



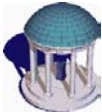
The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts
Spring 2003

Parameter Passing, Generics and Polymorphism, Exceptions

Stotts, Hernandez-Campos

1



Parameter Passing

- Pass-by-value
 - Input parameter
- Pass-by-result
 - Output parameter
- Pass-by-value-result
 - Input/output parameter
- Pass-by-reference
 - Input/Output parameter, no copy
- Pass-by-name
 - Textual substitution

2

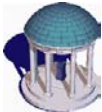


Pass-by-value

```
int m=8, i=5;
foo(m);
print m;    # prints 8
            # since m is passed by-value

...
proc foo (byval b) {
  b = i + b;
  # b is byval so it is essentially a local variable
  # initialized to 8 (the value of the actual back in
  # the calling environment)
  # the assignment to b cannot change the value of m back
  # in the main program
}
```

3

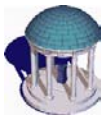


Pass-by-reference

```
int m=8, I=5;
foo(m);
print m;    # prints 13
            # since m is passed by-reference

...
proc foo (byref b) {
  b = i + b;
  # b is byref so it is a pointer back to the actual
  # parameter back in the main program (containing 8 initially)
  # the assignment to b actually changes the value in m back
  # in the main program
  # i accesses the variable back in the main via scope rules
}
```

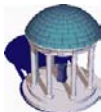
4



Pass-by-value-result

```
int m=8, I=5;
foo(m);
print m;    # prints 13
            # since m is passed by-value-result
...
proc foo (byvres b) {
  b = i + b;
  # b is byref so it is a pointer back to the actual
  # parameter back in the main program (containing 8 initially)
  # the assignment to b actually changes the value in m back
  # in the main program
  # i accesses the variable back in the main via scope rules
}
```

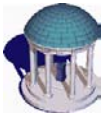
5



Pass-by-name

- Arguments passed by name are re-evaluated in the caller's referencing environment every time they are used
- They are implemented using a hidden-subroutine, known as a *thunk*
- This is a costly parameter passing mechanism
- Think of it as in-line substitution (subroutine code put in-line at point of call, with params substituted)
- Or, actual params substituted textually in the subroutine body for the formals

6



Pass-by-name

```
array A [1..100] of int;
int i=5;
foo(A[i],i);
print A[i]; # prints A[6]
...        # so prints 7
```

```
# good example
proc foo (name B,name k) {
  k = 6;
  B = 7;
}
```

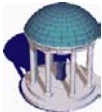
```
# text substitution does this
proc foo {
  i = 6;
  A[i] = 7;
}
```

```
array A [1..100] of int;
int i=5;
foo(A[i]);
print A[i]; # prints A[5]
...        # not sure what
```

```
# a problem here...
proc foo (name B) {
  int i = 2;
  B = 7;
}
```

```
proc foo {
  int i = 2;
  A[i] = 7;
}
```

7

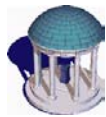


Pass-by-name

- Evaluate: $y = \sum_{1 \leq x \leq 10} 3x^2 - 5x + 2$
- In pass-by-name: $y := \text{sum}(3*x*x - 5*x + 2, x, 1, 10)$

```
real procedure sum (expr, i, low, high);
  value low, high;
  comment low and high are passed by value;
  comment expr and i are passed by name;
  real expr;
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do
    rtn := rtn + expr;
    comment the value of expr depends on the value of i;
  sum := rtn
end sum
```

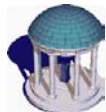
8



Ada

- **in** is a call-by-value
- **out** is a call-by-result
- **in/out** is call-by-value result
- Pass-by-value is expensive for complex types, so it can be implemented by passing either values or references
- However, programs can have different semantics with two solutions
 - These are illegal program in Ada

9



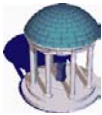
Ada Example

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

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

...
r.a := 3;
foo (r);
put (r.a);      -- does this print 4 or 5?
```

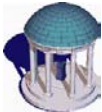
10



Summary

	implementation mechanism	permissible operations	change to actual?	alias?
value	value	read, write	no	no
in, const	value or reference	read only	no	maybe
out (Ada)	value or reference	write only	yes	maybe
value/result	value	read, write	yes	no
var, ref	reference	read, write	yes	yes
sharing	value or reference	read, write	yes	yes
in out (Ada)	value or reference	read, write	yes	maybe
name (Algol 60)	closure (thunk)	read, write	yes	yes

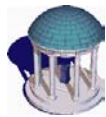
11



Generics, Polymorphism

- Polymorphism is the property of code working for arguments/data of different types
 - *Sort (list) works for list of int, list of string*
- ML allows this but at cost... dynamic type checking
- Generics, templates allow static type checking but some measure of polymorphism

12

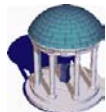


Generics, Polymorphism

```
generic compare (x,y: type T) returns bool{
    return x < y ;
}
...
Creal = new compare(T=real);
Cint = new compare(T=int);
Cstr = new compare(T=string);

