Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Exceptions

Detecting errors at runtime

- We want a way to detect and correct errors at runtime
- Many times, the author of a library (or other code module) can detect errors, but doesn't know how to correct them
- Today, we will use exceptions to separate error detection and error correction into two parts of a program

Motivation

- Recall our factorial function
- It is only valid for non-negative integers

```
//REQUIRES: n >= 0
//EFFECTS: returns n!
int factorial (int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

Motivation

- If we're asking the user for input, it would be easy to "accidentally" pass a negative value to factorial

```
int main() {
  string f; //function
  int n; //number
  while (cin >> f >> n) {
    if (f == "factorial") {
      cout << factorial(n) << endl;
    } else {
      cout << "try again" << endl;;
    }
  }
}
```

```
./a.out
factorial 5
120
factorial -5
????
```

Motivation

- Instead of the REQUIRES clause, let's look at another way of ensuring correct inputs: **runtime checking**.

- So, if we can't guarantee formally (via a specification) that the inputs are correct, maybe we can guarantee this by checking the inputs explicitly before using them in our program.

Determining legitimate output for illegitimate input

- There are three general strategies for determining legitimate output for illegitimate input:

1. "It's my problem!"
2. "I Give up!"
3. "It's your problem!"

1. “It’s my problem!”

- Try to “fix” things and continue execution by “coercing” legitimate inputs from illegitimate ones by either <u>modifying them</u> or <u>returning default outputs</u>
- For example:

```
//REQUIRES: n >= 0
//EFFECTS: returns n! for non-negative inputs and
//  1 for negative inputs
int factorial (int n) {
  // ...
}
```

1. “It’s my problem!”

```
//REQUIRES: n >= 0
//EFFECTS: returns n! for non-negative inputs and
//  1 for negative inputs
int factorial (int n) {
  // ...
}
```

- Problem: in this example, factorial is simply undefined for negative numbers, and trying to define it changes the rules of math

2. “I Give up!”

- Use `abort()` or `exit()`

```
//REQUIRES: n >= 0
//EFFECTS: returns n! for non-negative inputs and
//  crashes the program for negative inputs
int factorial (int n) {
  if (n < 0) exit(EXIT_FAILURE);
  // ...
}
```

2. “I Give up!”

```
//REQUIRES: n >= 0
//EFFECTS: returns n! for non-negative inputs and
//  crashes the program for negative inputs
int factorial (int n) {
  if (n < 0) exit(EXIT_FAILURE);
  // ...
}
```

- It is Not Nice to terminate a program this way
- What if there were open files? They could become corrupted.
- Exiting from a function deep in the call stack is (usually) not the way to handle an error!

3. “It's your problem!”

- Sometimes, the code that detects the error does not know how to correct the error
- One way to solve this is to encode failure in the return value

```
//EFFECTS: returns n! for non-negative inputs and
//  returns a negative number for negative inputs
int factorial (int n) {
  if (n < 0) return n;
  // ...
}
```

3. “It's your problem!”

- Encode failure in the return value

```
//EFFECTS: returns n! for non-negative inputs and
//  returns a negative number for negative inputs
int factorial (int n) {
  if (n < 0) return n;
  // ...
}
```

- Problem #1: we're still changing the rules of math

3. “It's your problem!”

- Problem #2: code that uses `factorial()` forgets to check

```
int main() {
  string f; //function
  int n; //number
  while (cin >> f >> n) {
    if (f == "factorial") {
      cout << factorial(n) << endl;
    } else {
      cout << "try again" << endl;;
    }
  }
}
```

3. “It's your problem!”

- Problem #3: code that uses `factorial()` gets messy

```
int main() {
  string f; //function
  int n; //number
  while (cin >> f >> n) {
    if (f == "factorial") {
      int result = factorial(n);
      if (result < 0)
        cout << "try again" << endl;
      else
        cout << result;
    } else {
      cout << "try again" << endl;
    }
  }
}
```

3. “It's your problem!”

- Encode “failure” in the **return values**
- Problem #4: sometimes you can't encode “failure” elegantly in the return values
- For example:

```
//EFFECTS: returns n^3
int cube(int n) {
  return n * n * n;
}
```

Exceptions

- *Exceptions* let us detect an error in one part of the program and correct it in a different part of the program

- For example, we could detect an error in `factorial()` and correct it in `main()`

Exception Propagation

- When an exception occurs, it *propagates* from a function to its caller until it reaches a handler

- This is called exception propagation, and it happens automatically

- In the worst case, an exception propagates up the call chain all the way to the caller of `main()`, at which point your program exits

- You can imagine exceptions as a multi-level return

Exception Handling in C++

- When code detects an error, it uses a `throw` statement
- Code that might cause an error goes in a `try{}` block
- Code that corrects an error goes in a `catch{}` block

- If the exception is successfully handled in the catch block, execution continues normally with the first statement following the catch block
- Otherwise, the exception is propagated to the enclosing block or to the caller if there is no enclosing block
- If an exception is propagated to the caller of `main()`, the program exits

Exception Handling in C++

- When code detects an error, it uses a `throw` statement
- Exceptions have types and values (just like variables)
- When we throw an exception, we specify a value for the exception type in a throw statement
- You can think of this value as being a kind of parameter to the exception, allowing some information describing the exception to be passed the handler
- Examples:
  - `int n = 0; throw n;`
  - `char c = 'e'; throw c;`

Terminology

- throw exception == raise exception
- catch block == exception handler

Exception Example

- When code detects an error, it uses a `throw` statement

```
//EFFECTS: returns n!, throws n for negative inputs
int factorial (int n) {
  if (n<0) throw n;
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

Exception Example

- When code detects an error, it uses a `throw` statement

```
//EFFECTS: returns n!, throws n for negative inputs
int factorial (int n) {
  if (n<0) throw n;
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

If `n` is non-negative, no exception is thrown and the function returns its result

Exception Example

- When code detects an error, it uses a `throw` statement

```
//EFFECTS: returns n!, throws n for negative inputs
int factorial (int n) {
  if (n<0) throw n;
  int result = 1;
  while (n > 1) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

If `n` is negative:

1. No more code from this function executes
2. Control passes “up the chain” to the caller

Exception Example

- Code that might cause an error goes in a `try{}` block

```
int main() {
  string f; //function
  int n; //number
  while (cin >> f >> n) {
    try {
      if (f == "factorial") {
        cout << factorial(n) << endl;
      }
    }
  }
}
```

Exception Example

- Code that corrects an error goes in a `catch{}` block
- A `catch{}` block goes directly after a `try{}` block
- A `catch{}` block matches the type from a `throw` statement

Exception Example

- Code that corrects an error goes in a `catch{}` block

```
int main() {
  string f; //function
  int n; //number
  while (cin >> f >> n) {
    try {
      if (f == "factorial") {
        cout << factorial(n) << endl;
      }
    } catch(int i) {
      cout << "try again" <<endl;
    }
  }
}
```

Exception Example

```
int main() {
  string f; //function
  int n; //number
  while (cin >> f >> n) {
    try {
      if (f == "factorial") {
        cout << factorial(n) << endl;
      }
    } catch(int i) {
      cout << "try again" <<endl;
    }
  }
}
```

```
int factorial (int n) {
  if (n<0) throw n;
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

```
./a.out
factorial 5
120
factorial -5
try again
```

Exercise

- Write this function

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

```
//EFFECTS: returns n choose k,
//  throws an exception for negative input
int combination(int n, int k);
```

- Question: do you need to add a `throw` statement? Why or why not?

- Add code to `main()` to call `combination()`

Unhandled exceptions

- When an exception is not caught by a catch block, it propagates all the way to the caller of `main()`, and the program exits

