Семинар 11
Виртуальные функции, Абстрактные классы, Шаблоны функций

Виртуальные функции

Если в определении нестатического метода класса добавить спецификатор virtual, то этот метод становится виртуальным. Виртуальные функции используются при наследовании. Если функция определена как виртуальная в базовом классе, то её повторное определение (даже без virtual) в производном классе также создаст виртуальную функцию.

Виртуальные функции

Чаще всего виртуальные функции применяются в абстрактных классах и при работе с указателями базового класса, настроенными на объект производного класса.

Чистая виртуальная функция – это функция следующего вида, не имеющая тела:
virtual тип имя (специф_парам) = 0;
Конструкция = 0 – чистый спецификатор.

Абстрактные классы

Если по смыслу задачи все виртуальные функции базового класса планируется переопределять в производных классах, то такой базовый класс можно определить как абстрактный.

Абстрактный класс – это класс, в котором есть хотя бы одна чистая виртуальная функция (см. предыдущий слайд)

Абстрактные классы

Свойства абстрактных классов:

- содержит хотя бы одну чистую виртуальную функцию
- нельзя создать объект абстрактного класса
- абстрактный класс может использоваться только в качестве базового
- в абстрактные классы могут входить поля данных и методы, в т.ч. не чистые виртуальные

Абстрактные классы

Свойства абстрактных классов (продолж.):

- можно определить конструктор, из которого можно вызывать методы абстрактного класса за исключением чистых виртуальных функций

Абстрактные классы

Пример. Базовый абстрактный класс.

```
class figure{
protected:
    double xc, yc, dx, dy;
public:
    figure(double x, double y, double dxx, double dyy)
        :xc(x), yc(y), dx(dxx), dy(dyy) {}
    void grow(double d){
        dx+=d; dy+=d;
    }
    virtual double area() = 0;
    virtual char * className() = 0;
};
```

Абстрактные классы

Пример. Производные классы.

```
struct ellipse: public figure{
    ellipse(double x, double y, double r1, double r2)
        :figure(x, y, r1, r2) {}
    virtual double area(){return (dx/2)*(dy/2)*3.14159;}
    virtual char * className() {return "ellipse";}
};
struct rectangle: public figure{
    rectangle(double x, double y, double w, double h)
        :figure(x, y, w, h) {}
    virtual double area(){return dx*dy;}
    virtual char * className() {return "rectangle";}
};
```

Абстрактные классы

Пример. Тело функции main():

```
int _tmain(int argc, _TCHAR* argv[])
{
ellipse E(1.1,2.2,3.0,4.0);
rectangle R(5.0,6.0,7.7,8.8);
cout << "Rectangle area = " << R.area() << endl;
cout << "Rectangle className = " << R.className() << endl;
return 0;
}
```

Абстрактные классы

Имея массив указателей типа базового класса на объекты производного класса, можно вызывать одноимённые методы для объектов различных классов.

Абстрактные классы

Пример. Вывод информации о всех объектах массива. Добавим в базовый класс оператор вывода:

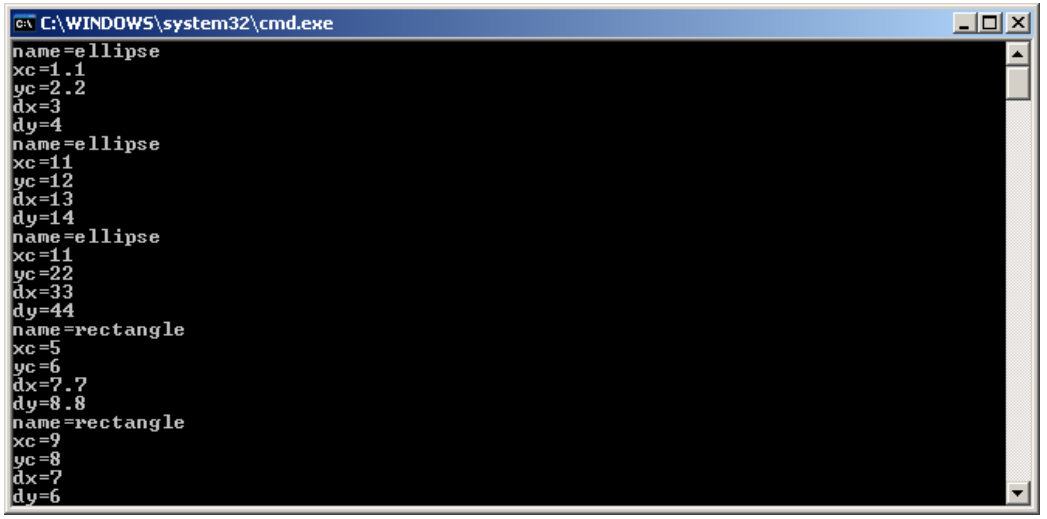
```
class figure{
protected:
        double xc, yc, dx, dy;
public:
        figure(double x, double y, double dxx, double dyy)
                :xc(x), yc(y), dx(dxx), dy(dyy) {}
        void grow(double d){
                dx+=d; dy+=d;
        }
        virtual double area() = 0;
        virtual char * className() = 0;
        friend ostream & operator << (ostream &out, figure& fig);
};
ostream & operator << (ostream &out, figure& fig){
out << "name=" << fig.className() << endl;
out << "xc=" << fig.xc << endl;
out << "yc=" << fig.yc << endl;
out << "dx=" << fig.dx << endl;
out << "dy=" << fig.dy << endl;
return out;}
```

