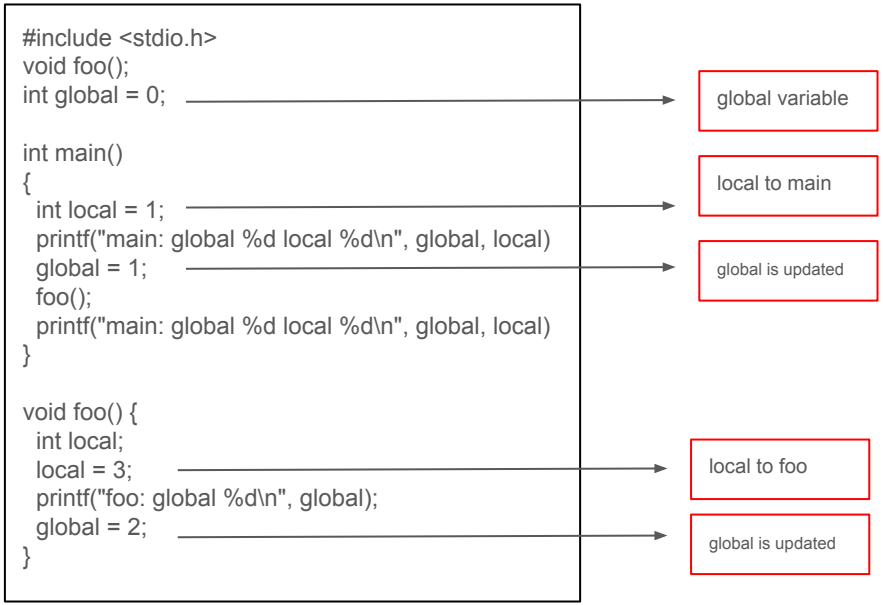
Data Types

int	integer (occupies at least 16 bits)
double	floating point (occupies at least 32 bits)
char	character (occupies at least 8 bits)

Different data types occupy different amounts of memory

Global Variables (Can access anywhere in the program)
Local Variables (Can access only in a particular region)



