UNIT 1

INTRODUCTION

Unit-03/Lecture-01

•Two fundamental abstraction facilities

–Process abstraction

•Emphasized from early days

–Data abstraction

•Emphasized in the1980s

Fundamentals of Subprograms [RGPV JUNE 2011 (5 MARKS)]

General Subprogram Characteristics

•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.

Basic Definition

•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

•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 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

•Function declarations in C and C++ are often called prototypes

•A subprogram declaration provides the protocol, but not the body, of the subprogram

•Parameter: A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram

•Argument: An actual parameter represents a value or address used in the subprogram call statement

Argument/Parameter Correspondence

•Positional

•The binding of actual parameters (arguments) to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth

•Safe and effective

•Keyword

•The name of the formal parameter to which an actual parameter (argument) is to be bound is specified with the actual parameter

•Advantage: Parameters can appear in any order, thereby avoiding parameter correspondence errors

•Disadvantage: User must know the formal parameter's names

Parameters

Actual/Formal Parameter Keyword Correspondence: Python Example

sumer

(length = my_length

,list = my_list

,sum = my_sum

)

Parameters: length, list, sum

Arguments: my_length, my_list, my_sum

Unit-03/Lecture-02

Formal Parameter Default Values

•In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)

•In C++, default parameters must appear last because parameters are position-ally associated

•Variable numbers of parameters

•C# methods can accept a variable number of parameters as long as they are of the same type

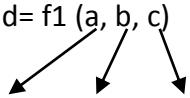
—the corresponding formal parameter is an array preceded by parameters.

•In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.

•In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk

- **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. It is Safe and effective

Caller d= f1 (a, b, c)



int f1(int x, float y, int z)

- **Keyword**- The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter

In Ada, Sumer (Length => My_Length, List => My_Array, Sum => My_Sum);

Advantage: Parameters can appear in any order.

Disadvantage: User must know the formal parameter's names.

Unit-03/Lecture-03

Procedures and Functions [RGPV JUNE 2014 (7 MARKS)]

•There are two categories of subprograms

•Procedures are collection of statements that define parameterized computations. These computations are enacted by single call statements. Procedure defines new statements. For Ex- Ada doesn't have a sort statement, a user can build a procedure to sort arrays of data and use a call to that procedure in place of unavailable sort statement.

•Functions structurally resemble procedures but are semantically modelled on mathematical functions. If a function is a faithful model, it produces no side effects that are; it modifies neither its parameter nor any variables defined outside the function. Such a pure function returns a value that is only desired effect.

•They are expected to produce no side effects.

•In practice, program functions have side effects.

Design Issues for Subprograms

•Are local variables static or dynamic?

•Can subprogram definitions appear in other subprogram definitions?

•What parameter passing methods are provided?

•Are parameter types checked?

•If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?

•Can subprograms be overloaded?

•Can subprogram be generic?

Local Referencing Environments [RGPV JUNE 2014 (7 MARKS)]

[RGPV JUNE 2013 (5 MARKS)]

Variables that are defined inside subprograms are called local variables. Local variables can be either static or stack dynamic "bound to storage when the program begins execution and are unbound when execution terminates."

Advantages of using stack dynamic:

a. Stack dynamic variables are essential for recursive subprograms.
b. Flexibility they provide the subprogram.
c. Storage for locals is shared among some subprograms.

Disadvantages:

a. Allocation/de allocation time- There is the cost of the time required to allocate initialize and de allocate such variables for each call to the subprogram.
b. Indirect addressing "only determined during execution."- Access to stack dynamic local variables must be indirect whereas accesses to static variables can be direct. This indirectness is because the place in the stack where a particular local variable will reside can be determined only during execution.
c. Finally when all local variables are stack dynamic, subprograms cannot be history sensitive, that is, they cannot retain data values of local variables between calls.

Ex- for a history- sensitive subprogram is one whose task is to generate pseudo random numbers. Each call to such a subprogram computes one pseudorandom number, using the last one it computed.

Advantages of using static variables:

a. Static local variables can be accessed faster because there is no indirection.

b. No run-time overhead for allocation and de allocation.

c. Allow subprograms to be history sensitive.

