Concepts of programing Languages

Subprograms Design -1

Sudan University of Science and Technology

Dr. Aghabi Nabil Abosaif

Topics

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

Introduction

- Two **fundamental abstraction** facilities
  - **Process abstraction**
    - Emphasized from early days (subprograms)
  - **Data abstraction**
    - Emphasized in the1980s

Fundamentals of Subprograms

- Most of subprograms, except the coroutines *will be described later*, have the following characteristics:

1. Each subprogram has a **single entry point.**
2. The calling program is **suspended** during execution of the called subprogram, which implies that **there is only one subprogram in execution at any given time.**
3. Control always **returns** to the **caller** when the called subprogram's execution **terminates**

Definition

- A subprogram is a block of code that **performs a specific task**.

- It can be invoked (or called) from **elsewhere in the program**, allowing for code **reuse** and **modularity**.

- Subprograms
  - improve **readability**,
  - reduce **redundancy**,
  - enhance **maintainability** by breaking complex tasks into smaller
  - more **manageable** units.

Subprograms History

- The first programmable computer, **Babbage's Analytical Engine**, built in the 1840s, had the capability of reusing collections of **instruction cards** at several different places in a program.

- In a modern programming language, such a **collection of statements** is written as a subprogram. This reuse results in savings in memory space and coding time.

- Also, The **methods** of object-oriented languages are closely related to the subprograms.

- The primary way methods differ from subprograms is the way they are **called** and their **associations** with **classes** and **objects**.

Basic Definitions

- The two fundamental kinds of subprograms, **procedures** and **functions**.

- 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 is said to be **active** if, after having been called, it has begun execution **but has not yet completed that execution.**

Basic Definitions -1

- A *subprogram* ***header*** is the first part of the definition, including *the name, the kind of subprogram, and the formal parameters*.

  - **def adder (parameters):** //*This is the header of a Python*
  - Ruby -> **def**.
  - JavaScript -> **function**.
  - In C, the header might be as follows:
    - **void** adder (parameters)

Basic Definitions -2

- The **body** of subprograms *defines its actions*.

  - In the C- based languages (and some others— For example, JavaScript) the body of a subprogram is **delimited by braces**.

  - In Ruby, an **end statement terminates** the body of a subprogram.

  - As with compound statements, the statements in the body of a Python function must be **indented** and the end of the body is indicated by the first statement that is not indented.

Basic Definitions -2

- One characteristic of **Python** functions that sets them apart from the functions of other common programming languages is that function **def** statements are executable.

- When a **def** statement is **executed**, it assigns the given name to the given function body.

- Until a function's def has been executed, the function cannot be called.

Basic Definitions -2

- If the **then clause** of this selection construct is executed, that version of the function fun can be called, **but not the version in the else clause**.

- Likewise, if the **else clause is chosen**, its version of the function can be **called but the one in the then clause cannot.**

```
if . . .
  def fun(. . .):
    . . .
else
  def fun(. . .):
    . . .
```

Basic Definitions -2

- All **Lua** functions are **anonymous**, although they can be defined using syntax that makes it appear as *though they have names*.
- For example, consider the following identical definitions of the function cube:
  - function cube(x) return x * x * x end
  - cube = function (x) return x * x * x end
- The first of these uses conventional syntax,
- The form of the second more accurately illustrates the namelessness of functions.

Basic Definitions -3

- The ***parameter profile*** (aka *signature*) of a subprogram is the *<u>number, order, and types</u>* of its parameters.

- The ***protocol*** is a subprogram's **parameter** *<u>profile</u>* and, if it is a function, it ***<u>return type</u>***

Basic Declaration

- **Function declarations** in C and C++ are often called *prototypes*. *Such declarations are often placed* ***in header files***.
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram.
- In both the cases of variables and subprograms, **declarations are needed for static type checking.**
  - In the case of subprograms, it is the type of the parameters that must be checked.

parameter

- A ***formal parameter*** is a *dummy variable* listed in the subprogram header and used in the subprogram.
  - Called dummy variables because they are not variables in the usual sense: In most cases, they **are bound to storage only when the subprogram is called**, and that binding is often through some other program variables.
- An ***actual parameter*** represents a *value or address* used in the subprogram call statement.

