Subprograms and implementations

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

The advantages of having subprograms in the programming language are:

1) It increases the readability of a program by structuring the program in terms of subprograms.
2) Subprograms helps to reuse a set of program lines.
3) The reuse of programs result in saving coding time and memory.
4) Subprograms provides abstraction to the implementation.

Fundamentals of Subprograms (characteristics)

1) Each subprogram has a single entry point.
2) The calling 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 returns to the caller, when the subprogram execution terminates.

Basics

- whenever we define a subprogram, it is compulsory to declare the following things:
  1) Subprogram header/prototype/declaration
  2) Subprogram call
  3) Subprogram definition.

```
int Sum(int, int);          ---> Subprogram declaration/prototype

main()                      ---> Formal parameters
{
  int a, b;
  a=10;                           actual parameters.
  b=20;
                            ---> Subprogram call
  printf("The Sum is %d", Sum(a,b));
}
  int Sum(int x, int y)     ---> Subprogram definition
   {
     return x+y;
   }
```

Subprogram header/declaration

- It is the first part of a subprogram. It serves various purposes:

  1) It tells the compiler that it is a subprogram.

  2) When more than one subprograms with are present, the header differentiates one from another by using the function name or number and type of arguments or return type.

Subprogram Call

- It is a request to execute that specific subprogram.

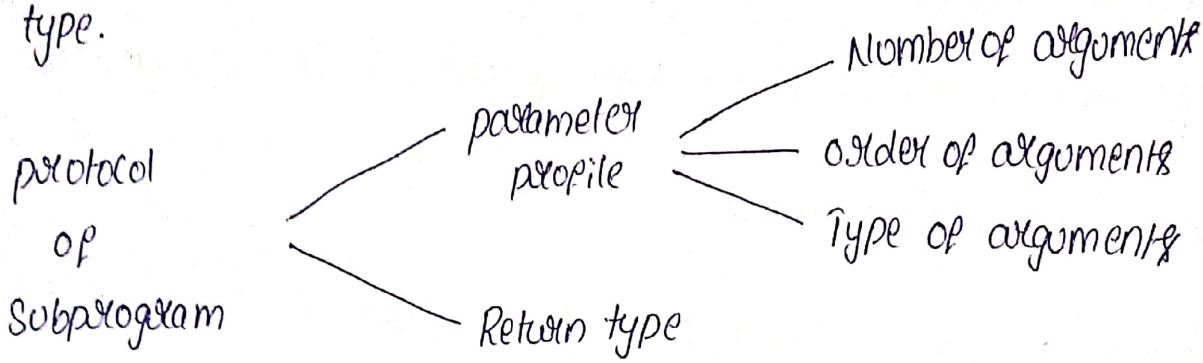
Subprogram definition

- Definition tells the actual action, which is performed by the subprogram. In C program, the body of the definition is delimited by curly braces.

Parameters

- The parameter profile of a subprogram contains the number, order and types of its formal parameters.

The protocol of a subprogram is its parameter profile plus and its return type.



Procedures and Functions

- There are two distinct categories of subprograms
  1) procedures
  2) Functions
- All subprograms are collections of statements that define parameterized computations.
- Functions return values, while procedures do not.

Design issues for subprograms

- The various design issues associated with functions are:
  1) Are local variables statically or dynamically allocated.
  2) can subprogram definitions appear in other subprogram definitions?
  3) what parameter-passing method or methods used?
  4) Are the types of the 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 closures supported.

Local referencing environments

- Subprograms can define their own local variables, there by defining their own local referencing environments.
- Variables defined inside Subprograms are called local variable, because their scope is usually the body of the Subprogram in which they are defined.
- Local variables memory allocated will be either Stack dynamic or Static.

Ex: Stack - C/C++

Static - FORTRAN

Storage class	Scope/visibility/access	memory	Default value
Auto	local/Block	Stack	Garbage
Register	local/Block	register	Garbage
static	File	Static	zero
Extern	Global/application	?	zero.

Stack Variable ⇒ Visibility/Scope - From declaration until scope is exited.

lifetime - From declaration until scope is exited.

Static Variable ⇒ lifetime - program runtime.

Visibility - determined by access modifiers.

Parameter passing

Nested Subprograms

- Nesting subprograms is to create a heirarchy of both logic and scopes.
- If a subprogram is needed only within another subprogram, we can nest the subprograms and hide the inner subprogram.
  - only Algol 60, Pascal, Ada and Algol 68 supports nested subprograms. JavaScript, Python, Ruby and Lua also supports them.
  - C, C++, Java doesn't support nested functions / nested subprograms.

