Классы в C++ являются естественным продолжением структуры struct в С.

* **Определения структур**

Структуры — это составные типы данных, построенные с использованием других типов. Рассмотрим следующее определение структуры:

**struct Time {**

**int hour; // 0-23**

**int minute; // 0-59**

**int second; // 0-59**

} ;

Ключевое слово struct начинает определение структуры. Идентификатор Time — тег (обозначение, имя-этикетка) структуры. Тэг структуры используется при объявлении переменных структур данного типа. В этом примере имя нового типа — Time. Имена, объявленные в фигурных скобках описания структуры — это элементы структуры. Элементы одной и той же структуры должны иметь уникальные имена, но две разные структуры могут содержать не конфликтующие элементы с одинаковыми именами. Каждое определение структуры должно заканчиваться точкой с запятой. Приведенное объяснение, как мы вскоре увидим, верно и для классов.

Определение Time содержит три элемента типа int — hour, minute и second (часы, минуты и секунды). Элементы структуры могут быть любого типа и одна структура может содержать элементы многих разных типов. Структура не может, однако, содержать экземпляры самой себя. Например, элемент типа Time не может быть объявлен в определении структуры Time. Однако, может быть включен указатель на другую структуру Time. Структура, содержащая элемент, который является указателем на такой же структурный тип, называется *структурой с самоадресацией.*

Предыдущее определение структуры данных не резервирует никакого пространства в памяти; определение только создает новый тип данных, который используется для объявления переменных. Переменные структуры объявляются так же, как переменные других типов. Объявление

**Time timeObject, timeArray[10], \*timePtr;**

объявляет timeObject переменной типа Time, timeArray — массивом с 10 элементами типа Time, a timePtr — указателем на объект типа Time.

**2.Доступ к элементам структуры**

Для доступа к элементам структуры (или класса) используются операции доступа к элементам — *операция точка* (.) и *операция стрелка* (->). Операция точка обращается к элементу структуры (или класса) по имени переменной объекта или по ссылке на объект. Например, чтобы напечатать элемент hour структуры timeObject используется оператор

**cout << timeObject.hour;**

Операция стрелка, состоящая из знака минус (-) и знака больше (>), записанных без пробела, обеспечивает доступ к элементу структуры (или класса) через указатель на объект. Допустим, что указатель timePtr был уже объявлен как указывающий на объект типа Time и что адрес структуры timeObject был уже присвоен timePtr. Тогда, чтобы напечатать элемент hour структуры timeObject с указателем timePtr, можно использовать оператор

**cout << timePtr->hour;**

Программа на рис. 1 создает определенный пользователем тип структуры Time с тремя целыми элементами: hour, minute и second. Программа определяет единственную структуру типа Time, названную dinnerTime, и использует операцию точка для присвоения элементам структуры начальных значений 18 для hour, 30 для minute и 0 для second. Затем программа печатает время в военном (24-часовом) и стандартном (12-часовом) форматах. Заметим, что функции печати принимают ссылки на постоянные структуры типа Time. Это является причиной того, что структуры Time передаются печатающим функциям по ссылке — этим исключаются накладные расходы на копирование, связанные с передачей структур функциям по значению, а использование const предотвращает изменение структуры типа Time функциями печати.





**3.Использование абстрактного типа данных Time с помощью класса**

Классы предоставляют программисту возможность моделировать объекты, которые имеют *атрибуты* (представленные как *данные-элементы)* и *варианты поведения* или *операции* (представленные как *функции-элементы).* Типы, содержащие данные-элементы и функции-элементы, обычно определяются в C++ с помощью ключевого слова class.

Функции-элементы иногда в других объектно-ориентированных языках называют *методами,* они вызываются в ответ на *сообщения,* посылаемые объекту. Сообщение соответствует вызову функции-элемента.

Когда класс определен, имя класса может быть использовано для объявления объекта этого класса. Рис. 6,2 содержит простое определение класса Time.

Определение нашего класса Time начинается с ключевого слова class. *Тело* определения класса заключается в фигурные скобки ({ }). Определение класса заканчивается точкой с запятой. Определение нашего класса Time, как и нашей структуры Time, содержит три целых элемента hour, minute и second.



