**Методология объектно-ориентированного программирования**

Методология *объектно-ориентированного программирования* пришла на смену процедурной или алгоритмической организации структуры программного кода.

Объектно-ориентированное программирование (ООП, Object-Oriented Programming) - совокупность принципов, технологий, а также инструментальных средств для создания программных систем на основе архитектуры взаимодействия *объектов*.

Распространение методологии *ООП* связано с процессом разработки программ. В частности, процедурно-ориентированная *декомпозиция* программ уступила *место* объектно-ориентированной, при которой в качестве отдельных структурных единиц программы рассматриваются не процедуры и функции, а *классы* и *объекты* с соответствующими свойствами и методами.

Как следствие, *программа* перестала быть последовательностью предопределенных на этапе кодирования действий, а преобразовалась в событийно управляемую. Последнее обстоятельство доминирует и при разработке широкого круга современных приложений. В этом случае каждая *программа* представляет собой бесконечный цикл ожидания заранее определенных событий. Инициаторами событий могут быть другие программы или пользователи, а при наступлении отдельного события *программа* выходит из состояния ожидания и реагирует на него вполне адекватным образом.

Основные принципы *ООП*: *абстракция*, *наследование*, *инкапсуляция* и *полиморфизм*.

***Абстракция*** (abstraction) - характеристика сущности, которая отличает ее от других сущностей. *Абстракция* определяет границу представления соответствующего элемента *модели* и применяется для определения фундаментальных понятий *ООП*, таких как *класс* и*объект*.

***Класс*** представляет собой *абстракцию* совокупности реальных *объектов*, которые имеют общий набор свойств и обладают одинаковым поведением. ***Объект*** в контексте ООП рассматривается как экземпляр соответствующего *класса*.

*Объекты*, которые не имеют идентичных свойств или не обладают одинаковым поведением, *по* определению, не могут быть отнесены к одному *классу*.

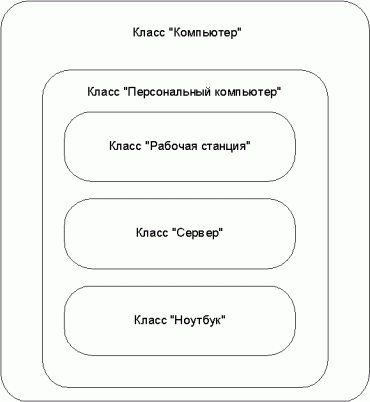
*Классы* можно организовать в виде иерархической структуры, которая *по* внешнему виду напоминает схему классификации в понятийной логике. *Иерархия* понятий строится следующим образом. В качестве наиболее общего понятия или категории берется понятие, имеющее наибольший объем и, соответственно, наименьшее содержание. Это самый высокий уровень *абстракции* для данной иерархии. Затем данное общее понятие конкретизируется, то есть уменьшается его объем и увеличивается содержание. Появляется менее общее понятие, которое на схеме иерархии будет расположено на уровень ниже исходного. Этот процесс конкретизации понятий может быть продолжен до тех пор, пока на самом нижнем уровне не будет получено понятие, дальнейшая *конкретизация* которого в данном контексте либо невозможна, либо нецелесообразна.

Принцип, в соответствии с которым знание о наиболее общей категории разрешается применять для более частной категории, называется ***наследованием***.

*Наследование* тесно связано с иерархией *классов*, определяющей, какие *классы* следует считать наиболее абстрактными и общими *по*отношению к другим *классам*. При этом если общий или родительский *класс* (предок) обладает фиксированным набором свойств и поведением, то производный от него *класс* (*потомок*) должен содержать этот же набор свойств и подобное поведение, а также дополнительные, которые будут характеризовать уникальность полученного *класса*. В этом случае говорят, что производный *класс*наследует свойства и поведение родительского *класса*.

