SUBPROGRAMS (FUNCTIONS, PROCEDURES)

There are two categories of subprograms

i). *Procedures* are collection of statements that define parameterized computations

ii). *Functions* structurally resemble procedures but are semantically modeled on mathematical functions

- They are expected to produce no side effects
- In practice, program functions have side effects

Semantics of subprogram calls might be the most important concept in the study of imperative languages.
Subprograms benefits:

- By reusing code, we save programming time and memory
- Process abstraction: - A subprogram *hides the details* of a computation. This increases program readability and writability

Subprogram fundamentals

All subprograms have a single *entry point*. The calling program unit is *suspended* during execution of a called subprogram. Control always returns to the caller when the subprogram is finished.
A subprogram is *active* when it has begun execution but not yet completed execution.
A ***subprogram definition*** describes the interface to a subprogram and its code (actions of the subprogram abstraction)
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

Subprogram headers

The HEADER is the first part of the definition. It usually contains:

- Some keyword signaling the beginning of a subprogram
- A NAME for the subprogram
- The PARAMETERS of the subprogram

The following are examples of headers in different programming languages

Subroutine Adder(parameters)	FORTRAN
procedure Adder(parameters)	Ada
void Adder(parameters)	C

The *parameter profile* (signature) of a subprogram is the number, order, and types of its parameters
The subprogram *protocol* is the parameter profile plus the return type (for functions only).
A *subprogram declaration* provides the protocol, but not the body, of the subprogram as shown below

```
int foo( int, float );
```
(in C or C++ language)

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

Subprogram parameters

Subprograms need to work on data and most languages allow access to non-local data, BUT
– Accessing non-local data is a *side effect*
– Side effects make it hard to predict results of a subprogram.
– Using non-local data in a subprogram is bad
It is better to use *parameter passing*.
Parameters in the subprogram header are called FORMAL parameters while parameters in the subprogram call are called ACTUAL parameters.

- A ***formal parameter*** is a dummy variable listed in the subprogram header and used in the subprogram
- An ***actual parameter*** represents a value or address used in the subprogram call statement

Parameter specification

i). 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. Notice that most languages use positional parameters

ii). Keyword: - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter. Parameters can appear in any order

E.g. in Python:

```
def f( x=1, y=2 ):
     return x + y + 2
```

f() returns 5
f(4) returns 8
f(4,5) returns 11
f(y=3) returns 6

Ada allows mixing of keyword and positional parameters in a call, but Python does not.
Some languages like C allow subprograms with a variable number of parameters.
Example: printf() allows an infinite number of parameters.

Formal Parameter Default Values
In certain languages (e.g., C++, Ada), formal parameters can have default values (if not actual parameter is passed). In C++, default parameters must appear last because parameters are positionally associated.

Subprogram semantics
How does a subprogram do its work for the caller?
i). It can modify non-locals visible to both the subprogram and the caller (BAD)
ii). It can modify parameters passed to the subprogram
iii). If the subprogram is a function, it can return a value to the caller.
True (mathematical) functions do not have any side effects, but oftentimes the functions we write in imperative programming languages need to have side effects.

Subprogram issues
The designer of a programming languages must make several decisions about subprograms:

- What parameter passing methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- Are locals statically or dynamically allocated?
- Can subprograms be nested?
- If subprograms can be passed as parameters, and subprograms can be nested, what is the referencing environment of the passed subprogram?
- Can subprograms be overloaded or generic?

Local referencing environments
A subprogram can reference its parameter and also its own *local variables*.
Locals can be static or stack dynamic.
Stack dynamic locals:

- Bound to storage when the subprogram begins execution
- Unbound from storage when the subprogram terminates

Advantages:
i). Shared storage with other subprograms
ii). Ability to support recursion
Disadvantages:
i). Cost of allocating storage at subprogram call time
ii). No history can be maintained between calls

Static locals in C
By default, all locals are stack-dynamic in C, but the static keyword gives you a static local as shown below:

```
int adder( int A[], int cA ) {
     static int sum = 0;
     int I;
     for ( I = 0; I < cA; I++ ) sum += A[I];
     return sum;
```

```
}
int main(){
     int A1[] = { 1, 2, 3, 4 };
     int A2[] = { 5, 6, 7, 8 };
     int sumboth;
     adder( A1, 4 );
     sumboth = adder( A2, 4 );
     printf("Sum of lists A1 and A2: %d\n", sumboth );
     return 0;
}
```

