

Chapter 9

Subprograms

Chapter 9 Topics

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms and Functions
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Overloaded Subprograms and Operator
- Generic Subprograms
- Coroutines

Subprograms (Functions)

- Fundamental to all programming languages
- Essential for:
 - Abstraction
 - Separation of concerns
 - Top-down design
- Behavior of subprograms is closely related to dynamic memory management and the stack

Abstraction

- Languages provide two fundamental abstraction facilities
 - Process abstraction
 - The only abstraction available in early imperative languages.
 - Any data structure was implemented as an array in Fortran
 - How would you do this with a tree?
 - Data abstraction
 - Strong emphasis in language design in the 1980's
 - Evolutionary Trail
 - Records
 - Abstract Data Types
 - Packages
 - Objects
 - Objects of course have been around since the 1960's (what goes around, comes around)

Terminology

- Some languages syntactically distinguish functions that return a value from functions that do not
 - Fortran: Functions and Subroutines
 - Ada/Pascal: Functions and Procedures
- C-Like languages do not make a syntactic distinction
 - Functions that do not return a value are called void functions
- At the machine level there is no distinction
- We will use the terms subroutine, subprogram and function interchangeably

Function calls

- Value-returning functions are r-values and can appear in expressions:
e.g., $x = (b*b - \text{sqrt}(4*a*c))/2*a$
- Non-value-returning functions are invoked as a separate statement
e.g., `strcpy(s1, s2);`

Assumptions

- Except for coroutines and concurrent calls, we consider that:
 - Each subprogram has a single entry point
 - Some languages such as Fortran do allow multiple entry points
 - The calling program is suspended during execution of the called subprogram
 - Control always returns to the caller when the called subprogram's execution terminates

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type

Basic Definitions (continued)

- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement

Design Issues for Subprograms

- Are local variables statically or dynamically allocated?
- Can subprogram definitions be nested?
- What parameter passing methods are provided?
- Are actual/formal parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprograms be generic?

Design Issues for Functions

- Are side effects allowed?
 - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
 - Most imperative languages restrict the return types
 - C allows any type except arrays and functions
 - C++ is like C but also allows user-defined types
 - Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
 - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
 - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
 - Javascript treats functions as first-class objects that can be returned from functions
 - Lua allows functions to return multiple values

Function/Subprogram Headers

- Fortran (parameter types defined on separate line)

```
subroutine avg(a,b,c)
  real a, b, c
```

- As a function

```
real function avg(a,b)
  real a, b
```

- C

```
void avg(float a, float b, float c);
float avg(float a, float b)
```

Function/Subprogram Headers

- Ada

```
procedure Avg (A, B: in Integer; C: out Integer)
function Avg(a,b: in Integer) returns Integer
```

Python def

- Python function definitions are executable

```
if n==3 :
    Def avg(a,b,c)
        Return (a+b+c)/3
else :
    Def avg(a,b)
        Return (a+b)/2
```

Parameters: Access to Data

- Subprograms can access data either through non-local variables or by parameters passed to the subprograms
- Parameters are more flexible
 - Accessing non-local variables binds the subprogram to the variable names
 - Parameters by contrast provide local names (aliases) for data passed by the caller

Parameters: Access to Computations

- Parameters can also describe computations as well as data
- Subprogram or function names or address can be passed as parameters

Actual and Formal Parameters

- Parameters in subprogram headers are called formal parameters
 - A local name for actual values or variables passed
 - Bound to storage only during subprogram activation
- Parameters in subprogram headers are called actual parameters
 - Bound to formal parameters when subprogram is activated
 - Different restrictions on forms than actual parameters
- Alternate terminology: Formal parameters may be just called "parameters" while actual parameters are called "arguments."

