HIGH-LEVEL PROGRAMMING 2

C++ Exceptions

by Prasanna Ghali

Classifying Errors

- Compile-time errors: Syntax errors, type errors
- Link-time errors: Errors found when combining object files into executable program
- Logic errors: Errors found by programmers looking for causes of erroneous results
- Run-time errors: Errors found by checks in running program

Logic Errors


```
using PDD = std::pair<double,double>;

PDD low_high(std::vector<double> const& v) {
  double low = 0.0, high = 0.0;
  for (double x : v) {
    high = (x > high) ? x : high;
    low  = (x < low)  ? x : low;
  }
  return std::make_pair(low, high);
}
```

```
-16.5 -23.2 -24.0 -25.7 -26.1 -18.6 -9.7 -2.4
7.5   12.6   23.8  25.3  28.0 34.8  36.7  41.5
40.3  42.6   39.7  35.4  12.6  6.5  -3.7  -14.3
```

```
76.5  73.5  71.0  73.6  70.1  73.5  77.6  85.3
88.5  91.7  95.9  99.2  98.2  100.6 106.3 112.4
110.2 103.6 94.9  91.7  88.4  85.2  85.4  87.7
```

Run-Time Errors

- Errors found by checks in running program
  - Errors detected by the computer [hardware and/or operating systems]
  - Error detected by a library [e.g., the standard library]
  - Error detected by user code

Sources of Errors (1/2)

- Poor specification: No clear sense of what program should do
- Incomplete programs: During development, not all cases are considered
- Unexpected arguments: Passing a value to function that can't handle it
- Unexpected input: Incorrect input provided by user

Sources of Errors (2/2)

- Unexpected state: What if data is incomplete or incorrect?
- Logical errors: Code that just doesn't do what it is supposed to do

Assertions

- *Assertion* specifies that a program satisfies certain conditions at particular points in its execution
  - *Preconditions*: specify input conditions to function
  - *Postconditions*: specify output conditions of function
  - *Invariants*: specify conditions over defined regions of program

Run-Time Assertions (1/2)

- Run-time assertions are implemented in C++ with macro `assert` declared in `<cassert>`

```
std::cout << "Enter positive integer: ";
int value;
std::cin >> value;
assert(value >= 0);
std::cout << "You entered: " << value << "\n";
```

Run-Time Assertions (2/2)

- Assertion checking can be turned off by defining NDEBUG flag to your compiler

```
// add following directive in source file
// containing assertion checking
#define NDEBUG
```

```
// or pass NDEBUG macro to compiler using flag D
g++ -D NDEBUG ...
```

Compile-Time Assertions

- Compile-time assertions are implemented in C++ with `static_assert` declaration

```
template <typename T, size_t N>
size_t f(T (&)[N]) {
  static_assert(N>=5 && "array size < 5");
  return N;
}

int ai[10];
std::cout << "f(ai): " << f(ai) << "\n";
double ad[4];
std::cout << "f(ad): " << f(ad) << "\n";
```

Run-Time Errors: Argument Errors


```
// calculate area of rectangle
int area(int length, int width) {
  return length*width;
}

// calculate area within frame
int framed_area(int x, int y) {
  return area(x - 2, y - 2);
}

int x = -3, y = 2, z = 4;
// ...
int area1    = area(x, y);
int area2    = framed_area(1, z);
double ratio = double(area1)/area3;
```

Two alternatives to deal with problem of argument errors with `area`:

1) Let caller of `area` deal with bad arguments
2) Let `area` [called function] deal with bad arguments

Caller Deals with Errors

- Messy, repetitive, and bloated code
- Caller needs to know details about how `framed_area` calls `area`
- Brittle code caused by changes to definition of `framed_area`

```
int main() {
  int x = -3, y = 2, z = 4;
  assert(x<=0 && "non-positive x");
  assert(y<=0 && "non-positive y");
  int area1 = area(x, y);

  assert(z<=2 && "non-positive 2nd argument to framed_area");
  assert(1<=2 && "non-positive 1st argument to framed_area");
  int area2 = framed_area(1, z);
}
```