Parameter passing methods

Ways in which parameters are transmitted to and/or from called subprograms

i). Pass-by-value
ii). Pass-by-result
iii). Pass-by-value-result
iv). Pass-by-reference
v). Pass-by-name

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

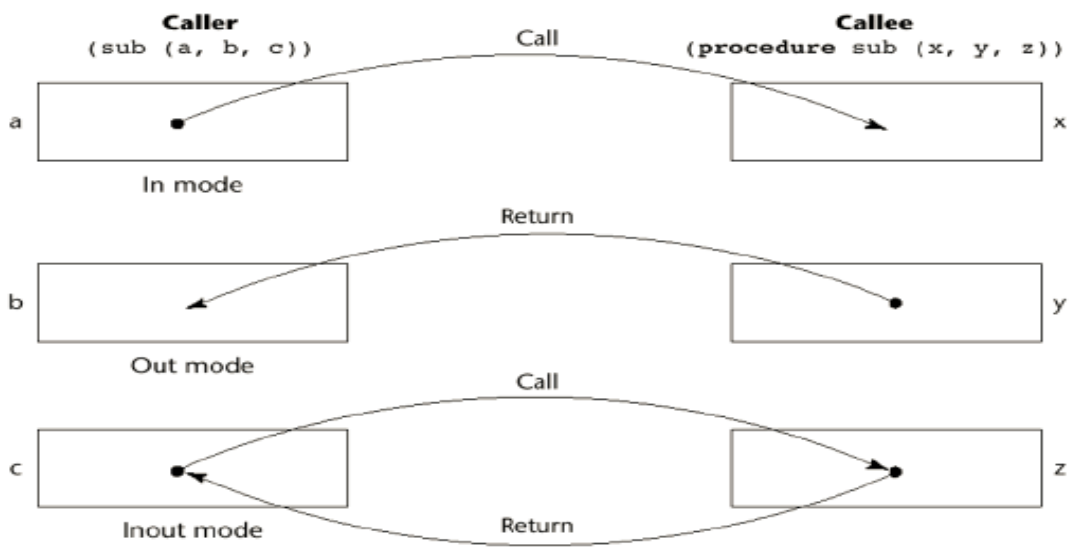
- Normally implemented by copying
- Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
- When copies are used, additional storage is required
- Storage and copy operations can be costly

Models of Parameter Passing

The following are various parameter passing methods:

i). IN-MODE: formal parameters receive data from corresponding actual parameter.
ii). OUT-MODE: final value of formal parameter is transmitted to the corresponding actual parameter
iii). INOUT-MODE: both input and output.

Data transmission can be by *copying* or through an *access path* (usually a pointer).
There are many possible implementations of these modes.



Pass by Value (In-Mode)

For in-mode parameters, we can use the actual parameter's value to initialize the corresponding formal parameter at subprogram call time. Usually, pass by value is accomplished by copying.

Advantage: - Simple

Disadvantages: - Additional storage is needed for the formal parameter, copy overhead might be significant.

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. It requires extra storage location and copy operation
Potential problem: sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1

Pass-by-Value-Result (Inout Mode)
It is a combination of pass-by-value and pass-by-result. Sometimes it is called pass-by-copy. Formal parameters have local storage
The disadvantages are similar to those of pass-by-result and pass-by-value

Pass-by-Reference (Inout Mode)
Instead of copying, an access path is provided. It is also called pass-by-sharing
Passing process is efficient (no copying and no duplicated storage)

Disadvantages:
- Access is indirect, through a pointer.
- Actual parameters can get changed accidentally
- Actual parameter collisions create aliases.

E.g. consider the following code snippet

```
int global;
void fun( int &x ) {
x = global + 1;
}
fun( global );
```

Inside fun(), x and global are aliases.

