Основы ООП

Наследование и полиморфизм

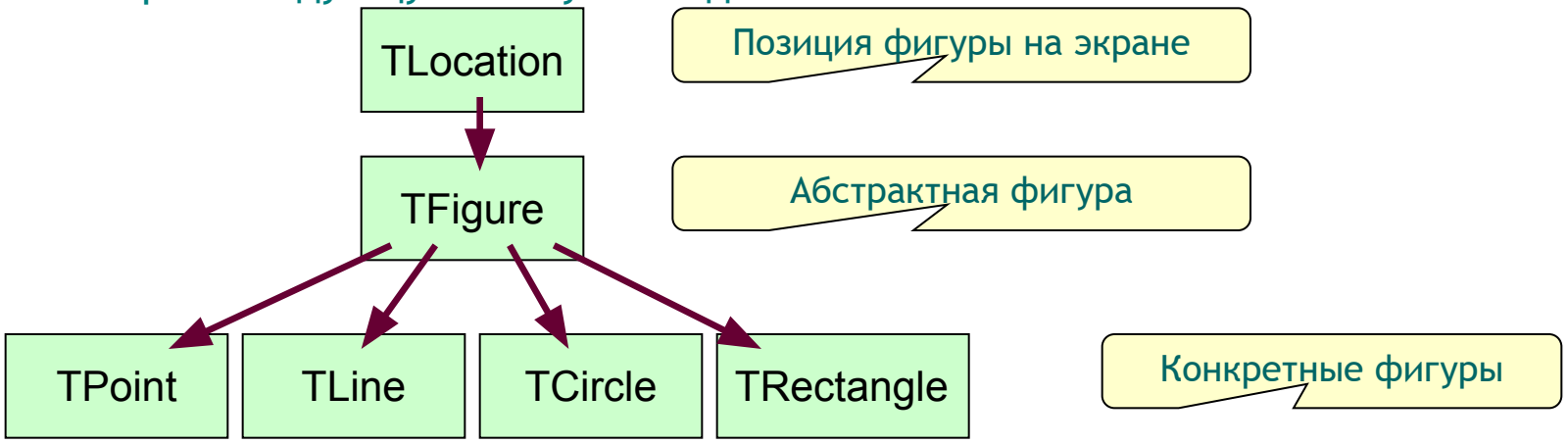
Наследование

Наследованием называется процесс порождения одного класса от другого класса с сохранением и/или перекрытием свойств и методов класса-предка и добавлением, при необходимости, новых свойств и методов в классы-потомки.

Набор классов, связанных отношением наследования, называют иерархией классов или схемой наследования.

Пример наследования. Фигуры на экране.

Рассмотрим следующую схему наследования:



Общим свойством всех фигур на экране, независимо от их формы, является наличие координат (класс TLocation).

Любая фигура на экране должна “уметь” себя отобразить, скрыть своё изображение, а также перемещаться по экрану (класс TFigure).

Наследование и полиморфизм

```
enum bool { false, true };
class TLocation {
protected:
     int X, Y;       // координаты фигуры на экране
public:
     TLocation ( int a, int b ) { X = a; Y = b; }      // конструктор
     int GetX ( ) { return X; }                        // получение координаты X
     int GetY ( ) { return Y; }                        // получение координаты Y
};
```

```
class TFigure : public TLocation {    // абстрактная фигура
protected:
     bool Vision;        // признак видимости фигуры на экране
public:
     TFigure ( int a, int b ) : TLocation ( a, b ) { }// конструктор
     void Show ( ) { }          // метод будет перекрыт у потомков класса
     void Hide ( ) { }          // метод будет перекрыт у потомков класса
     bool IsVisible ( ) { return Vision; }       // получение признака видимости
     void MoveTo ( int newX, int newY ) {  // перемещение фигуры по экрану
        bool V = IsVisible ( );       // признак видимости фигуры
        if ( V ) Hide ( );         // если фигура видима – скроем её
        X = newX; Y = newY;        // изменим координаты фигуры
        if ( V ) Show ( ); }              // если фигура была видима –
};                              // покажем её на новом месте
```

