Exceptions

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

Overview

- Introduction
- Exception Syntax
- Throwing an Exception
- Catching an Exception
- Constructors and Exception Handling

Exceptions

- Exceptions provide a systematic, object-oriented approach to handling runtime errors generated by C++ classes
- To qualify as an exception, such errors must
  - occur as a result of some action taken within a program, and
  - be ones the program itself can discover
- Examples:
  - A constructor in a user-written string class might generate an exception if the application tries to initialize an object with a string that is too long
  - A program can check if a file was opened or written to successfully and generate an exception if it was not

Why Do We Need a New Mechanism to Handle Errors?

- Let us look at how the process was handled in the past
- In C language programs, an error is often signaled by returning a particular value from the function in which it occurred
- For example, many math functions return a special value to indicate an error, and disk file functions often return NULL or 0 to signal an error
- Each time you call one of these functions, you check the return value

Obsolete error handling:

```
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
```

Problems With the Old Error Handling Mechanism

- The problem with this approach is that every single call to such a function must be examined by the program
- Surrounding each function call with an `if...else` statement and inserting statements to handle the error (or to call an error-handler routine) makes the listing long and hard to read
- Also, it is not practical for some functions to return an error value
  - For example, imagine a `min()` function that returns the minimum of two values
  - All possible return values from this function represent valid outcomes
  - There is no value left to use as an error return

Problems With the Old Error Handling Mechanism

- The problem becomes more complex when classes are used because errors may take place without a function being explicitly called
- For example, suppose an application defines objects of a class:

```
SomeClass obj1, obj2, obj3;
```

- How will the application find out if an error occurred in the class constructor?
- The constructor is called implicitly, so there is no return value to be checked

Exception Syntax

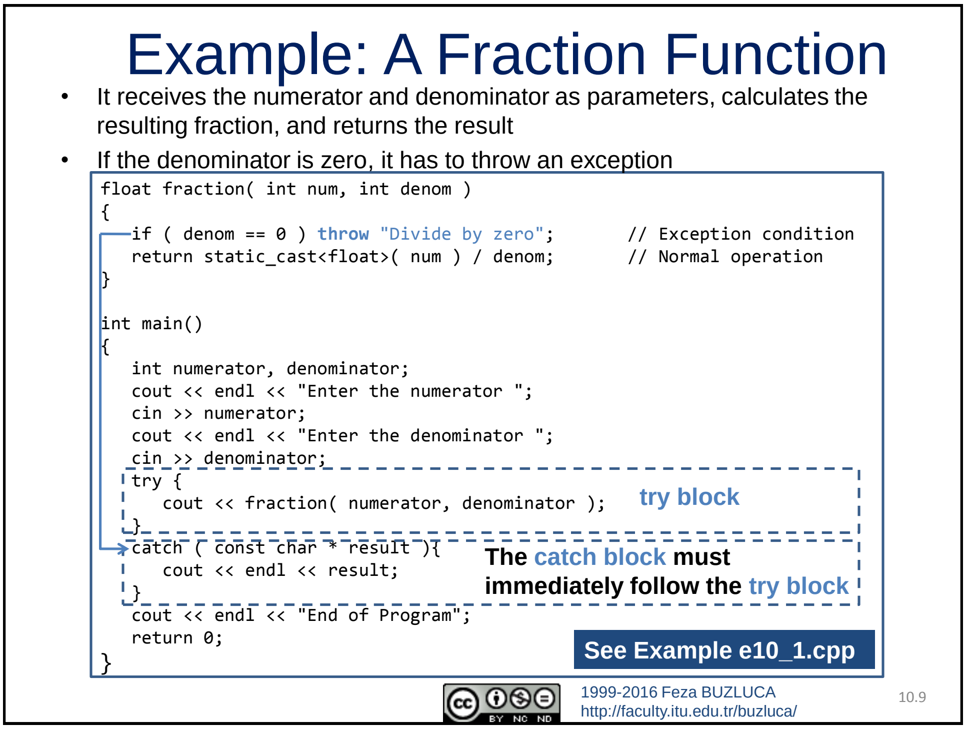
- If an error is detected in a member function, this member function informs the application that an error has occurred
- When exceptions are used, this is called throwing an exception
- In the application, a separate section of code is installed to handle the error
- This code is called an exception 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
- The exception mechanism uses three new C++ keywords:
  - `throw`
  - `catch`
  - `try`

