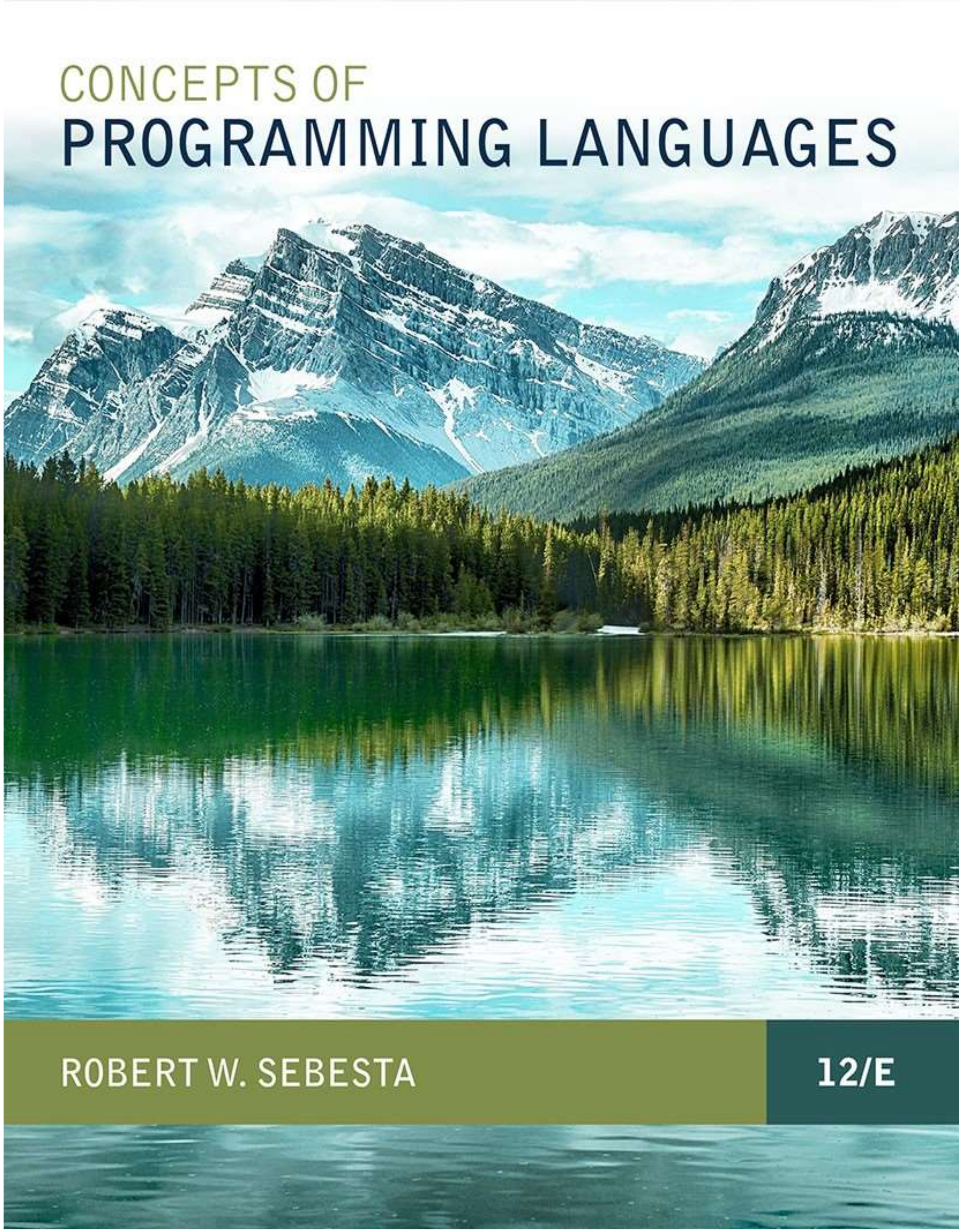
Lecture 7

Ch. 9: Subprograms

Ch. 10: Implementing Subprograms (brief coverage)



ISBN 0-321-49362-1

Topics

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Semantic Modes
- Conceptual Models
- Implementations
- Subprograms as Parameters
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded Operators

Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
    - Subprograms -- Discussed in this Lecture
  - Data abstraction
    - Emphasized since 1980s
    - ADTs -- Discussed at length in Lecture 11

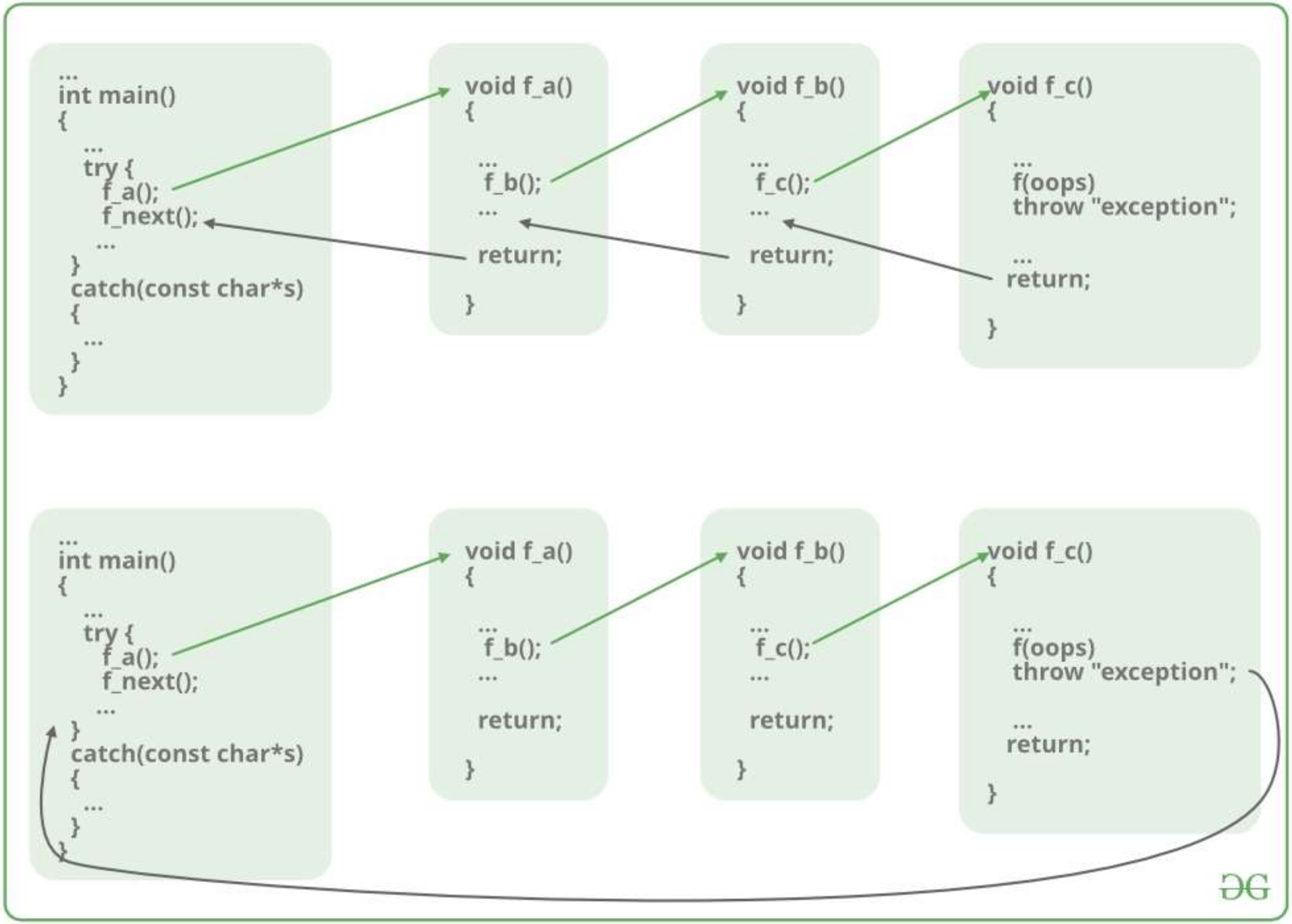
Process Abstraction -- Illustration

```
 main () {
     int a, b, c, d;
     print(a);
     print(“___”);
     print(“****”);
     print(b);
     print(“___”);
     print(“****”);
         …
}
```

```
display(int x) {
        print(x);
        print(“___”);
        print(“****”);
}
main() {
        int a, b, c, d;
        display(a);
        display(b);
        display(c);
        display(d);
}
```

