ЛАБОРАТОРНАЯ РАБОТА №7
СОЗДАНИЕ АБСТРАКТНЫХ КЛАССОВ

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:

Целью данной лабораторной работы является изучение и создание абстрактных классов в языке C++.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:

1. Виртуальные функции

Фнкция - член класса может содержать спецификатор virtual. Такая функция называется виртуальной. Спецификатор virtual может быть использован только в объявлениях нестатических функций-членов класса.

Если некоторый класс содержит виртуальную функцию, а производный от него класс содержит функцию с тем же именем и типами формальных параметров, то обращение к этой функции для объекта производного класса вызывает функцию, определённую именно в производном классе. Функция, определённая в производном классе, вызывается даже при доступе через указатель или ссылку на базовый класс. В таком случае говорят, что функция производного класса подменяет функцию базового класса. Если типы функций различны, то функции считаются разными, и механизм виртуальности не включается. Ошибкой является различие между функциями только в типе возвращаемого значения.

```
1) class Base
2)  { public:
3)     virtual void f1();
4)     virtual void f2();
5)     virtual void f3();
6)     void f();
7)   };
```

```
1) class Derived :  public Base
2)  { public:
3)     void f1();
4)     void f2(int);
5)     char f3();
6)     void f();
7)  };
8) / Скрывает Base::f2()
9) // Ошибка – различие только в типе возвращаемого
   значения!
```

```
1) Derived *dp = new Derived;
2) Base    *bp = dp;
3) // Преобразование указателя на производный класс в
   указатель на базовый класс
4)
5) bp->f(); // Вызов Base::f
6) dp->f();  // Вызов Derived::f
7) dp->Base::f(); // Вызов Base::f
```

```
1) dp->f1(); // Вызов Derived::f1
2) bp->f1(); // Всё равно вызов Derived::f1!!!
3) bp->Base::f1();// Вызов Base::f1 (использование явного
   квалификатора  блокирует механизм виртуальности)
```

```
1) bp->f2();
2) dp->f2(0);
3) dp->f2();
4) dp->Base::f2();
5)
6) // Вызов Base::f2
7) // Вызов Derived::f2
8) // Ошибка, т.к. не задан параметр
9) // Вызов Base::f2
```

Виртуальную функцию можно использовать, даже если у её класса нет производных классов. Производный класс, который не

нуждается в собственной версии виртуальной функции, не обязан её реализовывать.

Интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызывается, в то время как интерпретация вызова невиртуальной функции-члена класса зависит от типа указателя или ссылки, указывающей на этот объект.

Этот механизм делает производные классы и виртуальные функции ключевыми понятиями при разработке программ на C++. Базовый класс определяет интерфейс, для которого производные классы обеспечивают набор реализаций. Указатель на объект класса может передаваться в контекст, где известен интерфейс, определённый одним из его базовых классов, но этот производный класс неизвестен. Механизм виртуальных функций гарантирует, что этот объект всё равно будет обрабатываться функциями, определёнными для него, а не для базового класса.

Спецификатор virtual предполагает принадлежность функции классу, поэтому виртуальная функция не может быть ни глобальной функцией, ни статическим членом класса, поскольку вызов виртуальной функции нуждается в конкретном объекте для выяснения того, какую именно функцию следует вызывать.

Подменяющая функция в производном классе также считается виртуальной, даже при отсутствии спецификатора virtual.

Виртуальная функция может быть объявлена дружественной в другом классе.

Такое поведение, когда функции базового класса подменяются функциями производного класса, независимо от типа указателя или ссылки, называется полиморфизмом. Тип, имеющий виртуальные функции, называется полиморфным типом. Для достижения полиморфного поведения в языке C++ вызываемые функции-члены класса должны быть виртуальными, и доступ к объектам должен осуществляться через ссылки или указатели. При непосредственных манипуляциях с объектом (без использования указателя или ссылки) его точный тип известен компилятору, и поэтому полиморфизм времени выполнения не требуется.

Для реализации механизма виртуальности используются таблицы виртуальных функций. Каждый объект класса, имеющего виртуальные функции, содержит таблицу виртуальных функций, в

которой хранятся адреса виртуальных функций, определённых для того класса, к которому реально принадлежит объект. Поскольку при создании объекта его тип известен, компилятор может определить адреса виртуальных функций этого класса и записать их в таблицу виртуальных функций.

