G22.2110-003 Programming Languages - Fall 2012

Lecture 4

Thomas Wies

New York University

Review

Last week

- Control Structures
- Selection
- Loops
- Adding Invariants

Outline

- Subprograms
- Calling Sequences
- Parameter Passing
- Recursion

Sources:
PLP, 3.6, 6.6, 8.1 - 8.3

Subprograms

- The basic abstraction mechanism
- *Functions* correspond to the mathematical notion of computation:

$$\text{input} \longrightarrow \text{output}$$

- *Procedures* affect the environment, and are called for their side-effects
- Pure functional model possible but rare (HASKELL)
- Hybrid model most common: functions can have (limited) side effects

Activation Records

Recall that subroutine calls rely heavily on use of the *stack*.

Each time a subroutine is called, space on the stack is allocated for the objects needed by the subroutine.

This space is called a *stack frame* or *activation record*.

The *stack pointer* contains the address of either the last used location or the next unused location on the stack.

The *frame pointer* points into the activation record of a subroutine so that any objects allocated on the stack can be referenced with a static offset from the frame pointer.

Activation Records

Why not use an offset from the stack pointer to reference subroutine objects?

Activation Records

Why not use an offset from the stack pointer to reference subroutine objects?

There may be objects that are allocated on the stack whose size is unknown at compile time.

These objects get allocated above the frame pointer so that objects whose size is known at compile time can still be accessed quickly.

Activation Records

Why not use an offset from the stack pointer to reference subroutine objects?

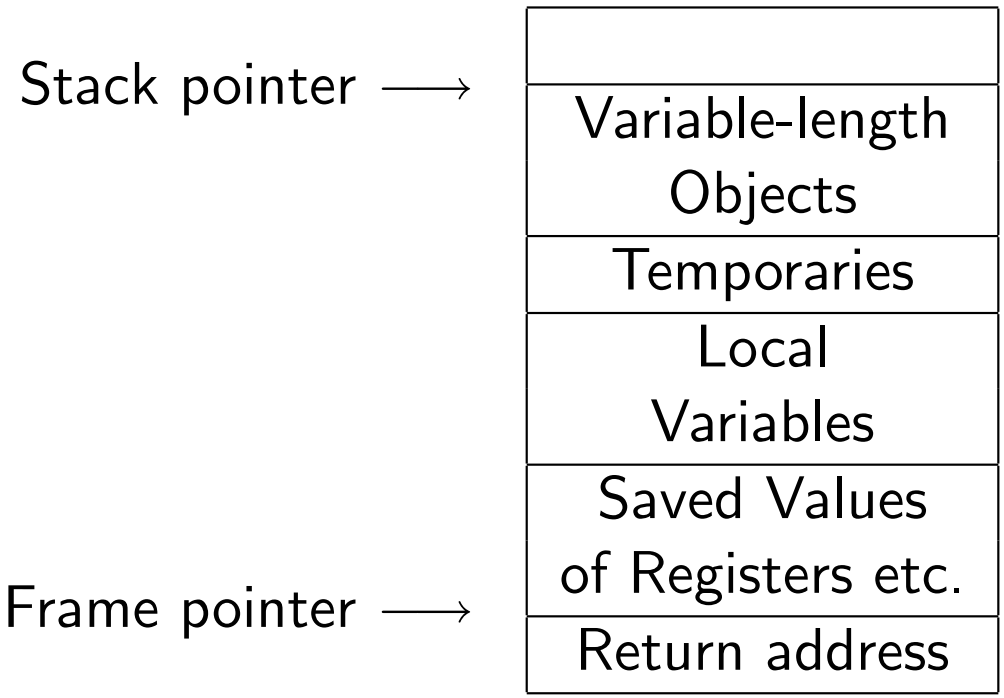
There may be objects that are allocated on the stack whose size is unknown at compile time.

These objects get allocated last so that objects whose size is known at compile time can still be accessed quickly via a known offset from the frame pointer.

Example

```
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
   ...
end
```

Typical Activation Record



Managing Activation Records

When a subroutine is called, a new activation record is created and populated with data.
The management of this task involves both the *caller* and the *callee*.

- The *calling sequence* refers to code executed by the caller just before and just after a subroutine call.
- The *prologue* refers to activation record management code executed at the beginning of a subroutine.
- The *epilogue* refers to activation record management code executed at the end of a subroutine.

Sometimes the term *calling sequence* is used to refer to the combined operations of the caller, prologue, and epilogue.

Calling Sequence

Calling a subroutine

- Pass parameters
- Save return address
- Update static chain
- Change program counter
- Move stack pointer
- Save register values, including frame pointer
- Move frame pointer
- Initialize objects

Calling Sequence

Finishing a subroutine

- Finalize (destroy) objects
- Pass return value(s) back to caller
- Restore register values, including frame pointer
- Restore stack pointer
- Restore program counter

Calling Sequence

Are there advantages to having the caller or callee perform various tasks?

Calling Sequence

Are there advantages to having the caller or callee perform various tasks?

If possible, have the callee perform tasks: task code needs to occur only once, rather than at every call site.

Calling Sequence

Are there advantages to having the caller or callee perform various tasks?

If possible, have the callee perform tasks: task code needs to occur only once, rather than at every call site.

Some tasks (e.g. parameter passing) must be performed by the caller.

Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?

Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?

Save all registers currently being used by caller.

Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?

Save all registers currently being used by caller.

What would the callee have to do to ensure proper saving of registers?

Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?

Save all registers currently being used by caller.

What would the callee have to do to ensure proper saving of registers?

Save all registers that will be used by callee.

Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?

Save all registers currently being used by caller.

What would the callee have to do to ensure proper saving of registers?

Save all registers that will be used by callee.

Which is better?

Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?

Save all registers currently being used by caller.

What would the callee have to do to ensure proper saving of registers?

Save all registers that will be used by callee.

Which is better?
Could be either one–no clear answer.

In practice, many processors (including MIPS and x86) compromise: half the registers are caller-save and half are callee-save.

Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?

Save all registers currently being used by caller.

What would the callee have to do to ensure proper saving of registers?

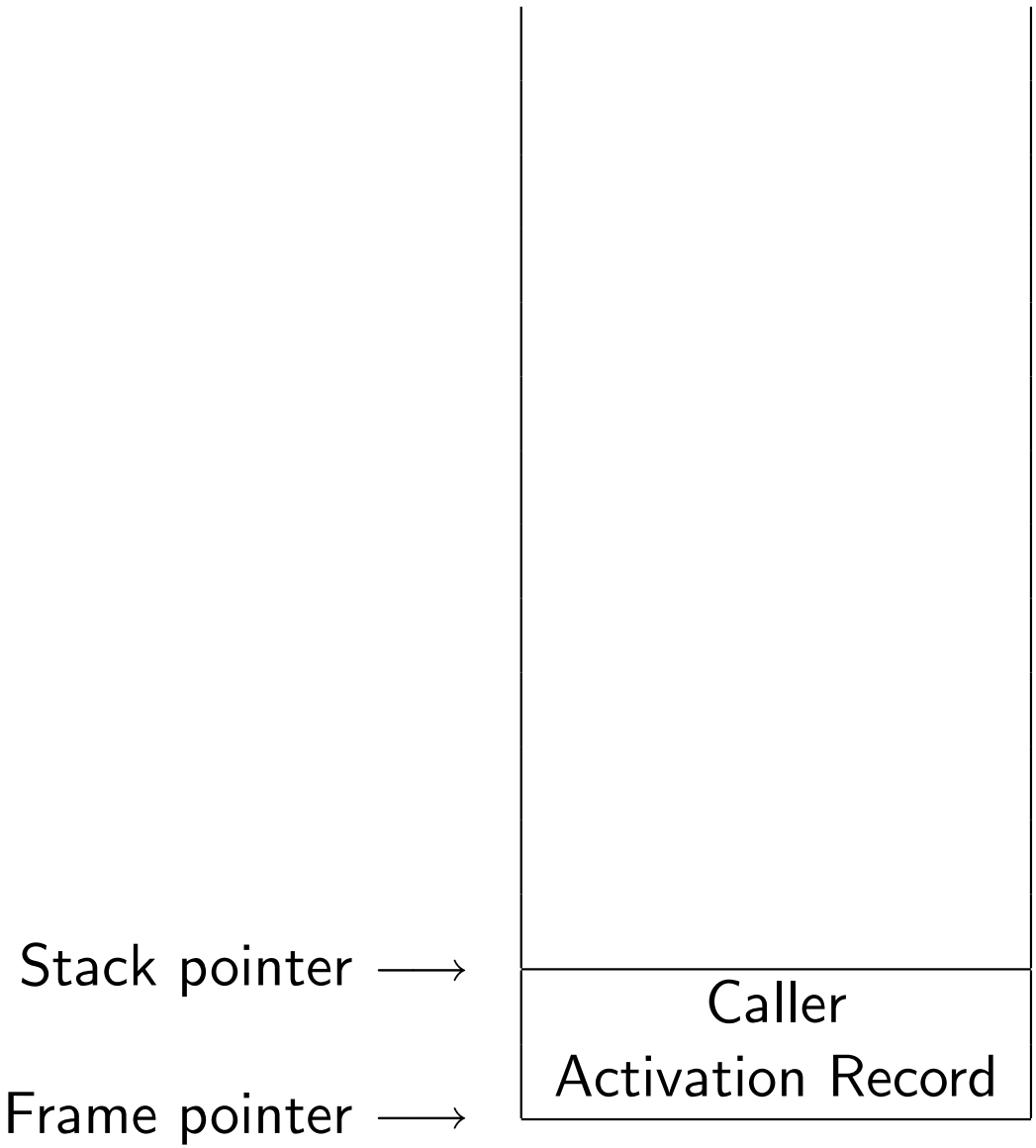
Save all registers that will be used by callee.