Actual/Formal Parameter

- The parameters in the subprogram header are called **formal parameters**.

- Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram.
- These parameters are called **actual parameters**

Parameter Correspondence

1. **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*

2. **Keyword**
   - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
     - sumer(lenth = my_length, sum = my_sum, list = my_array)

   - ***Advantage*:** Parameters can *appear in any order*, thereby avoiding parameter correspondence errors
   - ***Disadvantage*:** User must know the formal *parameter's names*

Parameter Correspondence(Cons.)

- In some languages, like Python ; Keyword and positional parameters **can be mixed**
- in a call, as in
- **sumer(my_length, sum = my_sum, list = my_array)**

- The only restriction with this approach is that after a keyword parameter appears in the list, all remaining parameters must be keyworded.

- This restriction is necessary **because** a position may no longer be well defined after a keyword parameter has appeared.

Formal Parameter Default Values

- In In Python, Ruby, C++, and PHP, formal parameters can have default values.
- Formal parameters can ***<u>have default values </u>***(if no actual parameter is passed)
- `def compute_pay(income,exemptions = 1,tax_rate)`
- `pay = compute_pay(20000.0, tax_rate = 0.15)`

Formal Parameter Default Values

- In C++, which does not support keyword parameters, the rules for default parameters are necessarily different.
- The default parameters must appear last, because parameters are positionally associated.
- A C++ function header for the compute_pay function can be written as follows:
- **float compute_pay(float income, float tax_rate,int exemptions = 1)**
- **pay = compute_pay(20000.0, 0.15);**

Procedures and Functions

- There are **two categories** of subprograms
  1. ***Procedures*** are *collection of statements* that define parameterized computations. Older languages (such as Fortran and Ada, support procedures).

  - The computations of a procedure are enacted by **single call statements.**

- Procedures **can produce results in the calling program unit** by two methods:
  1. If there are variables that **are not formal parameters** but are still **visible in both** the procedure and the calling program unit, the procedure can change them;

  2. If the procedure **has formal parameters** that **allow the transfer of data to the caller**, those parameters can be changed.

Procedures and Functions

- There are **two categories** of subprograms
  2. ***Functions*** structurally resemble procedures but are semantically modeled *on mathematical functions.*

- **Functions return values and procedures do not**
  - They *are expected to produce no side effects*
    - *that is, it* ***modifies*** *neither its parameters nor any variables defined* ***outside the function****.*
  - In practice, program *functions may have side effects.*

- Functions are **called by appearances** of their names in expressions, along with the required actual parameters.

- The **value produced by a function's** execution is returned to the calling code.

Design Issues for Subprograms
1. Local Referencing Environments

- **Subprograms can define their own variables, thereby defining local referencing environments.**
- Local variables can be **static**
  - defined inside subprograms, *their scope is usually the body* of the subprogram in which they are *defined*.
- Local variables can be **stack-dynamic**
  - they are bound to storage when the subprogram begins execution and are unbound from storage when that execution terminates.

  \- **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

Example- Local Referencing Environments

- In C and C++ functions, **locals are stack dynamic** unless specifically **declared to be static**.
- In the following function, the variable sum is static and count is stack dynamic.

```
int adder(int list[], int listlen) {
  static int sum = 0;
  int count;
  for (count = 0; count < listlen; count ++)
    sum += list [count];
  return  sum;
}
```

- The methods of **C++**, **Java**, **and C#** have **only stack-dynamic local variables.**
- In Python, a global variable is defined outside any function or method. You can reference a global variable inside a function **without declaring** it as global.

```
x = 10 # Global variable
 def print_x(): print(x) # No need for 'global' to read the global variable
print_x() # Outputs: 10
```

