Федеральное агентство связи

Хабаровский институт инфокоммуникаций (филиал)

Государственное образовательное учреждение

высшего профессионального образования

«Сибирский государственный университет телекоммуникаций и информатики»

Факультета «Инфокоммуникаций, радиосвязи и мультимедиа»

**Курсовая работа**

по дисциплине: «Объектно-ориентированное программирование»

на тему: «Перегрузка операторов в языке С++»

Вариант №18.

Выполнил: .

Проверил: .

Хабаровск 2011 г.

**СОДЕРЖАНИЕ**

Введение3

1. Теоретическая часть5

1.1. ООП как развитие идей структурного подхода5

1.2. Свойства ООП 8

1.3. Описание класса10

1.4. Описание объектов 14

1.5. Указатель *this* 16

1.6. Конструкторы 17

1.7. Статические элементы класса 22

1.8. Дружественные функции и классы 24

1.9. Деструкторы 26

1.10. Перегрузка операций 28

1.11. Указатели на элементы классов 32

2. Практическая часть34

2.1. Описание полей класса35

2.2. Функции – члены класса и дружественные функции36

2.3. Разработка текста программы37

2.4. Листинг программы39

2.5. Результат выполнения программы44

Заключение45

Список используемой литературы46

**Введение**

ООП возникло в результате развития идеологии процедурного программирования, где данные и подпрограммы (процедуры, функции) их обработки формально не связаны. Для дальнейшего развития объектно-ориентированного программирования часто большое значение имеют понятия события (так называемое событийно-ориентированное программирование) и компонента (компонентное программирование, КОП).

Формирование КОП от ООП произошло, как случилось формирование модульного от процедурного программирования: процедуры сформировались в модули — независимые части кода до уровня сборки программы, так объекты сформировались в компоненты — независимые части кода до уровня выполнения программы. Взаимодействие объектов происходит посредством сообщений. Результатом дальнейшего развития ООП, по-видимому, будет агентно-ориентированое программирование, где агенты — независимые части кода на уровне выполнения. Взаимодействие агентов происходит посредством изменения среды, в которой они находятся.

Языковые конструкции, конструктивно не относящиеся непосредственно к объектам, но сопутствующие им для их безопасной (исключительные ситуации, проверки) и эффективной работы, инкапсулируются от них в аспекты (в аспектно-ориентированном программировании). Субъектно-ориентированное программирование расширяет понятие объект посредством обеспечения более унифицированного и независимого взаимодействия объектов. Может являться переходной стадией между ООП и агентным программирование в части самостоятельного их взаимодействия.

Первым языком программирования, в котором были предложены принципы объектной ориентированности, была Симула. В момент своего появления (в 1967 году), этот язык программирования предложил поистине революционные идеи: объекты, классы, виртуальные методы и др., однако это всё не было воспринято современниками как нечто грандиозное. Тем не менее, большинство концепций были развиты Аланом Кэйем и Дэном Ингаллсом в языке Smalltalk. Именно он стал первым широко распространённым объектно-ориентированным языком программирования.

В настоящее время количество прикладных языков программирования (список языков), реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам. В области системного программирования до сих пор применяется парадигма процедурного программирования, и общепринятым языком программирования является язык C. Хотя при взаимодействии системного и прикладного уровней операционных систем заметное влияние стали оказывать языки объектно-ориентированного программирования. Например, одной из наиболее распространенных библиотек мультиплатформенного программирования является объектно-ориентированная библиотека Qt, написанная на языке C++.

1. **Теоретическая часть**
   1. **ООП как развитие идей структурного подхода**

В окончательном виде любая программа представляет собой набор инструкций процессора. Все, что написано на любом языке программирования - более удобная, упрощенная запись этого набора инструкций, облегчающая написание, отладку и последующую модификацию программы. Чем выше уровень языка, тем в более простой форме записываются одни и те же действия.

С ростом объема программы становится невозможным удерживать в памяти все детали, и становится необходимым структурировать информацию, выделять главное и отбрасывать несущественное. Этот процесс называется повышением степени абстракции программы.

Для языка высокого уровня первым шагом к повышению абстракции является использование функций, позволяющее после написания и отладки функции отвлечься от деталей ее реализации, поскольку для вызова функции требуется знать только ее интерфейс. Если глобальные переменные не используются, интерфейс полностью определяется заголовком функции.

Следующий шаг - описание собственных типов данных, позволяющих структурировать и группировать информацию, представляя ее в более естественном виде. Например, все разнородные сведения, относящиеся к одному виду товара на складе, можно представить с помощью одной структуры.

Для работы с собственными типами данных требуются специальные функции. Естественно сгруппировать их с описанием этих типов данных в одном месте программы, а также по возможности отделить от ее остальных частей. При этом для использования этих типов и функций не требуется полного знания того, как именно они написаны - необходимы только описания интерфейсов. Объединение в модули описаний типов данных и функций, предназначенных для работы с ними, со скрытием от пользователя модуля несущественных деталей является дальнейшим развитием структуризации программы.

Все три описанных выше метода повышения абстракции преследуют цель упростить структуру программы, то есть представить ее в виде меньшего количества более крупных блоков и минимизировать связи между ними. Это позволяет управлять большим объемом информации и, следовательно, успешно отлаживать более сложные программы.

Введение понятия класса является естественным развитием идей модульности. В классе структуры данных и функции их обработки объединяются. Класс используется только через его интерфейс - детали реализации для пользователя класса не существенны.

Идея классов отражает строение объектов реального мира - ведь каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами, свойствами и поведением. Программы в основном предназначены для моделирования предметов, процессов и явлений реального мира, поэтому удобно иметь в языке программирования адекватный инструмент для представления моделей.

Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы (напомню, что тип задает внутреннее представление данных в памяти компьютера, множество значений, которое могут принимать величины этого типа, а также операции и функции, применяемые к этим величинам).

Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом. Интерфейсом класса являются заголовки его открытых методов. Таким образом, класс как модель объекта реального мира является черным ящиком, замкнутым по отношению к внешнему миру.