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
  - Unless exception occurs
  - See illustration next slide



```
...
int main()
{
    ...
    try {
      f_a();
      f_next();
      ...
    }
    catch(const char*s)
    {
      ...
    }
}
```

```
void f_a()
{

   ...
   f_b();
   ...

   return;

}
```

```
void f_b()
{

   ...
   f_c();
   ...

   return;

}
```

```
void f_c()
{

    ...
    f(oops)
    throw "exception";

    ...
   return;

}
```

```
...
int main()
{
    ...
    try {
      f_a();
      f_next();
      ...
    }
    catch(const char*s)
    {
      ...
    }
}
```

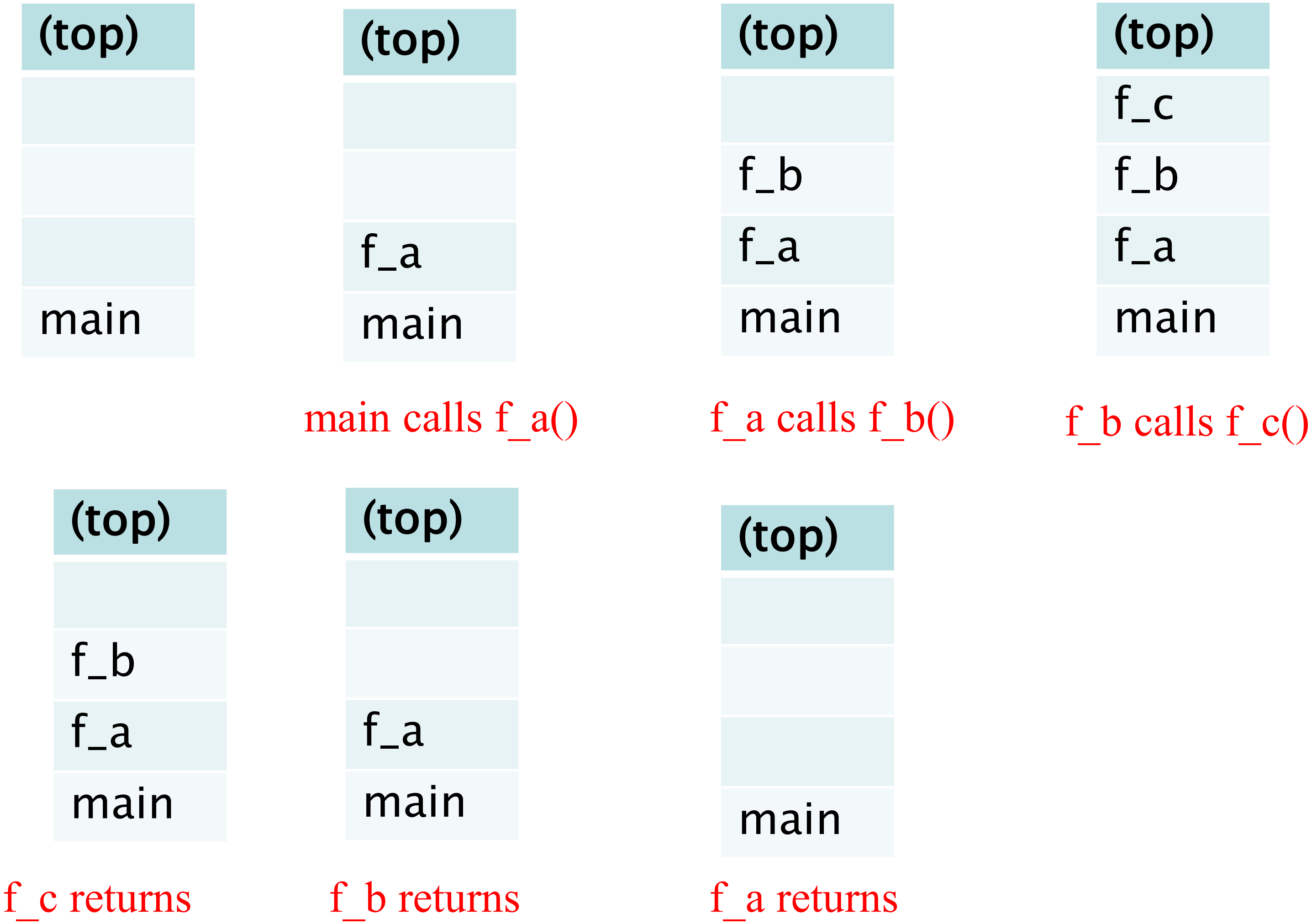
```
void f_a()
{

   ...
   f_b();
   ...

   return;

}
```

```
void f_b()
{

   ...
   f_c();
   ...

   return;

}
```

```
void f_c()
{

    ...
    f(oops)
    throw "exception";

    ...
   return;

}
```

Geeksforgeeks.org

Run-time stack



(top)

main

(top)

f_a

main

main calls f_a()

(top)

f_b

f_a

main

f_a calls f_b()

(top)

f_c

f_b

f_a

main

f_b calls f_c()

(top)

f_b

f_a

main

f_c returns

(top)

f_a

main

f_b returns

(top)

main

f_a returns

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)

- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
  - Function declarations in C and C++ are often called *prototypes*
- 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

Parameters and Arguments

```
void display (int x, double y) { … return; }

int main () {
        int count; double sum;
        display(count, sum); …
}
```

By position (in the above call)

```
        count -> x  sum -> y
```

By name/keyword:

```
        display (sum => y, count => x);  //Ada-like
```

Parameters: Default Values and Variable Length

- In certain languages (C++, Python, etc.), 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)

- Variable length argument
  - A feature that allows a function to receive any number of arguments, e.g. in the situations for a function to handle variable number of arguments according to requirement such as
    - 1) Sum of given numbers. 2) Minimum of given numbers.
  - C++, Java, Python etc. support this feature

Example: default (parameter) values

```
int cost(int items, double taxRate = 9.0, int base = 0) {
        return items * (1+taxRate) + base;
}
```

```
cout << cost(4);
cout << cost(4, 10.0);
cout << cost (4, 8.0, 1);
```

Variable Length Arguments: Examples

```
//C# code segment
static int Multiply(params int[] b) {
    int mul =1;
    foreach (int a in b) {        mul = mul*a;      }
    return mul;
}

//function call accepts zero or more arguments
 int mulVal1 = Multiply(5);
 int mulVal2 = Multiply(2, 3, 10);
```

Discussion Questions:

(1) How Java (Java's syntax) supports variable length arguments?

