Lecture 8: Scope, Symbol Table, & Runtime Stack

COMP 524 Programming Language Concepts
Stephen Olivier
February 10, 2008

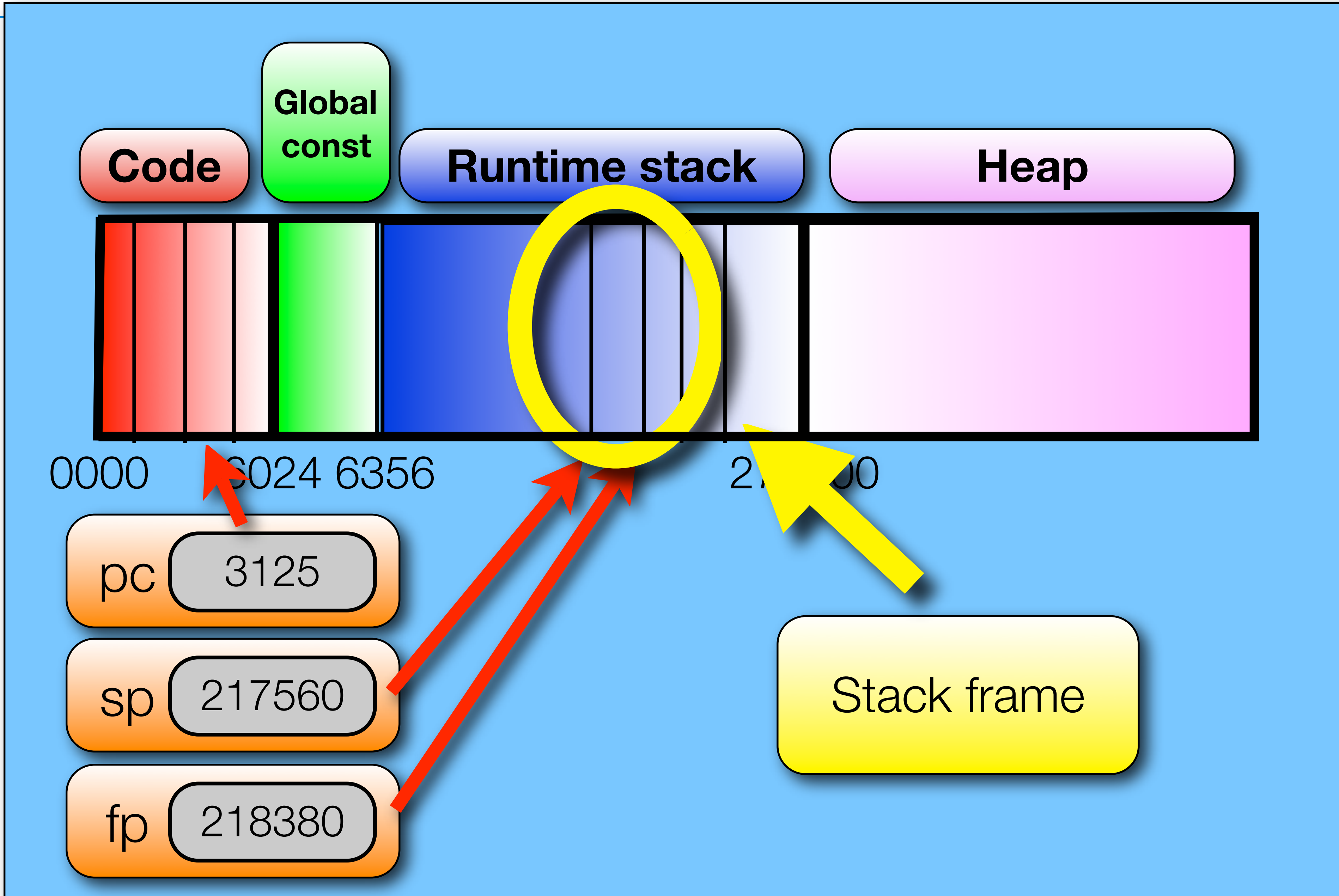
Based on notes by A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts

The University of North Carolina at Chapel Hill

Goal of Lecture

- Discuss scoping!

Sample Memory Layout



Scope

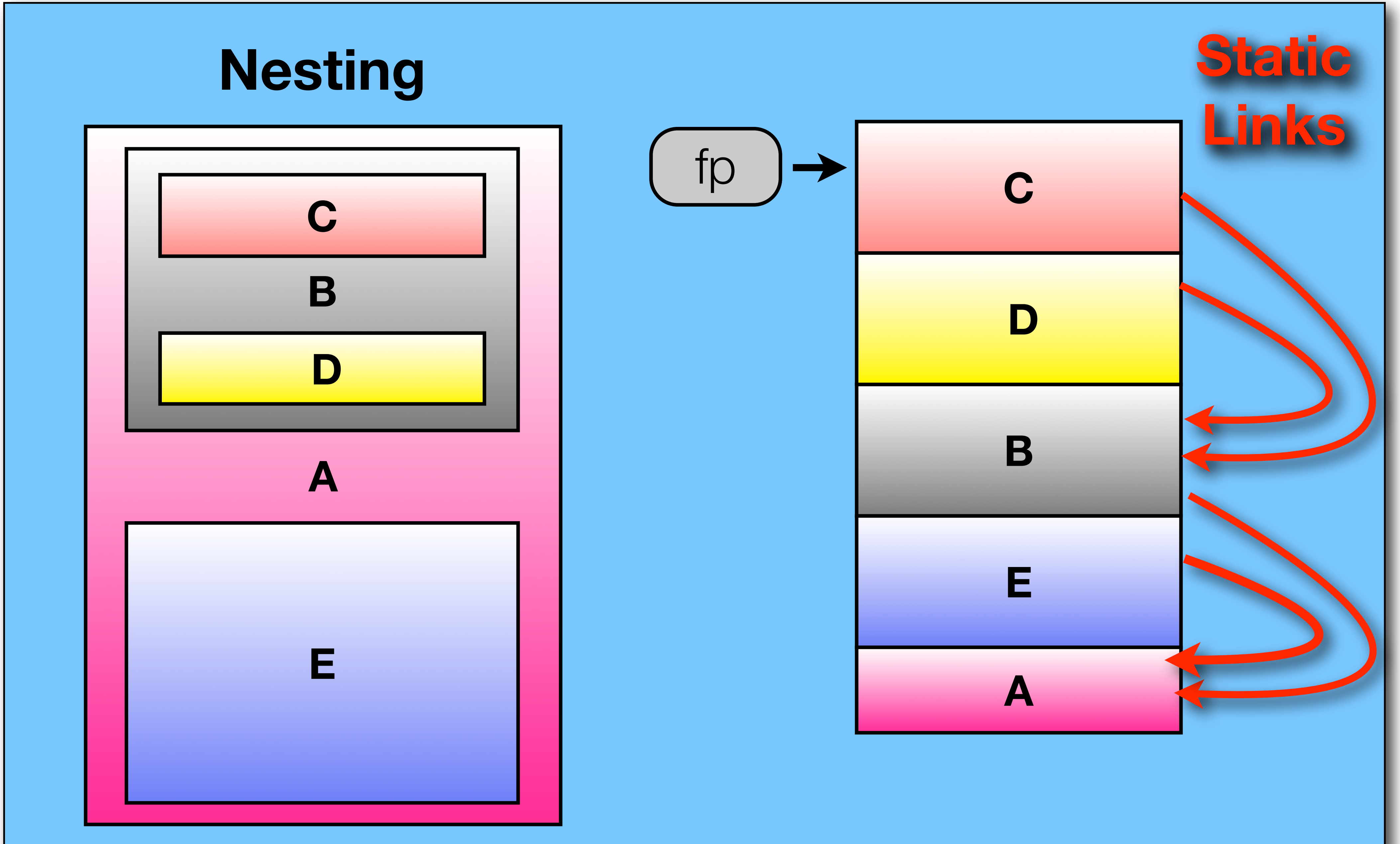
- Scope is the textual region of a program in which binding is active.
- Programming languages implement
  - Static Scoping (or lexical): Active bindings are determined using the text of the program at compile time
    - Most recent scan of the program from top to bottom
    - Closes nested subroutine rule.
  - Dynamic Scoping: active bindings are determined by the flow of execution at run time
- Current active binding called Referencing environment.

Nest Subroutines

- Nest subroutines are able to access parameters and local variables of the surrounding scope

```
procedure P1(A1);
  var X : real;
  procedure P2(A2);
    procedure P3(A3);
      X = 2;
    end
  end
end
```

Nested Subroutines--Determining Scope



Dynamic Scope

- Bindings between name and objects depend on the flow of control at run time
  - the current binding is the one found most recently during execution.

```
a:int;

procedure first()
  a:=1

procedure second()
  a:int
  first()

a:=2
if read_int() > 0
  second()
else
  first()
write_int(a)
```

Perl and Dynamic Scope

- Perl allows dynamic scope.
- If not declared otherwise, variables are **dynamically created**, **global**, and **persistent**.
  - **Dynamic** creation: Variables appear when referenced.
  - **Global**: Variables can be referenced in any and all code written
  - **Persistent**: Variables stay around until end of execution.

Perl and Dynamic Scope

```
$a = 1;
aFunc();
$d = $b+$c; #$d = 1 + 3 = 4

sub aFunc{
  $b=$a;
  $c=3;
}
```

Perl and Dynamic Scope

```
aFunc();

sub bFunc{
  $c = $a;
  #$c = 1 if run in or after aFunc
}

sub aFunc{
   $a = 1;
   bFunc();
}
```

Perl

- “my $abc”
  - Makes variable statically scoped
  - Only available to this subroutine
  - Not available to called subroutines or originating subroutines
  - Destroyed when execution exits the block it is in.