Pass-by-Name (Inout Mode)
This is inout-mode method. The actual parameter is textually substituted for the corresponding formal parameter in all of its occurrences in the subprogram. 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. It can be dangerous if used improperly
Frequently used in MACROS in C, e.g.

```
#include<stdio.h>
#define Swap(x, y) {int temp; temp = x; x = y; y = temp;}
int main(){
     int a=2, b=4;
     Swap(a,b);
     printf("A=%d\n B=%d", a,b);
     return 0;
}
```

Parameter passing methods in real languages

i). C uses pass by value only and for out or in out parameters, you must use pointers. The const keyword provides one-way access to pointers

ii). C++ adds a reference type for out/inout parameters
- Reference parameters are implicitly dereferenced.
- Semantics are pass-by-reference
- fun(const int &p1, int p2, int &p3) has a one-way pass by reference, a pass by value, and a pass by reference.

iii). Java
- All parameters are passed are passed by value
- Object parameters are passed by reference

iv). Ada
- Three semantics modes of parameter transmission: in, out, in out; in is the default mode e.g. `procedure Adder( A : in out integer; B : in Integer; C : out Float )`
- Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; in out parameters can be referenced and assigned

v). C#
- Default method: pass-by-value
- Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref

vi). PHP: very similar to C#

vii). Perl: all actual parameters are implicitly placed in a predefined array named @_

viii). Python uses pass by value only.

Type Checking

Experts agree that formal and actual parameters should be checked for type agreement. If not, programming errors cause difficult-to-debug problems.

E.g. C did not require type checking of function parameters (or even the number of parameters)

ANSI C does check for type compatibility

Problems still occur with pointers and functions like scanf():

```
double x;
scanf("%f", &x ); /* Should be "%lf" */
```

scanf() allows an arbitrary number of arbitrary pointer parameters i.e. the compiler can't check for type correctness.

Parameter passing implementation

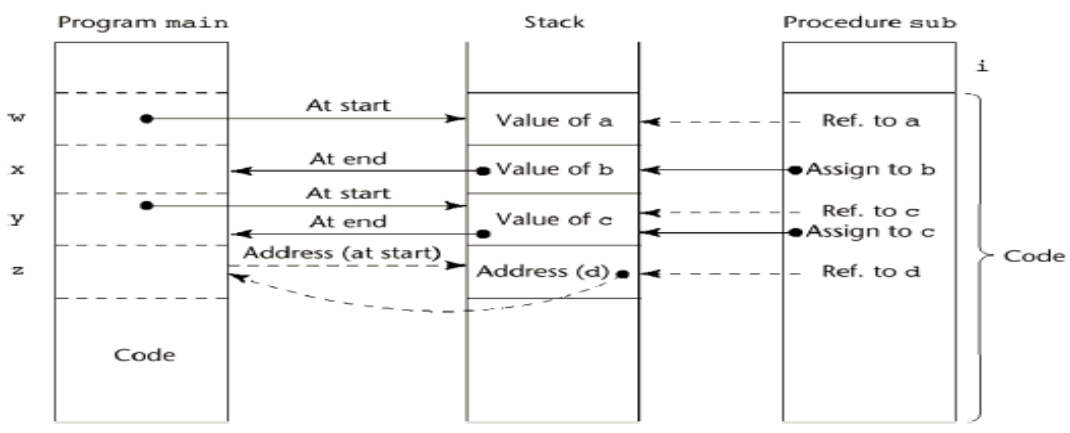
Almost all languages pass parameters using the *runtime stack.*

i). Pass by value parameters:
- Copied into stack locations at call time
- Stack locations serve as the corresponding formal parameters.

ii). Pass by result parameters:
- Allocated on the stack at call time
- Retrieved by the caller at subprogram termination time

iii). Pass by value-result parameters:
- Allocated on the stack
- Initialized from the actual parameter at call time
- Retrieved by the caller at subprogram termination time.

iv). Pass by reference parameters:
- Address of actual parameter is placed on the stack
- Tricky if the actual parameter is an expression or literal
- Inside the subprogram, indirect addressing is used



w: pass by value
x: pass by result
Y: pass by value-result
z: pass by reference.

Handling multidimensional arrays
In C, multidimensional arrays are arrays of arrays.
A 2-dimensional array is accessed as
address(A[i][j]) = address(A[0][0]) + i * ncols + j

Note that ncols is needed, but nrows is not needed.
Therefore, when passing A[][] to a function, ncols must be declared:

```
void fun( int A[][10] ) {
     …
}
int main( void ) {
     int A1[5][10];
     …
     fun( A );
     …
}
```

This prevents us from writing a function that can take arbitrary 2- dimensional arrays as parameters

One way around the problem is to mimic the 2- dimensional array with a 1- dimensional array as shown below:

```
void fun( int *A, int nrows, int ncols ) {
     …
     A[i*ncols+j] = x;
     …
}
int main( void ) {
     int A2[5][12];
     …
     fun( &A[0][0], 5, 12 );
     …
}
```

