Теоретическая часть второго домашнего задания

0. Повторите и перечислите какие типы памяти есть в С, как и когда на них выделяются переменные , их время жизни.

Областью действия (видимости) имени называется часть программы, в пределах которой можно использовать имя. Для автоматической (автоматические, локальные, **переменные** создаются при входе в функцию и уничтожаются при выходе из неё) переменной, объявляемой в начале функции, областью видимости является эта функция. Важно, что локальные переменные с одинаковыми именами, объявленные в разных функциях, не имеют никакого отношения друг к другу. Это также справедливо и для параметров функций, которые по сути являются локальными переменными.

Рассмотрим важные для понимания примеры объявлений переменных и функций:

```
main( ) { … }

int sp = 0;
double val[MAXVAL];

void push(double f) { … }
double pop(void) { … }
```

В языке С область видимости внешней переменной или функции распространяется от точки, в которой она объявлена, до конца компилируемого файла. Таким образом, в нашем примере переменные «*sp*» && «*val[]*» видны внутри функций «*push*» && «*pop*», но ни переменные, ни функции не видны в функции «*main*».

Если необходимо обратиться к внешней переменной до ее определения или если она определенная в другом файле исходного кода (см. модульное программирование), то надо вставить объявление с ключевым словом: *extern*.

Важно понимать различие между объявлением и определением внешней переменной. Объявлением сообщает её определенные свойства (тип), а определение выделяет место в памяти для ее хранения. Во всех файлах программы должно быть в общей сложности не более одного определения (где должна быть инициализация) внешней переменной.

(*источник: Керниган, Ритчи «Язык программирования С», стр. 93*)

1. Какие способы управления областями видимости в языке С мы знаем?

Для управления областями видимости мы знаем несколько основных команд. Во-первых, мы можем рассмотреть пример, заданный в (0) части теоретического домашнего задания. Там использовалось ключевое слово *external*. Также областью видимости можно управлять, заключая объявление переменных (теперь уже локальных) внутрь некоторой *функции( ) { … }* . В таком случае они должны будут прекратить свою жизнь при выходе из функции и возвращать указатель на них вне функции опасно для жизни.

Также существуют *статические переменные*. Если объявление внешней переменной или функции содержит слово *static,* то ее область действия ограничивается данным файлом исходного кода — от точки объявления и до конца. Это позволяет избежать конфликта, если в других файлах программы будут употребляться схожие имена.

Еще одной особенностью статических переменных является то, что при применении к внутренним переменным функций статические переменные продолжают существовать и по завершении функции, это является средством постоянного хранения скрытой информации внутри одной функции.

(*источник: Керниган, Ритчи «Язык программирования С», стр. 96*)

2. Какие способы управления областями видимости в языке C++ вы знаете? Что такое **namespace**?

Здесь реализуется схожий механизм с использованием команды static, то есть область видимости ограничивается лишь заданным файлом. Однако здесь также есть ключевое слово: namespace - пространство имен. С этой командой стоит работать осторожней. Потому что эта команда будет вызывать ту функцию, которую найдет компилятор, а это не всегда то, что мы ожидаем, так что стоит задавать это явно.

*(*источник: конспект с семинара по программированию 2*)*

В стандарте C++ вместо термина файл используется термин единица трансляции. Файловая модель - это не единственный способ организации информации в компьютере. Для простоты в книге (и конспекте) будет применяться термин «файл», но стоит помнить, что под этим также понимается и «единица трансляции».

Категории хранения влияют на то, как информация может совместно использоваться разными файлами. В языке C++11 есть 4 основных схемы хранения данных, они отличаются между собой продолжительностью хранения:

— **Автоматическая продолжительность хранения.** Переменные, объявленные внутри определения функции - включая параметры функции - имеют автоматическую продолжительность хранения. Они создаются, когда выполнение программы входит в функцию или блок, а после выхода используемая переменными память освобождается. В C++ существуют два вида автоматических переменных.

