CSC 127 Exception Handling

Preview

- Defensive Programming
- Exceptions
- Exception Handling

Defensive Programming

- *Defensive programming* is a methodology that makes programs more robust to failures
  - Increases the quality of software
- Failures may be due to
  - Erroneous user input (e.g. entering a date in the wrong format)
  - File format and access problems (e.g. end of file or disk full)
  - Networks failures
  - Problems with arithmetic (e.g. overflow)
  - Hardware and software interrupts (e.g. hitting the break key)

Exceptions

- **Exception:** “An abnormal condition that arises in a code sequence at run time”.

- An exception is a run-time error.

- Exception handling allows us to manage run-time errors in an orderly fashion.

Exception Handling Principles

- The three purposes of an exception handler:
  1. Recover from an exception to safely continue execution
  2. If full recovery is not possible, display or log error message(s)
  3. If the exception cannot be handled locally, clean up local resources and re-raise the exception to propagate it to another handler
- *Exception handling* makes defensive programming easier
  - An *exception* is an error condition, failure, or other special event
  - Exceptions are handled by *exception handlers* to recover from failures
  - *Built-in exceptions* are automatically detected by the run-time system and handled by internal handlers or the program aborts
  - Exceptions can be explicitly *raised* by the program
  - Exception handlers can be user-defined routines that are executed when an exception is raised

Exceptions

- Using exception handling, our program can automatically invoke an error-handling routine when an error occurs.

- C++ exception handling is built upon three keywords: **try, catch, and throw.**

Exception Handling in C++

- C++ has no built-in exceptions:
  - Exceptions are user-defined
  - STL defines a few useful exceptions
  - Exceptions have to be explicitly raised with **`throw`**

Exception Handling Fundamentals

- **Error prone program statements** that we may want to monitor for generation of exceptions are contained in a **try block.**
- **Syntax:**

```
try {
    // try block
}
```

Exception Handling Fundamentals

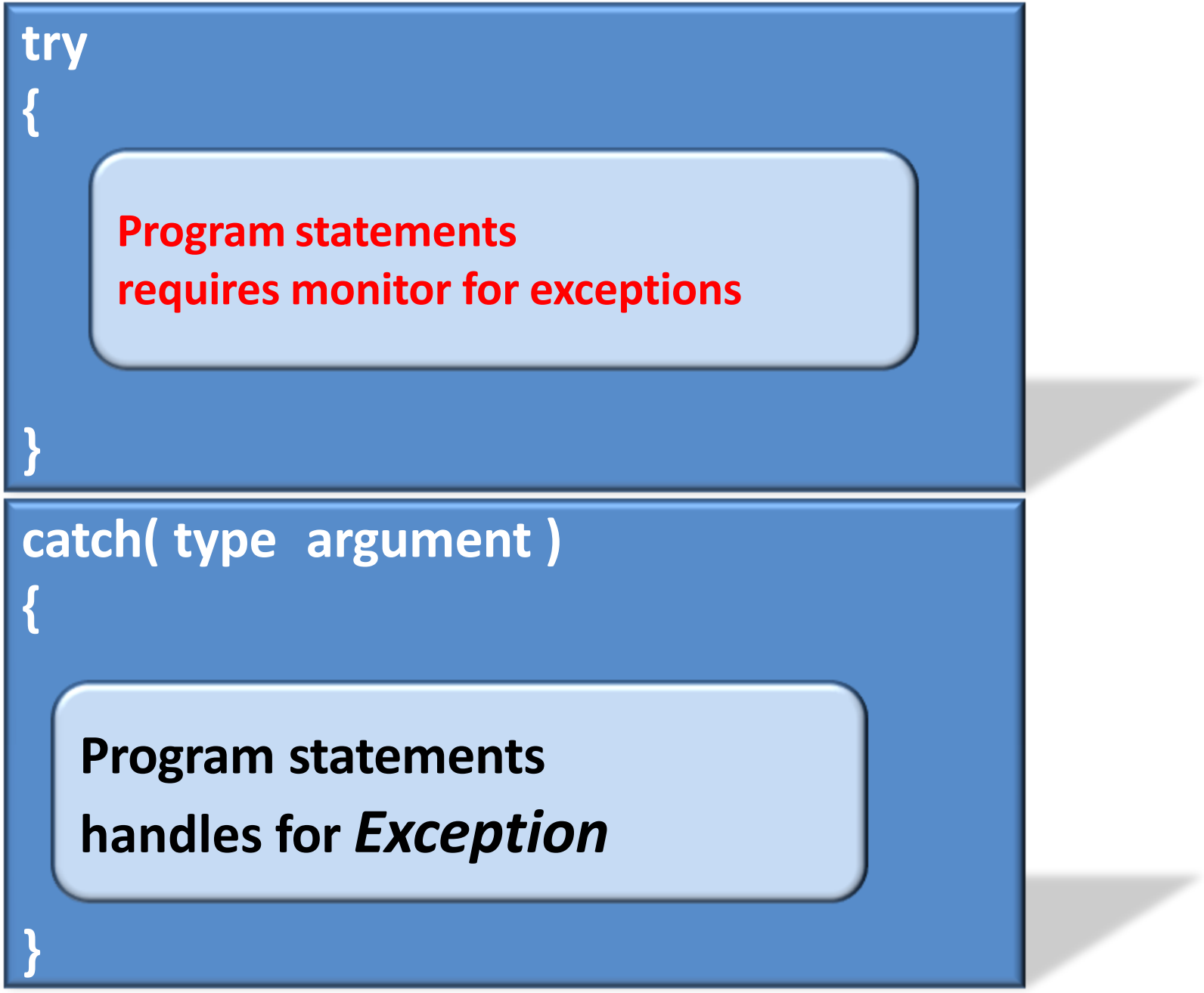
- If an exception (i.e., an error) occurs within the try block, then that exception is thrown using throw.
- Syntax:
  - throw exception;
- If an exception is to be caught, then throw must be execu either from within a try block or from any function called from within the try block (directly or indirectly).