(2) Java vs. C# in support of variable length arguments?

Procedures and Functions

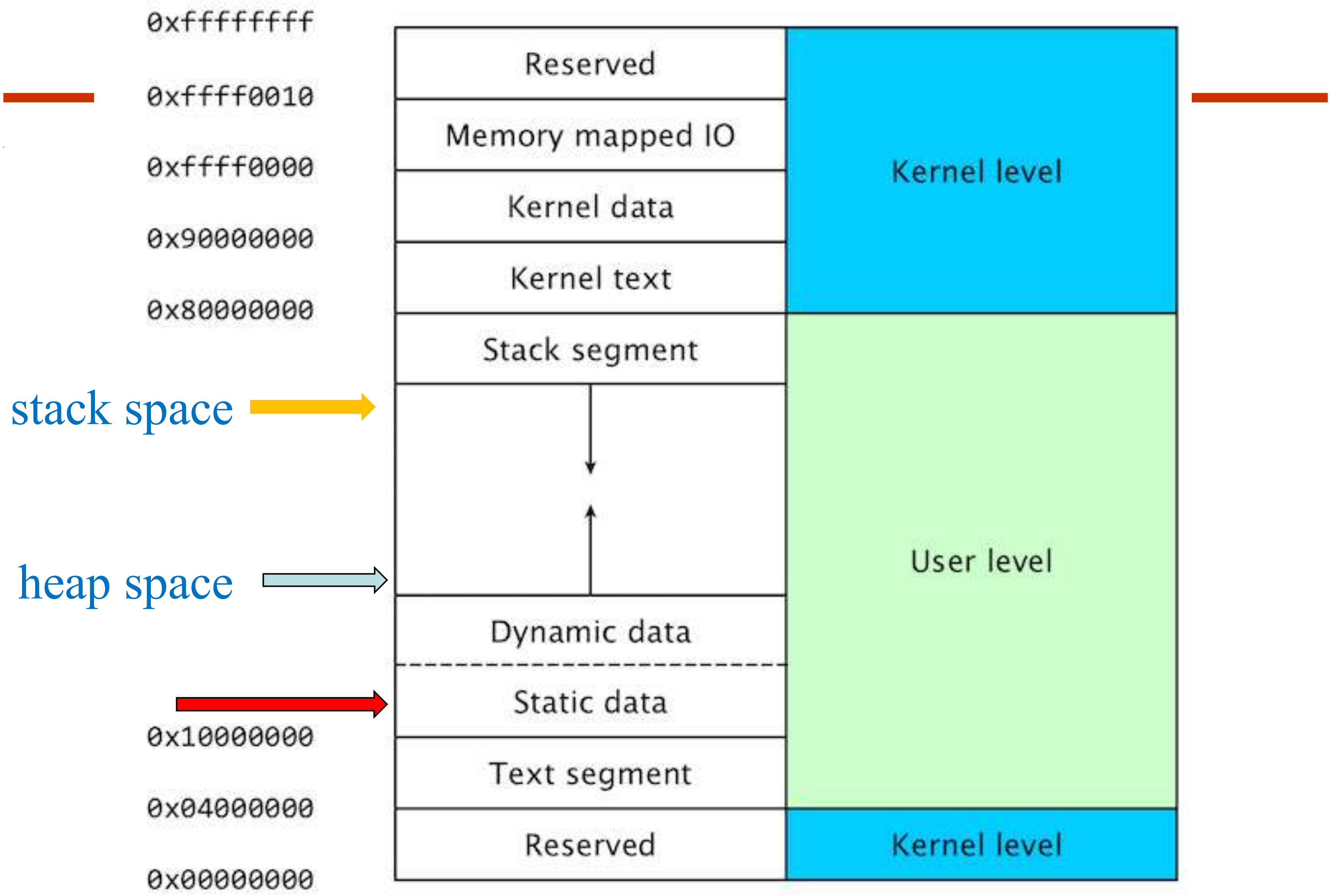
- There are two categories of subprograms
  - *Procedures* are collection of statements that define parameterized computations
    - Procedure doesn’t return a value
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions – a return value is expected.
    - Functions are expected to produce no side effects
    - In practice, program functions have side effects

Design Issues for Subprograms

- Local variables: static or dynamic variables?
  - Scope, referencing environment?
- Can subprogram definitions be nested?
  - i.e. 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?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?

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
  - In most contemporary languages, locals are stack dynamic
    - e.g in Java, C++, Python methods
- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables
  - In C-based languages, locals are by default stack dynamic, but can be declared static



0xffffffff

0xffff0010

0xffff0000

0x90000000

0x80000000

0x10000000

0x04000000

0x00000000

Reserved

Memory mapped IO

Kernel data

Kernel text

Stack segment

Dynamic data

Static data

Text segment

Reserved

Kernel level

User level

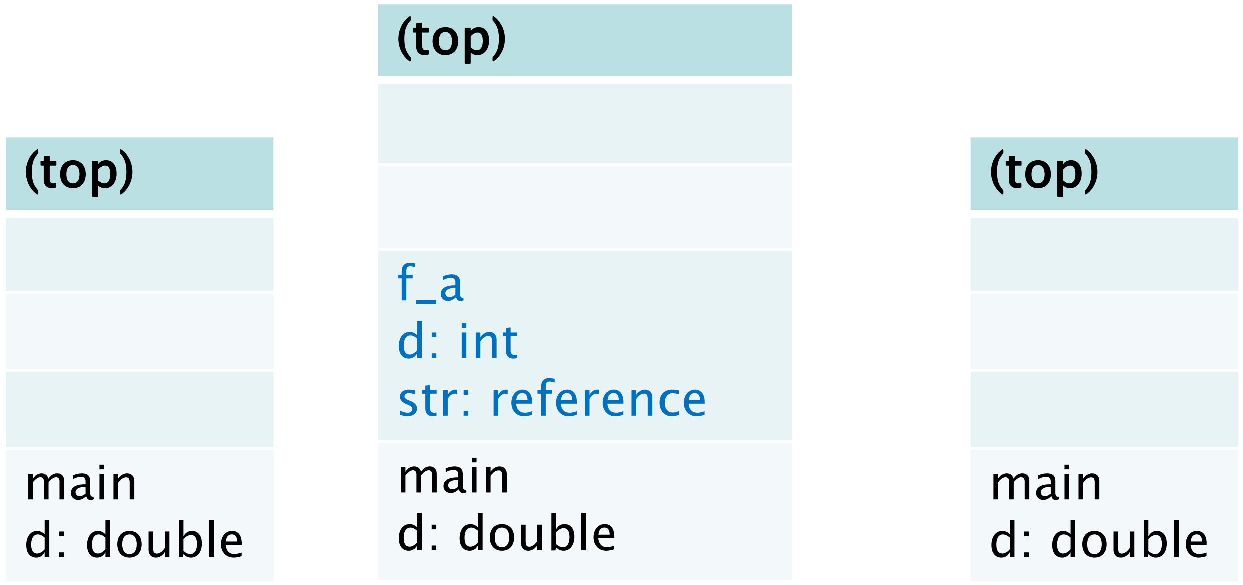
Kernel level

stack space

heap space

Figure source: it.uu.se

Run–time stack: Local variables



(top)

