Stack Allocation

Modern programming languages allow recursion, which requires *dynamic allocation*.

Each recursive call allocates a *new copy* of a routine's local variables.

The number of local data allocations required during program execution is not known at compile-time.

To implement recursion, all the data space required for a method is treated as a distinct data area that is called a *frame* or *activation record*.

Local data, within a frame, is accessible only while a subprogram is active.

In mainstream languages like C, C++ and Java, subprograms must return in a stack-like manner—the most recently called subprogram will be the first to return.

A frame is pushed onto a *run-time stack* when a method is called (activated).

When it returns, the frame is popped from the stack, freeing the routine's local data.

As an example, consider the following C subprogram:

```
p(int a) {
   double b;
   double c[10];
   b = c[a] * 2.51;
}
```

Procedure **p** requires space for the parameter **a** as well as the local variables **b** and **c**.

It also needs space for control information, such as the return address.

The compiler records the space requirements of a method.

The *offset* of each data item relative to the beginning of the frame is stored in the symbol table.

The total amount of space needed, and thus the size of the frame, is also recorded.

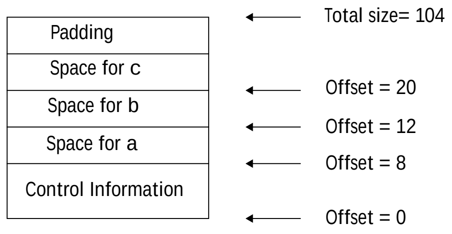
Assume **p**'s control information requires 8 bytes (this size is usually the same for all methods).

Assume parameter **a** requires 4 bytes, local variable **b** requires 8 bytes, and local array **c** requires 80 bytes.

Many machines require that word and doubleword data be *aligned*, so it is common to pad a frame so that its size is a multiple of 4 or 8 bytes.

This guarantees that at all times the top of the stack is properly aligned.

`Here is p`’s frame:



Padding
Space for c
Space for b
Space for a
Control Information

Total size= 104
Offset = 20
Offset = 12
Offset = 8
Offset = 0

Within **p**, each local data object is addressed by its offset relative to the start of the frame.

This offset is a fixed constant, determined at compile-time.

We normally store the start of the frame in a register, so each piece of data can be addressed as a **(Register, Offset)** pair, which is a standard addressing mode in almost all computer architectures.

For example, if register **R** points to the beginning of **p**’s frame, variable **b** can be addressed as **(R,12),** with **12** actually being added to the contents of **R** at run-time, as memory addresses are evaluated.

Normally, the literal **`2.51`** of procedure **p** is not stored in **p**’s frame because the values of local data that are stored in a frame disappear with it at the end of a call.

It is easier and more efficient to allocate literals in a *static area*, often called a *literal pool* or *constant pool*. Java uses a constant pool to store literals, type, method and interface information as well as class and field names.

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.

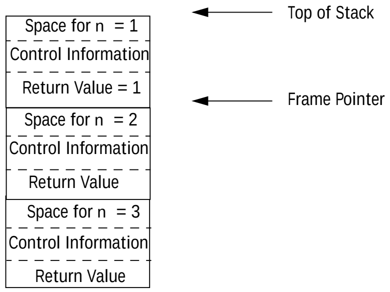
Instead a distinct register, often called the *frame pointer*, is used to access the current frame.

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:



Space for n = 1
Control Information
Return Value = 1
Space for n = 2
Control Information
Return Value
Space for n = 3
Control Information
Return Value
Top of Stack
Frame Pointer

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.

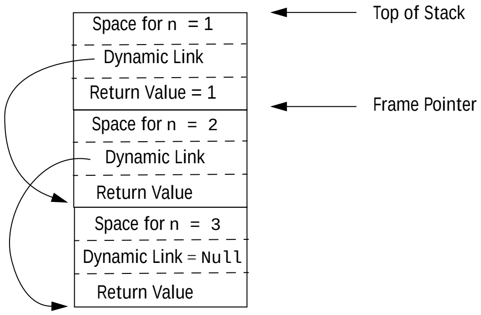
In simple cases this is done by adjusting the frame pointer by the size of the current frame.

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:



Space for n = 1
Dynamic Link
Return Value = 1
Space for n = 2
Dynamic Link
Return Value
Space for n = 3
Dynamic Link = Null
Return Value
Top of Stack
Frame Pointer

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:

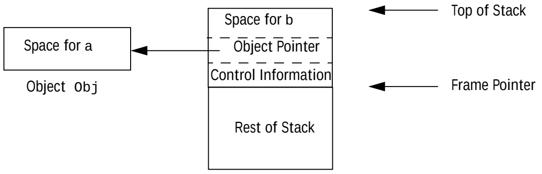
```
class K {
  int a;
  int sum(){
     int b;
  return a+b;
} }
```

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**.



Space for a

Object `Obj`

Space for b

Object Pointer

Control Information

Rest of Stack

Top of Stack

Frame Pointer

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);
}
```

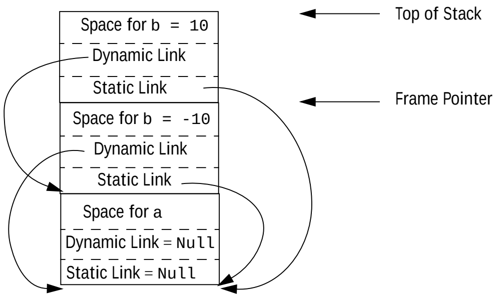
When **q** executes, it can access not only its own frame, but also that of **p**, in which it is nested.

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**.