Review

Last week

• Generic Programming

Outline

• Exceptions

Sources for today's lecture:

PLP, 8.5

Exceptions

General mechanism for handling abnormal conditions

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>How raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>predefined</td>
<td>constraint violations, I/O errors, communication errors, other illegalities</td>
<td>by the runtime system</td>
</tr>
<tr>
<td>user-defined</td>
<td>pop from empty stack</td>
<td>explicitly by user code</td>
</tr>
</tbody>
</table>

• exception handlers specify remedial actions or proper shutdown

• exceptions can be stored and re-raised later
One way to improve robustness of programs is to write code to explicitly handle errors.

How can we do this?

Traditionally, this was done by checking the result of each operation that can go wrong (e.g., popping from a stack, writing to a file, allocating memory).

Unfortunately, this has a couple of serious disadvantages:

1. it is easy to forget to check
2. writing all the checks clutters up the code and obfuscates the common case (the one where no errors occur)

Exceptions let us write clearer code and make it easier to catch errors.

Predefined exceptions in Ada

- Defined in Standard:
  - Constraint_Error: value out of range
  - Program_Error: illegality not detectable at compile-time: unelaborated package, exception during finalization, etc.
  - Storage_Error: allocation cannot be satisfied (heap or stack)
  - Tasking_Error: communication failure

- Defined in Ada.IO_Exceptions:
  - Data_Error, End_Error, Name_Error, Use_Error, Mode_Error, Status_Error, Device_Error
Handling exceptions

Any begin-end block can have an exception handler:

```plaintext
procedure Test is
  X: Integer := 25;
  Y: Integer := 0;
begin
  X := X / Y;
exception
  when Constraint_Error =>
    Put_Line("did you divide by 0?");
  when others =>
    Put_Line("out of the blue!");
end;
```

A common idiom

```plaintext
function Get_Data return Integer is
  X: Integer;
begin
  loop
    begin
      Get(X);
      return X;  -- if got here, input is valid, -- so leave loop
    exception
      when others =>
        Put_Line("input must be integer, try again");
        -- will restart loop to wait for a good input
    end;
  end loop;
end;
```

User-defined Exceptions

```plaintext
package Stacks is
  Stack_Empty: exception;
  ...
end Stacks;
```

package body Stacks is

```plaintext
procedure Pop (X: out Integer;
  From: in out Stack) is
begin
  if Empty(From)
    then raise Stack_Empty;
  else ...
end Pop;
...
end Stacks;
```

The scope of exceptions

- an exception has the same visibility as other declared entities: to handle an exception it must be visible in the handler (e.g., caller must be able to see Stack_Empty).
- an others clause can handle unnamable exceptions partially

```plaintext
when others =>
  Put_Line("disaster somewhere");
  raise;  -- propagate exception,
          -- program will terminate
```
Exception run-time model

What happens when an exception is raised?

1. When an exception is raised, the current sequence of statements is abandoned (e.g., current `Get` and `return` in example).

2. Starting at the current frame, each frame in the current dynamic scope is examined (want dynamic as opposed to static scopes because those are values that caused the problem).

3. As each frame is examined, if a handler is found, it is executed, and program execution resumes in that frame. Otherwise, the frame is discarded.

4. If no handler is found, the program terminates.

Note: A discarded frame (including the frame that raised the exception) is never resumed.

Exception information

- an Ada exception is a label, not a type: we cannot declare exception variables and assign to them
- but an exception occurrence is a value that can be stored and examined
- an exception occurrence may include additional information: source location of occurrence, contents of stack, etc.
- predefined package Ada.Exceptions contains needed machinery

Ada.Exceptions (part of std libraries)

```ada
package Ada.Exceptions is
  type Exception_Id is private;
  type Exception_Occurrence is limited private;

  function Exception_Identity (X: Exception_Occurrence) return Exception_Id;
  function Exception_Name (X: Exception_Occurrence) return String;

  procedure Save_Occurrence  
    (Target: out Exception_Occurrence;
     Source: Exception_Occurrence);
  procedure Raise_Exception (E: Exception_Id;
                             Message: in String := "");

end Ada.Exceptions;
```

Using exception information

```ada
begin
  ...
  exception
    when Expected: Constraint_Error =>
      -- Expected has details
      Save_Occurrence(Event_Log, Expected);

    when Trouble: others =>
      Put_Line("unexpected" &
               Exception_Name(Trouble) &
               " raised");
      Put_Line("shutting down");
      raise;
  end;
```

Exceptions in C++

- similar runtime model,...
- but exceptions are bona-fide types,
- and exception occurrences are first-class values
- handlers appear in `try/catch` blocks

```cpp
try {
    some_complex_calculation();
} catch (const RangeError& e) {
    // RangeError might be raised
    // in some_complex_calculation
    cerr << "oops\n";
} catch (const ZeroDivide& e) {
    // same for ZeroDivide
    cerr << "why is denominator zero?\n";
}
```