— **Статическая продолжительность хранения.** Переменные, объявленные за пределами определения функции либо с использованием ключевого слова static, имеют статическую продолжительность хранения. Они существуют в течение всего времени выполнения программы. В языке C++ существуют три вида переменных со статической продолжительностью хранения.

— **Потоковая продолжительность хранения (C++11)**. В наши дни многоядерные процессоры распространены практически повсеместно. Такие процессоры способны поддерживать множество выполняющихся задач одновременно. Это позволяет программе разделить вычисления на отдельные потоки, которые могут быть обработаны параллельно. Переменные, объявленые с ключевым словом thread_local, хранятся на протяжении времени существования содержащего их потока.

— **Динамическая продолжительность хранения.** Память, выделяемая операцией new, сохраняется до тех пор, пока она не будет освобождена с помощью операции delete или до завершения программы, смотря какое из событий наступит раньше. Эта память имеет динамическую продолжительность хранения и часто называется свободным хранилищем или … кучей (heap).

Касательно областей видимости наблюдается схожая с C ситуация. Повторим описание области видимости.

Область видимости определяет доступность имени в пределах файла. Например, переменная , определенная в функции, может быть использована только в этой функции* , но не в какой-либо другой, в то время как переменная, определенная в файле до определения функций, может применяться во всех функциях.

Связывание описывает, как имя может совместно использоваться разными файлами, а имя с внутренним связыванием — функциями внутри одного файла. Имена автоматических переменных не имеют никакого связывания, т.к. они не являются разделяемыми. Переменная с локальной видимостью видна лишь внутри блока { }.

Также существуют регистровые переменные. Ключевое слово *register* было первоначально введено в языке C, чтобы рекомендовать компилятору использовать для хранения автоматической переменной регистр центрального процессора. Идея заключалась в том, что это ускоряло доступ к переменной.

В языке C++ до версии C++11 использование было аналогичным, с тем отличием, что эта рекомендация была обобщена и начала указывать на тот факт, что переменная интенсивно используется, и, возможно, компилятору стоит уделить ей особенное внимание. В версии C++11 ключевое слово *register* остается просто способом идентифицировать переменную как автоматическую. Поскольку это может применяться только с переменными, которые будут автоматическими в любом случае, одна из причин использования этого ключевого слова — указать, что нужна автоматическая переменная,

Описание хранения	Продолжительность	Область видимости	Связывание	Способ объявления
Автоматическая	Автоматическая	Блок	Нет	В блоке
Регистровая	Автоматическая	Блок	Нет	В блоке, с использованием ключевого слова `register`
Статическая без связывания	Статическая	Блок	Нет	В блоке, с использованием ключевого слова `static`
Статическая с внешним связыванием	Статическая	Файл	Внешнее	Вне всех функций
Статическая с внутренним связыванием	Статическая	Файл	Внутреннее	Вне всех функций, с использованием ключевого слова `static`

возможно, с тем же самым именем, что и у внешней переменной. Ранее — *auto*.

Здесь мы можем наблюдать перегрузку ключевого слова static.

Элементы статических массивов и структур устанавливаются в 0 по умолчанию. Кроме того, все статические переменные обладают следующей особенностью инициализации: все биты не инициализированной статической переменной устанавливаются в 0. И такая переменная называется инициализированной 0.

В C++ есть новая операция - разрешение контекста « *::* ». Если поместить эту операцию перед именем переменной, будет использоваться глобальная версия этой переменной.

Спецификаторы класса хранения: *register, static, extern, thread_local, mutable.* Также есть CV - квалификаторы: *const, volatile.* Ключевое слово const указывает, что переменная после ее инициализации не может быть изменена программой. Ключевое слово volatile указывает, что значение в ячейке памяти может быть изменено, даже если в коде программы нет ничего такого, что может модифицировать ее содержимое. А вот с *mutable* все несколько более многогранно: с его помощью можно указать, что отдельный член структуры или класса может быть изменен, даже если переменная типа структуры (или класса) объявлена со спецификатором *const*.