Perl and Dynamic Scope (my)

```
$a = 1;
aFunc();
$d = $b+$c; #$d = undefined + undefined = 0

sub aFunc{
  my($b, $c)
  $b=$a;
  $c=3;
}
```

Perl and Dynamic Scope (my)

```
aFunc();
sub bFunc{
  $c = $a;
  #$c is undefined no matter if it
  #is run at or in bFunc
}

sub aFunc{
  my($a);
  $a = 1;
  bFunc();
}
```

Perl

- “local $var”
  - Makes variable dynamically scoped
  - “Temporary global”
  - Available to called subroutines, but not available to originating Subroutines
  - Destroyed when execution exits current block

Perl and Dynamic Scope (local)

```
$a = 1;
aFunc();
$d = $b+$c; #$d = undefined + undefined = 0

sub aFunc{
  local($b, $c)
  $b=$a;
  $c=3;
}
```

Perl and Dynamic Scope (local)

```perl
aFunc();
sub bFunc{
  $c = $a;
  #$c is undefined if bFunc is run
  #after aFunc, but is 1 if run
  #in $aFunc
}
sub aFunc{
  local($a);
  $a = 1;
  bFunc();
}
```

Lifetime vs Scope

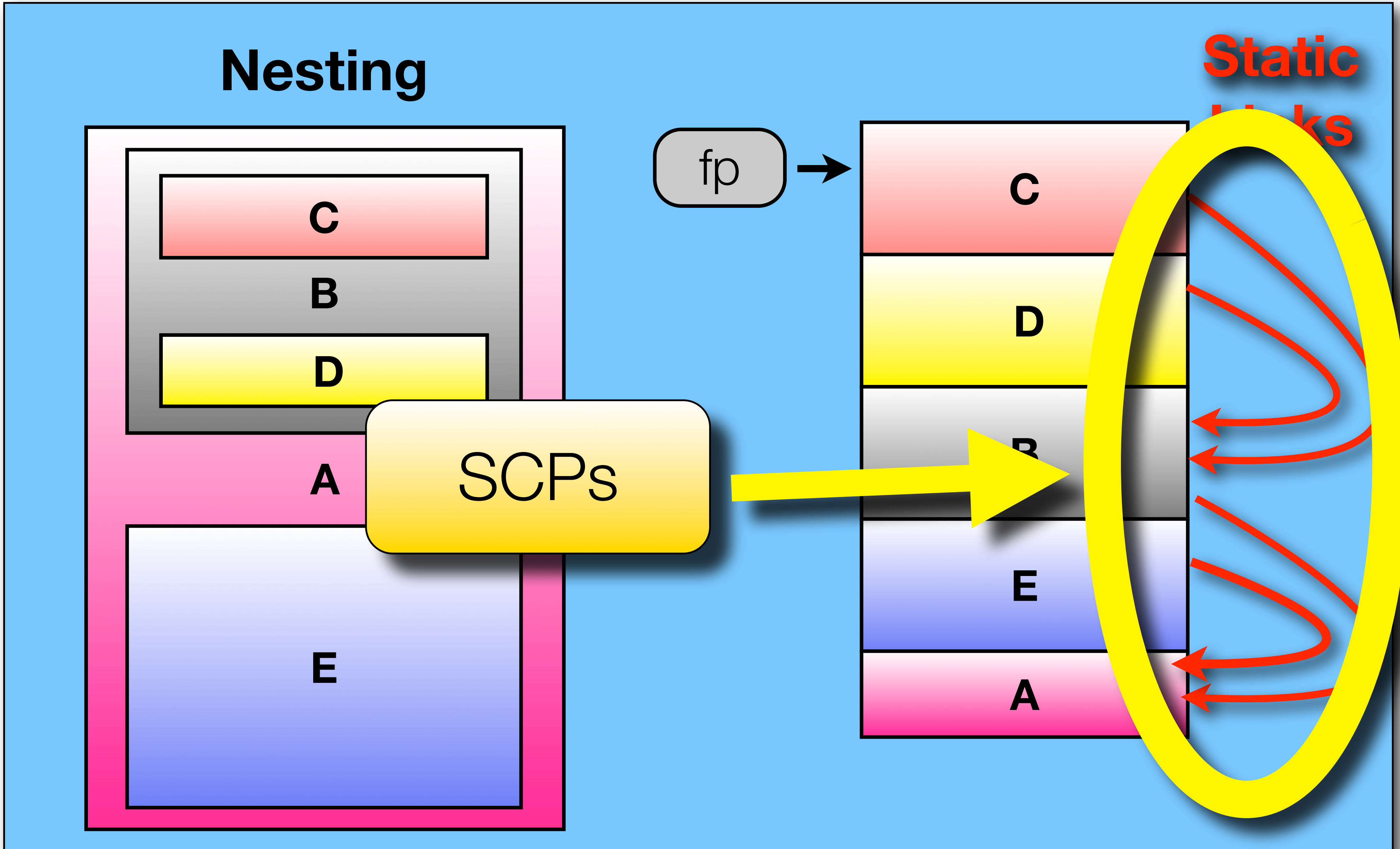
- Some objects exist only when scope is active
- ... however, this is not always the case.

```
class foo{
  public static int sum = 0;
  void vooDo(){ sum ++; }
}

//Where is sum? Its not active but it exists.
g1 = new foo;
g1.vooDo();
```

Static Chain

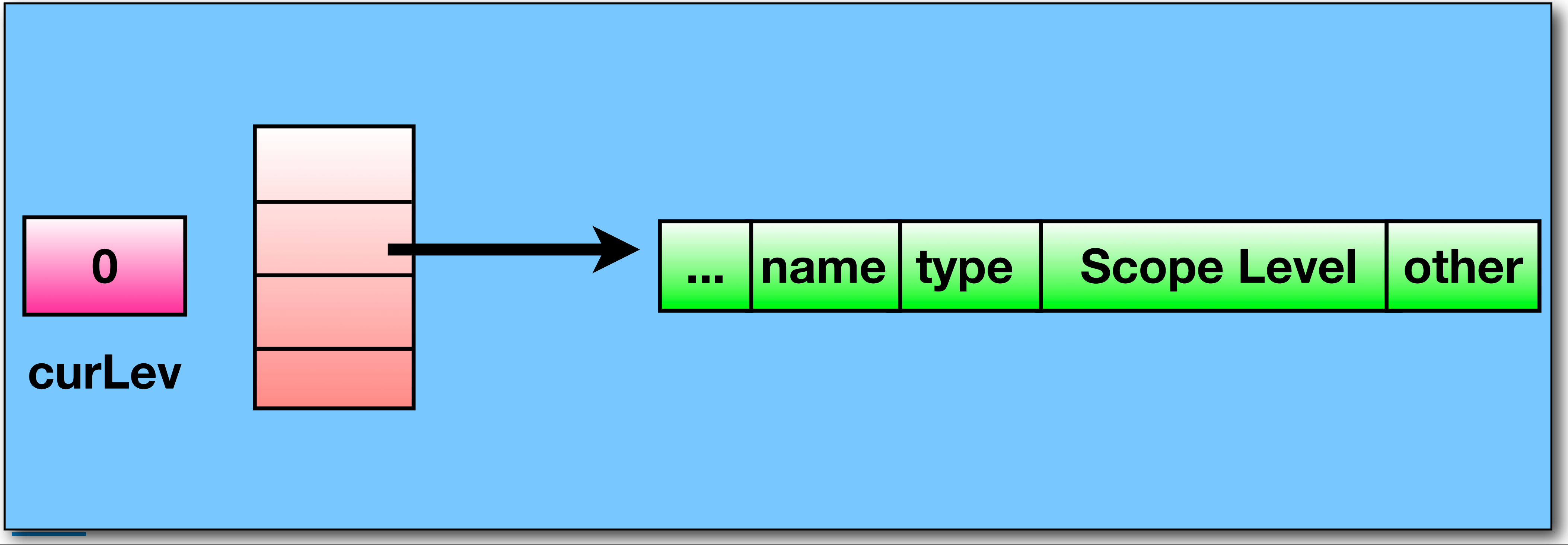
- For finding non-local bindings at run-time
- Each frame contains a **static chain pointer** (**SCP**), a pointer to the most recent frame on the next lexical level out.

Nested Subroutines--Determining Scope



Symbol Table

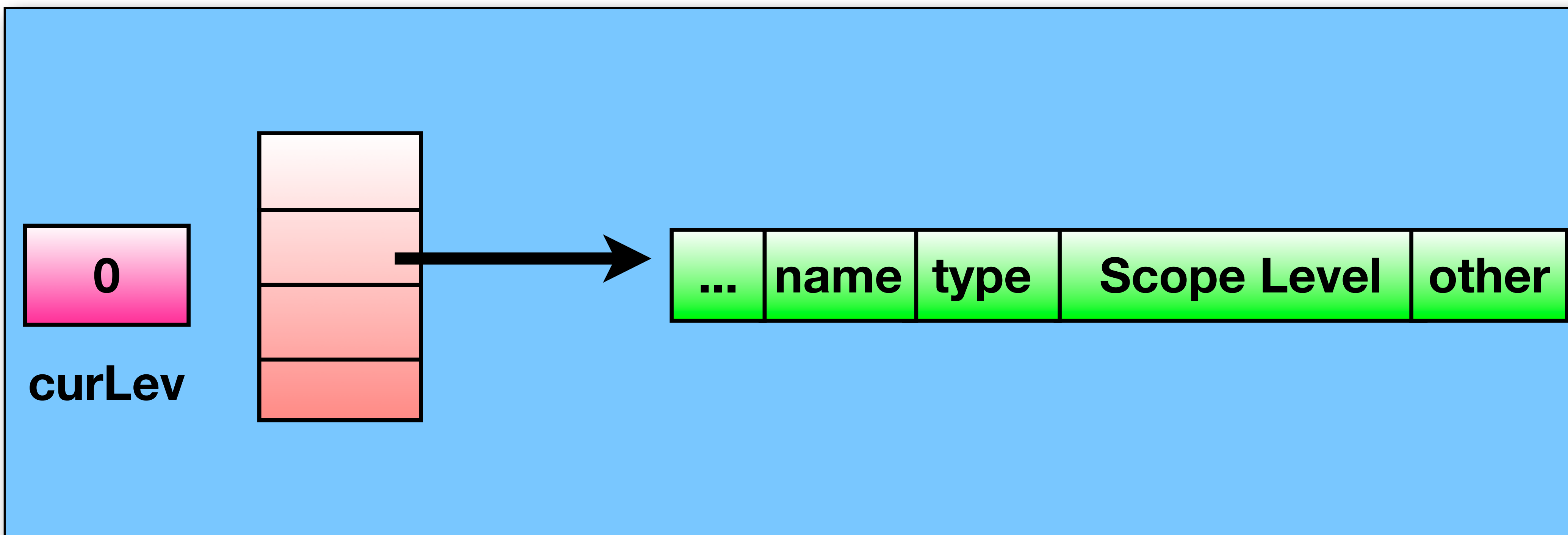
- In statically scoped languages, compilers keep track of names using a data structure called a symbol table.
- The symbol table might be retained after compiling and made available at runtime (e.g., for debugging)



Symbol Table

- In static ... s keep track of names ... ymbol table.
- The sy... r compiling and made a ... ugging)

Maps names to info about objects.
Just like a **hash** in Perl!