Exception Handling Fundamentals

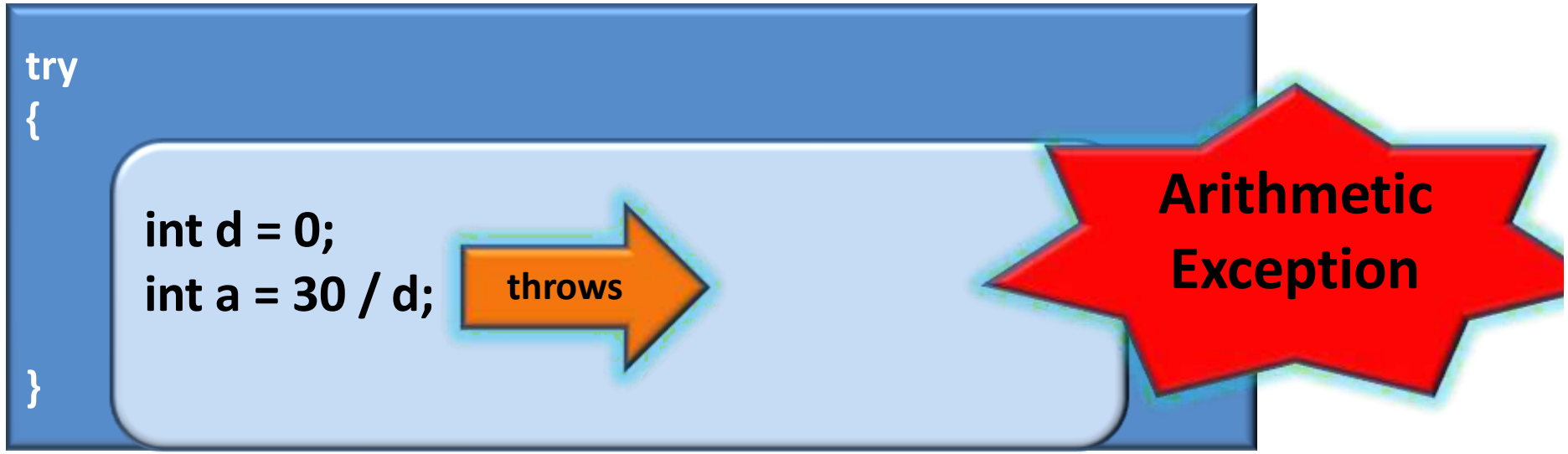
- The thrown exception is caught, using **catch block** and processed.

- **Syntax:**

```
catch (type  argument)
{
  // catch block
}
```


try
{
Program statements
requires monitor for exceptions
}
catch(type argument)
{
Program statements
handles for *Exception*
}

Using try & catch




```
catch(int e )
{
  printf("Division by zero.");
}
```

Exception Handling Fundamentals

- NOTE:
  - Throwing an unhandled exception causes the standard library function terminate() to be invoked. By default, terminate() calls abort() to stop your program.

Exception Handling in C++

- An exception in C++ is a type (typically a class):
  - **`class empty_queue`**
    `{ public empty_queue(queue q) { ... };`
    `...` *// constructor that takes a queue object for diagnostics*
    `};`
    declares an “empty queue” exception
  - **`short int eof_condition;`**
    declares a variable used to raise a "short int" exception

Exception Handling in C++ (cont'd)

- C++ exception handlers are attached to a block of statements with the **`try`**-block and a set of **`catch`**-clauses (or **`catch`**-blocks):

```
try {
   …
   … throw eof_condition;  // matches short int exception
   … throw empty_queue(myq);  // matches the empty_queue exception and
                              // passes the myq object to handler
   … throw 6;  // matches int exception and sets n=6
   …
} catch (short int) {
   … // handle end of file (no exception parameter)
} catch (empty_queue e) {
   … // handle empty queue, where parameter e is the myq empty_queue object
} catch (int n) {
   … // handle exception of type int, where parameter n is set by the throw
} catch (...) {
   … // catch-all handler (ellipsis)
}
```

Exception Handling in C++ (cont'd)

- A **`catch`**-block is executed that matches the type/class of **`throw`**
  - A **`catch`** specifies a type/class and an optional parameter
  - Can pass the exception object by value or by reference
  - The parameter has a local scope in the **`catch`**-block
- The **`catch`**(...) with ellipsis catches all remaining exceptions
- After an exception is handled:
  - Execution continues with statements *after* the **`try-catch`** and all local variables allocated in the try-block are deallocated
  - If no handler matches an exception (and there is no **`catch`** with ellipsis), the current function is terminated and the exception is propagated to the caller of the function:

```
try {
  afun(); // may throw empty queue exception that it doesn't catch
} catch (empty_queue)
{ ... // handle empty queue exception here
}
```

Exception Handling in C++ (cont'd)

- C++ supports *exception hierarchies*:
  - An exception handler for exception class *X* also catches derived exceptions *Y* of base class *X:*

```
class X {…};
class Y: public X {…};
…
try {
   …
} catch (X& e) {
   …  // handle exceptions X and Y
}
```

- In C++, functions and methods may list the types of exceptions they may raise:

```
int afun() throw (int, empty_queue)
{ ... }
```

where **`afun`** can raise **`int`** and **`empty_queue`** exceptions, as well as derived exception classes of **`empty_queue`**