Объектно-ориентированное программирование с использованием языка C++

ШАГ компьютерная АКАДЕМИЯ

Урок №5

Объектно-ориентированное программирование с использованием языка C++

Содержание

Еще раз о перегрузке...

Сегодня мы с вами подводим итоги по перегрузке операторов. Поэтому в данном разделе мы остановимся на рассмотрении еще нескольких примеров.

Оператор "круглые скобки" ()

Оператор "круглые скобки" чаще всего используется, в тех ситуациях, когда нам необходимо дать пользователю класса возможность изменить содержимое объекта уже после того, как последний создан. То есть, с помощью данного оператора мы убираем из класса функции типа SetValue:

```
#include <iostream>
using namespace std;
class MyPoint{
    int X;
    int Y;
public:
    MyPoint(){
        X=0;
        Y=0;
    }
    //функция, которая используется для изменения
    //значений полей объекта уже после его создания
    //именно от нее мы будем избавляться с помощью
    //перегрузки круглых скобок
    void SetValue(int x,int y){
       X=x;
       Y=y;
```

```
    }
    void Show(){
        cout<<X<<" "<<Y<<"\n\n";
    }
};

void main(){
    MyPoint P;
    P.Show();

    //изменение значений полей объекта
    P.SetValue(2,3);
    P.Show();
}
```

Как видите, приведенный пример класса достаточно прост. Упростим его еще больше :))). Для этого используем "круглые скобки" со следующим синтаксисом:

```
Тип_Возвращаемого_Значения operator () (список_аргументов);
```

Модифицируем код:

```
#include <iostream>
using namespace std;
class MyPoint{
    int X;
    int Y;
public:
    MyPoint(){
        X=0;
        Y=0;
    }

    // перегруженный оператор ()
    void operator()(int x,int y){
        X=x;
```

```
        Y=y;
    }
    void Show(){
        cout<<X<<" "<<Y<<"\n\n";
    }
};

void main(){
    MyPoint P;
    P.Show();

    //изменение значений полей объекта
    P(2,3);
    P.Show();
}
```

Достаточно просто, не правда ли??? Однако всегда есть особенности которые вам необходимо запомнить:

1. В отличие от бинарных и унарных операторов, оператор вызова функции может принимать любое количество аргументов (от 0 и больше), другими словами количество параметров функции ограничивается только здравым смыслом.
2. Перегруженный оператор вызова функции не изменяет механизма вызова функции, он изменяет только то, как оператор интерпретируется при его применении к объекту данного класса.

Перегрузка операторов new, new[], delete, delete[]

Сразу следует отметить, что операции new и delete предопределены для любого типа, в том числе и для абстрактного типа, определенного через механизм классов.

То есть создавать их перегрузку в классе не обязательно. Однако, бывает необходимо использовать для объекта какой-то необычный способ выделения памяти. Для этого new и delete можно перегрузить. Например, нужно выделять память и, в случае отсутствия её, создавать файл на диске, в который будет записываться информация. Осуществить такую перегрузку очень просто. Рассмотрим общий синтаксис:

```
Для выделения памяти под один объект и удаление её:
void* operator new(size_t размер){
    код оператора
    return указатель_на_память;
}

void operator delete(void* p){
    код оператора
}

Для выделения памяти под множество объектов и удаление её:
void* operator new[](size_t размер){
    код оператора
    return указатель_на_память;
}

void operator delete[](void* p){
    код оператора
}
```

Некоторые особенности.

1. Параметр размер будет содержать число в байтах, которое необходимо выделить для размещения объекта. Это значение будет сформировано автоматически, т.е. передавать явно его нет необходимости.