Имена в C++ могут относиться к переменным, функциям, структурам, перечислениям, классам, а также членам классов и структур. По мере увеличения размеров программных объектов возрастает вероятность конфликта имен. В частности, при использовании нескольких библиотек может возникать конфликт мет тех же классов List, Tree & Node. Опишем некоторую возможную проблему, называемую проблемой пространства имен: может потребоваться использовать класс List из одной библиотеки и класс Tree из другой, при этом каждый из них ожидает взаимодействия с собственной версией класса Node.

Стандарт C++ предоставляет средства пространства имен, которые обеспечивают более совершенное управление областью видимости имен. Здесь может быть полезным некоторое знакомство с терминологией пространств имен:

Декларированная область - это область, в которой могут делаться объявления. Например, глобальная переменная может быть объявлена вне всех функций, тогда

декларированной областью для нее будет являться файл. Если это сделать внутри функции, то ее декларированной областью будет соответсвующий блок.

Потенциальная область видимости переменной начинается с точки объявления и простирается до конца ее декларированной области объявления. Таким образом потенциальная область является более ограниченной.

Правила языка C++, касающиеся глобальных и локальных переменных, определяют своего рода иерархию пространств имен. В каждой декларированной области могут быть объявлены имена, не зависящие от имен, объявленных в других декларированных областях.

В настоящее время в C++ появилась возможность создавать именованные пространства имен за счет определения декларированной области нового вида, одним из основных назначений которых является предоставления места для объявления имен. Здесь ключевым словом является *namespace*. Пространства имен являются открытыми, это означает, что можно добавлять имена в уже существующие пространства:

```
namespace Jill {
        char * goose(const char * );
}
```

Для использования имен из определенных пространств на необходим какой-то способ доступа к именам заданного пространства имен. Простейший способ предусматривает использование операции разрешения контекста:

```
Jill::goose(); //использование соотв. функции из заданного пространства имен.
```

Для удобства есть объявления и директивы *using*. Объявление using добавляет имя в локальную или глобальную декларированную область, то есть мы работаем только с одним элементом пространна имен в некоторой области:

```
using Jill::goose(); // в локальном или глобальном смысле мы после этого всегда исп.
```

Директива *using* уже делает доступными все имена. При этом принцип работы вновь как с локальными и глобальными переменными.

Интересно, что объявления пространств имен могут быть вложенными, тогда для доступа к внутренним именам нужно писать полный путь через « *::* » (стр. 470). Также можно пользоваться директивами и объявлениями using внутри пространств имен.

Можно создавать псевдонимы для для пространства имен (namespace fw = fire_wall).

*(*источник: Пата «Язык программирования C++. Лекции и упражнения.», стр. 435+*)*

3. Для чего нужны ключевые слова **public:** и **private:** ?

Одним из основных принципов ООП является инкапсуляция. Этот механизм позволяет ограничить доступ одних компонент программ к другим.

Ключевыми словами здесь являются *private* и *public*. Эти метки позволяют управлять доступом к членам класса (в C++ также существует третье ключевое слово *protected*). Любая программа, которая использует объект определенного класса, может иметь непосредственный доступ к членам из раздела *public*. Доступ к членам объекта из раздела *private* программа может получить только через открытые функции-члены из раздела *public* (или через дружественные функции). Т.е. Открытые функция выступают своего рода посредниками между пользователем и приватным сектором.

Рассмотрим некоторые стандартные договоренности по объявлениям членов класса согласно принципам ООП. Единицы данных обычно размещаются в разделе private. Функции-члены, которые образуют интерфейс класса, размещаются в разделе *public*; в противном случае вызвать эти функции из программы не удастся. По умолчанию спецификатор доступа к объектам является *private*.

*(*источник: Пата «Язык программирования C++. Лекции и упражнения.», стр. 489*)*

4. Что особенного в конструкторе и деструкторе класса, что их отличает от других методов.

Существует ряд определенных стандартных функций, называемых *конструкторами* и *деструкторами*, которыми обычно должен быть снабжен класс. Одна из целей C++ состоит в том, чтобы сделать использование объектов классов подобным применению стандартных типов. И преградой на этом пути служит то, что данным класса разрешен только закрытий доступ, а это означает, что нужны специальные функции, способные до них добраться.