Ex: JavaScript

```
function f1(d)
{
  function f2(n)
  {
    return d+n;
  }
  return f2;
}
```

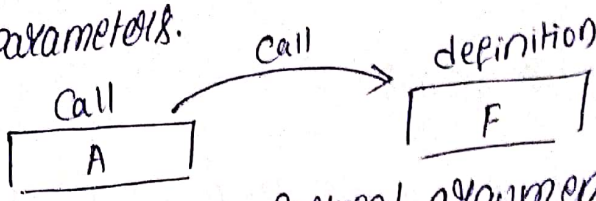
→ C, C++, Java doesn't support writing a function definition in another function.

→ like writing f2's definition in f1.

Parameter passing Methods

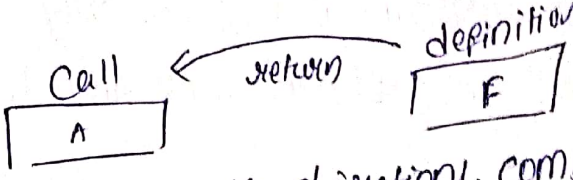
- parameter passing methods are the ways in which parameters are transmitted to and from called subprograms.
- parameter paying methods are based on three semantic models:

1) In-mode - values are transmitted from actual arguments to formal arguments / parameters.



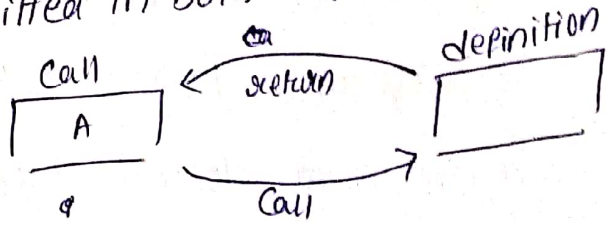
F ← A

2) out-mode - values are transmitted from formal arguments to actual arguments



A ← F

3) In out mode - values are transmitted in both directions. combination of above two.



- There are two conceptual models of how data transmission takes place in parameter transmission
  1) Actual value is copied
  2) An access path to the value is transmitted.
- Based on above conceptual models, the following are the various parameter paying methods:
  1) call by value
  2) call by
     1) pass by value
     2) pass by result
     3) pass by value-result
     4) pass by reference
     5) pass by Name.

Pass by Value - In mode implementation

- The value of the actual argument is used to initialize the corresponding formal parameter. The formal parameters acts as local variables in the subprogram or function.

Ex:
```
Void print (int x)
{
  printf("%d", x);
}
main()
{
  int y=10;
  print(y);
}
```

- The advantage of pass-by-value is that for scalars it is fast in both linking and access time.
- Disadvantage is additional storage is required for formal parameters.

Pass by result - out-mode implementation.

- A value is passed from the formal parameters to actual parameters.

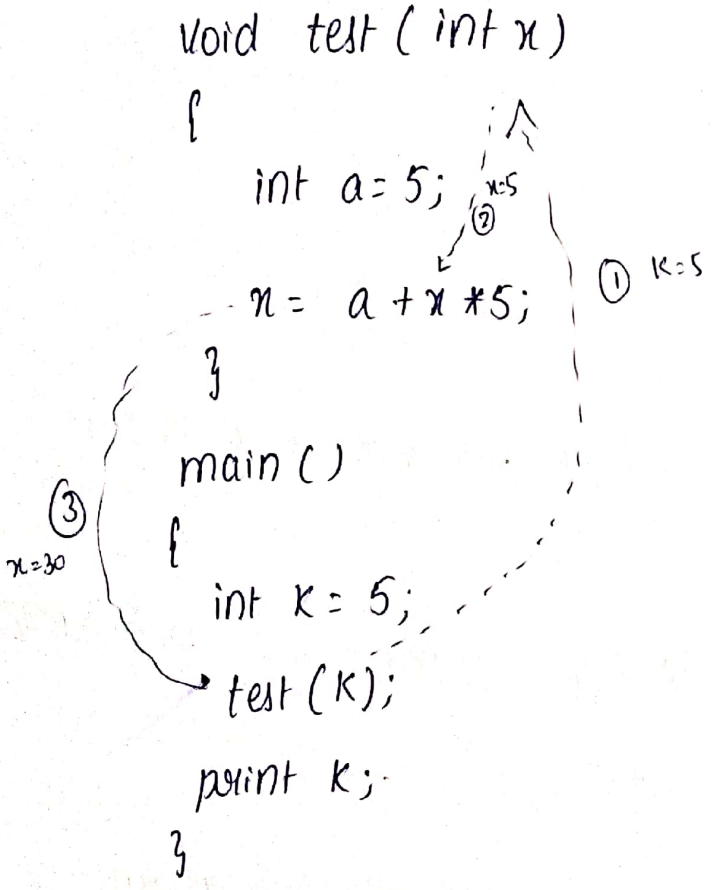
Ex: C++ program with "out" specifier on its formal parameters.

```
void fixer(out int x, out int y)
{
   x=17;
   y=35;
}

p. fixer(out a, out b);
```

→ x is copied into a and y is copied into b.

