Exceptions

Dr. Sanem Sarıel Talay

OOP – Exceptions

- Provide a systematic, OO approach to handle runtime errors
- To qualify as an exception, such errors must occur as a result of some action taken within a program which can discover them
  - a constructor in a user-written string class might generate an exception if the application tries to initialize an object with a string that's too long.
  - a program can check if a file was opened or written successfully and generate an exception if it was not.

Old approach to handle errors

```
if( somefunc() == ERROR_RETURN_VALUE )
  // handle the error or call error-handler function
else
  // proceed normally
if( anotherfunc() == NULL )
  // handle the error or call error-handler function
else
  // proceed normally
if( thirdfunc() == 0 )
  // handle the error or call error-handler function
else
  // proceed normally
```

Old approach - disadvantages

- Every single call to such a function must be examined by the program.
- Long listings, hard to read
- Sometimes, it is not simple to return an error value if there is already a return value - min()
- Errors may take place without a function being explicitly called:
  - SomeClass obj1, obj2, obj3;
  - The constructor is called implicitly, so there's no return value to be checked.

New approach - exceptions

- If an error is detected in a member function, this member function informs the application that an error has occurred: *throwing an exception*
- In the application, a separate section of code is provided to handle the error: exception handler or *catch block*: it catches the exceptions thrown by the member function.
- Any code in the application that uses objects of the class is enclosed in a *try block*.

Throwing an exception

- Syntax of a function f that throws an exception:

```
return_type f(parameters){
   if (exception_condition) throw exceptioncode;
       // normal operation
   return expression;
 }
```

- Here *exceptioncode* can be any variable or constant of any built-in type (as char, int, char *) or it can also be an object that defines the exception.

Exception example

- A fraction function takes the numerator and denominator as parameters, calculates the result of the fraction and returns it back.
- If the denominator is zero an exception must be thrown.

Exception example, con't

```
float fraction(int num, int denom) {
     if(denom==0) throw "Divide by zero";
     return static_cast<float>(num) / denom;
}
int main(){
     int numerator, denominator;
     cout << endl << "Enter the numerator ";
     cin >> numerator;
     cout << endl << "Enter the denominator ";
     cin >> denominator;
     try{
         cout << fraction(numerator, denominator);
     }
     catch (const char * result){
          cout << endl << result;
    }
   cout << endl << "End of Program";
   return 0;
}
```

See Example: e10_1.cpp

Throwing exceptions

- In a catch block you may catch only the type of the exception-code, if the code itself is not necessary.

```
catch (const char *){
  cout <<endl<< "ERROR";//The thrown data is unknown
}
```

- A function may throw more than one exception. For example if we don't want negative denominators, we can write the fraction function as follows:

```
float fraction(int num, int denom){
    if(denom == 0) throw "Divide by zero";
    if(denom < 0) throw "Negative denominator";
    return static_cast<float>(num) / denom;
}
```

Throwing exceptions, con't

- A function may also throw exceptions of different types. In this case, a separate catch block must be written for each exception type.

```
float fraction(int num, int denom){
  if(denom == 0) throw "Divide by zero";//throws char *
  if(denom < 0) throw "Negative denom.";//throws char *
   if(denom > 1000) throw -1;       // throws int
    return static_cast<float>(num) / denom;
}
try {
  cout << fraction(numerator , denominator);
} // Catch block for exceptions of type char *
catch (const char * result) {
          cout << endl << result;
} // Catch block for exceptions of type int
catch (int) {
          cout << endl << "ERROR";
}
```

See Example: e10_2.cpp

Throwing objects

- Like built-in data types, objects can also be thrown and caught as exceptions.
- Examine the example e10_3.cpp. In this program we have a class: Stack. This class includes two functions push and pop. If an error occurs, these functions throw an object of class Error.

See Example: e10_3.cpp

Exceptions in Constructors

- Exceptions are necessary to find out if an error occurred in the class constructor. Constructors are called implicitly and there's no return value to be checked.
- The creator of the String class does not allow the contents of the String to be longer than 10 characters.

Exceptions in Constructors, con't

```
class String{
   enum { MAX_SIZE = 10 };      // MAX_SIZE is a constant
   int size;
   char *contents;
 public:
   String(const char *);              // Constructor
   void print() const;                // A member function
   ~String();                         // Destructor
};
String::String(const char *in_data){
   cout<< "Constructor has been invoked" << endl;
   size = strlen(in data);
   if (size > MAX_SIZE) throw "String too long";
   contents = new char[size +1];            // +1 for null
   character
   strcpy(contents, in_data);
 }
```

Exceptions in Constructors, con't

```
int main(){
   char input[20];               // To take strings from keyboard
   String *str;                  // Pointer to objects
   bool again;                   // loop condition
   do{
    again = false;
    cout << " Enter a string: ";
    cin >> input;
    try{
        str = new String(input);         // calls the constructor
    }
    catch (const char *){
         cout << "String is too long" << endl;
         again = true;
    }
   }while(again);
   str->print();        //The creation of the object is guaranteed
    delete str;
       return 0;
```

See Example: e10_4.cpp