Содержание

Глава 1
Общие сведения

- Зачем нужно объектно-ориентированное программирование
- Характеристики объектно-ориентированных языков
- C++ и C
- Изучение основ
- Универсальный язык моделирования (UML)

Изучив эту книгу, вы получите основные навыки создания программ на языке C++, поддерживающем *объектно-ориентированное программирование* (ООП). Для чего нужно ООП? Каковы преимущества ООП перед такими традиционными языками программирования, как Pascal, C или BASIC? Что является основой ООП? В ООП существует две ключевые концепции — *объекты* и *классы*. Каков смысл этих двух терминов? Как связаны между собой языки C и C++?

Эта глава посвящена рассмотрению перечисленных вопросов, а также обзору средств языка, о которых пойдет речь в других главах книги. Не беспокойтесь, если материал, изложенный в этой главе, покажется вам чересчур абстрактным. Все методы и концепции, которые мы упомянем, будут подробно рассмотрены в последующих главах книги.

Для чего нужно объектно-ориентированное программирование?

Развитие объектно-ориентированного метода обусловлено ограниченностью других методов программирования, разработанных ранее. Чтобы лучше понять и оценить значение ООП, необходимо разобраться, в чем состоит эта ограниченность и каким образом она проявляется в традиционных языках программирования.

Процедурные языки

C, Pascal, FORTRAN и другие сходные с ними языки программирования относятся к категории *процедурных языков*. Каждый оператор такого языка является ука-

занием компьютеру совершить некоторое действие, например принять данные от пользователя, произвести с ними определенные действия и вывести результат этих действий на экран. Программы, написанные на процедурных языках, представляют собой последовательности инструкций.

Для небольших программ не требуется дополнительной внутренней организации (часто называемой термином *парадигма*). Программист создает перечень инструкций, а компьютер выполняет действия, соответствующие этим инструкциям.

Деление на функции

Когда размер программы велик, список команд становится слишком громоздким. Очень небольшое число программистов способно удерживать в голове более 500 строк программного кода, если этот код не разделен на более мелкие логические части. *Функция* является средством, облегчающим восприятие при чтении текста программы (термин *функция* употребляется в языках C и C++; в других языках программирования это же понятие называют подпрограммой или процедурой). Программа, построенная на основе процедурного метода, разделена на функции, каждая из которых в идеальном случае выполняет некоторую законченную последовательность действий и имеет явно выраженные связи с другими функциями программы.

Можно развить идею разбиения программы на функции, объединив несколько функций в *модуль* (зачастую модуль представляет собой отдельный файл). При этом сохраняется процедурный принцип: программа делится на несколько компонентов, каждый из которых представляет собой набор инструкций.

Деление программы на функции и модули является основой структурного программирования. Структурное программирование представляет собой нечто не вполне определенное, однако в течение нескольких десятков лет, пока не была разработана концепция объектно-ориентированного программирования, оно оставалось важным способом организации программ.

Недостатки структурного программирования

В непрекращающемся процессе роста и усложнения программ стали постепенно выявляться недостатки структурного подхода к программированию. Возможно, вам приходилось слышать «страшные истории» о том, как происходит работа над программным проектом, или даже самим участвовать в создании такого проекта: задача оказывается сложнее, чем казалось, сроки сдачи проекта переносятся. Все новые и новые программисты привлекаются для работы, что резко увеличивает расходы. Окончание работы вновь переносится, и в результате проект терпит крах.

Проанализировав причины столь печальной судьбы многих проектов, можно прийти к выводу о недостаточной мощи структурного программирования: как бы эффективно ни применялся структурный подход, он не позволяет в достаточной степени упростить большие сложные программы.

В чем же недостаток процедурно-ориентированных языков? Существует две основные проблемы. Первая заключается в неограниченности доступа функций к глобальным данным. Вторая состоит в том, что разделение данных и функций, являющееся основой структурного подхода, плохо отображает картину реального мира.

Давайте рассмотрим эти недостатки на примере программы складского учета. В такой программе глобальными данными являются записи в учетной книге. Различные функции будут получать доступ к этим данным для выполнения операций создания новой записи, вывода записи на экран, изменения существующей записи и т. д.

Неконтролируемый доступ к данным

В процедурной программе, написанной, к примеру, на языке C, существует два типа данных. *Локальные данные* находятся внутри функции и предназначены для использования исключительно этой функцией. Например, в программе складского учета функция, осуществляющая вывод записи на экран, может использовать локальные данные для хранения информации о выводимой записи. Локальные данные функции недоступны никому, кроме самой функции, и не могут быть изменены другими функциями.

Если существует необходимость совместного использования одних и тех же данных несколькими функциями, то данные должны быть объявлены как *глобальные*. Это, как правило, касается тех данных программы, которые являются наиболее важными. Примером здесь может служить уже упомянутая учетная книга. Любая функция имеет доступ к глобальным данным (мы не рассматриваем случай группирования функций в модули). Схема, иллюстрирующая концепцию локальных и глобальных данных, приведена на рис. 1.1.

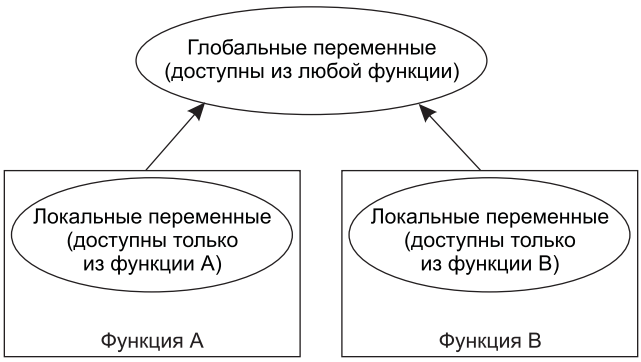

Рис. 1.1. Глобальные и локальные переменные

Большие программы обычно содержат множество функций и глобальных переменных. Проблема процедурного подхода заключается в том, что число возможных связей между глобальными переменными и функциями может быть очень велико, как показано на рис. 1.2.