Идея классов является основой объектно-ориентированного программирования (ООП). Основные принципы ООП были разработаны еще в языках Simula-67 и Smalltalk, но в то время не получили широкого применения из-за трудностей освоения и низкой эффективности реализации. В С++ эти концепции реализованы эффективно и непротиворечиво, что и явилось основой успешного распространения этого языка и внедрения подобных средств в другие языки программирования.

Идеи ООП не очень просты для практического использования (их неграмотное применение приносит гораздо больше вреда, чем пользы), а освоение существующих стандартных библиотек требует времени и высокого уровня первоначальной подготовки.

Конкретные переменные типа данных "класс" называются экземплярами класса, или объектами. Объекты взаимодействуют между собой, посылая и получая сообщения. Сообщение - это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций. Таким образом, с помощью ООП легко реализуется так называемая "событийно-управляемая модель", когда данные активны и управляют вызовом того или иного фрагмента программного кода.

* 1. **Свойства ООП**

Основными свойствами ООП являются инкапсуляция, наследование и полиморфизм.

Объединение данных с функциями их обработки в сочетании со скрытием ненужной для использования этих данных информации называется инкапсуляцией (encapsulation). Инкапсуляция повышает степень абстракции программы: данные класса и реализация его функций находятся ниже уровня абстракции, и при написании программы информация о них не нужна. Кроме того, инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним (например, при необходимости сменить способ хранения данных с массива на стек).

Инкапсуляция позволяет использовать класс в другом окружении и быть уверенным, что класс не испортит не принадлежащие ему области памяти, а также создавать библиотеки классов для применения во многих программах.

Наследование - возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы. Выделение общих черт различных классов в один класс-предок является мощным механизмом абстракции - ведь и любая наука начинается с абстрагирования и классификации, которые помогают справиться со сложностью рассматриваемой предметной области.

Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные - на ветвях и листьях. В С++ каждый класс может иметь сколько угодно потомков и предков. Иногда предки называются надклассами или суперклассами, а потомки - подклассами или субклассами.

Третьим китом, на котором стоит ООП, является полиморфизм - возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

Понятие полиморфизма используется в С++ весьма широко. Простым примером полиморфизма может служить перегрузка функций, когда из нескольких вариантов выбирается наиболее подходящая функция по соответствию ее прототипа передаваемым параметрам. Другой пример - использование шаблонов функций, когда один и тот же код видоизменяется в соответствии с типом, переданным в качестве параметра. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

Использование при программировании понятий, более близких к предметной области, благодаря представлению программы в терминах поведения объектов является большим преимуществом ООП. Однако проектирование такой программы представляет собой весьма сложную задачу, поскольку в процесс добавляется еще один важный этап - разработка иерархии классов.

Плохо спроектированная иерархия приводит к созданию сложных и запутанных программ. Важно до начала проектирования правильно определить, требуется ли вообще применять объектно-ориентированный подход. Если в иерархии классов нет необходимости, то, как правило, достаточно ограничиться модульной технологией.

* 1. **Описание класса**

***Класс*** - это описание определяемого типа. Любой тип данных представляет собой множество значений и набор действий, которые разрешается выполнять с этими значениями. Например, сами по себе числа не представляют интереса - нужно иметь возможность ими оперировать: складывать, вычитать, вычислять квадратный корень и т. д. В С++ множество значений нового типа определяется задаваемой в классе структурой данных, а действия с объектами нового типа реализуются в виде функций и перегруженных операций С++.

Данные класса называются полями (по аналогии с полями структуры), а функции класса - методами. Поля и методы называются элементами класса. Описание класса в первом приближении выглядит так:

*class <имя>{*

*[ private: ]*

*<описание скрытых элементов>*

*public:*

*<описание доступных элементов>*

*}; // Описание заканчивается точкой с запятой*

Спецификаторы доступа *private* и *public* управляют видимостью элементов класса. Элементы, описанные после служебного слова *private*, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Интерфейс класса описывается после спецификатора *public*. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций *private* и *public*, порядок их следования значения не имеет.

***Поля класса:***

1. могут быть простыми переменными любого типа, указателями, массивами и ссылками (т.е. могут иметь практически любой тип, кроме типа этого же класса, но могут быть указателями или ссылками на этот класс);
2. могут быть константами (описаны с модификатором *const*), при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
3. могут быть описаны с модификатором *static*, но не как *auto*, *extern* и *register*.

Классы могут быть ***глобальными*** (объявленными вне любого блока) и ***локальными*** (объявленными внутри блока, например, внутри функции или внутри другого класса). Обычно классы определяются глобально.

Локальные классы имеют некоторые особенности:

* локальный класс не может иметь статических элементов;
* внутри локального класса можно использовать из охватывающей его области типы, статические (*static*) и внешние (*extern*) переменные, внешние функции и элементы перечислений;
* запрещается использовать автоматические переменные из охватывающей класс области;
* методы локальных классов могут быть только встроенными (*inline*);
* если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

В качестве примера создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество щупалец, силу или наличие гранатомета) и поведение. Естественно, пример будет схематичен, поскольку приводится лишь для демонстрации синтаксиса.

*class monster*

*{*

*int health, ammo;*

*public:*

*monster(int he = 100, int am = 10)*

*{ health = he; ammo = am;}*

*void draw(int x, int y, int scale, int position);*

*int get\_health(){return health;}*

*int get\_ammo(){return ammo;}};*

В этом классе два скрытых поля - *health* и *ammo*, получить значения которых извне можно с помощью методов *get\_health()* и *get\_ammo()*. Доступ к полям с помощью методов в данном случае кажется искусственным усложнением, но надо учитывать, что полями реальных классов могут быть сложные динамические структуры, и получение значений их элементов не так тривиально. Кроме того, очень важной является возможность вносить в эти структуры изменения, не затрагивая интерфейс класса.

Методы класса имеют неограниченный непосредственный доступ к его полям. Внутри метода можно объявлять объекты, указатели и ссылки как своего, так и других классов.

В приведенном классе содержится три определения методов и одно объявление (метод *draw*). Если тело метода определено внутри класса, он является встроенным (*inline*). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости:

*void monster::draw(int x, int y, int scale, int position)*

*{ /\* тело метода \*/}*

Встроенные методы можно определить и вне класса с помощью директивы *inline* (как и для обычных функций, она носит рекомендательный характер):

*inline int monster::get\_ammo()*

*{return ammo;}*

Методы можно перегружать (это одно из проявлений полиморфизма), а также объявлять либо константными, либо статическими (но не одновременно).

