

Основы программирования на языке C++

Преподаватель:
Маслов Алексей Владимирович,
доцент кафедры общей физики

Содержание

1. Типы данных, операторы, выражения [лек. 1]
2. Управление (инструкции, блоки, циклы, переключатели) [лек. 2]
3. Массивы, структуры, указатели [лек. 3, 4]
4. Функции и структура программы, шаблоны функций [лек. 5,6]
5. Классы и объекты [лек. 7]
6. Работа с памятью, пространства имен [лек. 8]
7. Интерфейс (ввод-вывод)
8.

Объекты и классы

Объекты и классы

Процедурное программирование (в С) сводится к разбиению задачи на набор переменных, структур данных, разработке алгоритмов и выделению подпрограмм (функций).

Объектно-ориентированное программирование (ООП) – это особый концептуальный подход к проектированию программ. Заключается в создании новых типов данных, которые описывают решаемую задачу.

С++ расширяет язык С средствами, облегчающими применение ООП.

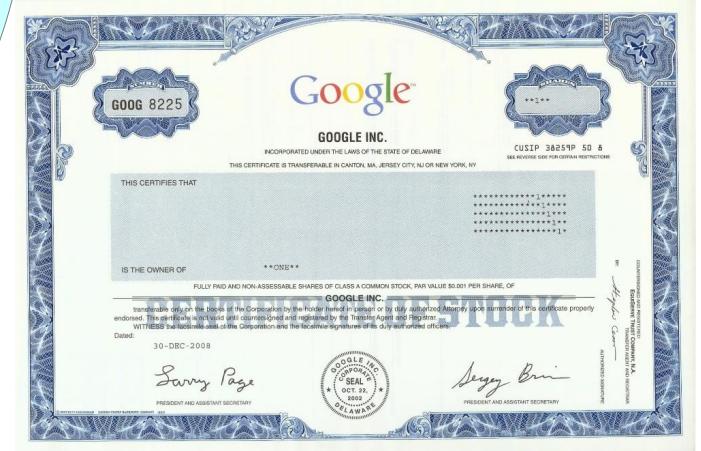
Классы – это наиболее важное расширение С++, предназначенное для реализации ООП.

Пример: Работа с акциями компаний

Акции – ценные бумаги, которые обеспечивают её владельцу долю в капитале компании, дают право на получение дивидентов и на управление компанией.



Так как мы оперируем с акциями, то давайте введём новый тип данных — акции и определим операции с ними.



Классы в C++

Класс – тип данных, который воплощает некоторую абстракцию (вещь, объект) . Он комбинирует представление данных и методов для манипулирования этими данными.

Определение класса напоминает определение структур. В отличие от стандартных структур языка С классы в C++ включают в качестве элементов не только данные, но и функции (методы).

Обычно спецификация класса состоит из двух частей.

- Объявление класса описывает его элементы (компоненты, члены): переменные (данные) и функции (методы). Методы играют роль интерфейса данного класса.
- Определения методов класса описывают реализацию функций, то есть что конкретно делают функции.

Грубо говоря, объявление класса предоставляет общий обзор класса, в то время как определения методов снабжают необходимыми деталями.

Пример: определение класса Stock для работы с акциями

```
class Stock { // объявление класса, его имя – новый тип данных
private: // идентифицирует элементы класса с ограниченным доступом
    std::string company; // компания (элемент класса – данные)
    long shares;          // количество акций
    double share_val;    // цена одной акции
    double total_val;    // полная стоимость
    void set_tot() { total_val = shares * share_val; } // элемент-функция
public: // элементы с открытым доступом (интерфейс)
.....
void acquire(string str, long num, double price); // элемент-функция
void buy(long num, double price); // элемент-функция
void sell(long num, double price);
void update(double price);
void show() const; // не меняет элементы объекта
.....
}
```

Ключевое слово `class` в C++ служит для определение класса.

Синтаксис `class Stock` идентифицирует `Stock` в качестве имени типа для нового класса. Это позволяет объявлять переменные, которые называются объектами (или экземплярами) типа `Stock`.

```
Stock Apple; // создание объекта с именем Apple типа Stock  
int ivar; // создание (декларирование) переменной ivar типа int  
Stock some_name2;
```

Необходимые операции представлены в виде элементов-функций (функций-членов или методов), таких как `sell()` и `update()`. Элемент-функция может быть определена на месте, как, например, `set_tot()`, либо представлена с помощью прототипа.

Связывание данных и методов в единое целое – отличительное свойство класса. В структуры, как правило, объединяют данные.

Управление доступом к элементам класса

Метки `private` и `public` определяют доступ к элементам класса.

`public`: Любая функция, которая использует объект класса, имеет прямой доступ к элементам из раздела `public`. Доступ осуществляется, используя точку (.), как и в случае структур.

`Stock Apple;`

```
Apple.buy(10, 34.56); // вызов метода buy
```

`private`: Прямого доступа к элементам объекта из раздела `private` нет. Доступ имеют только открытые элементы-функции из раздела `public` (или же через дружественные функции, см. дальше).

```
cout << Apple.shares; // неверно, нет открытого доступа к shares
```

Эта изоляция данных от прямого доступа со стороны программы называется сокрытием данных.

При проектировании класса разумно отделить открытый интерфейс от специфики реализации.