Callee Deals with Errors (1/2)


```
// calculate area of rectangle
int area(int length, int width) {
  assert(length<=0 && "non-positive x");
  assert(width<=0 && "non-positive y");
  return length*width;
}

// calculate area within frame
int framed_area(int x, int y) {
  int constexpr fw{2};
  assert(x-fw<=0 && "non-positive 1st argument");
  assert(y-fw<=0 && "non-positive 2nd argument");
  return area(x - fw, y - fw);
}
```

Callee Deals with Errors (2/2)

- Respectable reasons why callee cannot deal with errors
  - No access to function definition
  - The callee doesn't know what to do [except terminate the program]
  - Callee doesn't know where it was called from
  - Performance costs – for small function, cost of assertion is more than cost of calculating result!!!
- What to do?

Run-Time Errors: Another Example

- What're our options?

```
int string_to_int(std::string const& s) {
  std::istringstream iss{s};
  int ival;
  iss >> ival;
  if (iss.fail())
    // what should we do here?

  // see if there's anything left over; if so, fail
  char left_over;
  iss >> left_over;
  if (!iss.fail())
    return ival;
  else
    // what should we do here?
}
```

Run-Time Errors: Option 1

- Terminate program to prevent it from continuing with garbage value
- Response seems drastic and suboptimal
  - Doesn't give program chance to recover from problem
  - Seems silly to terminate large, complicated software system over a single string error

Run-Time Errors: Option 2

- Use function-call-and-return system for functions to return special values meaning "hey caller, this value indicates function failed to execute correctly"
  - Not possible to put aside a value such as `-1` to indicate failure
  - If each function returns an error using sentinel value(s), we might accidently check return value of one function against error code of another function
  - There is no correct value to return for certain types such as `std::vector<int>`
  - Most serious problem is that caller can ignore return value without encountering any warnings!!!

Unsolvable Problem?

- We'd like error-handling system that combines both options
  - Option 1 prevents program from continuing normally when error occurs
  - Option 2 provides mechanism to appropriately process an error
- How can we combine both options into single system?

C++ Exceptions (1/2)

- C++ feature called *exceptions* that completely bypasses function-call-and-return
- Separates error detection [by called function] and error handling [by calling function] so that detected error cannot be ignored

C++ Exceptions (2/2)

- C++ exception system consists of three parts:
  - `try`-block
  - `throw` expression
  - `catch`-clause

qroot: Run-Time Errors


```
double qroot(double a, double b, double c) {
  double d = b*b - 4.0*a*c;
  return (-b + sqrt(d)) / (2.0*a);
}

int main() {
  double a = 1.0, b = 5.0, c = 2.0;
  std::cout << "qroot: " << qroot(a, b, c) << '\n';
  a = 1.0; b = 2.0; c = 5.0;
  std::cout << "qroot: " << qroot(a, b, c) << '\n';
  a = 0.0; b = 5.0; c = 2.0;
  std::cout << "qroot: " << qroot(a, b, c) << '\n';
}
```

```
qroot a=1, b=5, c=2: -0.438447
qroot a=1, b=2, c=5: -nan
qroot a=0, b=2, c=5: -nan
```

try block (1/2)

- Specifies region of code designated as area where run-time errors might occur
- Code in try block executes as normal and jumps to code directly following try block once finished

```
try {
  double a = 1.0, b = 5.0, c = 2.0;
  std::cout << "qroot: " << qroot(a, b, c) << '\n';
}
```

try block (2/2)

- At some point, code in try block, such as call to qroot, will cause run-time error in qroot

```
try {
  a = 1.0; b = 2.0; c = 5.0;
  std::cout << "qroot: " << qroot(a, b, c) << '\n';
}
```

