The Procedure Abstraction
Part III: Establishing Addressability

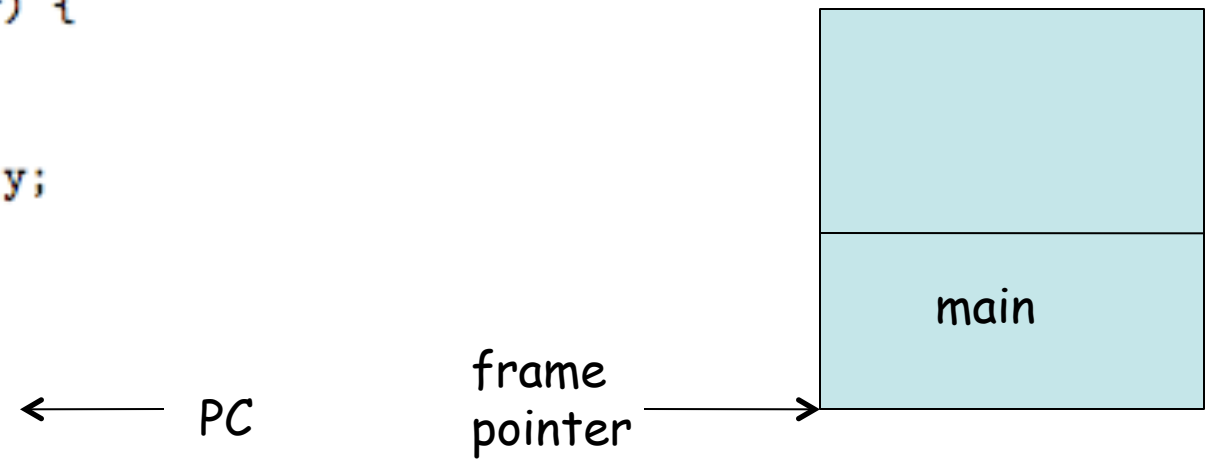
Simple C program


```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```

A. Source code

Simple C program with runtime stack

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```



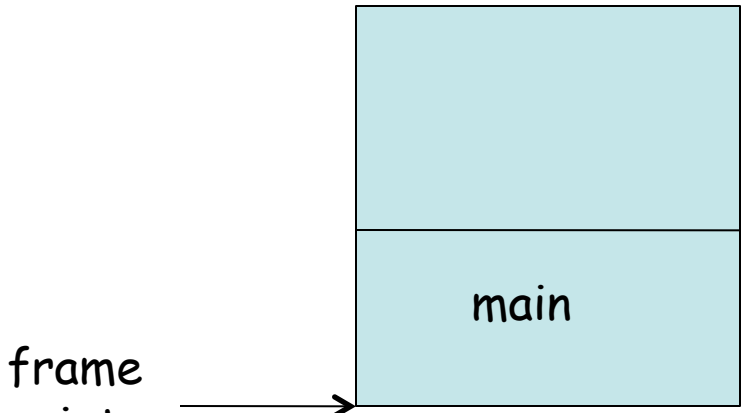
A. Source code

B. Runtime Stack

Simple C program with runtime stack

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```

PC



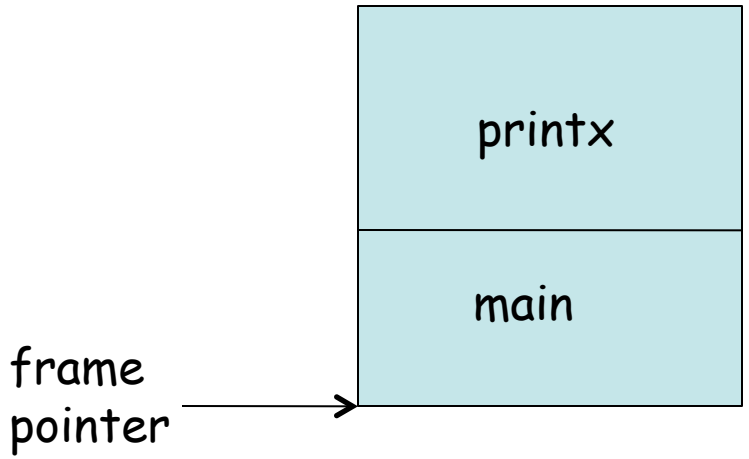
A. Source code

B. Runtime Stack

Simple C program with runtime stack

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();     <-- PC
  foo(z);
}
```