2. У параметра размер тип данных size_t. Данный тип — это псевдоним, под которым скрывается unsigned int.
3. В прототипе оператора new можно указать дополнительные параметры, которые будут передаваться при вызове new. При этом — самое главное, чтобы на первом месте в объявлении функции находился параметр типа size_t.
4. Перегруженная функция new должна возвращать указатель на выделенную память.
5. Функция delete получает указатель на область памяти, которую необходимо освободить, при этом, она обязана освободить эту память.
6. В delete также можно передавать дополнительные параметры.

Ну, а теперь, рассмотрим пример:

```
#include <iostream>
using namespace std;
//библиотека для функций
//выделения памяти
#include <malloc.h>

class SomeClass{
    int some;

public:
    //перегруженные операторы new и delete,
    //здесь также используются дополнительные
    //параметры, передаваемые внутрь операторов
    void * operator new(size_t size,char* str = "New");
    void   operator delete(void* ptr);
```

```
    //перегруженные операторы new [] и delete []
    void * operator new [] (size_t fullsize,
                                char* str = "New []");
    void operator delete [] (void* ptr);
};

void * SomeClass::operator new( size_t size,char* str)
{
    cout <<"\n"<<str<<"\n";
    /*
    Здесь, для выделения памяти используется
    стандартная функция void *malloc( size_t size );
    В неё передаётся size - количество байт, которое
    нужно выделить. Если память выделяется, то из
    malloc возвращается адрес начала выделенного блока.
    */
    void*ptr = malloc(size);

    if(!ptr){
        cout<<"\nError memory new!!!\n";
    }
    else{
        cout<<"\nMemory new - OK!!!\n";
    }
    return ptr;
}

void * SomeClass::operator new[]
                        ( size_t fullsize,char* str)
{
    cout <<"\n"<<str<<"\n";
    /*
    Здесь, для выделения памяти используется
    стандартная функция void *malloc( size_t size );
    В неё передаётся size - количество байт, которое
    нужно выделить. Если память выделяется, то из
    malloc возвращается адрес начала выделенного блока.
    */
```

```
    void*ptr = malloc(fullsize);

    if(!ptr){
        cout<<"\nError memory new[]!!!\n";
    }
    else{
        cout<<"\nMemory new[] - OK!!!\n";
    }
    return ptr;
}

void SomeClass::operator delete( void* ptr)
{
    /*
    Для очистки памяти используется стандартная
    функция void free( void * memblock );
    Функция free очищает память, memblock - адрес
    начала очищаемого участка
    */
    free(ptr);
    cout<<"\nDelete memory\n";
}

void SomeClass::operator delete[](void* ptr)
{
    free(ptr);
    cout<<"\nDelete [] memory\n";
}

void main()
{
    /*
    Вызывается оператор
    new( size_t size,char* str="New")
    на место size будет подставлен размер класса
    SomeClass, а str получит значение по умолчанию
    т.е. "New"
    */
```

```
    SomeClass *p = new SomeClass;

    /*
    Вызывается оператор
    new[](size_t size,char* str="New[]")
    на место size будет подставлен размер класса
    SomeClass, умноженный на количество запрашиваемых
    элементов а str получит значение по умолчанию
    т.е. "New[]"
    */
    SomeClass *r = new SomeClass [3];

    /*
    Вызывается оператор delete(void* ptr) на место
    ptr будет подставлен адрес памяти, выделенной
    под объект p
    */
    delete p;

    /*
    Вызывается оператор delete[](void* ptr)
    на место ptr будет подставлен адрес памяти
    выделенной под объект r
    */
    delete[]r;
}
```

Результат работы программы:

```
New
Memory new - OK!!!
New []
Memory new[] - OK!!!
Delete memory
Delete [] memory
```