В каждом классе есть метод, имя которого совпадает с именем класса. Он называется конструктором и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных. Типы данных *struct* и *union* являются специальными видами класса.

* 1. **Описание объектов**

Конкретные переменные типа данных "класс" называются экземплярами класса, или объектами. Время жизни и видимость объектов зависит от вида и места описания и подчиняется общим правилам С++:

*monster Vasia; // Объект класса monster с параметрами по умолчанию*

*monster Super(200, 300);// Объект с явной инициализацией*

*monster stado[100]; // Массив объектов с параметрами по умолчанию*

*/\* Динамический объект (второй параметр задается по умолчанию) \*/*

*monster \*beavis = new monster (10);*

*monster &butthead = Vasia; // Ссылка на объект*

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор.

Доступ к открытым (*public*) элементам объекта аналогичен доступу к полям структуры. Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель:

*объект.поле*

*указатель -> поле*

*(\*указатель).поле*

*объект.метод( параметры )*

*указатель -> метод( параметры )*

*(\*указатель).метод( параметры )*

Обращение к открытому полю и вызов метода для массива объектов:

*имя\_массива[ индекс ].поле*

*имя\_массива[ индекс ].метод( параметры )*

***Например:***

*int n = Vasia.get\_ammo();*

*stado[5].draw;*

*cout << beavis->get\_health();*

Получить или изменить значения *private* элементов можно только через обращение к соответствующим методам.

Можно создать константный объект, значения полей которого изменять запрещается. К нему должны применяться только константные методы:

*class monster{*

*...*

*int get\_health() const {return health;}*

*};*

*const monster Dead (0,0); // Константный объект*

*cout << Dead.get\_health();*

***Константный метод:***

* объявляется с ключевым словом const после списка параметров;
* не может изменять значения полей класса;
* может вызывать только константные методы;
* может вызываться для любых (не только константных) объектов.

Рекомендуется описывать как константные те методы, которые предназначены для получения значений полей.

* 1. **Указатель this**

Каждый объект содержит свой экземпляр полей класса. Методы места в классе не занимают и не дублируются для каждого объекта. Единственный экземпляр метода используется всеми объектами совместно, поэтому нестатический метод должен "знать", для какого объекта он вызван.

Каждый нестатический метод, помимо явно объявленных параметров, получает еще один скрытый параметр: константный указатель на объект, для которого он вызван. В С++ это указатель обозначается зарезервированным словом *this*. Когда имя параметра метода совпадает с именем поля класса, доступ к полю выполняется через этот указатель (например, *this -> num*).

Выражение *\*this* представляет собой разыменование указателя и имеет тип определяемого класса. Обычно это выражение возвращается в качестве результата, если метод возвращает ссылку на свой класс (*return \*this;*).

Для иллюстрации использования указателя *this* добавим в приведенный выше класс *monster* новый метод, возвращающий ссылку на наиболее здорового (поле *health*) из двух монстров, один из которых вызывает метод, а другой передается ему в качестве параметра (метод нужно поместить в секцию *public* описания класса):

*monster & the\_best(monster &M)*

*{*

*if( health > M.get\_health())*

*return \*this;*

*return M;*

*}*

*...*

*monster Vasia(50), Super(200);*

*// Новый объект Best инициализируется значениями полей Super*

*monster Best = Vasia.the\_best(Super);*

* 1. **Конструкторы**

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов.

* Конструктор не возвращает значения, даже типа *void*. Нельзя получить указатель на конструктор.
* Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
* Конструктор, который можно вызвать без параметров, называется конструктором по умолчанию.
* Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.
* Если программист не указал ни одного конструктора, компилятор создает его автоматически (кроме случая, когда класс содержит константы и ссылки, поскольку их необходимо инициализировать). Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы базовых классов.
* Конструкторы не наследуются.
* Конструктор не может быть константным, статическим и виртуальным (нельзя использовать модификаторы *const*, *virtual* и *static*).
* Конструкторы глобальных объектов вызываются до вызова функции *main*. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

При объявлении объектов вызывается один из конструкторов. При отсутствии инициализирующего выражения в объявлении объекта вызывается конструктор по умолчанию, при инициализации другим объектом того же типа - конструктор копирования, при инициализации полей - один из явно определенных конструкторов инициализации (т.е. конструкторов, которым передаются параметры для инициализации полей объекта).

Конструкторы часто вызываются неявно для создания временных объектов. Обычно это происходит в следующих случаях:

* при инициализации;
* при выполнении операции присваивания;
* для задания значений параметров по умолчанию;
* при создании и инициализации массива;
* при создании динамических объектов;
* при передаче параметров в функцию и возврате результатов по значению.

***Примеры:***

*monster Super(200, 300), Vasia(50);*

*monster X = monster(1000);*

В последнем операторе создается объект Х, которому присваивается безымянный объект со значением параметра *health = 1000* (значения остальных параметров устанавливаются по умолчанию).

При создании динамического массива вызывается конструктор без аргументов.

В качестве примера класса с несколькими конструкторами усовершенствуем описанный ранее класс *monster*, добавив в него поля, задающие цвет (*skin*) и имя (*name*):

*enum color {red, green, blue}; // Возможные значения цвета*

*class monster*

*{*

*int health, ammo;*

*color skin;*

*char \*name;*

*public:*

*monster(int he = 100, int am = 10);*

*monster(color sk);*

*monster(char \* nam);*

*...*

*};*

*//--------------------------------*

*monster::monster(int he, int am)*

*{ health = he; ammo = am; skin = red; name = 0;}*

*//--------------------------------*

*monster::monster(color sk)*

*{*

*switch (sk)*

*{*

*case red: health = 100; ammo = 10; skin = red; name = 0; break;*

*case green: health = 100;ammo = 20;skin = green;name = 0;break;*

*case blue: health = 100; ammo = 40; skin = blue;name = 0;break;*

*}*

*}*

*//--------------------------------*

*monster::monster(char \* nam)*

*{*

*/\* К длине строки добавляется 1 для хранения нуль-символа \*/*

*name = new char [strlen(nam) + 1];*

*strcpy(name, nam);*

*health = 100; ammo = 10; skin = red;*

*}*

*//--------------------------------*

*monster \* m = new monster ("Ork");*

*monster Green (green);*

Первый из приведенных выше конструкторов является конструктором по умолчанию, поскольку его можно вызвать без параметров. Объекты класса *monster* теперь можно инициализировать различными способами, требуемый конструктор будет вызван в соответствии со списком инициализации. При задании нескольких конструкторов следует соблюдать те же правила, что и при написании перегруженных функций - у компилятора должна быть возможность распознать нужный вариант.

