

Chapter 10

Implementing Subprograms

Implementing Subprograms

- ❖ The subprogram *call* and *return* operations are together called *subprogram linkage*
- ❖ Implementation of subprograms must be based on semantics of subprogram linkage
- ❖ Implementation:
 - ⇒ Simple subprograms
 - no recursion, use only static local variables
 - ⇒ Subprograms with stack-dynamic variables
 - ⇒ Nested subprograms

Simple Subprograms

❖ Simple

- ⇒ subprograms are not nested and all local variables are static
- ⇒ Example: early versions of Fortran

❖ Call Semantics require the following actions:

- ⇒ Save execution status of current program unit
- ⇒ Carry out parameter passing process
- ⇒ Pass return address to the callee
- ⇒ Transfer control to the callee

❖ Return Semantics require the following actions:

- ⇒ If pass by value-result or out-mode, move values of those parameters to the corresponding actual parameters
- ⇒ If subprogram is a function, move return value of function to a place accessible to the caller
- ⇒ Restore execution status of caller
- ⇒ Transfer control back to caller

Simple Subprograms

❖ Required Storage:

- ⇒ Status information of the caller
- ⇒ Parameters
- ⇒ return address
- ⇒ functional value (if it is a function)

❖ Subprogram consists of 2 parts:

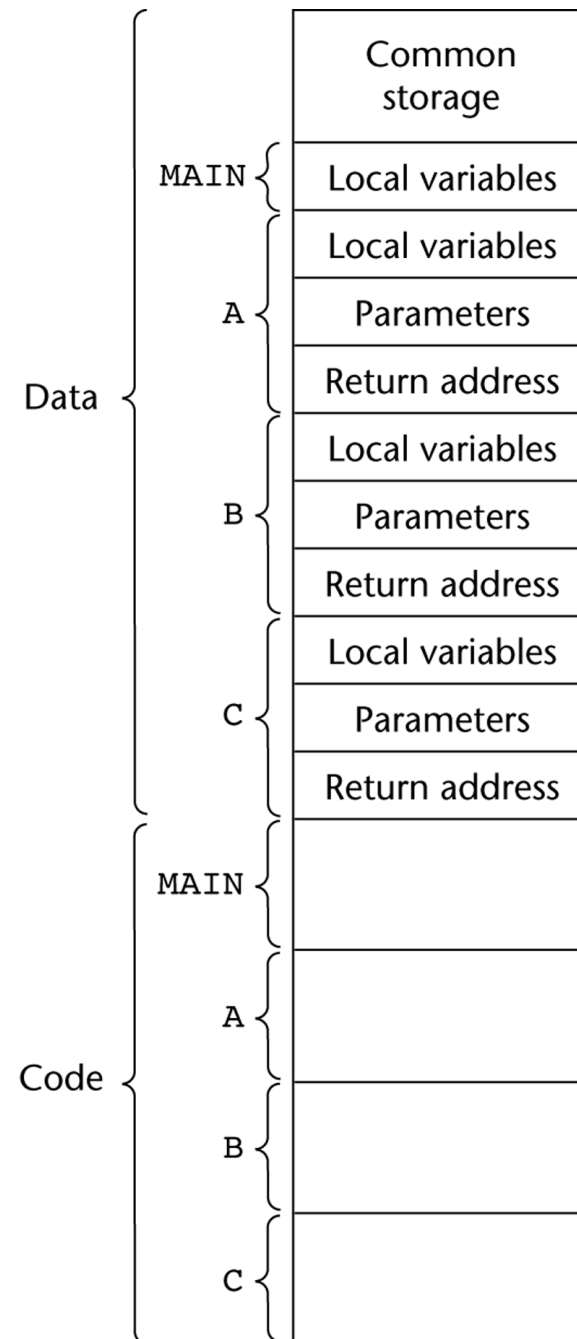
- ⇒ Subprogram code
- ⇒ Subprogram data
 - The format, or layout, of the noncode part of an executing subprogram is called an *activation record*
 - An activation record instance is a concrete example of an activation record (the collection of data for a particular subprogram activation)

Functional value
Local variables
Parameters
Return address

❖ Code and Activation record of a program with simple subprograms

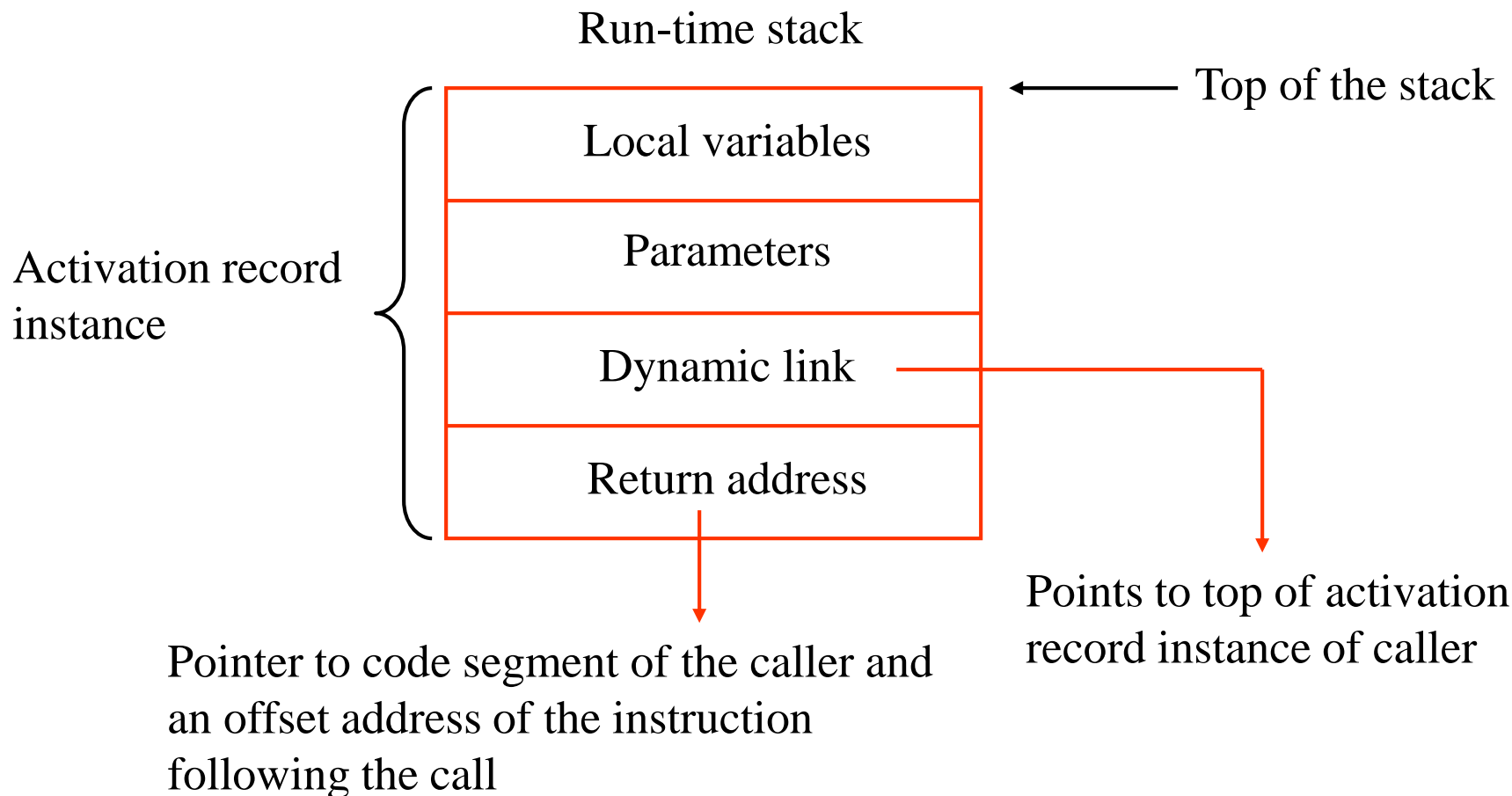
❖ Activation record instance for simple subprograms has fixed size. Therefore, it can be statically allocated

