

## Тема V. Сложные указатели. Указатели на члены класса.

- виртуальные функции должны быть членами класса
- при использовании библиотечных классов операторы всегда перегружаются глобальными функциями. Если такая функция должна иметь доступ к защищенным данным, объявляйте ее в классе friend
- если аргумент слева базового типа, объявляйте перегруженный оператор глобальной функцией. Если такая функция должна иметь доступ к защищенным данным, объявляйте ее в классе friend
- в остальных случаях рекомендуется перегружать оператор методом класса

- В некоторых случаях нужно уметь объявлять и использовать указатели на члены класса (реже на переменные, чаще на методы).
- Для демонстрации создадим простенькую иерархию классов:

```
class A{
public:
    int m_Private_a; // для примера в нарушение правил ООП
    int m_Public_a; // объявим общедоступную переменную
};

A(int a1=0, int a2=0){m_Private_a=a1; m_Public_a =a2;}

void f();
virtual void vf();

};

class B:public A{
public:
    ...
    virtual void vf();
};

int main()
{
    // Указатели на переменные класса
    A a(1,2);
    //a)
    //int* p = &A::m_Private_a; //ошибка доступа - cannot
    access private member
}
```

```

int* p = &a.m_Public_a; //p - обычный указатель, содержит
                        //адрес переменной m_Public_a
//использование такого указателя ничем не отличается от
                        //использования указателя на обычную
                        //переменную

int t = *p;

//()
```

```

// int A::*pa1 = &A::m_Private_a; //cannot access private
                                //member

int A::*pa = &A::m_Public_a; //a такая переменная ра
                            //содержит не адрес, а смещение
                            //переменной m_Public_a относительно
                            //начала объекта

//использование такого «указателя» специфично

int tmp = a.*pa; //a используется этим смещением
                    //синтаксически можно как указателем
                    //(но только посредством объекта или
                    //указателя на объект)
```

Замечание: так как статический метод класса по сути является глобальной функцией (раздел «Статические методы класса»), указатель на статический метод ничем не отличается от указателя на обычную глобальную функцию.

//void (\*pf) ()=A::f; //ошибка компилятора

void (A::\*pf) ()=A::f; //такой указатель rf содержит просто
адрес функции, но при вызове такой
функции компилятор должен
сформировать дополнительный
параметр - адрес того объекта, для
которого вызывается метод (this)

//указатели на виртуальные методы класса
или указателя на объект

A\* p1 = new A;

```

p1->vf();
(p1->*pvf1)();

A* p2 = new B;
void (B::*pvf2) () = B::vf; //то же самое значение, что и
                             //в pvf1

p2->vf();
(p2->*pvf1)();

}
```

Пример использования:

```

class A{
    int m_a;
public:
    //оба метода имеют одинаковый вид:
    void f1();
    void f2();
    ...
};
```

void (A::\*pvf1) () = &A::vf; //такой указатель содержит не
адрес функции, а адрес «заглушки»,
в которой
 а) по полученному адресу извлекается
 объекта
 б) по индексу, сопоставленному с
 vf,
 извлекается из
 метода
 в) косвенно вызывается метод

## Тема VI. Встроенные объекты (composition). Отношение между классами «содержит».

### Понятие встроенного объекта

В качестве переменной класса A может фигурировать объект другого класса. В этом случае говорят, что объект класса A **содержит** объект класса B. Или объект класса B **встроен** в объект класса A. Например:

```

// Какой из метод требует вызвать для каждого элемента массива, определяю случайным образом во время выполнения:

switch(rand()%2)

{
    case 0:
        pSample = A::f1;
        break;
    case 1:
        pSample = A::f2;
        break;
}

for(int i=0; i<sizeof(ar)/sizeof(A); i++)
{
    (ar[i]).*pSample(); // Вызов метода для каждого элемента массива посредством сформированного указателя
}

```

### Конструирование и уничтожение встроенных объектов.

```

int main()
{
    Rect r; // Создание объекта типа Rect
}

```

Последовательность создания объекта типа Rect выглядит следующим образом:

- компилятор, заранее зная, сколько памяти потребуется для объекта Rect (вместе с вложеннымми объектами Point), сразу же выделяет соответствующий объем памяти;

- если бы класс Rect был производным, компилятор сначала вызвал бы конструктор базового класса;
- вызываются конструкторы (в нашем случае **по умолчанию**) внедряемых объектов Point. Замечание: члены класса инициализируются в порядке их объявления в классе;
- вызывается конструктор Rect (в нашем случае **по умолчанию**).

**Замечание:** порядок разрушения объекта прямо противоположен порядку его создания. То есть в деструкторе Rect встроенные объекты Point гарантированно еще существуют!

### Передача параметров конструктора встроенным объектам.

#### Список инициализации конструктора.

Так как сначала вызываются конструкторы встроенных объектов, возникает та же проблема, что и при передаче параметров базовому классу. Существуют два способа передать параметры конструктора встроенным объектам:

- присваивание в теле конструктора;
- список инициализации конструктора.

Отличие:

Присваивание в теле конструктора	Список инициализации
----------------------------------	----------------------

1. Компилятор, заранее зная, сколько памяти потребуется для объекта Rect (вместе с внедреными объектами Point), сразу же выделяет соответствующий объем памяти.

2. Вызываются конструкторы **по умолчанию** внедряемых объектов Point.

3. Вызывается конструктор Rect в теле конструктора уже пронинциализированные внедренные объекты принимают новые значения

**Рекомендация:** предпочтите инициализацию присваиванию в конструкторах. Это уменьшает накладные расходы на лишний вызов функции.

Замечания:

1. В приведенном примере реализация метода совмещена с объявлением. Если объявление и реализация метода разнесены, то (как и в случае передачи параметров конструктору базового класса) синтаксическая конструкция « : » может быть указана только при определении метода.

```
//Файл rect.h
class Rect{
    ...
    Rect(int left, int top, int right, int bottom); // объявление конструктора
};

//Файл rect.cpp
Rect::Rect(int left, int top, int right, int bottom) :
    m_LeftTop(left, top),
    m_RightBottom(right, bottom)
{
    ...
}

//Реализация конструктора:
void Point::Point() {m_x = m_y = 0;} //так мы поступали с
                                         //переменными класса базового типа
                                         //до сих пор (присваивание в теле
                                         //конструктора)
```

2. Если программист явно с помощью списка инициализации не указал компилятору как следует создавать встроенные объекты, то они будут проинициализированы с помощью конструктора по умолчанию.

3. Переменные **базовых** типов можно также инициализировать с помощью списка (они тоже являются встроенными низкоуровневыми «объектами», только базового типа):

```
Point::Point() {m_x = m_y = 0;} //так мы поступали с
                                         //переменными класса базового типа
                                         //до сих пор (присваивание в теле
                                         //конструктора)

Point::Point(int x, int y) : m_x(x), m_y(y) //а можно и так
                                         //список инициализации
{
}
```

Если для встроенных объектов пользовательского типа использование списка инициализации дает несомненные преимущества, то для базовых типов особого выигрыша не получается, так как в обоих случаях низкоуровневый код генерируется одинаковый.

