PROCEDURES

CMPT 379 Lecture 21b

Lecture Overview

- Procedures and the procedure call abstraction
- Name spaces / scopes
- Frames
- Object-oriented mechanisms
- Parameter binding
- Standardized Linkages

Chapter 6

Procedures

- Procedures are the central abstraction of modern programming languages.
- They create a controlled execution environment for a piece of code.
- They have their own private named storage.
- They provide interfaces between system components.
- They are the basic unit of work for most compilers. They allow separate compilation.

Procedures

- We examine the techniques used to implement procedures and procedure calls.
- These consist of both compile-time and run-time algorithms and data structures.
- We study implementation of control, of naming, and of the call interface.
- These features make programming languages usable and large software systems possible.

Procedure Call Abstraction

- Procedural languages support an abstraction for procedure calls.
- Each language has a standard mechanism for invoking a procedure and mapping a set of arguments/parameters to the callee's name space.
- There is a provision for returning control to the caller and continuing execution at the point immediately after the call.
- Most languages allow a procedure to return a value or values to the caller. Such a procedure is called a function.
- Standard linkage conventions or calling sequences allow code written and compiled at a different time to be used together.

Name Spaces

- In most languages, each procedure creates a new and protected name space or scope.
- This allows the programmer to declare new names without concern for the surrounding context.
- Inside the procedure, these local declarations take precedence over other earlier declarations for the same names.
- A procedure's parameters allow the programmer to map values and variables from the caller into the callee's name space.

Name Spaces

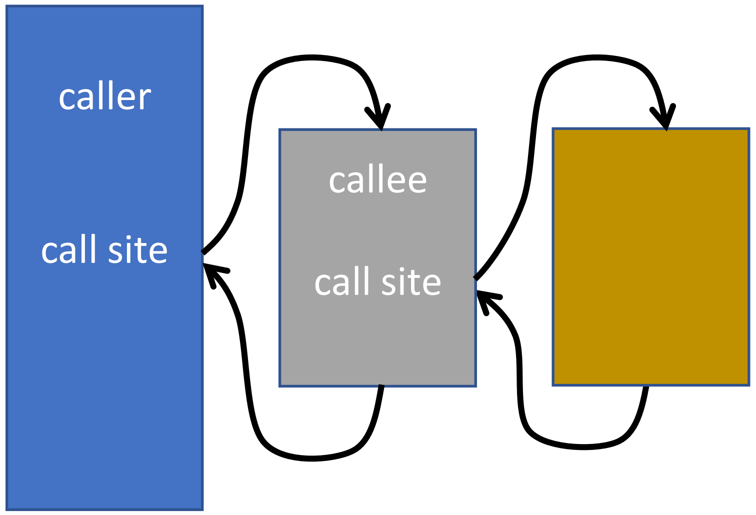
- The separate name space allows a procedure to function consistently when called from different contexts.
- At runtime, executing a call instantiates the callee's name space.
- The call must create or allocate storage for the objects declared by the callee.
- This allocation must be automatic and should be efficient.

External Interface

- Procedures define the critical interfaces among the parts of large software systems.
- The linkage conventions/calling sequences define rules that:
  - map names to locations,
  - preserve the caller's runtime environment,
  - create the callee's runtime environment, and
  - transfer control between caller and callee.
- Linkage conventions allow the development and use of libraries.
- Without them, programmers and compilers would both need more detailed knowledge about the callee of each procedure call.

Procedure Calls

- In procedural languages, a procedure call transfers control from the call site in the caller to the start of the callee.
- On exit from the callee, control returns to the point that immediately follows the call site.
- If the callee invokes other procedures, they return control in the same way.
- This "last to be called is first to return" is LIFO behaviour, and so it can be modelled with a stack.



Beyond Simple Procedural Languages

- In Object-Oriented languages, procedure calls and returns are mainly the same as procedural languages. There are differences in the mechanism to name the callee and locate it at runtime.
- Some languages (Scheme e.g.) allow a program to encapsulate a procedure and its runtime environment into a closure. This allows for later execution of the procedure. A stack is inadequate for the activation records / frames in these languages.
- Similar issues arise when a reference to a local variable can outlive a procedure's activation.