A. Source code

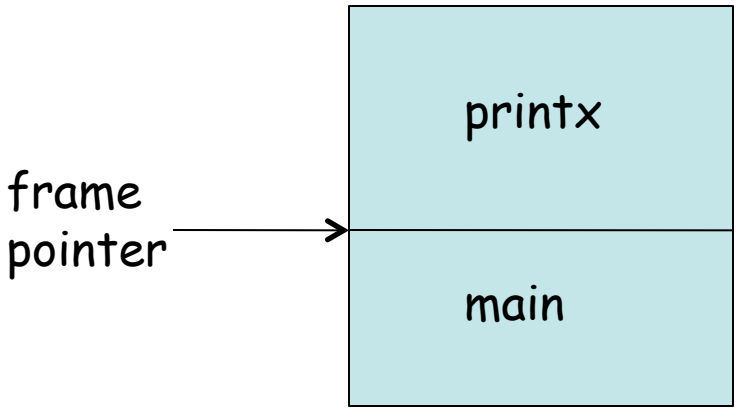


B. Runtime Stack

Simple C program with runtime stack

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```

PC



A. Source code

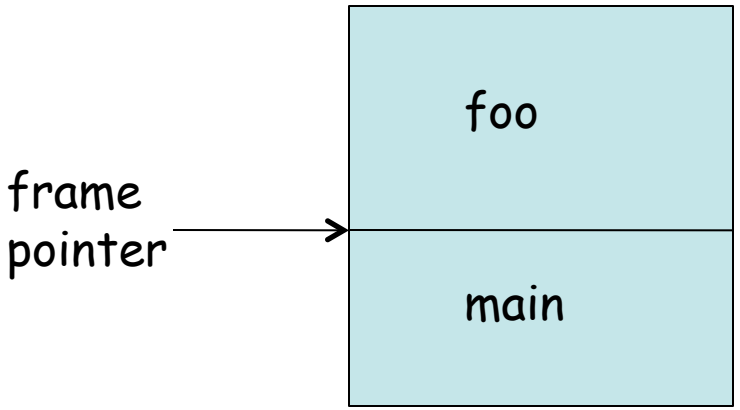
B. Runtime Stack

Simple C program with runtime stack

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```