Actual/Formal Parameter Correspondence

- How are actual parameters bound to the formal parameters?
- Positional
 - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
 - Safe and effective
- Keyword
 - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
 - Advantage: Parameters can appear in any order, thereby avoiding parameter correspondence errors
 - Disadvantage: User must know the formal parameter's names

Examples

- Keyword parameters (Python)

```
listsum(length = my_length,  
        list = my_array,  
        sum = my_sum
```
- Mixed positional / kw parameters (Python)

```
listsum(my_length,  
        list = my_array,  
        sum = my_sum
```
- After first keyword parameters all others must also be keyword params
- T-SQL

```
exec dbo.update_orders @date=GetDate(),  
                       @loc='oro'
```

Some Exceptions

- Perl - parameters aren't declared in a function header. Instead, parameters are available in an array `@_`, and are accessed using a subscript on this array.
- Smalltalk: unusual infix notation for method names

```
array at: index + offset put: Bag new  
array at: 1 put: self  
x < 4 ifTrue: ['Yes'] ifFalse: ['No']
```

Formal Parameter Default Values

- In many languages (ex: C++, Python, Ruby, Ada, PHP, Visual Basic), formal parameters can have default values
 - Default value is used if no actual value was supplied
- Ex: (Python)

```
def day_of_week(date, firstday = "Sunday")
```
- Syntactic rules can be complex
 - In C++, default parameters must appear last because parameters are positionally associated
 - In languages with keyword parameters, any parameter following an omitted parameter must be keyworded

Examples

- Default values (Python)

```
def compute_pay(income, exemptions=1, tax_rate)
```
- Example call

```
compute_pay(74000.0, tax_rate=0.273)
```
- Default values (C++)

```
Float compute_pay(float income, float tax_rate,  
                  int exemptions=1)
```
- Example call

```
myPay = compute_pay(74000, 0.273)
```

Variable Parameter Lists

- Variable numbers of parameters
 - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by `params`
 - In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
 - In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
 - In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods

Examples

- A C# function

```
public void DisplayList(params int[] list){  
    foreach (int next in list){  
        Console.WriteLine("Next value {0}",next);  
    }  
}
```
- Example calls

```
Myclass MyObject = New Myclass;  
int[] mylist = new int[6]{1,2,3,4,5,6};  
MyObject.DisplayList(mylist);  
MyObject.DisplayList(1,2*y-x,mylist[3],17);
```

Ruby Example

- Ruby does not support keyword parameters, but the last parameter in a function def can be designed as the array formal parameter by preceding it with *

```
def tester(p1,p2,p3,*p4)
  . . .
end
list = [2,4,6,8]
tester('first',{mon=>72,tue=>55,wed=>70},*list)
```

- Parameter assignments are:

```
p1 is 'first'
p2 is {mon=>72,tue=>55,wed=>70}
p3 is 2
p4 is [4,6,8]
```

Ruby Blocks

- Ruby provides built-in iterators that can be used to process the elements of arrays; e.g., each and find
 - Iterators are implemented with blocks, which can also be defined by applications
- Blocks can have formal parameters (specified between vertical bars)
 - they are executed when the method executes a `yield` statement

Ruby Blocks

```
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first, second = second, first + second
  end
end

puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " " }
puts
```

Local Referencing Environments

- Local Variables
 - Variables defined inside subprograms are called local variables
 - Local variables can be either static or stack-dynamic
 - Stack dynamic variables are essential to recursive programming
- In C terminology local variables are often called volatile variables because of stack dynamic allocation

Local Variables

- Stack-dynamic locals
 - Advantages
 - Support for recursion
 - Storage for locals is shared among some subprograms
 - Disadvantages
 - Allocation/de-allocation, initialization time
 - Indirect addressing
 - Subprograms cannot be history sensitive
- Local variables can be static
 - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Local Variables

- Default to stack-dynamic in most languages
 - Ada subprograms, and C++, C# and Java methods allow only stack-dynamic variables
 - Fortran variables are static unless a subroutine or function is declared to be recursive
- With nested subprograms and static scoping, restricted access to non-local variables is possible
 - Recent languages with nested subprograms include JavaScript, Python, Ruby and Lua