Дружественные функции

Дружественной функцией класса называется функция, которая, не являясь его компонентом, имеет доступ к его собственным (private) и защищенным (protected) компонентам. Функция не может стать другом класса "без его согласия". Для получения прав друга функция должна быть описана в теле класса со спецификатором friend. Именно при наличии такого описания класс предоставляет функции права доступа к защищенным и собственным компонентам. Например, так:

```
#include <iostream>
using namespace std;

//Класс - прямоугольник
class rect{

    //ширина и высота
    int Width, Height;
    //символ для отображения
    char Symb;
    //Прототип дружественной функции для замены символа:
    friend void friend_put(rect*r,char s);

public:

    //Конструктор.
    rect(int wi, int hi, char si)
    {
        Width = wi;
        Height = hi;
```

```
        Symb = si;
    }
    //Вывод фигуры на экран
    void display ()
    {
        cout<<"\n\n";
        for(int i=0;i<Height;i++){
            for(int j=0;j<Width;j++){
                cout<<Symb;
            }
            cout<<"\n\n";
        }
        cout<<"\n\n";
    }
};
//Дружественная функция замены
//символа в конкретном объекте:
void friend_put(rect*r, char s)
{
    //обращение к закрытому члену здесь допустимо
    //т.к. функция "дружит" с классом
    r->Symb = s;
}
void main ()
{
    //Создание объектов
    rect A(5,3,'A');
    rect B(3,5,'B');
    A. display ();
    B. display ();
    //замена символов с помощью
    //friend-функции
    friend_put(&A,'a');
    friend_put(&B,'b');
    A.display ();
    B.display ();
}
```

Комментарии к примеру

1. Функция friend_put описана в классе rect как дружественная и определена как обычная глобальная функция (вне класса, без указания его имени, без операции :: и без спецификатора friend).
2. Как дружественная она получает доступ к любым данным класса и изменяет значение символа того объекта, адрес которого будет передан ей как значение первого параметра.
3. В функции main создаются два объекта A и B, для которых определяются размеры фигуры и символы для вывода. Затем фигуры показываются на экран.
4. Функция friend_put успешно заменяет символы объектов, что демонстрирует повторный вывод на экран.

Некоторые особенности дружественных функций

Теперь, когда мы познакомились с механизмом создания дружественных функций, остановимся на некоторых важных моментах. Все то, что мы сейчас вам расскажем, связано с тем фактом, что дружественная функция не является компонентом класса. Итак:

1. Дружественная функция при вызове не получает указателя this.
2. Объекты классов должны передаваться дружественной функции только явно через аппарат параметров.
3. Дружественные функции нельзя вызывать через объекты классов, друзьями которых они являются, а

также через указатели на эти объекты. Иначе говоря, следующие действия запрещены:

```
имя_объекта.имя_функции
указатель_на_объект -> имя_функции
```

4. На дружественную функцию не распространяется действие спецификаторов доступа (public, protected, private), поэтому место размещения прототипа дружественной функции внутри определения класса безразлично.
5. Дружественная функция не может быть компонентной функцией того класса, по отношению к которому определяется как дружественная, зато она может быть просто глобальной функцией, а также компонентной функцией другого ранее определенного класса.
6. Дружественная функция может быть дружественной по отношению к нескольким классам.

Кое-что о применении

Использование механизма дружественных функций позволяет упростить интерфейс между классами. Например, дружественная функция позволит получить доступ к собственным или защищенным компонентам сразу нескольких классов. Тем самым из классов можно иногда убрать компонентные функции, предназначенные только для доступа к этим "скрытым" компонентам.

В качестве примера рассмотрим дружественную функцию двух классов "точка на плоскости" и "прямая на плоскости".

1. Класс "точка на плоскости" включает компонентные данные для задания координат (x, y) точки.
2. Компонентными данными класса "прямая на плоскости" будут коэффициенты A, B, C общего уравнения прямой A*x+B*y+C = 0.
3. Дружественная функция определяет уклонение заданной точки от заданной прямой. Если (a, b) — координаты конкретной точки, то для прямой, в уравнение которой входят коэффициенты A, B, C, уклонение вычисляется как значение выражения A*a+B*b+C.