But the pointer arithmetic is still hard to understand and error prone. The best we can do is write a macro:

```
#define AMAT(r,c) A[r*ncols+c]
void fun( int *A, int nrows, int ncols ) {
     …
     AMAT(i,j) = x;
     …
}
```

Most languages besides C help us a lot more. Python:

```
L = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

L can be queried for its size:

```
def sum(L):
     suml = 0;
     for i in range(len(L)):
          for j in range(len(L[i])):
               suml = suml + L[i][j]
     return suml
```

Subprogram or function names as parameters
Sometimes it is convenient to pass a function to another function.
Example: general-purpose numerical integration, in C:

```
float square( x ) { return x * x; }
```

```
float integrate( float (*f)(float), float x1, float x2, int n ) {
     int i;
     float width = ( x2 - x1 ) / (float)n;
     float x, fofx;
     float sum = 0;
     for ( i = 0; i < n; i++ ) {
          x = x1 + i * width + 0.5 * width;
          fofx = (*f)(x);
          sum += fofx * width;
     }
     return sum;
}
```

Now we can call

```
area = integrate(square, 0, 3, 100 );
```

Note: C and C++ functions cannot be passed as parameters but pointers to functions can be passed; parameters can be type checked

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.

Overloaded subprograms provide ad hoc polymorphism

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

In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)

Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

Generic Subprograms

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

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, e.g.

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

Simple subprogram linkage

Suppose there is no nesting, and all locals are static. This is how FORTRAN began its life.

For the CALL, the runtime system needs to

- Save the execution status of the current program unit
- Pass parameters
- Pass the return address to the callee
- Transfer control to the callee

For the RETURN, the system needs to

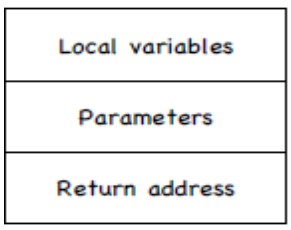
- Move out-mode parameters to corresponding actual parameters
- If subprogram is a function, put the return value in right spot
- Restore execution state
- Transfer control to the caller

Storage Requirements
We need storage for the following:

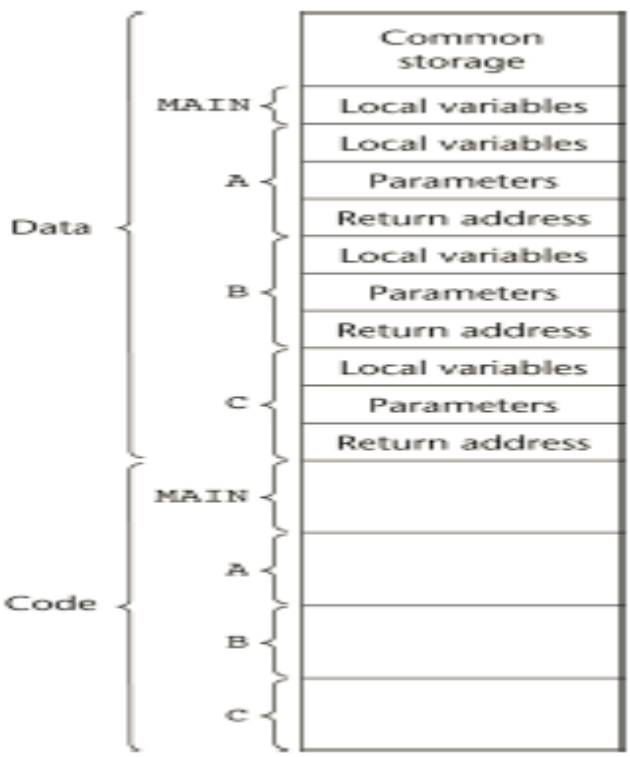
- The execution status info for caller.
- The parameters.
- The return address.
- The functional value (return value)

The data structure associated with a subprogram is called its *activation record*.
An *activation record instance* is a concrete example of an activation record, for a particular subprogram invocation.

Activation records
One way to lay out the activation record for a simple subprogram, with static local variables and no nesting. Static locals means no recursion, so there can only be one activation record instance for each subprogram.



Return address pointer to the next instruction in caller's code segment.
With multiple subprograms, we need one activation record instance for each subprogram, statically allocated. These activation record instances can be laid out linearly in memory, followed by the code for each subprogram as shown below.

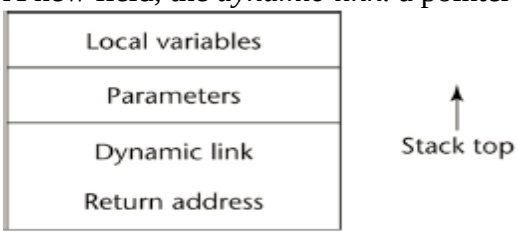


Independent subprogram compilation
Most compilers allow subprograms in different files to be compiled separately.
This means at run time, the subprograms and activation record instances must be loaded into memory from several different places. It is the LINKER's job to determine all of the subprograms used by a program, find the code for those subprograms, and load them into memory for execution.

Stack-dynamic local variables
Previously, we assumed no recursion and that all local variables were statically allocated.
What happens during subprogram linkage if we have stack-dynamic locals?
Subprogram linkage is more complex because

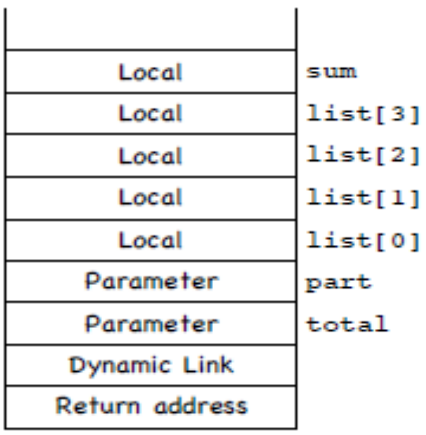
- The compiler has to generate code to cause implicit allocation and deallocation of the local variables.
- Recursion means there could be multiple simultaneous activations of one subprogram.
- Therefore, we need multiple activation record instances for one subprogram.
- Also, we need copies of the formal parameters and the dynamically allocated locals in the activation record.

Activation record with dynamically allocated locals
We add the locals and parameters to the activation record, and now we allocate activation record instances on the *runtime stack*.
A new field, the *dynamic link*: a pointer to the caller's activation record instance.



Example
Consider the following C function and its activation record