Собрание деталей реализации в одном месте и отделение их от интерфейса называется инкапсуляцией (encapsulation).

Примеры инкапсуляции:

- 1) Скрытие данных (помещение данных в раздел `private` класса)
- 2) Скрытие функциональных деталей реализации, как это сделано в классе `Stock` с функцией `set_tot()`.
- 3) Помещение определений функций класса в файл (*.cpp), отдельный от объявления класса (*.h).

Отчасти это напоминает использование обычных функций, когда пользователю надо только уметь вызывать функцию и совершенно не надо знать детали её реализации.

public или private?

Одним из главных принципов ООП (объектно-ориентированного программирования) является сокрытие данных, т.е. единицы данных обычно размещаются в разделе **private**.

Элементы-функции, которые образуют интерфейс класса, размещаются в разделе **public**; в противном случае вызвать эти функции из программы не удастся.

Как правило, закрытые элементы-функции применяются для управления деталями реализации, которые не формируют часть открытого интерфейса.

Использовать ключевое слово **private** в объявлении класса не обязательно, поскольку это спецификатор доступа к объектам класса по умолчанию.

```
class World {  
    float mass;      // по умолчанию private  
    char name [20]; // по умолчанию private  
    public:  
        void tellall(void);  
    ...  
}
```

Реализация элементов-функций класса

Определения элементов-функций очень похожи на определения обычных функций. Каждое из них имеет заголовок, тело, тип возврата и аргументы.

Но, кроме того, с ними связаны две специфических характеристики.

- 1) Для идентификации класса, которому принадлежит функция, используется операция разрешения контекста (::)
- 2) Методы класса имеют доступ к private-компонентам класса.

```
void Stock::buy(long num, double price) { // функция-элемент класса Stock
    shares += num; // доступ к private long shares
    share_val = price; // доступ к private double share_val, price
    set_tot(); // доступ к private set_tot(), элемент этого класса
}
```

```
void Stock::show(){ // функция-элемент класса Stock
    cout << "Company: " << company << ", " << "Shares: " << shares << ", ";
    cout << "Share Price: $" << share_val << ", ";
    cout << "Total value: $" << total_val << '\n' ;
}
```

Какой объект используется при вызове метода?

Stock st_1, st_2; // создание двух объектов

st_1.show(); // объект st_1 вызывает элемент-функцию (member function)

st_2.show(); // объект st_2 вызывает элемент-функцию

Каждый вновь созданный объект содержит хранилище для собственных внутренних элементов-данных класса.

Все объекты одного класса разделяют общий набор методов.

Вызов элемента-функции часто называется отправкой сообщения.
Отправка сообщения двум разным объектам вызывает один и тот же метод, который применяется к двум разным объектам

Использование классов

```
#include <iostream>
#include "stock.h" // заголовочный файл

int main(){
    Stock s1; // создание объекта s1 класса Stock
    s1.acquire("NanoSmart", 20, 12.50); //вызов метода
    s1.show(); // вызов метода
    s1.buy(15, 18.125);
    s1.show();
    return 0;
}
```

Встроенные методы

Любая функция с определением внутри объявления класса автоматически становится встроенной (*inline*). Это значит, что Stock::set_tot() является встроенной функцией.

Альтернатива:

```
class Stock{  
    private:  
        void set_tot(); // определение оставлено отдельным  
    public:  
};
```

```
inline void Stock::set_tot(){ // использование inline в определении  
    total_val = shares * share_val;  
}
```

Клиент-серверная модель разработки программ

В этой концепции, клиентом является программа, которая использует класс.

Объявление класса, включая его методы, образует сервер, который является ресурсом, доступным нуждающейся в нем программе.

Клиент взаимодействует с сервером только через открытый (public) интерфейс. Единственной ответственностью клиента и, как следствие – программиста, является знание интерфейса.

Ответственностью сервера и, как следствие – его разработчика, является обеспечение того, чтобы его реализация надежно и точно соответствовала интерфейсу.

Любые изменения, вносимые разработчиком сервера в класс, должны касаться деталей реализации, но не интерфейса.

Это аналогично использованию обычных функций в С:

`double y=sin(x); // мы знаем, как вызывать функции, но не их реализацию`

Конструкторы и деструкторы классов

```
int a = 3; // создание и инициализация переменной a  
struct movie s1 = {"Intersteller", 2014, 169}; // тоже для структуры  
Stock s1; // создание объекта s1 класса Stock
```

Как сделать использование объектов классов подобным применению стандартных типов данных?

К данным класса разрешён только закрытый доступ, а это означает, что единственный способ, с помощью которого программа может получить доступ к ним – через методы класса.

Следовательно, для успешной инициализации объекта понадобится придумать соответствующие методы.

Для автоматической инициализации используются специальные методы (функции), называемые *конструкторами класса*, которые предназначены для создания новых объектов и присваивания значений их элементам.

