CS24: Introduction to Computing Systems

Spring 2015

Lecture 11

Exception Handling

- Many higher-level languages provide exception handling
- Concept:
  - One part of the program knows how to detect a problem, but not how to handle it in a general way
  - Another part of the program knows how to handle the problem, but can't detect it
- When a problem is detected, the code <u>throws</u> an exception
  - An "exception" is a value representing the error
  - Frequently, an object that contains the error's details
  - The exception's <u>type</u> indicates the category of error
- Code that knows how to handle the problem can <u>catch</u> the exception
  - Provides an exception handler that responds to the error

Java Exception Handling

- Java exception handling uses **`try`/`catch`** blocks

```
public static void main(String[] args) {
    loadConfig(args);
    try {
        double x = readInput();
        double result = computeValue(x);
        System.out.println("Result = " + result);
    }
    catch (IllegalArgumentException e) {
        printError(e.getMessage());
    }
}
```

  - If input is invalid, **`computeValue()`** throws an exception
  - Execution immediately transfers to the exception handler for **`IllegalArgumentException`**

Java Exception Handling (2)

- Only exceptions from within **`try`** block are handled!

```
public static void main(String[] args) {
    loadConfig(args);
    try {
        double x = readInput();
        double result = computeValue(x);
        System.out.println("Result = " + result);
    }
    catch (IllegalArgumentException e) {
        printError(e.getMessage());
    }
}
```

  - If **`loadConfig()`** throws, the exception isn't handled here
  - **`try`**: "If *this* code throws, I want to handle the exceptions."
    - (Assuming the exception matches one of the **`catch`** blocks…)

Java Exception Handling (3)

- Code can report an exception by **`throw`**-ing it:

```
double computeValue(double x) {
    if (x < 3.0) {
        throw new IllegalArgumentException(
            "x must be at least 3");
    }
    return Math.sqrt(x - 3.0);
}
```

- Now the function can complete in two ways:
  - Normal completion: returns the computed result
  - Abnormal termination:
    - Function stops executing immediately when **`throw`** occurs
    - Program execution jumps to the nearest enclosing **`try`**/**`catch`** block with a matching exception type

Exceptions Within A Function

- Exceptions can be used within a single function

```
static void loadConfig(String[] args) {
  try {
    for (int i = 0; i < args.length; i++) {
      if (args[i].equals("-n")) {
        i++;
        if (i == args.length)
          throw new Exception("-n requires a value");
        ...
      }
      else if ...
    }
  }
  catch (Exception e) {
    System.err.println(e.getMessage());
    showUsage(); System.exit(1);
  }
}
```

- Used to signal an error in argument-parsing code

EXCEPTIONS SPANNING FUNCTIONS

- Exceptions can also span multiple function calls
  - Doesn't have to be handled by immediate caller of function that throws!
- Example:

```
Webpage loadPage(URL url) {
  try {
    InputStream in = sendHttpRequest(url);
    ...
  }
  catch (UnknownHostException e) ...
}

InputStream sendHttpRequest(URL url) {
  Socket sock = new Socket(url.getHost(), url.getPort());
  ...
}
```

- **`Socket`** constructor could throw an exception
  - Propagates out of **`sendHttpRequest()`** function…
  - Exception is handled in **`loadPage()`** function

EXCEPTION HANDLING REQUIREMENTS

- A challenging feature to implement!
  - Can throw objects containing arbitrary information
  - Exception can stay within a single function, or propagate across multiple function calls
  - Actual **`catch`**-handler that receives the exception, depends on who called the function that threw
    - A function can be called from multiple places…
    - A thrown exception should be handled by the nearest *dynamically-enclosing* **`try`**/**`catch`** block
- Also want exception passing to be <u>fast</u>
  - Ideally, won't impose any overhead on the program until an exception is actually thrown
  - Assumption: exceptions aren't thrown very often
    - *…hence the name "exception"…*
    - (Not always a great assumption these days, but oh well!)