Pass-by-Value-Result - Inout-mode implementation.

- combination of pass-by-value and pass-by-result.
- It is also called pass-by-copy or call-by-copy.
- In this the actual arguments are copied to the formal parameters at the subprogram entry and copied back at the subprogram termination.



→ K=30 is printed.

Pass by reference - Inout mode implementation.

- Rather than copying the values back and forth like in pass-by-value Result the pass by reference transmits an access path, usually just an address to the called subprogram.
- This provides the access path to the memory cells storing the actual parameter.

Ex:-
```
void test (int *x)
{
  *x = *x + 1;
}

main()
{
 int y = 10;
 test (&y);
 print y;          → y = 11 is printed.
}
```

Pass by Name

It is also an in-out mode implementation of parameter passing.
- When values are passed by name, the actual arguments are textually substituted for the corresponding formal parameter.
- This is both complex and inefficient. They add significant complexity to the program, thereby lowering its readability and reliability.

It is not widely used except for few languages.

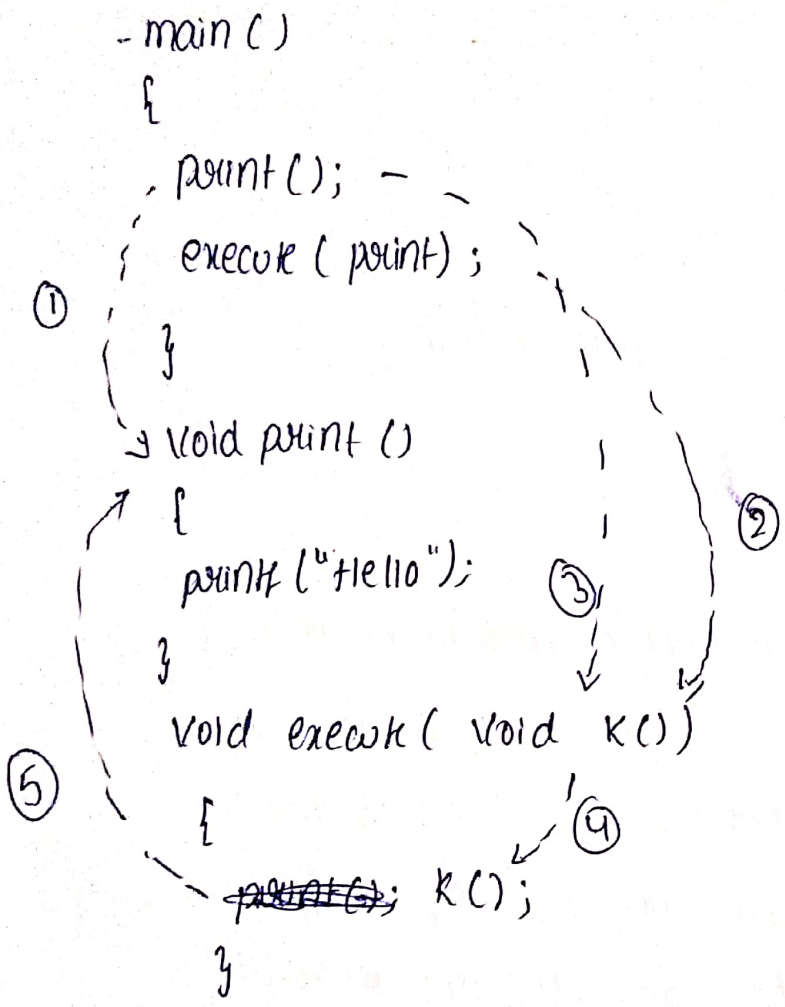
Type-checking parameters

- Types of actual parameters are to be checked for consistency with the types of the corresponding formal parameters.
- Without such type checking, errors are hard to be identified and the program gives undesired results.

Early programming languages such as Fortran 77 and the original version of C, did not require parameter type checking, but most later languages require it.

Parameters that are Subprograms

In programming, there are number of situations where subprogram names can be sent as parameters to other subprograms.



- In C and C++, functions cannot be passed as parameters, but pointers to functions can be passed.

```
void print()
{
 printf("Hello");
}
void execute (void (*f)())
{
 (*f)();
}
```

```
main()
{
 print();
execute (print);
}
```

overloaded subprograms

- overloaded subprogram is a subprogram that has the same name as another subprogram but different from the other in terms of number, order and types of its parameters and possibly in its return type of a function.