OUTPUT

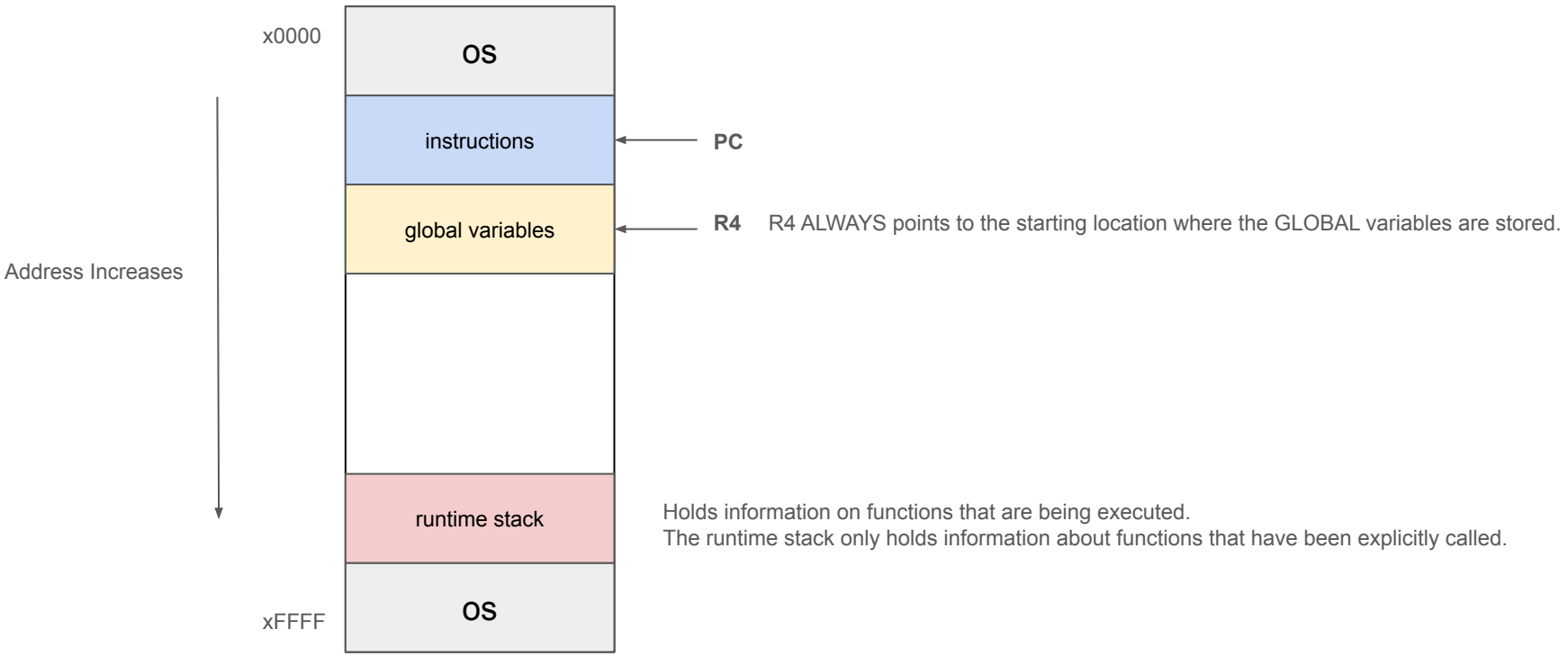
main: global 0 local 1

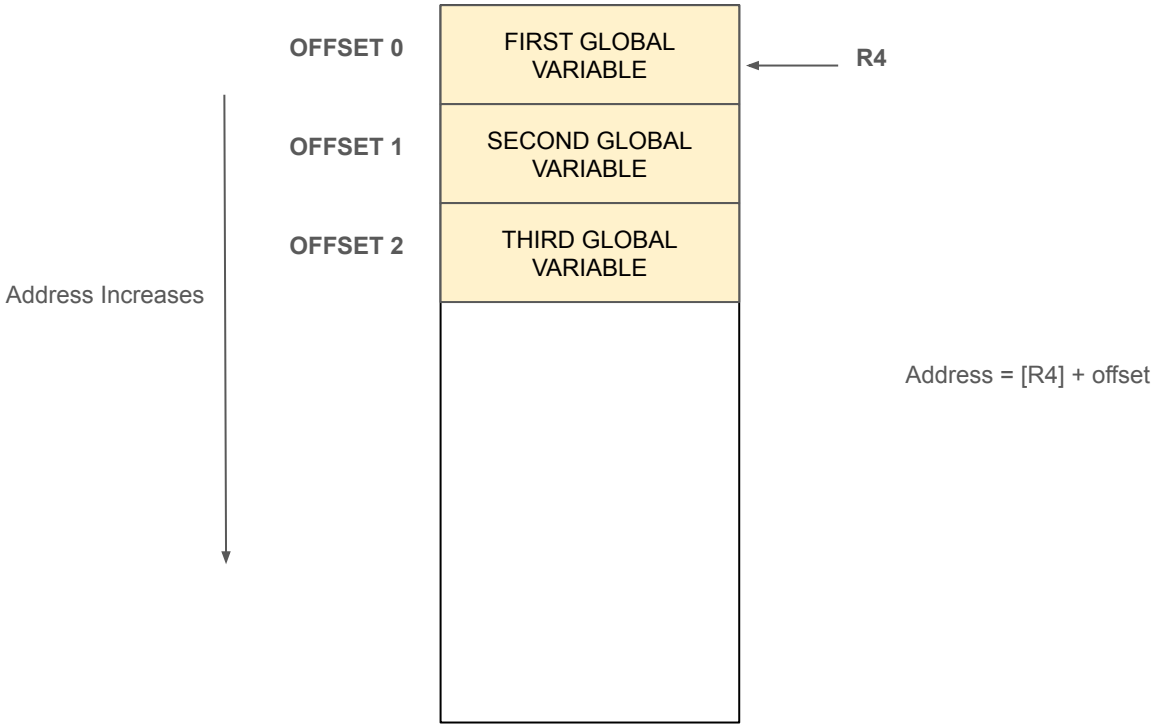
foo: global 1

main: global 2 local 1

Symbol Table

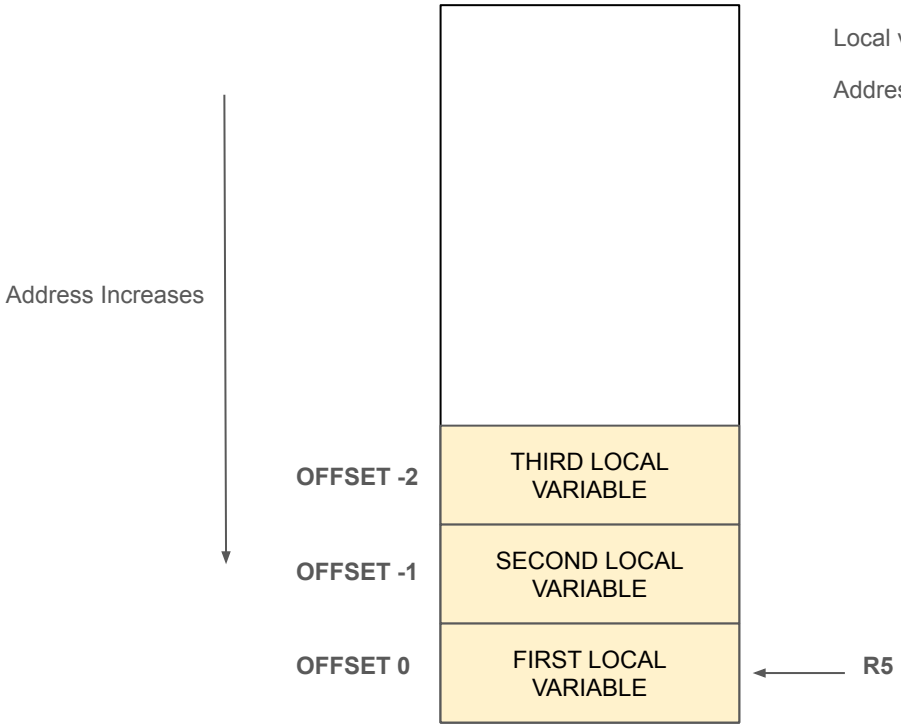
Records information about variables (Name of variables, their types, their Offset and their Scope)

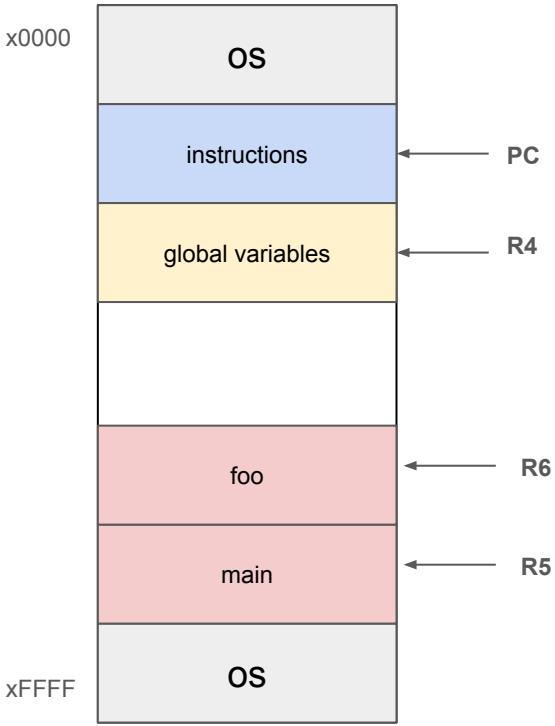

x0000
OS
instructions
PC
global variables
R4
R4 ALWAYS points to the starting location where the GLOBAL variables are stored.
Address Increases
runtime stack
Holds information on functions that are being executed.
The runtime stack only holds information about functions that have been explicitly called.
xFFFF
OS


OFFSET 0
FIRST GLOBAL VARIABLE
R4
OFFSET 1
SECOND GLOBAL VARIABLE
OFFSET 2
THIRD GLOBAL VARIABLE
Address Increases
Address = [R4] + offset

Local variables are stored in the stack frame.

Address = [R5] + offset






```
int main() {
  foo()
}

void foo() {
}
```

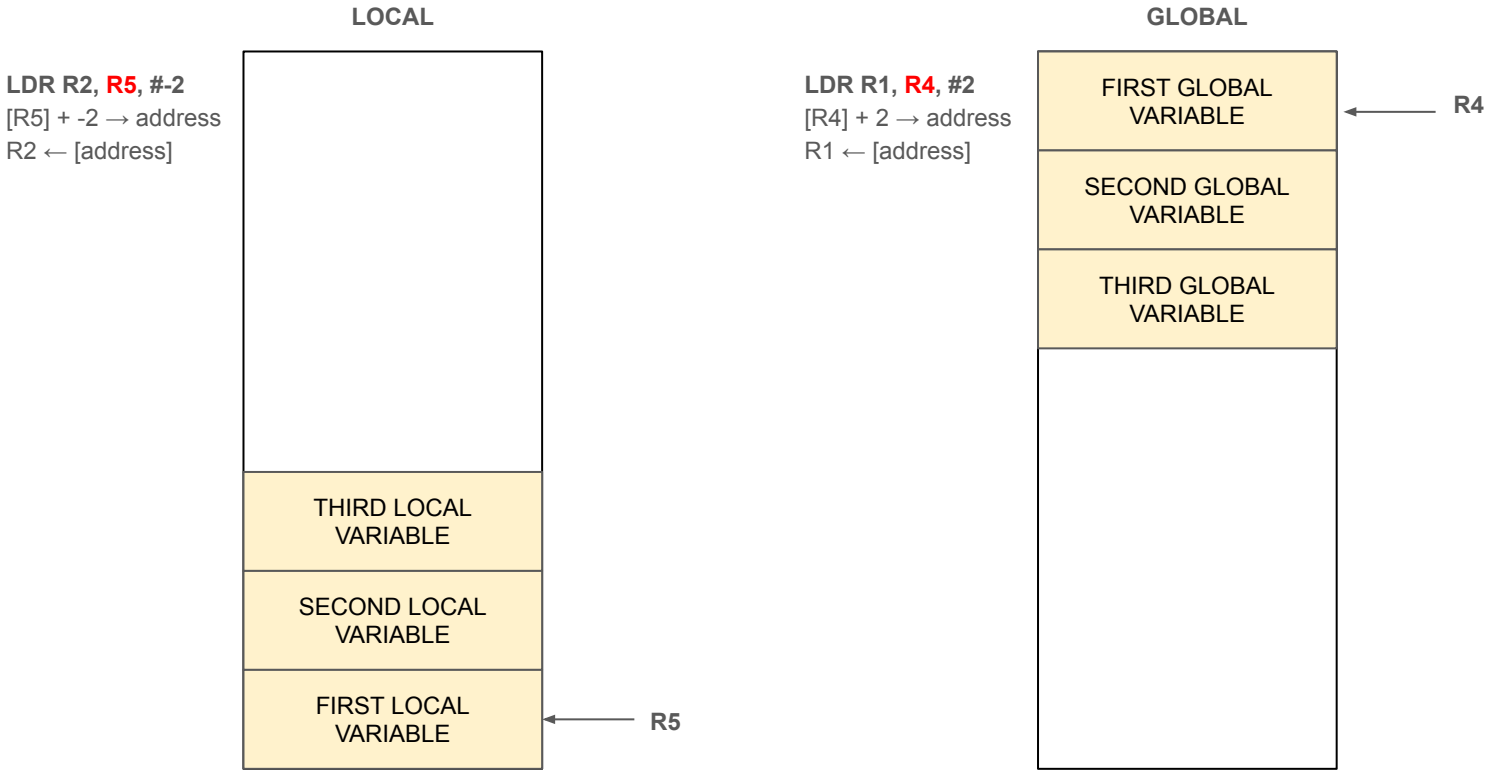
After function foo() has terminated, it is removed from the stack.