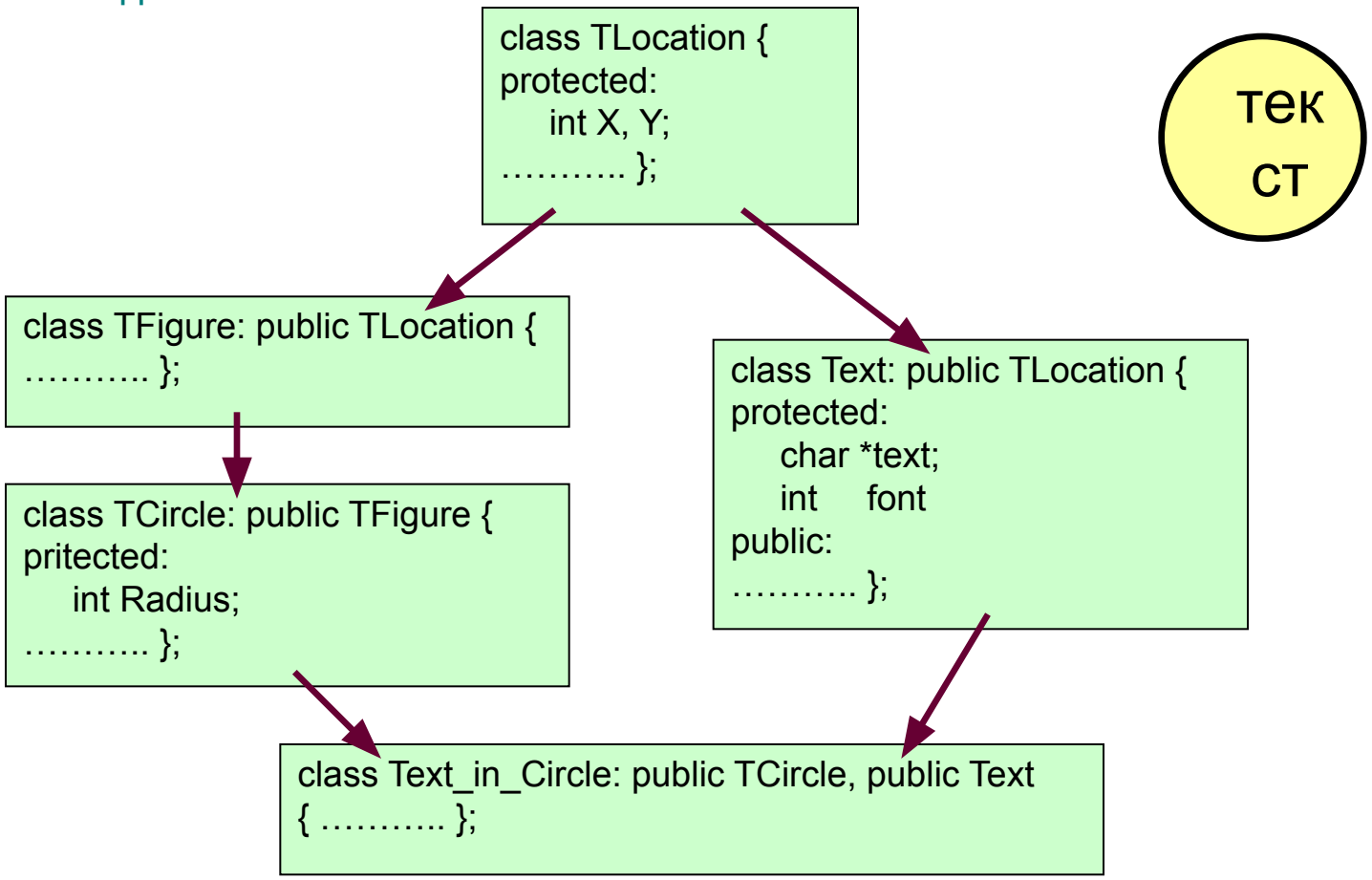
Наследование и полиморфизм

```
class TCircle : public TFigure {        // конкретная фигура - окружность
    int R;                              // радиус окружности
public:
    TCircle ( int a, int b, int r ) : TFigure ( a, b ) { R = r; }   // конструктор
    void Show ( ) {                     // перекрываем метод Show
      // используем графические примитивы из GDI+ для Win32
      // hdc - дескриптор контекста графического устройства
      // hdc всегда известен для окна, в котором будем рисовать
      Ellipse( hdc, X-R, Y-R, X+R, Y+R ); // рисуем окружность
      Vision = true; }          // фигура видна
    void Hide ( ) {                     // перекрываем метод Hide
      // сохраним цвет текущего пера (считаем, что фон полотна - белый)
      HGDIOBJ pen = SelectObject(hdc,GetStockObject(DC_PEN));
      SetDCPenColor(hdc, RGB(0,0,0) );  // установим белый цвет пера
      Ellipse( hdc, X-R, Y-R, X+R, Y+R );  // отобразим круг цветом фона
      SelectObject(hdc, pen );  // восстановим цвет текущего пера
      Vision = false; }                 // фигура не видна
};
```

