Subroutine and Control Abstraction (Objectives)

- Given a program with nested scope, the student will able to determine the bindings of variable references to declarations using static scope.
- Given a program with nested scope, the student will able to determine the bindings of variable references to declarations using dynamic scope.
- The student will be able to determine the meaning of programs with aliases and overloading.
- Given a procedure, the student will be able to determine how global variables, local variables and parameters are stored.
- Given a function or procedure call, the student will be able to describe methods for passing the parameters
- Given a linkage convention, the student will be able to determine which parts of the convention are the responsibility of the caller and which are for the callee

1

Procedure Abstraction - Issues

- Assign storage for all variables and compiler temporaries
- Generate code to compute addresses the compiler does not know at compile time
  - dependent on runtime behavior
- Interface with other programs, libraries, languages, OS

2

Abstractions Provided

- Name space
  - variables, values, procedure, etc.
  - local names not visible to outer procedures
  - local names obscure non-locals
  - each invocation has its own local variables
- Control abstraction
  - simple mechanism for invoking and returning from a procedure.
    - code for an invocation can be generated by the compiler without knowing what the callee source code looks like.
    - mechanism to return to calling procedure
- External interface
  - provide an environment where the program can call procedures written by others with confidence concerning the safety of data
    - external routines will not destroy my data
    - I will not destroy external data

3

Namespace: What’s in a name?

- What is the meaning of the following in C?

```
main() {
    char* name = (char*)malloc(200*sizeof(char));
    sprintf(name,”%s”,”fwip”);
    strncat(name,”fwop”,4);
    printf(“%s\n”,name);
}
```

How do you know?

4

What's in a name?

- How about the following?

```
void strncat(char *dest, char* src, int n) {
    strncpy(dest,src,n);
}

main() {
    char* name = (char*)malloc(200*sizeof(char));
    sprintf(name,"%s","fwip");
    strncat(name,"fwop",4);
    printf("%s\n",name);
}
```

5

What's in a name?

- How about the following?

```
void strncpy(char *dest, char* src, int n) {
    strcat(dest,src);
}

void strncat(char *dest, char* src, int n) {
    strncpy(dest,src,n);
}

main() {
    char* name = (char*)malloc(200*sizeof(char));
    sprintf(name,"%s","fwip");
    strncat(name,"fwop",4);
    printf("%s\n",name);
}
```

6

What is the point?

- The meaning of a variable is dependent upon the context in which it appears.
- How do we determine what the context is?
  - Scoping rules – determines the region of a program in which a binding active
- Two different scoping rules
  - Static scoping
    - Also called lexical scoping
    - Scope can be determined at compile time or by examining the text of the program in which it is used.
    - Use the lexically closest definition (more later)
  - Dynamic scoping
    - Scope determine at run-time by the most recent definition
    - Binding dependent upon run-time path of code

7

Name Spaces: Static Scoping

- Nested Scope (Pascal, Algol)
  - name refers to its lexically closest definition
- Fortran
  - global scope for procedures and common blocks
  - local scope for local variables
- C
  - global scope
  - file scope
  - local scope
  - { ... } scope
- Scheme
  - simple strategy of nesting from global inward

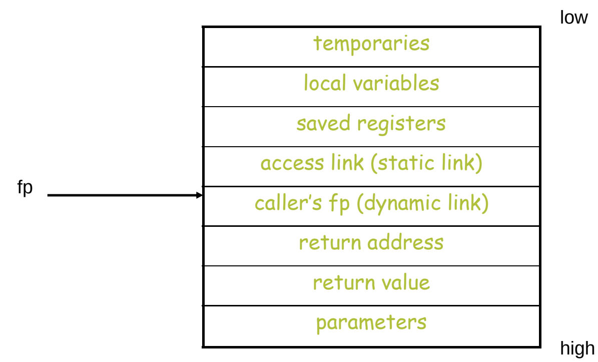
8

Activation Records

- Local variables
  - have a lifetime equivalent to the time while the invocation is active
  - hide locals from other outside procedures
- Activation Records accomplish this task
  - created as part of the standard procedure call
  - destroyed on exit from called procedure
  - include storage for locals, parameters, compiler temporaries
  - the activation record pointer or frame pointer (fp) serves as a base address, variables accessed as an offset to the fp
  - seamlessly handles recursion

