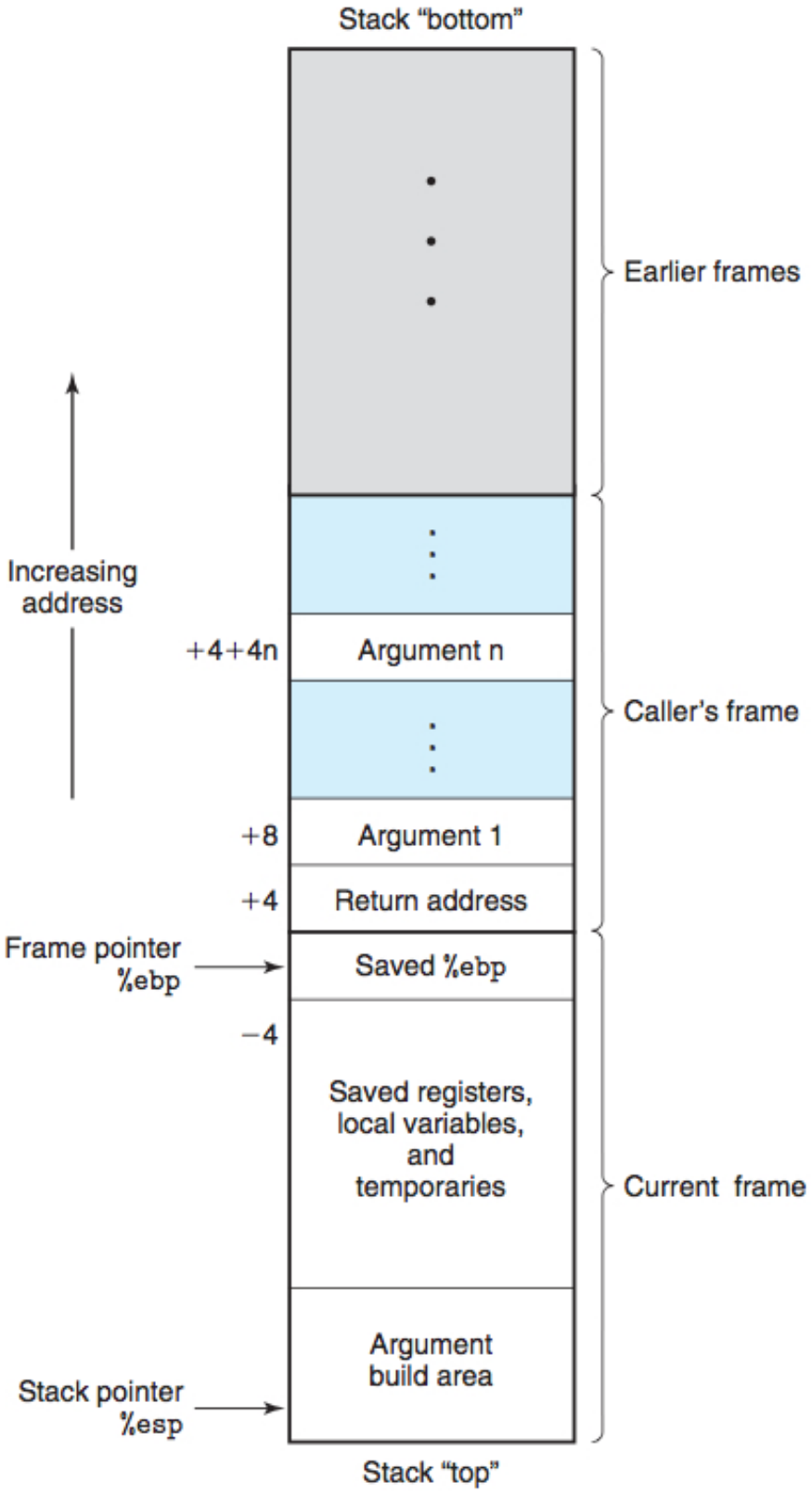
Procedure calls and Stack Frames

In assembly language, we refer to function calls as **procedures**. IA-32 makes use of the stack to pass arguments (parameters) to procedure calls. The portion of the stack allocateds for a single procedure call is called the **stack frame**. Every time execution changes context via a procedure call, the stack is grown and a new stack frame is generated. The two registers, %ebp and %esp index the current stack frame. %ebp is the **frame pointer** and points to the bottom of the current stack frame. %esp is the **stack pointer** and points to the top of the current stack frame.