Implementing Exception Handling

- With exception handling, there are two important points in program execution
- When execution enters a **`try`** block:
  - Some exceptions might be handled by this **`try`**/**`catch`** block…
  - May need to do some kind of bookkeeping so we know where to jump back to in case an exception is thrown
- When an exception is actually thrown:
  - Need to jump to the appropriate **`catch`** block
  - Need to access information from previous **`try`**-point, so that we can examine the proper set of catch blocks
  - This will frequently span multiple stack frames

Exceptions Within A Function

- When exception is thrown and caught within a single function:

```
void foo() {
    try {
        ...
        if (failed)
            throw new Exception();
        ...
    }
    catch (Exception e) {
        ...  // Handle the exception
    }
}
```

- In this case, can translate **`throw`** into a simple jump to the appropriate exception handler
  - Types are available at compile time

Exceptions Within a Function (2)

- Still need some way to pass exception object to the handler…

```
void foo() {
    try {
        ...
        if (failed)
            throw new Exception();
        ...
    }
    catch (Exception e) {
        ...  // Handle the exception
    }
}
```

- Assume there will be at most one exception in flight at any given time
- Store [reference to] the exception in a global variable

Exceptions Within A Function (3)

- One possible translation of our code:

```
void foo() {
    ...         // Code that sets up failed flag
    if (failed) {
        set_exception(new Exception());  // throw
        goto foo_catch_Exception;
    }
    ...         // Other code within try block
foo_end_try:  // End of try-block
    goto foo_end_trycatch;

foo_catch_Exception: {
        e = get_exception();
        ...  // Handle the exception
        goto foo_end_trycatch;
    }

foo_end_trycatch:
    return;
}
```

Exceptions Spanning Functions

- Not a good general solution! Normal case is to have exceptions spanning multiple function calls.
- Can't implement with **`goto`**, since **`goto`** can't span multiple functions!
  - Really can't hard-code the jump now, anyway…
- Really want a way to *record* where to jump, dynamically
  - i.e. when we enter **`try`** block
- Then, a way to jump back to that place, even across multiple function calls
  - i.e. when exception is thrown

```
int f(int x) {
  try {
    return g(3 * x);
  } catch (Exception e) {
    return -1;
  }
}

int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    throw new Exception();

  return Math.sqrt(x - 5);
}
```

`setjmp()` AND `longjmp()`

- C standard includes two very interesting functions:
- **`int setjmp(jmp_buf env)`**
  - Records current execution state into **`env`**, at exact time of **`setjmp()`** call
    - Information recorded includes callee-save registers, **`esp`**, and caller return-address
  - Always returns 0
- **`void longjmp(jmp_buf env, int val)`**
  - Restores execution state from **`env`**, back into all registers saved by **`setjmp()`**
  - **`esp`** is restored from **`env`**:
    - Any intervening stack frames are discarded
  - Stack is restored to the state as when **`setjmp()`** was called
    - Caller return-address on stack when **`setjmp()`** was called
  - Then **`longjmp()`** returns, with **`val`** in **`%eax`**
    - (or **`%eax`** = 1 if **`val`** is 0)
- To caller, it appears that **`setjmp()`** returned again!

setjmp() AND longjmp() (2)

- Previous example is simple enough to implement using **setjmp()** and **longjmp()**:

```
int f(int x) {
  try {
    return g(3 * x);
  } catch (Exception e) {
    return -1;
  }
}

int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    throw new Exception();

  return Math.sqrt(x - 5);
}
```

```
static jmp_buf env;

int f(int x) {
  if (setjmp(env) == 0)
    return g(3 * x);
  else
    return -1;
}

int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    longjmp(env, 1);

  return sqrtl(x - 5);
}
```

`setjmp()` AND `longjmp()` (3)

- When we enter **`try`** block, record execution state in case an exception is thrown
- If an exception is thrown, use **`longjmp()`** to return to where **`try`** was entered
  - *Stack frames of intervening function calls are discarded!*
- Return value of **`setjmp()`** indicates whether an exception was thrown
  - Example has only one kind of exception, so any return-value will do

```
static jmp_buf env;

int f(int x) {
  if (setjmp(env) == 0)
    return g(3 * x);
  else
    return -1;
}


int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    longjmp(env, 1);

  return sqrtl(x - 5);
}
```

