Functions and Recursions

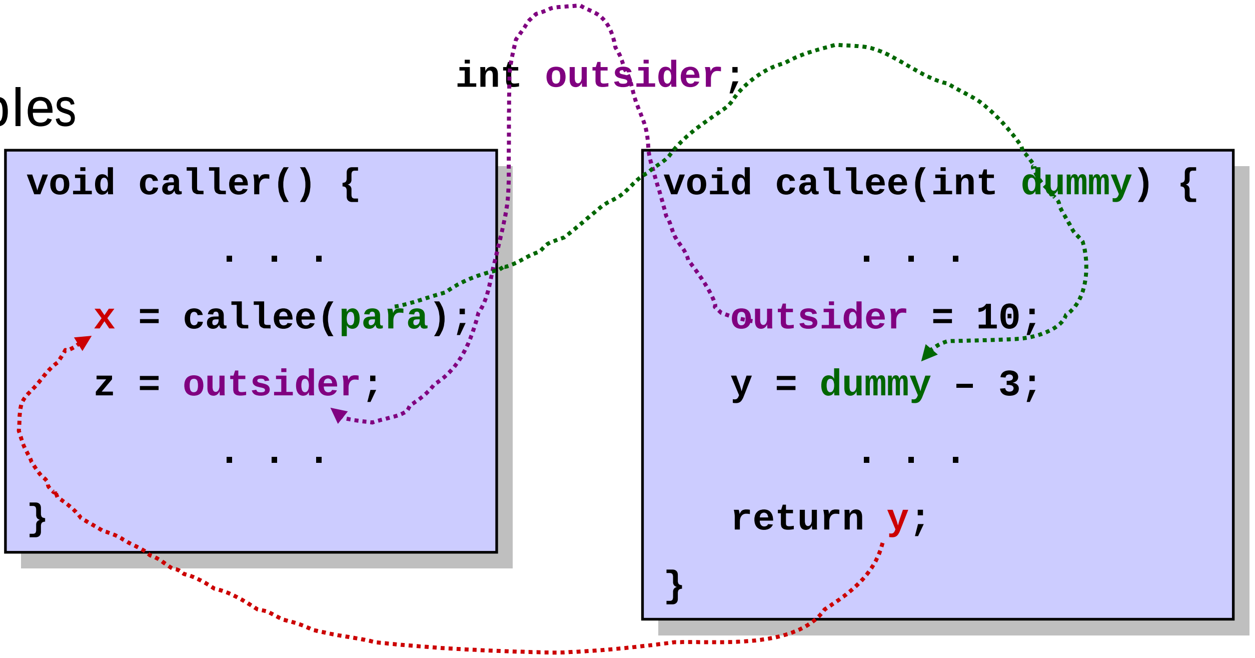
Kyuseok Shim
(Modified Professor Yunheung Paek‘s slides)
Seoul National University

Topics

- Function/Procedure calls
- Parameter passing
  - Call by value
  - Call by reference
- Higher-Order Functions (HOFs)

Function/Procedure calls

- Why is a **procedure call/invocation** a universal feature in programming languages?
- A procedure call involves a caller and a callee.
- The caller and the callee involved in the same call should communicate to exchange information necessary for the call. Then, how?
  - using global variables
  - using parameters and returning values



```
int outsider;
```

```
void caller() {
        . . .
    x = callee(para);
    z = outsider;
        . . .
}
```

```
void callee(int dummy) {
        . . .
    outsider = 10;
    y = dummy - 3;
        . . .
    return y;
}
```

Inlining

- Some languages such as C++ supports *explicit inlining*.

```
void foo() {
   int x,y,z;
   …
   y = bar(z,99);
   z = bar(88,y);
   …
}

inline int bar(int a, b) {
   int x,t;
   x = a * b;
   t = a - b;
   return x / t;
}
```

→

```
void foo() {
   int x,y,z,x1,t,x2,t1;
   …
   x1 = z * 99;
   t = z - 99;
   y = x1 / t;
   x2 = 88 * y;
   t1 = 88 - y;
   z = x2 / t1;
   …
}
```

- Pros and cons of inlining
  - It eliminates the overhead for procedure call.
  - Reckless use may increase the code size.
  - Inlined code is generally less readable and maintainable.

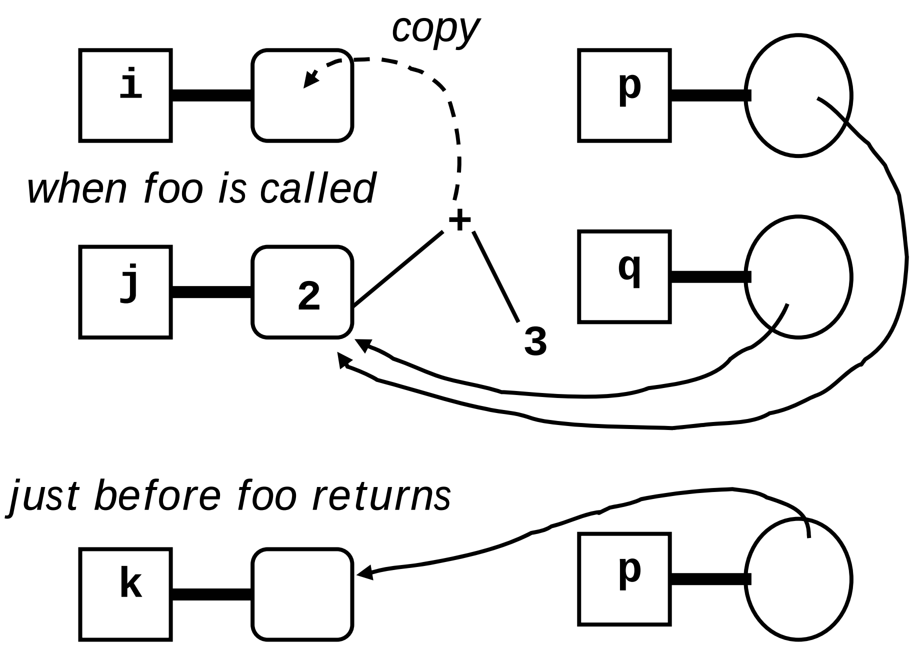
  → *So, inlining is ideal for a small procedure invoked within frequently executed regions (e.g., loops).*

Parameter passing

- parameter/argument passing
  - the study of the different ways of communication between a caller and a callee with parameters and results
- parameter passing methods
  - call-by-value
  - call-by-reference

Call-by-value

- When a procedure is called, the r-value of an actual argument is assigned to the l-value of the matching formal argument.
- secure because changes made on formal arguments do not affect the actual ones.



```
int k = i;
void foo(int i, int* p) {
        . . .
        p = &k;
}
void bar() {
        int j = 2;
        int* q = &j;
        . . .
        foo(j+3, q);
}
```