control is transferred to a new procedure by the *call* instruction. To make a direct call, the instruction is issued with the label for the procedure call. To make an indirect call, we use the * operator with an address or register. The call instruction pushes the return address onto the stack and then moves execution to the start of the called procedure. The *ret* instruction returns control from a procedure call. it pops the return address off the stack and jumps to the instruction stored by the proceeding call. We use register %eax for return values.

Register Usage Conventions

Because we have only a limited number of regsiters to work with and share among procedure calls, we need to make sure that when one procedure, *the caller* makes a call to another procedure *the callee*, the callee does not overwrite data saved in a register that the caller planned on using later:

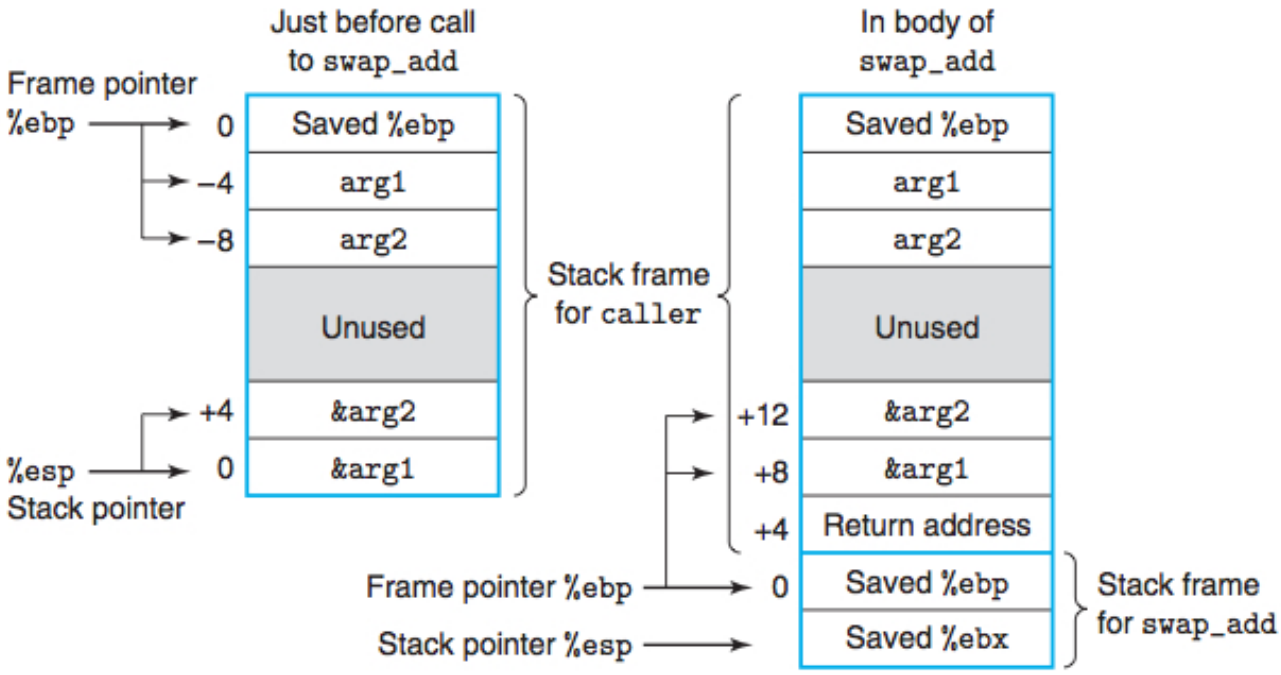
```
Caller:

    ... do something  ...
    movl %eax, %ebx          // copy some value into %ebx
    call callee
    imull $4,%ebx            // %ebx will have been changed by callee!!


Callee:

  ...  do stuff ...
 movl $0, %ebx
  ... do other stuff ...

  ret
  leave
```

To prevent this situation, we follow a set of register use conventions for the general purpose registers:

- Caller-Save Registers: %eax, %edx and %ecx. The callee may overwrite these registers. The caller must either not store information needed after the call in these registers or save the information to the stack prior to the call if the information is needed later.
- Callee-Save Registers: %ebx, %esi and %edi. The callee must save the values in these regsiters to the stack prior to overwriting them and must restore them before returning control to the caller.

