, что порождает избыточность кода. При этом очевидно, что класс `Human` является более общим, т. к. студенты и преподаватели – это прежде всего люди. Поэтому можно унаследовать классы `Student` и `Lecturer` от класса `Human`, передав производным классам «в наследство» его поле `name`.

Синтаксис наследования классов в языке C++ имеет следующий вид:

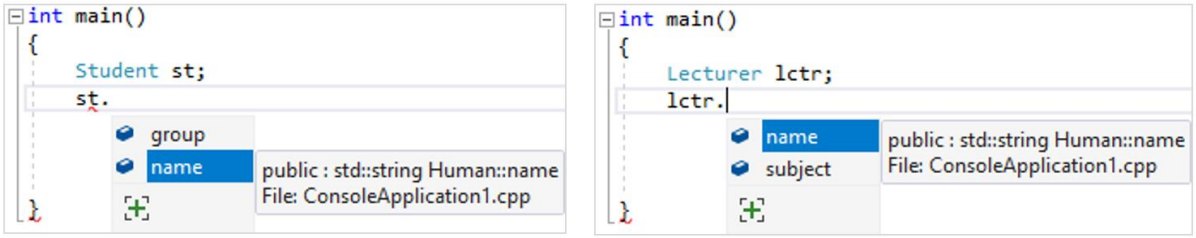
```
1   class DeliverClass : [Modifier] BaseClass{
2   // требуемый код
3   };
```

Здесь `DeliverClass` – производный класс, `BaseClass` – базовый класс, а `[Modifier]` – спецификатор доступа. Квадратные скобки у последнего сигнализируют о том, что он может отсутствовать (задан по умолчанию).

Изменим классы `Student` и `Lecturer`, добавив после их имен разделители «`:`», модификаторы наследования `public` и имя базового класса (от кого они наследуются).

```
1   class Human{
2      public:
3         string name;
4   };
5   class Student : public Human{
6      public:
7         string group;
8   };
9   class Lecturer : public Human {
10     public:
11        string subject;
12  };
```

Теперь при инициализации объектов производных классов в списке их членов содержатся как их уникальные поля (`group` и `subject`), так и унаследованное от базового класса поле `name` (рис. 4.1).



а) *б*)

Рис. 4.1 – Списки членов классов в MVS: `Student` (*а*) и `Lecturer` (*б*)

Механизм наследования можно считать аналогом функций в процедурном программировании. Если в коде содержатся функции, выполняющие близкие действия, то надо извлечь из них одинаковые части и поместить их в общую для них функцию. Тогда модифицированные функции будут одинаково вызывать эту общую функцию. Также и наследование позволяет упростить связи между элементами программы за счет их организации в виде иерархии классов (рис. 4.2). Для наглядного представления иерархии классов широко применяется унифицированный язык моделирования UML (прил. Ж).

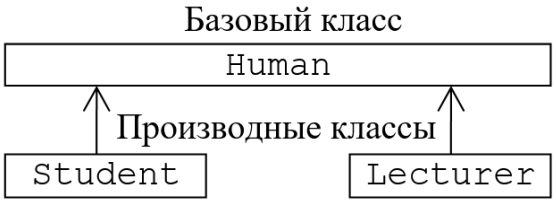


Рис. 4.2 – Иерархия классов: `Human` – `Student` и `Lecturer`

Построение иерархии классов производится с помощью выделения существенных признаков объекта, отличающих его от других объектов, при этом:

- объекты производных классов должны обладать всеми свойствами базового класса;
- наследуемые свойства базового класса не должны ограничивать использование собственных элементов производных классов;
- производные классы не должны приводить к ограничению свойств базового класса;
- классы, отличающиеся только специализацией, составляют один уровень иерархии.

В результате наследование позволяет повторно использовать ранее написанный программный код, тем самым сократить время на разработку программ. Кроме того, повторное использование кода также позволяет убедиться в его надежности. Так, чем в большем числе ситуаций используется программный код, тем больше возможностей обнаружить и устранить в нем ошибки.

4.2 О спецификаторах доступа

Выполним инкапсуляцию полей разработанных классов `Human`, `Student` и `Lecturer`, а доступ к ним организуем с помощью сеттеров и геттеров. Как было показано в предыдущем разделе, спецификатор `private` запрещает доступ к полям и методам класса за его пределами, из-за чего изменение поля `name` методами производных классов становится невозможным. Поэтому в классе `Human` наследуемое поле `name` объявлено как защищенное с помощью указания спецификатора доступа `protected`. Этот спецификатор *разрешает доступ к элементам базового класса как из самого класса, так и из производных от него классов*. Для демонстрации работы этого спецификатора в основной функции программы выполним создание объектов реализованных классов и вызов их методов.

```
1  #include <iostream>
2  using namespace std;
3  class Human {
4     public:
5        void getName() {
6           cout << "I am Human " << this->name << endl;
7        }
8        void setName(string inName) {
9           this->name = inName;
10       }
11    protected:
12       string name;
13 };
14 class Student : public Human {
15    public:
16           void getName() {
                cout << "I am Student " << this->name << ",
17 group=" << this->group << endl;
         }
```

```
18        void setName(string inName, string inGroup){
19           this->name = inName;
20           this->group = inGroup;
21        }
22     private:
23        string group;
24  };
25  class Lecturer : public Human {
26     public:
27        void getName() {
28                cout<< "I am Lecturer " << this->name << ",
29  subject=" << this->subject << endl;
          }
30        void setName(string inName, string inSubj) {
31          this->name = inName;
32          this->subject = inSubj;
33        }
34     private:
35        string subject;
36  };
37  int main(){
38     Human h;
39     h.setName("hum1");
40     h.getName();
41     Student student;
42     student.setName("stud1", "122-1");
43     student.getName();
44     Lecturer lecturer;
45     lecturer.setName("lect1", "programming");
46     lecturer.getName();
47     return 0;
48  }
49
```

Результат работы программы приведен на рисунке 4.3.

```
I am Human hum1
I am Student stud1, group=122-1
I am Lecturer lect1, subject=programming
```

Рис. 4.3 – Результат выполнения программы,
демонстрирующей особенности спецификатора `protected`

Из результатов видно, что в этом примере геттеры и сеттеры, являющиеся методами производных классов, могут изменять поле базового класса `name`. Однако, согласно принципу инкапсуляции, прямое обращение к элементам класса со спецификатором `protected` запрещено (рис. 4.4).

а) *б*)

Рис. 4.4 – MWS: прямое обращение (*а*) и ошибка доступа (*б*)
к элементу `name` со спецификатором `protected` класса `Human`

В результате спецификатор `protected` подобен `private` в том смысле, что доступ к членам класса, объявленным со спецификатором `protected`, можно получить извне только через интерфейсные (открытые) методы класса. При этом различие между спецификаторами `private` и `protected` проявляется только внутри классов, наследованных от базового класса. Так, члены производного класса могут иметь доступ к защищенным (`protected`) членам базового класса, но не имеют доступа к его закрытым (`private`) членам. Другими словами, защищенные элементы класса ведут себя как открытые для объектов базового и производных классов, а иначе – как закрытые, доступ к которым извне класса запрещен.

4.3 Отношения между производными и базовыми классами

Спецификаторы доступа `public`, `private` (по умолчанию) и `protected` могут применяться к базовому классу при его наследовании производными классами, т. е. действие того или иного спецификатора влияет на уровень доступности элементов базового класса.

Наследование называется открытым, если используется спецификатор `public`. Тогда справедливо, что объект производного класса является представителем объекта базового класса, но не наоборот, т. е. он полностью копирует свойства (поля) и поведение (методы) своего родителя. При этом все открытые (`public`) и защищенные (`protected`) члены базового класса становятся, соответственно, открытыми (`public` → `public`) и защищенными (`protected` → `protected`) членами производного класса. В то же время доступ из производного класса к закрытым (`private`) членам базового класса запрещен.

```
1  class A{
2     public:
3        int x;
4     protected:
5        int y;
6     private:
7        int z;
8  };
9  class B : public A{ // public->public, protected->
   protected
10     void printInfo{
11       cout << x; // нет ошибки
12       cout << y; // нет ошибки
13       cout << z; // ошибка, нет доступа к private
   элементу z базового класса A
14    };
15 };
```

Если при наследовании указан спецификатор доступа `protected`, то открытые (`public` → `protected`) и защищенные (`protected` → `protected`) члены базового класса становятся защищенными в производном классе. Такой вид «защищенного» наследования предполагает, что внутри всех производных классов в иерархии будут использоваться только члены, передающиеся механизмом наследования, и они будут защищены от влияния на себя за пределами действия своих классов.

```
1  class A{
2    public:
3       int x;
4    protected:
```

```
5       int y;
6    private:
7       int z;
8 };
9 class B : protected A{ // public-> protected, protected->
  protected
10    void printB{
11       cout << x; // x - protected элемент класса B
12       cout << y; // y - protected-элемент класса B
13       cout << z; // ошибка, нет доступа к private
  элементу базового класса A
14    };
15 };
16 class C : public B{ // public->public, protected->
  protected
17    void printC{
18       cout << x; // x - protected элемент класса C
19       cout << y; // z - protected элемент класса C
20       cout << z; // ошибка, нет доступа к private
  элементу базового класса A
21    };
22     };
```

Спецификатор доступа `private` реализует «закрытое» наследование. Это означает, что объекты производного класса не являются разновидностью объектов базового класса. При этом все открытые (`public` → `private`) и защищенные (`protected` → `private`) члены базового класса становятся закрытыми в производном классе. Поэтому дальнейшее наследование с любым спецификатором доступа бессмысленно (показано ниже на примере наследования класса `C` от класса `B`).

```
1  class A{
2     public:
3        int x;
4     protected:
5        int y;
6     private:
7        int z;
8  };
```

```
9  class B : private A{ // public->private, protected-> private
10    void printB{
11       cout << x; // x – private элемент класса B
12       cout << y; // y – private элемент класса B
13       cout << z; // ошибка, нет доступа к private элементу z
   базового класса A
14    };
15 };
16 class C : public B{ // «бессмысленное» наследование
17    void printC{
18       cout << x; // ошибка, нет доступа к private элементу
   базового класса B
19       cout << y; // ошибка, нет доступа к private элементу
   базового класса B
20       cout << z; // ошибка, нет доступа к private элементу
   базового класса B
21     };
22 };
```

4.4 Дружественные функции, методы и классы

Иногда требуется, чтобы функции, не входящие в состав классов, имели доступ к их закрытым элементам. Такие функции получили название *дружественных* (`friend`), их применение поверхностно рассмотрено в п. 3.3.8. При этом объявление «дружественности» не распространяется на переменные и поля.

Объявление дружественной функции начинается с ключевого слова `friend` и *должно находиться только в определении класса*, к элементам которого она будет иметь доступ.

```
1  #include <iostream>
2  using namespace std;
3  class A{
4      public:
5          friend void func(A&);
6      private:
7          int el1, el2;
8  };
9  void func(A& m_a) {
10     m_a.el1 = 2;
11         m_a.el2 = 4;
12     cout << m_a.el1 << " " << m_a.el2 << endl;
```

```
13 };
14 int main(){
15     A a;
16     func(a);
17     return 0;
18 }
```

Здесь в классе `A` объявлена дружественная функция `func()`, в качестве параметра которой указана ссылка на объект этого класса. Это позволяет этой функции, определенной вне класса, оперировать элементами класса, в данном случае его закрытыми полями. Несмотря на то, что дружественная функция объявляется внутри класса, тем не менее методом этого класса она не является. Поэтому не имеет значения, в закрытой (`private`) или открытой (`public`) части класса она объявлена.

Следует помнить, что дружественной функции не передается указатель `this`. При этом одна функция может быть дружественной для нескольких классов.

Важно, что метод одного класса может быть дружественным для другого класса. При этом надо сначала объявить класс (в следующем примере это `class A`), которому принадлежит дружественный метод, чтобы класс, для которого он становится дружественным (`class B`), «знал» о его существовании на этапе компиляции. В остальном реализация программного кода, в целом, аналогична случаю использования дружественной фикции, рассмотренному выше.

```
1  #include <iostream>
2  using namespace std;
3  class A;
4  class B {
5     public:
6        void func(A &m_a);
7  };
8  class A{
9     public:
10       friend void B :: func(A &);
11       int el1, el2;
12 };
13 void B :: func(A &m_a) {
14    m_a.el1 = 2;
15    m_a.el2 = 4;
```

```
16    cout << m_a.el1 << " " << m_a.el2 << endl;
17 };
18 int main(){
19    A a;
20    B b;
21    b.func(a);
22    return 0;
23 }
```

Если все методы одного класса являются дружественными для другого класса, то можно объявить дружественный класс.

```
1  #include <iostream>
2  using namespace std;
3  class B; // объявление класса B
4  class A {
5     public:
6        A(int i) { u = i;} // конструктор
7        friend B; // класс B становится дружественным A
8        void getData(){
9           cout << "Base class contain: u= " << u << endl;
10       }
11    private:
12       int u;
13 };
14 class B {
15    public:
16       void changeU(A &, int);
17       void getChangedU(A);
18 };
19 void B :: changeU(A &obj, int newVal){ // метод класса B
   (передача объекта класса и изменение u)
20    cout << "Set new value from friendly class" << endl;
21    obj.u = newVal;
22 }
23 void B :: getChangedU(A obj){ // метод класса B
24    cout << "Get new value from friendly class = "
   << obj.u << endl;
25 }
26 int main(){
```

```
27      A base(3);
28      base.getData();
29      B fr;
30      fr.changeU(base, 5);
31      fr.getChangedU(base);
32      return 0;
33        }
```

Здесь все методы класса `B` имеют доступ к закрытому полю `u` класса `A`.

4.5 Правила наследования методов

В рассмотренных примерах производилось наследование только полей базового класса, при этом аналогичным образом выполняется и наследование методов. Допустим, базовый класс `Human` содержит одно поле и ориентированные на него геттер и сеттер. Унаследуем класс `Student` от класса `Human`, определив дополнительно в нем новое поле, а также соответствующие ему сеттер и геттер.

```
1   class Human{
2      public:
3         void setName(const string &m_name) {
4            name = m_name;
5         };
6         string getName() { return name; }
7      private:
8         string name;
9   };
10  class Student : public Human{// «открытое» наследование
    (public->public, protected->protected)
11     public:
12        void setGroup (const string &m_group) {
13           group = m_group;
14        };
15        string getGroup() { return group; }
16     private:
17        string group;
18  };
19  int main(){
20     Student stud;
21     stud.setName("Alexey");
```

```
22    stud.setGroup("122-1");
23    cout << "Student name = " << stud.getName() << endl
<< "group = " << stud.getGroup() << endl;
24    return 0;
25      }
```

Здесь производный класс `Lecturer` унаследовал от `Human` открытые (`public`) методы `setName()` и `getName()`, с помощью которых можно обращаться к закрытому полю `name` класса `Human` (рис. 4.5).

```
Student name = Alexey
group = 122-1
```

Рис. 4.5 – Результат выполнения программы, демонстрирующей правила наследования методов

Очень важно, что базовый и производный классы могут иметь свои конструкторы и деструкторы. При этом они *не наследуются*, поэтому при создании объекта производного класса наследуемые им данные должны инициироваться конструктором базового класса.

Рассмотрим очередность вызова конструкторов и деструкторов базового и производного классов с помощью следующего примера.

```
1  #include <iostream>
2  using namespace std;
3  class BaseClass{
4     public:
5        BaseClass(){ cout<< "Constructor of base class"
<<endl; } // конструктор базового класса
6        ~BaseClass() { cout<< "Destructor of base class"
<< endl; } // деструктор базового класса
7  };
8  class DerivedClass: public BaseClass{
9     public:
10       DerivedClass() { cout << "Constructor of derived
class" << endl; } // конструктор производного класса
11       ~DerivedClass() { cout << "Destructor of derived
class" << endl; } // деструктор производного класса
12 };
13 int main(){
```

```
14      DerivedClass obj;
15      return 0;
16  }
```

Здесь конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. А вот деструкторы вызываются в обратном порядке (рис. 4.6).

```
Constructor of base class
Constructor of derived class
Destructor of derived class
Destructor of base class
```

Рис. 4.6 – Результат выполнения программы, демонстрирующей очередность работы конструкторов и деструкторов классов

Поскольку базовый класс «не знает» о существовании производного класса, любая инициализация его полей может быть выполнена независимо от производного класса. Вызов деструктора базового класса раньше деструктора производного класса может привести к преждевременному разрушению объекта производного класса, что часто недопустимо.

Если требуется передать аргумент конструктора производного класса в конструктор базового класса, то в производном используется расширенная запись конструктора.

```
1   #include <iostream>
2   using namespace std;
3   class BaseClass{
4      public:
5         BaseClass(int ii) {i = ii;}
6         ~BaseClass(){cout<<"BaseClass destructor "<<endl;}
7      private:
8         int i;
9   };
10  class DerivedClass: public BaseClass{
11     public:
12        DerivedClass(int nn, int m): BaseClass(m){n = nn;}
    // расширенная запись конструктора
13        ~DerivedClass(){cout<<"Derived destructor"<<endl;}
14     private:
15           int n;
```

```
16 };
17 int main(){
18    DerivedClass obj(10,11);
19    return 0;
20      }
```

Важно, что в отличие от конструкторов при реализации деструктора производного класса в нем не требуется явно вызывать деструктор базового класса, поскольку это будет сделано автоматически (рис. 4.7).

```
Derived destructor
BaseClass destructor
```

Рис. 4.7 – Результат выполнения программы, демонстрирующей очередность работы деструкторов производного и базового классов

Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов. Так, последовательно вызываются деструкторы производных классов, а затем деструктор базового класса.

Еще один важный момент состоит в том, что оператор присваивания `operator=` базового класса недоступен в производных классах. Причина заключается в том, что компилятор неявно создает оператор копирующего присваивания, который скрывает все унаследованные перегрузки `operator=`. Поэтому в производном классе надо объявлять о его использовании с помощью ключевого слова `using` и оператора разрешения области видимости «`::`».

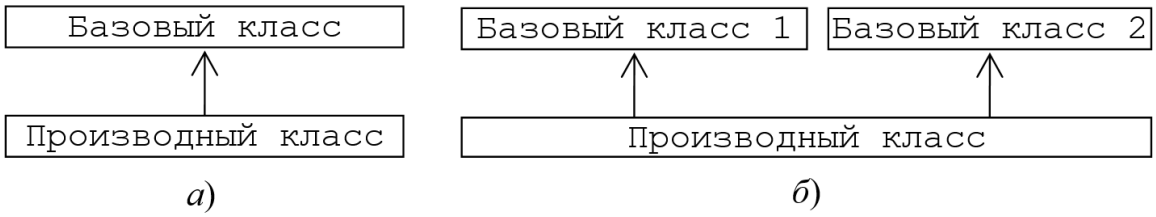
```
1  #include <iostream>
2  using namespace std;
3  class A {
4     public:
5          void operator= (int value){ i = value * value; }
6          void getVal(){ cout << i << endl;}
7      private:
8          int i;
9  };
10 class B : public A {
11     public:
12         using A :: operator=; // объявление
13 };
```

```
14 int main(){
15     A a;
16     a = 5; // 25
17     a.getVal();
18     B b;
19     b = 2;
20     b.getVal(); // 4
21     return 0;
22 }
```

Производный класс наследует методы базового класса, но не наследует конструкторы и деструктор, а также реализацию операции присваивания. Кроме того, дружественные функции и дружественные отношения между классами (дружественные методы) также не наследуются.

4.6 Множественное наследование

В рассмотренных ранее примерах использовалось простое наследование, при котором производный класс имеет только один базовый класс. В языке C++ также допускается использование множественного наследования, при котором у одного производного класса может быть несколько базовых классов (рис. 4.8).



а) *б*)

Рис. 4.8 – Простое (*а*) и множественное (*б*) наследование

Рассмотрим пример, когда класс `Laptop` наследуется одновременно от двух базовых классов – `Monitor` и `Computer`.

```
1 #include <iostream>
2 using namespace std;
3 class Computer {
4    public:
5       void turn_on(){ cout << "Welcome to OS" << endl;}
```

```
};
class Monitor {
   public:
      void show_image() {
         cout << "Imagine image here" << endl;
      }
};
class Laptop: public Computer, public Monitor {};
int main() {
   Laptop lap;
   lap.turn_on(); // вызов метода класса Computer
   lap.show_image(); // вызов метода класса Monitor
   return 0;
}
```

Здесь объект производного класса оперирует методами двух базовых классов (рис. 4.9). При этом следует помнить, что если наследуются несколько базовых классов, то их конструкторы явно вызываются конструктором производного класса или компилятор инициирует вызов конструкторов по умолчанию в порядке описания базовых классов.

```
Welcome to OS
Imagine image here
```

Рис. 4.9 – Результат выполнения программы, демонстрирующей наследование производного класса от двух базовых классов

Множественное наследование требует тщательного проектирования, так как его использование может привести к непредвиденным последствиям. Большинство таких последствий вызваны неоднозначностью в наследовании. (Поэтому многие языки программирования не поддерживают множественное наследование.)

```
#include <iostream>
using namespace std;
class Computer {
   public:
      void turn_on(){cout<< "Computer is on." <<endl;}
};
class Monitor {
```

```
8     public:
9        void turn_on(){cout<< "Monitor is on." <<endl;}
10 };
11 class Laptop: public Computer, public Monitor {};
12 int main() {
13    Laptop lap;
14    lap.turn_on(); // ошибка компиляции, неоднозначность
15    return 0;
16 }
```

Здесь класс `Laptop` наследует метод `turn_on()` от обоих базовых классов, и неясно, какой именно метод должен быть вызван. Эта проблема неоднозначности может быть решена за счет указания имени требуемого базового класса с помощью оператора разрешения области видимости «`::`».

```
1     lap.Monitor::turn_on();
2     lap.Computer::turn_on();
```

В этом примере конфликта имен удалось избежать. Но что делать, если разрабатываемый класс будет иметь, например, более двух базовых классов, которые, в свою очередь, тоже будут иметь свои базовые классы? Очевидный ответ – явно указывать версии методов, которые следует вызывать. Однако при этом возможность возникновения конфликтов имен увеличивается с каждым добавленным базовым классом.

Еще одной проблемой является «проблема ромба» (иногда ее называют «смертельный алмаз смерти»). Рассмотрим ее на иерархии классов, в которой классы `B` и `C` наследуют `A`, а класс `D` наследует `B` и `C` (рис. 4.10). Пусть каждый из классов `A`, `B` и `C` определяет свой метод `print()`. Тогда в случае вызова данного метода объектом класса `D` будет неясно, какая именно реализация метода `print()` будет вызвана: класса `A`, `B` или `C`.

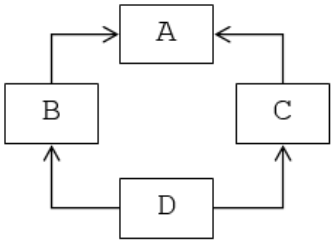


Рис. 4.10 – «Проблема ромба» при множественном наследовании

Разрешить «проблему ромба» можно за счет:

- переопределения метода в последнем производном классе;
- вызова требуемого метода конкретного базового класса;
- обращения к объекту производного класса как к объекту базового.

```
#include <iostream>
using namespace std;
class A {
   public:
      void print() {
         cout << "print function" << endl;
      }
};
class B: public A {};
class C: public A {};
class D: public B, public C{
   public:
      void print(){ // переопределение метода
         cout << "redefined function" << endl;
      }
};
int main(){
   D object;
   object.print(); // вызов переопределенного метода
   object.B::print(); // вызов метода базового класса
   static_cast<B&>( object ).print(); // обращение к
объекту производного класса как к объекту базового
   return 0;
}
```

Поскольку в C++ при инициализации объекта производного класса вызываются конструкторы всех родительских классов, то возникает и другая проблема, связанная с тем, что конструктор базового класса может быть вызван многократно. Для борьбы с этим используется виртуальное (ключевое слово `virtual`) наследование, которое предотвращает появление множественных объектов базового класса в иерархии наследования.

```
#include <iostream>
using namespace std;
class A {
```

```
4     public:
5        A(){ cout << "A constructor " << endl; }
6           void print() {
7              cout << "print function" << endl;
8           }
9  };
10 class B: virtual public A { // виртуальное наследование
11    public:
12       B(){ cout << "B constructor " << endl; }
13 };
14 class C: virtual public A { // виртуальное наследование
15    public:
16        C(){ cout << "C constructor " << endl; }
17 };
18 class D: public B, public C {}; // {} - обязательно!
19 int main(){
20    D object;
21    object.print();
22    return 0;
23 }
```

Здесь конструктор базового класса `A` будет вызван только один раз, а обращение к методу `print()`, без его переопределения в производном классе, не будет вызывать ошибку компиляции (рис. 4.11).

```
A constructor
B constructor
C constructor
print function
```

Рис. 4.11 – Результат выполнения программы, демонстрирующей виртуальное наследование

4.7 Сокрытие методов базового класса

В языке C++ невозможно удалить или ограничить функционал базового класса, кроме как непосредственно изменить его программный код. Однако в производном классе можно скрыть (запретить использовать) функционал, имеющийся в базовом классе. Это делается за счет изменения спецификаторов доступа и использования ключевого слова `using`. В результате любой член базового класса можно сделать закрытым в производном классе.

```
#include <iostream>
using namespace std;
class BaseClass{
    public:
      BaseClass(int val) : value(val){}
      int getValue() { return value; } // геттер
      int value; // value - public
};
class DerivedClass : public BaseClass{
   public:
      DerivedClass(int val) : BaseClass(val){}
   private:
      using BaseClass :: value;// теперь value - private
};
int main(){
   DerivedClass child(9);
   cout << child.getValue() << endl;//нет ошибки, геттер
   cout << child.value;// ошибка, т. к. value - private
   return 0;
}
```

Такой запрет использования позволяет инкапсулировать данные базового класса в производный. При этом нельзя изменить спецификатор доступа элемента базового класса с `private` на `protected` или `public`, т. к. производный класс не имеет доступа к закрытым элементам базового класса.

