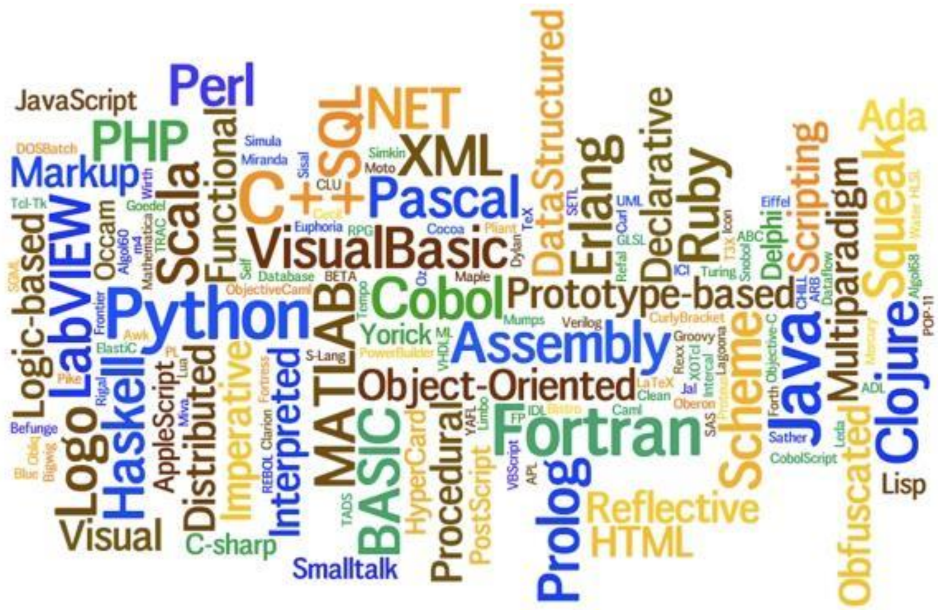


Programming Languages Subprograms

Programming Languages
Module 8 (Chapter 9 - PART 1)

Dr. Tamer ABUHMED
College of Computing

SUNG KYUN KWAN UNIVERSITY

Topics to be covered

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Overloaded Subprograms
- Generic Subprograms
- Design Issues for Functions
- User-Defined Overloaded Operators
- Closures
- Coroutines

Part 1

Fundamentals of Subprograms

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates

1

2

```
int addition(int a, int b)
{
int r;
r = a + b;
return r;
}


int main()
{
    int z;
z = addition(5, 3);
cout << "The result is " << z;
}
```

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
  - In Python, function definitions are executable; in all other languages, they are non-executable
  - In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of `Object`. They can be called without an object, like a function
  - In Lua, all functions are anonymous
- 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

formal parameters

```
int addition(int , int );

int main()
{ int a, b, z;
a = 5; b = 3;
z = addition(a , b);
cout << "The result is " << z;
}

int addition(int a, int b)
{
int r;
r = a + b;
return r;
}
```

actual parameters

Python function definitions are executable

```
def func():
    print('func()')
    i = 1
    if i == 1:
        def func1():
            print('func1')
    else:
        def func2():
            print('func2')
    func1()
    func2()
func()
```

def func2() will nover executed, so func2 is not exist

```
func2()
UnboundLocalError: local variable 'func2' referenced before assignment
```

Actual/Formal Parameter Correspondence

- 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

Binding of actual parameters to formal parameters (Positional) C++

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
- In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)

```
void point(int x = 3, int y = 4);
void main(){
point(1, 2); // calls point(1,2)
point(1);   // calls point(1,4)
point();    // calls point(3,4)
 }
```

```
void summation(int x, int y = 3 , int z);
//error : 'summation' : missing default parameter for parameter 3
```

```
float compute_pay(float income, float tax_rate, int exemptions = 1);
void main(){
compute_pay(1200.5, 0.02);  // exemptions = 1
 }
```

Binding of actual parameters to formal parameters (Keywords) Python

```
def display (ID , Name, Age = 0, Family = [] ):
    print ("ID: ", ID)
    print ("Name: ", Name)
    print ("Age: ", Age)
    for i in Family:
        print ("Family member: ", i)
display(ID = 154, Name = "Alice", Age = 23, Family = ["Father, Mother, Brother"])
```

```
def display (ID = 0, Name = "Alice",*, Age, Family ):
    print ("ID: ", ID)
    print ("Name: ", Name)
    print ("Age: ", Age)
    for i in Family:
        print ("Family member: ", i)

display(154 , "Alice", Age = 23, Family = ["Father, Mother, Brother"])
```

Positional

Keywords arguments

Variable subprogram Parameter Lists

- Variable numbers of parameters
  - C++ functions 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 va_list
- As C++, Java also support passing a variable number of arguments to methods. However, variable parameter in C++ must be as an argument to the end of the list

C++	Java

`#include <stdargs.h>`
`void F(int first, ...) {`
` int i = first;`
` va_list marker; // retrieve arguments`
` va_start(marker, first);`
`  while(i != -1)`
` i = va_arg(marker, int); va_end(marker);`
`  }`
	`void F(int... args)`
	`{`
	` for(int i : args) {}`
	`}`