❖ Since simple subprograms do not support recursion, there can be only one active version of a given subprogram



Subprograms with Stack-Dynamic Variables

- ❖ Compiler must generate code to cause implicit allocation and deallocation of local variables



Subprograms with Stack-Dynamic Variables

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

Local variable	sum
Local variable	list[3]
Local variable	list[2]
Local variable	list[1]
Local variable	list[0]
Parameter	part
Parameter	total
Dynamic link	
Return address	

Example: without Recursion

```

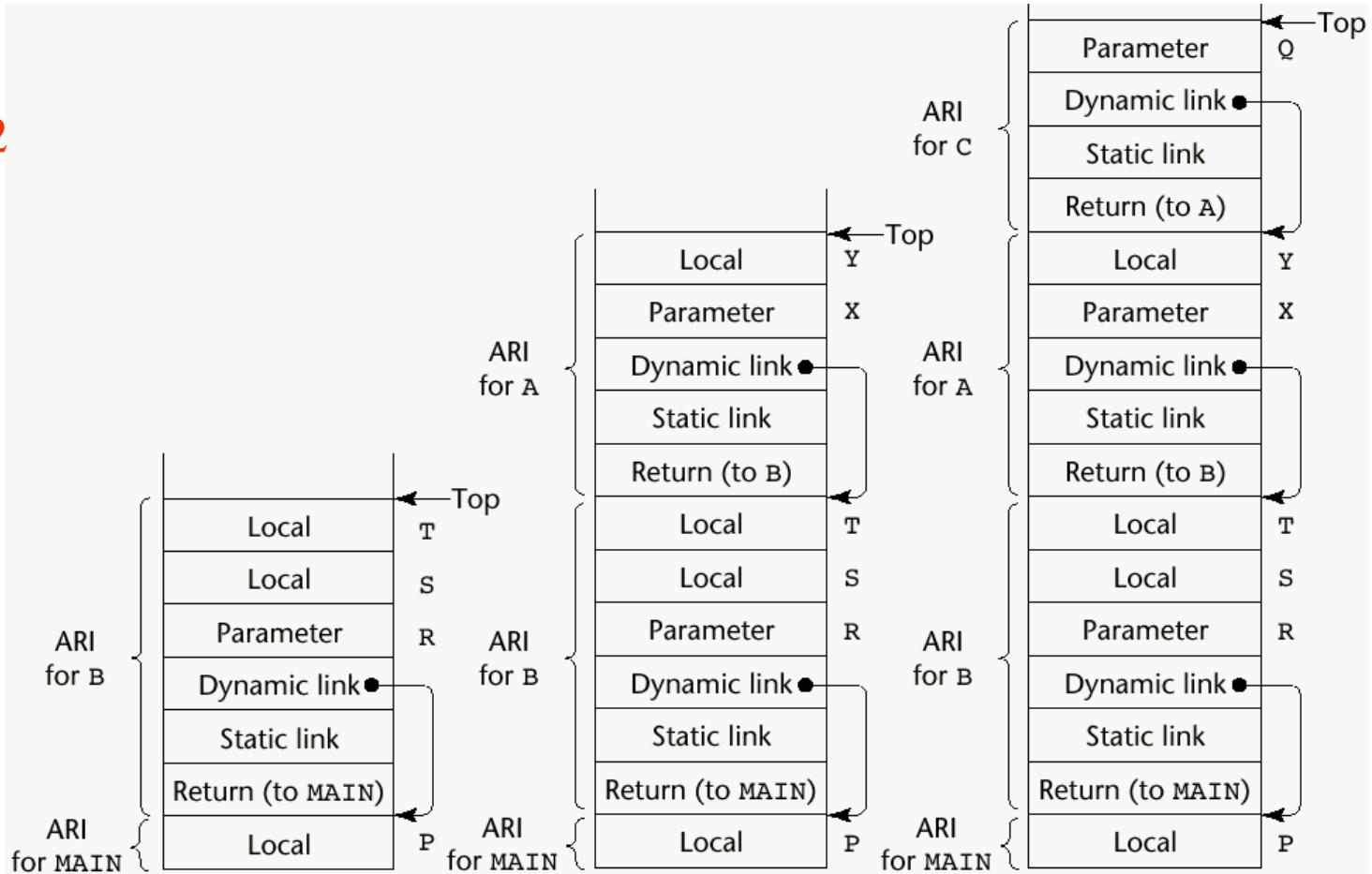
void A(int X) {
    int Y;
    ... ← 2
    C(Y);
}

void B(float R) {
    int S, T;
    ... ← 1
    A(S);
    ...
}

void C(int Q) {
    ... ← 3
}

void main() {
    float P;
    ...
    B(P);
    ...
}

```



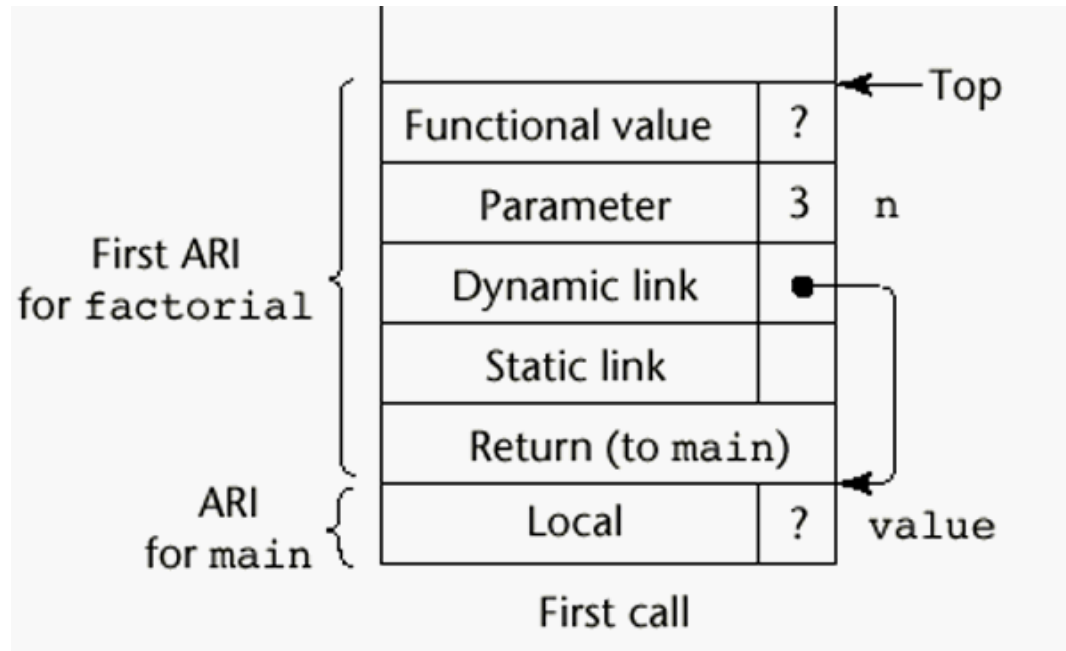
Collection of dynamic links present in the stack at any given time is called the dynamic chain