Можно также закрыть (запретить использовать) методы базового класса в производном, используя ключевое слово `delete`. (Аналогичный запрет рассматривался в п. 3.3.2 на примере конструктора по умолчанию класса `Complex`.)

```
#include <iostream>
using namespace std;
class BaseClass{
   public:
      BaseClass(int val) : value(val){}
      int getValue() { return value; } // геттер
   private:
      int value; // value - private
};
class DerivedClass : public BaseClass{
```

```
11    public:
12       DerivedClass(int val) : BaseClass(val){}
13       int getValue() = delete; // запрет геттера
14 };
15 int main(){
16    DerivedClass child(9);
17    cout<< child.getValue() << endl;//ошибка, нет геттера
18    BaseClass chield_(9);
19    cout<< chield_.getValue() << endl; //нет ошибки
20    return 0;
21 }
```

Здесь компилятор будет сообщать об ошибке вызова метода `getValue()` через объект класса `DerivedClass` (строка 17). Однако через объект базового класса все будет работать (строка 19), так как только в производном классе запрещено использовать метод `getValue()`.

4.8 Виртуальные функции и методы, абстрактные классы и интерфейсы

Кратко обсудим еще один инструментарий языка C++, который наравне с шаблонами классов позволяет обобщать реализацию схожих классов и тем самым поддерживает обобщенное программирование.

Виртуальный метод (функция) *– это метод (функция), который определяется в базовом классе с последующим переопределением его в производных классах.*

Синтаксис объявления виртуального метода (функции) имеет следующий вид:

```
1  virtual void print(){ // виртуальный метод (функция)
2     // требуемый код
3  };
```

Ключевое слово `virtual` гарантирует, что если метод (функция) не переопределен в производном классе, то компилятор выдаст ошибку.

```
1  #include <iostream>
2  using namespace std;
3  class BaseClass{
```

```
4     virtual void print(){ // виртуальный метод (функция)
5        cout << "Print from BaseClass" << endl;
6     };
7  };
8  class DerivedClass : public BaseClass{};
9  int main(){
10    DerivedClass child;
11    child.print(); // ошибка, нет переопределения
   виртуального метода (функции) в производном классе
12    return 0;
13 }
```

Виртуальный метод (функция), объявленный с использованием конструкции «= 0», называется чисто виртуальным.

```
1  class BaseClass{
2     public:
3        virtual void print()=0;// чисто виртуальный метод
4     // остальной код класса
5  };
```

*Класс считается **абстрактным**, если он содержит как минимум один чисто виртуальный метод. Если при наследовании от абстрактного класса повторяется объявление его чисто виртуальных методов, то производный класс также становится абстрактным.*

Стандарт языка C++ запрещает создавать объекты абстрактных классов. Такие классы используются, только когда необходимо создать семейство классов и вынести их общее поведение в отдельный базовый класс. При этом в каждом производном классе требуется переопределить его уникальные методы.

*Классы, не имеющие полей и состоящие из чисто виртуальных методов, называются **интерфейсными**.*

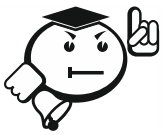
В интерфейсном классе объявляются лишь сигнатуры методов. Это сигнализирует о том, что производный класс должен уметь делать, а как именно, он «решает» сам.

Рассмотрим работу интерфейсов на примере класса двумерных геометрических фигур `Figure`.

```
class Figure{
  public:
   virtual float getPerimeter() = 0;
   virtual float getSquare() = 0;
};
```

Данный интерфейсный класс содержит два чисто виртуальных метода `getPerimeter()` и `getSquare()`, используемых для вычисления периметра и площади фигуры и не имеющих никаких реализаций. Поскольку для каждого вида фигур данные величины считаются по-разному, то создадим производные классы `Rectangle` (прямоугольник) и `Circle` (круг) со спецификатором наследования `public`.

```
class Circle : public Figure {
   float radius;
  public:
   Circle(float r){ this->radius = r; };
   float getPerimeter() override;
   float getSquare() override;
};
float Circle::getPerimeter(){ return 2*3.14*radii; };
float Circle::getSquare(){ return 3.14*radii*radii; };
class Rectangle : public Figure {
   float a, b;
  public:
   Rectangle(float ma, float mb){
      this->a = ma;
      this->b = mb;
   }
   float getPerimeter() override;
   float getSquare() override;
};
float Rectangle::getPerimeter() { return 2*(a+b); };
float Rectangle::getSquare() { return a*b; };
```



В языке C++ метод производного класса будет переопределять метод базового класса, только если его сигнатура полностью совпадает с сигнатурой базового класса. В случае наиболее распространенной ошибки, т. е. когда одна из сигнатур будет содержать обычный указатель, а другая – константный, переопределение методов не происходит. Чтобы избежать этого, рекомендуется для каждого метода, который переопределяет базовый метод, использовать ключевое слово (модификатор) `override`. Оно позволяет компилятору следить за тем, чтобы указанный метод переопределял метод базового класса.

Хотя в C++ нельзя создавать объекты абстрактных классов, можно создавать указатели на эти классы. При их инициализации можно использовать ссылки на объекты производного класса.

```
1  int main(){
2     Figure *obj;
3     Rectangle rect(2.5, 3.5);
4     obj = &rect; // или obj = new Rectangle(2.5, 3.5);
5     Circle circ(3.2);
6     obj = &circ; // или obj = new Cirle(3.2);
7     return 0;
8  }
```

С помощью таких указателей можно обращаться к реализациям методов, переопределенных на более низких уровнях иерархии наследования. Выбор той или иной реализации определяется типом объекта, который был использован при инициализации указателя.

```
1  int main(){
2     Figure *obj;
3     Circle circ(3.2);
4     Rectangle rect(2.5, 3.5);
5     obj = &rect;
6     cout << obj->getPerimeter()<<" " << obj->getSquare();
7     obj = &circ;
8     cout << obj->getPerimeter()<<" " << obj->getSquare();
9     return 0;
10 }
```

Здесь указатель на абстрактный класс `obj` инициализируется ссылкой на объект `rect` производного класса `Rectangle` с последующим вызовом двух его методов. Далее указатель повторно инициализируется ссылкой на объект класса `Circle` и аналогичным образом вызывает уже его методы. Выбор той или иной реализации виртуальных методов определяется классом объекта, на который ссылается указатель `obj` (рис. 4.12).

```
12 8.75
20.096 32.1536
```

Рис. 4.12 – Результат выполнения программы,
демонстрирующей работу с указателем на абстрактный класс

Также указатели на абстрактные классы можно использовать в качестве аргументов функций. Допустим, требуется написать функцию для проверки получаемых с помощью методов `getPerimeter()` и `getSquare()` результатов. Для этого внутри функции работа с объектом класса производится через указатель на абстрактный базовый класс, а при вызове функции в качестве аргумента передается ссылка на объект производного класса (рис. 4.13).

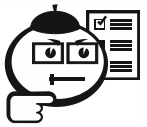
```
1  void checkFigure(Figure *ptr) {
2     float tmpS = ptr->getSquare();
3     float tmpP = ptr->getPerimeter();
4     if ((tmpP < 0) || (tmpS < 0))
5        cout << "No valid figure" << endl;
6     else
7        cout << "Figure is valid" << endl;
8  };
9  int main(){
10    Circle circ(4.5);
11    Rectangle rect(-2.5, 3.5);
12    checkFigure(&circ);
13    checkFigure(&rect);
14    return 0;
15 }
```

```
Figure is valid
No valid figure
```

Рис. 4.13 – Результат выполнения программы, демонстрирующей работу с функцией, принимающей в качестве аргумента указатель на абстрактный класс

Контрольные вопросы по разделу 4

1. Что такое наследование классов?
2. В чем различие между спецификаторами `protected` и `private`?
3. Какие могут быть отношения между производными и базовыми классами?
4. Чем отличается спецификатор наследования от спецификатора доступа?
5. Что такое дружественная функция и для чего она используется?
6. Какой класс называется дружественным?
7. Для чего применяется множественное наследование?
8. Какими способами можно разрешить неопределенность при множественном наследовании?
9. Как осуществляется сокрытие методов базового класса внутри производного?
10. Для чего используется ключевое слово `using` при наследовании?
11. Для чего используется ключевое слово `delete` при наследовании?
12. Что такое виртуальный метод (функция)?
13. Какой класс называется абстрактным?
14. Что такое чисто виртуальный метод (функция)?
15. Для чего применяются интерфейсные классы?



Задания для самостоятельного выполнения

1. Используя класс `Complex` из п. 3.3.10, с помощью наследования создать производный класс `ComplexResistance` (комплексное сопротивление), содержащий следующие поля: действительная и мнимая части сопротивления, угловая частота. В производном классе реализовать

дополнительный метод для вычисления модуля и аргумента комплексного сопротивления.

2. Определить иерархию классов для геометрических фигур: закрашенный треугольник, треугольник, четырёхугольник, ромб. Найти общие черты и выразить их в отдельных классах. Определить в классах метод `Draw` (рисовать), выводящий на экран название соответствующей фигуры.
3. Создать класс для определения двумерного массива размерностью 5×5. Реализовать конструктор класса для заполнения массива случайными числами от –20 до 20. Создать дружественную классу функцию, которая будет выполнять поиск максимального и минимального значений в массиве.
4. Создать абстрактный базовый класс `Equation` (уравнение) с виртуальным методом (функцией) нахождения значения функции (y) по заданному значению аргумента (x). Создать производные классы: `LinearEquation` (линейное уравнение), `QuadraticEquation` (квадратное уравнение) и `CubicParabolaEquation` (кубическая парабола).
5. Создать абстрактный базовый класс `Figure` (фигура) с виртуальной функцией вычисления площади. Создать производные классы `RectangleFigure` (прямоугольник), `CircleFigure` (круг) и `RightTriangleFigure` (прямоугольный треугольник) со своими реализациями метода вычисления площади фигуры.

5 Объектно-ориентированное программирование: полиморфизм

Был этот мир глубокой тьмой окутан.
Да будет свет! И вот явился Ньютон.
Но сатана недолго ждал реванша.
Пришел Эйнштейн – и стало все, как раньше.

А. Поуп, Дж. Сквайр
Пер. С. Я. Маршака

5.1 Вводные замечания

В предыдущих разделах изложены основополагающие принципы объектно-ориентированного программирования – инкапсуляция и наследование, после ознакомления с которыми логично перейти к последнему принципу ООП – полиморфизму.

*Под **полиморфизмом** понимают свойство программного кода менять свое поведение в зависимости от ситуаций, возникающих по ходу выполнения программы.*

Переходя на уровень реализации, полиморфизм подразумевает использование одного имени идентификатора (функции, оператора) для решения разных задач, т. е. выбор конкретного действия определяется типом данных. В более общем смысле идея полиморфизма состоит в использовании общего интерфейса для множества методов. Тогда полиморфизм является одним из основных приемов разработки гибкой архитектуры приложений, обеспечивающим повторное использование и расширяемость базы программного кода.

В языке C++ возможна реализация двух видов полиморфизма – статического и динамического. Статический полиморфизм достигается за счет использования перегруженных функций и операторов на этапе раннего связывания, а также шаблонов. Раннее связывание означает то, что на этапе компиляции имя идентификатора напрямую связано с машинным адресом, что накладывает определенные ограничения на работу программного кода. Динамический полиморфизм реализуется с помощью наследования и виртуальных функций (на этапе

позднего связывания), упоминавшихся в предыдущем разделе. Позднее связывание означает, что программный код вызова функций формируется по ходу выполнения программы, т. е. код не содержит указаний на то, какая конкретно функция должна быть вызвана.

Не все задачи требуют использования позднего связывания. Так, выбор конкретного типа реализации полиморфизма напрямую зависит от специфики и сложности решаемой задачи.

Далее приведены примеры реализации статического и динамического полиморфизма в языке C++.

5.2 Перегрузка методов класса

В подразд. 2.2 были показаны возможности языка C++ по использованию функций с одинаковыми именами, но разными типами данных аргументов и их числом. Объявление методов класса и функций в целом с одним именем называется перегрузкой. Рассмотрим подобную перегрузку в контексте ООП на примере методов класса для сложения двух и трех чисел (аргументы методов).

```
1  #include <iostream>
2  using namespace std;
3  class Calculator{
4     public:
5        int sum(int a, int b){
6           return a + b;
7        }
8        int sum(double a, double b, double c){
9           return a + b + c;
10       }
11 };
12 int main(){
13    Calculator C;
14    cout << C.sum(1,2) << endl << C.sum(1.0, 2.0, 3.0);
15    return 0;
16 }
```

Здесь первый метод принимает на два аргумента типа `int` и возвращает их сумму, а второй проделывает то же действие, но с тремя входными аргументами

типа `double`. В функции `main()` выполняется вызов этих методов через объект класса и вывод результата их работы на печать.

Методы (функции), отличающиеся только типом возвращаемых значений, не являются перегруженными. В этом случае компилятор выдаст ошибку.

Частным случаем перегрузки методов является перегрузка конструкторов в классе, упомянутая в п. 3.3.2. Рассмотрим пример класса `Sphere`, реализующего сферу. В классе перегружен конструктор, который может быть вызван без параметров (по умолчанию) и с одним параметром.

```
1  #include <iostream>
2  class Sphere{
3     public:
4        Sphere(){
5           R = 10;
6        }
7        Sphere(int r){
8           R = r;
9        }
10    private:
11       int R;
12 };
13 int main(){
14    Sphere S1;
15    Sphere S2(20);
16    return 0;
17 }
```

Использование перегрузки конструкторов позволяет повысить гибкость класса при его создании. Так, разработчик может выбрать оптимальный способ создания объекта в зависимости от решаемой задачи.

5.3 Перегрузка операторов

Перегрузка операторов, поверхностно рассмотренная в п. 3.3.8, также является частным случаем проявления полиморфизма, поскольку позволяет использовать одинаковый интерфейс для разных методов одного и того же оператора. В

случае когда оператор становится перегруженным, он приобретает дополнительные свойства в зависимости от контекста его применения, тем не менее, сохраняется возможность его использования в изначальном смысле. Например, оператор сложения `operator+` может быть перегружен по отношению к классам объекта таким образом, чтобы он добавлял содержимое одного объекта к другому. Но при этом он сохраняет свое «поведение» для остальных типов данных.

Перегружать можно только стандартные операторы, определенные в языке C++. При этом создавать новые операторы запрещено.

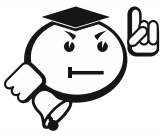
Ранее было упомянуто о существовании нескольких способов перегрузки операторов, к которым относят перегрузку через методы класса, обычные (внешние) и дружественные функции. Напомним, что подробное рассмотрение дружественных функций и классов приведено в подразд. 4.4.

Далее на примере пользовательского класса `Complex` из п. 3.3.10 рассмотрим перегрузки часто используемых операторов, а также приведем общие рекомендации по перегрузке операторов.

5.3.1 Бинарные операторы

Арифметические операторы

Арифметические операторы (+, –, *, /) относятся к наиболее часто используемым операторам языка C++. Эти операторы являются бинарными, т. е. работающими с двумя операндами. Важно отметить, что при перегрузке операторов нельзя изменять их приоритет.



При перегрузке бинарных операторов (в том числе и арифметических), работающих с операндами разного типа, надо предусматривать перегрузку для обеих ситуаций порядка указания операторов. Например, для оператора сложения это `operator+(Point& p, int v)` и `operator+(int v, Point& p)`.

Рассмотрим пример перегрузки операторов сложения для аргументов разных типов данных через *дружественные* функции. Для краткости изложения здесь и далее код объявления класса опущен.

```
1  Complex Complex::operator+(Complex& c, double v){
2     return Complex(c.real_d+v, c.imag_d);
3  }
4  Complex Complex::operator+(double v, Complex& c){
5     return c + v; // перегрузка через operator+(Complex&,
   double) из строки 2
6  }
```

Обратим внимание на то, что `operator+(double v, Complex& c)` перегружен с помощью вызова `operator+(Complex& c, double v)`. Так, на практике рекомендуется, когда это реализуемо, определять операторы вызовом другого, уже перегруженного оператора.

Оператор `operator+(double v, Complex& c)` не может быть перегружен через метод класса, т. к. в этом случае `double v` является левым операндом, на который указатель `this` указывать не может.

Операторы сравнения

Часть операторов перегружаются парами. Так, например, если необходимо перегрузить `operator>`, то надо перегружать и `operator<`. При этом следует помнить, что результатом выполнения оператора сравнения является переменная логического типа (`bool`).

Рассмотрим перегрузку операторов сравнения через *дружественные* функции на примере `operator==` и `operator!=`.

```
1  bool Complex::operator==(Complex& c1, Complex& c2){
2     return c1.real_d == c2.real_d && c1.imag_d ==
   c2.imag_d;
3  }
4  bool Complex::operator!=(Complex& c1, Complex& c2){
5     return !(c1 == c2);
6  }
```

Здесь, как и ранее, `operator!=` перегружен через уже перегруженный оператор `operator==`.

Операторы ввода и вывода

Если класс содержит большое число полей, то «ручной» вывод на печать их значений с помощью геттеров может неоправданно увеличить объем программного кода, а также значительно усложнить его читаемость. Поэтому рекомендуется либо реализовать отдельный метод для вывода на печать этих значений, либо воспользоваться перегрузкой `operator<<`. Аналогичным образом реализуется и перегрузка оператора ввода `operator >>`. Эти операторы также относятся к бинарным, левые операнды которых имеют тип `std::ostream&` и `std::istream&` соответственно. Тогда их перегрузка на примере *дружественных* функций будет иметь следующий вид:

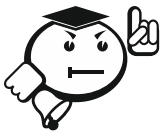
```
1  std::ostream&   Complex::operator<<(std::ostream   &out,
   Complex& c){
2     out << c.real_d << ", " << c.imag_d;
3     return out;
4  }
5  std::istream&    Complex::operator>>(std::istream    &in,
   Complex& c){
6     in >> c.real_d >> c.imag_d;
7  }
```

5.3.2 Унарные операторы

Арифметические операторы

Сначала рассмотрим перегрузку унарного минуса (`operator-`). Так как этот оператор применим только к одному объекту, то его перегрузка должна выполняться только через *метод* класса.

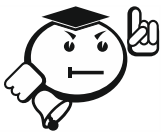
```
1  Complex Complex::operator-() const {
2     return Complex(-real_d, -imag_d);
3  }
```



Поскольку унарный минус не изменяет объект класса `Complex`, то метод перегрузки должен быть константным, чтобы оператор можно было использовать с константными объектами.

Операторы инкремента и декремента

Сложность перегрузки операторов инкремента (`operator++`) и декремента (`operator--`) заключается в том, что они имеют по две формы записи – постфиксную и префиксную. Для того чтобы компилятор отличал постфиксный оператор от префиксного, используется специальный фиктивный параметр. При этом имя переменной не указывается.



Префиксные операторы возвращают объект после его изменения, а постфиксные – в исходном состоянии, с последующим его изменением.

Важно помнить, что перегрузка операторов инкремента и декремента осуществляется через *метод* класса. Рассмотрим постфиксную и префиксную формы перегрузке `operator++`. Оператор декремента перегружается аналогично, поэтому соответствующий программный код опущен.

```
1  Complex& Complex::operator++(){ // префиксный инкремент
2      ++real_d;
3      ++imag_d;
4      return *this;
5  }
6  Complex    Complex::operator++(int){    //    постфиксный
   инкремент
7      Complex temp(real_d, imag_d);
8      ++(*this); // использование префиксного инкремента
9      return temp;
10 }
```

Обратим внимание на то, что здесь возврат исходного объекта при постфиксной записи реализован с помощью временного объекта. Кроме того, возврат значения по ссылке осуществить не удастся, поскольку этот временный объект будет уничтожен после выполнения тела метода. Как и ранее, постфиксная форма перегружена через уже перегруженную префиксную форму.

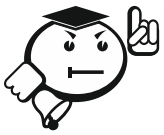
5.3.3 Особые операторы

Оператор индексации

Оператор индексации (`operator[]`) перегружается, когда надо получить доступ к закрытому полю класса, являющемуся массивом. При этом перегрузка

реализуется через *метод* класса. Рассмотрим это на примере класса `Student`, содержащего массив вещественных чисел – оценок по математике.

```
1  class Student{
2     public:
3        double& operator[] (const int i);
4     private:
5        double maths[20];
6  };
7  double& Student::operator[] (const int i) {
8     return maths[i];
9  }
```



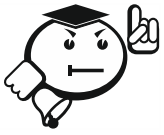
Поскольку `operator[]` имеет приоритет выше, чем оператор присваивания (`operator=`), метод его перегрузки должен возвращать ссылку на объект.

Оператор присваивания

Оператор присваивания (`operator=`) часто используется в ситуациях, когда надо скопировать содержимое одного объекта в другой, ранее созданный объект. Рассмотрим перегрузку оператора присваивания (`operator=`) через *метод* класса.

```
1  Complex& Complex::operator=(const Complex& c){
2     real_d = c.real_d;
3     imag_d = c.imag_d;
4     return *this;
5  }
```

Здесь уже все предельно понятно, поэтому комментарии опущены.



Для того чтобы можно было выполнять цепочку операций присваивания, например a = b = c, метод перегрузки `operator=` должен возвращать ссылку на левый операнд.

5.3.4 Правила перегрузки операторов

В данном подразделе приведены правила по перегрузке операторов. Однако, как сказал классик, «правил нет без исключений», поэтому также приведены и исключения из этих правил, выражающиеся в конечном счете в ограничениях перегрузок.

Общие правила по перегрузке операторов и исключения из них:

1. Перегружать можно только существующие операторы, нельзя их переименовывать или создавать новые.
2. Нельзя изменять приоритет оператора при его перегрузке.
3. Хотя бы один из операндов перегруженного оператора должен быть пользовательского типа данных.
4. Нельзя изменять количество операндов оператора (исключение – `operator()`).
5. Нельзя перегружать операторы для стандартных типов данных (`int`, `float` и пр.).
6. Нельзя перегружать: тернарный оператор (`operator?:`), оператор `sizeof()`, оператор разрешения области видимости (`operator::`), оператор доступа к члену класса (`operator.`) и оператор препроцессора (`operator#`).
7. Для унарных операторов следует использовать перегрузку через методы класса.
8. Для операторов присваивания (`operator=`), индексации (`operator[]`), вызова функции (`operator()`) следует использовать перегрузку через методы класса.
9. Для перегрузки бинарных операторов, которые изменяют левый операнд, например `operator+=`, следует использовать перегрузку через методы класса.
10. Для перегрузки бинарных операторов, которые не изменяют левый операнд, например `operator+`, следует использовать обычные (внешние) или дружественные функции.

5.3.5 Пользовательский класс Complex: часть 2

Здесь сведен воедино весь разработанный класс `Complex` из подразд. 3.3.10 с учетом перегрузок часто используемых операторов, рассмотренных выше. Пример работы с объектом этого модернизированного класса приведен в файле `main.cpp`.

// файл Complex.h

```
1  #ifndef COMPLEX_H
2  #define COMPLEX_H
3  #include <iostream>
4  using namespace std;
5  class Complex{
6      public: // объявление открытых членов класса
7          Complex(double, double); // конструктор 1
8          Complex(double); // конструктор 2
9          Complex(); // конструктор по умолчанию
10         void povComplex();
11         void setValue(double, double); // сеттер
12         Complex getValue(); // геттер
13         void setValueRe(double); // сеттер реальной части
14         double getValueRe(); // геттер реальной части
15         void setValueIm(double); // сеттер мнимой части
16         double getValueIm(); // геттер мнимой части
17         Complex operator-() const;
18         Complex& operator+=(Complex);
19         Complex& operator-=(Complex);
20         friend Complex operator+(Complex);
21         friend Complex operator-(Complex);
22         friend Complex operator+(Complex& c, double v);
23         friend Complex operator+(double v, Complex& c);
24         friend bool operator==(Complex& c1, Complex& c2);
25         friend bool operator!=(Complex& c1, Complex& c2);
26         friend bool operator>=(Complex& c1, Complex& c2);
27         friend bool operator<(Complex& c1, Complex& c2);
28         friend ostream& operator<< (ostream &out,
   Complex& c);
29         friend istream& operator>> (istream &in,
   Complex& c);
30         Complex& operator++();
```

```
31          Complex operator++(int);
32          Complex& operator--();
33          Complex operator--(int);
34          Complex& operator=(const Complex& c);
35          void showComplex ();
36      private: // объявление закрытых членов класса
37          double real_d; // реальная часть
38          double image_d; // мнимая часть
39  };
40  #endif // COMPLEX_H
```

```
                // файл Complex.cpp
1   #include "Complex.h"
2   using namespace std;
3   Complex::Complex(double x, double y): real_d(x),
    image_d(y) { }
4   Complex::Complex(double x): real_d(x), image_d(0) { }
5   Complex::Complex(): real_d(0), image_d(0) { }
6   void Complex::povComplex(){
7       real_d = real_d * real_d;
8       image_d = image_d * image_d;
9   }
10  void Complex::setValue(double re, double im){ // сеттер
11      real_d = re; image_d = im;
12   }
13  Complex Complex::getValue(){ // геттер
14      return Complex(real_d, image_d);
15  }
16  void Complex::setValueRe(double re){ // сеттер
17      real_d = re;
18  }
19  double Complex::getValueRe(){ // геттер
20      return real_d;
21  }
22  void Complex::setValueIm(double im){ // сеттер
23      image_d = im;
24  }
25  double Complex::getValueIm(){ // геттер
26      return image_d;
27  }
```

```
28 Complex Complex::operator-() const {
29     return Complex(-real_d, -imag_d);
30 }
31 Complex& Complex::operator+=(Complex c1){
32     real_d += c1.real_d;
33     image_d += c1.image_d;
34     return *this;
35 }
36 Complex& Complex::operator-=(Complex c1){
37     real_d -= c1.real_d;
38     image_d -= c1.image_d;
39     return *this;
40 }
41 Complex Complex::operator+(Complex c1){
42     Complex temp(this->real_d, this->image_d);
43     temp += c1; // через operator+=
44     return temp;
45 }
46 Complex Complex::operator-(Complex c1){
47     Complex temp(this->real_d, this->image_d);
48     temp -= c1; // через operator-=
49     return temp;
50 }
51 Complex Complex::operator+(Complex& c, double v){
52     return Complex(c.real_d+v, c.imag_d);
53 }
54 Complex Complex::operator+(double v, Complex& c){
55     return c + v; // через operator+(Complex&, double )
56 }
57 bool Complex::operator==(Complex& c1, Complex& c2){
58     return  c1.real_d  ==  c2.real_d  &&  c1.imag_d  ==
    c2.imag_d;
59 }
60 bool Complex::operator!=(Complex& c1, Complex& c2){
61     return !(c1 == c2); // через operator==
62 }
63 bool Complex::operator>=(Complex& c1, Complex& c2){
64     return c1.real_d >= c2.real_d && c1.imag_d >=
    c2.imag_d;
```

```
65 }
66 bool Complex::operator<(Complex& c1, Complex& c2){
67     return !(c1>=c2); // через operator>=
68 }
69 std::ostream&   Complex::operator<<(std::ostream   &out,
   Complex& c){
70     out << c.real_d << ", " << c.imag_d;
71     return out;
72 }
73 std::istream&    Complex::operator>>(std::istream    &in,
   Complex& c){
74     in >> c.real_d >> c.imag_d;
75 }
76 Complex& Complex::operator++(){
77     ++real_d;
78     ++imag_d;
79     return *this;
80 }
81 Complex Complex::operator++(int){
82     Complex temp(real_d, imag_d);
83     ++(*this); // использование префиксного инкремента
84     return temp;
85 }
86 Complex& Complex::operator--(){
87     --real_d;
88     --imag_d;
89     return *this;
90 }
91 Complex Complex::operator--(int){
92     Complex temp(real_d, imag_d);
93     --(*this); // использование префиксного декремента
94     return temp;
95 }
96 void Complex::showComplex (){
97     cout << "(" <<real_d << "," <<image_d << ")"<< endl;
98 }
```

// файл main.cpp

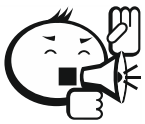
```
1 #include "Complex.h"
2 int main(){
```

```
3       Complex a(0.0,0.0);
4       Complex b(1.0,0.0);
5       Complex c = -b; // перегруженные operator-, operator=
6       Complex d = ++a; // перегруженный operator++
7       if(c == b){ // перегруженный operator==
8          std::cout << a + b << std::endl; // перегруженные
    operator<<, operator+
9       }
10      std::cin >> a; // перегруженный operator>>
11      // остальной код программы
12      return 0;
13  }
```

Кратко прокомментируем код файла `main.cpp`. В функции `main` создаются 4 объекта пользовательского типа `Complex` – `a`, `b`, `c` и `d`, при этом используется конструктор с двумя параметрами (конструктор 1), а также перегруженный оператор присваивания и операторы инкремента и унарного минуса. Далее, с помощью перегрузки оператора сравнения, выполняется проверка объектов на равенство и вывод их суммы (перегрузка бинарного плюса). После чего пользователь может переопределить значение одного из объектов, используя перегруженный оператор ввода.

