Reading Assignment

Get and read Chapter 11 of Crafting a Compiler featuring Java.

(Available from DoIt Tech Store)

Accessing Frames at Run-Time

During execution there can be many frames on the stack. When a procedure **A** calls a procedure **B**, a frame for **B**’s local variables is pushed on the stack, covering **A**’s frame. **A**’s frame can’t be popped off because **A** will resume execution after **B** returns.

For recursive routines there can be hundreds or even thousands of frames on the stack. All frames but the topmost represent suspended subroutines, waiting for a call to return.

The topmost frame is *active*; it is important to be able to access it directly.

The active frame is at the top of the stack, so the *stack top register* could be used to access it.

The run-time stack may also be used to hold data other than frames.

It is unwise to require that the currently active frame always be at *exactly* the top of the stack.

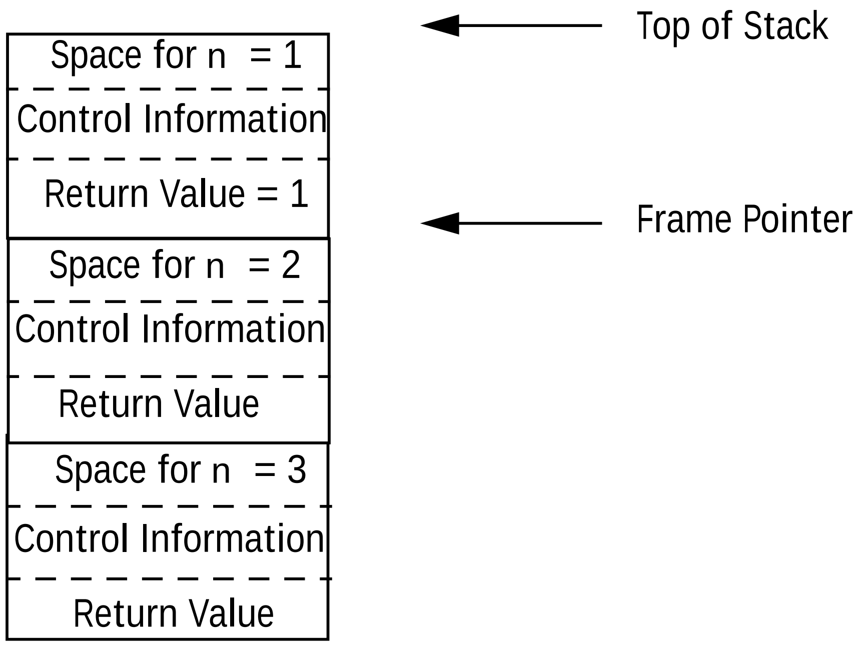
Instead a distinct register, often called the *frame pointer*, is used to access the current frame.

This allows local variables to be accessed directly as offset + frame pointer, using the indexed addressing mode found on all modern machines.

Consider the following recursive function that computes factorials.

```
int fact(int n) {
   if (n > 1)
        return n * fact(n-1);
   else return 1;
}
```

The run-time stack corresponding to the call **`fact(3)`** (when the call of **`fact(1)`** is about to return) is:



We place a slot for the function’s return value at the very beginning of the frame.

Upon return, the return value is conveniently placed on the stack, just beyond the end of the caller’s frame. Often compilers return scalar values in specially designated registers, eliminating unnecessary loads and stores. For values too large to fit in a register (arrays or objects), the stack is used.

When a method returns, its frame is popped from the stack and the frame pointer is reset to point to the caller’s frame.