Класс Stock с конструкторами и деструкторами

```
class Stock { // объявление класса, его имя – новый тип данных
private:
    ....
public: // элементы с открытым доступом (интерфейс)
    Stock(); // конструктор без аргументов
    Stock(const std::string &co, long n=0, double pr=0.0); // конструктор
    ~Stock(); // деструктор
    ....
}
```

Объявление и определение конструкторов

```
// Конструктор без аргументов  
// Stock(); прототип
```

```
Stock::Stock(){ // конструктор без аргументов  
company = "no name";  
shares = 0;  
share_val = 0.0;  
total_val = 0.0;  
}
```

```
// Конструктор с несколькими аргументами по умолчанию  
// Stock (const std::string & co, long n = 0, double pr = 0.0); прототип
```

```
Stock::Stock(const string & co, long n, double pr) { // определение  
company = co; // company – элемент класса  
shares = n;  
share_val = pr;  
set_tot (); // вызов закрытой (private) функции  
}
```

Способы инициализации с помощью конструкторов

Первый - вызвать конструктор явно:

```
Stock s3 = Stock("IBM", 4, 160.0);
```

Это устанавливает значение элемента company объекта s3 равным строке "IBM", значение shares равным 4 и т.д.

Второй способ - вызвать конструктор неявно:

```
Stock s3("IBM", 4, 160.0);
```

```
Stock s3 = Stock("IBM", 4, 160); // тоже самое в явном виде
```

Третий способ - списковая инициализация (C++11):

```
Stock hot_tip = {"IBM", 4, 160.0}; // неявный вызов конструктора
```

```
Stock jock {"IBM"}; // используется {"IBM", 0, 0.0}
```

```
Stock temp {};
```

// конструктор без аргументов

Конструкторы по умолчанию

Конструктор по умолчанию (default constructor) – это конструктор, который используется для создания объекта, когда не предоставлены явные инициализирующие значения.

`Stock s2; // используется конструктор по умолчанию`

Компилятор может сам создавать такой конструктор.

`Stock::Stock() { } // не производит инициализацию.`

После того, как вы определите хотя бы один конструктор класса, компилятор перестанет создавать конструктор по умолчанию и нужно создавать свой (если происходит инициализация типа `Stock s2;`). Можно также использовать конструктор с параметрами, если заданы все их значения по умолчанию.

`Stock::Stock(const string & co="No name", long n=0, double pr=0);`

Деструкторы

В случае использования конструктора для создания объекта программа отслеживает этот объект до момента его исчезновения. В этот момент программа автоматически вызывает специальную элемент-функцию с названием деструктор.

Деструктор имеет специальное имя, формируемое из имени класса и предваряющего его символа тильды (~).

```
~Stock();
```

Подобно конструктору, деструктор не имеет ни возвращаемого значения, ни объявляемого типа. Однако в отличие от конструктора, деструктор не должен иметь аргументов. Обычно деструктор не должен явно вызываться в коде.

Как правило, деструктор нужен для освобождения памяти, выделенной операцией new.

Использование const

Объекты с **const**: после инициализации нельзя менять элементы объекта ни напрямую (если они **public**), ни через вызов элементов-функций (если они меняют элементы).

```
const Stock land = Stock("Apple"); // land - постоянная
```

Элементы-функции с **const**:

```
void show() const; // обещает не изменять вызываемый объект  
land.show(); // не меняет элементы объекта land
```

Указатель `this`

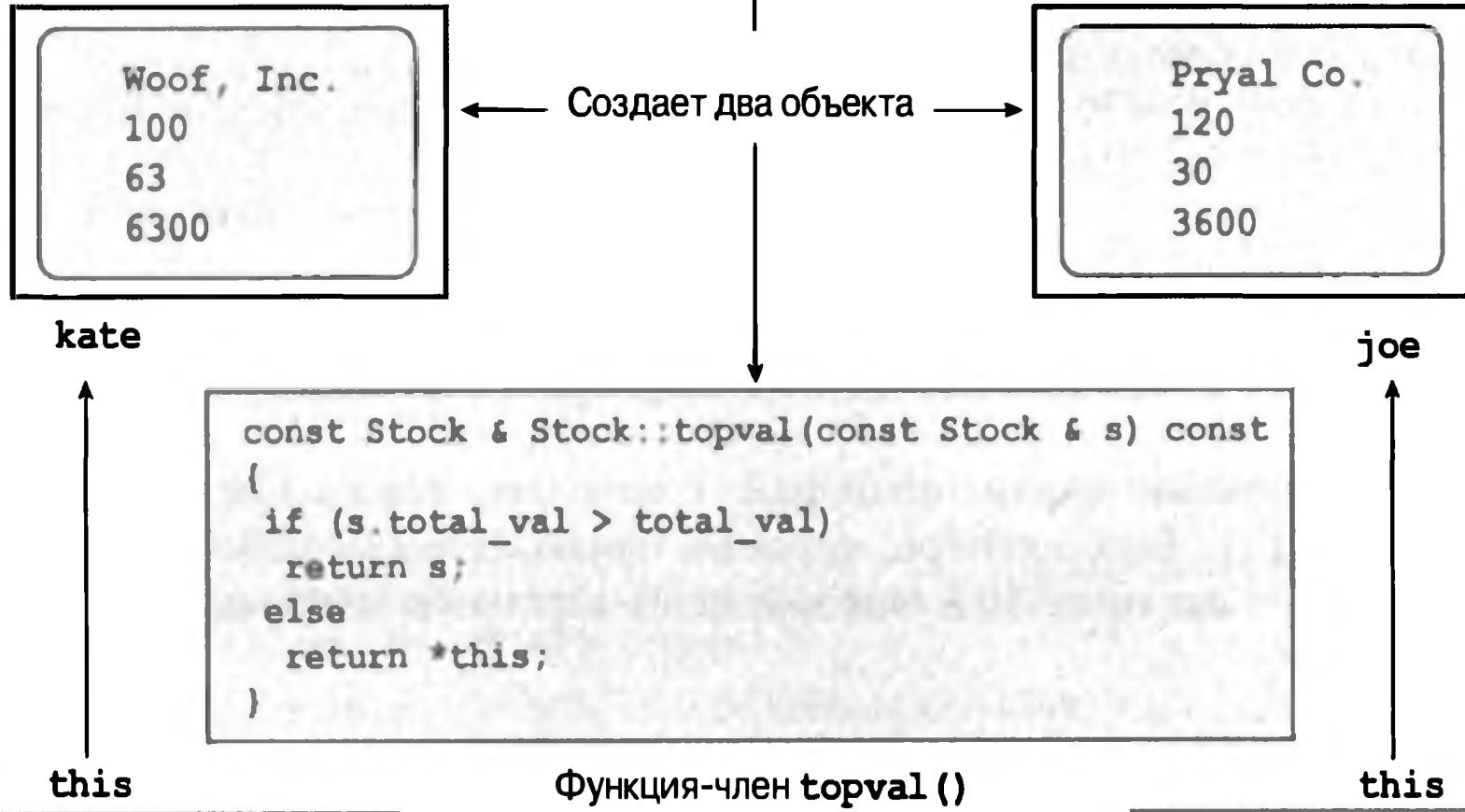
Помимо элементов декларируемых в явном виде, каждый объект имеет встроенный элемент `this` — указатель на себя.

