Subroutines and Control Abstraction

COMS W4115

Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science

Exceptions

Exceptions

How to handle an unexpected or unusual condition such as divide-by-zero or out-of-memory?

1. Return a usable value

   Not always the right thing to do

2. Return a different “status” value and always check this

   Tedious, and easy to accidentally omit

   Unix system calls use this

   Lots of overhead

3. Pass a closure for the error-handler

   Clutters, can add overhead

Returning a Status Value

Example: The Unix **open()** call:

> Upon successful completion, the open() function opens the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, −1 is returned, **errno** is set to indicate the error, and no files are created or modified.

Passing a Closure

Closure: Place to send control (instruction label) + environment (stack, registers, etc.)

Example: C’s setjmp/longjmp mechanism

A way to return from deeply nested functions.

A hack now part of the standard library

setjmp/longjmp Behavior and Usage

```
#include <setjmp.h>
jmp_buf closure;  /* address, stack */
void top(void) {
  switch (setjmp(closure)) {
    case 0: child(); break;
    case 1: /* longjmp called */ break;
} }
void child() { child2(); }
void child2() { longjmp(closure, 1); }
```

PL/I Pioneered Exception Handling

PL/I has a very flexible dynamic mechanism:

```
on ZERODIVIDE go to HandleZeroDivide;
```

Establishes a condition handler that persists until control leaves its block, or until it is overridden.

Tricky: currently-active handlers are a function of the dynamic execution of the program.

Unix Signal Handling

Unix provides a similar facility:

```
#include <stdio.h>
#include <signal.h>

void handleint() {
  printf("Got an INT");
}

void main() {
  signal(SIGINT, handleint);
  for (;;) { }
}
```

Exceptions: Lexically Bound

Dynamic behavior is a problem with PL/I and Unix mechanisms.

Too confusing

Too much overhead

Not structured

Better to make exceptions lexically bound like variables.

Idea is to treat is as an exceptional return from a procedure, not a cross-procedure goto.

Java’s Exception Mechanism

```
class MyException extends Exception {}

try {

  if (error) throw new MyException();

} catch (MyException e) {

  System.out.println("Caught Exception");

}
```

Java's Finally

```java
class E extends Exception {}

class Foo {
  public static void main(String[] args)
  { p(1); foo(args[0]); p(5); }

  static void foo(String s) {
    try {
      if (s.equals("a")) throw new E();
      if (s.equals("b")) return;
      p(2);
    } catch (E e) { p(3); }
    finally { p(4); } // Always executed
  }

  static void p(int v) { System.out.println(v); }
}
```

a	b	c
1	1	1
		2
3		
4	4	4
5	5	5

Types of Exception Objects

An exception is a built-in type in Ada:

```
declare empty_queue : exception;
```

It is another kind of object in Modula-3:

```
EXCEPTION empty_queue;
```

It is an ordinary object in C++:

```
class empty_queue {};
```

It extends the **`Exception`** class in Java:

```
class SyntaxError extends Exception {}
```

Valued Exceptions

```
class Syntax extends Exception {
  String file;
  int line;
  String exp;
  public Syntax(String f, int l, String e)
    { file = f; line = l; exp = e; }
  String toString() { return file + ":" +
    Integer.toString(line,10) +
    ":syntax error, expecting " + exp;
  }
}

throw new Syntax("hello.c", 10, "}");
```

Declaring Exceptions in Modula-3

Any raised exception must be listed:

```
EXCEPTION Fail, Reject;

PROCEDURE NewAccount (name: TEXT)
  RAISES (Reject) =
  BEGIN
    RAISE Reject; (* OK *)

    RAISE Fail; (* Run-time Error:
                   exception not listed *)
  END NewAccount
```

Declaring Exceptions in C++

If given, function may only throw listed exceptions

```
class Ex1 {};
class Ex2 {};
class Ex3 : Ex2 {};

void foo()
  { throw Ex1(); }    // OK

void bar() throw(Ex1)
  { throw Ex2(); }    // Run-time error

void baz() throw(Ex2)
  { throw Ex3(); }    // OK
```

Declaring Exceptions in Java

Only “checked” exceptions must be listed.

```
class Ex1 extends Exception {}
class Ex2 extends Ex1 {}
class Ex3 extends Exception {}

public void foo() throws Ex1 {
  throw new Ex1(); // OK
  throw new Ex2(); // OK
  throw new Ex3(); // Compile-time error
  throw new UnknownError(); // OK: Unchecked
}
```