setjmp()/longjmp() Example

- What happens with **f(5)** ?
- Things will go badly… ☺

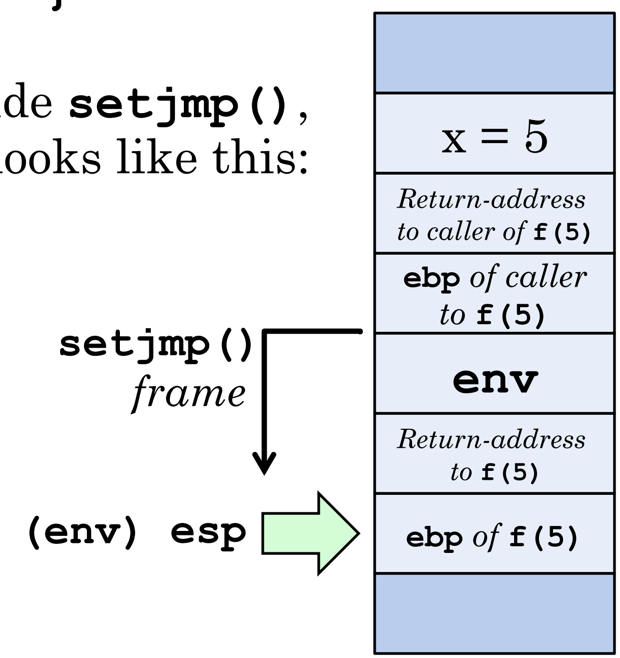
```
static jmp_buf env;

int f(int x) {
  if (setjmp(env) == 0)
    return g(3 * x);
  else
    return -1;
}

int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    longjmp(env, 1);

  return sqrtl(x - 5);
}
```

`setjmp()`/`longjmp()` EXAMPLE (2)

- **`f(5)`** calls **`setjmp()`** to prepare for an exception
  - Will return 0 since it's actually the **`setjmp()`** call
- **`setjmp()`** stores:
  - Callee-save registers, including current **`esp`**
  - **`setjmp()`**-caller's **`ebp`** and return address
    - (grab these from stack)
- **`env`** now holds everything necessary for **`longjmp()`** to act like it's **`setjmp()`**…

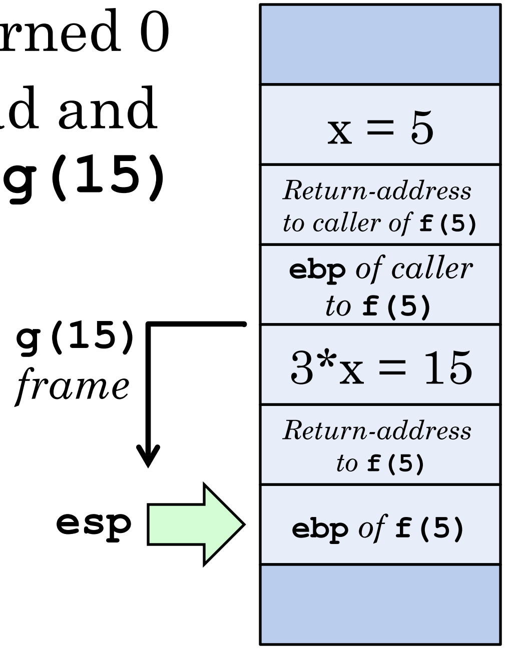
```
static jmp_buf env;

int f(int x) {
  if (setjmp(env) == 0)
    return g(3 * x);
  else
    return -1;
}
```

Inside **`setjmp()`**, stack looks like this:



`setjmp()/longjmp()` EXAMPLE (3)

- **`setjmp()`** returned 0
- **`f(5)`** goes ahead and calls **`g(3*x)`** = **`g(15)`**
- Now the stack looks like this:

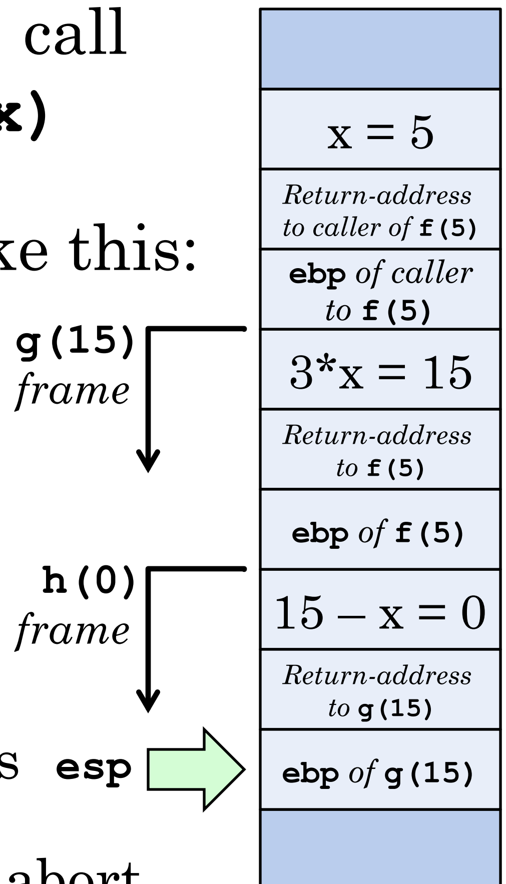



```
static jmp_buf env;

int f(int x) {
  if (setjmp(env) == 0)
    return g(3 * x);
  else
    return -1;
}

int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    longjmp(env, 1);

  return sqrtl(x - 5);
}
```