5.4 Виртуальные методы (функции)

Виртуальные функции, кратко описанные в подразд. 4.8, в сочетании с использованием наследования реализуют *динамический* полиморфизм. Напомним, что виртуальным методом (функцией) называется метод (функция), объявленный в базовом классе (с ключевым словом `virtual`) и переопределенный в производных классах. Тогда сигнатура виртуального метода (функции) определяет его интерфейс, а производные классы его реализуют.



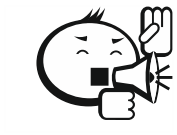
При вызове виртуального метода (функции) надо учесть особенности организации его (ее) вызова. Вызов виртуального метода (функции) должен производиться для указателя (или ссылки) базового класса, который указывает на экземпляр производного от него класса.

Рассмотрим пример динамического полиморфизма для классов, образующих иерархию с помощью наследования. Так, создадим базовый класс `Animal` и производный от него `Tiger`.

```
1  class Animal{ // абстрактный базовый класс
2     public:
3        virtual void eat() = 0; // чисто виртуальный метод
   (функция)
4  };
5  class Tiger : Animal{ // производный класс
6     public:
7        virtual void eat() override{ // переопределение
8           // требуемый код
9        }
10 };
11 void doEat(Animal *a){ // внешняя функция (по указателю)
12    a->eat(); // проявление полиморфизма
13 }
14 void doEat(Animal &a){ // внешняя функция (по ссылке)
15    a.eat(); // проявление полиморфизма
16 }
17 int main()
18    Animal *ptrA; // указатель на базовый класс
19    Animal &refA; // ссылка на базовый класс
20    Tiger t; // экземпляр производного класса
21    ptrA = &t; // указатель перемещен на объект Tiger
22    refA = t; // ссылка перемещена на объект Tiger
23    prtA->eat(); // полиморфизм – вызов Tiger::eat()
24    doEat(ptrA); // полиморфизм (передача указателя)
25    refA.eat(); // полиморфизм – вызов Tiger::eat()
26    doEat(refA); // полиморфизм (передача ссылки)
27    return 0;
28 }
```

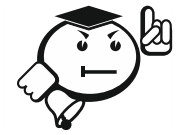
Проанализируем приведенный код. Так, в данном случае родительский класс `Animal` является абстрактным, поскольку он содержит одну чисто виртуальную функцию `eat()`. Поэтому невозможно создать объект родительского класса, однако можно объявить ссылку и указатель на этот класс. В производном

классе `Tiger` выполняется переопределение виртуальной функции со спецификатором `override`. В результате полиморфизм позволяет единообразно обращаться с разными объектами одной иерархии.



Метод (функция) производного класса считается переопределением виртуального метода (функции) базового класса в случае, если имена, аргументы, типы возвращаемых значений этих методов (функций) совпадают. Спецификатор `override` позволяет облегчить нахождение ошибок: если метод не переопределяет виртуальный метод (функцию), компилятор выдаёт ошибку.

Видно, что в приведенном коде определены две внешние функции `doEat()`, которые в качестве аргументов принимают указатель и ссылку на базовый класс соответственно. Тела этих функций содержат вызов виртуального метода (функции), так как на этапе компиляции неизвестно, метод какого класса будет вызван. Поэтому код вызова формируется по ходу выполнения программы. В функции `main()` создается экземпляр класса `Tiger`, после чего указатель и ссылка (базового класса) перемещаются на этот объект. В завершение выполняются вызовы внешних функций и вызов виртуального метода (функции) соответственно через операторов доступа к членам класса «`.`» (точка) и «`->`» (стрелка).

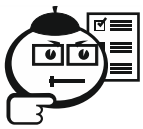


Правила организации динамического полиморфизма:

1. Классы должны образовывать иерархию наследования.
2. Классы должны содержать методы с одинаковым именем, списком параметров и типом возвращаемого значения и должны быть помечены как `virtual`.
3. Должен быть использован спецификатор `override` для всех переопределений методов.
4. Метод (функция) производного класса, совпадающий по сигнатуре с базовым, но объявленный как константный (`const`), не воспринимается как виртуальный.
5. Нельзя вызывать виртуальные методы (функции) в конструкторах или деструкторе класса.

Контрольные вопросы по разделу 5

1. Что такое полиморфизм в ООП?
2. В чем различие динамического и статического полиморфизма?
3. Что такое перегрузка методов (функций)?
4. Зачем используются перегрузки операторов?
5. Можно ли создавать новые операторы и перегружать их?
6. Какой тип перегрузки следует использовать для бинарных операторов, которые изменяют левый операнд?
7. Какой тип перегрузки следует использовать для бинарных операторов, которые не изменяют левый операнд?
8. Какой тип перегрузки следует использовать для унарных операторов?
9. Можно ли изменить приоритет оператора при его перегрузке?
10. Для чего используется ключевое слово `virtual`?
11. Для чего используется спецификатор `override`?



Задания для самостоятельного выполнения

1. Используя класс `Complex` из п. 5.3.5, перегрузить операторы: `operator*`, `operator/`, `operator*=`, `operator/=`, `operator>`, `operator<=`.
2. Создать класс для определения двумерного массива размерностью 5×5. Реализовать конструктор класса для заполнения массива случайными числами от –10 до 10. Переопределить оператор индексации.
3. Создать абстрактный базовый класс `Animal` (животное) с виртуальным методом (функцией) издавания звука (вывод текста в консоль). Создать производные классы: `Dog`, `Cat` и `Cow`, в каждом из которых переопределить метод базового класса (вывод в консоль надписи «`Woof`», «`Meow`», «`Moo`» соответственно).
4. Создать абстрактный базовый класс `Figure` (фигура) с виртуальным методом (функцией) вычисления площади. Создать производные классы `RectangleFigure` (прямоугольник), `CircleFigure` (круг) и `RightTriangleFigure` (прямоугольный треугольник) со своими реализациями метода вычисления площади фигуры.

Заключение

Мы строили, строили и наконец построили!

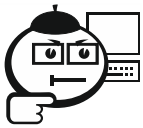
Э. Н. Успенский.
Приключения Чебурашки

Образование – замечательное дело, надо лишь хоть иногда вспоминать о том, что ничему, что стоит знать, научить невозможно.

Оскар Уайльд

Программирование – это искусство, а ему, как известно, нельзя научить, независимо от того, кто вас учит [11]. При этом авторы пособия убеждены, что к нему можно приобщить. Достижению именно этой цели посвящено данное пособие. Как уже было упомянуто, рассмотрены только некоторые особенности конструкций языка C++. Например, не затронута очень важная тема, посвященная параллельному программированию, а также только кратко рассмотрено использование стандартной библиотеки языка C++. При этом в пособии есть весь необходимый материал, на основе которого обучающийся уже самостоятельно может развивать свои знания и навыки. Так, рассмотрены главные принципы ООП: инкапсуляция, наследование и полиморфизм, а также обработка исключений и механизмы шаблонов, т. е. то, без чего невозможно дальнейшее совершенствование навыков создания программного обеспечения. При этом авторы надеются, что их работа оказалась ненапрасной.

Для того чтобы подстегнуть интерес читателя к дальнейшему изучению программирования, приведем ссылку на видео, где наглядно рассказывается об особенностях языков программирования, в том числе C++, и работы программных средств. Это, по замыслу авторов, позволит читателю определиться, куда «двигаться дальше».



О языках программирования и компиляторах:
что следует знать, чего можно не знать, что знать не нужно

Список использованных источников и литературы

Три магнитофона, три кинокамеры
заграничных, три портсигара
отечественных, куртка замшевая... три.

Х/ф «Иван Васильевич
меняет профессию»

1. Страуструп, Б. Программирование: принципы и практика с использованием C++ / Б. Страуструп. – 2-е изд. – М. : ИД «Вильямс», 2016. – 1328 с.
2. Босуэлл, Д. Читаемый код или программирование как искусство / Д. Босуэлл, Т. Флетчер. – СПб. : Питер, 2012. – 203 с.
3. GDB Online Debugger [Электронный ресурс]. – URL: https://www.onlinegdb.com (дата обращения: 01.06.2022).
4. Replit [Электронный ресурс]. – URL: https://replit.com (дата обращения: 01.06.2022).
5. Dev-C++ [Электронный ресурс]. – URL: http://www.dev-cpp.com (дата обращения: 01.06.2022).
6. Visual Studio [Электронный ресурс]. – URL: https://visualstudio.microsoft.com/ru/vs/ (дата обращения: 01.06.2022).
7. Qt Creator [Электронный ресурс]. – URL: https://www.qt.io/product/development-tools (дата обращения: 01.06.2022).
8. Страуструп, Б. Дизайн и эволюция C++ / Б. Страуструп. – М. : ДМК Пресс ; СПб. : Питер, 2006. – 448 с.
9. Эккель, Б. Философия C++. Введение в стандартный C++ / Б. Эккель. – 2-е изд. – СПб. : Питер, 2004. – 572 с.
10. Шилдт, Г. C++: базовый курс / Г. Шилдт. – 3-е изд. – М. : ИД «Вильямс», 2010. – 620 с.
11. Элкинс, Д. Почему нельзя научить искусству : пособие для студентов художественных ВУЗов / Д. Элкинс. – М. : ООО «Ад Маргинем Пресс», 2015. – 330 с.

Список рекомендуемой литературы

Выучить от сих до сих! Приеду – проверю!

Х/ф «Джентльмены удачи»

1. Арлоу, Д. UML 2 и унифицированный процесс. Практический объектно-ориентированный анализ и проектирование / Д. Арлоу, И. Нейштадт. – СПб. : Символ-Плюс, 2015. – 624 с.
2. Объектно-ориентированное программирование на C++ [Электронный ресурс] : учебник / И. В. Баранова, С. Н. Баранов, И. В. Баженова [и др.]. – Красноярск : СФУ, 2019. – 288 с. // «Лань» : электронно-библиотечная система. – URL: https://e.lanbook.com/book/157572 (дата обращения: 01.06.2022).
3. Босуэлл, Д. Читаемый код или программирование как искусство / Д. Босуэлл, Т. Флетчер. – СПб. : Питер, 2012. – 203 с.
4. О'Двайр, А. Осваиваем C++ 17 STL [Электронный ресурс] / А. О'Двайр. – М. : ДМК Пресс, 2018. – 352 с. // «Лань» : электронно-библиотечная система. – URL: https://e.lanbook.com/book/116126 (дата обращения: 01.06.2022).
5. Саттер, Г. Решение сложных задач на C++ : пер. с англ. / Г. Саттер. – М. : ИД «Вильямс», 2008. – 400 с. – Сер. C++ In-Depth.
6. Скворцова, Л. А. Объектно-ориентированное программирование на языке C++ [Электронный ресурс] : учеб. пособие / Л. А. Скворцова. – М. : РТУ МИРЭА, 2020. – 246 с. // «Лань» : электронно-библиотечная система. – URL: https://e.lanbook.com/book/163862 (дата обращения: 01.06.2022).
7. Страуструп, Б. Дизайн и эволюция C++ / Б. Страуструп. – М. : ДМК Пресс ; СПб. : Питер, 2006. – 448 с.
8. Страуструп, Б. Программирование: принципы и практика с использованием C++ / Б. Страуструп. – 2-е изд. – М. : ИД «Вильямс», 2016. – 1328 с.
9. Мейерс, С. Эффективный и современный C++: 42 рекомендации по использованию C++ 11 и C++ 14 : пер. с англ. / С. Мейерс. – М. : ИД «Вильямс», 2016. – 304 с.

10. Швец, А. Погружение в паттерны проектирования [Электронный ресурс] : интернет-издание / А. Швец. – 2018. – 406 с.
11. Шилдт, Г. C++: базовый курс : пер. с англ. / Г. Шилдт. – 3-е изд. – М. : ИД «Вильямс», 2010. – 620 с.

Приложение А
(информационное)
Инструкция по установке и настройке Microsoft Visual Studio

Все становится ясно и понятно. Никаких тайн и загадок.

М/ф «Смешарики»

Для установки Microsoft Visual Studio (MVS) нужно:

1. Перейти на сайт (MVS) по ссылке https://visualstudio.microsoft.com/ru/ и скачать установочный файл версии Community (рис. А.1).

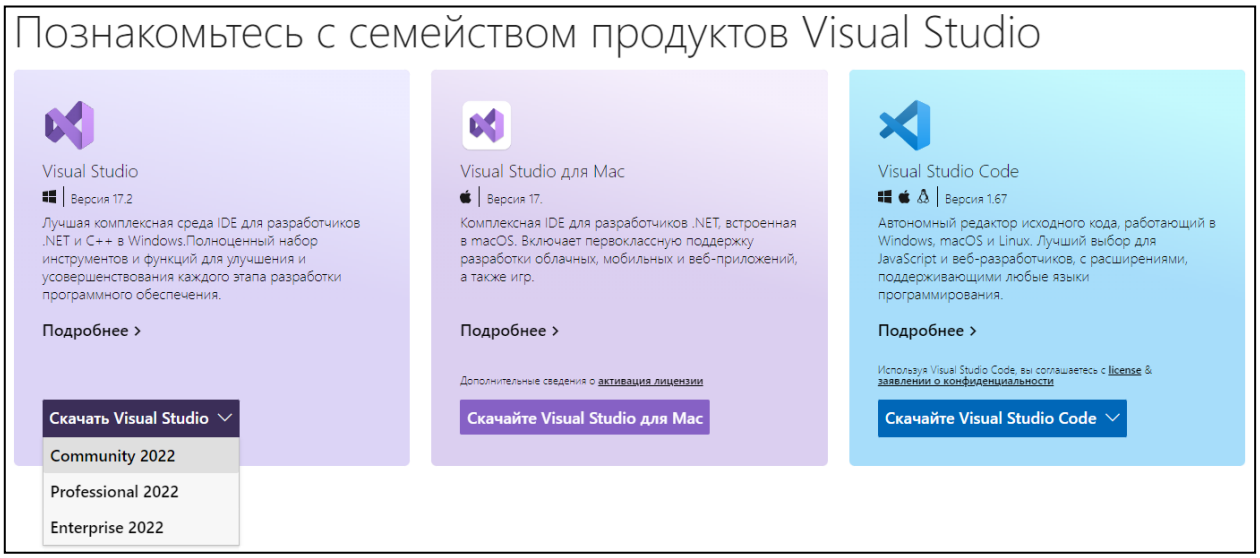


Рис. А.1 – MVS: семейство продуктов и выбор версии Community

2. Открыть скачанный файл. Далее нажать `Продолжить` и дождаться окончания установки.

3. В открывшемся окне выбрать пункт `Разработка классических приложений C++`, оставить каталог установки по умолчанию и нажать `Установить` (рис. А.2).

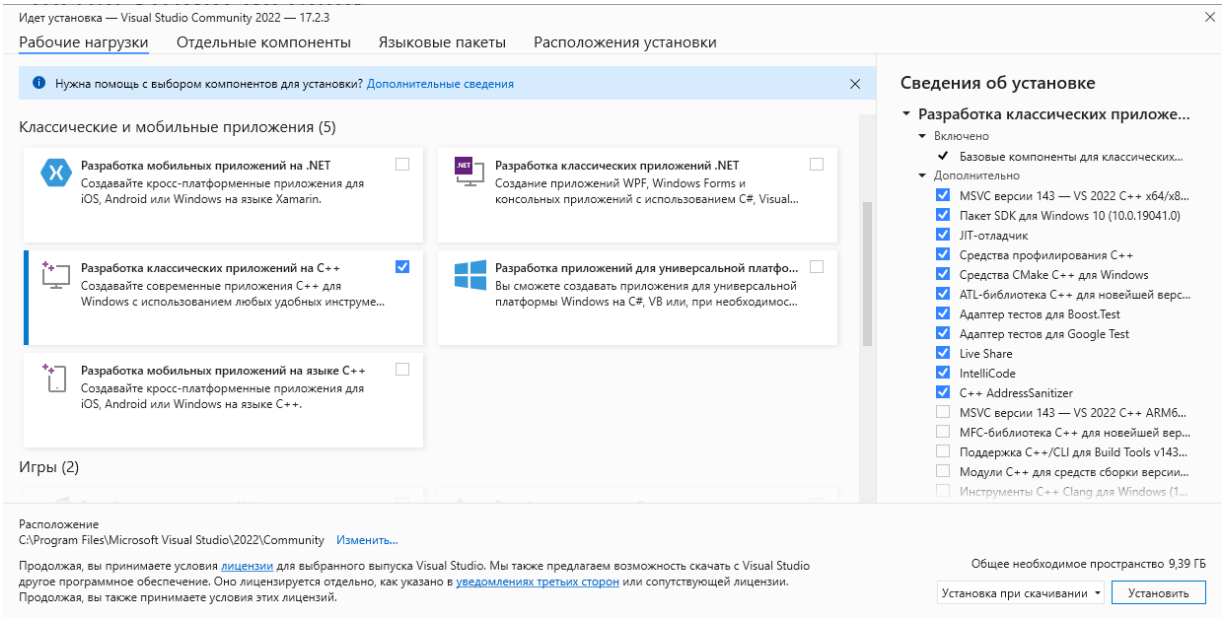
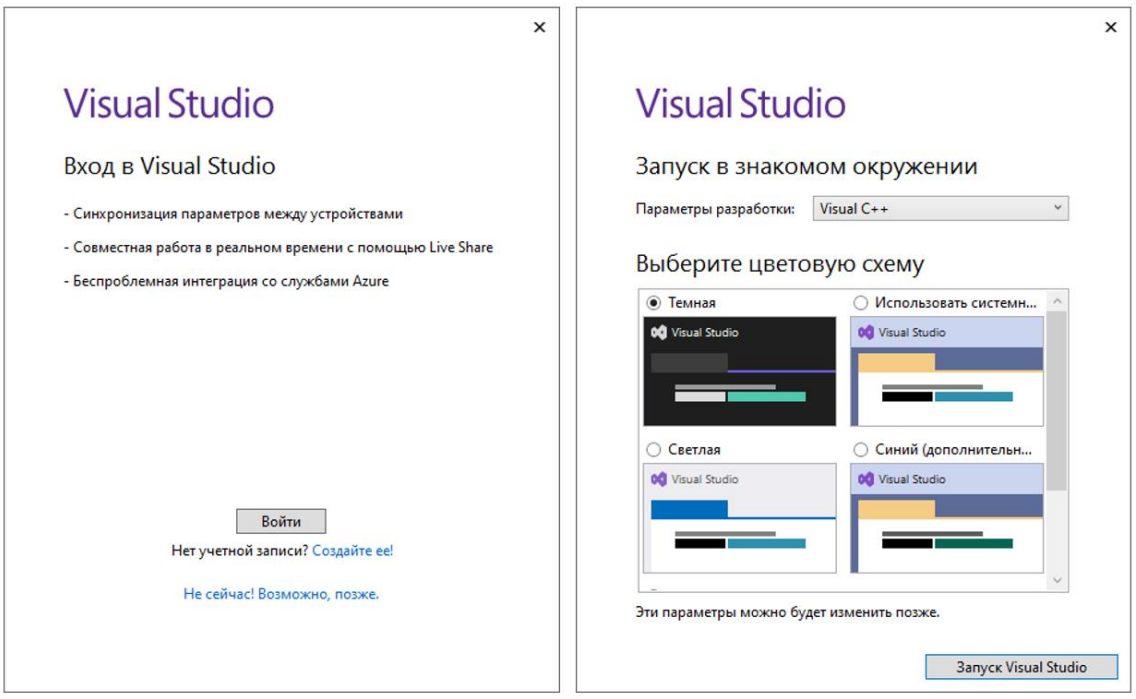


Рис. А.2 – MWS: параметры установки MVS

4. После установки создать учетную запись Microsoft или войти без нее (рис. А.3, *а*).

5. Выбрать параметры разработки `Visual C++` и требуемую тему оформления. Затем нажать `Запуск Visual Studio` (рис. А.3, *б*) и дождаться открытия MVS.



а) *б*)

Рис. А.3 – MVS: виды окон для входа в учетную запись Microsoft (*а*) и предварительной настройки окружения (*б*)

Приложение Б
(информационное)
Инструкция по установке и настройке Qt Creator

Твои мысли подобны кругам на воде, друг мой. В волнении исчезает ясность, но если ты дашь волнам успокоиться, ответ станет очевидным.

М/ф «Кунг-фу Панда»

Для установки Qt Creator (Qt) нужно:

1. Перейти на сайт Qt Framework по ссылке https://www.qt.io/download-open-source, нажать `Download the Qt Online Installer` (находится в середине страницы). На открывшейся странице нажать `Download` (рис. Б.1), тем самым запустив процесс скачивания установочного файла (`Qt Online Installer`).

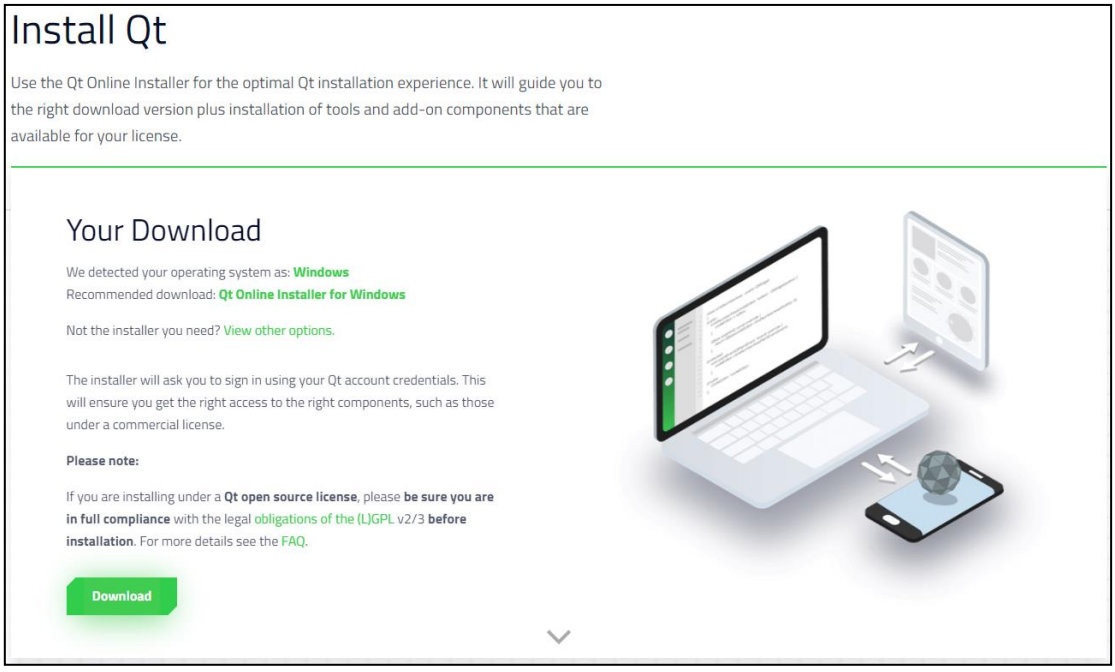


Рис. Б.1 – Qt: вид страницы загрузки `Qt Online Installer`

2. Открыть скачанный файл. В открывшемся окне (`Установка Qt`) во вкладке `Добро пожаловать` создать учетную запись или войти (при наличии созданной ранее учетной записи) в Qt, после чего нажать `Далее` (рис. Б.2).