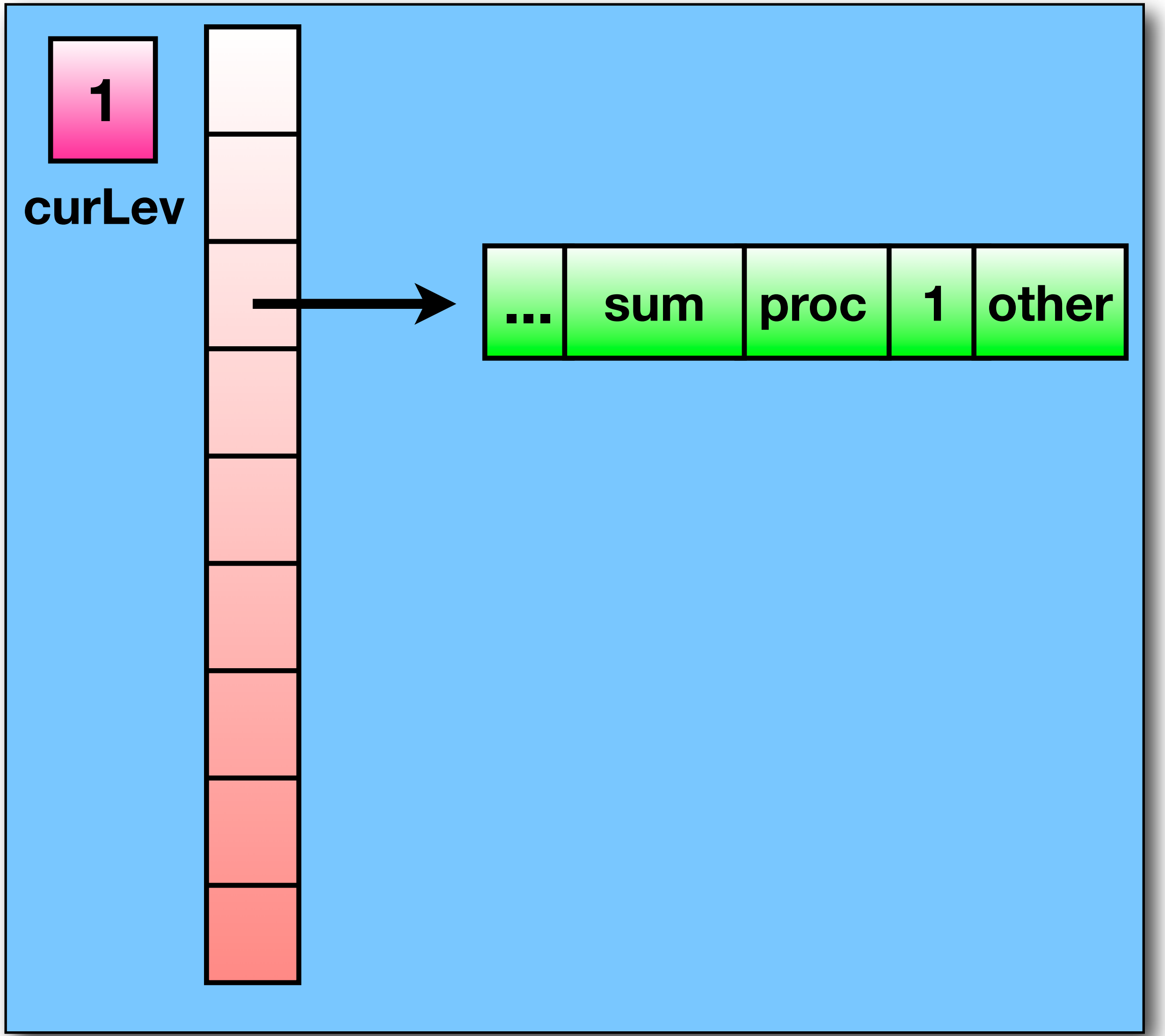


Symbol Table: Simplified

- Seeing a new name during parsing makes several things happen.
    1. **addName** to the ST
    2. Is the name a new scope? **addScope**
        a. **New Scopes**: Procedure/method names, nested blocks....
    3. Nesting Level (Lexical level) is counted as parsing goes
    4. Each Name is stored with its **scope number**
- Compiler keeps track of the lexical level in force when a name is declared
- Multiple entries are made for a name in the hash table. .. A new inner delectation “hides” an outer declaration.

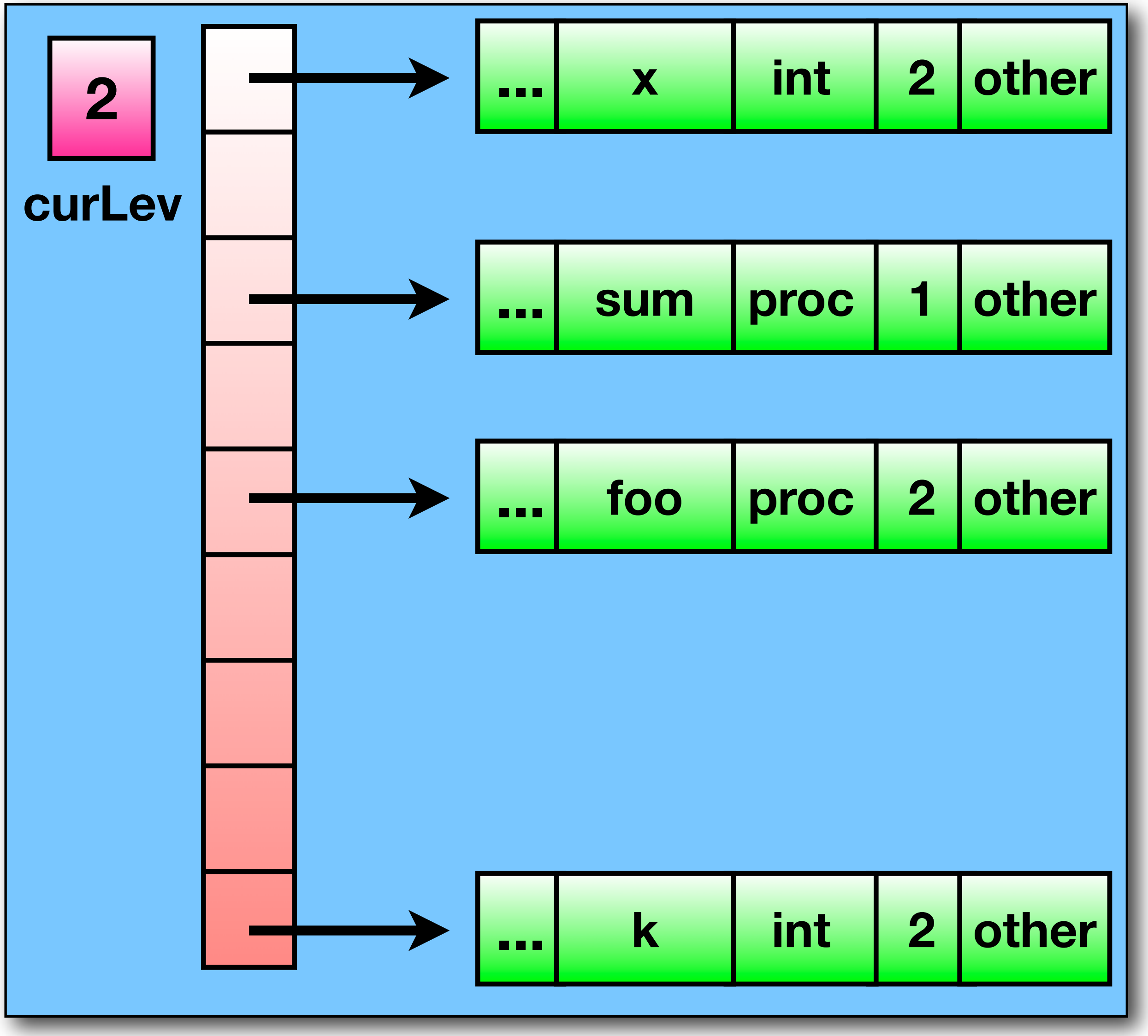
Sample Program

```
proc sum(int x){
  int k = 0;

  proc foo(){
    real sum = 0.0;

    proc inDo(int sum){
      return sum * x;
    }
  }
}
```



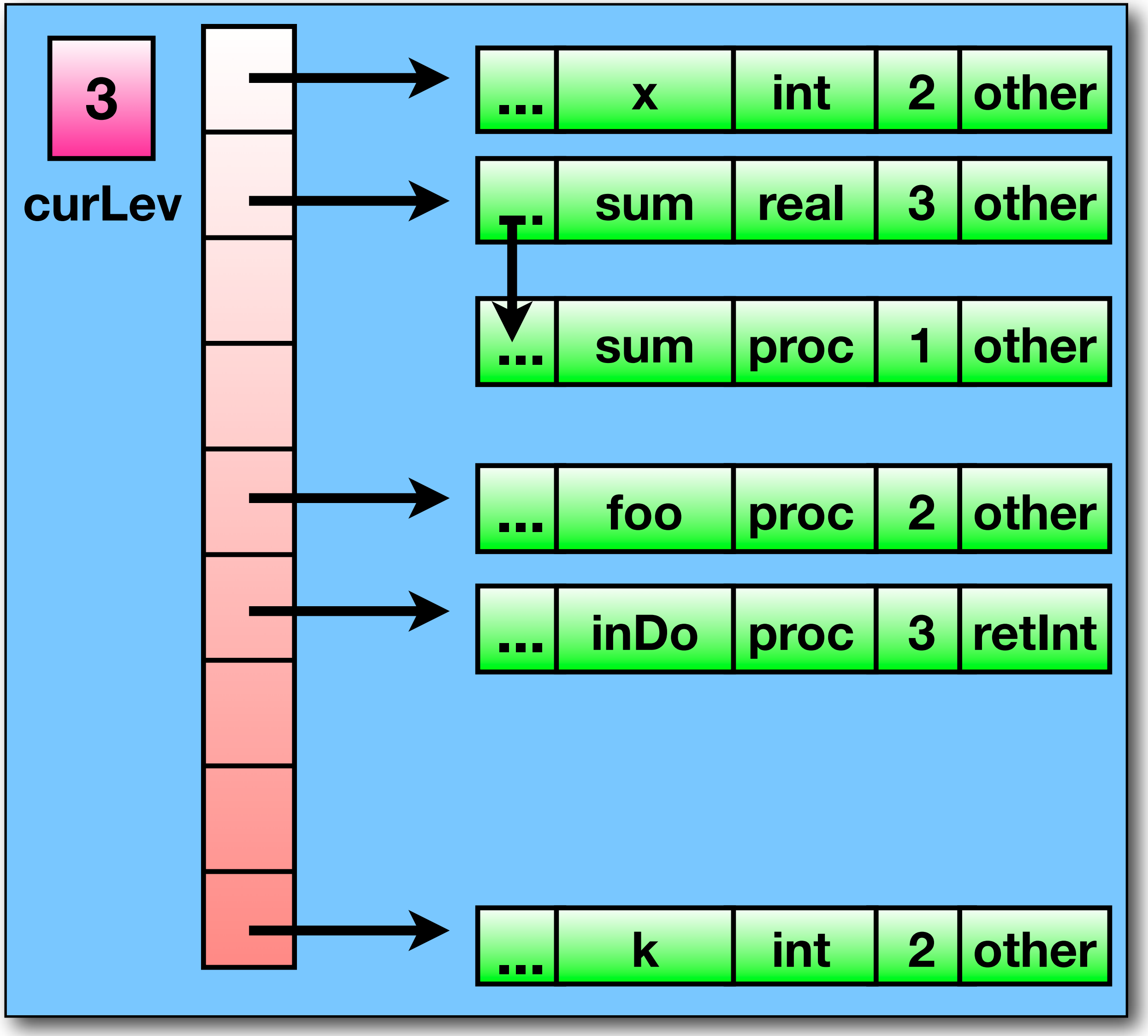
Sample Program

```
proc sum(int x){
  int k = 0;

  proc foo(){
    real sum = 0.0;

    proc inDo(int sum){
      return sum * x;
    }
  }
}
```