**Рис. 6.2.** Простое определение **кпассЗЧте**

Остальные части определения класса — новые. Метки public: (открытая) и private: (закрытая ) называются *спецификаторами доступа к элементам.* Любые данные-элементы и функции-элементы, объявленные после спецификатора доступа к элементам public: (и до следующего спецификатора доступа к элементам), доступны при любом обращении программы к объекту класса Time. Любые данные-элементы и функции-элементы, объявленные после спецификатора доступа к элементам private: (и до следующего спецификатора доступа к элементам), доступны только функциям-элементам этого класса. Спецификаторы доступа к элементам всегда заканчиваются двоеточием (:) и могут появляться в определении класса много раз и в любом порядке. В дальнейшем в тексте нашей книги мы будем использовать записи спецификаторов доступа к элементам в виде public и private (без двоеточия).

Определение класса в нашей программе содержит после спецификатора доступа к элементам public прототипы следующих четырех функций-элементов: Time, *setTime*, printMilitary и printStandard. Это — *открытые функции-элементы* или *открытый интерфейс услуг класса.* Эти функции будут использоваться *клиентами* класса (т.е. частями программы, играющими роль пользователей) для манипуляций с данными этого класса.

Обратите внимание на функцию-элемент с тем же именем, что и класс. Она называется *конструктором* этого класса. Конструктор — это специальная функция-элемент, которая инициализирует данные-элементы объекта этого класса. Конструктор класса вызывается автоматически при создании объекта этого класса. Мы увидим, что обычно класс имеет несколько конструкторов; это достигается посредством перегрузки функции.

После спецификатора доступа к элементам private следуют три целых элемента. Это говорит о том, что эти данные-элементы класса являются доступными только функциям-элементам класса и, как мы увидим в следующей главе, «друзьям» класса. Таким образом, данные-элементы могут быть доступны только четырем функциям, прототипы которых включены в определение этого класса (или друзей этого класса). Обычно данные -элементы перечисляются в части private, а функции-элементы — в части public. Как мы увидим далее, можно иметь функции-элементы private и данные public; последнее не типично и считается в программировании дурным вкусом.

Когда класс определен, его можно использовать в качестве типа в объявлениях, например, следующим образом:

**Time sunset, // объект типа Time**

**arrayOfTimes[5], // массив объектов типа Time**

**\*pointerToTime, // указатель на объект типа Time**

**SdinnerTime = sunset; //ссылка на объект типа Time**

Имя класса становится новым спецификатором типа. Может существовать множество объектов класса как и множество переменных типа, например, такого, как int. Программист по мере необходимости может создавать новые типы классов. Это одна из многих причин, по которым C++ *является расширяемым языком.*

Программа на рис. 6.3 использует класс Time. Эта программа создает единственный объект класса Time, названный t. Когда объект создается, автоматически вызывается конструктор Time, который явно присваивает нулевые начальные значения всем данным-элементам закрытой части private. Затем печатается время в военном и стандартном форматах, чтобы подтвердить, что элементы получили правильные начальные значения. После этого с помощью функции-элемента setTime устанавливается время и оно снова печатается в обоих форматах. Затем функция-элемент setTime пытается дать данным-элементам неправильные значения и время снова печатается в обоих форматах.

Снова отметим, что данные-элементы hour, minute и second предваряются спецификатором доступа к элементам private. Эти закрытые данные-элементы класса обычно недоступны вне класса (но, как мы увидим в главе 7, друзья класса могут иметь доступ к закрытым элементам класса.) Глубокий смысл такого подхода заключается в том, что истинное представление данных внутри класса не касается клиентов класса. Например, было бы вполне возможно изменить внутреннее представление и представлять, например, время внутри класса как число секунд после полуночи. Клиенты могли бы использовать те же самые открытые функции-элементы и получать те же самые результаты, даже не осознавая произведенных изменений. В этом смысле, говорят, что реализация класса *скрыта* от клиентов. Такое *скрытие информации* способствует модифицируемости программ и упрощает восприятие класса клиентами.





В нашей программе конструктор Time просто присваивает начальные значение, равные 0, данным-элементам, (т.е. задает военное время, эквивалентное 12 AM). Это гарантирует, что объект при его создании находится в известном состоянии. Неправильные значения не могут храниться в данных-элементах объекта типа Time, поскольку конструктор автоматически вызывается при создании объекта типа Time, а все последующие попытки изменить данные-элементы тщательно рассматриваются функцией setTime.