4. Если Вы реализовали конструктор по умолчанию класса Rect следующим образом:

```
Rect::Rect()
{
    ... //пустое тело
}
```

это означает, что Вас устраивает конструирование встроенных объектов с помощью default конструктора класса Point. Но, если Вы хотите создавать объекты класса Rect по умолчанию с другими значениями, то можно использовать список инициализации, например:

```
Rect::Rect() : m_LeftTop(1, 1), m_RightBottom(100, 100)
```

5. Если класс, с одной стороны, является производным, а, с другой стороны, содержит встроенные объекты, то требуется передать параметры как конструктору базового класса, так и конструкторам встроенных объектов. Для этого все вызовы указываются в одном списке через запятую в любом порядке (а выполняться будет в строго определенном – начиная с конструктора базового класса, а потом конструкторы встроенных объектов в порядке следования их в объявлении класса):

```
class ColoredRect:public Rect{
    ...
    int m_color;
public:
    ColoredRect(int left, int top, int right, int bottom, int color)
        :Rect(left, top, right, bottom),
         m_color(color)
    {
    };
}
```

## Когда без списка инициализации не обойтись

Для рассмотренных примеров можно было использовать список инициализации конструктора или присваивание в теле конструктора. Выбор влияет только на эффективность. Встречаются случаи, когда программист просто обязан использовать список инициализации. Если членами класса являются:

- константные встроенные объекты,
- ссылки.

И то, и другое должно быть проинициализировано при создании объекта!

Например:

```
class A{
    const int m_n;
    int& m_ref;
public:
```

A(int n, int& r) : m\_n(n), m\_ref(r) {} // подумайте:  
может ли параметр r передавать по  
значению???

/ / A() : m\_n(0), m\_ref(???){}; // для default конструктора  
непонятно, как сформировать адрес,  
который должна  
принципиализирована ссылка

};

int main()

{

int n=5;

F A(5,n);

/ / F f(); // ошибка компилятора – все константы и ссылки должны  
быть проинициализированы при  
создании!!!

}

## Порядок инициализации членов класса

Члены класса инициализируются в том порядке, в котором они указаны в классе, поэтому порядок их следования в списке инициализации не имеет при малейшего значения! Если программист не знает этого правила, могут возникнуть нетривиальные ошибки, например:

```
class A{
    int* m_p;
    int m_n;
public:
    A(int a) : m_n(a), m_p(new int[m_n]) {} // сначала
                                                // компилятором будет вызван оператор
                                                // new // будет вызван со случайнм
                                                // значением
                                                // m_n, а потом
                                                // проинициализирована переменная m_n
                                                // значением параметра n
    ...
};

Пример использования встроенных объектов для построения квадратов посредством прямоугольников
```

Замечание: этот прием используется в общенных классах стандартной библиотеки при создании классов-адаптеров. Идея заключается в том, что класс-адаптер содержит защищенный объект подходящего типа. Так как объект защищен, весь его public интерфейс извне недоступен. Поэтому пользователь может использовать только из методов класса-владельца. А владелец реализует свои методы, пользуясь только теми возможностями встроенного объекта и только таким образом, как ему удобно. Например, реализуем класс, абстрагирующий квадрат, посредством встроенного объекта класса прямоугольник.

```
class Rect {
protected:
    int l,r,t,b;
public:
    Rect(int x1,int y1,int x2,int y2){l=x1; r=x2; t=y1;
                                                b=y2;}
    void Inflate(int dl,int dr,int dt,int db){l-=dl; r+=dr;
                                                t-=dt; b+=db;}
};

ФПС СПбГУ
```

```

class Square{
protected:
public:
    void Inflate(int d) {m_r.Inflate(d, d, d, d); }

    Square(int x, int y, int d) :m_r(x, y, x+d, y+d) {}

    int main()
    {
        Square s(1, 1, 10);
        s.Inflate(2);
        //s.m_r.Inflate(1, 2, 3, 4); //ошибка доступа
    }
}

class Pair{
    char name[KEY]; // имя
    int phone; //номер телефона
    protected:
    public:
        Pair() {
            name[0]=0; //пустая строка
            phone = 911;
        }
        Pair(const char* key, int data)
        {
            strcpy(name, key, KEY-1);
            name[KEY-1] = 0;//если больше, чем KEY
            phone = data;
        }
        bool operator==(const char* k)
        {
            return strncmp(name, k, KEY)==0;
        }
}
friend class book;

const int NUM=10; //пусть в записной книжке не может быть больше
//основной класс, абстрагирующий записную книжку. Это не совсем обычный
массив, для которого спрavedлив произвольный доступ. Специфика такого
«Массива» заключается в том, что чтение и запись в массиве осуществляются по-
разному, поэтому реагируют эти две операции посредством двух разных
методов. Добавляются элементы в такой массив «на свободное место», а при
считывании из массива по указанному ключу ищется значение

class book{
    Pair arr[NUM]; //для упрощения введем встроенный массив
    const size_t KEY = 20;
}

Пример использования встроенных объектов - ассоциативный массив»
«ассоциативный массив»
Ассоциативный массив называется массивом, в котором хранятся пары
ключ/значение. Поиск значения происходит по ключу. В приведенном примере
рассматривается не полноценный ассоциативный массив, а, пожалуй, первое
приближение к его настоящей реализаций. Реализуем «записную книжку», в
которой будут храниться пары имя/телефон.

В реализуемом ассоциативном массиве введем ограничение – ключ должен
быть уникальным, то есть двух одинаковых имен быть не может.
Замечание: так как речь пойдет о массиве (не имеет значения его внутренняя
реализация), то очевидно потребуется перегрузка оператора [] . Если для
обычных массивов индекс может быть только целым значением, то
перегруженный оператор [] может принимать параметр любого типа.
Пусть ассоциативный массив содержит пары: имя, номер телефона
pair
    class Square{
protected:
public:
    void Inflate(int d) {m_r.Inflate(d, d, d, d); }

    Square(int x, int y, int d) :m_r(x, y, x+d, y+d) {}

    int main()
    {
        Square s(1, 1, 10);
        s.Inflate(2);
        //s.m_r.Inflate(1, 2, 3, 4); //ошибка доступа
    }
}

class Pair{
    char name[KEY]; // имя
    int phone; //номер телефона
    protected:
    public:
        Pair() {
            name[0]=0; //пустая строка
            phone = 911;
        }
        Pair(const char* key, int data)
        {
            strcpy(name, key, KEY-1);
            name[KEY-1] = 0;//если больше, чем KEY
            phone = data;
        }
        bool operator==(const char* k)
        {
            return strncmp(name, k, KEY)==0;
        }
}
friend class book;

const int NUM=10; //пусть в записной книжке не может быть больше
//основной класс, абстрагирующий записную книжку. Это не совсем обычный
массив, для которого спрavedлив произвольный доступ. Специфика такого
«Массива» заключается в том, что чтение и запись в массиве осуществляются по-
разному, поэтому реагируют эти две операции посредством двух разных
методов. Добавляются элементы в такой массив «на свободное место», а при
считывании из массива по указанному ключу ищется значение

class book{
    Pair arr[NUM]; //для упрощения введем встроенный массив
    const size_t KEY = 20;
}

Школа Практического Программирования
http://www.avalon.ru
+7(812)703-0202