Для иллюстрации принципа *наследования* можно привести следующий пример. Общий *класс* "*Компьютер*". Он определяется как*абстракция* свойств и поведения всех, реально существующих электронных вычислительных машин. При этом общими свойствами *класса*"*Компьютер*" могут быть такие, как наличие процессора, оперативной памяти, устройств ввода и вывода информации. Если в качестве производного рассмотреть *класс* "Персональный *компьютер*", то все выделенные выше свойства будет содержать и этот *класс*. Можно сказать, что *класс* "Персональный *компьютер*" наследует свойства родительского *класса* "*Компьютер*". Однако кроме перечисленных свойств классу-потомку будут присущи дополнительные, например, наличие системного блока и материнской платы с разъемом для установки микропроцессора.

В свою *очередь* *класс* "Персональный *компьютер*" может быть классом-предком для других *классов*, в частности "*Рабочая станция*", "*Сервер*" и "Ноутбук". С этой точки зрения все указанные *классы* наследуют свойства родительского *класса* "Персональный *компьютер*", а возможно и переопределяют некоторые из них. Описанная выше текстовая *информация* о соотношении *классов* в данном примере обладает одним серьезным недостатком, а именно, отсутствием наглядности. В этой связи возникает вопрос: а возможно ли представить иерархию *наследования* *классов* в визуальной форме? В понятийной логике для изображения понятий используются окружности или прямоугольники. С помощью этой графической нотации, иерархию *классов* для рассмотренного примера можно представить в виде вложенных прямоугольников или окружностей, каждый из которых соответствует отдельному *классу* ([рис. 1.1](http://www.intuit.ru/studies/courses/32/32/lecture/1000?page=1#image.1.1)).



**Рис. 1.1.**Иерархия вложенности классов для примера общего класса "Компьютер"

Подобное изображение обладает серьезным недостатком. Из представленного рисунка не ясно, изображена ли на нем *иерархия* понятий или *декомпозиция* *класса* "*Компьютер*" на его составные части. Как будет показано далее, использование нотации *UML* позволяет устранить данную неопределенность посредством введения в рассмотрение двух различных отношений: обобщения и агрегации (Лекция 6).

Следующий принцип *ООП* - *инкапсуляция*. ***Инкапсуляция*** характеризует сокрытие отдельных деталей внутреннего устройства *классов*от внешних по отношению к нему *объектов* или пользователей.

Клиенту, взаимодействующему с *объектом* *класса*, необязательно знать, каким образом осуществлен тот или иной элемент *класса*. Конкретная реализация присущих *классу* свойств и методов, которые определяют его поведение, является собственным делом данного*класса*. Более того, отдельные свойства и методы *класса* могут быть невидимы за его пределами, это относится к базовой идее введения различных категорий видимости для элементов *класса*.

На примере с *классом* "*Компьютер*" нетрудно проиллюстрировать *инкапсуляцию* следующим образом. Основным субъектом, который взаимодействует с *объектами* этого *класса*, является *пользователь*. Вполне очевидно, что не каждый *пользователь* в совершенстве знает внутреннее устройство того или иного компьютера. К тому же, отдельные детали этого устройства сознательно скрыты в корпусе системного блока или монитора. А в случае нарушения работы компьютера, которое является причиной неадекватности его поведения, необходимый ремонт выполняет профессиональный специалист.

*Инкапсуляция* ведет свое происхождение от деления модулей в некоторых языках программирования на две части или секции:*интерфейс* и реализацию. При этом в интерфейсной секции модуля описываются все объявления функций и процедур, а возможно и типов данных, доступных за пределами модуля. Указанные процедуры и функции являются способами оказания услуг внешним клиентам. В другой секции модуля, называемой реализацией, содержится программный код, который определяет конкретные способы реализации объявленных в интерфейсной части процедур и функций.

*Полиморфизм* также один из основных принципов *ООП*. Под ***полиморфизмом*** (греч. Poly - много, morfos - форма) понимается свойство*объектов* принимать различные внешние формы в зависимости от обстоятельств. Применительно к *ООП* *полиморфизм* означает, что действия, выполняемые одноименными методами, могут различаться в зависимости от того, к какому из *классов* относится тот или иной метод.

К примеру, три *объекта* соответствующих *классов*: двигатель автомобиля, электрический свет в комнате и персональный *компьютер*. Для каждого из них можно определить операцию выключить(). Однако результат выполнения этой *операции* будет отличаться для каждого из рассмотренных *объектов*. Так для двигателя автомобиля выполнение *операции* выключить() означает прекращение подачи топлива и его остановку. Выполнение *операции* выключить() для электрического света в комнате означает простой щелчок выключателя, после чего комната погружается в темноту. В последнем случае для персонального компьютера выполнение *операции*выключить() может быть причиной потери данных, если производится нерегламентированным образом.