В нижеописанной программе определены классы с общей дружественной функцией, в основной программе введены объекты этих классов и вычислено уклонение от точки до прямой:

```
#include <iostream>
using namespace std;

//Предварительное упоминание о классе line_.
class line_;

//Класс "точка на плоскости":
class point_
{
    //Координаты точки на плоскости.
        float x, y;
public:
        //Конструктор.
        point_(float xn = 0, float yn = 0)
        {
            x = xn;
            y = yn;
        }
        friend float uclon(point_,line_);
};
```

```
//Класс "прямая на плоскости":
class line_
{
    //Параметры прямой.
        float A, B, C;
public:
        //Конструктор.
        line_(float a, float b, float c)
        {
            A = a;
            B = b;
            C = c;
        }
        friend float uclon(point_,line_);
};

//Внешнее определение дружественной функции.
float uclon(point_ p, line_ l)
{
    //вычисление отклонения прямой
    return l.A * p.x + l.B * p.y + l.C;
}
void main()
{
    //Определение точки P.
    point_ P(16.0,12.3);

    //Определение прямой L.
    line_ L(10.0,-42.3,24.0);

    cout << "\n Result" << uclon(P,L) << "\n\n";
}
```

Дружественная перегрузка

Итак, мы рассмотрели дружественные функции и несколько примеров их применения. Однако одним из основных свойств этих специфических функций является то, что с их помощью можно осуществить перегрузку операторов. Такой тип перегрузки носит название дружественной.

Проиллюстрируем особенности оформления операции-функции в виде дружественной функции класса.

```
#include <iostream>
using namespace std;

//класс реализующий работу
//с логическим значением
class Flag
{
    bool flag;
    //дружественная функция (перегрузка
    //оператора ! - замена значения флага
    //на противоположное)
    friend Flag& operator !(Flag&f);

public:
    //Конструктор.
    Flag(bool iF)
    {
        flag = iF;
    }
    //Компонентная функция показа значения флага
```

```
    //в текстовом формате:
    void display(){
        if(flag) cout<<"\nTRUE\n";
        else cout<<"\nFALSE\n";
    }
};
//Определение дружественной
//операции-функции.
//(this не передается, поэтому 1 параметр)
Flag& operator !(Flag & f)
{
    //замена значения на противоположное
    f.flag=!f.flag;
    return f;
}

void main()
{
    Flag A(true);

    //показ начального значения
    A.display();

    //замена значения на противоположное
    //с помощью перегруженного оператора
    A=!A;
    //показ измененного значения
    A.display();
}

Результат выполнения программы:

TRUE
FALSE
```

Глобальная перегрузка

В C++ кроме двух известных вам разновидностей перегрузки (перегрузка в классе и дружественная перегрузка), существует еще одно понятие — глобальная перегрузка, осуществляемая во внешней области видимости.

Допустим, переменные a и b объявлены как объекты класса C. В классе C определен оператор C::operator+(C), поэтому

```
a+b означает a.operator+(b)
```

Однако, также возможна глобальная перегрузка оператора +:

```
C operator+(C,C) {....}
```

Такой вариант перегрузки тоже применим к выражению a+b, где a и b передаются соответственно в первом и втором параметрах функции. Из этих двух форм предпочтительной считается перегрузка в классе. Т. к. вторая форма требует открытого обращения к членам класса, а это отрицательно отражается на строгой эстетике инкапсуляции. Вторая форма может быть более удобной для адаптации классов, которые находятся в библиотеках, где исходный текст невозможно изменить и перекомпилировать.То есть добавить в класс перегрузку в качестве метода класса нереально.

Смешивать эти две формы в программе не рекомендуется. Если для некоторого оператора определены обе формы с одинаковыми типами формальных параметров, то использование оператора может создать двусмысленность, которая, скорее всего, окажется фатальной.

