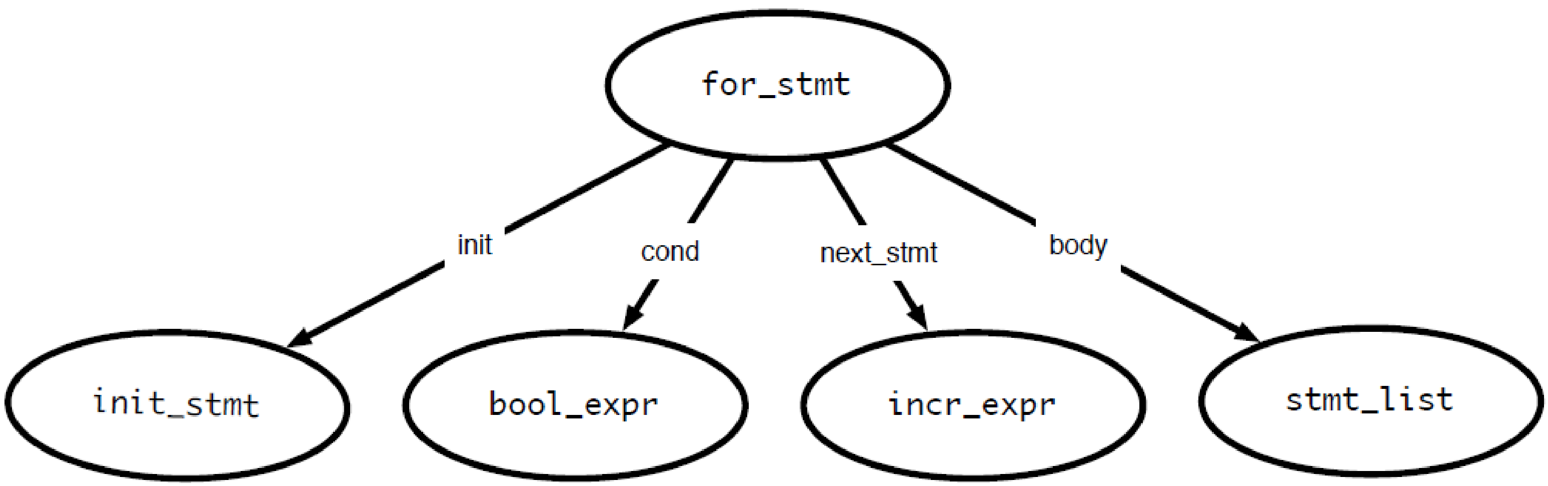
CS323: Compilers

Spring 2023

Week 7: Semantic Processing (for loop, switch, functions)

For loops

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```



Slide courtesy: Milind Kulkarni

Generating code: for loops

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```

↓

```
    <init_stmt>
LOOP:
    <bool_expr>
    j<!op> OUT
    <stmt_list>
INCR:
    <incr_stmt>
    jmp LOOP
OUT:
```

- Execute init_stmt first
- Jump out of loop if bool_expr is false
- Execute incr_stmt after block, jump back to top of loop
- Question: Why do we have the INCR label?

Slide courtesy: Milind Kulkarni

Switch statements

```
switch (<expr>)
  case <const_list>: <stmt_list>
  case <const_list>: <stmt_list>
  ...
  default: <stmt_list>
end
```

- Generated code should evaluate <expr> and make sure that some case matches the result
- Question: how to decide where to jump?

Slide courtesy: Milind Kulkarni

Deciding where to jump

- Problem: do not know *which label* to jump to until switch expression is evaluated
- Use a jump table: an array indexed by case values, contains address to jump to
  - If table is not full (i.e., some possible values are skipped), can point to a default clause
    - If default clause does not exist, this can point to error code
  - Problems
    - If table is sparse, wastes a lot of space
    - If many choices, table will be very large

Slide courtesy: Milind Kulkarni

Jump table example

Consider the code:
((xxxx) is address of code)

Case x is
(0010) When 0: stmts
(0017) When 1: stmts
(0192) When 2: stmts
(0198) When 3 stmts;
(1000) When 5 stmts;
(1050) Else stmts;

Table only has one
Unnecessary row
(for choice 4)

Jump table has 6 entries:

0	JUMP 0010
1	JUMP 0017
2	JUMP 0192
3	JUMP 0198
4	JUMP 1050
5	JUMP 1000

Slide courtesy: Milind Kulkarni

Jump table example

Consider the code:
((xxxx) Is address of code)

Case x is
(0010) When 0: stmts0
(0017) When 1: stmts1
(0192) When 2: stmts2
(0198) When 3 stmts3
(1000) When 987 stmts4
(1050) When others stmts5

Jump table has 6 entries:

0	JUMP 0010
1	JUMP 0017
2	JUMP 0192
3	JUMP 0198
4	JUMP 1050
. . .	JUMP 1050
986	JUMP 1050
987	JUMP 1000

Table only has 983 unnecessary rows. Doesn't appear to be the right thing to do! ***NOTE: table size is proportional to range of choice clauses, not number of clauses!***

Slide courtesy: Milind Kulkarni

Linear search example

Consider the code:
(xxxx) Is offset of local
Code start from the
Jump instruction

Case x is
(0010) When 0: stmts
(0017) When 1: stmts
(0192) When 2: stmts
(1050) When others stmts;

If there are a small number of choices, then do an in-line linear search. A straightforward way to do this is generate code analogous to an IF THEN ELSE.

If (x == 0) then stmts1;
Elseif (x = 1) then stmts2;
Elseif (x = 2) then stmts3;
Else stmts4;

O(n) time, n is the size of the table, for each jump.

Dealing with jump tables

```
switch (<expr>)
  case <const_list>: <stmt_list>
  case <const_list>: <stmt_list>
  ...
  default: <stmt_list>
end
```

```
    <expr>
    <code for jump table>
  LABEL0:
    <stmt_list>
  LABEL1:
    <stmt_list>
  ...
  DEFAULT:
    <stmt_list>
  OUT:
```

- Generate labels, code, then build jump table
  - Put jump table after generated code
- Why do we need the OUT label?
  - In case of `break` statements

Slide courtesy: Milind Kulkarni

Functions

- **Typical Syntax and Usage**

```
FUNCTION VOID bar(INT x, FLOAT y) BEGIN

END
```

Keywords

Return type

comma separated parameter declarations.

Declarations (string or variable decl) followed by statement declarations

```
FUNCTION void foo() BEGIN
INT a;
FLOAT b;
…
bar(a, b);

END
```

Calls function `bar`

Terms

```
void foo() {
  int a, b;
  ...
  bar(a, b);
}


void bar(int x, int y) {
    ...
}
```

- foo is the *caller*
- bar is the *callee*
- a, b are the *actual parameters* to bar
- x, y are the *formal parameters* of bar
- Shorthand:
  - argument = actual parameter
  - parameter = formal parameter

Different Kinds of Parameters

- Value
- Reference
- Result
- Value-Reference
- Read-only
- Call-by-Name

Value parameters

- "Call-by-value"
- Used in C, Java, default in C++
- Passes the value of an argument to the function
- Makes a copy of argument when function is called
- Advantages? Disadvantages?

Advantage: 'side-effect' free – caller can be sure that the argument is not modified by the callee

Disadvantage: Not efficient for larger sized arguments.

Value parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int y, int z) {
    y = 2;
    z = 3;
    print(x);
}
```

- What do the print statements print?

Reference parameters

- “Call-by-reference”
- Optional in Pascal (use “var” keyword) and C++ (use “&”)
- Pass the *address* of the argument to the function
- If an argument is an expression, evaluate it, place it in memory and then pass the address of the memory location
- Advantages? Disadvantages?

Advantage: Efficiency – for larger sized arguments
Disadvantage: results in clumsy code at times (e.g. check for null pointers)

Reference parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int &y, int &z) {
    y = 2;
    z = 3;
    print(x);
    print(y);
}
```

- What do the print statements print?

Result Parameters

- To capture the return value of a function
- Copied at the end of function into arguments of the caller
- E.g. `output` ports in Verilog `module` definitions

Result Parameters

```
int x = 1
void main () {
  foo(x, x);
  print(x);
}

void foo(int y, result int z) {
  y = 2;
  z = 3;
  print(x);
}
```

- What do the print statements print?

Value-Result Parameters

- “Copy-in copy-out”
- Evaluate argument expression, copy to parameters
- After subroutine is done, copy values of parameters back into arguments
- Results are often similar to pass-by-reference, but there are some subtle situations where they are different

Value-Result Parameters

```
int x = 1
void main () {
  foo(x, x);
  print(x);
}

void foo(int y, value result int z)
{
  y = 2;
  z = 3;
  print(x);
}
```

- What do the print statements print?