Which is better?
Could be either one–no clear answer.

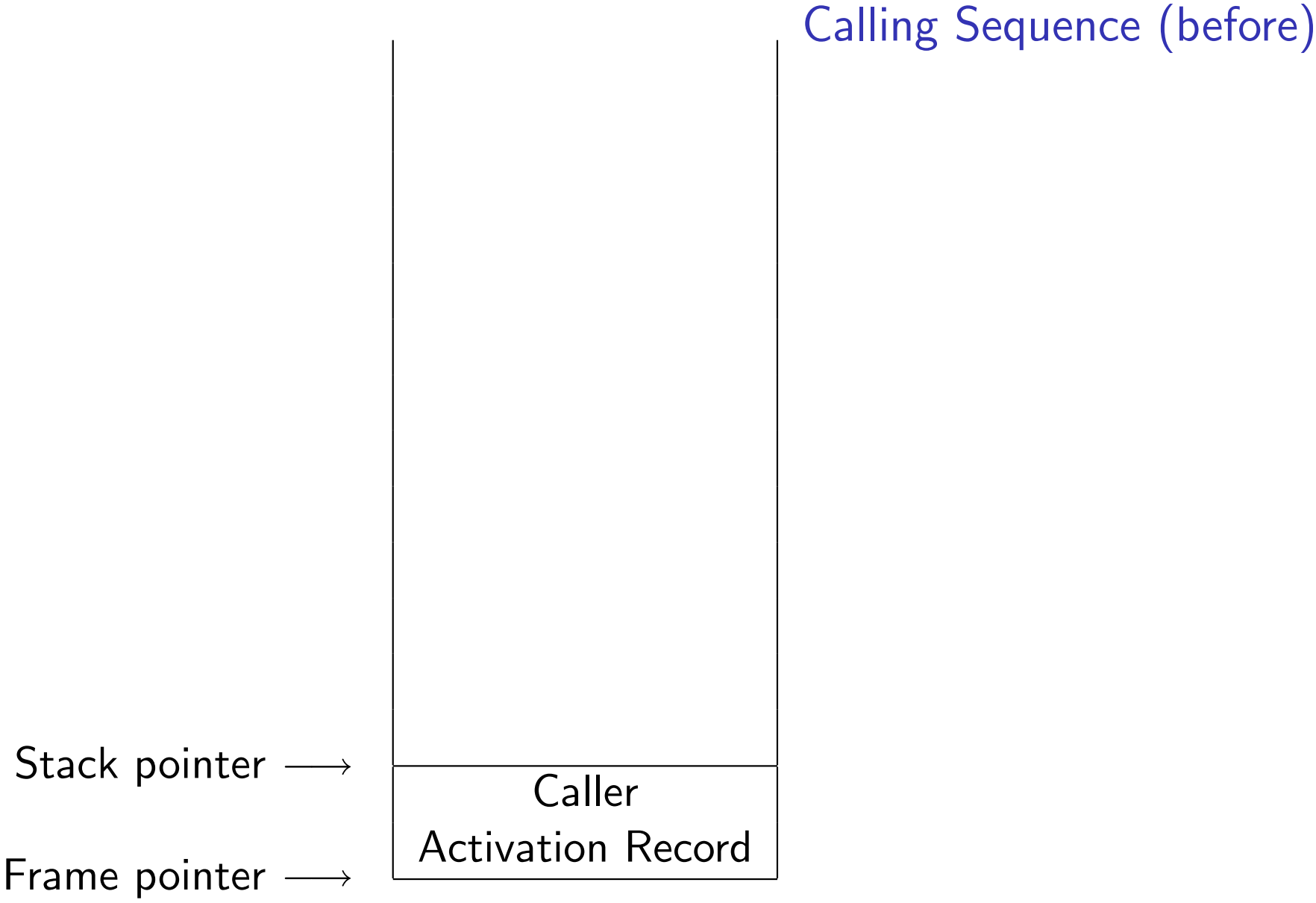
In practice, many processors (including MIPS and x86) compromise: half the registers are caller-save and half are callee-save.

Register Windows offer an alternative: each routine has access only to a small *window* of a large number of registers; when a subroutine is called, the window moves, overlapping a bit to allow parameter passing.