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 (such as char, int, char *), or
  - an object that defines the exception



Example: A Fraction Function

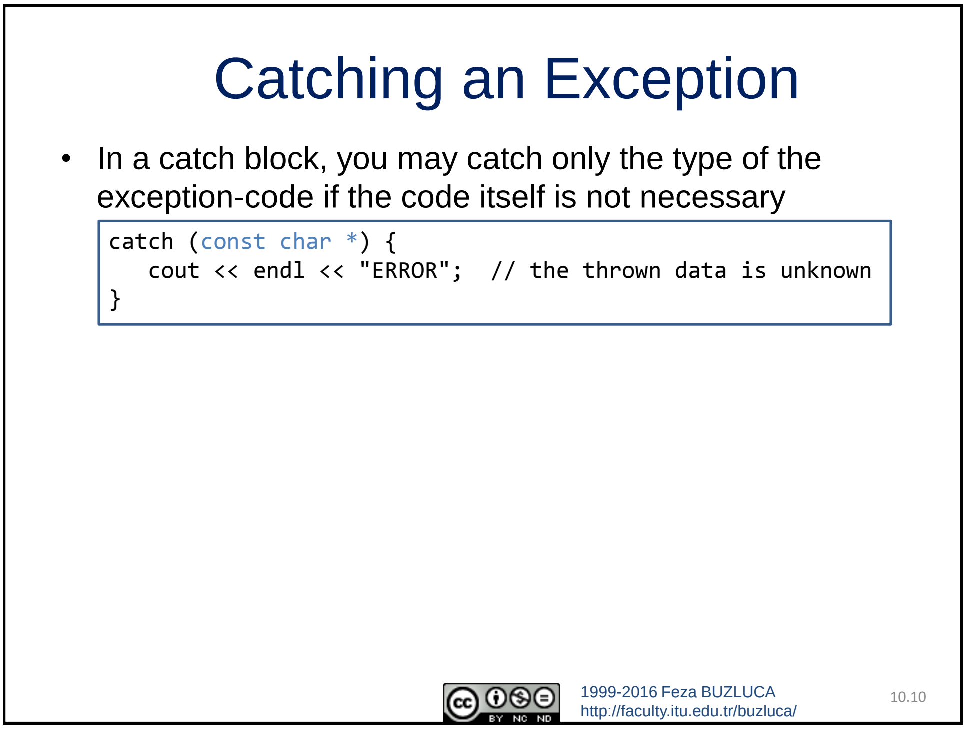
- It receives the numerator and denominator as parameters, calculates the resulting fraction, and returns the result
- If the denominator is zero, it has to throw an exception

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

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;
}
```

try block

The catch block must immediately follow the try block

See Example e10_1.cpp



Catching an Exception

- 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
}
```

Throwing Multiple Exceptions

- A function may throw more than one exception
- For example, if we do not want negative denominators, we can write the fraction function as:

```
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 Multiple Exceptions of Different Types

- A function may also throw exceptions of different types