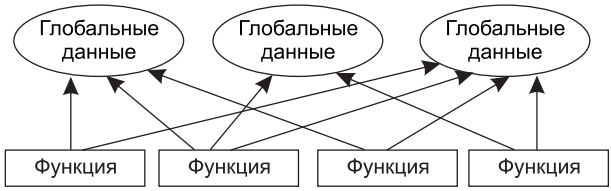

Рис. 1.2. Процедурный подход

Большое число связей между функциями и данными, в свою очередь, также порождает несколько проблем. Во-первых, усложняется структура программы. Во-вторых, в программу становится трудно вносить изменения. Изменение структуры глобальных данных может потребовать переписывания всех функций, работающих с этими данными. Например, если разработчик программы складского учета решит сделать код продукта не 5-значным, а 12-значным, то будет необходимо изменить соответствующий тип данных с `short` на `long`. Это означает, что во все функции, оперирующие кодом продукта, должны быть внесены изменения, позволяющие обрабатывать данные типа `long`. Можно привести аналогичный бытовой пример, когда в супермаркете изменяется расположение отделов, и покупателям приходится соответствующим образом менять свой привычный путь от одного отдела к другому.

Когда изменения вносятся в глобальные данные больших программ, бывает непросто быстро определить, какие функции необходимо скорректировать. Даже в том случае, когда это удается сделать, из-за многочисленных связей между функциями и данными исправленные функции начинают некорректно работать с другими глобальными данными. Таким образом, любое изменение влечет за собой далеко идущие последствия.

Моделирование реального мира

Вторая, более важная, проблема процедурного подхода заключается в том, что отделение данных от функций оказывается малопригодным для отображения картины реального мира. В реальном мире нам приходится иметь дело с физическими объектами, такими, например, как люди или машины. Эти объекты нельзя отнести ни к данным, ни к функциям, поскольку реальные вещи представляют собой совокупность *свойств* и *поведения*.

Свойства

Примерами свойств (иногда называемых характеристиками) для людей могут являться цвет глаз или место работы; для машин — мощность двигателя и количество дверей. Таким образом, свойства объектов равносильны данным в программах: они имеют определенное значение, например *голубой* для цвета глаз или *4* для количества дверей автомобиля.

Поведение

Поведение — это некоторая реакция объекта в ответ на внешнее воздействие. Например, ваш босс в ответ на просьбу о повышении может дать ответ «да» или

«нет». Если вы нажмете на тормоз автомобиля, это повлечет за собой его остановку. Ответ и остановка являются примерами поведения. Поведение сходно с функцией: вы вызываете функцию, чтобы совершить какое-либо действие (например, вывести на экран учетную запись), и функция совершает это действие.

Таким образом, ни отдельно взятые данные, ни отдельно взятые функции не способны адекватно отобразить объекты реального мира.

Объектно-ориентированный подход

Основополагающей идеей объектно-ориентированного подхода является объединение *данных* и *действий, производимых над этими данными*, в единое целое, которое называется *объектом*.

Функции объекта, называемые в C++ *методами* или *функциями-членами*, обычно предназначены для доступа к данным объекта. Если необходимо считать какие-либо данные объекта, нужно вызвать соответствующий метод, который выполнит считывание и возвратит требуемое значение. Прямой доступ к данным невозможен. Данные сокрыты от внешнего воздействия, что защищает их от случайного изменения. Говорят, что данные и методы *инкапсулированы*. Термины *сокрытие* и *инкапсуляция* данных являются ключевыми в описании объектно-ориентированных языков.

Если необходимо изменить данные объекта, то, очевидно, это действие также будет возложено на методы объекта. Никакие другие функции не могут изменять данные класса. Такой подход облегчает написание, отладку и использование программы.

Типичная программа на языке C++ состоит из совокупности объектов, взаимодействующих между собой посредством вызова методов друг друга. Структура программы на C++ приводится на рис. 1.3.

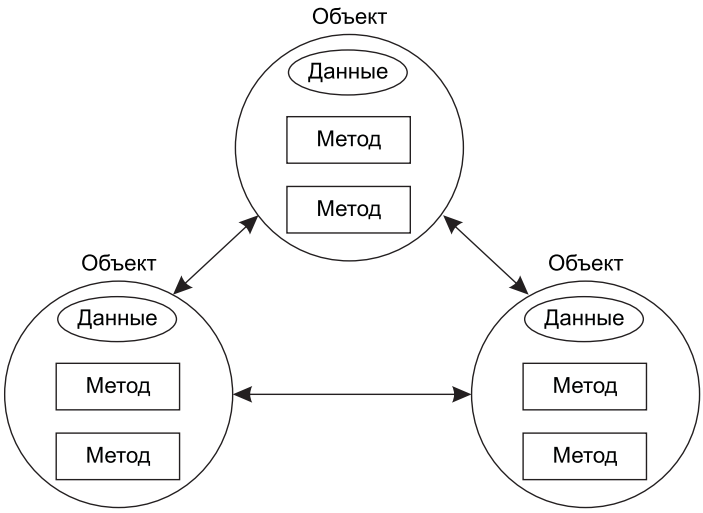

Рис. 1.3. Объектно-ориентированный подход