

# Scoping

---

---

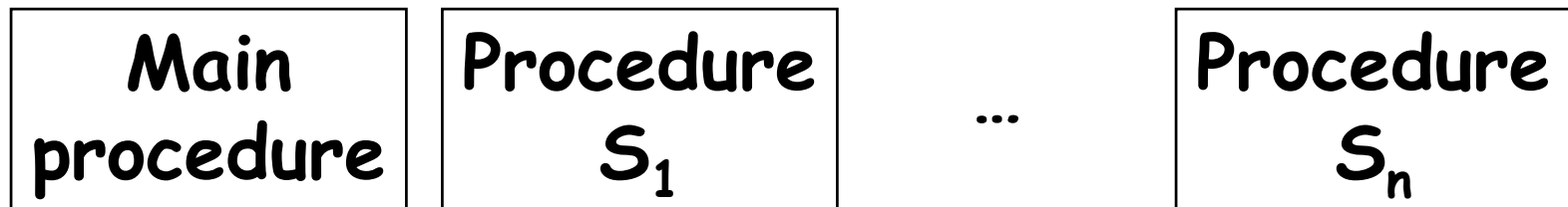
- Chapter 5, 9, 10

# Scope Rules of a Language

---

---

- Which entities (variables, procedures, ...) are **accessible** in which parts of a program? What is their **lifetime**?
- Example: FORTRAN has a set of subroutines (procedures)



- Proc names are visible everywhere
- Local vars are visible only in the declaring proc
- Global vars are visible everywhere

# Static Scope Rule

---

---

- Algol, Pascal, Modula-2, C, C++, Java, ...
- Entities accessible in a scope = entities declared in that scope + entities declared in surrounding scope (minus those with name conflicts) + entities declared in scopes surrounding that scope ...
- Each scope is a box whose sides are one-way mirrors; you can look out of the box, but you can't look into a box

# Example

---

---

```
class Point {
    public: Point(double x, double y);
           virtual void print(); virtual void add(Point* q);
    private: double x,y;
};
Point::Point(double x, double y) { this->x = x; this->y = y; }
void Point::print(); { cout<<x<<","<<y<<endl; }
void Point::add(Point* q) {
    q->print();
    {
        Point *q = new Point(100.0,100.0);
        this->x += q->x; this->y += q->y;
    }
    this->x += q->x; this->y += q->y;
}
int main(void) {
    Point* p1 = new Point(1.0,1.0); p1->print();
    Point* p2 = new Point(2.0,2.0); p1->add(p2); p1->print();
    return 0; }
```

# Dynamic Scope Rule

---

---

- LISP is the prime example
- Entities accessible in a scope = entities declared in that scope + entities declared in the **calling** scope (minus those with name conflicts) + entities declared in scope **calling** that scope ...
- We are not going to talk about it

# Compile time vs. Run time

---

---

- At compile time, we consider the scopes and their nesting
  - Determines which entities (variables, etc.) are accessible in which parts of the code
- At run time, each scope has a lifetime
  - Anything declared in this scope has this lifetime - it becomes alive at the start of the scope, and "dies" at the end of the scope

# Example

```
class Point {
    public: Point(double x, double y);
           virtual void print(); virtual void add(Point* q);
    private: double x,y;
};
Point::Point(double x, double y) { this->x = x; this->y = y; }
void Point::print() { cout<<x<<" "<<y<<endl; }
void Point::add(Point* q) {
    q->print();
    {
        Point *q = new Point(100.0,100.0);
        this->x += q->x; this->y += q->y;
    }
    this->x += q->x; this->y += q->y;
}

int main(void) {
    Point* p1 = new Point(1.0,1.0); p1->print();
    Point* p2 = new Point(2.0,2.0); p1->add(p2); p1->print();
    return 0; }
```

# Implementation of Static Scoping

---

---

- Consider no procedure nesting (C-like)
  - We have one global scope and then just separate local scopes for each procedure
    - all procedure names are in the global scope
    - global variables in the global scope; local variables in each local scope
- Four pieces of memory are used
  - **code segment**: code for all procedures
  - **global (static) segment**: the global variables
  - **run-time call stack**: the local variables
  - **heap segment**: dynamically-allocated entities