Тем не менее, глобальная перегрузка операторов обеспечивает симметрию, которая также обладает эстетической ценностью. Рассмотрим пример:

```
#include <iostream>
using namespace std;

//класс "точка"
class Point
{
    //координаты точки
    int X;
    int Y;
public:
    //конструктор
    Point(int iX,int iY){
        X=iX;
        Y=iY;
    }

    //показ на экран
    void Show(){
        cout<<"\n++++++++++++++++++++++\n";
        cout<<"X = "<<X<<"\tY = "<<Y;
        cout<<"\n++++++++++++++++++++++\n";
    }

    //перегруженный оператор +
    //метод класса для ситуации Point+int
    Point&operator+(int d){
        Point P(0,0);
        P.X=X+d;
        P.Y=Y+d;
        return P;
    }
    //функции доступа к privat-членам без них
    //глобальная перегрузка невозможна
```

```
    int GetX() const{
        return X;
    }
    int GetY() const{
        return Y;
    }
    void SetX(int iX){
        X=iX;
    }
    void SetY(int iY){
      Y=iY;
    }
};

//глобальная перегрузка для ситуации int + Point
//доступ к private-членам через специальные функции
Point&operator+(int d,Point&Z){
        Point P(0,0);
        P.SetX(d+Z.GetX());
        P.SetY(d+Z.GetY());
        return P;
}

void main()
{
    //создание объекта
    Point A(3,2);
    A.Show();

    //оператор-метод +
    Point B=A+5;
    B.Show();

    //глобальный оператор
    Point C=2+A;
    C.Show();
}
```

Без глобальной перегрузки задача int + Point не решается. Поскольку мы не можем получить доступ к "родному" целому типу (то есть к типу int) и переопределить его операции, обеспечить симметрию простым определением операторов класса не удастся. Потребуется решение с глобальными функциями.

> **Примечание:** Здесь мы могли бы применить дружественную перегрузку, и таким образом избавиться от "функций доступа к private-членам". Однако, если бы тело класса Point было бы для нас закрыто, то вписать в него функцию-друга было бы нереально.

Перегрузка ввода/вывода данных

Для того, что бы закрепить новую полученную информацию о перегрузке рассмотрим возможность перегрузить операторы << и >>. Для начала немного информации.

- Выполнение любой программы C++ начинаются с набором предопределенных открытых потоков, объявленных как объекты классов в файле-библиотеке iostream. Среди них есть два часто используемых нами объекта — это cin и cout.
- **cin** — объект класса istream (Потоковый класс общего назначения для ввода, являющийся базовым классом для других потоков ввода)
- **cout** — объект класса ostream (Потоковый класс общего назначения для вывода, являющийся базовым классом для других потоков вывода)
- Вывод в поток выполняется с помощью операции, которая является перегруженной операцией сдвига

влево <<. Левым ее операндом является объект потока вывода. Правым операндом может являться любая переменная, для которой определен вывод в поток. Например, оператор cout << "Hello!\n"; приводит к выводу в предопределенный поток cout строки "Hello!".

- Для ввода информации из потока используется операция извлечения, которой является перегруженная операция сдвига вправо >>. Левым операндом операции >> является объект класса istream.

Чтобы избежать неожиданностей, ввод-вывод для абстрактных типов данных должен следовать тем же соглашениям, которые используются операциями ввода и вывода для встроенных типов, а именно:

1. Возвращаемым значением для операций ввода и вывода должна являться ссылка на поток, чтобы несколько операций могли быть выполнены в одном выражении.
2. Первым параметром функции должен быть поток, из которого будут извлекаться данные, вторым параметром — ссылка или указатель на объект определенного пользователем типа.
3. Чтобы разрешить доступ к закрытым данным класса, операции ввода и вывода должны быть объявлены как дружественные функции класса.
4. В операцию вывода необходимо передавать константную ссылку на объект класса, поскольку данная операция не должна модифицировать выводимые объекты.