Read-only Parameters

- Used when callee will not change value of parameters
- Read-only restriction must be enforced by compiler
- E.g. `const` parameter in C/C++
- Enforcing becomes tricky when in the presence of aliasing and control flow. E.g.

```
void foo(readonly int x, int y) {
int * p;
if (...) p = &x else p = &y
*p = 4
}
```

Call-by-name Parameters

- The arguments are passed to the function before evaluation
  - Usually, we evaluate the arguments before passing them
- Not used in many languages, but Haskell uses a variant

```
int x = 1
void main () {
    foo(x+2);
    print(x);
}

void foo(int y) {
    z = y + 3; //expands to z = x + 2 + 3
    print(z);
}
```

Call-by-name Parameters

- Why is this useful?
  - E.g. to analyze certain properties of a program/function – termination

```
void main () {
    foo(bar());
}

void foo(int y) {
    z = 3;
    if(z > 3)
        z = y + z;
}
```

  - Even if `bar` has an infinite loop, the program terminates.

Program Layout in Memory

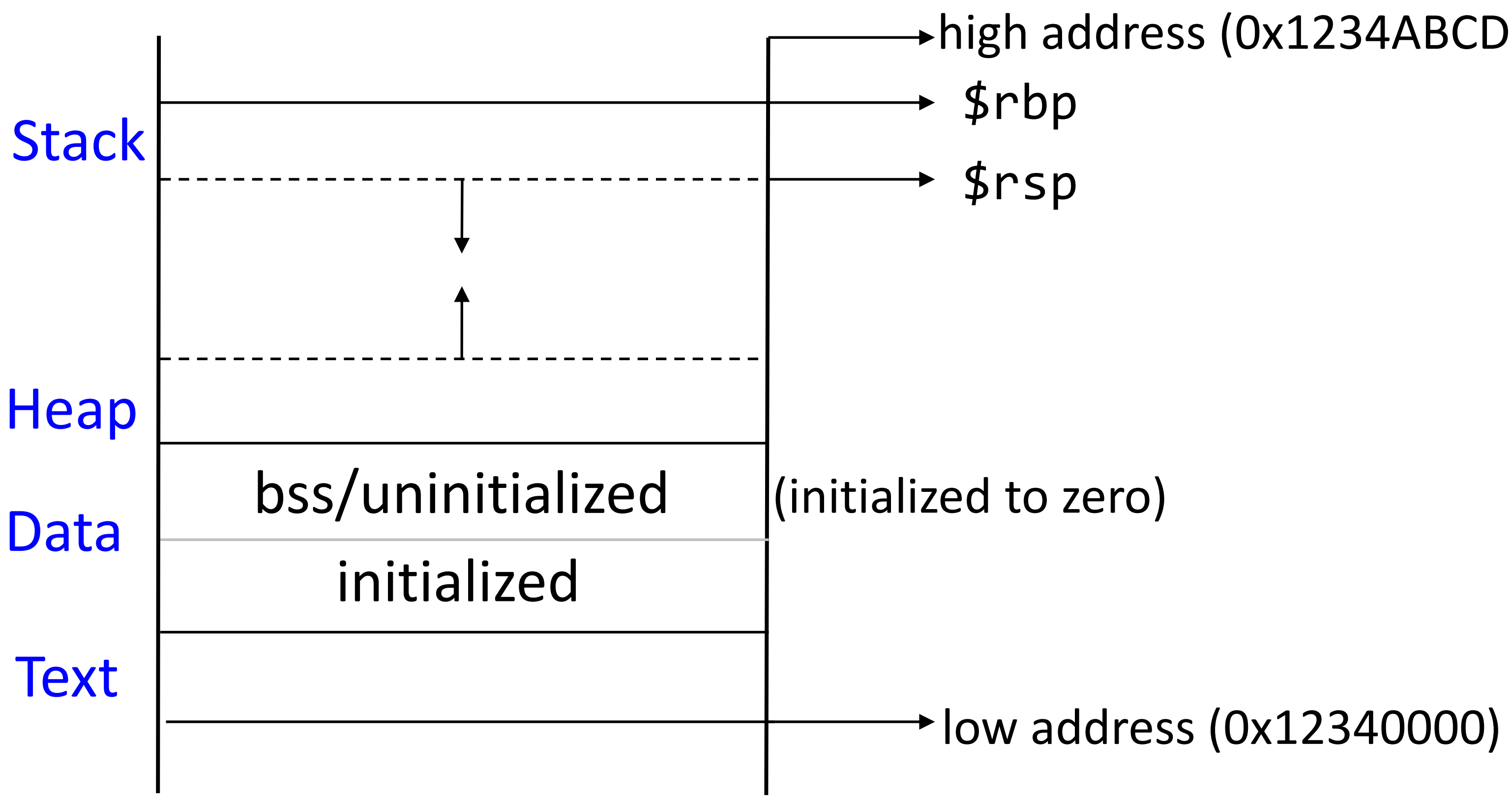
- Compiler assumes a *runtime environment* for execution of the program.
- A C/C++ program in Linux OS has 4 segments of memory
  - Every memory location is a *box* holding *data/instruction*

Program Layout in Memory

- A program's memory space is divided into four segments:

  1. Text
     - source code of the program
  2. Data
     - Broken into *uninitialized* and *initialized* segments; contains space for global and static variables. E.g. `int x = 7; int y;`
  3. Heap
     - Memory allocated using `malloc/calloc/realloc/new`
  4. Stack
     - Function arguments, return values, local variables, special registers.

Program Layout in Memory



Activation

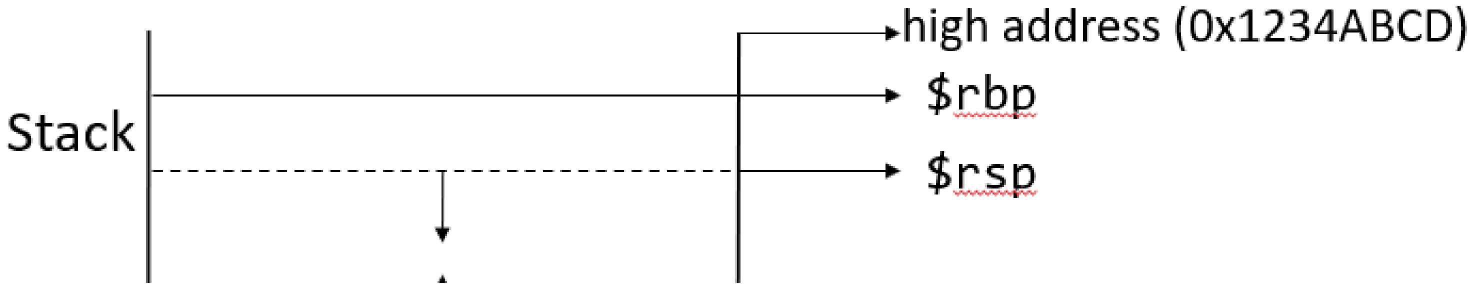
- A function call or invocation is termed an *activation*
- Calls to functions in a program form *activation tree*
  - Postorder traversal of the tree shows return sequence i.e. the order in which control returns from functions
  - Preorder traversal of the tree shows calling sequence
- In a sequential program, at any point in time, *control of execution is in any one activation*
  - All the ancestors of that activation are active i.e. have not returned

Activation

- Activations are managed through the help of *control stack*
- A function call (activation) results in allocating a chunk of memory called *activation record* or *frame* on the stack (also called *stack frame*)

