Robust programming – Exception handling

9 Exception handling

© 2006 Wouter Verkerke, NIKHEF

Introduction to Exception handling

- Exception handling = handling of anomalous events outside regular chain of execution
  - Purpose: Handle error conditions that cannot be dealt with locally

- Your traditional options for dealing with an error
  1. Terminate the program
     - *Not acceptable for embedded program, e.g high voltage controller*
  2. Return a value representing an error
     - *Often return type has no 'reserved values' that can be used as error flag*
  3. Return a legal value but set a global error flag
     - *You make the tacit assumption that somebody will check the flag*
  4. Call a user supplied error function
     - *Just passing the buck – user function also has no good option to deal with problem*

© 2006 Wouter Verkerke, NIKHEF

Throwing and catching

- Simple example of C++ exception handling
  - Exception is 'thrown' in case of run-time problem
  - If exception is not caught (as in example below) program terminates

```
int main() {
   double x(-3) ;
   double y = calc(x) ;
}

double calc(double x) {
  if (x<0) {
    // run-time error condition
    throw x ;
  }
  return sqrt(x) ;
}
```

© 2006 Wouter Verkerke, NIKHEF

385

Exceptions and automatic variables

- How is throwing an exception better than calling abort()?
  - If exception is thrown destructor is called for all automatic variables up to point where exception is handled (in this case up to the end of `main()`)
  - In example below, buffers of output file `ofs` flushed, and file closes properly prior to program termination

```
int main() {
   SomeClass obj ;
   double x(-3) ;
   double y = calc(x) ;
}

double calc(double x) {
  ofstream ofs ;
  if (x<0) {
    // run-time error condition
    throw x ;
  }
  return sqrt(x) ;
}
```

time line
1) throw f
2) ofs destructor (closes file)
3) obj destructor
4) exit

© 2006 Wouter Verkerke, NIKHEF

386

Catching exceptions

- You can also deal explicitly with exception ‘catching’ the exception
  - You can pass information to the handler via the thrown object

```
int main() {
   double x(-3) ;
   try {
     double y = calc(x) ;
   }
   catch (double x) {
     cout << “oops, sqrt of “
          << ”negative number:”
          << x << endl ;
   }
}

double calc(double x) {
  if (x<0) {
    // run-time error condition
    throw x ;
  }
  return sqrt(x) ;
}
```

Exceptions are caught in this {} block

Exceptions handled in this block

Display details on error using information passed by exception (value of negative number in this case)

© 2006 Wouter Verkerke, NIKHEF

Catching deep exceptions

- Exceptions are also caught by `try{}`, `catch{}` if they occur deeply inside nested function calls

```
int main() {
   double x(-3) ;
   try {
     double y = wrapper(x) ;
   }
   catch (float x) {
     cout << “
          << ”
          << e
   }
}
```

```
double wrapper(double x) {
  // do some other stuff
  return calc(x) – 5 ;
}
```

```
double calc(double x) {
  if (x<0) {
    // run-time error condition
    throw f ;
  }
  return sqrt(x) ;
}
```

Solving the problem

- You can try to solve the problem in the catch block
  - If you fail, call `throw` inside the catch block to indicate that you give up

```
int main() {
   double x(-3) ;
   int* array;
   try {
     array = allocate(10000) ;
   }
   catch (int size) {
     cout << "error allocating array of size "
          << size << endl ;

     // do some housekeeping

     if (problem_solved) {
        array = allocate(size) ;
     } else {
        throw ; // give up handling error here
     }
   }
}
```

If allocate throws an exception again it will not be caught by surrounding catch block

© 2006 Wouter Verkerke, NIKHEF

389

Solving the problem in steps

- A chain of error handlers
  - A rethrow allows a higher level error handler to deal with the problem if you can't

```
int main() {
   double x(-3) ;
   try {
     double y = wrapper(x) ;
   }
   catch (float x) {
     // second level error handling
   }
}

double wrapper(double x) {
  // do some other stuff
  try {
      calc(x) ;
  }
  catch (float x) {
     // first resort error handling
     if (!problem_solved) {
        throw ; // forward
     }
  }
}

double calc(double x) {
   throw f ;
}
```

390

There are exceptions and exceptions

- Usually more than one kind of error can occur
  - Give each error its own type of exception. Each type that is thrown can be assigned a separate catch handler:

```
int main() {
   double x(-3) ;
   try {
     double y = calc(x) ;
   }
   catch (int x) {
     cout << "oops, sqrt of "
          << "negative integer number"
          << endl ;
   }
   catch (float x) {
     cout << "oops, sqrt of "
          << "negative floating point number"
          << endl ;
   }
}
```