Единственным способом обойти трудность определения последовательности действий при инициализации переменных является автоматическая инициализация объектов при их создании. Для этого в C++ предлагаются специальные функции-члены, называемые конструкторами класса, которые предназначены для создания новых объектов и присваивания значений их членам-данным. Если говорить точнее, то C++ регламентирует имя для таких функций-членов, а также синтаксис их вызова, тогда как наша задача — написать определение этого метода. Имя метода конструктора совпадает с именем класса. Прототип и заголовок конструктора обладают интересным свойством: несмотря на то, что конструктор не имеет возвращаемого значения, они не объявляются с типом void. Разница в том, что программа автоматически вызовет конструктор при объявлении объекта.

Конструктор по умолчанию - это *конструктор*, который используется для создания объекта, когда не предоставлены явные инициализирующие значения.

Stock( ); // конструктор по умолчанию к классу Stock

Совет
При проектировании класса обычно должен быть предусмотрен конструктор по умолчанию, который неявно инициализирует все переменные-члены класса.

В случае использования конструктора для создания объекта программа отлеживает этот объект до момента его исчезновения. В этот момент программа автоматически вызывает специальную функцию-член *деструктор*. Он служит благой цели: соскребает мусор с окраин города, например, если в конструкторе используется функция new, то деструктор должен обратиться к delete для освобождения выделенной памяти.

~Stock( ); // деструктор по умолчанию к классу Stock

Подобно конструктору, деструктор не имеет ни возвращаемого значения, ни объявленого типа. Однако в отличие от конструктора деструктор не должен иметь аргументы. В случае, когда он не имеет обязанностей, его можно кодировать как функцию, которая ничего не делает. Однако для того, чтобы видеть ее вызов можно ее заполнить:

```
Stock::~Stock( ) {
        cout << «Bye, » << company << «!\n»;
}
```

Деструктор вызывается автоматически при уничтожении объекта класса, поэтому он должен существовать. Если этого не предусмотреть, компилятор неявно задаст конструктор по умолчанию и, если обнаружит код, который ведет к уничтожению объекта, также неявно задаст деструктор.

*(*источник: Пата «Язык программирования C++. Лекции и упражнения.», стр. 500*)*

5. Для чего нужды операторы **new** и **delete**?

Эти операторы связаны с динамическим выделением памяти. Динамическая память может выделяться в одной функции и освобождаться в другой. В отличие от автоматической памяти, динамическая память не подчиняется схеме LIFO. Порядок выделения и освобождения памяти определяется по тому, когда и как применяются операции *new* и *delete*.

Несмотря на то, что концепции схем хранения неприменимы к динамической памяти, они применимы к автоматическим и статистическим переменным-указателям, используемым для отслеживания динамической памяти. Например:

```
float * p_fees = new float [20] ;
```

Некоторое количество памяти (лучше использовать то, какой размер знаешь) остаются занятыми до тех пор, пока операция *delete* не разлучит их. Однако этот указатель перестает существовать по выходе из блока. Если требуется возможность работы с другими функциями, то задавать нужно глобально, а в самой функции использовать *external*. Рассмотрим инициализацию с помощью new.

Если требуется создать и инициализировать хранилище для одного из встроенных скалярных типов, необходимо указать имя типа и инициализирующее значение, заключенное в круглых скобках:

```
int *pi = new int (6); // *pi устанавливается в 6.
double * pd = new double (99.99); // *pd устанавливается в 99.99
```

Синтаксис с круглыми собаками также может использоваться с классами, которые имеют подходящие конструкторы, но об этом позже (или нет).

