CS 012 • Handout 2

A Brief Introduction To Stacks And Function Activation Frames

wagner@cs.ucr.edu
Department of Computer Science and Engineering
University of California, Riverside

January 2, 2003

Data Structures: Stacks

Suppose you've written a program which defines and uses an array of a fixed size and, of course, of a given type. Say, for instance, it's an array of *char*'s of size 10.[1]

The interesting property of an array is that you can directly set and retrieve *any* of its elements in a single operation; if the array is named A, then the assignment $A[7]$ = 'P' sets the 8-th element to the character 'P'.[2] However, if you needed to search the array for a given character, you'd have to check *every* element until you find the one you're looking for, unless the array is sorted, in which case you can use a faster search algorithm called *binary search*. In general, however, you're stuck with a search that may take as many steps as the size of the array.

This informal analysis shows that we can characterize an array by the kind and speed of the operations we can perform on it: access to any element (an operation known as *random access*) is fast, but searching an array is comparatively expensive, potentially taking as many steps as the size of the array. Element deletion is equally expensive, since we may have to relocate a large portion of the array elements after the deletion.

An array is an example of a *data structure*: an arrangement of memory cells for which a very specific set of operations is defined, some of which may be fast and others which may

[1]Since the array's type and size are both known at compile time, the compiler is able to generate code to allocate enough memory for the entire array, namely, 10 bytes in our example (recall that a *char* takes up a single byte).

[2]Not the 7-th element, because array indices start at 0.

be slow.

Another example of a data structure is a *stack*, whose behavior is what one would expect from its name. Just as you add or take a tray only from the top of the stack of trays in the cafeteria, a stack data structure lets you insert a new element onto its top or retrieve (from the top) the last element inserted. In the data structure jargon, the insertion operation is called a *push* and the retrieval operation is called a *pop*. The important thing to remember about a stack is that you can only insert or retrieve at the top of the stack; all other elements are *not* accessible until they are exposed at the top by successive pops.

From the definition above, it's clear that pushing and popping are very fast operations. But, like an array, searching the stack requires you to pop every element until you find the one you're looking for. It might seem, therefore, that a stack is very similar in behavior to an array. Not so! An array allows fast random access, whereas random access in a stack is as slow as searching it.

Building a stack

How does one actually build a stack? There are different ways to implement a stack, each with its own advantages and disadvantages. I'll present here the simplest one: an array implementation.

Suppose, then, that you have a fixed-size array `stack` of, say, characters. In order to build a stack data structure from this array, we also need to define an integer, `top`. For simplicity, suppose the array has size 5. Initially, `top` will have a value of -1, indicating that the stack is empty. As we push elements into the stack or pop elements from it, `top` will change accordingly, either increasing or decreasing. When `top` equals (size of the array minus 1), we'll know then that the stack is full and no more pushes should be allowed. Similarly, if we perform enough pops, `top` will reach -1 again, indicating that the stack is empty and we should not allow any more pops from it.

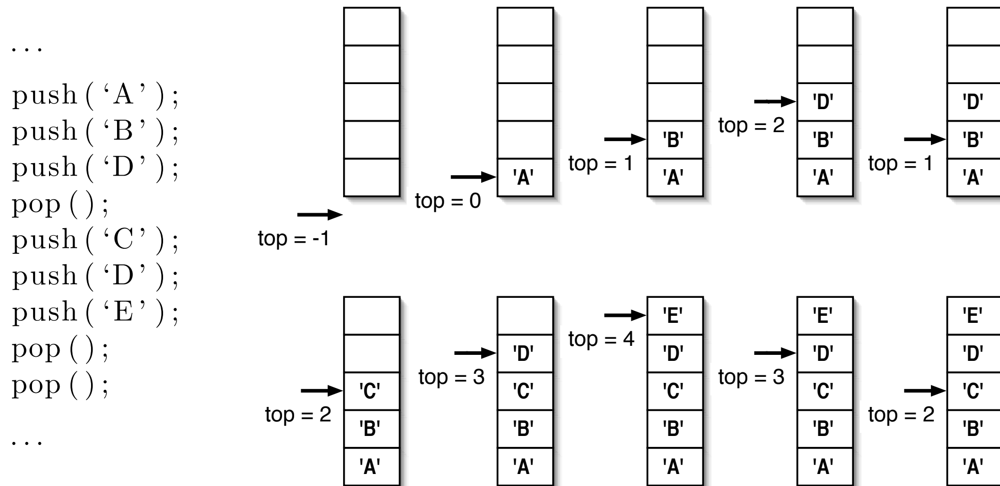
The actual operations `push` and `pop` can be defined as follows:[3]

```
void push(char c)
{
    top++;
    stack[top] = c;
}
```

```
char pop()
{
    char c = stack[top];
    top--;
    return c;
}
```