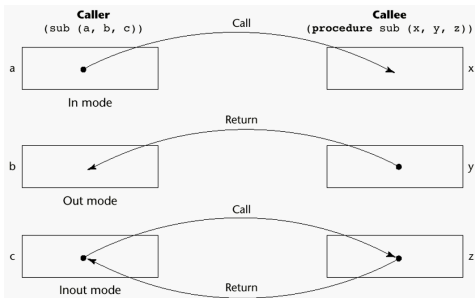
Parameter Passing Methods

- We will consider both semantic models and implementation models
- Semantic models are concerned with the effects of assignments to formal parameters
- Implementation models are techniques of achieving a desired semantic model

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Semantic Models



Conceptual Models of Transfer

1. Actual values can be copied (to caller, callee or both)
2. Provide a reference or an access path rather than copying values

Pass-by-Value (In Mode)

- Assignment to the formal parameter has no effect outside the subprogram
- The value of the actual parameter is used to initialize the corresponding formal parameter
 - Normally implemented by providing a copy of the actual parameter on the stack
 - Can be implemented by providing a reference or transmitting an access path but not recommended (enforcing write protection is not easy)
 - *Disadvantages:* additional storage is required (stored twice) and the actual copy operation can be costly (for large parameters)

Swap Function

- This won't work in C

```
void swap (int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```
- This will work (pass by reference)

```
void swap (int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```
- To call

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

Swap in Java

- Same reasoning in Java

```
void swap (Object a, Object b) {  
    Object temp = a;  
    a = b;  
    b = temp;  
}
```

- But you can swap array elements

```
void swap (Object [] A, int i, int j) {  
    int temp = A[i];  
    A[i] = A[j];  
    A[j] = temp;  
}
```

Java Swapping Using a Wrapper Class

```
// MyInteger: similar to Integer, but can change value  
class MyInteger {  
    private int x; // single data member  
    public MyInteger(int xIn) { x = xIn; } // constructor  
    public int getValue() { return x; } // retrieve value  
    public void insertValue(int xIn) { x = xIn; } // insert  
}  
  
public class Swapping {  
    // swap: pass references to objects  
    static void swap(MyInteger rWrap, MyInteger sWrap) {  
        // interchange values inside objects  
        int t = rWrap.getValue();  
        rWrap.insertValue(sWrap.getValue());  
        sWrap.insertValue(t);  
    }  
  
    public static void main(String[] args) {  
        int a = 23, b = 47;  
        System.out.println("Before. a: " + a + ", b: " + b);  
        MyInteger aWrap = new MyInteger(a);  
        MyInteger bWrap = new MyInteger(b);  
        swap(aWrap, bWrap);  
        a = aWrap.getValue();  
        b = bWrap.getValue();  
        System.out.println("After. a: " + a + ", b: " + b);  
    }  
}  
  
// from http://www.cs.utsa.edu/~wagner/CS2213/swap/swap.html
```

Swapping with C++ preprocessor

- Also from

<http://www.cs.utsa.edu/~wagner/CS2213/swap/swap.html>

```
#define swap(type, i, j) {type t = i; i = j; j = t;}  
  
int main() {  
    int a = 23, b = 47;  
    printf("Before swap. a: %d, b: %d\n", a, b);  
    swap(int, a, b);  
    printf("After swap. a: %d, b: %d\n", a, b);  
    return 0;  
}  
  
• Preprocessed output  
  
int main() {  
    int a = 23, b = 47;  
    printf("Before swap. a: %d, b: %d\n", a, b);  
    { int t = a ; a = b ; b = t ; }  
    printf("After swap. a: %d, b: %d\n", a, b);  
    return 0;  
}
```

Pass-by-Result (Out Mode)

- When a parameter is passed by result:
 - no value is transmitted to the subprogram
 - the corresponding formal parameter acts as a local variable
 - its value is transmitted to caller's actual parameter when control returns, by physical copy
 - Require extra storage location and copy operation
- Potential problem: sub(p1, p1)
 - whichever formal parameter is copied back will represent the current value of p1
 - Order determines value