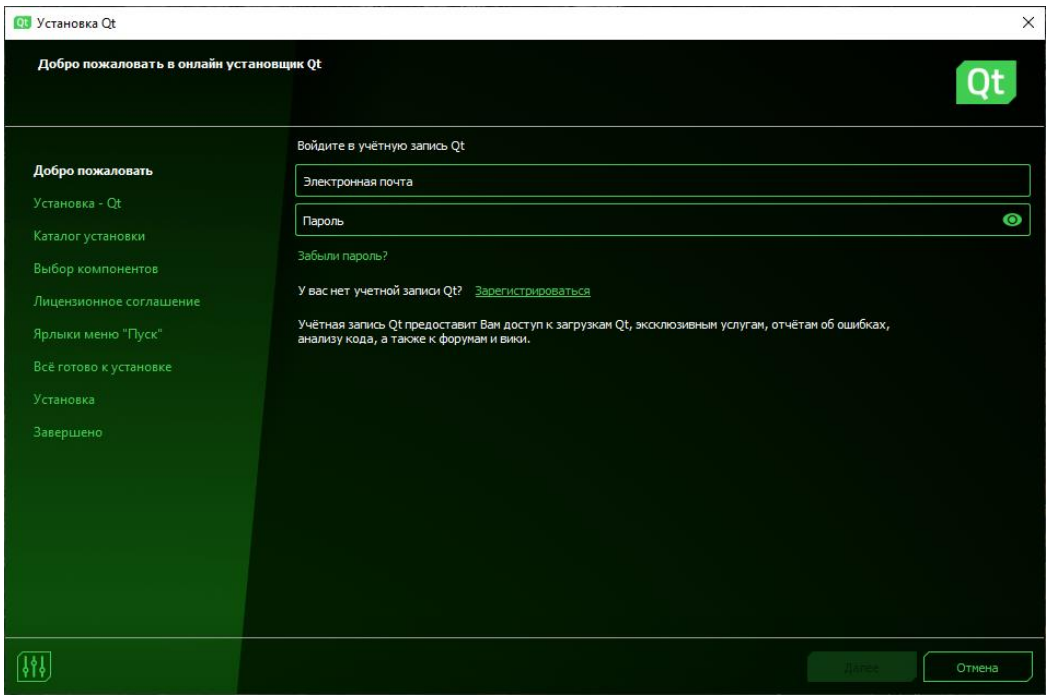


Рис. Б.2 – Qt: вид страницы `Установка Qt`, меню `Добро пожаловать`

3. Ознакомиться с обязательствами по использованию Qt в качестве открытого программного обеспечения, выбрать пункт `Я – частное лицо` и нажать `Далее` (рис. Б.3).

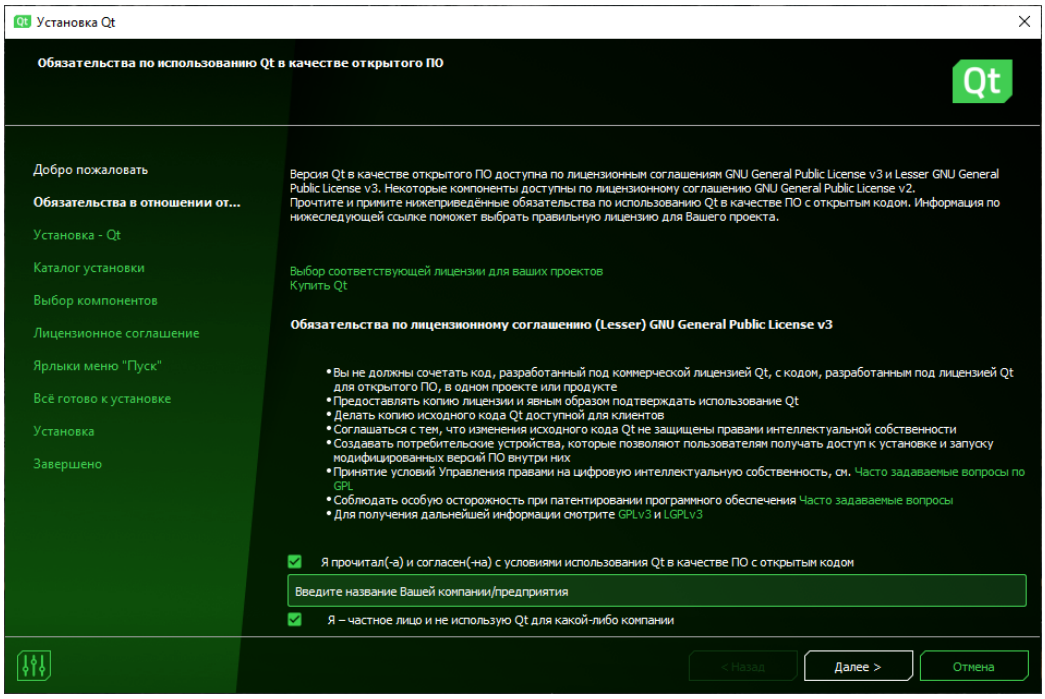


Рис. Б.3 – Qt: меню `Обязательства в отношении использования Qt ...`

4. Настроить требуемый вариант отправки анонимной статистики о работе Qt и нажать `Далее` (рис. Б.4).

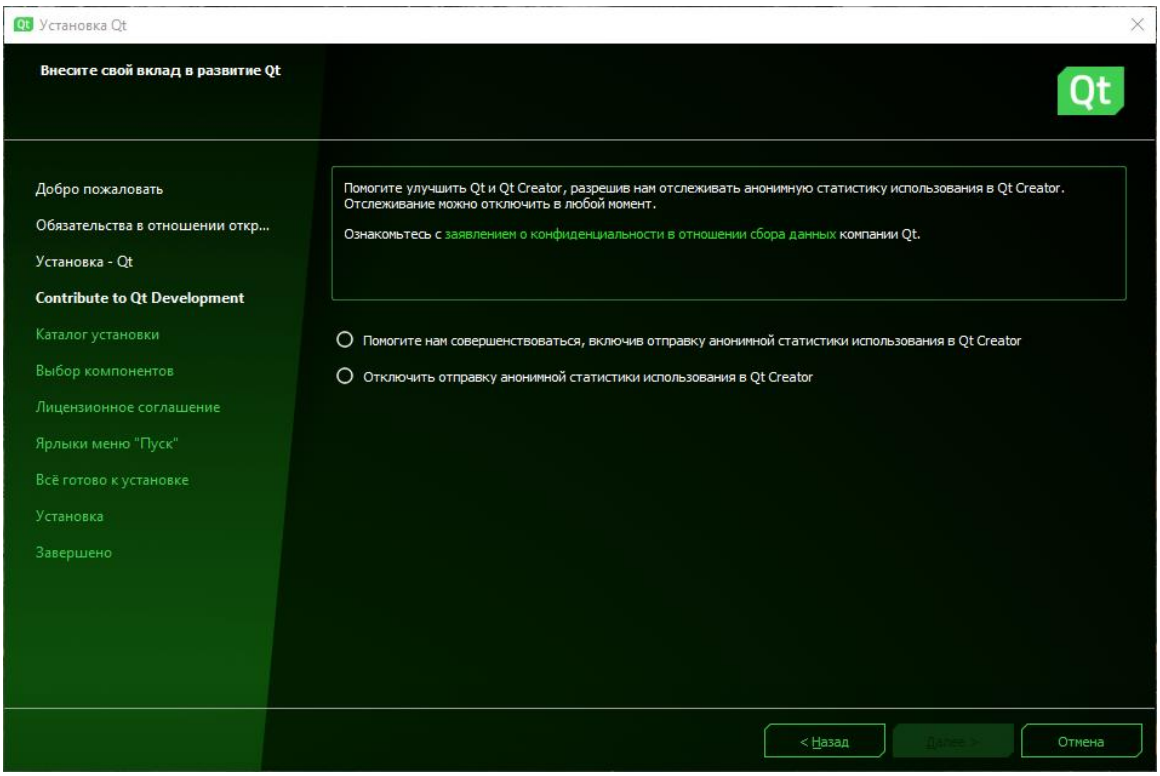


Рис. Б.4 – Qt: меню `Contribute to Qt Development`

5. Выбрать каталог (рекомендуется оставить по умолчанию), активировать `Выборочная установка` и нажать `Далее` (рис. Б.5).

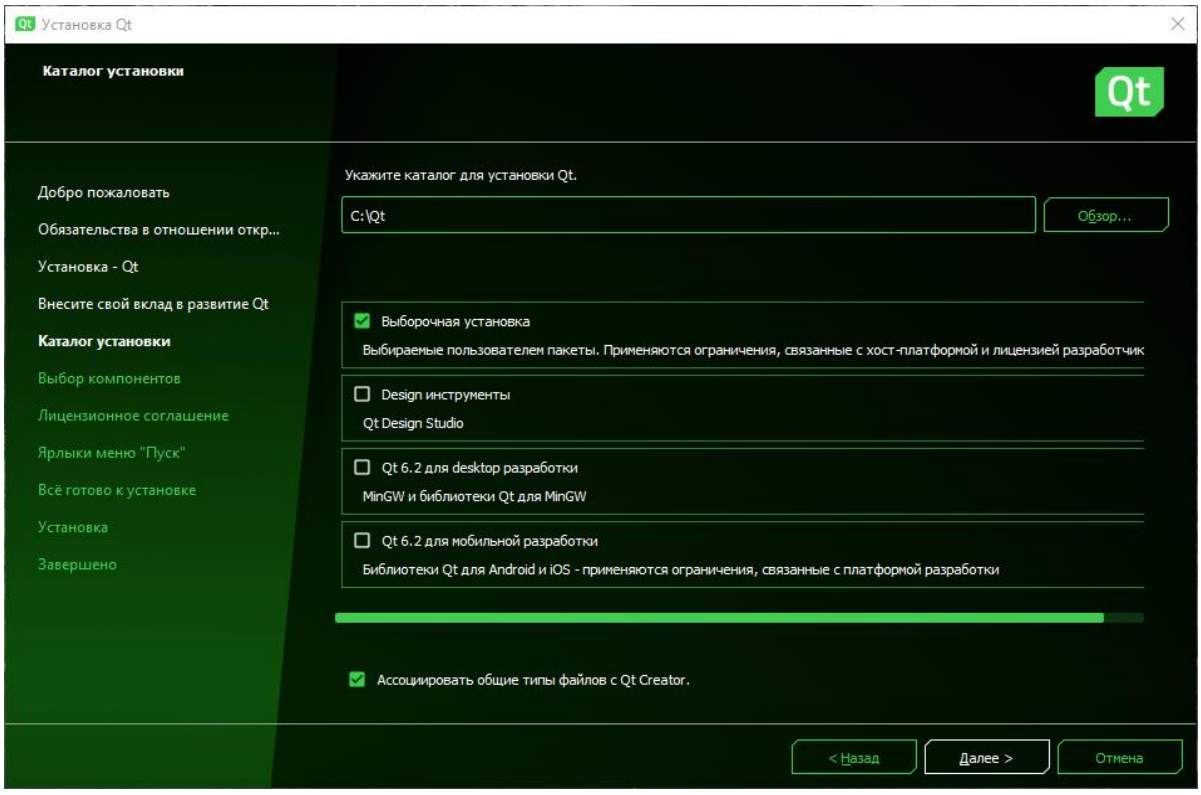


Рис. Б.5 – Qt: меню `Contribute to Qt Development`

6. В меню `Выбор компонентов` активировать следующие пункты:

- `Qt` (рис. Б.6):
  - ✓ MSVC 2019 64-bit;
  - ✓ Sources;
  - ✓ Qt 5 Compatibility Module;
  - ✓ Qt Data Visualization.

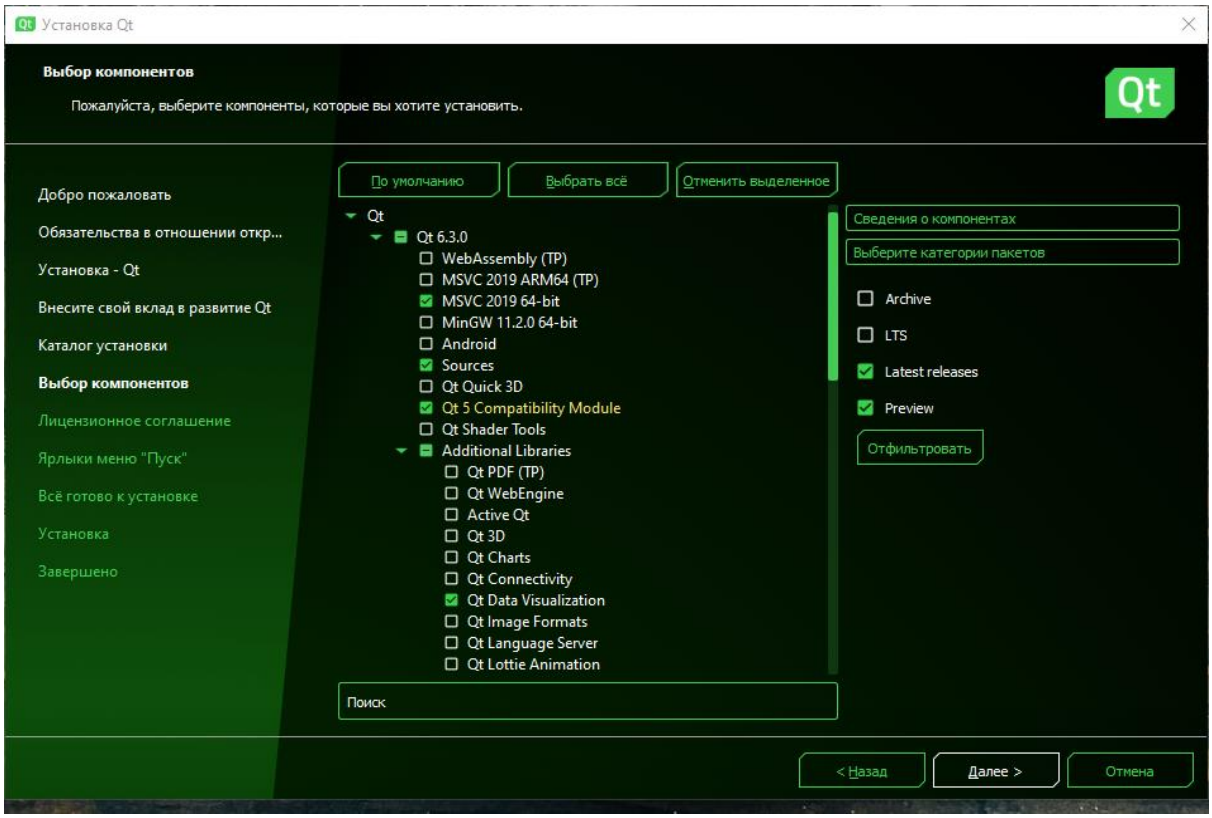


Рис. Б.6 – Qt: меню `Выбор компонентов`, пункт `Qt`

- `Developer and Designer Tools` (рис. Б.7):
  - ✓ Qt Creator 7.0.2;
  - ✓ Debugging Tools for Windows;
  - ✓ Qt Creator 7.0.2 Debug Symbols;
  - ✓ Qt Design Studio 3.4.0;
  - ✓ CMake 3.21.1 64-bit;
  - ✓ Ninja 1.10.2.

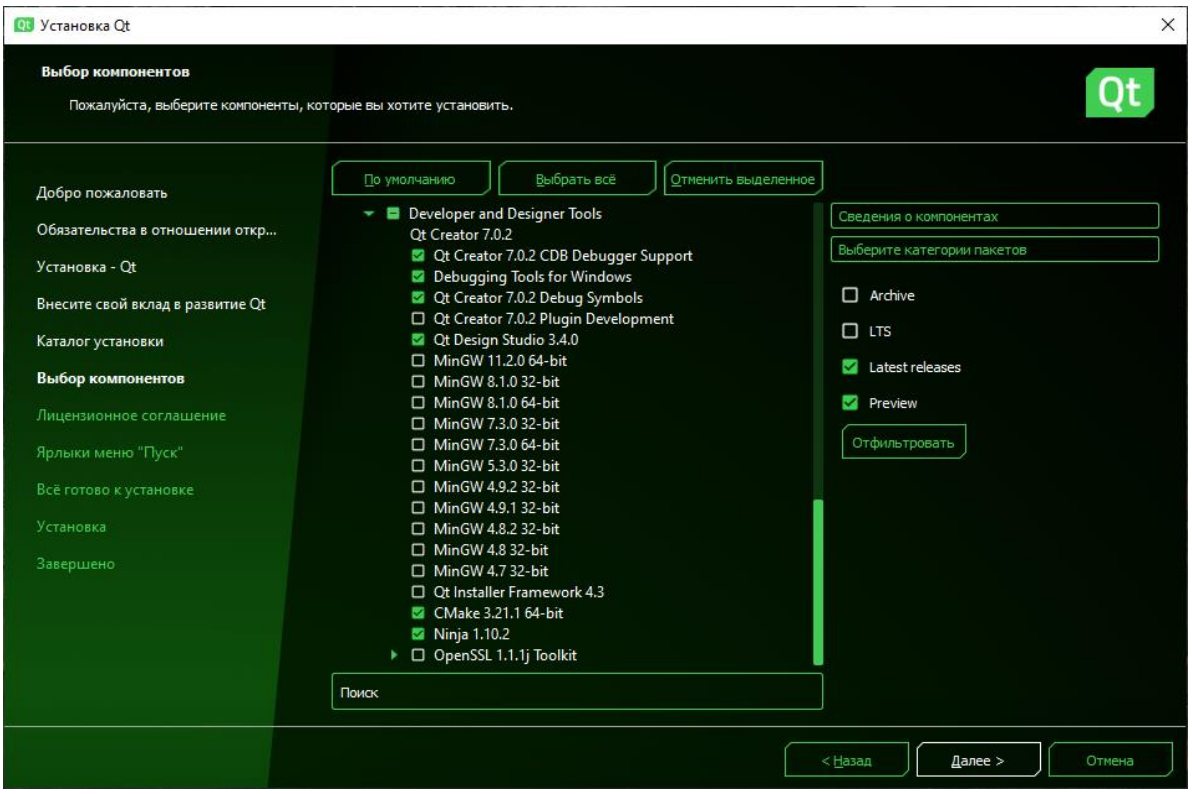


Рис. Б.7 – Qt: меню `Выбор компонентов`, пункт `Developer and Designer Tools`

7. Согласиться с лицензионным соглашением и нажать `Далее` (рис. Б.8).

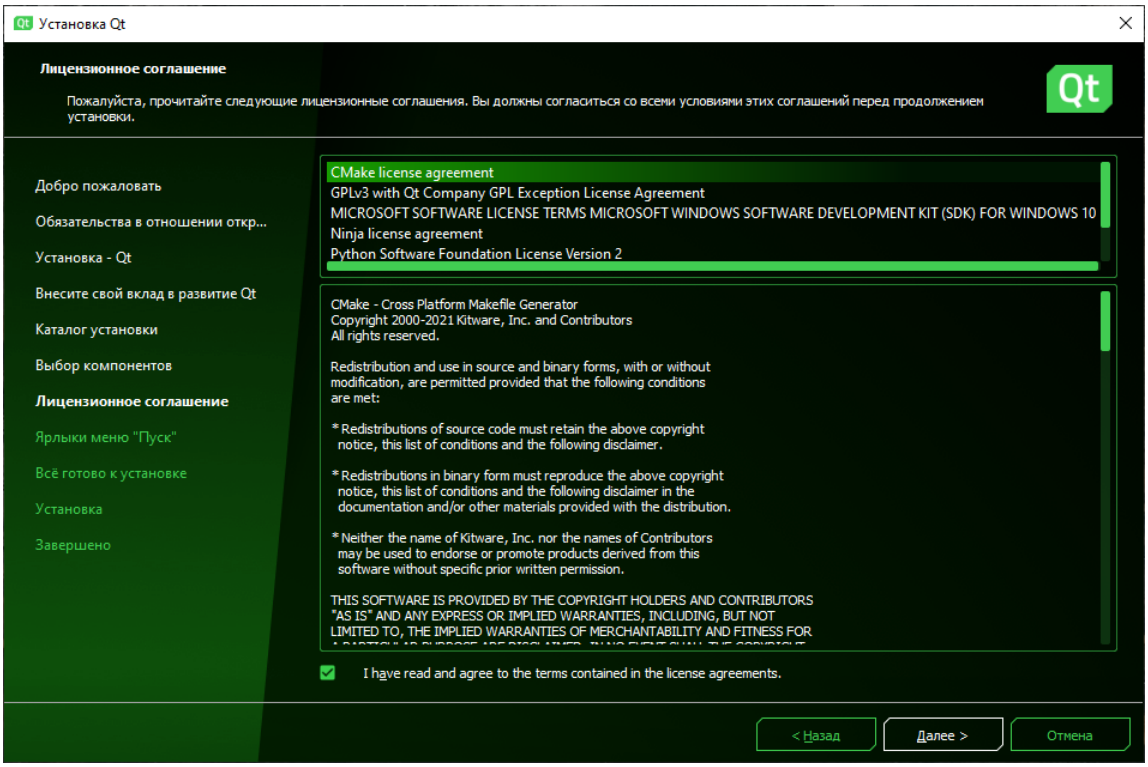


Рис. Б.8 – Qt: меню `Лицензионное соглашение`

8. Ввести требуемое имя папки, которое будет отображаться в меню Пуск, нажать Далее (рис. Б.9), после чего на открывшейся странице нажать Установить.

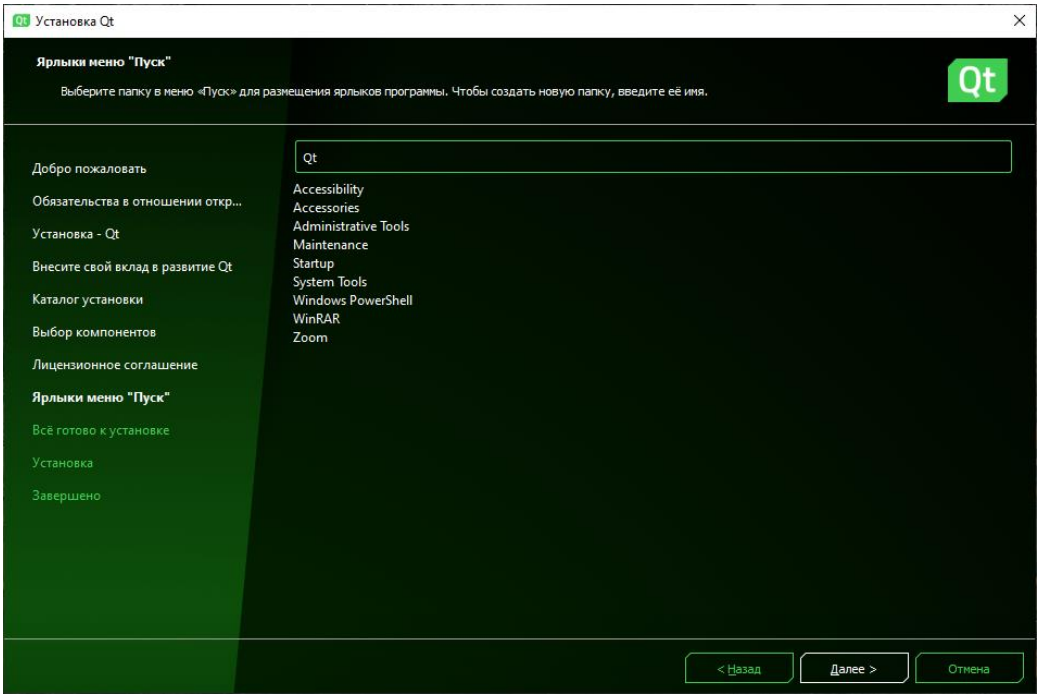


Рис. Б.9 – Qt: меню Ярлыки меню "Пуск"

Приложение В
(информационное)
Рекомендации по именованию элементов программного кода

Как вы яхту назовете, так она и поплывет.

М/ф «Приключения капитана Врунгеля»

Существует несколько подходов к именованию классов, их полей, методов и объектов, а также переменных и функций. Однако не стоит забывать, что в большом коллективе разработчиков используется, как правило, один из этих подходов. Это позволяет добиться удобочитаемости кода, написанного разными разработчиками. При этом каждый из подходов имеет свои преимущества и недостатки, но их рассмотрение выходит за рамки данного пособия. Заинтересованный читатель может самостоятельно рассмотреть эти подходы и выработать свой собственный, который будет использовать при написании программ.

В пособии подобный стиль не был широко использован, т. к. рассматриваемые небольшие примеры были интуитивно понятными и хорошо читаемыми. Кроме того, авторы стремились к минимализму при представлении программного кода. Тем не менее, авторы рекомендуют придерживаться следующих правил именования элементов программного кода:

1. При задании имен элементов программного кода использовать слова на английском языке, а не транслитерацию:

`faylModeli` – плохо;

`fileModel` – хорошо.

2. При использовании аббревиатур и сокращений для задания имен использовать нижний регистр, слова общего использования (`compute`, `calculate`, `find`) не сокращать, а слова, специфичные для предметной области (`html`, `cpu`), не расшифровывать:

`loadSTEPModel()` – плохо;

`loadStepModel()` – хорошо.

3. Для парных элементов кода использовать составные имена, содержащие слова-антонимы (`get/set`, `add/remove`, `create/destroy`, `start/stop`, `insert/delete`, `increment/decrement`, `begin/end`, `first/last`, `up/down`, `min/max`, `next/previous`, `old/new`, `open/close`, `show/hide`, `suspend/resume` и т. д.).

4. Пространствам имен задавать имена, используя нижний регистр (`namespace mode`, `namespace modelantenna`).