Обратите внимание на то, что метод MoveTo, разработанный для абстрактной фигуры TFigure, теперь будет работать для любой конкретной фигуры.

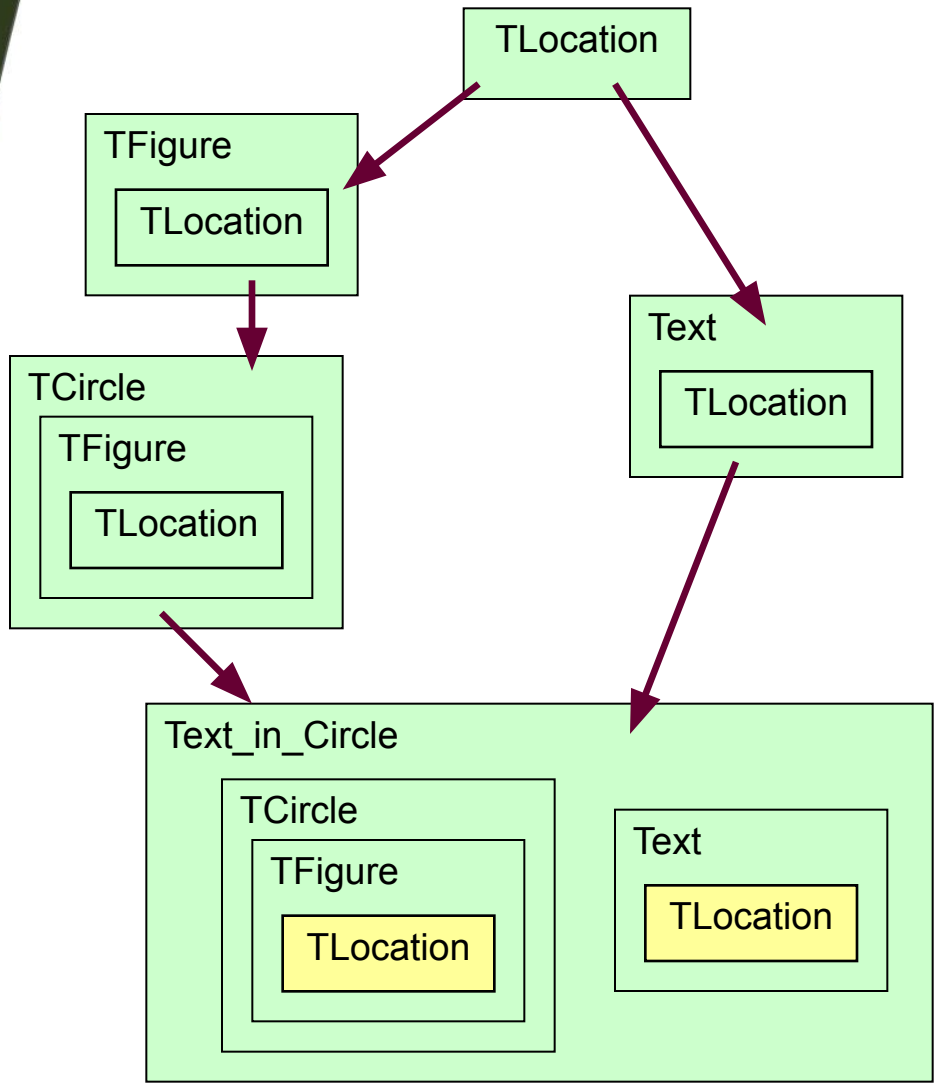
Множественное наследование

Множественное наследование означает, что у классов-потомков может быть более, чем один непосредственный класс-предок. В C++ такое допускается. Рассмотрим некоторый визуальный объект, который представляет собой текстовую строку, заключенную в окружность. Построим схему наследования:



Наследование и полиморфизм

Поскольку классы-потомки наследуют все данные и методы классов-предков, в итоге имеем следующую картину:



В результате такой схемы наследования, все данные и методы класса TLocation будут представлены в итоговом классе Text_in_Circle в двух экземплярах.

Возникает неоднозначность: каким экземпляром данных пользоваться?

Данная неприятная ситуация преодолевается в C++ путем применения виртуальной схемы наследования (о ней позже).

Виртуальные функции

Переопределим ранее введенные классы TFigure и TCircle следующим образом:

```
class TFigure : public TLocation {    // абстрактная фигура
protected:
        bool Vision;              // признак видимости фигуры на экране
public:
         TFigure ( int, int ) ;          // конструктор
virtual       ~TFigure ( ) ;          // виртуальный деструктор
virtual void Show ( ) ;               // виртуальная функция
virtual void Hide ( ) ;               // виртуальная функция
          bool IsVisible ( ) ;         // получение признака видимости
          void MoveTo ( int, int ) ;// перемещение фигуры по экрану
};
```

```
class TCircle : public TFigure {       // конкретная фигура - окружность