- **Implicit Local Variables**, If you assign to a global variable inside a method without declaring it as global, Python treats it **as a local variable**, and the global variable remains **unchanged**.

```
x = 10   # Global variable
def modify_x(): x = 20   # Local variable, doesn't affect the global variable
modify_x()
print(x)    # Outputs: 10 (global variable not modified)
```

- **To modify a global** variable inside a function, you must **declare it as global**.

```
x = 10  # Global variable
def modify_x():
   global x   # Declare the global variable
   x = 20      # Modify the global variable
modify_x()
print(x)     # Outputs: 20 (global variable modified)
```

- **Local variables** are defined **inside** a method and exist only within that method.

```
def calculate_square(a):
  result = a * a # Local variable 'result' return result

print(calculate_square(5)) # Outputs: 25
# 'result' is not accessible here.
```

- All local variables in Python methods are **stack dynamic**.

Design Issues for Subprograms 2.Parameter-passing methods

- They are the ways in which parameters are **transmitted to and/or from** called subprograms.
  - Semantics models of parameter-passing methods.
  - Implementation models

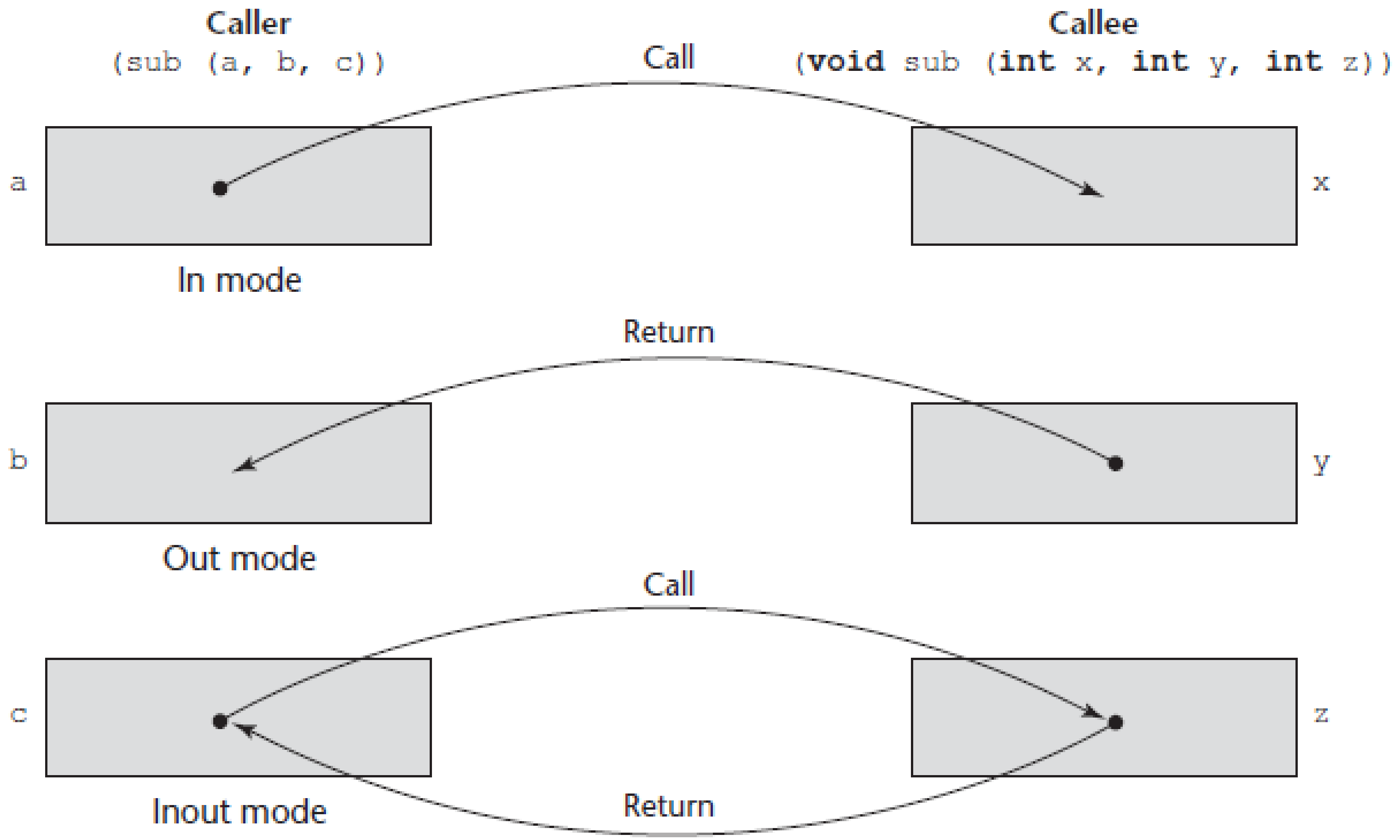
Semantic Models of Parameter Passing

- **In mode**: They can **receive data** from the corresponding actual parameter;
- **Out mode**: They can **transmit data** to the actual parameter;
- **Inout mode**: They can do **both**.

Semantic Models of Parameter Passing

- Consider a subprogram that takes two arrays of ***int*** values as parameters— ***list1*** and ***list2***.
- The subprogram must add ***list1*** to ***list2*** and return the result as a revised version of ***list2***.
- Furthermore, the subprogram must create a new array from the two given arrays and return it. For this subprogram,
- **list1** should be **in mode**, because it is not to be changed by the subprogram.
- **list2** must be **inout mode**, because the subprogram needs the given value of the array and must return its new value.
- The **third array** should be **out mode**, because there is no initial value for this array and its computed value must be returned to the caller.

The three semantics models of parameter passing when physical moves are used



Implementation Models of Parameter Passing

- **Conceptual Models of Transfer**