Отметим, что данные-элементы класса не могут получать начальные значения в теле класса, где они объявляются. Эти данные-элементы должны получать начальные значения с помощью конструктора класса или им можно присваивать значения через функции.

Функция с тем же именем, что и класс, но со стоящим перед ней *символом тильда* (~), называется *деструктором* этого класса (наш пример не включает деструктор). Деструктор производит «завершающие служебные действия» над каждым объектом класса перед тем, как память, отведенная под этот объект, будет повторно использована системой.

Заметим, что функции, которыми класс снабжает внешний мир, предваряются меткой public. Открытые функции реализуют все возможности класса, необходимые для его клиентов. Открытые функции класса называют *интерфейсом* класса или *открытым интерфейсом.*

Отметим использование *бинарной операции разрешения области действия* (::) в каждом определении функции-элемента, следующем за определением класса на рис. 6.3. После того, как класс определен и его функции- элементы объявлены, эти функции-элементы должны быть описаны. Каждая функция-элемент может быть описана прямо в теле класса (вместо включение прототипа функции класса) или после тела класса. Когда функция-элемент описывается после соответствующего определения класса, имя функции предваряется именем класса и бинарной операцией разрешения области действия Поскольку разные классы могут иметь элементы с одинаковыми именами, операция разрешения области действия «привязывает» имя элемента к имени класса, чтобы однозначно идентифицировать функции-элементы данного класса.

Интересно, что функции-элементы printMilitary и printStandard не получают никаких аргументов. Это происходит потому, что функции-элементы неявно знают, что они печатают данные-элементы определенного объекта типа Time, для которого они активизированы. Это делает вызовы функций-элементов более краткими, чем соответствующие вызовы функций в процедурном программировании. Это уменьшает также вероятность передачи неверных аргументов, неверных типов аргументов или неверного количества аргументов.

* **Область действия класс и доступ к элементам**

**класса**

Данные-элементы класса (переменные, объявленные в определении класса) и функции-элементы (функции, объявленные в определении класса) имеют *областью действия класс.* Функции, не являющиеся элементом класса, имеют *областью действия файл.*

При области действия класс элементы класса непосредственно доступны всем функциям-элементам этого класса и на них можно ссылаться просто по имени. Вне области действия класс к элементам класса можно обращаться либо через имя объекта, либо ссылкой на объект, либо с помощью указателя на объект.

Функции-элементы имеют внутри класса *область действия функцию:* переменные, определенные в функции-элементе, известны только этой функции. Если функция-элемент определяет переменную с тем же именем, что и переменная в области действия класс, последняя делается невидимой в области действия функция. Такая скрытая переменная может быть доступна посредством операции разрешения области действия с предшествующим этой операции именем класса. Невидимые глобальные переменные могут быть доступны с помощью унарной операции разрешения области действия.

Операции, использованные для доступа к элементам класса, аналогичны операциям, используемым для доступа к элементам структуры. *Операция выбора элемента точка* (.) комбинируется для достуна к элементам объекта с именем объекта или со ссылкой на объект. *Операция выбора элемента стрелка* (->) комбинируется для доступа к элементам объекта с указателем на объект.

Программа на рис. 6.4 использует простой класс, названный Count, с открытым элементом данных х типа int и открытой функцией-элементом print, чтобы проиллюстрировать доступ к элементам класса с помощью операций выбора элемента. Программа создает три экземпляра переменных типа Count — counter, counterRef (ссылка на объект типа Count) и counterPtr (указатель на объект типа Count). Переменная counterRef объявлена, чтобы

ссылаться на counter, переменная counterPtr объявлена, чтобы указывать на counter. Важно отметить, что здесь элемент данных х сделан открытым просто для того, чтобы продемонстрировать способы доступа к открытым элементам. Как мы уже установили, данные обычно делаются закрытыми (private), чему мы и будем следовать в дальнейшем.



* **Отделение интерфейса от реализации**

Один из наиболее фундаментальных принципов разработки хорошего программного обеспечения состоит в отделении интерфейса от реализации. Это облегчает модификацию программ. Что касается клиентов класса, то изменения в реализации класса не влияют на клиента до тех пор, пока интерфейс класса, изначально предназначенный для клиента, остается неизменным (функции класса можно расширять за пределы исходного интерфейса).