     int R;                        // радиус окружности
public:
        TCircle ( int a, int b, int r ) : TFigure ( a, b ) { R = r; }   // конструктор
virtual      ~TCircle ( ) ;     // виртуальный деструктор
virtual void Show ( ) { ... }       // перекрываем виртуальную функцию Show
virtual void Hide ( )  { ... }// перекрываем виртуальную функцию Hide
};
```

Наследование и полиморфизм

В чём суть использования виртуальных функций? Важно это записать и понять.

1. Если в сигнатуре функции-члена класса слово «virtual» не использовалось, тогда имя функции и исполняемый код функции связываются друг с другом на этапе компиляции. Связывание выполняет компилятор. Данный механизм называется ранним связыванием. Методы потомков перекрывают одноименные методы предков на этапе компиляции кода.
2. Если в сигнатуре функции-члена класса использовалось слово «virtual», тогда имя функции и её исполняемый код связываются друг с другом непосредственно на этапе исполнения кода. Связывание выполняет конструктор объекта. Данный механизм называется поздним связыванием. Методы потомков перекрывают одноименные методы предков только на этапе исполнения кода.

Когда работает механизм позднего связывания, компилятор отыскивает в исходном коде все виртуальные функции и формирует таблицу виртуальных функций, которую упрощенно можно представлять себе так:

< имя-функции >	< адрес-кода >
…….	…….
…….	…….
…….	…….

Адреса исполняемого кода в таблицу заносят конструкторы объектов по мере того, как эти объекты создаются. Это и есть полиморфизм. Обеспечивается динамичность и гибкость кода, без его повторной компиляции.