Activation Records (Frames)

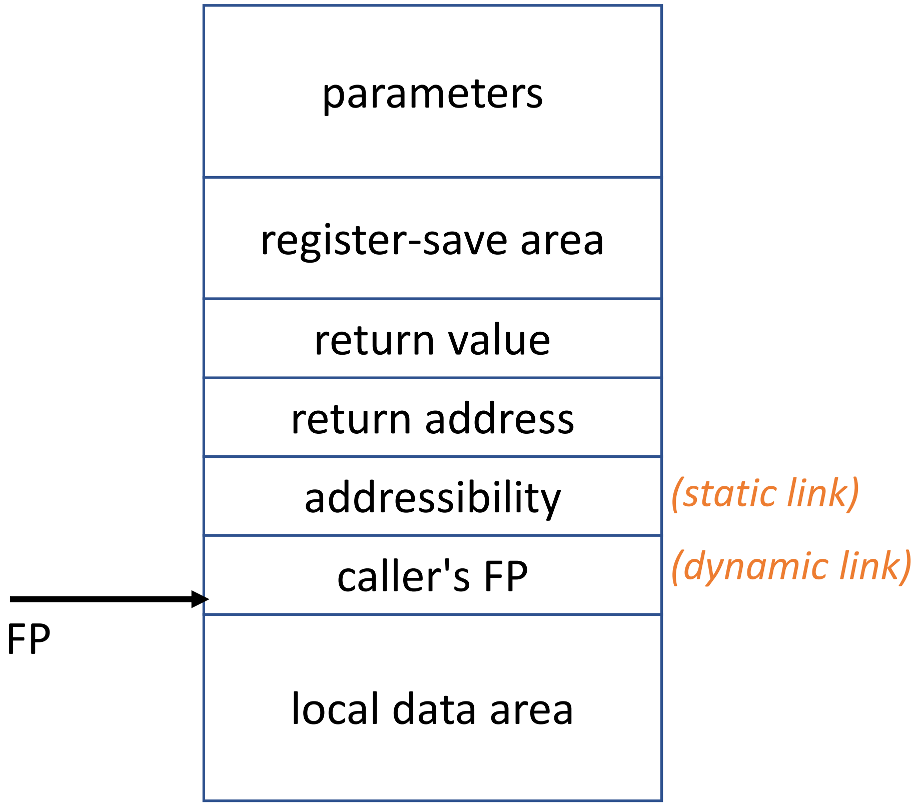
For each scope activation, the compiler needs a location to store the variables defined in the scope.

For procedures and their simple subscopes, this location is provided by an activation record or frame.

Each currently-executing instance of a procedure gets an activation record, so recursive procedures may have several activation records at any one time.

Also included in an activation record are the procedure's parameters and bookkeeping information associated with the procedure call.

