Computer Science 1MC3

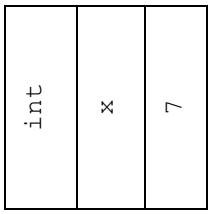
Lab 8 – Stack Frames and the Heap

A function's stack holds it parameters, return addresses, and local variables. A stack is used by the computer to allocate memory when running a program.

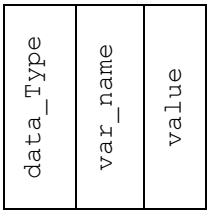
The computer will allocate memory from the top of the stack to the bottom of the stack, this is called first in last out (filo). To conceptualize this imagine making a pile of books. You can only take the top book off the pile.

Simple Stack Frames

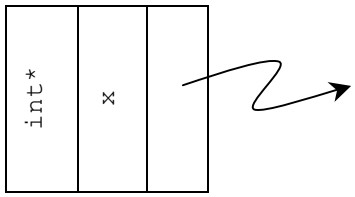
Let us consider the trivial declaration `int x=7;` the stack frame for this would look like:



In general a stack frame is:



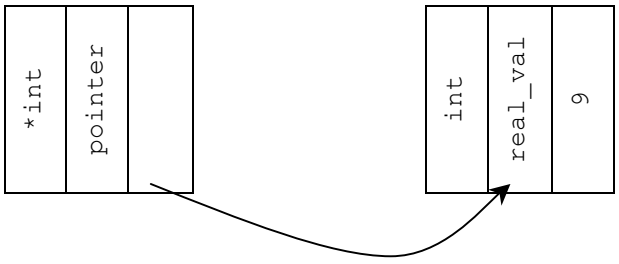
The value of a pointer would just be an arrow. For instance `int *x;`



A curved arrow implies that the pointer has not been initialized to anything yet.

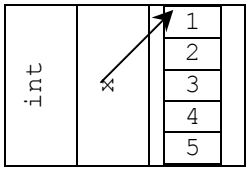
However if we had:

```
int real_val=9;
int *pointer;
pointer=&real_val
```



Arrays

Stack frames for arrays are a little bit different. It is necessary to make one box for every value in the array. For instance `int x[]={1,2,3,4,5}`



Please note that x is a pointer to the first array element.

Abstract Data Types

We will demonstrate how to make stack frames of abstract date types from the following complex example:

```
struct computer {

      int height;
      int width;
      char[] make;

}

typdef struct computer compType;

compType boxArray[]={ {1,4"IBM}, {7,2"Apple} };
```

compType	boxArray	compType array index 0	int	height	1
			int	width	4
			char	make →	"IBM" *technically this should be an array*
		compType array index 1	int	height	7
			int	width	2
			char	make →	"Apple" *should also technically be an array*

Now it is possible to make a pointer to any element in this stack. So say we wanted a pointer to boxArrays first element. We could do it like this:

```
compType *pointer;

pointer=&compType[1];
```

Would give you a stack frame:

*compType	pointer	

So when making a stack frame we just start off with the basic `dataType|name|value` frame and continually fill in the `value` column with either another frame or a value. This definition is recursive in the sense that we are filling a column, in a column, in a column . . . in a frame.

Dynamic Allocation On The Heap

At some point in your computer science careers it may be necessary to declare an array without knowing what size it has to be in the first place. This is what we call a dynamic array, it is dynamic in the sense that we can manipulate the length of it at run time. To do this we use `malloc`, a function which will reserve some memory on the **heap**. The heap is a lot like the stack but highly unorganized; stack memory is linear, with memory cells right after each other, whereas the heap has giant holes (like Swiss cheese).

In order to declare some space on the memory we will require a pointer to reference it. So if we would like to declare an integer array of a dynamic size we would do as follows:

```
int *arrayname;

arrayname = (int*)malloc(desired_array_size*sizeof(int));
```

In general:

```
data_type *pointer;

pointer = (data_type*)malloc(deserired_length*sizeof(data_type));
```

Please note that great caution is required when using the heap. The heap is often filled with values from other programs. You will not receive errors for going out of bounds on an array or even worse you may forget to initialize a variable and then call it and get a value anyways. This causes great headaches when debugging code.

Homework

1. Create the stack frame for:

```
struct car {

      int year;
      int model_num;
      char[] company;

}

typdef struct car carType;

carType carsArray[]={ {2001,1112,"Toyota"}, {1999,1254,"BMW"} };
```

2. Create the stack frame for a pointer to the BMW car.

3. What is FILO? Explain. Can you figure out what FIFO means?

4. Create two functions. Function 1 will return the length of the first word in a sentence and Function 2 will return the first word. You will need to dynamically allocate memory for a character array in the second function.

```
int firstWordLength (char *sentence);

char* firstWord (char *sentence);
```