*Полиморфизм* объектно-ориентированных языков связан с перегрузкой функций, но не тождествен ей. Важно иметь в виду, что имена методов и свойств тесно связаны с *классами*, в которых они описаны. Это обстоятельство обеспечивает определенную *надежность*работы программы, поскольку исключает случайное применение метода для решения несвойственной ему задачи.

Наиболее существенным обстоятельством в развитии методологии *ООП* явилось осознание того, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Прежде, чем начать *программирование* *классов*, их свойств и методов, необходимо определить сами эти *классы*. Более того, нужно дать ответы на следующие вопросы: сколько и какие*классы* нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания *классам* требуемого поведения, а также установить взаимосвязи между *классами*. Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной *предметной области*, для которой разрабатывается*программа*. Все эти обстоятельства привели к появлению специальной методологии, получившей название методологии *объектно-ориентированного анализа и проектирования* (ООАП).

### Методология объектно-ориентированного анализа и проектирования

Необходимость анализа *предметной области* до начала написания программы была осознана при разработке масштабных проектов. Процесс создания баз данных существенно отличается от написания программного кода для решения вычислительной задачи. Так, при проектировании базы данных возникает необходимость в предварительной разработке концептуальной схемы или *модели*, которая отражала бы общие взаимосвязи *предметной области* и особенности организации соответствующей информации.

***Предметная область*** (domain) - часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы. Другими словами, *предметная область* включает в себя только те *объекты* и взаимосвязи между ними, которые необходимы для описания требований и условий решения конкретной задачи.

Выделение исходных или базовых компонентов *предметной области*, требуемых для решения той или иной задачи, представляет, в общем случае, нетривиальную проблему. Сложность данной проблемы проявляется в неформальном характере процедур или правил, которые можно применять для этой цели. Более того, эта работа должна выполняться совместно со специалистами или экспертами, хорошо знающими *предметную область*. Например, если разрабатывается *база данных* для обслуживания пассажиров крупного аэропорта, то в проектировании концептуальной схемы *базы данных* должны принимать участие штатные сотрудники аэропорта. Эти сотрудники хорошо знают весь процесс обслуживания пассажиров или данную *предметную область*. Сложность моделирования*предметной области* и разработки корпоративных информационных систем привело к появлению новой методологии *объектно-ориентированный анализ и проектирование*.

***Объектно-ориентированный анализ и проектирование*** (ООАП, *Object-Oriented Analysis*/Design) - технология разработки программных систем, в основу которых положена объектно-ориентированная методология представления *предметной области* в виде*объектов*, являющихся экземплярами соответствующих *классов* .

Методология ООАП тесно связана с концепцией автоматизированной разработки программного обеспечения (*Computer* Aided *Software**Engineering*, *CASE*). К первым *CASE*-средствам отнеслись с определенной настороженностью. Со временем появились как восторженные отзывы об их применении, так и критические оценки их возможностей. Причин для столь противоречивых мнений было несколько. Первая из них заключается в том, что ранние *CASE*-средства были простой надстройкой над системой управления базами данных (*СУБД*).*Визуализация* процесса разработки концептуальной схемы *БД* имеет немаловажное *значение*, тем не менее, она не решает проблем создания приложений других типов.

Вторая причина связана с графической нотацией, реализованной в *CASE*-средстве. Если языки программирования имеют строгий*синтаксис*, то попытки предложить подходящий *синтаксис* для визуального представления концептуальных схем *БД*, были восприняты далеко не однозначно. На этом фоне разработка и стандартизация унифицированного языка моделирования *UML* вызвала воодушевление у всего сообщества корпоративных программистов.

В рамках ООАП исторически рассматривались три графических нотации:

* диаграммы "сущность-связь" (*Entity-Relationship Diagrams*, *ERD*),
* диаграммы *функционального моделирования* (*Structured Analysis* and *Design Technique*, *SADT*),
* диаграммы потоков данных (*Data Flow Diagrams*, *DFD*).