Disadvantages:

a. Inability to support recursion.

b. Their storage can't be shared with the local variables of other inactive subprograms.

Ex:

```
int adder( int list[ ], int listlen) {

static int sum = 0;

int count; //count is stack-dynamic

for (count = 0; count < listlen; count++)

sum += list[count];

return sum;

}
```

Here the variable sum is static and count is stack dynamic. Ada, C++, Java and C# have only stack dynamic local variables.

Unit-03/Lecture-04

Parameter Passing Methods [RGPV JUNE 2014(7 MARKS)]

[RGPV JUNE 2013 (10 MARKS)]

Following are the ways in which parameters are transmitted to and/or from called subprograms

– Pass-by-value

– Pass-by-result

– Pass-by-value-result

– Pass-by-reference

– Pass-by-name

Formal parameters are characterized by one of three distinct semantic models-

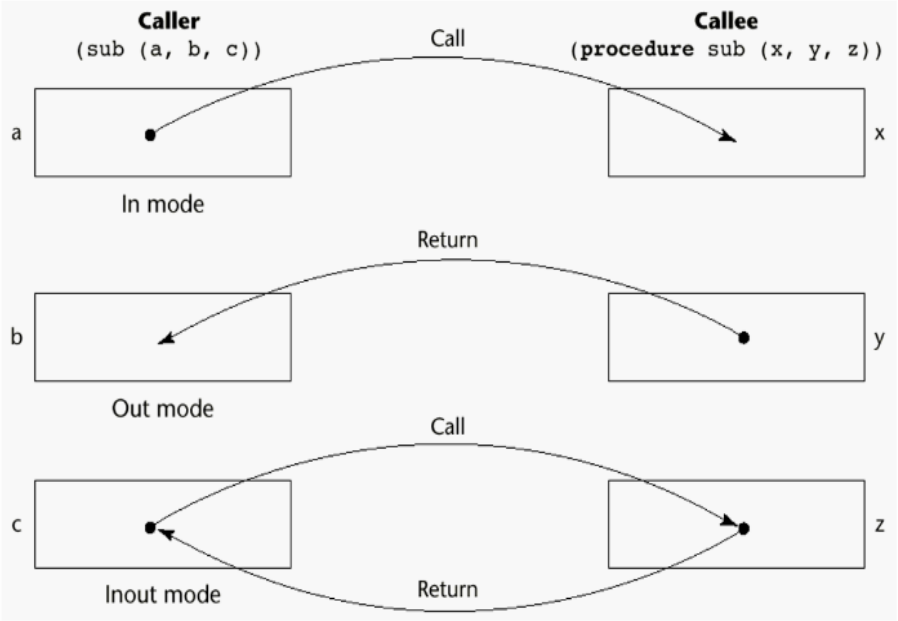
i. They can receive data from the corresponding actual parameters.
ii. They can transmit data to the actual parameter.
iii. They can do both.

These three semantic models are called in mode, out mode, and in out mode respectively.

Data transmission takes place by two ways-

- An actual value is moved.
- An access path is transmitted.

Models of Parameter Passing



1. **Pass by Value (In mode)**

The value of the actual parameter is used to initialize the corresponding formal parameter

–Normally implemented by copying

–Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)

–Disadvantages (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)

–Disadvantages(if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

2. **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 problem: sub(p1, p1); whichever formal parameter is copied back will represent

the current value of p1

3. **Pass by Value Result (In out 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

4. **Pass by Reference (In out 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)

5. **Pass By Name (In out 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

Unit-03/Lecture-05

Overloaded Subprograms

An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment

–Every version of an overloaded subprogram has a unique protocol

–Unique protocol means that the number, order, or types of parameters must differ or the return type must differ

•C++, Java, C#, and Ada include predefined overloaded subprograms and also allow users to write multiple versions of subprograms with the same name.

Because each version of an overloaded subprogram has a unique parameter profile, the compiler can disambiguate occurrences of calls to them by the different type parameters.

But when parameter coercions are allowed, complicate the disambiguation process enormously. The issue is that if no method's parameter profile matches the number and types of the actual parameters in a method call, but two or more methods have parameter profiles that can be matched through coercions which method should be called. A language designer has to decide how to rank all the different coercions so that the compiler can choose the method that best matches the call.