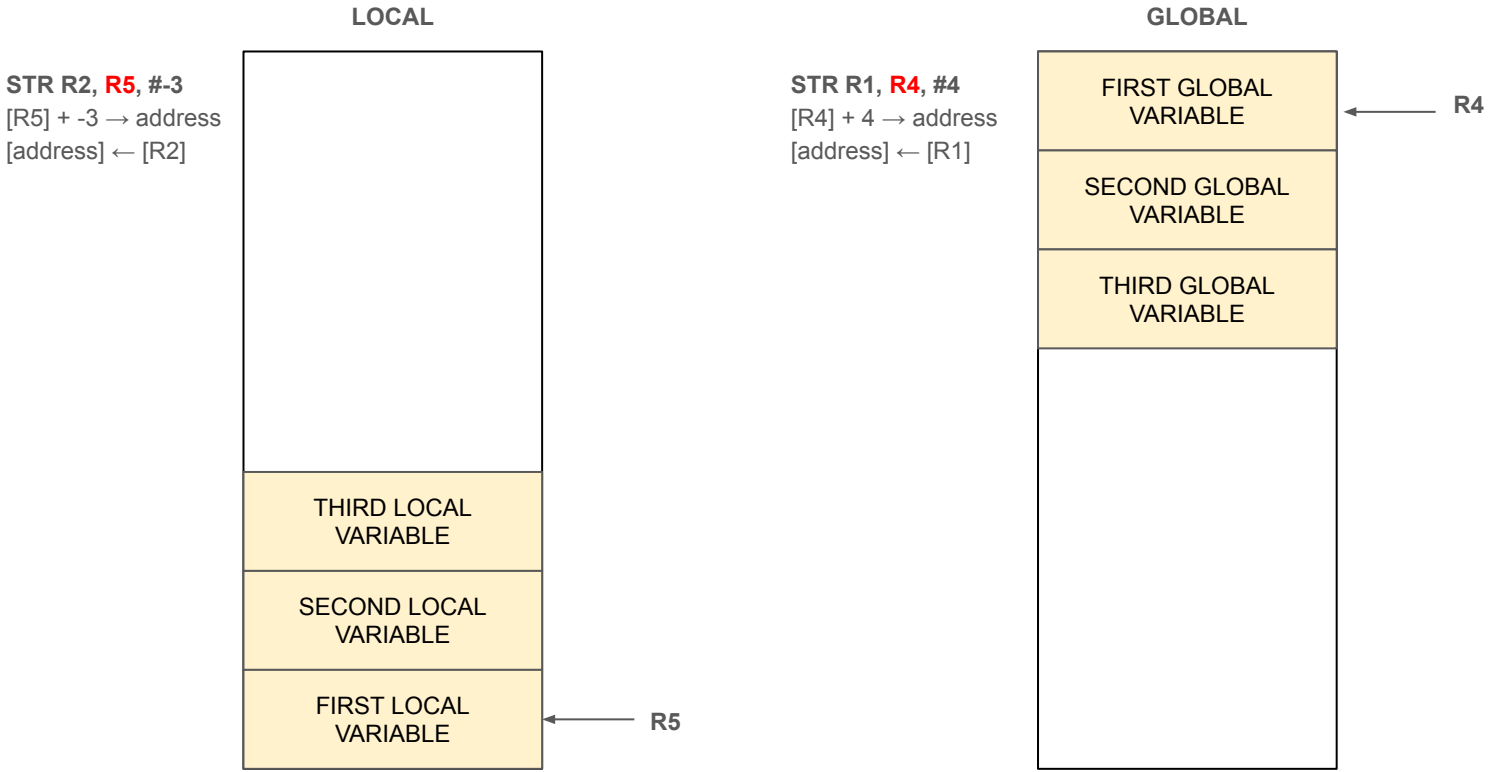
R6 points to the top of the runtime stack.
R5 points to the bottom of the stack frame of the current function.
Each function has their own stack frame.

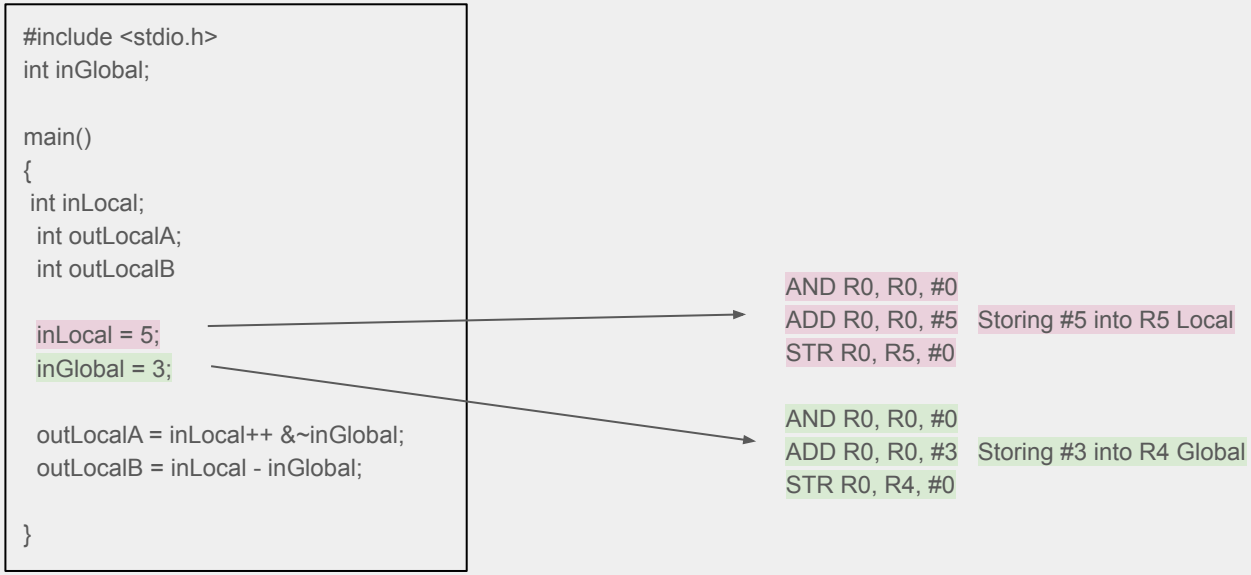
```
#include <stdio.h>
int inGlobal;

main()
{
 int inLocal;
  int outLocalA;
  int outLocalB

  inLocal = 5;
  inGlobal = 3;

  outLocalA = inLocal++ &~inGlobal;
  outLocalB = inLocal - inGlobal;

}
```

Name	**Type**	**Offset**	**Scope**
inGlobal	int	0	Global
inLocal	int	0	Main
outLocalA	int	-1	Main
outLocalB	int	-2	Main

LDR/STR


LOCAL
LDR R2, R5, #-2
[R5] + -2 → address
R2 ← [address]
THIRD LOCAL VARIABLE
SECOND LOCAL VARIABLE
FIRST LOCAL VARIABLE
R5
GLOBAL
LDR R1, R4, #2
[R4] + 2 → address
R1 ← [address]
FIRST GLOBAL VARIABLE
SECOND GLOBAL VARIABLE
THIRD GLOBAL VARIABLE
R4