При вызове какого-то метода (включая конструкторы и деструкторы) значение указателя `this` передаётся неявным образом и внутри метода `this` будет указывать на вызывающий объект.

Внутри метода вызвавший его объект становится `*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.

```
Stock kate("Woof, Inc.", 100, 63);  
Stock joe("Pryal Co.", 120, 30);
```



Вызывает `topval()` с `kate`, поэтому
s — это `joe`, `this` указывает на `kate`,
а `*this` представляет собой `kate`

Вызывает `topval()` с `joe`, поэтому
s — это `kate`, `this` указывает на `joe`,
а `*this` представляет собой `joe`

Использование указателя this для идентификации элементов объекта

```
void Stock::update(double price) { // функция-элемент класса Stock
    share_val = price; // изменение share_val на price
    set_tot();          // вызов метода set_tot(), элемент этого класса
}
```

```
Stock s3 = Stock("IBM", 4, 160.0);
s3.update(200); // изменили цену акции
```

```
void Stock::update(double share_val) { // функция-элемент класса Stock
    // share_val = share_val; // так работать не будет
    this->share_val = share_val; // надо использовать указатель this
    set_tot();                // доступ к private set_tot(), элемент этого класса
}
```

Массив объектов

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

Такое объявление требует либо отсутствия у класса явно определённых конструкторов (при этом используются неявные, ничего не делающие конструкторы), либо, как и в рассматриваемом случае - чтобы был явно определён конструктор по умолчанию.

```
mystuff[0].update(120.0); // применяет update() к первому элементу
```

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

Для инициализации элементов массива можно использовать конструктор. В этом случае необходимо вызывать конструктор для каждого индивидуального элемента:

```
const int STKS = 4;  
Stock mystocks[STKS] = {  
    Stock("NanoSmart", 12.5, 20),  
    Stock(),  
    Stock("Monolithic Obelisks", 130, 3.25)  
};
```

Объект `mystocks[3]` также будет инициализироваться неявным вызовом конструктора по умолчанию `Stock()`.

Перегрузка операций

C++ позволяет определять несколько функций с одинаковыми именами и разной сигнатурой (списками аргументов). Это перегрузка функций или функциональный полиморфизм.

Цель такой перегрузки – позволить использовать одно и то же имя функции для некоторой базовой операции, несмотря на то, что она применяется к данным разных типов.

Перегрузка операций расширяет концепцию такой перегрузки на операции, позволяя трактовать их множеством способов.

Например, операция `*`, когда применяется к адресу, выдает значение, хранимое по этому адресу. Но использование `*` с двумя числовыми величинами означает их перемножение.

С++ позволяет распространить перегрузку операций на *пользовательские* типы, разрешая, например, применять символ + для сложения двух объектов.

Например, общей вычислительной задачей является сложение двух массивов. Обычно это выглядит подобно следующему циклу for:

```
for (int i = 0; i < 20; i++)
    arr3[i] = arr1[i] + arr2[i]; // поэлементное сложение
```

Но в С++ можно определить класс, который представляет массивы и перегружает операцию + таким образом, что станет возможным приведенный ниже код:

```
arr3 = arr1 + arr2; // сложить два объекта-массива
```

Пример перегрузки – сложение двух переменных из класса string, используя знак +.

```
string a = "Hello, ";
string b = "World!";
string c = a + b; // перегруженная операция
```

Другие операции (-,*,/) не перегружены для string.

Для перегрузки операции используется специальная форма функции, называемая функцией операции (или функция оператора, operator function).

