Ve 280

Programming and Introductory Data Structures

Exception

Learning Objectives:

Understand when exceptions are useful

Understand how they work

Know how to implement them

Outline

- Exception: the Concepts
- Exception Handling in C++

Exceptions

Motivation

- We want a means of **recognizing** and **handling** unusual conditions in the program at runtime
  - E.g., the program opens a file that does not exist!

- Another example: **partial functions**.
  - A function that does not produce meaningful results for **all possible** values of its input type.
  - One particular way of preventing a partial function from receiving invalid inputs: **the REQUIRES specification**.
  - However, a REQUIRES clause is just a comment and cannot **enforce** the specification…

Exceptions

Motivation

- Instead of the REQUIRES clause, there is another way of ensuring correct inputs: **runtime checking**.
  - The idea is to check the inputs **explicitly** before using them in our program.

- One nice things about REQUIRES, is that we don't have to figure out what constitutes "bad" input.

- For runtime checking, we do… ☹

Exceptions

Determining legitimate output for illegitimate input

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

1. "It's my problem!"
   - Try to "fix" things and continue execution by "enforcing" legitimate inputs from illegitimate ones by
     - either <u>modifying the inputs</u>
     - or <u>returning default outputs</u> that make sense in the context
   - For example, `list_rest()` could return an empty list if input is an empty list.
   - Such behavior must be explained in the specification!

Exceptions

Determining legitimate output for illegitimate input

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

1. "It's my problem!"

   - However, this strategy fails whenever there is no "default" behavior for the function with the given illegal inputs.
   - For example, what is division over 0?
     - Division over 0 is simply undefined, and trying to define it changes the rules of math.

Exceptions

Determining legitimate output for illegitimate input

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

2. "I Give up!"

    - Use something like `assert()`.
    - `assert(condition)` terminates the program if condition is not true.

```
list_rest (list_t l)
// REQUIRES: list is not empty
{
     assert(!list_isEmpty(l));
}
```

Exceptions

Determining legitimate output for illegitimate input

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

2. "I Give up!"

    - However, it is Not Nice to terminate a program this way.
    - There are some situations where this type of "hard exit" is ok, but there is usually some more things to do before terminating.
        - For example, free the allocated memory.
    - Usually, exiting from a function deep in the call stack is not the way to do it.

Exceptions

Determining legitimate output for illegitimate input

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

3. "It's your problem!"
   The caller of the function

   - Encode "failure" in the **return values**.
     - Example: factorial() use 0 to encode negative input.
   - Unfortunately, you often can't encode "failure" elegantly in the return values.
   - For example, `list_first()` can return **any** integer, so no special value is available to encode "the list is empty!".
   - Compared to the other two, this is usually the strategy that you use.

Exceptions

It's your problem!

- To fully implement this strategy for runtime checking,
  - Every writer of **every function** must:
    1. Be diligent in checking for illegitimate inputs.
    2. Make sure to pass back the proper encoded "failure" return values.
  - Every writer of **every call** to one of these functions must:
    1. Be diligent in examining these returned values.
    2. Be diligent in acting on these returned values.

Exceptions

It's your problem!

- In practice, this strategy is unworkable for several reasons:

1. You get lazy.

    - You say to yourself, "This kind of error cannot **possibly** occur here, so I'll just omit this check for it."
    - Others may get lazy and not want to check for your return values.

Exceptions

It's your problem!

- In practice, this strategy is unworkable for several reasons:

2. You **forget** to check.

    - For example, if `foo` calls `bar`, `bar` calls `baz`, and `baz` returns an error; `bar` will probably notice, but `bar` has to remember to pass this to `foo`!

Exceptions

It's your problem!

- In practice, this strategy is unworkable for several reasons:

3. It gets unwieldy.

    - If you are ruthlessly diligent about it, your code becomes unmanageable.
    - You have to write too much error handling code, and it becomes hopelessly intertwined with the "normal-case" code.
    - In other words, this doesn't scale well.

- So, we need some mechanism to help deal with these runtime errors…

Exceptions

Dealing with runtime errors

- Fortunately, such a mechanism for dealing with runtime errors has been around for a long time.
- It is called an **exception handling mechanism**.
- **Exception**: something bad that happens in a block of code, such as a bad parameter that prevents the block from continuing to execute.