Абстрактные классы

Пример. Вывод информации о всех объектах массива. Добавим функцию вывода для всех элементов массива

```
void showInfo(figure * fig[], int k){
        for(int i=0; i<k; i++)
                cout << *fig[i];
}

int _tmain(int argc, _TCHAR* argv[])
{
        ellipse E1(1.1,2.2,3.0,4.0);
        ellipse E2(11, 12, 13, 14);
        ellipse E3(11, 22, 33, 44);
        rectangle R1(5.0,6.0,7.7,8.8);
        rectangle R2(9, 8, 7, 6);
        figure *fig[] = {&E1, &E2, &E3, &R1, &R2}; // Это массив указателей на объекты абстр. класса
        showInfo(fig, 5);
        return 0;
}
```



Шаблоны функций

Шаблоны функций служат для реализации функционального полиморфизма – т.е. для того, чтобы функция с одним именем могла принимать параметры различных типов. Определяя шаблон функций, вы определяете семейство функций. Формат определения:

template<список_параметров_шаблона>
Определение_шаблонной_функции

Шаблоны функций

Пример шаблонной функции и её вызова

```
template<typename T>
void swap(T* x, T* y){
	T z = *x;
	*x = *y;
	*y = z;
}

int _tmain(int argc, _TCHAR* argv[])
{
	double a = 3.1;
	double b = 4.1;
	swap(&a,&b); // Можно вызвать явно: swap<double>(&a,&b)
	// или снова неявно swap<>(…)
	cout << "a=" << a << " b=" << b << endl;
	return 0;
}
```

Шаблоны функций

Терминология.
Шаблон семейства функций служит для формирования конкретных определений функций, называемых *специализациями шаблонной функции*. Процесс формирования специализации называют *инстанцированием шаблона*.

Шаблоны функций

Пример инстанцирования

Шаблон:

```
template<typename T>
void swap(T* x, T* y){
	T z = *x;
	*x = *y;
	*y = z;
}
```

Вызов:

```
double a = 3.1, b = 4.1;
swap(&a,&b);
```

Специализация:

```
void swap(double* x, double* y){
	T z = *x;
	*x = *y;
	*y = z;
}
```

Шаблоны функций

Терминология. Продолжение.
Процесс определения типов специализации по типам параметров, переданных при вызове функции, называется *выведением аргументов шаблона функции.*

Шаблоны функций

Основные свойства шаблона функций:

- Список параметра шаблона не может быть пустым
- Каждый аргумент в списке параметров шаблона начинается с typename или class, аргументов может быть несколько
- Кроме типизирующих параметров у шаблона могут быть нетипизирующие параметры, тогда они явно специфицируются в заголовке шаблона
- Аргумент, соответствующий типизирующему параметру шаблона, может иметь любой тип, такой, что тело функции имеет смысл для этого типа

Шаблоны функций

Пример нетипизирующих параметров в шаблоне

```
template<int START, typename T>
T sum(T arr[], int len=1){
	T summa=T(0);
	for(int i=START; i<START+len; i++)
	summa+=arr[i];
	return summa;
}

int _tmain(int argc, _TCHAR* argv[])
{
	float f[] = {2.1, 3.2, 4.3, 5.4};
	int i[] = {9, 8, 7, 6, 5, 4, 3};
	cout << "sum<1>(f,2)=" << sum<1>(f,2) << endl;
	cout << "sum<2, int>(i, 4)=" << sum<2, int>(i, 4);
	return 0;
}
```

Шаблоны функций

Пример случая, когда тело функции неприменимо к конкретному вызову (см. 4-е свойство на предыдущем слайде)

```
template<typename T>
T absval(T x){
	return (x < T(0)? -x : x);
}

int _tmain(int argc, _TCHAR* argv[])
{
	double a = -31.9;
	cout << absval(a); // OK
	char * hw = "Hello world";
	cout << absval(hw); // NOT OK
	// error: унарная операция «-» (минус) недопустима для
	// типа char *
	return 0;
}
```

Шаблоны функций

Основные свойства шаблона функций:

- Нельзя использовать в заголовке шаблона параметры с одинаковыми именами
- Имя типизирующего параметра имеет в шаблонной функции все права имени типа (см. предыдущий пример – функция absval возвращает значение типа T, принимает в качестве параметра значение типа T, и создаёт внутри тернарного оператора безымянный экземпляр типа T)

Шаблоны функций

Основные свойства шаблона функций:

• При вызове типы аргументов, соответствующие одинаково параметризированным параметрам, должны быть одинаковы.

Шаблоны функций

Пример некомпилируемого кода
(см. предыдущее свойство)

```
template<typename T>
T greater(T x, T y){
        return (x > y? x : y);
}

int _tmain(int argc, _TCHAR* argv[])
{
        float f = 3.14;
        int i = 3;
        cout << greater(f, i); // Ошибка: T greater(T,T):
        // в шаблоне параметр "T" неоднозначен
        return 0;
}
```