Функция оператора имеет следующую форму, в которой X - это символ перегружаемой операции:

operatorX(список аргументов)

operator+() – перегрузка оператора +

operator*() – перегрузка оператора *

operator@() – нельзя, так как оператора @ не существует.

Перегрузка оператора на примере класса Time

```
class Time {  
private:  
    int hours, minutes;  
public:  
    Time(int h=0, int m=0);  
    Time Sum(const Time & t) const;  
    void show() const {std::cout << hours << ":" << minutes << std::endl;};  
};  
  
Time::Time(int h, int m){ hours=h; minutes=m; }  
  
Time Time::Sum(const Time & t) const{//  
    Time s;  
    s.minutes = minutes + t.minutes;  
    s.hours = hours + t.hours + s.minutes / 60;  
    s.minutes %= 60;  
    return s;  
}  
  
int main (void) {  
    Time coding(2, 40);  
    Time fixing(5, 55);  
    Time testing(2, 10);  
  
    Time total;  
    total = coding.Sum(fixing);  
    total.show();  
    total = coding.Sum(fixing).Sum(testing);  
    //total = (coding.Sum(fixing)).Sum(testing);  
    total.show();  
}
```

Для сложения двух объектов мы сначала используем метод Sum()

Метод Sum() вызывается объектом Time, принимая второй объект Time в качестве аргумента, а возвращает объект Time.

8:35
10:45

```

class Time {
private:
    int hours, minutes;
public:
    Time(int h=0, int m=0){ hours=h; minutes=m; }
    //Time Sum(const Time & t) const;
    Time operator+(const Time & t) const;
    void show() const {std::cout << hours << ":" << minutes << std::endl; }
};

//Time Time::Sum(const Time & t) const{//  

Time Time::operator+(const Time & t) const{
    Time s;  

    s.minutes = minutes + t.minutes;  

    s.hours = hours + t.hours + s.minutes / 60;  

    s.minutes %= 60;  

    return s;
}  

int main (void) {
    Time coding(2, 40);
    Time fixing(5, 55);
    Time testing(2, 10);

    Time total;
    // total = coding.Sum(fixing);
    total = coding + fixing; //total = coding.operator+(fixing);
    total.show();
    // total = coding.Sum(fixing).Sum(testing);
    total = coding + fixing + testing;
    //total = coding.operator+(fixing).operator+(testing);
    total.show();
    return 0;
}

```

Заменим метод Sum() на operator+().
 Это позволяет использовать удобное обозначение в виде оператора:
 total=coding+fixing;

последовательное
применение операторов

Ограничения на перегрузку операторов

- Хотя бы один объект, на который распространяется действие перегруженного оператора, должен быть введён пользователем (user-defined). Операторы для встроенных типов языка C++ не перегружаются.
- Перегруженный оператор должен сохранять синтаксис (количество operandов) изначального оператора.

`int x;`

`Time t1;`

`% x; // неправильно, так как оператор % (остаток) требует два операнда`
`% t1; // тоже неправильно, так как используется один операнд`

- Нельзя использовать новые символы

- Нельзя перегружать некоторые операторы, например: `sizeof`, `.`(доступ к члену), `::` (область определения), `?:` (оператор условия)

Можно перегружать следующие операции:

+	-	*	/	%	^
&		~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
	++	--	,	->*	->
()	[]	new	delete	new []	delete []

Что такое «друзья» (friends) в C++?

Существует три разновидности «друзей»:

- *дружественные функции*;
- дружественные классы; (изучать не будем)
- дружественные элементы-функции (изучать не будем).

Объявляя функцию другом класса (дружественной классу), вы позволяете ей иметь те же привилегии доступа, что и у элементов-функций класса.

Дружественные функции не вызываются объектами, то есть это отдельные функции с привилегированным доступом к элементам класса.

Перегрузка оператора *

```
class Time {  
private:  
    int hours, minutes;  
public:  
    Time(int h=0, int m=0);  
    Time operator*(double mult) const;  
    void show() const {std::cout << hours << ":" << minutes << std::endl;}  
};  
  
Time::Time(int h, int m){ hours=h; minutes=m; }  
  
Time Time::operator*(double mult) const {  
    Time result;  
    long tot_minutes = mult*(hours * 60 + minutes);  
    result.hours = tot_minutes/ 60;  
    result.minutes = tot_minutes % 60;  
    return result;  
}  
  
int main (void) {  
    Time A(2, 40);  
    Time B=A*2;  
  
    B.show();  
}
```

Операция * вызывается объектом слева от знака *, беря в качестве аргумента число справа.

5:20

Левый операнд – это вызывающий объект. То есть

`A = B * 2;`

транслируется в следующий вызов метода:

`A = B.operator*(2);`

Приведенный ниже оператор не соответствует методу:

`A = 2 * B; // так работать не будет, слева должен быть объект, а не число`

Для умножения числа на объект можно написать функцию, не являющуюся элементом класса, и перегружающую операцию `*`:

`A = operator* (2, B); // или A=2*B;`

Заметим, что оператор `*` требует два аргумента.

Эта функция должна иметь следующий прототип:

`Time operator* (double m, const Time & t) ;`

Однако такая функция не имеет непосредственного доступа к закрытым данным класса. Для обеспечения такого доступа ее можно сделать дружественной классу `Time`.