- Note that the stack frame from calling **`setjmp()`** is long gone…
  - (along with the return-address to where **`setjmp()`** was called from)
  - **`env`** still contains these values!

setjmp()/longjmp() Example (4)

- Now in **g(15)** call
- **g** calls **h(15-x)** = **h(0)**
- Stack looks like this:



- Problem:
  - **h()** can't handle values less than 5
  - **h()** needs to abort the computation

```c
static jmp_buf env;

int f(int x) {
  if (setjmp(env) == 0)
    return g(3 * x);
  else
    return -1;
}

int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    longjmp(env, 1);

  return sqrtl(x - 5);
}
```

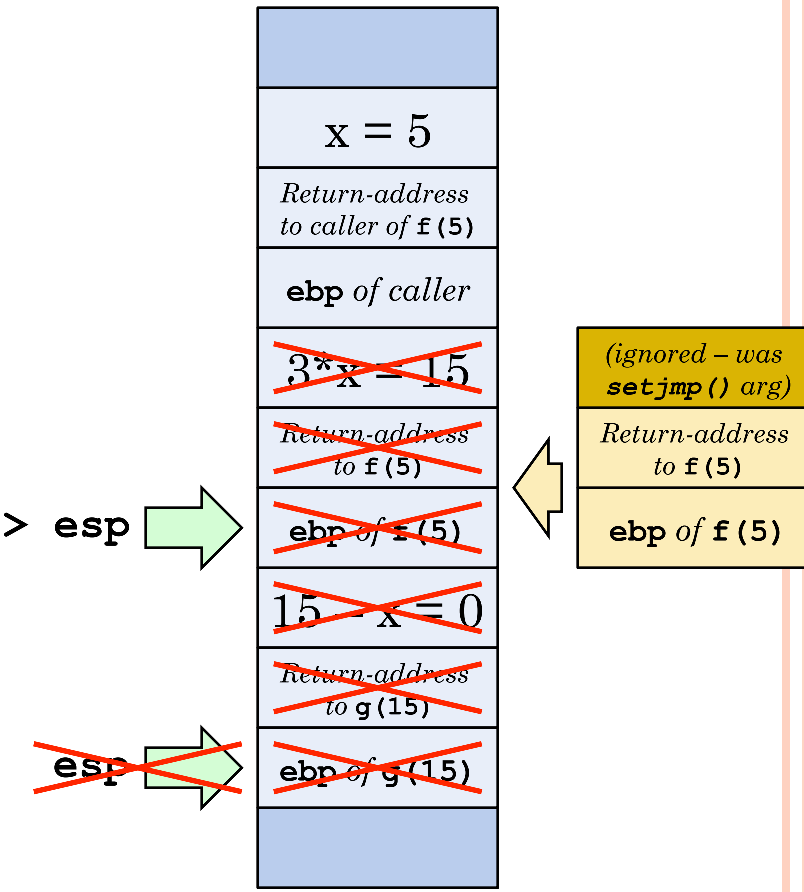
`setjmp()/longjmp()` Example (5)

- `h(0)` needs to abort!
  - Got a bad argument…
- `h()` "throws an exception"
  - Calls `longjmp()` to switch back to nearest enclosing `try`-block
  - `env` contains details of where nearest enclosing `try`-block is…
- `longjmp()` restores execution state back to execution in `f()`
- `f()` "catches the exception"
  - It now sees `setjmp()` return a nonzero result, indicating there was an exception…
  - `f()` returns -1 as final result

```
static jmp_buf env;

int f(int x) {
  if (setjmp(env) == 0)
    return g(3 * x);
  else
    return -1;
}

int g(int x) {
  return h(15 - x);
}

int h(int x) {
  if (x < 5)
    longjmp(env, 1);

  return sqrtl(x - 5);
}
```

`setjmp()/longjmp()` Example (6)

```
int f(int x) {
  if (setjmp(env) == 0)
    ...
}
...
int h(int x) {
  if (x < 5)
    longjmp(env, 1);
    ...
}
```



- When **`h()`** calls **`longjmp()`**, **`esp`** and caller **`eip`**/**`ebp`** are restored from **`env`**
- When **`longjmp()`** returns, execution resumes in **`f()`**, "back at **`setjmp()`**"
  - Caller has no idea who returned back!
- Result of **`setjmp()`** indicates error
  - *(but it's technically **`longjmp()`**'s result…)*
- **`f()`** handles the error appropriately

How do these things work?!