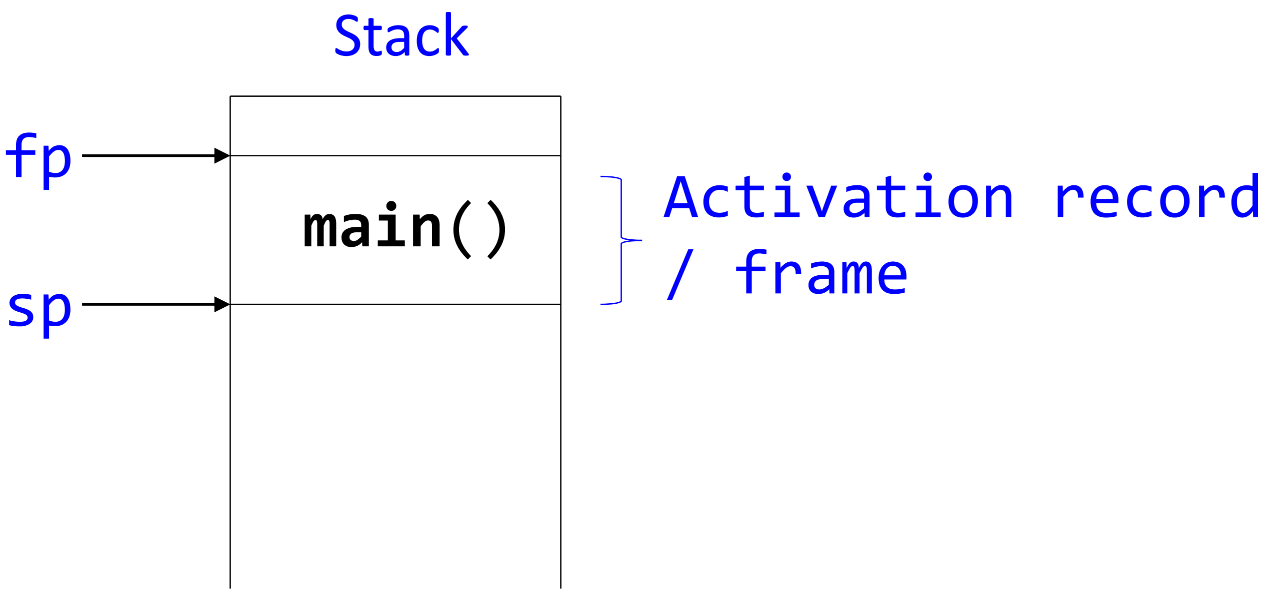
Activation Record

- A *sub-segment* of memory on the stack
  - Special registers `$rbp` and `$rsp` track the bottom and top of the stack frame. These are the names in x86 architecture.



- `$rbp` – base pointer or frame pointer (`fp`)
- `$rsp` – stack pointer (`sp`)

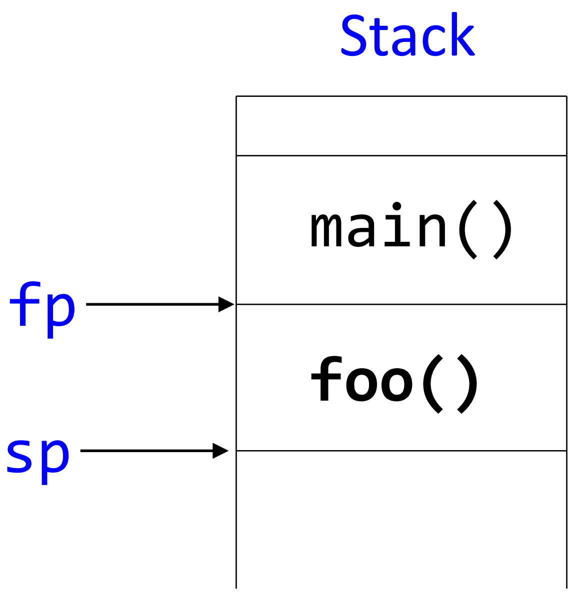
Activation Record - Example




```
main() {
    …
    foo();
    …
}

foo() {
    bar();
    …
    baz();
}
```

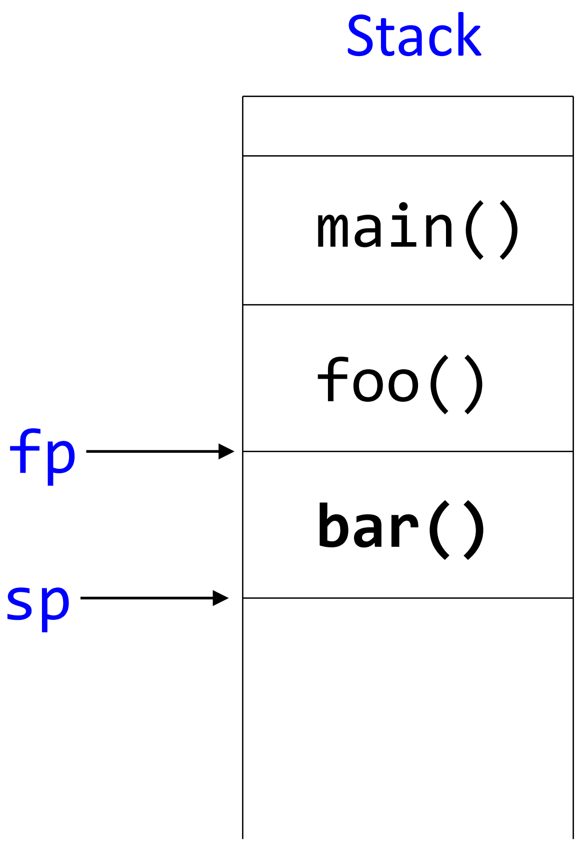
Activation Record - Example




```
main() {
    …
    foo();
    …
}

foo() {
    bar();
    …
    baz();
}
```

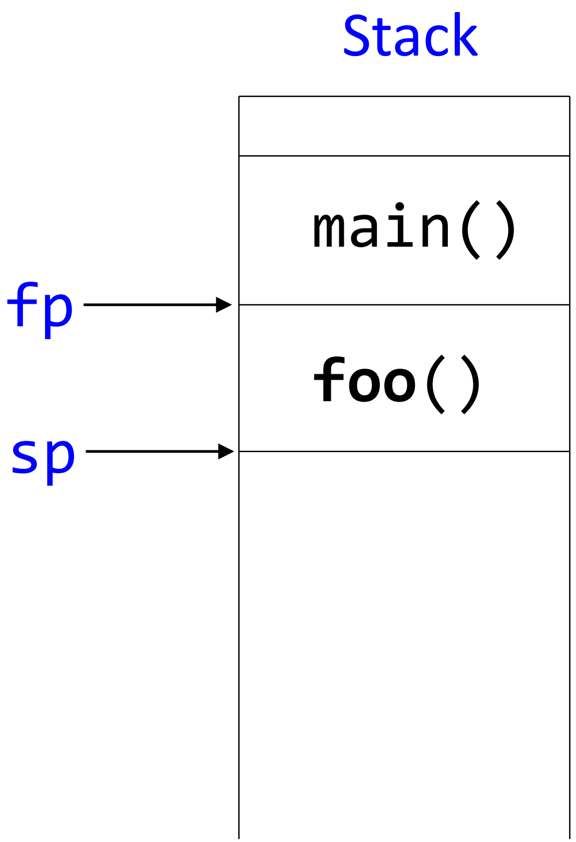
Activation Record - Example




```
main() {
    …
    foo();
    …
}

foo() {
    bar();
    …
    baz();
}
```

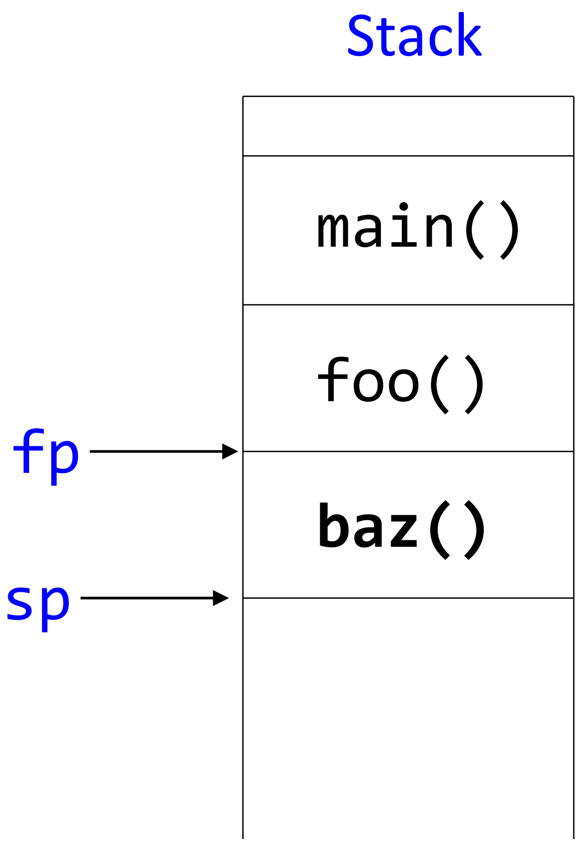
Activation Record - Example




```
main() {
   …
   foo();
   …
}

foo() {
   bar();
   …
   baz();
}
```