5. В именах функций и методов, реализующих *вычисления*, использовать поясняющие слова `compute` или `calculate` (`calculateFarFieldData()`), *поиск* – `find` (`findMaxValue()`), *инициализацию* – `initialize` или `init` (`initializeData()`).

6. Функциям и методам, возвращающим значения, задавать имена в зависимости от того, что они возвращают, а функциям и методам, не возвращающим значения (процедурам) (`void`), – в зависимости от того, что они делают.

7. Классам и структурам (пользовательским типам данных) задавать имена, используя смешанный регистр, начиная со строчной буквы (`CommandClass`, `MainWindow`).

8. Объектам (экземплярам) класса (или структуры) и переменным с «большой» областью видимости (переменные в функциях и методах) задавать имена, используя смешанный регистр, начиная с прописной буквы (`command`, `mainWindow`). Кроме того, объектам классов следует задавать имена, названные по их типу и назначению (`«Point startingPoint;»`, `«Name firstName;»`). Переменным с «малой» областью видимости (счетчик циклов) и итераторам задавать только короткие имена (`i`, `j`, `k`, `l`, `m`, `n` – для числовых переменных (`i` использовать во внешнем цикле, а `j` и `k` – во вложенных) и `c`, `d`, `s` – для строк и символов):

«`auto testVariable = foo(i, j);`» – плохо;

«`auto testVariable = foo(year, dayOfYear);`» – хорошо.

9. В имена переменных и полей, описывающие число объектов, включать префикс `n` (`nLines`, `nPoints`).

10. Булевым переменным, полям, функциям и методам задавать имена, используя префикс `is` (`isVisible`, `isChecked()`, `isFinished()`), а иногда – `has`. При этом в именах не использовать отрицание:

«`bool isNotEmpty;`» – плохо;

«`bool isEmpty;`» – хорошо.

11. Методам классов и функциям задавать глагольные имена, используя нижний регистр, начиная со строчной буквы (`addCommand()`). При этом не следует использовать имена объекта класса в именах его методов, например для объекта `line` класса `Line`:

`line.getLineLength()` – плохо;

`line.getLength()` – хорошо.

12. Закрытым (`private`) полям класса задавать имена, используя смешанный регистр, начиная с ведущего подчеркивания («`int _numSegm;`»). Если поле представляет измеряемую величину, то в имя надо (по возможности) включать название единицы измерения поля (`int _timeoutSeconds`).

13. Интерфейсным (`public`) методам класса, предоставляющим доступ к его закрытым (`private`) членам, задавать имена, начинающиеся с приставок `get` и `set` (`model.getName()`, `model.setName(name)`).

14. Типу данных в шаблоне задавать имя `T` или `Type` (`template <class T>`). Если в шаблоне используется несколько типов данных, то указывать их полные имена (`template <class Key, class Hash, class Allocator>`).

15. Константным переменным и значениям в перечислениях (`enum`) задавать имена, используя верхний регистр и подчеркивание в качестве разделителя (`PI, COLOR_RED, COLOR_GREEN`).

16. Массивам объектов задавать имена, используя множественное число («`int values[];`», «`vector<Points> points;`»).

17. В имена классов – обработчиков исключений добавлять постфикс `Exception` (`class AcessException`).

18. Не использовать дополнительное именование указателей, кроме ситуаций, где особо важен тип данных:

«`Point* ptrPoint;`» – плохо;

«`Point* point;`» – хорошо.

19. Использовать (по возможности) псевдонимы имен для именования составных типов данных («`using vector_d = std::vector<double>;`», «`using ui = unsigned int;`»).

Приложение Г
(информационное)
Стандартная библиотека шаблонов

– Да где же красавец-то?! Он же кривой!

– Ну где кривой, где кривой-то?! К тому же, если вот так смотреть, то и ничего и не кривой!.. А совсем даже наоборот!

М/ф «Добрыня Никитич и Змей Горыныч»

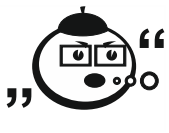
Вводные замечания

Если читающий добрался до материала, изложенного здесь, то базовые знания по ОПП и навыки по программированию на C++ он уже получил. Возможно, при изучении материала основной части пособия могли появляться вопросы типа: почему же в языке C++ не всегда и не все сделано интуитивно понятно? Дело в том, что язык C++ – это универсальный язык программирования, в котором есть практически все, что может когда-либо пригодится программисту в его деятельности. А за универсальность, как известно, надо чем-то платить. Этой платой, по сути, и стало использование сложного синтаксиса языка. Тем не менее, не все так сложно, как может показаться на первый взгляд.

Еще раз отметим, что язык C++ является строго типизированным, поэтому явное задание типа данных является ключевым аспектом его использования. Так, в подразд. 2.3 и 3.3 рассмотрен механизм создания шаблонов: функций и классов. В частности, на примере пользовательских классов `Complex` и `Vector` показана его универсальность, которая позволяет использовать обобщенное программирование при разработке программ. Механизм шаблонов широко используется в стандартной библиотеке шаблонов (*Standard Template Library, STL*), которая является частью стандартной библиотеки языка C++. Она содержит шаблонные классы и функции общего назначения, реализующие большое число алгоритмов и типов данных. Поэтому прежде чем создавать свои массивы, писать циклы и реализовывать функции, работающие с этими массивами, а также созда-

вать новые классы, следует сначала заглянуть в стандартную библиотеку шаблонов, где наверняка можно найти готовое решение, которое позволит избежать ненужных ошибок и тем самым ускорить процесс разработки программ.

Для подтверждения сказанного обратимся к мнению эксперта. Г. Саттер в своей книге «Решение сложных задач на C++» написал следующее:



«*... Однако я начну с одного, пожалуй, самого полезного замечания: зачем писать класс* `Complex`*, если таковой уже имеется в стандартной библиотеке? Помимо всего прочего, стандартный класс разработан с учетом многолетнего опыта лучших программистов и не болен «детскими болезнями крутизны в программизме». Будьте скромнее и воспользуйтесь трудами ваших предшественников. Вместо написания собственного кода повторно используйте имеющийся, в особенности код стандартной библиотеки. Это быстрее, проще и безопаснее. Вероятно, наилучший способ решить все проблемы – это не использовать класс* `Complex` *вообще, заменив его стандартным классом* `std::complex`» [5].

Отметим, что главная идея организации STL – это использование шаблонов классов, которые применимы почти со всеми типами данных. (Поэтому, когда речь идет о шаблонных классах стандартной библиотеки, лучше использовать указание на пространство имен `std`, например `std::complex`, чтобы отделять их от пользовательских классов.)

Может возникнуть резонный вопрос: а почему мы сразу не использовали эту библиотеку? Ответ предельно прост. Прежде чем ее использовать, нужно понять, как она устроена и работает, а для этого необходимо сначала освоить синтаксис языка C++ и осознать, что такое ООП. Тогда ваши классы будут получаться более «правильными», а программы – легко читаемыми.

Далее приведены *краткие сведения* о работе с этой библиотекой. Поэтому заинтересованному читателю в дальнейшем целесообразно более подробно ознакомиться с ее функциональными возможностями и убедиться в том, насколько она обширна.

Компоненты библиотеки

Основу STL составляют три компонента: *контейнеры*, *итераторы* и *алгоритмы*. Назначение первых – это управление коллекциями однотипных элементов (объектов), вторые используются для направленного перебора элементов контейнеров, а третьи реализуют средства манипуляции с ними.

Контейнеры: последовательные

Контейнеры делятся на последовательные, ассоциативные и адаптеры. Последовательные контейнеры, судя по их общему названию, предназначены для последовательного хранения элементов. Основными последовательными контейнерами являются: `std::vector<Type>`, `std::deque<Type>`, `std::list<Type>`, `std::array<Type>`, `std::forward_list<Type>`, `std::basic_string<Type>`. Здесь `Type` – требуемый тип данных. Также есть более новые `std::unordered_map<Type>`, `std::span<Type>` и `std::string_view<Type>`. (Здесь и далее контейнеры указаны с минимальными наборами параметров.)

Кратко рассмотрим особенности работы с некоторыми из них. Начнем с `std::vector<Type>` (вектор). При этом используем лишь некоторые его методы, позволяющие выполнять операции над ним.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  void print(const vector<int>& vec){
5      for (auto val : vec)
6          cout << val << ' ';
7      cout << endl;
8  }
9  int main(){
10     vector<int> vecA; // пустой
11     vector<int> vecB(5); // 0 0 0 0 0
12     vector<int> vecC(5,9); // 9 9 9 9 9
13     cout << "max size=" << vecA.max_size() << endl; //
   max size=2305843009213693951
14     cout << "max size="<< vecC.max_size() << endl; //
   max size=2305843009213693951
15     for (auto i = 0; i < 5; i++){
16             vecA.push_back(rand()/1e8);
```

```
17          vecB.push_back(i); // добавление в конец вектора
значения i
18      }
19      print(vecA);
20      cout << "before swap:" << endl;
21      cout << "vecB ";
22      print(vecB);
23      vecC.resize(2); // задание нового размера - 9 9
24      vecC.insert(vecC.begin()+1, 1313); // добавление
элемента - 9 1313 9
25      vecC.pop_back(); // удаление последнего элемента - 9
1313
26      cout<< "vecC ";
27      print(vecC);
28      vecC.swap(vecB); // обмен значениями между двумя
векторами
29      cout << "after swap:" << endl;
30      cout << "vecB ";
31      print(vecB); // 9 1313
32      cout << "vecC ";
33      print(vecC); // 0 0 0 0 0 0 1 2 3 4
34      vecC.erase(vecC.end()-1); // удаление по позиции
35      print(vecC); // 0 0 0 0 0 0 1 2 3
36      vecC.erase(vecC.begin()+5); // удаление по позиции
37      print(vecC); // 0 0 0 0 0 1 2 3
38      vecC.erase(vecC.begin(), vecC.begin()+5); //
удаление диапазона элементов
39      print(vecC); // 1 2 3
40      return 0;
41  }
```

Здесь понимание использованного синтаксиса уже не должно вызывать больших сложностей. Так, сначала с использованием конструкторов создаются объекты шаблонного класса `std::vector<int>`, а затем для этих объектов вызываются требуемые методы, т. е. все то, что было проделано при разработке пользовательских классов и последующей работе с их объектами. При этом для вывода на печать элементов вектора использована внешняя функция, использующая цикл `for` для диапазона.

Одним из главных достоинств контейнера `std::vector<Type>`, отличающего его от динамически созданного массива с помощью ключевого слова `new`, является то, что память под его объекты автоматически выделяется и, что не менее важно, очищается. Теперь рассмотрим, как можно обращаться к элементам этого контейнера с помощью перегруженного `operator[]` и метода `at()`, а также аннигилировать контейнер (очистить его содержимое).

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main(){
5      vector<int> vecA={3, 4, 5}; // инициализация
6      vecA[1]=13; // обращение к элементу
7      vecA.resize(4); // изменение размера
8      vecA[3]=14; // добавление нового элемента
9      int &val = vecA.at(1); // at() - возвращает ссылку
   на элемент в положении, указанном в скобках
10     val = 109;
11     for(auto i = 0; i < vecA.size(); i++) // size()-
   определение размера контейнера
12         cout << vecA.at(i) << " "; // или cout <<
   vecA[i] << " ";
13     cout <<endl<< "before: size vector= " <<vecA.size();
14     vecA.clear(); // аннигиляция контейнера
15     cout <<endl<< "after: size vector= " <<vecA.size();
16     return 0;
17      }
```

При рассмотрении функционала контейнера `std::vector<Type>` может возникнуть вопрос: а матрицы тоже можно создавать? Да, конечно. При этом полученные массивы в общем случае сложно назвать матрицей, поскольку их вид может оказаться не вполне привычным, как в следующем примере.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main(){
5      vector<vector<int>> vecA{{1,2}, {5,6}, {9,10}};
6      vecA[1].push_back(13);
7      vecA.resize(4);
```

```
    vecA[3].push_back(-4);
    for (auto row : vecA){
        for (auto val : row)
            cout << val << " ";
        cout << endl;
    }
    return 0;
}
```

Класс `std::deque<Type>` (двухсторонняя очередь) подобен `std::vector<Type>`, но обеспечивает более эффективную вставку и удаление элементов, особенно при работе с начальной частью контейнера. Так, например, в отличие от `std::vector<Type>` у `std::deque<Type>` есть методы для добавления элемента в начало контейнера – `push_front()` и его удаления – `pop_front()`.

```
#include <iostream>
#include <deque>
using namespace std;
int main(){
    deque <int> deqA{1,2,3}; // 1 2 3
    deqA.push_back(4); // 1 2 3 4
    deqA.push_back(5); // 1 2 3 4 5
    deqA.pop_front(); // 2 3 4 5
    deqA.push_front(0); // 0 2 3 4 5
    for(auto val : deqA)
        cout << val << " ";
    cout << endl;
    return 0;
}
```

Еще одним последовательным контейнером является шаблонный класс `std::list<Type>` (список), который предоставляет доступ только к началу и концу контейнера. Тогда для того чтобы обратиться к элементу в середине контейнера, его надо последовательно перебрать. Поэтому в классе реализованы методы сортировки. Кроме того, этот контейнер отличается линейным временем доступа к его элементам.

```
#include <iostream>
#include <list>
using namespace std;
bool check (const int& val) {
```

```
5       return (val % 2);
6   }
7   int main(){
8       list <int> listA{1,2,3,4,1,2,3,4};
9       listA.push_front(-1);// -1 1 2 3 4 1 2 3 4
10      listA.push_back(5); // -1 1 2 3 4 1 2 3 4 5
11      listA.reverse(); // 5 4 3 2 1 4 3 2 1 -1
12      listA.sort(); // -1 1 1 2 2 3 3 4 4 5
13      listA.remove(2); // -1 1 1 3 3 4 4 5
14      listA.remove_if(check); // 4 4
15      listA.unique(); // проверка на уникальность - 4
16      for(auto val : listA)
17          cout << val << " ";
18      cout << endl;
19      return 0;
20  }
```

Шаблонный класс `std::array<Type>` (массив) позволяет создать «стандартный» массив однотипных элементов, а имеющиеся в нем методы – легко обращаться к его элементам. При этом важно, что длина контейнера должна быть известна во время компиляции.

```
1   #include <iostream>
2   #include <array>
3   using namespace std;
4   int main(){
5       array <int,5> arrA{1,2,3,4,5};
6       array <int,5> arrB;
7       arrB.fill(10); // 10 10 10 10 10
8       for(auto val : arrB)
9           cout << val << " ";
10      cout << endl;
11      cout << arrA.front() << endl; // 1
12      cout << arrA[1] << endl; // 2
13      return 0;
14  }
```

Контейнеры: ассоциативные

Ассоциативные контейнеры отличаются от последовательных тем, что они дополнительно сортируют свои элементы, поэтому иногда их еще называют отсортированные ассоциативные контейнеры. Они организуют парное хранение и извлечение элементов. Каждая пара содержит ключ сортировки и значение. При этом значение ключа для каждой такой пары должно быть уникальным и впоследствии не может быть изменено. Библиотека содержит 4 таких контейнера: `std::set<Key, Type>`, `std::multiset<Key, Type>`, `map<Key, Type>`, `multimap<Key, Type>`. Здесь `Key` – ключ сортировки. При этом доступ к элементам контейнеров осуществляется через указатели.

Элементы контейнера шаблонного класса `std::set<Key, Type>` являются уникальными и служат ключевыми значениями, по которым данные автоматически упорядочены. Задание ключа в данном контейнере не является обязательным. При этом важно, что значения контейнера не могут быть изменены. Поэтому при необходимости их следует удалить и поместить заново.

```
1  #include <iostream>
2  #include <set>
3  using namespace std;
4  using zooSet = set<int>;
5  int main(){
6      zooSet setA{11,21,3};
7      zooSet setB=setA;
8      setA.insert(1001); // добавление элемента со
   значением 1001 в контейнер
9      setA.insert({0, -11, -3, -5}); // добавление
   элементов
10     setA.erase(11); // удаление элемента со значением 11
   из контейнера
11     setB.swap(setA);
12     for (auto i = setA.begin(); i != setA.end(); i++)
13         cout << ' ' << *i;
14     cout << endl;
15     for (auto i = setB.begin(); i != setB.end(); i++)
16         cout << ' ' << *i;
17     return 0;
18 }
```

Рассмотрим пример использования контейнера `map<Key, Type>`.

```
1  #include <iostream>
2  #include <map>
3  using namespace std;
4  using zooMap = map<string, int>;
5  int main(){
6      zooMap mapA= {
7          { "Tiger", 3}, { "Wolf", 4},{ "Camel", 2}
8      };
9      mapA.insert ( pair<string,int>("Elephant", 2) );
10     mapA.insert ( pair<string,int>("Ant", 5000000) );
11     mapA.insert ( pair<string,int>("Camel", 1) );
12     cout << "The our zoo contains ("<< mapA.size() <<
   ") :" << endl;
13     for (auto i = mapA.begin(); i != mapA.end(); i ++)
14         cout << i->first << "- " << i->second << endl;
15     cout << "Is there an elephant in the zoo? - " <<
   boolalpha << mapA.contains("Elephant") << endl;
16     cout << "Is there an kangaroo in the zoo? - " <<
   mapA.contains("Kangaroo") << endl;
17     return 0;
18 }
```

Здесь производится сортированный по ключу вывод значений элементов контейнера, имеющих тип данных `pair` (п. 3.3.7). Кроме того, следует обратить внимание на строку 11, в которой происходит попытка добавления в контейнер элемента с уже существующим ключом. В результате этот элемент не будет добавлен. При этом ошибки компиляции не будет. Ситуацию можно исправить за счет использования контейнера `std::multiset<Key, Type>`, в котором ограничения на уникальность ключа нет. Важно, что в этом случае используется тот же заголовочный файл (`<map>`).

```
1  #include <iostream>
2  #include <map>
3  using namespace std;
4  using zooMap = multimap<string, int>;
5  int main(){
6      zooMap mapA= {
7          { "Wolf", 4},{ "Camel", 2}
8      };
```

```
9       mapA.insert ( pair<string,int>("Camel", 1) ); //
    добавление верблюда
10      for (auto i = mapA.begin(); i != mapA.end(); i ++)
11          cout << i->first << "- " << i->second << endl;
12      return 0;
13       }
```

Контейнеры: адаптеры

Адаптивные контейнеры, как следует из их названия, позволяют адаптировать их под конкретные задачи. Используется 3 вида таких контейнеров: `std::queue<Type>` (работает по принципу «первым поступил – первым обслужен»), `std::stack<Type>` («последним поступил – первым обслужен») и `std::prority_queue<Type>` («максимальный элемент всегда первый»).

Итераторы

Как уже было сказано выше, итераторы реализуют унифицированный перебор элементов контейнера. В библиотеке используется 5 их основных видов: итераторы чтения (вывода) и записи (ввода), однонаправленные и двунаправленные итераторы, а также итераторы произвольного доступа.

Итераторы *– это объекты, предназначенные для перебора и предоставления доступа к отдельным контейнерным элементам.*

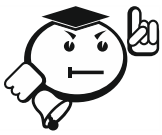
Итераторы могут быть использованы через методы классов контейнеров `end()` и `begin()`, перегруженные операторы и глобальные функции `end(Conteiner)` и `begin(Conteiner)`. Отметим, что в рассмотренных выше примерах использования контейнеров некоторые из итераторов уже были задействованы, например, в последнем (строки 10 и 11), хотя этого явно не видно, т. к. в цикле `for` (строке 10) использовано автоматическое определение типа данных с помощью ключевого слова `auto`.

Отдельно отметим, что в рассмотренных примерах использования контейнеров `std::set<Key, Type>`, `std::map<Key, Type>` и `std::multimap<Key, Type>` при выводе их значений для наглядности использован «стандартный» цикл `for`, а остальных контейнеров – цикл `for` для диапазона (п. 2.3.2). Следующий пример наглядно демонстрирует, какие следует сделать изменения при смене одного цикла `for` другим и неиспользовании ключевого слова `auto`.

```
1  #include <iostream>
2  #include <set>
3  #include <map>
4  using namespace std;
5  int main(){
6  // set
7      set<int> setA = {11,21,3};
8      cout << " -- set using -- " << endl;
9      for (auto i = setA.begin(); i != setA.end(); i++) //
   методы
10         cout << *i << ' ' ;
11     cout << endl;
12     for (auto i = begin(setA); i != end(setA); i++) //
   глобальные функции
13         cout << *i << ' ' ;
14     cout << endl;
15     for (set<int>::iterator i = setA.begin(); i !=
   setA.end(); i++) // явное указание итератора
16         cout << *i << ' ' ;
17     cout << endl;
18     for (auto i : setA)
19         cout << i << " ";
20 // map, multimap
21     map<string, int> mapA = {{"Wolf", 4},{"Camel", 2}};
22     cout << endl << " -- map using -- " << endl;
23     for (auto i = mapA.begin(); i != mapA.end(); i ++)
   // методы
24         cout << i-> first << "- " << i-> second << endl;
25     for (auto i = begin(mapA); i != end(mapA); i ++) //
   глобальные функции
26         cout << i-> first << "- " << i-> second << endl;
27     for (map<string,int>::iterator i = mapA.begin(); i
   != mapA.end(); i ++) // явное указание итератора
28         cout << i-> first << "- " << i-> second << endl;
29     for (auto &i : mapA)
30         cout << i.first << "- " << i.second << endl;
31     set<int>::iterator it1;
32     set<int>::iterator it2;
```

```
33      return 0;
34  }
```

Здесь следует обратить внимание на требуемую смену операторов разыменования указателей при использовании ключевого слова `auto` вместо явного указания итератора (`::iterator`). (Для других контейнеров соответствующие модификации делаются аналогично.)



При работе с контейнером предпочтительно (повышает читаемость кода) использовать ключевое слово `auto` и цикл `for` для диапазона вместо явного использования итераторов и стандартного цикла `for`.

Алгоритмы

После того как были рассмотрены контейнеры и итераторы, можно перейти к «самому интересному», а именно к алгоритмам. Важно помнить, что алгоритмы работают с контейнерами только через итераторы. Они находятся в заголовочных файлах `<algorithm>` и `<numeric>`. Ниже кратко рассмотрим некоторые из них – материал носит лишь ознакомительный характер.

Следующий пример демонстрирует работу контейнера `vector<Type>` и итератора в совокупности с некоторыми алгоритмами сортировки и поиска. При этом, во избежание «перегрузки материала», алгоритмы использованы только с обязательными параметрами, т. е. указание дополнительных параметров не производилось.

Для наглядности некоторые алгоритмы сортировки закомментированы. Читатель может самостоятельно поэкспериментировать с ними. Для этого нужно построчно раскомментировать требуемую строку с алгоритмом и закомментировать остальные, т. к. в противном случае произойдет последовательная и бессмысленная сортировка уже отсортированного контейнера.

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>
4   using namespace std;
5   int main( ){
6       vector<int> vec={325,457,457,886,112,792,457};
7       cout << "--before--" << endl;
```

```
8       for (auto i : vec) cout<< i << " " ;
9   // варианты сортировки
10      sort(vec.begin(), vec.end()); // алгоритм быстрой
    сортировки
11  //    sort(vec.begin()+1, vec.end()-1); // алгоритм
    быстрой сортировки (от 2-го элемента до предпоследнего)
12  //    sort(vec.begin()+1, vec.begin()+3); // алгоритм
    быстрой сортировки (со 2-го по 4-й)
13  //    stable_sort(vec.begin(), vec.end()); // алгоритм
    сортировки слиянием
14  //    partial_sort(vec.begin(), vec.end()-5, vec.end());
    // алгоритм частичной сортировки
15      cout << endl << "--after--"<<endl;
16      for (auto i : vec) cout<< i << " " ;
17      pair<vector<int>::iterator, vector<int>::iterator>
    resRange;
18      resRange = equal_range(vec.begin(), vec.end(), 457);
    // алгоритм поиска диапазона
19      cout << endl << "The vector contains " <<
    resRange.second - resRange.first << " elements with the
    value of 457" << endl;
20      for(auto i = begin(vec); i < end(vec); i++) {
21        if(i == resRange.first)
22          cout << "{ ";
23        if(i == resRange.second)
24          cout << "} ";
25        cout << *i << " " ;
26    }
27    return 0;
28  }
```

Алгоритм `sort(start, stop)` предназначен для быстрой сортировки (рис. Г.1). Однако он не гарантирует, что близкие по значениям элементы будут корректно упорядочены. Алгоритм `stable_sort(start, stop)`, наоборот, гарантирует это (в примере не показано).

```
--before--
325 457 457 886 112 792 457
--after--
112 325 457 457 457 792 886
```

Рис. Г.1 – Результат работы алгоритма
`sort(vec.begin(), vec.end())`

Эти алгоритмы позволяют выполнять частичную сортировку по принципу «от (`start`) и до (`stop`)». Для этого используются сдвиги итератора. При этом элементы, не попавшие в указанный диапазон, остаются на прежних местах в контейнере.

