Control Abstraction

Dr.. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

09, 2013

Outline

Subprogram Definition

Subprogram definition consists of:

- **Specification**
  - Subprogram name
  - Parameters
    - input + output
    - order
    - type
    - parameter passing mechanisms: by value, by reference, by name,...
  - Behaviour of the subprogram
- **Implementation**:
  - Local data
  - Collection of statements as **subprogram body**

định nghĩa của chương trình con (Subprogram) bao gồm:
Specification (Đặc tả):
Subprogram name (Tên chương trình con)
Parameters (Tham số): Input, Output, thứ tự, kiểu dữ liệu, cơ chế truyền tham số (by value, by reference, by name,...).
Behaviour of the subprogram (Hành vi của chương trình con)
Implementation (Triển khai):
Local data (Dữ liệu cục bộ)
Collection of statements as subprogram body (Tập hợp các câu lệnh tạo thành thân chương trình con)

Example

```
def apply(interval: Int,
          repeats: Boolean = true)
          (op: => Unit) {
  val timeOut = new javax.swing.AbstractAction() {
    def actionPerformed
    (e: java.awt.event.ActionEvent) = op
  }
  val t = new javax.swing.Timer(interval, timeOut)
  t.setRepeats(repeats)
  t.start()
}
```

1. How many subprogram definitions are there in the above code?

Trong đoạn code trên, có 2 subprogram definition:
apply
actionPerformed

2. How many parameters are there in each subprogram definition?

Subprogram apply có 3 tham số: interval: Int, repeats: Boolean = true, và op: => Unit.

Subprogram actionPerformed có 1 tham số: e: java.awt.event.ActionEvent.

Subprogram Activation

Khi một chương trình con (subprogram) được gọi để thực thi, nó sẽ được "kích hoạt" (activation). Quá trình kích hoạt này bao gồm các bước sau:
Khởi tạo: Khi chương trình con được gọi, một "activation" (hoạt động) của chương trình con đó được tạo ra.
Hủy bỏ: Khi chương trình con thực hiện xong nhiệm vụ của nó, "activation" đó sẽ bị hủy.

An activation of a subprogram:

- is created when the subprogram is invoked
- is destroyed when the subprogram completed its execution

An activation includes

- Static part: Code segment
- Dynamic part: Activation record
  - formal parameters
  - local data
  - return address
  - other links

Mỗi "activation" sẽ bao gồm hai phần:

Static part (Phần tĩnh): Là phần mã lệnh (code segment) của chương trình con. Phần này không thay đổi trong suốt quá trình thực thi.
Dynamic part (Phần động): Là "activation record" (bản ghi hoạt động), chứa các thông tin thay đổi trong quá trình thực thi, bao gồm:
Các tham số hình thức (formal parameters)
Dữ liệu cục bộ (local data)
Địa chỉ trả về (return address)
Các liên kết khác (other links)

"activation" giống như là một "phiên bản" làm việc của chương trình con. Mỗi khi chương trình con được gọi, một phiên bản làm việc mới được tạo ra để thực hiện các lệnh. Khi thực hiện xong, phiên bản đó sẽ biến mất.

Subprogram Mechanisms

- Simple Call-Return
- Recursive Call
- Exception Processing Handler
- Coroutines
- Scheduled Subprograms
- Tasks

một số cơ chế phổ biến mà các ngôn ngữ lập trình sử dụng để quản lý việc thực thi các chương trình con. Các cơ chế này bao gồm:

Simple Call-Return (Gọi và trả về đơn giản): Đây là cơ chế cơ bản nhất, khi một chương trình con được gọi, nó sẽ thực thi xong rồi trả kết quả về chỗ gọi. Không có đệ quy, gọi rõ ràng, một lối vào và trả về ngay lập tức.
Recursive Call (Gọi đệ quy): Cho phép một chương trình con gọi lại chính nó (đệ quy trực tiếp) hoặc gọi một chương trình con khác mà cuối cùng lại gọi lại nó (đệ quy gián tiếp).
Exception Processing Handler (Xử lý ngoại lệ): Cơ chế này dùng để xử lý các tình huống bất thường xảy ra trong quá trình thực thi chương trình, ví dụ như lỗi chia cho 0, lỗi truy cập bộ nhớ....
Coroutines: Cho phép các chương trình con tạm dừng thực thi và trả quyền điều khiển cho chương trình gọ , sau đó có thể tiếp tục thực thi từ chỗ tạm dừng.
Scheduled Subprograms (Chương trình con được lên lịch): Việc thực thi chương trình con không bắt đầu ngay khi nó được gọi, mà được lên lịch theo thời gian hoặc độ ưu tiên.
Tasks: Các đơn vị thực thi có thể chạy đồng thời với các đơn vị khác, có thể trên máy đa xử lý hoặc chia sẻ thời gian trên máy đơn xử lý.

Simple Call-Return

No recursion (Không đệ quy): Tức là chương trình con không gọi lại chính nó.
Explicit Call Site (Vị trí gọi tường minh): Mình thấy rõ ràng chỗ nào trong code gọi chương trình con.
Single Entry Point (Một điểm vào): Mỗi lần gọi là chỉ có một điểm bắt đầu thực thi duy nhất của chương trình con.
Immediate Control Passing (Chuyển giao điều khiển ngay lập tức): Khi gọi chương trình con, quyền điều khiển chuyển ngay cho nó, và khi nó xong thì trả về ngay.
Single Execution (Thực thi đơn): Chương trình con thực thi một lần rồi trả về.

Basic Features

- No recursion
- Explicit Call Site
- Single Entry Point
- Immediate Control Passing
- Single Execution

Recursive Call

- Be able to call recursive
  - Direct Recursice Call
  - Indirect Recursive Call (Mutual Recursive)
- Other features same as Simple Call-Return

Direct Recursive Call (Đệ quy trực tiếp): Là khi một chương trình con gọi trực tiếp đến chính nó.
Indirect Recursive Call (Mutual Recursive) (Đệ quy gián tiếp hay đệ quy tương hỗ): Là khi một chương trình con gọi một chương trình con khác, và chương trình con kia lại gọi ngược lại chương trình con ban đầu.
Ngoài ra, các đặc điểm khác của Recursive Call cũng giống như Simple Call-Return.

Exception Processing Handler

- May have no explicit call site
- Used in
  - Event-Driven Programming
  - Error Handler

Example,

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp(){
     if (length(V)==0) throw new EmptyExcp();
     else ...
};
...
try{...
   average(W);
   ...
}
catch (EmptyExcp e) {write("Array empty"); }
```

Cái này dùng để xử lý mấy tình huống "oái oăm" xảy ra khi chạy chương trình. Ví dụ như là chia cho số 0, truy cập vào vùng nhớ không được phép, hoặc là file không tồn tại...
Mấy cái "ngoại lệ" này có thể:

May have no explicit call site (Có thể không có vị trí gọi tường minh): Tức là nó có thể xảy ra ở đâu đó trong chương trình mà mình không biết trước được.

Used in (Được dùng trong):
Event-Driven Programming (Lập trình hướng sự kiện): Ví dụ như khi người dùng click chuột, gõ phím... nếu có lỗi gì xảy ra thì bộ xử lý ngoại lệ sẽ lo.
Error Handler (Xử lý lỗi): Cái này thì rõ rồi, dùng để xử lý các loại lỗi khác nhau.

Ngoại lệ giống như mấy cái sự cố bất ngờ. Mình không biết khi nào nó xảy ra, nhưng mình có thể chuẩn bị sẵn các phương án để xử lý khi nó xảy ra.

Exception Mechanisms

Khi một ngôn ngữ lập trình hỗ trợ việc xử lý ngoại lệ, nó cần phải quy định rõ ràng mấy cái này:

which exceptions can be handled and how they can be defined (những loại ngoại lệ nào có thể được xử lý và cách định nghĩa chúng):
Ví dụ, trong Java thì ngoại lệ phải là một lớp con của Throwable. Trong Ada thì nó là các giá trị của một kiểu đặc biệt. Còn C++ thì có thể là bất kỳ giá trị nào.
how an exception can be raised (cách phát sinh một ngoại lệ):

A language must specify:

- *which* exceptions can be handled and *how* they can be defined
- *how* an exception can be raised
- *how* an exception can be handled

Ngoại lệ có thể được phát sinh bởi người dùng (ví dụ như khi click chuột, gõ phím), bởi hệ điều hành, bởi một đối tượng (ví dụ như Timer), hoặc bởi chính lập trình viên (thông qua lệnh throw).
how an exception can be handled (cách xử lý một ngoại lệ):

Cái này liên quan đến việc định nghĩa một khối code được bảo vệ (protected block) để "bắt" ngoại lệ, và định nghĩa bộ xử lý ngoại lệ (exception handler) tương ứng với khối đó.
Ngoài ra, ngôn ngữ cũng cần quy định về "termination semantic" (ngữ nghĩa kết thúc) khi xảy ra ngoại lệ, ví dụ như là có tiếp tục thực thi sau khi xử lý ngoại lệ hay không (resumable vs. non-resumable), và nếu không tiếp tục thì có "unwind stack" (giải phóng stack) hay không.

How an exception can be defined

- Java: subclass of *Throwable*
- Ada: values of a special type
- C++: any value

Mỗi ngôn ngữ lập trình có cách riêng để định nghĩa một loại ngoại lệ mới:

Java: Ngoại lệ được định nghĩa bằng cách tạo ra một lớp con của lớp Throwable.

Ada: Ngoại lệ là các giá trị của một kiểu đặc biệt.

C++: Ngoại lệ có thể là bất kỳ giá trị nào.

How an exception can be raised

Ngoại lệ có thể được "tung ra" (raised) từ nhiều nguồn khác nhau:

By user interaction (Do tương tác của người dùng): Ví dụ như khi mày click chuột, di chuyển chuột, hoặc thay đổi nội dung ô nhập liệu, có thể phát sinh ngoại lệ nếu có lỗi xảy ra trong quá trình xử lý sự kiện đó.

Raising exception

- By user interaction
  (Click, MouseMove, TextChange, ...)
- By operating system
- By an object (Timer)
- By programmer (throw)

By operating system (Do hệ điều hành): Hệ điều hành có thể phát sinh ngoại lệ, ví dụ như lỗi truy cập bộ nhớ trái phép.

By an object (Do một đối tượng): Một đối tượng cụ thể trong chương trình cũng có thể phát sinh ngoại lệ, ví dụ như đối tượng Timer trong ví dụ ở slide.

By programmer (Do lập trình viên): Mày có thể tự phát sinh ngoại lệ trong code bằng cách sử dụng từ khóa throw.

Example in Scala

sử dụng Timer để phát sinh sự kiện, sau đó có thể liên kết với việc xử lý ngoại lệ (mặc dù ví dụ này chủ yếu minh họa cách dùng Timer và hàm bậc cao):