9

Activation Record



10

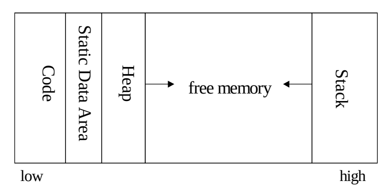
Local Storage

- The ends of the activation record can change from procedure to procedure. The rest is the same size.
- Space for local data
  - leave space for each local variable of known size
  - for unknown size, reserve space for a pointer
- Space for saved register values
  - caller/callee saved registers
  - space for values in registers that are/may be used in the callee

11

Allocating Activation Records

- Stack allocation
  - caller updates stack pointer to reflect the new information it puts on stack
  - callee extends the stack space for its local information
  - variables on stack cannot outlive the lifetime of the procedure
- Heap allocation
  - for languages where local variable can outlive a procedure execution
    - Scheme, Lisp
    - first-class functions
    - garbage collection required
- Static allocation
  - for leaf procedures



12

Example

- Trace the activation records at each point in executing the following code

```
 int f(int n) {
       int m;
       m = n * 4;
       return m;
}

int g(int m) {
       int f = m + 2;
       return f;
}

main() {
       int n = f(7) + g(2);
       printf("%d\n",n);
}
```


Nested Subroutines

- Some languages allow nested subroutines – Pascal, Scheme, Lisp, PL/I
  - declarations refer to the closest nested declaration
    - a name that is introduced in a declaration is known in the scope in which it is declared and all internally nested scope, unless it is hidden by another declaration of the same name in one or more nested scopes.

Example Nested Subroutines

```
procedure P1 (a : integer; x : array [1..10] of integer);
    procedure P2 (b : integer);
           procedure P3 (c : integer);
           begin
             … = a + b + c
           end;
    begin
      … = P3(b)
    end;
    procedure P3 (a : array[1..10] of integer);
           procedure P2 (a : array[1..10] of integer);
           begin
             … = P2(a) + P3(a) (* stop here *)
           end;
    begin
      … = P2(x) + P3(x)
    end;
begin
    … = P2(a) + P3(x)
end;
```

To which declaration is each variable and function reference bound?
Trace the activation records until execution reaches the indicated point?

Establishing Addressability

- for global and static variables, reference a label in the static data area (_gp in our project)
- for locals, offset off of $fp (%rbp in x86-64)
- What do we do for local variables of procedures at outer nesting depths?
  - access link (static link)
  - display

Scoping Support

- There are two parts to supporting scope
  - compile time
  - run time
- At compile time, the compiler must model the set of names that are accessible at a particular point in the code
  - block-structured symbol tables
- The compiler must generate code, for the runtime, to access names within its lexical scope
  - access links or displays

17

Handling Nested Scope

- We want the lexically most recent declaration of a variable.
- Operations needed
  - insert(name,p) – insert record for name at nesting level p
  - lookup(name) – return the most recent record for name
  - delete(p) – delete all records at nesting level p
- Simple strategy
  - Use a stack of hash tables
  - When entering a new scope push a new table on the stack
  - Search tables from top to bottom of the stack
  - pop table off of stack when leaving a scope

18

Example

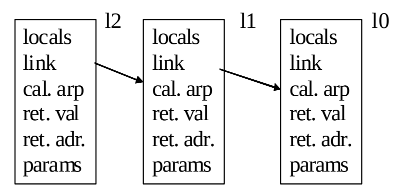
```
PROGRAM test1;
VAR x,y,z : INTEGER;

    PROCEDURE p1(x,y:INT)
    VAR g : INTEGER;

     PROCEDURE p2(g:FLOAT)
       BEGIN
         x = z + g + y;
         p3(x)
       END;

        PROCEDURE p3(x:INT)
          BEGIN
            x = z + g + y;
            p1(x, y)
          END;

       BEGIN
         p2(x)
       END;

     BEGIN
         p1(x, y)
     END;
```