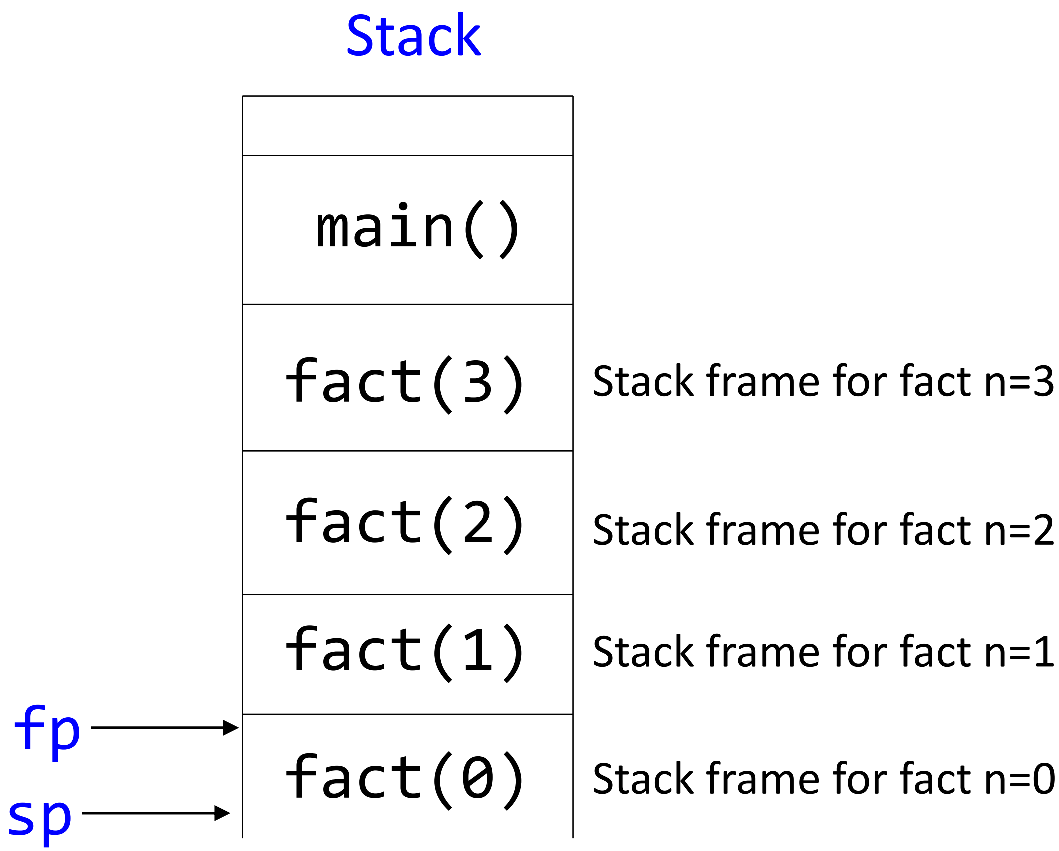
Activation Record - Example




```
main() {
    …
    foo();
    …
}

foo() {
    bar();
    …
    baz();
}
```

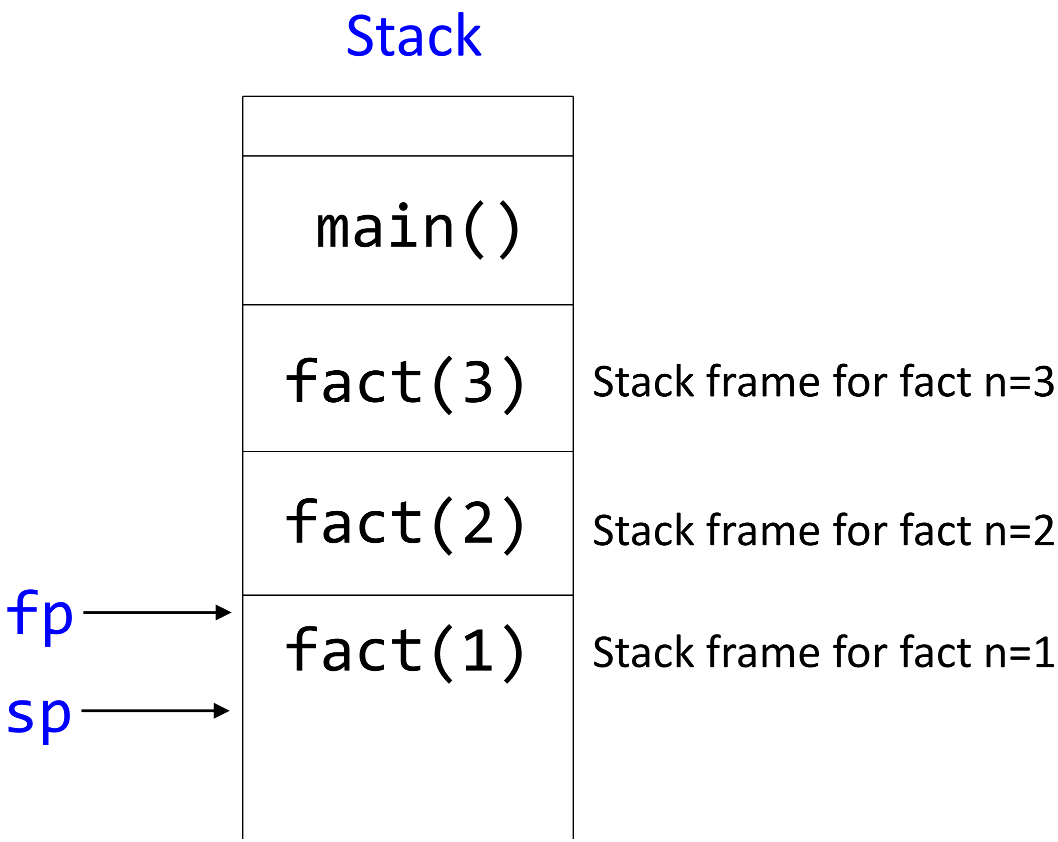
Activation Record – Example (Recursive Functions)




```
main() {
   …
   fact(3);
   …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Activation Record – Example (Recursive Functions)

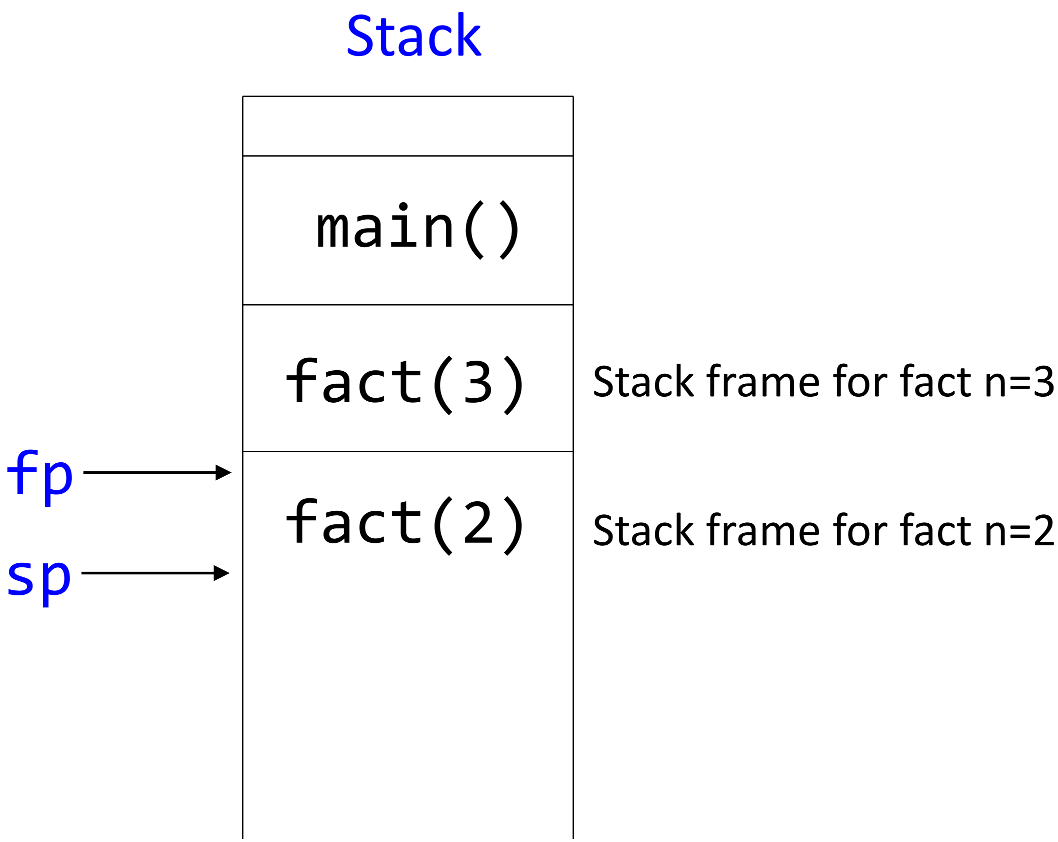



```
main() {
    …
    fact(3);
    …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=0 popped off. 1 Returned.