Существует еще один способ инициализации полей в конструкторе - с помощью списка инициализаторов, расположенным после двоеточия между заголовком и телом конструктора:

*monster::monster(int he, int am):*

*health (he), ammo (am), skin (red), name (0){}*

Поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением. Без этого способа не обойтись при инициализации полей-констант, полей-ссылок и полей-объектов. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

***Конструктор копирования***

***Конструктор копирования*** - это специальный вид конструктора, получающий в качестве единственного параметра указатель на объект этого же класса:

*T::T(const T&) { ... /\* Тело конструктора \*/ }*

где *T* - имя класса. Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:

* при описании нового объекта с инициализацией другим объектом;
* при передаче объекта в функцию по значению;
* при возврате объекта из функции.

Если программист не указал ни одного конструктора копирования, компилятор создает его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

Запишем конструктор копирования для класса *monster*. Поскольку в нем есть поле *name*, содержащее указатель на строку символов, конструктор копирования должен выделять память под новую строку и копировать в нее исходную:

*monster::monster(const monster &M)*

*{*

*if (M.name)*

*{*

*name = new char [strlen(M.name) + 1];*

*strcpy(name, M.name);*

*}*

*else name = 0;*

*health = M.health; ammo = M.ammo; skin = M.skin;*

*}*

*...*

*monster Vasia (blue);*

*monster Super = Vasia; // Работает конструктор копирования*

*monster \*m = new monster ("Ork");*

*monster Green = \*m; // Работает конструктор копирования*

* 1. **Статические элементы класса**

С помощью модификатора *static* можно описать статические поля и методы класса.

***Статические поля***

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора:

*class A*

*{*

*public:*

*static int count;*

*}*

*...*

*A::count = 0;*

Статические поля доступны как через имя класса, так и через имя объекта:

*/\* будет выведено одно и то же \*/*

*A \*a, b; \* cout << A::count << a->count << b.count;*

На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как *private,* нельзя инициализировать с помощью операции доступа к области действия, как описано выше. Им можно присвоить значения только с помощью статических методов.

Память, занимаемая статическим полем, не учитывается при определении размера объекта операцией *sizeof*. Статические поля нельзя инициализировать в конструкторе, так как они создаются до создания любого объекта.

Классическое применение статических полей - подсчет объектов. Для этого в классе объявляется целочисленное поле, которое увеличивается в конструкторе и уменьшается в деструкторе.

***Статические методы***

Статические методы могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, поскольку им не передается скрытый указатель *this*. Обращение к статическим методам производится так же, как к статическим полям - либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

Статические методы не могут быть константными (*const*) и виртуальными (*virtual*).

* 1. **Дружественные функции и классы**

Иногда желательно иметь непосредственный доступ извне к скрытым полям класса, то есть расширить интерфейс класса. Для этого служат дружественные функции и дружественные классы.

***Дружественная функция***

Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом *friend*. В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель *this* ей не передается. Одна функция может "дружить" сразу с несколькими классами.

Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие спецификаторов доступа, место размещения ее объявления в классе безразлично.

В качестве примера ниже приведено описание двух функций, дружественных классу *monster*. Функция *kill* является методом класса *hero*, а функция *steal\_ammo* не принадлежит ни одному классу. Обеим функциям в качестве параметра передается ссылка на объект класса *monster*.

*class monster; // Предварительное объявление класса*

*class hero*

*{*

*...*

*void kill(monster &);*

*};*

*class monster*

*{*

*...*

*friend int steal\_ammo(monster &);*

*/\* Класс hero должен быть определен ранее \*/*

*friend void hero::kill(monster &);*

*};*

*int steal\_ammo(monster &M){return --M.ammo;}*

*void hero::kill(monster &M){M.health = 0; M.ammo = 0;}*

***Дружественный класс***

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого, весь класс объявляется дружественным с помощью ключевого слова *friend.* В приведенном ниже примере класс *mistress* объявляется дружественным классу *hero*:

*class hero{*

*... friend class mistress;}*

*class mistress{*

*... void f1();*

*void f1();}*

Функции *f1* и *f2* являются дружественными по отношению к классу *hero* (хотя и описаны без ключевого слова *friend*) и имеют доступ ко всем его полям.

Объявление *friend* не является спецификатором доступа и не наследуется. Обратите внимание на то, что класс сам определяет, какие функции и классы являются дружественными, а какие нет.

* 1. **Деструкторы**

***Деструктор*** - это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. Деструктор вызывается автоматически, когда объект выходит из области видимости:

* для локальных переменных - при выходе из блока, в котором они объявлены;
* для глобальных - как часть процедуры выхода из *main*;
* для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции *delete* (автоматический вызов деструктора при выходе указателя из области действия не производится).

При уничтожении массива деструктор вызывается для каждого элемента удаляемого массива. Для динамических объектов деструктор вызывается при уничтожении объекта операцией *delete*. При выполнении операции *delete[]* деструктор вызывается для каждого элемента удаляемого массива.

Имя деструктора начинается с тильды (*~*), непосредственно за которой следует имя класса. Деструктор:

* не имеет аргументов и возвращаемого значения;
* не может быть объявлен как *const* или *static*;
* не наследуется;
* может быть виртуальным;
* может вызываться явным образом путем указания полностью уточненного имени; это необходимо для объектов, которым с помощью *new* выделялся конкретный адрес.

Если деструктор явным образом не определен, компилятор создает его автоматически.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически - иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор для рассматриваемого примера должен выглядеть так :

*monster::~monster() {delete [] name;}*

* 1. **Перегрузка операций**

С++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Эта дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в С++, за исключением: . .\* ?: :: # ## *sizeof* .