Exceptions and inheritance

A handler names a class, and can handle an object of a derived class as well:

```cpp
class Matherr { }; // a bare object, no info
class Overflow : public Matherr {...};
class Underflow : public Matherr {...};
class ZeroDivide : public Matherr {...};

try {
    weatherPredictionModel(...);
} catch (const Overflow& e) {
    // e.g., change parameters in caller
} catch (const Matherr& e) {
    // Underflow, ZeroDivide handled here
} catch (...) {
    // handle anything else (ellipsis)
}
```

Exceptions in JAVA

- Model and terminology similar to C++:
  - exceptions are objects that are thrown and caught
  - `try` blocks have handlers, which are examined in succession
  - a handler for an exception can handle any object of a derived class
- Differences:
  - all exceptions are extensions of predefined class `Throwable`
  - checked exceptions are part of method declaration
  - the `finally` clause specifies clean-up actions (in C++, cleanup actions are idiomatically done in destructors)

```java
struct ZeroDivide {
    int lineno;
    ZeroDivide (...) { ... } // constructor
    ...
};
...
if (x == 0)
    throw ZeroDivide(...); // call constructor // and go
```

Defining and throwing exceptions

The program throws an object. There is nothing in the declaration of the type to indicate it will be used as an exception.

```java
struct ZeroDivide {
    int lineno;
    ZeroDivide (...) { ... } // constructor
    ...
};
...
```
Exception class hierarchy

- any class extending `Exception` is a **checked** exception
- system errors are extensions of `Error`; these are **unchecked** exceptions

Checked exceptions must be either handled or declared in the method that throws them; this is checked by the compiler.

### Exceptions in Java

If a method might throw an exception, callers should know about it

```java
public void replace (String name, Object newValue) throws NoSuch
{
    Attribute attr = find(name);
    if (attr == null) throw new NoSuch(name);
    newValue.update(attr);
}
```

### Mandatory cleanup actions

Some cleanups must be performed whether the method terminates normally or throws an exception.

```java
public void parse (String file) throws IOException {
    BufferedReader input =
        new BufferedReader(new FileReader(file));
    try {
        while (true) {
            String s = input.readLine();
            if (s == null) break;
            parseLine(s); // may fail somewhere
        }
    } finally {
        if (input != null) input.close();
    } //regardless of how we exit
}
```

### Exceptions in ML

- runtime model similar to Ada/C++/Java
- `exception` is a single type (like a `datatype` but dynamically extensible)
- declaring new sorts of exceptions:
  ```ml
  exception StackUnderflow
  exception ParseError of { line: int, col: int }
  ```

- raising an exception:
  ```ml
  raise StackUnderflow
  raise (ParseError { line = 5, col = 12 })
  ```
**Exceptions in ML**

- handling an exception:

  \[
  \text{expr}_1 \ \text{handle} \ \text{pattern} \Rightarrow \text{expr}_2
  \]

  If an exception is raised during evaluation of \(\text{expr}_1\), and \text{pattern} matches that exception, \(\text{expr}_2\) is evaluated instead.

**A closer look**

exception DivideByZero

fun f i j =
  if j <> 0
    then i div j
  else raise DivideByZero

(f 6 2
 handle DivideByZero => 42) (* evaluates to 3 *)

(f 4 0
 handle DivideByZero => 42) (* evaluates to 42 *)

Typing issues:

- the type of the body and the handler must be the same
- the type of a \text{raise} expression can be \textbf{any type}
  (whatever type is appropriate is chosen)

**Call-with-current-continuation**

Available in Scheme and SML/NJ; usually abbreviated to \text{call/cc}.

A \textit{continuation} represents the computation of “rest of the program”.

\text{call/cc} takes a function as an argument. It calls that function with the current continuation (which is packaged up as a function) as an argument.

If this continuation is called with some value as an argument, the effect is as if \text{call/cc} had itself returned with that argument as its result.

The current continuation is the “rest of the program”, starting from the point when \text{call/cc} returns.

\[
\begin{align*}
\text{(call/cc (lambda (c) (c 5)))} & \quad ; \text{returns 5} \\
\text{(call/cc (lambda (c) 5))} & \quad ; \text{so does this} \\
\text{(call/cc (lambda (c) (+ 1 (c 5))))} & \quad ; \text{ditto}
\end{align*}
\]

**The power of continuations**

We can implement many control structures with \text{call/cc}:

- \textbf{return}:
  \[
  \begin{align*}
  (\lambda (x) \\
  \text{(call/cc (lambda (ret) \\
  \text{... \quad ; body of function \\
  \text{(ret 76) \quad ; call continuation with result \\
  \text{... \quad ; ditto)}))})
  \end{align*}
  \]

- \textbf{goto}:
  \[
  \begin{align*}
  \text{(begin \\
  \text{... \\
  \text{(call/cc (lambda (k) (set! here k)) \quad ; set label \\
  \text{... \\
  \text{(here ()) \quad ; \text{’goto’ here \\
  \text{... \\
  \end{align*}
  \]
Exceptions via call/cc

Exceptions can also be implemented by call/cc:

- Need global stack: handlers
- For each try/catch:
  (call/cc (lambda (k)
    (begin
      (push handlers (lambda ()
        (begin
          (pop handlers)
          (catch-block)
          (k ()))))
      (try-block)
      (pop handlers)))))
- For each raise:
  ((top handlers)) ; call the top function on
  ; the handlers stack