```
object Timer {
  def apply(interval: Int,
            repeats: Boolean = true)
            (op: => Unit) {
    val timeOut = new javax.swing.AbstractAction() {
      def actionPerformed
      (e: java.awt.event.ActionEvent) = op
    }
    val t = new javax.swing.Timer(interval, timeOut)
    t.setRepeats(repeats)
    t.start()
  }
}
Timer(2000) { println("Timer went off") }
Timer(10000, false) { println("10 seconds are over!") }
```

Đoạn code này định nghĩa một Timer sẽ chạy một đoạn code (op) sau một khoảng thời gia interval). Nếu có lỗi xảy ra trong op, nó có thể được bắt và xử lý như một ngoại lệ. Ví dụ dưới cùng cho thấy cách gọi Timer với các khoảng thời gian và hành động khác nhau.

How an exception can be handled

Cách xử lý ngoại lệ (How an exception can be handled):
Để xử lý ngoại lệ, mình cần làm mấy việc sau:
Define the protected block (Định nghĩa khối được bảo vệ): Đây là đoạn code mà mình muốn theo dõi xem có ngoại lệ xảy ra hay không.
Define exception handler (Định nghĩa bộ xử lý ngoại lệ): Cái này là đoạn code sẽ được chạy khi một ngoại lệ cụ thể xảy ra trong khối được bảo vệ.

- Define *the protected block* to intercept the exception for being handled
- Define exception handler associated with the protected block

Termination Semantic

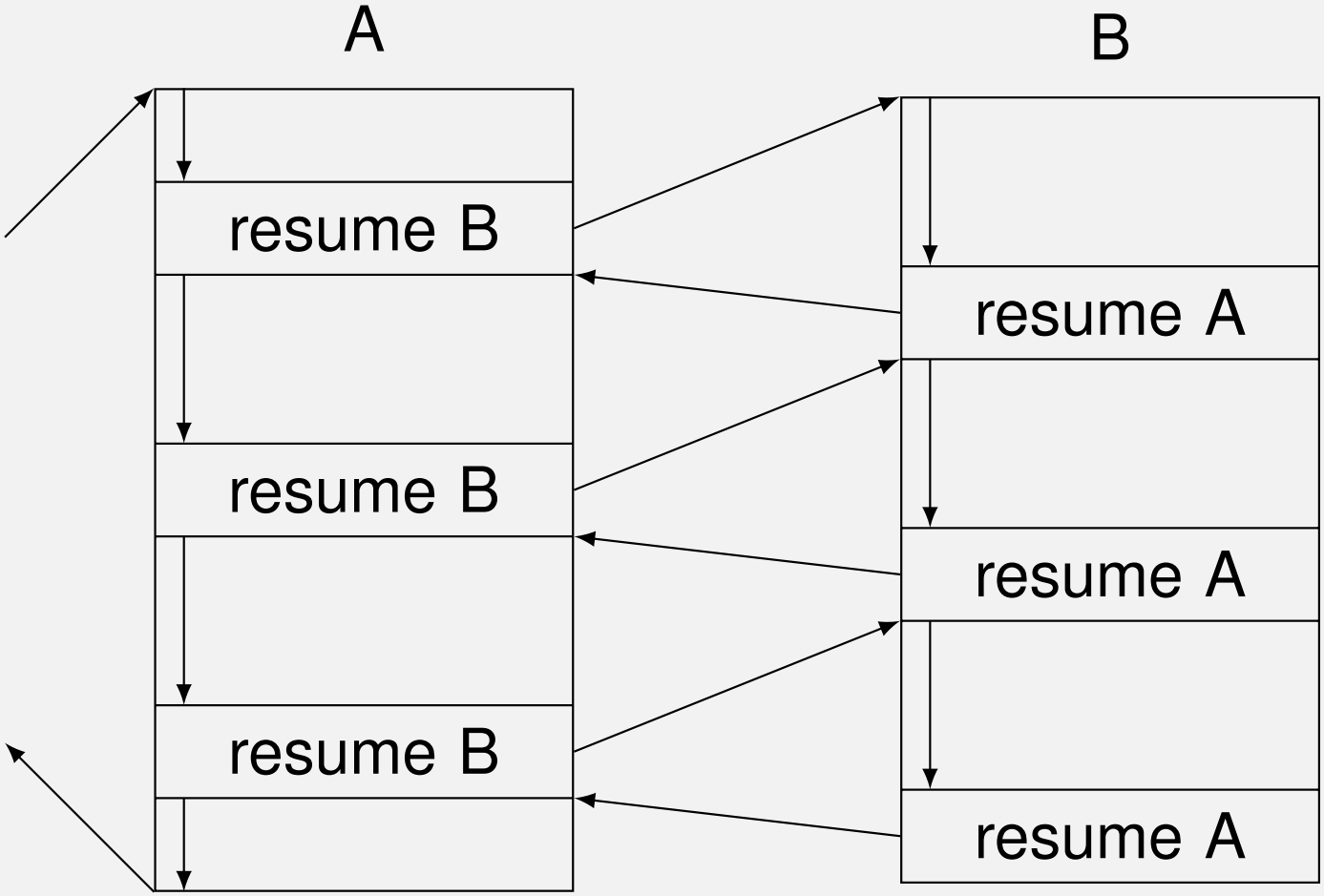
- non-resumable (common) + stack unwinding
- resumable
  - at the statement causing the error
  - after the statement causing the error

Về ngữ nghĩa kết thúc (Termination Semantic), tức là chuyện gì sẽ xảy ra sau khi ngoại lệ được xử lý:

Non-resumable (Không tiếp tục được): Đây là trường hợp phổ biến. Sau khi ngoại lệ được xử lý, chương trình sẽ không tiếp tục thực thi từ chỗ gây ra lỗi nữa. Thường đi kèm với "stack unwinding", tức là giải phóng các hàm đã gọi trên stack cho đến khi gặp bộ xử lý ngoại lệ phù hợp.
Resumable (Có thể tiếp tục được): Ít phổ biến hơn. Sau khi xử lý ngoại lệ, chương trình có thể tiếp tục thực thi, có thể là ngay tại câu lệnh gây lỗi hoặc sau câu lệnh gây lỗi.

A coroutine may postpone its execution and control is back to caller. Its execution later is resumed at the place it postponed.

Một coroutine (chương trình con tương trợ) có thể tạm dừng việc thực thi của nó và trả quyền điều khiển lại cho bên đã gọi nó. Sau đó, nó có thể tiếp tục chạy lại từ chính xác cái chỗ mà nó đã tạm dừng trước đó.



Cái hình vẽ này cho thấy sự tương tác giữa hai coroutine, gọi là A và B.

Ban đầu, có thể là A bắt đầu chạy (mũi tên đi vào A từ bên trái).
Trong A, nó gặp lệnh resume B. Lúc này, A sẽ tạm dừng lại ngay tại dòng lệnh đó và chuyển quyền điều khiển sang cho B. Mũi tên từ A chỉ sang B tượng trưng cho việc này.
B bắt đầu chạy (hoặc tiếp tục chạy từ chỗ nó dừng lần trước).
Trong B, nó gặp lệnh resume A. Lúc này, B sẽ tạm dừng và chuyển quyền điều khiển lại cho A. Mũi tên từ B chỉ sang A tượng trưng cho việc này.
Quan trọng là khi A nhận lại quyền điều khiển, nó sẽ tiếp tục chạy từ ngay sau dòng lệnh resume B mà nó đã tạm dừng trước đó.
Cứ thế, A và B luân phiên nhau chạy, mỗi lần gặp resume là nhường quyền điều khiển cho bên kia và "đánh dấu" lại chỗ dừng để lần sau quay lại chạy tiếp.
Khác với hàm gọi thông thường là gọi xong là chạy hết rồi trả về chỗ gọi nó, coroutine có thể tạm dừng bất cứ lúc nào và cho bên khác chạy, rồi sau đó cái bên kia lại có thể "nhường" lại quyền điều khiển để coroutine ban đầu tiếp tục chạy tiếp. Nó giống như hai người đang làm việc chung, thay phiên nhau làm.

Tasks

"Tasks" (tác vụ) trong lập trình là các đơn vị công việc có khả năng chạy đồng thời (concurrently) với các tác vụ khác.
Đặc điểm của Tasks:
Có thể thực thi đồng thời với các tác vụ khác.
Có thể chạy trên máy có nhiều bộ xử lý (multi-processor machine) hoặc trên máy chỉ có một bộ xử lý bằng cách sử dụng kỹ thuật chia sẻ thời gian (time sharing).

- able to execute concurrently with other tasks
- run on multi-processor machine or
- single processor machine using time sharing

Issue?

khi các tác vụ chạy đồng thời, sẽ có một số vấn đề phát sinh cần phải xử lý :

- Synchronization
  - Race condition
  - Deadlock
- Communication

Synchronization (Đồng bộ hóa): Làm sao để các tác vụ phối hợp nhịp nhàng với nhau, tránh xung đột khi cùng truy cập vào tài nguyên chung

Race condition (Tình trạng chạy đua): Xảy ra khi kết quả của chương trình phụ thuộc vào thứ tự mà các tác vụ truy cập hoặc sửa đổi dữ liệu dùng chung. Nếu thứ tự thực thi thay đổi, kết quả cũng thay đổi, dẫn đến lỗi không mong muốn.

Deadlock (Tắc nghẽn): Xảy ra khi hai hoặc nhiều tác vụ chờ đợi lẫn nhau một cách vô hạn để giải phóng tài nguyên, dẫn đến tất cả đều bị kẹt lại và không thể tiếp tục thực thi.

Communication (Truyền thông): Làm sao để các tác vụ có thể trao đổi thông tin hoặc dữ liệu với nhau.

Example in Scala

ví dụ bằng Scala để minh họa việc sử dụng parallel collections (.par) để thực thi các thao tác trên mảng một cách song song (parallel processing), đây là một dạng của Tasks:

```
val pa = (0 until 10000).toArray.par

pa.map(_ + 1)

pa map { v => if (v % 2 == 0) v else -v }

pa.fold(0) { _ + _ }

var a = 0

pa foreach { a += _ }
```

val pa = (0 until 10000).toArray.par // Tạo một parallel array từ dãy số từ 0 đến 9999 [cite: 25, 26]
pa.map(_ + 1) // Cộng 1 cho mỗi phần tử, thực hiện song song [cite: 26]
pa map { v => if (v % 2 == 0) v else -v } // Đổi dấu các số lẻ, thực hiện song song [cite: 26]
pa.fold(0) { _ + _ } // Tính tổng các phần tử, có thể thực hiện song song [cite: 27]

var a = 0
pa foreach { a += _ } // Cộng dồn các phần tử vào biến 'a'. Chú ý: thao tác này có thể gặp vấn đề Race condition vì nhiều tác vụ cùng ghi vào biến 'a' mà không có đồng bộ hóa. [cite: 27]

Ví dụ này cho thấy cách Scala hỗ trợ việc tạo ra các "tasks" ngầm khi sử dụng .par trên collection, cho phép các thao tác như map và fold được thực hiện trên nhiều nhân xử lý cùng lúc. Tuy nhiên, dòng cuối cùng (pa foreach { a += _ }) là một ví dụ tiềm ẩn vấn đề "race condition" nếu không có biện pháp đồng bộ hóa phù hợp, vì nhiều tác vụ có thể cùng cố gắng cập nhật biến a cùng lúc.

Scheduled subprograms

Cái này khác với cách gọi chương trình con thông thường. Với Scheduled subprograms, khi mày gọi nó, nó không bắt đầu chạy ngay lập tức đâu. Thay vào đó, việc thực thi của nó sẽ được "lên lịch" bởi một bộ điều phối (scheduler).

Việc lên lịch này có thể dựa trên:

scheduled by time (lên lịch theo thời gian): Ví dụ như slide ghi "CALL A AT TIME = CURRENT_TIME + 10", tức là gọi chương trình con A để chạy sau 10 đơn vị thời gian kể từ bây giờ.
scheduled by priority (lên lịch theo độ ưu tiên): Ví dụ "CALL B WITH PRIORITY 7", tức là gọi chương trình con B với độ ưu tiên là 7. Bộ điều phối sẽ dựa vào độ ưu tiên này để quyết định khi nào nên cho B chạy so với các tác vụ khác.

- The execution of callee is NOT started when it is invoked
  - scheduled by time
    CALL A AT TIME = CURRENT_TIME + 10
  - scheduled by priority
    CALL B WITH PRIORITY 7
- Controlled by a scheduler

Cả quá trình này được Controlled by a scheduler (Kiểm soát bởi một bộ điều phối). Bộ điều phối này giống như một người quản lý công việc, nó sẽ xem xét các yêu cầu chạy chương trình con và quyết định khi nào thì thực sự cho chúng chạy dựa trên các quy tắc lên lịch (thời gian, độ ưu tiên, v.v.).

Formal and Actual Parameter

"Formal and Actual Parameter" (Tham số hình thức và Tham số thực tế)

Definition

- Formal parameters:int foo(float x,bool& y);
  - just a simple name
  - close to a variable declaration
  - combine with symbols relating to parameter passing mechanism
- Actual parameters/Arguments:foo(4*a,b)
  - an expression

Formal-Actual Corresponding

- by position
  int foo(float a,int b) $\Leftarrow$ foo(x+1,y-2)
- by name
  int foo(float a,int b) $\Leftarrow$ foo(b = x+1, a = y-2)

Definition (Định nghĩa):

Formal parameters (Tham số hình thức):

Là mấy cái tên biến mà bạn khai báo trong định nghĩa của chương trình con. Ví dụ trong int foo(float x, bool& y);, thì x và y là tham số hình thức.

Nó giống như tên gọi đơn giản thôi.

Gần giống với việc khai báo biến cục bộ trong chương trình con.

Có thể kết hợp với các ký hiệu liên quan đến cơ chế truyền tham số (ví dụ: & trong C++ cho truyền tham chiếu).

Actual parameters/Arguments (Tham số thực tế / Đối số):

Là mấy cái giá trị hoặc biểu thức mà bạn truyền vào khi gọi chương trình con. Ví dụ khi gọi foo(4*a, b), thì 4*a và b là tham số thực tế.

Chúng thường là các biểu thức cần được tính toán giá trị trước khi truyền đi.

Formal-Actual Corresponding (Sự tương ứng giữa tham số hình thức và thực tế):

Làm sao để chương trình biết tham số thực tế nào sẽ "điền vào" tham số hình thức nào? Có hai cách phổ biến được slide nhắc đến:

by position (Theo vị trí):

Cách này phổ biến nhất. Tham số thực tế đầu tiên tương ứng với tham số hình thức đầu tiên, tham số thực tế thứ hai tương ứng với tham số hình thức thứ hai, và cứ thế tiếp tục.