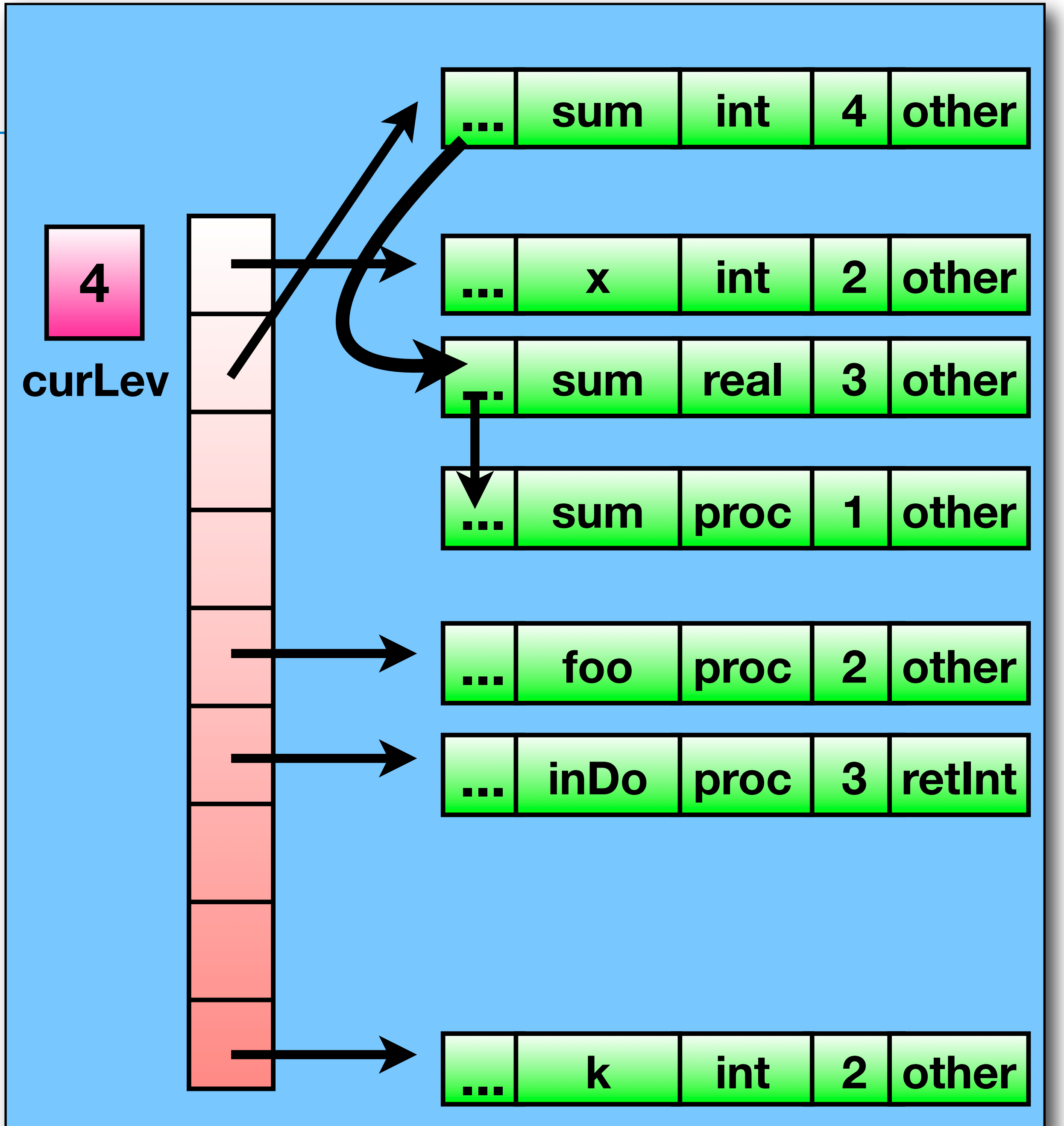
Sample Program

```
proc sum(int x){
  int k = 0;

  proc foo(){
    real sum = 0.0;

    proc inDo(int sum){
      return sum * x;
    }
  }
}
```



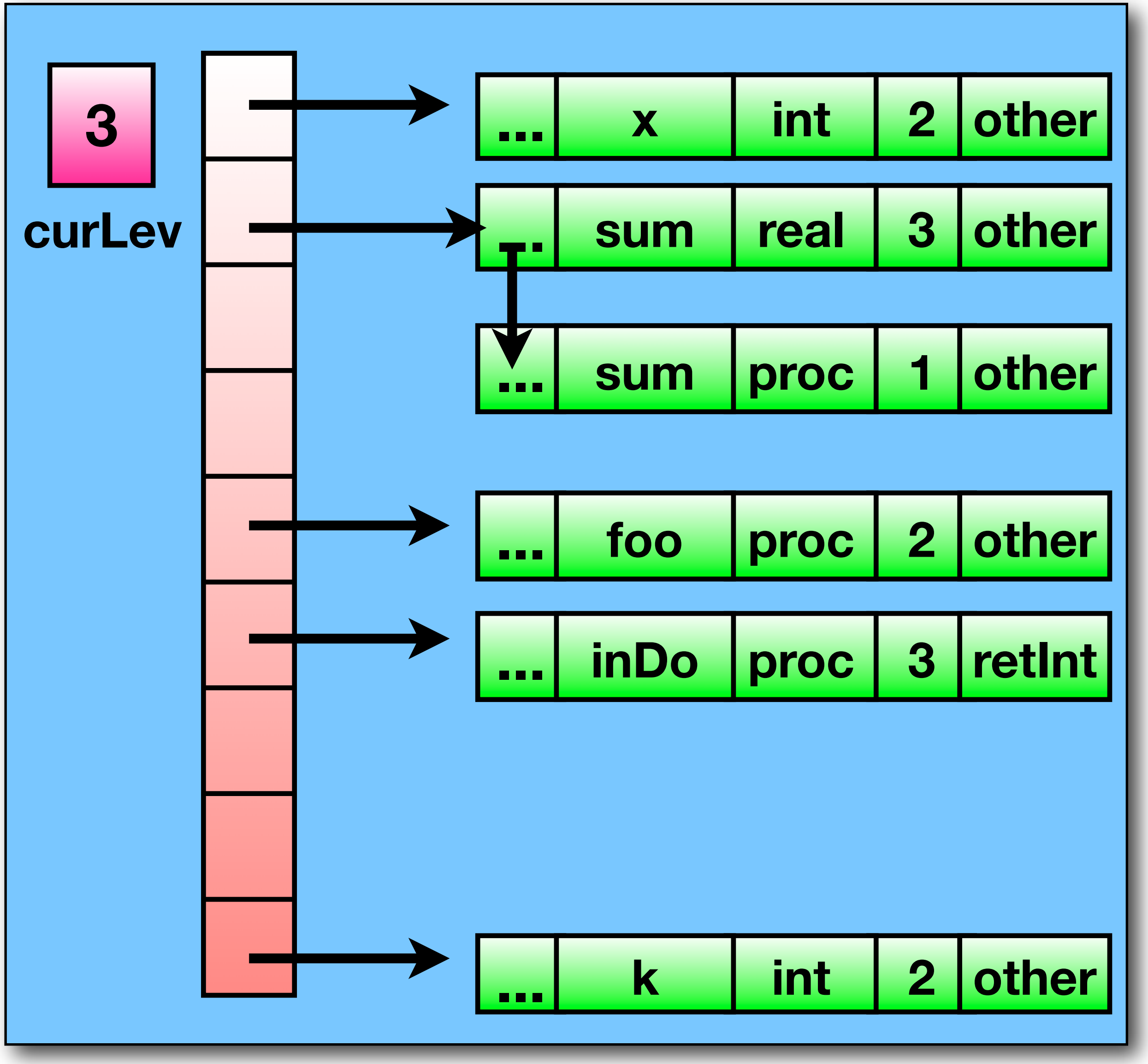
Sample Program

```
proc sum(int x){
  int k = 0;

  proc foo(){
    real sum = 0.0;

    proc inDo(int sum){
      return sum * x;
    }
  }
}
```