При вызове виртуальной функции её адрес определяется не на этапе компиляции, а во время выполнения программы. Из таблицы виртуальных функций берётся элемент с определённым номером и вызывается функция, находящаяся по этому адресу. Таким образом, для вызова виртуальной функции требуется одна дополнительная операция выбора значения из таблицы виртуальных функций.

Соображения эффективности ни в коем случае не должны считаться препятствием для использования виртуальных функций. Для реализации механизма виртуальности требуется количество памяти, равное размеру указателя, на каждую виртуальную функции и одна операция выбора значения из памяти на каждый вызов виртуальной функции. Однако другие методы, реализующие подобный механизм, также потребуют накладных расходов. Например, при использовании поля типа также требуется память и время на выполнение проверок, причем если количество памяти невелико, то временные затраты, особенно при большом количестве производных классов, могут быть несопоставимо больше. Кроме того, механизм виртуальных функций позволяет легко модифицировать программы и добавлять новые классы.

1.1. Виртуальные деструкторы

Многие классы требуют некоторой формы очистки до уничтожения объекта. Так как базовый класс не может знать, требуется ли в производном классе такая очистка, он должен предположить, что требуется. Необходимая очистка будет гарантированно производиться в том случае, если деструктор в базовом классе объявлен как виртуальный. В этом случае деструктор производного класса будет подменять деструктор базового класса, и правильный деструктор будет вызываться даже по указателю или ссылке на базовый класс, как и в случае с другими виртуальными функция. Подмена деструктора производится даже несмотря на то, что формально деструкторы в

разных классах имеют разные имена. Однако деструктор является специальной функцией и существует в каждом классе в единственном экземпляре. Поэтому создание и использование виртуальных деструкторов возможно. Виртуальный деструктор производного класса будет вызывать деструкторы базовых классов.

1.2. Виртуальные конструкторы

После знакомства с виртуальными деструкторами возникает вопрос: «Может ли конструктор быть виртуальным?» Ответ – нет, но желаемый эффект можно получить достаточно просто.
Для того чтобы создать объект, конструктор должен знать его точный тип. Следовательно, конструктор не может быть виртуальным. Более того, конструктор является не совсем обычной функцией. В частности, он взаимодействует с процедурами управления памятью способом, недоступным обычным функциям-членам класса. Как следствие, невозможно получить указатель на конструктор.
Оба ограничения можно обойти, определив функцию, которая вызывает конструктор и возвращает созданный объект.

```
1) class Base
2)  { ...
3)    public:
4)     Base();
5)     Base(const Base& b);
6)     virtual Base* Create() { return new Base();      }
7)     virtual Base* Clone()  { return new Base(*this); }
8)   };
```

Так как функции вроде Create() и Clone() являются виртуальными и создают (косвенно) объекты, их часто называют «виртуальными конструкторами». Однако на самом деле они не являются конструкторами в обычном понимании, просто каждая из них использует конструктор для создания подходящего объекта.
Для создания объекта собственного типа производный класс может заместить функции Create() и/или Clone().

```
1) class Derived : public Base
2)  { ...
3)    public:
4)     Derived();
5)     Derived(const Derived& d);
6)     Derived* Create() { return new Derived();       }
7)     Derived* Clone()  { return new Derived(*this); }
8)   };
```

Теперь имея объект класса Base или производного от него класса, можно создать новый объект такого же типа.

```
1) void f(Base *p)
2)  { Base *p1 = p->Create(); }
3) // Указатель, присвоенный p1, имеет корректный, но
   неизвестный тип
```

Значения, возвращаемые функциями Derived::Create() и Derived::Clone(), имеют тип Derived*, а не Base*. Это позволяет при необходимости создавать новые объекты без потери информации о типе.

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

Тип подменяющей функции должен быть таким же, как тип виртуальной функции, которую она подменяет, за исключением того, что допускаются послабления по отношению к типу возвращаемого значения. Если исходный тип возвращаемого значения был B*, то тип возвращаемого значения подменяющей функции может быть D* при условии, что B является открытым базовым классом для D. Аналогично, вместо B& тип возвращаемого значения может быть ослаблен до D&.

2. Абстрактные классы

Механизм абстрактных классов служит для представления общих понятий, которые фактически используются лишь для порождения более конкретных понятий. Абстрактный класс можно также употреблять как определение интерфейса, в котором производные классы обеспечивают разнообразие реализаций.

