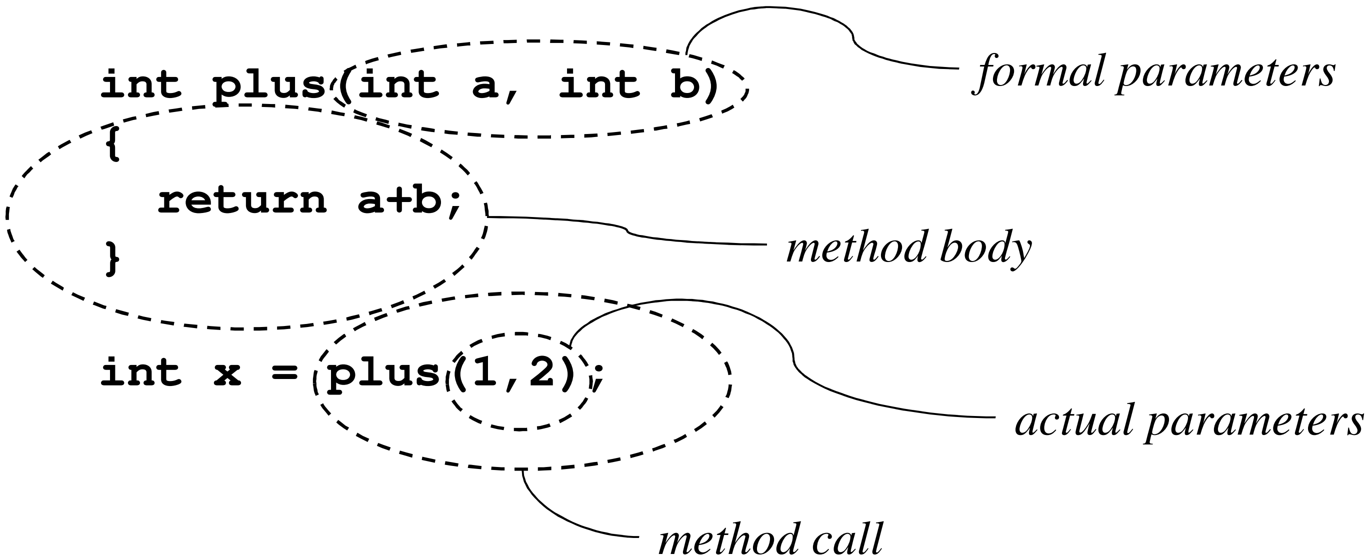
Parameters

Parameter Passing



- How are parameters passed?
- Looks simple enough…
- We will see seven techniques

Outline

- 18.2 Parameter correspondence
- Implementation techniques
  - 18.3 By value
  - 18.4 By result
  - 18.5 By value-result
  - 18.6 By reference
  - 18.7 By macro expansion
  - 18.8 By name
  - 18.9 By need
- 18.10 Specification issues

Parameter Correspondence

- A preliminary question: how does the language match up parameters?
- That is, which formal parameters go with which actual parameters?
- Most common case: *positional parameters*
  - Correspondence determined by positions
  - *n*th formal parameter matched with *n*th actual

Keyword Parameters

- Correspondence can be determined by matching parameter names
- Ada:

```
DIVIDE(DIVIDEND => X, DIVISOR => Y);
```

- Matches actual parameter **x** to formal parameter **DIVIDEND**, and **Y** to **DIVISOR**
- Parameter order is irrelevant here

Mixed Keyword And Positional

- Most languages that support keyword parameters allow both: Ada, Fortran, Dylan, Python
- The first parameters in a list can be positional, and the remainder can be keyword parameters

Optional Parameters

- Optional, with default values: formal parameter list includes default values to be used if the corresponding actual is missing
- This gives a very short way of writing certain kinds of overloaded function definitions

Example: C++

```
int f(int a=1, int b=2, int c=3) { body }
```

```
int f() {f(1,2,3);}
int f(int a) {f(a,2,3);}
int f(int a, int b) {f(a,b,3);}
int f(int a, int b, int c) { body }
```

Unlimited Parameter Lists

- Some languages allow actual parameter lists of unbounded length: C, C++, and scripting languages like JavaScript, Python, and Perl
- Library routines must be used to access the excess actual parameters
- A hole in static type systems, since the types of the excess parameters cannot be checked at compile time

```
int printf(char *format, ...) { body }
```

Outline

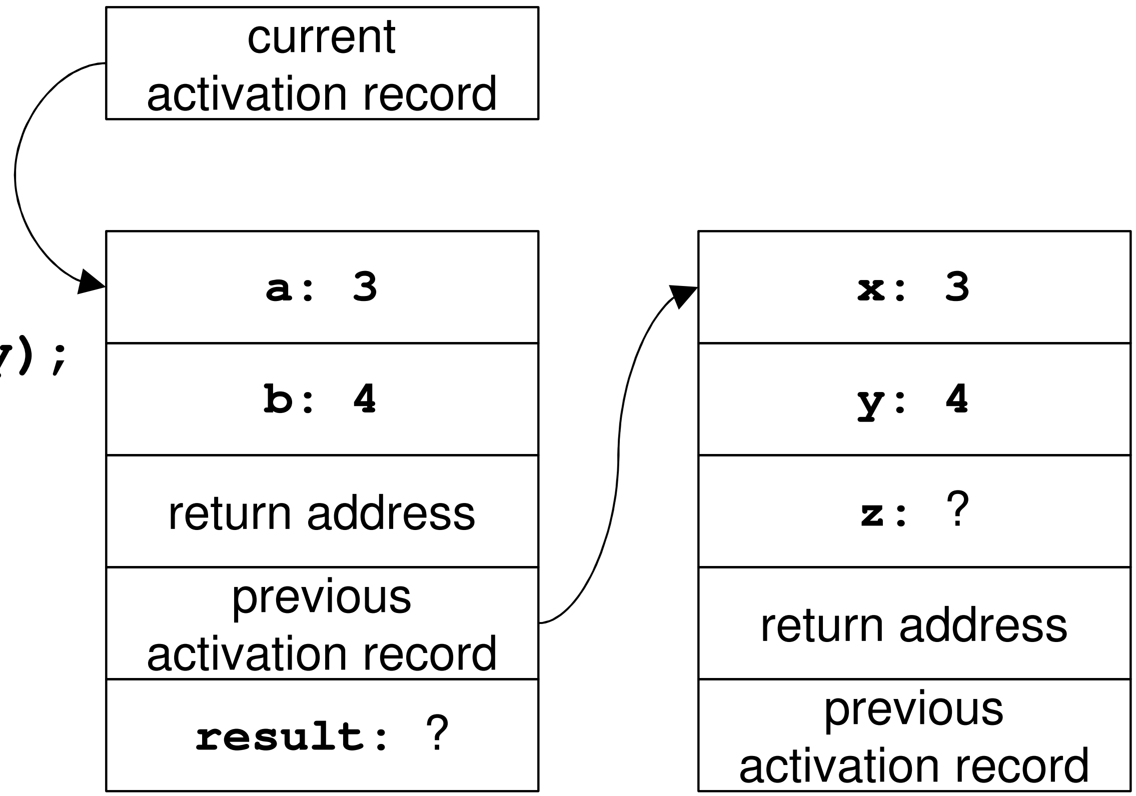
- 18.2 Parameter correspondence
- Implementation techniques
  - 18.3 By value
  - 18.4 By result
  - 18.5 By value-result
  - 18.6 By reference
  - 18.7 By macro expansion
  - 18.8 By name
  - 18.9 By need
- 18.10 Specification issues

By Value

For by-value parameter passing, the formal parameter is just like a local variable in the activation record of the called method, with one important difference: it is initialized using the value of the corresponding actual parameter, before the called method begins executing.