Перегрузка операций осуществляется с помощью функций специального вида (функций-операций) и подчиняется следующим правилам:

* сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо) по сравнению с использованием в стандартных типах данных;
* нельзя переопределить операцию по отношению к стандартным типам данных;
* функция-операция не может иметь аргументов по умолчанию;
* функции-операции наследуются (за исключением =).

Функцию-операцию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс (особый случай: функция-операция, первый параметр которой - стандартного типа, не может определяться как метод класса).

Функция-операция содержит ключевое слово *operator*, за которым следует знак переопределяемой операции:

*тип operator операция ( список параметров) { тело функции }*

***Перегрузка унарных операций***

Унарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода без параметров, при этом операндом является вызвавший ее объект, например:

*class monster*

*{... monster & operator ++() {++health; return \*this;}}*

*monster Vasia;*

*cout << (++Vasia).get\_health();*

Если функция определяется вне класса, она должна иметь один параметр типа класса:

*class monster*

*{... friend monster & operator ++( monster &M);};*

*monster& operator ++(monster &M) {++M.health; return M;}*

Если не описывать функцию внутри класса как дружественную, нужно учитывать доступность изменяемых полей (в данном случае поле *health* недоступно извне, так как описано со спецификатором *private*, поэтому для его изменения требуется использование соответствующего метода, не описанного в приведенном примере).

Операции постфиксного инкремента и декремента должны иметь первый параметр типа *int*. Он используется только для того, чтобы отличить их от префиксной формы:

*class monster*

*{... monster operator ++(int){monster M(\*this); health++; return M;}};*

*monster Vasia;*

*cout << (Vasia++).get\_health();*

***Перегрузка бинарных операций***

Бинарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

*class monster*

*{ ...*

*bool operator >( const monster &M)*

*{*

*if( health > M.get\_health()) return true;*

*return false;*

*}*

*};*

Если функция определяется вне класса, она должна иметь два параметра типа класса:

*bool operator >(const monster &M1, const monster &M2)*

*{ if( M1.get\_health() > M2.get\_health()) return true;*

*return false;*

*}*

***Перегрузка операции присваивания***

Операция присваивания определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля ссылок на динамически выделяемую память, необходимо определить собственную операцию присваивания. Чтобы сохранить семантику операции, операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент - ссылку на присваиваемый объект:

*monster& operator = (const monster &M)*

*{*

*if (&M == this) return \*this; // Проверка на самоприсваивание*

*if (name) delete [] name;*

*if (M.name)*

*{*

*name = new char [strlen(M.name) + 1];*

*strcpy(name, M.name);*

*}*

*else name = 0;*

*health = M.health; ammo = M.ammo; skin = M.skin;*

*return \*this;*

*}*

Операцию присваивания можно определять только в теле класса. Она не наследуется.

***Перегрузка операции приведения типа***

Можно определить функции-операции, которые будут осуществлять преобразование класса к другому типу. Формат:

*operator имя\_нового\_типа ();*

Тип возвращаемого значения и параметры указывать не требуется. Можно определять виртуальные функции преобразования типа.

***Пример:***

*monster::operator int(){return health;}*

*...*

*monster Vasia; cout << int(Vasia);*

* 1. **Указатели на элементы классов**

К элементам классов можно обращаться с помощью указателей. Для этого определены операции .\* и ->\*. Указатели на поля и методы класса определяются по-разному.

Формат указателя на метод класса:

*возвр\_тип (имя\_класса::\*имя\_указателя)(параметры);*

Например, описание указателя на метод класса *monster*

*int get\_health() const {return health;}*

будет иметь вид:

*int (monster:: \*pget)();*

Такой указатель можно задавать в качестве параметра функции. Это дает возможность передавать в функцию имя метода:

*void fun(int (monster:: \*pget)())*

*{*

*(\*this.\*pget)(); // Вызов функции через операцию .\**

*(this->\*pget)(); // Вызов функции через операцию ->\**

*}*

Можно настроить указатель на конкретный метод с помощью операции взятия адреса:

*pget = & monster::get\_health;// Присваивание значения указателю*

*monster Vasia, \*p;*

*p = new monster;*

*int Vasin\_health = (Vasia.\*pget)();//Вызов функции через операцию .\**

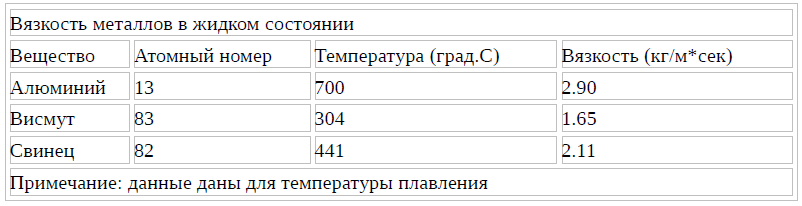
*int p\_health = (p->\*pget)(); // Вызов функции через операцию ->\**

* Указателю на метод можно присваивать только адреса методов, имеющих соответствующий заголовок.
* Нельзя определять указатель на статический метод класса.
* Нельзя преобразовывать указатель на метод в указатель на обычную функцию, не являющуюся элементом класса.

Как и указатели на обычные функции, указатели на методы используются в том случае, когда возникает необходимость вызвать метод, имя которого неизвестно. Однако в отличие указателя на переменную или обычную функцию, указатель на метод не ссылается на определенный адрес памяти. Он больше похож на индекс в массиве, поскольку задает смещение. Конкретный адрес в памяти получается путем сочетания указателя на метод с указателем на определенный объект.