[3]Actually, these functions should test for stack-empty and stack-full conditions but, for simplicity, we'll ignore those tests here.

As an example, the figure below shows the result of applying the following sequence of pushes and pops:



Note that popping the stack does *not* clear the array entry at the top of the stack prior to popping it. Deleting that entry is never necessary and would be a waste of time, since `top` keeps track of where the stack really ends.

Activation Frames

What are stacks useful for? One of the most important uses for a stack is in implementing the mechanism by which local variables are instantiated within a block of code and also the mechanism by which function calls work. Before we discuss these mechanisms in detail, however, we need to review how memory is allocated on a typical computer.

Memory layout

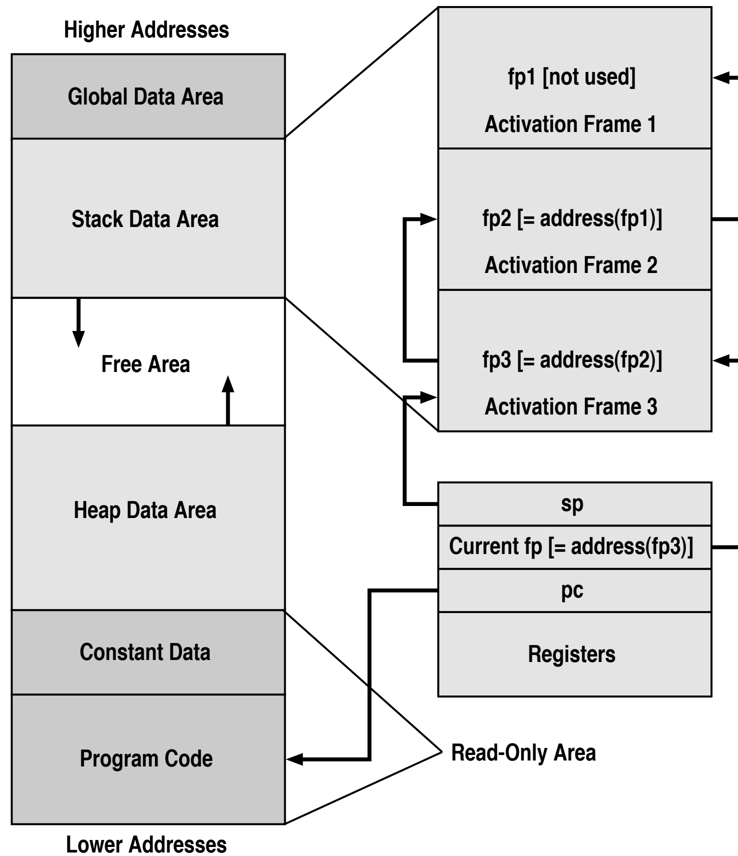
Typically, there are five distinct sections of memory which are used by a running program:

- **Read-only** data area: this area of memory is where both the program's executable code and all its constants are stored;
- **Global/static** data area: this area is used to store the program's global variables;
- **Stack/local** data area: the contents of this area change as the program enters and exits code blocks and executes function calls;

- **Heap/dynamic** data area: this is the area of memory where dynamic arrays and other dynamic data structures (such as objects in object-oriented languages) are instantiated on demand;

- **Register/temporary** data area: this area is available for fast access and for specialized uses. In particular, this is the area of memory where the program counter, the stack pointer, and the current frame pointer are stored (more on these below).