What Exceptions are Caught

In Ada, either exact name match or “others”

```
declare ex1 : exception;
procedure foo ... is
begin
  begin
    ...
    raise ex1;
  exception
    when ex1 => ... -- handles ex1
    when others => ... -- everything else
  end;

end foo;
```

What Exceptions are Caught

C++ supports inheritance and “...”

```
class Ex1 {};
class Ex2 : Ex1 {};

try {
  throw Ex1();
  throw Ex2();
} catch (Ex1 e) { /* Ex2 or Ex1 */ }
catch (...) { /* any others */ }
```

Obvious Way to Implement Exceptions

```
try {                      push(Ex, Exhandler);

  throw Ex;                  throw(Ex);
                             pop();
                             goto Exit;
} catch (Ex e) {           Exhandler:
  foo();                     foo();
}                          Exit:
```

push() adds a handler to a stack

pop() removes a handler

throw() finds first matching handler

Problem: imposes overhead even with no exceptions

Implementing Exceptions Cleverly

Real question is the nearest handler for a given PC.

```
1  void foo() {
2
3    try {
4      bar();
5    } catch (Ex1 e) { H1: a(); }
6
7  }

8  void bar() {
9
10   try {
11     throw Ex1();
12   } catch (Ex2 e) { H2: b(); }
13
14 }
```

1–2	Reraise
3–5	H1
6–9	Reraise
10–12	H2
13–14	Reraise

look in table

no match, reraise

look in table

Parameters

Call-By-Value

The default in C

```
void foo(int x) {
  x = x + 10; // Does not change y
  printf("%d ", x);
}

void main() {
  int y = 0;
  foo();
  printf("%d ", y);
}
```

Prints “10 0”

Call-By-Reference

In C, you must explicitly use pointers and take addresses

```
void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

void main() {
  int x = 2, y = 3;

  swap(&x, &y);
}
```

Call-By-Reference

C++ references simplify the syntax

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void main() {
  int x = 2, y = 3;
  swap(x, y);      // Works
}
```

Java’s Object References

This prints “5 6”: ints are passed by value.

```
class Foo {
  public static void swap(int x, int y)
  { int tmp = x; x = y; y = tmp; }

  public static void p(int i)
  { System.out.println(Integer.toString(i,10)); }

  public static void main(String[] args) {
    int x = 5, y = 6;
    swap(x,y); p(x); p(y); // Does not swap
  }
}
```

Java’s Object References

This prints “6 5”: objects are passed by reference

```
class MyInt {
  int v;
  MyInt(int vv) { v = vv; }
  int get() { return v; }
  void set(int vv) { v = vv; };
}

class Foo {
  public static void swap(MyInt x, MyInt y)
  { int tmp = x.get(); x.set(y.get()); y.set(tmp); }

  public static void p(int i)
  { System.out.println(Integer.toString(i,10)); }

  public static void main(String[] args) {
    MyInt x = new MyInt(5);
    MyInt y = new MyInt(6);
    swap(x, y); p(x.get()); p(y.get()); // Swaps
  }
}
```

Aliases

Pass-by-reference can cause strange behavior:

```
int sum3(int &x, int &y) {
  x = x * 3;
  y = y * 3;
  return x + y;
}

int x = 2, y = 3;
sum3(x, y); // OK : returns 15

int w = 5;
sum3(w, w); // Returns 90!
```

Pass-By-Reference vs. -Value

Pass-by-value ensures caller can’t modify the value.

No sticky alias problems

Inefficient for large objects.

Pass by Value/Result

Ada has copy in/copy out semantics.

```
procedure foo(a : in integer,
              b : out integer,
              c : in out integer) in
begin
  c = c + a;
  b = a + 2;
  a = a + 1;
end foo;

x, y, z : integer;

x := 1; z := 5;
foo(x,y,z);
-- x = 1    unchanged
-- y = 3    copied from x
-- z = 6    copied then out
```

No Aliasing in Ada

Copy in/copy out semantics:

```
function sum3(x : in out integer,
              y : in out integer) return integer is
begin
  x := x * 3;
  y := y * 3;
  return x + y;
end

w : integer;

w := 5;
sum3(w, w); -- Returns 30, w = 15
```

Large Ada Objects May Be Passed by Value or Reference

```
type t is record
  a, b : integer;
end record;

procedure foo(s : in out t) is begin
  r.a := r.a + 1;
  s.a := s.a + 1;
end foo;

r : t;
r.a := 3;
foo(r);
put(r.a); -- Prints 4 or 5: erroneous program
```