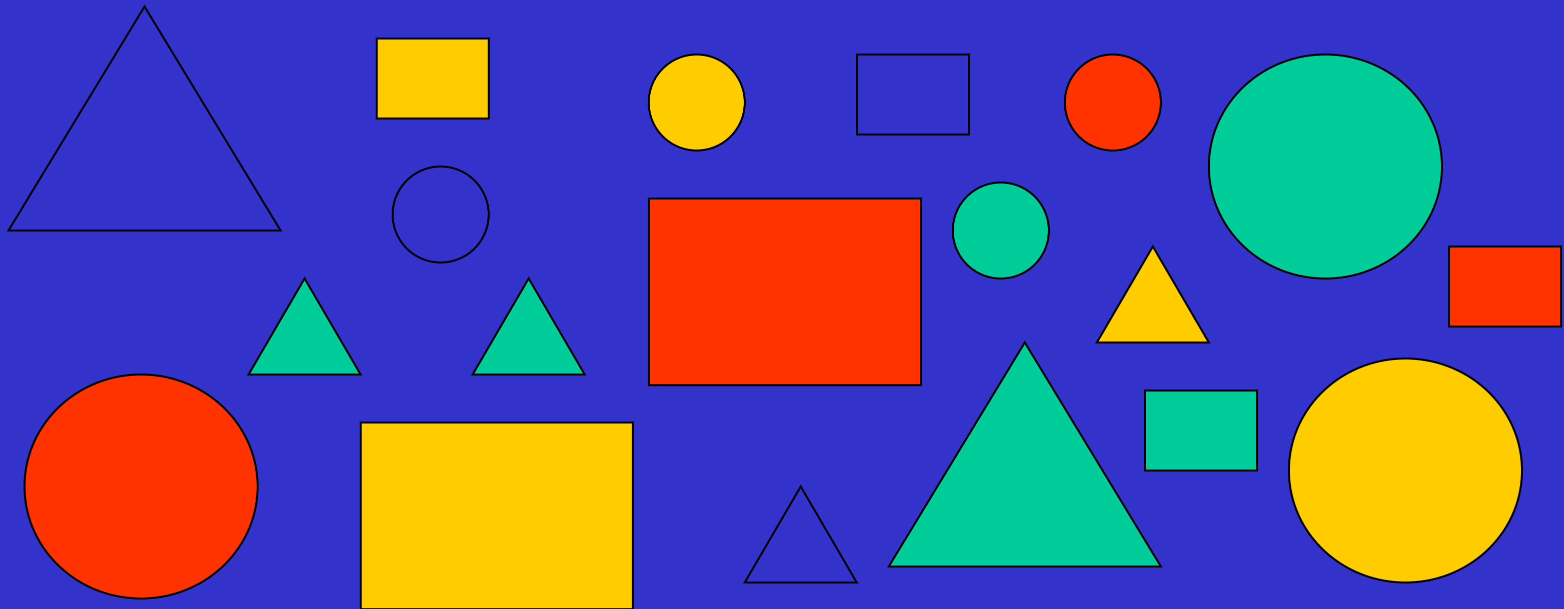
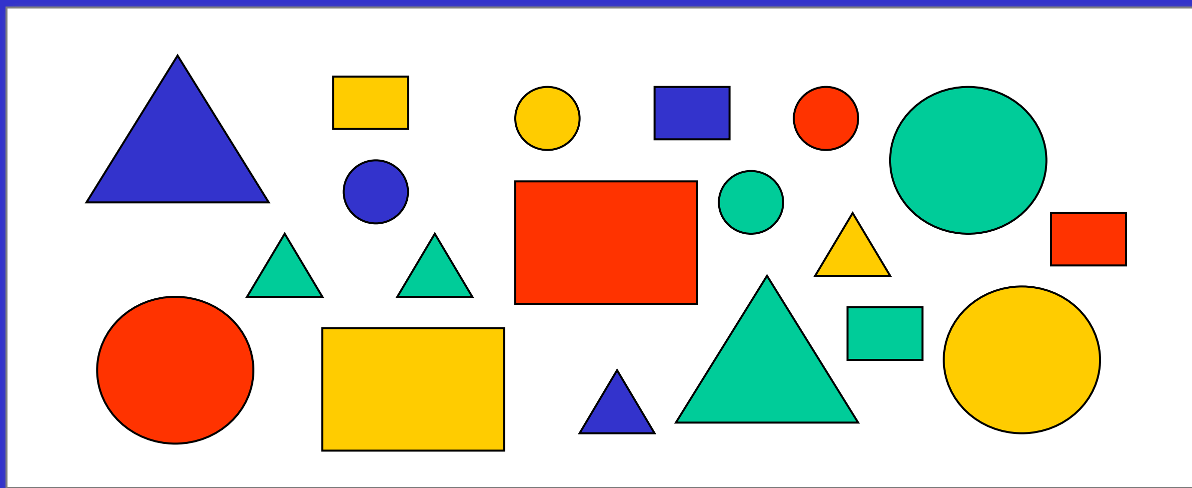
Тема 5. Объектно-ориентированное программирование. Наследование и полиморфизм