Subprograms with Stack-Dynamic Variables

- ❖ Recursion adds possibility of multiple simultaneous activations of a subprogram
 - ⇒ Each activation requires its own copy of formal parameters and dynamically allocated local variables, along with return address

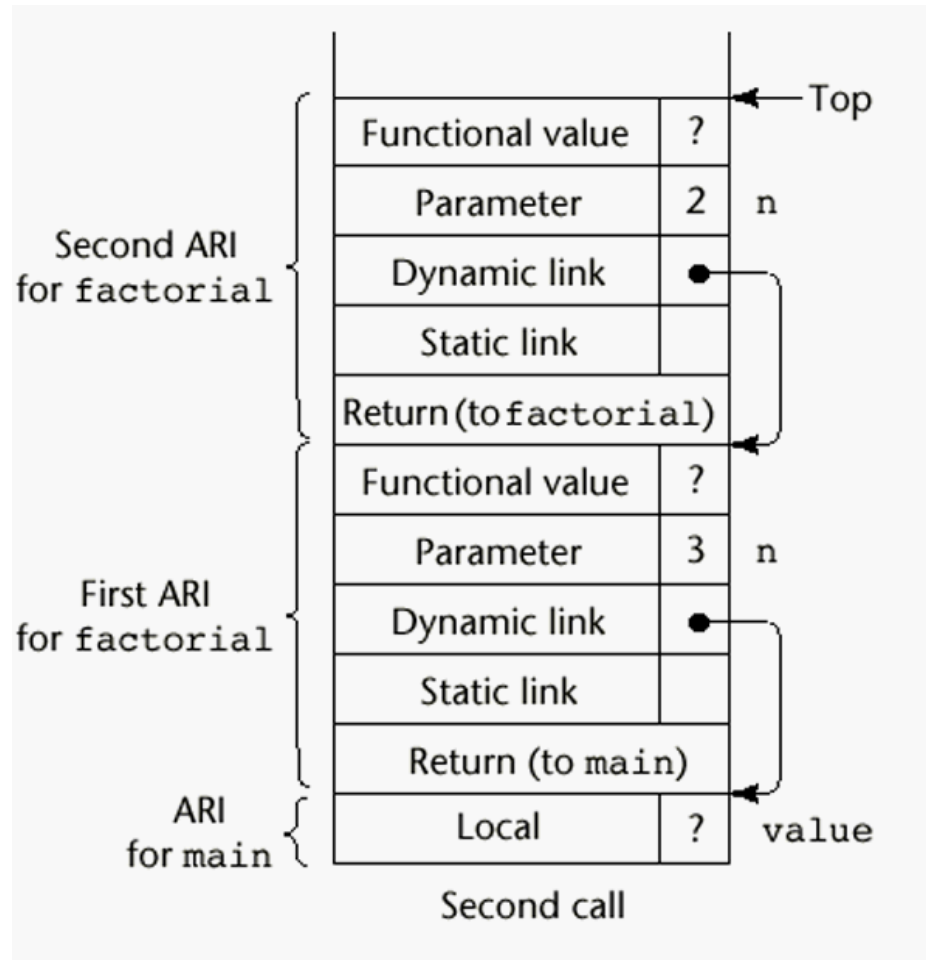
Subprograms with Recursion

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



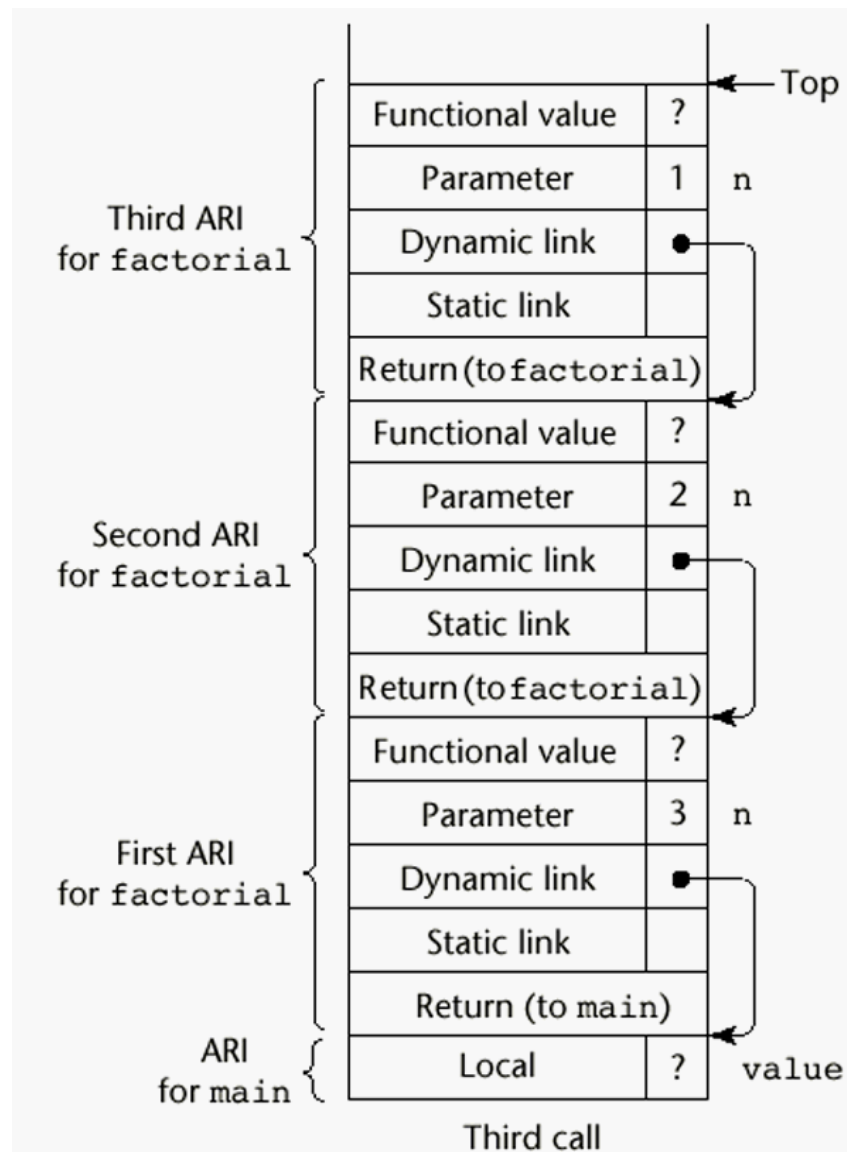
Subprograms with Recursion

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



Subprograms with Recursion

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

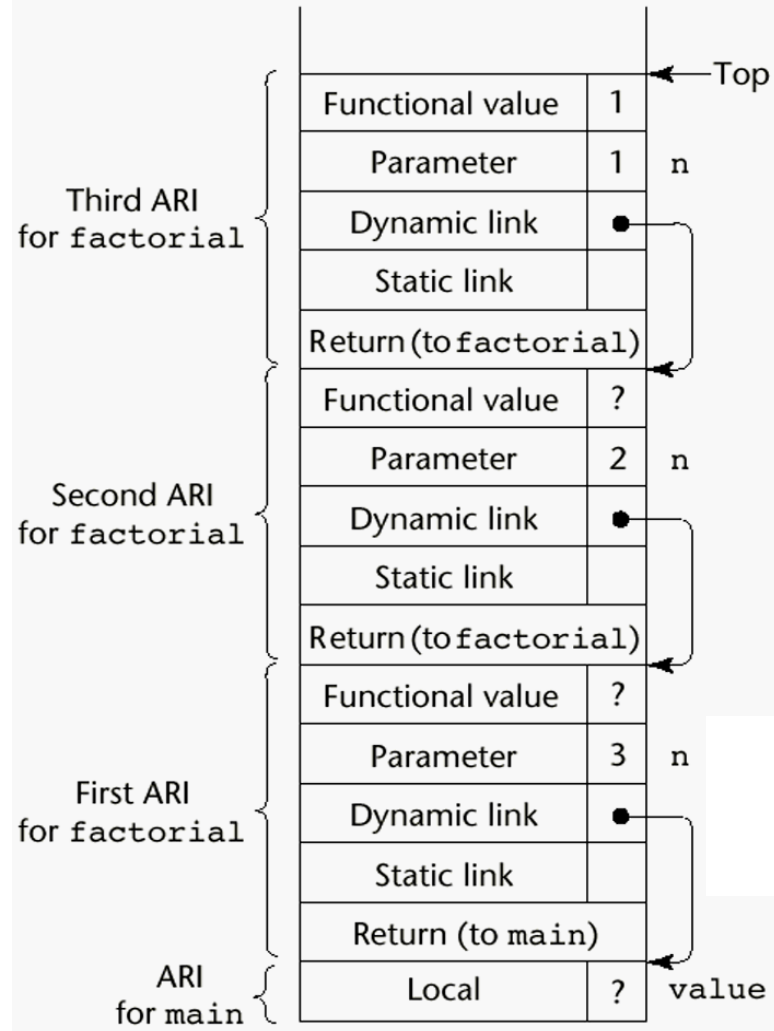


Subprograms with Recursion

```

int factorial (int n) {
    ... ← 1
    if (n <= 1)