```
void sub( float total, int part ) {
     int list[4];
     float sum;
     …
}
```



Creation of Activation Records
Activation records are now created and destroyed dynamically.
When a subprogram is *called*, we allocate an activation record instance (ARI) on the *runtime stack*.

- The runtime stack is a special stack created and managed automatically by the operating system
- The runtime stack is normally supported by hardware

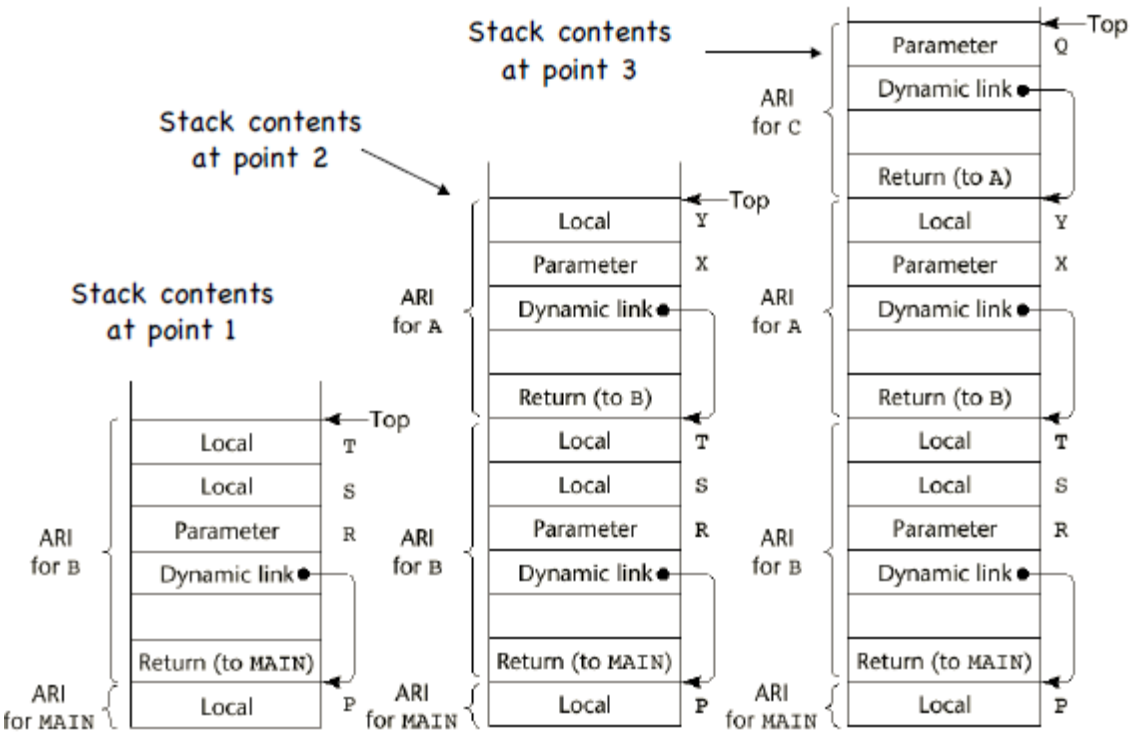
The called subprogram is *active* until it returns.
When a subprogram becomes *inactive*, its activation record instance is destroyed.