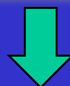
Понятие абстракции

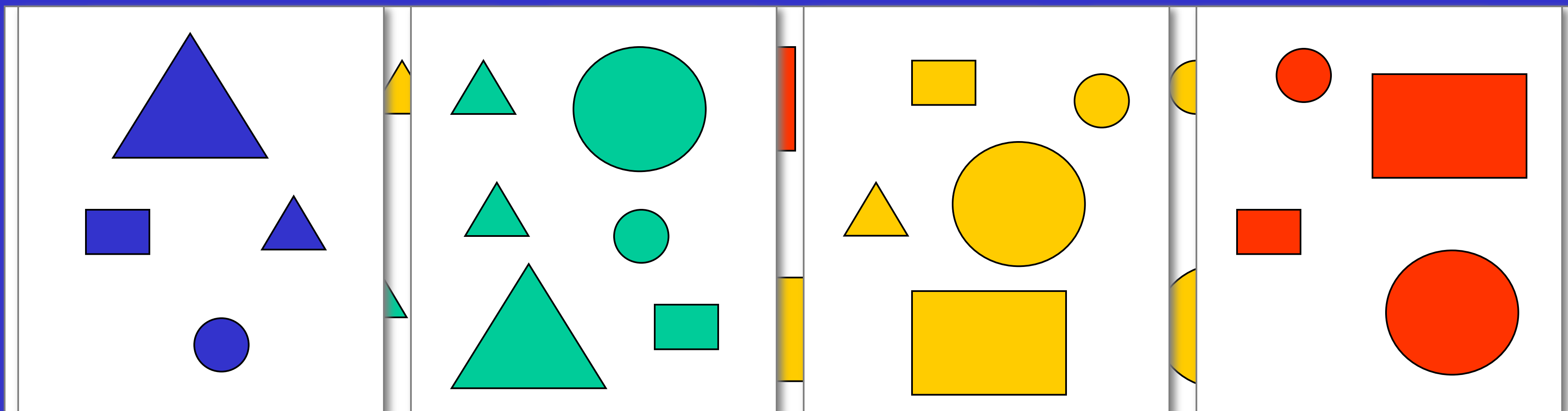


Абстракция в объектно-ориентированном программировании — это выделение существенных характеристик объектов, которые отличают их от всех других объектов, четко определяя концептуальные границы каждого класса.