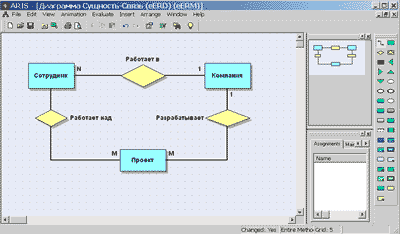
Диаграммы "сущность-*связь*" (*ERD*) предназначены для графического представления *моделей* данных разрабатываемой программной системы и предлагают набор стандартных обозначений для определения данных и отношений между ними. С помощью этого вида диаграмм можно описать отдельные компоненты концептуальной *модели* данных и совокупность взаимосвязей между ними.

Основными понятиями данной нотации являются понятия сущности и связи. При этом под сущностью (*entity*) понимается *произвольное*множество реальных или абстрактных *объектов*, каждый из которых обладает одинаковыми свойствами и характеристиками. В этом случае любой рассматриваемый *объект* может быть экземпляром одной и только одной сущности, должен иметь уникальное имя или*идентификатор*, а также отличаться от других экземпляров данной сущности.

*Связь* (*relationship*) определяется как *отношение* или *ассоциация* между отдельными сущностями. Примерами связей могут являться родственные отношения, в частности "отец-сын" или производственные - "начальник-подчиненный". Другой тип связей задается отношениями "иметь в собственности" или "обладать свойством". Различные типы связей графически изображаются в форме ромба с соответствующим именем данной связи.

Графическая *модель* данных строится таким образом, чтобы связи между отдельными сущностями отражали не только семантический характер соответствующего отношения, но и дополнительные аспекты обязательности связей, а также *кратность* участвующих в данных отношениях *экземпляров сущностей*. *Нотация* диаграмм (*ERD*) реализована в различных программных средствах. Пример диаграммы*ERD*, разработанной с помощью средства моделирования бизнес-процессов *ARIS* \text{\textregistered}, изображен на [рис.1.2](http://www.intuit.ru/studies/courses/32/32/lecture/1000?page=2#image.1.2).

Ограниченность диаграмм *ERD* проявляется при конкретизации концептуальной *модели* в более детальное *представление* моделируемой программной системы, которое кроме статических связей должно содержать информацию о поведении или функционировании отдельных ее компонентов.

[](http://www.intuit.ru/EDI/25_12_15_1/1450995683-14467/tutorial/92/objects/1/files/1-2big.gif)

[увеличить изображение](http://www.intuit.ru/EDI/25_12_15_1/1450995683-14467/tutorial/92/objects/1/files/1-2big.gif)  
**Рис. 1.2.**Диаграмма "сущность-связь" для примера сотрудников компании, работающих над различными проектами

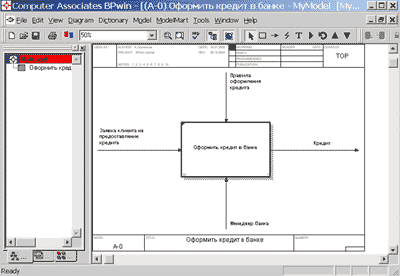
В рамках диаграмм *функционального моделирования* было разработано несколько графических языков моделирования, которые получили следующие названия:

* Нотация *IDEF0* - для документирования процессов производства и отображения информации об использовании ресурсов на каждом из этапов проектирования систем
* Нотация IDEF1 - для документирования информации о производственном окружении систем
* Нотация *IDEF2* - для документирования поведения системы во времени

*Нотация* *IDEF2* никогда не была полностью реализована. *Нотация* IDEF1 в 1985 году была расширена и переименована в *IDEF1X*. Методология *IDEF* нашла применение в правительственных и коммерческих организациях, поскольку в 1993 году появился стандарт *FIPS*правительства США для двух технологий *IDEF0* и *IDEF1X*. В течение последующих лет этот стандарт продолжал активно развиваться и послужил основой для реализации в некоторых *CASE*-средствах, наиболее известным из которых является AllFusion *Process Modeler* \text{\textregistered}(новое название BPwin \text{\textregistered}) компании *Computer* Associates.