Для инициализации обычной структуры или массива необходим стандарт C++11 и фигурные скобки списковой инициализации. Новый стандарт позволяет следующее:

```
struct where {double x; double y; double z;};
where * one = new where {2.5, 5.3, 7.2}; // C++11
int * ar = new int [4] {1, 2, 3, 4}; // C++11
```

Также можно применять инициализацию с помощью фигурных скобок для переменных с одиночным значением:

```
int *pin = new int { }; // !TODO
double * pdo = new double {99.99};
```

*(*источник: Пата «Язык программирования C++. Лекции и упражнения.», стр. 457*)*

new — оператор языка программирования C++, обеспечивающий выделение динамической памяти в куче. За исключением формы, называемой «размещающей формой new», new пытается выделить достаточно памяти в куче для размещения новых данных и, в случае успеха, возвращает адрес выделенного участка памяти. Однако, если *new* не может выделить память в куче, то он передаст (*throw*) исключение типа «*std::bad_alloc*». Это устраняет необходимость явной проверки результата выделения. После встречи компилятором ключевого слова new им генерируется вызов конструктора класса.

*(*источник: wikipedia*)*

6. Что такое и для чего нужен указатель **this**?

До сих пор каждая функция-член класса имела дело только с одним объектом: тем, который ее вызывал. Однако иногда методу может понадобиться иметь дело с двумя объектами и для этого обращаться к любопытному указателю по имени *this*.

Иногда объявление класса может включать в себя отображение данных, но при этом ему может недоставать аналитических возможностей для того, чтобы достать какую-то внутреннюю информацию. Самый простой способ сообщить программе о хранимых данных - это предусмотреть методы, возвращающие эти данные, например:

```
class Stock {
private:
        …
        double total_val;
        …
public:
        double total( ) const { return total_val; }
        …
};
```

Однако можно воспользоваться другим подходом, который поможет разобраться с указателем *this*. Это весьма крупная конструкция, так что будьте внимательны.

Подход заключается в определении функции-члена, которая будет просматривать два объекта класса и возвращать ссылку на больший из них. Попытка реализовать эту идею вызовет некоторые интересные вопросы, которые мы сейчас рассмотрим.

Во-первых, как написать функцию, работающую с двумя объектами с целью их сравнения? Пусть мы ее назвали topval (). stock1.topval( ) обращается к данным объекта stock1, stock2.topval( ) обращается к данным объекта stock2. Если нужно, чтобы метод сравнивал два объекта, второй объект потребуется передать в виде аргумента. Для эффективности можно передавать по ссылке. То есть метод topval( ) должен принимать аргумент типа const Stock &.

Во-вторых, как результат метода будет передаваться в вызывающую программу? Самый прямой путь - заставить метод возвращать ссылку на объект, который имеет большее значение total_val. Таким образом, метод сравнения двух объектов будет иметь следующий прототип:

```
const Stock & topval (const Stock & s) const;
```

Эта функция имеет неявный доступ к одному объекту и явный - ко второму, и она возвращает ссылку на один из двух объектов. Слово const внутри скобок указывает, что функция не будет модифицировать объект, к которому получает явный доступ, а слово const, которое следует за скобками, устанавливает, что функция не будет изменяться объект, на который ссылается неявно. Тип возврата функции также является ссылкой const.

Предположим, что мы хотим сравнить два объекта Stock — stock1 и stock2 — и присвоить объекту top тот из них, которое имеет наибольшее значения total_val.

```
top = stock1.topval( stock2 ); // верны оба варианта
top = stock2.topval( stock1 );
```

Решение проблемы, обозначенной в дополнении, в C++ заключается в применении специального указателя this. Он указывает на объект, который использован для вызова функции-члена (обычно this передается методу в качестве скрытого аргумента). Таким образом, вызов *stock1.topval(stock2)* устанавливает значение this равным адресу объекта stock1 и делает его доступным методу topval( ). Аналогичным образом, вызов *stock2.topval(stock1)* устанавливает значение this равным адресу объекта stock2. Вообще все методы класса получают указатель this, равный адресу объекта, который вызвал метод. Фактически total_val внутри total( ) является сокращенной нотаций *this->total_val*. *(операция -> используется для доступа к членам структуры через указатель на нее)

Однако то, что необходимо вернуть из метода — это не *this*, поскольку *this* представляет собой адрес объекта. Вам нужно вернуть сам объект, а это обозначается выражением **this.* (операция * к указателю дает значение, на которое оно указывает).