Рисунок 6.5 разбивает программу на рис. 6.3 на ряд файлов. При построении программы на C++ каждое определение класса обычно помещается в *заголовочный файл,* а определения функций-элементов этого класса помещаются в *файлы исходных кодов* с теми же базовыми именами. Заголовочные файлы включаются (посредством **#include)** в каждый файл, в котором используется класс, а файлы с исходными кодами компилируются и компонуются с файлом, содержащим главную программу.

Программа на рис. 6.5 состоит из заголовочного файла **timel.h,** в котором объявляется класс **Time,** файла **timel.cpp,** в котором описываются функции- элементы класса **Time,** и файла **fig6\_5.cpp,** в котором описывается функция **main.** Выходные данные этой программы идентичны выходным данным программы на рис. 6.3.





Заметим, что объявление класса заключено в следующие директивы препроцессора (смотри главу 17):

**// Предотвращение многократного включения заголовочного файла #ifndef Т1МЕ1\_Н fdefine Т1МЕ1\_Н**

**#endif**

При построении больших программ в заголовочные файлы будут помещаться также и другие определения и объявления. Приведенные выше директивы препроцессора предотвращают включение кода между #ifndef и #endif, если определено имя Т1МЕ1\_Н. Если заголовок еще не включался в файл, то имя Т1МЕ1\_Н определяется директивой #define и операторы заголовочного файла включаются в результирующий файл. Если же заголовок уже был включен ранее, Т1МЕ1\_Н уже определен и операторы заголовочного файла повторно не включается. Попытки многократного включения заголовочного файла обычно случаются в больших программах с множеством заголовочных файлов, которые могут сами включать другие заголовочные файлы. Замечание: по негласному соглашению в приведенных выше директивах используется имя символической константы, представляющее собой просто имя заголовочного файла с символом подчеркивания вместо точки.

**5.Инициализация объектов класса: конструкторы**

После создания объекта его элементы могут быть инициализированы с помощью функции *конструктор.* Конструктор — это функция-элемент класса с тем же именем, что и класс. Программист предусматривает конструктор, который затем автоматически вызывается при создании объекта (создании экземпляра класса). *Данные-элементы класса не могут получать начальные значения в определении класса.* Они либо должны получить эти значения в конструкторе класса, либо их значения можно установить позже, после создания объекта. Конструкторы не могут указывать типы возвращаемых значений или возвращать какие-то значения.

Когда объявляется объект класса, между именем объекта и точкой с запятой можно в скобках указать *список инициализации элементов.* Эти начальные значения передаются в конструктор класса. Скоро мы увидим несколько примеров подобных *вызовов конструкторов.*

**6.Использование конструкторов с аргументами**

**по умолчанию**

Конструктор из timel.cpp (рис. 6.5) присваивает нулевые (т.е. соответствующие 12 часам по полуночи в военном формате времени) начальные значения переменным hour, minute и second. Конструктор может содержать значения аргументов по умолчанию. Программа на рис. 6.8 переопределяет функцию конструктор Time так, чтобы она включала нулевые значения аргументов по умолчанию для каждой переменной. Задание в конструкторе аргументов по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если в вызове конструктора не указаны никакие значения. Созданный программистом конструктор, у которого все аргументы — аргументы по умолчанию (или который не требует никаких аргументов), называется *конструктором с умолчанием,* т.е. конструктором, который можно вызывать без указания каких-либо аргументов.



Для каждого класса может существовать только один конструктор с умолчанием. В этой программе конструктор вызывает функцию-элемент setTime со значениями, передаваемыми конструктору (или значениями по умолчанию), чтобы гарантировать, что значение, предназначенное для hour, находится в диапазоне от 0 до 23, а значения для minute и second — в диапазоне от 0 до 59. Если значение выходит за пределы диапазона, оно устанавливается равным нулю с помощью setTime (это пример гарантии того, что данные- элементы будут в непротиворечивом состоянии).





**Рис. 6.8.** Использование конструктора с аргументами по умолчанию (часть 4 из 4)

**Использование деструкторов**

*Деструктор* — это специальная функция-элемент класса. Имя деструктора совпадает с именем класса, но перед ним ставится символ *тильда* (~). Это соглашение о наименовании появилось интуитивно, потому что, как мы увидим в последующих главах, операция тильда является поразрядной операцией дополнения, а по смыслу деструктор является дополнением конструктора.