Процесс моделирования *IDEF* представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной *модели* системы какой-либо *предметной области*. Функциональная *модель* *IDEF* отображает структуру процессов функционирования системы и ее отдельных подсистем, то есть, выполняемые ими действия и связи между этими действиями. Для этой цели строятся специальные *модели*, которые позволяют в наглядной форме представить последовательность определенных действий. Исходными строительными блоками любой *модели* нотации *IDEF0* процесса являются *деятельность* (*activity*) и стрелки (arrows).

Одна из наиболее важных особенностей нотации *IDEF0* - постепенное введение все более детальных представлений *модели* системы *по*мере разработки отдельных диаграмм. Построение *модели* *IDEF0* начинается с представления всей системы в виде простейшей диаграммы, состоящей из одного блока процесса и стрелок ICOM, служащих для изображения основных видов взаимодействия с*объектами* вне системы. Поскольку исходный процесс представляет всю систему как единое целое, данное *представление* является наиболее общим и подлежит дальнейшей декомпозиции. Пример представления общей *модели* процесса оформления кредита в банке, разработанной с помощью *CASE*-средства AllFusion *Process Modeler* \text{\textregistered}, изображен на [рис. 1.3](http://www.intuit.ru/studies/courses/32/32/lecture/1000?page=2#image.1.3).

[](http://www.intuit.ru/EDI/25_12_15_1/1450995683-14467/tutorial/92/objects/1/files/1-3big.gif)

[увеличить изображение](http://www.intuit.ru/EDI/25_12_15_1/1450995683-14467/tutorial/92/objects/1/files/1-3big.gif)  
**Рис. 1.3.**Пример исходной диаграммы IDEF0 для процесса оформления кредита в банке

В конечном итоге *модель* *IDEF0* представляет собой набор иерархически взаимосвязанных диаграмм с сопроводительной документацией, которая разбивает исходное *представление* сложной системы на отдельные составные части. Детали каждого основного процесса представляются в виде более подробных процессов на других диаграммах. В этом случае каждая *диаграмма* нижнего уровня является декомпозицией процесса из более общей диаграммы. Поэтому на каждом шаге декомпозиции более общая *диаграмма* конкретизируется на ряд детальных диаграмм.

Основной недостаток данной методологии связан с отсутствием явных средств для объектно-ориентированного представления *моделей*сложных систем. Некоторые аналитики отмечают важность знания и применения нотации *IDEF0*, однако отсутствие возможности реализации соответствующих графических *моделей* в объектно-ориентированном программном коде существенно сужают *диапазон*решаемых с ее помощью задач.

В основе графического моделирования информационных систем с помощью диаграмм потоков данных лежит специальная технология построения диаграмм потоков данных *DFD*. В разработке методологии *DFD* приняли участие многие аналитики, среди которых следует отметить Э. Йордона. Он *автор* одной из первых графических нотаций *DFD*.

Недостаток рассмотренных нотаций связан с отсутствием явных средств для объектно-ориентированного представления *моделей*сложных систем, а также сложных алгоритмов обработки данных. Поскольку на рассмотренных типах диаграмм не указываются характеристики времени выполнения отдельных процессов и передачи данных между процессами, то *модели* систем, реализующих синхронную обработку данных, не могут быть адекватно представлены в этих нотациях. Все эти особенности методов структурного системного анализа ограничили возможности широкого применения соответствующих нотаций и послужили основой для разработки унифицированного языка моделирования *UML*.

Абстрактные типы данных

Класс Stock довольно специфичен. Однако программисты часто определяют

классы для представления более общих концепций. Например, использование классов —

хороший способ реализации того, что специалисты в области вычислительной

техники называют абстрактным типом данных (abstract data type — ADT). Как и можно

было предположить, ADT описывает данные в общей манере, без деталей, связанных

с языком или реализацией. Рассмотрим для примера стек. Используя стек, данные

можно сохранять так, что они всегда будут добавляться или удаляться с его вершины.

Например, программы на C++ применяют стек для управления автоматическими

переменными. Когда новые переменные создаются, они добавляются на вершину стека, а

когда уничтожаются, то удаляются из нее.