© 2006 Wouter Verkerke, NIKHEF

391

The catchall

- When you specify multiple exception handlers they are tried in order
  - *The catch(…) handler catches any exception*. Useful to put last in chain of handlers

```
int main() {
   double x(-3) ;
   try {
     double y = calc(x) ;
   }
   catch (int x) {
     // deal with int exception
   }
   catch (float x) {
     // deal with float exception
   }
   catch (…) {
     // deal with any other exception
   }
}
```

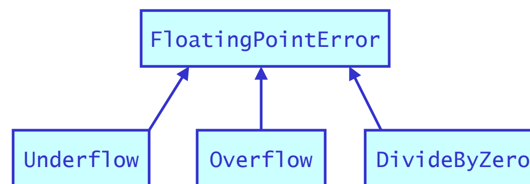
© 2006 Wouter Verkerke, NIKHEF

392

Exceptions and objects – hierarchy

- So far example have thrown floats and ints as exceptions
- What else can we throw?
  - Actually, *anything*, including objects!

- Throwing objects is particularly nice for several reasons
  1. Errors often have a hierarchy, just like objects can have
  2. Objects can carry more information, error status, context etc …
  - Example of exception hierarchy



Throwing objects

- Example of hierarchical error handling with classes
  - The hierarchy of throwable classes

```
class FloatingPointError {
  FloatingPointError(float val) : value(val) {}
 public:
  float value ;
} ;

class UnderflowError : public FloatingPointError {
  UnderflowError(float val) : FloatingPointError(val) {}
} ;

class OverflowError : public FloatingPointError {
  OverflowError(float val) : FloatingPointError(val) {}
} ;

class DivZeroError : public FloatingPointError {
  DivZeroError(float val) : FloatingPointError(val) {}
} ;
```

© 2006 Wouter Verkerke, NIKHEF

Catching objects

- Hierarchical handling of floating point errors
  - Note that if we omit the DivZeroError handler that exception is still caught (but then by the FloatingPointError handler)

```
void mathRoutine() {
  try {
    doTheMath() ;
  }

  // Catches divide by zero errors specifically
  catch(DivZeroError zde)  {
    cout << "You divided " << zde.value
         << " by zero!" << endl ;
  }

  // Catches all other types of floating point errors
  catch(FloatingPointError fpe) {
    cout << "A generic floating point error occurred,"
         << " value = " << fpe.value << endl ;
  }
}
```

© 2006 Wouter Verkerke, NIKHEF

395

Throwing polymorphic objects

- We can simplify error handling further by throwing polymorphic objects

```
class FloatingPointError {
  FloatingPointError(float val) : value(val) {}
  virtual const char* what() { return "FloatingPointError" ; }
 public:
  float value ;
} ;

class UnderflowError : public FloatingPointError {
  UnderflowError(float val) : FloatingPointError(val) {}
  const char* what() { return "UnderflowError" ; }
} ;

class OverflowError : public FloatingPointError {
  OverflowError(float val) : FloatingPointError(val) {}
   const char* what() { return "OverflowError" ; }
} ;

class DivZeroError : public FloatingPointError {
  DivZeroError(float val) : FloatingPointError(val) {}
   const char* what() { return "DivZeroError" ; }
} ;
```

© 2006 Wouter Verkerke, NIKHEF

396

Catching polymorphic objects

- Handling of all `FloatingPointErrors` in a single handler
  - Of course only works if action for all type of errors is the same (or implemented as virtual function in exception class)
  - If structurally different error handling is needed for one particular type of `FloatingPointError` you can still insert separate handler

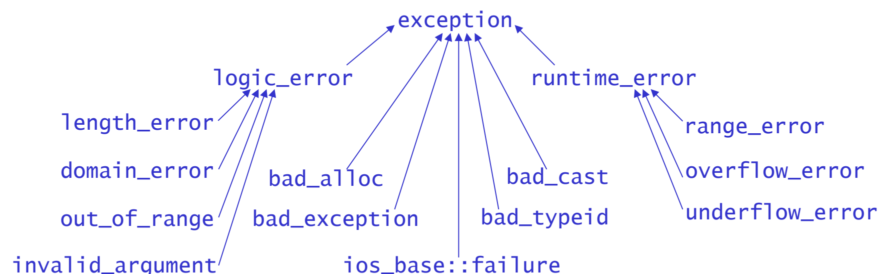
```
void mathRoutine() {
  try {
    doTheMath() ;
  }

  // Catches all types of floating point errors
  catch(FloatingPointError fpe) {
    cout << fpe.what() << endl ;
  }
}
```

© 2006 Wouter Verkerke, NIKHEF

397

Standard Library exception classes

- The Standard Library includes a hierarchy of objects to be thrown as exception objects
  - Base class '`exception`' in header file `<exception>`
- Hierarchy of standard exceptions
  - Member function `what()` returns error message



  - Note that catching `class exception` does not guarantee catching all exceptions, people are free to start their own hierarchy

© 2006 Wouter Verkerke, NIKHEF

398