Exceptions

Exception Handling

- When an exception occurs, the block of the normal-case code is exited, and control is passed to another block of code (the **error handling** code).
- This error handling code then tries to correct the problem.
- In pictures:

Exceptions

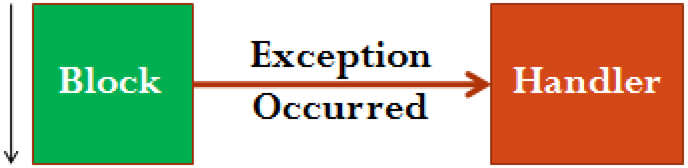
Exception Handling

- Exception handling lets us separate the normal code from the error handling code, with a conceptual "goto" between the two.

- Conceptually, normal part and error handling part are separate, but in C++, error handling part could appear in the same function as normal part.

Exceptions

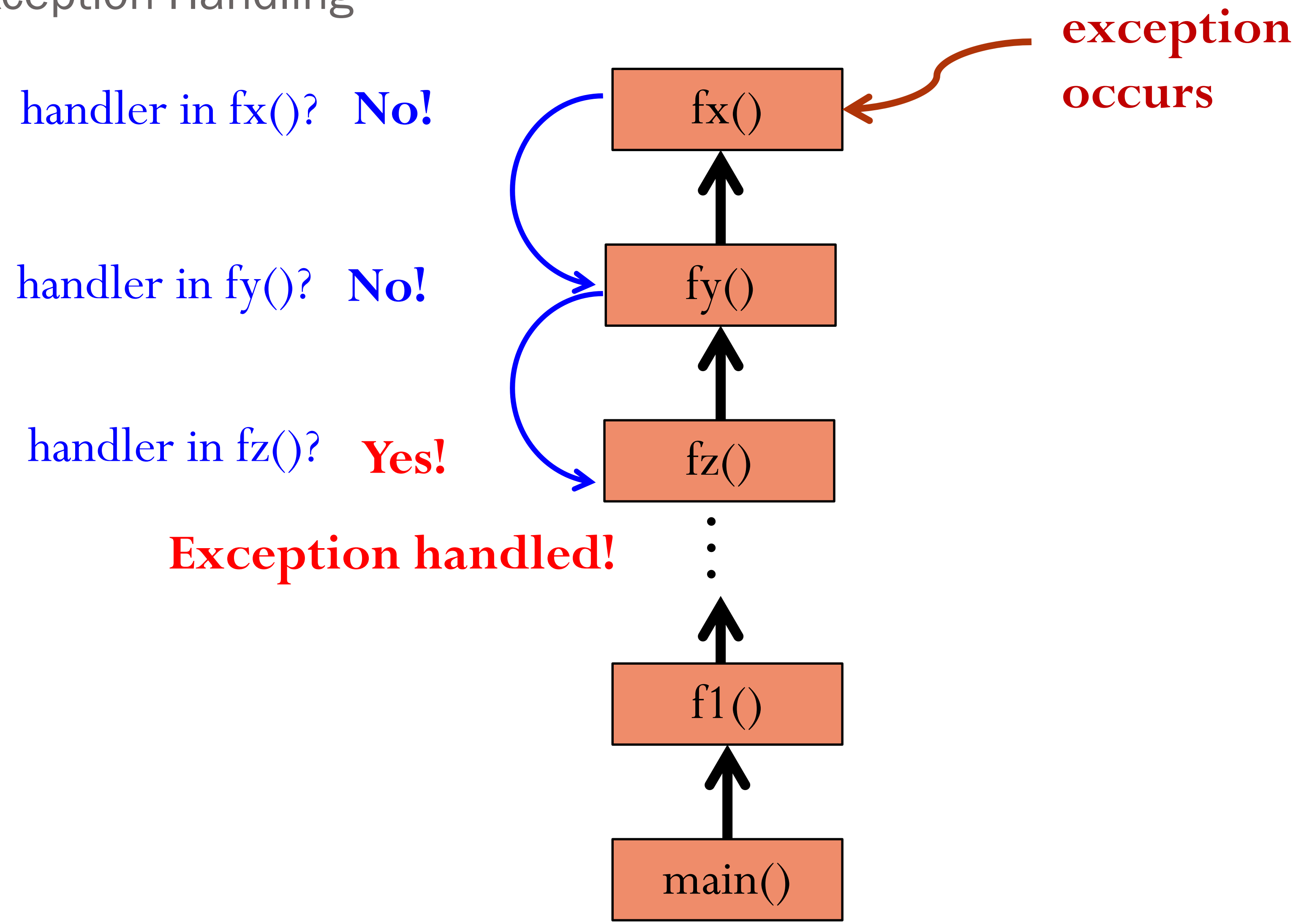
Exception Handling



- An important mechanism for exception handling is the **exception propagation**, which specifies where to find the handler.
  - First, the remaining part of the function where exception happens is searched for the handler. If found, exception is resolved.