```

Школа Практического Программирования

ՀԱՅԱՍՏԱՆԻ

+7(812)703-0207

```
int m_n; // индекс первого свободного в массиве (или количества записей)
```

```
public:
```

```
book() {m_n = 0; }

...
int& operator[](const char* key) {
    for(int i=0; i<m_n; i++)
    {
        if(ar[i]==key) // ??
            return ar[i].phone;
    }
}

if(m_n<NUM)
{
    strcpy(ar[m_n].name, key);
    ar[m_n].name [KEY-1]=0;
    return ar[m_n].phone;
} else { std::cout<<"full!"; }

}
int main()
{
    Book b; // создаем пустую записную книжку
    B["Marina"] = 1111111;
    B["Alex"] = 2222222;
    cout<< b["Marina"]; // будет выведено 111111
    cout<< b["Boris"]; // будет выведено Not Found
}
```

## Указатели на объекты в качестве членов данных класса.

В качестве переменной класса может фигурировать указатель на объект другого класса.

```
class Y{
    ...
    x* m_px; // переменная класса Y, которая является указателем на объект типа X или массив объектов типа X. При создании экземпляра класса X компилятор вынуждает память под указатель m_px, который будет инициализирован (или не инициализирован) по общим правилам.
}

...
}

Замечание 1: как только в классе появляется указатель — скорее всего, будет динамически выделяться память, поэтому Ваша задача заключается в обеспечении корректного значения этого указателя! => в таком классе должны быть предусмотрены корректно реализованные:
```

- default constructor — Y(){m\_px=0}; - вовремя обнуленный указатель избавит Вас от мучительных поисков ошибок, возникающих при случайном использовании неинициализированного указателя.
- деструктор ~Y()~{delete [] m\_px;} и при необходимости он должен быть объявлен virtual. (С нулевым указателем delete будет работать корректно, но трудно сказать, какие неприятности Вас ожидают в случае «случайного» значения m\_px)
- конструктор копирования
- оператор присваивания

Замечание 2: часто в качестве члена данных класса X фигурирует указатель типа X\* (на объект того же типа). Это позволяет создавать стековые структуры данных типа списков и деревьев

## Ссылки на указатели

Это прием часто используется для того, чтобы вместо пары функций – Set/Get для получения и присваивания нового значения члену класса – указателю. Особенно этот прием удобен при работе со списками:

```
class A{
```

```

A* pPrevious; //указатель на предыдущий элемент
A* pNext; //указатель на следующий элемент
...
public:
    ...
    A* & Next() { return pNext; } //возвращаемая ссылка на
                                  //указатель фактически замаскированным двойным указателем
                                  //=> эту функцию можно использовать как справа, так и слева от «==»
                                  //читать/писать)
A*& Previous() { return pPrevious; }
};

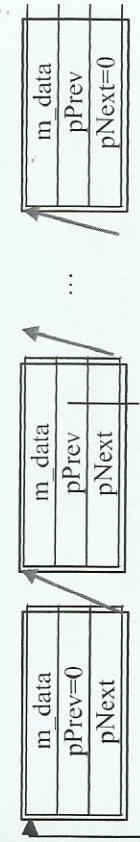
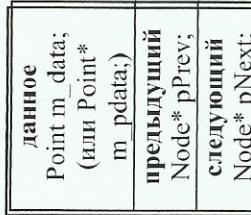
int main()
{
    A a, a1;
    ...
    A* p = a.Next(); //в правой части присвоения
    a.Previous() = &a1; //в левой части присваивания - иначе ошибка компилятора "left operand = must be lvalue"
    // в правой части присваивания
    A a2 = * (a.Next()); //в правой части присваивания
}

```

## Объектно-ориентированный двухсвязный список (однородный). Вложенные объявления классов

Обычно список реализуется посредством вспомогательного класса-обертки. Каждый объект такого класса-обертки содержит собственно данное и дополнительную служебную информацию (связи с соседними элементами).

Замечание: если хранящиеся данные одного и того же типа (список однородный), то эффективнее хранить эти данные как встроенные объекты. Если же требуется объединить объекты разных производных классов (с общим базовым классом), то единственной возможностью является хранение в «обертке» указателя базового типа (а сами объекты должны быть созданы динамически).



```

class List
{
    class Node // вложенные объявления класса. Так как класс Node
    {
        ...
        // если бы не было второго объявления, здесь компилятор выдал бы ошибку!
        A* pA = a2.Next();
    };
    ...
    class Node // вложенный класс Node
    {
        ...
    };
};

```

```

    вспомогательный, то все его члены
    защищены - private

{
    friend class List; // даю права всем методам класса
    List пользователь на любом членами
    класса Node

    Node* pPrev;
    // указатель на предыдущий элемент

    Node* pNext;
    // указатель на следующий элемент

    Point m_data;
    // данные
};

~Node();
Node(Node* p, Node* n, const Point* );
// конструктор

```

примера мы ограничимся одним-единственным данным – указателем на начало списка

```

};

// данные класса List. Замечание: списки реализуют по-разному. В
// качестве данных можно хранить количество элементов, указатель на
// последний элемент в списке...

Node* pHead;
// для
// иначе мы ограничимся одним-
// единственным данным – указателем
// на начало списка

public:
List() {pHead=0; }
~List();

int Size();
void AddToHead(const Point&); // добавить в начало списка
bool Remove(const Point&); // исключить из списка первый
// элемент, значение которого
// совпадает с параметром

...
};

// деструктор списка должен обеспечить освобождение динамически
// захваченной памяти
while( pHead!=0 )

```

```

#include "list.h"

List::Node::~Node() // класс тоже является областью видимости
{
    // Наш деструктор умеет исключать удалаемый элемент из списка
    // Коррекция следующего

    if(pNext !=0) pNext->pPrev = pPrev;
    // Коррекция предыдущего

    if(pPrev!=0) pPrev->pNext = pNext;
}

List::Node::Node(Node* p, Node* n, const Point* d):
m_data(*d) // встроенный
// будет копией параметра
{
    // Наш конструктор умеет создавать очередной узел и подключать
    // его в список, а так как создавать
    // объекты сможет только класс List,
    // то есть гарантия, что указатели p
    // и p указывают на два соседних узла
    // списка.

    // В текущем элементе установлены указатели на его соседей
    pPrev = p;
    pNext = n;

    // Если существует следующий –
    // скорректировали у него
    // указатель на предыдущий

    if( n !=0) n->pPrev = this;

    // Если существует предыдущий –
    // скорректировали у него
    // указатель на следующий

    if( p!=0 ) p->pNext = this;
}

List::~List()
{
    // деструктор списка должен обеспечить освобождение динамически
    // захваченной памяти
}

```

```

{
    Node* p = pHead; //указатель на уничтожаемый Node
    pHead = pHead->pNext; //устанавливаем на следующий
    delete p; //удаляем предыдущий
}

int List::Size()
{
    Node* p = pHead;
    int n=0;
    while (p!=0)
    {
        p = p->pNext;
        n++;
    }
    return n;
}

```

Node\* p = pHead; //указатель на уничтожаемый Node  
pHead = pHead->pNext; //устанавливаем на следующий  
delete p; //удаляем предыдущий

Замечание 1: мы реализовали далеко не все операции для работы со списком. Подумайте, что еще обязательно требуется реализовать для такого сложного класса.

Замечание 2: реализаций двухвязных списков много. Полезным приемом является реализация списка с помощью фиктивных элементов (Node) — «стражей». Именно такую реализацию требуется разработать в лабораторной работе.