Call-by-value

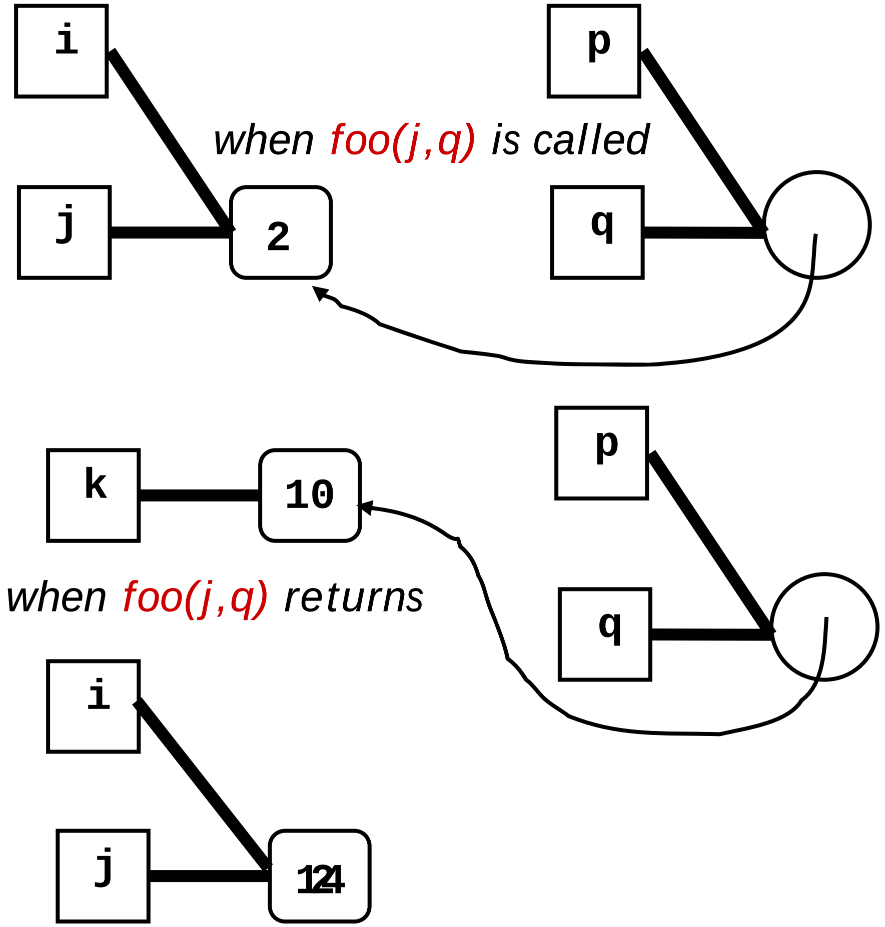
- can be expensive.
  - *Pascal*

```
type long_list = array [1..10000] of real;
var a : long_list;
procedure soar (d : long_list);
         . . .
end;
begin
         soar(a);
end;
```

- Typically, not appropriate if a callee wants to return multiple results.

Call-by-reference/location

- When a procedure is called, the l-value of an actual argument is shared with the matching dummy argument.

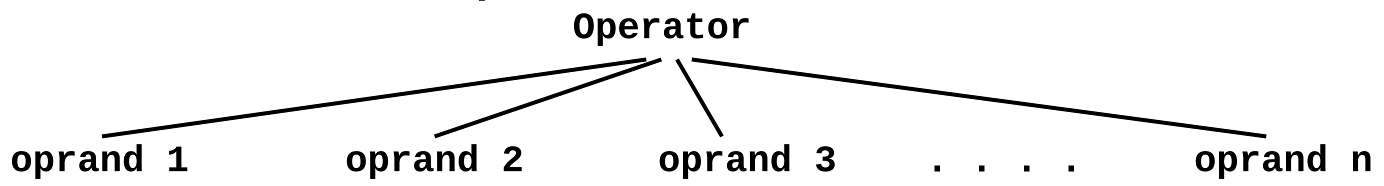


```
int k = 10;
void foo(int &i, int* &p) {
        i = 7;
        *p = i + *p;
        p = &k;
}
void bar() {
        int j = 2;
        int* q = &j;
        . . .
        foo(j, q);
        cout << j << *q;
        foo(j+3, q);
}
```

error in principle, but...

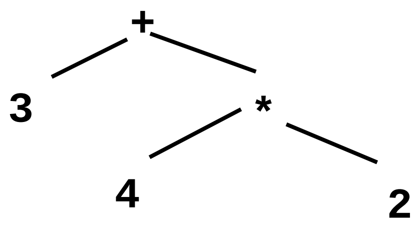
Evaluation order of operator/operands

- Given an expression, should we always evaluate all the operands before the operator?