Ví dụ: int foo(float a, int b) được gọi bằng foo(x+1, y-2). Lúc này, x+1 sẽ là giá trị cho a, và y-2 sẽ là giá trị cho b. Thứ tự của chúng trong lời gọi hàm và trong định nghĩa hàm phải khớp nhau.

by name (Theo tên):

Cách này ít phổ biến hơn. Bạn chỉ định rõ tham số thực tế nào sẽ tương ứng với tham số hình thức nào bằng cách dùng tên của tham số hình thức.

Ví dụ: int foo(float a, int b) được gọi bằng foo(b = x+1, a = y-2). Mặc dù b được viết trước a trong lời gọi, nhưng do bạn chỉ định rõ b = x+1 và a = y-2, nên x+1 sẽ là giá trị cho tham số hình thức b, và y-2 sẽ là giá trị cho tham số hình thức a. Thứ tự trong lời gọi không quan trọng bằng việc chỉ định tên.

Hiểu đơn giản, tham số hình thức là cái "chỗ chứa" dữ liệu trong định nghĩa hàm, còn tham số thực tế là "dữ liệu thật" mà bạn bỏ vào cái chỗ chứa đó khi dùng hàm. Có thể bỏ vào theo thứ tự hoặc chỉ rõ tên chỗ chứa.

Parameter Passing

- Input-Output
  - By value-result
  - By reference
  - By name
- Input Only
  - By value
  - By constant reference
- Output Only
  - By result
  - As a result of a function

"Parameter Passing" (Truyền tham số). Cái này là cách dữ liệu được chuyển giữa chương trình gọi (caller) và chương trình con được gọi (callee).

Slide chia ra làm 3 loại chính dựa trên mục đích sử dụng của tham số:

Input-Output (Vừa là đầu vào, vừa là đầu ra): Tham số này vừa dùng để truyền dữ liệu từ chương trình gọi vào chương trình con, và sau khi chương trình con xử lý, kết quả thay đổi trên tham số này sẽ được trả ngược về chương trình gọi. Có 3 cơ chế chính cho loại này:
By value-result (Theo giá trị - kết quả): Giá trị của tham số thực tế được sao chép vào tham số hình thức khi chương trình con bắt đầu. Chương trình con làm việc với bản sao này. Khi chương trình con kết thúc, giá trị cuối cùng của tham số hình thức được sao chép ngược trở lại vào tham số thực tế ban đầu. (giống như pass by value nhưng có thêm chép giá trị cuối vào cái ban đầu trở lại)
By reference (Theo tham chiếu): Thay vì sao chép giá trị, địa chỉ bộ nhớ của tham số thực tế được truyền vào chương trình con. Tham số hình thức lúc này hoạt động như một bí danh (alias) cho tham số thực tế. Bất kỳ thay đổi nào trên tham số hình thức bên trong chương trình con sẽ trực tiếp ảnh hưởng đến tham số thực tế bên ngoài.
By name (Theo tên): Đây là cơ chế ít phổ biến hơn, thường được dùng trong các ngôn ngữ lập trình chức năng. Biểu thức của tham số thực tế không được tính toán giá trị ngay khi gọi hàm, mà nó được truyền dưới dạng một "công thức" hoặc "khuôn mẫu" (thường gọi là thunk). Bất cứ khi nào tham số hình thức được sử dụng trong chương trình con, cái "công thức" này mới được tính toán lại giá trị của nó tại thời điểm đó và trong môi trường thực thi hiện tại của chương trình con.

Input Only (Chỉ là đầu vào): Tham số này chỉ dùng để truyền dữ liệu từ chương trình gọi vào chương trình con. Chương trình con sẽ sử dụng giá trị này nhưng không được phép thay đổi nó theo cách mà sự thay đổi đó ảnh hưởng đến tham số thực tế bên ngoài. Có 2 cơ chế chính:
By value (Theo giá trị): Giá trị của tham số thực tế được sao chép vào tham số hình thức khi chương trình con bắt đầu. Chương trình con làm việc với bản sao này. Mọi thay đổi trên bản sao sẽ không ảnh hưởng đến tham số thực tế.
By constant reference (Theo tham chiếu hằng số): Địa chỉ của tham số thực tế được truyền vào, tương tự như By reference. Tuy nhiên, ngôn ngữ lập trình đảm bảo rằng chương trình con không thể thay đổi giá trị mà tham chiếu đó trỏ tới (ví dụ dùng từ khóa const trong C++). Cách này hiệu quả khi truyền các đối tượng lớn vì không cần sao chép toàn bộ dữ liệu, nhưng vẫn đảm bảo an toàn dữ liệu đầu vào.

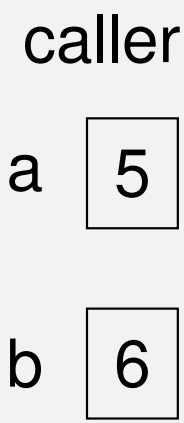
Output Only (Chỉ là đầu ra): Tham số này chỉ dùng để chương trình con trả kết quả về cho chương trình gọi. Chương trình gọi không truyền dữ liệu đầu vào qua tham số này. Có 2 cách:
By result (Theo kết quả): Giá trị của tham số hình thức bên trong chương trình con khi nó kết thúc sẽ được sao chép ngược trở lại vào tham số thực tế tương ứng. Giống By value-result nhưng không có bước sao chép từ tham số thực tế vào tham số hình thức lúc bắt đầu.
As a result of a function (Là kết quả trả về của hàm): Đây là cách phổ biến nhất. Giá trị kết quả được trả về trực tiếp bởi hàm thông qua từ khóa return, chứ không thông qua tham số nào cả. Ví dụ như int foo() ... return 0;, kết quả là 0 được trả về trực tiếp.
Mỗi cơ chế truyền tham số này có ưu và nhược điểm riêng về hiệu năng, độ an toàn dữ liệu và cách hoạt động. Việc lựa chọn cơ chế nào phụ thuộc vào ngôn ngữ lập trình và yêu cầu cụ thể của bài toán.

Input-Output Parameters

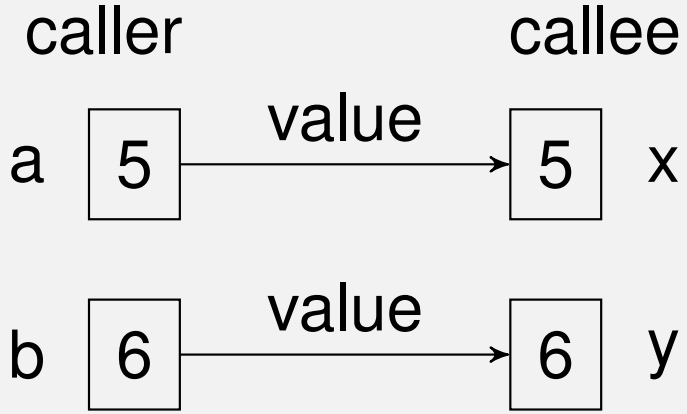
- Pass by value-result

Input-Output Parameters

- Pass by value-result



- Pass by value-result
  findMax(a,b) $\Rightarrow$ **int** findMax(**int** x,**int** y) {...}

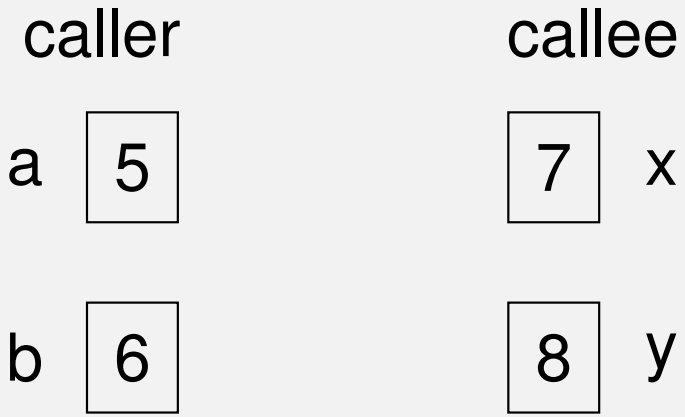


Bên caller (chương trình gọi), biến a có giá trị là 5, b có giá trị là 6.
Khi gọi hàm findMax(a, b), giá trị của a (là 5) được sao chép sang tham số hình thức x bên callee (hàm findMax). Mũi tên "value" từ a sang x chỉ điều này.
Tương tự, giá trị của b (là 6) được sao chép sang tham số hình thức y bên callee. Mũi tên "value" từ b sang y chỉ điều này.
Lúc này, bên callee, x có giá trị 5, y có giá trị 6. Chúng là bản sao độc lập với a và b.

Input-Output Parameters

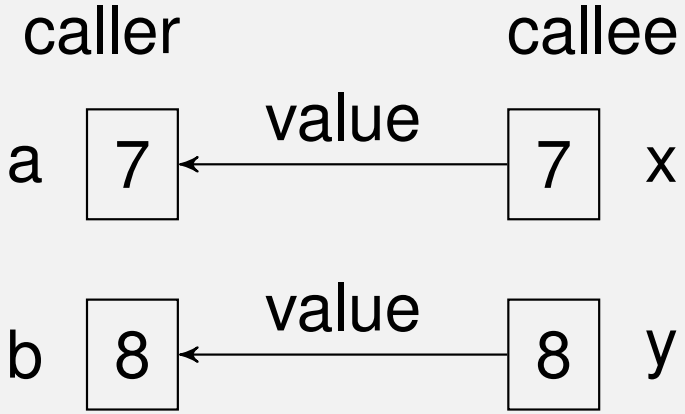
- Pass by value-result

int findMax(**int** x,**int** y) {...}



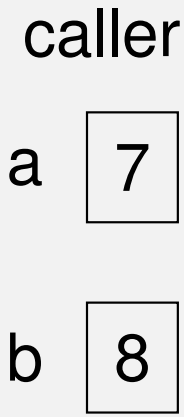
Bên trong hàm findMax (callee), giá trị của các biến cục bộ x và y đã bị thay đổi trong quá trình tính toán (ví dụ: hàm này tìm max rồi gán lại, hoặc làm gì đó khác). Giả sử giá trị của x giờ là 7 và y là 8.
Lưu ý là giá trị của a và b ở bên caller vẫn giữ nguyên là 5 và 6, vì x và y chỉ là bản sao thôi.

- Pass by value-result
  findMax(a,b) $\Leftarrow$ **int** findMax(**int** x,**int** y) {...}



Hàm findMax đã thực thi xong. Theo cơ chế Pass by value-result, giá trị cuối cùng của các tham số hình thức (x và y) sẽ được sao chép ngược trở lại vào các tham số thực tế tương ứng (a và b).
Giá trị cuối cùng của x (là 7) được sao chép ngược về a. Mũi tên "value" từ x sang a chỉ điều này.
Giá trị cuối cùng của y (là 8) được sao chép ngược về b. Mũi tên "value" từ y sang b chỉ điều này.

- Pass by value-result



Quá trình sao chép ngược đã hoàn tất.
Bây giờ, ở bên caller, giá trị của a đã được cập nhật thành 7 và giá trị của b đã được cập nhật thành 8.

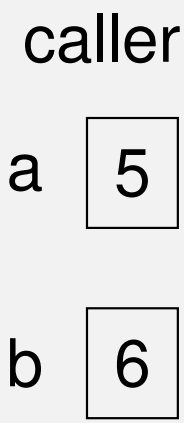
Input-Output Parameters

- Pass by value-result
- Pass by reference

Input-Output Parameters

- Pass by value-result

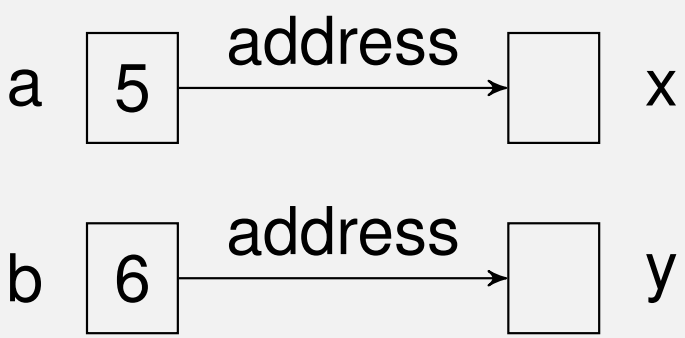
- Pass by reference



Input-Output Parameters

- Pass by value-result

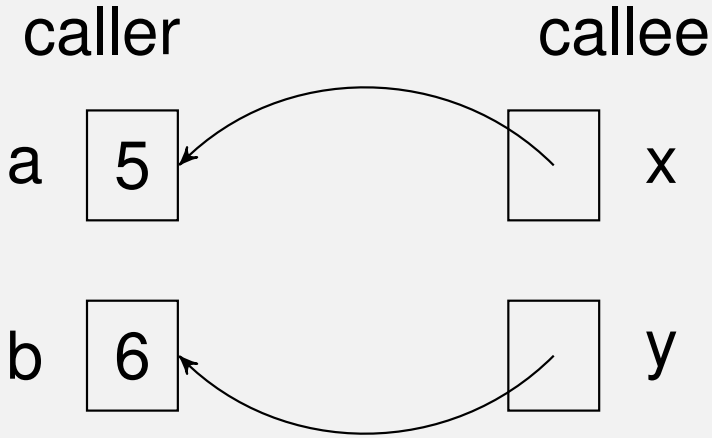
- Pass by reference
  findMax(a,b) $\Rightarrow$ **int** findMax(**int**& x,**int**& y) {...}



- Pass by value-result

- Pass by reference

int findMax(**int**& x,**int**& y) {...}

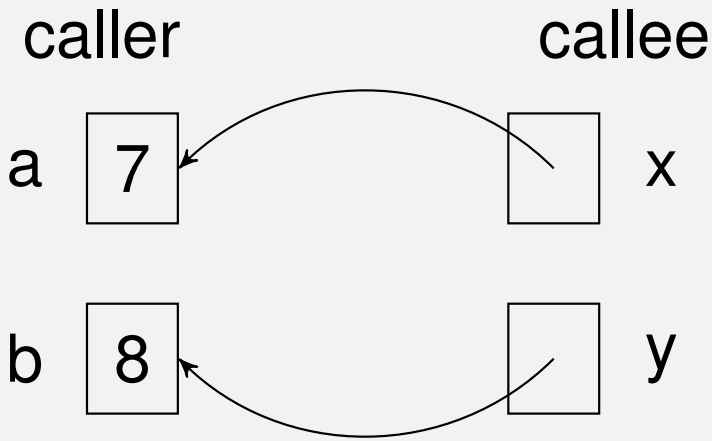


Lời gọi là findMax(a, b), và định nghĩa hàm bên callee là int findMax(int& x, int& y). Cái ký hiệu & ở đây chỉ ra rằng x và y là tham chiếu.
Thay vì sao chép giá trị như Pass by value-result, ở đây địa chỉ bộ nhớ của a được truyền cho x, và địa chỉ bộ nhớ của b được truyền cho y. Mũi tên "address" chỉ điều này. Lúc này, x không phải là một bản sao giá trị của a nữa, mà x trở thành một cái tên khác (bí danh - alias) dùng để chỉ cùng ô nhớ với a. Tương tự, y là bí danh cho b.

- Pass by value-result

- Pass by reference

int findMax(**int**& x,**int**& y) {...}



Hình vẽ dùng các đường cong nối x với a và y với b để thể hiện rằng chúng đang trỏ đến cùng một vị trí trong bộ nhớ. Khi chương trình con (callee) thao tác với x hoặc y, nó đang trực tiếp làm việc trên ô nhớ của a và b ở bên caller.

Input-Output Parameters

- Pass by value-result

- Pass by reference
  findMax(a,b) $\Leftarrow$
  caller

  a | 7 |

  b | 8 |