  - If not, the **caller** of the function issuing the exception is searched for the handler. If found, done!
  - If still not, the **caller of the caller** is searched … So on and so forth.
  - In the worst case, the exception propagates up the call chain all the way to **the caller of main()**, at which point your program exits.

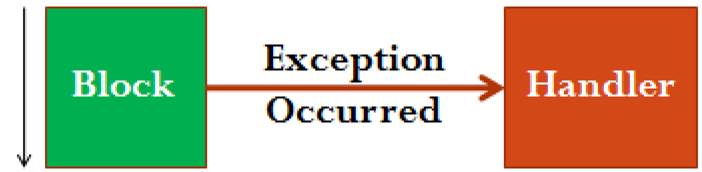
Exceptions

Exception Handling



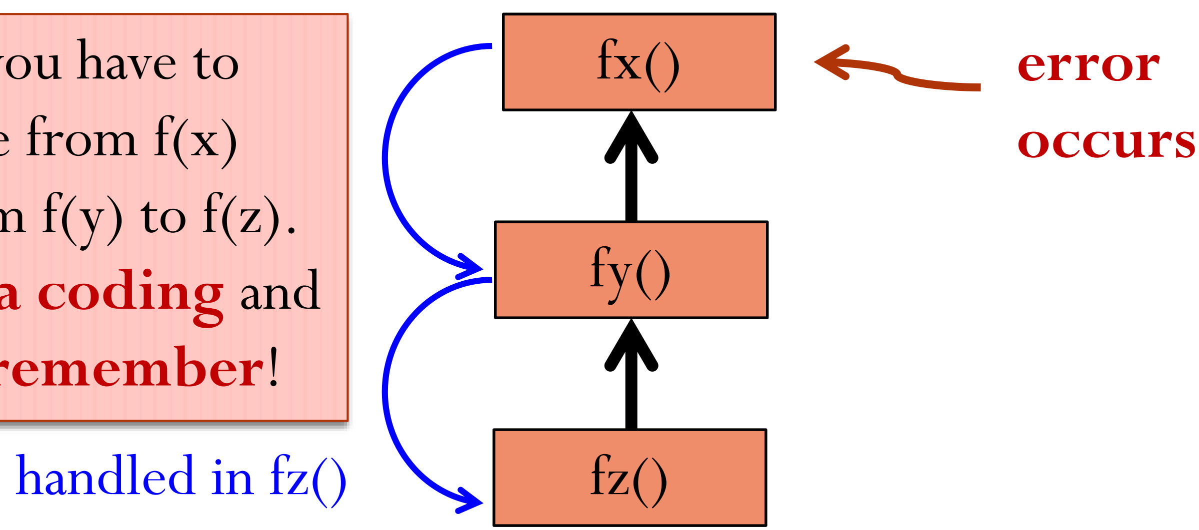
Exceptions

Exception Handling



- An exception handling mechanism is merely a neat way to **automatically** pass an exceptional condition up the call chain, until it is handled somewhere.
- This mechanism doesn't require you to propagate the error **yourself**!
  - It prevents you from having to encode things in return values.

In old method, you have to pass return value from f(x) to f(y), then from f(y) to f(z). This needs **extra coding** and requires you to **remember**!