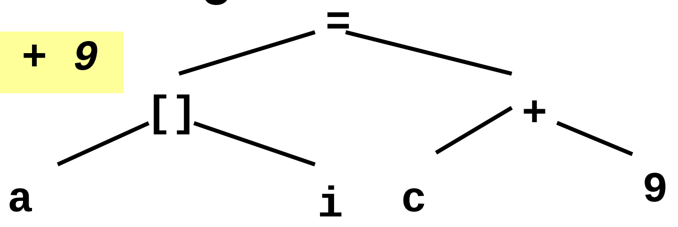


- *Yes!* - almost all kinds of expressions → **eager evaluation**

`3+4*2`

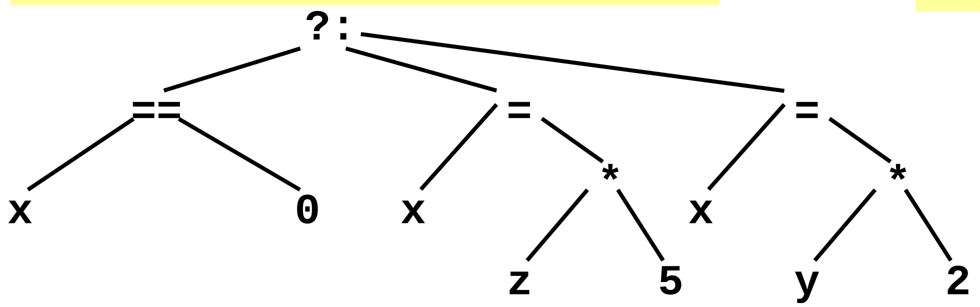


`a[i] = c + 9`

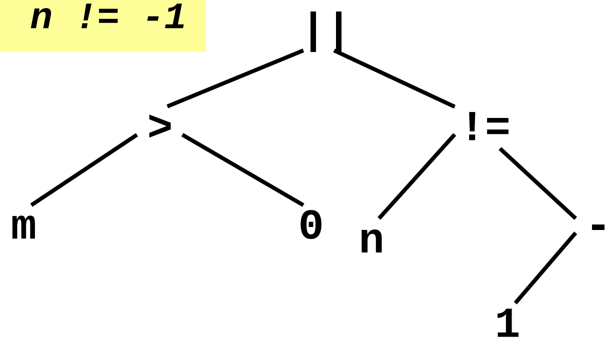


- *No!* - a few special expressions → **lazy evaluation**

`x == 0 ? x = z * 5 : x = y * 2`



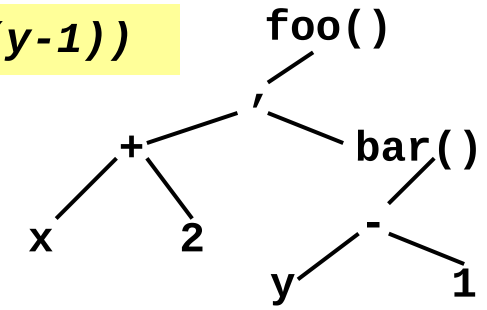
`m > 0 || n != -1`



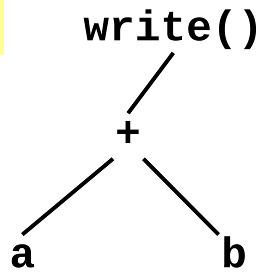
A procedure call as an expression

- A procedure call is an expression which consists of a procedure id (operator) and actual arguments (operands).

`foo(x+2,bar(y-1))`



`write a+b`



- Parameter passing models determine …
  1. how to map actual arguments to formal arguments.
  2. the evaluation order of operands and operators: *eager or lazy.*
- eager evaluation: call-by-value, cal-by-reference, call-by-sharing
- lazy evaluation: call-by-name, call-by-need

Why lazy evaluation?

- necessary to prevent infinite loops in recursion.

  // Suppose that C++ if-expression ?: does not use lazy evaluation, but eager evaluation.

  ```
  int fac(int n) { return (n == 0 ? 1 : n*fac(n - 1)); }
  ```

  // Now, what happens if we call **`fac(2)`***?*

  *fac(2) → (2==0 ? 1 : 2*fac(1))*
  → (2==0 ? 1 : n(1==0 ? 1 : 1*fac(0)))*
  → (2==0 ? 1 : n(1==0 ? 1 : 1*(0==0 ? 1 : 1*fac(-1))))*
  → (2==0 ? 1 : n(1==0 ? 1 : 1*(0==0 ? 1 : 1*(-1==0 ? 1 : 1*fac(-2)))))*

  . . .

  →*C++ allows lazy evaluation for if-expressions and relational operations.*

Traditional view of functions

- Ordinary data objects have *first-class values*.

  integers, real numbers, characters, strings …

- The traditional view of a function $f : \mathcal{D} \rightarrow \mathcal{R}$

  “a static piece of code for mapping values of first-class input values to first-class output values”

  → Such functions are said to be **first-order**.

- In many programming languages such as C/C++, Java, and Fortran, functions are first-order.

Higher-order functions

- In some other languages called *functional programming languages*, functions themselves are considered as first-class values so that they can be passed as inputs to and returned as outputs from other functions.
- *Higher-Order Function* (or *Functional*) → a function that takes functions as parameters or returns as outputs
  - HOFs are the essence of functional languages.
- HOFs are a powerful features of a functional language.
  - Treating functions as values increase the expressive power of a language. → functions handling functions : sums, derivatives
  - They help abstract out common control patterns, leading to very concise programs. → repetitive applications of similar tasks

HOFs taking functions as parameters

- The *summation* in mathematics (denoted by the notation Σ) is a HOF since it takes a function *f* as a parameter.

$$\sum_{j=1}^{m} f(j) = f(1) + f(2) + \ldots + f(m)$$

- The HOF can be represented with *type expressions*:

sum : *(α→β) × int × int→β* *when the function* **f : ** *α→β*

- Notation Σ makes a mathematical expression concise & brief by capturing the common patterns among the expression.

$$\sum_{i=1}^{l} \sum_{j=1}^{m} \sum_{k=1}^{n} f(i, j, k) = f(1,1,1) + \ldots + f(1,1,n) + f(1,2,1) + \ldots + f(l, m, n)$$

HOFs in non-functional languages

- Many non-functional languages support HOFs that take functions as parameters (Fortran, C++, . . .)
- They typically use *pointers* to support function as arguments.

```
C++        void (*efct) (string);                 // pointer to function
```

- They allow two operations on the function pointers.
  - *take the address of functions from the pointers*
  - *call the functions thru the addresses*

```
void error(string s) { . . . }
void announce(string t) { . . . }
efct = &error;                 // efct points to function error
efct(“Divided by zero”);       // call error  thru efct
efct = &announce;              // now, efct points to function announce
efct("No more assignment for 프방!");   // call  announce thru efct
```

→ *The pointers can be passed to other functions as arguments or results.*

- They are not as flexible as functional languages

```
int inc(int i) { . . . }
efct = &inc;               // illegal: since “inc” has different types of
                                     return value and input arguments of “efct”
```

Blocks

- A block
  - is a section of code that consists of *a set of declarations* and *a sequence of statements.*
  - provides its own environment or scope for variables.
  - allocates storage to variables local to a block when execution enters the block; the storage is deallocated when the block is exited.
  - is delimited by keywords or special characters
    - *procedure bodies* *ex: Fortran* → **`function . . . end`**
    - *begin/end* *ex: Algol* → **`begin . . end`**
    - *special characters* *ex: C* → **`{ . . . }`**
- The programming languages that allow programs to define blocks are called **block-structured** languages.
  - block-structured: Pascal, PL/I, Algol, C/C++, Scheme
  - non-block-structured: Cobol, Basic, Assembly

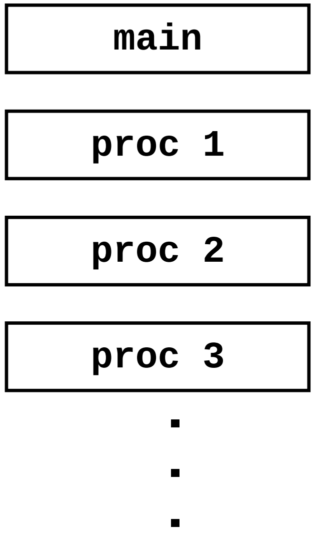
Terminology for blocks

- A block enclosed by other blocks is called a **nested** block. A block enclosing other blocks is called a **nesting** block.
- The variables **declared** (or **bound**) in a block are called **local** variables. Bindings of local variables of a block are visible only inside the block.
- The declarations of local variables in a block are **implicitly inherited** by nested blocks. It is not allowed to export a declaration to nesting blocks.
- **Non-local** variables of a block are those whose declarations are implicitly inherited from nesting blocks. They are not bound in the block, but bound in one of the nesting blocks.
- **Global** variables are those bound in the outermost nesting block; thereby, their bindings are visible in entire program, and they are accessible anywhere in a program.