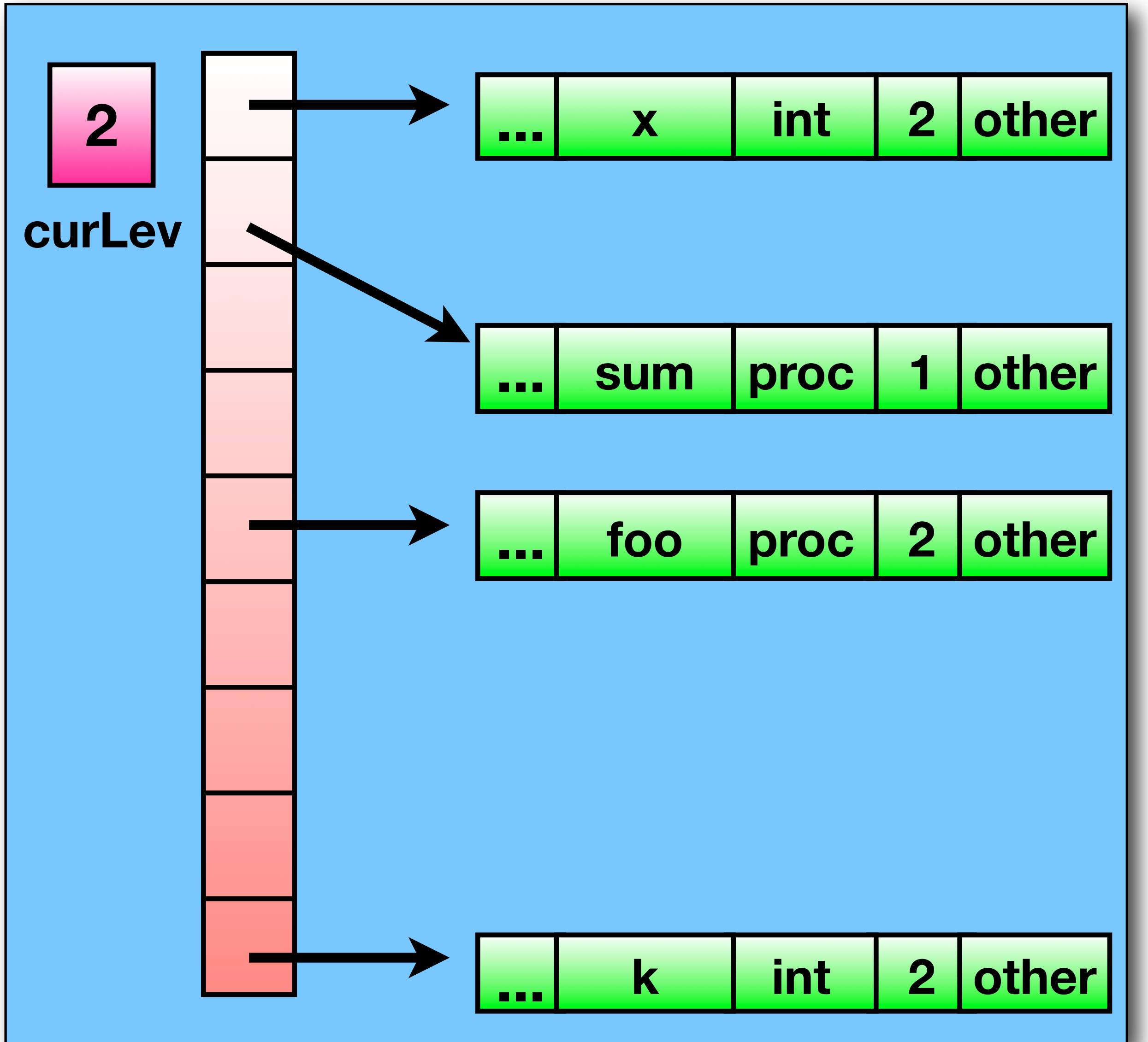
Sample Program

```
proc sum(int x){
  int k = 0;

  proc foo(){
    real sum = 0.0;

    proc inDo(int sum){
      return sum * x;
    }
  }
}
```



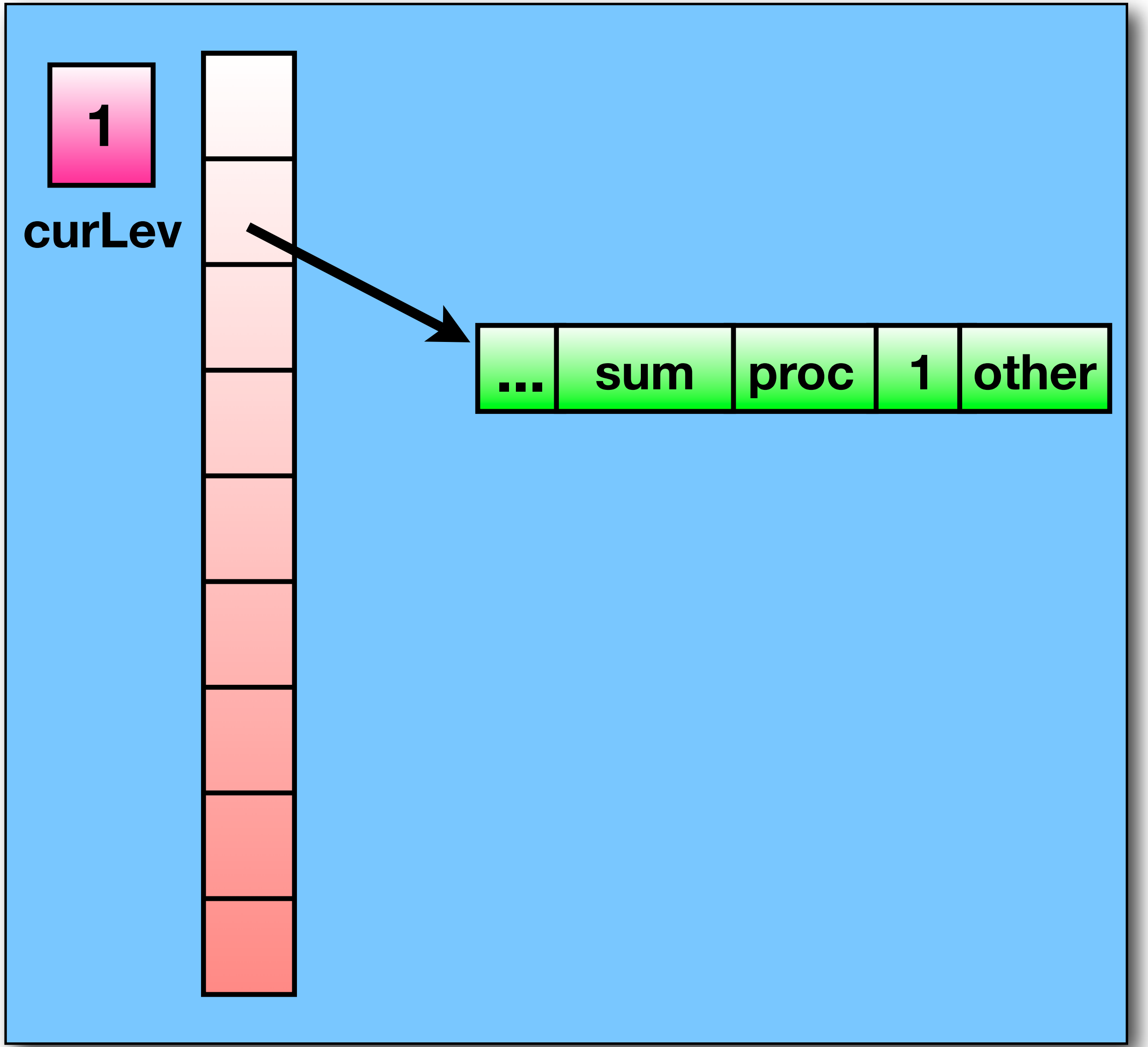
Sample Program

```
proc sum(int x){
  int k = 0;

  proc foo(){
    real sum = 0.0;

    proc inDo(int sum){
      return sum * x;
    }
  }
}
```



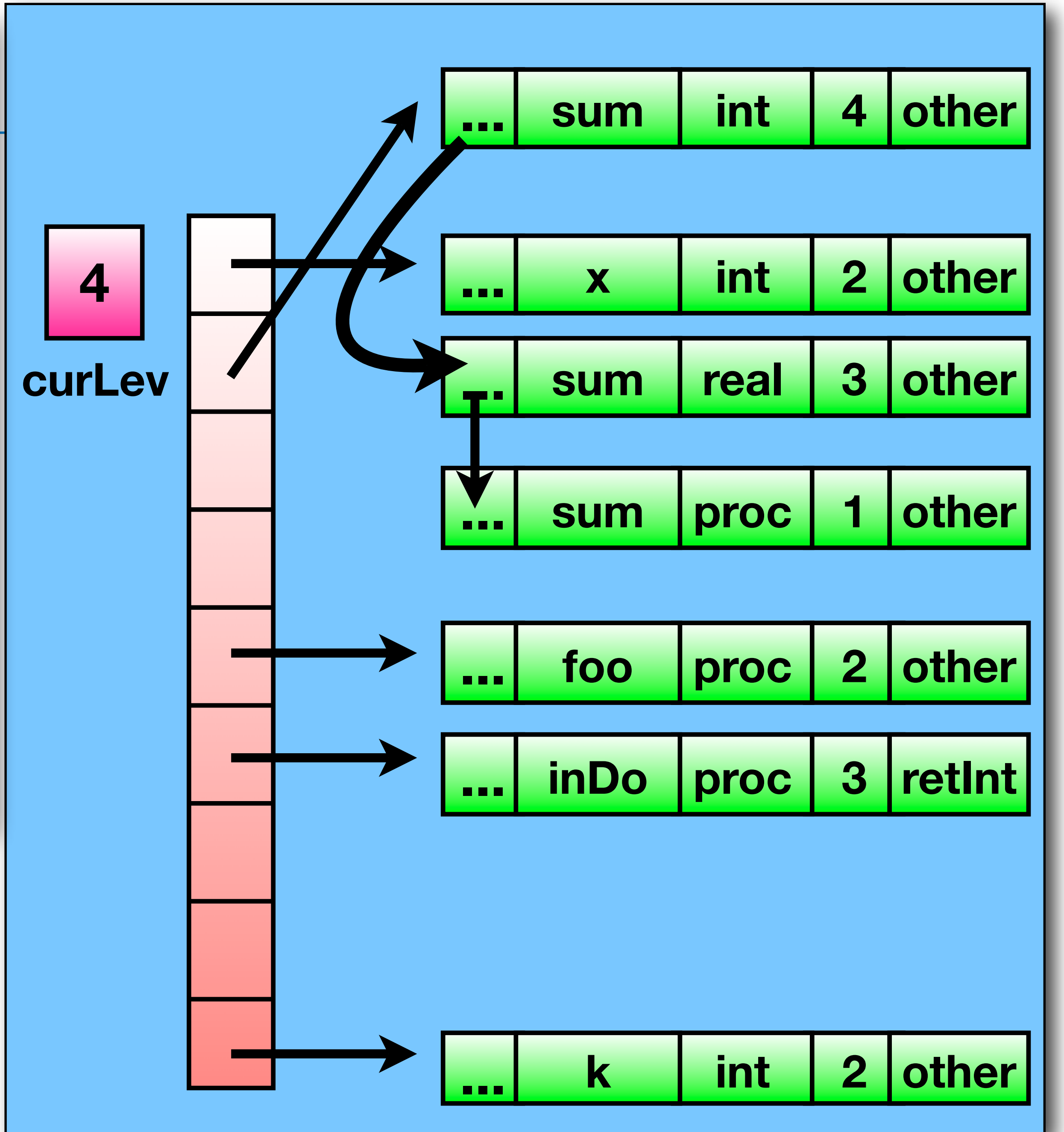
Sample Program

```
proc sum(int x){
  int k = 0;

  proc foo(){
    real sum = 0.0;

    proc inDo(int sum){
      return sum * x;
    }
  }
}
```