```
void fun1 ( int x ) {
     int y;
     … /* Point 2 */
     fun3( y );
     …
}
void fun2 ( float r ) {
     int s, t;
     … /* Point 1 */
     fun1( s );
```

Example

main() calls fun2()
fun2() calls fun1()
fun1() calls fun3()

```
        …
}
void fun3 ( int q ) {
        … /* Point 3 */
}
void main ( ) {
        float p;
        …
        fun2( p );
        …
}
```



Runtime stack

main() has no parameters or dynamic link so its activation record instance (ARI) only contains the local variable p.

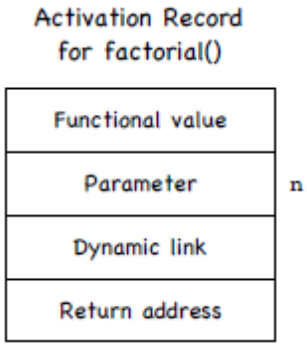
The sequence of dynamic links is called the *dynamic chain* or the *call chain*. It tells you the sequence of function calls that led to the current stack state.

The LOCAL_OFFSET is the offset of a local variable within the activation record. In fun1(), the offset of y is 3 (0th element is the return address, 1st is the dynamic link, 2nd is the parameter x)

Example with recursion

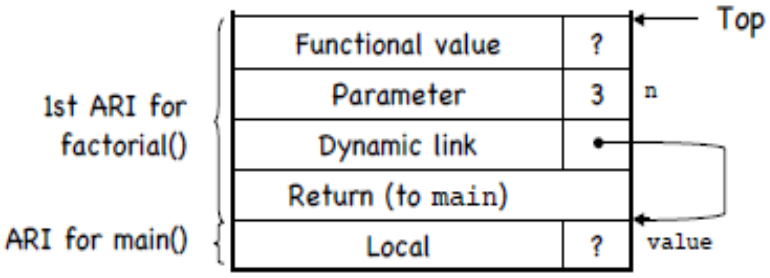
There's nothing special about recursion.