Абстрактный класс – это класс, который может использоваться лишь в качестве базового класса для некоторого другого класса. Класс является абстрактным, если он содержит хотя бы одну чистую виртуальную функцию.

Виртуальная функция называется чистой, если в объявлении функции внутри объявления класса задан чистый спецификатор = 0.

```
1) class Shape
2)  { public:
3)      virtual void draw() = 0; // Чистая виртуальная
   функция
4)      ...
5)   };
```

Абстрактный класс нельзя употреблять в качестве типа объектов, типа параметров функций, типа возвращаемого функцией значения или как тип при явном приведении типа. Можно, однако, объявлять указатели и ссылки на абстрактный класс.

```
1) Shape  s; // Ошибка: объект абстрактного класса
2) Shape *s; // Всё правильно
3) Shape  f(); // Ошибка
4) void   f(Shape s); // Ошибка
5) Shape& f(Shape &s); // Всё правильно
```

Чистые виртуальные функции наследуются и остаются чистыми виртуальными функциями, таким образом, производный класс, в котором чистая виртуальная функция не переопределена, остаётся абстрактным классом.

3. Пример «Геометрические фигуры»

Файл Shapes.h

```
1) #define SHAPES
2)
3) class Shapes
   // Класс Shapes является абстрактным, т.к. содержит
   чистые виртуальные функции
4)  { protected:
5)     static int count;
6)     int color;
7)     int left, top, right, bottom;
8)     Shapes()  { count++; }
9)
10)    public:
11)     enum {LEFT, UP, RIGHT, DOWN};
12)     virtual ~Shapes() { count--; }
13)
14)     static int GetCount() { return count;  }
15)     int Left()   const { return left;   }
16)     int Top()    const { return top;    }
17)     int Right()  const { return right;  }
18)     int Bottom() const { return bottom; }
19)     virtual void Draw() = 0;
   // Чистые виртуальные функции
20)     virtual void Move(int where, const Shapes
   *shape) = 0;
21)     virtual Shapes* NewShape() = 0;
22)     virtual Shapes* Clone()    = 0;
23)   };
```

Файл Shapes.cpp

```
1) #include "Shapes.h"
2) int Shapes::count = 0;
```

Файл Circle.h

```
1) #if !defined(SHAPES)
2)  #include "Shapes.h"
3) #endif
4)
5) class Circle : public Shapes
6)  { private:
7)     int cx, cy, radius;
8)    public:
9)     Circle(int x = 0, int y = 0, int r = 0, int c =
   0);
10)     ~Circle() { }
11)     void Draw();
12)     void Move(int where, const Shapes *shape);
13)     Circle* NewShape() { return new Circle(); }
14)     // При прямом вызове конструктора копирования не
   осуществляется вызов конструктора базового класса, и
   поэтому, в данном случае,
15)     // не происходит увеличения значения переменной
   Shapes::count, содержащей количество объектов класса
   Shapes
16)     Circle* Clone()    { Circle *p = new Circle();
   *p = *this; return p ; }
17)   };
```

Файл Circle.cpp

```
1) #include "Circle.h"
2)
3) Circle::Circle(int x, int y, int r, int c)
4)  { cx    = x; cy = y;   radius = r;
5)    color = c;
6)    left  = cx - radius; top    = cy - radius;
7)    right = cx + radius; bottom = cy + radius;
8)   }
9)
10) void Circle::Draw()
11)  { ... }
```

```
12)
13) void Circle::Move(int where, const Shapes *shape)
14)  {
15)    switch (where)
16)     { case LEFT:
17)        cx = shape->Left() - radius;
18)        cy = (shape->Top() + shape->Bottom()) / 2;
19)        break;
20)       case UP:
21)        cx = (shape->Left() + shape->Right()) / 2;
22)        cy = shape->Top() - radius;
23)        break;
24)       case RIGHT:
25)        cx = shape->Right() + radius;
26)        cy = (shape->Top() + shape->Bottom()) / 2;
27)        break;
28)       case DOWN:
29)        cx = (shape->Left() + shape->Right()) / 2;
30)        cy = shape->Bottom() + radius;
31)        break;
32)      }
33)    left  = cx - radius; top    = cy - radius;
34)    right = cx + radius; bottom = cy + radius;
35)   }
```

