Implementing Subprograms

Call/Return Semantics

- Subprogram linkage
- At call
  1. Parameter passing
  2. Allocation of storage for locals
  3. Saving status of calling routine
  4. Transfer of control
  5. Saving return address
  6. Establish access to non-locals
- At return
  1. Parameter return
  2. Return value
  3. Deallocation of storage
  4. Restore status of calling routine
  5. Return of control

Simple Subprograms

- Non-nested and static locals
- Storage
  - Code, locals, parameters
  - Activation record (AR)
  - Also status of calling and return address
  - Fixed and can be statically allocated
- Linker
  - Loads separately compiled parts
  - Resolves external references
Stack-dynamic Locals

- Storage allocation/deallocation for parameters & locals
- Supports recursion
  - Requires multiple ARs
- AR
  - Size and layout usually known at compile-time
  - LIFO call/return implies stack for ARs
  - Dynamic link is pointer to base of AR of caller
- Environment Pointer (EP)
  - Points to base of AR of currently executing routine
  - Offset for addressing – offset computed at compile time
  - Used to reset stack
- E.g.
  - Dynamic chain
  - Offsets
Figure 10.3
A typical activation record for a function with calculation.

void sub(float total, int part) {
    int list[3];
    float sum;
    ...
}

Figure 10.4
The activation record for functions with.

void main(float r) {
    int s, t;
    ...
    fun3(a);
    ...
}

void main(int x) {
    int y;
    ...
    fun4(y);
    ...
}

void main(int y) {
    int x;
    ...
    fun5(x);
    ...
}

void sub(float x, float y) {
    int m, n;
    ...
    fun6(m, n);
    ...
}

main calls fun1
fun1 calls fun2
fun2 calls fun3
Recursion

- Requires dynamic local storage
  - E.g. stack dynamic
- E.g.
  - Multiple ARs for same function (factorial)
  - Only one active
  - Local offsets are still correct

```c
int factorial(int n) {
    if (n <= 1)
        return 1;
    else return (n * factorial(n - 1));
}

void main() {
    int value;
    value = factorial(3);
}
```

Figure 10.5
The activation record for two levels in a program.
Nested Subprograms

- Static scoping
- All non-locals must be in some AR
  - Locate correct AR
  - Offset into AR
- ARs of static ancestors must be on stack
  - Static scoping semantics
- Static chain
  - Each AR has pointer (static link) to AR of its static ancestor
  - Could resolve non-local references by searching up static chain
(depth,offset) pairs
- Static depth – number routines nested within
- Nesting-depth – difference between static depth of accessing routine and static depth of declaring routine
- Offset is local offset in AR

E.g.
Establishing static link
- Compute nesting depth as difference between static depth of caller and static depth of declaring routine
- Follow static chain nesting depth links to locate AR for static link

Note
- References get more expensive as more non-local

Blocks
- As anonymous subroutines
- Locals as additional (hidden) locals for routine
Dynamic Scoping

- Most recently declared
- Deep access
  - Search up dynamic chain
  - Expensive access
- Shallow access
  - Separate stack for each name
    - Reference top of stack
    - At declaration, push on correct stack
  - Central table by name
    - If active, relocate
    - Expensive allocation