        return 1;
    else
        return n*factorial(n-1);
    ... ← 2
}

void main() {
    int value;
    value = factorial(3);
    ... ← 3
}
    
```

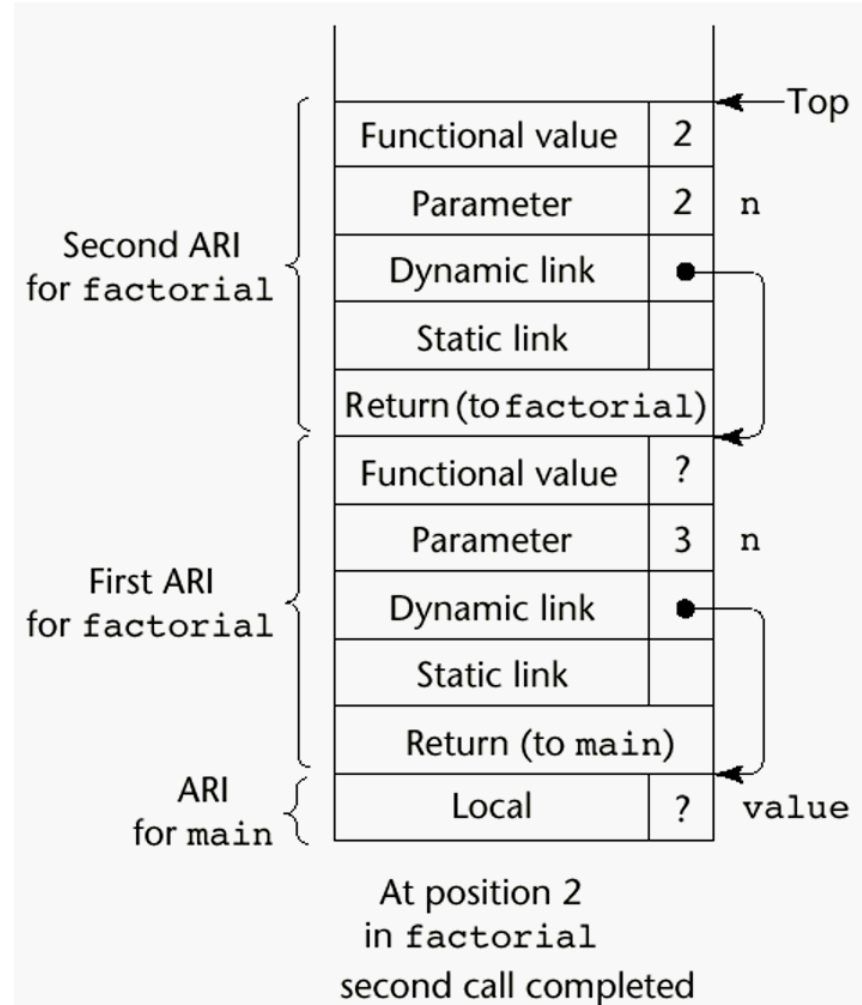


At position 2
in factorial
third call completed

Subprograms with Recursion

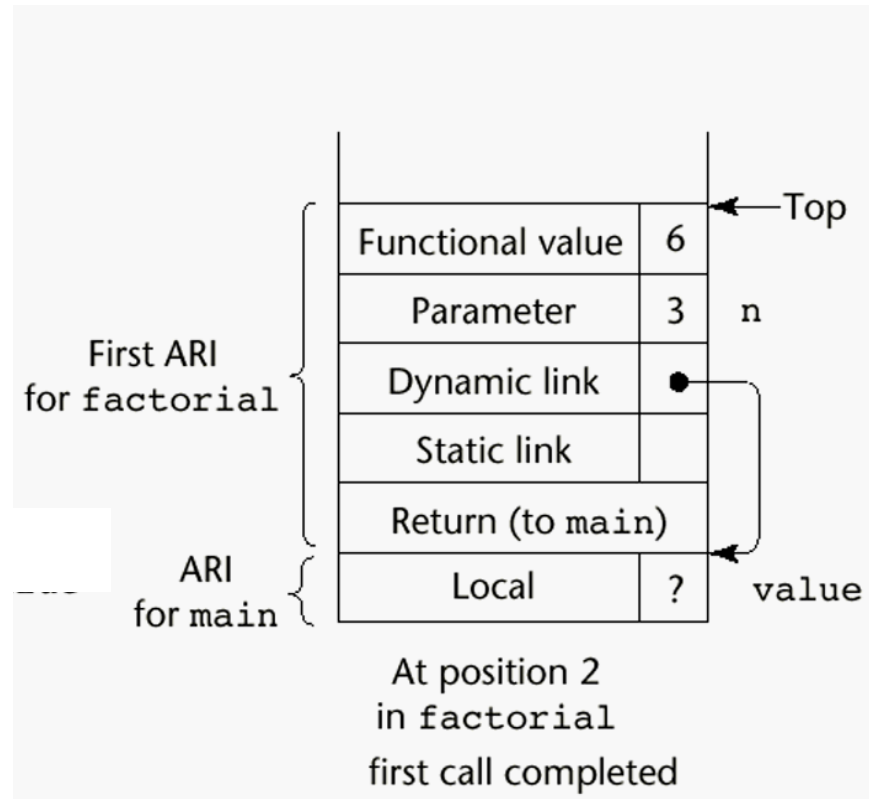
```

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



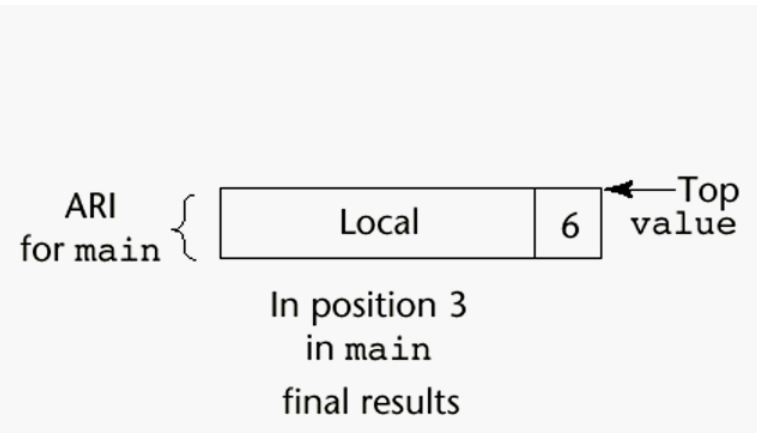
Subprograms with Recursion

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



Subprograms with Recursion

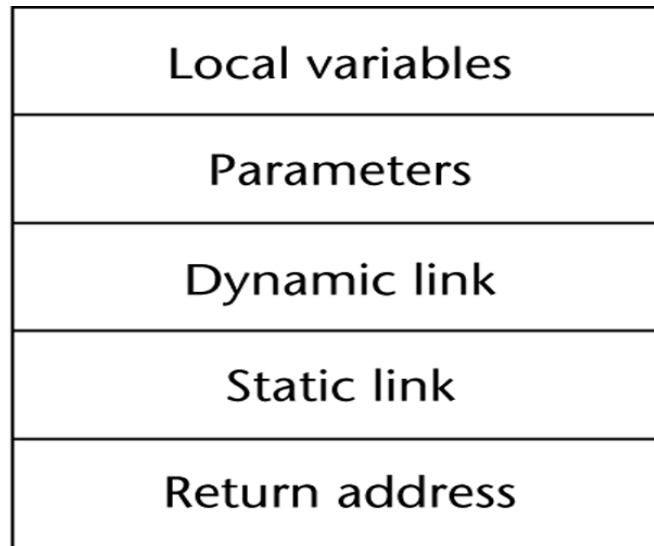
```
int factorial (int n) {  
    ... ← 1  
    if (n <= 1)  
        return 1;  
    else  
        return n*factorial(n-1);  
    ... ← 2  
}  
  