Types of blocks

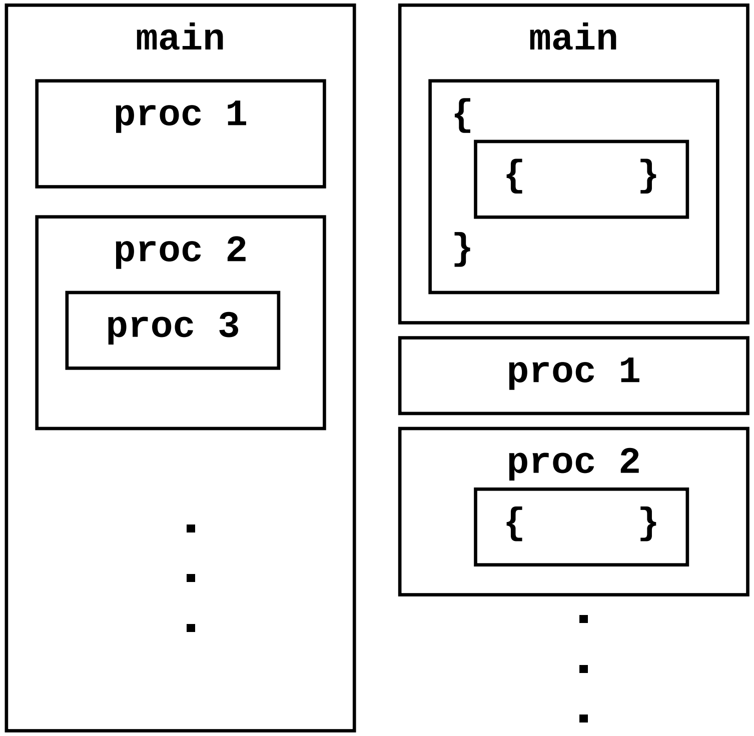
- Disjoint block structure
  - The body of a procedure is a block.
  - There is no nesting of blocks.

ex: Fortran

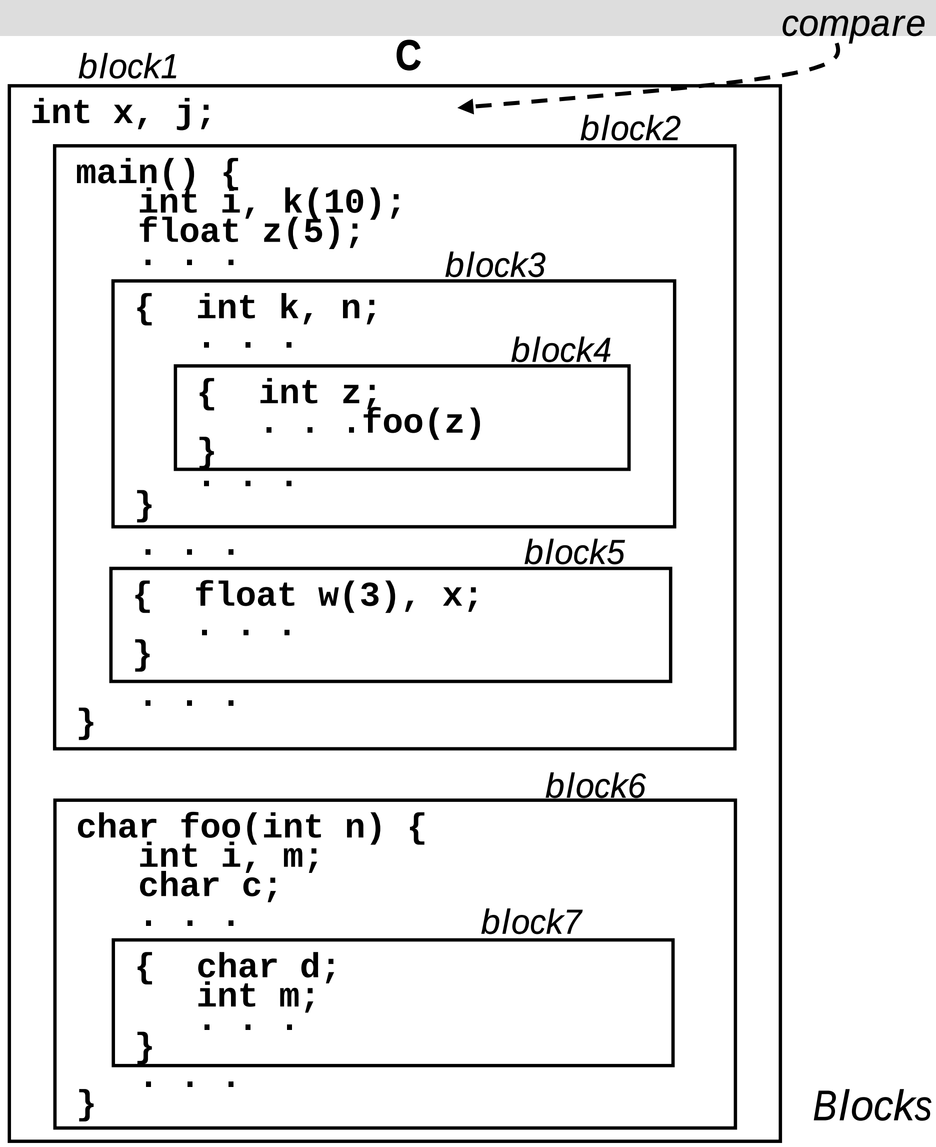


- Nested block structure
  - A block contains other blocks nested inside it.

ex: Pascal, Algol, C, Scheme



Nested block structure



```
int x, j;
main() {
   int i, k(10);
   float z(5);
   . . .
   {  int k, n;
      . . .
      {  int z;
         . . .foo(z)
      }
      . . .
   }
   . . .
   {  float w(3), x;
      . . .
   }
   . . .
}

char foo(int n) {
   int i, m;
   char c;
   . . .
   {  char d;
      int m;
      . . .
   }
   . . .
}
```

Blocks communicate with non-locals or parameters.