main
d: double

(top)

f_a
d: int
str: reference

main
d: double

(top)

main
d: double

main calls f_a()

f_a() returns

Subprogram Basics: Summary

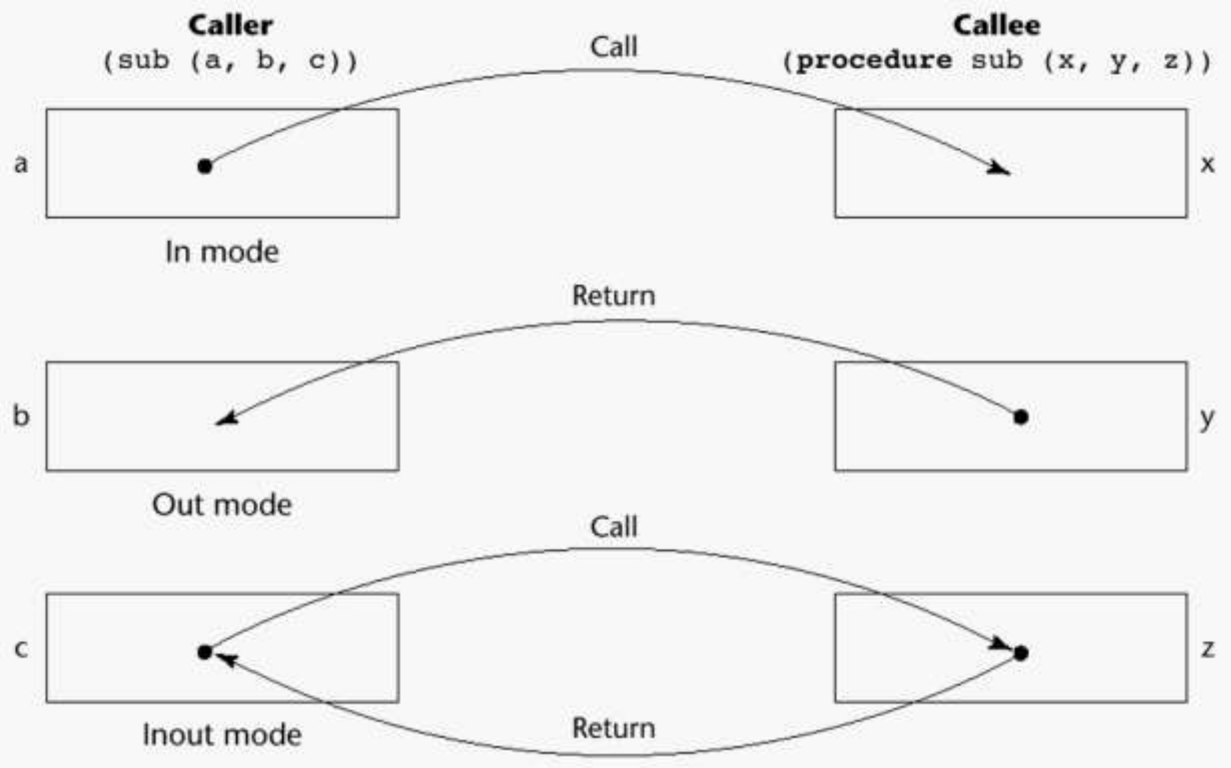
- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Parameters can be bound by position or keywords
- Some languages allow default parameters and/or variable length parameters
- Next lecture will discuss more about parameter passing

Parameter Passing

- Parameter passing is the mechanism used to pass arguments (actual parameters) to parameters (formal parameters, i.e. dummy variables) defined a procedure (subroutine) or function.
  - The most common methods are to pass the value of the actual parameter (call by value), or to pass the address of the memory location where the actual parameter is stored (call by reference).

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode



Example: Parameter Passing modes

```
void modeDemo (int item, int & count, double & cost) {. //C++
        cost = item * count * 1.3;
        count ++;
        return;
}

int unit = 5; double total; int amount = 8;
modeDemo (unit, amount, total);
```

Theoretically: unit -> item (in); amount <-> count (in out);
total <- cost (out)

C# (not C++) clearly indicates the out mode.

Conceptual Models of Transfer

- Arguments (actual parameters) are passed to formal parameters in either of the following ways:
  - Physically move a value
  - Move an access path to a value

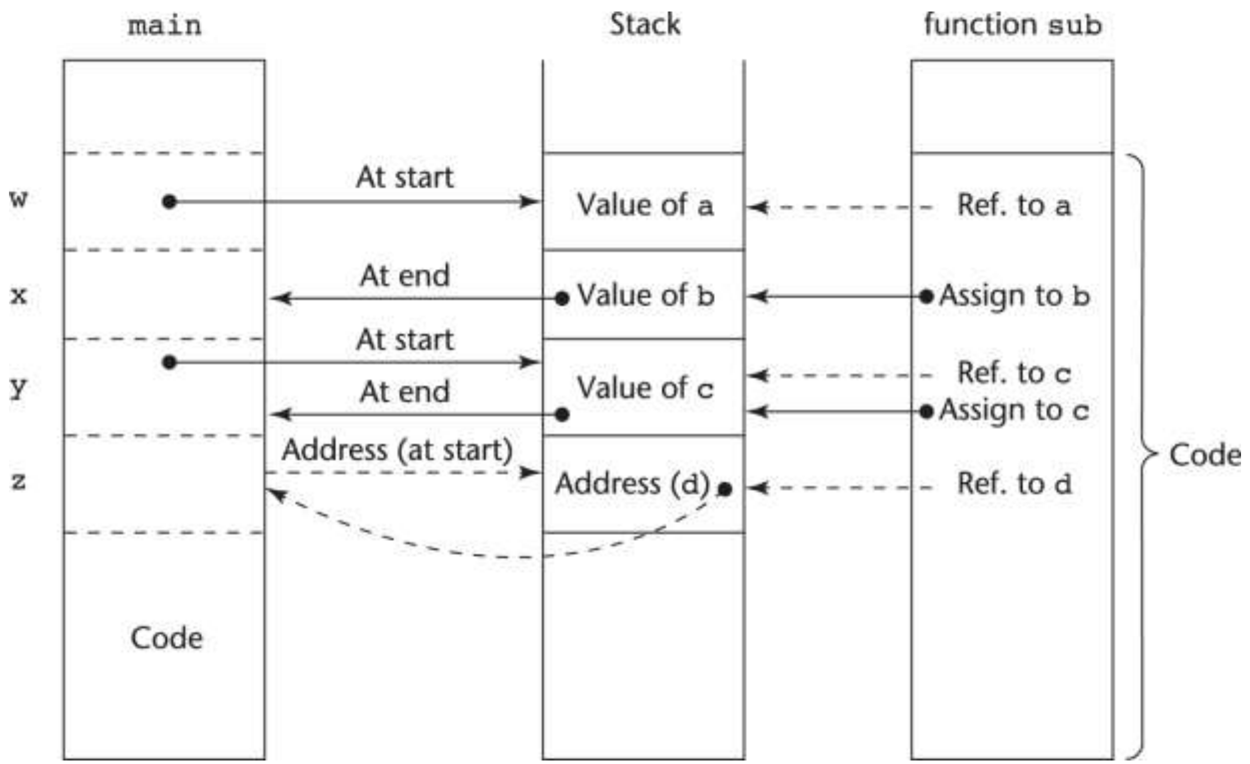
- Practically we have to the following models
  - Pass by value
  - Pass by result
  - Pass by value-result
  - Pass by reference
  - Pass by name