In Ada, the return type of an overloaded function can used to disambiguate calls. Therefore two overloaded functions can have the same parameter profile and differ only in their return types. Ex- If an Ada program has 2 functions named Fun, both of which take an Integer parameter but one returns an Integer and one returns a float the following call would be legal:-

```
A, B: Integer;

A: = B + Fun (7);
```

In this call, the call to Fun is bound to the version of Fun that returns an Integer because choosing the version that returns a float would cause a type error.

Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls.

Ex- In C++

```
void fun (float b= 0.0);

void fun();

.....

fun ();
```

The call is ambiguous and will cause a compilation error.

Unit-03/Lecture-06

Generic Subprograms

A generic or polymorphic subprogram takes parameters of different types on different activations

• Overloaded subprograms provide ad hoc polymorphism

• 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

Examples of parametric polymorphism: C++ template

```
<Class Type>

Type max (Type first, Type second) {

return first > second ? first: second;

}
```

• The above template can be instantiated for any type for which operator > is defined. For example

```
int max (int first, int second) {

return first > second? first : second;

}
```

Another example of Generic Subprogram in C++

A generic subprogram (function) for swapping integer, float and character type elements-

```
# include <iostream.h>

# include <conio.h>

void swap (T &a, T &b)

{
```

```
                    T temp;
                    temp = a;
                    a = b;
                    b = temp;
}
Void main ()
{
                    Int x=10, y=20;
                    Float a=1.2, b= 2.4;
                    Cout<<"Swapping integer values \n";
                    Cout<<"values of x and y before swapping\n";
                    Cout<<" X="<<x<<"Y="<<y;
                    Swap (x,y);
                    Cout<<" values of x and y after swapping\n";
                    Cout<<" X="<<x<<"Y="<<y;
                    Cout<<"Swapping float values \n";
                    Cout<<"values of a and b before swapping\n";
                    Cout<<" A="<<a<<"B="<<b;
                    Swap (a,b);
                    Cout<<" values of a and b after swapping\n";
                    Cout<<" A="<<a<<"B="<<b;
                    getch();
}
```

Unit-03/Lecture-07

Co routines **[RGPV JUNE 2011(10 marks)]**

[RGPV JUNE 2014 (10 marks)] [JUNE 2013(5 MARKS)]

• A co routine is a subprogram that has multiple entries and controls them itself

• Also called symmetric control: caller and called co routines are on a more equal basis

• A co routine call is named a resume

• The first resume of a co routine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the co routine.

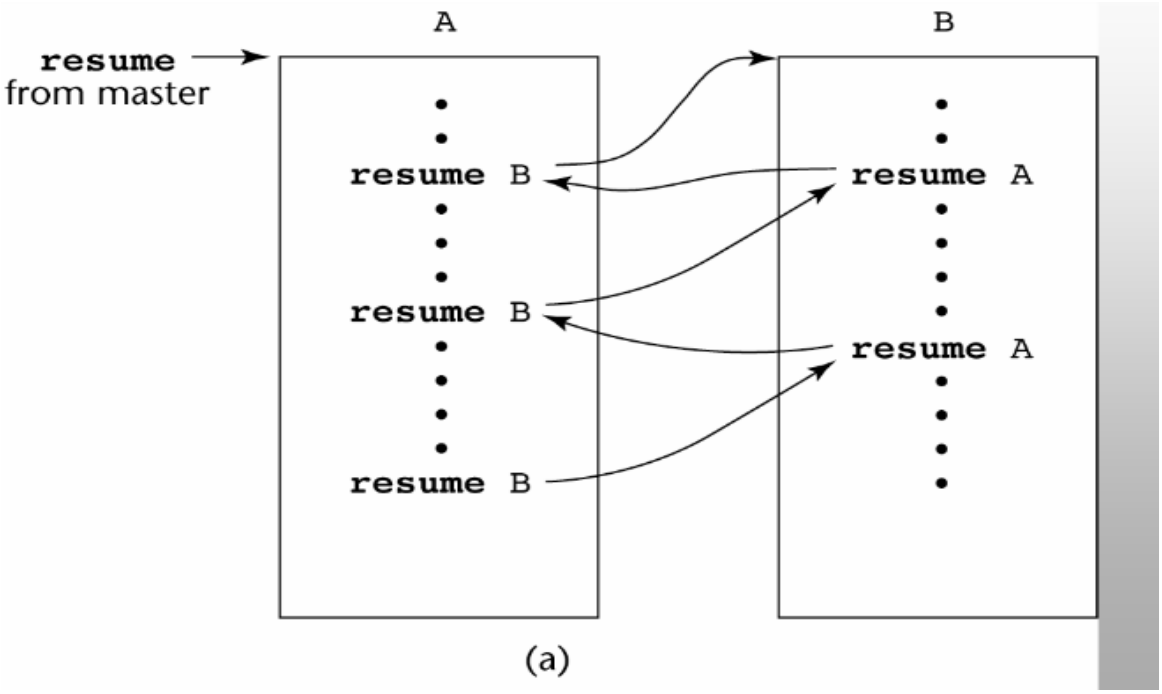
• Co routines repeatedly resume each other, possibly forever