Классификация объектов по различным признакам





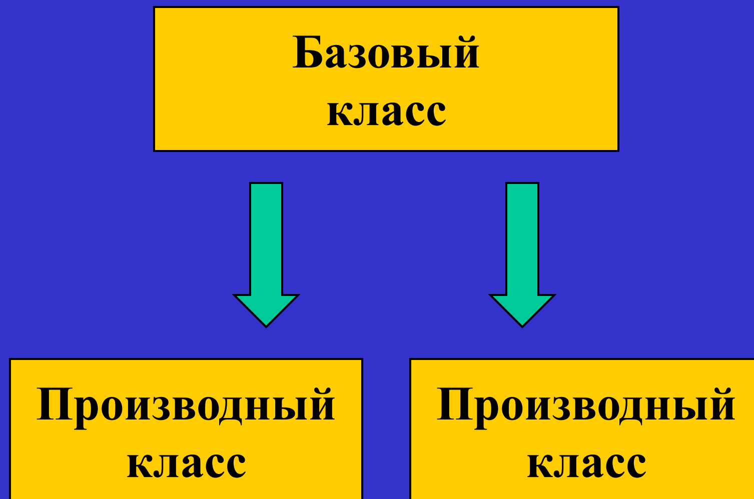


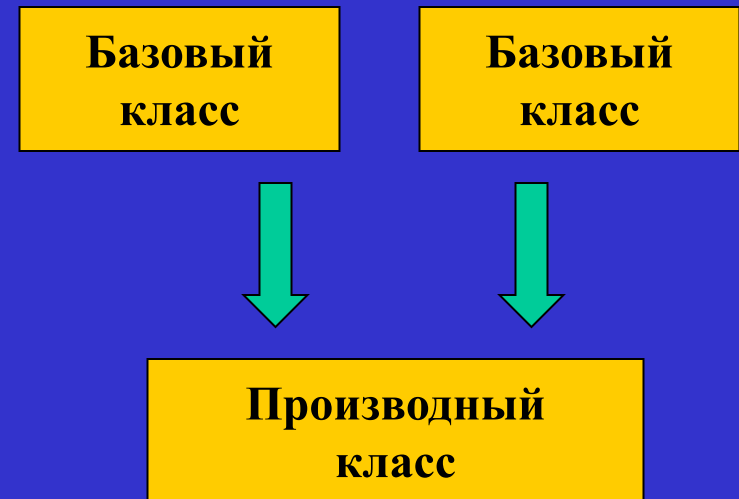
Иерархия абстрактного и конкретного

Понятие наследования

Наследование — важная составляющая концепции объектно-ориентированного программирования, благодаря которой возможно создание новых классов на базе уже существующих.
Новые классы (производные) получают все элементы наследуемых классов (базовых).

Простое и множественное наследования





При простом наследовании производный класс имеет только один базовый класс. Базовый класс может иметь несколько производных классов. Иерархия классов имеет вид дерева.

При множественном наследовании производный класс имеет несколько родительских классов. Иерархия классов имеет вид ориентированного графа.

Критерии «хорошей» иерархии классов

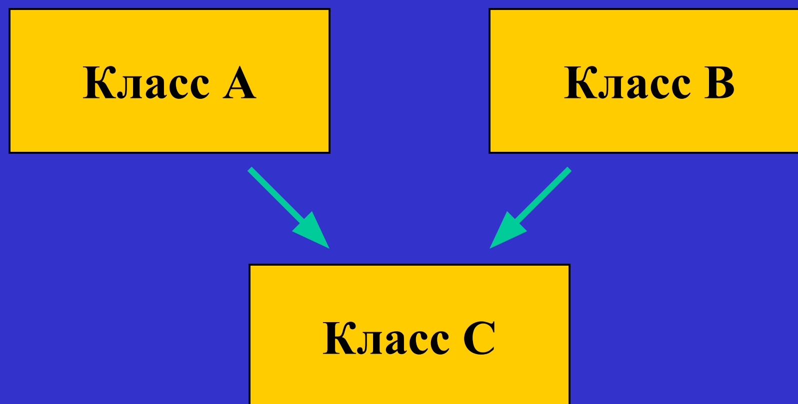
Критерий включения. Объекты производных классов должны обладать всеми свойствами базовых классов.

Критерий независимости. Наследуемые свойства не должны ограничивать использование собственных элементов производных классов.

Критерий специализации. Если производный класс приводит к специализации (ограничению) свойств базового класса, то это является нарушением принципа независимости. Классы, которые отличаются только специализацией, обычно составляют один уровень иерархии.

Критерий единства. Наследование и полиморфизм обеспечивают единообразное выполнение аналогичных функций классов. Базовые классы используются для создания интерфейса между классами и «внешним миром».