- **`setjmp()`** and **`longjmp()`** <u>must</u> be implemented in assembly language
  - No C mechanism for saving and restoring registers
  - No C mechanism for manipulating the stack this way
- Implementation is also very platform-specific!
  - Size of **`jmp_buf`** corresponds to how many registers need to be saved and restored
  - Linux on IA32 uses 8 dwords
  - MacOS X on IA32 uses 18 dwords (!)
  - MacOS X on PPC uses 192 dwords (!!!)
    - RISC processors tend to have a large number of registers, due to load/store architecture
  - Specification is ambiguous about exactly what needs to be saved…

How do these things work?! (2)

- Implementing **`setjmp()`/`longjmp()`** is surprisingly straightforward
  - Simply requires understanding of stack frames in cdecl
- In **`setjmp()`**, must know how to save the caller's execution state, to fake a return from **`setjmp()`**
  - Return-address where caller invoked **`setjmp()`** from
  - **`esp`** value inside **`setjmp()`**
  - (also the callee-save registers, since they will change before **`longjmp()`** is called…)
- In **`longjmp()`**, just need to manipulate the stack to restore this execution state, then **`ret`** !
  - Caller will see return-value in **`eax`** like usual
  - Returns to where caller invoked **`setjmp()`** from
  - *They'll never know the difference!*

Multiple Catch Blocks

- A **`try`** block can have multiple **`catch`** blocks

```
Webpage loadPage(String urlText) {
  try {
    Socket s = httpConnect(urlText);
    ...
  }
  catch (MalformedURLException e) {
    ...
  }
  catch (UnknownHostException e) {
    ...
  }
}
```

- Easy to support this with **`setjmp()`** and **`longjmp()`**
  - **`longjmp()`** can simply pass a different integer value for each kind of exception
  - Compiler can assign integer values to all exception types

Multiple Catch Blocks (2)

- One possible translation:

```
jmp_buf env;
...
  switch (setjmp(env)) {
  case 0:    /* Normal execution */
    ...      // Translation of
    ...      //   Socket s = httpConnect(urlText);
    break;

  case 1037: /* Caught MalformedURLException */
    ...
    break;
  case 1053: /* Caught UnknownHostException */
    ...
    break;
  }
```

- Code that calls **`longjmp()`** passes exception-type in call
- *Many details left out, involving variable scoping, etc.*

Nested Exception Handlers

- One major flaw in our implementation:
  - **`try/catch`** blocks can be nested within each other!
  - We only have one **`jmp_buf`** variable in our example
  - A nested **`try/catch`** block would overwrite the outer **`try`**-block's values stored in the **`jmp_buf`**
- Solution is straightforward:
  - Introduce a "try-stack" for managing the **`jmp_buf`** values of nested **`try/catch`** blocks
  - When we enter into a new **`try`**-block, push the new **`try/catch`** handler state (**`jmp_buf`**) onto try-stack
  - This is separate from the regular program stack
  - (It doesn't strictly *have* to be separate, but to keep things simple, we will keep it separate!)

Nested Exception Handlers (2)

- Once we have a try-stack for nested handlers, need some basic exception handling operations
- When program enters a **`try`** block:
  - Call **`setjmp()`**, and if it returns 0 then push the **`jmp_buf`** onto the **`try`**-stack
  - **Note:** Cannot call **`setjmp()`** in a separate helper function that does these things for us! ***Why not?***
    - Left as an exercise for the student…
- When program exits the **`try`**-block without any exceptions:
  - Need to pop the topmost **`jmp_buf`** off of the try-stack
  - Can do this in a helper function

Nested Exception Handlers (3)

- When an exception is thrown:

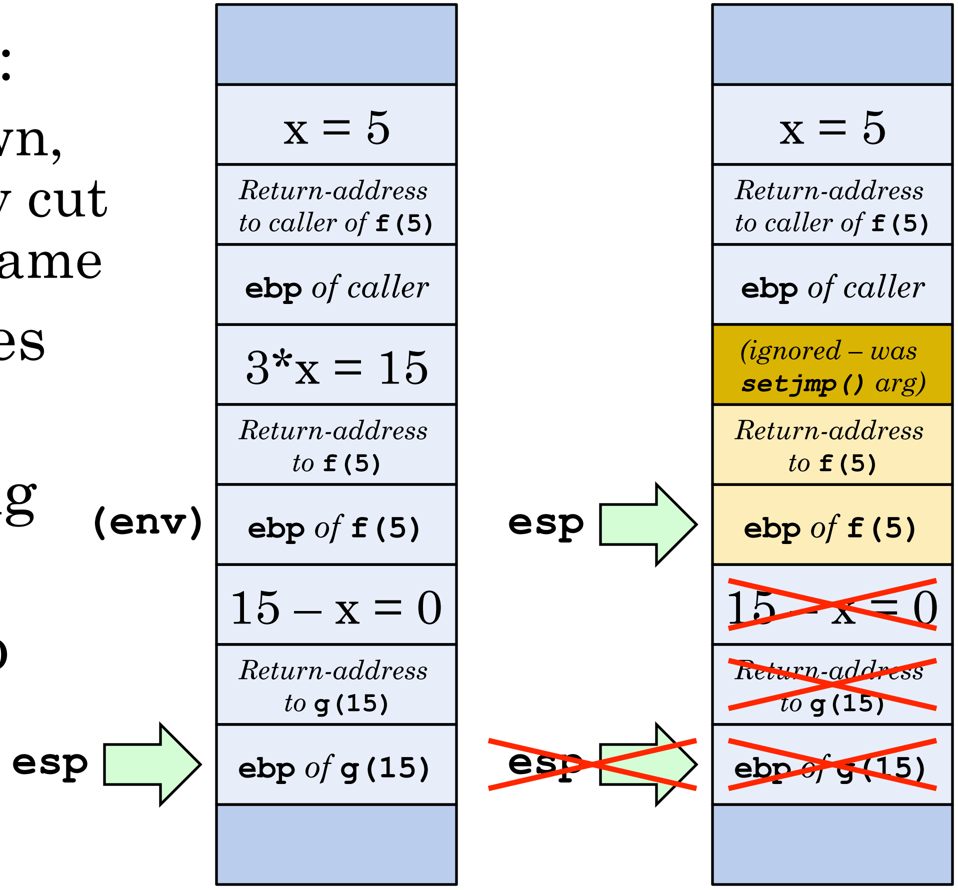
  `void throw_exception(int exception_id)`

  - Helper function that pops the topmost **`jmp_buf`** off of the try-stack, and then uses it to do **`longjmp(exception_id)`**
- If an exception isn't handled by a **`try`**/**`catch`** block, or if **`catch`**-block re-throws the exception:
  - Just invoke **`throw_exception()`** again with same ID
  - Next enclosing **`try`**-block's **`jmp_buf`** is now on top of stack
  - (Do this in **`default`** branch, or **`else`**-clause if using **`if`**.)

- With these tools in place, can easily handle nested exception-passing scenarios
- An example of this is provided in Assignment 4! ☺

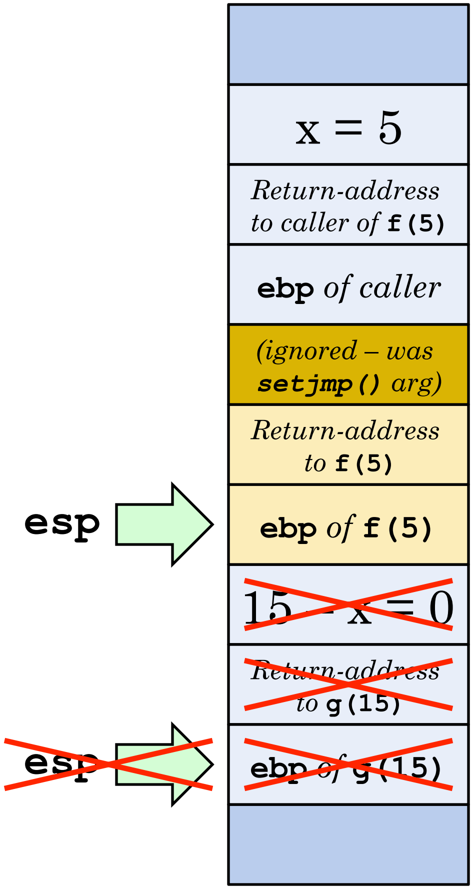
STACK CUTTING

- This kind of exception-handling implementation is called <u>stack cutting</u>
- From previous example:
  - When exception is thrown, the stack is immediately cut down to the handler's frame
- Intervening stack frames are simply eliminated!
- *Very fast* for propagating exceptions…
- Unacceptable if cleanup needs to be done for intervening functions!