### Специфика использования безразмерных массивов

Безразмерные массивы (unsize arrays) – это специфическая особенность компилятора Microsoft. Используются только как члены данных классов и структур при соблюдении следующих ограничений:

- безразмерный массив может быть только последним членом данных.
- такой класс не может быть базовым для другого класса
- если объект такого класса встречается в другой класс, он тоже должен быть последним
- не может иметь виртуального базового класса
- оператор sizeof примененный к такому классу, возвращает количество байт без учета безразмерного массива
- нельзя создавать массивы объектов такого класса

```

void List::AddToHead(const Point& o) // добавить в начало
                                         списка
{
    pHead = new Node(0, pHead, &o);
}

bool List::Remove( Point& ref) //если такой был найден и удален
                                - true, если такого в списке не
                                было - false
{
    Node* p = pHead;

```

void List::AddToHead(const Point& o) // добавить в начало списка

bool List::Remove( Point& ref) //если такой был найден и удален – true, если такого в списке не было – false

Например:

```
class A
{
    unsigned m_n; // количество элементов в массиве
    char m_ar[]; // безразмерный массив
public:
    void set(int n, const char* p) {m_n = n; strcpy(m_ar, p);}
};
```

**Замечание:** при создании объектов такого типа тоже следует помнить, что компилятор не знает, сколько памяти требуется выделить для такого массива, поэтому по умолчанию сам ничего не выделяет.

```
int main()
{
    //так нельзя!
    A a; //для массива память не выделена
}
```

int n = sizeof(a); //4
//по списку инициализаторов компилятор может посчитать, сколько требуется выделить памяти, но для того, чтобы использовать список инициализаторов, все данные должны быть **Public**
// A a = { sizeof("ABC"), "ABC" }; //ошибка компилятора, так как данные защищены

//можно явно выделить требуемый объем памяти динамически:
char ar[80];
cin >> ar;
A\* pa = reinterpret\_cast<A\*>(new char[sizeof(int)] + strlen(ar) + 1);
pa->set(strlen(ar), ar);

## Предварительное неполное объявление класса (**forward declaration**)

На момент использования любого идентификатора компилятор должен знать его свойства, то есть в том месте текста Вашей программы, где компилятор встречает Ваш пользовательский тип, он уже должен «видеть» его объявление. Но иногда встречаются такие ситуации, когда классы «сылаются» друг на друга:

```
class A{//объявление класса A
B* pB; //на этот момент времени компилятор должен знать свойства B
...
B Func(B&); //аналогично
};

class B{//объявление класса B
A a; //на этот момент времени компилятор должен знать свойства A
...
};

class A{
    ...
};

class B{
    ...
};
```

В такой ситуации как бы Вы не переставляли места объявления, компилятор будет выдавать ошибки. Для разрешения таких взаимных зависимостей можно (а иногда и необходимо) указать компилятору, что используемый идентификатор является именем класса, а его объявление компилятор встретит чуть позже. Этот прием называется предварительным неполным объявлением класса. Вывручает он не во всех случаях, а только тогда, когда компилятору достаточно знать, что используемое имя – это имя класса:

```
class B; // предварительное неполное объявление
class A{
    B* pB; //OK – компилятор знает, сколько памяти под указатель, а все осталное можно отложить до конкретного обращения к этому указателю
};

B Func(B); // OK
/B b; //ошибка, так как компилятор должен знать сколько зарезервировать памяти
```

```
};

class B{... A a;};
```

## Тема VII. Статические члены класса

### Статические данные.

Часто возникает следующая ситуация: все объекты класса оперируют с одним и тем же, единственным для всех экземпляров, глобальным данным (данные, которые характеризуют количество или взаимосвязь **всех существующих на данный момент объектов данного типа**). То есть появляются переменные, которые имеют отношение к классу в целом, но не входят (не являются частью) в каждый объект данного класса. Для таких данных вводится понятие и ключевое слово **static**.

Специфика:

- статические переменные размещаются в статической области памяти на стадии компоновки (когда еще не создано ни одного экземпляра класса!) независимо от того, где создается сам объект.
- существует в единственном экземпляре (то есть компоновщик выделяет под нее память только один раз!) независимо от того, сколько создано экземпляров данного класса. Это означает, что для переменных класса с ключевым словом static при создании экземпляра класса компилятор место не резервирует!

```
class X{
    int m_x; //обычная переменная класса (в каждом объекте
              //типа X компилятор отводит под
              //такую переменную память в объекте)

    static size_t count; //статическая переменная (в объекте
                        //память под такую переменную не
                        //резервируется!)

public:
    ...
};
```

- Статическую переменную класса необходимо определить следующим образом (вне функций и тем более вне объявления класса - как глобальные переменные) для того, чтобы компилятор отвел под эту переменную место в статической памяти:

**X.cpp**

```
size_t X::nCount; // объявление и инициализация статической
                  // переменной nCount класса X нулем
                  // по умолчанию.
```

- Определяется и инициализируется статическая переменная одинаково – **независимо от спецификатора доступа:**

```
X::nCount = 1;
cout<<X::count; //???
X x1;
size_t n = sizeof(x1); //???
cout<<x1.count; //???
{
    cout<<x1.count; //???
}
X x2=x1;
cout<<x2.count; //???
cout<<x2.count; //???
```

**Обращение «извне».** Статическая переменная класса **по сути является глобальной**, заключенной в пространство имен (имя класса), поэтому к public статической переменной можно обращаться посредством cout<<X::nCount.

С другой стороны, **формально** статическая переменная является членом класса, поэтому ничто не мешает обращаться кней посредством объекта: cout<<x.nCount<<endl>->nCount;

- Обращение к статической переменной внутри методов класса для программиста – ничем не отличается от обращения к обычной переменной класса, для компилятора – подставляет адрес статической переменной, а не вычисляет базат-смещение
- Следствие: статические члены класса никогда не должны инициализироваться в конструкторе!

Для примера рассмотрим класс, который автоматически «контролирует» количество объектов данного типа, существующих на данный момент выполнения программы. Для решения этой задачи заведем статическую переменную-счетчик и предусмотрим модификацию этой переменной при создании нового объекта и при удалении существующего.

```
class X{
public:
    static int nCount; // счетчик объектов типа X.
    X() {m_x=0; nCount++; }
    ... // подумайте: какие еще методы требуется реализовать?
    ~X() {nCount--; }

};

int main()
{
    static A a; // ошибка – использование неопределенного класса A
}
```

В объявлении класса можно использовать встроенные статические данные того же типа. Например, таким образом можно задать default значение (одинаковое для всех экземпляров класса):

```
class A{
public:
    static int aDefault; //OK
    A(); // ошибка – использование неопределенного класса A
};

A::aDefault = 1;
```

Такое default значение **можно изменить** в процессе выполнения, а значения параметров по умолчанию – нет!