19

Run-time Access to Local Variables of Other Procedures

- Basically, how do we have access to the memory locations in the block structured-symbol table at run-time for a particular program point?
  - All variables at each scope level are put in one activation record
  - one link to lexically enclosing AR (not necessarily the caller's), or the variables at the next outer level

```
procedure l0;
  procedure l1;
    procedure l2;
```



20

Setting Up Access Links to Model Symbol Table

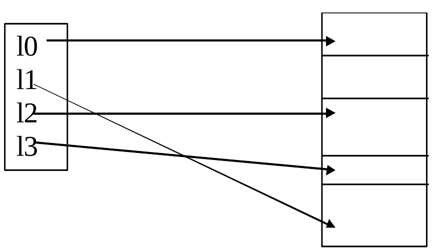
- Caller at level m, callee at level n
  - case n = m + 1
    - callee uses caller's arp as the access link
  - case n = m
    - callee's access link is the same as the caller's access link
  - case n < m
    - callee's access link is the level n – 1 access link for the caller

Example: Access links and Non-local Access

```
PROGRAM test1;
VAR x,y,z : INTEGER;

    PROCEDURE p1(x,y:INT)
    VAR g : INTEGER;

      PROCEDURE p2(g:FLOAT)
        BEGIN
          x = z + g + y;
          p3(x)
        END;

  PROCEDURE p3(x:INT)
    BEGIN
      x = z + g + y;
      p1(x, y)
    END;

  BEGIN
    p2(x)
  END;

BEGIN
  p1(x, y)
END;
```


Another Approach: Use a Global Display

- Allocate a globally accessible array to hold the ARPs of the most recent instance of a procedure called at each level



Code to Support Displays

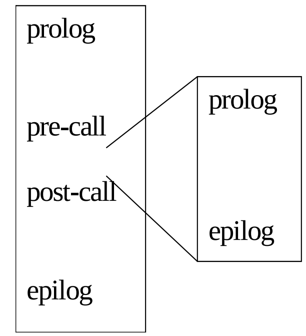
- Add a slot in the AR to hold the old FP stored in the display
- On call, copy FP from the display to new AR, store new FP in display
- On return, copy FP from AR to the display
- Example: Do previous example with displays

Comparison

- call overhead
  - display is constant overhead
  - access link depends on the call
- reference overhead
  - display constant for any non-local access
  - access link depends on how many levels out the declaration occurs

Linkage Convention

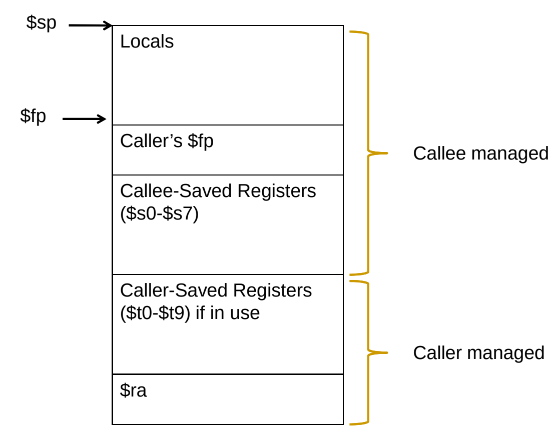
- What happens at a call?
  - pre-call – set up AR
  - post-return – undo pre-call actions
  - prolog – extend AR
  - epilog – undo prolog code



Linkage Convention

	Caller	Callee
	pre-call	post-call
Call	allocate AR eval & store params store ret. addr. & FP save caller-saved regs jump to callee	save callee-saved regs extend AR for locals find static data area initialize locals fall through to code
	post-return	pre-return
Return	deallocate basic AR restore caller-saved regs restore reference parameters	restore callee-saved regs discard local data restore caller's FP jump to return address

Mips Calling Convention For Project 3



Mips Calling Convention

$sp →	Locals	Callee managed
$fp →	Caller's $fp	Callee managed
	Callee-Saved Registers ($s0-$s7)	Callee managed
	arg 0 arg 1 arg 2 … arg n-1	Caller managed
	Caller-Saved Registers ($t0-$t9) if in use	Caller managed
	$ra	Caller managed

