BLG252E-OBJECT ORİENTED PROGRAMMİNG

20.04.2015

Practice Session 3

What will we cover?

- Templates
- Exception Handling

What will we do?

- We will design a **GenericArray** class, an object of which contains an array of elements of a selected type (built in types or objects from user defined classes). The array is initialized with the **size** information.
- As a user defined class, we will create a **Fraction** class to represent fractional numbers with a numerator and a denominator.
- **GenericArray** objects will throw an exception whenever array index is out of bounds.
- We will design **GenericArray** as a generic class **template**, so that it runs smoothly with the test programs provided.

Test Program by Using Built-in Type **int**

```
int main() {
    // To test GenericArray by using built-in type int
    GenericArray<int> m1(5); // creates an empty 5-element-integer array
    GenericArray<int> m2(3); // creates an empty 3-element-integer array
    for (int i = 0; i <= 5; i++) {
        try {
            m1[i] = i;
        } catch (const string & err_msg) { // exception handler
            cout << err_msg << endl; // writes "index out of bounds"
        }
    }
    GenericArray<int> m3 = m2 = m1;
    // does not throw an exception if sizes are not equal,
    // the required precautions are taken
    // now, both m2 and m3 have 5 elements
```

Test Program by Using Built-in Type **int**

```
for (int i = 0; i <= 5; i++) {
    try {
        cout << m3[i] << " ";
    } catch (const string & err_msg) { // exception handler
        cout << err_msg << endl; // writes "index out of bounds"
    }
}

if (m1.contains(3))
    cout << "Element 3 is contained in the array" << endl;
else
    cout << "Element 3 is not contained in the array" << endl;

cout << endl;
```

Test Program by Using User Defined Class **Fraction**

```
// To test GenericArray by using user-defined class Fraction
GenericArray<Fraction> m4(3); // An array with three empty spaces
Fraction cObj1(3, 5); // A Fraction object
Fraction cObj2 = cObj1;
Fraction cObj3(3, 4);
cObj2.setDenom(7); // sets the denominator of the Fraction object as 7

try {
    m4[0] = cObj1;
    m4[1] = cObj2;
    m4[2] = cObj3;
} catch (const string & err_msg) { // exception handler
    cout << err_msg << endl; // writes "index out of bounds"
}
```

Test Program by Using User Defined Class **Fraction**

```
    for (int i = 0; i <= 3; i++) {
        try {
            cout << m4[i] << " ";
        } catch (const string & err_msg) { // exception handler
            cout << err_msg << endl; // writes "index out of bounds"
        }
    }

    if (m4.contains(Fraction(3, 7)))
        cout << "The element is contained in the array" << endl;
    else
        cout << "The element is not contained in the array" << endl;

    return 0;
}
```

Expected Output

```
index out of bounds
0 1 2 3 4 index out of bounds
Element 3 is contained in the array

3/5 3/7 3/4 index out of bounds
The element is contained in the array
```

Solutions (Fraction class)

```
// A class to represent fractional numbers  with a numerator and a denominator.
class Fraction{
    unsigned int num,denom;
 public:
    // default constructor
    Fraction(){ num= 1; denom = 1;}
    // constructor with initialization parameters
    Fraction(unsigned int num_in, unsigned int denom_in) {
        num=num_in;
        if (denom_in==0)
            denom=1;
        else
            denom=denom_in;
    }
    // set methods
    void setNum(unsigned int num_in) {num = num_in;}
    void setDenom(unsigned int denom_in) {denom = denom_in;}
```

Solutions (Fraction class)

```
    // print method
     void print() const{
        if (num == 0)
            cout<<0;
        else
            cout << num << "/" << denom;
     }
     // overloading comparison operator to be used by GenericArray.contains(const Type&)
     bool operator==(const Fraction& inObject) const{
         if (num == inObject.num && denom == inObject.denom)
             return true;
         else
             return false;
     }
};
```

Solutions (overloading operator<<)

```
// overloading operator<< to be able to print Fraction objects by using cout
// operator<< should return an ostream,
// so that "chained" output like: cout << fObj1 << " " << fObj2 << endl will work
// it must be defined as! a free function, not a class method!!
ostream& operator<< (ostream &out, const Fraction& inObject){
     if (num == 0)
          out<<0;
       else
          out << num << "/" << denom;
    return out;
}
```

Solutions (GenericArray class)

```
// A generic dynamic array for any type(built-in and user defined)
template <class Type>
class GenericArray{
    Type *elements;
    int size;

public:
    // Constructor creates a dynamic array with given size
    GenericArray(int s){
        size = s;
        elements = new Type[size];
    }

    // destructor to give dynamically allocated space back
    ~GenericArray(){
        delete[] elements;
    }
```

Solutions (GenericArray class)

```
// copy constructor is necessary as we need to allocate space dynamically for elements
GenericArray(const GenericArray& inArray){
    size = inArray.size;
    elements = new Type[size];
    for (int i = 0; i < size; i++ )
        elements[i] = inArray.elements[i];
}

// check if the specified element is contained in the array or not
bool contains(const Type& element) const{
    for (int i = 0; i < size; i++ )
        if (elements[i] == element)
            return true;
    return false;
}
```

Solutions (GenericArray class)

```
// operator= is overloaded so that it can assign different sized arrays
 const GenericArray& operator= (const GenericArray& inArray) {
     // If given array has a different size,
     if (inArray.size != size){
         // resize contained array
         delete[] elements;
         size = inArray.size;
         elements = new Type[size];
     }
     // assign elements of given array to contained array
     for (int i = 0; i < size; i++ )
         elements[i] = inArray.elements[i];
     // return the resulting GenericArray object to be able to make assignments in a chain
     return *this;
 }
```

Solutions (GenericArray class)

```
    // operator[] is overloaded to be able to return an element by its index
    Type& operator[](int index){
        // throw an exception whenever the given index is out of array boundaries
        if (index < 0 || index > size-1)
            throw string("index out of bounds");
        return elements[index];
    }
}
```