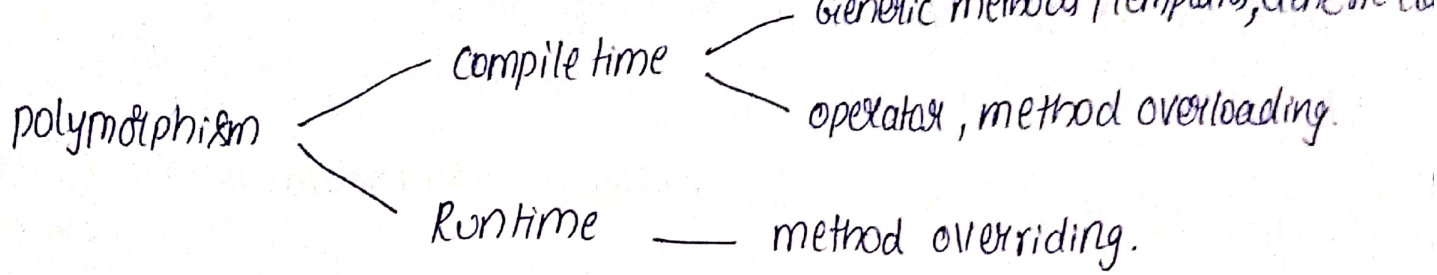
- The meaning of a call to an overloaded subprogram is determined by the actual parameter list.

- C++, Java, Ada and C# include predefined overloaded subprograms.

```
void add(int, int);
void add( int, int, int);

main()
{
  int x=10, y=20, z=30;

  add (x, y);
  add (y, z);
  add (x, y, z);
}
void add(int a, int b, int c)
{
   printf(" %d", a+b+c);

}
void add( int a, int b)
{
  printf(" %d", a+b)
}
```

Generic Subprograms

- Polymorphism is about having many forms. It can be of two types in subprograms



- Ad-hoc polymorphism is a type of ~~polym~~ polymorphism that is provided by overloaded subprograms.
- Subtype polymorphism means that a variable of type T can access any object of type T or any type derived from T.
- Parametric polymorphism is provided by a subprogram that takes generic parameters that describe the types of the parameters of the subprogram.

Generic Functions

- C++ supports generic functions in the name of template functions.

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

main()
{
   int x = 10, y=20;

   printf(" The max is %d", max(x,y));
}
```

Referencing environments of passed Subprograms

- There are three choices:

1) Shallow binding - The environment of the call statement that enacts the passed subprogram.

2) Deep binding - The environment of the definition of the passed subprogram.

3) Adhoc binding - The environment of the call statement that passed the subprogram as an actual parameter.

Ex:

```
function Sub1()
{
  var x;
  function Sub2()
  {
    print x;
  };
  function Sub3()
  {
    var x;
    x=3;
    Sub4(Sub2);
  };
  function Sub4(Subx)
  {
    var x;
    x=4;
    Subx();
  };
  x=1;
  Sub3();
};
```

Adhoc → binding of local x in Sub3. output is 3

Deep → Sub2's execution, reference to x is bound to local x in Sub1
x=1;

Shallow → reference to x in Sub4 by execution Sub2. x=4.

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?

Functional Side Effects

- A function can be prevented from causing side effects through its parameters.
- Most imperative language function can have either pass-by-value or pass-by-reference, thus allowing functions that cause side effects.
- Pure functional languages such as Haskell, do not have variables, so their functions cannot have side effects.

Types of returned values

- C allows any type to be returned by its functions except arrays and functions. Both of these can be handled by pointer type return values.
- C++ also allows user-defined types or classes to be returned from its functions.

Number of return values

- In most languages, only a single value can be returned from a function.

- Ruby allows the return of more than one value from a method.
- Lua also allows multiple value return

```
return 3, sum, index.
```

- The function call for the above statement would be as follows:

```
a, b, c = fun();
```

Semantics of calls and Returns

- The subprogram call and return operations are together called subprogram linkage.
- The implementation of subprograms must be based on the semantics of the subprogram linkage of the language.
- A subprogram call has numerous actions associated with it.

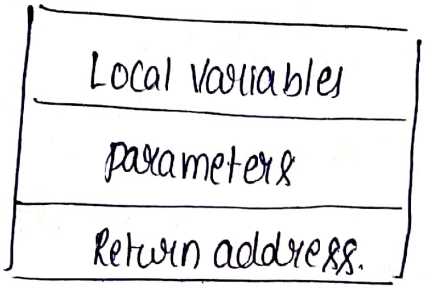
1) The call process must implement the parameter-passing method used.
2) If local variables are not static, the call process must allocate storage for the locals and bind the variables to storage.
3) The execution status is everything needed to resume the execution of the calling program unit.
4) The calling process also must arrange to transfer control to the code of the subprogram and ensure that control can return to the proper place on completion.
5) If the language supports nested subprograms, the call process must create some mechanism to provide access to non local variable that are visible to the called subprogram.

Implementing Simple Subprograms

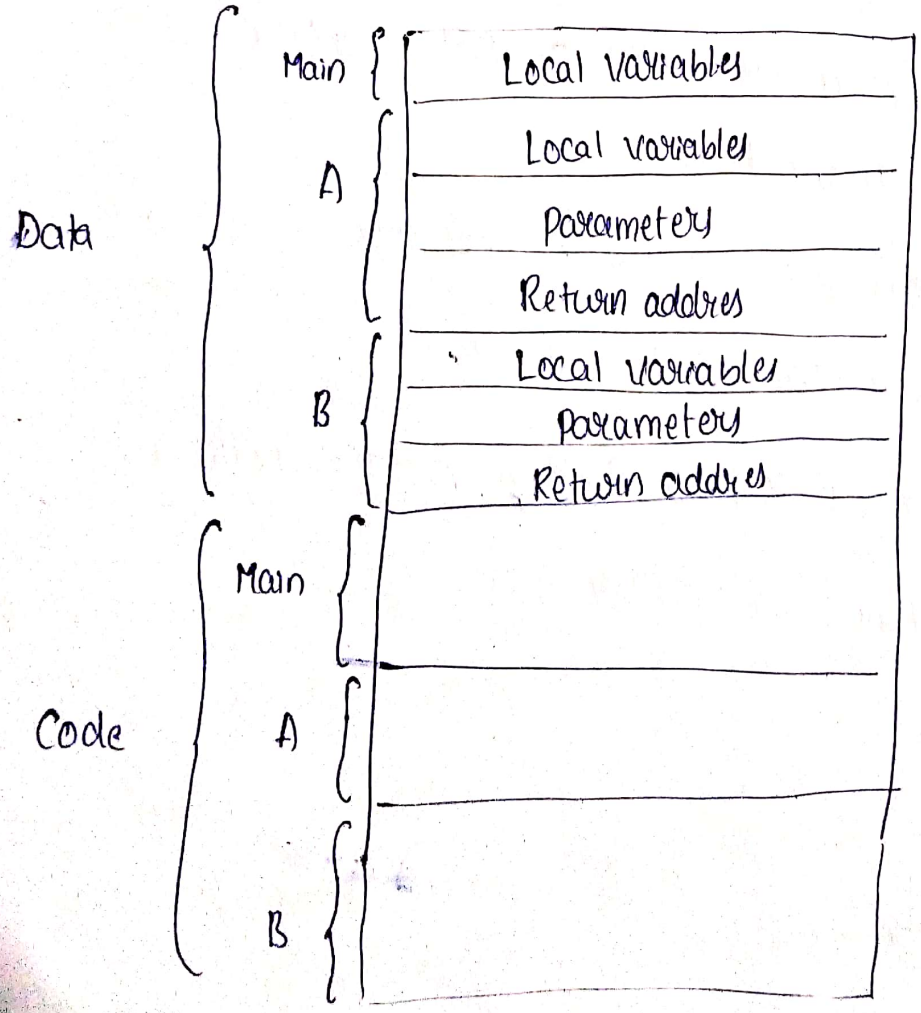
- By "simple" means that subprograms cannot be nested and all local variables are static.
- The semantics of a call to a "simple" subprogram requires the following actions:
    1) Save the execution status of the current program unit.
    2) Compute and pass the parameters.
    3) Pass the return address to the called.
    4) Transfer control to the called.
