Computer Architecture: Take IV

Examples by Jerry, Nick, and Julie.

A simple function

The simple **Add** function takes two integers as parameters, computes their **sum** in a local variable, and then returns that value.

```
static int Add(int one, int two)
{
   int temp, sum;

   sum = one + two;
   return sum;
}
```

The activation record is 20 bytes total: 8 bytes for the parameter block, 4 bytes for the return address, and 8 bytes for the local variables.

Add AR

(total size 20 bytes)

		type	size	offset
	two	int	4	16
	one	int	4	12
	Saved PC	address	4	8
	temp	int	4	4
SP ->	sum	int	4	0

left to right

top to bottom
right to left

Doing the most straightforward translation (no concern for optimization), the generated code for the body of **Add** will look like this:

prior to RET, SP points to return address; RET sets PC to it and SP = SP + 4

```
SP = SP - 8          ; make space for local variables
R1 = M[SP + 12]      ; load value of parameter one into R1
R2 = M[SP + 16]      ; load value of parameter two into R2
R3 = R1 + R2         ; do addition
M[SP] = R3           ; store result in local variable sum
RV = M[SP]           ; copy sum's value into RV register (return value)
SP = SP + 8          ; clean up space used for local variables
RET                  ; return to caller, pick up at saved address
```

You could change the C code to not bother with the local variables (**temp** isn't even used, and **sum** isn't necessary), but in fact, a smart optimizing compiler can already recognize they aren't needed and remove them for you. When a value is used only temporarily, it is likely to exist only in a register and never be written to the stack at all:

```
                        ; eliminate all local variables, no change to SP
    R1 = M[SP + 4]      ; load value of parameter one into R1
    R2 = M[SP + 8]      ; load value of parameter two into R2
    RV = R1 + R2        ; compute sum directly into RV register
    RET                 ; no need to clean up locals, there aren't any!
```

The Calling Function

What kind of code is generated to make a function call? The caller has responsibility for making space for the parameters and assigning their values, along with saving the return state and transferring control. The function **Caller** has no parameters, one local integer, and calls the **Add** function from the previous example:

```
static void Caller(void)
{
   int num = 10;

   num = Add(num, 45);
   num = 100;
}
```

Caller AR

(total size 8 bytes)

		type	size	offset
	Saved PC	address	4	4
SP ->	num	int	4	0

Here is the code generated for the **Caller** function, note how it handles the call to **Add**.

```
SP = SP - 4         ; make space for local variable num

M[SP] = 10          ; assign local variable num constant value 10


R1 = M[SP]          ; load up the value of num (before we change SP
                    ; so we don't have to deal with changed offsets)

SP = SP - 8         ; push space for parameter block of Add's AR
M[SP + 4] = 45      ; initialize parameters in the activation record
M[SP] = R1

CALL <Add>          ; the CALL instruction makes space on the stack
                    ; for the return address, saves the PC value there
                    ; and then assigns the PC to the address of the first
                    ; instruction of the Add fn (which transfers control)

SP = SP + 8         ; when control returns here, pop the params off stack
M[SP] = RV          ; read return value from RV and store in num

M[SP] = 100         ; assign num constant value 100

SP = SP + 4         ; clean up storage for locals
RET                 ; return, no value stored in RV since fn has no return
```

Although the caller makes space for parameters, it does not allocate space for the locals. This is because as the outside caller, you don't know how much space is required for the local variables of some other random function. For example, when calling **strlen**, how can you tell how many bytes it needs for local variables? It is the job of the **callee** to allocate that space. When **Add** starts executing, the first thing it does is push space onto the stack for the locals, without initializing any values in that space (see code on previous page).

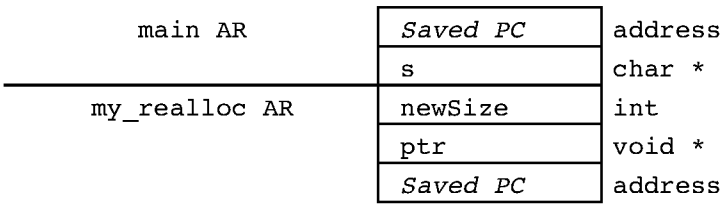
A pointer parameter

Note that parameters in C are passed by value, so a copy of the contents are made on the stack. Any changes made to the parameter in the context of the callee function are lost when control returns to the caller. When we want to change a value within a called function, we pass the address of the value and then the callee code dereferences the pointer to access and change the data.