LOCAL
STR R2, R5, #-3
[R5] + -3 → address
[address] ← [R2]
THIRD LOCAL VARIABLE
SECOND LOCAL VARIABLE
FIRST LOCAL VARIABLE
R5
GLOBAL
STR R1, R4, #4
[R4] + 4 → address
[address] ← [R1]
FIRST GLOBAL VARIABLE
SECOND GLOBAL VARIABLE
THIRD GLOBAL VARIABLE
R4


#include <stdio.h>
int inGlobal;
main()
{
int inLocal;
int outLocalA;
int outLocalB
inLocal = 5;
inGlobal = 3;
outLocalA = inLocal++ &~inGlobal;
outLocalB = inLocal - inGlobal;
}
AND R0, R0, #0
ADD R0, R0, #5 Storing #5 into R5 Local
STR R0, R5, #0
AND R0, R0, #0
ADD R0, R0, #3 Storing #3 into R4 Global
STR R0, R4, #0

b=1		
a = b++	POST-INCREMENT Assign current value of b first, then increase the value of b by 1.	a=1 b=2
a = ++b	PRE-INCREMENT First increase value of b by one, then add it to a.	a = 2 b=2

```
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;
  int outLocalA;
  int outLocalB

  inLocal = 5;
  inGlobal = 3;

  outLocalA = inLocal++ &~inGlobal;
  outLocalB = inLocal - inGlobal;

}
```

inLocal AND NOT(inGlobal)

```
LDR R0, R5, #0      Retrieve inLocal into R0
ADD R1, R0, #1      Increment inLocal
STR R1, R5, #0      Store inLocal in R5 / Update it

LDR R1, R4, #0      Retrieve inGlobal
NOT R1, R1          Bitwise NOT of inGlobal
AND R2, R0, R1      Bitwise AND with pre-increment inLocal and NOT inGlobal
STR R2, R5, #-1     Stores the result in inLocalA (offset -1)
```

```c
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;
  int outLocalA;
  int outLocalB

  inLocal = 5;
  inGlobal = 3;

  outLocalA = inLocal++ &~inGlobal;
  outLocalB = inLocal - inGlobal;

}
```

inLocal - inGlobal

```
LDR R2, R5, #0
LDR R3, R4, #0
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3

STR R2, R5, #-2
```

Retrieve inLocal into R2
Retrieve inGlobal into R3
Negative inGlobal
Increment inGlobal
Compute inLocal - inGlobal into R2

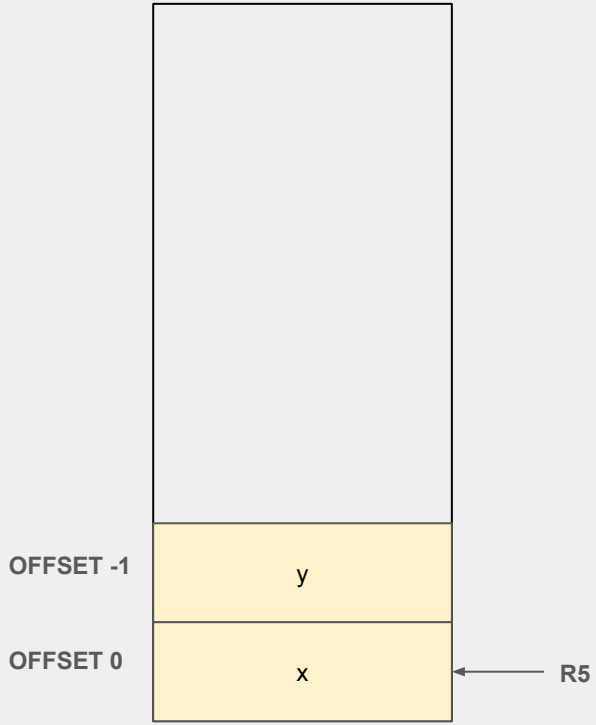
Store the result into outLocalB

2's Complement conversion of positive to negative

Conditional Statements

```
int main() {
int x, y;
if (x == 2) y = 5;
}
```

```
LDR R0, R5, #0
ADD R0, R0, #-2
BRnp NOT_TRUE

AND R1, R1, #0
ADD R1, R1, #5
STR R1, R5, #-1

NOT_TRUE
...
```

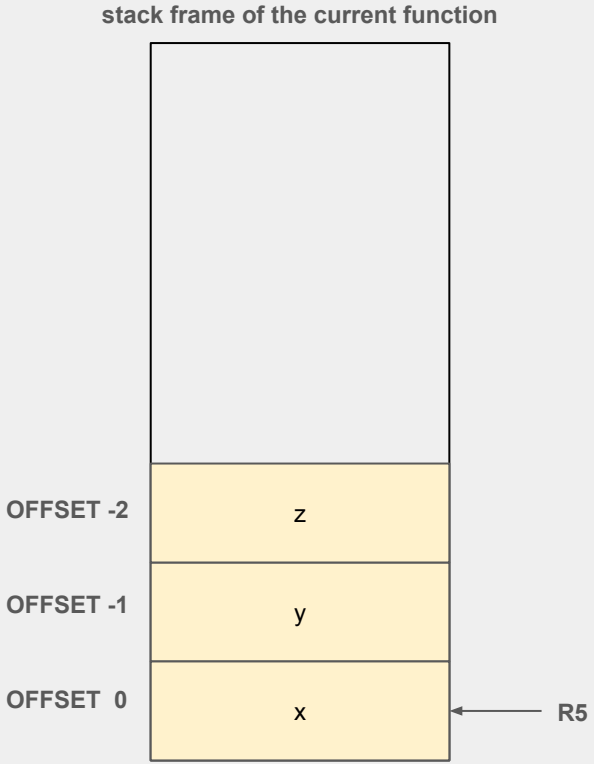


if (x)

- In C, an expression is considered true if it evaluates to a non-zero value.
- An expression is considered false if it evaluates to zero.

```
int x, y, z;
if (x) {
  y++;
  z--;
}
else {
  y--;
  z++;
}
```

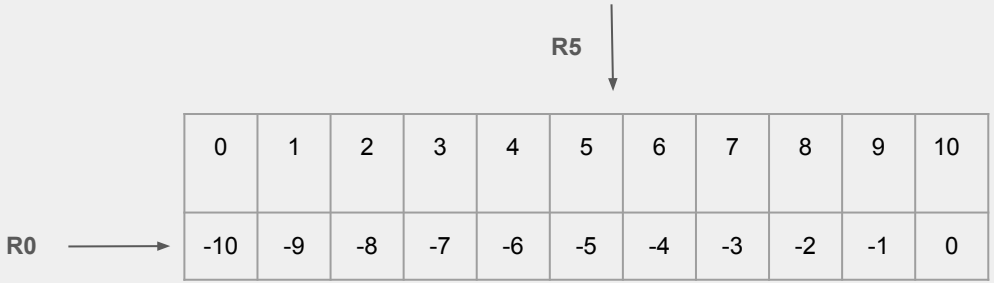
```
        LDR R0, R5, #0
        BRz ELSE

        LDR R1, R5, #-1
        ADD R1, R1, #1
        STR R1, R5, #-1
        LDR R1, R5, #-2
        ADD R1, R1, #-1
        STR R1, R5, #-2
        BRnzp DONE

ELSE    LDR R1, R5, #-1
        ADD R1, R1, #-1
        STR R1, R5, #-1
        LDR R1, R5, #-2
        ADD R1, R1, #1
        STR R1, R5, #-2
DONE
        ...
```