```
float fraction( int num, int denom )
{
   if ( denom == 0 ) throw "Divide by zero";      // throws char *
   if ( denom < 0 ) throw "Negative denominator"; // throws char *
   if ( denom > 1000 ) throw -1;                  // throws int
   return static_cast<float>( num ) / denom;
}
```

Catch Blocks for Different Exception Types

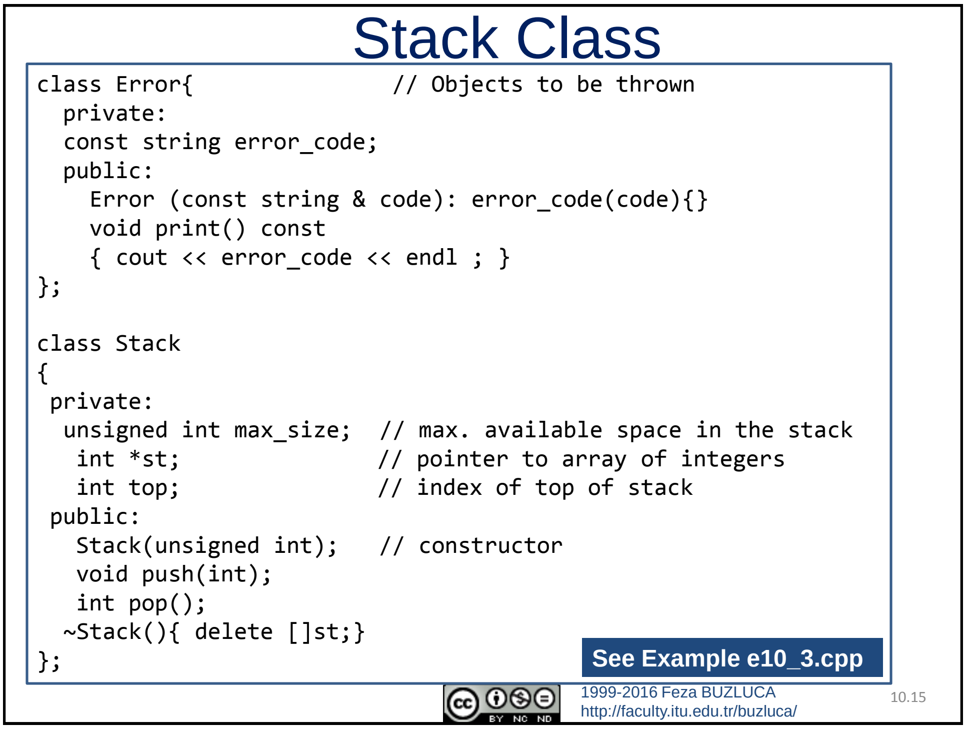
- If a function throws exceptions of different types, then a separate catch block must be written for each exception type

```
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 (value is not taken)
catch ( int ) {
   cout << endl << "ERROR";
}
```

See Example e10_2.cpp

Objects Can Also Be Thrown and Caught As Exceptions

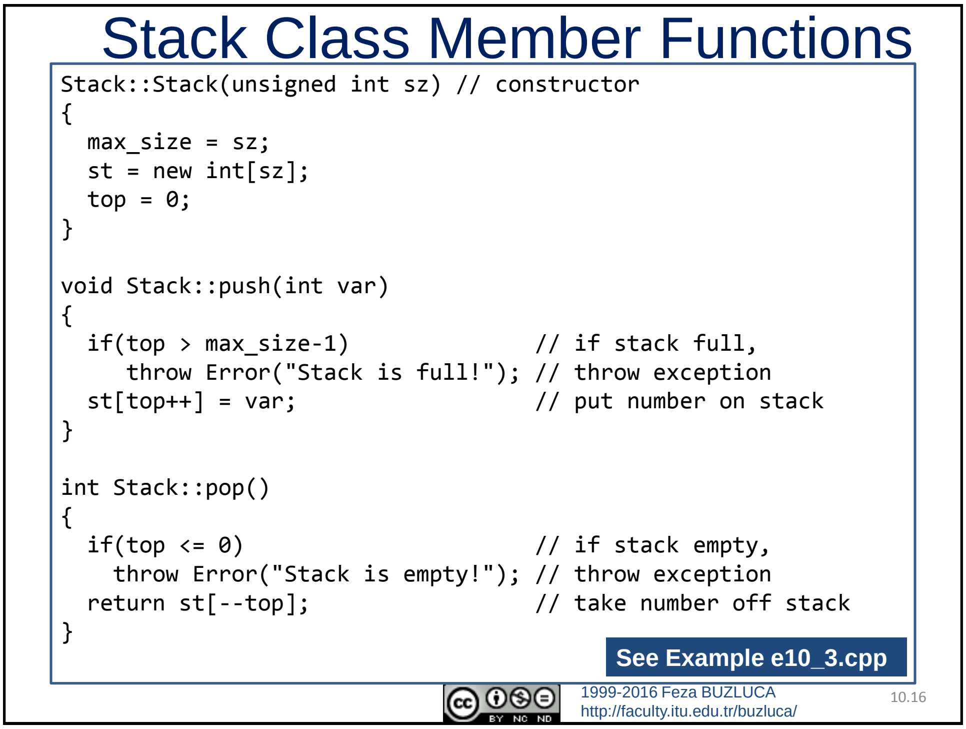
- Like built-in data types, objects can also be thrown and caught as exceptions
- **See Example e10_3.cpp**
  - In this program, we have a class: `Stack`
  - This class includes two functions: `push` and `pop`
  - If an error occurs, these functions throw an object of class `Error`



Stack Class