void main() {  
    int value;  
    value = factorial(3);  
    ... ← 3  
}
```



Nested Subprograms

❖ Support for static scoping

- ⇒ Implemented using static link (also called static scope pointer), which points to the bottom of the activation record instance of its static parent



Nested Subprograms

❖ Static chain

⇒ links all static ancestors of executing subprogram

❖ Static_depth

⇒ an integer associated with static scope that indicates how deeply it is nested in outermost scope

❖ Chain offset

⇒ Difference between static_depth of procedure containing reference to variable x and static_depth of procedure containing declaration of x

procedure A is

 procedure B is

 procedure C is

 ...

 end; -- of C

 ...

end; -- of B

...

end; -- of A

❖ Static_depths of A, B, and C are 0, 1, and 2, respectively

❖ If procedure C references a variable declared in A, the chain_offset of that reference is 2

Nested Subprograms

```
program MAIN_2;  
  var X : integer;  
  procedure BIGSUB;  
    var A, B, C : integer;  
    procedure SUB1;  
      var A, D : integer;  
      begin { SUB1 }  
        A := B + C; <-----1  
      end; { SUB1 }  
    procedure SUB2(X : integer);  
      var B, E : integer;  
      procedure SUB3;  
        var C, E : integer;  
        begin { SUB3 }  
          SUB1;  
          E := B + A; <-----2  
        end; { SUB3 }  
      begin { SUB2 }  
        SUB3;  
        A := D + E; <-----3  
      end; { SUB2 }  
    begin { BIGSUB }  
      SUB2(7);  
    end; { BIGSUB }  
begin  
BIGSUB;  
end. { MAIN_2 }
```

Calling sequence:

Main_2 calls Bigsub

Bigsub calls Sub2

Sub2 calls Sub3

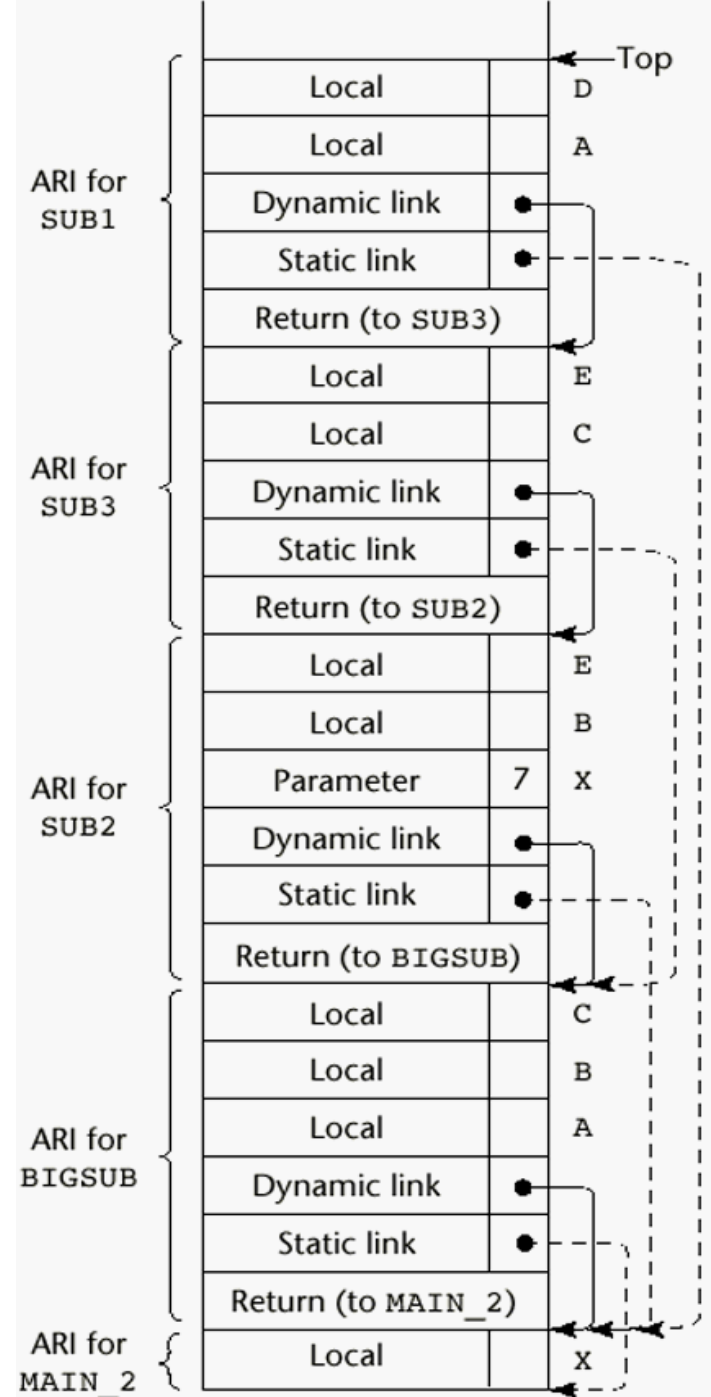
Sub3 calls Sub1

Example