1. **Практическая часть**

**Задание. Вариант 18.**



Составить программу, в которой будут вводиться 7 - 10 строк таблицы и выводится на экран таблица - сразу же после ввода и после сортировки ее по значениям в первом столбце. Программа должна быть составлена с использованием класса, необходимо предусмотреть функции доступа к полям класса (функции ***get()*** и ***set()***) и необходимо перегрузить следующие операторы:

* Присваивание;
* Сложение (в случае символьных полей сложение понимается как конкатенация);
* Соответствие;
* Потоковый ввод;
* Потоковый вывод.
  1. **Описание полей класса.**

Класс, описывающий *Вязкость металлов в жидком состоянии* имеет следующие поля:

* Вещество. – **char \***.
* Атомный номер. – **unsigned int**.
* Температура. – **unsigned int**.
* Вязкость. – **float**.
  1. **Функции – члены класса и дружественные функции**

Класс ***moy*** имеет следующие функции-члены:

*void setall(char \*a, unsigned int ,unsigned int , float );*

*void getall(char \*a, unsigned int b, unsigned int c, float d);*

*void showall(void);*

*moy operator = (moy o1);*

*int operator == (moy o1);*

*moy operator + (moy o1);*

а также два конструктора – по умолчанию и инициализирующий:

*moy () { vyazkost =0; temperatura =0; nomer =0; \*(veshestvo=newchar[1])='\0';}*

*moy (char \*a, unsigned int b, unsigned int c, float d);*

Класс ***moy*** имеет следующие дружественные функции:

*friend ostream & operator<<(ostream &stream, moy &o1);*

*friend istream & operator>>(istream &stream, moy &o1);*

*friend void shapka(void);*

*friend void linebuild(void);*

*friend int isvalid(int a, int b);*

* 1. **Разработка текста программы**

Для удобства работы с программой введём переменную-константу, которая будет отвечать за количество экземпляров класса ***moy***. То есть будем работать с массивом объектов класса ***moy*** размерностью ***N***.

***const*** *N=3****;***

Перейдём к описанию объявленных выше функций:

*moy* ***operator =****(moy o1);*

Данная функция производит перегрузку оператора присваивания, то есть выполняет присваи-вание полей одного объекта класса полям другого объекта этого же класса. При перегрузке бинарного оператора левый операнд передаётся функции неявно, а правый передаётся в каче-стве аргумента. Возвращаемый тип - сам класс, т.е ***moy***. В целом эта функция - аналог функции ***set***.

***int operator ==****(moy o1);*

Функция позволяет установить соответствие между экземплярами одного класса. Т.е. её тело включает проверку на равенство значений числовых полей и сравнение содержимого строк, которое выполняется с помощью функции ***strcmp***. В зависимости от того, тождественны ли экземпляры функцией возвращается 0 или 1.

*moy* ***operator +****(moy o1);*

Функция ***operator+()*** возвращает объект типа ***moy*** Отметим тот факт, что временный объект ***tr*** используется внутри функции ***operator+()*** для хранения результата и является возвращаемым объектом. Отметим также, что ни один из операндов не меняется. Назначение переменной ***tr*** легко понять.

В данной ситуации оператор + был перегружен способом, аналогичным своему традиционному арифметическому использованию. Поэтому и было важно, чтобы ни один из операндов не менялся. Таким образом, временный объект необходим для хранения результата.

***friend*** *ostream &* ***operator<<****(ostream &stream, moy &o1);*

Первый параметр является ссылкой на объект типа **ostream**. Это означает, что поток **stream** должен быть потоком вывода. Второй параметр получает выводимый объект. Обратите внимание, что пользовательская функция вывода возвращает ссылку на поток **stream**, который имеет тип **ostream**. Это необходимо, если перегруженный оператор << должен использоваться в ряде последовательных выражений ввода/вывода.

***friend*** *istream &* ***operator>>****(istream &stream, moy &o1);*

Пользовательские функции ввода возвращают ссылку, на поток **istream**, который является по-током ввода. Первый параметр тоже является ссылкой на поток ввода. Второй параметр - это ссылка на объект, получающий вводимую информацию.

Так же, как и пользовательская функция вывода, пользовательская функция ввода не может быть функцией-членом. Хотя внутри такой функции может быть выполнена любая операция, лучше ограничить ее работу вводом информации.

*friend void shapka(void);*

Данная функция является вспомогательной. Она формирует шапку таблицы при вызове перегруженной формы оператора **<<**

***friend int isvalid****(****int*** *a,* ***int*** *b);*

Функция, описанная выше выполняет проверку данных, которые вводит пользователь. Т.е проверяет соответствие номеров класса на существование.

* 1. **Листинг программы**

#include <iostream.h>

#include <stdlib.h>

#include <conio.h>

#include <string.h>

#include <math.h>

#include <stdio.h>

#include <iomanip.h>

const N=3;

class moy {

char \*veshestvo;

unsigned int nomer;

unsigned int temperatura;

float vyazkost;

friend ostream & operator<<(ostream &stream, moy &o1);

friend istream & operator>>(istream &stream, moy &o1);

friend void shapka(void);

friend void linebuild(void);

friend int isvalid(int a, int b);

public:

moy () { vyazkost =0; temperatura =0; nomer =0; \*(veshestvo = new char[1])='\0';}

moy (char \*a, unsigned int b, unsigned int c, float d);

void setall(char \*a, unsigned int ,unsigned int , float );

void getall(char \*a, unsigned int b, unsigned int c, float d);

void showall(void);

moy operator = (moy o1);

int operator == (moy o1);

moy operator + (moy o1);

};

ostream & operator<<(ostream &stream, moy &o1) {

shapka();

stream<<"|"<<setw(15)<<o1.veshestvo<<" | ";

stream<<setw(9)<<o1. nomer <<"|";

stream<<setw(18)<<o1. temperatura <<" |";

stream<<setw(17)<<o1. vyazkost <<" |";

linebuild();

return stream;

}

istream & operator>>(istream &stream, moy &o1) {

cout<<"Veshestvo, Atomniyi nomer, Temperatura, Vyazkost: \n";

stream>>o1.veshestvo;

stream>>o1. nomer;

stream>>o1. temperatura;

stream>>o1. vyazkost;

return stream;

}

int isvalid(int a, int b){

if (((a>N-1) || (a<0)) || ((b>N-1) ||(b<0)))

{

cout<<" Oshibka! Ekzempliara s takim imenem ne syshestvyet.\n";

getch();

return -1;

}

else if (a==b)

{

cout<<" Oshibka! Ekzempliar ne mozet bit zapisan sam v sebya.\n";

getch();

return -2;

}

return 0;