Абстрактные классы

Абстрактные классы используются для порождения производных классов.
«Нет ничего практичнее хорошей абстракции» (перефразируя Р.Кирхгофа).
Переопределим абстрактную фигуру ещё раз следующим образом:

```
class TFigure : public TLocation {    // абстрактная фигура
protected:
        bool Vision;              // признак видимости фигуры на экране
public:
         TFigure ( int, int ) ;         // конструктор
virtual      ~TFigure ( ) ;             // виртуальный деструктор
virtual void Show ( ) = 0 ;             // чистая виртуальная функция
virtual void Hide ( ) = 0 ;             // чистая виртуальная функция
          bool IsVisible ( ) ;          // получение признака видимости
          void MoveTo ( int, int ) ;  };    // перемещение фигуры по экрану
```

Класс называется абстрактным, если он содержит хотя бы одну чистую виртуальную функцию. Создание объектов абстрактных классов бессмысленно, т.к. такие объекты будут нежизнеспособны. Чистые виртуальные функции должны быть перекрыты в классах-потомках в обязательном порядке.

Таким образом, абстрактные классы как бы «вынуждают» своих потомков реализовывать определенные минимальные модели поведения, обеспечивая надлежащую функциональность производных классов.

Виртуальная схема наследования

Виртуальная схема наследования позволяет избегать неоднозначности в процессе множественного наследования.

Это достигается за счёт применения механизма позднего связывания и использования ссылочных типов.

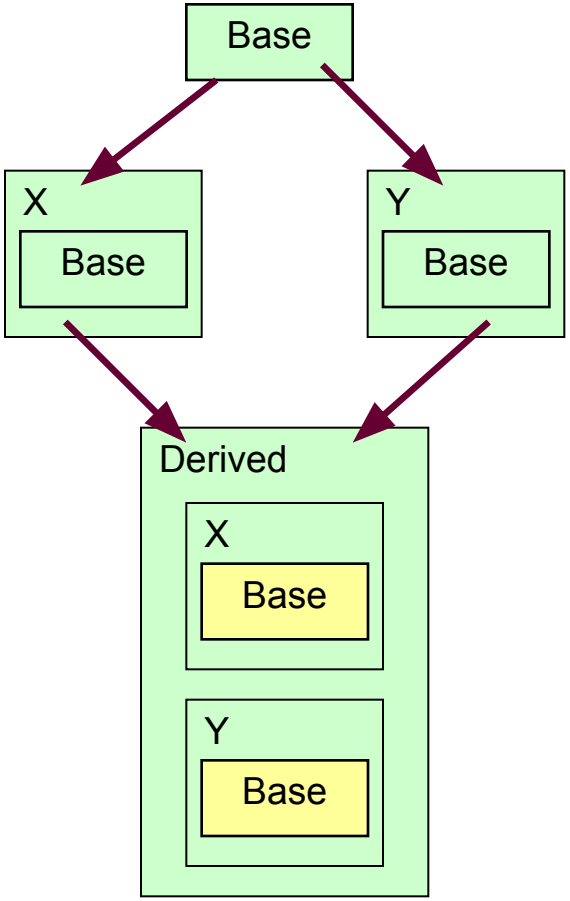
При этом, вся внутренняя работа выполняется компилятором и конструкторами объектов. Разработчику кода необходимо лишь указать ключевое слово «virtual» при описании процесса наследования.

Пример. Пусть есть такие описания:

