Виртуальные функции и абстрактные классы

```
class Base
 { public:
    virtual void f1();
    virtual void f2();
    virtual void f3();
    void f();
  };

class Derived : public Base
 { public:
    void f1();
    void f2(int);
    char f3();
    void f();
  };
```

```
Derived *dp = new Derived;
Base    *bp = dp;

bp->f();          // Base::f
dp->f();          // Derived::f
dp->Base::f();      // Base::f

dp->f1();         // Derived::f1
bp->f1();         // Derived::f1!!!
bp->Base::f1();     // Base::f1

bp->f2();         // Base::f2
dp->f2(0);        // Derived::f2
dp->f2();
dp->Base::f2();     // Base::f2
```

```
class Base
 { ...
   public:
    Base();
    Base(const Base& b);
    virtual Base* Create()
      { return new Base();      }
    virtual Base* Clone()
      { return new Base(*this); }
  };
```

```
class Derived : public Base
 { ...
   public:
    Derived();
    Derived(const Derived& d);
    Derived* Create()
      { return new Derived();       }
    Derived* Clone()
      { return new Derived(*this); }
  };
```

```
void f(Base *p)
 { Base *p1 = p->Create(); }

void f2(Derived *p)
 { Derived *p2 = p->Clone(); }
```

```
class Shape
 { public:
    virtual void draw() = 0;
   ...
  };

Shape  s;
Shape *s;
Shape  f();
void   f(Shape s);
Shape& f(Shape &s);
```

```
#define SHAPES

class Shapes
 { protected:
    static int count;
    int color;
    int left, top, right, bottom;
    Shapes()  { count++; }
   public:
    enum {LEFT, UP, RIGHT, DOWN};
    virtual ~Shapes() { count--; }
    static int GetCount() { return count;  }
    int Left()   const { return left;   }
    int Top()    const { return top;    }
    int Right()  const { return right;  }
    int Bottom() const { return bottom; }
    virtual void Draw() = 0;
    virtual void Move(int where, const Shapes *shape) = 0;
  };
```

```
#include "Shapes.h"

int Shapes::count = 0;
```

```
#if !defined(SHAPES)
 #include "Shapes.h"
#endif

class Circle : public Shapes
 { private:
    int cx, cy, radius;
   public:
    Circle(int x = 0, int y = 0, int r = 0, int c = 0);
    ~Circle() { }
    void Draw();
    void Move(int where, const Shapes *shape);
  };
```

```
#include "Circle.h"

Circle::Circle(int x, int y, int r, int c)
 { cx    = x; cy = y;   radius = r;
   color = c;
   left  = cx - radius; top    = cy - radius;
   right = cx + radius; bottom = cy + radius;
  }

void Circle::Draw()
 { ... }

void Circle::Move(int where, const Shapes *shape)
 { ... }
```

```
#if !defined(SHAPES)
 #include "Shapes.h"
#endif

class Triangle : public Shapes
 { private:
    int x1, y1, x2, y2, x3, y3;
   public:
    Triangle(int x1 = 0, int y1 = 0, int x2 = 0, int y2 = 0,
             int x3 = 0, int y3 = 0, int c = 0);
    ~Triangle() { }
    void Draw();
    void Move(int where, const Shapes *shape);
  };
```

```
#include "Triangle.h"

int Max(int a, int b, int c);
int Min(int a, int b, int c);

Triangle::Triangle(int x1, int y1, int x2, int y2,
                   int x3, int y3, int c)
 { this->x1 = x1; this->y1 = y1;
   this->x2 = x2; this->y2 = y2;
   this->x3 = x3; this->y3 = y3;
   color  = c;
   left   = Min(x1, x2, x3); top    = Min(y1, y2, y3);
   right  = Max(x1, x2, x3); bottom = Max(y1, y2, y3);
  }

void Triangle::Draw()
 { ... }

void   Triangle::Move(int where, const Shapes *shape)
 { ... }
```

```
#include "Circle.h"
#include "Triangle.h"

void main()
 { Shapes* shapes[10];

   shapes[0] = new Circle(100, 100, 30, 50);
   shapes[1] = new Triangle(0, 0, 20, 0, 0, 20, 90);
   shapes[2] = new Circle(200, 200, 50, 20);

   for(int i = 0; i < Shapes::GetCount(); i++)
    shapes[i]->Draw();

   for(int i = 1; i < Shapes::GetCount(); i++)
    shapes[i]->Move(Shapes::LEFT, shapes[i - 1]);

   for(int i = 0; i < Shapes::GetCount(); i++)
    shapes[i]->Draw();

   for (int i = 0, n = Shapes::GetCount(); i < n; i++)
    delete shapes[i];
  }
```

```
template <class T> class Stack
 { private:
    enum { SIZE = 3 };
    T stack[SIZE];
    T *cur;
   public:

    class StackError
     { public:
        virtual ~StackError() { }
        virtual Stack* GetPtr() = 0;
        virtual void Print()    = 0;
      };

    class StackEmpty : public StackError
     { private:
        Stack *stack;
       public:
        StackEmpty(Stack *p) : stack(p) { }
        Stack* GetPtr() { return stack; }
        void Print();
      };
```

```
  class StackFull : public StackError
   { private:
      Stack *stack;
      T n;
     public:
      StackFull(Stack *p, T i) : stack(p), n(i) { }
      Stack* GetPtr() { return stack; }
      T GetValue()    { return n;     }
      void Print();
    };

  Stack()  { cur = stack; }
  ~Stack() { }
  T   Push(const T& n);
  T   Pop();
  int IsEmpty() { return cur == stack; }
  T   operator >> (T& s)       { s = Pop(); return s; }
  T   operator << (const T& s) { return Push(s);      }
};
```

```
#include <iostream>
#include "Stack.h"

template <class T> T Stack<T>::Push(const T& n)
 { if (cur - stack < SIZE)
    { *cur++ = n; return n; }
   else
    throw StackFull(this, n);
  }

template <class T> T Stack<T>::Pop()
 { if (cur != stack)
    return *--cur;
   else
    throw StackEmpty(this);
  }
```

```
template <class T> void Stack<T>::StackEmpty::Print()
 { std::cout << "Attempt to get a value from the empty stack
                at the address " << GetPtr() << std::endl;
  }

template <class T> void Stack<T>::StackFull::Print()
 { std::cout << "Attempt to put a value " << GetValue() <<
               " to the full stack at the address " <<
               GetPtr() << std::endl;
  }
```

```
#include "Stack.cpp"

void main()
 { Stack<int> is;
   int n;

   try
    { is << 1;
      is << 2;
      is << 3;
      is >> n;
      printf("%d\n", n);
      is >> n;
      printf("%d\n", n);
      is >> n;
      printf("%d\n", n);
      is >> n;
      printf("%d\n", n);
    }
   catch (Stack<int>::StackError& s)
    { s.Print(); }
```

```
  Stack<char> cs;
  char c;

  try
   { cs << 'a';
     cs << 'b';
     cs << 'c';
     cs << 'd';
     cs << 'e';
     cs >> c;
     printf("%c\n", c);
     cs >> c;
     printf("%c\n", c);
    }
  catch (Stack<char>::StackError& s)
   { s.Print(); }
}
```