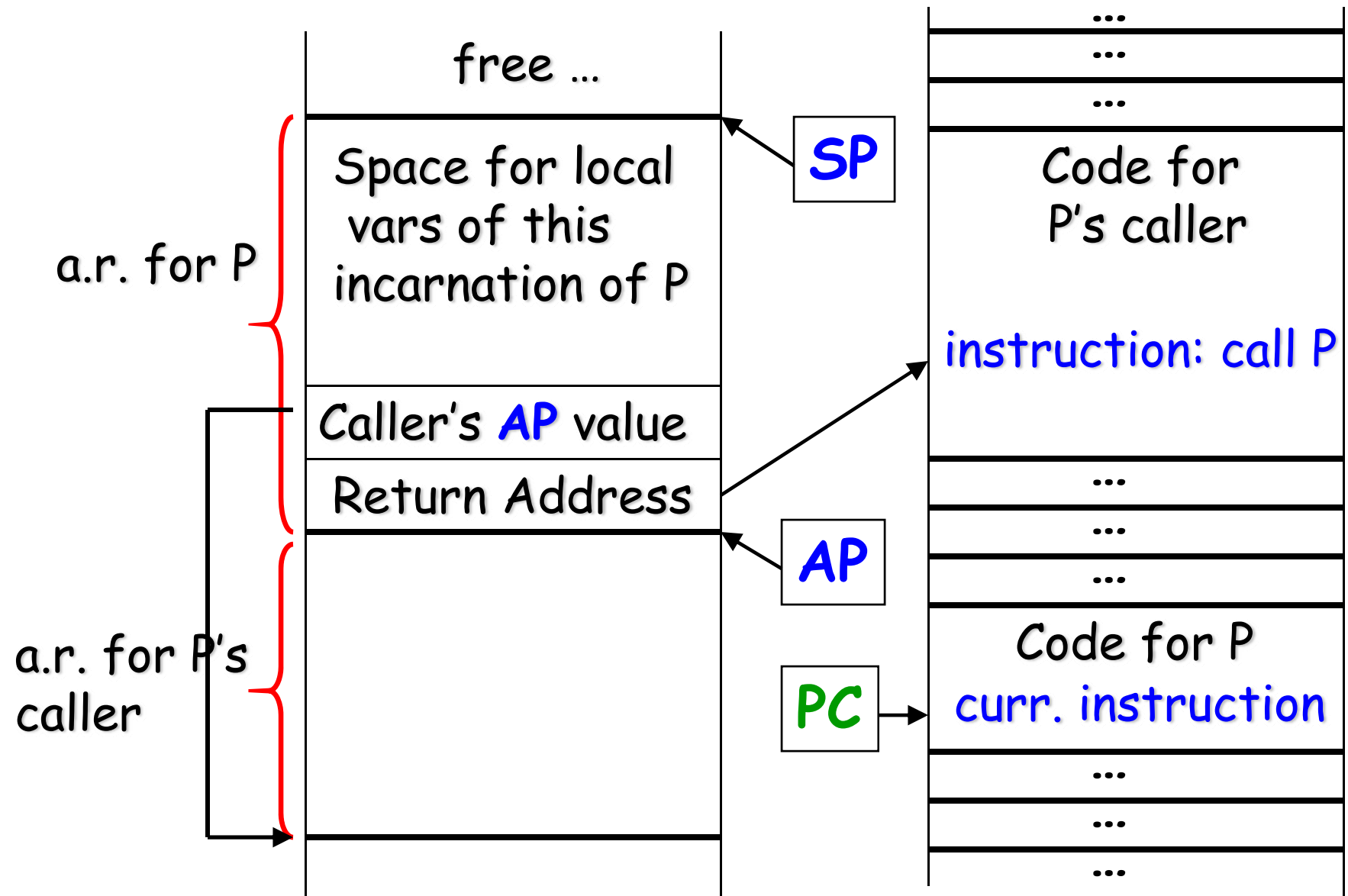
# Run-time Call Stack

---

---

- When a procedure  $P$  begins execution:
  - An **activation record** (a.r.) for *that incarnation* of  $P$  is created on the stack
    - The a.r. will contain space for local variables of that incarnation of  $P$
  - During this incarnation of  $P$ , the **a.r. pointer** (**AP**) register will contain the (starting) address of this activation record
  - The **stack pointer** (**SP**) register will contain the address of the location immediately beyond this a.r.
- When this incarnation of  $P$  finishes, control returns to the caller,  $SP$  is set to the current  $AP$ , and  $AP$  set to the address of the activation record of the caller

# Call Stack: Simple Implementation



# Compile-time Code Generation

---

---

- What code does the compiler produce to make this work?
  - **Mem** is the memory, an array of memory locations
  - **SP** is the stack pointer; points to the next free element of **Mem**
  - **AP** is activation record pointer; points to the first element of the current a.r.
    - Current activation record is from **Mem[AP]** through **Mem[SP-1]**
  - **PC** is the program counter

# Code at "call P"

---

---

- Save return address
  - $\text{Mem}[\text{SP}] := \text{PC} + 4; //$  assuming 4 byte instr.
- Save pointer to caller's activation record
  - $\text{Mem}[\text{SP} + 4] := \text{AP}; //$
- Allocate space for new a.r. of P
  - $\text{AP} := \text{SP}$  and  $\text{SP} := \text{SP} + n$
  - $n$  is the size of P's a.r.; known at compile time
- Jump to P
  - $\text{PC} :=$  address of first instruction in P; known at compile time