`throw` Expression (1/2)

- Function causing run-time error [such as `qroot`] will use `throw` expression to indicate it has encountered something it can't handle
  - We say `throw` *raises* an exception
- Like `return`, `throw` accepts single parameter that indicates an object to throw so that exception [error] handler has access to extra information about error

throw Expression (2/2)


```
double qroot(double a, double b, double c) {
  double d = (b * b) - (4.0 * a * c);

  // protected against sqrt(-x) and division by 0
  if (d < 0.0) {
    throw (d);                    // throw double
  } else if (a == 0.0) {
    throw ("Division by 0.");  // throw const char *
  }

  // we only reach this point if no exception was thrown
  return (-b + sqrt(d)) / (2.0 * a);
}
```

catch-Clause

- When some code [such as qroot] in try block throws an exception, nearest matching catch-clause will "catch" thrown exception

```
// code here ...
try {
  // code that might throw an exception and must be protected
} catch (char const *p) { // catch value of type const char pointer
  // code to handle char pointer exception from try block above
} catch (int i) { // catch value of type int
  // code to handle int exception from try block above
} catch (exception e) { // catch an "exception" object
  // code to handle exception object from try block above
} catch (...) { // deal with all other exceptions
  ...
}
// more code here ...
```

qroot With throw Expressions (1/3)


```
double qroot(double a, double b, double c) {
  double d = (b * b) - (4.0 * a * c);
  if (d < 0.0) {
    throw(d);                    // throw double
  } else if (a == 0.0) {
    throw("Division by 0.");  // throw const char *
  }
  return (-b + sqrt(d)) / (2.0 * a);
}

int main() {
  try { // protect code
    std::cout << "qroot a=1, b=5, c=2: " << qroot(1, 5, 2) << '\n';
  } catch (char const *message) { // catch char const* const
    std::cout << message << '\n';
  } catch (double value) {  // catch double exception
    std::cout << value << '\n';
  }
}
```

qroot With throw Expressions (2/3)


```
double qroot(double a, double b, double c) {
  double d = (b * b) - (4.0 * a * c);
  if (d < 0.0) {
    throw(d);                    // throw double
  } else if (a == 0.0) {
    throw("Division by 0.");  // throw const char *
  }
  return (-b + sqrt(d)) / (2.0 * a);
}

int main() {
  try { // protect code
    std::cout << "qroot a=1, b=2, c=5: " << qroot(1, 2, 5) << '\n';
  } catch (char const *message) { // catch char const* const
    std::cout << message << '\n';
  } catch (double value) {  // catch double exception
    std::cout << value << '\n';
  }
}
```

qroot With throw Expressions (3/3)


```
double qroot(double a, double b, double c) {
  double d = (b * b) - (4.0 * a * c);
  if (d < 0.0) {
    throw(d);                    // throw double
  } else if (a == 0.0) {
    throw("Division by 0.");  // throw const char *
  }
  return (-b + sqrt(d)) / (2.0 * a);
}

int main() {
  try { // protect code
    std::cout << "qroot a=0, b=5, c=2: " << qroot(0, 5, 2) << '\n';
  } catch (char const *message) { // catch char const* const
    std::cout << message << '\n';
  } catch (double value) {  // catch double exception
    std::cout << value << '\n';
  }
}
```

Unwinding of Stack (1/2)


```
double qroot(double a, double b, double c) {
  double d = (b * b) - (4.0 * a * c);
  if (d < 0.0) {
    throw(d);
  } else if (a == 0.0) {
    throw("Division by 0.");
  }
  return (-b + sqrt(d)) / (2.0 * a);
}

void f1() {
  std::cout << "Starting f1...\n";
  qroot(1.0, 5.0, 3.0);
  std::cout << "Ending f1...\n";
}

void f2() {
  std::cout << "Starting f2...\n";
  f1();
  std::cout << "Ending f2...\n";
}
```

```
int main() {
  try { // protect code
    std::cout << "Starting main...\n";
    f2();
    std::cout << "Ending main...\n";
  } catch (const char *msg) {
    std::cout << msg << '\n';
  } catch (double value) {
    std::cout << value << '\n';
  }
}
```