Давайте посмотрим на свойства стека в абстрактном смысле. Прежде всего, стек

содержит множество элементов. (Это свойство делает его контейнером — т.е. еще

более общей абстракцией.) Вдобавок стек характеризуется операциями, которые на нем

можно выполнять:

• создание пустого стека;

• добавление элемента в вершину стека (т.е. заталкивание (push) элемента);

• удаление элемента из вершины стека (т.е. выталкивание (pop) элемента);

• проверка, полон ли стек;

• проверка, пуст ли стек.

Это описание может быть сопоставлено с объявлением класса, в котором

общедоступные функции-члены предоставляют интерфейс, реализующий операции над

стеком. Закрытые данные-члены будут обеспечивать хранение информации в стеке.

Концепция класса хорошо соответствует подходу ADT.

Раздел private должен позаботиться о хранении данных. Например, можно

использовать обычный массив, динамически распределенный в памяти массив либо

какую-то более развитую структуру данных вроде связного списка. Однако открытый

интерфейс класса должен скрывать точные детали представления. Наоборот, он должен

быть выражен в общих понятиях, таких как создание стека, заталкивание элемента

и т.д. В листинге 10.10 показан один из возможных подходов. Предполагается, что тип

bool реализован. Если же это не так, то вместо bool с false и true можно

использовать int со значениями 0 и 1.

Листинг 10.10. stack. h

// stack.h -- определение класса для абстрактного типа данных — стека

#ifndef STACK\_H\_

#define STACK\_H\_

typedef unsigned long Item;

class Stack

{

private:

enum {MAX = 10}; // константа, специфичная для класса

Item items[MAX]; // хранит элементы стека

int top; // индекс вершины стека

public:

Stack();

bool isemptyO const;

bool isfull() const;

Объекты и классы 527

// push () возвращает false, если стек полон, и true - в противном случае

bool push(const Item & item); // добавляет элемент в стек

// pop () возвращает false, если стек пуст, и true - в противном случае

bool pop(Item & item); // выталкивает элемент с вершины стека

};

#endif

В примере, приведенном в листинге 10.10, раздел private показывает, что стек

реализован с помощью массива, но раздел public никак не отражает этот факт. То

есть обычный массив можно заменить, скажем, динамическим массивом, не меняя

интерфейс класса. Это означает, что изменение реализации стека не требует внесения

изменений в код программы, которая будет его использовать. Вы просто

перекомпилируете код реализации стека и скомпонуете его с кодом программы.

Представленный интерфейс несколько избыточен, т.к. pop () и push () возвращают

информацию о состоянии стека (пуст или полон) вместо того, чтобы иметь тип void.

Это обеспечивает дополнительные возможности по управлению переполнением стека

и его очисткой. Можно использовать isempty () и isfull () для проверки перед

попытками модификации стека или с помощью возвращаемых значений push () и pop ()

определять, удалась ли соответствующая операция.

Вместо того чтобы определять стек в терминах некоторого конкретного типа,

класс описывает его в терминах общего типа Item. В данном случае заголовочный

файл использует typedef для указания, что Item является unsigned long. Если вы

хотите создать стек для хранения элементов типа double или структур, измените

typedef, а объявление класса и определения методов останутся прежними. Шаблоны

классов (см. главу 14) предлагают более мощный метод изоляции типа хранимых

данных от проектного решения класса.

Далее потребуется реализовать методы класса. В листинге 10.11 показан один из

возможных вариантов.

Листинг 10.11. stack.срр

// stack.срр — функции-члены класса Stack

#include "stack.h"

Stack::Stack() // создание пустого стека

top = 0;

bool Stack::isempty () const

return top == 0;

bool Stack::isfull () const

return top == MAX;

bool Stack: :push (const Item & item)

if (top < MAX)

{

items[top++] = item;

return true;

}

else

return false;

}

528 Глава 10

bool Stack: :pop (Item & item)

{

if (top > 0)

{

item = items [—top];

return true;

}

else

return false;

Конструктор по умолчанию гарантирует, что все стеки будут создаваться пустыми.