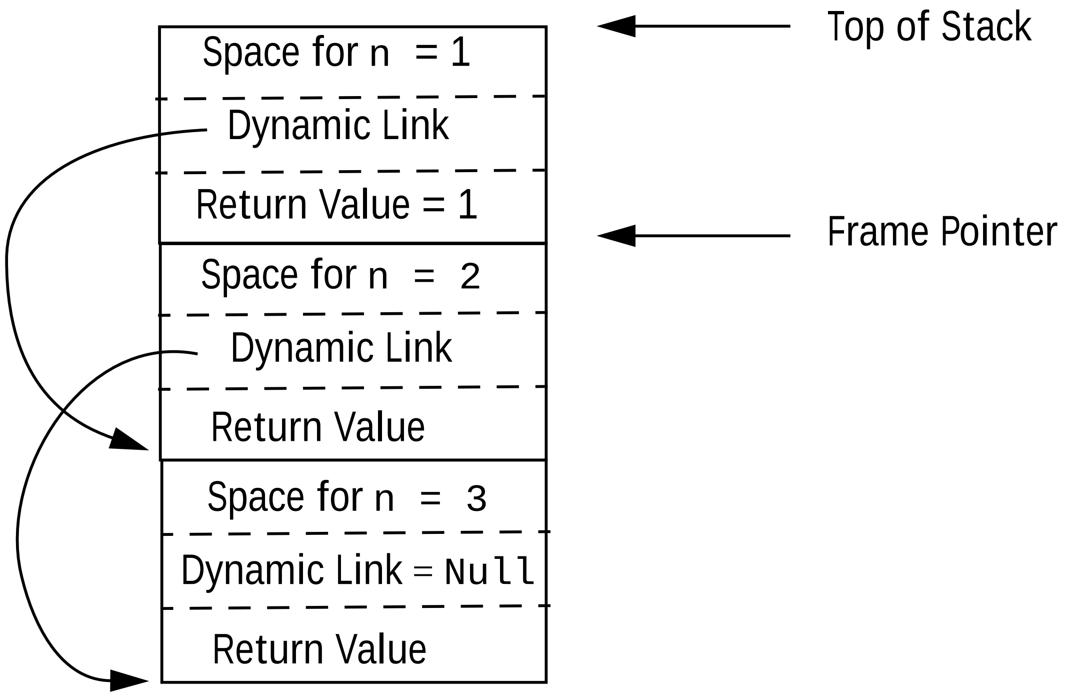
In simple cases this is done by adjusting the frame pointer by the size of the current frame.

Dynamic Links

Because the stack may contain more than just frames (e.g., function return values or registers saved across calls), it is common to save the caller's frame pointer as part of the callee's control information.

Each frame points to its caller's frame on the stack. This pointer is called a *dynamic link* because it links a frame to its dynamic (run-time) predecessor.

The run-time stack corresponding to a call of **`fact(3)`**, with dynamic links included, is:



Classes and Objects

C, C++ and Java do not allow procedures or methods to nest.

A procedure may not be declared within another procedure.

This simplifies run-time data access—all variables are either global or local.

Global variables are statically allocated. Local variables are part of a single frame, accessed through the frame pointer.

Java and C++ allow classes to have *member functions* that have direct access to instance variables.

Consider:

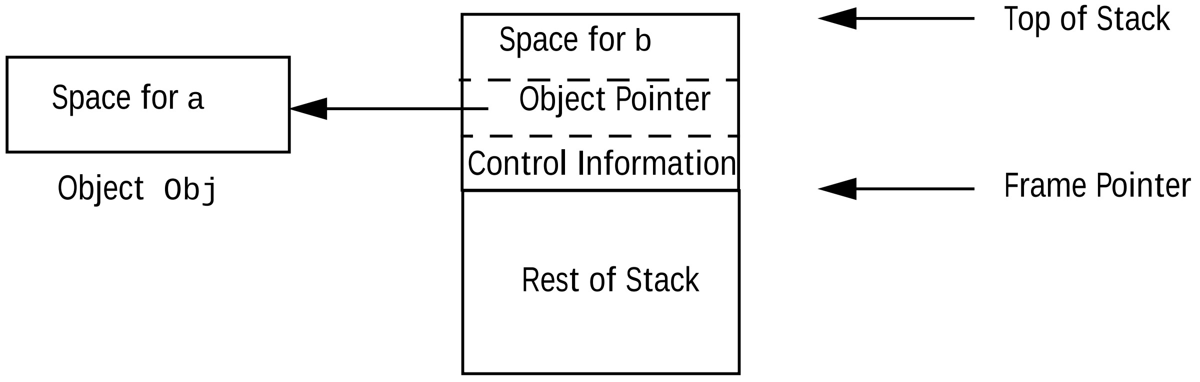
```
class K {
  int a;
  int sum(){
     int b;
  return a+b;
} }
```

Each object that is an instance of class **K** contains a member function **sum**. Only one translation of **sum** is created; it is shared by all instances of **K**.