Unwinding of Stack (2/2)


```
double qroot(double a, double b, double c) {
  double d = (b * b) - (4.0 * a * c);
  if (d < 0.0) {
    throw(d);
  } else if (a == 0.0) {
    throw("Division by 0.");
  }
  return (-b + sqrt(d)) / (2.0 * a);
}

void f1() {
  std::cout << "Starting f1...\n";
  qroot(0.0, 5.0, 3.0);
  std::cout << "Ending f1...\n";
}

void f2() {
  std::cout << "Starting f2...\n";
  f1();
  std::cout << "Ending f2...\n";
}
```

```
int main() {
  try { // protect code
    std::cout << "Starting main...\n";
    f2();
    std::cout << "Ending main...\n";
  } catch (const char *msg) {
    std::cout << msg << '\n';
  } catch (double value) {
    std::cout << value << '\n';
  }
}
```

Rethrowing Exceptions (1/2)


```
double qroot(double a, double b, double c) {
  double d = (b * b) - (4.0 * a * c);
  if (d < 0.0) {
    throw(d);
  } else if (a == 0.0) {
    throw("Division by 0.");
  }
  return (-b+sqrt(d))/(2.0*a);
}

void f1() {
  try {
    std::cout << "Starting f1...\n";
    qroot(0.0, 5.0, 3.0);
    std::cout << "Ending f1...\n";
  } catch (const char *msg) {
    std::cout << "Caught exception f1...
    throw;
  }
}
```

```
int main() {
  try { // protect code
    std::cout << "Starting main...\n";
    f2();
    std::cout << "Ending main...\n";
  } catch (const char *msg) {
    std::cout << msg << '\n';
  } catch (double value) {
    std::cout << value << '\n';
  }
}
```

```
void f2() {
  std::cout << "Starting f2...\n";
  f1();
  std::cout << "Ending f2...\n";
}
```

Exception Classes (1/5)

- You can throw any type of value: `throw 3;`
- And, later catch it with a clause that uses same type:

```
try {
  // call function that may throw exception ...
} catch (int a) {
  // deal with exception here ...
}
```

Exception Classes (2/5)

- Throwing primitive values is usually not good idea
- Throwing strings makes more sense but no conversions are performed when a value is thrown!!!

```
int main() {
  try {
    throw "hello world";
  } catch (std::string const& exp) {
    std::cout << "catching exception: " << exp << "\n";
  }
}
```

Exception Classes (3/5)

- To avoid such problems, more common to throw *exception objects*

```
class MyApplicationError {
  std::string reason;
public:
  MyApplicationError(std::string const& r) : reason(r) {}
  std::string const& what() const { return reason; }
};
```

- Errors are now indicated by throwing an instance of this class

Exception Classes (4/5)


```
class MyApplicationError {
  std::string reason;
public:
  MyApplicationError(std::string const& r) : reason(r) {}
  std::string const& what() const { return reason; }
};

int main() {
  try {
    // do stuff ...
    throw MyApplicationError("illegal value");
    // do more stuff ...
  } catch (MyApplicationError const& e) {
    std::cerr << "Caught exception " << e.what() << "\n";
  }
}
```

Exception Classes (5/5)

- Standard library provides hierarchy of *standard exception classes* in `<stdexcept>`
- See implementation of exception class for `hlp2::Str`

noexcept Exception Specification

- C++11 provides noexcept specification to specify that a function cannot throw or will not throw

```
void foo(int) noexcept;
void boo(int);
```

- Benefits of this qualification:
  - Overhead of stack unwinding is absent and can improve execution speed of program
  - Callers need not worry about checking for thrown exceptions