Out Mode Example: C#

```
void fixer(out int x; out int y){  
    x = 42;  
    y = 33;  
}  
  
// what happens with this code?  
f.fixer(out a, out a);  
  
• Or this code?  
  
void doit(out int x; int ndx){  
    x = 42;  
    ndx = 2;  
}  
  
int sub = 3;  
f.doit(out myList[sub], sub);
```

Pass-by-Reference (Inout Mode)

- Pass reference or access path (usually just the address)
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
 - Potentials for unwanted side effects (parameter collisions)
 - Aliasing

Distinguishing ref & value parameters

- If languages support both pass-by-value and pass-by-reference, the distinction must be made explicit in the in the parameter list
- Ex: Pascal
by value

```
function f(x, y: integer): integer;
```


by reference

```
procedure swap(var x, y: integer);
```

Swap Function

- C with pass by reference

```
void swap (int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```
- To call

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

Reference Parameters must be l-values

- Since an address is passed, you can't pass a literal value as a reference parameter

```
swap (a, b) //OK  
swap(a+1, b) // Not OK  
swap(x[j],x[j+1]) // OK
```
- In Fortran all parameters are reference
- Some early compilers had an interesting bug

```
Subroutine inc(j)  
    j = j + 1  
End Subroutine
```
- A call to `inc(1)` would leave the constant with the value 2 for the remainder of the program

Using r-values as arguments

- Some languages (e.g., Fortran) allow non l-values as arguments for reference parameter
- Solution is to create a temporary variable and pass that address

Parameter Collisions / Aliasing

- A C++ function:

```
void fun(int &p1, int &p2)
```
- What happens here

```
fun(diff, diff);  
fun(list[i], list[j]);
```
- What happens here

```
void fun(int list[], int &p2)  
fun(list, list[j]);
```

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
 - Think of it as copy-in, copy-out
- Formal parameters have local storage
- Disadvantages:
 - Those of pass-by-result
 - Those of pass-by-value
- Advantages:
 - Same as pass by reference

Pass by value-result

- Identical to pass-by-reference except when aliasing is involved
- An Ada-syntax swap:

```
Procedure swap3(a : in out Integer,  
  b : in out Integer) is  
  temp : Integer  
Begin  
  temp := a;  
  a := b;  
  b := temp;  
end swap3;
```

A normal call

- Suppose swap3 is called with
 swap3(c, d)
- The actions of swap3 with this call are

```
addr_c = &c    - Move first parameter address in  
addr_d = &d    - Move second parameter address in  
a = *addr_c   - Move first parameter value in  
b = *addr_d   - Move second parameter value in  
temp = a  
a = b  
b = temp  
*addr_c = a   - Move first parameter value out  
*addr_d = b   - Move second parameter value out
```

A call with an array element

- Now consider
 swap3(i, list[i])
- The actions of swap3 with this call are
 addr_i = &i - Move first parameter address in
 addr_listi = &list[i] - Move 2nd parameter addr in
 a = *addr_i - Move first parameter value in
 b = *addr_listi - Move 2nd parameter value in
 temp = a
 a = b
 b = temp
 *addr_i = a - Move first parameter value out
 *addr_listi = b - Move 2nd parameter val out
- Operation is correct because addresses are computed on copy-in

Aliasing

- A program in C-like syntax
 int i = 3; /* global */
 void fun (int a, int b) {
 i = b;
 }
 void main() [
 int list[10];
 list[i] = 5;
 fun(i, list[i]);
]

Aliasing

- a and i are aliases. With pass by reference we have
 addr_i = &i - Move first parameter address in
 addr_listi = &list[i] - Move 2nd parameter addr in
 a = *addr_i - a <- 3 (i)
 b = *addr_listi - b <- 5 (list[3])
 i = b - global i <- 5
- But pass by value-result adds
 *addr_i = a - global i <- 3
 *addr_listi = b - list[i] <- 5

Pass-by-Name

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding
- An interesting idea that was developed with Algol and virtually abandoned because of the semantic complexity

Pass by Name

- Example of *late binding*, since evaluation of the argument is delayed until its occurrence in the function body is actually executed.
- An argument *passed by name* behaves as though it is textually substituted for every occurrence of the corresponding parameter in the function body.
- Originally used in Algol 60/Algol 68
- Dropped by successors: Pascal, Modula, Ada
- Associated with lazy evaluation in functional languages e.g., Haskell

Pass by Name

- Implications of pass-by-name:
 1. The argument expression is re-evaluated each time the formal parameter is accessed.
 2. The procedure can change the values of variables used in the argument expression and thus change the expression's value.

Swap again

- Cannot write a general purpose swap