Giả sử bên trong hàm findMax, giá trị của x được thay đổi thành 7 và y thành 8.
Vì x và y là bí danh cho a và b, nên ngay lập tức, giá trị trong ô nhớ của a trở thành 7 và của b trở thành 8.
Mày có thể thấy giá trị hiển thị bên phía caller cũng thay đổi thành 7 và 8, đồng bộ với giá trị bên callee thông qua mối liên kết tham chiếu.

Input-Output Parameters

- Pass by value-result
- Pass by reference
- Pass by name

Input-Output Parameters

- Pass by value-result
- Pass by reference
- Pass by name
  findMax(a,b) $\Rightarrow$ **int** findMax(**int**$\Rightarrow$ x,**int**$\Rightarrow$ y) {...}

Input-Output Parameters

- Pass by value-result
- Pass by reference
- Pass by name

int findMax(**int**⇒ x,**int**⇒ y) {...}

a ≡ x

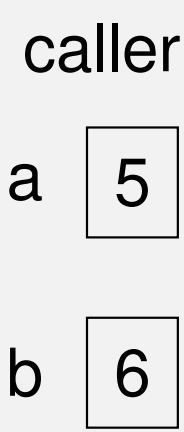
b ≡ y

Input Only Parameters

- Pass by value

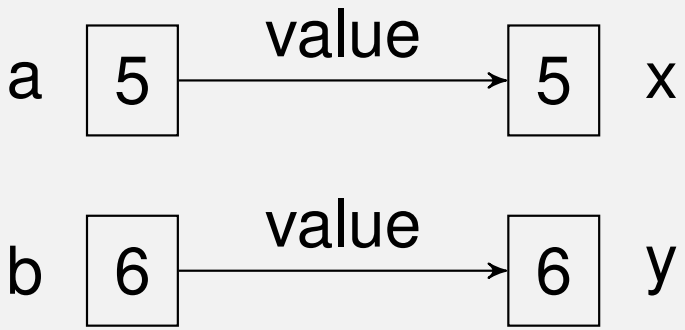
Input Only Parameters

- Pass by value



Input Only Parameters

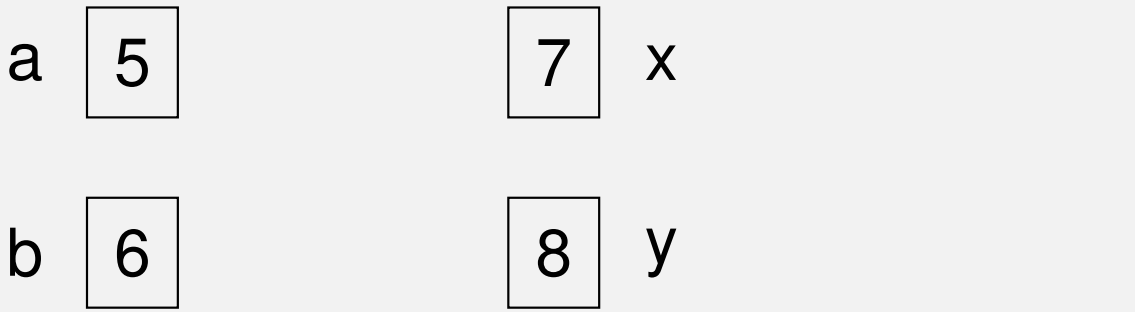
- Pass by value
  findMax(a,b) $\Rightarrow$ **int** findMax(**int** x,**int** y) {...}



Input Only Parameters

- Pass by value

int findMax(**int** x,**int** y) {...}



Input Only Parameters

- Pass by value
  findMax(a,b) $\Leftarrow$
  caller

  a | 5 |

  b | 6 |

(Vẫn giữ giá trị cũ, vì pass by value nên nó chỉ thay đổi giá trị copy)

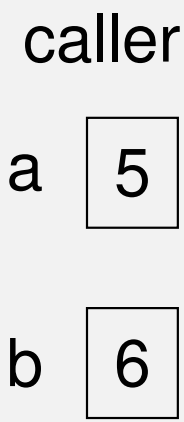
Input Only Parameters

- Pass by value
- Pass by constant reference

Input Only Parameters

- Pass by value

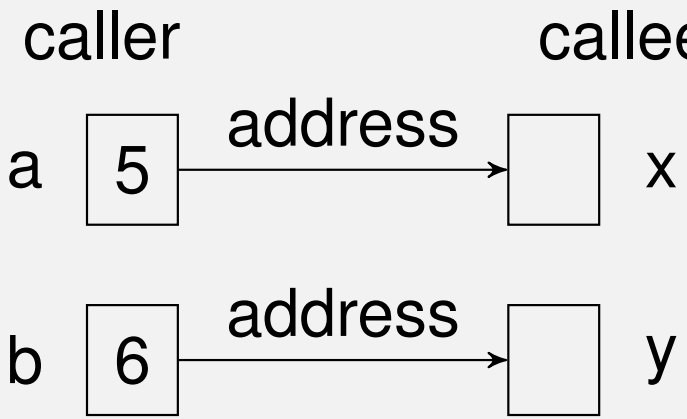
- Pass by constant reference



Input Only Parameters

- Pass by value

- Pass by constant reference
  findMax(a,b) $\Rightarrow$ **int** findMax(**const int**& x,**const int**& y) {...}



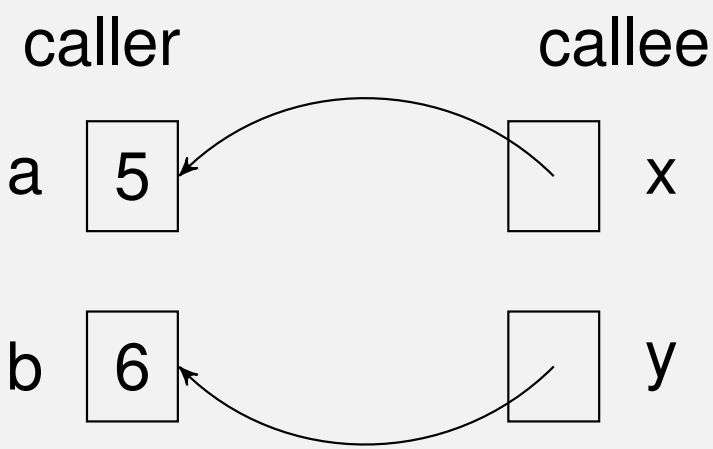
Input Only Parameters

- Pass by value

- Pass by constant reference

int findMax(**const int**& x,**const int**& y)

{...}



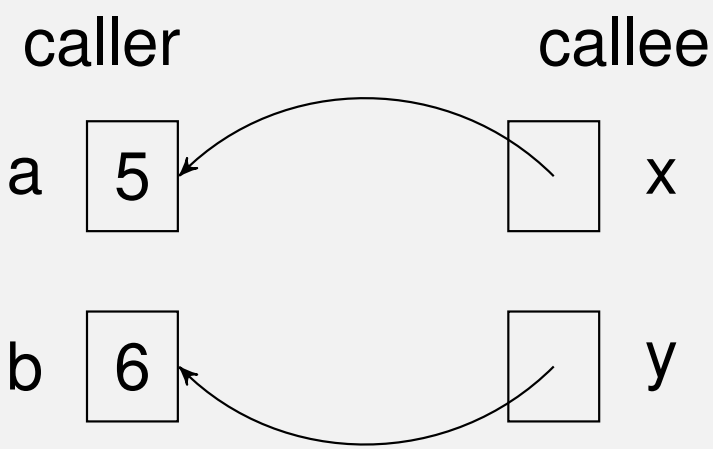
Input Only Parameters

- Pass by value

- Pass by constant reference

int findMax(**const int**& x,**const int**& y)

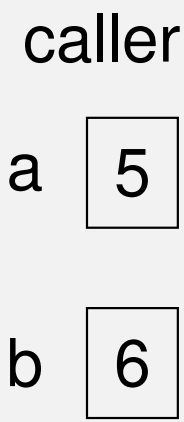
{...}



Input Only Parameters

- Pass by value

- Pass by constant reference
  findMax(a,b) $\Leftarrow$



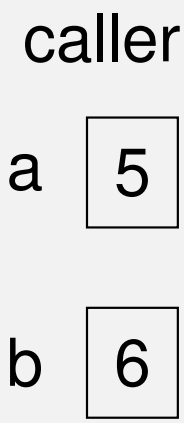
Output Only Parameters

- Pass by result

- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Output Only Parameters

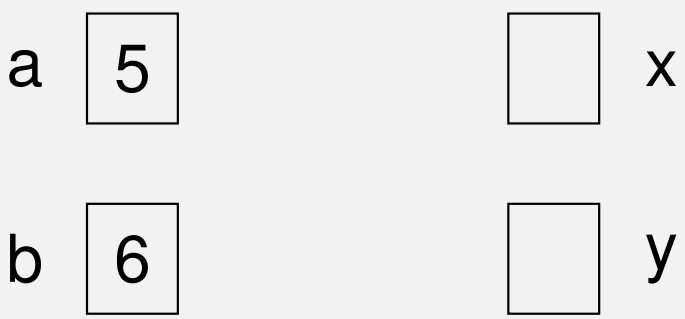
- Pass by result



- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Output Only Parameters

- Pass by result
  findMax(a,b) $\Rightarrow$ **int** findMax(**int** x,**int** y) {...}



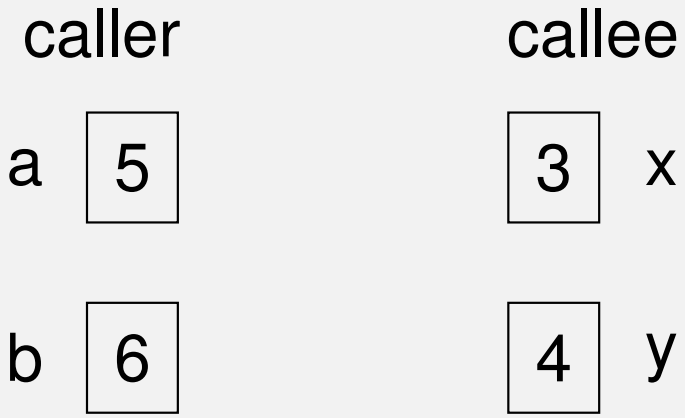
khi truyền vào hàm findMax (callee), các tham số hình thức x và y được tạo ra, nhưng không được gán giá trị ban đầu từ a và b. Chúng coi như chưa có giá trị gì lúc này.

- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Output Only Parameters

- Pass by result

int findMax(**int** x,**int** y) {...}

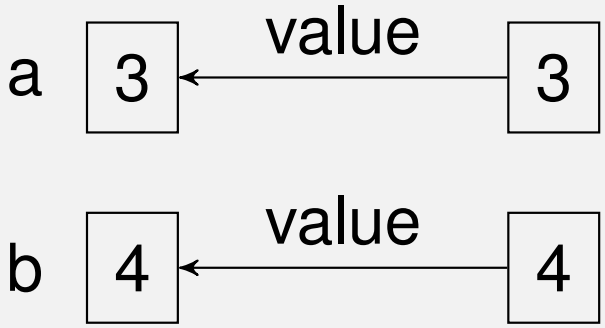


- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Bên trong hàm findMax, chương trình con sẽ tính toán và gán các giá trị nào đó cho x và y (3, 4).
Khi hàm findMax kết thúc, giá trị cuối cùng của x và y sẽ được sao chép ngược trở lại vào biến a và b ở bên caller.
Tức là, dữ liệu chỉ đi một chiều từ callee ra caller thông qua việc sao chép ngược kết quả cuối cùng. Giá trị ban đầu của a và b không ảnh hưởng đến giá trị khởi tạo của x và y.

Output Only Parameters

- Pass by result
  findMax(a,b) $\Leftarrow$
  caller



- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Higher-order Functions

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

Example, in stdlib.h of C, there is a built-in sorting function

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));

int
int_sorter(const void *first_arg ,const void *second_arg)
    int first = *(int*)first_arg;
    int second = *(int*)second_arg;
    if (first < second)  return -1;
    else if (first == second) return 0;
    else return 1;
}
```