Итак, рассмотрим пример подобной перегрузки:

```
#include <iostream>
using namespace std;

//класс "точка"
class Point
{
    //координаты точки
    int X;
    int Y;
public:

    //конструктор
    Point(int iX,int iY){
        X=iX;
        Y=iY;
    }
    //дружественные функции перегрузки ввода
    //и  вывода данных
    friend istream& operator>>(istream& is, Point& P);
    friend ostream& operator<<(ostream& os, const Point& P);
};

//ввод данных через поток
istream& operator>>(istream&is, Point&P){
    cout<<"Set X\t";
    is >> P.X;
    cout<<"Set Y\t";
    is >> P.Y;
    return is;
}
//вывод данных через поток
ostream& operator<<(ostream&os, const Point&P){
    os << "X = " << P.X << '\t';
    os << "Y = " << P.Y << '\n';
    return os;
}
```

```
void main()
{
    //создание объекта
    Point A(0,0);

    //одиночный ввод и вывод
    cin>>A;
    cout<<A;

    //множественное выражение
    Point B(0,0);
    cin>>A>>B;
    cout<<A<<B;
}
```

Примечание: Кстати!!! В одном из примеров мы использовали константный метод — метод, который не имеет право изменять поля класса. Однако, если какое-то поле объявлено со спецификатором mutable, его значение МОЖНО менять в методе типа const.

Дружественные классы

Пора узнать, что "дружить" могут не только функции. Класс тоже может быть дружественным другому классу.

Особенности "дружбы" между классами

1. Дружественный класс должен быть определен вне тела класса, "предоставляющего дружбу".
2. Все компонентные функции класса-друга будут являться дружественными для другого класса без указания спецификатора friend.
3. Все компоненты класса доступны в дружественном классе, но не наоборот.
4. Дружественный класс может быть определен позже (ниже), чем описан как дружественный.

Рассмотрим простой пример, иллюстрирующий вышесказанное.

```
#include <iostream>
using namespace std;

//упоминание о классе,
//который будет описан ниже
class Banana;

//класс, который будет
//дружественным
```

```
class Apple{
public:
    void F_apple(Banana ob);
};
//класс, который "позволяет" с собой "дружить"
class Banana{
    int x, y;
public:
    Banana(){
        x=y=777;
    }
    //реализация дружбы
    friend Apple;
};

//функция, которая
//автоматически становится "другом"
void Apple::F_apple(Banana ob)
{
    //обращение к private - членам
    cout<<ob.x<<"\n";
    cout<<ob.y<<"\n";
}

void main(){
    Banana b;
    Apple a;
    a.F_apple(b);
}
```

Статические члены класса

Каждый объект одного и того же класса имеет собственную копию данных класса. Можно сказать, что данные класса тиражируются при каждом определении объекта этого класса. Отличаются они друг от друга именно по "привязке" к тому или иному объекту. Это не всегда соответствует требованиям решаемой задачи. Например, при формировании объектов класса может потребоваться счетчик объектов. Безусловно, такой счетчик можно сделать компонентом класса, но иметь его нужно только в единственном числе. Чтобы компонент класса был в единственном экземпляре и не тиражировался при создании каждого нового объекта класса, он должен быть определен в классе как статический и иметь атрибут static.

Статические поля класса

Итак, поле класса можно объявить со служебным словом static. Память под такие поля резервируется при запуске программы, то есть еще до того, как программист явно создаст первый объект данного абстрактного типа. При этом все объекты, сколько бы их ни было, используют эту заранее созданную одну — единственную копию своего статического члена.