But be careful about what is by reference and what is by value! Note that the pointer itself is still a normal parameter that is copied to the stack and changes to the pointer inside the calling function will only affect the local copy. For example, think about why you have to catch the return value from **realloc**. Why doesn't **realloc** just change the pointer you passed as a parameter when it is necessary to move the data? Here's a sketch of that scenario:

```
void main(void)
{
   char *s = malloc(100);
   my_realloc(s, 200);
   // after return, s still holds same address as before
}

void my_realloc(void *ptr, int newSize)
{
   ptr = malloc(newSize);  // why isn't this enough to change the pointer?
}
```



It's important to understand why assigning to **ptr** in the **my_realloc** function has no effect on the local variable **s** back in **main**. To get this to work, we would need to pass the pointer itself by reference into the function so that it can reach back to the local variable and change its contents:

```
void main(void)
{
   char *s = malloc(100);
   my_realloc(&s, 200);
                // after return, s now holds new address
}

void my_realloc(void **ptr, int newSize)
{
   *ptr = malloc(newSize); // follow param back to change
}
```

now can derefence and change the bits in the s block of main AR

A struct parameter

The **Binky** function takes a **struct** as a parameter and has a local **struct** declared on the stack. What does the activation record look like for **Binky**? What code will be generated to set the denominator fields in the parameter **struct** and local **struct** variable?

```
struct fraction {
   int numerator;
   int denominator;
};


static void Binky(struct fraction param)
{
   struct fraction local;

   local.denominator = 1;
   param.denominator = 2;
}
```

Binky AR

(total size 20 bytes)

		type	size	offset
	param.den	int	4	16
	param.num	int	4	12
	Saved PC	address	4	8
	local.den	int	4	4
SP ->	local.num	int	4	0

```
SP = SP - 8        ; make space for local variable

M[SP + 4] = 1      ; set local.denominator = 1
M[SP + 16] = 2     ; set param.denominator = 2

SP = SP + 8        ; clean up locals
RET                ; return to caller, no return value stored
```

The calling function

How does a **struct** get passed as a parameter? Like other parameters in C, it is passed by value, so a copy is made on the stack. The function **Caller** has no parameters, a local **struct fraction**, and calls **Binky** from the previous example:

```
static void Caller(void)
{
   struct fraction actual;

   Binky(actual);
}
```

Caller AR

(total size 12 bytes)

		type	size	offset
	Saved PC	address	4	8
	actual.denom	int	4	4
SP ->	actual.num	int	4	0

Here is the code generated in **Caller** to set up for the call to **Binky**. Note that **structs**, like all C parameters, are passed by value. A complete copy of the **struct** is made and copied to the stack.

```
SP = SP - 8     ; make space for local variable (left uninitialized)
R1 = M[SP]      ; store the value of the two fields of the struct "actual"
R2 = M[SP + 4]  ; (before we change the SP which will make things messy)


SP = SP - 8     ; push space for the parameter block of Binky's AR

M[SP + 4] = R2  ; initialize the parameter in the AR. This means
M[SP] = R1      ; copying both fields. For a small struct, it is possible
                ; to store the fields temporarily in register(s) and then
                ; copy to stack in steps. However, for larger structures,
                ; it will be necessarily to store the address of the
                ; structure in a register and then use a specialized
                ; copy function (something like memcpy) to copy its
                ; contents to the stack. Something similar is done when
                ; returning a struct as the return value.
Call <Binky>
SP = SP + 8     ; clean up parameters

SP = SP + 8     ; clean up locals

RET             ; return to caller, no return value stored
```

Passing **structs** as parameters (or as return values) is usually quite expensive. A copy can't easily be made if the **struct** is larger than a register. It is often preferred to pass structures by address (even when you don't intend to modify them) to avoid this

expense. The **const** modifier can be used to show you don't intend the contents to be modified.

A function with a local array

An example with an array declared locally. Note that all the memory for the array elements is allocated as space on the stack when you use this type of declaration.

```
static void Apple(void)
{
   int i;
   short scores[4];

   scores[i] = 10;
}
```

Apple AR

(total size 16 bytes)

no scores pointer variable on stack

		type	size	offset
	Return addr	address	4	12
	i	int	4	8
	scores[3]	short	2	6
	scores[2]	short	2	4
	scores[1]	short	2	2
SP ->	scores[0]	short	2	0

Here is the code for the **Apple** function, the most interesting part being the function body which assigns the constant **10** to the **i**th member of the scores array:

```
SP = SP - 12          ; make space for locals (left uninitialized)

R1 = M[SP + 8]        ; load value of i into R1
R2 = R1 * 2           ; multiply i by size of element (short = 2 bytes)
R3 = SP + R2          ; add offset to base address of scores array
M[R3] =.2 10          ; assign array element, copy only 2 bytes!

SP = SP + 12          ; clean up stack
RET                   ; return to caller
```

Looking at the above activation record, do you see why it is not possible to do something like this?

```
scores = (short *) malloc(sizeof(short) * 25);
```

no scores pointer variable on stack

How much more expensive would it be to allocate a **1000** member array instead of just **4**? How does this compare to using **malloc** to allocate an array?