Trends of parameter passing:

5 models => 2 models (by value & reference)
=> 1 model (pass by value)

New model: Pass by assignment (Python etc.)

Implementation on Run-time Stack: Example



Function header: `void sub(int a, int b, int c, int d)`

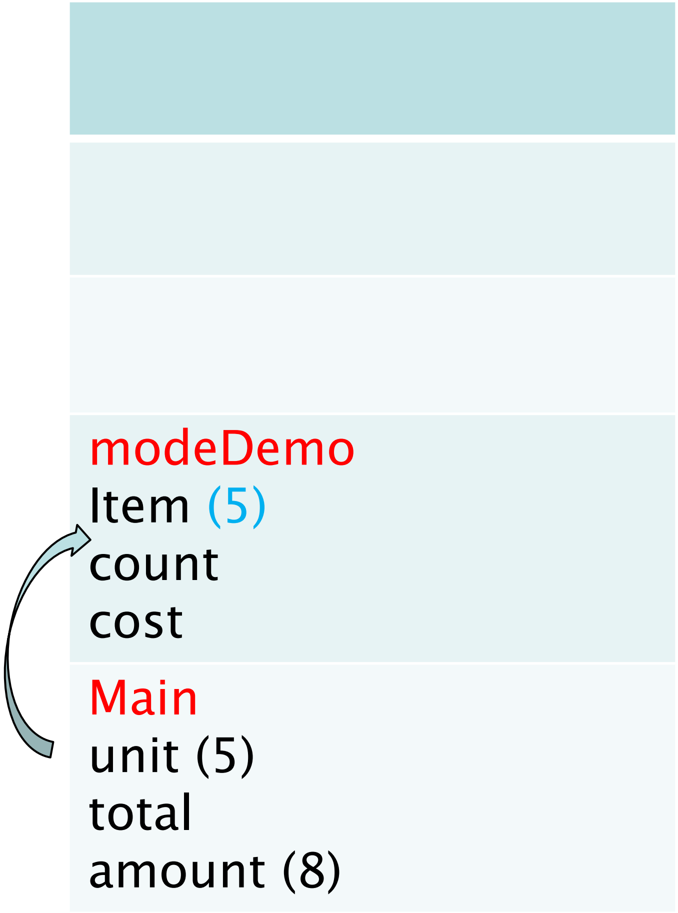
Function call in main: `sub(w, x, y, z)`

(assume: pass w by value, x by result, y by value-result, z by reference)

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - implemented by copying (physically moving a value)
  - 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)
  - Popularly used in scenarios where the subprogram is only "using" the parameter for some computation, not changing it

Pass by value: Illustration



modeDemo
Item (5)
count
cost

Main
unit (5)
total
amount (8)

2 separate locations;
value: copy

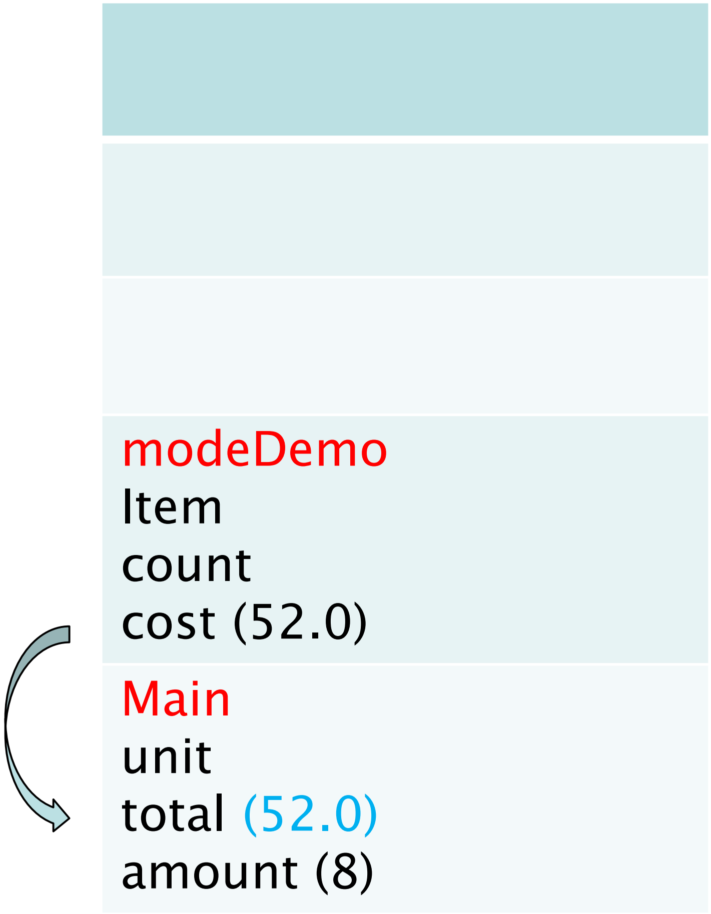
Pass-by-Result (Out Mode)

- When a parameter is passed by result,
  - no value is transmitted to the subprogram;
  - formal parameter acts as a local variable with its value transmitted to caller's actual parameter when control is returned to the caller, by physical move
- Potential problems

```
void sub (x, y ) {     //assume x and y are pass-by-result parameters
        x = 5; y = 6;
        return;
}
```

  - sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1
  - sub(list[a], a); Compute address of list[a] at the beginning of the subprogram or end?
- No longer practically used, though variants exist
  - e.g. C#'s out parameter

Pass by value: Illustration



modeDemo
Item
count
cost (52.0)

Main
unit
total (52.0)
amount (8)