Advantages of block structure

- The block structure improves *readability* of programming by delimiting the scope of a binding, whiling nested blocks allow some bindings to be shared.
  - → The storage location of shared bindings can be used for communication between different blocks.
- It saves *storage* because the binding of a variable needs to be remembered only as long as the innermost nesting block is executed.
  - → Upon return of a block, the storage for the local variables can be deallocated unless a variable is static.
- It provides a mechanism for structuring programs, which may improve *writability* of programming.
  - → For instance, a given task is decomposed to several subtasks. A main procedure performs the whole task by distributing the subtasks to its sub-procedures within it.

Problems w/ globals in block structure

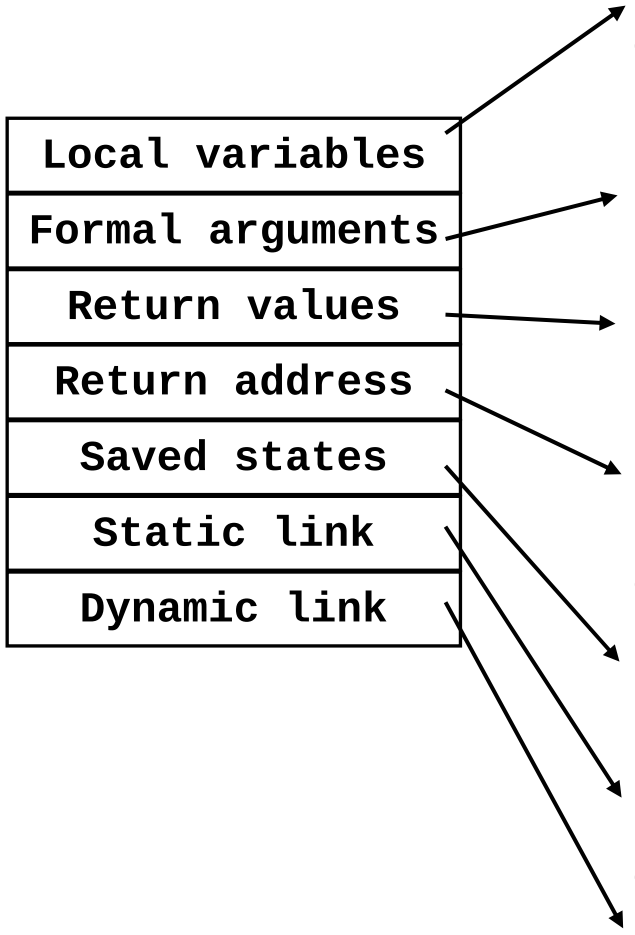
- It is generally difficult to exercise sharing bindings (or declarations) effectively.
- So, there is a tendency to move the declarations to the outermost block, which results in many global variables in a program. This exacerbates the following problems:
  - **Side-effects:** Debugging/maintaining programs are more difficult
  - **Indiscriminate accesses:** Due to implicit inheritance of bindings, all bindings in a block can be accessed by all nested blocks even when they are not supposed to. This results in less secure code.

    e.g.) typos in a nested block may not be recognized, and yet producing incorrect output.
  - **Screening problems:** The visibility of a declaration in a block can be accidentally lost when a variable with the same name is re-declared in an intervening nested block. This often happens when a program is large.

Implementation of block structure

- When a program block is invoked in a block-structured language, the body of the block is executed.
- Each execution of the body is called an **activation** of the block.
- Associated with each activation of a block is storage of the variables declared in the block and any additional information needed for the activation.
- The storage associated with an activation is called an **activation record** (AR).
- Components of an AR may vary depending on languages.

Activation record



The size of storage for local variables can be easily calculated at compile time if the language uses static type binding.

If the block is a procedure, additional storage is needed for arguments

If the block is a function which has to return the result, storage is allocated in the activation record.

In order to resume execution of the caller after the current block is exited, the address of the caller's code to return to should be kept.

Miscellaneous info about the caller site when this block was invoked.

It points to the activation record of its innermost enclosing block.

It points to the activation record of the caller.

Storage types for the implementation

- Static location
  - The addresses of static variables are fixed before run time.
  - Some storage is reserved for the variables at compile time.
- Stack
  - A stack is used to manage allocation/deallocation of ARs.
  - A language that holds ARs in a stack is said to obey a **stack-discipline**.

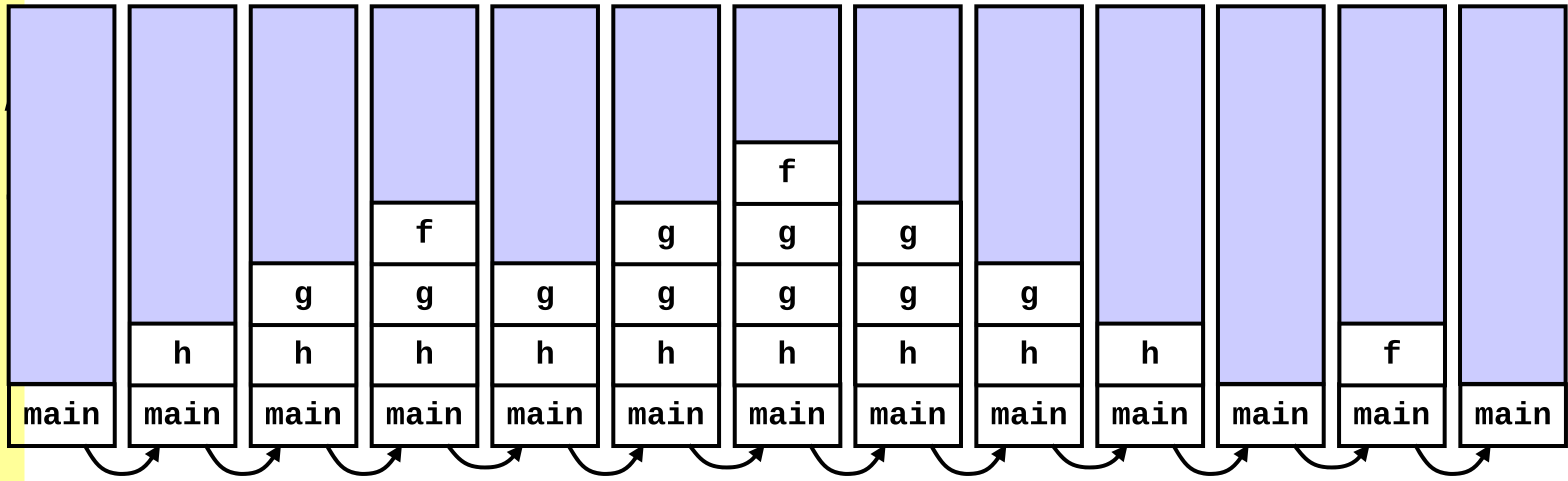
    → Most traditional imperative programming languages such as C and Pascal obey the discipline.
- Heap
  - A heap is more expensive but more flexible than a stack.