PC

A. Source code

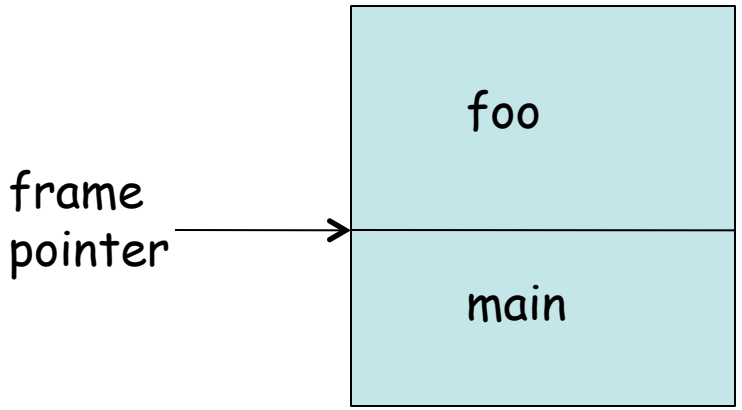


B. Runtime Stack

Simple C program with runtime stack

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```

PC



A. Source code

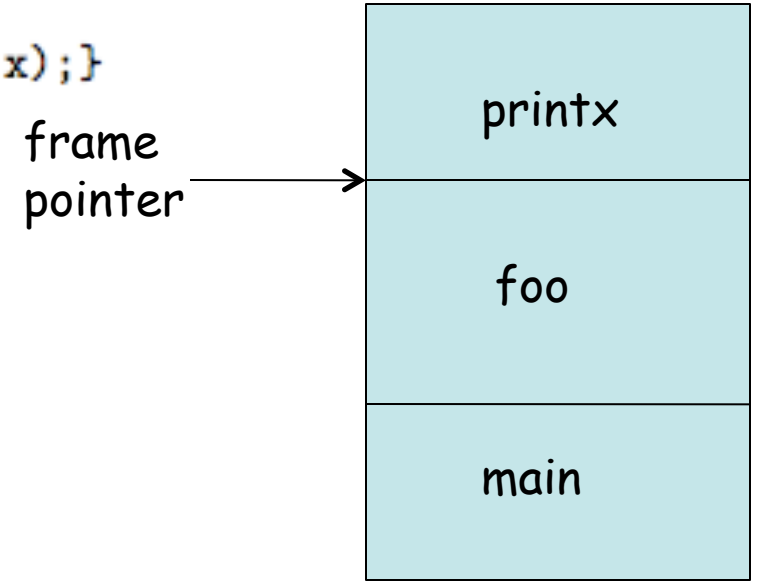
B. Runtime Stack

Simple C program with runtime stack

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```