**Замечание 2:** статические константы целого типа можно пропинчизировать в членопредставлении объявления класса:

**Замечание 2:** статические константы целого типа можно проницализировать  
использовав в объявлении класса:

```
class CObject{
```

**Замечание 2:** статические константы целого типа можно пронициализировать членопостроенно в объявлении класса:

```
class A{
public:
    const static int n=1; //это простая константа, вместе
    // которой компилятор будет просто
    // подставлять значение. Область
    // видимости такой константы
    // ограничена классом
    static int count;//=1; - ошибка
};

int A::count; //обязательно!

//const int A::n; // необязательно - если хотим зарезервировать память

virtual const char* GetClassName() const =0;
};

class MyObject:public CObject{
    static char m_name[]; //это только объявление, поэтом
    // могу размер не указывать
    // эквивалентно объявлению extern
    char m_name[];
};

public:
    virtual const char* GetClassName() const { return m_name; }
};
```

```

int main()
{
    int tmp = A::n; // компилятор подставит 1
    char MyObject::m_name[] = "MyObject";
    //а это определение
    //совместное с инициализацией
}

```

переменной `т_памте` имени своего класса и переопределает виртуальный

ההנומינציה הדרתית מושגתה על ידי דוד קפלן, מנהל תיאטרון גשר, על שמו של יוסי עיני, שזכה בפרס על תרומתו לתרבות ישראליות.

Създаден е и новият квартал на града, който се намира в южната част на града.

Выполнения.

Замечание: примененный ниже механизм определения класса объекта в период выполнения применимся в библиотеке MFC и появился задолго до того, как определение типа в период выполнения (runtime type information – RTTI) ввели в стандарт C++. Стандартные классы в языке C++ определяют типы данных в виде таблиц, виртуальных функций выводит имя класса «шаблоном» объекта // посредством

спецификации СТП, ЧАП и АЛП в библиотеке используется прежний механизм.

Пример является упрощением.

Все классы MFC (кроме **вспомогательных**) наследуются от **CObject**. Если свойства класса хотели узнать имя класса объекта, достаточно было бы определить в классе **СObject** виртуальную функцию:

```
virtual CString GetClassName() const;
```

которой ограничена именем класса, поэтому (если она public) фактически она может быть доступна извне.

Можно также посредством объекта или указателя на объект.

<http://www.avalon.ru>

Школа Практического Программирования

λύτρα

+7(812)703-0202

Самое главное отличие статического метода класса от обычного заключается в том, что компилятор при вызове статической функции не формирует «невидимый» параметр, содержащий адрес объекта, поэтому:

- следствие 1 – в таких функциях указатель `this` не существует!!
- следствие 2 – обратиться к нестатическим данным класса в такой функции невозможно
- следствие 3 – статическая функция не может быть `virtual`

### Использование статических методов для доступа извне к private или protected static-данным класса

Если бы статическая переменная `nCount` класса `X` была защищенной, для доступа к ней извне класса потребовался бы `public` статический метод класса.

Например:

```
class X{
protected:
    static int nCount;
public:
    X() {nCount++;}
    ~X() {nCount--;}
    ...
    static int GetCount() {return nCount;}
};
```

Объявляется и инициализируется статическая переменная одинаково – независимо от спецификатора доступа:

```
int X::nCount = 0;
```

Использование статических методов:

```
int main()
{
    cout<<x::GetCount(); //???
    x x1(10);
    cout<<x1::GetCount(); //???
```

## Производящие функции классов («виртуальный конструктор»).

- Производящей функцией называется функция, которая инициализирует вызов оператора `new` для динамического создания экземпляра класса.
- Если конструктор класса защищен, то компилятор «не позволяет» создать объект такого класса традиционным способом:

```
class A{
    int m_a;
public:
    A(int a) {m_a=a;} //private конструктор

    int main()
    {
        //A a(1); //ошибка доступа (для того, чтобы компилятор имел право вызвать конструктор, он должен быть public)

        X();
    }
}
```

Закрытые конструкторы (объявленные со спецификатором `private` или `protected`) **не допускают создания объектов класса обычными пользователями**, поэтому для создания объекта требуется выполнение одного из следующих условий:

- конструктор вызывается `public` статическим методом класса;
  - конструктор вызывается методом `friend`-класса.
- Когда это нужно? – если разработчик класса делает конструктор защищенным, он тем самым заставляет пользователя класса создавать объекты только предусмотренным разработчиком способом. Например:

```
const int MAX=100; //максимально возможное количество в системе

class A{ //базовый класс, который будет контролировать в программе как все объекты класса A, так и все объекты производных классов посредством статического счетчика и статического массива, в
```

```

котором будут храниться указатели
базового типа

int m_a;

protected:
    A(int a=0) {m_a=a; if(m_count<MAX) m_ar[m_count++] = this;
    }

    virtual ~A() {m_count--;
    }

    static unsigned int m_count; //счетчик объектов как типа
    A, так и любого производного от A
    типа

    static A* m_ar[MAX]; // для упрощения считаем, что
    объектов не может быть больше MAX
public:
    static A* Create(int a){return new A(a);} //производящая
    функция

    void Destroy(){delete this;}
};

class B:public A{
    int m_b;
protected:
    B(int a, int b):A(a){m_b=b;}
public:
    static A* Create(int a, int b){return new B(a,b);}
    //статическая функция не может
    быть виртуальной, так как она не
    принимает в качестве «невидимого»
    параметра адрес объекта!!!
};

}

int main()
{
    // B b(1,2); //ошибка - конструктор защищен
    A* p = B::Create(1,2); //объект будет создан только таким
    //... образом, как предусмотрел
    // вызов посредством указателя на базовый класс
    // виртуальных методов производного
    // класса
    p->Destroy(); //вызывается метод базового класса. При
    // выполнении delete
    // сначала вызывается оператора delete
    // деструктор класса B, потом
    // деструктор класса A, потом
    // освобождается памяти
}

```

**анализ:**

котором будут храниться указатели базового типа

int m\_a;

protected:

A(int a=0) {m\_a=a; if(m\_count<MAX) m\_ar[m\_count++] = this;

}

virtual ~A() {m\_count--;

}

static unsigned int m\_count; //счетчик объектов как типа A, так и любого производного от A типа

static A\* m\_ar[MAX]; // для упрощения считаем, что объектов не может быть больше MAX

public:

static A\* Create(int a){return new A(a);} //производящая функция

void Destroy(){delete this;}

};

class B:public A{

int m\_b;

protected:

B(int a, int b):A(a){m\_b=b;}

public:

static A\* Create(int a, int b){return new B(a,b);}

//статическая функция не может быть виртуальной, так как она не принимает в качестве «невидимого» параметра адрес объекта!!!

};

## Тема VIII. Структуры и объединения C++

В C++ структуры и объединения имеют существенные отличия от структур и объединений C.

### Структура C++

Структуры C могли содержать только данные. Они предоставляли программисту удобную возможность упаковки данных. Структура C++ - это почти что полноценный класс. Это понятие осталось в C++ для совместимости с C.

У классов и структур C++ много общего:

- в структурах C++ (в отличие от структур C) могут быть функции-члены,
- в них можно объявлять секции public, protected, private,
- структуры C++ могут иметь конструкторы, деструкторы,
- участвовать в наследовании,
- иметь виртуальные функции
- ...

Однако структуры C++ от классов формально одно-единственное: спецификатор доступа по умолчанию – public (для совместимости со структурами C).