• Co routines provide quasi-concurrent execution of program units (the co routines); their execution is interleaved, but not overlapped

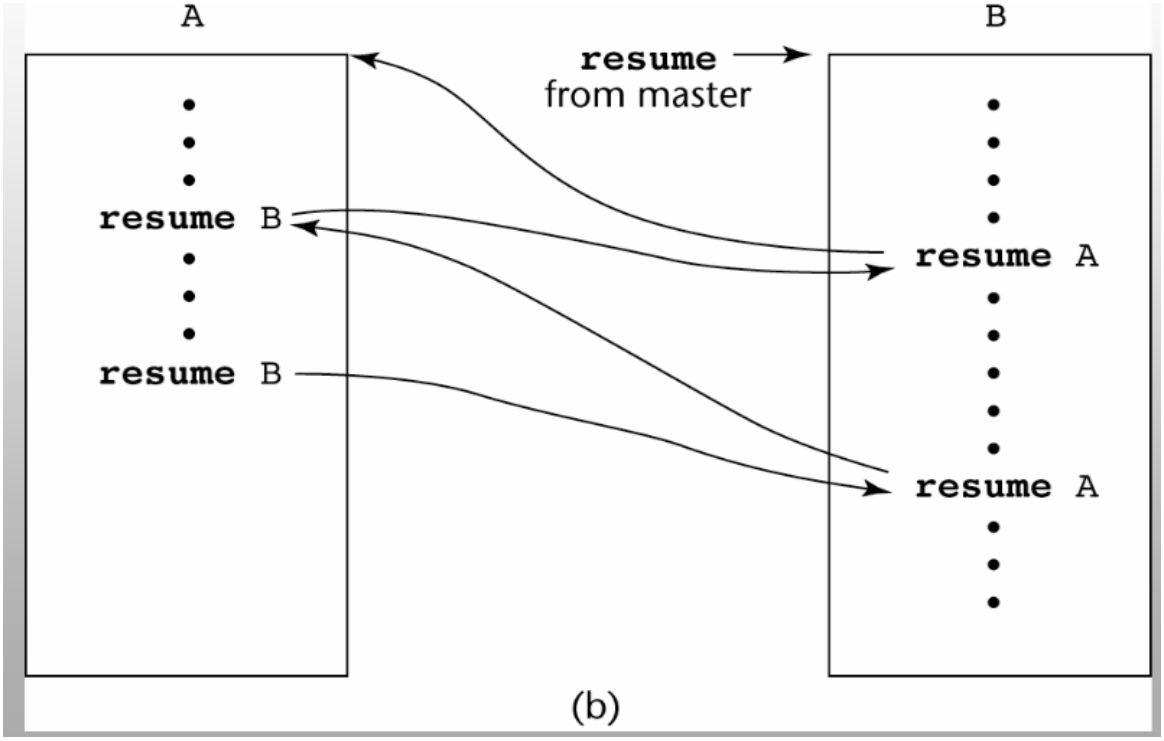
Only one co routine executes at a given time. Rather than executing to their other ends, co routines often partially execute and then transfer control to other routines. When restarted a co routine resumes execution just after the statement it used to transfer control elsewhere. This sort of interleaved execution sequence is related to the way multiprogramming operating systems work. In the case of co routines, this is sometimes called Quasi Concurrency.