PC

A. Source code

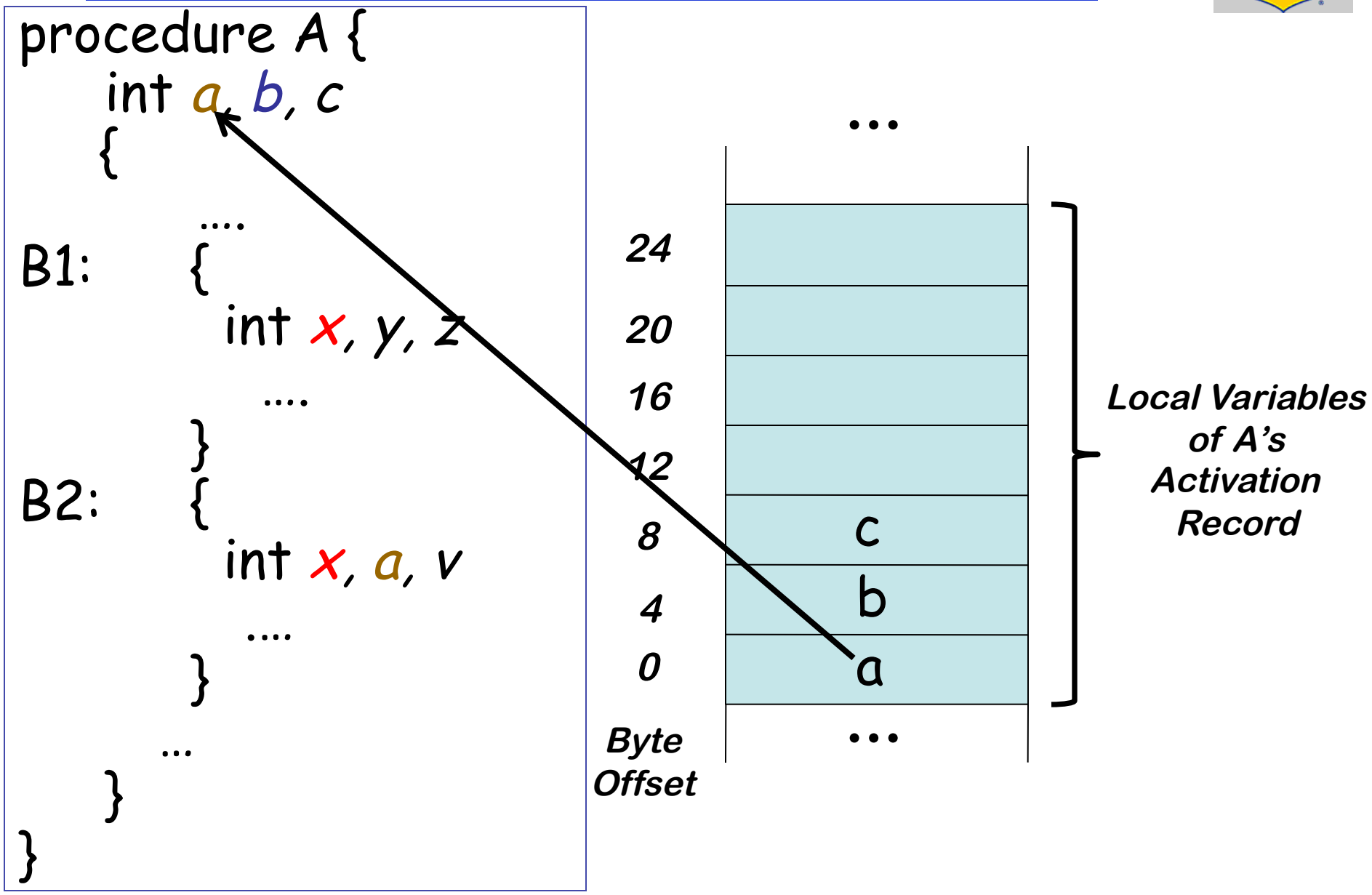


B. Runtime Stack

Local Storage

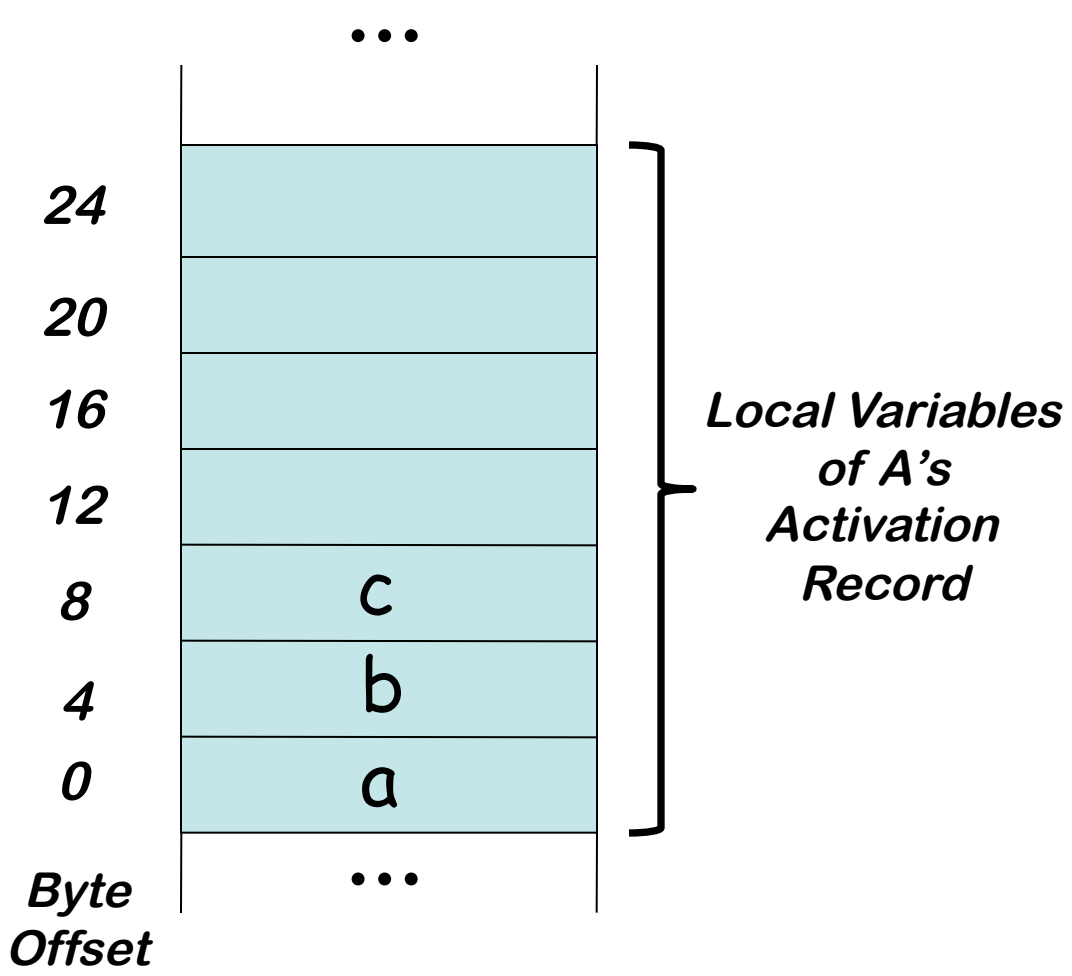
- Central part of activation record is static
  - → All fields have known fixed lengths
- Code can acccess items at fixed offsets from frame pointer (ARP)
- End of AR reserved for variable-length data