Activation Record – Example (Recursive Functions)

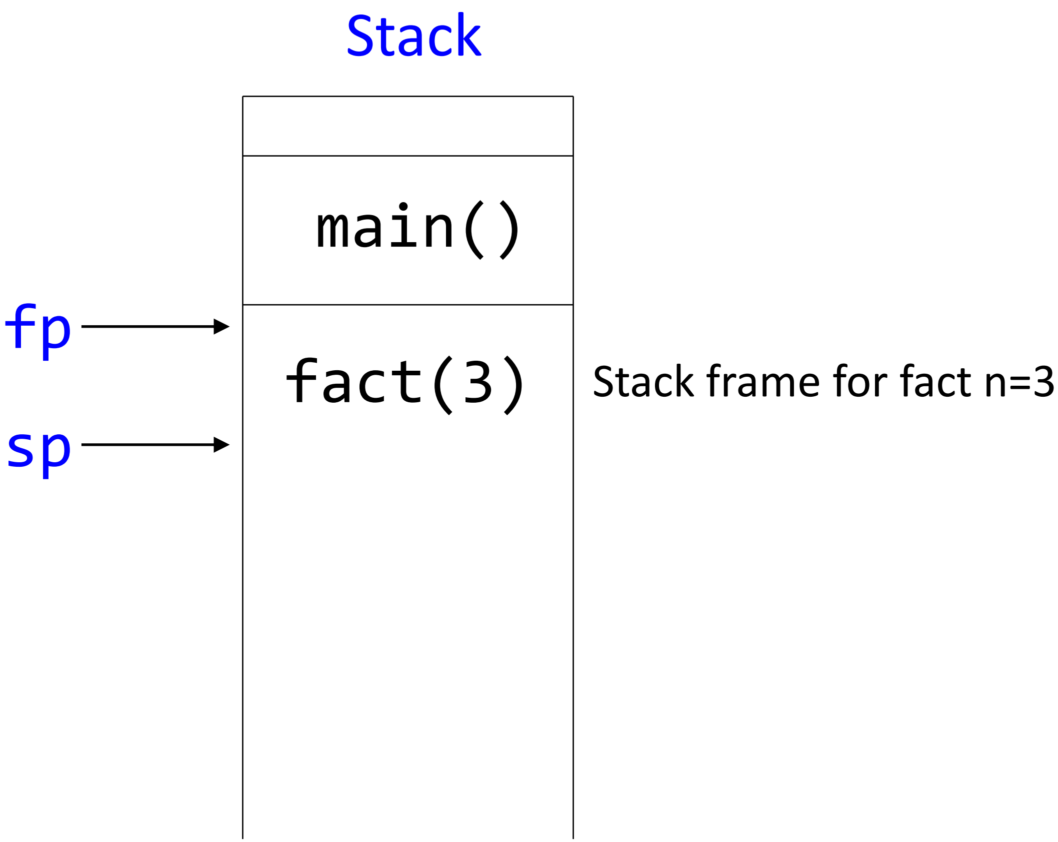



```
main() {
   …
   fact(3);
   …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=1 popped off. 1 Returned.

Activation Record – Example (Recursive Functions)

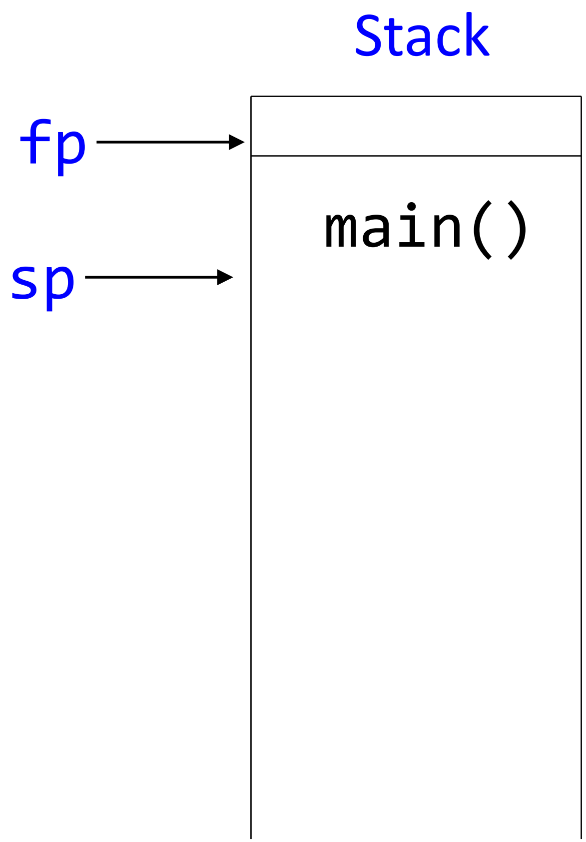



```
main() {
    …
    fact(3);
    …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=2 popped off. 2 Returned.

Activation Record – Example (Recursive Functions)




```
main() {
   …
   fact(3);
   …
}

fact(int n) {
if (n=0) return 1
return n*fact(n-1)
}
```

Stack frame for n=3 popped off. 6 Returned.

Activation Record

- What happens when a function is called?
  1. `fp` and `sp` get adjusted
  2. Memory for the activation record is allocated on stack
     - The size of the memory allocated depends on local variables used by the called function (consult function's symbol table for this)
  3. Each invocation of a function has its own instantiation of local variables
- When the function call returns:
  - Memory for the activation record is destroyed

Activation Record

- What is stored in the activation record?

  Depends on the language being implemented:
  - Temporaries
  - Local vars
  - Saved registers
  - Return address, previous `fp`
  - Return value
  - Actual Params
- Who stores this information?
  - Caller
  - Callee

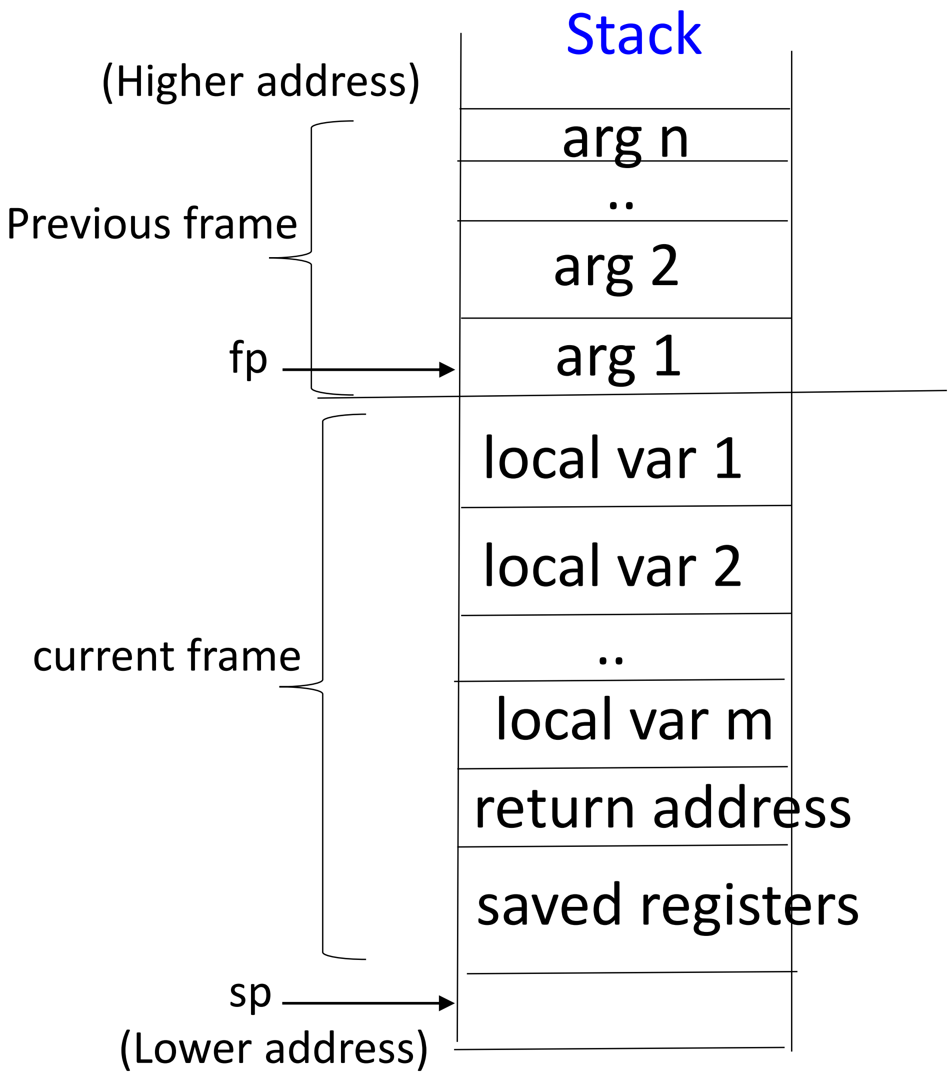
  together execute *calling sequence* and *return sequence*