}

moy moy::operator + (moy o1) {

moy tr;

int i,j;

delete[] tr.veshestvo;

tr.veshestvo =new char [strlen(veshestvo)+strlen(o1.veshestvo)+2];

strcpy(tr.veshestvo,veshestvo);

strcat(tr.veshestvo, o1.veshestvo);

tr. nomer = nomer +o1. nomer;

tr. temperatura = temperatura +o1. temperatura;

tr. vyazkost = vyazkost +o1. vyazkost;

return tr;

}

int moy::operator == (moy o1) {

if (temperatura!= o1. temperatura) {cout << " Dannie ekzempliari klassa ne ravni."; getch();}

else if (ceil(vyazkost)!=ceil(o1. vyazkost)) {cout << " Dannie ekzempliari klassa ne ravni."; getch();}

else if (strcmp(veshestvo,o1.veshestvo) != 0) {cout << " Dannie ekzempliari klassa ne ravni."; getch();}

else if (nomer!= o1. nomer) {cout << " Dannie ekzempliari klassa ne ravni."; getch();}

else cout<<" Ekzemplyari klassa ravni."; getch();

return 0;

}

moy moy::operator = (moy o1) {

delete [] veshestvo;

veshestvo = new char [strlen(o1.veshestvo)+2];

if (!veshestvo) {

cout<<" Oshibka! Pamiat ne videlena.";

exit(1);

}

strcpy(veshestvo,o1.veshestvo);

nomer = o1. nomer;

temperatura = o1. temperatura;

vyazkost = o1. vyazkost;

return o1;

}

moy:: moy (char \*a, unsigned int b, unsigned int c, float d) {

veshestvo=new char [strlen(a)+1];

strcpy(veshestvo,a);

nomer =b;

temperatura =c;

vyazkost =d;

}

void moy::setall(char \*a, unsigned int b, unsigned int c, float d) {

strcpy(veshestvo,a);

nomer =b;

temperatura =c;

vyazkost =d;

}

void moy::showall(void) {

cout<< veshestvo <<" ";

cout<< nomer <<" ";

cout<< temperatura <<" ";

cout<< vyazkost <<" ";

}

void moy::getall( char \* a, unsigned int b, unsigned int c, float d)

{

delete[] a;

a=new char [strlen(veshestvo)+1];

strcpy(a, veshestvo);

b= nomer;

c= temperatura;

d= vyazkost;

cout<<a<<" ";

cout<<b<<" ";

cout<<c<<" ";

cout<<d<<" ";

cout<<"\n";

}

void shapka(void)

{

cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

cout<<"| Vyazkost metallov v ghidkom sostoyanii |\n";

cout<<"|---------------------------------------------------------------------------|\n";

cout<<"| Veshestvo | Atomniyi nomer | Temperatura | Vyazkost |\n";

cout<<"|---------------------------------------------------------------------------|\n";

}

void linebuild(void) {

cout<<"\n|--------------------------------------------------------------------------|\n";

}

int main(void) {

char \*n;

unsigned int t;

unsigned int s;

float h;

short i,k;

unsigned int s1;

short q,q1;

moy obj[N]={ moy ("Alyuminiyi", 13,700,2.90),

moy ("Vismut", 83,304,1.65),

moy ("Svinec", 82,441,2.11)};

clrscr();

cout.precision(2);

cout<<" Veshestvo, Atomniyi nomer, Temperatura, Vyazkost: \n";

for (i=0; i<N; i++) {

obj[i].showall();

cout<<"\n";

}

cout<<"\nPeregryzka operatora prisvaivania '='.\n";

cout<<" Vvedite nomera ekzemplyarov klassa: zameniaimi b zameniashi >\n";

cin>>q;

cin>>q1;

if (isvalid(q, q1)!=0) exit(-1);

obj[q] = obj[q1];

cout<<" Veshestvo, Atomniyi nomer, Temperatura, Vyazkost: \n";

for(i=0; i<N; i++) {

obj[i].showall();

cout<<"\n";

}

cout<<"\nPeregryzka operatorov sootvetstviya '=='.\n";

cout<<" Vvedite nomera ekzemplyarov klassa, kotorie nyzhno sravnit >\n";

cin>>q;

cin>>q1;

if (isvalid(q, q1)!=0) exit(-1);

obj[q] == obj[q1];

cout<<"\nPeregryzka operatorov symmi '+'.\n";

cout<<" Vvedite nomera ekzemplyarov klassa, kotorie nyzhno slozhit >\n";

cin>>q;

cin>>q1;

moy temp;

if (isvalid(q, q1)!=0) exit(-1);

temp=obj[q]+obj[q1];

cout<<" Veshestvo, Atomniyi nomer, Temperatura, Vyazkost: ";

temp.getall(n,t,s,h);

cout<<"\nPeregryzka operatorov potovogo vvoda '>>'.";

cout<<"\nVvedite nomer ekzemplyara klassa: ";

cin>>q;

if ((q>N-1)||(q<0)) {cout<<" Oshibka! Ekzempliara s takim indeksom ne syshestvyet."; getch(); exit(-1);}

cin>>obj[q];

cout<<"\n Peregryzka operatorov potovogo vivoda '<<'.";

cout<<"\n Vvedite nomer ekzemplyara klassa: ";

cin>>q;

if ((q>N-1)||(q<0)) {cout<<" Oshibka! Ekzempliara s takim indeksom ne syshestvyet."; getch(); exit(-1);}