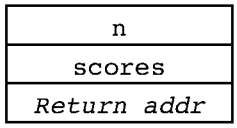
A function with an array parameter

All arrays in C are passed by reference, thus **Banana** here will receive the base address of the array as its first parameter.

```
static void Banana(short scores[], int n)
{
   scores[n] = 10;
}
```

Banana AR

(total size 12 bytes)



	type	size	offset
n	int	4	8
scores	short *	4	4
Return addr	address	4	0

Here is the code for the **Banana** function, which assigns the constant **10** to the **n**th member of the scores array. Note how it is different (minutely but importantly) from the similar code in the **Apple** function:

```
                        ; no locals, so no need to make space
R1 = M[SP + 8]          ; load n into R1
R2 = R1 * 2             ; multiply n by size of element (short = 2 bytes)
R3 = M[SP + 4]          ; load base address of scores array
R4 = R3 + R2            ; add offset to base address
M[R4] =.2 10            ; assign array element, copy only 2 bytes!
RET
```

Notice that having just the base address of the array stored in our activation record is different that having the array itself stored in the activation record. In this example, it would be legal to reassign where **scores** points with this code:

```
scores = (short *) malloc(sizeof(short) * 25);
```

scores pointer variable on stack

What does this line of code do? Does you see why this is legal here but wasn't in the previous example?

Also worth mentioning is that you can declare an array parameter with a lot of different notation:

```
static void Banana(short scores[], int n)
static void Banana(short *scores, int n)
static void Banana(short scores[10], int n)
```

But all of these generate exactly the same code, have the same behavior, and the same activation record. There is absolutely no difference in them, the notations by convention indicate how the argument will be used in the function body.

There is one slightly obscure way to make a copy of certain arrays when passing them as a parameter, do you see what it is?

Passing an array as a parameter

What if the previous **`Apple`** function made a call to **`Banana`**?

```
static void Apple(void)
{
   int i;
   short scores[4];

   Banana(scores, i);
}
```

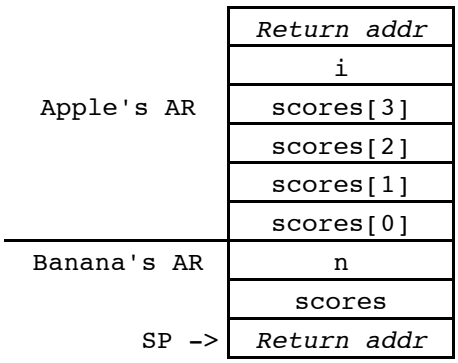
Code for the **`Apple`** function, now with a call to **`Banana`**:

```
SP = SP - 12          ; make space for locals (left uninitialized)

R1 = M[SP + 8]        ; load i into R1
R2 = SP               ; load base address of scores array
SP = SP - 8           ; make space for params to Banana function
M[SP + 4] = R1        ; assign second param
M[SP] = R2            ; assign first param
CALL <Banana>         ; jump to Banana function
SP = SP + 8           ; clean up parameter block from Banana

SP = SP + 12          ; clean up locals
RET
```

While this code is executing, the stack will look like this:



Using a function pointer

When you pass a function pointer as a parameter, it is literally tracking the address of the sequence of instructions for the named function. Setting up for the function call is the same as for named function, the generated code uses the activation record as described by the declared type of the function pointer. At the call instruction, control goes to the passed address, rather than a specific named function. For example, this function compares two unknown things via a function pointer and returns true or false based on the returned result:

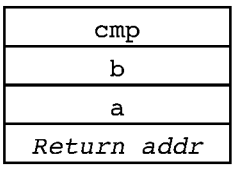
use typedef for function pointer to shorten the length of parameter description

```
typedef int (*compareFn)(const void *, const void *);

static bool AreEqual(const void *a, const void *b, compareFn cmp)
{
   if (cmp(a, b) == 0)      // not the most compact way to write this
      return true;          // but simplest to generate code for
   else
      return false;
}
```

AreEqual AR

(total size 16 bytes)



		type	size	offset
	cmp	fn ptr	4	12
	b	void *	4	8
	a	void *	4	4
SP ->	*Return addr*	address	4	0