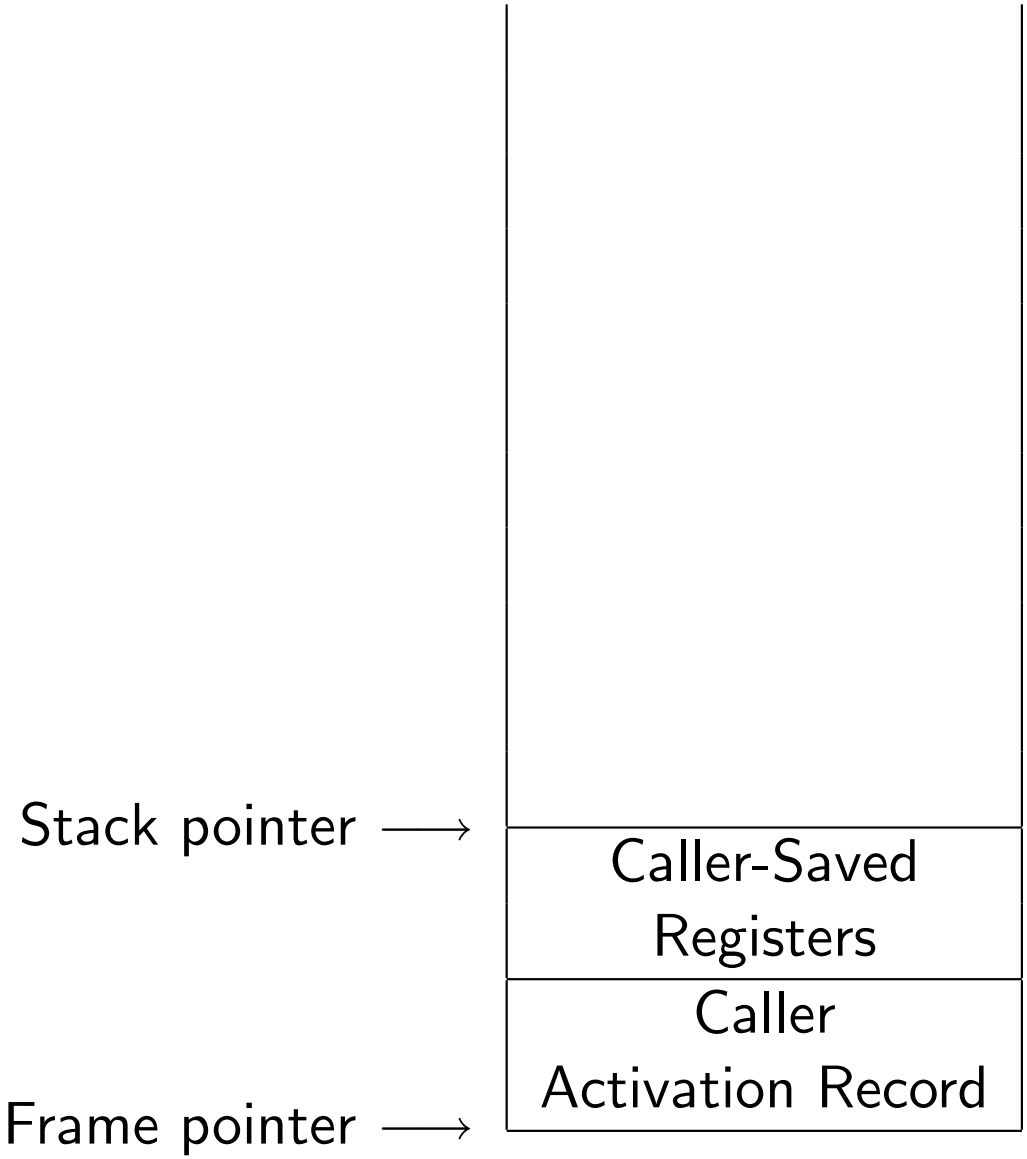
Calling a Subroutine



Calling a Subroutine



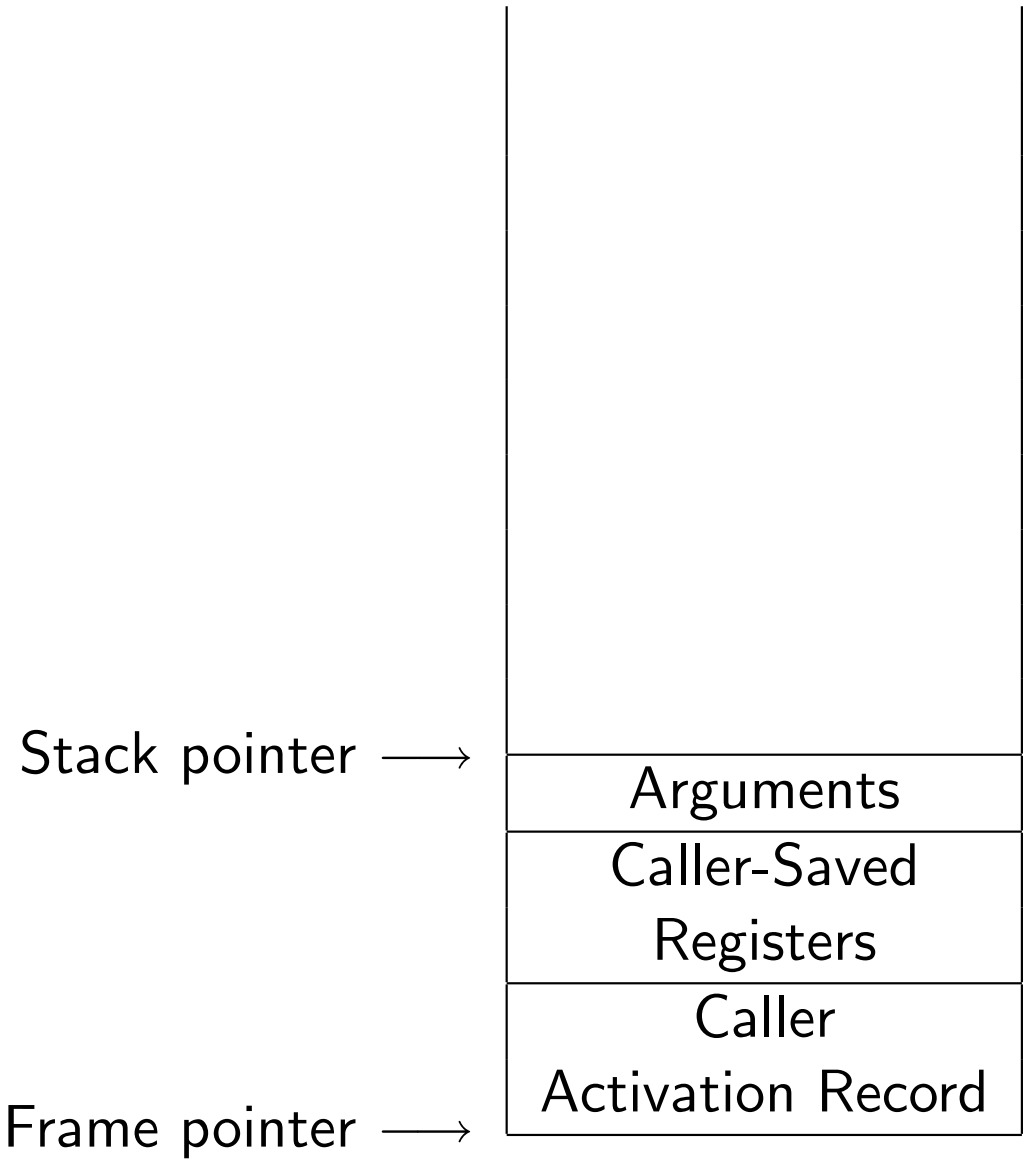
Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers

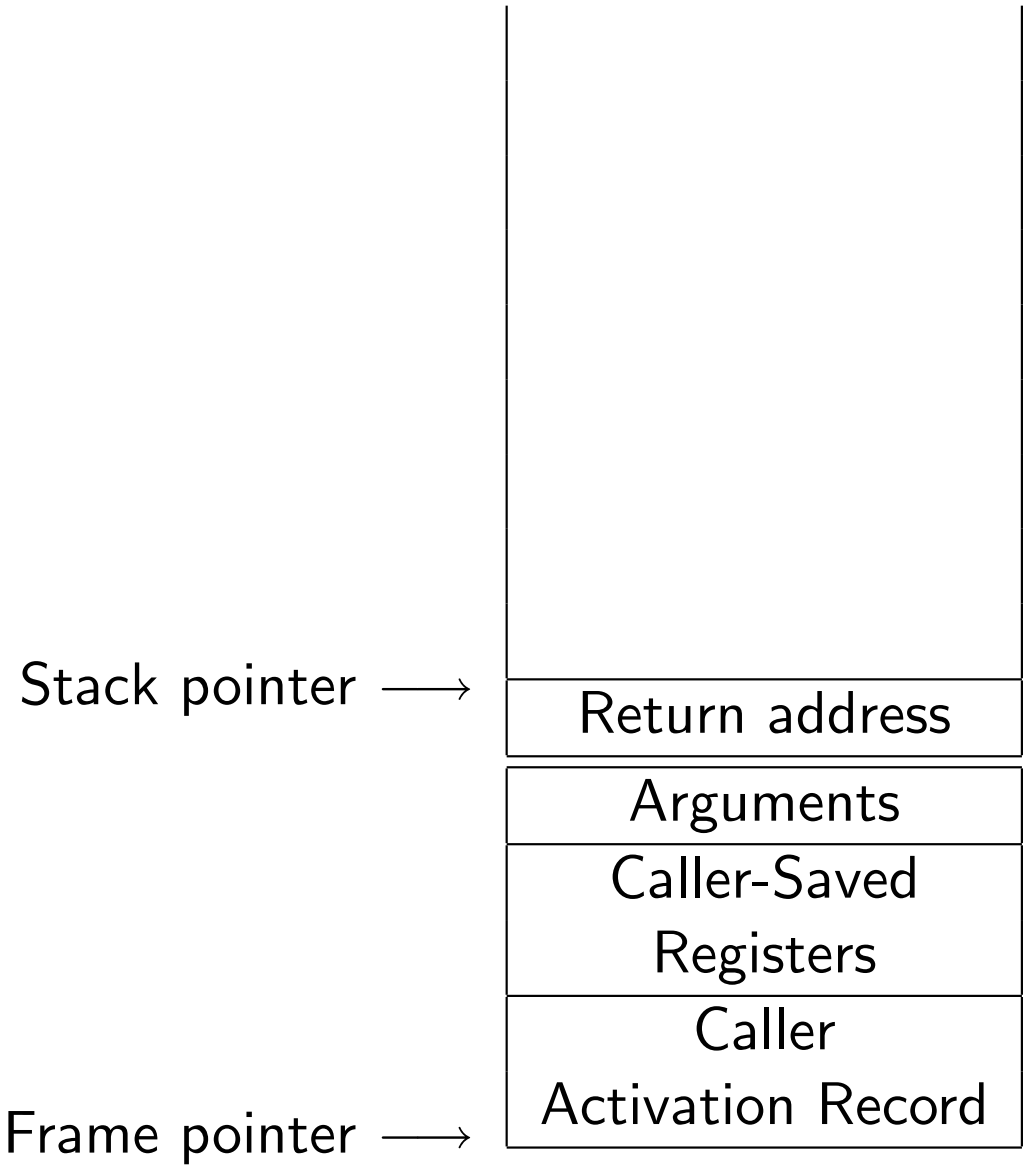
Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack

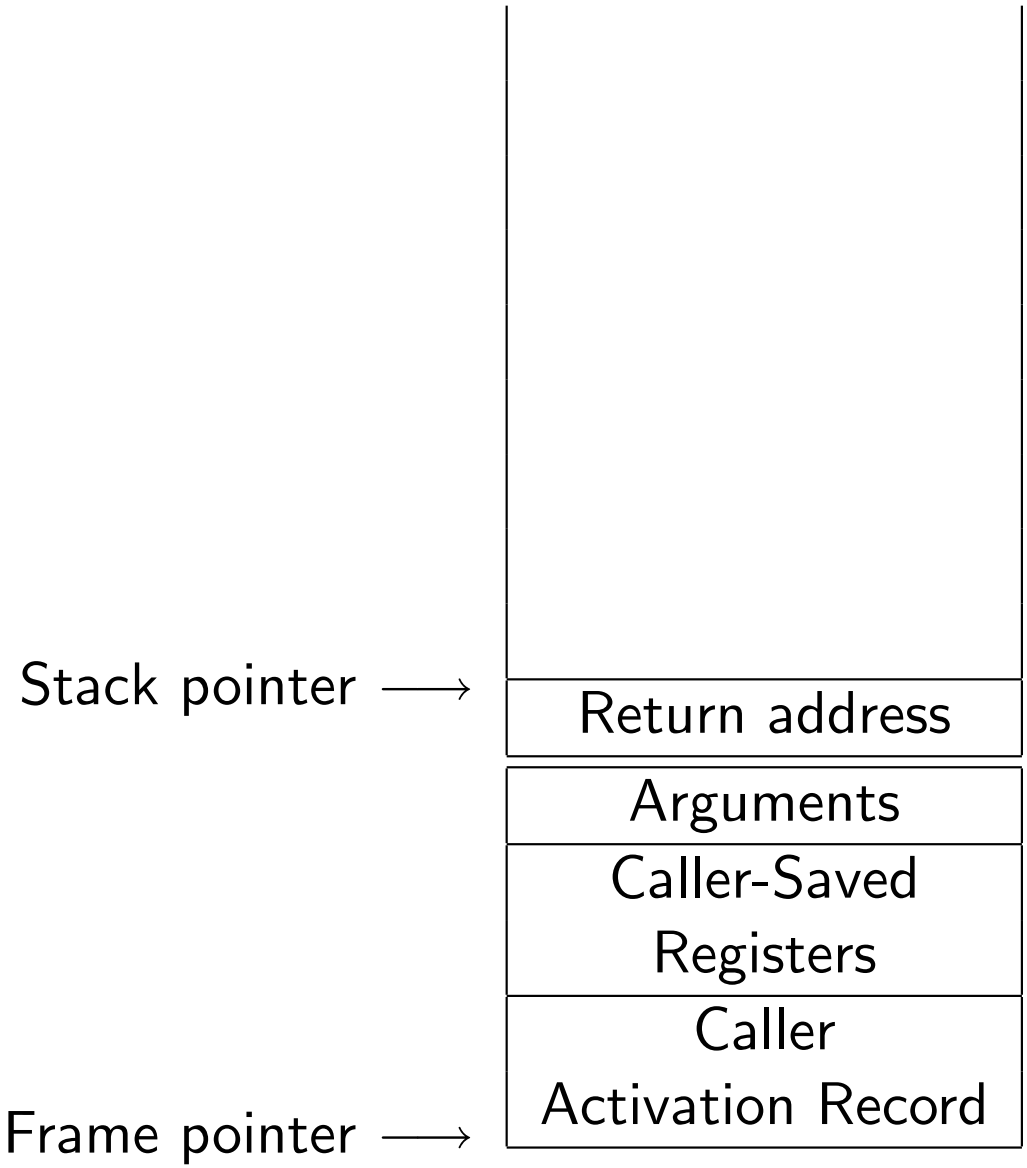
Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Typical Calling Sequence

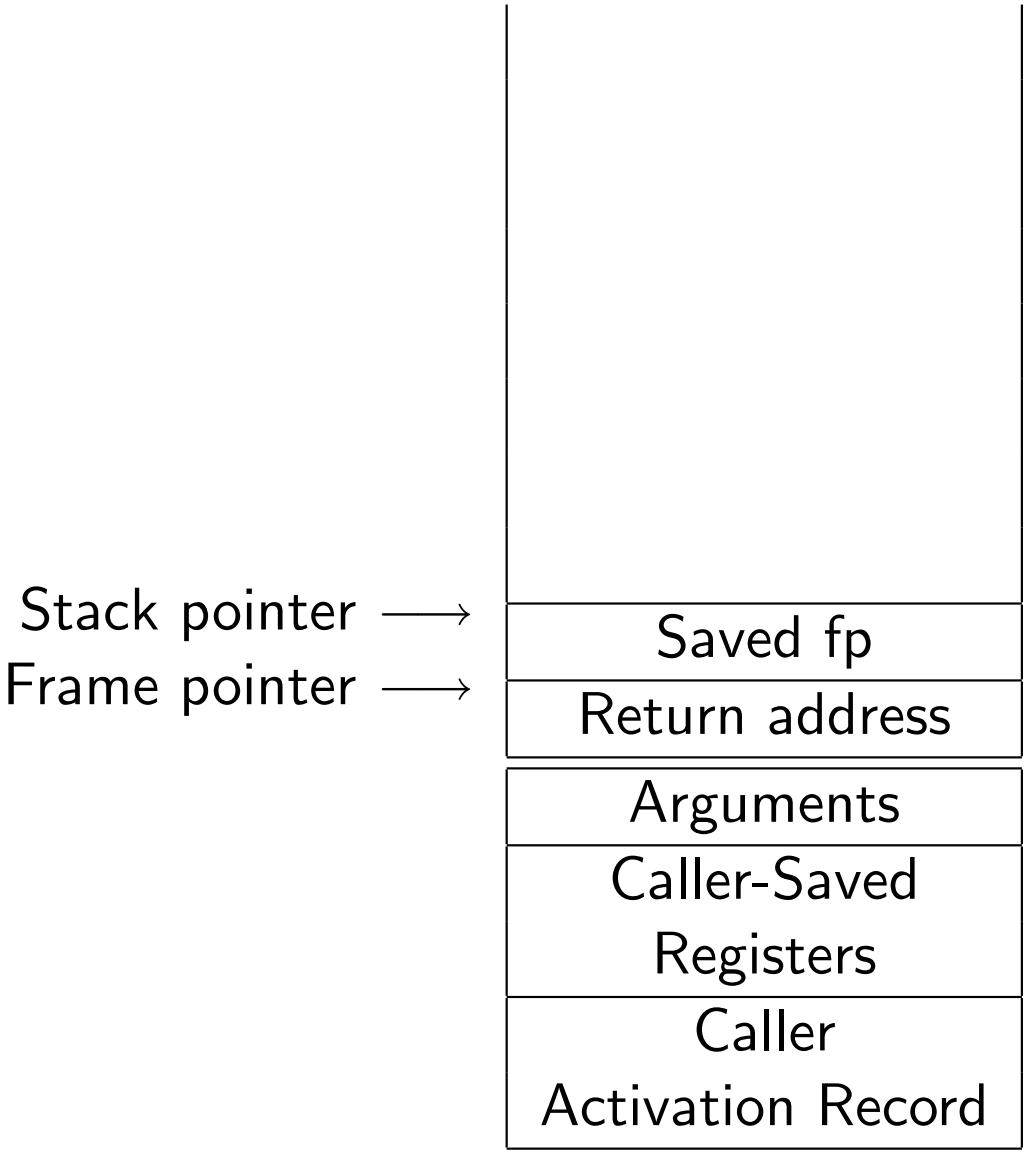


Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

Typical Calling Sequence



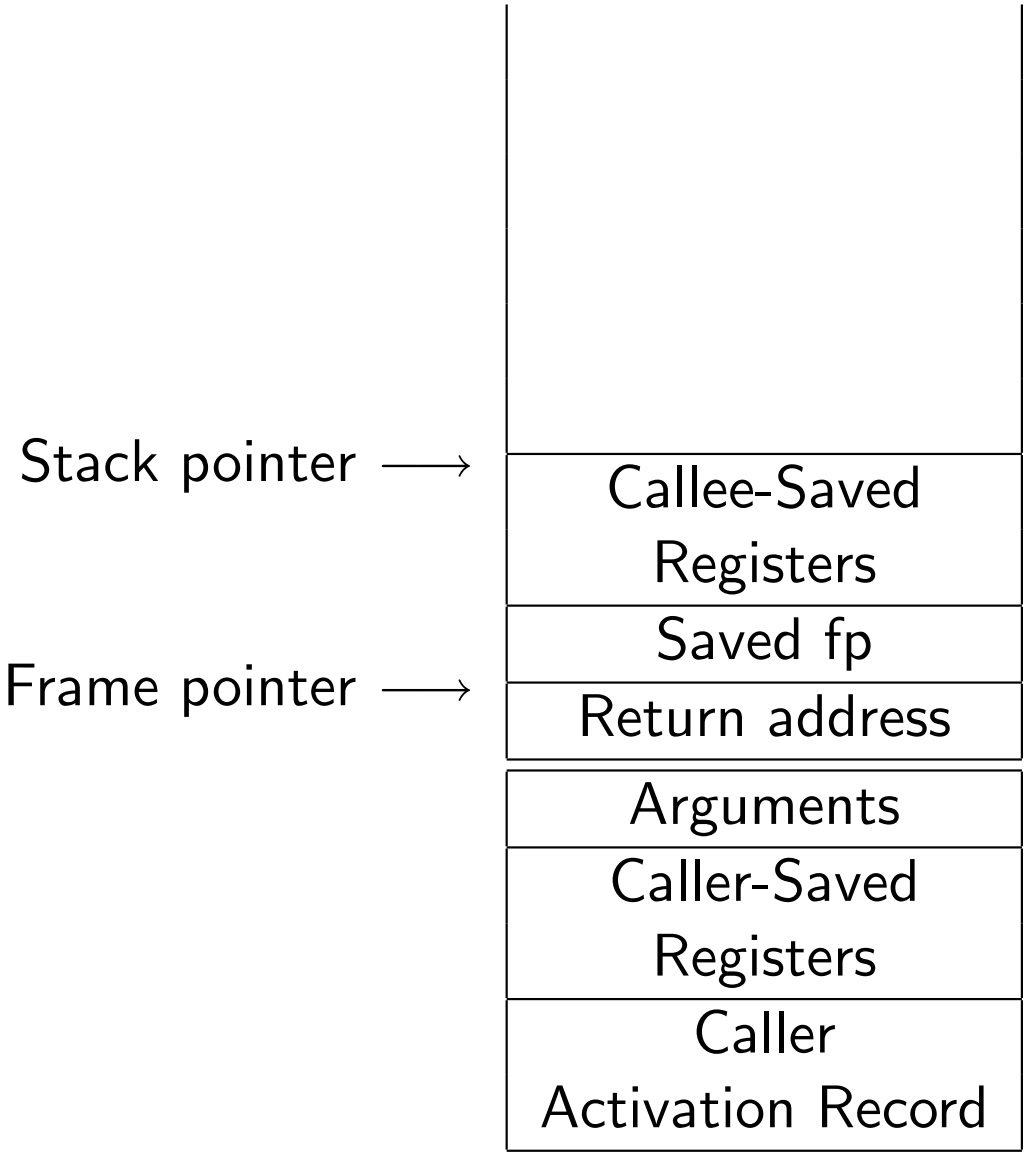
Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