- Simplest method
- Widely used
- The only method in real Java

```
int plus(int a, int b) {
  a += b;
  return a;
}

void f() {
  int x = 3;
  int y = 4;
  int z = plus(x, y);
}
```

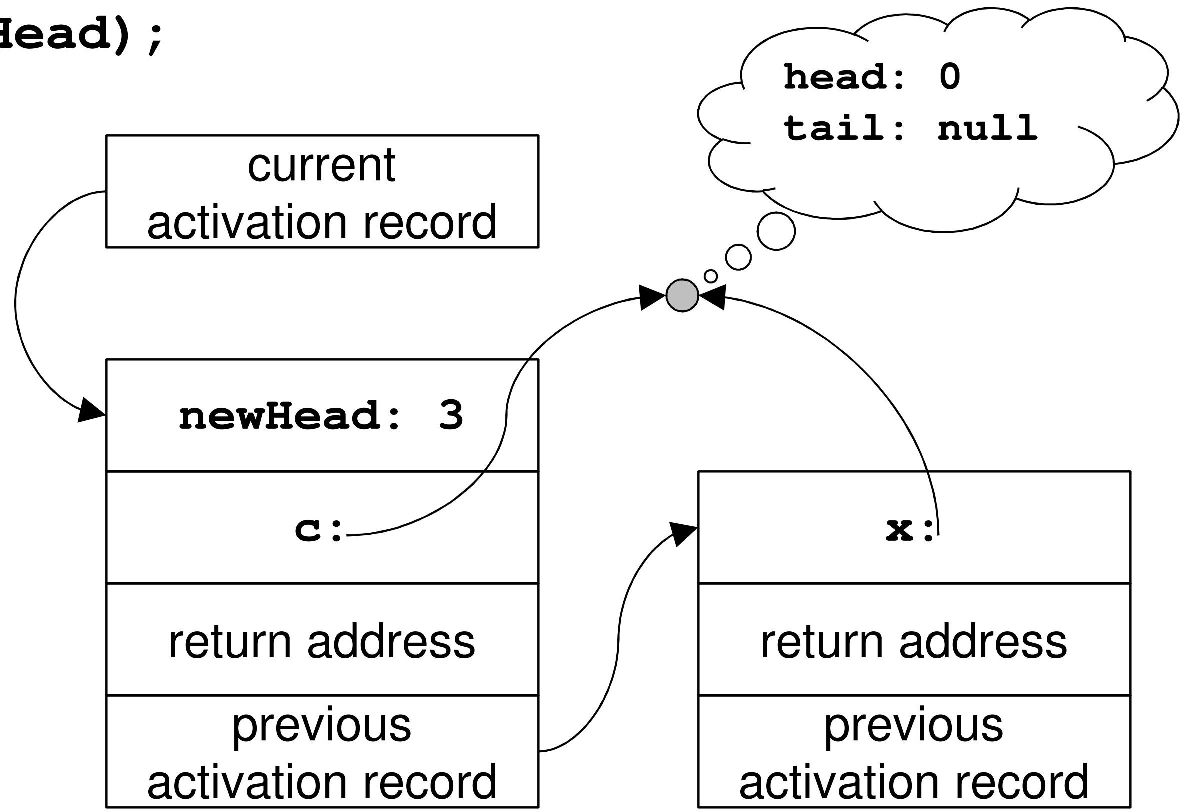


When **plus** is starting

Changes Visible To The Caller

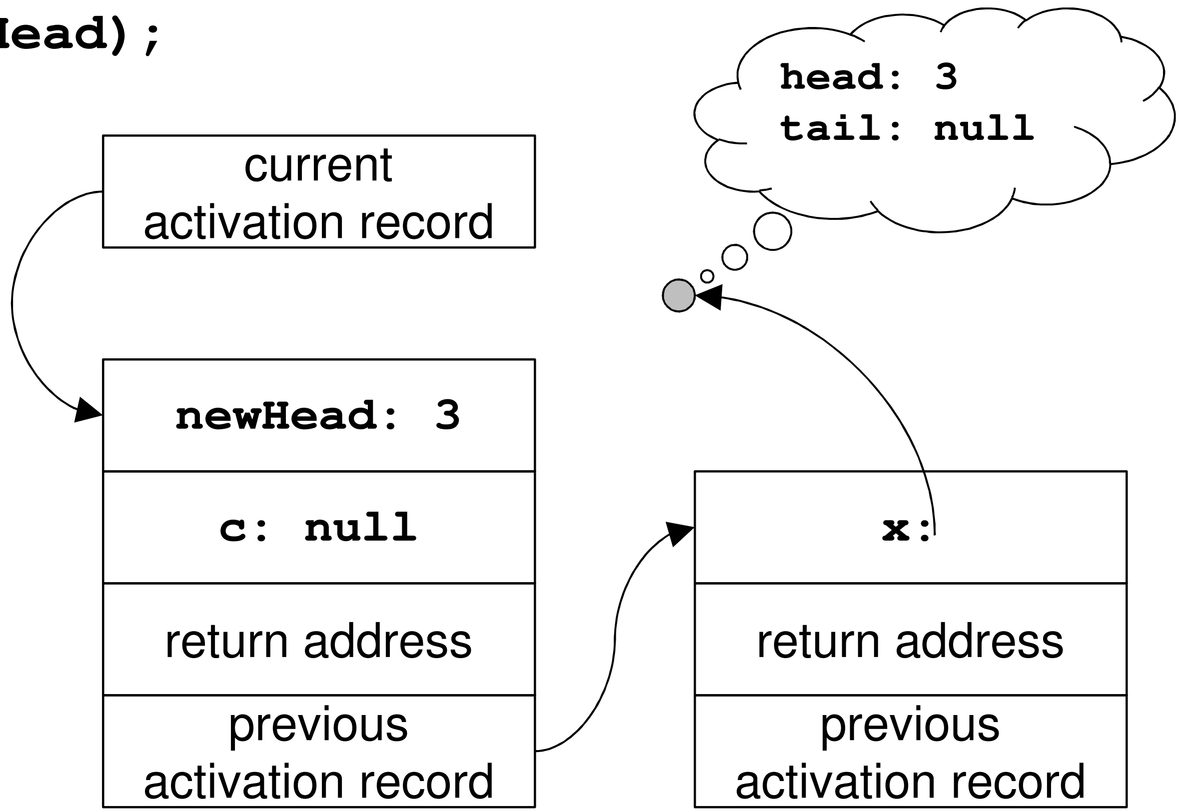
- When parameters are passed by value, changes to a formal do not affect the actual
- But it is still possible for the called method to make changes that are visible to the caller
- The value of the parameter could be a pointer (in Java, a reference)
- Then the actual cannot be changed, but the object referred to by the actual can be

```
void f() {
  ConsCell x = new ConsCell(0,null);
  alter(3,x);
}

void alter(int newHead, ConsCell c) {
  c.setHead(newHead);
  c = null;
}
```