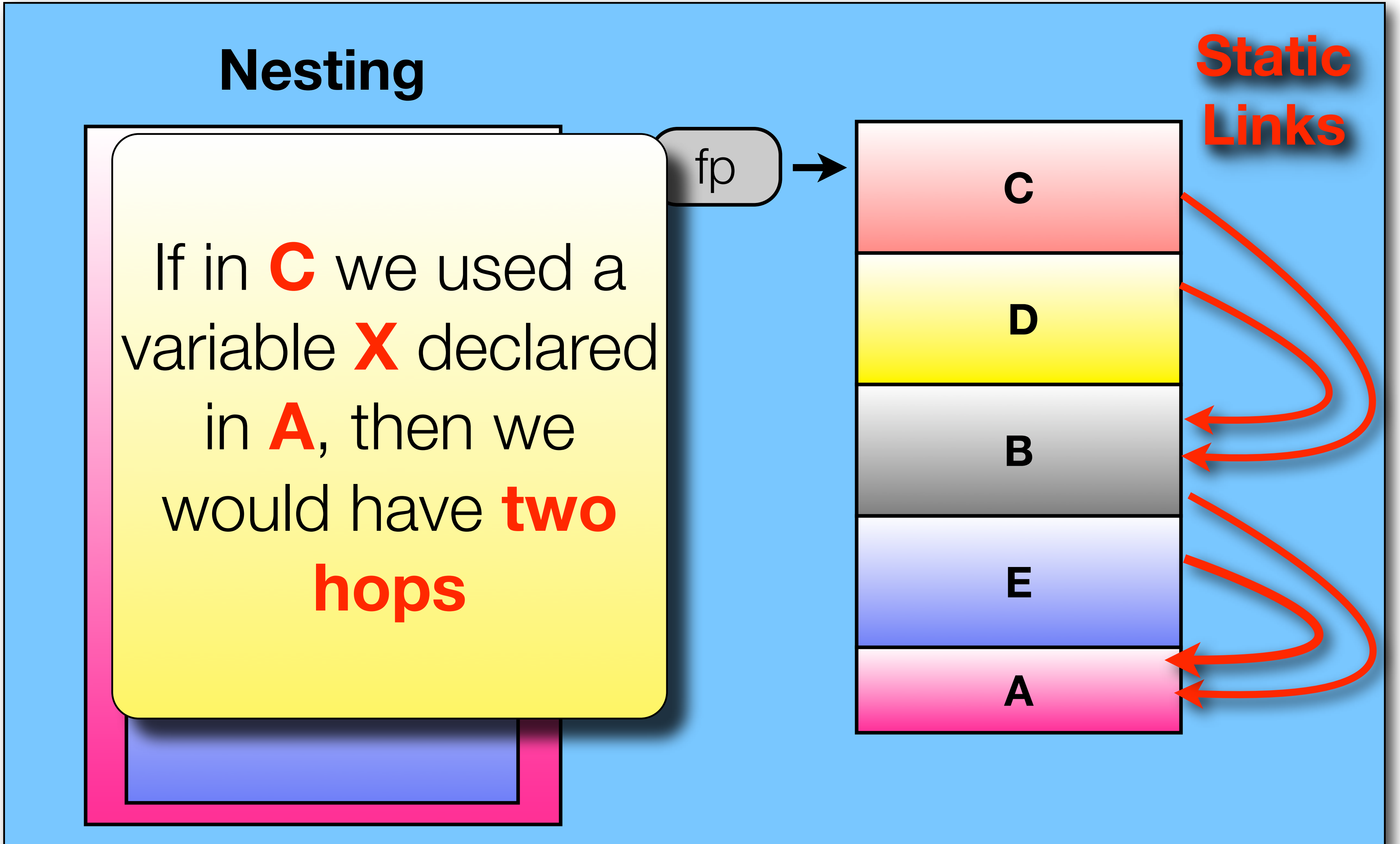


The scope tells you how many static chain hops you need to make, *i.e.*, **Current scope minus your scope**.