    → Typically, it is used for dynamic/pointer variables.
  - Functional languages use a heap for activation record allocation in order to treat functions/procedures as first-class citizens. (*why?*)
  - Also, some newer imperative programming languages such as Modula-3 and Oberon use a heap for AR allocation.

AR implemented in a stack

- Some observations on uses of ARs
  - Recursion has significant implications for language implementations of block structure. To support recursion, a separate AR has to be allocated for each procedure block invocation (*why?*)
  - When a block is exited, the life-time of the local variables ends. The AR is no longer needed after returning from the block.
- ARs can be efficiently managed with an **LIFO stack.**

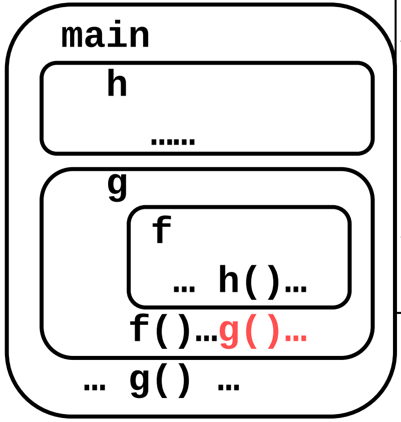
```
f() { . . .}
g() {
  static int i=0
  ..f()..
  if(i++==0) g()
}
h() {
  ..g()..
}
main() {
  ..h()..f()..
}
```



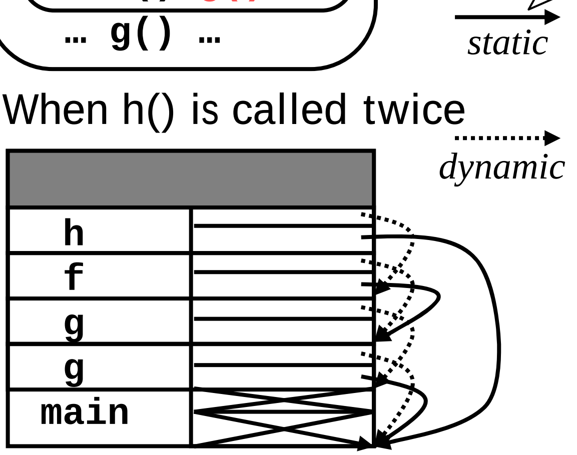
Static and dynamic links in a stack

- A dynamic link is used to restore access to the AR where the current block is activated: that is, *the AR of the caller of the block*.
- A static link in an AR of a block points to the AR of the next nesting block.
- Assume that X is a block whose AR is currently on the top of the stack when a new block Y is invoked. The dynamic and static link values of a new AR of Y are:

Static link는 코드의 block 구조에 의해 결정됨 반면에 dynamic link는 어떤 함수를 불렀을 때 결정됨



When h() is called twice

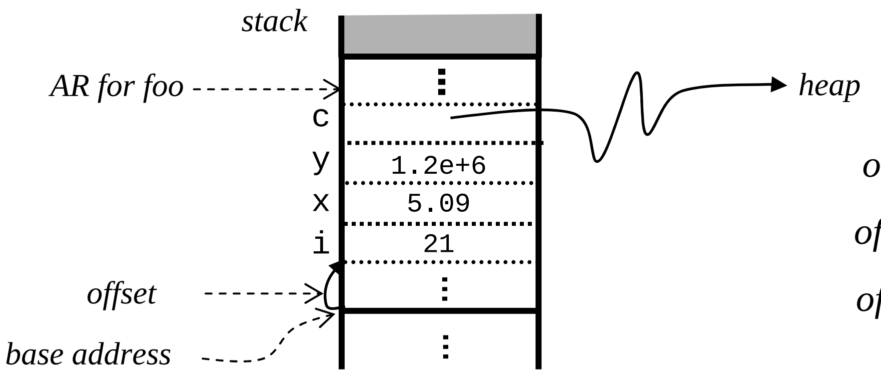


Access to (non)-local data in a stack

- Local accesses are fast:

  address of a local variable = address of base of current AR + an offset

```
foo (int i) {
   double x, y;
   char* c;
   …
}
```



offset(x) = *offset*(i) + 4

offset(y) = *offset*(x) + 8

offset(c) = *offset*(y) + 8

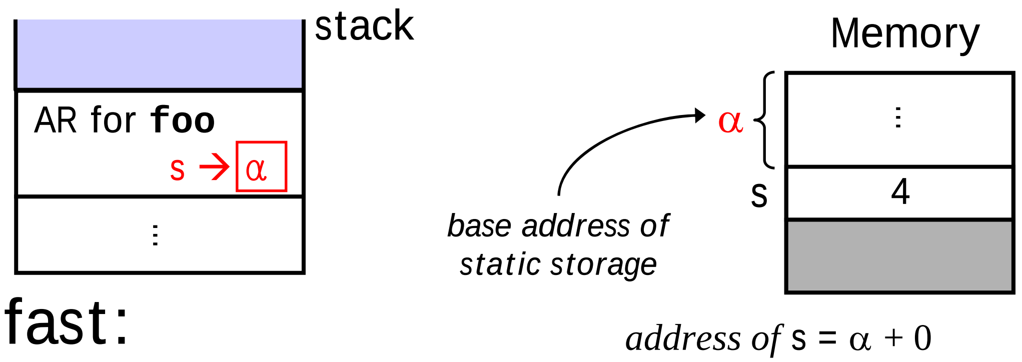
… = *offset*(c) + 4

- Nonlocal accesses are slower because they require extra pointers chasing following static or dynamic links.

Storage allocation for static variables

- A static variable declared in a block should retain its value between activations of the block.
  - If static variables are stored in ARs, this requirement cannot be met because the AR for each activation is removed after the activation is killed and, thereby, the values of all the variables in the AR is lost.
  - One solution is to store static variables in separate memory space with fixed addresses. For this, the compiler reserves some static storage space for static variables when it compiles the program.

```
foo (int i) {
    static int s = 0;
    …
    s++;
    …
}
```