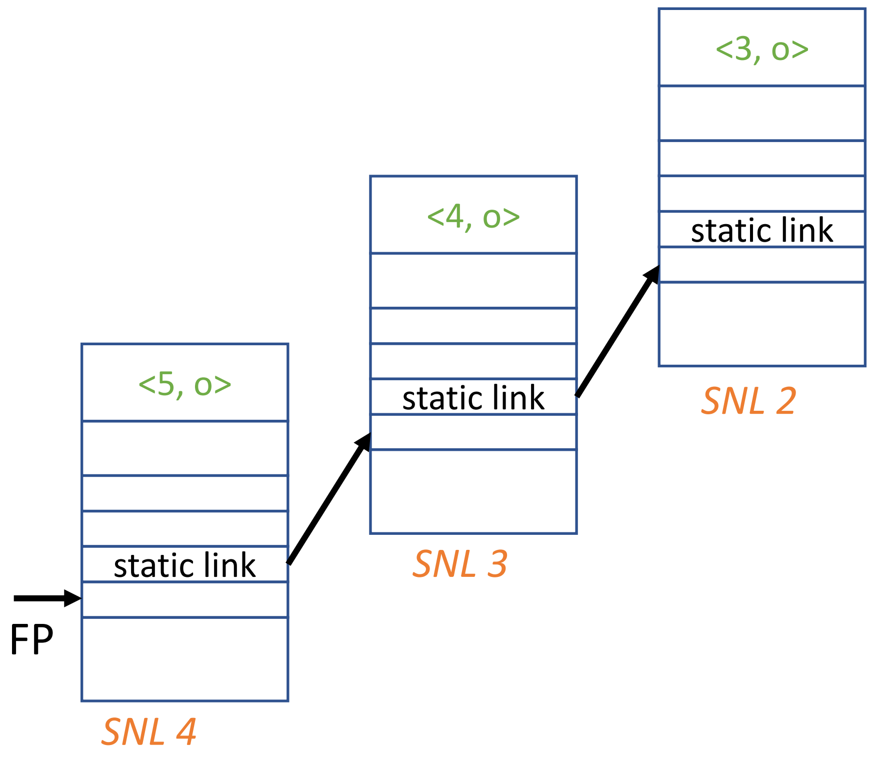
Typical Frame



All data in the frame is accessed through the FP.

Because procedures access their data frequently, most compilers dedicate a register to the FP.

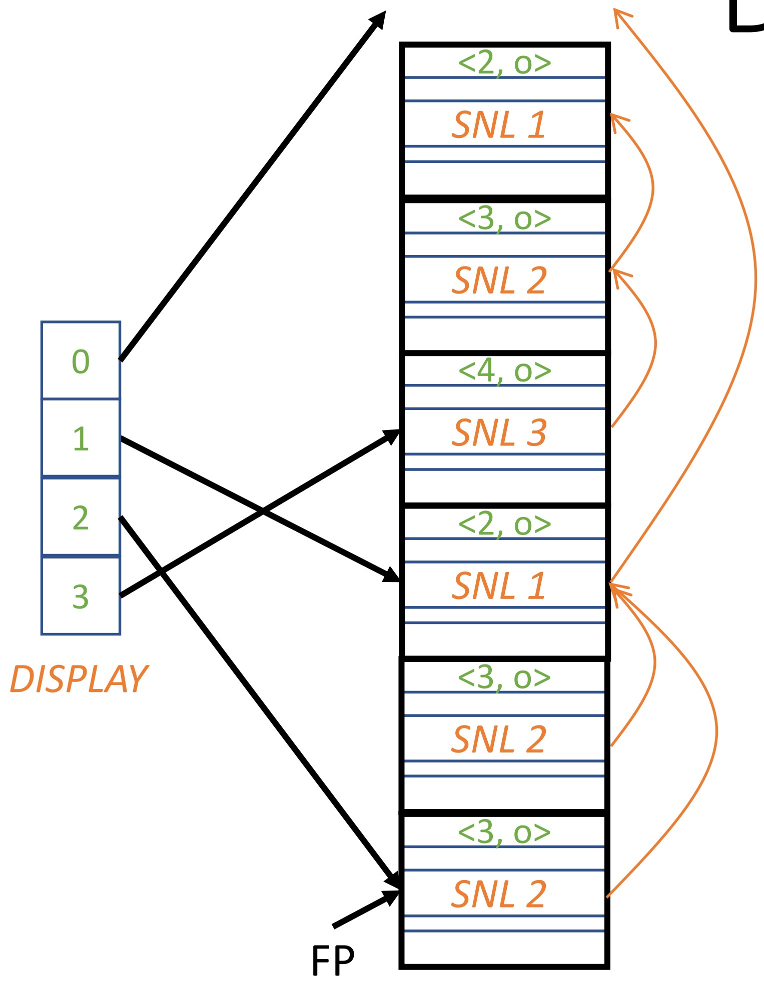
Access to Nonlocal Variables: Static chaining



A static coordinate is <m, o> where m is a static nesting level and o is an offset.

If we're executing a procedure at static nesting level k, and we want a variable at static nesting level m, we follow the static links of frames k-m+1 times, then use the offset.

Access to Nonlocal Variables: Display



A display is an array A where A[i] is the FP to the frame of the latest still-active procedure of static nesting level i.