When **alter** is starting

```
void f() {
  ConsCell x = new ConsCell(0,null);
  alter(3,x);
}

void alter(int newHead, ConsCell c) {
  c.setHead(newHead);
  c = null;
}
```



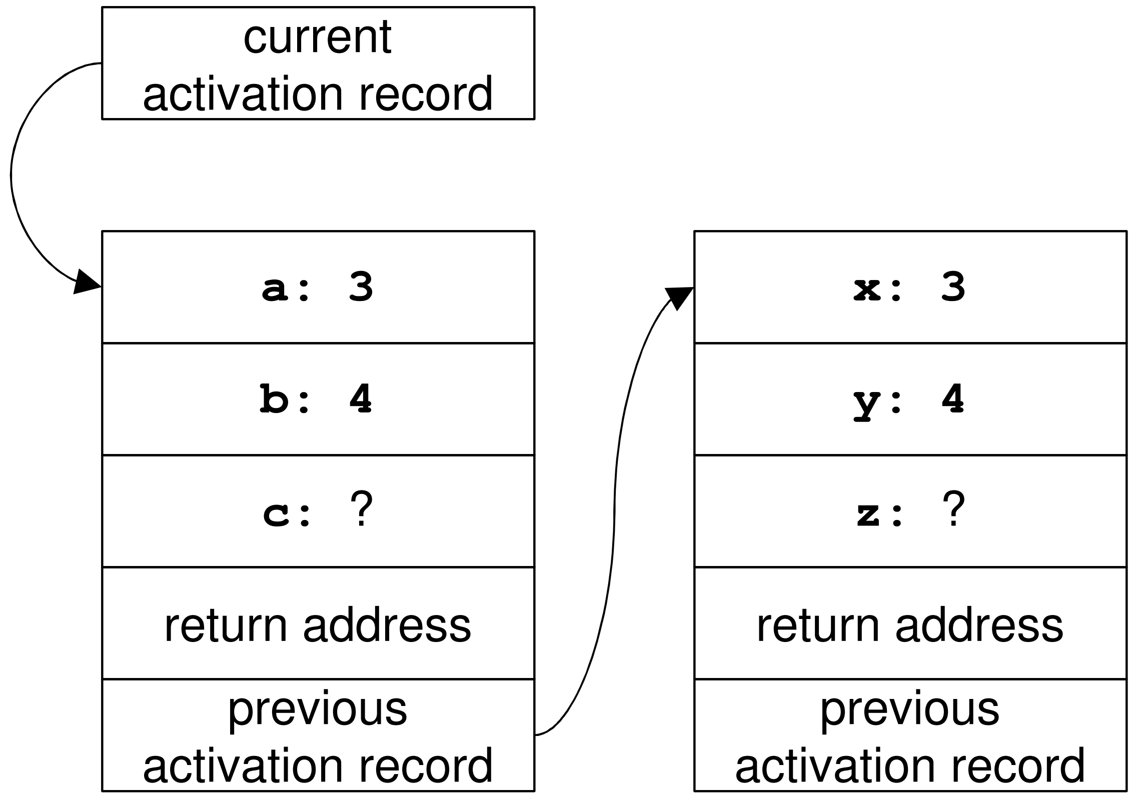
When **alter** is finishing

By Result

For by-result parameter passing, the formal parameter is just like a local variable in the activation record of the called method—it is uninitialized. After the called method finished executing, the final value of the formal parameter is assigned to the corresponding actual parameter.