Файл Triangle.h

```
1) #if !defined(SHAPES)
2)  #include "Shapes.h"
3) #endif
4)
5) class Triangle : public Shapes
6)  { private:
7)     int x1, y1, x2, y2, x3, y3;
8)    public:
```

```
9)     Triangle(int x1 = 0, int y1 = 0, int x2 = 0, int
   y2 = 0, int x3 = 0, int y3 = 0, int c = 0);
10)     ~Triangle() { }
11)     void Draw();
12)     void Move(int where, const Shapes *shape);
13)     Triangle* NewShape() { return new Triangle(); }
14)     // При прямом вызове конструктора копирования не
   осуществляется вызов конструктора базового класса, и
   поэтому, в данном случае,
15)     // не происходит увеличения значения переменной
   Shapes::count, содержащей количество объектов класса
   Shapes
16)     Triangle* Clone()    { Triangle *p = new
   Triangle(); *p = *this; return p ; }
17)   };
```

Файл Triangle.cpp

```
1) #include "Triangle.h"
2)
3) int Max(int a, int b, int c);
4) int Min(int a, int b, int c);
5)
6) Triangle::Triangle(int x1, int y1, int x2, int y2, int
   x3, int y3, int c)
7)  { this->x1 = x1;
8)    this->y1 = y1;
9)    this->x2 = x2;
10)   this->y2 = y2;
```

```
11)    this->x3 = x3;
12)    this->y3 = y3;
13)    color  = c;
14)    left   = Min(x1, x2, x3);
15)    top    = Min(y1, y2, y3);
16)    right  = Max(x1, x2, x3);
17)    bottom = Max(y1, y2, y3);
```

```
18)    }
19)
20)  void Triangle::Draw()
21)   { ... }
22)
23)  void   Triangle::Move(int where, const Shapes *shape)
24)   { int dx, dy;
25)
26)     switch (where)
27)      { case LEFT:
28)          dx = shape->Left() - right;
29)          dy = (shape->Top() - top + shape->Bottom() -
   bottom) / 2;
30)          break;
31)        case UP:
32)          dx = (shape->Left() - left + shape->Right() -
   right) / 2;
33)          dy = shape->Top() - bottom;
34)          break;
35)        case RIGHT:
36)          dx = shape->Right() - left;
37)          dy = (shape->Top() - top + shape->Bottom() -
   bottom) / 2;
38)          break;
39)        case DOWN:
40)          dx = (shape->Left() - left + shape->Right() -
   right) / 2;
41)          dy = shape->Bottom() - top;
42)          break;
43)       }
44)     x1 += dx; y1 += dy;
45)     x2 += dx; y2 += dy;
46)     x3 += dx; y3 += dy;
47)     left   = Min(x1, x2, x3);
48)     top    = Min(y1, y2, y3);
49)     right  = Max(x1, x2, x3);
50)     bottom = Max(y1, y2, y3);
51)    }
52)  ;
```

Файл main.cpp

```
1) #include "Circle.h"
2) #include "Triangle.h"
3)
4) void main()
5)  { Shapes* shapes[10];
   // Т.к. класс Shapes является абстрактным,
6)
   // можно объявить массив указателей, но не массив
   фигур
7)    shapes[0] = new Circle(100, 100, 30, 50);
8)    shapes[1] = new Triangle(0, 0, 20, 0, 0, 20, 90);
9)    shapes[2] = new Circle(200, 200, 50, 20);
10)    shapes[3] = shapes[0]->NewShape();
11)    shapes[4] = shapes[1]->NewShape();
12)    shapes[5] = shapes[0]->Clone();
13)    shapes[6] = shapes[1]->Clone();
14)
15)    for(int i = 0; i < Shapes::GetCount(); i++)
16)     shapes[i]->Draw();
17)
18)    for(int i = 1; i < Shapes::GetCount(); i++)
19)     shapes[i]->Move(Shapes::LEFT, shapes[i - 1]);
20)
21)    for(int i = 0; i < Shapes::GetCount(); i++)
22)     shapes[i]->Draw();
23)
24)    for(int i = 0, n = Shapes::GetCount(); i < n;
   i++)
25)     delete shapes[i];
26)   }
```

ЗАДАНИЕ

Создать базовый абстрактных класс «Человек», имеющий нереализованную виртуальную функцию вывода информации на

экран. Затем создать классы «Ученик» и «Босс», уна- следованные от него.