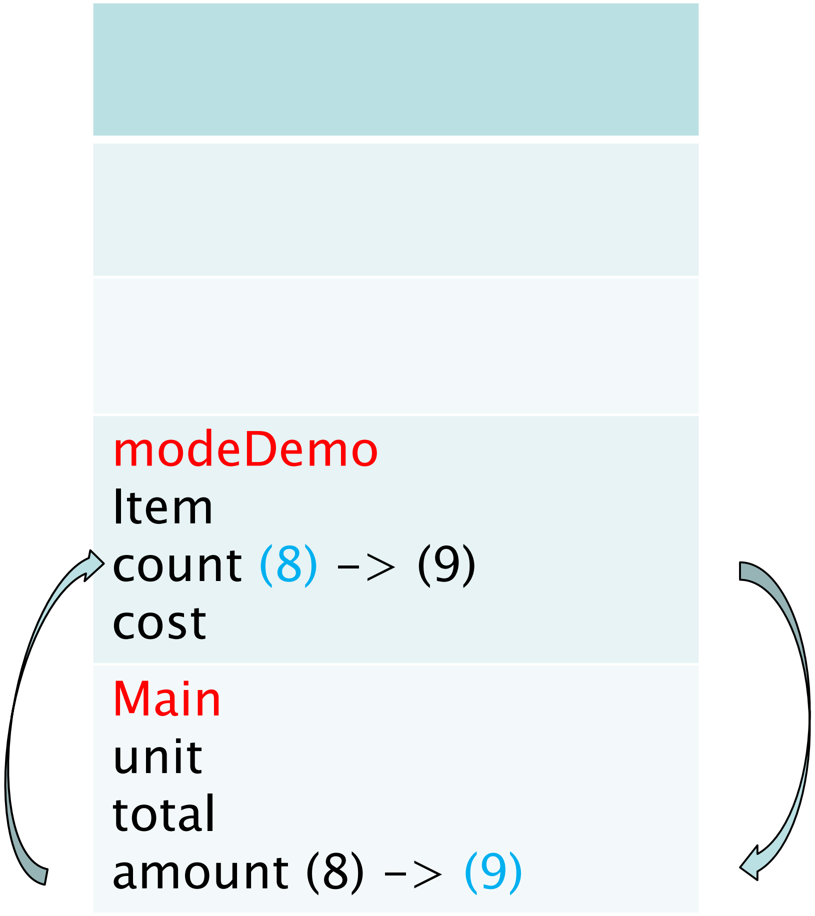
2 separate locations;
value: copy

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

- Pass-by-reference, another method of inout mode, is more efficient than pass-by-value-result, thus popularly used.

Pass by value-result: Illustration



modeDemo

Item

count (8) -> (9)

cost

Main

unit

total

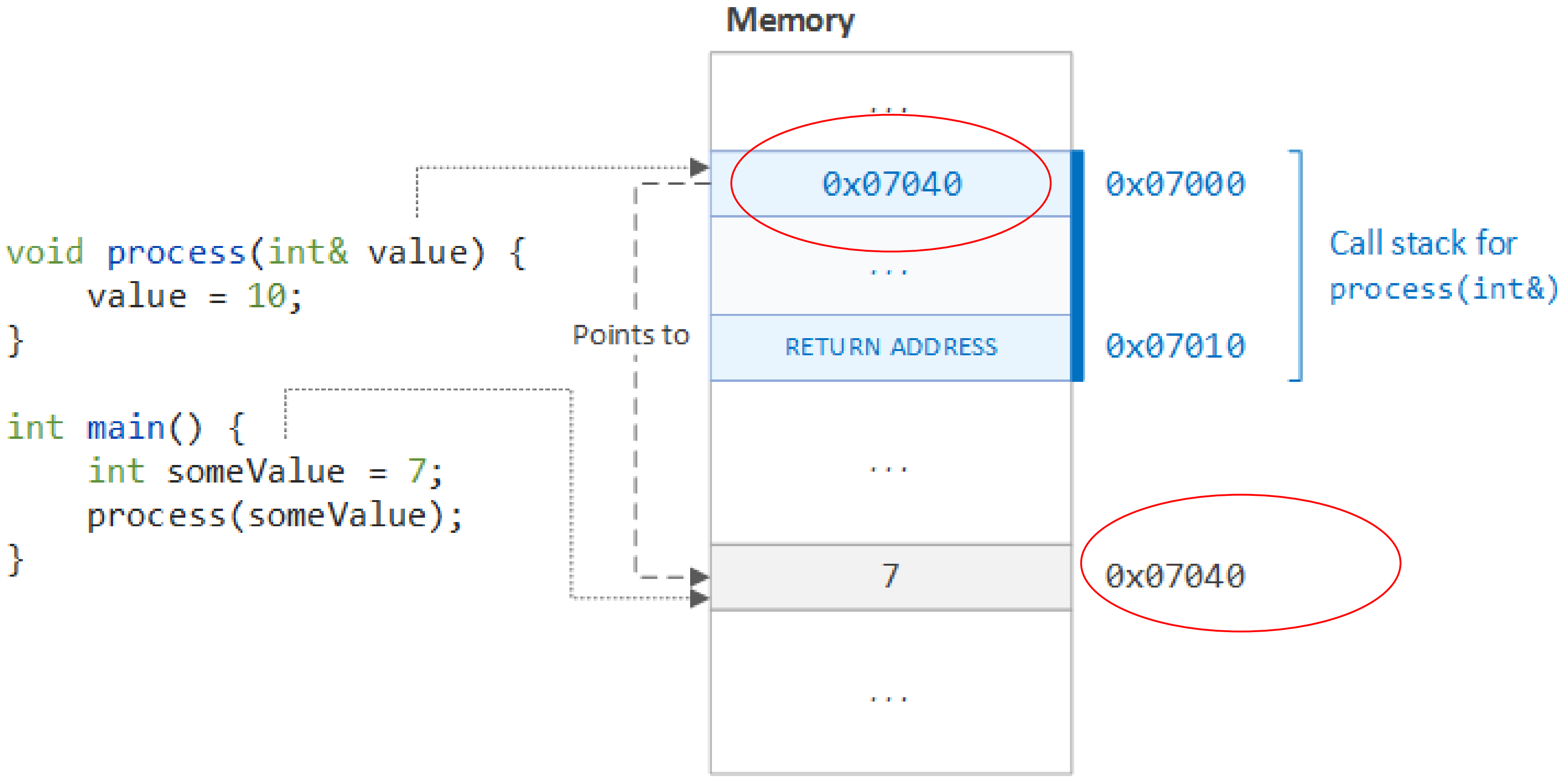
amount (8) -> (9)

2 separate locations; value: copy over and copy back

Pass-by-Reference (Inout Mode)

- Pass an access path (i.e. address)
  - Actual parameter and formal parameter shares memory locations, i.e. becomes *aliases*
  - 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), e.g.

    ```
    void fun (x, y) { … }    //assume x, y call-by-ref parameters
    fun(total, total);  fun(list[i], list[j]);  fun(list[i], i); //collisions
    ```
- Efficient and popularly used method
  - To resolve collision issues: use with caution



```
void process(int& value) {
    value = 10;
}

int main() {
    int someValue = 7;
    process(someValue);
}
```

The memory address of someValue is copied to value location; thus value is a reference to someValue.

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, e.g.