Алгоритм `partial_sort(start, middle, stop)` сортирует элементы от начала (`start`) и до конца (`stop`) так, чтобы элементы до середины (`middle`) находились в порядке возрастания и тем самым оказывались наименьшими элементами контейнера (рис. Г.2).

```
--before--
325 457 457 886 112 792 457
--after--
112 325 457 886 457 792 457
```

Рис. Г.2 – Результат работы алгоритма
`partial_sort(vec.begin(), vec.end()-5, vec.end())`

Задача алгоритма поиска `equal_range(start, stop, value)` состоит в поиске в контейнере требуемых значений (`value`). Выходными параметрами являются итераторы начала и конца диапазона, содержащего это значение (рис. Г.3).

```
--before--
325 457 457 886 112 792 457
--after--
112 325 457 457 457 792 886
The vector contains 3 elements with the value of 457
112 325 { 457 457 457 } 792 886
```

Рис. Г.3 – Результат работы (для отсортированного контейнера) алгоритма
`equal_range(vec.begin(), vec.end(), 457)`

Когда в контейнере содержится только один искомый элемент, итераторы совпадают. Если контейнер не будет предварительно отсортирован, то результат

работы алгоритма может оказаться не вполне ожидаемым. Пример этого (для предварительно увеличенного с семи до десяти элементов исходного контейнера) показан на рисунке Г.4. (Поэтому для получения требуемого решения рекомендуется предварительно сортировать контейнеры.)

```
--before--
325 457 457 886 112 792 457 457 111 567
--after--
325 457 457 886 112 792 457 457 111 567
The vector contains 4 elements with the value of 457
325 { 457 457 886 112 } 792 457 457 111 567
```

Рис. Г.4 – Результат работы для несортированного контейнера алгоритма `equal_range(vec.begin(), vec.end(), 457)`

Рассмотрим еще несколько алгоритмов поиска. Первый – это алгоритм `find()`.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main( ){
   vector<int> vec={325,457,457,886,112,792,457};
   cout << "--input vector-- "<< endl;
   for (auto i : vec) cout<< i << " " ;
   cout << endl<<"search result for element 007"<< endl;
   vector<int>::iterator res;
   res = find(vec.begin(), vec.end(), 007);
   cout << "value=" << *res << ", index=" << res -
vec.begin() <<endl;
   return 0;
}
```

Здесь алгоритм `find(start, stop, value)` возвращает итератор, указывающий на позицию первого вхождения элемента с требуемым значением (`value`) в анализируемом контейнере (рис. Г.5, *а*). Если ни одного элемента с требуемым значением не найдено, то возвращаемый итератор указывает на последний элемент контейнера (рис. Г.5, *б*).

```
--input vector--
325 457 457 886 112 792 457
search result for element 457
value=457, index=1
```

а)

```
--input vector--
325 457 457 886 112 792 457
search result for element 007
value=0, index=7
```

б)

Рисунок Г.5 – Результат работы алгоритма `find()`: `find(vec.begin(), vec.end(), 457)` (*а*) и `find(vec.begin(), vec.end(), 007)` (*б*)

Рассмотрим использование алгоритма `find_end()`, предназначенного для поиска (под)последовательности элементов контейнера в (под)последовательности элементов другого контейнера. Его возвращаемым значением является итератор, указывающий на первый элемент найденной (под)последовательности в проверяемой (под)последовательности.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5  int main( ){
6     vector<int> vec = {113,117,112,325,457,457,886,112,
   792,457,1};
7     vector<int> vec2={325,457,457,886,112,792,457};
8     cout << "--input vectors-- "<< endl;
9     for (auto i : vec) cout<< i << " " ;
10    cout << endl;
11    for (auto i : vec2) cout<< i << " " ;
12    vector<int>::iterator res;
13    res = find_end(vec.begin(), vec.end(),
   vec2.begin()+2, vec2.begin()+5);
14    cout << endl << "value=" << *res << ", index=" <<
   res - vec.begin() <<endl;
15    return 0;
16 }
```

Здесь происходит поиск подпоследовательности (457, 886, 112, 792) из контейнера vec2 во всей последовательности элементов контейнера vec (рис. Г.6).

```
--input vectors--
113 117 112 325 457 457 886 112 792 457
1
325 457 457 886 112 792 457
value=457, index=5
```

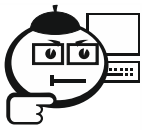
Рис. Г.6 – Результат работы алгоритма `find_end(vec.begin(), vec.end(), vec2.begin()+2, vec2.begin()+5)`

Если совпадений не найдено, то итератор указывает на последний элемент проверяемой (под)последовательности (рис. Г.7).

```
--input vectors--
113 117 112 1 2 4 5 3 7 8 1
325 457 457 886 112 792 457
value=8, index=9
```

Рис. Г.7 – Результат работы алгоритма `find_end(vec.begin(), vec.end()-2, vec2.begin()+2, vec2.begin()+5)`

Подробнее о стандартной библиотеке шаблонов можно узнать из рекомендуемой литературы и онлайн-справочника корпорации Microsoft, доступного по следующей ссылке.



Файлы заголовков стандартной библиотеки C++

Приложение Д
(информационное)
Инструкция по созданию проекта в Microsoft Visual Studio

Я, товарищ директор, того, на этого…

М/ф «Следствие ведут Колобки»

Для создания проекта в Microsoft Visual Studio (MVS) нужно:

1. После запуска MVS открыть меню `Файл`, затем выбрать `Создать` → `Проект` (рис. Д.1). Результат показан на рисунке Д.2.

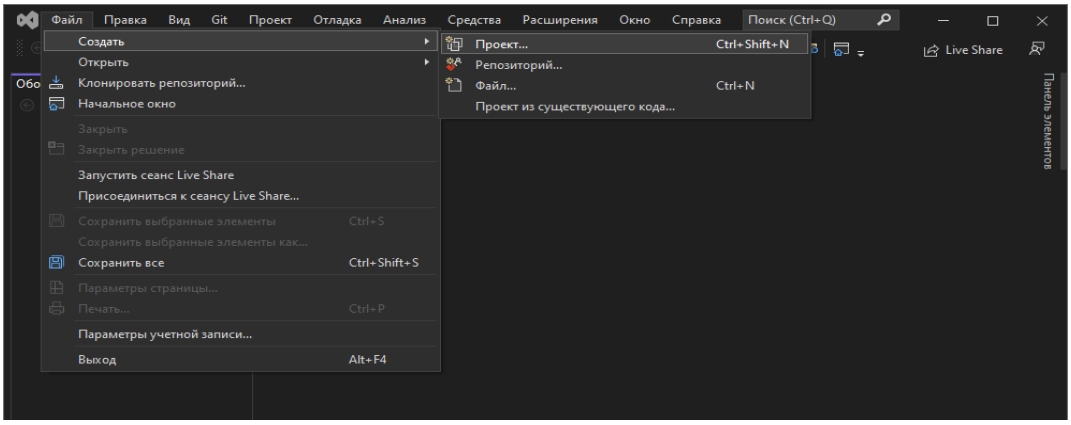


Рис. Д.1 – MVS: вид меню для создания нового проекта

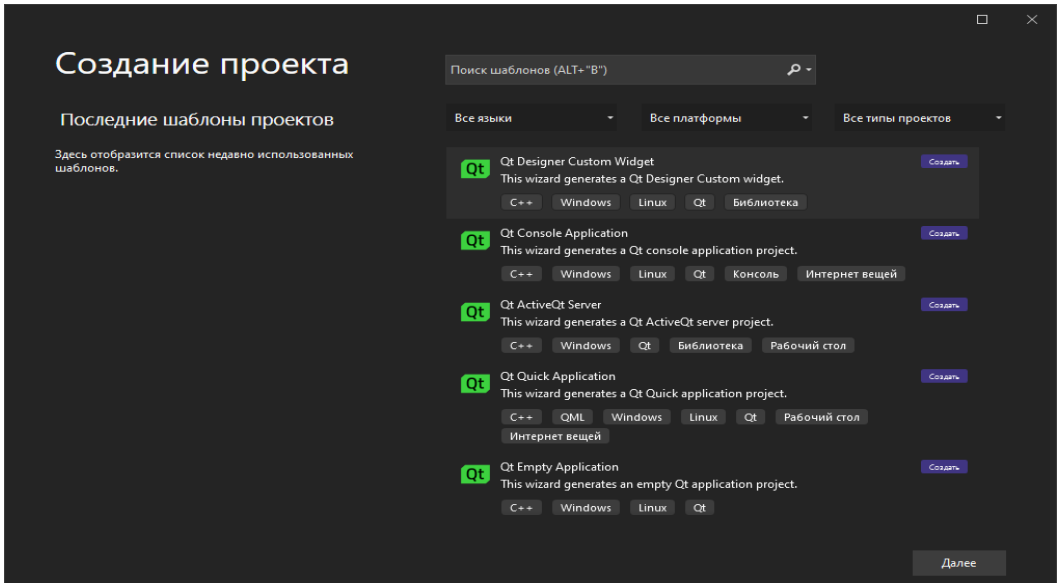


Рис. Д.2 – MVS: вид окна `Создание проекта`

2. Настроить фильтры проекта, для этого выбрать `C++`, `Windows`, `Консоль` и пункт `Консольное приложение` (рис. Д.3), после чего нажать `Далее`.

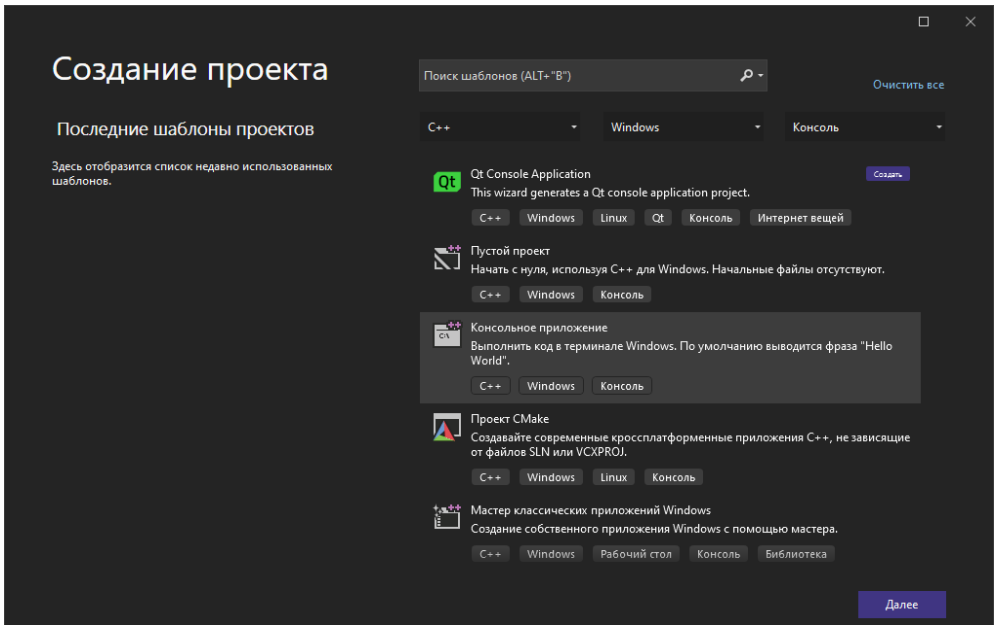


Рис. Д.3 – MVS: вид окна шаблона настройки консольных проектов на языке C++ для Windows

3. В открывшемся окне ввести требуемые имена проекта и решения (в MVS проекты помещаются в решения), а также путь к директории для их хранения, после чего нажать `Создать` (рис. Д.4).

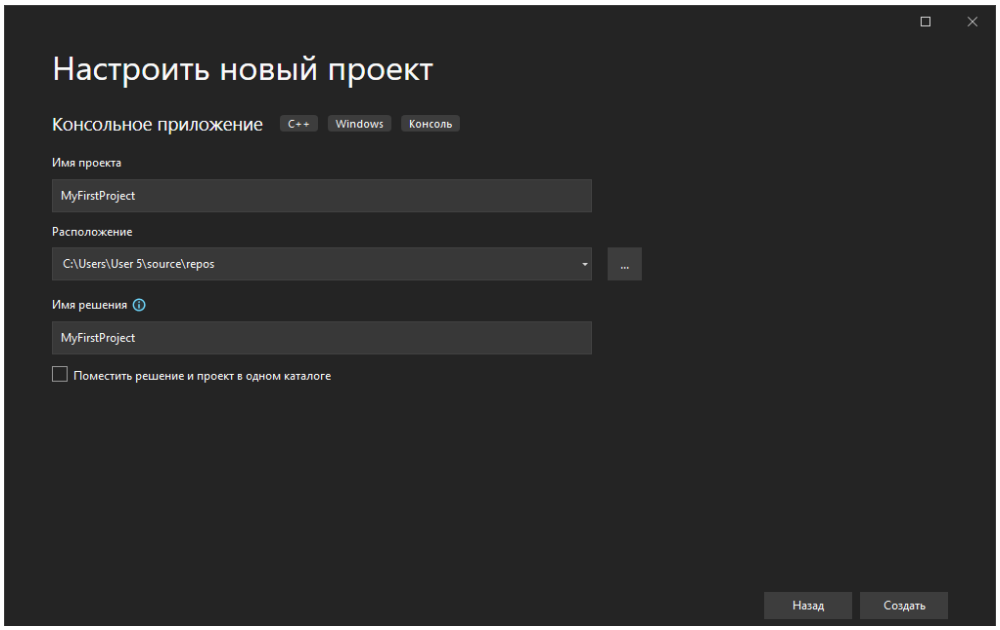


Рис. Д.4 – MVS: вид окна с настройками новых решения и проекта

В результате будет создано решение с включенным в него проектом консольного приложения, содержащим исполняемый файл, в который помещена функция `main()` (рис. Д.5).

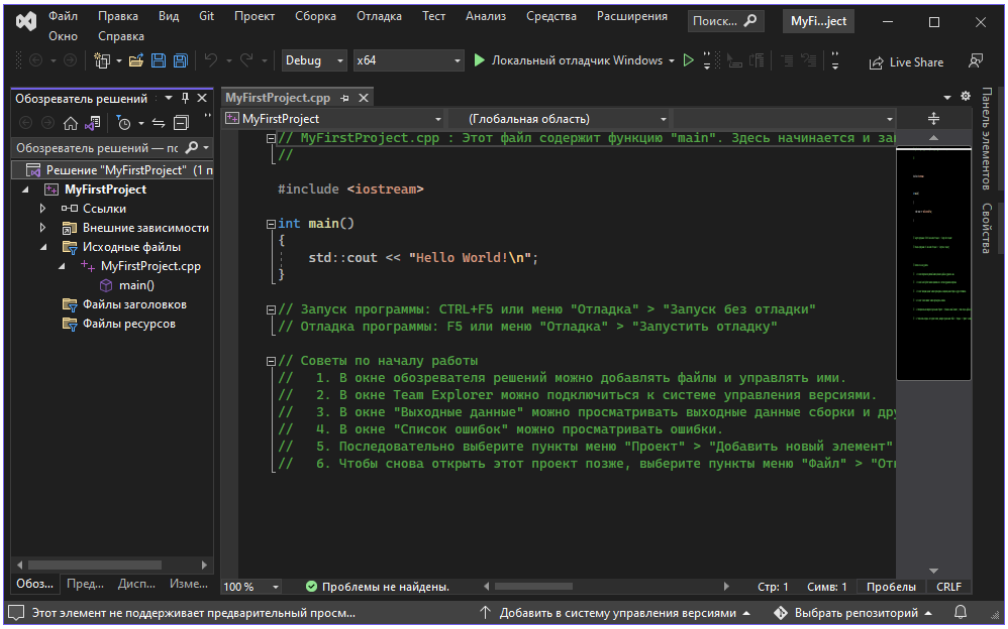


Рис. Д.5 – MVS: вид дерева решения и кода из исполняемого файла

4. Добавить (при необходимости) дополнительные файлы в проект, для чего нажать правой кнопкой мыши по папке `Исходные файлы` в дереве решения, в которую будет добавлен создаваемый файл. Затем перейти в пункт меню `Добавить → Создать элемент` (рис. Д.6).

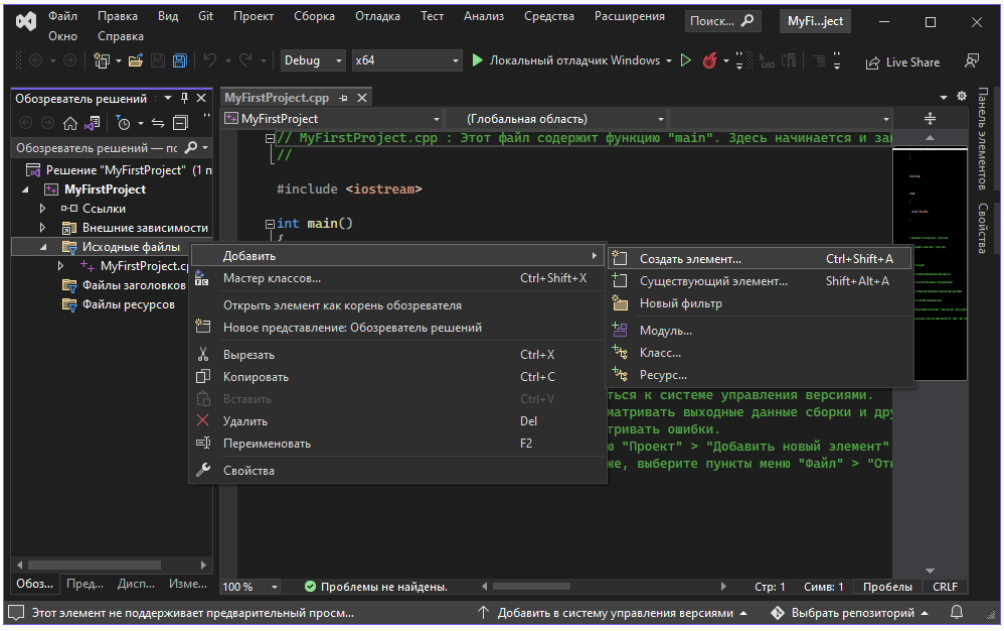


Рис. Д.6 – MVS: добавление файла в созданный ранее проект

5. Во всплывающем окне выбрать тип файла, задать его имя и нажать `Добавить` (рис. Д.7).

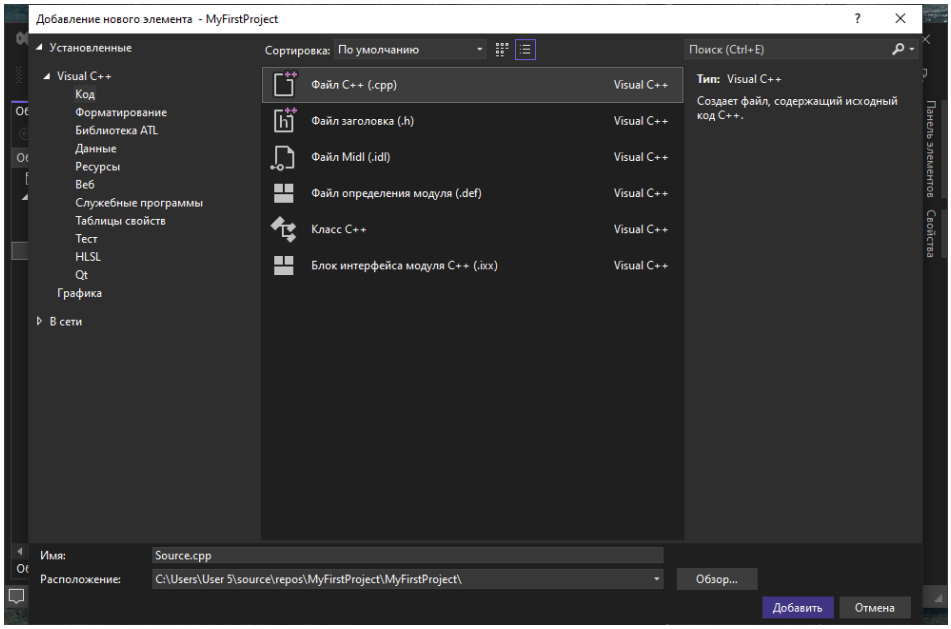


Рис. Д.7 – MVS: вид окна `Добавление нового элемента`

Рассмотрим пример добавления в проект файла, содержащего объявление класса `MyClass`. Для этого, согласно п. 5, в окне `Добавление нового элемента` нужно выбрать тип файла `Класс C++` (или `Файл C++`), после чего в открывшемся окне ввести имя класса и нажать `Добавить`. Полученный результат представлен на рисунке Д.8.

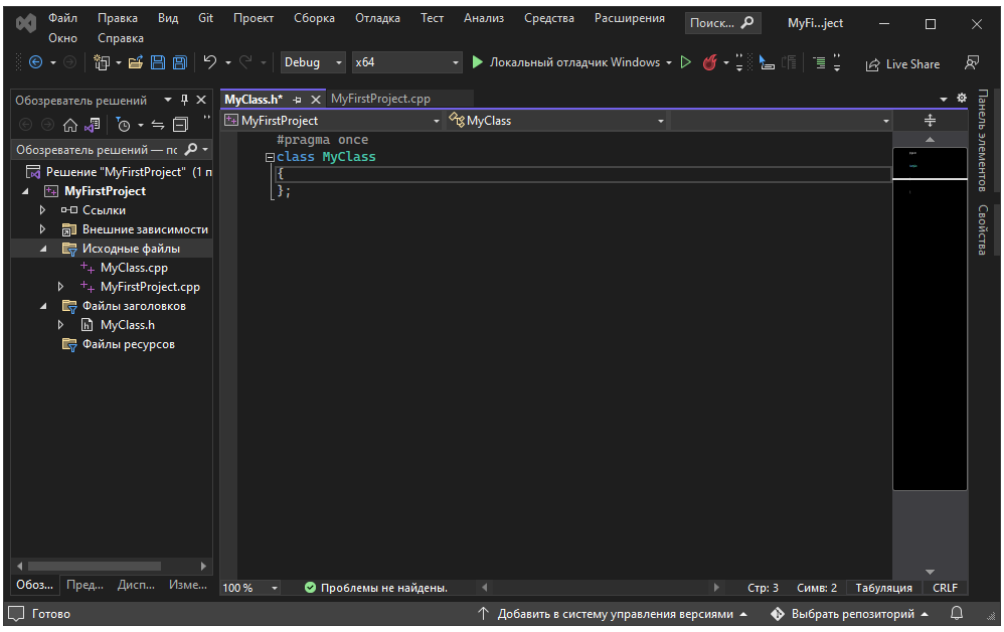


Рис. Д.8 – MVS: вид проекта после добавления файла с классом `MyClass`

Приложение Е
(информационное)
Инструкция по созданию проекта в Qt Creator

В принципе – все просто: нужно собрать команду, стать авторитетом и показать фокус.

Х/ф «Каникулы строгого режима»

Для создания проекта в Qt Creator (Qt) нужно:

1. В главном окне Qt нажать `Create Project` (рис. Е.1).

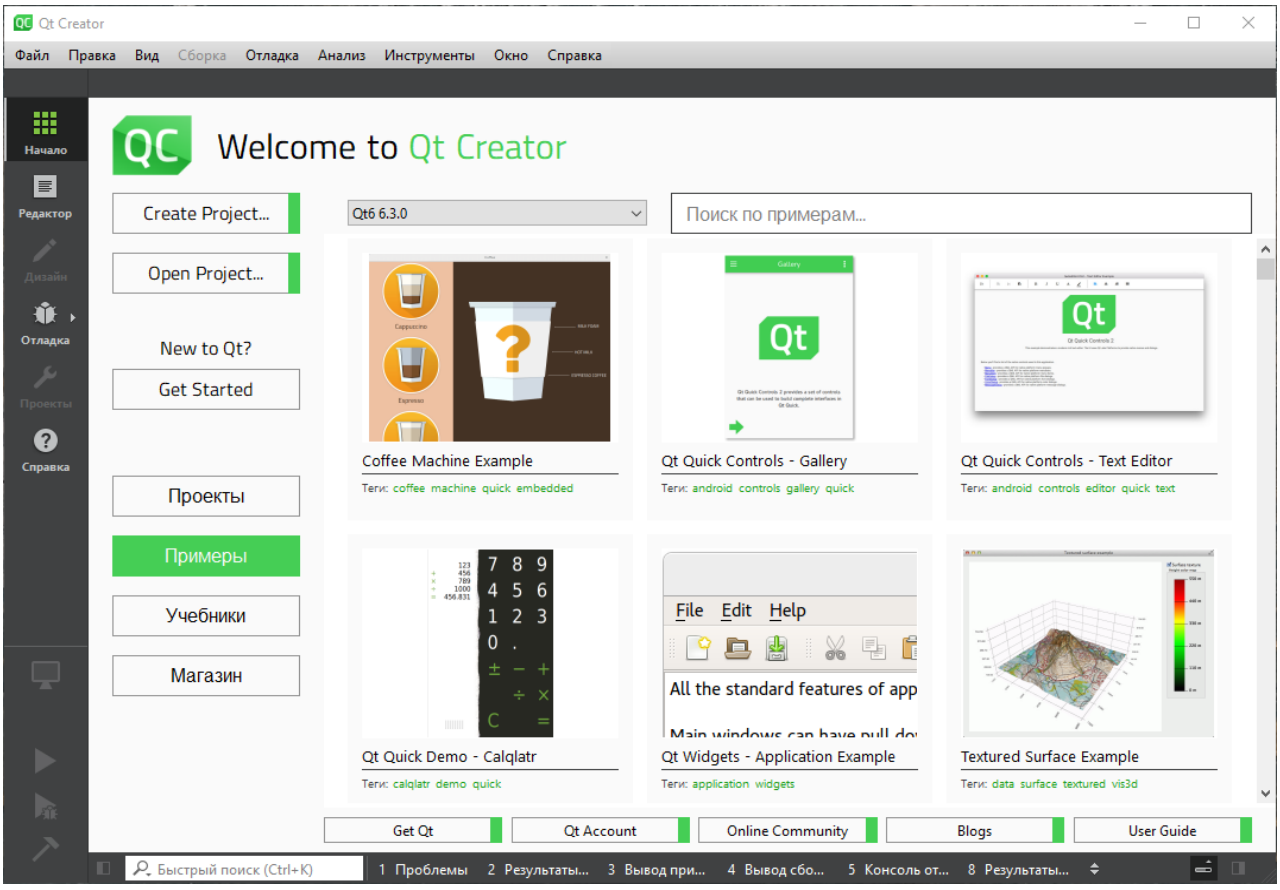


Рис. Е.1 – Qt: вид главного окна при запуске

2. Выбрать (в открывшемся окне) шаблон `Проект без Qt` → `Приложение на языке C++` и нажать `Choose` (рис. Е.2).

