

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Наследование (продолжение), шаблоны функций

ВОПРОСЫ

- ◎ Чем отличается `public` наследование от `private` наследования?
- ◎ Что такое абстрактный класс?
- ◎ Каким образом реализуется механизм виртуальных функций?
- ◎ Какие особенности необходимо учитывать при работе деструкторов в процессе наследования?
- ◎ Сформируйте иерархию классов, содержащую понятия «прямоугольник» и «квадрат»

```
class Rectangle {  
public:  
    virtual void setHeight(int newHeight);  
    virtual void setWidth(int newWidth);  
    virtual int height() const;        // возвращают текущие значения  
    virtual int width() const;  
...  
};  
void makeBigger(Rectangle& r)          // функция увеличивает площадь r  
{  
    int oldHeight = r.height();  
    r.setWidth(r.width() + 10);        // увеличить ширину r на 10  
    assert(r.height() == oldHeight); // убедиться, что высота r  
}                                      // не изменилась
```

```
class Square: public Rectangle {...};  
Square s;  
...  
assert(s.width() == s.height()); // должно быть справедливо для  
// всех квадратов  
makeBigger(s); // из-за наследования, s является  
// Rectangle, поэтому мы можем  
// увеличить его площадь  
assert(s.width() == s.height()); // По-прежнему должно быть справедливо  
// для всех квадратов
```

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

- ◎ Множественное наследование означает, что класс имеет несколько базовых классов.
- ◎ При этом, если в базовых классах есть одноименные элементы, может произойти конфликт идентификаторов, который устраняется с помощью операции доступа к области видимости.

ПРИМЕР РЕАЛИЗАЦИИ МНОЖЕСТВЕННОГО НАСЛЕДОВАНИЯ

7

```
class A
{public:
    void doSmth(){ /*...*/ }
};

class B
{public:
    void doSmth(){ /*...*/ }
    void doSmthElse(){ /*...*/ }
};
class C: public A, public B { /*...*/ };
```

ПРОБЛЕМЫ МНОЖЕСТВЕННОГО НАСЛЕДОВАНИЯ

- ◎ Использование в программе метода, определенного в обоих базовых классах приведет к ошибке, поскольку компилятор не в состоянии разобраться, метод какого из базовых классов требуется вызвать.

```
C some;  
some.B::doSmth();
```

- ◎ Если у базовых классов есть общий предок, это приведет к тому, что производный от этих базовых класс унаследует два экземпляра полей предка, что чаще всего является нежелательным.

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

```
class Animal
{ public:
    virtual void eat(); // Метод определяется для данного класса
};

class Mammal : public Animal
{ public:
    virtual Color getHairColor();
};

class WingedAnimal : public Animal
{ public:
    virtual void flap();
};

// A bat is a winged mammal
class Bat : public Mammal, public WingedAnimal {};
// обратите внимание, что метод eat() не переопределен в Bat

Bat bat;
```

КАК ЕСТ ЛЕТУЧАЯ МЫШЬ?

- ◎ Но как летучая мышь ест? Что происходит при вызове метода `bat.eat()`?
- ◎ у каждого наследника (`WingedAnimal`, `Mammal`) метод `eat()` определен по-своему.
- ◎ «Ромбическое наследование» (Diamond Inheritance):
 - ◎ сущность `Animal` единственна по сути;
 - ◎ `Bat` – это `Mammal` и `WingedAnimal`
 - ◎ но свойство животности (`Animalness`) летучей мыши (`Bat`), оно же свойство животности млекопитающего (`Mammal`) и оно же свойство животности `WingedAnimal` – по сути это одно и то же свойство.

ПРЕДСТАВЛЕНИЕ КЛАССОВ В ПАМЯТИ

- ◎ При наследовании классы предка и наследника просто помещаются в памяти друг за другом.
- ◎ Таким образом объект класса Bat это на самом деле последовательность объектов классов (Animal, Mammal, Animal, WingedAnimal, Bat), размещенных последовательно в памяти.
- ◎ При этом Animal повторяется дважды, что и приводит к неоднозначности.

ВИРТУАЛЬНОЕ МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

- ◎ Чтобы избежать дублирования, требуется при наследовании общего предка определить его как виртуальный класс

```
class monster { ... };
class daemon: virtual public monster { ... };
class lady: virtual public monster { ... };
class god: public daemon, public lady { ... };
```

- ◎ Класс god содержит только один экземпляр полей класса monster.

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

```
class Animal
{
public:
    virtual void eat();
};

// Two classes virtually inheriting Animal:
class Mammal : public virtual Animal {
public:
    virtual Color getHairColor();
};

class WingedAnimal : public virtual Animal
{
public:
    virtual void flap();
};

// A bat is still a winged mammal
class Bat : public Mammal, public WingedAnimal {};
```

ПРЕДСТАВЛЕНИЕ КЛАССОВ В ПАМЯТИ

- ◎ Теперь, часть Animal объекта класса Bat::WingedAnimal *та же самая*, что и часть Animal, которая используется в Bat::Mammal, и можно сказать, что Bat имеет в своем представлении только одну часть Animal и вызов Bat::eat() становится однозначным.
- ◎ Виртуальное наследование реализуется через добавление указателей на виртуальную таблицу vtable в классы Mammal и WingedAnimal
- ◎ Таким образом, Bat представляется, как (vtable*, Mammal, vtable*, WingedAnimal, Bat, Animal).