Создание друзей

Первый шаг в создании дружественной функции предусматривает помещение прототипа в объявление класса и предварение его префиксом в виде ключевого слова `friend`:

```
friend Time operator* (double m, const Time & t);  
// размещается в объявлении класса
```

На основе этого прототипа можно сделать два вывода.

- Несмотря на то что функция `operator* ()` присутствует в объявлении класса, она не является элементом (методом) класса. Поэтому она не вызывается через операцию членства `(.)`.
- Несмотря на то что функция `operator* ()` не является элементом класса, она имеет те же права доступа, что и методы класса.

Перегрузка оператора * (дружественная функции)

```
class Time {  
private:  
    int hours, minutes;  
public:  
    Time(int h=0, int m=0);  
    Time operator*(double mult) const;  
    friend Time operator* (double mult, const Time & t);  
    void show() const{std::cout << hours << ":" << minutes << std::endl;}  
};  
  
Time::Time(int h, int m){ hours=h; minutes=m; } // constructor  
  
Time Time::operator*(double mult) const {  
    long tot_minutes = mult*(hours * 60 + minutes);  
    return Time(tot_minutes/60, tot_minutes%60);  
}  
  
Time operator*(double mult, const Time & t){  
    long tot_minutes = mult*(t.hours * 60 + t.minutes); //access to private  
    return Time(tot_minutes/60, tot_minutes%60);  
}  
  
int main (void) {  
    Time A(2, 40);  
    Time B=A*2; // Time B = A.operator*(2);  
    Time C=2*A; // Time C = operator*(2, A);  
  
    B.show();  
    C.show();  
    return 0;  
}
```

Поскольку перегруженная * не является элементом класса, добавлять квалификатор **Time::** не нужно. Ключевое слово **friend** также не используется в определении.

5:20
5:20

Перегрузка оператора * (обычная функции)

```
class Time {  
private:  
    int hours, minutes;  
public:  
    Time(int h=0, int m=0);  
    Time operator*(double mult) const;  
    //friend Time operator* (double mult, const Time & t);  
    void show() const{std::cout << hours << ":" << minutes << std::endl;};  
};  
  
Time::Time(int h, int m){ hours=h; minutes=m; } // constructor  
  
Time Time::operator*(double mult) const {  
    long tot_minutes = mult*(hours * 60 + minutes);  
    return Time(tot_minutes/60, tot_minutes%60);  
}  
  
Time operator*(double mult, const Time & t){ // not a friend of Time  
    return t*mult; // access to private only through the member *  
}  
  
int main (void) {  
    Time A(2, 40);  
    Time B=A*2; // Time B = A.operator*(2);  
    Time C=2*A; // Time C = operator*(2, A);  
  
    B.show();  
    C.show();  
    return 0;  
}
```

Здесь мы переписали операцию * так, что нам не нужен доступ к закрытым элементам класса.

Внутри новой операции * мы используем другую операцию *, которая является элементом класса.

Операцию * можно сделать и дружественной классу Time.

Сравнение двух подходов: Функция – элемент класса или друг на примере сложения. На практике можно использовать оба подхода.

```
class Time {  
    ....  
public:  
    ....  
    Time operator+(const Time &) const;  
    ....  
};  
  
Time Time::operator+(const Time & t) const  
{  
    Time sum;  
    ....  
    return sum;  
}  
int main(void){  
    Time t1(1,20);  
    Time t2(2,30);  
    Time tot;  
    tot = t1 + t2;  
    // tot = t1.operator+(t2);  
    ....  
}
```

```
class Time {  
    ....  
public:  
    ....  
    friend Time operator+(const Time &, const Time &);  
    ....  
};  
  
Time operator+(const Time &t1, const Time &t2)  
{  
    Time sum;  
    ....  
    return sum;  
}  
int main(void){  
    Time t1(1,20);  
    Time t2(2,30);  
    Time tot;  
    tot = t1 + t2;  
    // tot=operator+(t1,t2);  
    ....  
}
```

Операторы: бинарные (требуют два операнда) и унарные (требуют один operand) (binary and unary operators)

```
class Time {  
private:  
    int hours, minutes;  
public:  
    Time operator-() const;  
};
```

До этого мы рассматривали перегрузку бинарных операторов +, *.

Здесь рассмотрим перегрузку унарного оператора смены знака.

```
Time Time::operator-() const { // операция смены знака, унарный  
    return Time(-hours, -minutes);  
}
```

```
int main(){  
    Time t1(10,22);  
    Time t2 = -t1; // эквивалентно полной записи t1.operator-()  
    return 0;  
}
```

Так как оператор смены знака (-) унарный, то его вызывает объект справа (t1). Аргументов этот оператор не требует.

Обратите внимание, что для символа (-) мы можем перегрузить и бинарную операцию вычитания.