  - Physically move a value
  - Move an access path to a value.

1.Pass-by-Value (In Mode)

- The value of the actual parameter is *used to initialize* the corresponding formal parameter
  - Normally **implemented by copying**
  - Also,it Can be implemented **by transmitting *an access path but not recommended*** (enforcing write protection updated 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 update **write-protect** in the called subprogram.

1.Pass-by-Value (In Mode) Example

```
public class CallByValueExample {
    // Function that changes the value of x
    public static void changeValue(int x) {
        x = 10;  // This will not change the original value in main()
    }

    public static void main(String[] args) {
        int number = 5;
        changeValue(number);  // Call by value
        System.out.println("Number after function call: " + number);  // Prints 5
    }
}
```

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- Pass-by-Result : <u>Compute address</u> o at the *<u>beginning</u>* of the subprogram or <u>end</u>?

- the implementor may be able to choose <u>between two different times</u> to evaluate the addresses of the actual parameters: **at the time of the call** or **at the time of the return.**

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

If the address of **list[sub]** is evaluated at the **time of the call**, the value **17** will be stored in **list[21]** (since sub = 21 at the time of the call).

If the address of **list[sub]** is evaluated at the **time of return**, the value **17** will be stored in **list[42]** (since sub = 42 at the time of return).

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)

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.
- Pass-by-name parameters are both **complex** to implement and **inefficient**.
- They also add significant complexity to the program, thereby lowering its **readability** and **reliability**

Pass-by-Name (Inout Mode) Example

```
#include <iostream>


#define SQUARE(x) ((x) * (x))
 // Macro definition, equivalent to pass-by-name
int main() {
    int a = 5;
    int b = 10;
    std::cout << "Result: " << SQUARE(a + b) << std::endl;
    // Macro definition, equivalent to pass-by-name
   return 0;
}
```

- When we call SQUARE(a + b), the textual substitution happens, and the result is computed as:

```
((a + b) * (a + b)) -> ((5 + 10) * (5 + 10)) = 225
```

To Be Continued