EXCEPTIONS

- Exceptions provide a systematic, object-oriented approach to handle **runtime** errors generated by C++ classes.
- To qualify as an exception, such errors must occur due to some action taken within a program and must be the ones the program itself can discover.
  - For example, a constructor in a user-written `Point` class might generate an exception if the application tries to initialize an object with coordinates that are beyond the limits.
  - A constructor can generate an exception if it cannot allocate memory for a dynamic class member. `ptr = new Type[x]; // is ptr == nullptr?`
  - A program can check if a file was opened or written successfully and generate an exception if it was not.

Error handling without exceptions:

- In C/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.

```
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
```

Error handling without exceptions (contd):

Problems (without exceptions):

- The program must examine every single call to such a function.

  Surrounding each function call with an `if/else` statement and inserting statements to handle the error (or to call an error-handler routine) make the listing long and hard to read.

- 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's no value left to use as an error return.

- The problem becomes more complex when classes are used because errors may take place without a function (constructor) being explicitly called.

  For example, suppose an application defines objects of a class:

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

  How will the application determine if an error occurred in the class constructor?

  The constructor is called implicitly, so there's no return value to be checked.

Exception Syntax

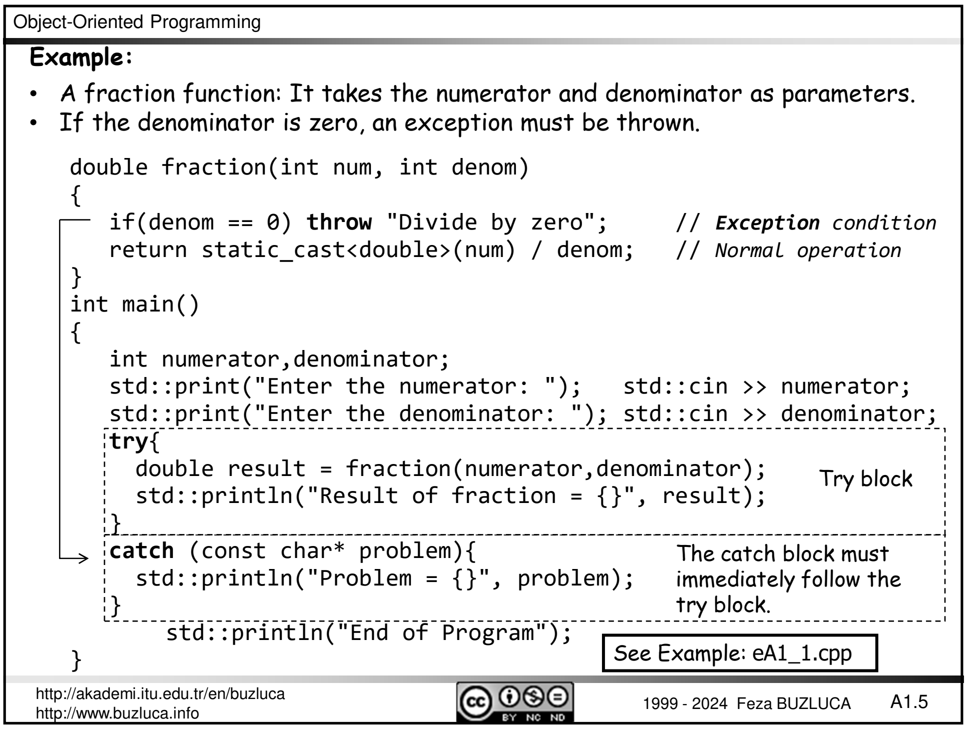
- If an error is detected in a function, it informs the application that an error has occurred.
- When exceptions are used, this is called *throwing an exception*.
- In the application, a separate section of code is installed to handle the error.
- This code is called an **exception handler** or ***catch block***: it catches the exceptions thrown by the function.
- Any code in the application that uses objects of the class is enclosed in a ***try block***.
- The exception mechanism uses three keywords: **throw**, **catch**, and **try**.

Throwing an exception:

Syntax of a function `anyFunction` that throws an exception:

```
return_type anyFunction( parameters ) {
       if ( exception_condition ) throw exception code; // break
          ...  // normal operation
       return expression;
   }
```

Here *exception code* 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.