Storage for Blocks within a Single Procedure



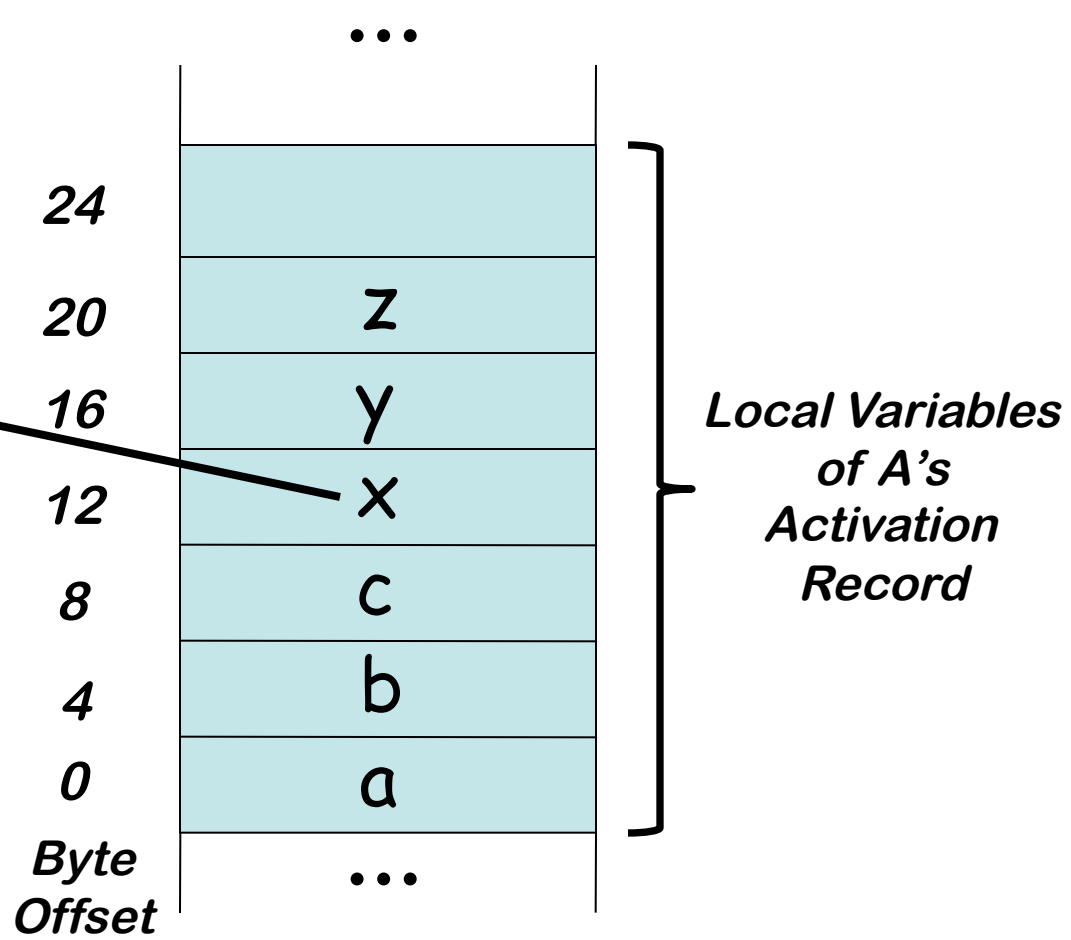
What about x in scope's B1 and B2?

```
procedure A {
    int a, b, c
    {
        ....
B1:     {
          int x, y, z
            ....
        }
B2:     {
          int x, a, v
            ....
        }
      ...
    }
}
```