Application Binary Interface (ABI)

- How is data organized on the activation record?
  - ABI is the specification on how data is provided to functions
    - Caller saves or callee saves
  - ABI is meant to deliver interoperability between different compilers
    - Compile the function using one compiler to create an object code, Link object code with other code compiled using a different compiler

forms the *calling convention*

Typical Activation Record



Callee accesses arguments using +ve offset from FP:
argument1 = memory[FP]
argument2 = memory[FP+1] ..

Callee accesses local variables using –ve offset from FP:
local var 1=memory[FP-1]
local var 2=memory[FP-2]

Function call: Peeking at Activation Record

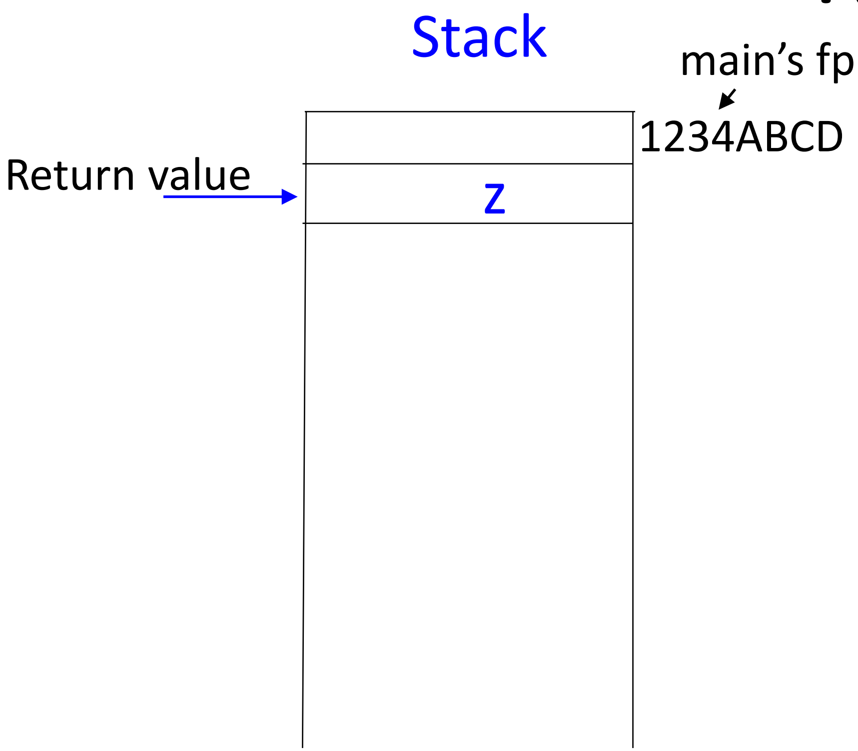
```
main() {
    …
    foo();
    …
}
```

- When `main` calls function `foo`
  1. The following are pushed on to the stack:
     1. `foo`'s arguments
     2. Space to hold `foo`'s return value
     3. Address of the next instruction executed (in `main`) when `foo` returns (return address)
     4. Current value of `$rbp` (frame pointer)

     `$rsp` is automatically updated (decremented) to point to current top of the stack.

  2. `$rbp` is assigned the value of `$rsp`

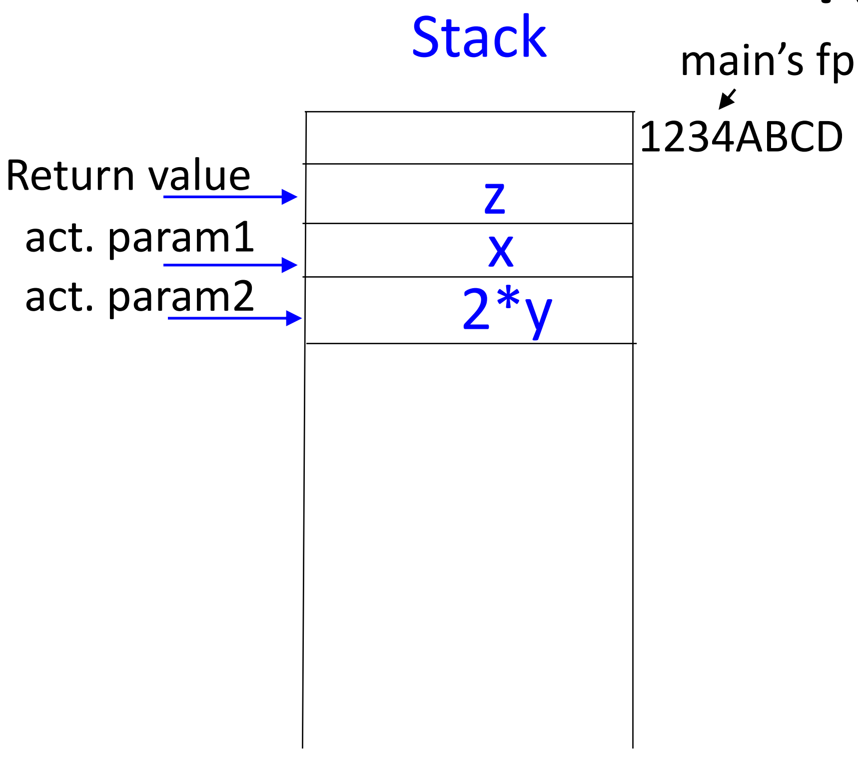
Function call: Peeking at Activation Record




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

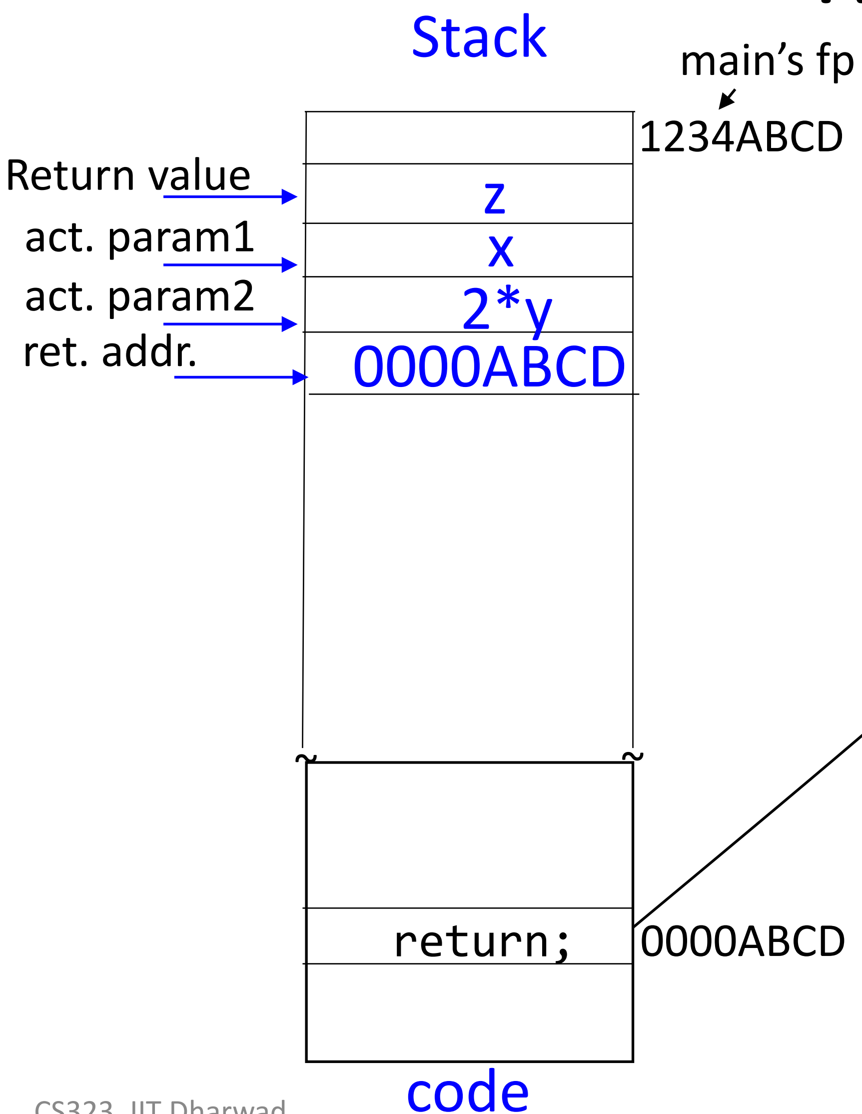
Function call: Peeking at Activation Record