Variable subprogram Parameter Lists

- Python support passing variable *positional* and *keyword* arguments

```python
# variable positional arguments
def function(*args):
    for i in args:
        print(i)


function(564,64,6,6)
```

```python
def function2(**kwargs):
    for i in kwargs.items():
        print(i)

function2(x = 2, y = 7, z = 8)

def function3(*args,**kwargs):
    for i in args:
        print(i)
    for i in kwargs.items():
        print(i)
function3(8,x = 2, y = 7, z = 8)
```

Procedures and Functions

- There are two categories of subprograms
  - *Procedures* are collection of statements that define parameterized computations
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
    - They are expected to produce no side effects
    - In practice, program functions have side effects

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are 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 subprogram be generic?
- If the language allows nested subprograms, are closures supported?

Local Referencing Environments

- Local variables can be stack-dynamic
  - 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 Referencing Environments: Examples

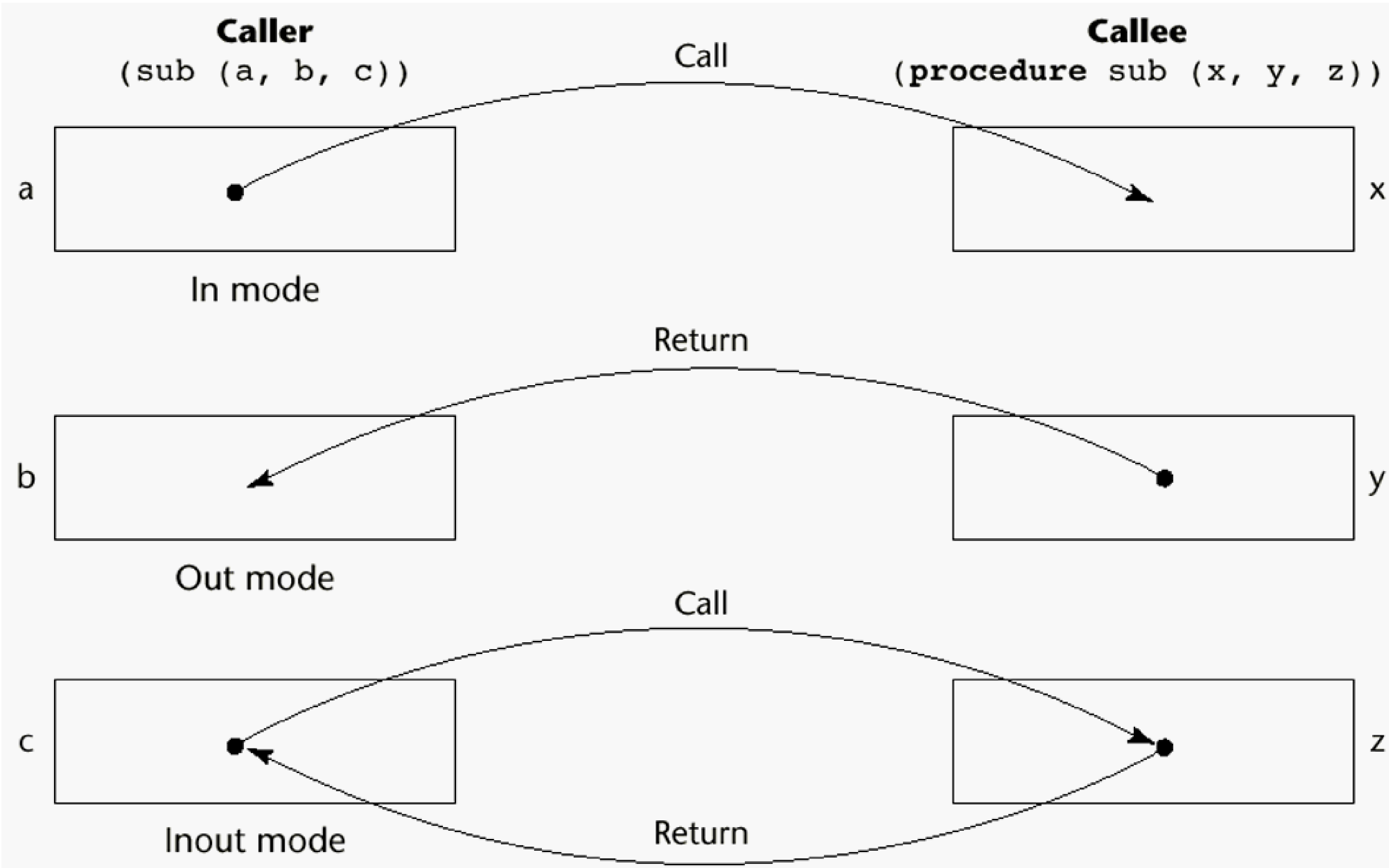
- In most contemporary languages, locals are stack dynamic
- In C-based languages, locals are by default stack dynamic, but can be declared `static`
- The methods of C++, Java, and C# only have stack dynamic locals
- In Lua, all implicitly declared variables are global; local variables are declared with `local` and are stack dynamic

Semantic Models of Parameter Passing

- Formal parameters are characterized by one of three distinct semantics models:

1) In mode : They can receive data from the corresponding actual parameter;
2) Out mode: they can transmit data to the actual parameter; or
3) Inout mode: they can do both.