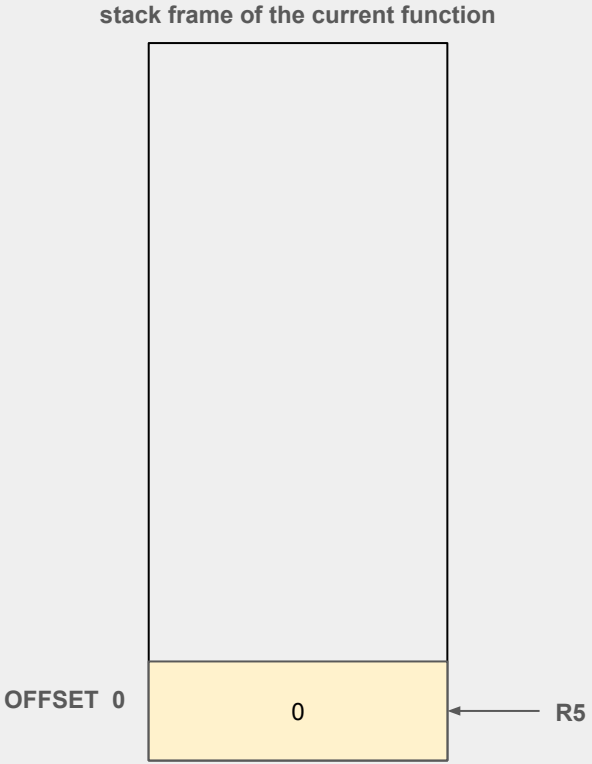


```
int x;
x = 0
while (x < 10) {
  x= x + 1
}
```

```
for (x = 0); x < 10; x++)
```

```
        AND R0, R0, #0
        STR R0, R5, #0

LOOP    LDR R0, R5, #0
        ADD R0, R0, #-10
        BRzp DONE

        LDR R0, R5, #0
        ADD R0, R0, #1
        STR R0, R5, #0
        BRnzp LOOP
DONE
```





Functions

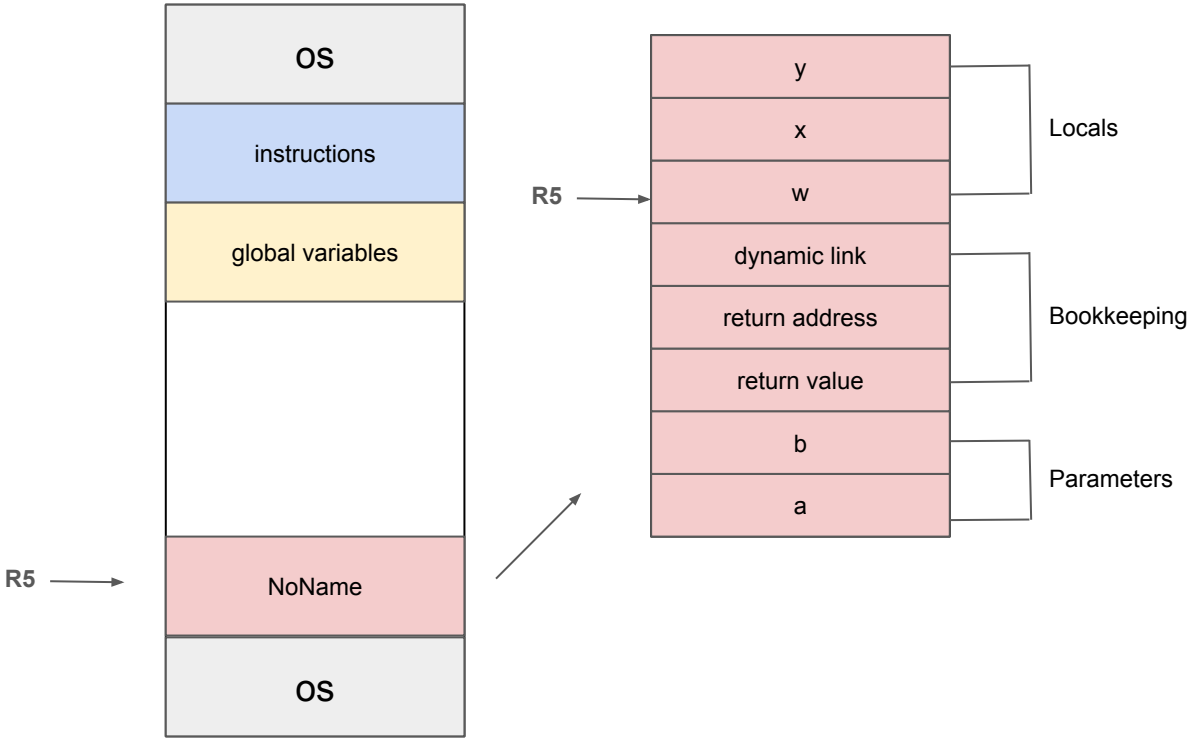
Zero or multiple arguments are passed in.

Single or No result is returned.

Return value is always a particular type, if there is no return value, the return type is void.

R6 ALWAYS points to the top of the runtime stack.

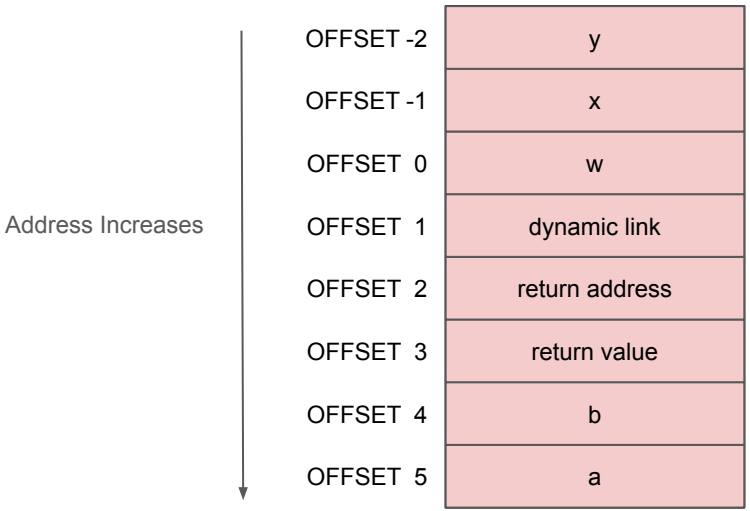
```
int NoName (int a, int b)
{
  int w, x, y;

  return ;
}
```



Dynamic Link
Address of stack frame for caller of function, so upon termination we can restore R5.

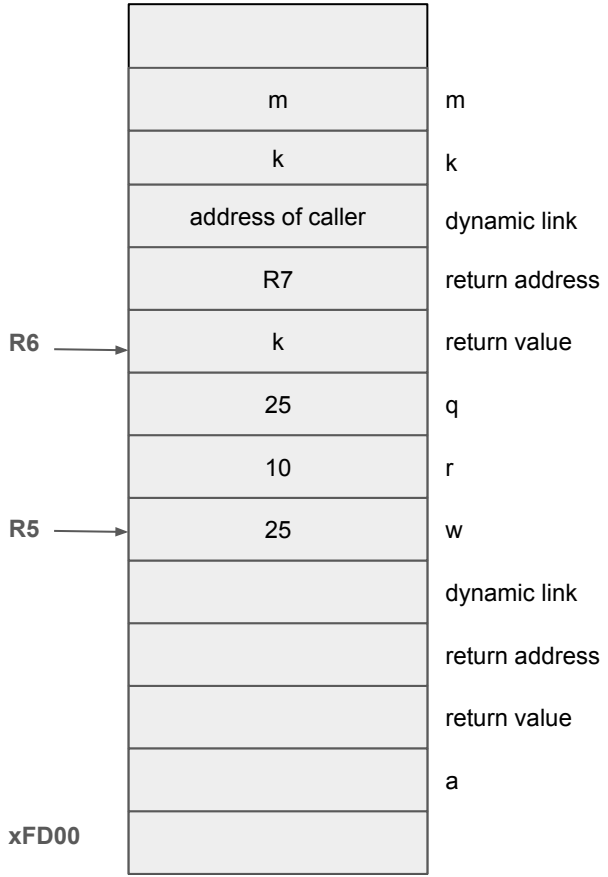
Return Address
Address in caller that we need to execute after the function terminates