1. Save old fp, set new fp

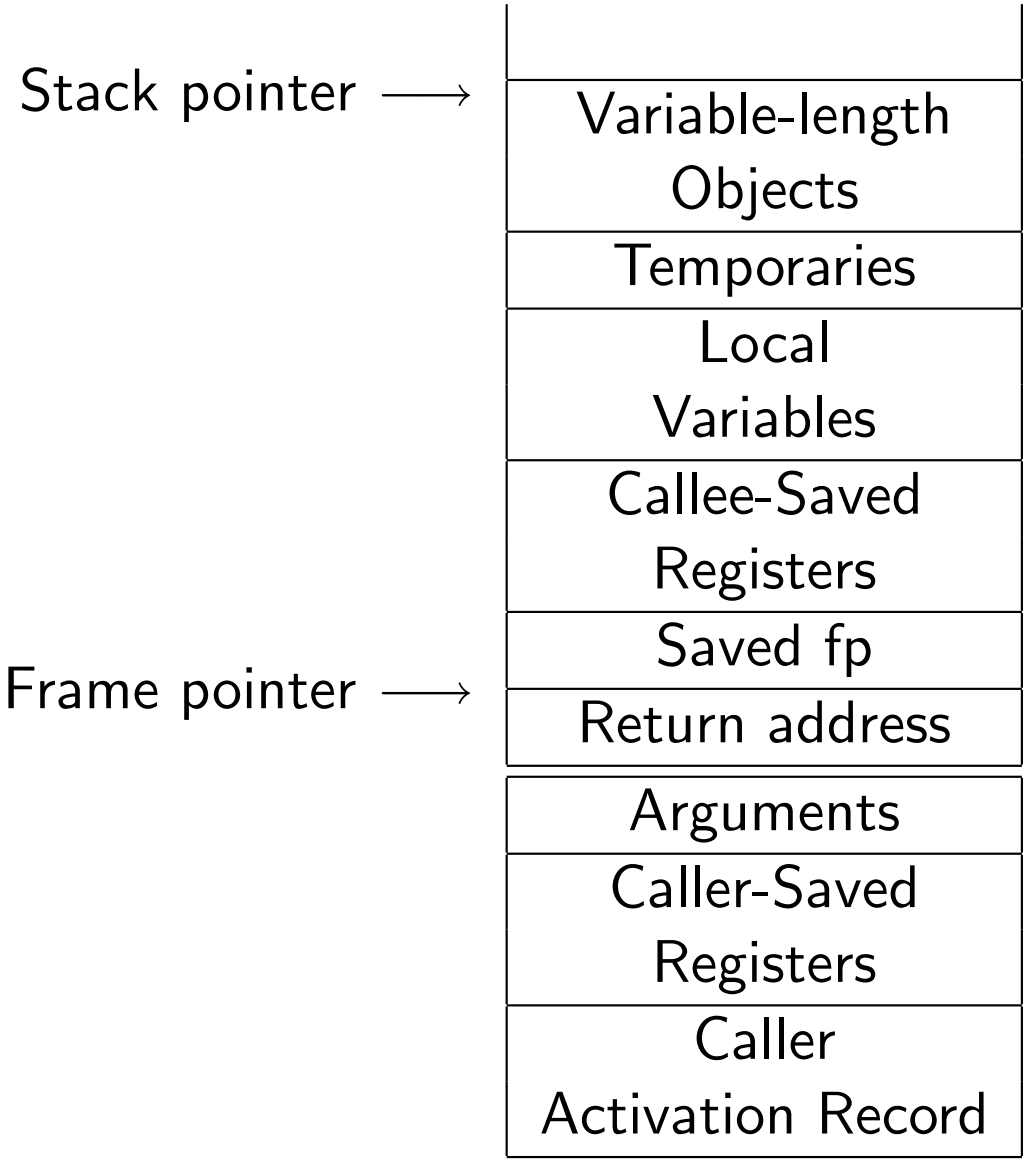
Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers

Typical Calling Sequence



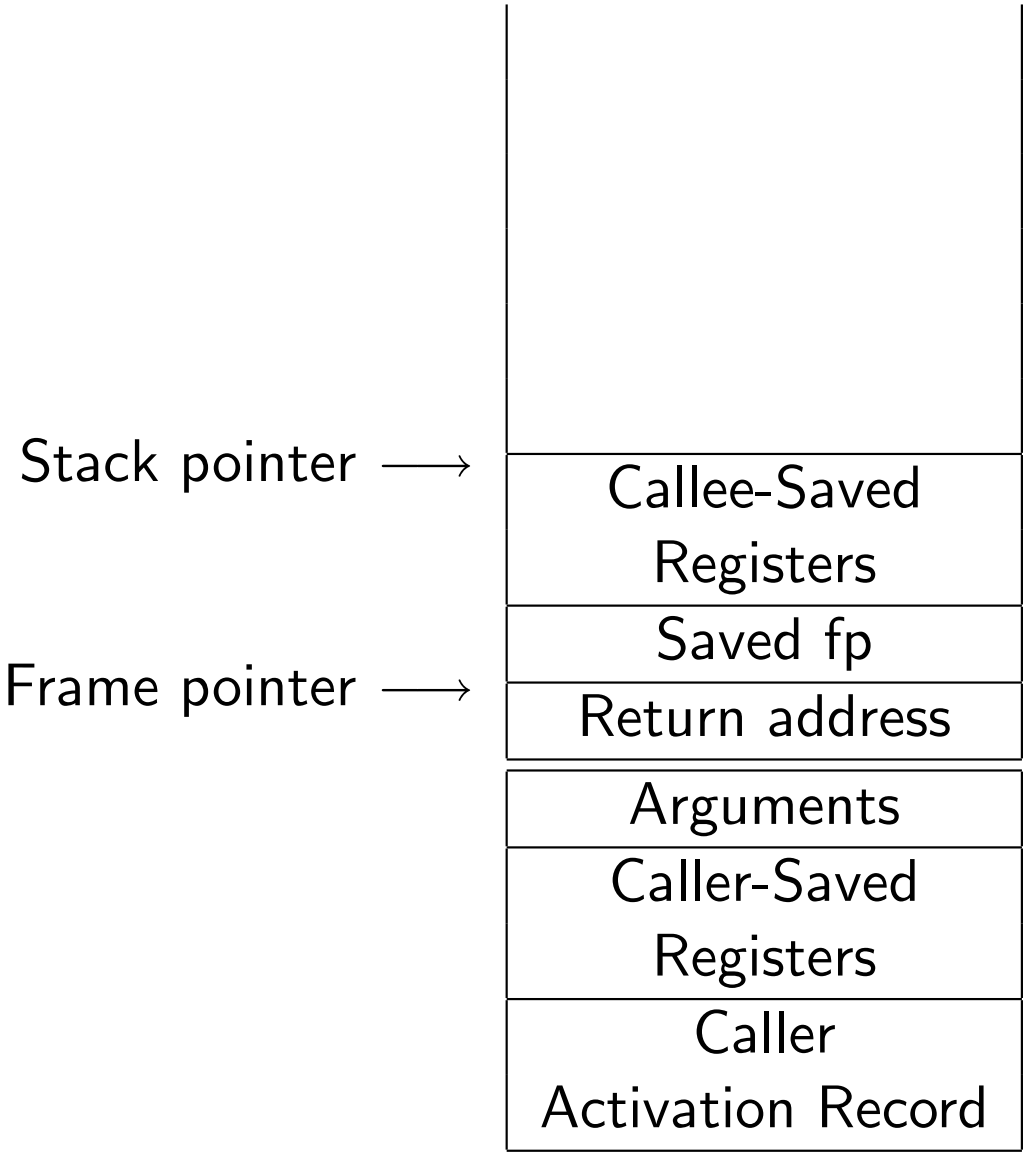
Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