Example: Pass-by-name Elegancy

```
real procedure Sum(j, lo, hi, Ej);   //ALGOL 60 program
          value lo, hi;                //value paramters
          integer j, lo, hi;   real Ej;   //parameter type declaration
begin
          real S;   //local variable
          S := 0;
          for j := lo step 1 until hi do
                    S := S + Ej;
                    Sum := S
end;

Sum(i, 1, n, x[i]);            //x[1]+x[2]+...+x[n]
Sum(i, 1, n, x[i]*y[i]);       //x[1]*y[1]+x[2]*y[2]+...+x[n]*y[n]
// in each loop iteration,  Ej is re-evaluated as it's literally substituted by actual parameter
```

Parameter Passing: Example

```
//C#

void Method(ref int arg) {
    arg = arg + 10;
}

int number = 25;
Method(ref number);
Console.WriteLine(number);
```

Pass-by-Assignment

- Python example

https://medium.com/school-of-code/passing-by-assignment-in-python-7c829a2df10a

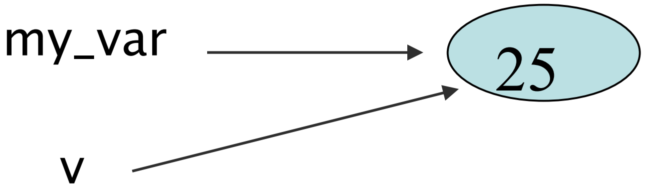
```
my_var = 25
def my_method(v):
    v += 10
    return
my_method(my_var)
print (my_var)
```

- Pass-by-reference vs. Pass-by-assignment

Note: In Python 25 is an object!

- By assignment

After parameter passing



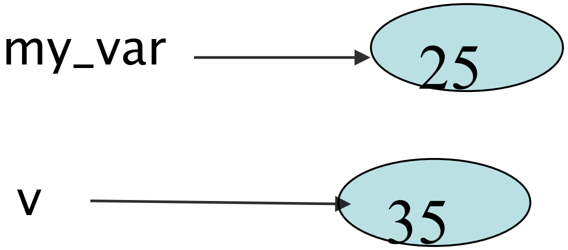
my_var

v

25

Object 25 is immutable!

After v = v + 10



my_var

v

25

35

Pass-by-Assignment

- Python example

```
my_var = [25, 10]
def my_method(v):
        v[0] = v[0] + 10
        return
my_method(my_var)
print (my_var[0])
```

Pass-by-Assignment

- Python example

```
my_var = [25, 10]
def my_method(v):
        v[0] = v[0] + 10
        return
my_method(my_var)
print (my_var[0])
```

List elements are mutable!

Discussion Questions

- What model(s) Java use for parameter passing?

- In Java, how to implement parameter passing of a dynamically allocated heap array?

  e.g.

  ```
  public static void f (int [] data) { data[0] = 1; return}

  int[] my_data = new int [5];
  for (int j = 0; j< 5; j++) my_data[j] = 10;
  f(my_data);
  //what will be my_data[0] now?
  ```

- Illustration of above parameter passing?

Parameter Passing Methods of Major Languages

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters

- C++
  - Pass-by-value
  - Pass-by-reference (& parameter)
    - A special pointer type called reference type is introduced in C++

- Java
  - All non-object parameters are passed are passed by value
    So, no method can change any of these parameters
  - Object parameters are passed by reference via object references
    - No language supported pass-by-reference method needed.

Parameter Passing Methods of Major Languages (continued)

- Fortran 95+ and Ada
  - Parameters can be declared to be in, out, or inout mode
    - Actual models (by copy or access path) may vary by implementation
- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref
  - out parameter is same as ref except no initial needed.
- Python and Ruby
  - Use pass-by-assignment (all data values are objects);
  - The actual parameter is assigned to the formal parameter
  - Mutable and immutable objects play important role
    - For immutable objects, the actual parameter will not change if formal parameter changes (see example next slide)

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: it is always required
- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables (i.e. object references) do not have types (objects do), so parameter type checking is not possible

Multidimensional Arrays as Parameters

- For most languages arrays are passed by reference
  - For one-dimensional array, a starting address of the array is passed
  - The length of the array may not known by the subprogram. That may cause memory leak problem.
    - C++ example in next slide

- If a multidimensional array is passed to a subprogram, the compiler needs to know the size and shape of that array to build the storage mapping function
  - A starting address and total number of elements not enough
    - Since a 2-D array is stored linearly in memory, for an array of 16 elements how can you tell if it's a 4 x 4 array or a 2 x 8 array?

Passing of 1-D array: C++ Example

```
#include <iostream>

void f(int x[]) {
    for (int i=0; i<8; i++)
        std::cout << x[i] << " ";
    std::cout<<"\n";
    return;
}
int main() {
  int a[10] ={1,1,1,1,1,1,1,1,1,1};
  int b[5] ={1,2,3,4,5};
  f(a);
  f(b);
  return 0;
}
```

Output:

```
1 1 1 1 1 1 1 1

1 2 3 4 5 0 -201043027 21990
```

//by programiz online compiler
//note: in call of f(b), 3 garbage values (0 -201043027 21990) added.

Multidimensional Arrays as Parameters: C and C++

- In C/C++ programmer is required to include the declared sizes of all but the first subscript in the formal parameter
  - e.g. for 3-D array, the 2nd and 3rd dimension must be given in formal parameter

```
void pass3D (int m3d [][20][30])
```

- Another solution: pass a pointer of the array and the sizes of the dimensions as additional parameters
  - the user must include in the code the storage mapping function in terms of the size parameters

Multidimensional Arrays as Parameters: Java and C#

- Arrays are objects
  - they are all single-dimensioned, but the elements can be arrays, i.e. multidimensional arrays are represented by array of 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

Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size
    - Programmer may use a const (constant) array that provides one-way access under pass-by-reference

Parameter Passing: Summary

- Three semantic modes of parameter passing: in mode, out mode, and inout mode
- The conceptual models of parameter passing
  - Pass by value
  - Pass by result
  - Pass by value-result
  - Pass by reference
  - Pass by name
  - Pass by assignment

Subprogram Names as Parameters

- It is sometimes convenient to pass subprogram names as parameters
- Very popular in functional programming
  - Higher order functions, or functional forms
  - Or, simply function as parameter

- Questions to consider
  - Why we'd have introduce this feature
  - Language support (syntax) of this feature
  - Pros and Cons
    - Which language(s) supporting it, which are not?