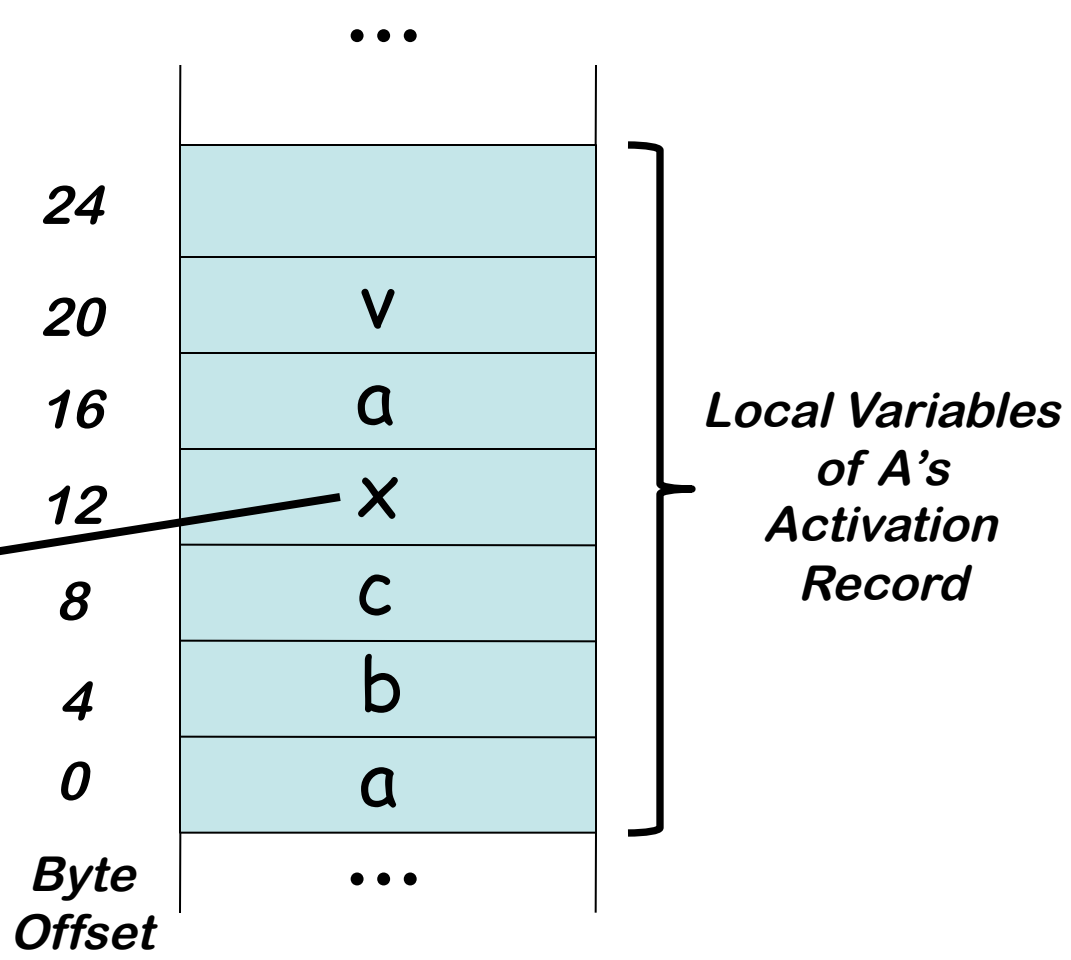
What about x in scope's B1 and B2?

```
procedure A {
    int a, b, c
    {
        ....
B1:     {
          int x, y, z
            ....
        }
B2:     {
          int x, a, v
            ....
        }
      ...
    }
}
```



What about x in scope's B1 and B2?

```
procedure A {
    int a, b, c
    {
        ....
B1:     {
          int x, y, z
            ....
        }
B2:     {
          int x, a, v
            ....
        }
      ...
    }
}
```



Storage for Blocks within a Single Procedure

- Could implement activation record for each block. Too expensive!
- Share storage for blocks that don't overlap
  - → B2 and B3 do not overlap
  - → Offsets computed statically

  Blocks B1 and B2 share same offsets!

Activation Record Details

How does the compiler find the variables?

- They are at known offsets from AR pointer
- Leads to a "loadAO" inst.
  - → loadAO arp, 8 # arp register and offset 8 bytes
- Compiler creates a **static coordinate** of variable

Activation Record Details

Variable-length data

- If AR can be extended, put it after local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

- Must generate explicit code to store the values
- Among the procedure's first actions

Establishing Addressability

Must create base addresses

- Global & static variables
  - → Construct a label by mangling names (*i.e.*, &_fee)

Establishing Addressability

- Local variables
  - → Convert to static data coordinate and use **ARP** + offset

- Local variables of other procedures
  - → Convert to static coordinates
  - → Find appropriate **ARP**
  - → Use that **ARP** + offset

Must find the right AR

Need links to nameable ARs

What about Nested Subprograms?

- Cannot determine distance to AR of non-local variable

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```

Body of **zab**

What about Nested Subprograms?

- Cannot determine distance to AR of non-local variable

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```

Body of **baz**

What about Nested Subprograms?

- Cannot determine distance to AR of non-local variable

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```

Body of **foo**

What about Nested Subprograms?