- The semantics of a return from a simple subprogram requires the following actions:
    1) If there are pass-by-value-result or out-mode parameter, the current values of those parameters are moved or made available to the corresponding actual parameters.
    2) The execution status of the caller is restored.
    3) Control is transferred back to the caller.
- The call and return actions require storage for the following:
    1) Status information about the caller.
    2) Parameters
    3) Return address
    4) Return value for functions.
    5) Temporaries used by the code of the subprograms.

- A simple subprogram consists of two separate parts: the actual code of the subprogram, which is constant and the local variables and data, which can change when the subprogram is executed.

- The format or layout of the noncode part of the subprogram is called an "activation record" because the data it describes is relevant only during the execution of the subprogram.

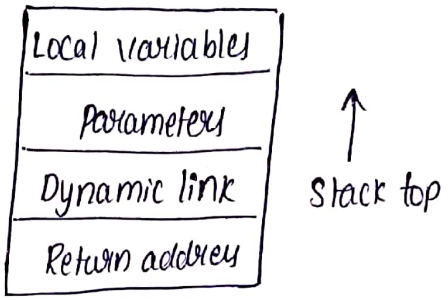


- The code and activation records of a program with two subprams is as follows



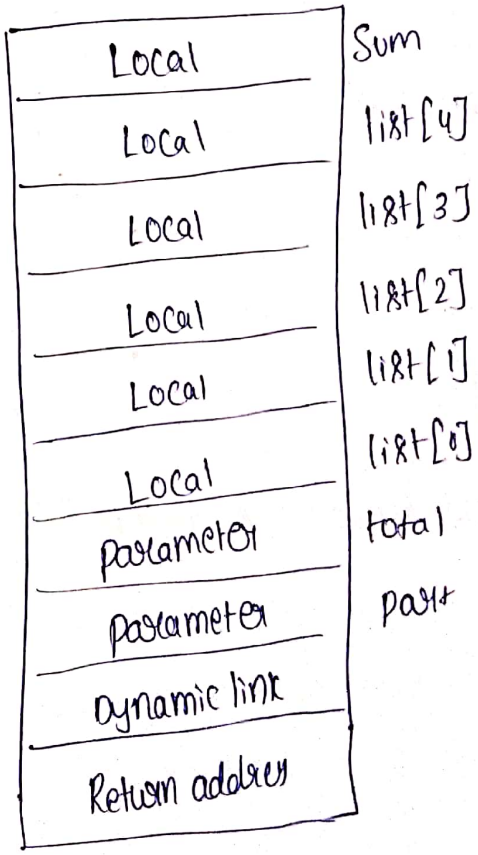
Subprograms with Stack-dynamic local variables

- One of the main advantages of stack-dynamic local variables is the support of recursion.
- To support this a more complex activation record is required.
- In languages with stack-dynamic local variables, activation record instances must be created dynamically. A typical activation record is as follows.