```

program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C; <-----1
      end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
          SUB1;
          E := B + A; <-----2
        end; { SUB3 }
      begin { SUB2 }
        SUB3;
        A := D + E; <-----3
      end; { SUB2 }
    begin { BIGSUB }
      SUB2(7);
    end; { BIGSUB }
  begin
    BIGSUB;
  end. { MAIN_2 }

```



Nested Subprograms

❖ At position 1 in SUB1:

⇒ A - (0, 3) =====> (chain_offset, local_offset)

⇒ B - (1, 4)

⇒ C - (1, 5)

❖ At position 2 in SUB3:

⇒ E - (0, 4)

⇒ B - (1, 4)

⇒ A - (2, 3)

❖ At position 3 in SUB2:

⇒ A - (1, 3)

⇒ D - an error

⇒ E - (0, 5)

Nested Subprograms

❖ Drawbacks

- ⇒ A nonlocal reference is slow if the number of scopes between the reference and the declaration of the referenced variable is large
- ⇒ Time-critical code is difficult, because the costs of nonlocal references are hard to estimate

❖ Displays

- ⇒ Alternative to static chains
- ⇒ Store **static links in a single array called display**, instead of storing in the activation records
- ⇒ Accesses to nonlocals require exactly two steps for every access, regardless of the number of scope levels
 - Link to correct activation record is found using a statically computed value called the `display_offset`
 - Compute `local_offset` within activation record instance

Blocks

❖ User-specified local scope for variables

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

❖ Blocks can be implemented using static chain

❖ **Blocks are treated as parameterless subprograms** that are always called from same place in the program

⇒ Every block has an activation record

⇒ An instance is created every time a block is executed

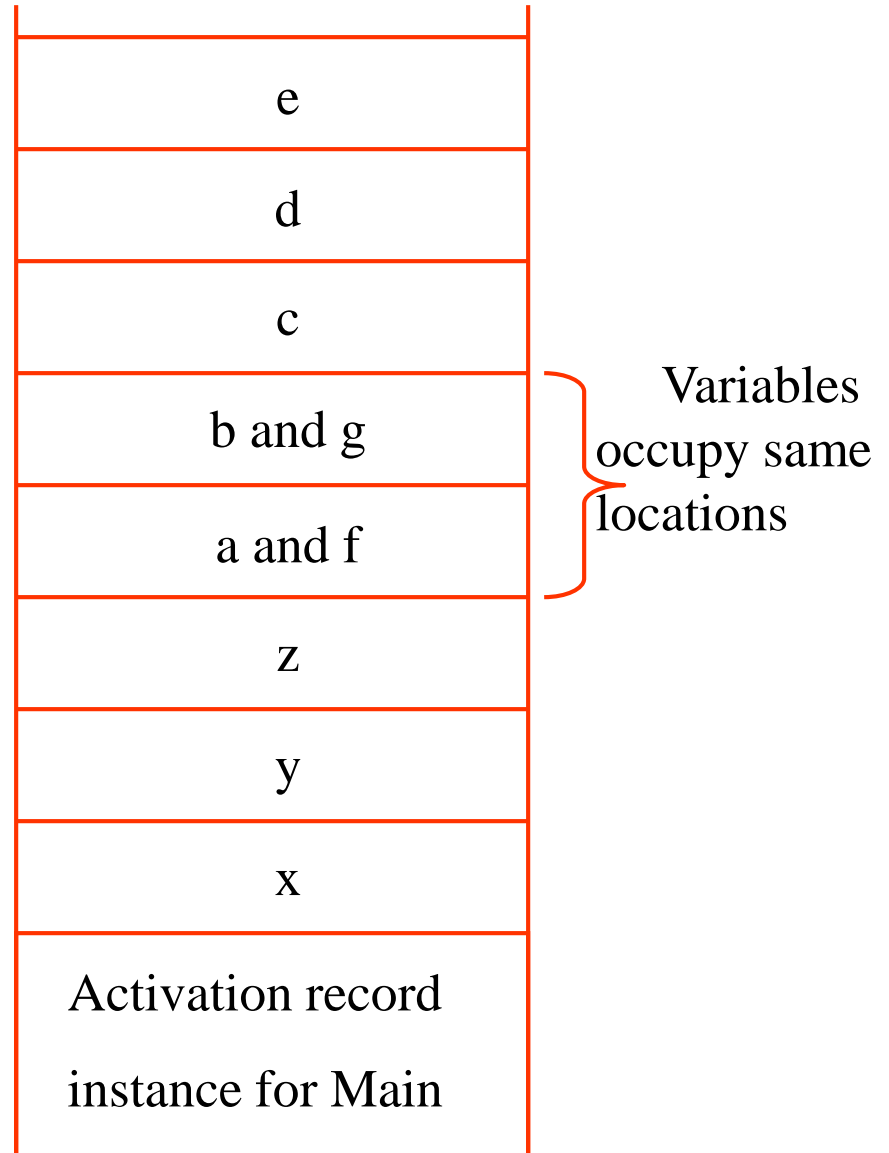
❖ Alternative implementation

⇒ Amount of space can be allocated statically

⇒ Offsets of all block variables can be statically computed, so block variables can be addressed exactly as if they were local variables

Blocks

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



Subprogram Implementation

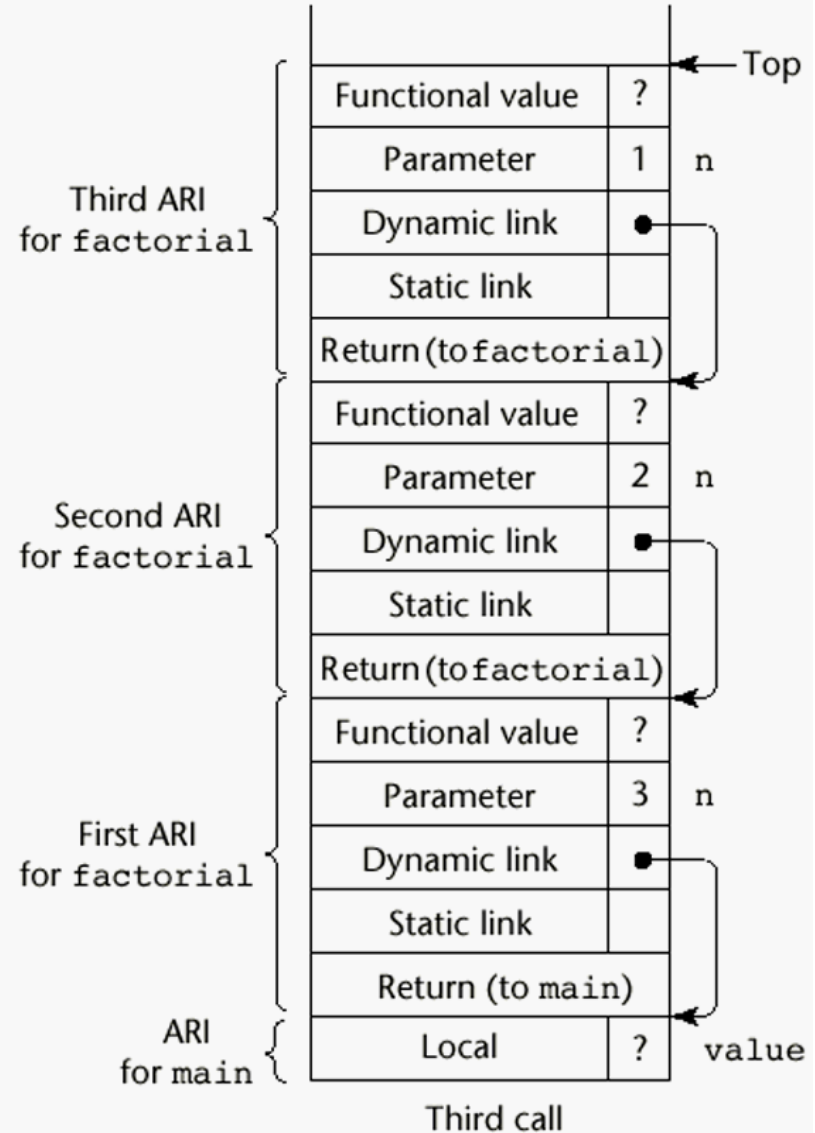
⇒ Activation record on the stack

- Parameters
- Return address
- Local variables
- Static link
- Dynamic link