Putting it all together:Example-

Take a look at our C code: swap_add.c and the same code in IA-32: swap_add.s



Example:

```
Given a C function named fun with the following code body:

   *p = d;
    return x-c;

and the following IA32 implementation:

   movsbl       12(%ebp), %edx
   movl         16(%ebp), %eax
   movl         %edx, (%eax)
   movswl       8(%ebp), %eax
   movl         20(%ebp), %edx
   subl         %eax, %edx
   movl         %edx, %eax

what is the C function prototype?
  ? fun(?, ?, ?, ?)

Answer:

   int fun(short c,char d,int *p, int x)
```

Arrays

Notice for any data type T and an integer constant N, the declaration:

T A[N]

produces

1. an allocation of L * N bytes of *contiguous* memory where L is the size (in bytes) of data type T.
2. An identifier (variable) A that points to the beginning of the array.

Example: Complete the table for the following array declarations:

```
short S[7];
short *T[3];
short **U[6];
long double V[8];
long double *W[4]
```

Array	Element Size	Total Size	Start Addr	Element i
S	2	14	s_0	s_0+2i
T	4	12	t_0	t_0+4i
U	4	24	u_0	u_0+4i
V	12	96	v_0	v_0+12i
W	4	16	w_0	w_0+4i

Remember operand specifiers the form: (R_a, R_b, s)?

This is a natural way to access array elements in IA32:
movl (%edx, %ecx, 4), %eax // %edx points the first element in the array, %ecx holds the value of *i* and the scaling factor corresponds to the element size

Pointer Airthmetic

In C, arithmetic on pointers is inplicitly scaled according to the size of the data type. If *p* is a pointer to type *T* and the value of *p* is *x_p*, then expressions using pointer arithmetic are evaluated as such:

p+i = x_p+*Li** where *L* is the size of data type *T*

Example:Given the starting address of array E and our indexing integer i are stored in registers %edx and %ecx, we evaluate pointer expressions in IA32 assembly as follows:

Expression	Type	Value	IA32
E	int *	x_e	movl %edx, %eax
E[0]	int	M[x_e]	movl (%edx), %eax
E[i]	int	M[x_e+4i]	movl (%edx, %ecx, 4), %eax
&E[2]	int *	x_e+8	leal 8(%edx), %eax
E+i-1	int *	x_e+4i-4	leal -4(%edx, %ecx, 4), %eax
*(E+i-3)	int *	M[x_e+4i-12]	movl -12(%edx, %ecx, 4), %eax
&E[i]-E	int	i	movl %ecx, %eax

Two-dimensional arrays

For arrays of the form: **T D[R][C];**, we access element D[i][j] as follows:

&D[i][j]= x_d + L(C * i + j) where L is the size in bytes of type T and C is the column height.

Heterogeneous Data Structures

Remember in C structs allow us to create custom data types that group objects of any data type into a single type. When memory for a struct object is allocated, the components are stored in a contiguous region of memory. the value of a struct variable is actually a pointer to the first byte of this memory region.

```
struct  rectangle {
  int i;
  int j;
  int a[3];
  int *p;
};
```

```
     rectangle g;

      g--------------> Offset  0        4       8                                20
                      Contents [    i   |   j   |   a[0]   |   a[1]   |   a[2]   |   p   ]
```

Given: struct rectangle *r is in register %edx, what does the follow code do?

```
    movl        (%edx), %eax
    movl        %eax, 4(%edx)
```

r->j=r->i