UNIT-3

SUBPROGRAMS AND BLOCKS

Subprograms are the fundamental building blocks of programs and therefore among the most important concepts in programming language design.

① Fundamentals of subprograms -

→ General subprogram characteristics -

(1) Each subprogram has a single entry point.

(2) The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.

(3) Control always return to the caller when the subprogram execution terminates.

→ Basic definitions -

(1) Subprogram definition - describes the interface to and the subprogram abstraction.

(2) Subprogram call - Explicit request that a specific subprogram be executed

(3) Subprogram header - first part of the definition.

(4) Format - data type subprogram_header (parameters);

(5) Parameter profile - Contains the number, order and types of its formal parameters

(6) Protocol - parameter profile plus, if it is a function, its return type.

(7) Prototypes - function declaration.

→ Parameters -

(1) Formal Parameters - parameters in the sub-program header.

(2) Actual Parameters - bound to the formal parameters.

(3) Positional Parameters - First actual parameter is bound to the first formal parameter and so forth.

(4) Keyword Parameters - Name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call.

(5) Array formal Parameter - The hash item can be followed by a single parameter preceded by an asterisk.

→ Procedure and Functions - (Categories of sub-program)

(1) Function can return value but procedure do not.

(2) Function can be used as procedures but vice versa is not possible

(3) Procedure can produce results in the calling program unit by -

(i) Procedure can change variable of calling program unit - that are visible

(ii) Through formal parameters

② Scope and lifetime of a variable -

Scope of a variable is from its declaration to the end of the method but the lifetime of a variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

③ Static scope and dynamic scope -

(lexical scope) Static scope is named because the scope of a variable can be statically determined, that is prior to execution.

Dynamic scope is based on the calling sequence of subprograms, not their spatial relationship to each other. Thus, the scope can be determined only at run time.

```
function big() {
  function sub1() {
    var x = 7; //ignored
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

```
function big() {
  function sub1() {
    var x = 7;
  }
  function sub2() {
    var y = x;
    var z = 3;
  }
  var x = 3;
}
```

Can't decide

④ Design Issues for subprograms -

Overloaded subprogram - one that has the same name as another subprogram

Generic subprogram - computation can be done on data of different types in different calls.

Closure - Nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program.

Design issues for subprograms are -

(1) Are local variables statically or dynamically allocated?
(2) Can subprogram definitions appear in other subprograms definitions?
(3) What parameter-passing method or methods are used?
(4) Are the types of actual parameters checked against the types of the formal parameters?
(5) If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
(6) Can subprograms be overloaded?
(7) Can subprograms be generic?
(8) If the language allows nested subprograms, are closure supported?

⑤ Local Referencing Environments -

→ local variables -

Variables that are defined inside subprograms. local variables can be static or stack dynamic (they are bound to storage when the subprogram begins execution and are unbound from storage when that execution terminates).

```
int adder (int list[], int listlen) {
  static int sum = 0;      // Static local variable
  int count;               // Stack dynamic local variable
  for (count=0; count < listlen; count++)
    sum += list [count];
  return sum;
}
```

→ Nested subprograms - subprogram within another subprogram.

⑥ Parameter Passing Methods –

→ Semantics models of parameter passing – Three models are –

(1) In mode – They can receive data from the corresponding actual parameter.

(2) Out mode – They can transmit data to the actual parameter.

(3) Inout mode – They can do both.

→ Implementation models of parameter passing –

(1) Pass-by-value – The value of the actual parameter is used to initialize the corresponding formal parameter. (in mode semantics)

(2) Pass-by-result – No value is transmitted to the subprogram and it return a result or variable (out mode semantics)

Problem – void Finer (out int x, out int y) {

x = 17;

y = 35;

}

f. Finer (out a, out a);

y is assigned the value of a but x is assigned to 17.

(3) Pass-by-value-result – Combination of pass-by-value and pass-by result (inout mode semantics). Also called pass-by copy.

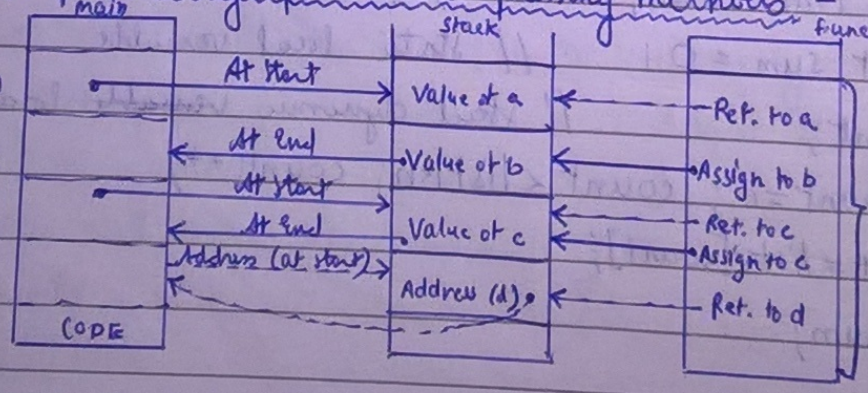
(4) Pass-by-Reference – Transmits an access path, usually just an address, to the called subprogram. (inout mode semantics)

Problem – void fune (int &first, int &second)

function calls as fun (total, total) Then collision occurs.

(5) Pass-by-Name – When parameters are passed by name, the actual parameter is, in the subprogram. (inout mode semantics)

→ Implementing parameter-passing methods –



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

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

⑦ Overloaded subprograms –

It is a subprogram that has the same name as another subprogram in the same referencing environment.