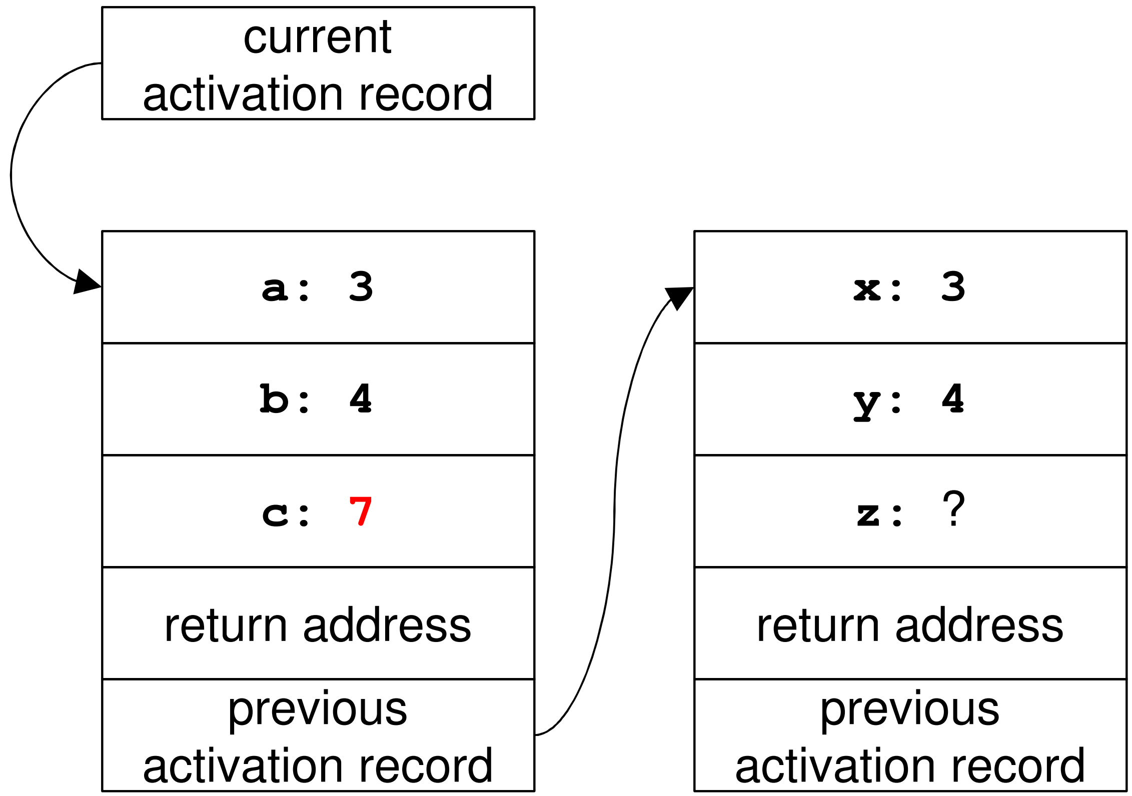
- Also called *copy-out*
- Actual must have an lvalue
- Introduced in Algol 68; sometimes used for Ada

```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



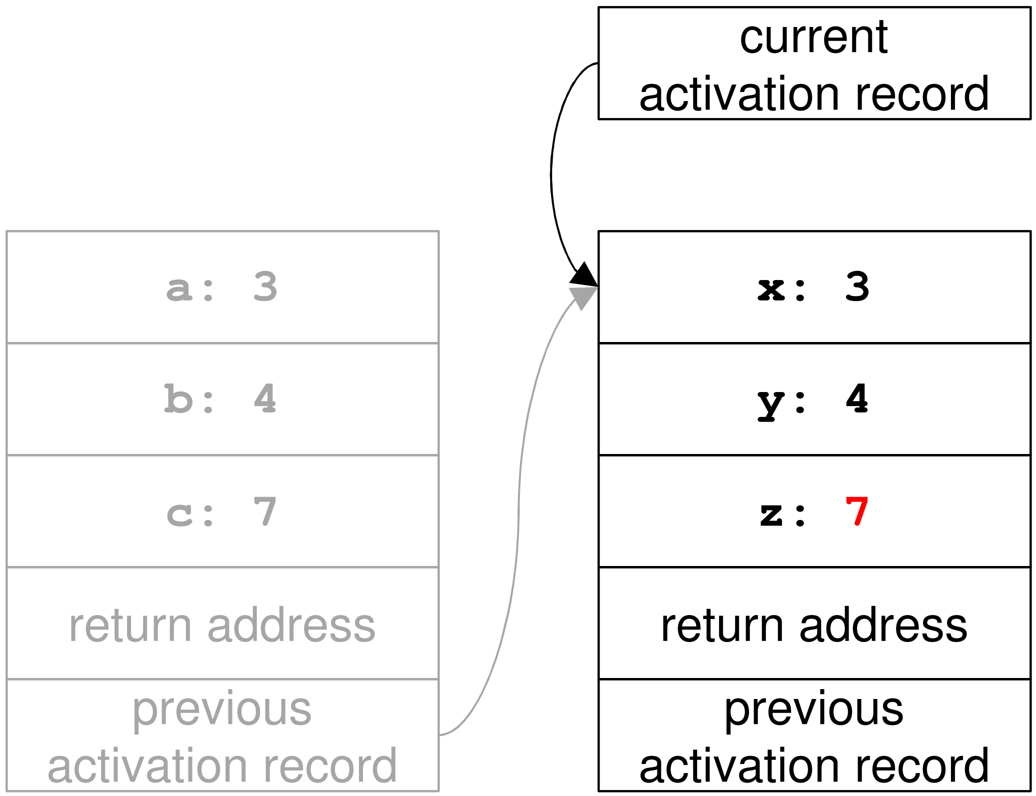
When **plus** is starting