Код функций-членов pop () и push () гарантирует корректное управление вершиной

стека. Подобного рода гарантии — это одно из обстоятельств, которые делают ООП

надежным. Предположим, что вы решили создать отдельный массив для

представления стека и независимую переменную, представляющую индекс вершины стека. В этом

случае вы отвечаете за правильность кода при каждом создании нового стека. Без

защиты, предоставляемой закрытыми данными, всегда существует возможность сделать

ошибку и изменить данные нежелательным образом.

Давайте протестируем стек. Код в листинге 10.12 моделирует деятельность клерка,

обрабатывающего заказы на покупки, который берет их со стопки на столе, используя

характерный для стека алгоритм LIFO (last-in, first-out — последним пришел, первым

обслужен).

Листинг 10.12. stacker.срр

// stacker.срр -- тестирование класса Stack

#include <iostream>

#include <cctype> // или ctype.h

#include "stack.h"

int main ()

{

using namespace std;

Stack st; // создание пустого стека

char ch;

unsigned long po;

// A - добавление заказа, Р - обработка заказа, Q - завершение

cout « "Please enter A to add a purchase order, \n"

« "P to process a PO, or Q to quit.\n";

while (cin » ch && toupper(ch) != 'Q')

{

while (cin.get() != '\n')

continue;

if (lisalpha(ch))

{

cout « ' \a';

continue;

}

switch(ch)

{

case 'A':

case 'a': cout « "Enter a PO number to add: "; // запрос номера заказа

cin » po;

if (st.isfulK) )

cout « "stack already full\n"; // стек уже полон

else

Объекты и классы 529

st .push (po) ;

break;

case 'P':

case 'p': if (st. isempty () )

cout « "stack already empty\n"; // стек уже пуст

else {

st.pop(po);

cout « "PO #" « po « " popped\n"; // заказ вытолкнут

}

break;

}

cout « "Please enter A to add a purchase order,\n"

<< "P to process a PO, or Q to quit.Xn";

}

cout « "Bye\n";

return 0;

Небольшой цикл while в листинге 10.12, который избавляется от остатка строки,

пока что не является совершенно необходимым, однако он пригодится в

модифицированной версии программы, которая будет рассматриваться в главе 14. Ниже показан

пример запуска программы:

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

A

Enter a PO number to add: 17885

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

P

PO #17885 popped

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

A

Enter a PO number to add: 17 965

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

A

Enter a PO number to add: 18002

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

P

PO #18002 popped

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

P

PO #17965 popped

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

P

stack already empty

Please enter A to add a purchase order,

P to process a PO, or Q to quit.

Q

Bye

Массив объектов

Часто, как и в примерах со Stock, требуется создавать несколько объектов одного

класса. Можно создать отдельные объектные переменные, как это делалось до сих пор

в примерах настоящей главы, но больше смысла будет в создании массива объектов.

Это может выглядеть подобно прыжку в неизвестность, но фактически массив

объектов объявляется таким же способом, как и массивы любых стандартных типов:

Stock mystuff[4]; // создание массива из 4 объектов Stock

Вспомните, что программа всегда вызывает конструктор по умолчанию, когда

создает объекты класса без явной инициализации. Такое объявление требует либо

отсутствия у класса явно определенных конструкторов (при этом используются неявные,

ничего не делающие конструкторы), либо, как и в представленном случае — чтобы

был явно определен конструктор по умолчанию. Каждый элемент — mystuff [0],

mystuff [1] и т.д. — является объектом класса Stock, а потому может применяться с

методами Stock:

mystuff [0] .updateO; // применяет updateO к первому элементу

mystuff[3].show(); // применяет show() к 4-му элементу

const Stock \* tops = mystuff[2].topval(mystuff[1]) ;

// сравнивает 2-й и 3-й элементы и устанавливает tops

// в указатель на тот из них, у которого больше значение total\_val

Для инициализации элементов массива можно использовать конструктор. В этом

случае необходимо вызывать конструктор для каждого индивидуального элемента:

const int STKS = 4;

Stock stocks[STKS] = {

Stock("NanoSmart", 12.5, 20),

Stock("Boffo Objects", 200, 2.0),

Stock("Monolithic Obelisks", 130, 3.25),

Stock("Fleep Enterprises", 60, 6.5)