```
procedure swap(a, b);
  integer a, b;
begin
  integer t;
  t := a; => t := i
  a := b; => t := a[i]
  b := t; => a[i] := t (but i has changed)
End;
```

- Swap (i, j) works fine but swap (i, a[i]) does not because the second assignment a := b changes the value of i

Another Simple Pass by Name example

```
procedure inc2(i, j);
  integer i, j;
  begin
    i := i+1; => k <- k+1
    j := j+1  => A[k] <- A[k]+1
  end;
inc2 (k, A[k]);
```

Jensen's Device

```
real procedure SIGMA(x, i, n);
  value n;
  real x; integer i, n;
begin
  real s;
  s := 0;
  for i := 1 step 1 until n do
    s := s + x;
  SIGMA := s;
End
```

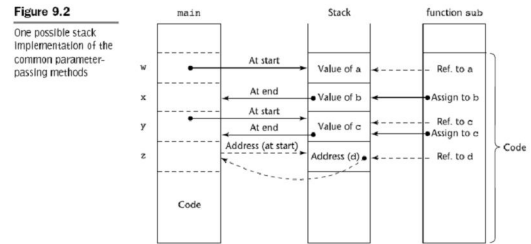
Jensen's Device

```
s := s + x;
• Consider
s := SIGMA(a, b, c);      (computes a * c)
  for i := 1 step 1 until n do
    s := s + a;
s := SIGMA(x[i], i, n);  (computes x1+x2+...+xn)
  for i := 1 step 1 until n do
    s := s + x[i];
s := SIGMA(x[i]*y[i], i, n); (x1*y1+ x2*y2+...+ xn*yn)
  for i := 1 step 1 until n do
    s := s + x[i] * y[i];
s := SIGMA(1/i, i, n);   (1/1 + 1/2 + 1/3 ... + 1/n)
  for i := 1 step 1 until n do
    s := s + 1/i;
```

Implementing Parameter-Passing Methods

- In most language parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack
- A subtle but fatal error can occur with pass-by-reference and pass-by-value-result: a formal parameter corresponding to a constant can mistakenly be changed

Stack Implementation



```
void sub(int a, int b, int c, int d)
. . .
Main()
  sub(w,x,y,z)
//pass w by val, x by result, y by value-result, z by ref
```

Parameter Passing: Some Major Languages

- C
 - Pass-by-value
 - Pass-by-reference is achieved by using pointers as parameters
- C++
 - A special pointer type called reference type for pass-by-reference
 - Reference params are implicitly dereferenced
 - Can have const reference parameters
- Java
 - All parameters are passed are passed by value
 - Object parameters are passed by reference so objects can be changed
 - But reference parameters cannot point to scalars
- Ada
 - Three semantics modes of parameter transmission: *in*, *out*, *in out*; *in* is the default mode
 - Formal parameters declared *out* can be assigned but not referenced; those declared *in* can be referenced but not assigned; *in out* parameters can be referenced and assigned

Parameter Passing: Some Major Languages

- Fortran 95
 - Parameters can be declared to be *in*, *out*, or *inout* mode using *Intent*. Otherwise pass by reference
- C#
 - Default method: pass-by-value
 - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with *ref*
- PHP: very similar to C#
- Perl: all actual parameters are implicitly placed in a predefined array named *@_*
 - Changes made to elements of *@_* will be reflected in actual parameter if it was a value
- Python and Ruby use pass-by-assignment
 - Every variable is a reference to an object
 - Effectively the same as pass by reference but some data objects are immutable

Type Checking Parameters

- Important for reliability
 - FORTRAN 77 and original C: none
 - Pascal, FORTRAN 90, Java, and Ada: always required
- C
 - Functions can be declared without types in headers:


```
double sin(x){
  double x; /* no type checking */
```
 - Or by prototypes with types


```
double sin( double x)
```
 - The semantics of this code differ for each call


```
int ival; double dval;
dval = sin(ival) /* not coerced with 1st def */
```

Type Checking Parameters

- C99 and C++ require formal parameters in prototype form
 - But type checks can be avoided by replacing last parameter with an ellipsis


```
int printf(const char* fmt_string, ...);
```
- Python, Ruby, PHP, Javascript etc.
 - NO type checking

Multidimensional Arrays as Parameters

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function

Multidimensional Arrays C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

```
void fn(int matrix [][][10])
...
void fn(int *matptr, int nr, int nc){
...
*(matptr + (row*nc*SizeOf(int)) + col*SizeOf(int))
= x;
```

Multidimensional Arrays: Ada

- Ada - not a problem
 - Constrained arrays - size is part of the array's type
 - Unconstrained arrays - declared size is part of the object declaration

```
function matsum(mat : in mat_type) return Float is
sum: Float := 0.0;
begin
  for row in mat'range(1) loop
    for col in mat'range(2) loop
      sum := sum + mat(row, col);
    end loop;
  end loop;
return sum;
end matsum;
```

Multidimensional Arrays: Fortran

- Formal parameter that are arrays have a declaration after the header
 - For single-dimension arrays, the subscript is irrelevant
 - For multidimensional arrays, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function

Multidimensional Arrays: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (Length in Java, Length in C#) that is set to the length of the array when the array object is created

```
float matsum(float mat[][]) {
  float sum = 0.0;
  for (int r=0; r < mat.length; r++){
    for (int c=0; c < mat[r].length; c++){
      sum += sum + mat[r, c];
    }
  }
  return sum;
}
```

Design Considerations for Parameter Passing

- Two important considerations
 - Efficiency
 - One-way or two-way data transfer
- The usual conflict between safety and efficiency
 - Good programming practices suggest limited access to variables, which means one-way whenever possible
 - But pass-by-reference is more efficient to pass structures of significant size

Subprogram as Parameters

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
 1. Are parameter types checked?
 2. What is the correct referencing environment for a subprogram that was sent as a parameter?
- These are often simply referred to as “function parameters”
- Note that some languages (e.g., JavaScript, Lisp, Scheme) allow anonymous function parameters as well as named function parameters

Function Parameters: Type Checking

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked
- FORTRAN 95 type checks
- Ada does not allow subprogram parameters; an alternative is provided via Ada's generic facility
- Java does not allow method names to be passed as parameters

Function Parameters: Referencing Environment

- When a language allows nested subprograms what is the referencing environment?
 - *Shallow binding*: The environment of the call statement that enacts the passed subprogram
 - Most natural for dynamic-scoped languages
 - *Deep binding*: The environment of the definition of the passed subprogram
 - Most natural for static-scoped languages
 - *Ad hoc binding*: The environment of the call statement that passed the subprogram

Referencing Environment: Example

```
function sub1(){
  var x;
  function sub2(){
    alert(x);
  }
  function sub3(){
    var x;
    x = 3;
    sub4(sub2)
  }
  function sub4(subx){
    var x;
    x = 4;
    subx();
  }
  x = 1;
  sub3();
}
```

- Shallow binding
 - Reference to x is bound to local x in sub4 so output is 4
- Deep binding
 - Referencing environment of sub2 is x in sub 1 so output is 1
- Ad hoc binding
 - Referencing environment of sub2 is x in sub 3 so output is 3
- In this case Javascript uses ad hoc binding

Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
 - Every version of an overloaded subprogram has a unique protocol
 - Unique protocol means that the number, order, or types of parameters must differ or the return type must differ
- C++, Java, C#, and Ada include predefined overloaded subprograms and also allow users to write multiple versions of subprograms with the same name
- The problem of disambiguation is significant

Overloaded Names: Disambiguation

- Consider the following two function prototypes:

```
double fun (int a, double b)
double fun (double a, int b)
```
- How should the compiler resolve this call?