STACK CUTTING (2)

- Can perform cleanup for intervening functions, if we keep track of what needs to be done
  - e.g. manage a list of resources that need cleaned up after each function returns
  - When exception is thrown, can use this info to clean up properly
  - Starting to get a bit *too* complicated…
- For languages with GC, don't really have much to clean up from functions
  - Just drop object-references from stack
  - Garbage collector will detect that objects are no longer reachable, and will eventually reclaim the space



Stack Unwinding

- Another solution to the exception-propagation problem: <u>stack unwinding</u>
  - Solution used by Java Virtual Machine, most C++ implementations, etc.
  - Unlike stack cutting, each stack frame is cleaned up individually. Much better for resource management!
- Remember, the important times in exception handling are:
  - When we are inside of a **`try`**-block – a thrown exception might be handled by this **`try`**/**`catch`**
  - When an exception is actually thrown
- The compiler generates an <u>exception table</u> for every single function in the program
  - All exception handling is driven from these tables

Exception Tables

- Each function has an exception table, containing:
  - A range of addresses [*from_eip*, *to_eip*], specifying the instructions the **`try`**-block encapsulates
  - An exception that the **`try`**-block can handle
  - The address of the handler for that exception
- Our example from before:

```
Webpage loadPage(String urlText) {
  try {
    Socket s = httpConnect(urlText);
    ...
  }
  catch (MalformedURLException e) {
    ...
  }
  catch (UnknownHostException e) {
    ...
  }
}
```

Exception table for **`loadPage()`**

from_eip	to_eip	exception	handler_eip
0x3019	0x315C	malformed_url	0x316B
0x3019	0x315C	unknown_host	0x3188

Exception Tables (2)

- When an exception is thrown within a function:
- Two important pieces of information!
  - What is the current program-counter?
  - What is the type of the exception that was thrown?
- Exception table for the current function is searched

Exception table for **`loadPage()`**

from_eip	to_eip	exception	handler_eip
0x3019	0x315C	malformed_url	0x316B
0x3019	0x315C	unknown_host	0x3188

  - Try to find a row where the program-counter is in the specified range, also having the same exception type
  - If found, dispatch to the specified exception handler
- If no matching row is found, the current stack frame is cleaned up, and process repeats in parent frame

Nested Try/Catch Example

- Code with nested **`try/catch`** blocks
- Compiler generates an exception table for each function:

Exception table for `f(x)`

from_eip	to_eip	exception	handler_eip
0x2005	0x203B	a	0x2043
0x2005	0x203B	b	0x204C

Exception table for `g(x)`

from_eip	to_eip	exception	handler_eip
0x2116	0x214A	b	0x2159
0x2116	0x214A	c	0x215E

Exception table for `h(x)`

from_eip	to_eip	exception	handler_eip

```
int f(int x) {
  try {
    return g(x * 3);
  } catch (A a) {
    return -5;
  } catch (B b) {
    return -10;
  }
}

int g(int x) {
  try {
    return h(8 - x);
  } catch (B b) {
    return -15;
  } catch (C c) {
    return -20;
  }
}

int h(int x) {
  if (x > 23)
    throw new A();
  else if (x < -15)
    throw new B();

  return x - 1;
}
```

Nested Try/Catch (2)

- Call `f(-9)`
- `f(-9)` calls `g(-9 * 3)` = `g(-27)`
- `g(-27)` calls `h(8 - -27)` = `h(35)`

- **Important point:**
  - So far, no overhead for entering **`try`**-blocks, or any other aspect of exception handling!