Example:

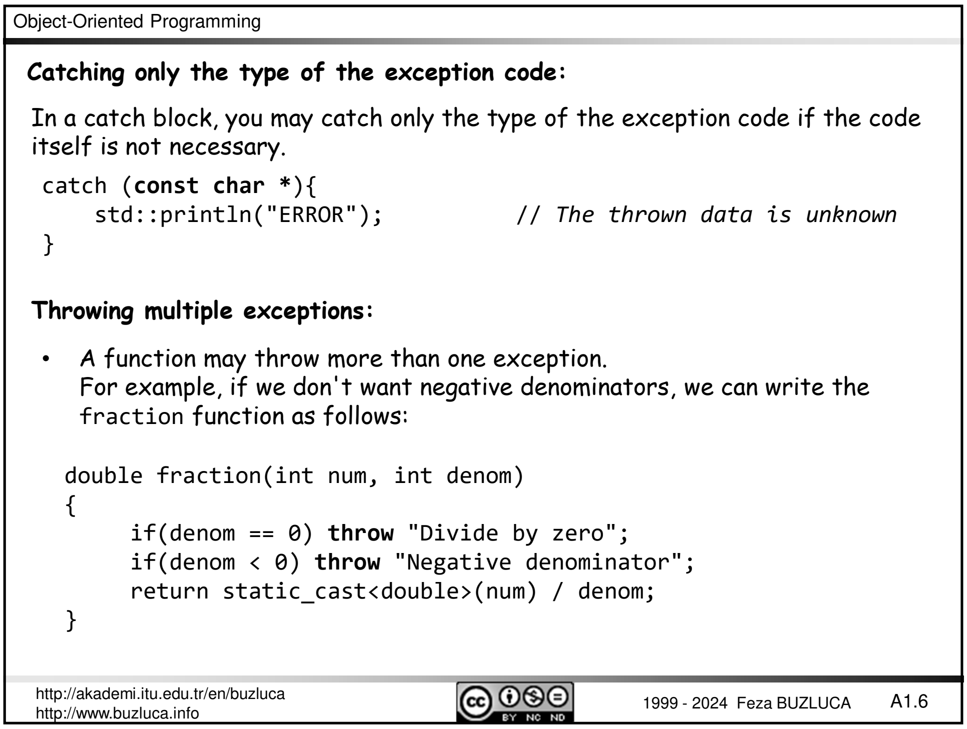
- A fraction function: It takes the numerator and denominator as parameters.
- If the denominator is zero, an exception must be thrown.

```
double fraction(int num, int denom)
{
    if(denom == 0) throw "Divide by zero";      // Exception condition
    return static_cast<double>(num) / denom;    // Normal operation
}
int main()
{
    int numerator,denominator;
    std::print("Enter the numerator: ");   std::cin >> numerator;
    std::print("Enter the denominator: "); std::cin >> denominator;
    try{
      double result = fraction(numerator,denominator);
      std::println("Result of fraction = {}", result);
    }
    catch (const char* problem){
      std::println("Problem = {}", problem);
    }
         std::println("End of Program");
}
```

Try block

The catch block must immediately follow the try block.

See Example: eA1_1.cpp



Catching only the type of the exception code:

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

```
catch (const char *){
    std::println("ERROR");          // The thrown data is unknown
}
```

Throwing multiple exceptions:

- 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:

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

Throwing exceptions of different types:

- *A function may also throw multiple exceptions of different types.*

```
double 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<double>(num) / denom;
}
```

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

```
try {
     result = fraction(numerator , denominator);
}
catch (const char * problem) {              // Catch block for char *
    std::println("Problem = {}", problem);
}
catch (int) {                  // Catch block for int (value is not taken)
    std::println("ERROR");
}
```

See Example: A1_2.cpp

Throwing objects as an exceptions:

- Like built-in data types, objects can also be thrown and caught as exceptions.

Example:

Objects of class `Error` can be thrown as expetions.

```
class Error{               // Objects to be thrown
  private:
    const std::string error_code;
  public:
    Error (const std::string & code): error_code(code){}
    void print() const
      { std::println("{}", error_code); }
};
```

In a catch block, we can catch objects of class `Error` and call its member functions.

```
catch(const Error &e)                  // exception handler
{
  e.print();
}
```

See Example: eA1_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's no return value to be checked.

Example:

The creator of the `String` class does not allow the contents of the `String` to be longer than `MAX_SIZE` characters.

```
String::String(const char *in_data)
{
   m_size = std::strlen(in_data);
   if (m_size > MAX_SIZE) throw "String too long";
   m_contents = new char[m_size +1];   // Allocate memeory if size is OK
   for (std::size_t index{ 0 }; index < m_size + 1; index++)
           m_contents[index] = in_data[index];
}
```

Example (contd):

```
int main()
{
 char input[20];
 String* str{};
 bool again;
 do{
  again = false;
  std::print(" Enter a string: "); std::cin >> input;
  try{
   str = new String{ input }; // calls the constructor to create an obj.
  }
  catch (const char * error){
   std::println("{}", error);
   again = true;
  }
 }while(again);
 str->print();
 delete str;
 return 0;
}
```

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

See Example: eA1_4.cpp