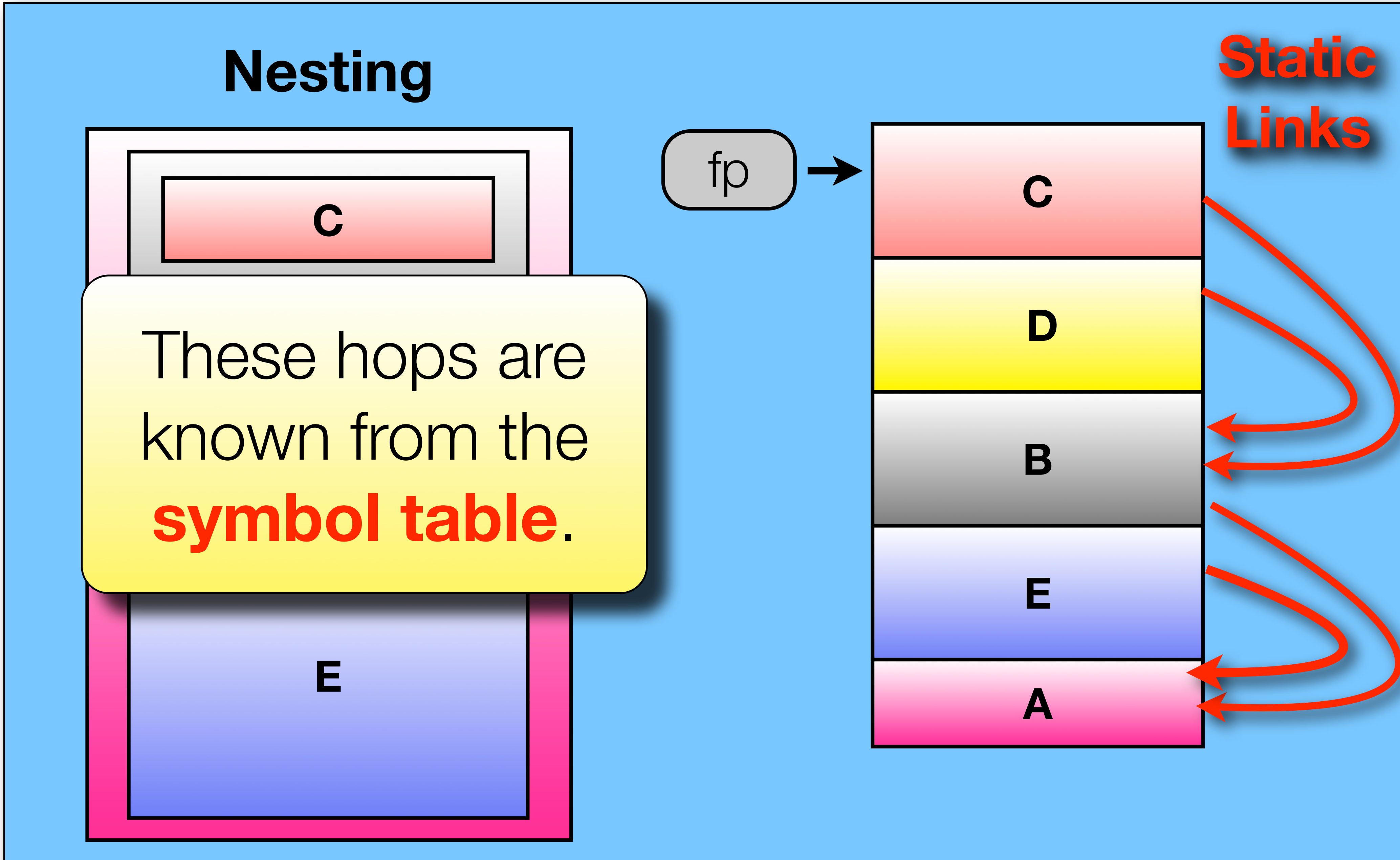
}
}



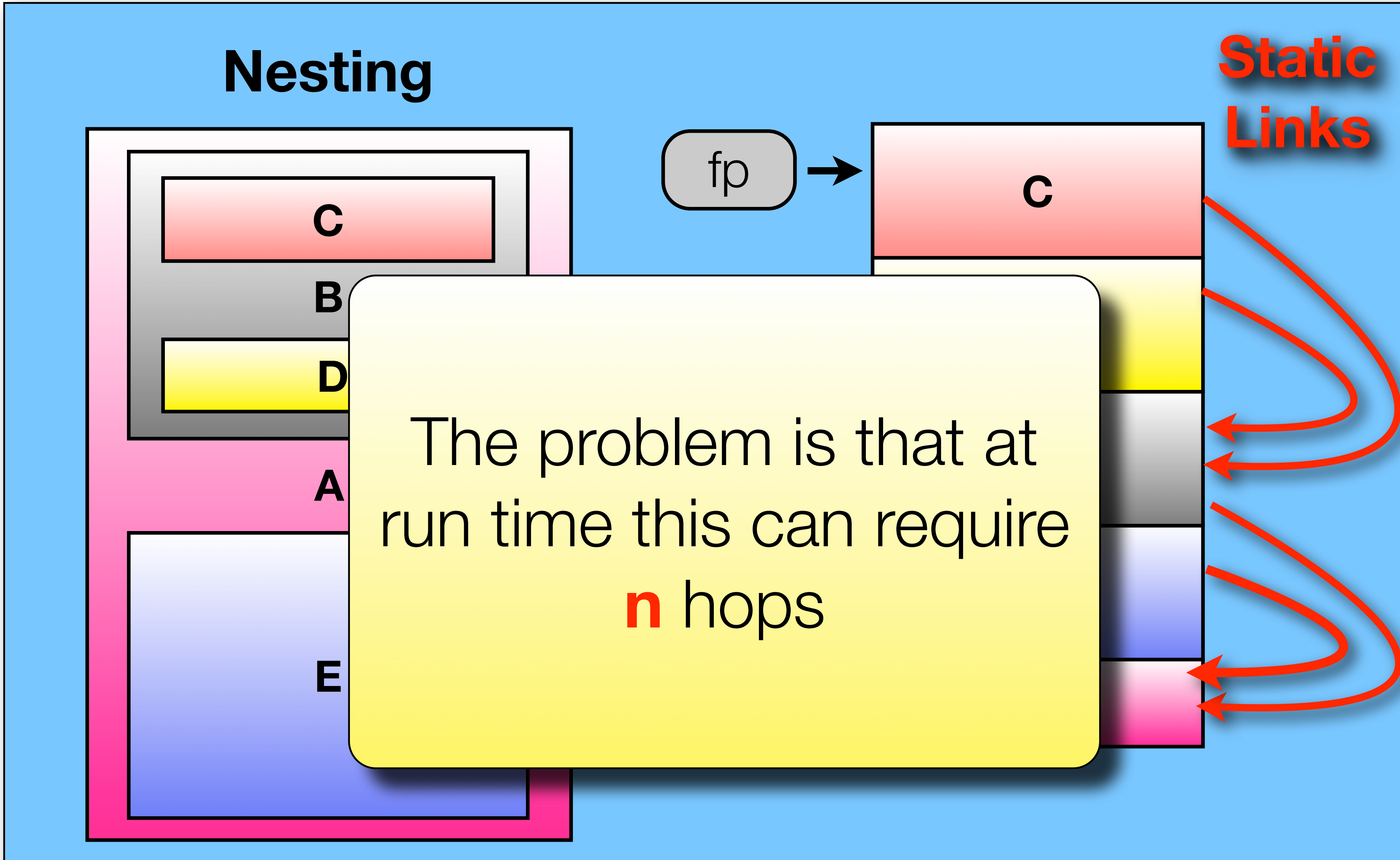
Nested Subroutines--Determining Scope



Nested Subroutines--Determining Scope

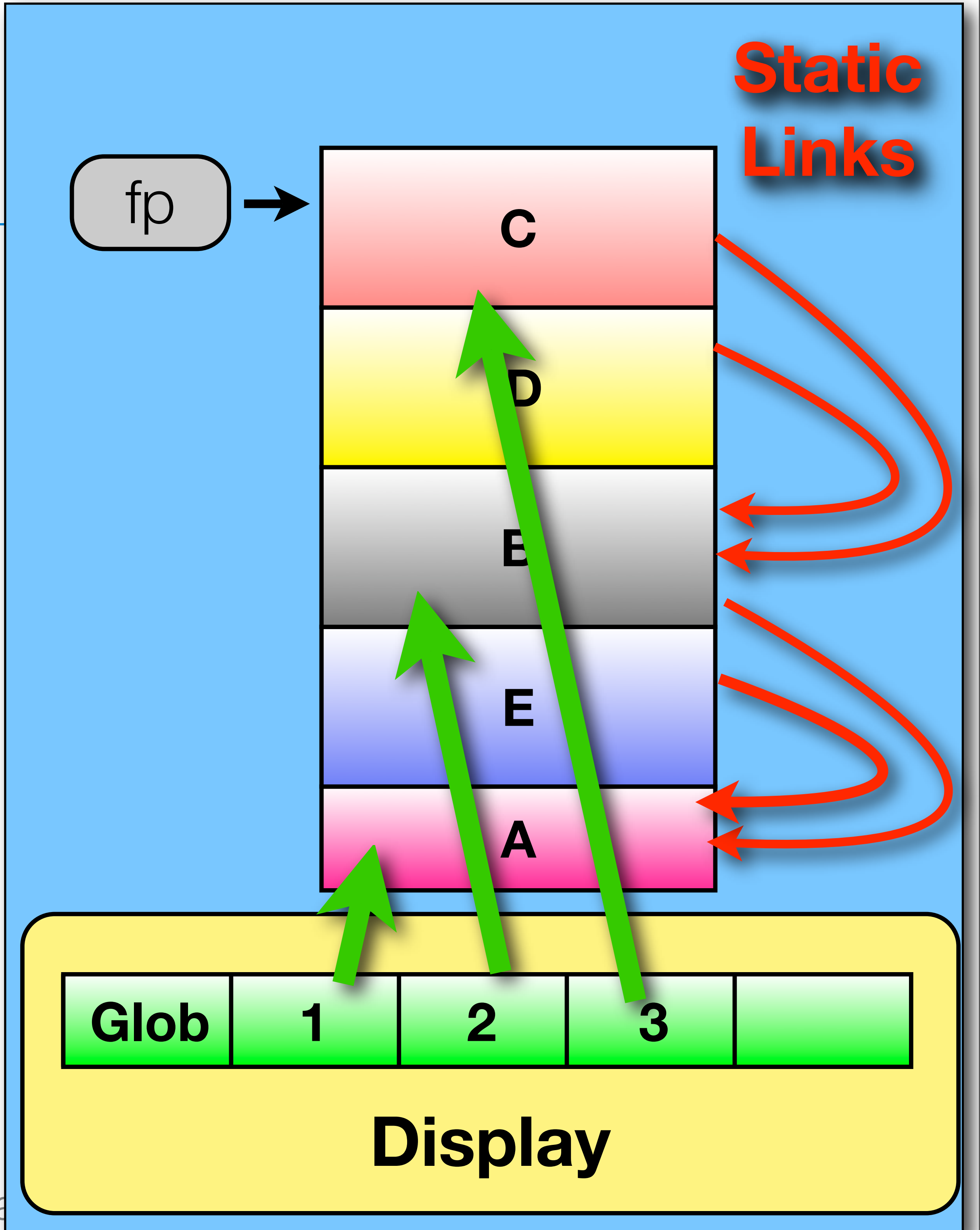


Nested Subroutines--Determining Scope



Display

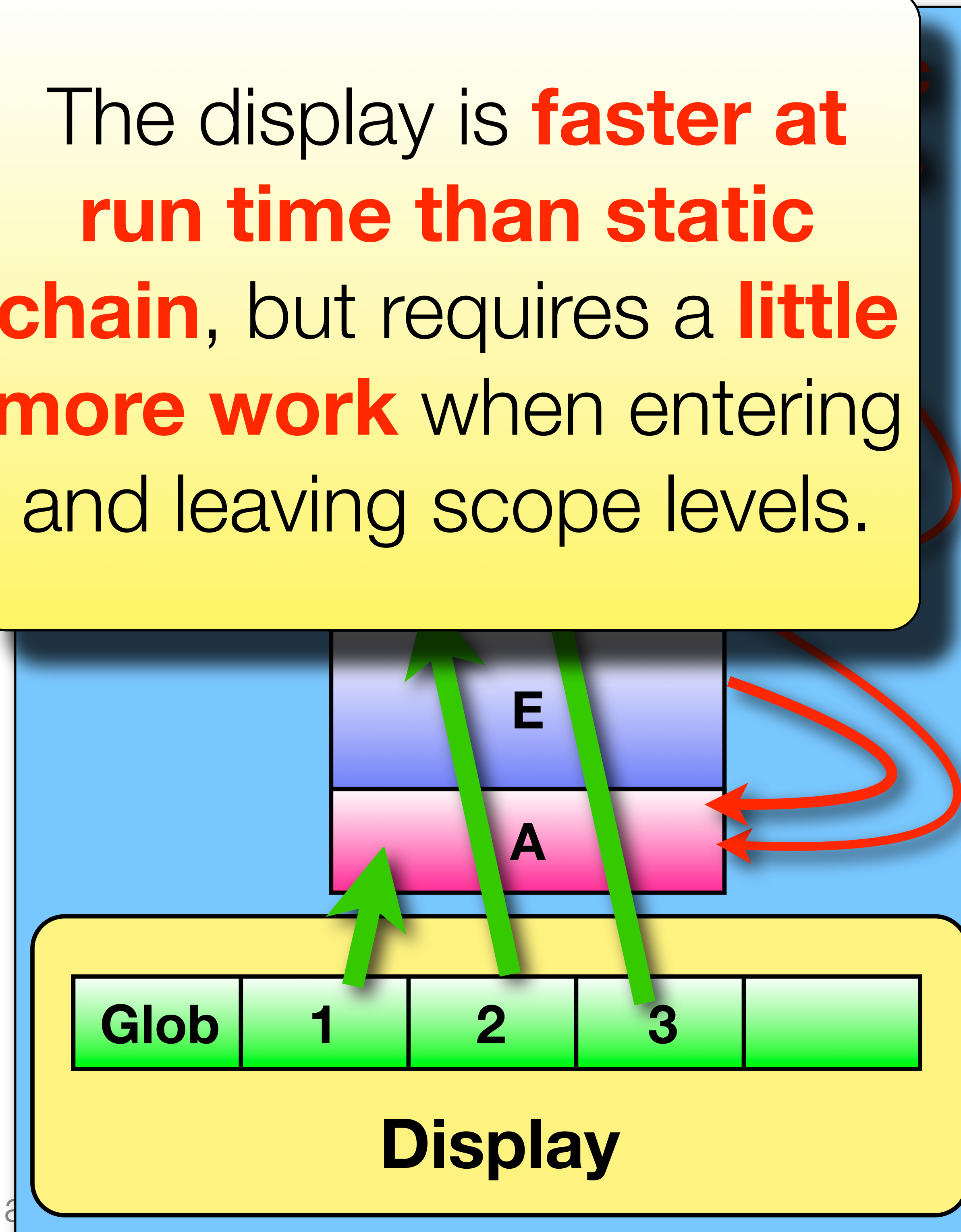
- The **display** is a small array that replaces the static chain, where the **jth element of the display contains a pointer to the jth nesting level**.



Display

- The **display** is a small array that replaces the static chain, where the **jth element of the display contains a pointer to the jth nesting level**.

The display is **faster at run time than static chain**, but requires a **little more work** when entering and leaving scope levels.



Dynamic Chain

- **Dynamic Chain Pointer** (**DCP**)
- Shows sequence of stack frames in dynamic (call) order.
- Allows implementation of dynamic scope.

We'll talk about this later!

Static Scope: Modules

- Many Modern languages are **more complicated in their scope rules than PASCAL and C**
- **Modules** are a means to **explicitly manipulate** scopes and names visibility.
  - e.g., Namespaces in C++ are modules.
- They are **not nested** in general
- Objects inside a module can **see each other** (subject to normal lexical scoping)
- Objects outside...able to see in?

Modules

```
namespace fooSpace{
  int bar;
}

void main(){
  bar = 3; //WRONG!!!
  fooSpace.bar = 3; //RIGHT!!!
}
```

Module as manager & as Type

- Two ways to view a module:
  - Module-as-manager means that the module acts as a collection of objects.
    - e.g., namespaces in C++
  - Module-as-type means that the module acts an object type that can have multiple object instances.
    - e.g., classes in C++.

Module as manager & as Type

```
namespace fooSpace{
  int bar;
}

void main(){
  fooSpace.bar = 3;
}
```

```
class fooClass{
public:
  int bar;
}

void main(){
  fooClass qud,zod;
  qud.bar = 3;
  zod.bar = 4;
}
```

Import/Export

- Objects in a module are not visible outside unless **exported**.
    - e.g., In C++ classes, objects are exported via **"public"**
- In some languages, Objects outside are not visible inside the module unless **imported**.
    - e.g., in C++ classes & namespaces, objects are imported via "**.**" as in "***namespacename.variable***" or "**using namespace *namespacename***"
- Bindings made in a module are **inactive outside**, but **not gone**.

Modules

```
namespace fooSpace{
  int bar;
}

void main(){
  bar = 3; //WRONG!!!
  fooSpace.bar = 3; //RIGHT!!!
}
```

Open Scope vs. Closed

- **Open scope**: Names **do not have to be imported explicitly** to be visible.
  - For example, Nested subroutines in Pascal
  - We can see the names in outer lexical scopes without having to ask for the ability.
- **Closed** scope: Names **must be imported explicitly** to be visible
  - Modules in C++, Perl, etc...