- But, we know that `h(35)` is going to throw…

```
int f(int x) {
  try {
    return g(x * 3);
  } catch (A a) {
    return -5;
  } catch (B b) {
    return -10;
  }
}

int g(int x) {
  try {
    return h(8 - x);
  } catch (B b) {
    return -15;
  } catch (C c) {
    return -20;
  }
}

int h(int x) {
  if (x > 23)
    throw new A();
  else if (x < -15)
    throw new B();

  return x - 1;
}
```

Nested Try/Catch (3)

- **`h(35)`** throws exception **`A`**
- Use our exception tables to direct the exception propagation

- **`h(35)`** throws **`A`**. **`eip`** = **`0x226C`**.
  - Check exception table for **`h`**:

Exception table for **`h(x)`**

from_eip	to_eip	exception	handler_eip

  - Nothing matches. *(duh…)*
  - Clean up local stack frame, then return to caller of **`h`**

```
int f(int x) {
  try {
    return g(x * 3);
  } catch (A a) {
    return -5;
  } catch (B b) {
    return -10;
  }
}

int g(int x) {
  try {
    return h(8 - x);
  } catch (B b) {
    return -15;
  } catch (C c) {
    return -20;
  }
}

int h(int x) {
  if (x > 23)
    throw new A();
  else if (x < -15)
    throw new B();

  return x - 1;
}
```

NESTED TRY/CATCH (4)

- Now inside of **`g(-27)`**
- **`g(-27)`** throws **`A`**. **`eip`** = **`0x2123`**.
  - Check exception table for **`g`**:

Exception table for **`g(x)`**

from_eip	to_eip	exception	handler_eip
0x2116	0x214A	b	0x2159
0x2116	0x214A	c	0x215E

  - **`g`** does have entries in its table, but none match combination of exception **`A`** and **`eip`** = **`0x2123`**.
  - Again, clean up local stack frame, then return to caller of **`g`**

```
int f(int x) {
  try {
    return g(x * 3);
  } catch (A a) {
    return -5;
  } catch (B b) {
    return -10;
  }
}

int g(int x) {
  try {
    return h(8 - x);
  } catch (B b) {
    return -15;
  } catch (C c) {
    return -20;
  }
}

int h(int x) {
  if (x > 23)
    throw new A();
  else if (x < -15)
    throw new B();

  return x - 1;
}
```

Nested Try/Catch (5)

- Finally, back to **`f(-9)`**
- **`f(-9)`** throws **`A`**. **`eip`** = **`0x201B`**.
  - Check exception table for **`f`**:

Exception table for **`f(x)`**

from_eip	to_eip	exception	handler_eip
0x2005	0x203B	a	0x2043
0x2005	0x203B	b	0x204C

  - **`f`** also has exception table entries, and the first entry matches our combination of exception type and instruction-pointer value!
  - Dispatch to specified handler:
    - **`return -5;`**
  - Exception propagation is complete.

```
int f(int x) {
  try {
    return g(x * 3);
  } catch (A a) {
    return -5;
  } catch (B b) {
    return -10;
  }
}

int g(int x) {
  try {
    return h(8 - x);
  } catch (B b) {
    return -15;
  } catch (C c) {
    return -20;
  }
}

int h(int x) {
  if (x > 23)
    throw new A();
  else if (x < -15)
    throw new B();

  return x - 1;
}
```

Comparison of Methodologies

- Stack cutting approach is optimized for the exception-handling phase
  - Transfers control to handler code in one step
    - (Presuming resources don't need to be cleaned up from intervening function calls…)
- Additional costs in the normal execution paths!
  - Need to record execution state every time a **`try`**-block is entered
  - Need to push and pop these state records, too!
- These costs will *quickly* add up in situations where execution passes through many **`try`**-blocks

Comparison of Methodologies (2)

- Stack unwinding approach is optimized for the normal execution phase
  - No exception-handler bookkeeping is needed at run-time…
  - …because all bookkeeping is done at compile-time!
- Additional costs in the exception-handling paths
  - Each function call on stack is dealt with individually
  - Must search through each function's exception table, performing several comparisons per record
- If a program *frequently* throws exceptions, especially from deep within call-sequences, this will definitely add up

Comparison of Methodologies (3)

- Typical assumption is that exception handling is a relatively uncommon occurrence
  - *(That's why we call them exceptions!!!)*
  - Additionally, most languages have resources to clean up, within each function's stack frame
  - e.g. even though Java has garbage collection, it also has **`synchronized`** blocks; monitors need unlocked
- Most common implementation: stack unwinding
- Many other optimizations are applied to exception handling as well!
  - Dramatically reduce or even eliminate overhead of searching for exception handlers within each function