Motivation: A sort function in Java

```
import java.util.Arrays;
int[] arr = { 13, 7, 6, 45, 21, 9, 101, 102 };
Arrays.sort(arr);
```

- What will be the result? i.e. what order will the list be sorted into? Ascending? Descending?
  - By default: ascending order {6, 7, 9, 13, 21, 45, 101, 102}

- What about I want to sort into descending order?
  - Group Discussion
    - Java or any other language
    - Write code

Discussion: A sort function in Java

```
import java.util.Arrays;
int[] arr = { 13, 7, 6, 45, 21, 9, 101, 102 };
Arrays.sort(arr);
```

- What will be the result? i.e. what order will the list be sorted into? (default: Ascending)
- What about I want to sort into a different order?

```
Arrays.sort(arr, Collections.reverseOrder());
```

reverseOrder(): a function as parameter

- What about I'm sorting an array of objects into various different orders? E.g. score in descending? If same score, name in ascending?

arrS = {("John", 89), ("Marv", 95), ("Jess", 89), ("Terry", 92), …}

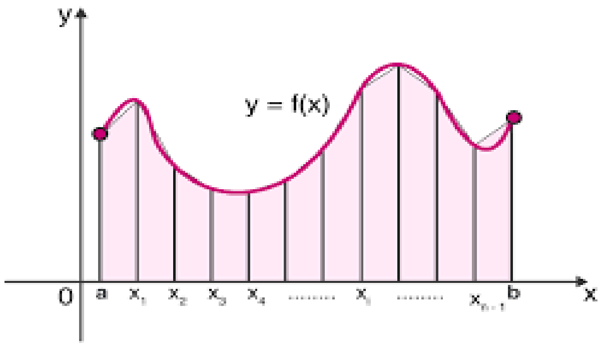
=> {("Marv", 95), ("Terry", 92"), ("Jess", 89), ("John", 89), …}

Function as parameter in sort (pseudo code)

```
void bubbleSort(int a[], int n, order_fun)
{
    int i, j;
    for (i = 0; i < n-1; i++)
      for (j = 0; j < n-i-1; j++)
        if (order_fun(a[j], a[j+1]))      //any order you may define
            swap(a[j], a[j+1]);    //swap two array elements
}
boolean order1 (x, y) { return x < y; } //descending
bubbleSort(arr, 8, order1);
boolean order2 (x, y) {
   return (x.score == y.score ? x.name > y.name : x.score < y.score);}
bubbleSort(arrS, Size, order2);
```

Subprograms as parameters: Applications

- Sorting
  - Sorting in ascending order, descending order
  - Sorting student objects based on names in ascending order, based on GPA in descending order
- Map functions (in map-reduce)
  - popular in cloud computing, further discussion in Chapter 15
- Integration
  - Integrating sin(x) in [a,b] vs. integrating cos(x), any f(x) in [a,b]?
  - Same integration method => no need to code again
  - Can the Integration function take the f(x) as parameter?



Language Support

- C/C++: Function Pointer
  - Can pass a pointer to a function as parameter
- Python (and languages in FP paradigm)
  - Fully support it

```
def convert (fun, lst) :
      result = []
      for x in lst :
                result = result.append(fun(x))
      return result

print(convert(math.factorial, [1,2,3,4])       => [1, 2, 6, 24]
def cube(x): return x**3
print(convert(cube, [1,2,3,4])                  => [1, 8, 27, 64]
```

Subprogram Name as Parameters

- Pros and Cons
  - Pros: previous examples
  - Cons: see issues below

- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

Subprogram Referencing Environment

- The *referencing environment* indicates the scope and visibility of the local variables
  - i.e. a subprogram uses a variable x that is not defined inside itself, where should we look for x?
- *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
- Example: next slide

```
function sub1() {  //JavaScript
        var x;
        function sub2() {
                alert(x); // what is x's value?
        };
        function sub3() {
                var x; x = 3;
                sub4(sub2);
        };
        function sub4(subx) {
                var x; x = 4; subx();
        };
        x = 1;
        sub3();
        };
```

Under shallow binding:

x=4

//sub2 called from sub4

Under deep binding:

x=1

//sub2's static parent is sub1

Under Ad Hoc binding:

x=3

//the environment that
//passed sub2 is sub3
//what sub4 called is subx()
//it is inside sub3 that subx()
//bound to sub2()

sub1() -> sub3() -> sub4(sub2) ->(subx binds to sub2) -> subx()/sub2() -> x?

Lambda Expressions/functions

- Anonymous functions
  - Popular with AWS
- Example (Python)

```
x = lambda a : a + 10
print(x(5))
```

- More to discuss at functional programming

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, e.g. C++ examples:

    ```
    void print (int);
    void print (double, char);
    int print (Student &, int);
    ```

  - In C++ the return type cannot be used to resolve overloading ambiguities while Ada allows that.

- Many languages such as Ada, Java, C++, and C# support subprogram overloading
  - allow users to write multiple versions of subprograms with the same name
  - Good for readability
  - Popularly used in constructors

Discussion: overloaded subprograms

- How to resolve ambiguity?
  - Usually we call a subprogram base on its name
  - Now, with a number of subprograms with the same name, how to decide which one to call?

- Review the subprogram terminology we studied earlier
  - Which of the following is Java/C++ used to resolve ambiguity?
    - Protocol, prototype, declaration, definition, …

User-Defined Overloaded Operators

- Operators can also be overloaded in many languages including Ada, C++, Python, and Ruby
  - However, the syntax for overloading a lot different

- A Python example

```
def __add__ (self, second) :
    return Complex(self.real + second.real, self.imag + second.imag)
```

Use: `x + y` or `x.__add__(y)` //assume x and y are complex numbers

- A C++ Example

```
class Complex {
        friend Complex operator+(Complex a, Complex b) { … }
        …
};
```

Use: `Complex x, y, z; …;  z = x+y;`

Question: Does Java support overloaded operators? Why or why not?

Generic Subprograms

- A *generic* or *polymorphic* *s*ubprogram takes parameters of different types on different activations

- Overloaded subprograms provide *ad hoc polymorphism*

- In OOP, polymorphism usually refers to *subtype polymorphism* where a variable of type T can access any object of type T as well as any type derived from T

- 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*
  - A cheap compile-time substitute for dynamic binding

Generic Subprograms in C++

- C++
  - Generic subprograms are preceded by a template clause that lists the generic variables, which can be type names or class names
  - “real” versions of a generic subprogram are created implicitly when the subprogram is called

```
template <typename Type>          //<class Type> may also be used
  Type max(Type first, Type second) {
  return first > second ? first : second;
  }
 int a=5, b=3, c;
 c = max(a, b);              //Type is unified with int
 double x = 3.5, y=1.5, z;
 z = max(x,y);               //Type is unified with double
```

Generic Subprograms in Java

- Generic subprograms introduced in Java since Java 5.0
- Example:

  ```
  public static <T> T doIt(T[] list) { … }
  ```

  - T is the name of the generic type
  - The parameter list is an array of generic elements
  - A call below bounds String to T

    ```
    doIt<String>(myList);
    ```

- The generic type must be of a class that implements the `Comparable` interface

Generic Subprograms in Java (continued)

- Wildcard types

  `Collection<?>` is a wildcard type for collection classes

```
void printCollection(Collection<?> c) {
    for (Object e: c) {
      System.out.println(e);
    }
 }
```

  – Works for any collection class

Question: Generics in Java vs. that in C++? Which one has better design?

Discussion: Write a generic sort method in Java (header only) and two calls to sort an array integers, an array of (name, score) objects.

Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Three modes of parameter passing: in mode, out mode, and inout mode popularly implemented as call by value, and call by reference
- Some languages allow subprogram as parameters
  - Referencing environment varies in this case
- Subprograms can be overloaded and some languages even allow operator overloading
- Subprograms can be generic