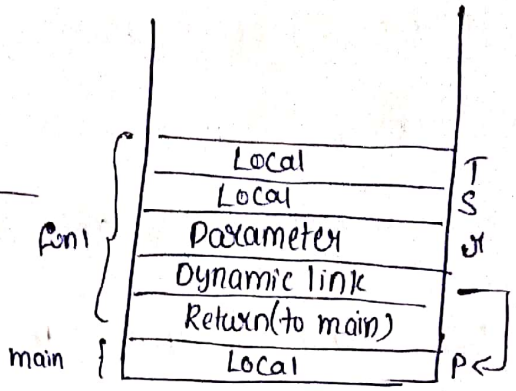
- The dynamic link is a pointer to the base of the activation record instance of the caller.

Ex:-
```
void Sub (float total, int part)
{
  int list[5];
  float sum;
}
```

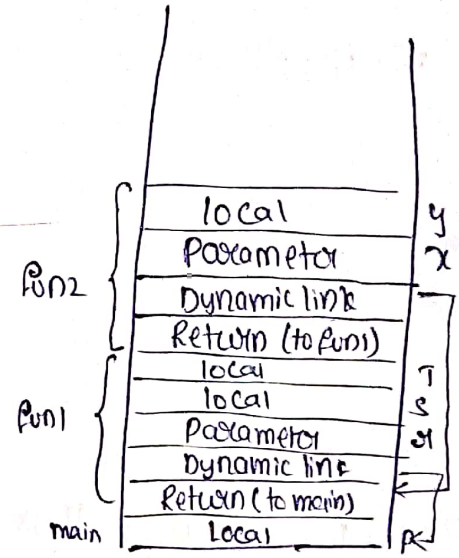


Example without Recursion

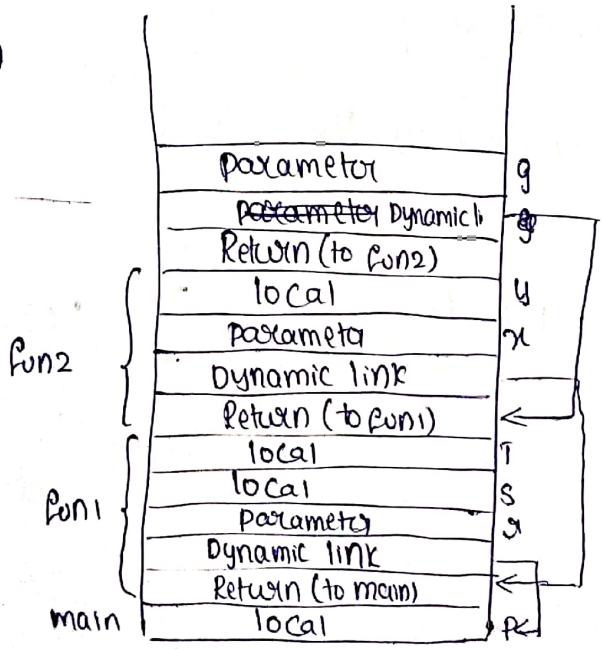
```
Void fun1 (float r)
{
  int s, t;
  ...
  ...
  fun2(s);
}
Void fun2 (int x)
{
  int y;
  ...
  fun3(y)
}
Void fun3(int g)
{
  ...
}
main()
{
float p;
fun1(p);
}
```



Activation record at instance 1



Activation record at instance 2.



Activation Record at instance 3.

→ The sequence of function calls in the program are:

main calls fun1

fun1 calls fun2

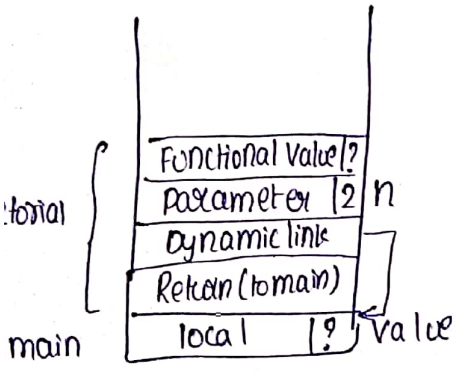
fun2 calls fun3.

Example with Recursion

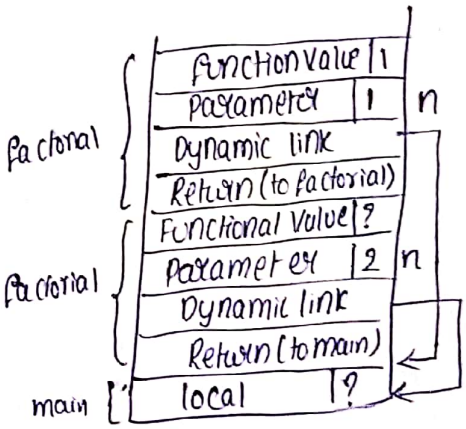
```
int factorial (int n)
{
  if (n<=1)
    return 1;
  else
    return (n*factorial(n-1));
  }
 void main()
 {
  int value;
  Value = factorial(2);
  }
```

The sequence of activation records is as follows.

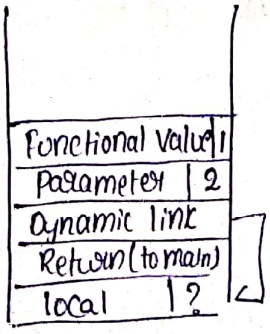
- First call of factorial(2)



- second call factorial(1)



- Second call completed



- First call completed

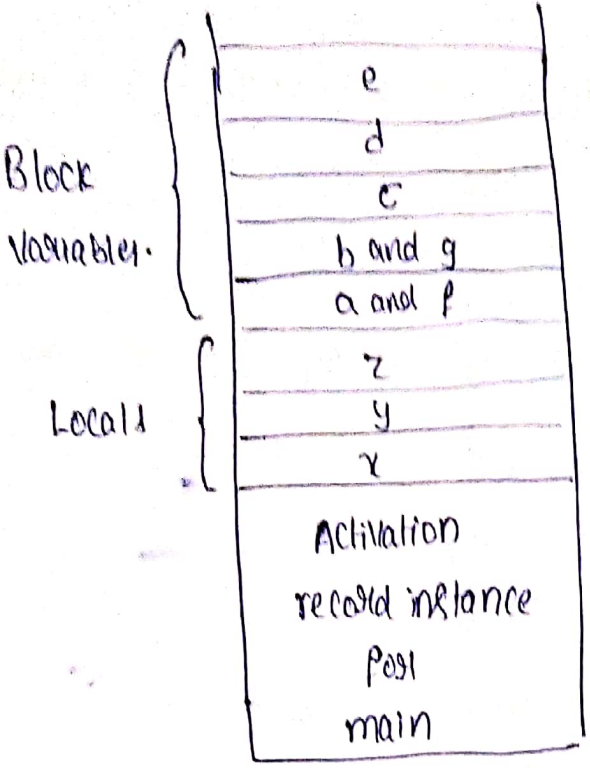
Blocks

- Blocks provide user-defined local scopes.
- Blocks can be implemented by the process described in the last topic.
- Blocks can also be implemented in somewhat simpler and more efficient way.

Ex:-