Наследование в C++




```
class A
{
    ...
};

class B
{
    ...
};
```

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

Ограничение доступа при наследовании

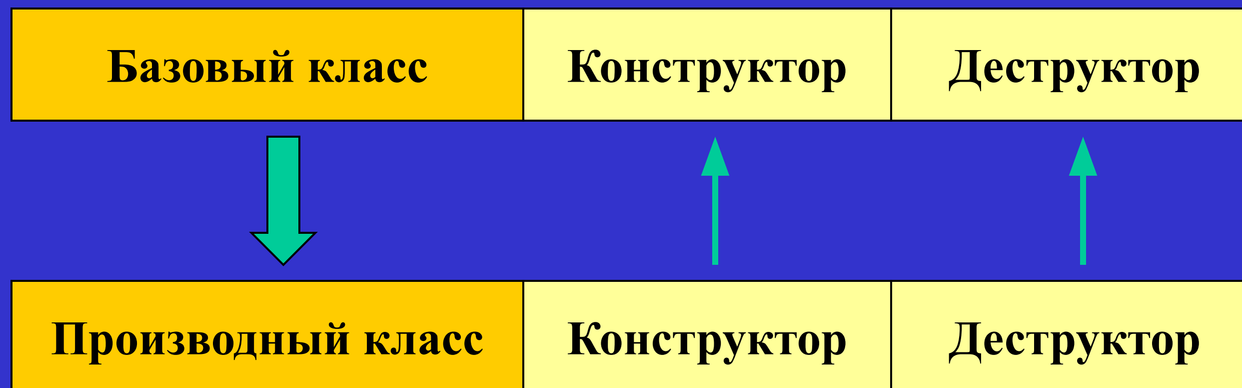
Ограничение доступа при наследовании

Атрибут доступа, указанный при наследовании	Атрибут доступа базового класса	Атрибут доступа производного класса
public	public	public
	protected	protected
	private	private
protected	public	protected
	protected	protected
	private	private
private	public	private
	protected	private
	private	private

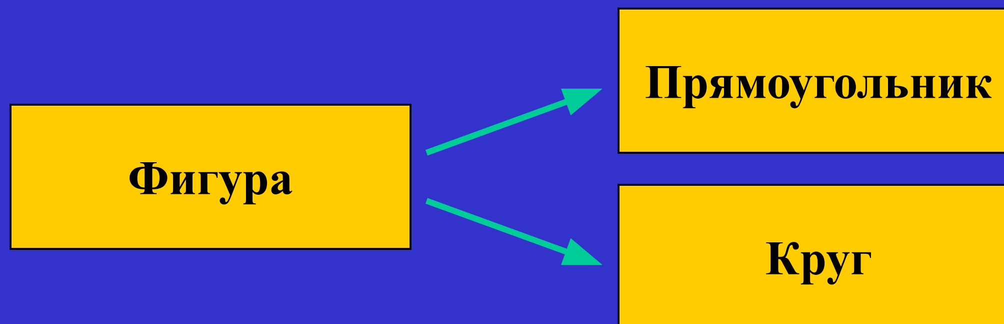
Наследование и конструкторы

Конструкторы и деструкторы не наследуются. При создании объекта производного класса наследуемые им данные должны инициироваться конструктором базового класса. Если наследуются несколько базовых классов, то их конструкторы вызываются явно конструктором производного класса или компилятор инициирует вызов конструкторов по умолчанию в порядке описания базовых классов.

Конструкторы производных классов могут передавать аргументы конструкторам базовых классов.



Пример наследования и использования конструкторов




```
class Figure
{
protected:
   TColor color;
public:
   Figure(TColor Color);
   TColor Color() { return(color); }
};

Figure::Figure(TColor Color)
{
   color = Color;
}
```

Пример наследования (продолжение)

```
class Rectangle : public Figure
{
private:
   float width;
   float height;
public:
   Rectangle(float Width, float Height, TColor Color);
   float Area() { return(width*height); }
};

Rectangle::Rectangle(float Width, float Height, TColor Color) :
           Figure(Color)
{
   width  = Width;
   height = Height;
}
```

Пример наследования (продолжение)