When **sum** executes it needs *two* pointers to access local and object-level data.

Local data, as usual, resides in a frame on the run-time stack.

Data values for a particular instance of **K** are accessed through an object pointer (called the **this** pointer in Java and C++). When **obj.sum()** is called, it is given an extra *implicit parameter* that a pointer to **obj**.



When **a+b** is computed, **b**, a local variable, is accessed directly through the frame pointer. **a**, a member of object **obj**, is accessed indirectly through the object pointer that is stored in the frame (as all parameters to a method are).

C++ and Java also allow inheritance via subclassing. A new class can extend an existing class, adding new fields and adding or redefining methods.

A subclass **D**, of class **C**, maybe be used in contexts expecting an object of class **C** (e.g., in method calls).

This is supported rather easily—objects of class **D** always contain a class **C** object within them.

If **C** has a field **F** within it, so does **D**. The fields **D** declares are merely *appended* at the end of the allocations for **C**.

As a result, access to fields of **C** within a class **D** object works perfectly.

Handling Multiple Scopes

Many languages allow procedure declarations to nest. Java now allows classes to nest.

Procedure nesting can be very useful, allowing a subroutine to directly access another routine’s locals and parameters.

Run-time data structures are complicated because multiple frames, corresponding to nested procedure declarations, may need to be accessed.

To see the difficulties, assume that routines *can* nest in Java or C:

```
int p(int a){
   int q(int b){
      if (b < 0)
           q(-b);
      else return a+b;
   }
   return q(-10);
}
```

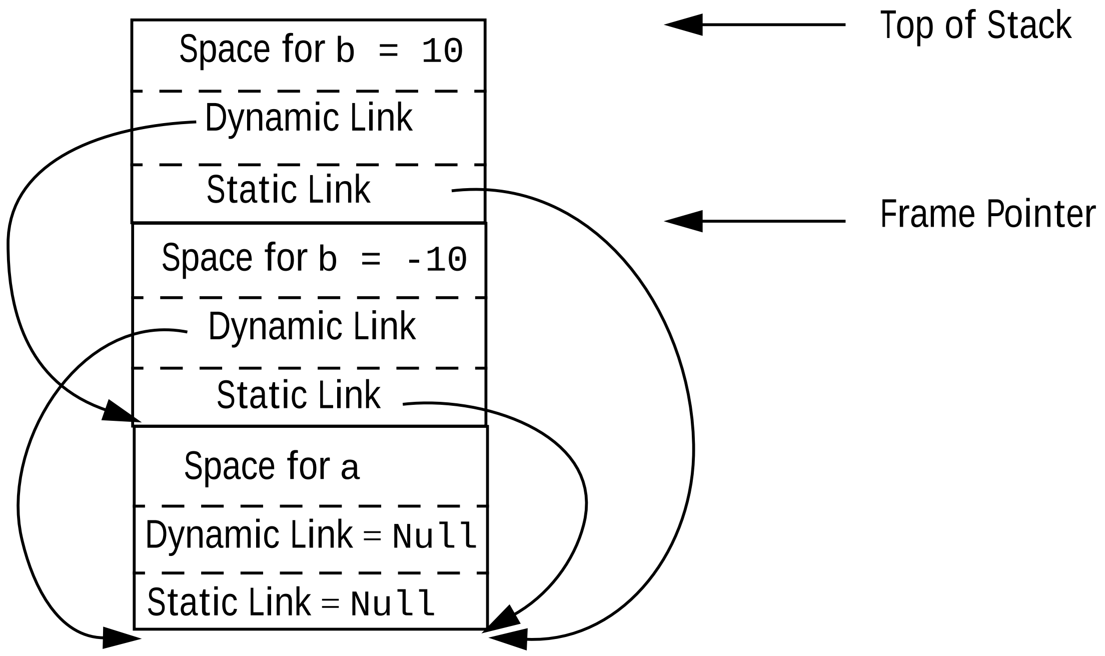
When **q** executes, it can access not only its own frame, but also that of **p**, in which it is nested.

If the depth of nesting is unlimited, so is the number of frames that must be made accessible. In practice, the level of nesting actually seen is modest—usually no greater than two or three.