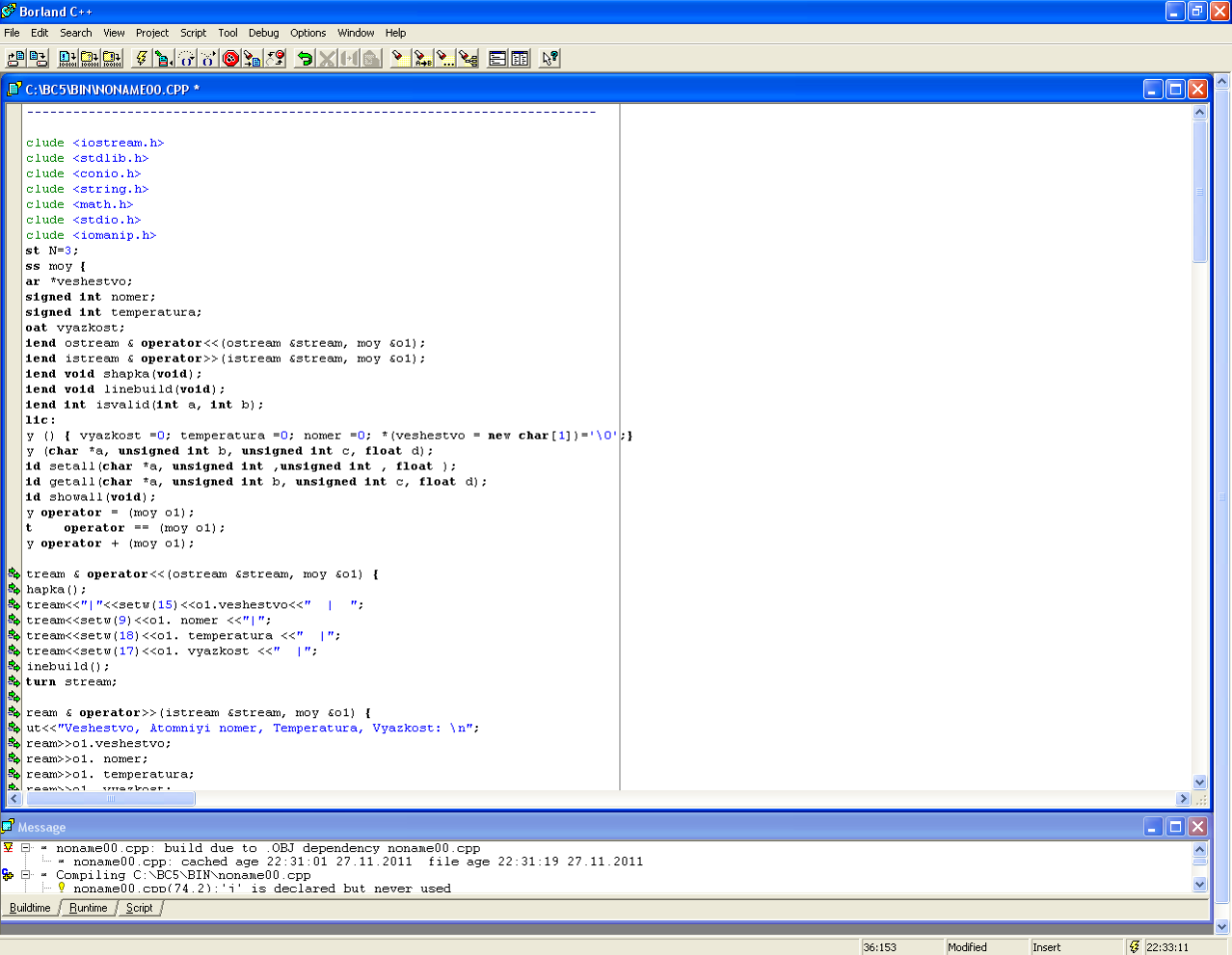
cout<<obj[q];

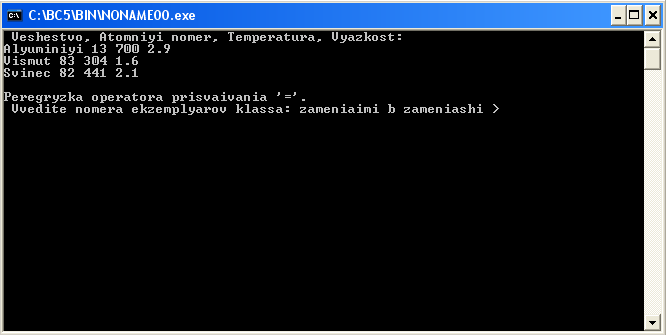
getch();

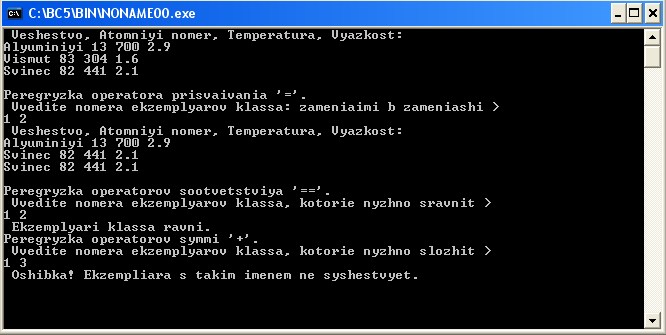
return 0;

}

* 1. **Результат выполнения программы**







**Заключение**

Объектно-ориентированное программирование является в настоящее время основой всей индустрии прикладного программирования благодаря выигрышу в конкурентной борьбе с альтернативными технологиями программирования. В промышленном программировании только в системном программировании позиции объектно-ориентированного программирования еще не очень сильны. Поэтому, с одной стороны, теоретические рассуждения о непригодности объектно-ориентированного программирования не соответствуют наблюдаемой на практике ситуации. С другой стороны, нельзя считать, что объектно-ориентированное программирование во всех случаях является наилучшей из методик программирования.

Процедурное программирование лучше подходит для случаев, когда важны быстродействие и потребляемые ресурсы, объектное - когда важна управляемость проекта и его модифицируемость, а также безопасность программ. Процедурное программирование обычно лучше подходит для небольших проектов, объектное - для больших.

**Список используемой литературы**

1. Иан Грэхем Объектно-ориентированные методы. Принципы и практика = Object-Oriented Methods: Principles & Practice. — 3-е изд. — М.: «Вильямс», 2004. — С. 880.

2. Антони Синтес Освой самостоятельно объектно-ориентированное программирование за 21 день = Sams Teach Yourself Object-Oriented Programming in 21 Days. — М.: «Вильямс», 2002. — С. 672.

3. Бертран Мейер Объектно-ориентированное конструирование программных систем + CD . Интернет-университет информационных технологий - ИНТУИТ.ру, Русская Редакция, 2005

4. Биллиг В.А. Основы программирования на C# . Интернет-университет информационных технологий - ИНТУИТ.ру, 2006

5. “Новые языки программирования и тенденции их развития”, Ушкова В., 2005 г.

6. Объектно-Орентированное программирование в C++, Р. Лафоре, 4-е 2004 г.