All of these areas are allocated by the operating system immediately prior to starting execution. Graphically, at some point during the program's execution, these different memory regions are typically laid out as in the figure below:



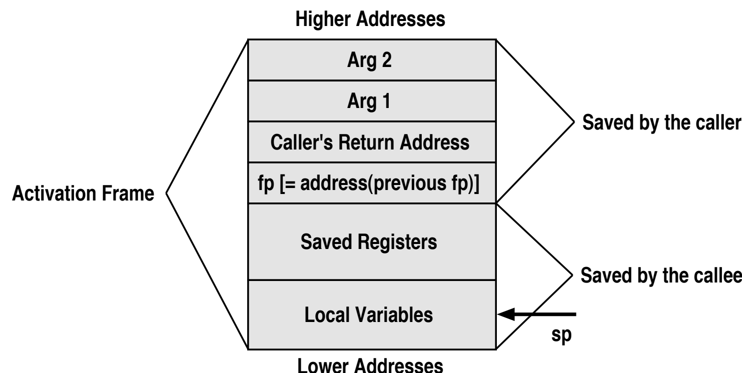
Note that the sizes of the global data, constant data, and program code areas are all known at compile time, so the compiler knows how much memory to allocate for those areas.

The stack and heap areas, due to their dynamic nature, are of variable size so the best the compiler can do is allocate a reasonable chunk of memory for both areas combined. It's the programmer's responsibility to manage the program's use of heap and stack memory to prevent an *out-of-memory* condition. Note also how the stack is set-up: it grows towards lower addresses.

A separate area of memory stores register data, the program counter (pc), the stack pointer (sp), and the current frame pointer (fp). In this simple handout, we will not talk much about registers. The program counter is the address of the next program instruction to execute, and the stack pointer is the address of the current top of the stack.

Frame Pointers

Every time the program execution encounters a function call, a structure called an *activation frame* is appended to the stack area. This structure is filled partially by the block calling the function and partially by the function being called. The figure below illustrates a typical activation frame.



The calling block pushes into the stack first the arguments to the function being called (typically in reverse order), and then a *return address*, that is, the address of the next instruction in the program's code after the instruction which prompted the function call (this is calculated from the program counter pc). This is done so that, when the function finishes executing and returns, the processor knows which instruction to execute next. The calling block also pushes into the stack the address in memory of the *previous* activation frame pointer. This is necessary because a function may call another function which may call yet another function and so on, and there must be a way to return from all these nested calls. Each of these calls is associated with its own activation frame, and the frames are linked in a chain, each pointing to its predecessor.

Once this preliminary information is stored into the activation frame, execution is transferred to the function being called. However, before the function's code is actually executed, the current register values are also added to the activation frame, again by pushing those values into the stack. Now the processor can freely execute the code for the function because the current state has been saved and will be restored when the function returns.

As the function executes, the processor can easily access the arguments to the function — they are located at addresses above fp. In addition, local variables defined by the function are also added to the activation frame and can be accessed equally as easily, this time at addresses below fp. It's the responsibility of the stack pointer (sp) to keep track of which of these variables is currently being accessed. When the function finishes executing and is about to return, all local variables are popped off the stack, the saved register values are also popped and stored back into the registers, the program counter is set to the return address stored in the activation frame, and the current frame pointer is updated to point to the same address as that stored in the activation frame's fp.

Thus, in the first figure above, when the function associated with activation frame 3 is about to return, the current fp variable is set to the address of fp2. Similarly, when the function associated with the activation frame 2 returns, the current frame pointer is set to the address of fp1.

Recursive function calls

The discussion above makes no mention of what kind of function calls we're dealing with and, so, it applies equally as well to *recursive calls*, which are function calls which call themselves.

Typically, the number of *non*-recursive nested calls in a program is small which means that the overhead of having to save and restore all that information, both in terms of memory used and in terms of processing speed, is not a major concern.

For recursive calls, however, care must be taken to ensure that the recursion doesn't go too deep, especially if memory is tight. That's why, if possible, recursive functions should be converted to non-recursive ones (that is, *iterative* functions). Sometimes that conversion is possible, other times it is not. The conditions under which this conversion is possible will be discussed in class.