```
class Error{               // Objects to be thrown
  private:
  const string error_code;
  public:
    Error (const string & code): error_code(code){}
    void print() const
    { cout << error_code << endl ; }
};

class Stack
{
 private:
  unsigned int max_size;  // max. available space in the stack
   int *st;               // pointer to array of integers
   int top;               // index of top of stack
 public:
   Stack(unsigned int);   // constructor
   void push(int);
   int pop();
  ~Stack(){ delete []st;}
};
```

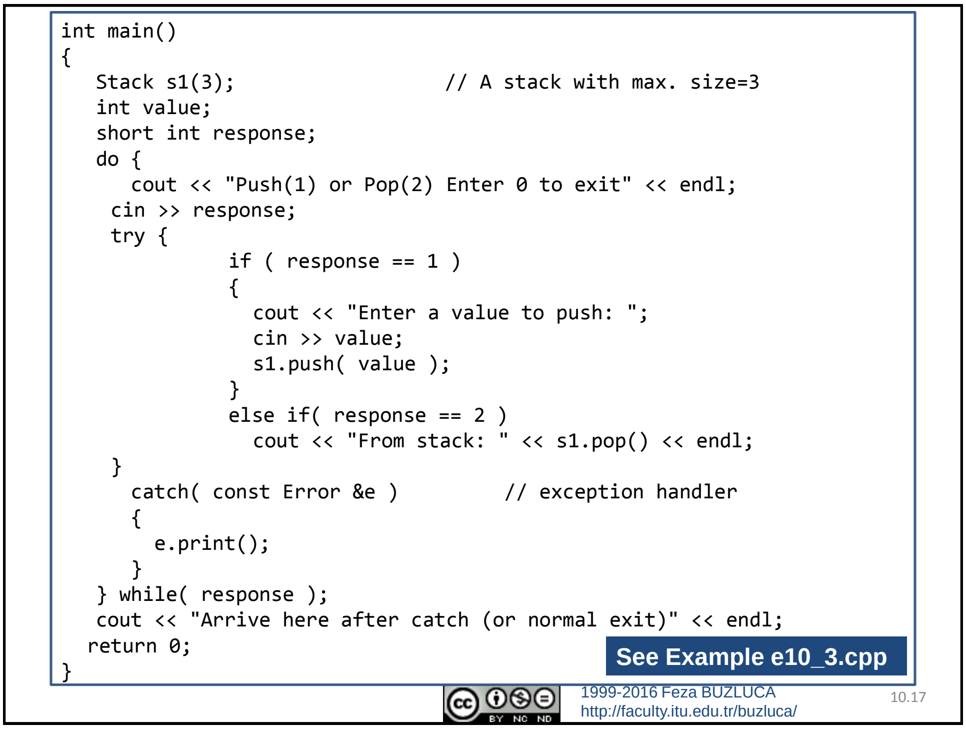
See Example e10_3.cpp



Stack Class Member Functions

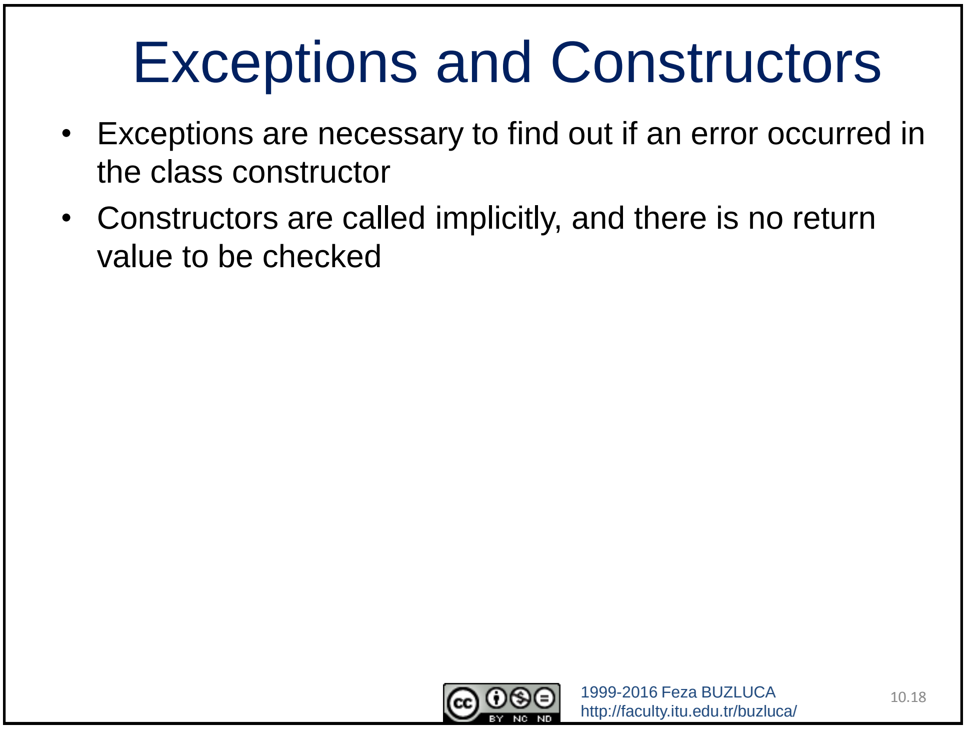
```
Stack::Stack(unsigned int sz) // constructor
{
  max_size = sz;
  st = new int[sz];
  top = 0;
}