Một hàm được gọi là hàm bậc cao khi nó có một trong hai đặc điểm sau (hoặc cả hai):

Nó nhận các hàm khác làm tham số đầu vào (accepts functions as its input parameters). Cái này khá phổ biến.
Nó trả về một hàm như là kết quả đầu ra (as its out parameters). Cái này ít phổ biến hơn, nhưng cần thiết trong lập trình chức năng.

ví dụ trong thư viện chuẩn C (stdlib.h), đó là hàm sắp xếp qsort.

Cái khai báo của hàm qsort là:
void qsort (void * base , size_t nmemb, size_t size , int (* compar) (const void * , const void *)) ;

Để ý cái tham số cuối cùng: int (* compar) (const void * , const void *). Đây chính là một con trỏ hàm (pointer to a function). Điều này có nghĩa là hàm qsort nhận một hàm khác làm tham số đầu vào. Cái hàm được truyền vào này (gọi là compar) có nhiệm vụ so sánh hai phần tử với nhau. qsort sử dụng cái hàm so sánh này để biết được thứ tự của các phần tử khi sắp xếp.

int_sorter:
Hàm này nhận hai con trỏ void*, ép kiểu chúng về con trỏ int, lấy giá trị và so sánh hai số nguyên. Nó trả về -1 nếu first nhỏ hơn second, 0 nếu bằng nhau, và 1 nếu first lớn hơn second. Cái hàm qsort cần một hàm so sánh có kiểu trả về và tham số đúng như vậy.

Higher-order Functions

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

Example, in stdlib.h of C, there is a built-in sorting function

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));

int main() {
    int array[10],i;
    /* fill array */
    for ( i = 0; i < 10; ++i )
        array[i] = 10 - i;
    qsort(array, 10, sizeof(int),int_sorter);
    for (i = 0; i < 10; ++i)
        printf ("%d\n" ,array[i]);
}
```

slide tiếp theo cho thấy cách qsort được sử dụng trong hàm main:

Ở dòng qsort (array , 10, sizeof (int) , int_sorter) ;, hàm qsort được gọi với các tham số: mảng cần sắp xếp (array), số lượng phần tử (10), kích thước mỗi phần tử (sizeof(int)), và quan trọng nhất là truyền hàm int_sorter vào làm tham số cuối cùng.

Như vậy, qsort là một ví dụ về hàm bậc cao vì nó nhận một hàm khác (int_sorter) làm tham số đầu vào để thực hiện công việc của mình.

Về phần hàm bậc cao trả về hàm khác, slide có nhắc đến nhưng không có ví dụ cụ thể trên slide này. Ý tưởng là một hàm có thể tạo ra (hoặc trả về) một hàm mới để sử dụng sau này.

Nói tóm lại, hàm bậc cao là những hàm có khả năng xử lý các hàm khác như dữ liệu thông thường (nhận làm tham số hoặc trả về làm kết quả).

What is non-local environment?

- Deep binding
- Shallow binding

Example, Static scope: z = 6

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}
...
{int x = 4;
 int z = g(f); //deep binding
}
```

1 + 3 (x+y)
4 (h(3))
2 (+ x)
6 (g(f))

Trong phạm vi tĩnh, biến x mà hàm f nhìn thấy được xác định lúc biên dịch dựa trên cấu trúc lồng nhau của code. Hàm f được định nghĩa ở ngoài cùng, nơi int x = 1;. Vì vậy, f luôn nhìn thấy biến x toàn cục có giá trị là 1.
"Deep binding" cũng củng cố điều này: f mang theo môi trường định nghĩa của nó (nơi x = 1).
Khi g(f) được gọi:
Bên trong g, có int x = 2;.
Lệnh return h(3) + x; thực thi. h ở đây chính là hàm f.
Lời gọi h(3) (tức là f(3)) được thực hiện. f sẽ tính x + y.
Vì phạm vi tĩnh và deep binding, f nhìn thấy x toàn cục là 1, và tham số y là 3. Kết quả của f(3) là 1+3=4.
g sau đó tính 4+x (với x cục bộ của g là 2).
Kết quả là 4+2=6.
Kết quả: z = 6 (Slide ghi "Static scope: z = 6", phù hợp với Deep binding trong phạm vi tĩnh).

▸ Skip Scala Example

Nó liên quan đến việc khi mình truyền một hàm làm tham số cho hàm khác, thì cái hàm được truyền đó sẽ truy cập đến các biến nằm ngoài phạm vi của nó (non-local variables) như thế nào.

Slide đề cập đến mấy khái niệm này:

What is non-local environment? (Môi trường không cục bộ là gì?)

Khi một hàm được định nghĩa, nó có thể truy cập các biến được khai báo bên trong nó (biến cục bộ - local variables) và các tham số của nó.

Môi trường không cục bộ (non-local environment) của một hàm là tập hợp các biến nằm ngoài phạm vi trực tiếp của hàm đó, nhưng nó vẫn có thể truy cập được theo quy tắc phạm vi (scoping rule) của ngôn ngữ. Ví dụ, các biến toàn cục, hoặc các biến trong các khối lệnh/hàm bao quanh nó.

Deep binding (Ràng buộc sâu):

Khi một hàm được truyền làm tham số, và nó cần truy cập đến một biến không cục bộ, "Deep binding" có nghĩa là hàm đó sẽ sử dụng giá trị của biến không cục bộ tại môi trường mà hàm đó được ĐỊNH NGHĨA.
Tức là, nó "nhớ" cái môi trường xung quanh nó lúc nó được tạo ra.

Shallow binding (Ràng buộc nông):

Ngược lại với Deep binding, "Shallow binding" có nghĩa là hàm được truyền sẽ sử dụng giá trị của biến không cục bộ tại môi trường mà hàm đó được GỌI (thực thi).
Tức là, nó lấy giá trị của biến không cục bộ trong môi trường hiện tại của cái hàm đang gọi nó.

What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Deep binding: z = 9

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
     int x = 2;
     return h(3) + x; //shallow binding
}
...
{int x = 4;
 int z = g(f); //deep binding
}
```

2 3 4

▸ Skip Scala Example

What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Shallow binding: z = 7

```
int x = 1;
int f(int y){ return x+y; }
                    2  3

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}                   5      2
...
{int x = 4;
 int z = g(f); //deep binding
}
```

Khi g(f) được gọi:
Bên trong g, có int x = 2;.
Lời gọi h(3) (tức f(3)) xảy ra.
Do "Shallow binding" và phạm vi động, khi f được gọi từ bên trong g, f sẽ nhìn thấy biến x gần nhất trong chuỗi gọi hàm, đó là x cục bộ của g có giá trị là 2.
f(3) tính x + y với x=2 (của g) và y=3 (tham số của f). Kết quả là 2+3=5.
g sau đó tính 5+x (với x cục bộ của g là 2).
Kết quả là 5+2=7.

▸ Skip Scala Example

Example in Scala

```
object FileMatcher {
    private def filesHere =
          (new java.io.File(".")).listFiles

    def filesEnding(query: String) =
      for (file <- filesHere;
           if file.getName.endsWith(query))
        yield file}

    def filesContaining(query: String) =
        for (file <- filesHere;
             if file.getName.contains(query))
          yield file

    def filesRegex(query: String) =
       for (file <- filesHere;
            if file.getName.matches(query))
          yield file
  }
```

Example in Scala

```
object FileMatcher {
    private def filesHere =
        (new java.io.File(".")).listFiles

    def filesMatching(query: String,
        matcher: (String, String) => Boolean) = {
      for (file <- filesHere;
          if matcher(file.getName, query))
        yield file
    }
    def filesEnding(query: String) =
        filesMatching(query, _.endsWith(_))

    def filesContaining(query: String) =
        filesMatching(query, _.contains(_))

    def filesRegex(query: String) =
        filesMatching(query, _.matches(_))
}
```

What returns as functions

- Code
- Environment

khi một hàm trả về một hàm khác, thì cái "hàm được trả về " đó bao gồm hai phần:

Code (Mã lệnh): Chính là thân của cái hàm được trả về đó.
Environment (Môi trường): Đây là tập hợp các biến không cục bộ mà cái hàm được trả về cần truy cập đến. Quan trọng là cái môi trường này được "ghi nhớ" lại khi hàm đó được tạo ra hoặc được trả về. Cái này thường gọi là closure.

Example,

```
// Khai báo hàm F trả về một hàm không tham số, trả về int
void->int F () {
    int x = 1; // Biến x cục bộ của hàm F
    int g () {
        return x+1; // Hàm g truy cập biến x không cục bộ (của F)
    }
    return g; // Hàm F trả về hàm g
}
// Trong chương trình chính hoặc một chỗ khác
void->int gg = F(); // Gọi hàm F, kết quả trả về (là hàm g cùng môi trường của nó) được gán cho biến gg
int z = gg(); // Gọi hàm được lưu trong gg (chính là hàm g)
```

What returns as functions

- Code
- Environment

Example,

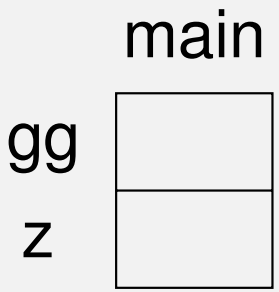
```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```