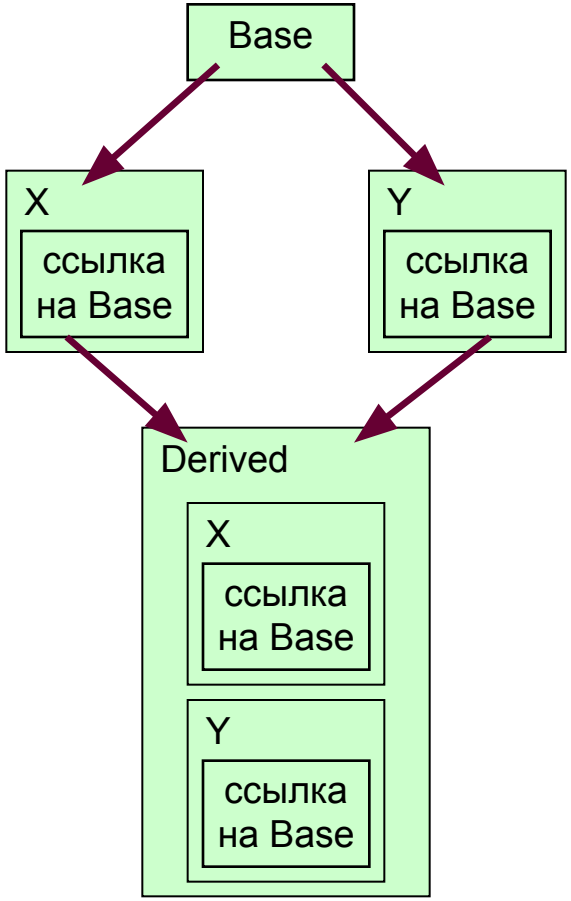
```
// Вариант 1 - используем обычную схему множественного наследования
class Base { ….. };
class Derived : public Base, public Base { ….. };     // так нельзя (синтаксис!)
class X : public Base { ….. };
class Y : public Base { ….. };
class Derived : X, Y { ….. };     // так можно, в Derived попадёт две копии Base

// Вариант 2 - используем виртуальную схему множественного наследования
class X : virtual public Base { ….. };
class Y : virtual public Base { ….. };
class Derived : X, Y { ….. };     // так можно, причём в Derived
                    // избежим неоднозначности с Base
```

Наследование и полиморфизм

Вариант 1 – обычная схема множественного наследования



Вариант 2 – виртуальная схема множественного наследования



Ссылки на объекты классов

Предположим, в нашем распоряжении имеется некая функция

```
int GetDelta ( int&, int& );
```

которая отслеживает движения мыши в каждый малый интервал времени по координатам X и Y (соответственно первый и второй аргументы функции). Функция возвращает «истину», если движение мыши имело место, и «ложь», если мышь оставалась неподвижной. Рассмотрим следующий фрагмент кода:

```
void Drag ( TFigure &Any )  {
    int X, dX, Y, dY;                  // координаты и их изменение
    Any.Show();                 // отрисовать фигуру
    X = Any.GetX();                    // получить текущую координату X
    Y = Any.GetY();                    // получить текущую координату Y
    while ( GetDelta ( dX, dY ) )       // пока движется мышь
      {  X =+ dX;  Y += dY;  Any.MoveTo ( X, Y );  }  // движется и фигура
}
```

Обратите внимание: функция Drag будет вслед за мышью перемещать по экрану любую фигуру, ничего не зная о форме этой конкретной фигуры. Важно лишь, чтобы эта фигура была потомком абстрактной фигуры TFigure.

Такое поведение возможно, потому что все фигуры-потомки TFigure способны выдавать свои координаты, а также себя изображать, стирать или передвигать.

Конструкторы

Конструктором называется функция-член класса, которая всегда вызывается при создании объектов класса. Назначение конструктора – привести объект в «рабочее» состояние. Например: получить необходимые ресурсы, открыть файлы, инициализировать данные, осуществить позднее связывание.

Имя конструктора должно совпадать с именем класса. Конструкторы не имеют возвращаемых значений. Конструкторы могут быть перегружены, как любая другая функция. Конструкторы не могут быть виртуальными функциями.

Существует три вида конструкторов: «нуль-конструктор (или конструктор по умолчанию)», «конструктор копирования» и конструктор общего вида.

Нуль-конструктор аргументов не имеет. Если в описании класса не было указано ни одного конструктора, нуль-конструктор создаётся автоматически.

Конструктор копирования имеет ровно один аргумент, который должен быть ссылкой на объект того же класса, что и создаваемый объект.

Все прочие конструкторы относятся к конструкторам общего вида.