1. Save old fp, set new fp
2. Save callee-save registers

Typical Calling Sequence



Calling Sequence (before)

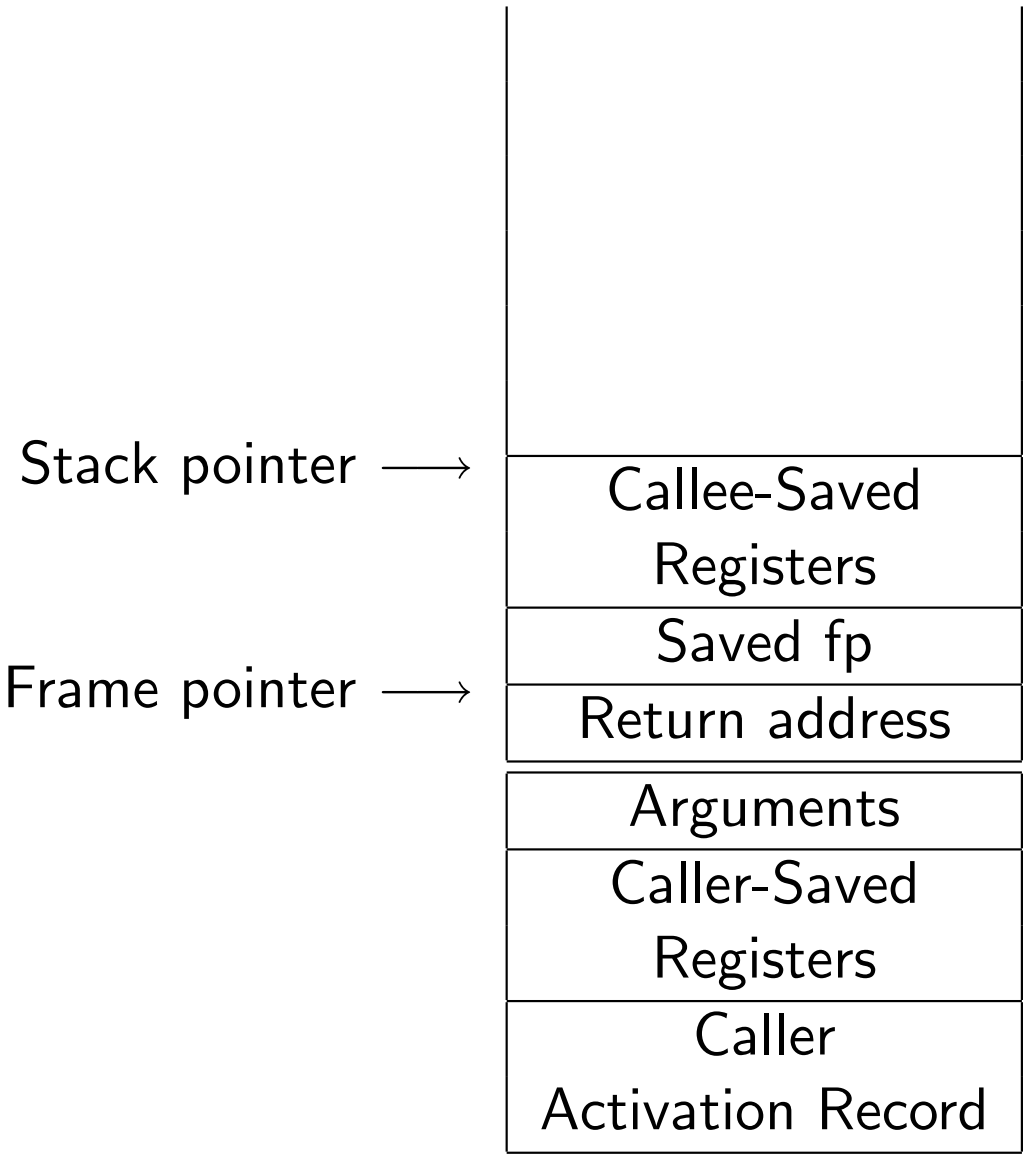
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