Co routines are created in an application by a program unit called Master Unit, which is not a co routine. When created, co routine executes their initialization code and then returns control to that master unit.

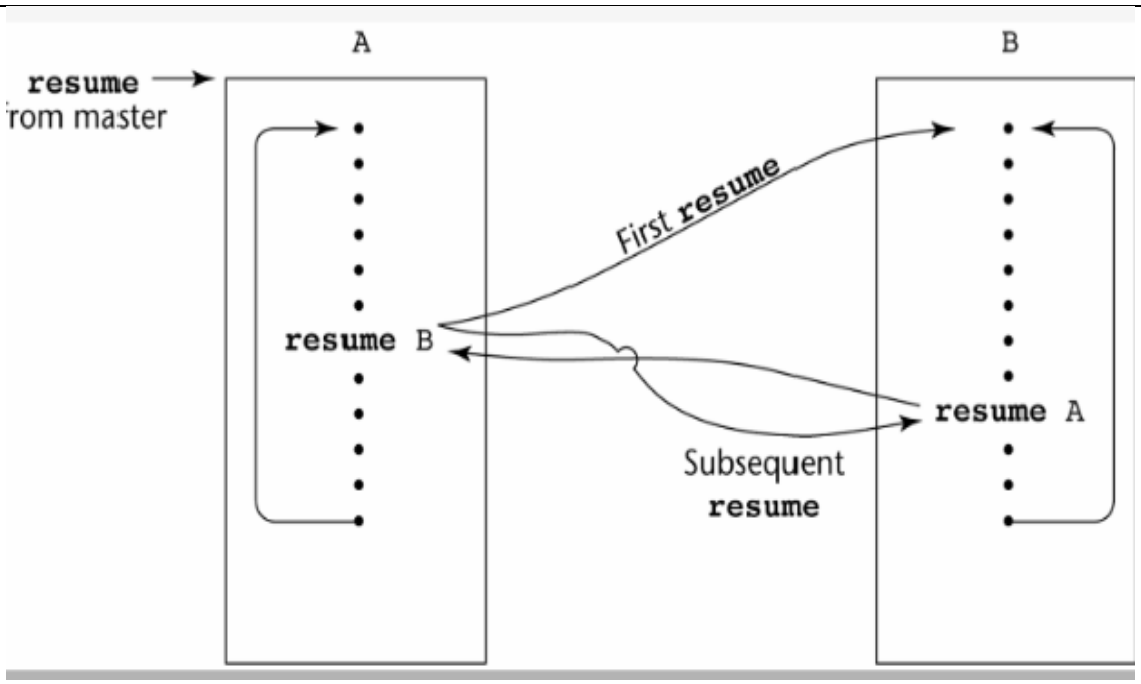
Possible Execution controls



Possible Execution controls



Possible Execution Controls with Loops


A
B
resume
from master
First resume
resume B
resume A
Subsequent
resume

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain scope visibility and lifetime of variable?	JUNE 2012	10 marks
Q.2	What do you mean by current instruction pointer and current environment pointer? How is it used for recursive subprograms?	JUNE 2012	10 marks
Q.3	Discuss the design issues for subprograms?	JUNE 2012	10 marks
Q.4	What do you mean by co routines? Explain	JUNE 2011	10 marks
Q.5	Write short note on Fundamentals of subprograms?	JUNE 2011	5 marks
Q.6	What is the difference between procedure and functions? Explain with suitable example?	JUNE 2014	7 marks
Q.7	Explain the following implementation models for parameter passing with an example. i) Pass -By-value ii) Pass-By-value-Result iii) Pass-By-Reference iv) Pass-By-Name	JUNE 2014	7 marks
Q.8	What do you mean by referencing environment of sub program? Discuss its several components?	JUNE 2014	7 marks
Q.9	What do you understand by coroutines? How do we achieve control transfer between coroutines?	JUNE 2014	7 marks
Q.10	Explain the following (i) Coroutines (ii) Local Referencing Environment	JUNE 2013	10 marks
Q.11	What are the three semantic models of parameter passing methods? Explain?	JUNE 2013	10 marks