- Access to static data is fast:

  address of static data = base address of static storage + offset

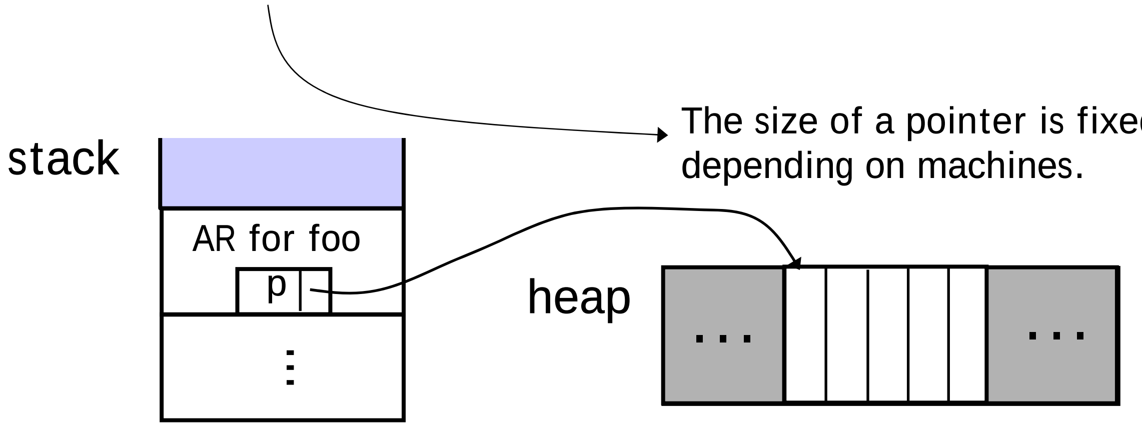
  → the base address and offsets can be determined at compile-time.

Heap allocation/deallocation

- If ARs are managed with a heap(*the area of memory used for dynamic memory allocation*), life times of the ARs need not be tied to the LIFO flow of control between activation.
- Even after control returns from a procedure block, an AR for the block can stay in storage. That is, the local variables are bound as long as needed.
  - Even in imperative languages, the size of an AR may not be determined when the AR is created because of **dynamic arrays**.
  - So, languages that use a stack for AR allocation still need a heap to allocate dynamic structures and to put pointers to them in the AR.

The size of a pointer is fixed depending on machines.

```
foo() {
   int* p;
    …
   p = new int[5];
    …
}
```



Allocation of dynamic arrays/lists

- Most languages support dynamic allocation primitives.

Pascal

```
type item = ˆlist;
     list = record
              head : integer;
              tail : item
           end;
var p : item;
begin
   new(p);
   p^.head = 3;
   p^.tail = nil;
```

// p = {3}

C++

```
list* p = new list;
p->head = 3;
p->tail = '\0';
```

→ The primitives allocate storage for a list/struct/record on a heap and store a pointer to it in **p** that is located in the AR on a stack.

How to deallocate dynamic data?

- use deallocation primitives: **`dispose`** (Pascal), **`delete`** (C++)

```
h() { ... int* p = f(); ... delete p; p = f(); … }
```

→ *versatile and flexible, but more difficult and less secure because the user must deallocate dynamic arrays explicitly.*

Common errors w/ dynamic allocation

- Explicit deallocation may cause **dangling pointers**.

```
void f () {
   char* c = d = "this is a list";
   delete c;
   ...
   cout << d;                          //Error! The string may no longer exist
}
```

- Mixing stack-allocated variables and pointers may cause errors.

```
float* g() {
   float* s = new float;
   float t;
   ...
   return &t;
}                           // s is garbage if it is not explicitly deallocated in g
void h() {
   float* r = g();          // no syntax error but r is a dangling pointer
   ...
}
```

Recursive structures

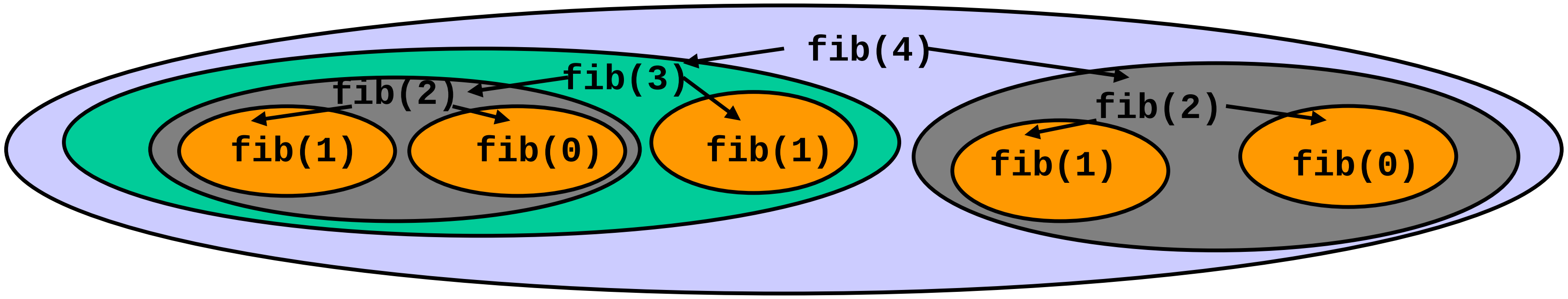
- A function *f* is **recursive** if it contains an application of *f* in its definition.

C++

```
 int fib(int n) {
   return ((n==0||n==1) ?
     1 : fib(n-1)+fib(n-2)));
}
```

→ *How does Scheme implement recursion?*

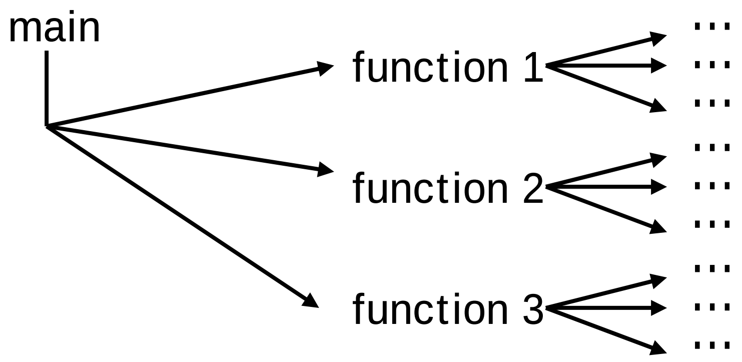
- Recursion simplifies programming by exploiting the *divide-and-conquer* method → *"divide a large problem into smaller ones"*



→ Can you rewrite the finobacci function without using recursion, and find how many lines you need for your version?

More facts about recursion

- Recursion allows users to implement their algorithms in the applicative style rather than the imperative style.



Applicative/Functional Programming

```
main {
        do this;
        do that;
        ...
        do this;
        do that;
}
```

Procedural/Imperative Programming

- Recursion can be expensive if not carefully used.