Dynamic Scope

- In a dynamically scoped language, the scope is determined at run-time.
  - the most recent binding encountered during execution is used
  - scope cannot, in general, be determined at compile time.
- At run-time, the binding is determined by scanning the stack of activation records from top to bottom until the first binding of the name is found

Example

- Evaluate the following using dynamic scoping. Show the stack of activation records at the point where the writeln occurs.

```
program f1;
 var x : integer;
  procedure f2(a: integer);
    begin
           writeln(a+x)
    end;
  procedure f3(a : integer);
    var x : integer;
     begin
        x := 3;
        f2(x)
     end;
begin
   x : = 1;
   f3(5);
end;
```


Practice Problem

- Evaluate the following using dynamic scoping. Show the stack of activation records at the point where the writeln occurs.

```
program f1;
  var x : integer;
   procedure f2(a: integer);
     begin
            writeln(a+x)
     end;
   procedure f3(a : integer);
     var x : integer;
      begin
         x := 3;
      end;
begin
   x : = 1;
   f3(5);
   f2(5)
end;
```

Shallow vs. Deep Binding

- Shallow binding refers to when the (non-local) referencing environment of a procedure instance is the referencing environment in force at the time the procedure is invoked (i.e., the one in which the procedure is invoked)
- Deep binding refers to when the (non-local) referencing environment of a procedure instance is the referencing environment in force at the time the procedure's declaration is elaborated (i.e., the one in which the procedure was passed as an argument)
- Both of these binding methods can be applied using either static or dynamic scope rules

33

```
program BindingExample (input, output);

procedure A (I : integer; procedure P);

    procedure B;
    begin
        writeln (I);
    end;

begin (* A *)
    if I > 1 then
        P
    else
        A (2, B);
end;

procedure C; begin end;

begin (* main *)
    A (1, C);
end.
```

34

Aliasing

- Aliasing occurs when more than one name refers to the same memory location.

```
int foo(int* y, int* x)
      *y += *x;
     return *x * 2;
}

int a = 5;
int b = foo(&a,&a);
printf("%d\t%d\n",a,b);
```

- Aliasing is generally considered bad:
  - hard to understand
  - less efficient code
    - impossible for compiler to prove no aliasing in general

35

Overloading

- Overloading uses a single name to mean different things based upon the context (often types). In C++,

  void print(int a);
  void print(float a);

  refer to different methods.
- How can the compiler differentiate the use?
  - name mangling is one approach
  - this is not applicable to objects and virtual functions, a different mechanism is needed.

36

Parameter Passing

- call-by-value
  - evaluate the parameter and pass the result
- call-by-reference
  - for variables pass the address
  - for expressions, evaluate, store on stack, pass address
- call-by-value/result
  - evaluate the parameter and pass the result
  - callee stores final value on stack
  - store result back in variable memory location
- call-by-name
  - encapsulate the parameter in a function
  - re-evaluate on every reference in called procedure

Example

- Interpret the following program using call-by-value, call-by-reference, call-by-value-result and call-by-name.

```
int x = 3;
int y = 4;

void f1(int a, int b) {
   a = a + b;
   printf(“x = %d\n”, x);
   printf(“b = %d\n”, b);
}

main () {
      f1(x, x + y);
      printf(“x = %d\n”, x);
}
```


Practice Problem

- Interpret the following program using call-by-value, call-by-reference, call-by-value-result and call-by-name semantics.

```
int x =10;
int y = 20;
void f1(int a, int b) {
   y = a;
   printf (“b = %d\n”,b);
   b = a + 10;
}

main () {
      f1(x +y,y);
      printf(“x = %d\n”,x);
      printf(“y = %d\n”,y) ;
}
```


Practice Problem

```
char *L = “012345”;
char str[10];

void f(char *L1, char* L2) {
   L1 = &L1[1];
   printf(“%s\n”,L);
   printf(“%s\n”,L2);
}

main () {
   f(L, strcpy(dummy,&L[1]));
   printf(“L = %s\n”,L);
}
```

Pros and Cons

Method	Pros	Cons
By-value	No bad effects	No side effects Potentially expensive
By-reference	Inexpensive Side effects	Aliasing
By-value-result	Side effects No aliasing	Expensive
By-name	Converges if possible side effects	Expensive "Strange" behavior