```
int factorial( int n ) {
        /* point 1 */
        if ( n <= 1 ) return 1;
        else return ( n * factorial( n - 1 ));
        /* point 2 */
}
int main( void ) {
        int value;
        value = factorial( 3 );
        /* point 3 */
}
```

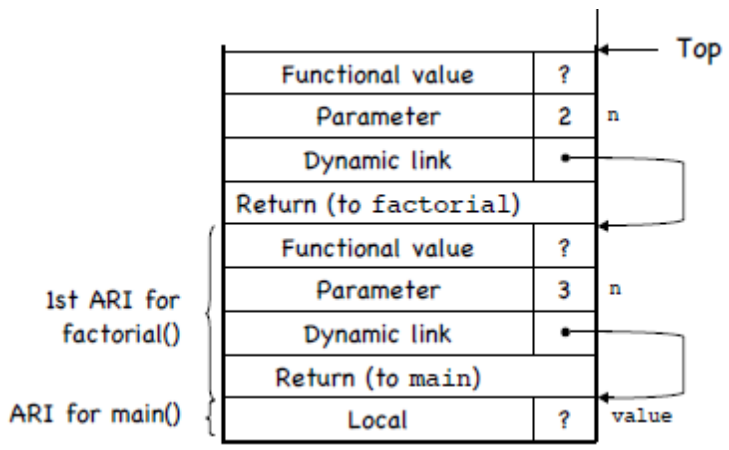


Stack contents for factorial example

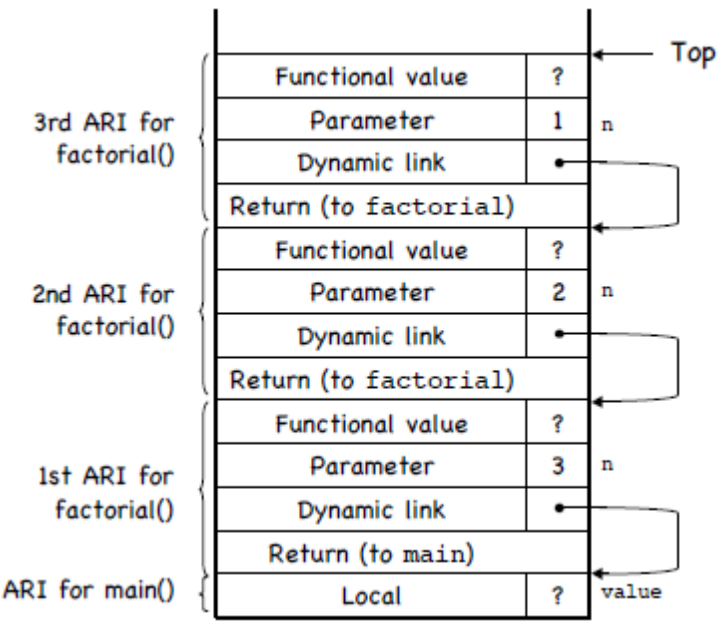
ARI at position 1 in factorial (1st call)



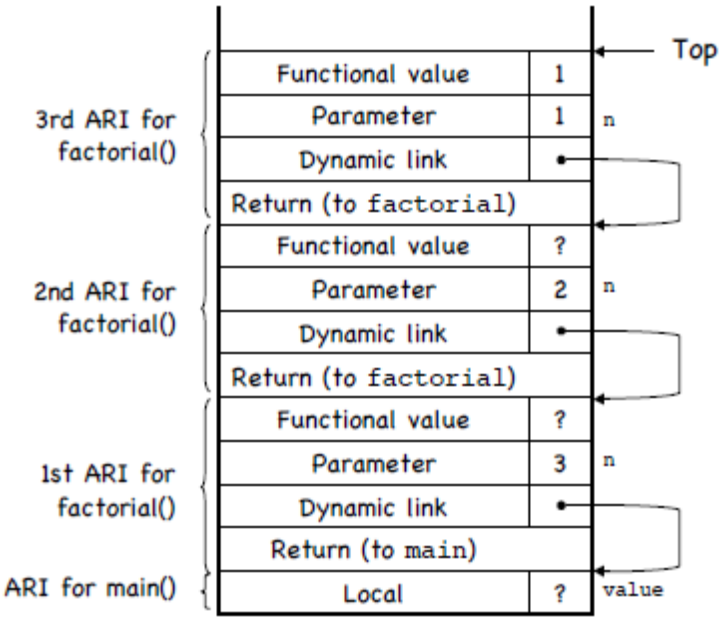
ARI at position 1 in factorial (2nd call)



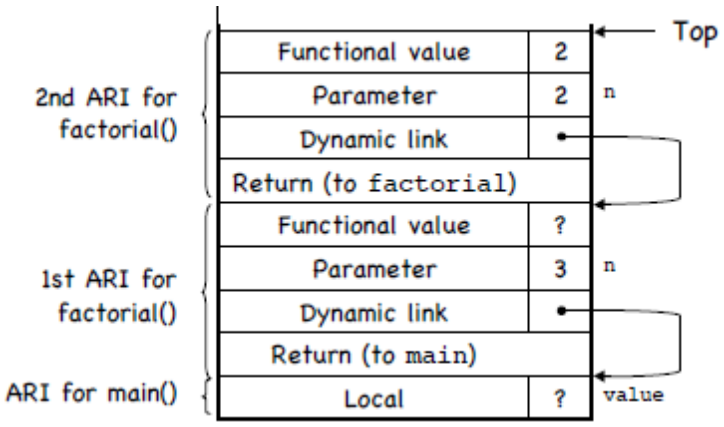
ARI at position 1 in factorial (3rd call)



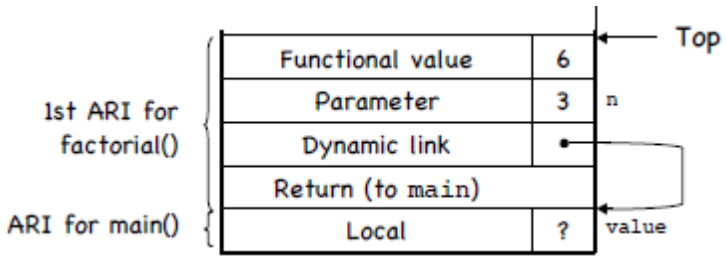
ARI at position 2 in factorial (3rd call completed)



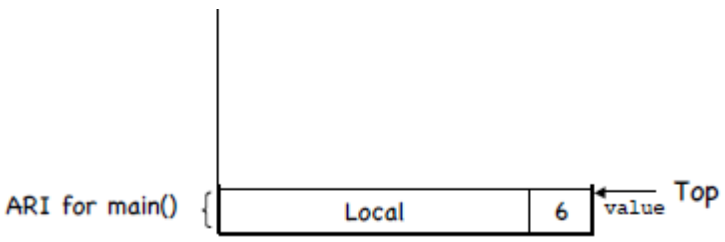
ARI at position 2 in factorial (2nd call completed)



ARI at position 2 in factorial (1st call completed)



ARI at position 3 in main (final results)



Nested subprograms

Most languages besides C (like Pascal, Python, Ada, etc.) allow nested subprograms. This means to find a non-local reference, we have to search several enclosing scopes.

For non-local references, we have to find the activation record instance of the correct enclosing subprogram. Usually we do this with *static chaining*.

For static chaining, we add a *static link* to the activation record. The static link points to the *static* parent's ARI.

The dynamic link still points to the *dynamic* parent's ARI.

Nested subprogram example (Ada)