- Can determine number of activation records down the stack

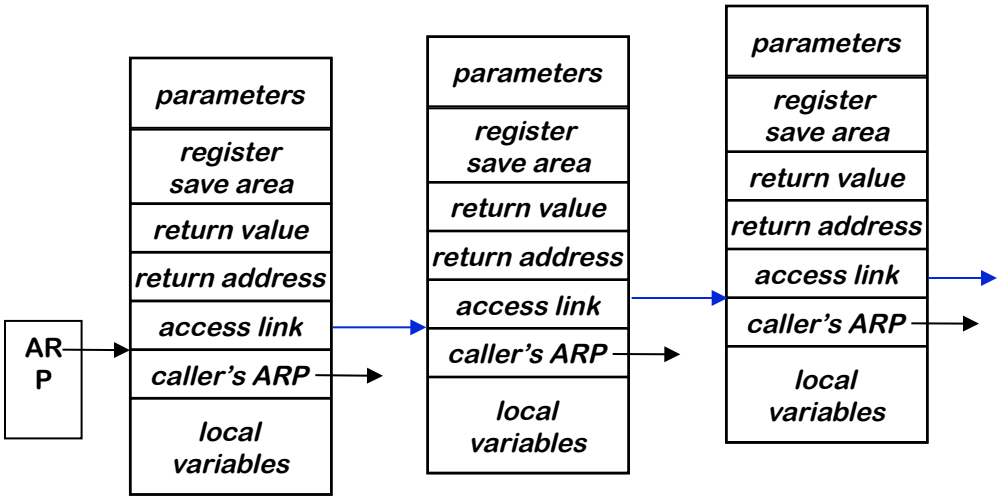
Use Static Coordinate and Access Links

- Name is translated into a *static coordinate*
  - → ‹ *level,offset* › pair
  - → "*level*" is lexical nesting level of the procedure
  - → "*offset*" is *unique* within that scope

Establishing Addressability

Using access links

- Each AR has a pointer to AR of lexical ancestor
- Lexical ancestor need not be the caller



- Reference to <*p*,16> runs up access link chain to *p*
- Cost of access is proportional to lexical distance

Establishing Addressability

Using access links

SC	Generated Code
<2,8>	loadAI r_0, 8 ⇒ r_2
<1,12>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0,16>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, -4 ⇒ r_1 loadAI r_1, 16 ⇒ r_2

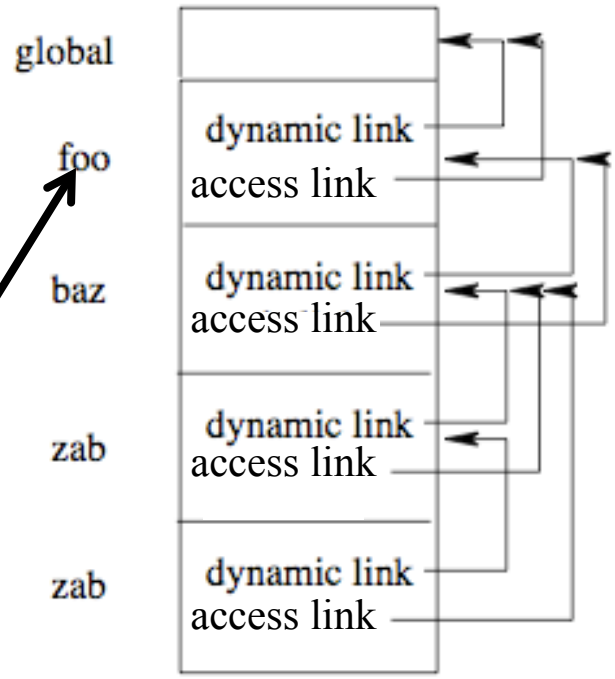
Access & maintenance cost varies with level

All accesses are relative to ARP (r_0)