```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



When **plus** is ready to return

```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



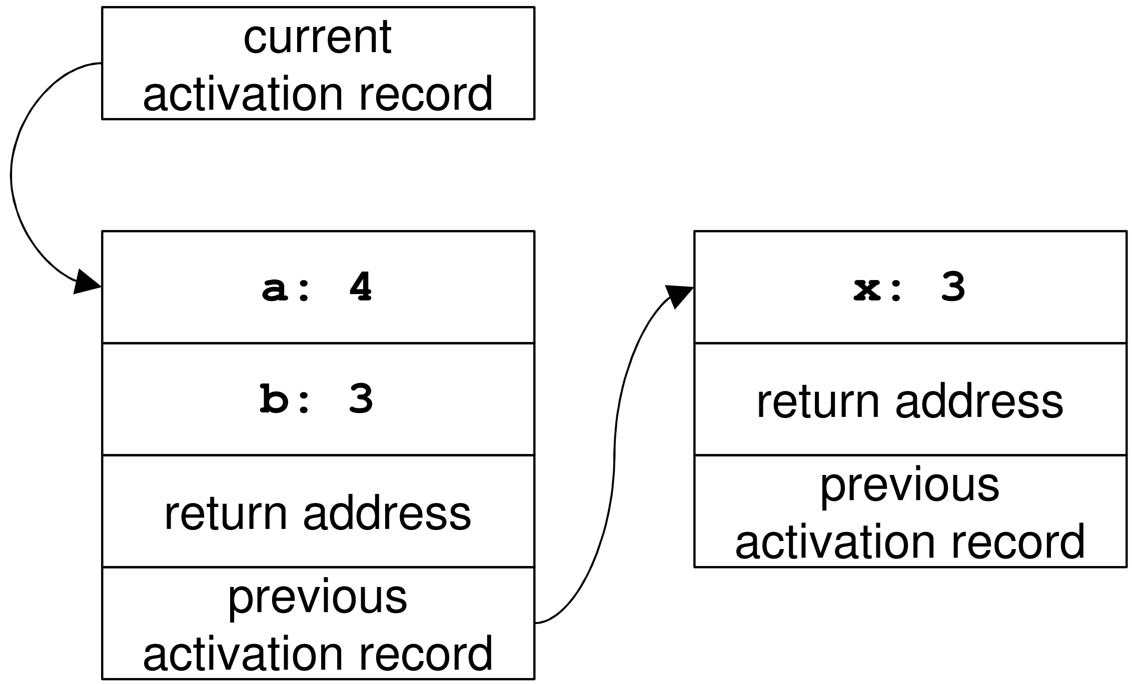
When **plus** has returned

By Value-Result

For passing parameters by value-result, the formal parameter is just like a local variable in the activation record of the called method. It is initialized using the value of the corresponding actual parameter, before the called method begins executing. Then, after the called method finishes executing, the final value of the formal parameter is assigned to the actual parameter.

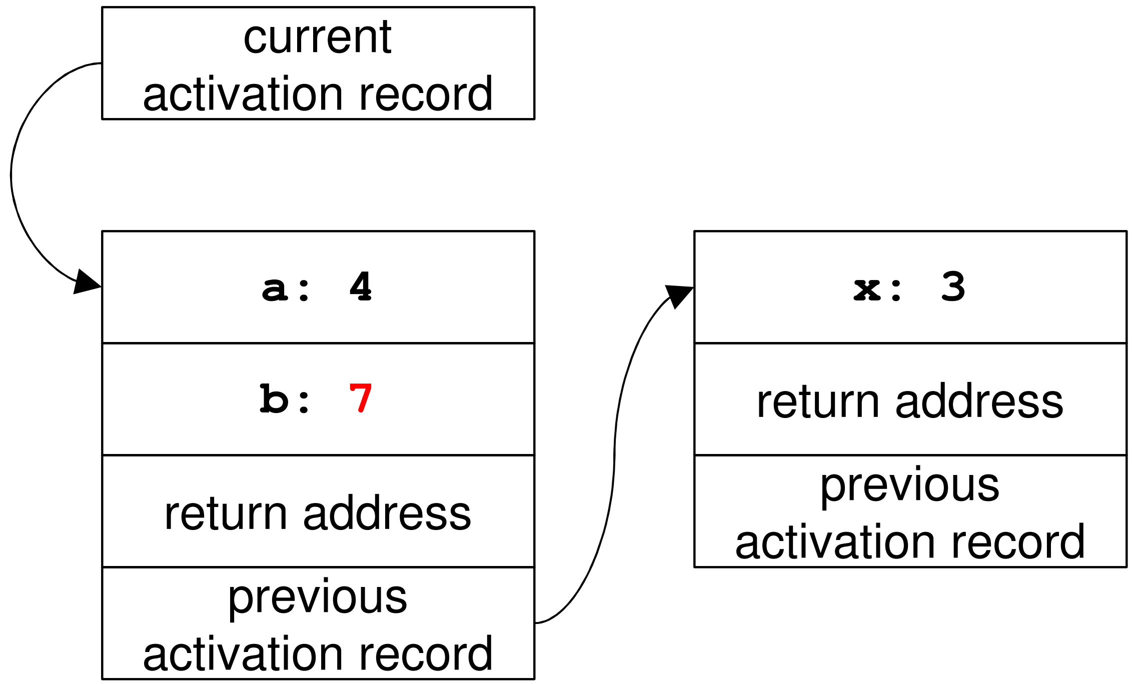
- Also called *copy-in/copy-out*
- Actual must have an lvalue

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



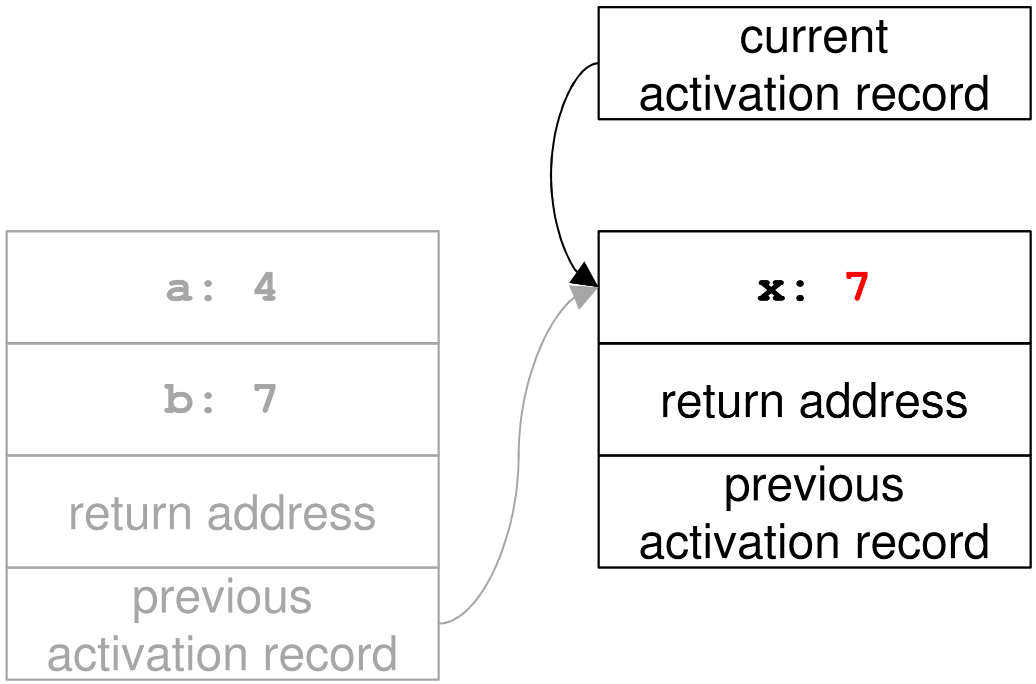
When **`plus`**
is starting

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



When **`plus`** is ready to return

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



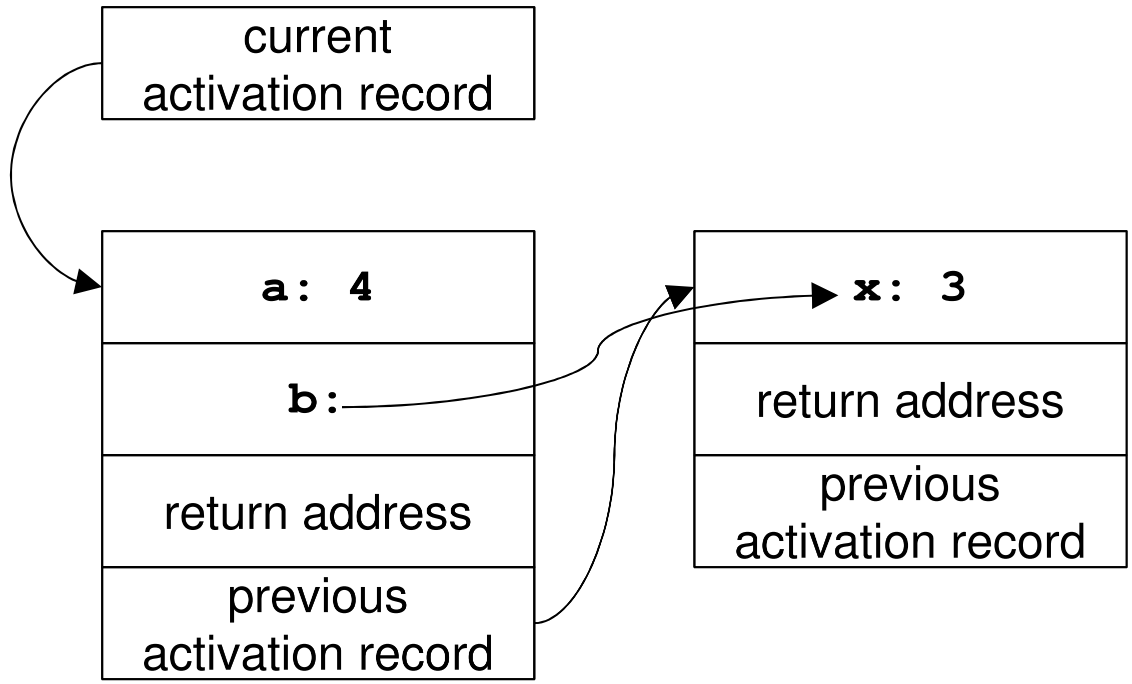
When **plus**
has returned

By Reference

For passing parameters by reference, the lvalue of the actual parameter is computed before the called method executes. Inside the called method, that lvalue is used as the lvalue of the corresponding formal parameter. In effect, the formal parameter is an alias for the actual parameter—another name for the same memory location.

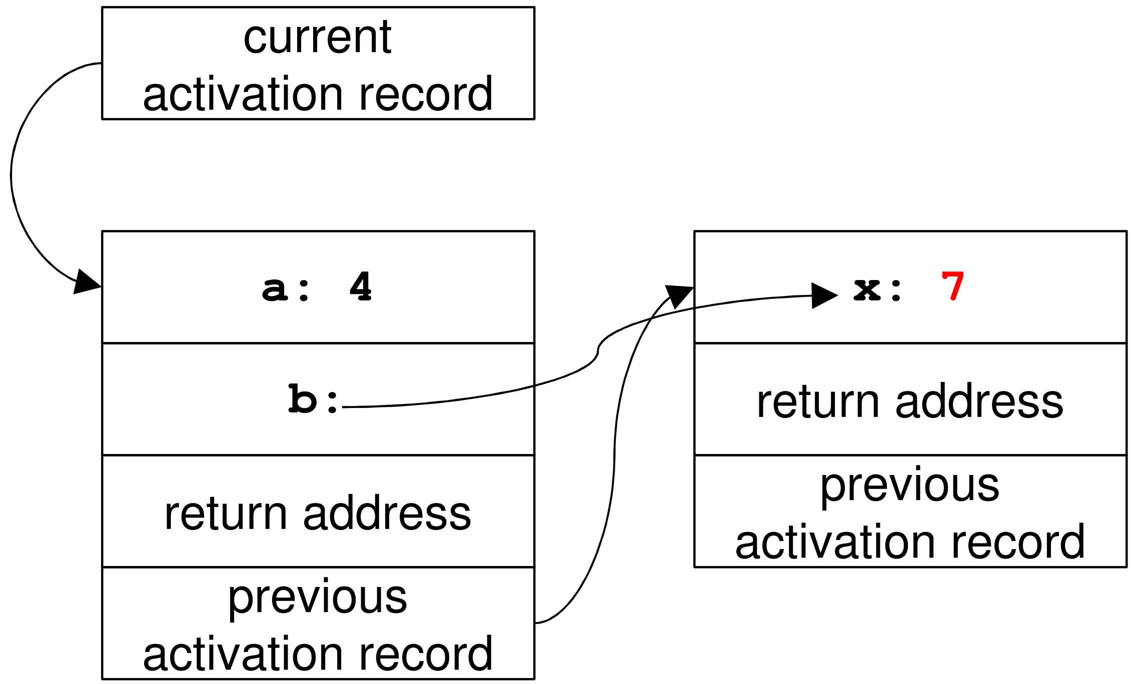
- One of the earliest methods: Fortran
- Most efficient for large objects
- Still frequently used