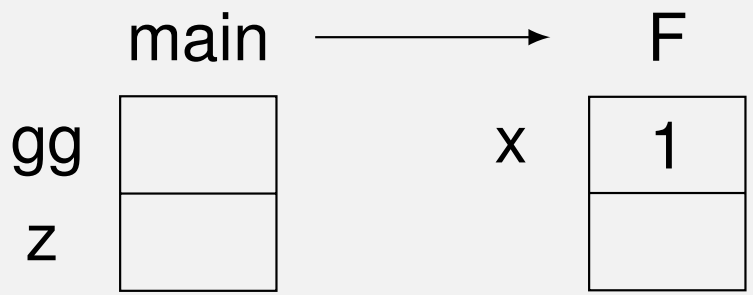
Ở dưới cùng, có cái khối main hoặc chương trình gọi.
Trong main, gọi hàm F() và gán kết quả cho gg.
Khi F được gọi, nó tạo ra môi trường riêng của nó, bao gồm biến cục bộ x với giá trị là 1.
Bên trong F, hàm g được định nghĩa. g cần dùng đến biến x của F.
Hàm F thực hiện lệnh return g;. Lúc này, F không chỉ trả về cái code của hàm g đâu, mà nó còn "đóng gói" (closure) luôn cái môi trường của nó tại thời điểm đó, bao gồm cả biến x=1.
Cái "gói" này (gồm code của g và môi trường x=1) được gán cho biến gg trong main.
Sau đó, lệnh int z = gg(); được thực thi. Cái này gọi cái hàm đã được lưu trong gg.
Khi gg() (chính là g) chạy, nó cần tính x + 1. Nó sẽ nhìn vào cái môi trường mà nó đã được "đóng gói" cùng (closure environment).
Trong môi trường đó, nó thấy biến x có giá trị là 1.
Vì vậy, nó tính 1+1=2.
Giá trị 2 này được trả về và gán cho biến z.

What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

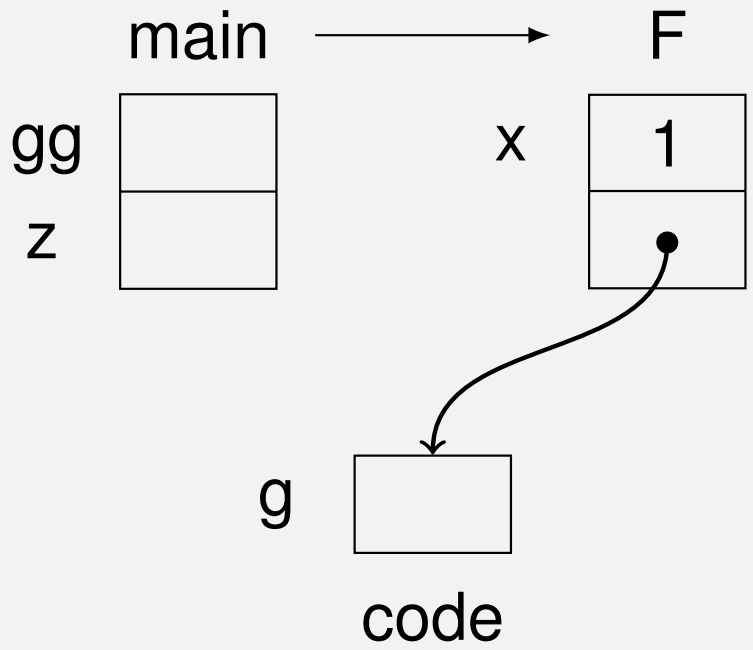


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

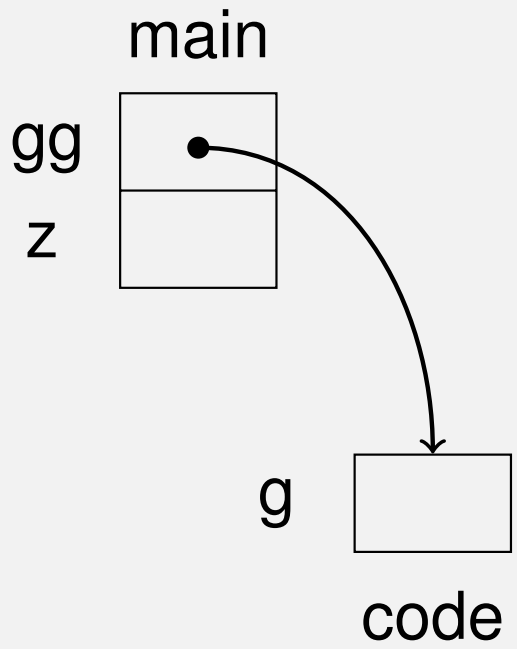


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

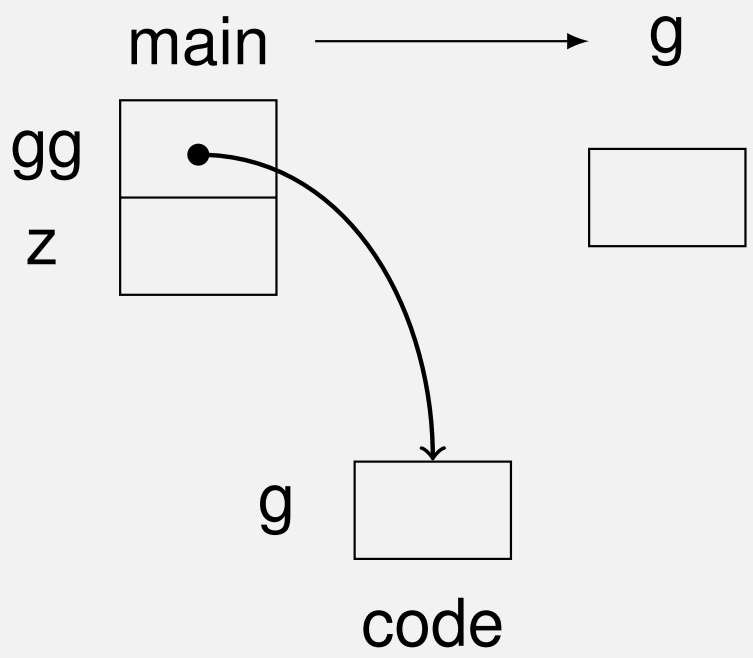


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

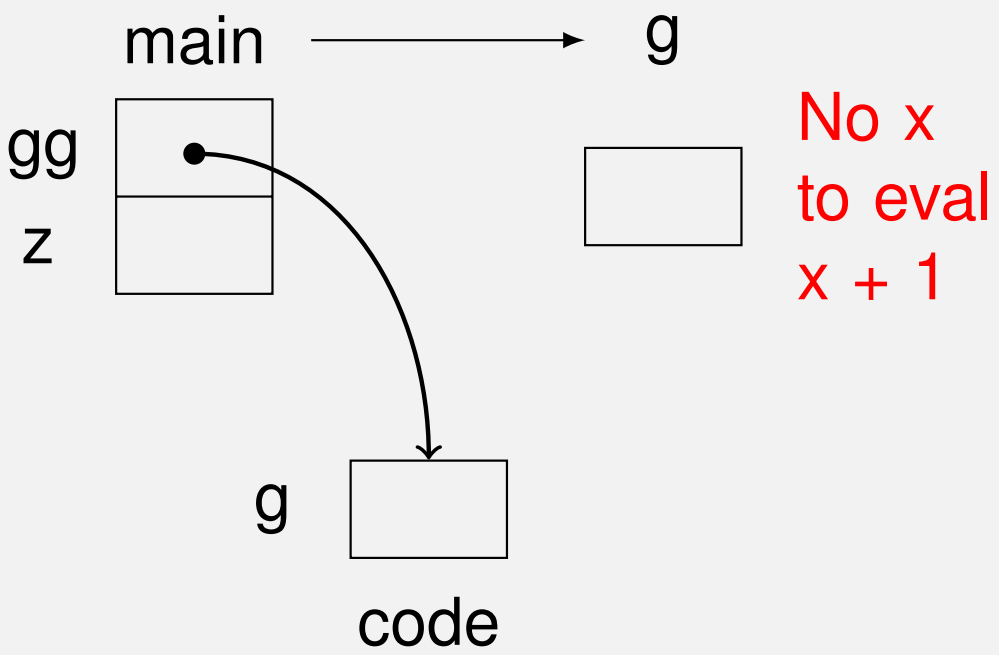


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```



Summary

- Subprogram mechanisms
  - Simple Call-Return
  - Recursive Call
  - Exception
  - Coroutine
  - Scheduled Call
  - Tasks
- Parameter Passing
- Higher-order Functions