Establishing Addressability

Using access links

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```

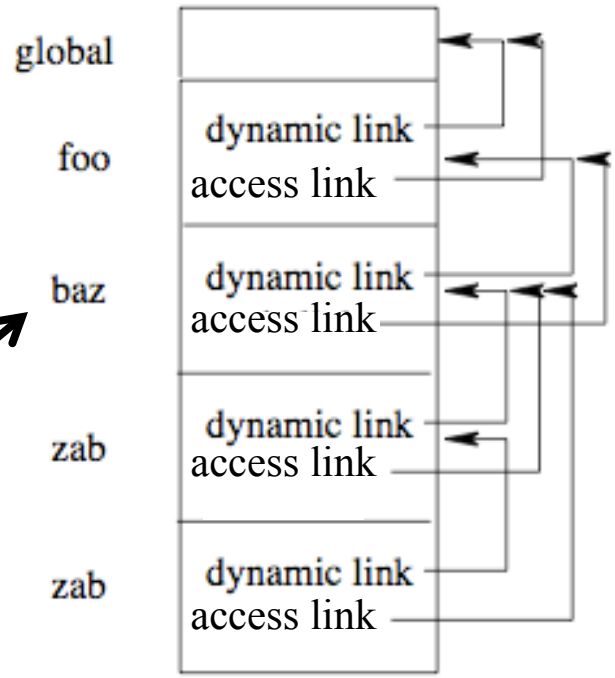


Establishing Addressability

Using access links

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```

PC

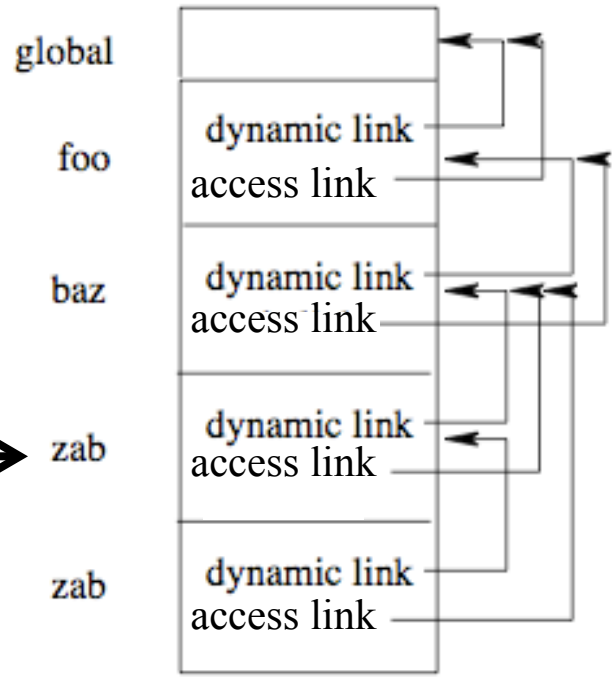


Establishing Addressability

Using access links

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```

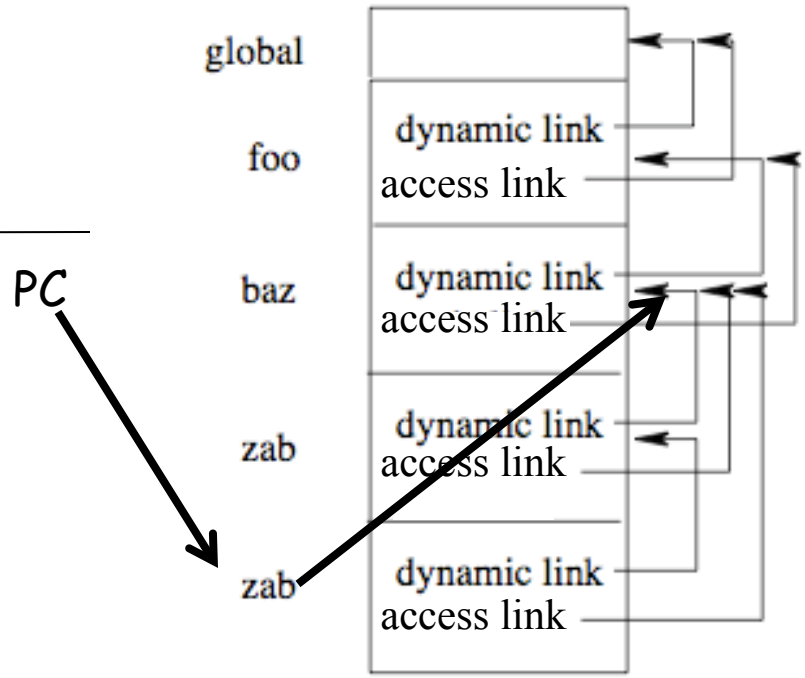
PC



Establishing Addressability

Using access links

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```



Back to Activation Records

If activation records are stored on the stack

- Easy to extend — simply bump top of stack pointer
- Caller & callee share responsibility
  - → Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own **ARP**
  - → Callee can push space for local variables (fixed & variable size)

Back to Activation Records

If activation records are stored on the heap

- Hard to extend
- Caller passes everything it can in registers
- Callee allocates AR & stores register contents into it
  → Extra parameters stored in caller's **AR** !