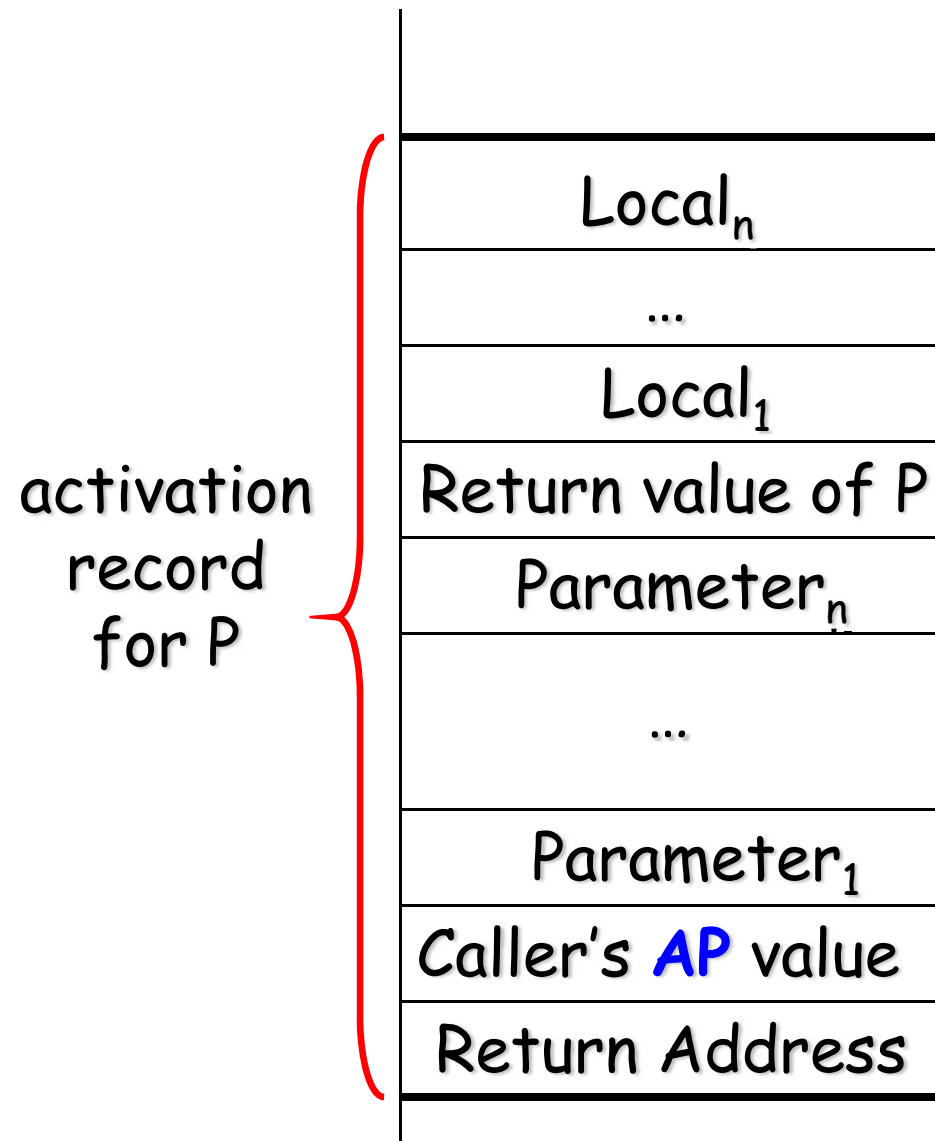
## At the end of P

---

---

- Pop the activation record from the run-time call stack and go back to the caller: restore **AP**, **SP**, reset **PC**
  - **SP** := **AP**
  - **AP** := **Mem**[**AP**+4]
  - **PC** := **Mem**[**SP**]
- Exercise: consider the Point example from earlier and "run" it by hand

# Call Stack: Parameters and Returns



- The formal parameters and the return values are at offsets (w.r.t. **AP**) that are known at compile time
- The caller of P can access them using its value of **SP** (the top of the stack), before and after the call

# Parameter Passing Mechanisms

---

---

- **Call-by-value:** C, Pascal, C++, Java, ...
  - The parameter is a local variable; initialized to the value of the corresponding argument.
  - procedure Swap( x, y) // does not work
    - { var z; z := x; x := y; y := z }
- **Call-by-reference:** C++, Pascal, ...
  - The parameter is not a new variable, but a new reference to the corresponding argument
- Also: call-by-result, call-by-value-result, call-by-name

# Example: C

---

---

- C does not have call by reference
  - Just call by value
- Using pointers, programmers usually “simulate” call by reference

```
void foo() {  
    int x;  
    int * y;  
    y = &x;  
    increment(y);  
}
```

```
void increment (int *f) { *f = *f + 1; }
```

# Classification of Memory Entities

---

---

- **Static**: global variables in C, Pascal, etc.
  - All FORTRAN variables (no recursion)
  - Space allocated at the "static" memory segment
- **Semi-static**: local variables of procedures in statically-scoped languages
  - Space in the a.r. for each incarnation of the procedure
  - Allocated at the start of the incarnation
  - Deallocated when the incarnation finishes
  - Size of a.r. known at compile time

# Classification of Memory Entities

---

---

- **Dynamic**: space allocated and de-allocated as the program executes
  - On the **heap memory segment**; programmer is responsible for memory management
- C: malloc() and free()
- C++:  $A^* a = \text{new } A(); \dots \text{delete } a;$
- Java:  $A a = \text{new } A();$  garbage collection