Перегрузка операции << для созданного класса Time

Заметим, что cout – глобальный объект класса ostream (поток вывода).

```
int x=3;
```

```
cout << x; // операция направления значения переменной x в поток вывода
```

Мы можем перегрузить оператор << для класса Time следующим образом:

```
void operator<<(ostream & os, const Time & t) {  
    os << t.hours << " hours, " << t.minutes << " minutes";  
}
```

Декларирование << как друга (для доступа к элементам hours и minutes) позволит написать:

```
cout << trip; // будет работать, cout - первый аргумент, trip - второй аргумент
```

Однако уже вот так применить не получится:

```
cout << "Trip time: " << trip << " (Tuesday) \n"; // не будет работать
```

Что надо сделать, чтобы использовать оператор << последовательно как с числами?

```
int x=3, y=5;
```

```
cout << x << y; // (cout << x) << y; сначала применяется к x, а затем к y.
```

Тогда для последовательного применения << такая операция должна возвращать ссылку на ostream, а не void. Тогда сделаем так:

```
ostream & operator<< (ostream & os, const Time & t) {  
    os << t.hours << " hours, " << t.minutes << " minutes";  
    return os;  
}
```

Конец лекционного материала

Динамическая память и классы

Предположим, что мы хотим разработать новый класс `String` для работы со строками. Нам надо тогда предусмотреть место для их хранения, написать операции создания, копирования и т.д.

Динамическая память и классы

```
class String { // новых класс, похожий на стандартный std::string
private:
    char *str; // указатель на память, где хранится строка
    int len;
    static int num_strings; // количество созданных объектов типа String
.....
```

```
public:
.....
```

```
};
```

- Особые элементы-функции, создаваемые в C++ автоматически:
1. Конструктор без аргументов (default constructor) A::A(){}
 2. Деструктор (default destructor) A::~A(){}
 3. Конструктор копирования (default copy constructor) A::A(const &A)
 4. Оператор присвоения (default assignment operator)
 5. Оператор получения адреса (указатель this)

String::String() { } // неявный конструктор (implicit constructor) без аргументов

```
String::String(){ // явный конструктор (explicit constructor) без аргументов
    операторы;
}
String::String(const char *s="...") { // может использоваться как констр. без аргумен.
    операторы;
}
```

Динамическая память и классы

Использование операции `new` для инициализации элементов-указателей объекта требует особой внимательности. В частности, вы должны следовать таким рекомендациям.

1. Если для инициализации указателя в конструкторе применяется операция `new`, то в деструкторе нужно использовать операцию `delete`.
2. Операции `new` и `delete` должны быть согласованными. Операции `new` должна соответствовать операция `delete`, а операции `new []` – операция `delete []`.
3. Если применяется несколько конструкторов, все они должны единообразно использовать операцию `new` – либо все со скобками, либо все без скобок. В классе существует только один деструктор, и все конструкторы должны быть совместимы с ним. При этом допустимо инициализировать указатель с помощью операции `new` в одном конструкторе и с помощью нулевого указателя (`NULL` или `nullptr` в C++11) – в другом, поскольку к нулевому указателю можно применять операцию `delete` (со скобками или без них).

4. Если класс содержит элементы, которые являются указателями, инициализированными операцией new, необходимо определить явно конструктор копирования (создание новых объектов из существующих, пересылка объектов как аргументов и т.д.), в котором инициализация одного объекта другим выполняется с помощью глубокого копирования. Без явного определения при компилировании автоматически создается версия, в которой просто копируются все элементы объекта. Конструктор копирования должен выделять память для хранения копируемых данных и копировать эти данные, а не только их адрес. Он должен обновлять все статические члены класса, чьи значения затрагиваются данных процессом.

```
String s1=String("abc"); // конструктор объекта из строки String::String(const char* );
String s2=s1; // конструктор копирования String::String(const String &);
String *s2=new String(s1); // тоже используется конструктор копирования
String s3;
s3=s1; // используется оператор присваивания =, в зависимости от
       // реализации, может вызываться конструктор копирования
String::String(const String & st) { // конструктор копирования
    num_strings++; // при необходимости обновление статического элемента
    len = st.len; // та же длина, что и у копируемой строки
    str = new char [len + 1] ; // выделение памяти
    std::strcpy(str, st.str) // копирование строки в новое место
}
```

5. Необходимо определить операцию присваивания, в которой копирование одного объекта в другой осуществляется с помощью глубокого копирования.

```
String & String::operator=(const String & st) {  
    if (this == &st) // присваивание объекта самому себе  
        return *this; // делать ничего не надо  
    delete [] str; // освобождение старой строки  
    len = st.len; // найти размер  
    str = new char [len + 1]; // получение памяти для новой строки  
    std::strcpy(str, st.str); // копирование строки  
    return *this; // возврат ссылки на вызвавший объект  
}
```

То есть метод должен проверить наличие присваивания объекта самому себе, освободить память, на которую ранее указывал элемент-указатель, скопировать данные, а не только их адрес, и возвратить ссылку на вызвавший объект.

```
String s1=String("Hello");  
String s2, s3;  
s3=s2=s1; // полная запись s3.operator=(s2.operator=(s1));
```

Типичные ошибки в конструкторе и деструкторе

```
String::String() { // конструктор без аргументов
    str = "default string"; // неверно: не хватает new []
    len = std::strlen(str);
}
```

```
String::String(const char * s) { // конструирование из строки
    len = std::strlen (s);
    str = new char;      // неверно: не хватает []
    std::strcpy(str, s); // неверно: некуда копировать
}
```

```
String::~String() { // деструктор
    delete str; // неверно, нужно использовать delete [] str;
}
```

