Example 1 - C like language

```
bool test( float a, int b )
{
    return a < b;
}
int f()
{
    float big = 0;
    int small = 100;
    int count = 0;
    bool b = test( big, small);
    while ( b ) {
        big = big * 1.1;
        int diff = int(big)
        small = small - diff;
        count = count + 1;
        b = test(big, small);
    }
    return count;
}
int main() { f(); }
```

Locals Collected for `test`

scope	id	type	size	offset	"padding"
1	a	float	8	0	0
1	b	int	4	8	0
total size 12					

It looks like the machine has 4B words, and double words for `float`.

Locals Collected for `f`

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					

It looks like the compiler is using 1B for `bools`.

`f`'s local data in symbol table

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					

`test`'s local data in symbol table

scope	id	type	size	offset	"padding"
1	a	float	8	0	0
1	b	int	4	8	0
total size 12					

Stack Setup for `f`→`test` Call

addr	contents	description
3044	`main`'s FP (address)	Nearish MAX MEMORY
3040	`main`'s RA (address)	Exec code address
3036	`f`'s <ret> value location	SP for `main` upon return
	4B padding	to d-word align `f`'s FP
3028	`f`'s `diff`	`diff` at (FP,20)
3024	`f`'s `b`	`b` at (FP,16)
3020	`f`'s `count`	`count` at (FP,12)
3016	`f`'s `small`	`small` at (FP,8)
3008	`f`'s `big`	`big` at (FP,0)
3004	`f`'s FP (address)	value of 3008
3000	`f`'s RA (address)	Exec code address
2996	`test`'s <ret> value location	SP for `f` upon return
	4B padding	to d-word align `test`'s FP
2988	`test`'s `b` parameter	`b` at (FP,8)
2984	`test`'s `a` parameter	`b` at (FP,0)

Recall that we've discussed in lecture how the **caller function** sets up a **callee function**'s **stack frame** with *control information* and room for a return value (pink); call parameters (green) and space for local variables as well (blue). Recall also we said this was a **pretty basic approach to stacks**.

`f`'s local data in symbol table

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					

`test`'s local data in symbol table

scope	id	type	size	offset	"padding"
1	a	float	8	0	0
1	b	int	4	8	0
total size 12					

Stack Setup for `f`→`test` Call

addr	contents	description
3044	`main`'s FP (address)	Nearish MAX MEMORY
3040	`main`'s RA (address)	Exec code address
3036	`f`'s <ret> value location	SP for `main` upon return
	4B padding	to d-word align `f`'s FP
3028	`f`'s `diff`	`diff` at (FP,20)
3024	`f`'s `b`	`b` at (FP,16)
3020	`f`'s `count`	`count` at (FP,12)
3016	`f`'s `small`	`small` at (FP,8)
3008	`f`'s `big`	`big` at (FP,0)
3004	`f`'s FP (address)	value of 3008
3000	`f`'s RA (address)	Exec code address
2996	`test`'s <ret> value location	SP for `f` upon return
	4B padding	to d-word align `test`'s FP
2988	`test`'s `b` parameter	`b` at (FP,8)
2984	`test`'s `a` parameter	`b` at (FP,0)

Recall that we've discussed in lecture how the **caller function** sets up a **callee function**'s **stack frame** with *control information* and room for a return value (pink); call parameters (green) and space for local variables as well (blue). Recall also we said this was a **pretty basic approach to stacks**.

Consider standard C library functions such as `qsort(3)` and `bsearch(3)` (their documentation is readily available online) — why won't this approach to stack setup work for these functions?

Better Stack Setup

Having the **caller** setup stack space for the **callee's** local variables won't work because in cases where the caller and callee are in two different source files (also called "translation units"), **the caller won't know how many local variables the callee requires space for**.

This is in fact **the typical case**.

The usual approach is the **caller** sets up **at least**

1. Stack space for the return value (communicated via SP value on the call)
2. (and maybe some) control information,

The **callee** (immediately upon entry) adjusts the FP and SP for local parameter storage (among other things perhaps).

When the callee `return`s, it pops its FP by the size of its frame (which it knows, of course), Sets SP=FP, and writes the return value on the stack (presuming the software architecture does not pass return values by register).