```
void func (float b = 0.0);
void func ();
....
fun ();   // The call is ambiguous & will cause a compilation error.
```

⑧ Generic Subprograms –

Parametric polymorphism subprograms are often called generic subprograms. that means it takes generic parameters that are used in type expressions that describe the types of the parameters of the subprogram

→ Generic functions in C++ – template <template parameters>

Eg – template <class Type>

```
Type max (Type first, Type second) {
    return first > second ? first : second;
}
```

→ Generic methods in Java 5.0 – generic class <T>

Eg – public static <T> T doIt (T[] list) { ... }

Call of to doIt → doIt<String> (mylist);

→ Generic methods in C# 2005 –

```
class MyClass {
    public static T DoIt <T> (T pt) { ... } }
```

→ Generic functions in F# –

```
let getLast (a, b, c) = c;;
```

⑨ Design Issues for functions –

The following design issues are specific to functions –

(1) Are side effects allowed?

(2) What types of values can be returned?

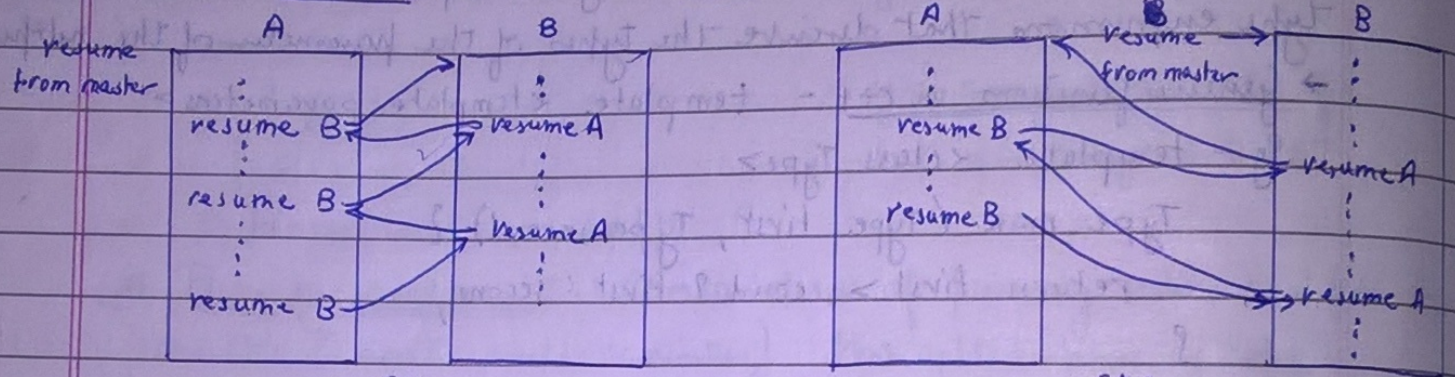
(3) How many values can be returned?

(10) ~~The~~ Overloaded operators -

```
Eg. Complex operator + (Complex &second) {
      return Complex (real + second.real, imag + second.imag);
    }
```

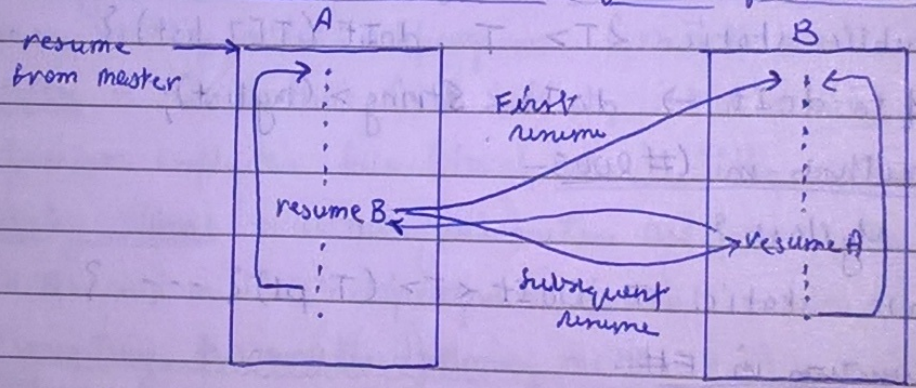
(11) Co-routines - Special kind of subprogram

It can have multiple entry points, which are controlled by the coroutines themselves. Coroutines must be history sensitive and thus have static local variables. Coroutines often begin at points other than its beginning. Because of this, the invocation of coroutine is called resume rather than call.



(a) (b)

Two possible execution control sequences for two coroutines without loops



Coroutine execution sequence with loops