Static Links

Two approaches are commonly used to support access to multiple frames. One approach generalizes the idea of dynamic links introduced earlier. Along with a dynamic link, we’ll also include a *static link* in the frame’s control information area. The static link points to the frame of the procedure that statically encloses the current procedure. If a procedure is not nested within any other procedure, its static link is **`null`**.

The following illustrates static links:



As usual, dynamic links always point to the next frame down in the stack. Static links always point down, but they may skip past many frames. They always point to the most recent frame of the routine that statically encloses the current routine.

In our example, the static links of both of **q**’s frames point to **p**, since it is **p** that encloses **q**’s definition.

In evaluating the expression **a+b** that **q** returns, **b**, being local to **q**, is accessed directly through the frame pointer. Variable **a** is local to **p**, but also visible to **q** because **q** nests within **p**. **a** is accessed by extracting **q**’s static link, then using that address (plus the appropriate offset) to access **a**.

Displays

An alternative to using static links to access frames of enclosing routines is the use of a *display*.

A display generalizes our use of a frame pointer. Rather than maintaining a single register, we maintain a *set of registers* which comprise the display.

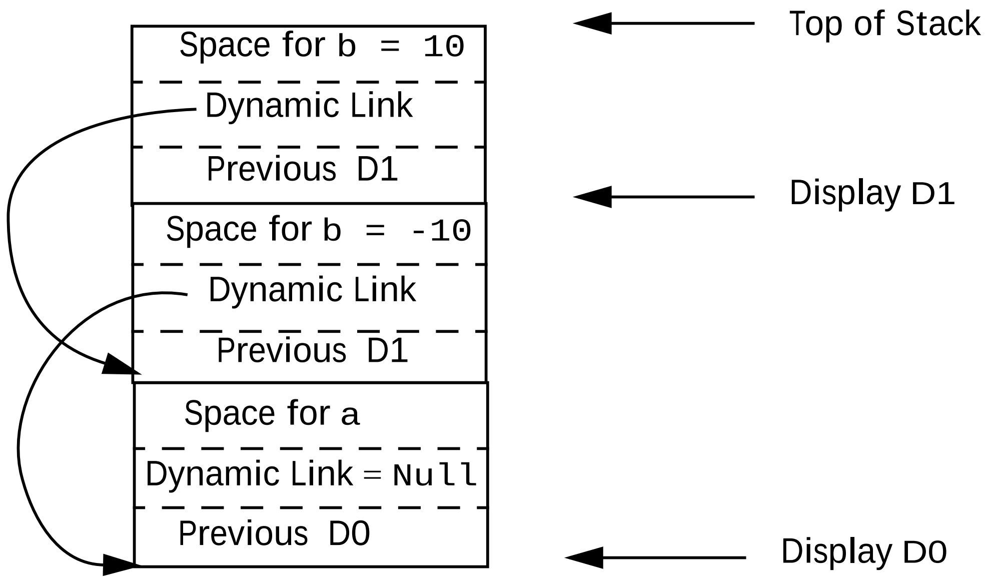
If procedure definitions nest *n* deep (this can be easily determined by examining a program's AST), we need *n+1* display registers.

Each procedure definition is tagged with a nesting level. Procedures not nested within any other routine are at level 0. Procedures nested within only one routine are at level 1, etc.

Frames for routines at level 0 are always accessed using display register **D0**. Those at level 1 are always accessed using register **D1**, etc.

Whenever a procedure **`r`** is executing, we have direct access to **`r`**’s frame plus the frames of all routines that enclose **`r`**. Each of these routines must be at a different nesting level, and hence will use a different display register.

The following illustrates the use of display registers:



Since **q** is at nesting level 1, its frame is pointed to by **D1**. All of **q**'s local variables, including **b**, are at a fixed offset relative to **D1**.

Since **p** is at nesting level 0, its frame and local variables are accessed via **D0**. Each frame's control information area contains a slot for the previous value of the frame's display register.

A display register is saved when a call begins and restored when the call ends. A dynamic link is still needed, because the previous display values doesn’t always point to the caller’s frame.

Not all compiler writers agree on whether static links or displays are better to use. Displays allow direct access to all frames, and thus make access to all visible variables very efficient. However, if nesting is deep, several valuable registers may need to be reserved. Static links are very flexible, allowing unlimited nesting of procedures. However, access to non-local procedure variables can be slowed by the need to extract and follow static links.