Outline

- Exception: the Concepts
- Exception Handling in C++

Exception Handling

C++ Terminology

- **Throwing an exception:** the act of making the program aware that an exception just occurred.
- **Catching an exception:** the act of responding to the exception that occurred.
- **Exceptions occur** in a block of code called a **try block**.
- **Exceptions are handled** in a separate but related block of code called a **catch block**.
- Alternative names:
  - throwing exceptions → raising exceptions
  - catch block → exception handler

Exception Handling

C++ Terminology

- In pictures:

```
void foo() {
    try { Block }
    catch (Type var) { Handler }
}
```

Exception Handling

Usage in C++

- Exceptions have **types** and **objects** (just like variables).
- We first need to **declare** an exception type, which can either be a basic type or a user-defined type, such as a **struct** or a **class**.
- When we throw an exception, we specify an **object** of the exception type in a **throw statement**.

```
int n = -1;
if(n < 0) throw n;
// The exception type is int
// We throw an object n of int type
```

- You can think of this object as being a kind of parameter of the exception, allowing some information describing the exception to be passed to the handler.

Exception Handling

Usage in C++

- We can define an exception type ourselves, using **struct** or **class**.

- Example: `struct NegInt_t { int val; };`

- Throw an exception of `NegInt_t` type

```
if(n < 0) {
   NegInt_t error;
   error.val = n;
   throw error;
}
```

Exception Handling

Usage in C++

- For the factorial function, we'll add a check for a negative parameter, and a throw statement if it is encountered.

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

Exception Handling

Usage in C++

- Now we can call `factorial()` inside a try block, with a catch block to handle the error:

```
void foo(int i) {
  try {
    cout << factorial(i) << endl;
  }
  catch (int v) {
    cout << "Error: negative input: ";
    cout << v << endl;
  }
}
```

The catch block will catch an object of exception type `int`, and store this object in `v`.

Exception Handling

Usage in C++

```
void foo(int i) {
      try { … }
      catch (int v) { … }
}
```

- You can think of the catch block as "protecting" the try block to which it is attached.
- You cannot write a catch block unless you have a try block before it.
- On the other hand, you can throw an exception **from** anywhere, instead of just within a try or catch block.
  - See the previous `factorial()` example

Exception Handling

Usage in C++

- Exception will be **propagated** along the calling function stack. Only the **first** catch block with the **same type** as the thrown exception object will handle the exception
  - If the current function f() does not have a matching catch block, it will propagate to the caller of f()
  - If no matching catch blocks, propagate to the caller of main, and program exits

```
void foo(int i) {
  try { //throw an int }
  catch (double v) {
    // will not catch the
    // exception with int type
  }
}
```

Exception Handling

Using in C++

- If the exception is successfully handled in the catch block, execution continues normally with the first statement following the catch block.

```
void foo(int i) {
      try { … }
      catch (int v) { … }
      ... // Do something next
}
```

Next to do

Exception Handling

Usage in C++

- Now suppose `foo`'s catch block can't handle the exception. It can propagate the exception by throwing it again:

```
void foo(int i) {
  try {
    cout << factorial(i) << endl;
  }
  catch (int v) {
    cout << "Error: negative input: ";
    cout << v << endl;
    throw v;
  }
}
```

Exception Handling

Usage in C++

```
void foo(int i) {
  try {
    cout << factorial(i) << endl;
  }
  catch (int v) {
    cout << "Error: negative input: ";
    cout << v << endl;
    throw v;
  }
}
```

Here the handler explicitly propagates the exception to `foo`'s caller after printing a message to standard output.

Exception Handling

Usage in C++

- In general, a try block can have associated a catch with more than one type of exception:

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

Exception Handling

Usage in C++

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

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

Exception Handling

Usage in C++

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

Usage in C++

- Finally, we need some way of telling the caller that a function can throw an exception, so that the caller can be prepared to handle it.
- We do this via the specification comment.
- The EFFECTS clause must state it:

```
int factorial(int n);
// EFFECTS: returns n! if n>=0
//          throws int n if n<0.
```

References

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 16 Exception Handling