Правильно:

```
String::String(const String & st) { // конструктор копирования
    len = st.len;
    str = new char [len + 1]; // правильно: выделение памяти
    std::strcpy(str, st.str); // правильно: копируется значение st
}
```

```
String::String() { // создание пустой строки
    len = 0;
    str = new char[1]; // используется new с []
    str[0] = '\0';
}
```

```
String::String(){
    static const char *s = "C++";
    len = std::strlen(s);
    str = new char[len + 1];
    std::strcpy(str, s);
}
```

```
String::String(){
    len = 0;
    str = 0; // NULL, or, with C++11, str = nullptr;
}
```

Указатели и объекты

Указатель на объект объявляется как обычно:

```
String *glamour; // объявление указателя без инициализации
```

Указатель можно инициализировать адресом существующего объекта:

```
String sayings[10]; // массив из 10 объектов класса String
```

```
String *second = &sayings[1];
```

```
String *third = sayings + 2;
```

Указатель можно инициализировать с помощью операции new, при этом создается новый объект:

```
String *favorite = new String (sayings[2]);
```

Использование операции new с классом вызывает соответствующий конструктор класса для инициализации вновь созданного объекта:

```
String *gleep = new String; // вызов конструктора по умолчанию  
String *glop = new String ("ox ox ox"); // вызов конструктора  
                                // String (const char *)  
String *favorite = new String(sayings[2]); // вызов конструктора  
                                         // String(const String &)
```

Для доступа к методу класса через указатель применяется операция ->:
if (sayings[i].length () < shortest->length())

Для получения объекта применяется операция разыменования (*):

```
String *first=&sayings[0];  
if (sayings [i] < *first) // перегруженное сравнение значений объектов  
first = &sayings[i]; // присваивание адреса объекта
```

Автоматические преобразования и приведения типов в классах

Преобразования встроенных типов языка C++:

`long count = 8; // значение 8 типа int преобразуется в тип long`

`double time =11; // значение 11 типа int преобразуется в тип double`

`int side = 3.33; // значение 3.33 типа double преобразуется в тип int (3)`

`int * p = 10; // ошибка, несовместимые типы.`

Но можно использовать приведение типа:

`int * p = (int *) 10; // нормально, но особого смысла нет`

Как мы можем осуществлять конверсию между встроенными типами и пользовательскими типами?

Любой конструктор класса, принимающий единственный аргумент, может служить функцией преобразования, преобразуя тип аргумента в тип класса.

Конструктор вызывается автоматически, когда значение типа аргумента присваивается объекту.

Предположим, что имеется класс String с конструктором, который принимает один аргумент типа char *: `String::String(const char *)`;

Тогда можно:

```
String bean = "pinto"; // преобразует тип char * в тип String с помощью  
// неявного вызова конструктора String(const char *);
```

```
explicit String(const char *); // использование explicit в декларации
```

Если объявлению конструктора предшествует ключевое слово `explicit`, то конструктор может быть использован только для явного преобразования:

```
String bean = String ("pinto") ; // преобразует тип char * в тип String явным образом  
String bean = (String) "pinto";  
String bean = "abs"; // не работает, если декларировано explicit String(const char *);
```

В зависимости от перегрузки операторов, могут вызываться различные функции.

```
String bean;
```

```
bean = "pinto";
```

После вызова конструктора **String::String()**, может быть следующее:

1. Если есть перегрузка оператора присваивания

```
String & String::operator=(const char *s);
```

то будет вызываться он.

2. Если нет такой перегрузки, то может вызываться конструктор

```
String::String(const char * s);
```

а затем перегруженный оператор

```
String & String::operator=(const String & s);
```

Для того, чтобы узнать какие функции вызываются, то можно внутри функции что-то распечатать. Иногда также удобно использовать переменную **__func__**, которая имеет тип **char *** и содержит имя функции.

```
cout << __func__ << endl; // распечатает имя выполняемой функции
```

Но перегруженные функции имеют одинаковые имена.

Автоматическая конверсия из встроенного типа в созданный класс (на примере типа `char *` и класса `String`)

Конверсия при помощи конструктора `String::String(const char *)` происходит при:

1. При инициализации объекта `String` типом `char*`;
2. При присвоении объекту типа `String` значения типа `char*`;
3. При пересылке значения `char *` в качестве аргумента функции, ожидающей тип `String`.
4. При попытке возврата значения типа `char *` функцией, которая должна возвращать `String`.
5. Во всех перечисленных ситуациях, когда какой-либо встроенный тип может однозначно конвертироваться `char *`.

Конверсия из созданного класса в другой тип

Для преобразования из класса в другой тип (int, char, double и т.д.) потребуется определить функцию преобразования и предоставить инструкции о том, как выполнять это преобразование.

Функция преобразования должна быть функцией-элементом, без спецификация возвращаемого типа, без аргументов. Если она преобразует в тип имяТипа, то ее прототип должен выглядеть следующим образом:

```
operator имяТипа();
```

```
operator int(void); // декларация оператор преобразования String в int  
operator int(); // или так
```

```
operator int(void){return len;} // декларация и пример определения
```

Можно также использовать explicit перед декларацией.

```
String s="abs"; // создание строки
```

```
int a = int(s); // явный вызов преобразования
```

```
int a = (int)s; // явный вызов преобразования
```

```
int a = s; // неявный вызов преобразования
```

Конец лекционного материала