Generic inc (a: type T) returns T {
    return a + 1;
}
Cint = new compare(T=int);
Cstr = new compare(T=string); # NO... compiler rejects... why?
```

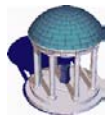
13



Exception Handling

- An exception is an unexpected or unusual condition that arises during program execution
 - Raised by the program or detected by the language implementation
- Example: read a value after EOF reached
- Alternatives:
 - *Invent* the value (e.g. -1)
 - Always return the value and a status code (must be checked every time)
 - Pass a closure (if available) to handle errors

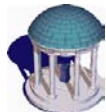
14



Exception Handling

- Exceptions move error-checking out of the *normal* flow of the program
 - No special values to be returned
 - No error checking after each call
- Exceptions in Java
 - <http://java.sun.com/docs/books/tutorial/essential/exceptions/>

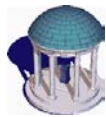
15



Exception Handlers Pioneered in PL/1

- Syntax: **ON** condition
 statement
- The nested statement is not executed when the **ON** statement is encountered, but when the condition occurs
 - E.g. overflow condition
- The binding of handlers depends on the flow of control
- After the statement is executed, the program
 - terminates if the condition is considered irrecoverable
 - continues at the statement that followed the one in which the exception occurred
- *Dynamic binding of handlers and automatic resumption can potentially make programs confusing and error-prone*

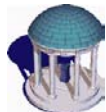
16



Exception Handlers

- Modern languages make the exception handler lexically bound, so they replace the portion of the code yet-to-be-completed
- In addition, exceptions that are not handled in the current block are propagated back up the *dynamic chain*
 - The dynamic chain is the sequence of dynamic links
 - Each activation record maintains a pointer to its caller, a.k.a. the *dynamic link*
 - This is a restricted form of dynamic binding

17



Exception Handlers Java

- Java uses lexically scoped exception handlers

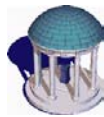
```
try {
    int a[] = new int[2];
    a[4];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("exception: " + e.getMessage());
    e.printStackTrace();
}
```

- The dynamic chain is available using

`printStackTrace()`

– <http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Throwable.html>

18

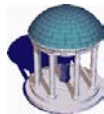


Exception Handlers

Use of Exceptions

- Recover from an unexpected condition and continue
 - *E.g.* request additional space to the OS after out-of-memory exception
- *Graceful* termination after an unrecoverable exception
 - Printing some helpful error message
 - » *E.g.* Dynamic link and line number where the exception was raised in Java
- Local handling and propagation of exception
 - Some exceptions have to be resolved at multiple level in the dynamic chain
 - *E.g.* exceptions can be reraised in Java using the **throw** statement

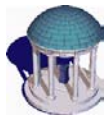
19



Returning Exceptions

- Propagation of exceptions effectively *makes* them *return values*
- Consequently, programming languages include them in subroutine declarations
 - Modula-3 requires all exceptions that are not caught internally to be declared in the subroutine header
 - C++ make the list of exception optional
 - Java divides them up into *checked* and *unchecked* exceptions (RuntimeExceptions do not have to be declared/caught)
 - » *E.g.* ArithmeticException vs. IOException

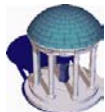
20



Hierarchy of Exceptions

- In PL/1, exception do not have a type
- In Ada, all exceptions are of type `exception`
 - Exception handler can handle one specific exception or all of them
- Since exceptions are classes in Java, exception handlers can capture an entire class of exceptions (parent classes and all its derived classes)
 - Hierarchy of exceptions

21



Implementation

- Linked-list of dynamically-bound handlers maintained at run-time
 - Each subroutine has a default handler that takes care of the epilogue before propagating an exception
 - This is slow, since the list must be updated for each block of code
- Compile-time table of blocks and handlers
 - Two fields: starting address of the block and address of the corresponding handler
 - Exception handling using a binary search indexed by the program counter
 - Logarithmic cost on the number of handlers

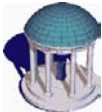
22



Java

- Each subroutine has a separate exception handling table
 - Thanks to independent compilation of code fragments
- Each stack frame contains a pointer to the appropriate table

23



C

- Exception can be simulated
- **setjmp()** can store a representation of the current program state in a buffer
 - Returns 0 if normal return, 1 if return from long jump
- **longjmp()** can restore this state
- Example:

```
if (!setjmp(buffer)) {  
    /* protected code */  
} else {  
    /* handler */  
}
```

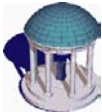
24



C

- The state is usually the set of registers
- `longjmp()` restores this set of registers
 - http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_20.html
- Is this good enough?
- Changes to variables before the long jump are committed, but changes to registers are ignored
- If the handler needs to see changes to a variable that may be modified in the protected code, the programmer must include the `volatile` keyword in the variable's declaration

25



Reading Assignment

- Read Scott
 - Sect. 8.3
 - Sect. 8.5
 - Sect. 8.4

26