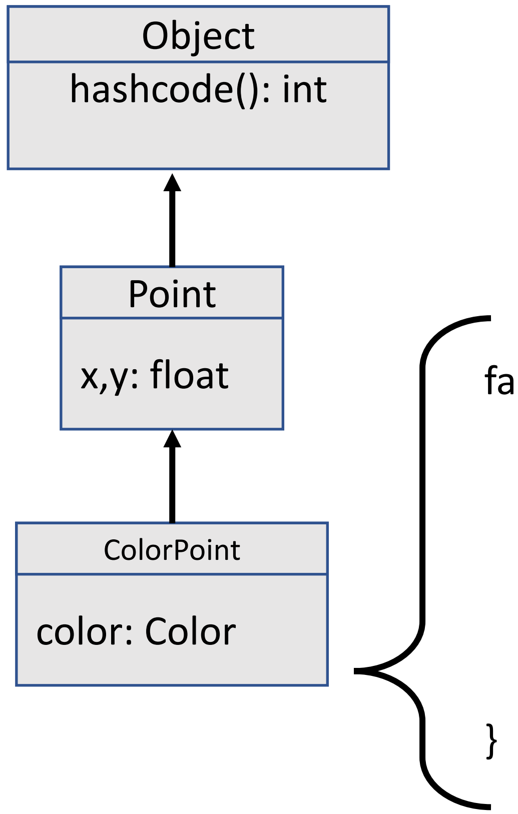
If we're executing a procedure at any static nesting level k, and we want a variable at static nesting level m, we follow the pointer display[m] and apply the offset.

Object-oriented Languages

A scope in an Object-oriented language (OOL) can inherit names from multiple hierarchies.

- As in procedural languages, they inherit from lexically containing scopes.
- They also inherit from their class's scope, and its superclass's scope, its superclass's superclass's scope, etc.
- In languages with multiple inheritance, like C++, they can inherit from multiple superclass hierarchies.

Object-oriented Languages



```
fadeColor(Color c) {
    int i = 0;
    int j = 0;
    while (i < MAX) {
        float k;
        <code>
    }
}
```

<code> has access to:

- k from its own scope
- i and j from the lexically enclosing scope.
- c from the procedure's scope.
- color from its class
- x and y from its class's superclass.
- hashcode() from its superclass's superclass.

Object-oriented Languages

The compiler must keep symbol tables for any class-level scopes that may be used in the code being compiled.

This includes symbol tables for qualified references -- any code that uses for instance

```
r = color.getRed();
```

where color is of type Color, must be aware of the symbol table for Color.

Many languages require the programmer to specify which symbol tables are active by importing them.

Runtime support for OOLs

Objects need to store their instance variables. Since object lifetimes are not necessarily tied to any procedure's lifetime, they cannot be stored on a frame. Rather, they are stored in an Object Record (OR) on the heap.

The OR typically includes some system-related information, most commonly a pointer to information about its class.

In some languages, classes are first-class meaning they are objects in their own right, having an OR and the ability to change at runtime.

Lambda Lifting

Typically a call to an object's methods look something like:

```
Shape myShape()
...
myShape.twist(arg1, arg2, arg3 ...)
```

Compilers translate this call to something like:

```
-Shape-twist(myShape, arg1, arg2, arg3 ...)
```

That is, the particular *twist* for the class of myShape is called, with myShape as the first argument, and the other arguments following. This is called lambda lifting. Many languages have a standard name for this hidden first argument, such as self or this. Generically, this object is known as the receiver.

Formals and Actuals

A procedure typically takes a number of parameters. The parameters as they appear in the procedure are called formal parameters or formals.

A procedure call has expressions for corresponding to each of the formal parameters; these expressions are called actual parameters or actuals.

```
foo(int a, int c) {
    int b = (a+c)/2;
    return b * b;
}
…

    int y = foo(2, 4+x);
```

Call by Value

Call by Value is a parameter-binding convention where the caller evaluates the actuals and passes their values to the callee.

Any modification of a value parameter in the callee is not visible in the caller.

Call by Name

Call by Name is a parameter-binding convention where a reference to a formal parameter in the callee behaves exactly as if the actual parameter had been textually substituted in its place, with appropriate renaming.

Basically, it's like a #define from C, but was the natural parameter-binding in Algol 60.

Call by Name is difficult to implement and difficult to comprehend, and it has fallen into disfavour.

There's an example in the text.