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

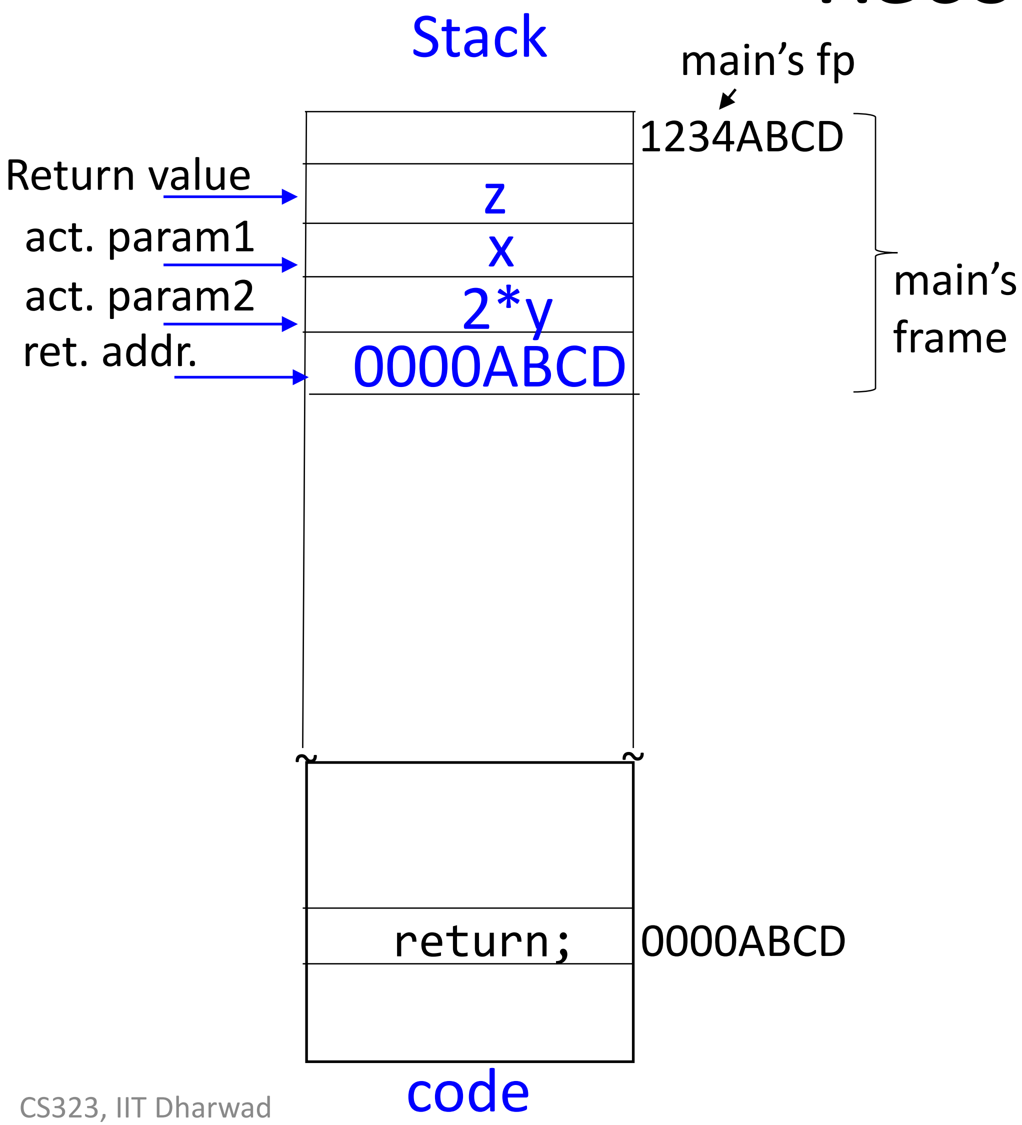
Function call: Peeking at Activation Record




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

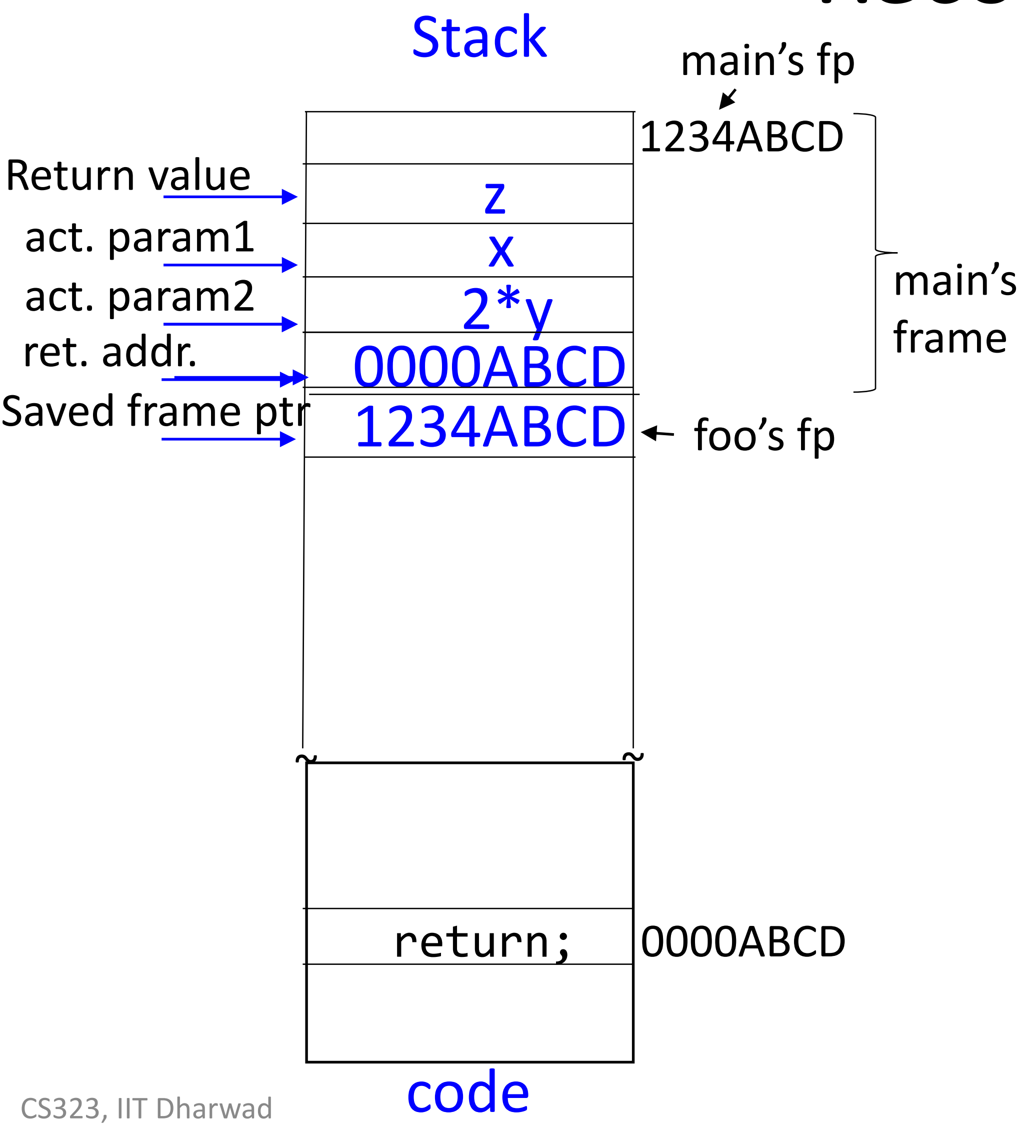
Function call: Peeking at Activation Record




```
main() {
    z=foo(x, 2*y);
    return;
}

int foo(int a, int b) {
    int l1, l2
    l1=a;
    l2=b;
    return l1+l2;
}
```