Деструктор класса вызывается при уничтожении объекта — например, когда выполняемая программа покидает область действия, в которой был создан объект этого класса. На самом деле деструктор сам не уничтожает объект — он выполняет *подготовку завершения* перед тем, как система освобождает область памяти, в которой хранился объект, чтобы использовать ее для размещения новых объектов.

Деструктор не принимает никаких параметров и не возвращает никаких значений. Класс может иметь только один деструктор — перегрузка деструктора не разрешается.

* **Использование данных-элементов и функций-элементов**

Закрытые данные-элементы можно изменять только с помощью функций-элементов (и дружественных функций) класса. Типичным изменением могла бы быть корректировка баланса клиентов банка (например, закрытого элемента данных класса BankAccount) с помощью функции-элемента computelnterest.

Классы часто предусматривают открытые функции-элементы, позволяющие клиентам класса устанавливать (т.е. записывать) или получать (т.е. читать) значения закрытых данных-элементов. Эти функции можно не называть конкретно «set» или «get», но все же часто применяют именно эти названия. Более точно, функция-элемент, которая устанавливает элемент данных in- terestRate, должна была бы называться setlnterestRate, а функция-элемент, которая читает элемент данных interestRate, должна была бы называться getlnterestRate. Читающие функции обычно называют также функциями «запросов».

Казалось бы, что предоставление возможностей как установки, так и чтения, по существу, то же самое, что задание открытых данных-элементов. Однако в C++ существует одна тонкость, которая делает этот язык таким привлекательным для создания программного обеспечения. Если данные-элементы открытые, то они могут быть прочитаны или записаны по желанию с помощью любой функции в программе. Если данные-элементы закрытые, то открытая функция «get», позволяет другим функциям читать при желании эти данные, причем функция «get» управляет форматированием и отображением данных. Открытая функция «set» может, а более точно — должна тщательно анализировать любую попытку изменить значение закрытого элемента данных. Это должно обеспечивать гарантию, что новое значение соответствует этому элементу данных. Например, должны быть отвергнуты попытки установить день месяца, равный 37, отрицательный вес человека, численную величину символьного значения, значение оценки, равное 185, при шкале оценок от 0 до 100 и т.д.

Клиенты класса должны быть уведомлены о попытке присвоить данным- элементам неверные значения. По этой причине функции записи данных- элементов класса часто оформляются так, чтобы они возвращали значения, указывающие на то, что была совершена попытка присвоить объекту класса неверное значение. Это дает возможность клиентам класса проверить значения, возвращаемые функцией записи, чтобы определить, является ли объект, которым они манипулируют, допустимым, и предпринять соответствующие действия, если объект недопустимый. В упражнениях вас попросят изменить программу на рис. 6.10 так, чтобы возвращать соответствующие значения ошибок из записывающих функций.

Программа на рис. **6.10** расширяет наш класс **Time** так, чтобы он включал функции чтения и записи закрытых данных-элементов **hour, minute** и **second.** Функции записи жестко управляют установкой данных-элементов. Попытки задать любым данным-элементам неправильные значения вызывают присваивание этим данным-элементам нулевых значений (и, таким образом, приведение данных-элементов в непротиворечивое состояние). Каждая функция чтения просто возвращает соответствующее значение данных-элементов. Программа сначала использует функции записи, чтобы задать правильные значения закрытым данным-элементам объекта **t** класса **Time,** затем использует функцию чтения, чтобы вывести эти значения на экран. Далее функции записи пытаются задать элементам **hour** и **second** неправильные значения, а элементу **minute** — правильное, и затем функции чтения направляют эти значения на экран. Результат подтверждает, что неправильные значения вызывают установку данных-элементов в нулевое состояние. В итоге программа устанавливает время 11:58:00 и прибавляет 3 минуты при вызове функции **incrementMinutes.** Функция **incrementMinutes** не является элементом класса; поэтому она использует функции-элементы записи и чтения для соответствующего увеличения элемента **minute.** Это функционирует правильно, но снижает производительность из-за многократных вызовов функций. В следующей главе мы обсудим запись дружественных функций как средства устранения этого недостатка.

Функции записи особенно важны с точки зрения разработки программного обеспечения, поскольку они могут производить проверку правильности данных. Функции записи и чтения имеют и другие достоинства при разработке программного обеспечения.