Call by Reference

Call by Reference is a parameter-binding convention where each parameter is passed as a reference. If the actual parameter is a variable, then its address is passed. If the actual parameter is an expression, the caller evaluates the expression, stores the result in its own frame, and passes a pointer to that result.

Inside the callee, each reference to a call-by-reference formal parameter needs an extra level of indirection. Any redefinition of a reference formal is reflected in the actual. Also, any reference formal might end up bound to a variable that is accessible by another name inside the callee. When this happens, the names are called aliases. Aliasing can create counterintuitive behaviour.

Aliasing

Here's an example of aliasing in C.

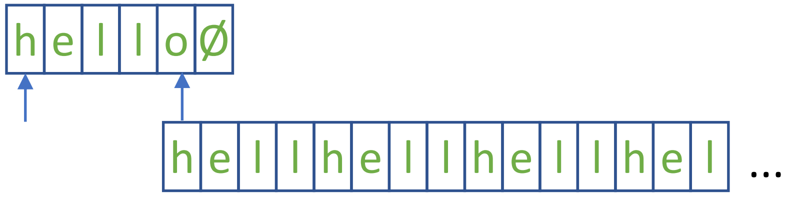
A simple implementation of strcpy (string copy) is:

```
void strcpy(char *s, char *t)  {
        while(*t != 0)
                *s++ = *t++;
        *s = 0;
}
```

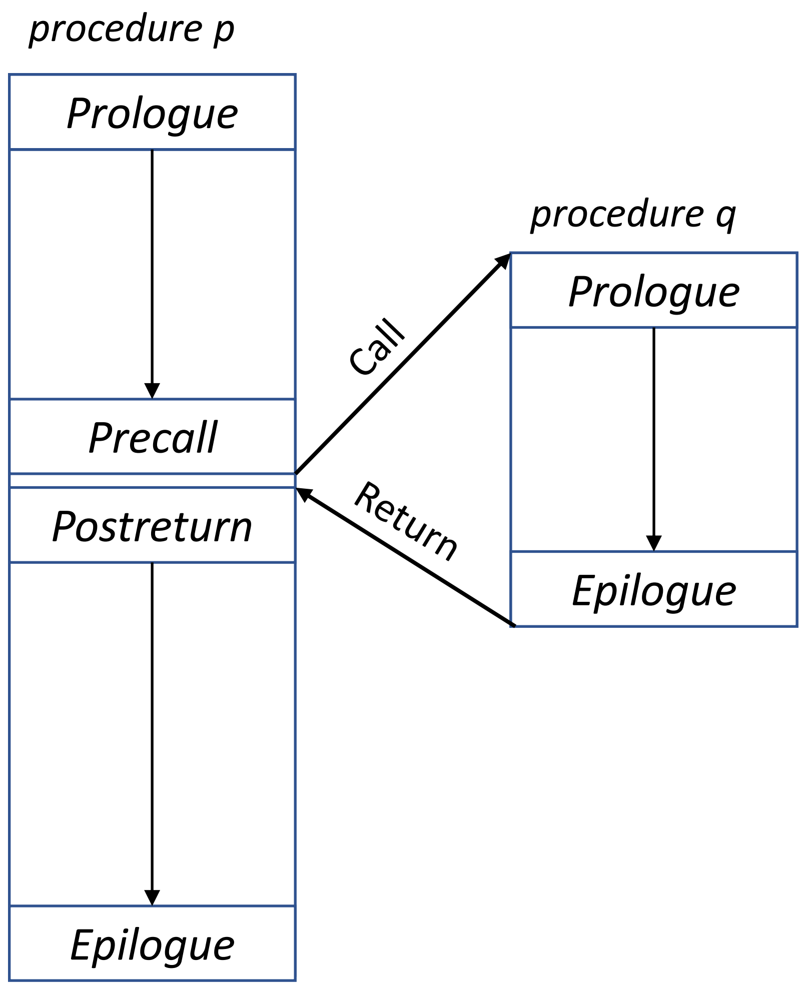
But what if we call it this way:

```
        char *u = "hello";
        strcpy(u+4, u);
```

strcpy would never stop, filling memory with "hell"!



Standardized Linkages



The precall is also known as the caller entrance handshake; the prologue is the callee entrance handshake; the epilogue is the callee exit handshake; the postreturn is the caller exit handshake.