Code for `AreEqual` showing how it sets up and calls a function via pointer:

```
                        ; no locals, so no space created
R1 = M[SP + 8]          ; load b into R1
R2 = M[SP + 4]          ; load a into R2
R3 = M[SP + 12]         ; load cmp into R3

SP = SP - 8             ; make space for params of the fn
M[SP + 4] = R1          ; assign second param
M[SP] = R2              ; assign first param

CALL R3                 ; call to address stored in R3

SP = SP + 8             ; remove parameters when function returns
BNE RV, 0, PC + 12      ; if return value not zero jump to else
RV = 1                  ; assign return value to true
RET                     ; return to caller
RV = 0                  ; assign return value to false
RET                     ; return to caller
```

call the address of function, instead of symbolic figure

It is absolutely imperative that the function pointer passed to **`AreEqual`** matches the prototype of a **`compareFn`**. What will happen if it doesn't? What will happen if a **`NULL`** or an incorrect pointer is passed as the compare function?

Passing a function pointer as a parameter

Passing a function pointer as a parameter means taking the address of its compiled code and assigning the parameter to hold that address, nothing too complicated, actually.

```
int CompareStrings(const void *a, const void *b)
{
   return strcmp((char *)a, (char *)b);
}


static void Caller(void)
{
   int same;
   char *s, *t;

   same = AreEqual(s, t, CompareStrings);
}
```

function pointer: address to compiled code in memory

Caller AR

(total size 16 bytes)

		type	size	offset
	Return addr	address	4	12
	same	int	4	8
	s	char *	4	4
SP ->	t	char *	4	0

Code for **Caller** showing how it sets up and calls a function passing a function pointer:

```
SP = SP - 12              ; create space for locals (left uninitialized)
R1 = M[SP]                ; load t into R1
R2 = M[SP + 4]            ; load s into R2
R3 = <CompareStrings>     ; load address of CompareStrings fn into R3

SP = SP - 12              ; make space for params of AreEqual
M[SP + 8] = R3            ; assign third param (the function pointer)
M[SP + 4] = R1            ; assign second param
M[SP] = R2                ; assign first param

CALL <AreEqual>           ; call to AreEqual

SP = SP + 12              ; remove parameters when function returns
M[SP + 8] = RV            ; assign return value to local variable same
SP = SP + 12              ; clean up space used for locals
RET                       ; return to caller
```

A function with pointers and typecasts

Taking all of the things we earlier discussed about generating code for pointers and typecasts, we can make quite a nasty little function that puts it all together to test how well you understand it all:

```
struct person {
   int age, id;
   struct person *next;
};

static char Muppets(struct person bert, struct person *ernie)
{
   struct person **oscar;

   ((struct fraction *)bert.next)->denominator = 0;

   ernie = &bert;
   oscar = &ernie;

   (**oscar).next = ernie;
   return bert.age;
}
```

Muppets AR

(total size 24 bytes)

		type	size	offset
	ernie	person *	4	20
	bert.next	person *	4	16
	bert.id	int	4	12
	bert.age	int	4	8
	Return addr	address	4	4
SP ->	oscar	person**	4	0

As always, begin by making space for the locals:

```
SP = SP - 4          ; make space for oscar
```

Now, consider the code generated for the first line of C:

```
R1 = M[SP + 16]      ; retrieve bert.next
                     ; now that bert.next is safely tucked
                     ; into R1, pretend that R1 points
                     ; to a struct fraction
M[R1 + 4] = 0        ; note the offset of four needed to
                     ; access the denominator field of fraction
```

On to the next two lines of assignments:

```
R1 = SP + 20         ; compute address of ernie
R2 = SP + 8          ; compute base address of bert
M[SP + 20] = R2      ; store bert's address in ernie
```

```
M[SP] = R1          ; store ernie's address in oscar
```

Now for the next line... Deep breath.

```
R1 = M[SP]          ; load oscar, R1 now contains a struct
                    ; person **
R1 = M[R1]          ; deref, now R1 contains a struct person *
R2 = M[SP + 20]     ; store ernie in a register
M[R1 + 8] = R2      ; if R1 stores the address of a struct
                    ; person, then R1+8 is the address of
                    ; that struct's next field.  We store
                    ; to that field.
```

And finally, the return statement:

```
R3 =  M[SP + 8]     ; load bert.age field
RV =.1 R3           ; copy lowest byte into return value
SP = SP + 4         ; clean up space used for locals
RET                 ; return control to caller
```

Note that amount of code generated is not proportional to the amount of thinking and drawing you need to do in order to arrive at it!