**На практике** (по договоренности между программистами) структуры используются вместо классов лишь при соблюдении следующих условий:

- структура не содержит виртуальных функций
- если участвует в наследовании, то не является производной от чего-либо, кроме разве что другой структуры, и не является базой для чего-либо, кроме разве что другой структуры (то есть не стоит наследовать структуру от класса...)
- нормальные программисты C++ используют обычно структуры для маленьких наборов данных с тривиальными функциями.

```
struct Rect{
    int left, top, right, bottom; // по умолчанию public
};
```

### Объединения C++

Объединения C++, кроме того что позволяют интерпретировать одни и ту же область памяти (то есть ее содержимое) по-разному, могут содержать также методы (в отличие от объединений C). По умолчанию – спецификатор доступа public.

```
union U {
```

```
    int m_i;
    char m_ch[4];
    U(int n) {m_i = n;}
```

};

```
int main()
```

```
{
    U ob(0x11223344);
    char ch = ob.m_ch[0];
    int n = ob.m_i;
```

}

Специфика:

- само объединение может иметь конструктор и деструктор, а его встроенные члены не могут,
- объединение не может быть производным от чего-либо
- не участвует в наследовании => в объединениях не используется спецификатор protected
- не может иметь статических членов
- размер union определяется его наибольшим данным:

```
union U {
private:
    int m_i;
    char m_ch[4];
    double m_d;
```

```

public:
    U (int x){m_i = x;}
    int GetInt() {return m_i;}
    int GetChar(int i){if(..) return m_ch[i];}
    ...
};

int main()
{
    U ob(0x11223344);
    //ob.m_i = 5; //ошибка
    int tmp = ob.GetChar(1);
    size_t n = sizeof(ob); //8
}

```

### Анонимные объединения (классы, структуры)

Иногда просто требуется сообщить компилятору, что требуется разместить несколько переменных по одному и тому же адресу => нельзя создать экземпляр анонимного объединения => обращение к членам объединения не посредством объекта, а просто по имени переменной. Для локального использования.

```

int main()
{
    {
        union{
            int i;
            char ch[4];
        };
        i=10;
        ch[3] = 0xff;
    }
    // до конца блока можно пользоваться
}

```

## Тема IX. Защищенное наследование. Отношение между классами – «подобен»

### Цель

Если спецификатор наследования производного класса `private` или `protected`, `public` интерфейс базового класса извне (посредством объекта или указателя на объект) становится недоступен:

```
class A{
    ...
public:
    void fA ();
};

class B:private/*protected*/ A{
    ...
int main()
{
    A a;
    a.fA(); //OK
    B b;
    //b.fA(); //ошибка компилятора - метод недоступен
    //A* pA = new B; //ошибка - преобразование существует, но
    // недоступно!
}

}

При защищенном наследовании доступ к public членам базового класса
возможен только из методов производного класса. Пользователи
производного класса не имеют доступа к базовой части, что позволяет
разработчику производного класса приспособить базовую часть для своих
целей, то есть:
• использовать те методы базового класса, которые удобно
использовать производному классу только определенным образом,
```

+7(812)703-0202

- а те понятия базового класса, которые «не справедливы» для производного, запрещены.

Например, таким образом можно построить иерархию классов:

```
class Rect {
protected:
    int l, r, t, b;
public:
    Rect(int x1,int y1,int x2,int y2){l=x1; r=x2; t=y1;
    b=y2;}
    ...
    void Inflate(int d1,int dr,int dt,int db){l-=d1; r+=dr;
    t-=dt; b+=db;}
};

class Square:protected Rect{
public:
    void Inflate(int d){Rect::Inflate(d,d,d,d); // метод
    базового класса вызывается таким
    образом, чтобы квадрат
    не
    превратился в прямоугольник
    Square(int x, int y, int d):Rect(x,y,x+d,y+d){}
};

int main()
{
    Square s(1,1,10);
    s.Inflate(2); //квадрат остался квадратом
    //s.Inflate(1,2,3,4); //ошибка компилятора
    //компилятор обмануть трудно:
    //s.Rect::Inflate(1,2,3,4); //ошибка компилятора
    //Rect r = s; //ошибка компилятора
    //Rect* pr =new Square(10,10,100); //ошибка компилятора
}
```

ФПС СПбГУ

**Замечание:** при защищенному наследовании внутренние взаимоотношения между двумя классами не меняются, то есть в методах производного класса доступны **protected** понятия базового класса.

### friend

Все друзья **Square** имеют также доступ к защищенным членам **Rect**, наследники - нет

```
class Rect{
protected:
    int l,r,t,b;
public:
    void f() {
        Rect(x1,y1,x2,y2){l=x1; r=x2; t=y1;
        b=y2;}
        void Inflate(int d,int dr,int dt,int db){l-=d; r+=dr;
        t-=dt; b+=db;}
    }
};

class Square:protected Rect{
public:
    void Inflate(int d){Rect::Inflate(d,d,d,d);}
    Square(int x, int y, int d):Rect(x,y,x+d,y+d){}
    friend void GF(Square& s);
};

void GF(Square& s){
    s.f(); //OK
}
```

### Защищенное наследование при построении иерархий классов

В предыдущем примере взаимоотношения внутри классов сохранялись, но если мы продолжим строить иерархию классов, то защищенная базовая часть должна

быть изолирована от всех остальных не непосредственных наследников, поэтому для таких наследников появляются дополнительные области видимости:

```
class A{
    int m_a;
protected:
    int m_a1;
    void fA1();
public:
    void fA2();
};
```

```
class B:private /*protected*/ A{
    int m_b;
protected:
    int m_B1;
    void fB1() //разрешено обращаться к public и protected
               //членам A
    fA1();
    m_a1++;
    fA2();
};

public:
    void fB2();
};
```

Разница в спецификаторах **private** и **protected** проявляется для наследников класса B:

```
class C:public B{
public:
```

```

void fC() {
    fB1();
    m_a1++; // protected - OK, private - inaccessible
    member
    fA1(); // protected - OK, private - inaccessible
    member

    fA2(); // protected - OK, private - not accessible
    because 'B' uses 'private' to
    inherit from 'A'
}

int main()
{
    A a;
    // c. fA2() ; // private - inaccessible member , protected - not
    // accessible
    c. fB2();

    // A* pA = new C; // ошибка
    B* pA = new C; // OK
}

```

Изменение вида доступа к элементу базового класса из производного класса в зависимости от спецификатора доступа, указываемого при объявлении производного класса

Спецификатор доступа, указываемый при объявлении производного класса	Изменение вида доступа к элементу базового класса посредством производного класса		
	в базовом public	в базовом protected	в базовом private
class B : <b>public</b> A{...};	в производном public	в производном protected	в производном недоступен
class C: public B{...};			
class B : <b>protected</b>	в производном	в производном	

## Тема X. Множественное наследование

Одиночное наследование (каким бы на самом деле сложным оно не было) предполагает наследование свойств от одного **непосредственного** предка. Множественное наследование позволяет производному классу наследовать свойства не только предков «по одной линии», но и свойства двух (трех,...) родителей. Иерархия классов при этом становится сложнее и вырождается в граф классов.

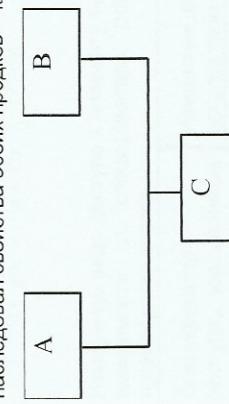
Отношение к множественному наследованию неоднозначно:

- с одной стороны, заманчиво, так как позволяет наследовать свойства нескольких предков (то есть зачастую моделирует реальные жизненные ситуации);
- с другой стороны, работает медленнее, дополнительный расход памяти, труда в реализации (так как подкидывает программисту массу «подводных камней»). Поэтому множественное наследование часто сравнивают с «яичком Пандоры», то есть с источником бед, которые не актуальны при одиночном наследовании.