};

В приведенном коде применяется стандартная форма инициализации массива:

разделенный запятой список значений, заключенный в фигурные скобки. В таком случае

каждое значение представлено вызовом метода конструктора. Если класс имеет более

одного конструктора, для разных элементов можно использовать разные конструкторы:

const int STKS = 10;

Stock stocks[STKS] = {

Stock("NanoSmart", 12.5, 20),

StockO ,

Stock("Monolithic Obelisks", 130, 3.25),

};

В коде элементы stocks[0] Hstocks[2] инициализируются с помощью

конструктора Stock (const string & со, long n, double pr), a stock [1] — посредством

конструктора Stock (). Поскольку такое объявление инициализирует массив только

частично, оставшиеся семь членов инициализируются конструктором по умолчанию.

Объекты и классы 521

В листинге 10.9 эти принципы применяются в короткой программе, которая

инициализирует четыре элемента массива, отображает их содержимое и проверяет

элементы в поисках того, который имеет наибольшее значение totalval. Поскольку

total () сравнивает только два объекта за раз, для просмотра всего массива в

программе используется цикл for. Для отслеживания элемента с наибольшим значением

totalval применяется указатель на Stock. Код в этом листинге использует

заголовочный файл из листинга 10.7 и файл методов из листинга 10.8.

Листинг 10.9. usestock2. срр

// usestok2.срр — использование класса Stock

// Компилировать вместе с stock20.cpp

#include <iostream>

#include "stock20.h"

const int STKS = 4;

int main ()

{

// Создание массива инициализированных объектов

Stock stocks[STKS] = {

Stock("NanoSmart", 12, 20.0),

Stock("Boffo Objects", 200, 2.0),

Stock("Monolithic Obelisks", 130, 3.25),

Stock("Fleep Enterprises", 60, 6.5)

};

std::cout « "Stock holdings:\n";

int st;

for (st = 0; st < STKS; st+ + )

stocks[st].show();

// Установка указателя н первый элемент

const Stock \* top = &stocks[0];

for (st = 1; st < STKS; st ++)

top = &top->topval(stocks[st]);

// Теперь top указывает на самый ценный пакет акций

std::cout « "\nMost valuable holding:\n";

top->show() ;

return 0;

Ниже показан вывод программы из листинга 10.9:

Stock holdings:

Company: NanoSmart Shares: 12

Share Price: $20.000 Total Worth: $240.00

Company: Boffo Objects Shares: 200

Share Price: $2.000 Total Worth: $400.00

Company: Monolithic Obelisks Shares: 130

Share Price: $3.250 Total Worth: $422.50

Company: Fleep Enterprises Shares: 60

Share Price: $6.500 Total Worth: $390.00

Most valuable holding:

Company: Monolithic Obelisks Shares: 130

Share Price: $3.250 Total Worth: $422.50

Относительно листинга 10.9 следует отметить один момент: большая часть работы

приходится на проектирование класса. Когда оно завершено, написание программы

становится достаточно простым.

522 Глава 10

Между прочим, знание об указателе this позволяет заглянуть "за кулисы" C++.

Например, исходная реализация для Unix использовала утилиту с front, которая

выполняла преобразование программ на C++ в программы на С. Для поддержки

определений методов все, что нужно было сделать — это преобразовать определение метода

C++ вроде такого:

void Stock::show() const

{

cout « "Company: " « company

« " Shares: " << shares << f\nf

<< " Share Price: $" << share\_val

« " Total Worth: $" « total\_val << ' \n•;

}

в следующий код на языке С:

void show(const Stock \* this)

{

cout << "Company: " << this->company

« " Shares: " << this->shares « f\nf

<< " Share Price: $" << this->share\_val

« " Total Worth: $" « this->total\_val « ' \n' ;

}

To есть квалификатор Stock: : преобразуется в аргумент функции, который

представляет собой указатель на Stock, после чего этот указатель используется для доступа

к членам класса.

Аналогичным образом преобразуются вызовы функций наподобие следующего:

top.show ();

в такой вид:

show(stop);

В той же манере указателю this присваивается адрес вызывающего объекта.

(Реальные детали этого процесса могут оказаться более сложными.)