→ *Compare these two functions that compute the factorial*

compute the factorial with recursion

```
int fac(int n) {
   return (n==0 ? 1 : n*fac(n-1));
}
```

compute the factorial with iteration

```
int fac2(int n) {
  int p = 1;
   for (; n > 0; n--) p = n * p;
     return p;
}
```

Comparison of **`fac`** and **`fac2`**

Computation of **`fac(4)`**

```
fac(4)
4 * fac(3)
4 * (3 * fac(2))
4 * (3 * (2 * fac(1)))
4 * (3 * (2 * (1 * fac(0))))
4 * (3 * (2 * (1 * 1)))
4 * (3 * (2 * 1))
4 * (3 * 2))
4 * 6
24
```

function call

return

five function calls

four words to store the temporal data

Computation of **`fac2(4)`**

```
fac2(4)
p = 1  < -------  n = 4
p = 4  < -------  n = 3
p = 12 < -------  n = 2
p = 24 < -------  n = 1
                  n = 0
```

return p

one function call

one word to store the temporal data

The main problem with the recursive version is that **`fac`** needs more memory space and function calls as the problem size **n** increases. In contrast, **`fac2`** always needs only 1 function call and 1 word regardless of the value of n. → *Suppose* n *is 1000!*

Tail recursion

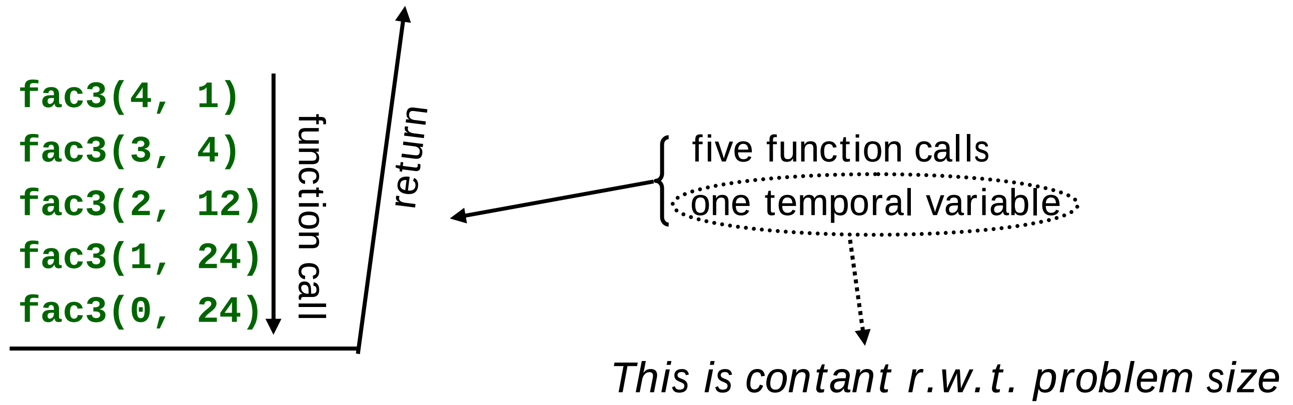
- A function f is **tail-recursive** if it is a recursive function that returns either *a value without needing recursion* or *the result of a recursive activation*.

Ex: `void fac3(int n, int& p) { if (n > 0) { p*=n; fac3(n-1,p);} }`

no ***int****!*

→ cf: Neither **`fib`** nor **`fac`** is tail-recursive.

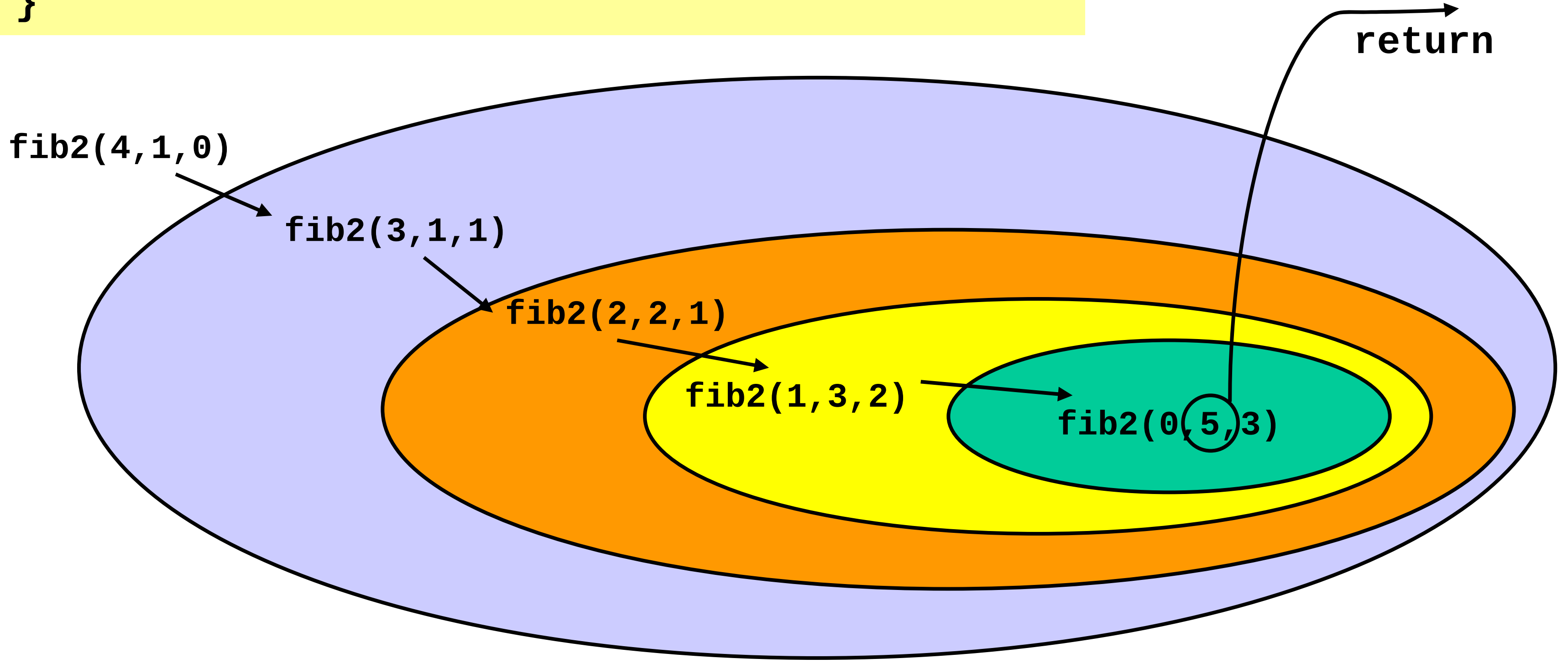
- What are tail-recursive functions so great about?
  - *While still taking advantage of recursion, they can execute programs in constant space.*



Application of tail recursion

- Write a tail-recursive version of fib.

```
void fib2(int n, int& l, int& r) {
     if (n > 0) { l+=r; r=l-r; fib2(n-1,l,r); }
}
```



return
fib2(4,1,0)
fib2(3,1,1)
fib2(2,2,1)
fib2(1,3,2)
fib2(0,5,3)