void Stack::push(int var)
{
  if(top > max_size-1)              // if stack full,
     throw Error("Stack is full!"); // throw exception
  st[top++] = var;                  // put number on stack
}

int Stack::pop()
{
  if(top <= 0)                      // if stack empty,
    throw Error("Stack is empty!"); // throw exception
  return st[--top];                 // take number off stack
}
```

See Example e10_3.cpp



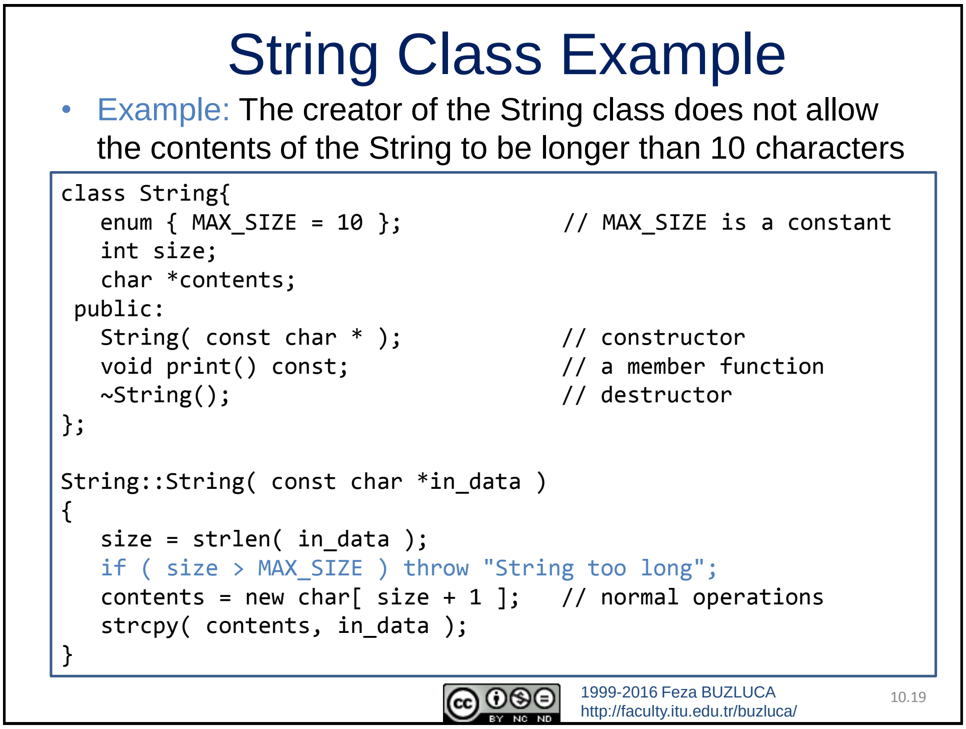
```
int main()
{
   Stack s1(3);                    // A stack with max. size=3
   int value;
   short int response;
   do {
      cout << "Push(1) or Pop(2) Enter 0 to exit" << endl;
    cin >> response;
    try {
              if ( response == 1 )
              {
                cout << "Enter a value to push: ";
                cin >> value;
                s1.push( value );
              }
              else if( response == 2 )
                cout << "From stack: " << s1.pop() << endl;
    }
      catch( const Error &e )         // exception handler
      {
        e.print();