```

int factorial (int n) {
    ...
    if (n <= 1)
        return 1;
    else
        return n*factorial(n-1);
    ...
}

void main() {
    int value;
    value = factorial(3);
    ...
}
    
```



Subprogram Implementation

- ❖ Bad design of subprogram implementation may result in network security problems
- ❖ Buffer overflow attack
 - ⇒ A type of vulnerability used by hackers to compromise the integrity of a system
 - ⇒ Problem is due to
 - Lack of safety feature in language design
 - bad coding by programmers

Buffer overflow attack

- ❖ The effectiveness of the buffer overflow attack has been common knowledge in software circles since the 1980's
- ❖ The Internet Worm used it in November 1988 to gain unauthorized access to many networks and systems nationwide
- ❖ Still used today by hacking tools to gain “root” access to otherwise protected computers
- ❖ The fix is a very simple change in the way we write array accesses; unfortunately, once code that has this vulnerability is deployed in the field, it is nearly impossible to stop a buffer overflow attack

Overview of Buffer Overflow Attacks

- ❖ The buffer overflow attack exploits a common problem in many programs.
- ❖ In several high-level programming languages such as C, “boundary checking”, i.e. checking to see if the length of a variable you are copying is what you were expecting, is not done.

```
void main(){  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

```
void myFunction(char *str)  
{  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```

Overview of Buffer Overflow Attacks

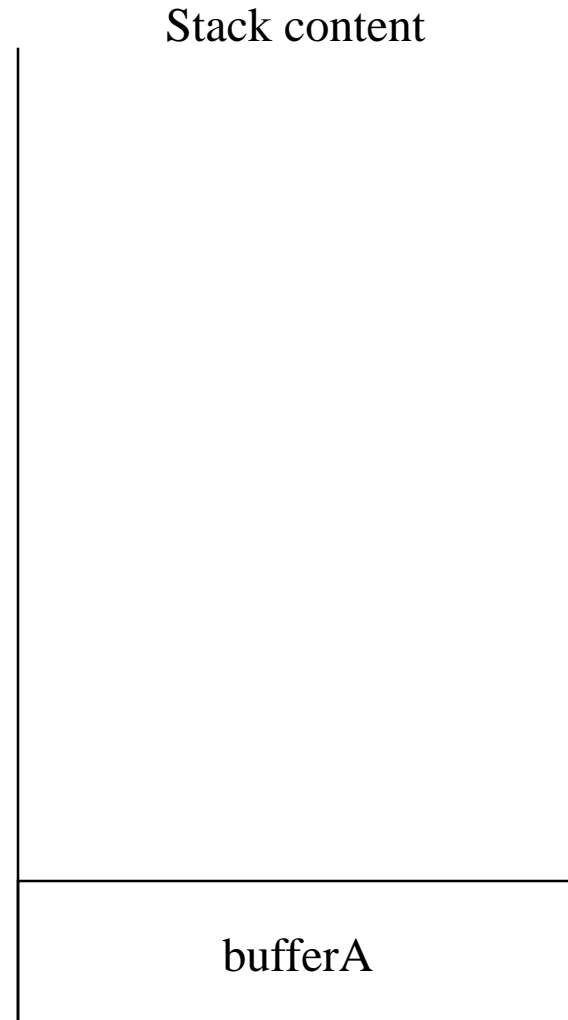
```
void main(){  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

```
void myFunction(char *str)  
{  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```

- main() passes a 256 byte array to myFunction(), and myFunction() copies it into a 16 byte array!
- Since there is no check on whether bufferB is big enough, the extra data overwrites other unknown space in memory.
- This vulnerability is the basis of buffer overflow attacks
- How is it used to harm a system?
 - It modifies the system stack

Overview of Buffer Overflow Attacks

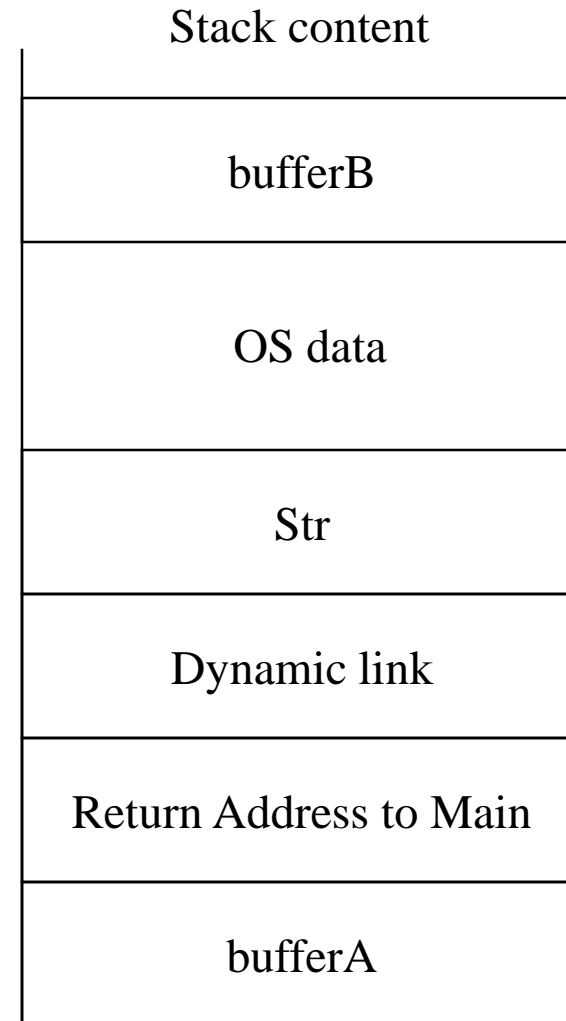
```
void main(){  
    char bufferA[256];  
    myFunction(bufferA);  
}
```



Overview of Buffer Overflow Attacks