Статический член класса должен быть инициализирован после определения класса и до первого описания объекта этого класса. Данное действие производится с

помощью так называемого полного или квалифицированного имени статического члена, которое имеет вид:

```
имя_класса::имя_статического_члена
```

Рассмотрим пример. Реализуем класс object_, в статическом члене которого хранится число существующих в каждый момент времени объектов типа object_.

```
#include <iostream>
#include <string.h>
using namespace std;

class object_{
    char *str;
public:
    //статическое поле класса
    static int num_obj;

    //конструктор
    object_ (char *s){
        str = new char [strlen (s) + 1];
        strcpy ( str, s );
        cout <<"Create " << str <<'\n';

        //Увеличиваем значение счетчика
        num_obj ++ ;
    }

    //деструктор
    ~object_ (){
        cout <<"Destroy " << str << '\n';
        delete str;

        //уменьшаем значение счетчика
        num_obj --;
    }
};
```

```
//Инициализация. Об этом говорит
//ключевое слово int!
int object_::num_obj = 0;

//создание глобальных объектов
object_ s1 ("First global object.");
object_ s2 ("Second global object.");

//вспомогательная функция
void f (char *str) {
    //Локальный объект
    object_ s(str);
    //явное обращение к статическому полю
    //без указания объекта
    cout <<"Count of objects - "
                          <<object_::num_obj<<".\n";
    cout <<"Worked function f()" <<".\n";
}

void main () {
    //явное обращение к статическому полю
    cout <<"Now, count of objects"
                          <<object_::num_obj<<".\n";
    object_ M ("Object in main ().");

    //обращение к статическому полю через объект
    cout <<"Now, count of objects" << M.num_obj <<".\n";

    f ("Local object.");
    f ("Another local object.");

    cout << "Before finish main() count of objects - "
                          <<object_::num_obj<<".\n";
}

Результаты работы программы:
Create First global object.
```

```
Create Second global object.
Now, count of objects 2.
Create Object in main ().
Now, count of objects3.
Create Local object.
Count of objects - 4.
Worked function f().
Destroy Local object.
Create Another local object.
Count of objects - 4.
Worked function f().
Destroy Another local object.
Before finish main() count of objects - 3.
Destroy Object in main ().
Destroy Second global object.
```

Примечание: Обратим внимание, что конструкторы для глобальных объектов вызываются до функции main (), а деструкторы после main().

На статические данные класса распространяются правила статуса доступа. Если статические данные имеют статус private, то к ним извне можно обращаться через компонентные функции. Другими словами, получается, что, если к моменту необходимости обращения к статическому полю типа private, объект класса еще не определен, обращение невозможно. Согласитесь, хотелось бы иметь возможность обойтись без имени конкретного объекта при обращении к статическим данным класса. Эта задача решается с помощью статической компонентной функции-метода.

Статические методы класса

Перед объявлением функции-члена класса можно поставить служебное слово static и она становится — статической компонентной функцией. Такая функция сохраняет все основные особенности обычных (нестатических) функций класса. К ней можно обращаться, используя имя уже существующего объекта класса либо указатель на такой объект.

Кроме того, статическую компонентную функцию можно вызвать следующим образом:

```
имя_класса::имя_статической_функции
```

Статические функции-члены позволяют получить доступ к приватным статическим данным-членам класса, не имея еще ни одного объекта данного типа в программе.

Но, внимание!!!

Для статической компонентной функции не определен указатель this. Когда это необходимо, адрес объекта, для которого вызывается статическая функция-член, должен быть передан ей явно в виде аргумента.