```
void plus(int a, by-reference int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



When **plus** is starting

```
void plus(int a, by-reference int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



When **plus** has made the assignment

Implementing Reference

```
void plus(int a, by-reference int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```

Previous example

```
void plus(int a, int *b) {
  *b += a;
}
void f() {
  int x = 3;
  plus(4, &x);
}
```

C implementation

By-reference = address by value

Aliasing

- When two expressions have the same lvalue, they are *aliases* of each other
- There are obvious cases:

```
ConsCell x = new ConsCell(0,null);
ConsCell y = x;

A[i]=A[j]+A[k];
```

- Passing by reference leads to less obvious cases…

Example

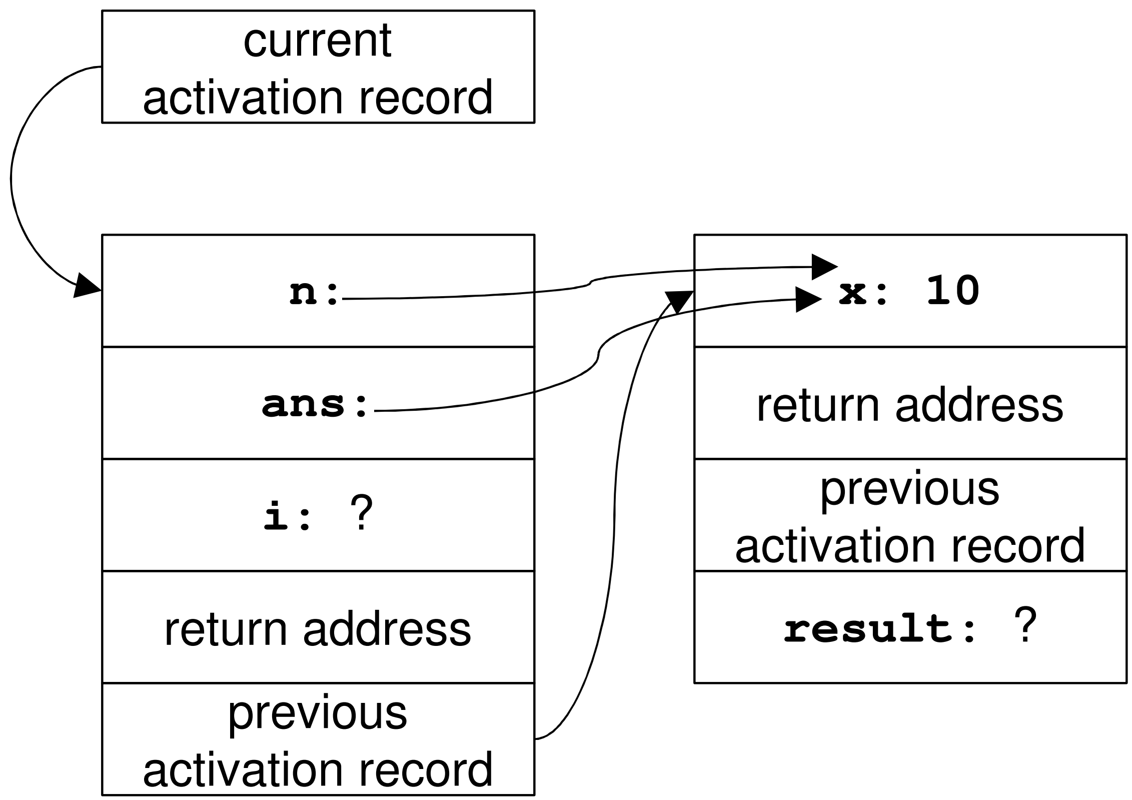
```
void sigsum(by-reference int n,
            by-reference int ans) {
  ans = 0;
  int i = 1;
  while (i <= n) ans += i++;
}
```

```
int f() {
  int x,y;
  x = 10;
  sigsum(x,y);
  return y;
}
```

```
int g() {
  int x;
  x = 10;
  sigsum(x,x);
  return x;
}
```

```
void sigsum(by-reference int n,
            by-reference int ans) {
  ans = 0;
  int i = 1;
  while (i <= n) ans += i++;
}

int g() {
  int x;
  x = 10;
  sigsum(x,x);
  return x;
}
```