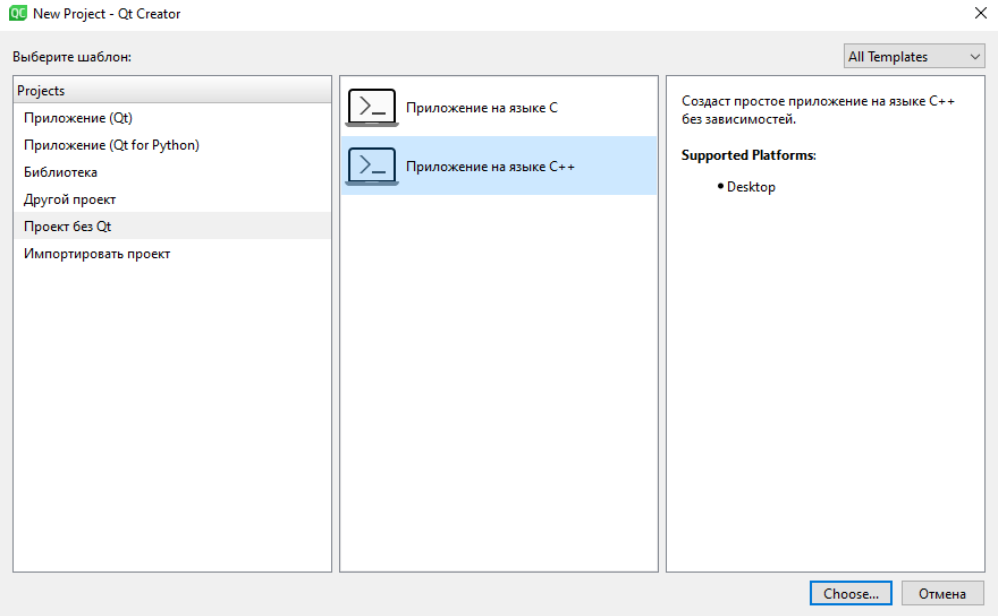


Рис. Е.2 – Qt: вид окна с выбором типа нового проекта

3. Указать требуемое имя проекта и директорию для его хранения, после чего нажать `Далее` (рис. Е.3).

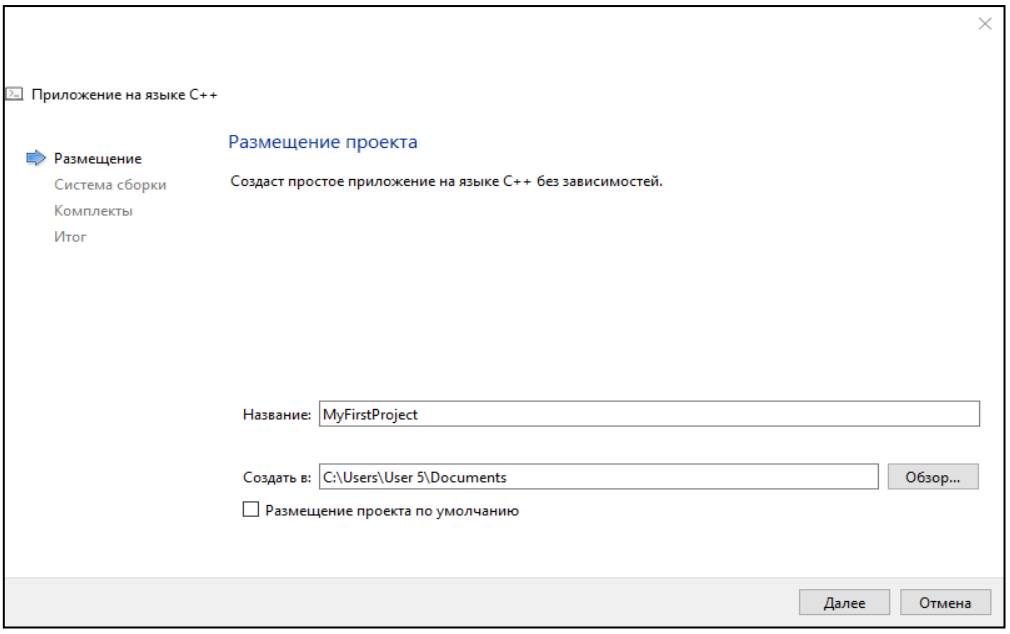


Рис. Е.3 – Qt: вид окна для указания имени и директории проекта

4. Выбрать систему сборки (Стандартная система сборки – `qmake`. Для создания проектов без Qt рекомендуется использовать `CMake`), после чего нажать `Далее` (рис. Е.4). Затем, оставив имя основного класса по умолчанию, нажать `Далее`.

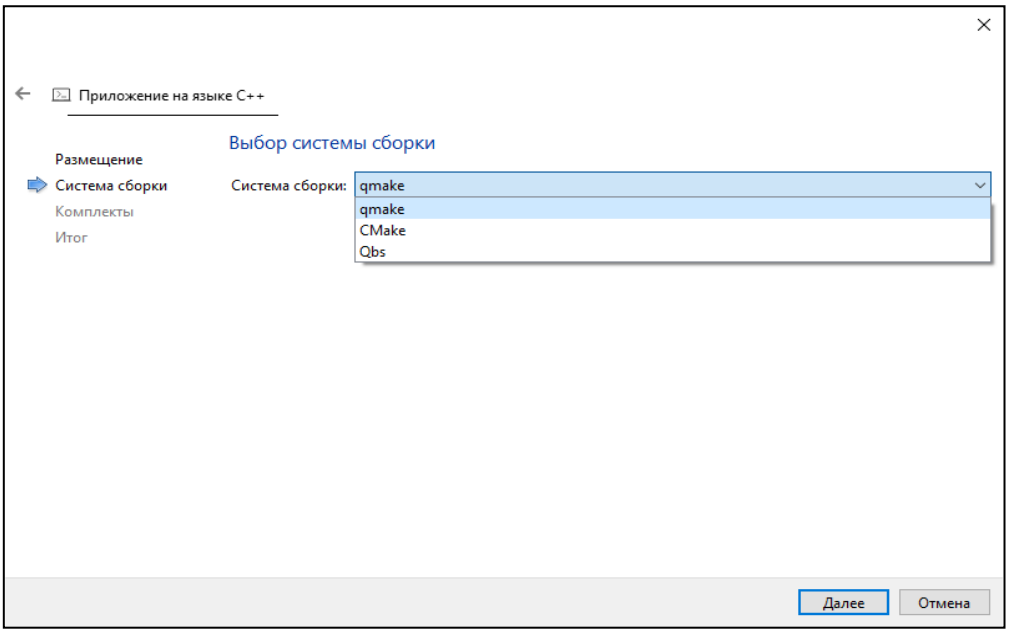


Рис. E.4 – Qt: вид окна для выбора системы сборки

5. Выбрать требуемый комплект средств (Qt, компилятор и разрядность) и нажать `Далее` (рис. E.5). (При первом запуске комплект, как правило, один.)

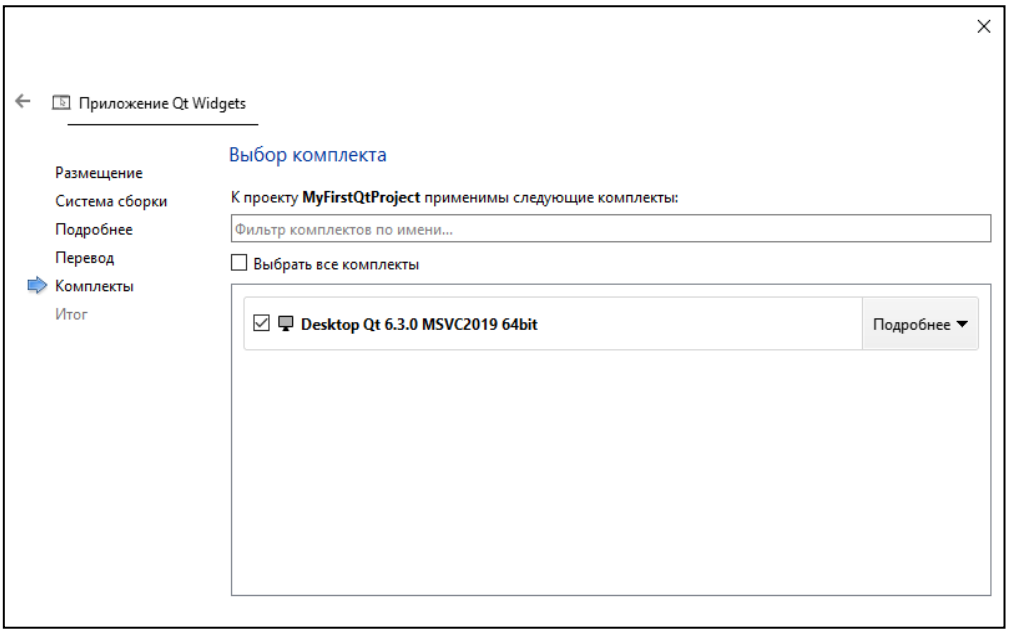


Рис. E.5 – Qt: вид окна для выбора комплекта средств

6. Указать `Нет` в поле контроля версий и нажать `Завершить` (рис. E.6).

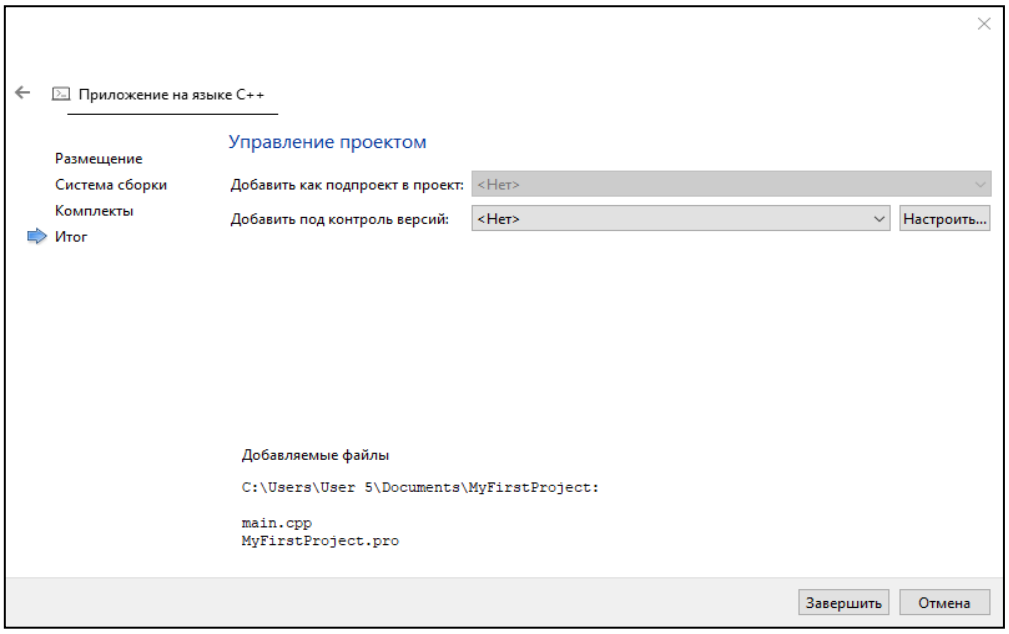


Рис. Е.6 – Qt: вид окна для настройки системы контроля версий

В результате будет создан требуемый проект, содержащий исполняемый (исходный) файл, в который помещена функция `main()`. Для запуска проекта используется зеленая кнопка `Запустить`, находящаяся в левом нижнем углу окна проекта. Результат выполнения программы отображается в нижней части окна проекта (рис. Е.7). Процесс добавления новых файлов в проект аналогичен, описанному для Microsoft Visual Studio (прил. Д).

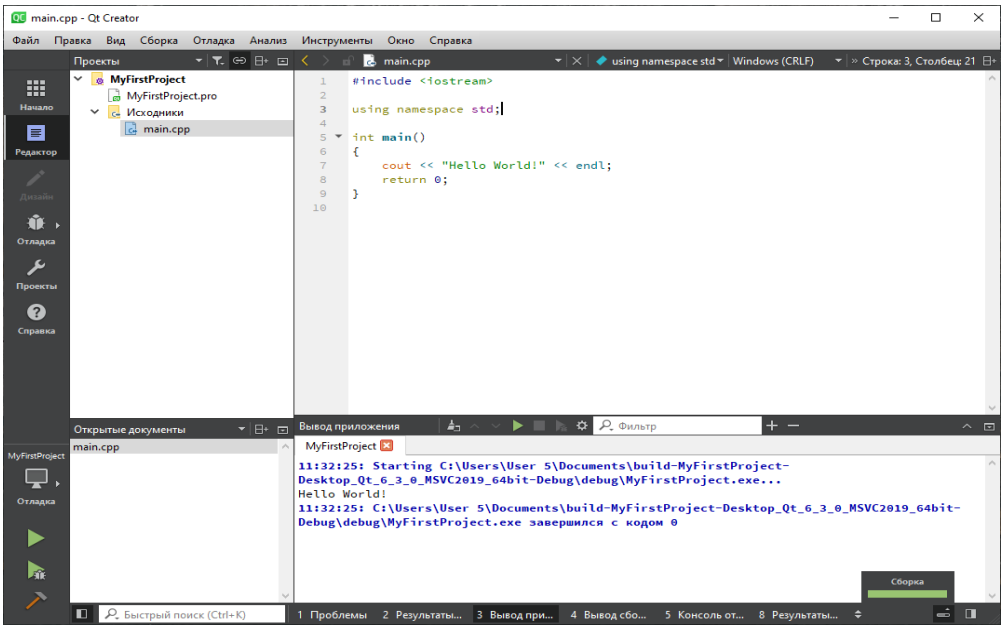


Рис. Е.7 – Qt: вид главного окна проекта

Приложение Ж
(информационное)
Унифицированный язык моделирования UML

– Ну и что это? Что это за народное творчество?

– Эх ты, серость! Это индейская национальная народная изба. «Фиг-вам» называется!

– Дожили! Мы его, можно сказать, на помойке нашли, отмыли, очистили от очисток, а он нам тут фигвамы рисует...

М/ф «Зима в Простоквашино»

Вводные замечания

Графическое представление результатов разработки является наиболее наглядным представлением мыслительной работы человека. Так, для процедурного программирования часто используются блок-схемы. Однако они слабо или вообще не пригодны для описания взаимосвязей между классами и моделирования информационных систем в целом, а также проектирования сложного программного обеспечения. Для этого был разработан унифицированный язык моделирования (UML), в полной мере пригодный для визуализации, спецификации, документирования и проектирования программного обеспечения. Далее вкратце рассмотрим базовые строительные блоки языка UML.

К основным блокам языка UML относят: *диаграммы*, *сущности* и *связи*. Под диаграммами понимают совокупность сущностей и связей, графически изображенных в виде связанного графа. Сущности – объекты, являющиеся основными элементами моделей и соединяющиеся между собой связями.

Диаграммы

В UML существует 13 видов различных диаграмм, используемых при разработке архитектур программного обеспечения и информационных систем. *Диаграмма классов* представляет собой совокупность классов, интерфейсов, а также связей между ними. Данные диаграммы чаще всего используются для моделирования объектно-ориентированных систем. *Диаграмма объектов* аналогична диаграмме классов, с учетом того что отображаются не классы, а их экземпляры.

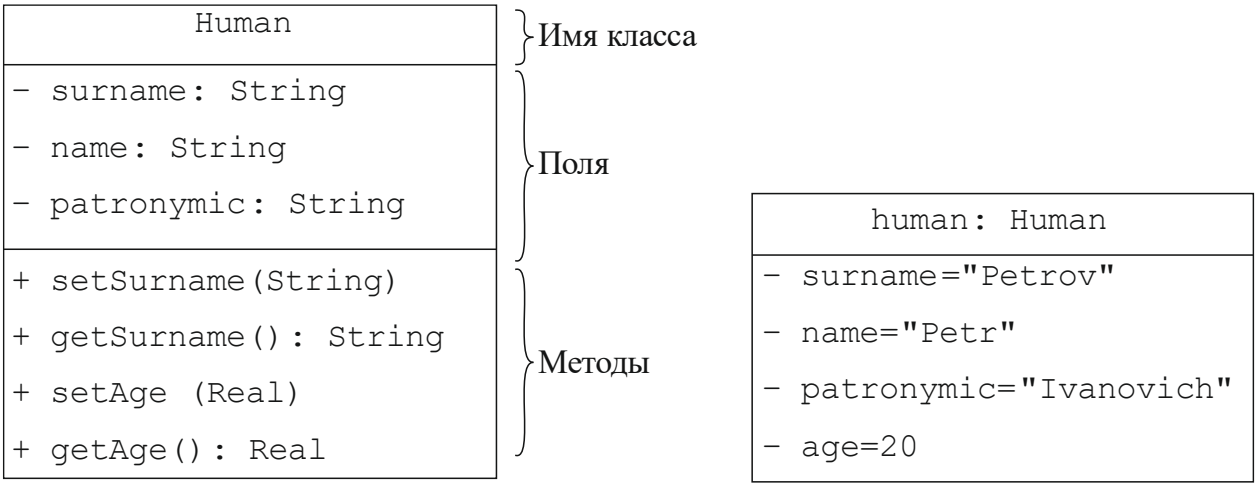
Диаграммы последовательности и *диаграммы коммуникаций* являются разновидностью диаграмм взаимодействия и предназначены для динамического описания моделируемой системы. Эти диаграммы могут содержать набор объектов, их ролей и передаваемых сообщений.

Сущности

Сущности языка UML делятся на 4 вида: структурные, поведенческие, группирующие, аннотирующие.

Структурные сущности представляют собой статические части модели, т. е. некоторые концептуальные или физические элементы. К ним относят классы и интерфейсы.

Классы изображают в виде прямоугольников, которые содержат имена классов, их полей и методов (рис. Ж.1, *а*). При этом поля и методы характеризуются уровнем видимости (по аналогии со спецификаторами доступа): «+» для `public`, «-» – `private`, «#» – `protected`, а также типом данных, который указывается после двоеточия, например, `real_d: Double`. (Есть еще один уровень видимости: «~» – `package`. Атрибут с такой видимостью недоступен для классов, определенных за пределами пакета.)



а) *б*)

Рис. Ж.1 – Обозначение класса (*а*) и его объекта (*б*)

Также поле может содержать идентификатор кратности, определяющий количество элементов в массиве (например, `maths[10]`). В случае если число элементов массива заранее неизвестно (динамический массив), используют обозначение `[*]`. Объекты классов изображаются аналогично, но они содержат только поля (рис. Ж.1, *б*).

Для отображения иерархии классов при наследовании родительский класс на языке UML принято называть суперклассом, а производные от него – подклассами. Для примера на рисунке Ж.2 приведена диаграмма, на которой показано, что подклассы `Student` (студент) и `Lecturer` (преподаватель) наследуются от суперкласса `Human` (человек).

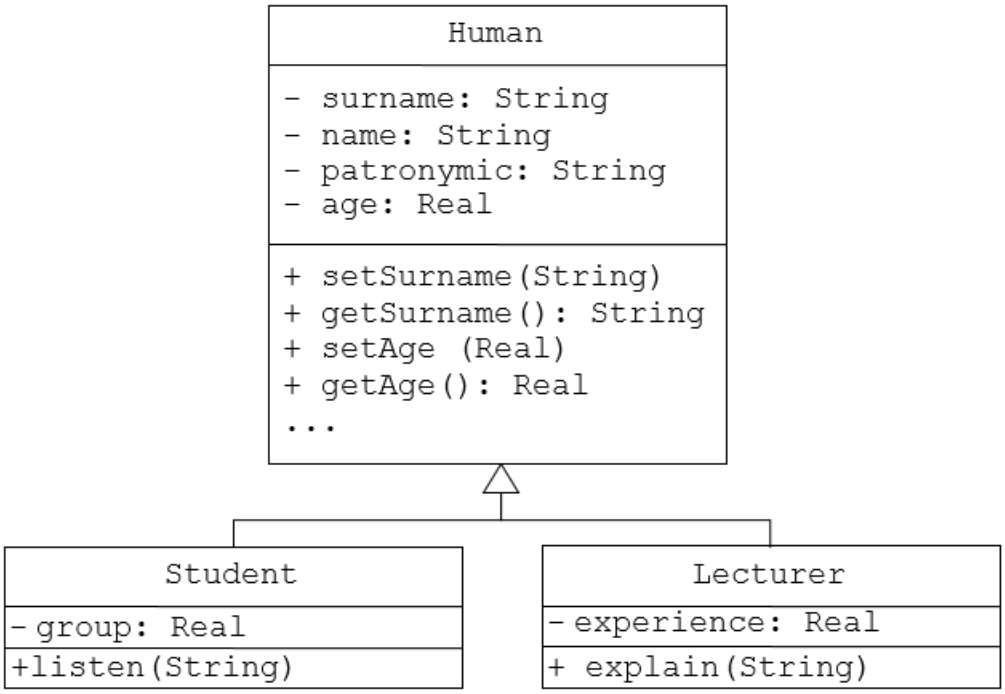


Рис. Ж.2 – Диаграмма классов

В производные классы добавляют только характерные для них элементы, т. е. поля и методы (если в производном классе нет переопределений методов базового класса), т. к. они уже содержат элементы базового класса. Так, например, для преподавателя (класс `Lecturer`) характерно иметь преподавательский стаж и уметь объяснять, а для студента (класс `Student`) характерны такие элементы, как номер учебной группы и способность слушать.

При построении иерархии классов следует помнить об абстракции данных. Так, при решении одной задачи условный класс `Student` может быть представлен одними элементами, а другой – другими. Например, в программе подсчета успеваемости студента должен обязательно присутствовать метод для подсчета его успеваемости, а в программе, ведущей учет названий выпускных квалификационных работ, – поле для их хранения и методы для их задания и считывания.

Интерфейсные классы (или просто интерфейсы) представляют собой набор методов, описывающих поведение объекта или его части. Напомним, что

класс считается интерфейсным, если он содержит только чисто виртуальные методы (подразд. 4.8), которые должны быть переопределены в наследованных от него классах. Графически интерфейсы изображаются аналогично классам, но над их именами добавляется так называемый стереотип `<<interface>>`.

Так, например, для класса `Human` может быть объявлен интерфейс `AlarmClock` (будильник), имеющий виртуальный метод `wakeUp()` (просыпаться). Тогда в каждом классе, реализующем этот интерфейс, должен быть переопределен этот метод с учетом особенностей классов (рис. Ж.3). Для пробуждения студента может использоваться, например, холодная вода, а для преподавателя – играющий «Интернационал» («Вставай, проклятьем заклейменный...»), или наоборот.

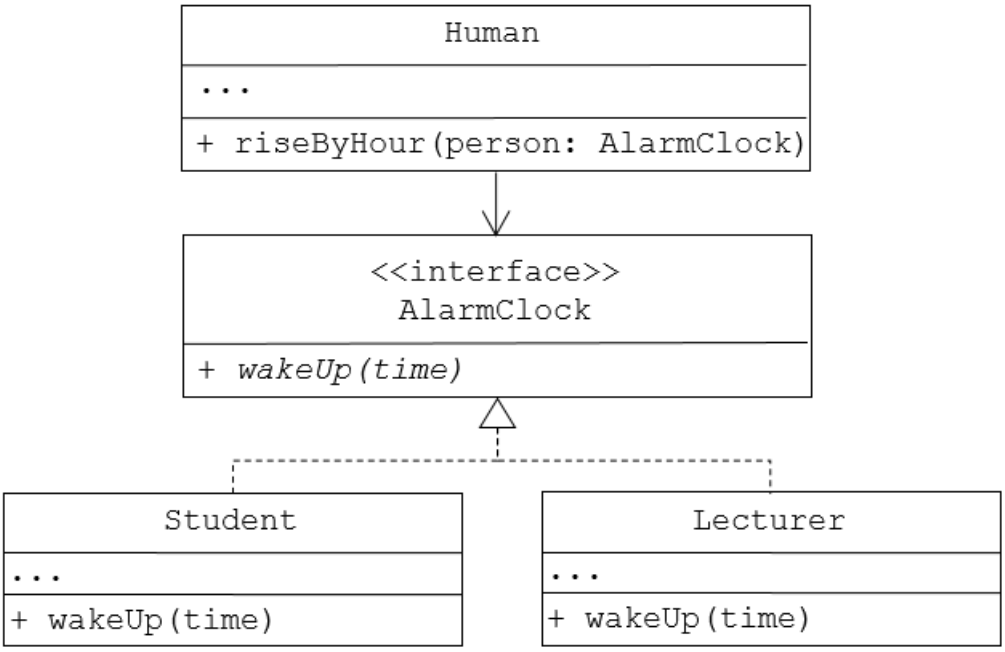


Рис. Ж.3 – Диаграмма классов, содержащая интерфейс

Помимо классов и интерфейсов к структурным сущностям относят кооперации, варианты использования, активные классы, компоненты, узлы. Их рассмотрение рекомендуется для самостоятельного изучения.

Поведенческие сущности используются для описания поведения модели во времени и пространстве.

К ним относят:

- *взаимодействие*, отражающее обмен сообщениями между объектами модели и изображающееся в виде линии со стрелкой и именем операции;

- *автомат*, описываемый в виде последовательных состояний объекта и включающий в себя состояния, переходы, события и действия;
- *деятельность*, специфицирующая последовательность шагов исполняемого процесса.

В результате во *взаимодействии* ключевое внимание отводится набору взаимодействующих объектов, в *автомате* – жизненному циклу одного объекта, а в *деятельности* описывается последовательность шагов безотносительно объекта, который их выполняет.

Группирующие сущности являются организационными частями UML-моделей, главной из которых является *пакет*. Пакет – механизм общего назначения для организации проектных решений, упорядочивающий конструкции реализации. Пакет может содержать в себе поведенческие, структурные и даже другие группирующие сущности. В отличие от элементов модели пакеты существуют только на этапе разработки, т. е. полностью концептуальны. Графически пакеты отображаются в виде папки с закладкой с указанием имени данного пакета. Кроме пакетов к группирующим сущностям относят: фреймворки, модели и подсистемы, рассмотрение которых также выносится на самостоятельное изучение.

Аннотирующие сущности отражают элементы, поясняющие части UML-моделей. К основной аннотирующей сущности относится *примечание* (комментарий), графически отображающееся в виде прямоугольника с загнутым углом и текстовым комментарием внутри.