```
void main(){  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

```
void myFunction(char *str)  
{  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```

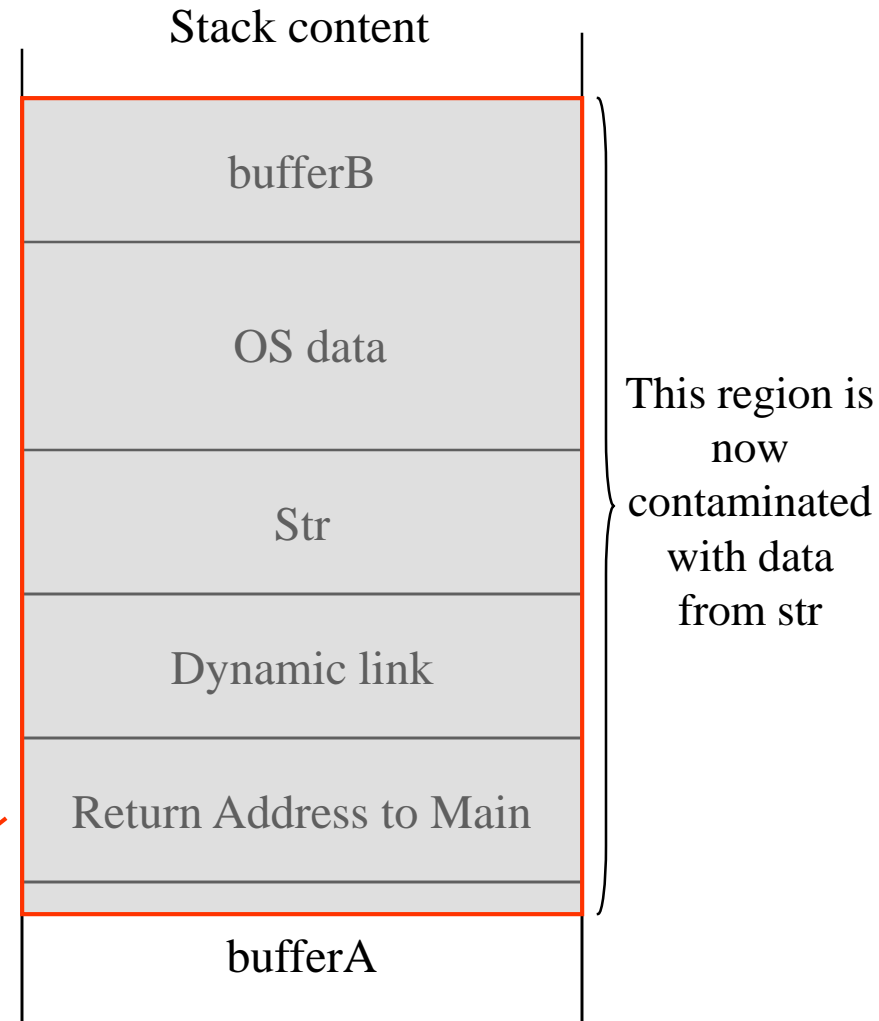


Overview of Buffer Overflow Attacks

```
void main(){  
    char bufferA[256];  
    myFunction(bufferA);  
}
```

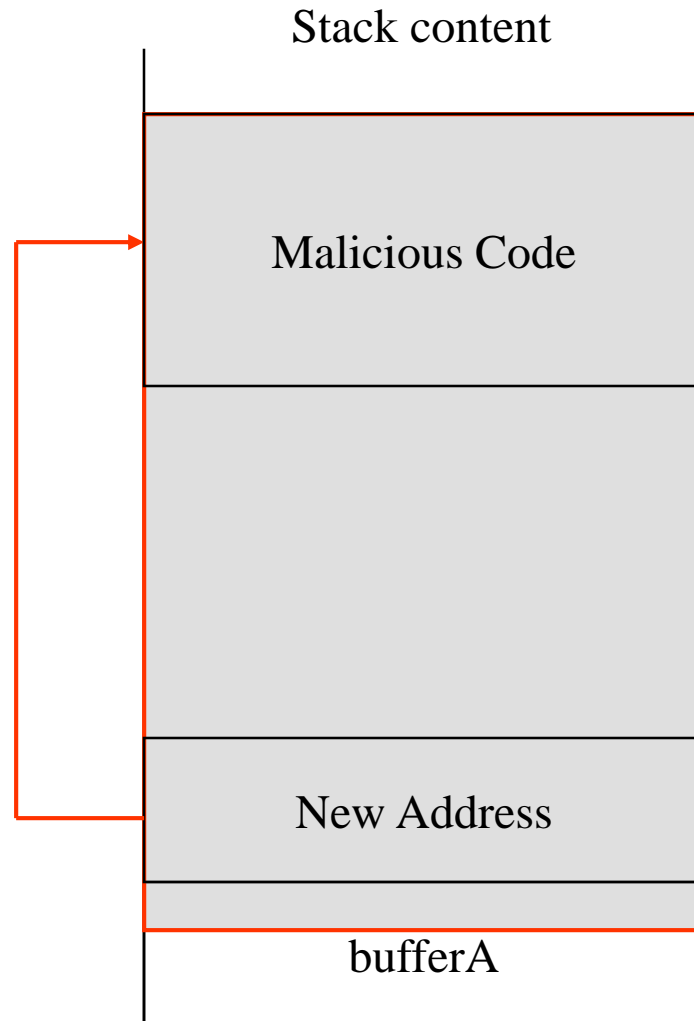
```
void myFunction(char *str)  
{  
    char bufferB[16];  
    strcpy(bufferB, str);  
}
```

May overwrite the
return address!!



Overview of Buffer Overflow Attacks

- If the content of str is carefully selected, we can point the return address to a piece of code we have written
- When the system returns from the function call, it will begin executing the malicious code



A Possible Solution

```
void main(){
    char bufferA[256];
    myFunction(bufferA, 256);
}
```

```
void myFunction(char *str, int len)
{
    char bufferB[16];
    if (len <= 16)
        strcpy(bufferB, str);
}
```

Buffer Overflow Attack

CERT/CC Vulnerability Notes - Microsoft Internet Explorer provided by Comcast

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites Media

Address <http://www.kb.cert.org/vuls> Go Links

[Vulnerability Notes](#)

[Vulnerability Notes Help](#)

[Information](#)

View Notes By

[Name](#)

[ID Number](#)

[CVE Name](#)

[Date Public](#)

[Date Updated](#)

[Severity Metric](#)

Other CERT/CC Knowledgebases

[AIRCERT](#)

Other Documents

[Advisories](#)

of these vulnerabilities are available from this web page in a searchable database format, and are published as "CERT Vulnerability Notes". The notes are very similar to advisories, but they may have less complete information. In particular, solutions may not be available for all the vulnerabilities in this database.

[CERT Advisories](#) (which are sent to the CERT/CC mailing list and are published on the web) are limited to vulnerabilities that meet a certain [severity threshold](#). Unfortunately, it is difficult to establish a metric ranking the severity of a vulnerability that is appropriate for all systems. For example, a severe vulnerability in a rarely used application might not qualify for publication as a CERT Advisory. However, this information might be very important to a system administrator who runs the rarely used application. In such cases, CERT Vulnerability Notes provide a mechanism to publish information about these less severe vulnerabilities.

You can [search](#) or browse Vulnerability Notes by several key fields, including [name](#), [vulnerability ID number](#), [CVE name](#), [date updated](#), [date public](#), or [metric](#). You can also customize database queries to obtain specific information, such as the [ten most recently updated vulnerabilities](#) or the [twenty vulnerabilities with the largest metric score](#).

The CERT/CC Vulnerability Note database is fully "Common Vulnerabilities and Exposures" (CVE) compatible. You can search for vulnerabilities by CVE name or browse our list of vulnerabilities in CVE name order. This information allows you to easily cross reference vulnerabilities described here with those at other sites. More information about CVE can be found at their [web site](#).

Recent Vulnerability Notes

- [VU#602204](#) OpenSSH PAM challenge authentication failure
- [VU#209807](#) Portable OpenSSH server PAM conversion stack corruption
- [VU#333628](#) OpenSSH contains buffer management errors
- [VU#792284](#) WS_FTP Server vulnerable to buffer overflow when supplied overly long "APPE" command
- [VU#219140](#) WS_FTP Server vulnerable to buffer overflow when supplied overly long "STAT" command
- [VU#41870](#) Sun Solstice AdminSuite ships with insecure default configuration
- [VU#108964](#) Sendmail contains buffer overflow in ruleset parsing
- [VU#784980](#) Sendmail prescan() buffer overflow vulnerability
- [VU#258564](#) Linux NFS utils package "rpc.mountd" contains off-by-one

Internet