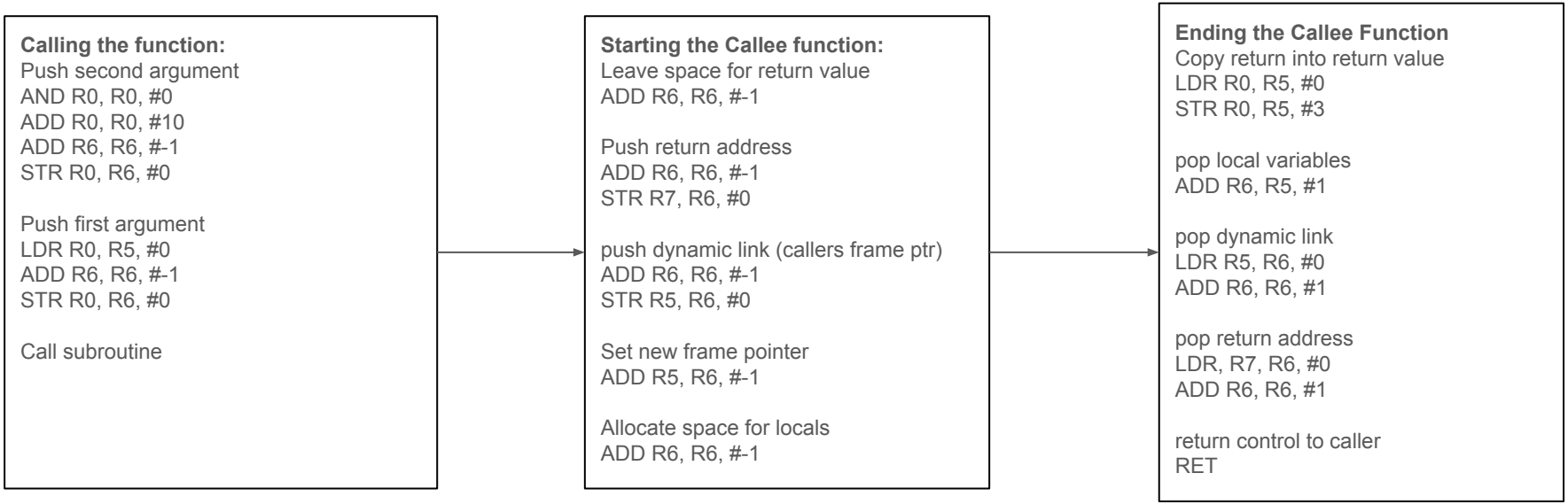
Return Value
Reserves location for holding a return value.


Address Increases
OFFSET -2
y
OFFSET -1
x
OFFSET 0
w
OFFSET 1
dynamic link
OFFSET 2
return address
OFFSET 3
return value
OFFSET 4
b
OFFSET 5
a

```
int function2 (int a) {
  int w;
  …
  w = function1(w, 10);
  …
  return w;
}

int function1 (int q, int r) {
  int k;
  int m;
  …
  return k;
}
```

```
AND R0, R0, #0
ADD R0, R0, #10
ADD R6, R6, #-1
STR R0, R6, #0

LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0

JSR function1

ADD R6, R6, #-1

ADD R6, R6, #-1
STR R7, R6, #0

ADD R6, R6, #-1
STR R5, R6, #0

ADD R5, R6, #-1

ADD R6, R6, #-1

LDR R0, R5, #0
STR R0, R5, #3

ADD R6, R5, #1

LDR R5, R6, #0
ADD R6, R6, #1

LDR, R7, R6, #0
ADD R6, R6, #1

RET
```



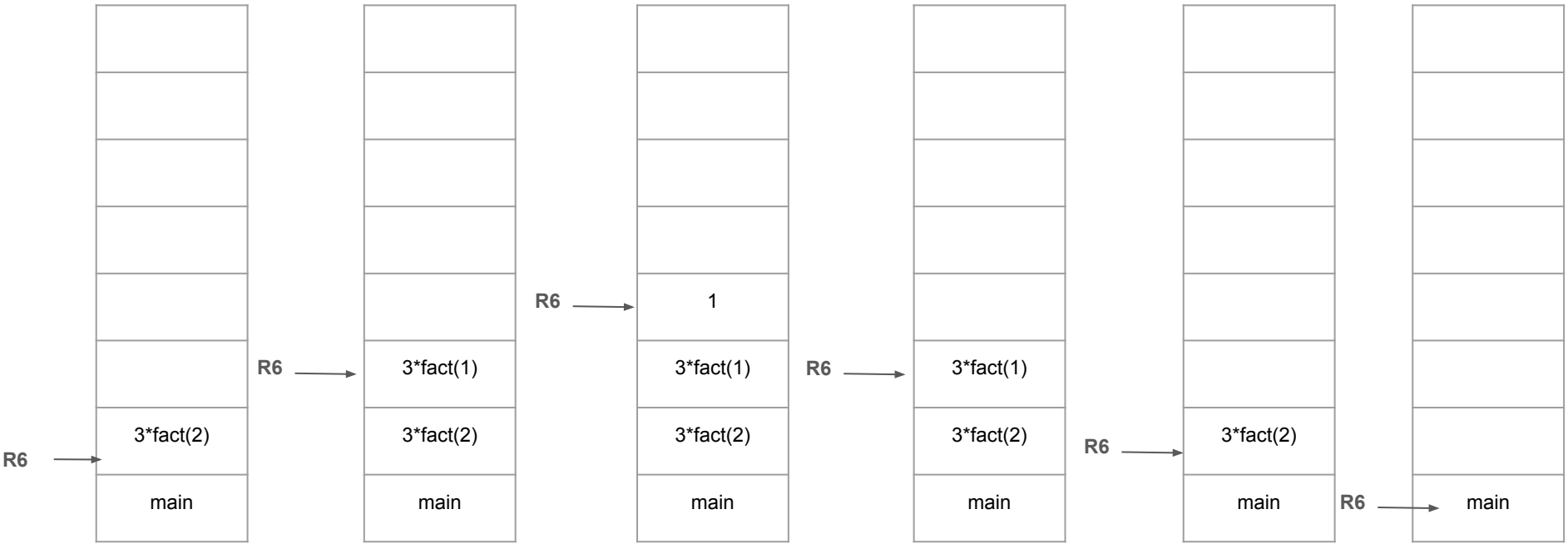

Calling the function:
Push second argument
AND R0, R0, #0
ADD R0, R0, #10
ADD R6, R6, #-1
STR R0, R6, #0
Push first argument
LDR R0, R5, #0
ADD R6, R6, #-1
STR R0, R6, #0
Call subroutine
Starting the Callee function:
Leave space for return value
ADD R6, R6, #-1
Push return address
ADD R6, R6, #-1
STR R7, R6, #0
push dynamic link (callers frame ptr)
ADD R6, R6, #-1
STR R5, R6, #0
Set new frame pointer
ADD R5, R6, #-1
Allocate space for locals
ADD R6, R6, #-1
Ending the Callee Function
Copy return into return value
LDR R0, R5, #0
STR R0, R5, #3
pop local variables
ADD R6, R5, #1
pop dynamic link
LDR R5, R6, #0
ADD R6, R6, #1
pop return address
LDR, R7, R6, #0
ADD R6, R6, #1
return control to caller
RET

Recursion

Whenever a recursive function is invoked, a new activation record is pushed onto the stack.