```
int z = (int)fun(1,2);
```
- If:
 - No method's parameter profile matches the profile of the call
 - Two or more can be matched through coercions
- Then:
 - Choose best match by ranking coercions
- Not a simple process!

User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- An Ada example

```
function "*" (A,B: in Vec_Type): return Integer
is
  Sum: Integer := 0;
begin
  for Index in A'range loop
    Sum := Sum + A(Index) * B(Index)
  end loop;
  return sum;
end "*";
...
c = a * b; -- a, b, and c are of type Vec_Type
```

Polymorphism and Generics

- From Greek, means having many forms
- Defn: A function or operation is *polymorphic* if it can be applied to any one of several related types and achieve the same result.
- Example: overloaded built-in operators and functions
 - + * / == != ...
 - + is also used for string concatenation in many langs
- An advantage of polymorphism is that it enables code reuse - why reinvent the wheel every time you need to sort an array of a different type?

Generic Subprograms

- A generic or polymorphic subprogram takes parameters of different types on different implementations or activations
 - We usually use polymorphic to refer to functions that have the same behavior for different types
 - Overloaded subprograms provide "ad hoc" polymorphism; no expectation of similar behavior
 - Dynamically typed languages such Python, Ruby and Javascript provide more general polymorphism as long as operators are defined for actual types passed
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*

Parametric Polymorphism

- A compile-time substitute for dynamic binding
- Type binding is deferred to compile time from programming time
- Still static rather than dynamic binding

Bubble Sort in Ada (Integers)

```
procedure sort (in out a: list) is
begin
  for i in a'first .. a'last-1 loop
    for j in i+1 .. a'last loop
      if a(i) > a(j) then
        begin t : integer := a(i);
          a(i) := a(j);
          a(j) := t;
        end if;
      end loop;
    end loop;
  end sort;
```

Generic Ada Sort

```
generic
  type element is private;
  type list is array(natural range <>) of element;
  with function ">"(a, b : element) return boolean;
package sort_pck is
  procedure sort (in out a : list);
end sort_pck;
package body sort_pck is
  procedure sort (in out a : list) is
  begin
    for i in a'first .. a'last - 1 loop
      for j in i+1 .. a'last loop
        if a(i) > a(j) then
          declare t : element;
          begin
            t := a(i); a(i) := a(j); a(j) := t;
          end;
        end if;
      end loop;
    end loop;
  end sort;
end sort_pck;
```

Instantiation of Generics

```
package integer_sort is
  new generic_sort( Integer, ">" );
package integer_sort is
  new generic_sort( Integer, ">" );
```

A Generic Stack (Ada 83 LRM)

```
generic
  SIZE : POSITIVE;
  type ITEM is private;
package STACK is
  procedure PUSH(E : in ITEM);
  procedure POP (E : out ITEM);
  OVERFLOW, UNDERFLOW : exception;
end STACK;
```

A Generic Stack (Ada 83 LRM)

```
package body STACK is
  type TABLE is array (POSITIVE range <>) of ITEM;
  SPACE : TABLE(1 .. SIZE);
  INDEX : NATURAL := 0;
  procedure PUSH(E : in ITEM) is
  begin
    if INDEX >= SIZE then
      raise OVERFLOW;
    end if;
    INDEX := INDEX + 1;
    SPACE(INDEX) := E;
  end PUSH;
  procedure POP(E : out ITEM) is
  begin
    if INDEX = 0 then
      raise UNDERFLOW;
    end if;
    E := SPACE(INDEX);
    INDEX := INDEX - 1;
  end POP;
end STACK;
```

Instantiating and Using Generic Stack

```
package STACK_INT is new STACK
  (SIZE => 200, ITEM => INTEGER);
package STACK_BOOL is new STACK(100, BOOLEAN);

. . .