Open Scope vs. Closed

```
sub foo()
  a:int;

  sub bar()
    a = 2
  end
end
```

```
namespace fooSpace{
  int bar;
}

void main(){
  fooSpace.bar = 3;
}
```

Referencing in Modules

- We need a **more complicated symbol table** to generate code for non-local referencing at run-time
- Seeing a new name during parsing makes several things happen.
  - Scopes are **counted and numbered serially**
  - Nesting level is also counted implicitly: **scope stack**

Scope Stack Example: Code

```
type T = record F1:int; F2:real; end;
Var V:T;
Module M;
  export I; import V;
  var I : int;
  proc P1(A1:real, A2:int):real
    END-P1
  proc P2(A3:real);
    var I: int;
    with V DO... END;
  END-P2;
END-M;
```

1

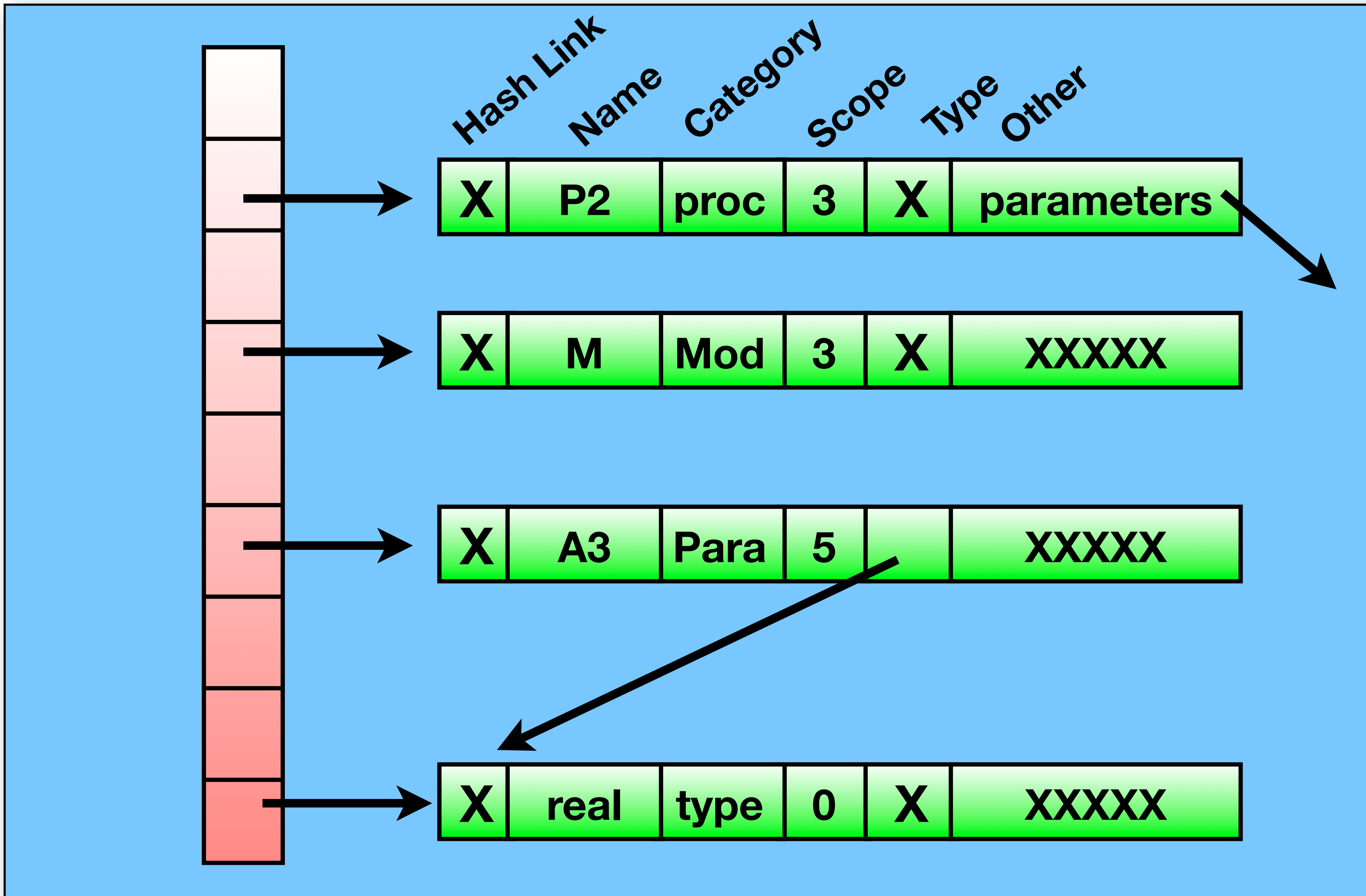
2

3

4

5

Scope Stack Example: Symbol Table



Scope Stack

- A **scope stack** indicates the **order and scopes** that compose the current referencing environment.

Scope	Closed?	Other	
2		rec v	with v
5			P2
3	X		M
1			Global

Scope Stack Example: Code

Lets look at this line

```
type                                    nd;
Var V
Module M;
  export I; import V;
  var I : int;
  proc P1(A1:rea    A2:int):real
    END-P1
  proc P2(A3:real);

    with V DO... END;

END-M;
```

1

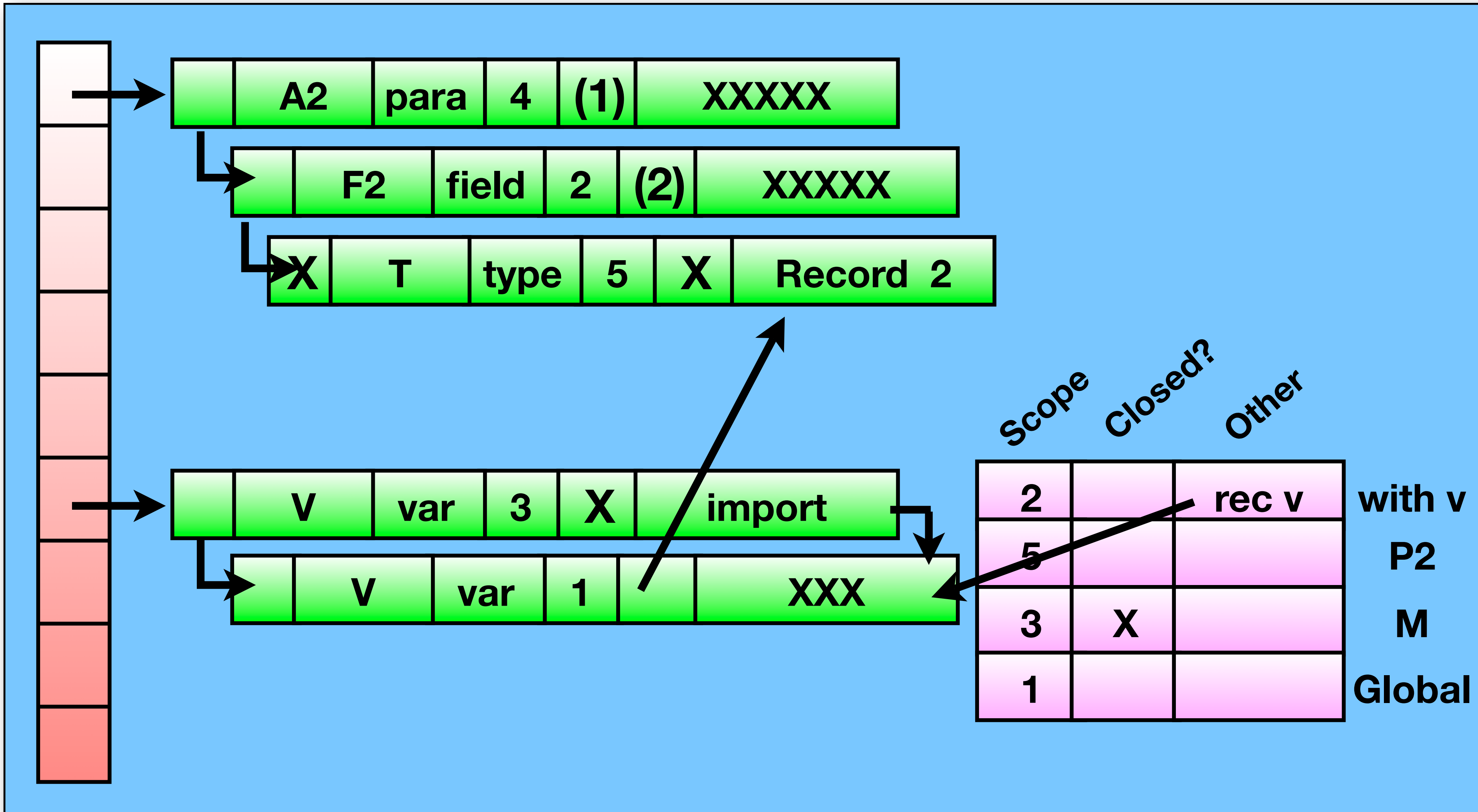
2

3

4

5

Scope Stack With Symbol Table



When a name is seen (parsing)

- When a name is seen during parsing
  - If it's a **declaration** -- hash name and create new entry
  - If it's a **new scope** -- push onto scope stack.
  - If it's a **reference** -- look up, then scan down the scope stack to see if the scope of the name is visible.
  - If it's a **module** -- begin making new entries for the imported names
- When a name is looked up.
  - Hash the name in the table to get entry
  - Hops... stack depth - level where name's scope is found on.

Binding within a Scope: Aliasing

- Aliasing: two names refer to a single object.
  - What are aliases good for? (Absolutely nothing? No!!)
    - space saving
    - linked data structures
  - Also, aliases arise in parameter passing as an unfortunate side effect.

```
double sum, sum_squares;
void acc(double &x){
  sum += x;
  sum_squares += x*x;
}
acc(sum);
```

Since x is passed by reference, this adds the value to sum, then takes the new value and squares that!

Bindi

- Alias
  - Wh
    - space saving
    - linked data structures
  - Also, aliases arise in pa[...]eter passing as an unfortunate side effect.

```
double sum, sum_squares;
void acc(double &x){
  sum += x;
  sum_squares += x*x;
}
acc(sum);
```

Binding within a Scope: Overloading & Coercion

- Overloading
  - Overloaded names can refer to more than one object in a given scope
  - Some overloading happens in almost all languages
    - Typical for arithmetic operators for numerical types
- Coercion
  - Compiler converts types automatically as required by context
- Overloading and coercion are prominent in C++

Polymorphism

- Single subroutine accepts unconverted arguments of unconverted types
- Subtype polymorphism
  - Commonly paired with inheritance in OO languages
- Parametric polymorphism
  - Explicit (genericity): programmer specifies type in “metadata”
    - C++ templates and Java (v. 5+) generics
  - Implicit: type inferred by compiler or interpreter

Overloading vs Genericity

- Overloading in C++ requires multiple functions

```
int min (int a, int b)
  { return ( (a < b) ? a : b ); }

float min (float a, float b)
  { return ( (a < b) ? a : b ); }
```

- Genericity in C++ using a single function template:

```
template <class T>
T min (T a, T b) { return ( (a < b) ? a : b ); }
```