A solution based on a recursive function is not as efficient as a loop-based solution.

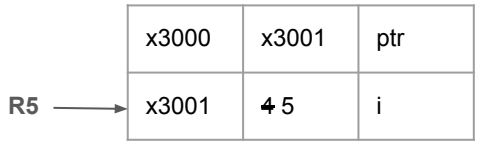
```
int factorial(int n) {
  int i;
  int result = 1;
  for (i = 1; i <= n; i++)
    result *= i;
  return result;
}
```



Pointers

int *p; (p is a pointer to an integer)
*p (returns the value pointed to by p
&z (returns the address of variable z)

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1
```




```
i = 4;
AND R0, R0, #0
ADD R0, R0, #4
STR R0, R5, #0

ptr = &i;
ADD R0, R5, #0
STR R0, R5, #-1

*ptr = *ptr + 1
LDR R0, R5, #-1
LDR R1, R0, #0
ADD R1, R1, #1
STR R1, R0, #0
```

Pointer Variables: Always store addresses of other variables.
Dereferencing Pointers (*ptr): Accesses or modifies the value at the address the pointer holds.

```
void swap2(int* i, int* j) {
  int temp;
  temp = *i;
  *i = *j:
  *j = temp;
}

int main() {
  int a = 12, b=13;
  swap2(&a, &b);
}
```

swap2(x3000, x3001)

x3002	12	temp
x3001	12	b
x3000	13	a

swap2(x3000, x3001)

x3001	13	b
x3000	12	a

In this example, in the main() function, two integers, and and b are declared and stored into memory somewhere.

The addresses of a and b are passed into the swap2() function. Hence, swap2(x3000, x3001)

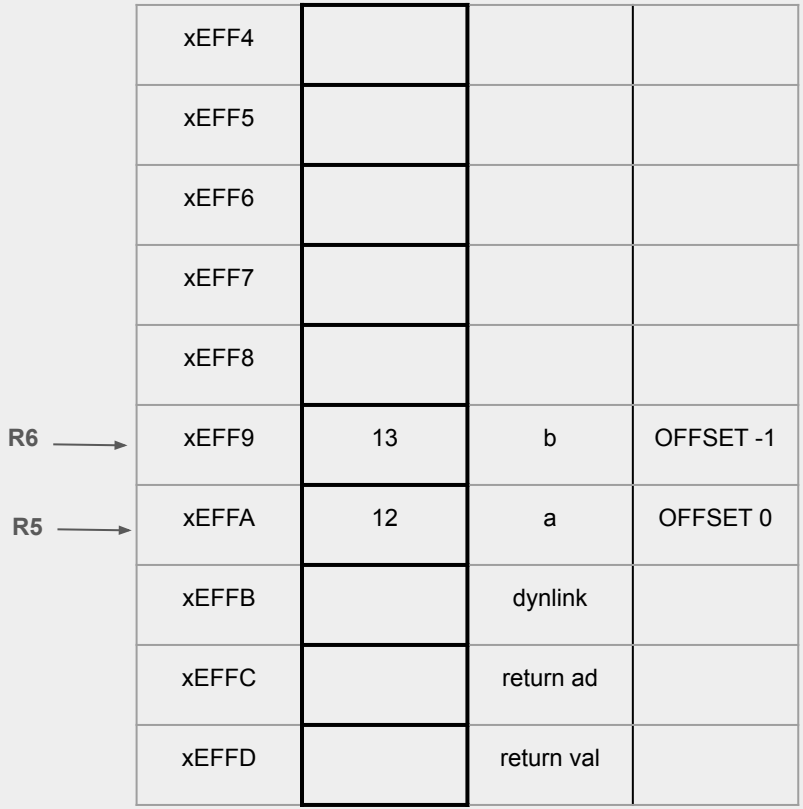
In the swap2() function, the parameters are pointers, since x3000 and x3001 are passed, the content of x3000 and x3001 are what's passed in.

a temp variable is initiated, and it stored what x3000 holds. temp = 12

then the content of x3000 is assigned to x3001, so i =j

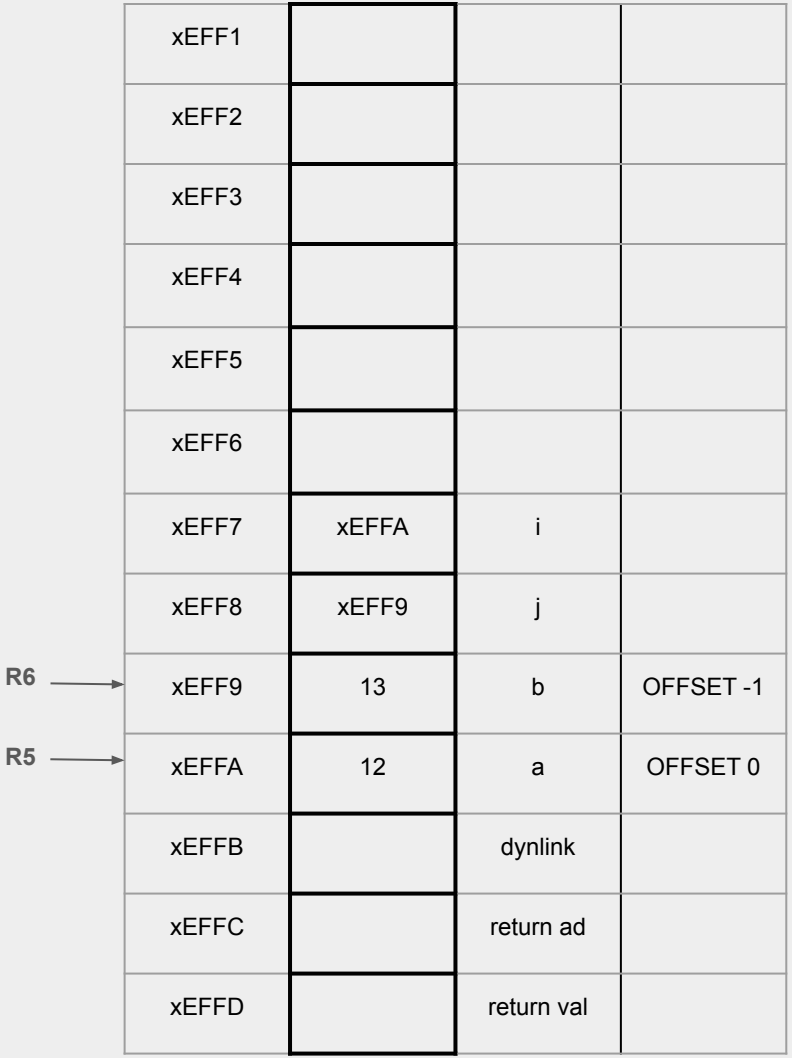
Finally, j is assigned the content of temp, so j = i