ОСОБЕННОСТИ ВИРТУАЛЬНОГО НАСЛЕДОВАНИЯ

- ➊ Размер объектов классов, использующих множественное виртуальное наследование, больше по сравнению со случаем, когда виртуальное наследование не используется.
- ➋ Доступ к данным-членам виртуальных базовых классов также медленнее, чем к данным невиртуальных базовых классов.
- ➌ Не применяйте виртуальных базовых классов до тех пор, пока в этом не возникнет настоятельная потребность. По умолчанию используйте невиртуальное наследование.
- ➍ Если все же избежать виртуальных базовых классов не удается, старайтесь не размещать в них данных (классы-интерфейсы)

ШАБЛОНЫ ФУНКЦИЙ

ШАБЛОНЫ ФУНКЦИЙ

- ◎ При создании функций иногда возникают ситуации, когда две функции выполняют одинаковую обработку, но работают с разными типами данных
- ◎ например, одна функция сортирует массивы типа `string`, а другая типа `float`.
- ◎ Часто используют шаблоны функций для быстрого определения нескольких функций, которые с помощью одинаковых операторов работают с параметрами разных типов или имеют разные типы возвращаемых значений.

ОПИСАНИЕ ШАБЛОНА ФУНКЦИИ

- ◎ Шаблон функции определяет типонезависимую функцию.

```
template< typename T >
void sort( T array[], int size ); // прототип: шаблон sort
// объявлен, но не определён

template< typename T >
void sort( T array[], int size ) // объявление и определение
{
    T t;
    for (int i = 0; i < size - 1; i++)
        for (int j = size - 1; j > i; j--)
            if (array[j] < array[j-1])
            {
                t = array[j];
                array[j] = array[j-1];
                array[j-1] = t;
            }
}
```

- ◎ Буква Т данном случае представляет собой общий тип шаблона.

ВЫЗОВ ШАБЛОННОЙ ФУНКЦИИ

- ◎ После определения шаблона внутри вашей программы вы объявляете прототипы функций для каждого требуемого вам типа.

```
int i[5] = { 5, 4, 3, 2, 1 };
```

```
sort< int >( i, 5 );
```

```
char c[] = "бвгда";
```

```
sort< char >( c, strlen( c ) );
```

```
// ошибка: у sort< int > параметр int[] а не char[]
```

```
sort< int >( c, 5 );
```

ВЫВЕДЕНИЕ ШАБЛОННОЙ ФУНКЦИИ

- ◎ В некоторых случаях компилятор может сам вывести (логически определить) значение параметра шаблона функции из аргумента функции.

```
int i[5] = { 5, 4, 3, 2, 1 };
sort( i, i + 5 ); // вызывается sort< int >

char c[] = "бвгда";
sort( c, c + strlen( c ) ); // вызывается sort< char >
```

ГЕНЕРАЦИЯ ШАБЛОННЫХ ФУНКЦИЙ

- ◎ Для каждого набора параметров компилятор генерирует новый экземпляр функции. Процесс создания нового экземпляра называется *инстанцированием шаблона*.
- ◎ Например, компилятор создал две специализации шаблона функции sort (для типов char и int)
- ◎ Для каждого возможного значения параметра компилятор будет создавать новые и новые экземпляры функций, которые будут отличаться лишь одной константой.

ПАРАМЕТРЫ ШАБЛОНОВ

- ◎ Параметрами шаблонов могут быть: параметры-типы, параметры обычных типов, параметры-шаблоны.
- ◎ Для параметров любого типа можно указывать значения по умолчанию.

```
template< class T1,           // параметр-тип
          typename T2,           // параметр-тип
          int I,                 // параметр обычного типа
          T1 DefaultValue,       // параметр обычного типа
          template< class > class T3, // параметр-шаблон
          class Character = char // параметр по умолчанию
          >
```

ОШИБКИ ШАБЛОНОВ

- Ошибки, связанные с использованием конкретных параметров шаблона, нельзя выявить до того, как шаблон использован.
- Например, шаблон **sort** сам по себе не содержит ошибок, однако использование его с типами, для которых операция '<' не определена, приведёт к ошибке

```
struct A
{
    int a;
};

A mas[5] = ...;
sort( mas, 5 );
```

- Если ввести операцию '<' до первого использования шаблона, то ошибка будет устранена.

Итого

- ◎ Множественное наследование обеспечивает возможность наследования от нескольких базовых классов одновременно
- ◎ Для разрешения ромбовидного наследования необходимо использовать виртуальное наследование
- ◎ Для реализации универсальных функций, обеспечивающих реализацию универсального алгоритма на нескольких типах данных необходимо использовать шаблонные функции