1. Save old fp, set new fp
2. Save callee-save registers

Epilogue

Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

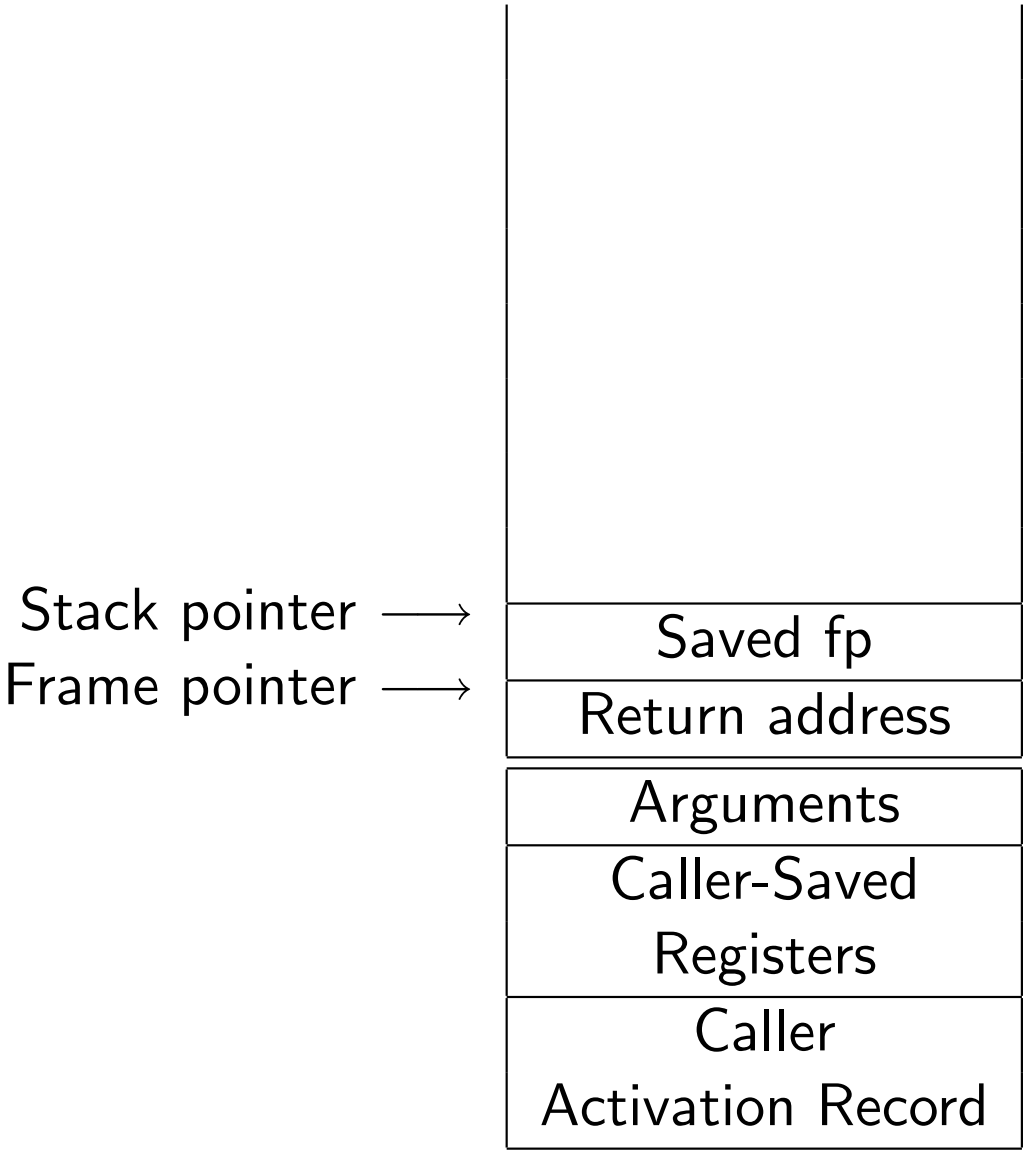
Prologue

1. Save old fp, set new fp
2. Save callee-save registers

Epilogue

1. Restore callee-save registers

Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

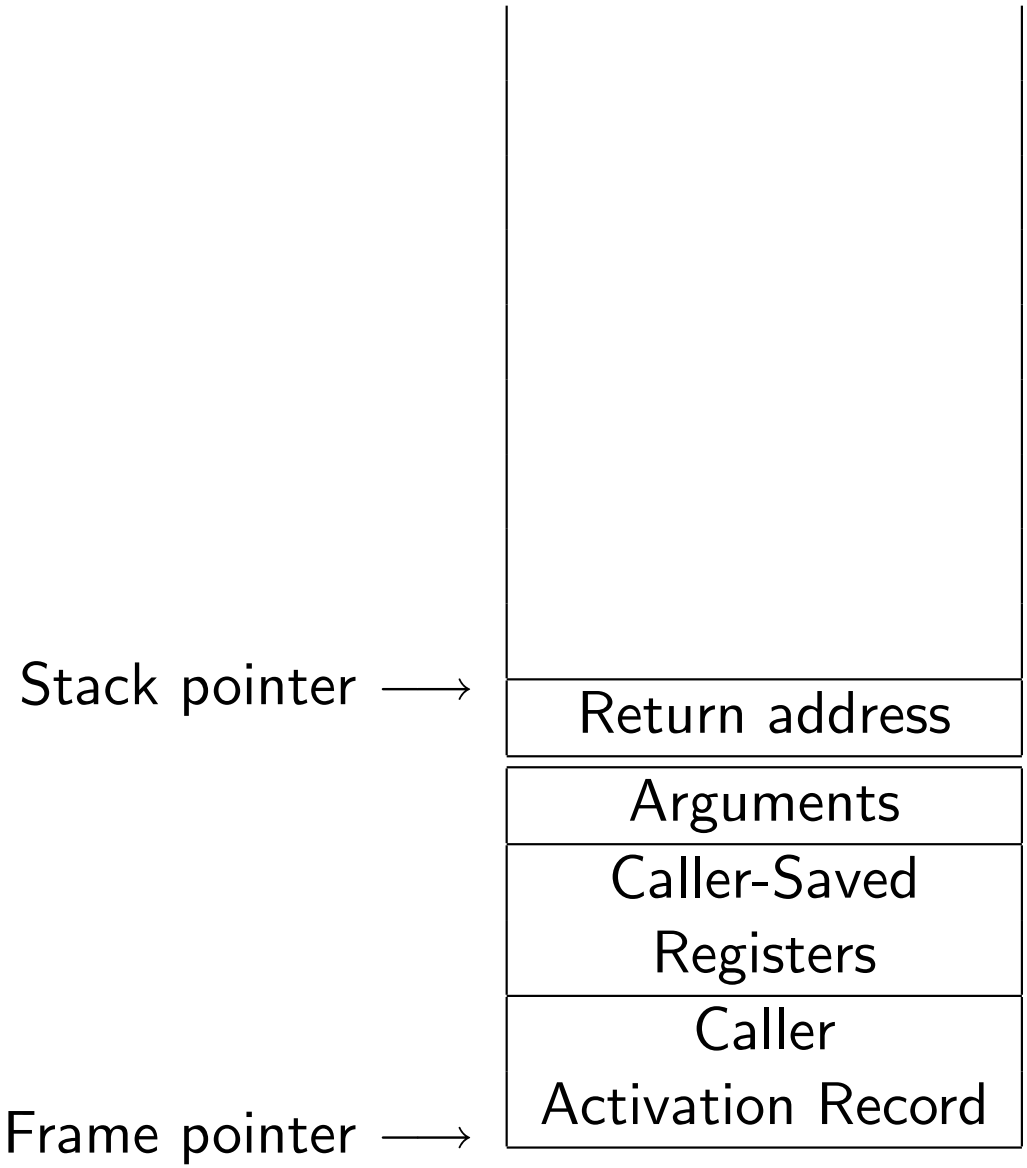
Prologue

1. Save old fp, set new fp
2. Save callee-save registers

Epilogue

1. Restore callee-save registers
2. Restore frame pointer

Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

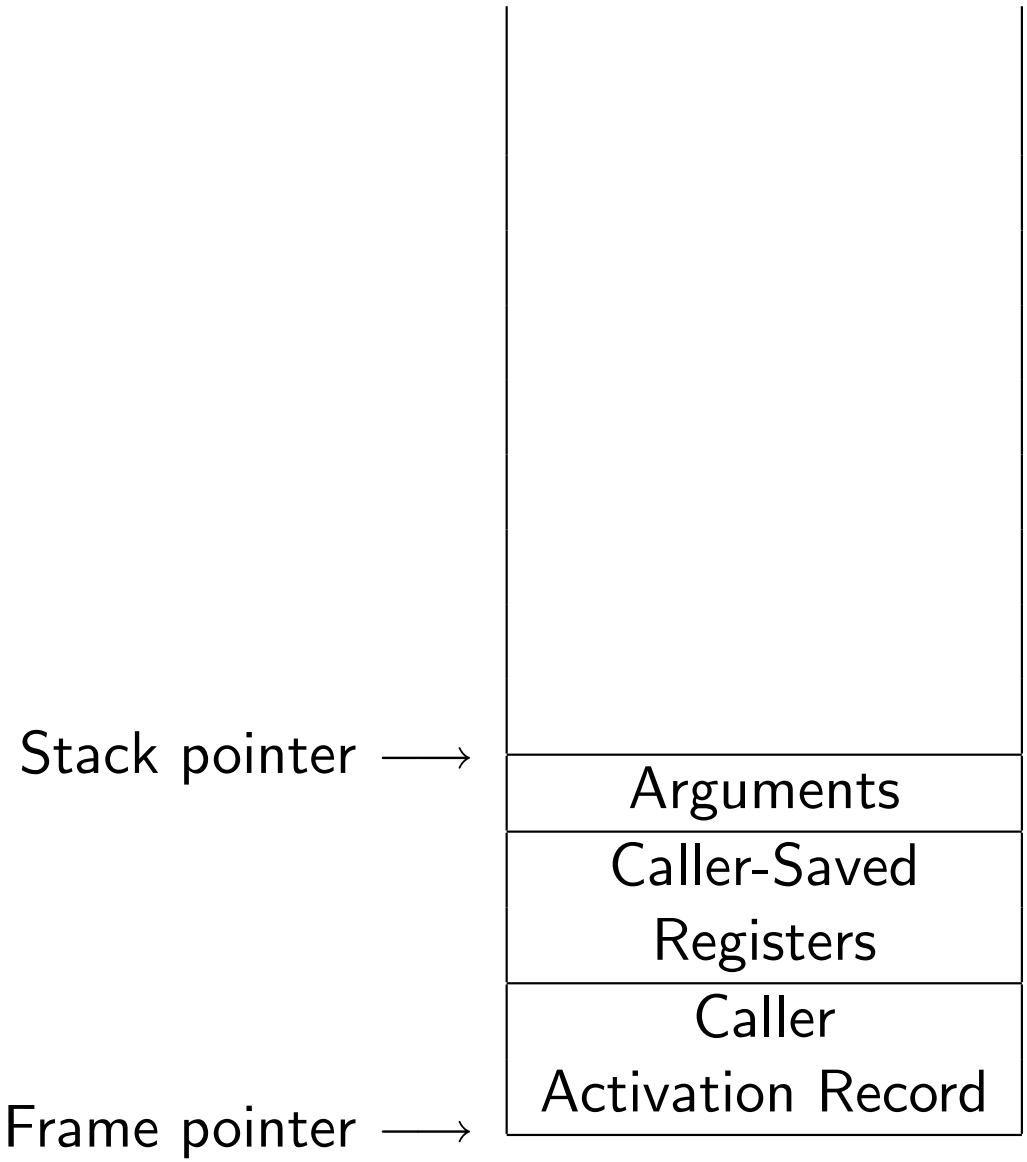
Prologue

1. Save old fp, set new fp
2. Save callee-save registers

Epilogue

1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

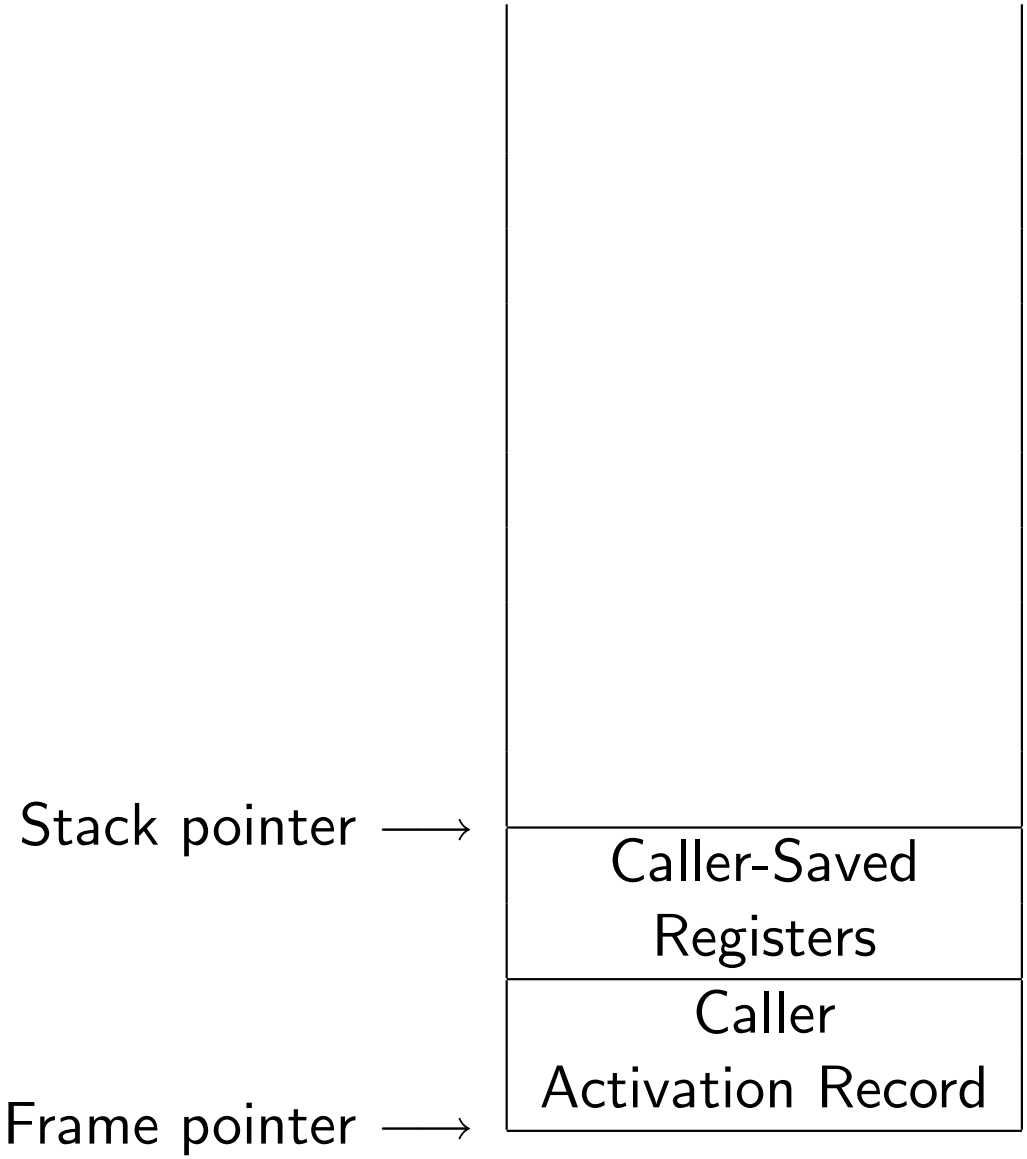
1. Save old fp, set new fp
2. Save callee-save registers

Epilogue

1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Calling Sequence (after)

Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

1. Save old fp, set new fp
2. Save callee-save registers

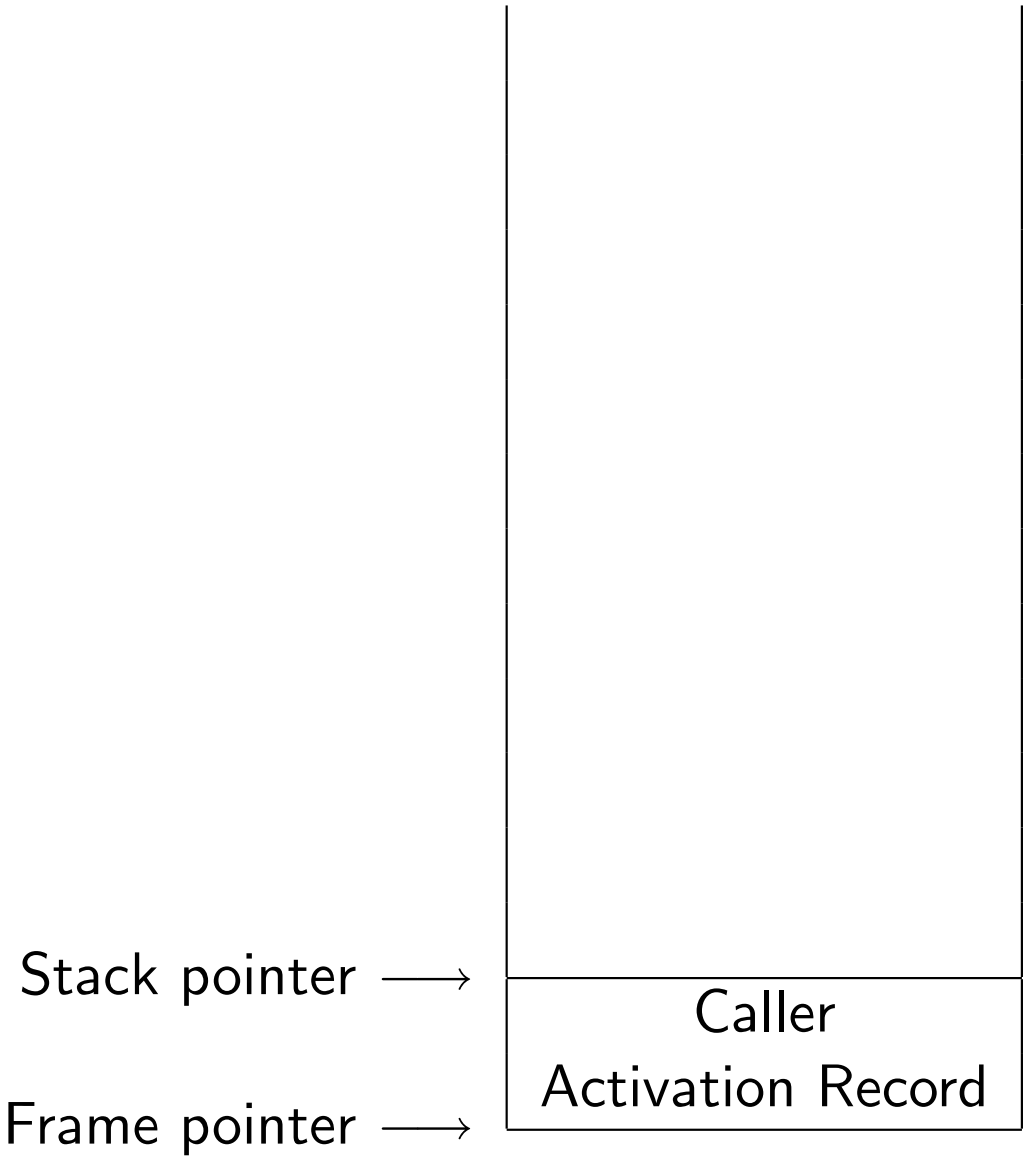
Epilogue

1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Calling Sequence (after)

1. Restore caller-save registers

Typical Calling Sequence



Calling Sequence (before)

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

1. Save old fp, set new fp
2. Save callee-save registers

Epilogue

1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Calling Sequence (after)

1. Restore caller-save registers

Optimizations

Leaf routines

A *leaf routine* is one which does not call any subroutines.

Leaf routines can avoid pushing the return address on the stack: it can just be left in a register.

If a leaf routine is sufficiently simple (no local variables), it may not even need a stack frame at all.

Optimizations

Leaf routines

A *leaf routine* is one which does not call any subroutines.

Leaf routines can avoid pushing the return address on the stack: it can just be left in a register.