When **sigsum** is starting

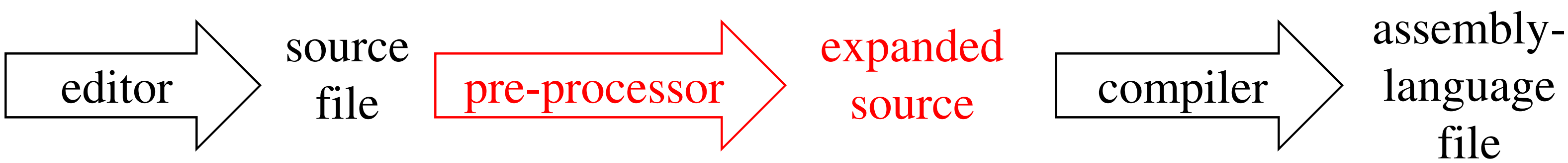


By Macro Expansion

For passing parameters by macro expansion, the body of the macro is evaluated in the caller's context. Each actual parameter is evaluated on every use of the corresponding formal parameter, in the context of that occurrence of that formal parameter (which is itself in the caller's context).

- Like C macros
- Natural implementation: textual substitution before compiling

Macro Expansions In C



- An extra step in the classical sequence
- Macro expansion before compilation

source file:

```
#define MIN(X,Y) ((X)<(Y)?(X):(Y))
a = MIN(b,c);
```

expanded source:

```
a = ((b)<(c)?(b):(c))
```

Preprocessing

- Replace each use of the macro with a copy of the macro body, with actuals substituted for formals
- An old technique, used in assemblers before the days of high-level languages
- It has some odd effects…

Repeated Evaluation

- Each actual parameter is re-evaluated every time it is used

source file:

```
#define MIN(X,Y) ((X)<(Y)?(X):(Y))
a = MIN(b++,c++);
```

expanded source:

```
a = ((b++)<(c++)?(b++):(c++))
```

Capture Example

source file:

```
#define intswap(X,Y) {int temp=X; X=Y; Y=temp;}
int main() {
  int temp=1, b=2;
  intswap(temp,b);
  printf("%d, %d\n", temp, b);
}
```

expanded source:

```
int main() {
  int temp=1, b=2;
  {int temp= temp ;  temp = b ;  b =temp;} ;
  printf("%d, %d\n", temp, b);
}
```

Capture

- In a program fragment, any occurrence of a variable that is not statically bound is *free*
- When a fragment is moved to a different context, its free variables can become bound
- This phenomenon is called *capture*:
  - Free variables in the actuals can be captured by definitions in the macro body
  - Also, free variables in the macro body can be captured by definitions in the caller

By Name

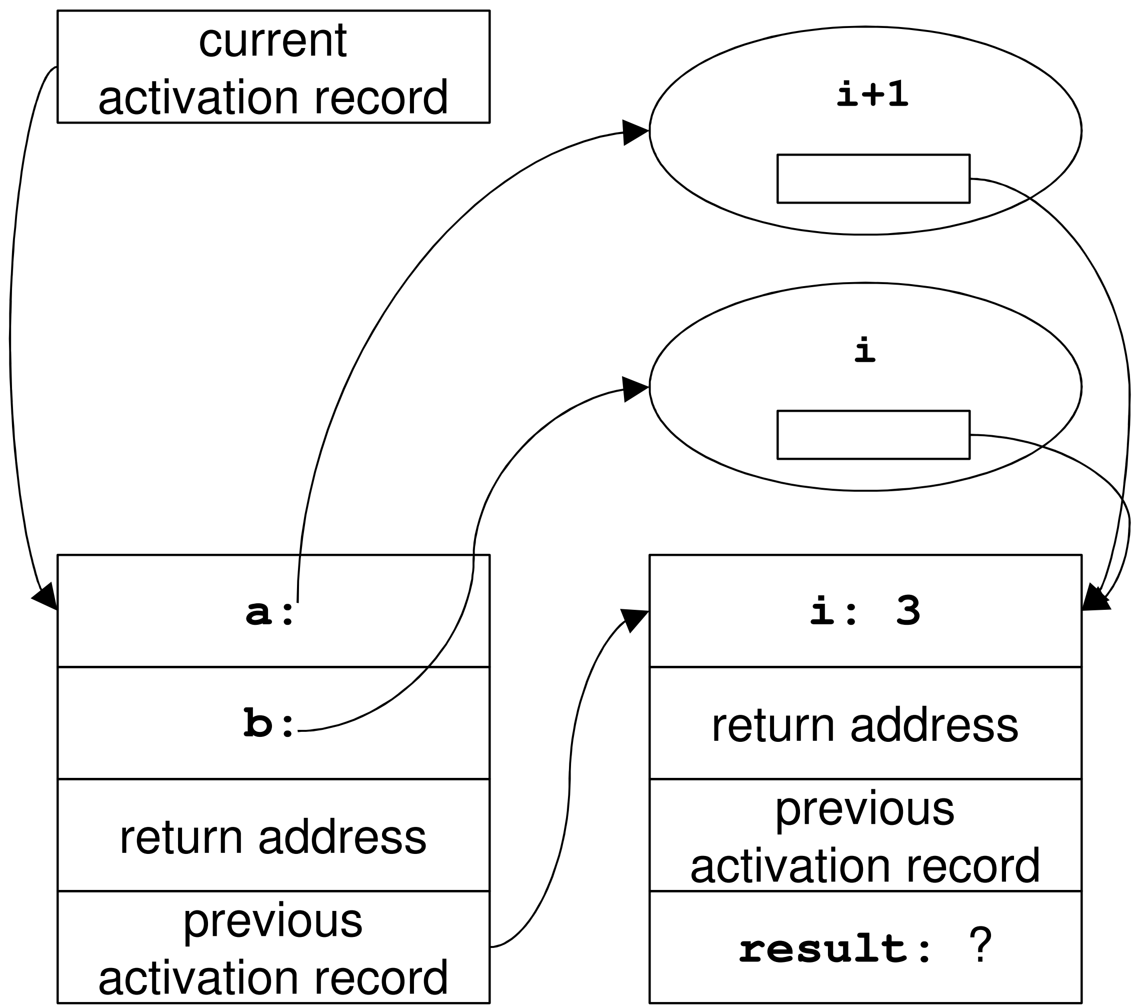
For passing parameters by name, each actual parameter is evaluated in the caller's context, on every use of the corresponding formal parameter.

