COP4020 Programming Languages

Exception Handling

Robert van Engelen & Chris Lacher

Overview

- Defensive programming
- Ways to catch and handle run-time errors in programming languages that do not support exception handling
- Exception handling in C++
- Exception handling in Java

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)

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

Signal Handlers

- UNIX signal handlers are a very simple form of exception handlers
  - The **`signal`**() function allows signals to be caught or ignored:

```
signal(SIGINT, SIG_DFL);        // default setting
signal(SIGINT, SIG_IGN);        // ignore
signal(SIGINT, handle_quit);    // use handler
…
void handle_quit(int sig)
{ … }
```

- Some useful signals:
  - SIGINT interrupt program
  - SIGFPE floating point exception
  - SIGKILL kill program
  - SIGBUS bus error
  - SIGSEGV segmentation violation
  - SIGPIPE write on pipe with no reader
  - SIGALRM real-time timer expired

Catching Runtime Errors w/o Exception Handlers

- C, Fortran 77, and Pascal do not support exception handling
- Other mechanisms to handle errors add "clutter" and obscures the logic of a program

- Method 1: Functions return special error values
  - Example in C:

```
int somefun(FILE *fd)
{ ...
  if (feof(fd)) return -1; // return error code -1 on end of file
  return value;            // return normal (positive) value
}
```

  - Every function return has to be tested for values indicating an error

```
int val;
val = somefun(fd);
if (val < 0) ...
```

  - Forgetting to test can lead to disaster!

Catching Runtime Errors w/o Exception Handlers (cont'd)

- Method 2: Functions and methods set global/object status variable
  - Global variable holds error status, e.g. in C:

```
int somefun(FILE *fd)
{ errstat = 0;                    // reset status variable
  ...
  if (feof(fd)) errstat = -1;     // error detected
  return value;                   // return a value anyway
}
```

  - Another method is to include a status parameter, e.g. in C:

```
int somefun(FILE *fd, int *errstat)
{ *errstat = 0;                    // reset status parameter
  ...
  if (feof(fd)) *errstat = -1;     // error detected
  return value;                    // return a value anyway
}
```

  - Must to check status variable after each function call

Catching Runtime Errors w/o Exception Handlers (cont'd)

- Method 3: Pass an error-handling function when invoking a function or method
  - Example in C:

```
int somefun(FILE *fd, void (*handler)(int))
{ ...
  if (feof(fd)) handler(-1);  // error detected: invoke handler
  return value;               // return a value
}
```

Catching Runtime Errors w/o Exception Handlers (cont'd)

- Method 4: Use setjmp/longjmp in C to dynamically jump back out through multiple function invocations
  - Example:

```
#include <setjmp.h>
int main()
{ jmp_buf jbuf; // jump buffer holds execution context
  …
  if (setjmp(jbuf) != 0) // setjmp returns 0 on initialization of context
    … // handle longjmp's error here
  …
  somefun();
}
int somefun()
{ …
  if (some_error)
    longjmp(jbuf, 10); // jump to the setjmp(), which returns 10
  …
}
```

  - Warning: use longjmp in C only, because destructors of local objects won't be invoked

Exception Handling Implementation

- In most programming languages, exception handlers are attached to a specific collection of program statements that can raise exception(s)
- When an exception occurs in this collection of statements, a handler is selected and invoked that matches the exception
- If no handler can be found, the exception is propagated to exception handlers of the outer scope of the statements, or if no handler exists in the outer scope, to the caller of the subroutine/method
- When propagating the unhandled exception to the caller, the current subroutine/method is cleaned up:
  - Subroutine frames are removed and destructor functions are called to deallocate objects

Exception Handling in C++

- C++ has no built-in exceptions:
  - Exceptions are user-defined
  - STL has some useful exceptions, I/O streams uses exceptions
  - Exceptions have to be explicitly raised with **`throw`**
- 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
} 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`**

Exception Handling in Java

- All Java exceptions are objects of classes that are descendants of class **`Throwable`**
- Classes **`Error`** and **`Exception`** are descendants of **`Throwable`**
  - **`Error`**: built-in exceptions such as "out of memory"
  - **`Exception`**: all user-defined exceptions are derived from **`Exception`**
  - **`RuntimeException`**: base class for built-in dynamic semantic errors, such as **`IOException`** for I/O exceptions
- Example user-defined exception:

```
class MyException extends Exception
{ public MyException() {};
  public MyException(String msg)
  { super(msg); // class Exception handles the message
  }
}
```

Exception Handling in Java (cont'd)

- An exception is explicitly raised with **`throw`**:
- Examples:

```
throw new MyException();
throw new MyException("some error message");
```

- The syntax of the **`catch`**-handlers in Java is the same as C++, but Java also has an optional **`finally`** block:

```
try {
   ...
} catch (MyException e) {
   ...  // catch exceptions that are (descendants of) MyException
} catch (Exception e) {
   ...  // a catch-all handler: all exceptions are descendants of Exception
} finally {
   ...  // always executed for clean-up operations
}
```

Exception Handling in Java (cont'd)

- The **`finally`** block is always executed, even when a **`break`** or **`return`** statement is executed in the try-block
- A **`catch`**-handler for an exception also handles exceptions that are descendents of that exception class
- After an exception is handled in a **`catch`**-block:
  - Execution continues with the statements in the **`finally`** block (if any)
  - And then the statements that follow the **`try-catch`** blocks
- If no handler matches the raised exception:
  - The current method is terminated
  - The **`finally`** block executed (if any)
  - Exception is propagated to the caller

Exception Handling in Java (cont'd)

- Java class methods must list the exceptions that they may raise:
  - Those that can be raised by the system and are not locally caught
  - Those that explicitly raised by **`throw`** and are not locally caught
  - For example:

```
class GradeList
{ ...
  void BuildDist() throws IOException
  { ... // I/O operations here may raise IOException
  }
  ...
}
```

- The Java compiler will verify the list of exceptions for completeness
- The exception classes **`Error`** and **`RuntimeException`** and their descendants are *unchecked exceptions* and are not verified by the compiler and do not need to be listed
- There are no default exception handlers or catch-all handlers
  - For a catch-all: **`Exception`** catches any exception