```
procedure Main_2 is
     X : Integer;
     procedure Bigsub is
             A, B, C : Integer;
             procedure Sub1 is
                     A, D : Integer;
                     begin -- of Sub
                     A := B + C;  <-------------------------  1
                     end;  -- of Sub1
             procedure Sub2(X : Integer) is
                     B, E : Integer;
                     procedure Sub3;
                             C, E : Integer;
                             begin -- of Sub3
                             Sub1;
                             E := B + A;  <-----------------  2
                             end; -- of Sub3
                     begin -- of Sub2
                     Sub3;
                     A := D + E;  <-------------------------  3
                     end; -- of Sub2
             begin -- of Bigsub
             Sub2(7);
             end; -- of Bigsub
     begin -- of Main_2
     Bigsub;
     end;  -- of Main_2
```

Blocks

Some languages, including C, allow user-specified local scopes, called *blocks* as shown below

```
{
     int temp;
     temp = list[upper];
     list[upper] = list[lower];
     list[lower] = temp;
}
```

The scope of temp is limited to the block enclosed by braces { }.

Block implementation

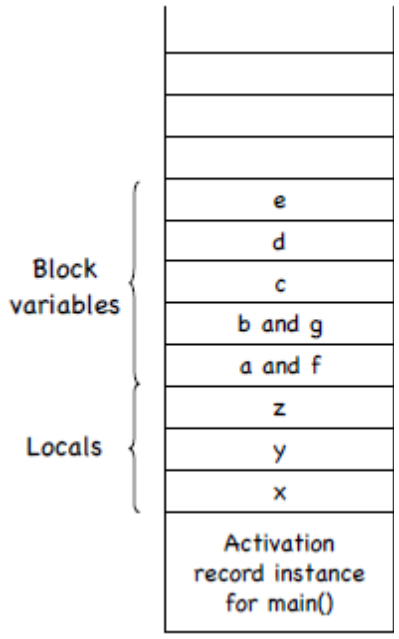
Method 1: treat blocks as parameterless subprograms and use static chaining, similar to nested subprograms.

Method 2: allocate space in the subprogram's activation record for *all possible blocks*.

Example: Consider the code segment below

```
int main( void ) {
     int x, y, z;
     while( … ) {
          int a, b, c;
          …
          while( … ) {
               int d, e;
               …
          }
     }
```

```
        while( … ) {
              int f, g;
              …
        }
        …
}
```


e
d
c
b and g
a and f
z
y
x
Activation record instance for main()
Block variables
Locals