Замечание: далеко не все объектно-ориентированные языки поддерживают множественное наследование.

### Простой пример (без полиморфизма)

Построим простую иерархию классов. Хотим, чтобы производный класс C наследовал свойства обоих предков – классов A и B:



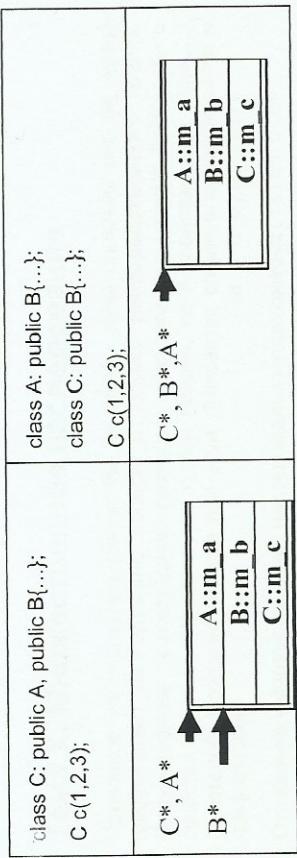
```

class A{
    int m_a = a;
    void FA() {};
public:
    A(int a){m_a = a;}
}

class B{
    int m_b = b;
    void FB() {};
public:
    B(int b){m_b = b;}
}
  
```

Специфика:

- класс C имеет конструкторы A(a), B(b), A(a), B(b), C(c). //Замечание: независимо от того порядка, в котором в списке конструкторов указывается базовых классов, компилятор их вызывает в том порядке, в котором они следуют в списке наследования;
- Порядок вызова конструкторов: A() -> B() -> C(). Конструкторы вызываются в том порядке, в котором они указаны в списке наследования!!!
- Деструкторы – в обратном порядке
- Компилятор строит объект в порядке объявления базовых классов. При этом экземпляр класса C будет в памяти выглядеть: (память отводится в порядке указания родительских классов)



- Вызов методов в таком простом случае ничем не отличается от простого наследования.

```

void main()
{
    C c(1,2,3);
    c.FA();
    ...
}
  
```

- При одиночном наследовании преобразование от derived\* к base\* происходит следующим образом: адрес остается тем же самым, а тип объекта, на который указывает указатель, изменяется. При **A\* a = &c;** // преобразование от производного к первому базовому классу – адрес не меняется. Здесь та же ситуация, что и при простом наследовании потому, что адрес подобъекта A совпадает с адресом **B\* b = &c;** //тоже получаем правильный результат, но затраты компилятора гораздо больше – адрес преобразуется к подобъекту «B» - адрес **изменяется!**

### Множественное наследование и полиморфизм

При наличии виртуальных функций ситуация усугубляется:

```

class A{
    int m_a;
public:
    virtual void f1() { }
    virtual void f2() { }
};

int main()
{
    ...
}
  
```

```

class B{
    int m_b;
public:
    virtual void f1() { }
    virtual void f2() { }
};

int main()
{
    ...
}
  
```

```

class C: public A, public B{
    int m_c;
public:
    ...
}
  
```

```
C c(4,5,6);
```

```
A* pA = &c; // адрес объекта С совпадает с адресом его
           // базовой части А => указатель на
           // vftab - первое данное с нулевым
           // смещением

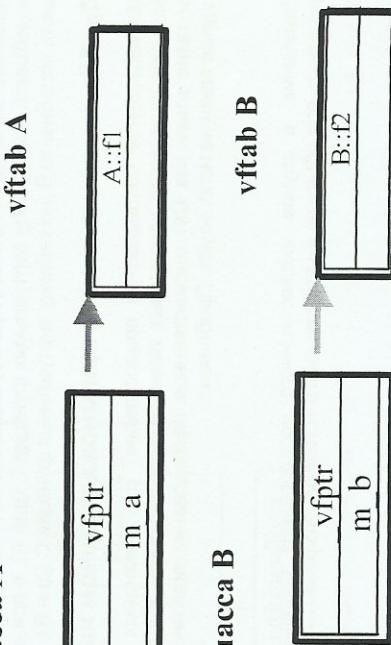
B* pB = &c; // адрес объекта С не совпадает с адресом его
           // базовой части В => но указатель на
           // vftab - это снова первое данное с
           // нулевым смещением

pA->f1(); // посредством С (для А) :: vftab

pB->f2(); // посредством С (для В) :: vftab
...
```

}

### Объект класса А



### Объект класса В

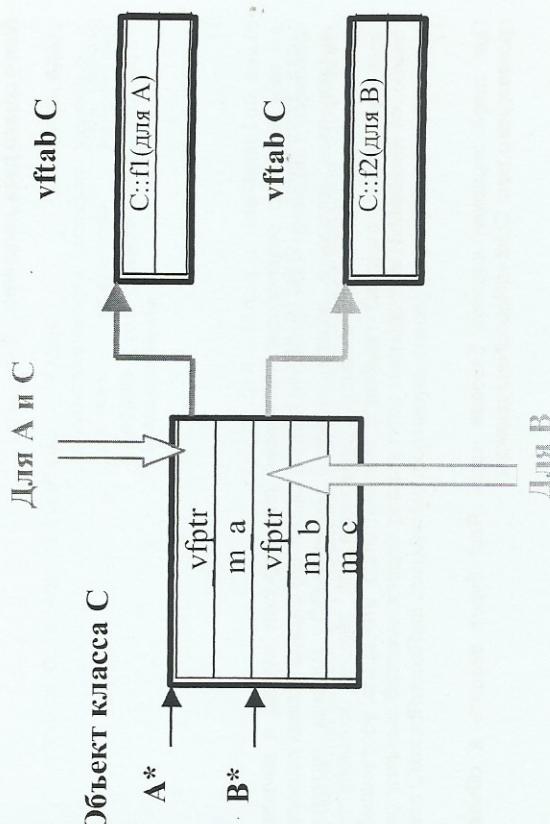
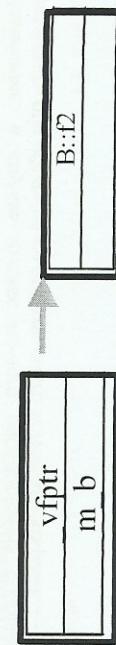


Таблица виртуальных функций две. Обе содержит адреса виртуальных функций производного класса. Поэтому при обращении и по A\*компилятор «видит» часть подобъекта A и таблицу виртуальных функций C (для A), а при обращении по B\* часть подобъекта B, но таблицу виртуальных функций C (для B).