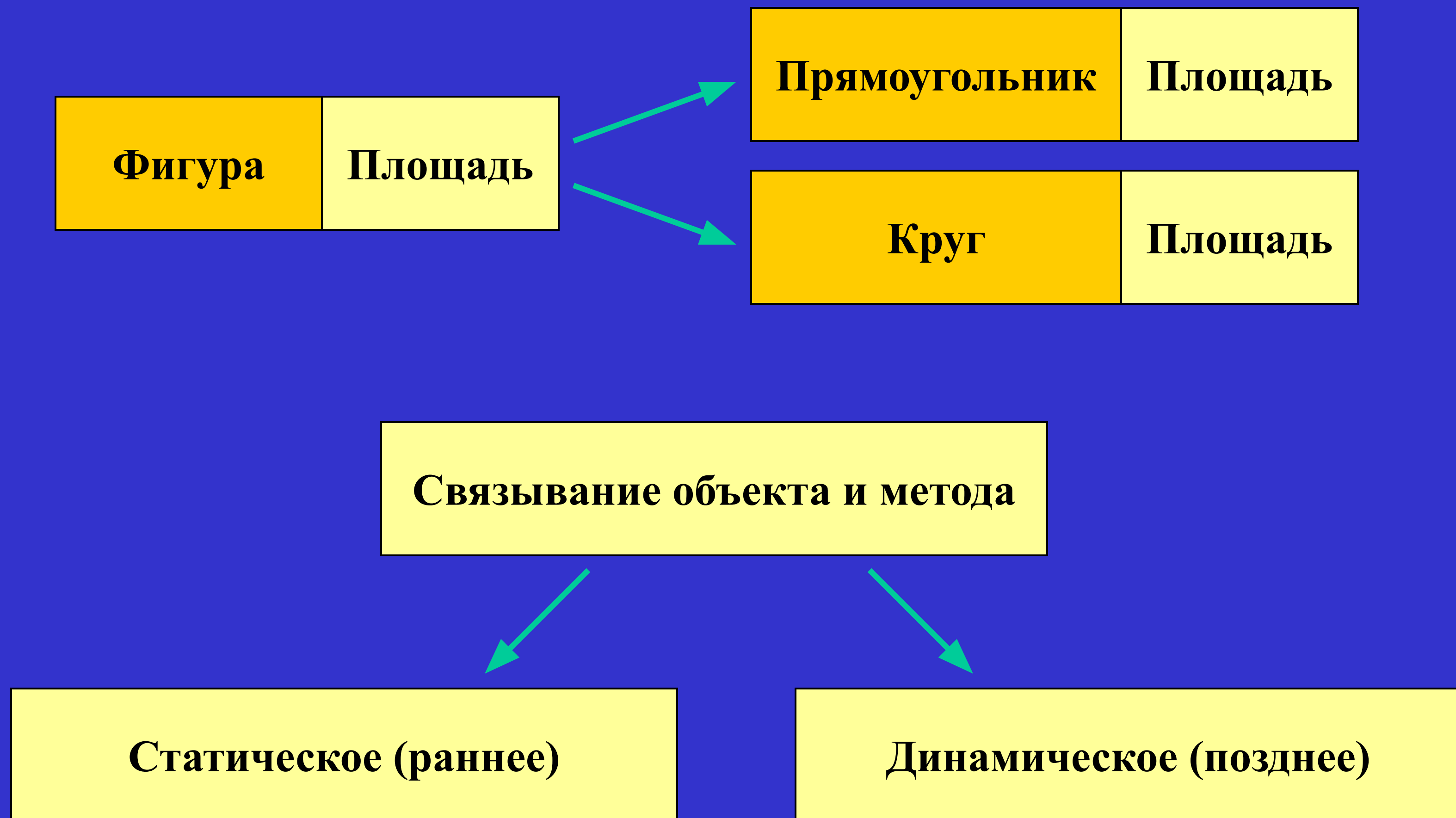
```
class Circle : public Figure
{
private:
   float radius;
public:
   Circle(float Radius, TColor Color);
   float Area() { return(M_PI*radius*radius); }
};

Circle::Circle(float Radius, TColor Color) : Figure(Color)
{
   radius = Radius;
}
```

Полиморфизм

Понятие полиморфизма тесно связано с концепцией наследования. Оно означает феномен различного выполнения одноименных функций в разных классах.

Перегрузка функций и операторов также является одной из составляющих полиморфизма.

Пример переопределения методов в С++



Пример раннего связывания

```
class Figure
{
...
public:
    float Area() { return(-1); }
};

class Rectangle : public Figure
{
...
public:
    float Area() { return(width*height); }
}

class Circle : public Figure
{
...
public:
    float Area() { return(M_PI*radius*radius); }
}
```

Пример раннего связывания

```
void main()
{
   Figure f; Rectangle r(200, 100); Circle c(150);

   ShowMessage(f.Area());          // метод класса Figure
   ShowMessage(r.Area());          // метод класса Rectangle
   ShowMessage(c.Area());          // метод класса Circle

   ShowMessage(r.Figure::Area());    // метод класса Figure
   ShowMessage(c.Figure::Area());    // метод класса Figure

   Figure *fp = &f; Figure *rp = &r; Figure *cp = &c;

   ShowMessage(fp->Area());        // метод класса Figure
   ShowMessage(rp->Area());        // метод класса Figure
   ShowMessage(cp->Area());        // метод класса Figure

   Rectangle *rp1 = &r; Circle *cp1 = &c;

   ShowMessage(rp1->Area());       // метод класса Rectangle
   ShowMessage(cp1->Area());       // метод класса Circle
}
```

Пример позднего связывания

```
class Figure
{
...
public:
    virtual float Area() { return(-1); }
};

class Rectangle : public Figure
{
...
public:
    virtual float Area() { return(width*height); }
}

class Circle : public Figure
{
...
public:
    virtual float Area() { return(M_PI*radius*radius); }
}
```