Рассмотрим пример:

```
#include <iostream>
using namespace std;

class prim{
    int numb;
    //статическое поле
    static int stat_;

public:
    prim (int i) {
        numb=i;
    }
```

```
    /*
    Статическая функция. Указатель this не
    определен, поэтому выбор объекта
    осуществляется по явно переданному указателю.
    Поле stat_ не требует указателя на объект,
    т.к. оно общий для всех  объектов класса prim.
    */
    static void func (int i, prim *p = 0) {
        //если хотя бы один объект есть
        if(p)
            p->numb = i;
        //если объектов класса нет
        else
            stat_ = i;
    }

    /*
    Статическая функция обращается только к
    статическому члену класса, никаких указателей
    не требуется.
    */
    static void show(){
        cout<<"stat_="<<stat_<<"\n\n";
    }

    //показ нестатического члена
    void show2(){
        cout<<"numb="<<numb<<"\n\n";
    }
};

//Инициализация статического члена класса.
int prim::stat_ = 8;

void main(){
    /*
    До создания объектов типа prim возможен
```

```
    единственный способ обращения к статической
    функции-члену.
    */
    prim::show ();

    //Можно изменить значение статического члена класса.
    prim::func(10);

    /*
    После создания объекта типа prim можно обратиться
    к статической функции обычным для абстрактных
    типов способом.
    */

    //Создается объект obj и его поле numb
    //становится равным 23.
    prim obj(23);
    obj.show2();

    //Можно изменить значение созданного объекта.
    prim::func(20, &obj); //obj.numb 20.
    obj.show2();

    obj.func(27, &obj); //obj.numb 27.
    obj.show2();
}
```

Шаблон Синглетон (Singleton pattern)

Одним из применений статических членов является конструкция под названием Singleton pattern. Данная конструкция позволяет создавать только один экземпляр класса, и обеспечивает глобальный доступ к этому экземпляру.

Иногда бывает очень важно, чтобы класс мог создать только один экземпляр (объект). Например, система может иметь несколько принтеров, но должен быть

только один спулер принтера. Как мы можем гарантировать, что имеется только один экземпляр класса, и что этот экземпляр доступен?! Можно, например, объявить глобальную переменную. Однако, данный способ плох тем, что мы во-первых засоряем пространство имен, а во-вторых не можем предотвратить создание нескольких экземпляров класса.

Лучшим решением будет класс, который сам отслеживает создание экземпляров и доступ к единственному объекту. Мы можем сделать так, чтобы класс гарантировал, что никакой другой экземпляр не может быть создан.

Надо написать класс, у которого можно создать только один экземпляр, но этим экземпляром должны пользоваться объекты других классов. Вот, пожалуй, самая простая из схем, реализующих эту задачу.

```
#include <iostream>
using namespace std;

class Singleton{

private:
    //указатель на единственный экземпляр класса
    static Singleton*s;
    int k;

    //закрытый конструктор
    Singleton(int i){
        k = i;
    }
public:
    //функция для получения указателя на
    //единственный экземпляр класса
```

```
    static Singleton*getReference(){
        return s;
    }

    //получение значения нестатического члена класса
    int getValue(){
        return k;
    }

    //перезапись значения нестатического члена класса
    void setValue(int i){
        k = i;
    }
};

// Инициализация статического члена класса.
Singleton* Singleton::s=new Singleton(3);

void main(){

    //получение указателя на
    //единственный экземпляр класса
    Singleton*p=Singleton::getReference();

    //работа с компонентом объекта
    cout<<p->getValue()<<"\n\n";
    p->setValue(p->getValue()+5);
    cout<<p->getValue()<<"\n\n";
}
```

Комментарии к примеру

- Класс Singleton окончательный — его нельзя расширить.
- Его конструктор закрытый — никакой метод не может создать экземпляр этого класса.
- Единственный экземпляр s класса Singleton — статический, он создается внутри класса. Однако мож-

но получить указатель на этот экземпляр методом getReference(), изменить состояние экземпляра s методом setValue() или просмотреть его текущее состояние методом getValue().

Безусловно, это только схема — класс Singleton надо еще наполнить полезным содержимым, но идея выражена ясно и полностью.

Домашнее задание

1. Создайте класс Время, в котором реализованы операции сложения, вычитания, сравнения, ввода и вывод на экран. Возможность конвертации времени из американского формата am (pm): 10:00 pm = 22:00, 12:00 pm =00:00
2. Создать класс для работы с матрицами. Предусмотреть, как минимум, функции для сложения матриц, умножения матриц, транспонирования матриц, присваивания матриц друг другу, установка и получение произвольного элемента матрицы. Необходимо перегрузить соответствующие операторы.