```
void main()
{
   int x, y, z;

      while ()
      {
         int a, b, c;

            while ()
            {
               int d, e;
            }
      }
   while ()
   {
     int f, g;
   }
}
```

- For the above program, static-memory layout could be used as shown below:



Implementing Dynamic Scoping

- There are two distinct ways in which local variables and non local references can be implemented in a dynamic-scoped language:

1) Deep access
2) Shallow access.

Deep Access

- If local variables are stack dynamic and are part of the activation records in a dynamic-scoped language, references to non local variables can be resolved by searching through the activation record instances of other subprograms that are currently active, beginning with the one most recently activated.

A dynamic chain is followed.

The dynamic chain links together all subprograms activation record instances in the reverse order in which they are activated.

- This method is called deep access, because access for a variable may require searches deep into the stack.

Ex:

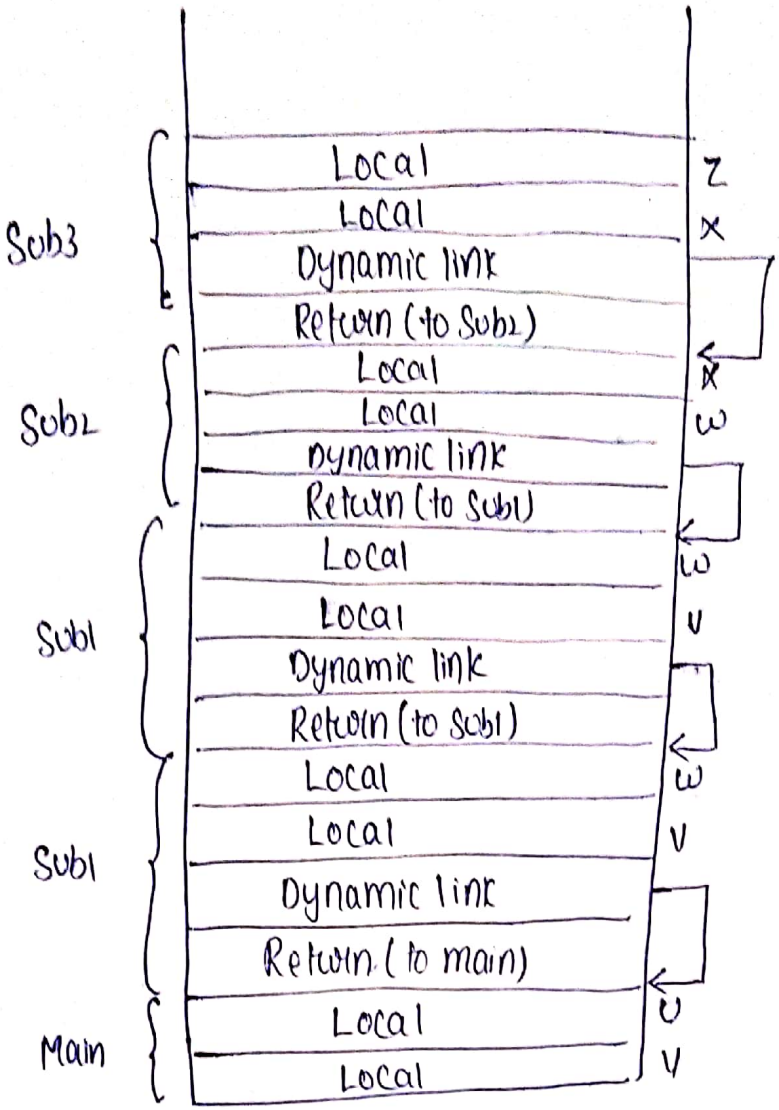
```
Void main()
{
  int v, u;
  Sub1();
}
Void Sub1()
{
  Sub1();
  Sub2();
}
Void Sub2()
{
  int w, x;
  Sub3();
}
Void Sub3()
{
  int x, z;
  x = u+v;
}
```

→ This program has following sequence of function calls:

main calls Sub1
Sub1 calls Sub1
Sub1 calls Sub2
Sub2 calls Sub3.

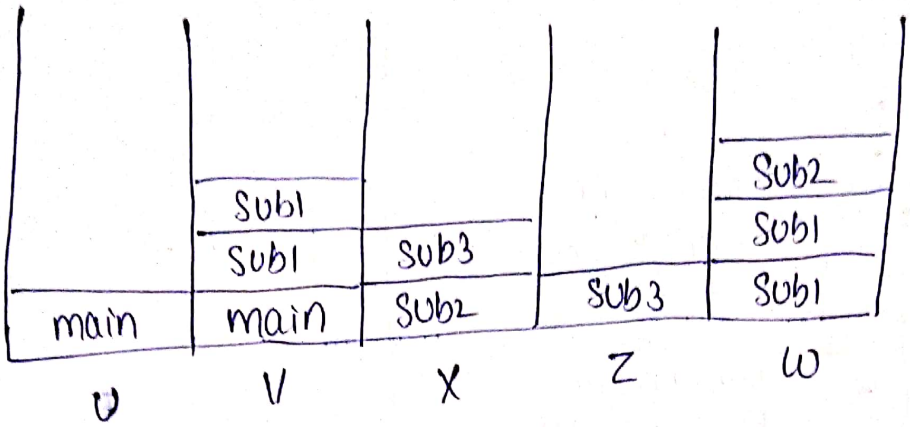
- Consider the references to the variable x, u and v in Sub3.
- The reference to x is in Sub3.
- The reference to u is found by searching all of the activation record instances on the stack, untill main.
- The reference to v is found in Sub1.

- The following figure shows the activation records or stack during the execution of Sub3.



Shallow Access

- In shallow access method, variables declared in subprograms are not stored in the activation records of those subprograms.
- with dynamic scoping there is at least one visible version of a variable of any specific name at a given time.
- So, each variable has a separate stack in a complete program.
- The variable stack for the same above example is as follows:


Sub1
Sub1
Sub3
Sub2
Sub2
Sub1
main
main
Sub2
Sub3
Sub1
U
V
X
Z
W