```
int main() {
  int a = 12, b=13;
  swap2(&a, &b);
}
```



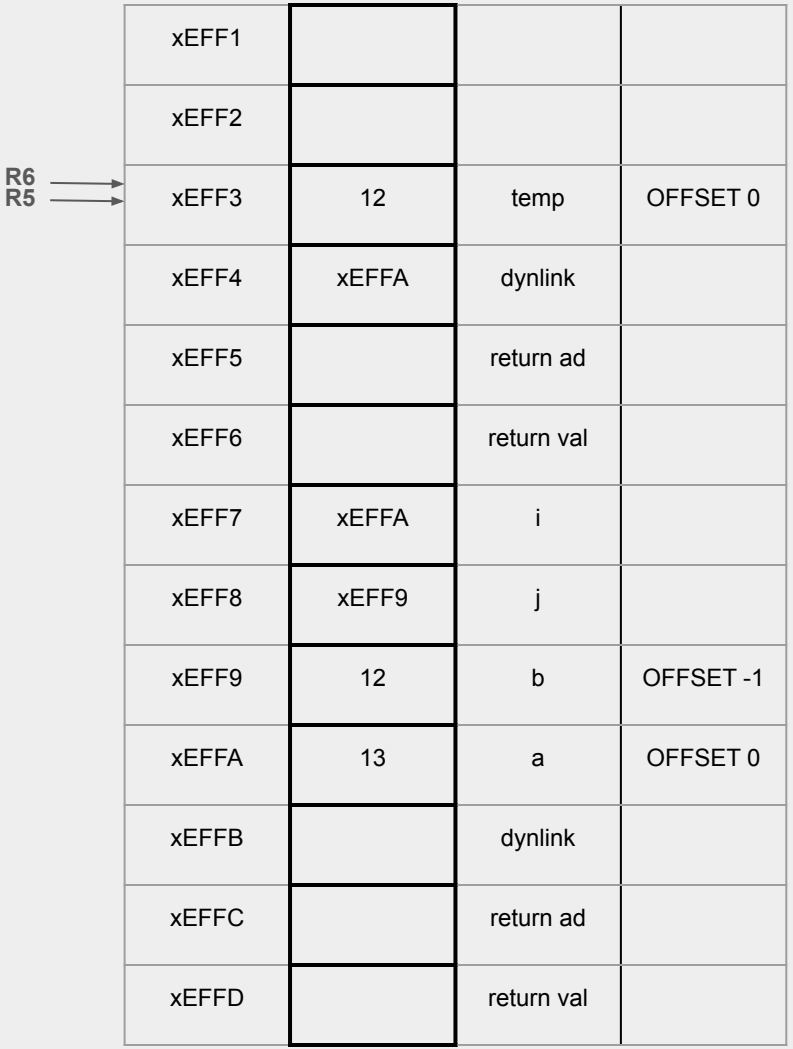
```
int main() {
  int a = 12, b=13;
  swap2(&a, &b);
}

void swap2(int* i, int* j) {
```

```
ADD R0, R5, #-1   ------>  R0 = xEFF9
ADD R6, R6, #-1
STR R0, R6, #0

ADD R0, R5, #0    ------>  R0 = xEFFA
ADD R6, R6, #-1
STR R0, R6, #0
```



```
int temp;
temp = *i;
*i = *j:
*j = temp;
```

```
LDR R0, R5, #4  ──→  R0 = xEFFA
LDR R1, R0, #0  ──→  R1 = 12
STR R1, R5, #0  ──→  Store 12 into R5

LDR R1, R5, #5  ──→  R1 = xEFF9
LDR R2, R1, #0  ──→  R2 = 13
STR R2, R0, #0  ──→  Store 13 into R0

LDR R2, R5, #0  ──→  R2 = 12
STR R2, R1, #0  ──→  Store 12 into R1
```



Arrays

Declaration: type variable[number_of_elements]
Array Reference: variable[index]

The last element of the array is stored in the stack frame first.

```
int grid[10], x
x = grid[3] + 1
grid[6] = 5

ADD R0, R5, #-9      R0 = &grid[0]
LDR R1, R0, #3       R1 = grid[3]
ADD R1, R1, #1       R1 = grid[3] + 1
STR R1, R5, #-10     Store grid[3] + 1 into x

AND R0, R0, #0       R0 = 0
ADD R0, R0, #5       R0 = 5
ADD R1, R5, #-9      R1 = x3009
STR R0, R1, #6       Store 5 into x3003
```

grid[x+1] = grid[x] + 2

```
LDR R0, R5, #-10        R0 = x
ADD R1, R5, #-9         R1 = x3009
ADD R1, R0, R1          R1 = x3009 + x
LDR R2, R1, #0          R2 = x
ADD R2, R2, #2          R2 = x+2

LDR R0, R5, #-10        R0 = x
ADD R0, R0, #1          R0 = x+1
ADD R1, R5, #-9         R1 = x3009
ADD R1, R0, R1          R1 = x3009 + x+1
STR R2, R1, #0          Store x+2 into x3009 + x+1
```

	x300A	x	OFFSET -10
	x3009	grid[0]	OFFSET -9
	x3008	grid[1]	OFFSET -8
	x3007	grid[2]	OFFSET -7
	x3006	grid[3]	OFFSET -6
	x3005	grid[4]	OFFSET -5
	x3004	grid[5]	OFFSET -4
	x3003	grid[6]	OFFSET -3
	x3002	grid[7]	OFFSET -2
	x3001	grid[8]	OFFSET -1
R5 →	x3000	grid[9]	OFFSET 0

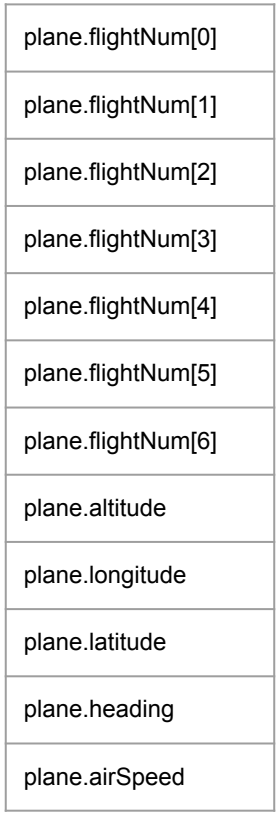
```
void foo(int* a, int b);
int main() {
  int x = 2
  int y = 3
  foo(&x, y);
}

void foo(int * a, int b) {
  int m[4], n;
  n = b;
  m[0] = a;
  m[1] = *a
  m[2] = &b;
  a = 4;
  b = 5;
}
```

```
  foo(&x, y);

void food(int * a, int b) {
  int m[4], n;
  n = b;
  m[0] = a;
  m[1] = *a
  m[2] = &b;
  a = 4;
  b = 5;
}
```

	xAFF1	3	n
	xAFF2	xAFFC	m[0]
	xAFF3	2	m[1]
	xAFF4	xAFFA	m[2]
	xAFF5		m[3]
	xAFF6	xAFFC	
	xAFF7	ret add	
	xAFF8	ret val	
	xAFF9	4	a
	xAFFA	5	b
	xAFFB	3	y
R5 →	xAFFC	2	x
	xAFFD		dyn link
	xAFFE		ret add
	xAFFF		ret val

OOP

```
struct flightType plane;

struct flightType {
  char flightNum[7]
  int altitude;
  int longitude;
  int latitude;
  int heading;
  int airSpeed;
};
```



You can also use the flightType name to declare other structs.

```
struct flyer {
  char name[20];
  struct flightType flight;
}
```

C provides a way to define a data type by giving a new name to a predefined type.

```
typedef <type> <name>;

typedef struct flightType Flight;
```

```
int x;
Flight plane;
int y;

plane.altitude = 0;
```

```
struct flightType plane;

struct flightType {
  char flightNum[7]
  int altitude;
  int longitude;
  int latitude;
  int heading;
  int airSpeed;
};
```

Address of an element with index i = starting address + i

OFFSET -13		y
OFFSET -12	INDEX 0	plane.flightNum[0]
OFFSET -11	INDEX 1	plane.flightNum[1]
OFFSET -10	INDEX 2	plane.flightNum[2]
OFFSET -9	INDEX 3	plane.flightNum[3]
OFFSET -8	INDEX 4	plane.flightNum[4]
OFFSET -7	INDEX 5	plane.flightNum[5]
OFFSET -6	INDEX 6	plane.flightNum[6]
OFFSET -5	INDEX 7	plane.altitude
OFFSET -4	INDEX 8	plane.longitude
OFFSET -3	INDEX 9	plane.latitude
OFFSET -2	INDEX 10	plane.heading
OFFSET -1	INDEX 11	plane.airSpeed
OFFSET 0 (R5)		x