- Like macro expansion without capture
- Algol 60 and others
- Now unpopular

Implementing By-Name

- The actual parameter is treated like a little anonymous function
- Whenever the called method needs the value of the formal (either rvalue or lvalue) it calls the function to get it
- The function must be passed with its nesting link, so it can be evaluated in the caller’s context

```
void f(by-name int a, by-name int b) {
  b=5;
  b=a;
}

int g() {
  int i = 3;
  f(i+1,i);
  return i;
}
```



When **f** is starting

Comparison

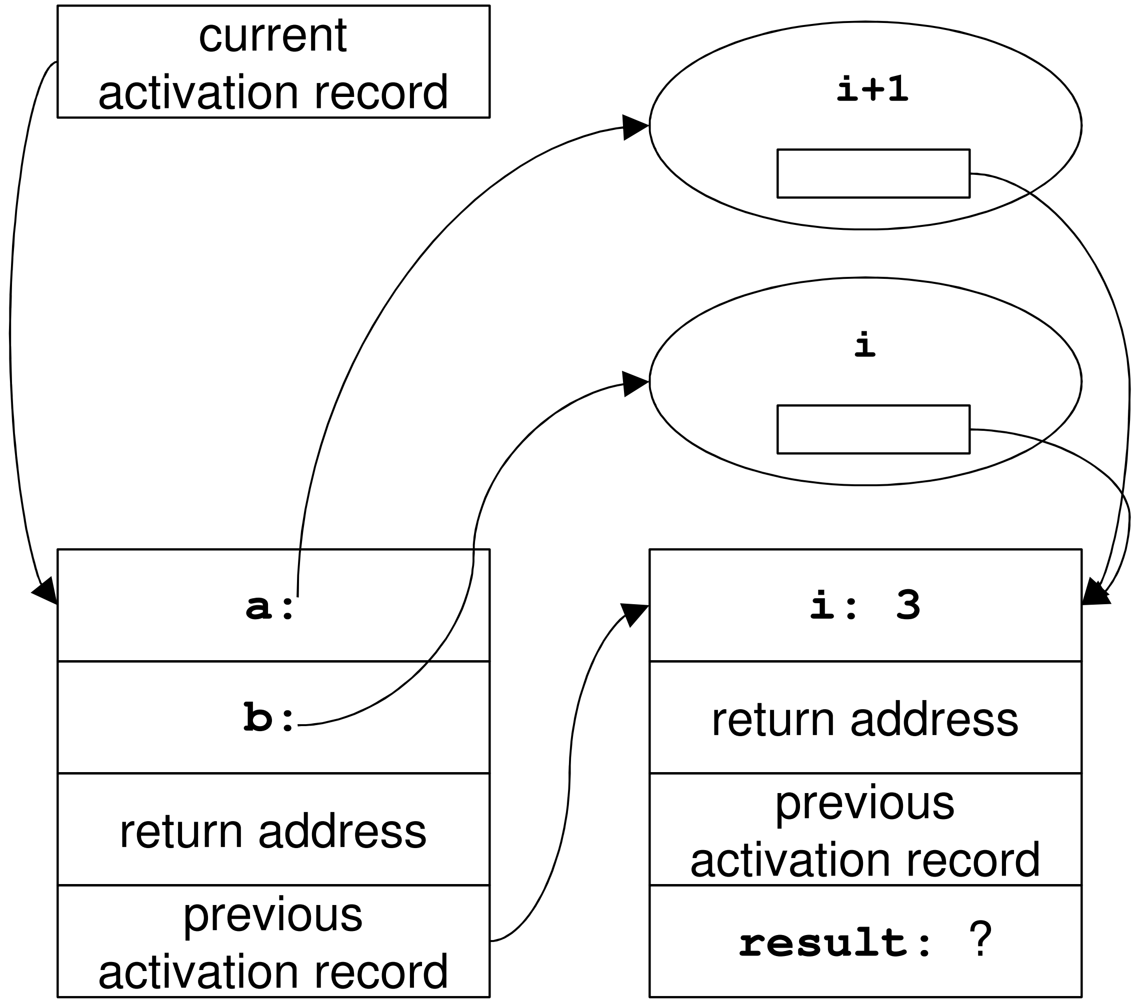
- Like macro expansion, by-name parameters are re-evaluated every time they are used
- (Can be useful, but more often this is merely wasteful)
- Unlike macro expansion, there is no possibility of capture

By Need

For passing parameters by need, each actual parameter is evaluated in the caller's context, on the first use of the corresponding formal parameter. The value of the actual parameter is then cached, so that subsequent uses of the corresponding formal parameter do not cause reevaluation.

- Used in lazy functional languages (Haskell)
- Avoids wasteful recomputations of by-name

```
void f(by-need int a, by-need int b) {
  b=a;
  b=a;
}

void g() {
  int i = 3;
  f(i+1,i);
  return i;
}
```



When **f** is starting

Laziness

```
boolean andand(by-need boolean a,
               by-need boolean b) {
  if (!a) return false;
  else return b;
}

boolean g() {
  while (true) {
  }
  return true;
}

void f() {
  andand(false,g());
}
```

Here, **`andand`** is short-circuiting, like ML's **`andalso`** and Java's **`&&`** operators.

The method **`f`** will terminate.

Same behavior for by-name and macro expansion.

Outline

- 18.2 Parameter correspondence
- Implementation techniques
  - 18.3 By value
  - 18.4 By result
  - 18.5 By value-result
  - 18.6 By reference
  - 18.7 By macro expansion
  - 18.8 By name
  - 18.9 By need
- 18.10 Specification issues

Specification Issues

- Are these just implementation techniques, or part of the language specification?
- Depends on the language:
  - Without side-effects, parameter-passing technique may be undetectable by the programmer
  - Even with side effects, some languages specify the parameter passing technique only partially

Without Side Effects

- Big question: are parameters always evaluated (*eager evaluation*), or only if they are really needed (*lazy evaluation*)?
- Cost model may also be used by the programmer (more in Chapter 21):
  - Is re-evaluation of a formal expensive?
  - Does parameter-passing take time proportional to the size of the object?

With Side Effects

- A program can detect which parameter-passing technique is being used by the language system
- But it may be an implementation detail that programs are not supposed to depend on—it may not be part of the specification of the language
- Case in point: Ada

Ada Modes

- Three parameter-passing modes:
  - **`in`**: these can be read in the called method, but not assigned—like constants
  - **`out`**: these must be assigned and cannot be read
  - **`in out`**: may be read and/or assigned
- Ada specification intentionally leaves some flexibility for implementations

Ada Implementations

- Copying is specified for scalar values:
  - **`in`** = value, **`out`** = result, **`in out`** = value/result
- Aggregates like arrays and records *may* be passed by reference instead
- Any program that can detect the difference (like some of our earlier examples) is not a legal Ada program

Conclusion

- Today:
  - How to match formals with actuals
  - Seven different parameter-passing techniques
  - Ideas about where to draw the line between language definition and implementation detail
- These are not the only schemes that have been tried, just some of the most common
- The CS corollary of Murphy's Law:

Inside every little problem there is a big problem waiting to get out