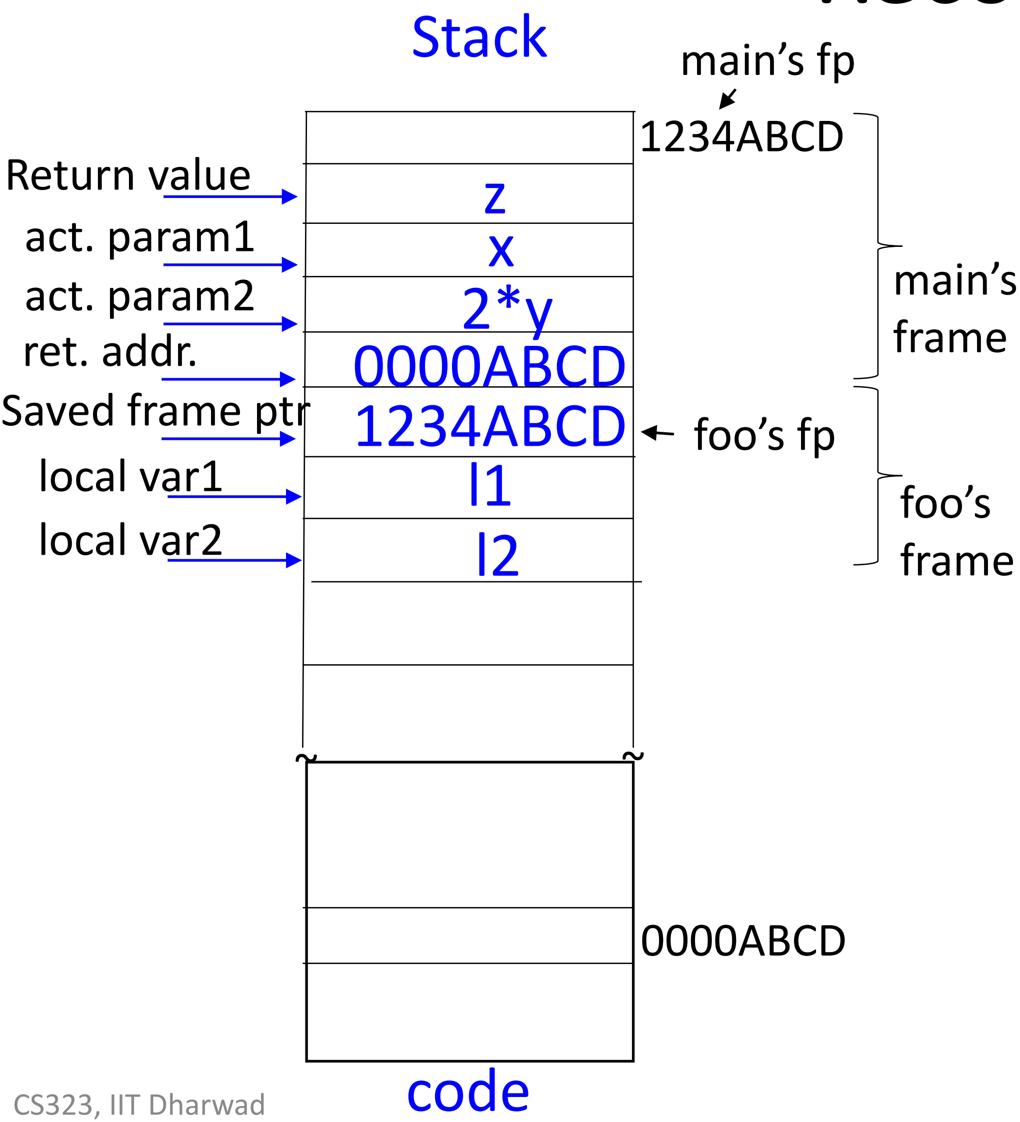
Function call: Peeking at Activation Record




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

Function call: Peeking at Activation Record




```
main() {
    z=foo(x, 2*y);
    return;
}

int foo(int a, int b) {
    int l1, l2
    l1=a;
    l2=b;
    return l1+l2;
}
```

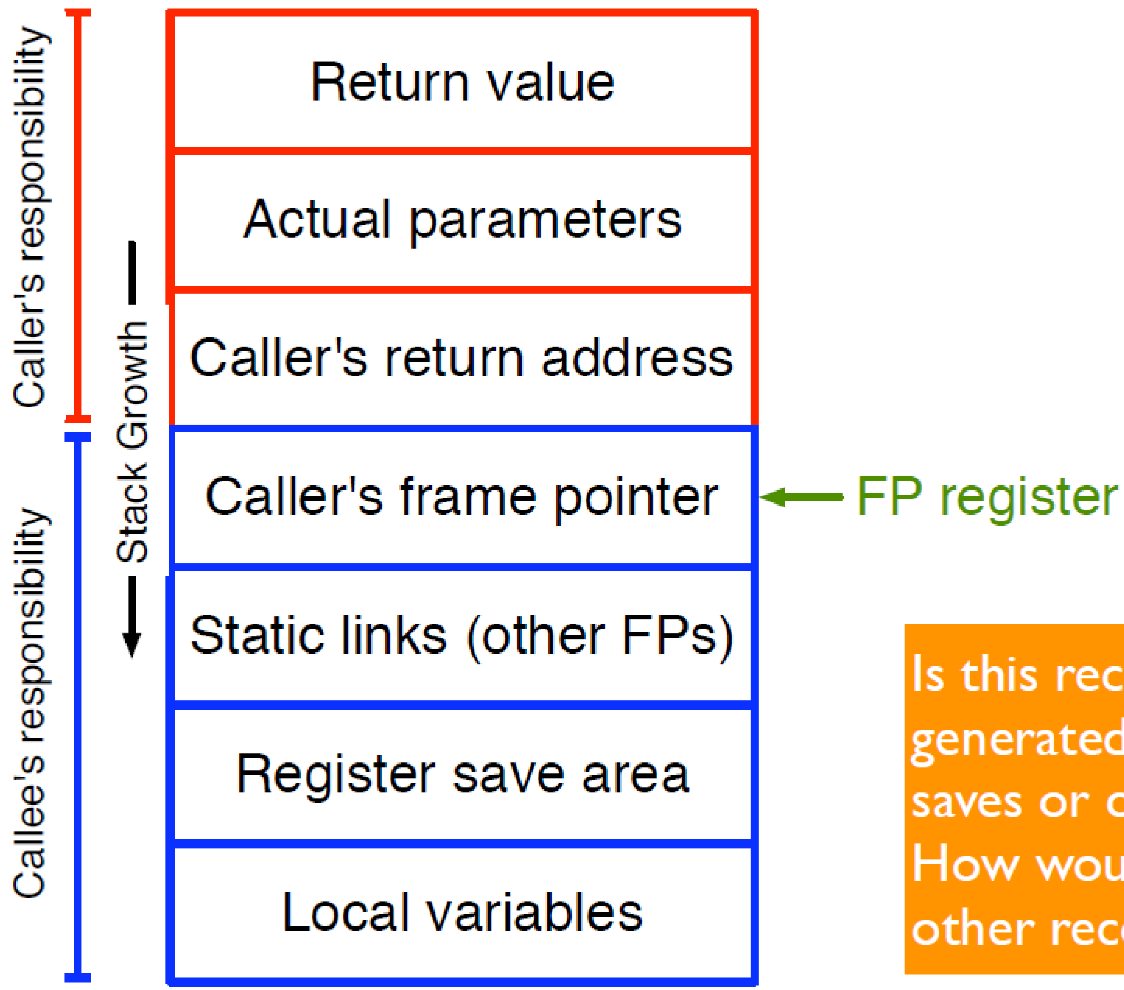
Function calls – Register Handling

- **Did not use registers** in the previous example (for parameter passing)
- Registers are faster than memory. So, compiler should keep parameters in registers whenever possible
- Modern calling convention places first few arguments in registers (arg1 in r1, arg2 in r2, arg3 in r3…) and the remaining in memory.
  - In x86 C-ABI, first 6 arguments are passed in registers
- What if callee wants to use registers r1, r2, r3 etc. for local computation? Callee must save the registers in its stack frame.

Function calls – Register Handling

- Two options: caller saves or callee saves
- Caller Saves
  - Caller pushes all the registers it is using on to the stack before calling the function
  - Restores the registers after the function returns
- Callee Saves
  - Callee pushes all the registers it is *going to use* on the stack immediately after being called
  - Restores the registers just before it returns

Activation records



Is this record generated for callee-saves or caller-saves? How would the other record look?