*(*источник: Пата «Язык программирования C++. Лекции и упражнения.», стр. 514*)*

7. Что такое **std::cout** и **std::cin**?

С помощью нового оператора разрешения контекста это является командой компьютеру: использовать функцию *cout & cin* из пространства имен под псевдонимом std. В данном пространстве имен *cout (Console Output, консольный вывод)* и *cin (Console Input, консольный ввод)* позволяют пользователю взаимодействовать с программой.

Между тем, пока рассмотрим реализацию `topval()`. Она порождает небольшую проблему. Вот часть реализации, иллюстрирующая проблему:

```
const Stock & Stock::topval(const Stock & s) const
{
  if (s.total_val > total_val)
       return s;          // объект-аргумент
  else
       return ?????;      // вызывающий объект
}
```

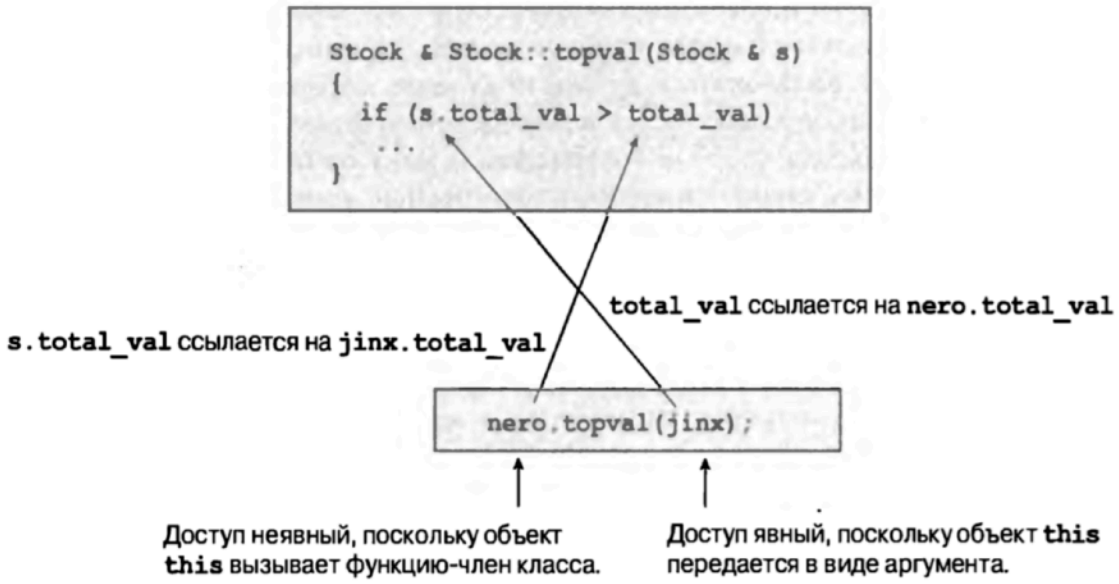

Рис. 10.3. *Доступ к двум объектам из функции-члена*

Здесь `s.total_val` — это суммарное значение объекта, переданное в виде аргумента, а `total_val` — суммарное значение объекта, которому сообщение передается. Если `s.total_val` больше `total_val`, то функция возвращает `s`. В противном случае она возвратит объект, использованный для вызова метода. (В терминологии ООП — это объект, которому передано сообщение `topval`.) Существует одна проблема: на чем вызывать объект? Если сделать вызов `stock1.topval(stock2)`, то `s` — это ссылка на `stock2` (т.е. псевдоним для `stock2`), но псевдонима для `stock1` не существует.