Models of Parameter Passing



Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

```
int main() {
        int x = 10, y = 20;
        swap(x, y);
        /* no change!
      x == 10, y == 20 */
```

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 is returned to the caller, by physical move
  - Require extra storage location and copy operation
- Potential problems:
  - `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of `p1`
  - `sub(list[sub], sub);` Compute address of list[sub] at the beginning of the subprogram or end?

```
void DoIt(out int x, int index){
x = 17;
index = 42;
}
sub = 21;
f.DoIt(list[sub], sub);
```

```
void Fixer(out int x, out int y) {
x = 17;
y = 35;
}
. . .
f.Fixer(out a, out a);
```

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

```
fun(total, total);  fun(list[i], list[j];  fun(list[i], i);
```

```
void swap(int& i, int& j)
{
    int t = i;
    i = j;
    j = t;     }
```

```
int main() {
      int x = 10, y = 20;
      swap(x, y);
      /* changed!
     x == 10, y == 20 */
```

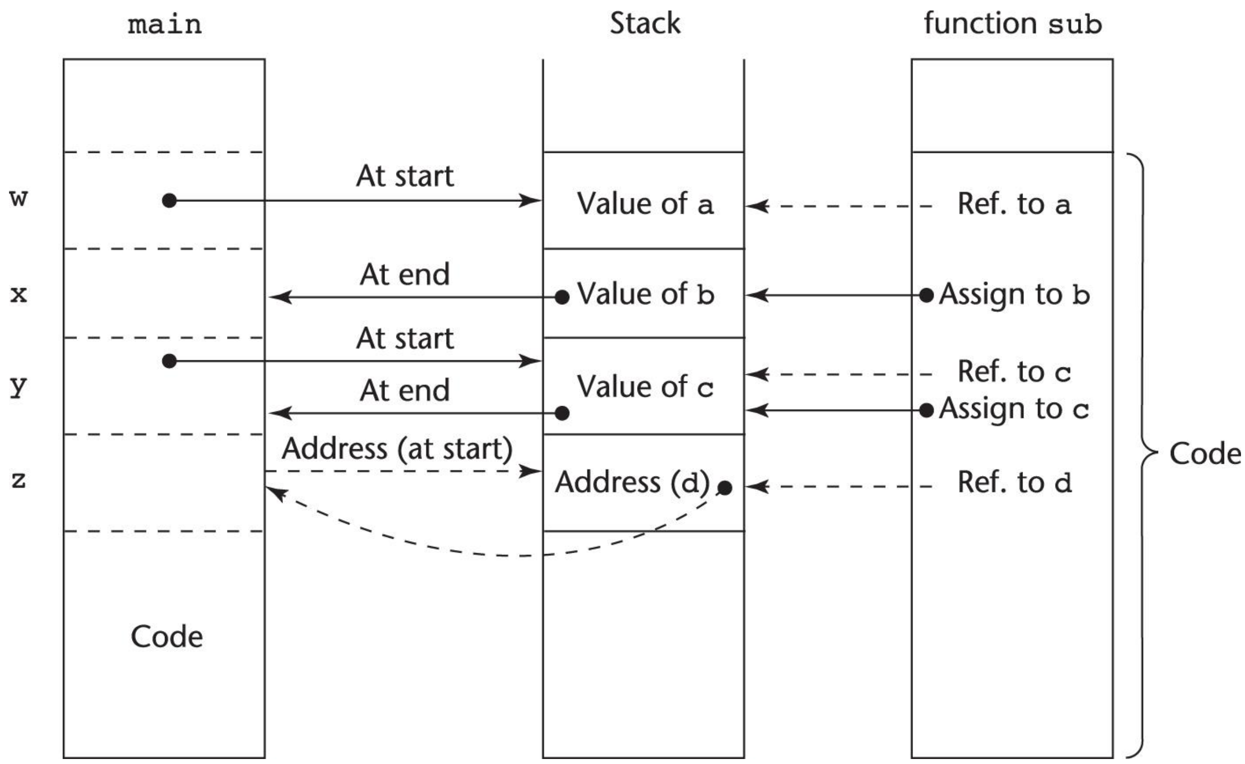
Pass-by-Name (Inout Mode)

- 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
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated

Implementing Parameter-Passing Methods

- In most languages 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

Implementing Parameter-Passing Methods



Function header: `void sub(int a, int b, int c, int d)`
Function call in main: `sub(w, x, y, z)`
(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)

Parameter Passing Methods of 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
- Java
  - All parameters are passed by value
  - Object parameters are passed by reference
- 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 Methods of Major Languages (continued)

- Fortran 95+
  - Parameters can be declared to be in, out, or inout mode
- 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#, except that either the actual or the formal parameter can specify ref
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal

Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
-