      }
   } while( response );
   cout << "Arrive here after catch (or normal exit)" << endl;
  return 0;
}
```

See Example e10_3.cpp



Exceptions and Constructors

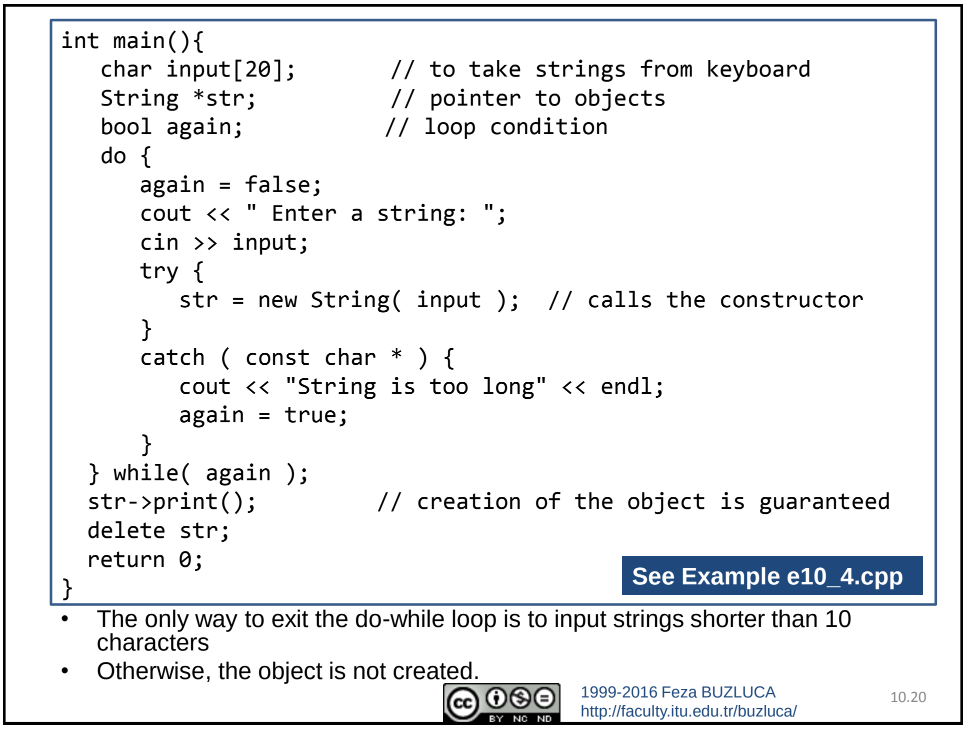
- Exceptions are necessary to find out if an error occurred in the class constructor
- Constructors are called implicitly, and there is no return value to be checked



String Class Example

- Example: The creator of the String class does not allow the contents of the String to be longer than 10 characters

```
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 )
{
   size = strlen( in_data );
   if ( size > MAX_SIZE ) throw "String too long";
   contents = new char[ size + 1 ];   // normal operations
   strcpy( contents, in_data );
}
```



```
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();        // creation of the object is guaranteed
  delete str;
  return 0;
}
```

See Example e10_4.cpp

- The only way to exit the do-while loop is to input strings shorter than 10 characters
- Otherwise, the object is not created.