Конструктор объекта явно или неявно вызывается всякий раз, когда объект создается, независимо от класса памяти объекта.

Конструкторы глобальных (внешних) объектов вызываются до того, как получит управление функция main(). Конструкторы объектов в динамической памяти вызываются операцией new. Конструкторы локальных объектов вызываются, когда становится активной область действия соответствующего объекта.

Для объектов классов, которые связаны отношениями наследования, порядок вызова конструкторов регулируется специальными правилами (см. далее).

Наследование и полиморфизм

Пример 1. Три вида конструкторов

```
class X  {        // виды конструкторов
.................
public:
    X ( ) ;         // 0-конструктор (конструктор по умолчанию)
    X ( const X& ) ;   // конструктор копирования
    X ( int n = 0 ) ;     // конструктор общего вида
};
void main ( void ) {
    X one;         // вызывается 0-конструктор
    X two ( 1 );  // вызывается конструктор общего вида
    X three = 1; // вызывается конструктор общего вида
    X four = one;       // вызывается конструктор копирования
    X five ( two ); }    // вызывается конструктор копирования
```

Выбор конкретного кода конструктора при создании объекта класса производится аналогично тому, как выбирается конкретный код при вызове перегруженных функций.

Основной принцип – сигнатура конструктора (т.е. количество и типы аргументов конструктора) должна соответствовать количеству и типам аргументов при вызове конструктора.

Если в классе нет ни одного конструктора, тогда автоматически создаётся и вызывается 0-конструктор («конструктор по умолчанию»).

Наследование и полиморфизм

Пример 2. Вызов конструкторов при наследовании

```
class Derived : Base1, virtual Base2  { ….. };  // два базовых класса
Derived X;       // порядок вызова: Base2(); Base1(); Derived()

class Table  { ….. }   // некоторый класс
class A {
     Table B;       // вложенный объект
     Table C;       // вложенный объект
     int n;         // элемент данных
     …..
public:
     A ( int size ) ;       // конструктор
    ~A ( ) ; } ;     // деструктор
A :: A ( int size ) : B ( size ), C ( size )  { n = size; }   // список инициализации
```

Правила вызова конструкторов следующие:

1. Сначала вызываются конструкторы базовых классов, затем вызываются конструкторы производных классов (сначала создаются объекты-предки, затем объекты-потомки).
2. Конструкторы виртуальных классов (виртуальная схема наследования) запускаются ранее конструкторов прочих классов.
3. Для вложенных объектов и производных классов могут использоваться списки инициализации.

Деструкторы

Деструктором называется функция-член класса, которая всегда вызывается при завершении существования объектов класса. Назначение деструктора – освободить полученные ресурсы, закрыть файлы, др. Имя деструктора должно начинаться со значка “~” (тильда) и в остальном совпадать с именем класса.

Деструкторы не имеют аргументов и не имеют возвращаемых значений. Деструкторы вызываются неявно и они не могут быть перегружены. Деструкторы могут быть виртуальными функциями. Деструкторы объектов вызываются в порядке, обратном порядку вызова конструкторов объектов. Деструкторы глобальных объектов вызываются после завершения работы функции main ( ).

```
class A1  { ..................public: virtual  ~A1 ( ) ;  } ;       // класс A1
class A2 : public A1 { .....        public: virtual  ~A2 ( ) ;  } ;       // класс A2
class A3 : public A2 { .....        public: virtual  ~A3 ( ) ;  } ;       // класс A3
// создадим три объекта классов и посмотрим на работу их деструкторов
void main ( void ) {
A1 *p [ 3 ] ;      // массив указателей на объекты классов
p [ 0 ] = new A1;      // создадим объект класса A1
p [ 1 ] = new A2;      // создадим объект класса A2
p [ 2 ] = new A3;      // создадим объект класса A3
delete p[1];      //  порядок вызова деструкторов:   ~A2();  ~A1();
delete p[2]; } ;  //  порядок вызова деструкторов:   ~A3();  ~A2();  ~A1();
```