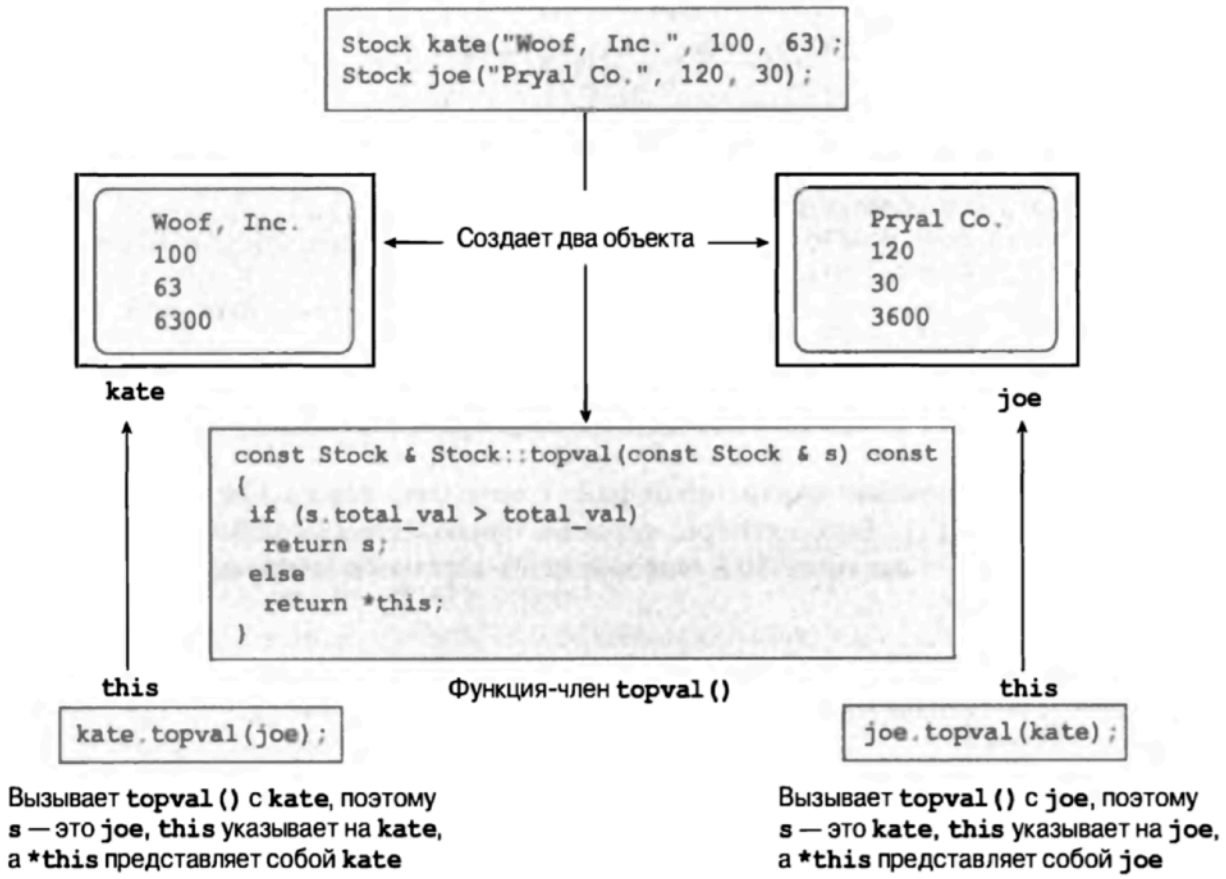

Рис. 10.4. *this указывает на вызвавший объект*

Теперь можно завершить определение метода, используя *this в качестве псевдонима вызвавшего объекта:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s;           // объект-аргумент
   else
      return *this;       // вызывающий объект
}
```

Тот факт, что возвращаемое значение представляет собой ссылку, означает, что возвращаемый объект является тем же самым объектом, который вызвал данный метод, а не копией, переданной механизмом возврата. В листинге 10.7 приведен новый заголовочный файл.

На заметку!

Каждая функция-член, включая конструкторы и деструкторы, имеет указатель this. Специфическим свойством this является то, что он указывает на вызывающий объект. Если метод нуждается в получении ссылки на вызвавший объект в целом, он может использовать выражение *this. Применение квалификатора const после скобок с аргументами заставляет трактовать this как указатель на const; в этом случае вы не можете использовать this для изменения значений объекта.