Lecture 9: Parameter Passing, Generics and Polymorphism, Exceptions

COMP 524 Programming Language Concepts
Stephen Olivier
February 12, 2008

Based on notes by A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts

The University of North Carolina at Chapel Hill

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**: Effectively textual substitution

Pass-by-value

```
int m=8, i=5;
foo(m);
print m; # print 8

proc foo(int b){
  b = b+5;
}
```

Pass-by-reference

```
int m=8;
foo(m);
print m; # print 13

proc foo(int b){
  b = b+5;
}
```

Pass-by-value-result

```
int m=8;
foo(m);
print m; # print 13

proc foo(int b){
  b = b+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 an in-line substitution (subroutine code put in-line at point of call, with parameters substituted).
- Or, actual parameters substituted textually in the subroutine body for the formulas.

Pass-by-name

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

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

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

```
array A[1..100] of int;
int i=5;
foo(A[i]);
print A[i]; #print A[5]=??

#BAD Example
proc foo(name B){
  int i=2;
  B=7;
}

#text sub does this
proc foo{
  int i=2;
  A[i]=7;
}
```

Pass-by-name

- Evaluate $$y = \sum_{1 \leq x \leq 10} 3x^2 - 5x + 2$$

- In pass-by-name: $y := \textsf{sum}(3 \cdot x \cdot x - 5 \cdot x + 2, x, 1, 10)$

```
real proc sum(expr, i, low, high);
  value low, high;
  real expr;
  integer i,low,high;
begin
  real rtn;
  rtn := 0;
  for i:= low step 1 until high do
    rtn := rtn + expr;
  sum:=rtn
end sum;
```

Ada

- `in` is call-by-value
- **out** is 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
  - This is “erroneous” in Ada.

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?
```

Summary

	Implementatio n mechanism	Permissible Operations	Changes to actual?	Alias?
value	Value	read, write	no	no
in, const	Val or ref	read only	no	maybe
out (Ada)	Val or ref	write only	yes	maybe
value/result	Val	read, write	yes	no
var, ref	Ref	Read, write	yes	yes
sharing	val or ref	Read, write	yes	yes
in out (Ada)	val or ref	Read, write	yes	maybe
Name (Algo 60)	Closure (thunk)	Read, write	yes	yes

Other Parameter Passing Features

- Variable length parameter lists
  - flexible in C using "..."
  - C++, C#, Java require that all are the same type
- Named parameters
  - Eliminates requirement for programmer to match the order of formal parameters to the order of actual parameters
- Default parameters
  - Default value provided in the definition of the subroutine

Return Values

- Allow return values of complex types?
    - Some restrict to primitive types and pointers
- Allow subroutine closures to be returned?
    - A closure bundles a reference to a subroutine and a referencing environment
    - Available in some imperative languages
    - C instead allows function pointers to be returned

Return Values

- How to specify the return value?
  - Fortran, Algol, Pascal: same name as the function name

```
function  add_five ( value1 : integer ) : integer;
   begin
         add_five := value1 + 5;
   end;
```

  - C, C++, Java: *return* statement
  - Other languages: name for function result provided in header

```
procedure a () returns retvar : int
    retval := 5;
end
```

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 each time)
  - Pass a closure (if available) to handle errors

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

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.

Exception Handlers

- Modern languages make 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.

Exception Handlers

- 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();
}
```

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 exception have to be resolved at multiple level in the dynamic chain.
  - e.g., Exceptions can be reraised in Java using the throw statement.

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++ makes the list of exception optional
  - Java divides them up into **checked** and **unchecked** exceptions

Hierarchy of Exceptions

- In PL/1, exceptions 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.

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 of the number handlers.

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.

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

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

C

- The state is usually the set of registers
- `longjmp()` restores this set of registers
- 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.

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
- Many functional languages allow this but at cost... dynamic type checking
- Generics, templates allow static type checking but some measure of polymorphism.

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 reject
```