```
int main() {
  // ...
  while (cin >> f >> n) {
      //no try{} block! ...
      if (f == "combination") {
        int k;
        cin >> k;
        cout << combination(n, k) << endl;
      }
      //no catch{} block! ...
   }
}
```

```
./a.out
combination -5 4
terminate called after
throwing an instance of
'int'
Aborted (core dumped)
```

Type discrimination

- A `try{}` block can have multiple `catch{}` blocks to handle different exception types

```
try {
  if (foo) throw 4;
  // some statements go here
  if (bar) throw 2.0;
  // more statements go here
  if (baz) throw ‘a’;
}
catch (int n) { }
catch (int d) { }
catch (char c) { }
catch (...) { }
```

Type discrimination

```
try {
   if (foo) throw 4;
   // some statements go here
   if (bar) throw 2.0;
   // more statements go here
   if (baz) throw 'a';
}
catch (int n) { }
catch (int d) { }
catch (char c) { }
catch (...) { }
```

The type of the thrown exception is matched, in order, to the list of catch blocks. The first matching catch block is executed

Type discrimination

```
try {
  if (foo) throw 4;
  // some statements go here
  if (bar) throw 2.0;
  // more statements go here
  if (baz) throw 'a';
}
catch (int n) { }
catch (int d) { }
catch (char c) { }
catch (...) { }
```

The last handler is a **default handler**, which matches any exception type. It can be used as a "catch-all" in case no other catch block matches

Exception types

- Code often uses custom types to describe errors
- For example:

```
class NegativeError {};

class InputError {};
```

- We use the class mechanism to declare custom types

Exception types

- When an error is detected, create a `NegativeError` object and `throw` it

```
//EFFECTS: returns n!, throws NegativeError for n<0
int factorial (int n) {
  if (n<0) throw NegativeError();
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

Exception types

- To correct an error, the `catch{}` block matches the type

```
int main() {
  //...
  while (cin >> f >> n) {
    try {
      //...
    } catch (NegativeError n) {
      cout << "try a positive number" << endl;
    } catch (...) {
      cout << "try again" <<endl;
    }

  }

}
```

```
./a.out
combination -5 4
try a positive
number
```

Exercise: What is the output?

```
class GoodbyeError {};
void goodbye() {
  cout << "goodbye!\n";
  GoodbyeError e; throw e;
  cout << "goodbye() returns\n";
}
```

```
class HelloError {};
void hello() {
  cout << "hello world!\n";
  goodbye();
  cout << "hello() returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  } catch (HelloError he) {
    cout << "HelloError\n";
  } catch (GoodbyeError ge) {
    cout << "GoodbyeError\n";
  }
  cout << "main() returns\n";
  return 0;
}
```

Exercise: What is the output?

```
class GoodbyeError {};
void goodbye() {
  cout << "goodbye!\n";
  GoodbyeError e; throw e;
  cout << "goodbye() returns\n";
}
```

```
class HelloError {};
void hello() {
  cout << "hello world!\n";
  try { goodbye(); }
  catch(GoodbyeError x)
  { throw HelloError(); }
  cout << "hello() returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  } catch (HelloError he) {
    cout << "HelloError\n";
  } catch (GoodbyeError ge) {
    cout << "GoodbyeError\n";
  }
  cout << "main() returns\n";
  return 0;
}
```

```
class Error {
  string msg;
public:
  Error(string s) : msg(s) {}
  string get_msg() { return msg;}
};
```

```
void goodbye() {
  cout << "goodbye!\n";
  throw Error("goodbye error");
  cout << "goodbye() returns\n";
}
```

```
void hello() {
  cout << "hello world!\n";
  try { goodbye(); }
  catch (Error e)
  { throw Error("hello error");}
  cout << "hello() returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  } catch (Error e) {
    cout << e.get_msg()
         << endl;
  } catch (...) {
    cout << "Unknown error"
         << endl;
  }
  cout << "main() returns\n";
  return 0;
}
```