### Проблемы, возникающие при множественном наследовании

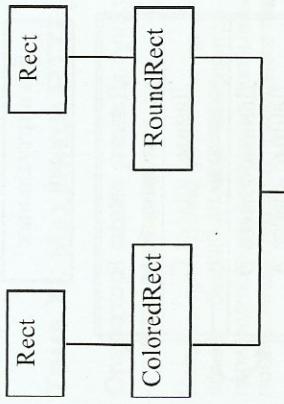
Множественное наследование предоставляет ряд очевидных преимуществ + большое количество «подводных камней», поэтому такие библиотеки как MFC (и такие языки как C#) множественное наследование практически (во всяком случае явно) не используют. Проблемы:

Замечание: в случае множественного наследования базовый класс нельзя указывать при объявлении производного более одного раза:

```
class A{...};  
class B : public A, public A{...}; // ошибка  
  
Но тем не менее, базовый класс может косвенно передаваться производному  
более одного раза, например:  
  
class Rect{...;  
protected:  
    int left, right, top, bottom;  
    ...;
```

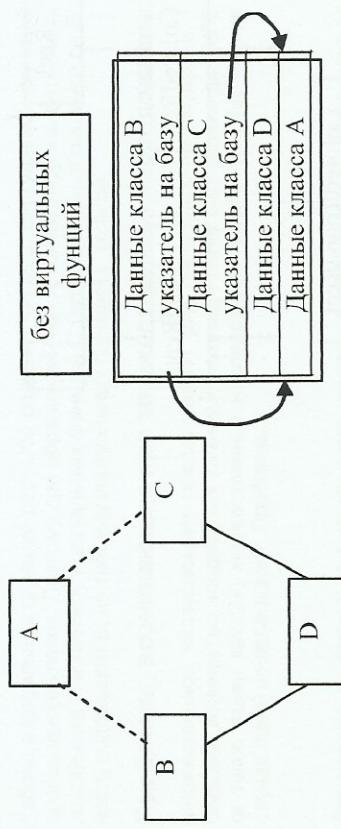
```
};  
  
class ColoredRect : public Rect{...};  
class RoundRect : public Rect {...};  
class ColoredRoundRect : public ColoredRect, public RoundRect  
{  
    int GetLeft();  
};
```

При такой иерархии классов базовая часть Rect будет входить в объект производного типа ColoredRoundRect дважды:



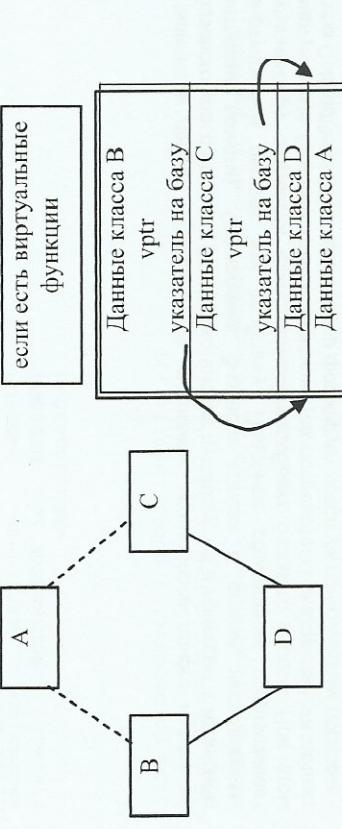
```
int ColoredRoundRect::GetLeft() {  
    //return left; //ошибка. Так как объект ColoredRoundRect  
    //подобъекта класса Rect, доступ к члену данных left неоднозначен!  
  
    return ColoredRect::value; //для устранения неоднозначности следует с помощью оператора разрешения областей видимости кнуть наследования  
}
```

Наличие в дереве наследования нескольких экземпляров одного и того же базового класса вредно (не только вносит путаницу, но и приводит к излишним затратам памяти). Объявление базового класса в списке наследования виртуальный решает эту проблему. Подобное объявление предписывает компилятору создавать единственный подобъект базового класса. При этом виртуальные базовые классы реализуются посредством указателей => объект типа ColoredRoundRect компилятор формирует стек приழдивым образом:



```
class Rect{...};
class ColoredRect : virtual public Rect{...};
class RoundRect : virtual public Rect{...};
class ColoredRoundRect : public ColoredRect, public RoundRect
{
    int GetLeft();
};
```

```
;
```



## Тема XI. Механизм RTTI (run-time type identification)

### Оператор static\_cast и указатель на классы, связанные наследованием

Неважное приведение указателя на объект производного типа к указателю на объект базового типа компилятор прекрасно выполняет сам, так как производный класс содержит полные определения своих базовых классов. Такое повышающее (upcast) приведение типа всегда безопасно!

B b;

A\* pA = &b;

Верно и следующее утверждение: если объект на самом деле является объектом производного типа, а в нашем распоряжении имеется указатель на базовый класс, то корректным должно являться и преобразование указателя "вниз" по иерархии классов вплоть до преобразования такого указателя к указателю на фактический (целевой) тип – понижающее (downcast).

Для приведения указателя на базовый тип к указателю производного типа можно использовать оператор явного приведения типа static\_cast – это механизм времени компиляции!

Пример:

```
// классы A и B связаны наследованием
class A{
    ...
};

class B:public A{
    ...
};

// C – «автономный» класс
class C{
};
```

### //Глобальная функция, принимающая указатель базового типа

```
void F(A* pA) // посредством такого указателя можно:
    - вызвать virtual-функцию целевого класса
    - вызвать обычный метод базового класса
    - оперировать данными базового класса
```

// Но если нужно вызвать обычный метод производного класса или обратиться к данным производного класса, то нужен указатель B\* :

B\* pB = static\_cast<B\*>(pA); // с точки зрения компилятора все корректно, так как классы связаны наследованием!

```
pB->fB();
```

```
int main()
{
    B b;
```

```
F(&b); // компилятор преобразует адрес B* к A* – это всегда корректно
```

// Классы A и D не связаны наследованием, поэтому компилятор не допустит такой ситуации

```
D d;
F(&d);
// // ошибка компилятора – такого преобразования нет!
```

```
// Но!
```

```
A a;
F(&a);
// ошибки компилятора нет, но результат некорректный, так как на самом деле объект A, а не B!
```

Замечание: компилятор проверяет только тот факт, что

<http://www.avalon.ru>

ФПС СПбГУ

- оба класса связаны наследованием;
- наследование открытое (public).

Поэтому компилятор считает такое приведение корректным. А на самом деле такое преобразование небезопасно! Поэтому возникла **необходимость в способе проверки возможности такого преобразования.**

Замечание: static\_cast обычно используется для неполиморфных типов (в классах нет виртуальных функций).

## Динамическая идентификация типа

```
///Но если нужно вызвать обычный метод производного класса или
обратиться к данным производного класса, то нужен указатель B* :
```

```
B* pB = static_cast<B*>(pA); // с точки зрения компилятора
все корректно, так как классы
связаны наследованием!
```

```
pB->fB();
```

```
int main()
{
    B b;
```

```
F(&b); // компилятор преобразует адрес B* к A* – это всегда
корректно
```

// Классы A и D не связаны наследованием, поэтому компилятор не допустит такой ситуации

```
D d;
F(&d);
// // ошибка компилятора – такого преобразования нет!
```

```
// Но!
```

```
A a;
F(&a);
// ошибки компилятора нет, но результат некорректный, так как на самом деле объект A, а не B!
```

Замечание: компилятор проверяет только тот факт, что

Для подключения RTTI (так как механизм дорогой и просто так работать не будет):

- в опциях проекта ProjectProperties(C/C++|Language – должен быть включен флаг Enable Run-Time Type Info (тогда в опциях командной строки компилятору будет указан ключ /GR)).
- #include <typeinfo> - объявлен класс type\_info

+7(812)703-0202

Школа Практического Программирования