Связи

Выделяют четыре основных типа связей: обобщение, реализация, зависимость и ассоциация.

Связь *обобщение* используется для отображения иерархии родитель – потомок и изображается в виде прямой линии с пустой треугольной стрелкой, указывающей на родителя (рис. Ж.2).

Связь *реализация* используется для демонстрации связи между интерфейсами и классами, реализующими этот интерфейс, и изображается в виде пунктирной линии с пустой треугольной стрелкой (рис. Ж.3).

Зависимость графически представляется в виде пунктирной линии (может быть со стрелкой) и семантически отображает связь между объектами модели, при которой изменения одного элемента (независимого) приводят к изменению другого элемента (зависимого). Стрелка отношения зависимости направлена от зависимого класса к независимому. На рисунке Ж.4 показан пример зависимости

двух классов, один из которых (`plotWindow`) отвечает за графическое отображение результатов неких расчетов, а другой (`plotData`) содержит данные для них. В результате изменение данных дает изменение графика.

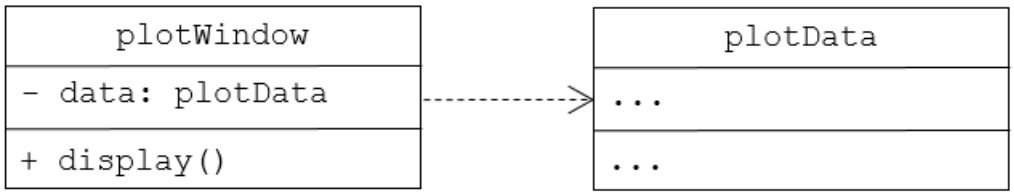


Рис. Ж.4 – Пример зависимости в UML

Ассоциация – структурная связь, отражающая набор связей между экземплярами классов. Обычно с помощью ассоциаций на диаграммах классов показывают, что один класс использует функционал другого. Рассмотрим ассоциацию на примере класса отображения графиков (`plotWindow`) и класса, отвечающего за текстовое отображение результатов (`consoleWindow`).

Пусть класс консоли содержит метод `displayData`, который может быть вызван по нажатию на соответствующую кнопку окна с графиком (рис. Ж.5). Ассоциации также могут содержать кратность, обозначающую число объектов во взаимодействии. В случае если кратность не указывается, подразумевается кратность [0..*]. Для статических классов – [1]. Помимо этого, ассоциация может содержать информативную метку, поясняющую суть отношения, а также роли участников.

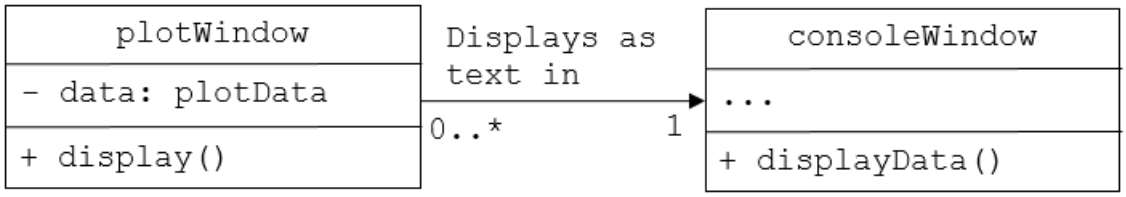


Рис. Ж.5 – Пример ассоциации в UML

Отношения *агрегация* и *композиция* являются частным случаем ассоциации и используются в ситуациях, когда необходимо отобразить связь целого с его частями. Основное отличие этих связей состоит в том, что при *агрегации* класс-часть может существовать обособленно и без класса-целого. Так, например, класс `Personal computer` (персональный компьютер) агрегирует такие элементы, как `Monitor` (монитор), `Mouse` (мышь), `Keyboard` (клавиатура), однако эти объекты могут существовать и без класса-целого (рис. Ж.6).

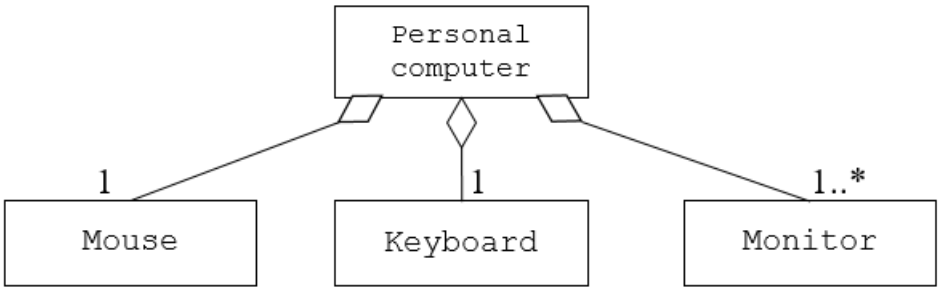


Рис. Ж.6 – Пример агрегации в UML

В противовес этому при *композиции* части не могут существовать отдельно без целого. Например, такие элементы интерфейса, как `Buttons` (кнопки), `Popup windows` (выпадающие окна), `Edit boxes` (поля ввода) в совокупности могут образовать класс `Window`, однако их отдельное существование не имеет смысла (рис. Ж.7).

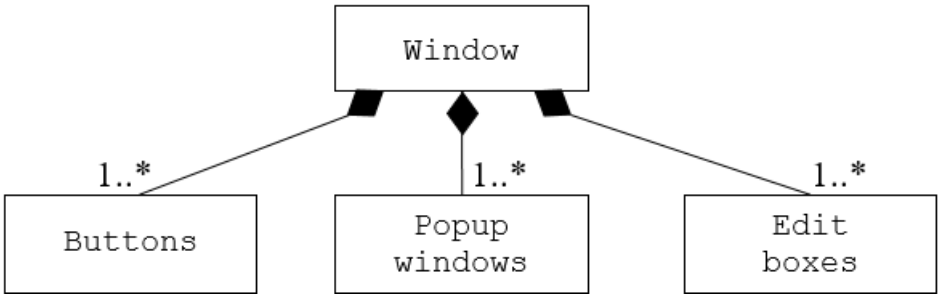


Рис. Ж.7 – Пример композиции в UML

Приложение З
(информационное)
Шаблоны проектирования

Таковы суровые законы жизни.
Или, короче выражаясь, жизнь
диктует нам свои суровые законы.

И. Ильф, Е. Петров.
Золотой теленок

Шаблон проектирования – это часто встречаемое решение определенной задачи при проектировании программной архитектуры. В отличие от готовых функций и библиотек шаблон представляет собой не какой-то конкретный код, а общую концепцию или пример решения той или иной задачи, которую надо «подстроить» под нужды разрабатываемой программы. На данный момент разработано достаточно большое число шаблонов, отличающихся по сложности, детализации, масштабу проектируемых программ и пр. Далее рассмотрены лишь два из них, поэтому заинтересованному читателю в дальнейшем целесообразно расширить свои знания о других шаблонах.

По назначению выделяют следующие группы шаблонов:

- порождающие (отвечают за гибкость создания объектов без внесения в программу лишних зависимостей);
- структурные (определяют различные способы построения связей между объектами);
- поведенческие (отвечают за формирование эффективной коммуникации между объектами).

Шаблоны проектирования представляют собой проверенные опытом других программистов решения, использование которых позволяет значительно сократить время на разработку программ и стандартизировать программный код.

Шаблоны основываются на ряде базовых принципов разработки:

1. «Инкапсулировать то, что изменяется», т. е. надо разделять изменяющиеся и неизменяющиеся компоненты программы.

2. Программировать на уровне интерфейсов, т. е. код должен зависеть от абстракций, а не от конкретных классов. Гибкость программной архитектуры выражается в том, что ее можно расширять, не «разрушая» существующий код.

Наиболее очевидный способ определения взаимодействия между двумя объектами разных классов – сделать один класс зависимым от другого. Однако есть другой способ решения данной задачи:

1) определить, что нужно одному объекту от другого и какие методы он вызывает;
2) описать эти методы в отдельном интерфейсе;
3) унаследовать первый класс от созданного интерфейса;
4) сделать второй класс зависимым от интерфейса, а не от первого класса.

3. Использовать композицию вместо наследования, где это возможно. Наследование является основным способом повторного использования кода, уже описанного в ранее разработанных классах. Однако такой подход обладает рядом недостатков:

- производный класс не может отказаться от интерфейса или реализации базового класса (своего родителя), поэтому требуется реализовывать все его абстрактные методы, даже если они не будут использоваться;
- переопределение методов в производных классах может изменять логику поведения базового класса;
- любое изменение в базовом классе может «сломать» поведение производных классов;
- множественное наследование может приводить к чрезмерному расширению иерархии классов.

Покажем, как работает композиция, на примере шаблона проектирования `Strategy` (стратегия) и описания модельного ряда транспортных средств производителя (`Transport`). Пусть имеется два вида этих средств: легковые (`Car`) и грузовые (`Truck`) автомобили. Транспортные средства каждого вида могут иметь как электрические (`Electric`), так бензиновые (`CombustionEngine`) двигатели. Кроме того, они могут иметь ручное управление (`Manual`) или автопилот (`Autopilot`). Тогда получится достаточно большая иерархия классов, т. к. производные классы нельзя наследовать от двух и более базовых классов (рис. 3.1). Дальнейшее добавление новых параметров транспортного средства приводит к еще большему увеличению иерархии классов и тем самым создает проблему дублирования программного кода.

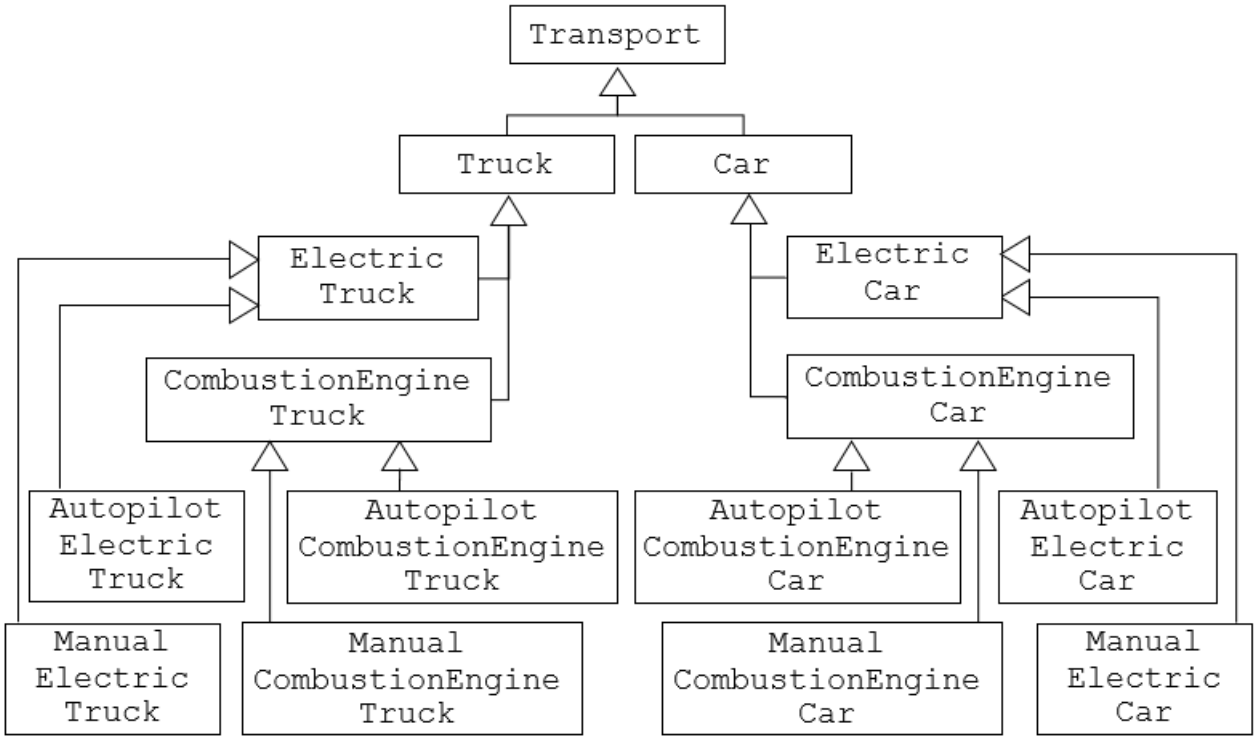


Рис. 3.1 – Модельный ряд производителя транспортных средств: иерархия классов

Как видно, в данном примере есть близкие по функционалу классы, отличающиеся лишь поведением, поэтому целесообразно сделать один общий класс, а разные варианты поведения вынести в отдельные классы. Именно в делегировании реализации поведения класса другому классу и кроется идея композиции, используемой в шаблоне `Strategy`.

Структура шаблона `Strategy` приведена на рисунке 3.2. Шаблон определяет, что схожие алгоритмы, которые могут часто изменяться или расширяться, должны быть помещены (инкапсулированы) в отдельные стратегии (классы), что обеспечивает их взаимозаменяемость. Тогда класс, который изначально сам выполнял некий алгоритм через свой метод, будет играть роль контекста (посредника), ссылаясь на нужную стратегию, тем самым делегируя ей отработку алгоритма. При этом для смены алгоритма нужно только подставить в контекст другую стратегию. Наконец, чтобы контекст не зависел от конкретных стратегий, они должны иметь общий интерфейс.

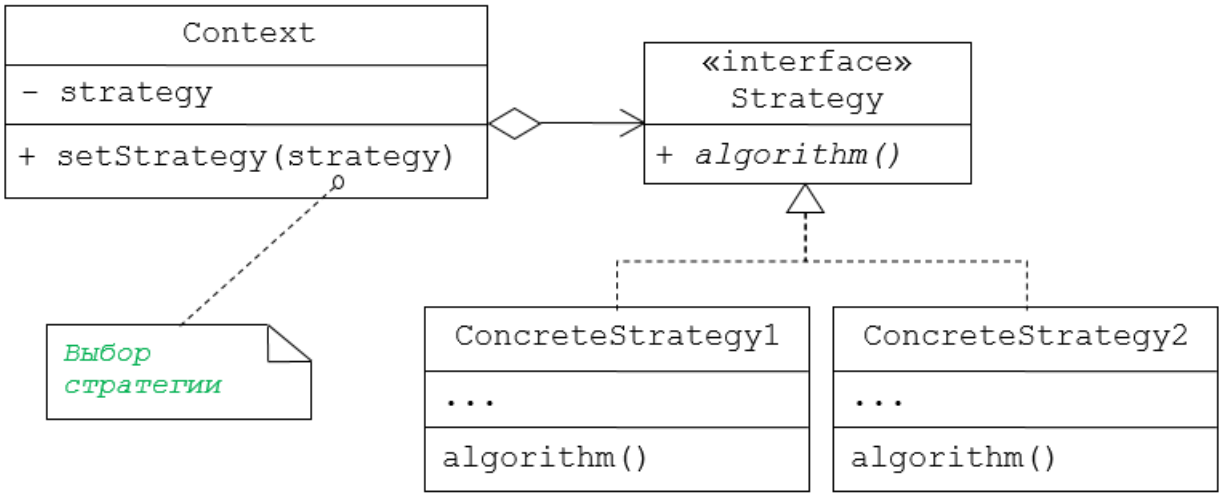


Рис. 3.2 – Структура шаблона `Strategy`

В программе, реализованной с применением шаблона `Strategy`, контекст *хранит указатель на объект конкретной стратегии*, работая с ним через общий интерфейс. Шаблон определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма. Для контекста неважно, какая именно вариация алгоритма будет выбрана, так как все они имеют одинаковый интерфейс. Во время выполнения программы контекст получает вызовы и делегирует их объекту конкретной стратегии. Поэтому надо создать объект конкретной стратегии и передать его в контекст либо через конструктор, либо в какой-то другой решающий момент, используя сеттер. Благодаря этому контекст не знает о том, какая именно стратегия сейчас выбрана.

Продемонстрируем пример использования шаблона `Strategy`.

```
1  #include <iostream>
2  #include <string.h>
3  using namespace std;
4  class Strategy; // объявление интерфейса
5  class Context{
6    public:
7      Context(){
8          operation = nullptr;
9      }
10     void setStrategy(Strategy*);
11     void useStrategy();
12   protected:
13     Strategy* operation;
14 };
```

```
15 class Strategy{ // интерфейс
16   public:
17     virtual void algorithm() = 0;
18 };
19 class ConcreteStrategy1: public Strategy{
20   public:
21     virtual void algorithm(){
22         cout << "algorithm of ConcreteStrategy1"<< endl;
23     }
24 };
25 class ConcreteStrategy2: public Strategy{
26   public:
27     virtual void algorithm(){
28         cout << "algorithm of ConcreteStrategy2"<< endl;
29     }
30 };
31 void Context::setStrategy(Strategy* s){
32     operation = s;
33 }
34 void Context::useStrategy(){
35     operation->algorithm();
36 }
37 int main(){
38     Context context;
39     ConcreteStrategy1 str1;
40     ConcreteStrategy2 str2;
41     context.setStrategy(&str1);
42     context.useStrategy();
43     context.setStrategy(&str2);
44     context.useStrategy();
45     return 0;
46 }
```

В результате для рассматриваемого примера с модельным рядом производителя транспортных средств иерархия классов примет вид, представленный на рисунке 3.3.

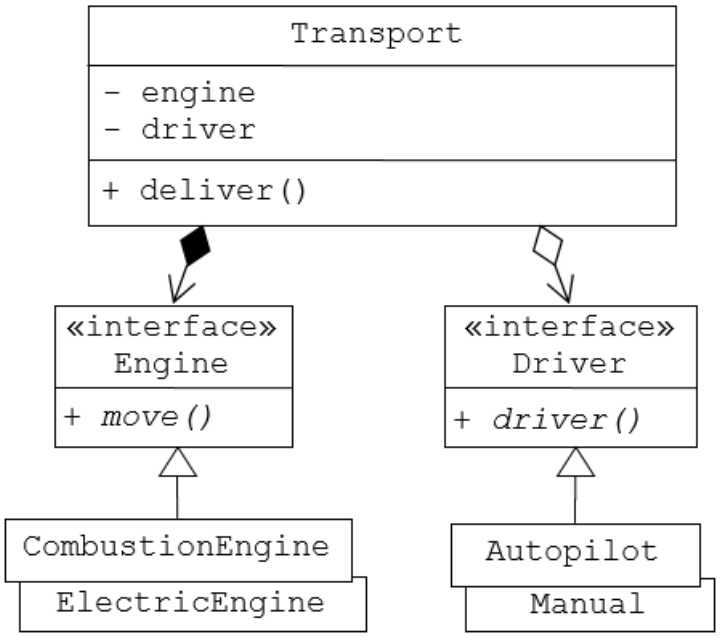


Рис. 3.3 – Модельный ряд производителя транспортных средств: иерархия классов по шаблону `Strategy`

Шаблон `Factory Method` (фабричный метод или метод виртуального конструктора) относится к группе порождающих шаблонов, который определяет общий интерфейс для создания объектов в базовом классе, позволяя производным классам изменять тип создаваемых объектов.

Предположим, что надо разработать программу для управления грузовыми перевозками (`Logistics`). Так как для перевозок предполагается использовать автомобили, весь код будет работать с объектами класса `Truck` (грузовой автомобиль). Однако если в дальнейшем может возникнуть потребность в добавлении в программу нового вида транспорта, например морского (`Ship`), то придется внести много изменений, т. к. большая часть программного кода «жестко» привязана к классу `Truck`. В этом и может помочь шаблон `Factory Method`, предписывающий создавать объекты не напрямую с помощью оператора `new`, а через вызов фабричного метода (рис. 3.4). (Не стоит переживать, т. к. оператор `new` все равно будет использован.) При этом допускается переопределение фабричного метода в производных классах, что позволяет изменять тип создаваемых объектов. Тогда все возвращаемые объекты должны иметь общий интерфейс.

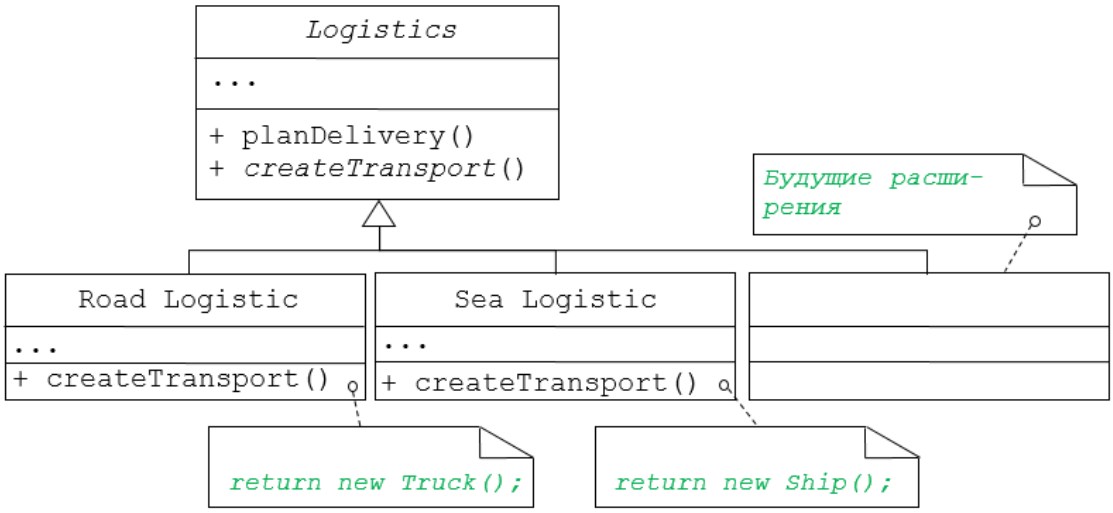


Рис. 3.4 – Структура шаблона `Factory Method`

Производные классы могут производить объекты других классов, но следующих с ними одному и тому же интерфейсу (рис. 3.5). Каждый из этих классов реализует фабричный метод по-своему. Так, например, грузовые автомобили везут грузы по земле, а корабли – по морю. Тогда фабричный метод из класса `Truck` вернет объект «грузовой автомобиль», а класс `Ship` – «корабль». Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный класс `Transport` (транспорт). Для него будет важно, чтобы объект имел метод `deliver()` (доставить), а как он это сделает, неважно.

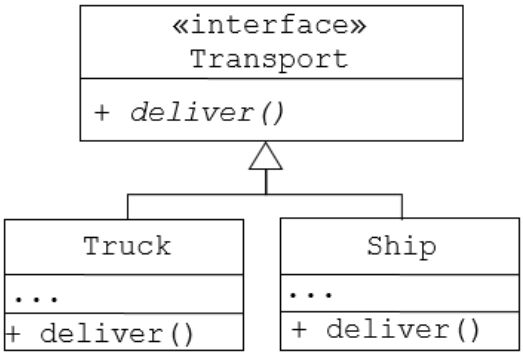


Рис. 3.5 – Общий интерфейс классов в шаблоне `Factory Method`

Структура шаблона приведена на рисунке 3.6.

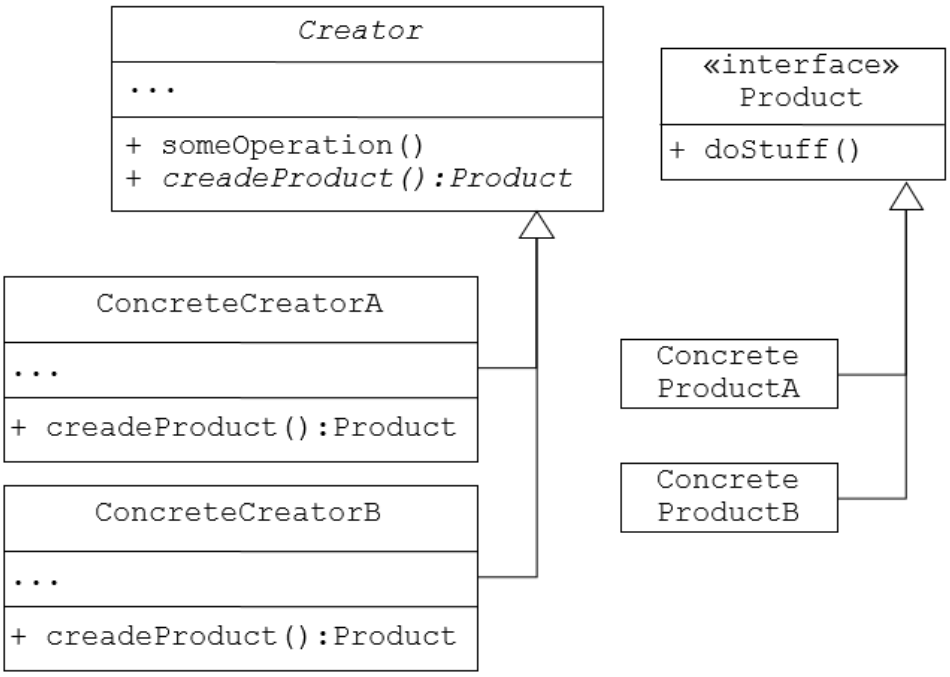


Рис. 3.6 – Структура шаблона `Factory Method`

Класс `Product` (продукт) определяет общий интерфейс объектов, которые могут произвести класс `Creator` (создатель) и его подклассы. `Creator` объявляет фабричный метод, создающий объекты через общий интерфейс `Product`. Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Однако он может возвращать и какой-то продукт по умолчанию. Стоит отметить, что создание продуктов не является единственной и главной функцией класса `Creator`. Обычно он содержит и другой полезный код работы с продуктом.

Рекомендации по использованию шаблона:

- заранее неизвестно, объекты каких типов надо создавать;
- программа должна быть независимой от процесса создания новых объектов и расширяемой;
- создание новых объектов надо делегировать производным классам из базового.

Учебное издание

Александр Витальевич Демаков
Алексей Андреевич Квасников
Сергей Петрович Куксенко

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Корректор А. Н. Миронова
Оригинал-макет Т. Н. Мосуновой

Подписано к публикации 22.12.2022.

Издательство «Эль Контент»
634061, г. Томск, ул. Киевская, д. 57, оф. 27

ISBN 978-5-4332-0302-0