Пример позднего связывания

```
void main()
{
   Figure f; Rectangle r(200, 100); Circle c(150);

   ShowMessage(f.Area());          // метод класса Figure
   ShowMessage(r.Area());          // метод класса Rectangle
   ShowMessage(c.Area());          // метод класса Circle

   ShowMessage(r.Figure::Area());     // метод класса Figure
   ShowMessage(c.Figure::Area());     // метод класса Figure

   Figure *fp = &f; Figure *rp = &r; Figure *cp = &c;

   ShowMessage(fp->Area());        // метод класса Figure
   ShowMessage(rp->Area());        // метод класса Rectangle
   ShowMessage(cp->Area());        // метод класса Circle

   Rectangle *rp1 = &r; Circle *cp1 = &c;

   ShowMessage(rp1->Area());       // метод класса Rectangle
   ShowMessage(cp1->Area());       // метод класса Circle
}
```

Чисто виртуальные методы

```
class Figure
{
...
public:
   virtual float Area() = 0;
};

class Rectangle : public Figure
{
...
public:
   virtual float Area() { return(width*height); }
}

class Circle : public Figure
{
...
public:
   virtual float Area() { return(M_PI*radius*radius); }
}
```

Чисто виртуальные методы не имеют тела и предназначены для задания интерфейса производных классов.

Абстрактные классы

```
class Figure
{
...
public:
   virtual float Area() = 0;
};

void main()
{
   Figure f;
...
}
```

Классы, которые включают хотя бы один чисто виртуальный метод, называются абстрактными. Для абстрактных классов не могут быть созданы объекты. Абстрактные классы всегда находятся на вершине иерархии классов и предназначены для задания интерфейса и придания иерархии единообразия.

```
[C++ Error] test.cpp(50): E2352 Cannot create instance of abstract class 'Figure'
[C++ Error] test.cpp(50): E2353 Class 'Figure' is abstract because of 'Figure::Area() = 0'
```

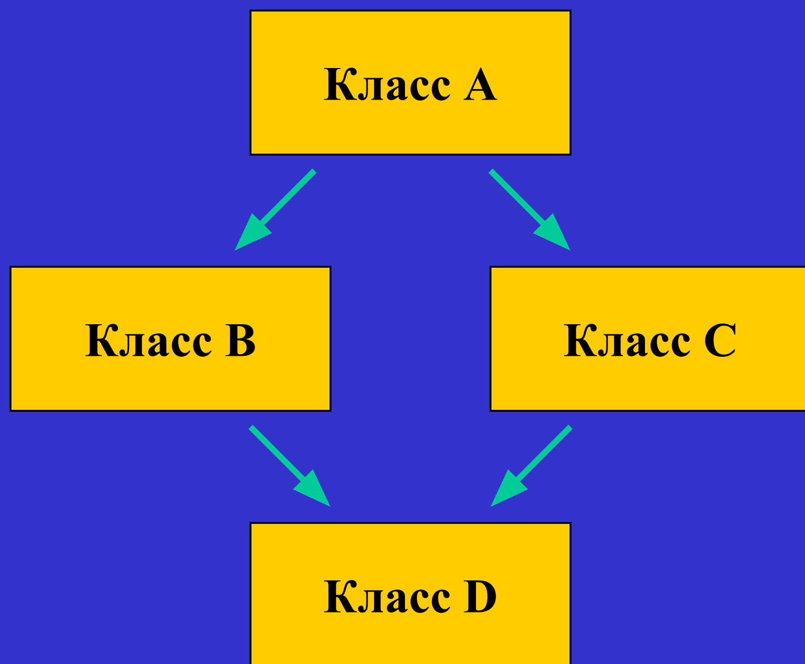
Конфликты при множественном наследовании

```
class A1
{
public:
   int    a;
   void   Func();
};

class A2
{
public:
   int    a;
   void   Func();
};

class B : public A1, public A2
{
...
};
```

```
void main()
{
   B b;
   b.a = 5;          // Ошибка
   b.Func();         // Ошибка
   b.A1::a = 6;      // Правильно
   b.A1::Func();     // Правильно
   b.A2::a = 7;      // Правильно
   b.A2::Func();     // Правильно
}
```

Для устранения конфликтов имен при множественном наследовании используется оператор расширения области видимости

Виртуальные базовые классы




```
class A
{
...
}

class B : virtual public A
{
...
}

class C : virtual public A
{
...
}

class D : public B, public C
{
...
}
```