STACK_INT.PUSH(N);
STACK_BOOL.PUSH(TRUE);
```

C++ Templates

- Basic implementation mechanism similar to macro expansion

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a > b)? a : b;
  return (result);
}
int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i, j);
  n=GetMax<long>(l, m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Generics through subclassing

- Most OO languages such as Java have collection classes that store Objects (and every object is a subclass of Object)
- Object variables are pointers or references
- Easy to implement generic collection classes

A Simple Stack in Java

- Note inner class Node (nested scope)

```
public class Stack {
    private class Node {
        Object val;
        Node next;
        Node(Object v, Node n) {
            val = v; next = n;
        }
    }
    private Node stack = null;
    ...
    public void push(Object v) { stack = new Node(v, Stack); }
}
```

Java Generics using Interface

- A Generic sort:

```
public static void sort (Comparable [] a) {
    for (int i = 0; i < a.length; i++)
        for (int j = i+1; j < a.length; j++)
            if (a[i].compareTo(a[j]) > 0 {
                Comparable t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}

public interface Comparable {
    public abstract int compareTo(Object o);
}
```

Ada Generics

- Ada
 - Versions of a generic subprogram are created by the compiler when explicitly instantiated by a declaration statement
 - Generic subprograms are preceded by a `generic` clause that lists the generic variables, which can be types or other subprograms

Generic Subprograms (continued)

- C++
 - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the `&` operator
 - Generic subprograms are preceded by a `template` clause that lists the generic variables, which can be type names or class names

Generic Subprograms (continued)

- Java 5.0
 - Differences between generics in Java 5.0 and those of C++ and Ada:
 1. Generic parameters in Java 5.0 must be classes
 2. Java 5.0 generic methods are instantiated just once as truly generic methods
 3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
 4. Wildcard types of generic parameters

Generic Subprograms (continued)

- C# 2005
 - Supports generic methods that are similar to those of Java 5.0
 - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type

Coroutines

- A *coroutine* is a subprogram that has multiple entry points and controls them itself
 - Coroutines maintain state between activations
- Also called *symmetric control*: caller and called coroutines are on a more equal basis
- Some programming languages with direct support for coroutines
 - C# F# Go Haskell Javascript
 - Lua Perl Prolog Python Ruby

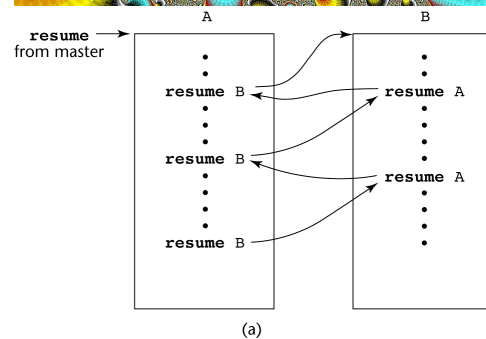
Coroutines

- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

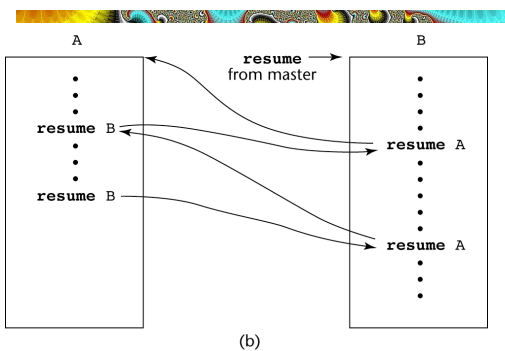
Coroutine Applications

- Card Games: each coroutine represents one player
- Producer-Consumer: one routine produces items and queues them; the other consumes them and removes them (ex. Mail program)
- Efficient traversal of complex data structures
- Note that coroutines are very similar to multiple threads and can be used for many of the same applications
 - But coroutines can never execute in parallel

Coroutines Illustrated: Possible Execution Controls



Possible Execution Controls



Execution Controls with Loops

