

9. Exception Handling

Overview

- Defensive programming
- Ways to catch and handle run-time errors without exception handling
- Exception handling in C++
- Exception handling in Java

Note: These notes cover Section 8.5 of the textbook, upto and including 8.5.2. Helpful extra hand-out material is available in class.

Defensive Programming

- *Defensive programming* is a technique that makes programs more *robust* to unexpected events such as run-time errors
 - Less program "*crashes*" at run time increases the quality of software
- Unexpected events may occur due to
 - Erroneous user input (e.g. entering a date in the wrong format)
 - File input and output problems (e.g. end of file or disk full)
 - Problems with arithmetic (e.g. overflow)
 - Hardware and software interrupts (e.g. hitting the break key)
- Programming language implementation of *exception handling* can make defensive programming easier
 - An *exception* is a special unexpected error condition at run time
 - Built-in exceptions may be detected automatically by the language implementation
 - Exceptions can be explicitly *raised*
 - Exceptions are handled by *exception handlers* to recover from error conditions.
 - Exception handlers are user-defined program fragments that are executed when an exception is raised

Catching Run-Time Errors Without Exception Handling

- C, Fortran 77, and Pascal do not support exception handling
- Other ways of catching and handling run-time errors have to be invented when a language does not support exception handling
- Adds "clutter" that obscures a program
- Method 1: Functions can 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 Run-Time Errors without Exception Handling (cont'd)

- Method 2: Functions and procedures can set status values
 - 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
}
```
 - Need to check status value after each function call
- Method 3: Pass an error-handling procedure to a subroutine
 - Example in C:


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

Exception Handlers

- Purposes of an exception handler:
 1. Recover from an exception to safely continue execution
 2. If full recovery is not possible, print error message(s)
 3. If the exception cannot be handled locally, clean up local resources and reraise the exception to propagate it to another handler
- In most languages, exception handlers can be attached to a collection of program statements
- When an exception occurs in the collection of statements, a handler is selected 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
 - When propagating the unhandled exception to the caller, the current subroutine is cleaned up: subroutine frames are removed and destructor functions are called to remove objects

◦ `short int eof_condition;`

declares a variable used to throw a "short int" exception

- Exception handlers are attached to a collection of statements using `try-catch`:
 - `try` {
 - ...
 - ... `throw eof_condition;` // matches short int exception
 - ... `throw empty_queue(myq);`
 - ... `throw 6;` // matches int exception
 - ...
 - } `catch (shortint)` {
 - ... // handle end of file
 - } `catch (empty_queue e)` {
 - ... // handle empty queue, where e is an empty_queue object
 - } `catch (int n)` {
 - ... // handle exception of type int, where n contains number
 - } `catch (...)` {
 - ... // catch-all handler

Exception Handling in C++

- No built-in exceptions:
 - Exceptions are user defined
 - Exceptions have to be explicitly raised by `throw`
- An exception is a type or a class:
 - `class empty_queue`

```
{ public empty_queue(queue q) { ... };  
    ... // constructor that takes a queue object for diagnostics  
};
```

declares an empty queue exception

Exception Handling in C++ (cont'd)

- A `catch`-block is executed that matches the type/class of the `throw` parameter
- A `catch` specifies a type/class and an *optional* parameter that is the object passed by `throw` to the `catch` just like call-by-value parameter passing, where the parameter has a *local* scope in the `catch`-block
- An *optional* `catch(...)` with ellipsis catches all remaining exceptions that do not have a matching handler
- After an exception is handled in a `catch`-block, execution continues with statements *after* the `try-catch` construct and all dynamic variables allocated in the `try`-block before the `throw` are deallocated
- If no handler matches a `throw` (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 (this example passes no empty_queue object)
- Functions can list the types of exceptions it can raise
 - `int afun() throw (int, empty_queue) { ... }`
where `afun` can raise `int` and `empty_queue` exceptions, as well as *subclasses* 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`: Java built-in interpreter exceptions such as "out of memory"
 - `Exception`: user-defined exception classes are subclasses of `Exception`
 - `RuntimeException` for dynamic semantic errors and `IOException` for I/O exceptions are predefined descendants of `Exception`
- Example user-defined exception declared as a descendant of `Exception`:
 - `class MyException extends Exception`

```
{ public MyException() {};  
  public MyException(String msg)  
  { super(msg); // let class Exception handle the message  
  }  
}
```
- An exception is raised by `throw`:
 - `throw new MyException();`
 - `throw new MyException("some specific error message");`
 - `MyException e = new MyException();`
 - ...
`throw e;`

Exception Handling in Java (cont'd)

- The syntax of the `try-catch` handlers in Java are the same as C++
- `try-catch` also has an optional `finally` block
 - `try` {
 ...
} `catch` (MyException e) {
 ... // *catch exceptions that are (descendants of)*
MyException
} `catch` (Exception e) {
 ... // *catch-all handler: all exceptions are descendants*
of Exception
} `finally` {
 ... // *always executed for user-defined clean-up*
operations
}
- The `finally` block is always executed even when a `break` or `return` statement appears in the protected `try`-block
- A handler of an exception also handles exceptions that are descendants of that exception class
- After an exception is handled in a `catch`-block, execution continues with the statements *after* the `try-catch` construct and all dynamic variables allocated in the `try`-block before the `throw` are deallocated
- If no handler matches a `throw`, the current function is terminated and the exception is propagated to the *caller* of the function

Exception Handling in Java (cont'd)

- Java methods *must* list the exceptions that it can raise
- A list of exceptions a method can raise is given using the `throws` keyword
 - `class` GradeList
 { `int` newGrade;
 ...
 `void` BuildDist() `throws` IOException
 { ... // *I/O operations that may raise IOException*
 }
 ...
 }
- The Java compiler will verify the list of exceptions for completeness
- Exception classes `Error` and `RuntimeException` and their descendants are unchecked exceptions and not verified by the compiler
- There are no default exception handlers or catch-all handlers
 - For a catch-all: `Exception` catches `Exception` and all its descendants