If a leaf routine is sufficiently simple (no local variables), it may not even need a stack frame at all.

Inlining

Another optimization is to *inline* a function: inserting the code for the function at every call site.

Optimizations

Leaf routines

A *leaf routine* is one which does not call any subroutines.

Leaf routines can avoid pushing the return address on the stack: it can just be left in a register.

If a leaf routine is sufficiently simple (no local variables), it may not even need a stack frame at all.

Inlining

Another optimization is to *inline* a function: inserting the code for the function at every call site.

What are advantages and disadvantages of inlining?

Optimizations

Leaf routines

A *leaf routine* is one which does not call any subroutines.

Leaf routines can avoid pushing the return address on the stack: it can just be left in a register.

If a leaf routine is sufficiently simple (no local variables), it may not even need a stack frame at all.

Inlining

Another optimization is to *inline* a function: inserting the code for the function at every call site.

What are advantages and disadvantages of inlining?

- *Advantages*: avoid overhead, enable more compiler optimizations
- *Disadvantages*: increases code size, can't always do it (i.e. recursive procedures)

Parameter Passing

Definitions

- *Formal parameters* are the names that appear in the declaration of the subroutine.
- *Actual parameters* or *arguments* refer to the expressions passed to a subroutine at a particular call site.

```
// formal parameters: a, b, c
function f (int a, int b, int c)
   ...

// arguments: i, 2/i, g(i,j)
f(i, 2/i, g(i,j));
```

Parameter passing

Modes

What does a reference to a formal parameter in the execution of a subroutine mean in terms of the actual parameters?

It depends on the parameter *mode*.

- *by value*: formal is bound to value of actual
- *by reference*: formal is bound to location of actual
- *by copy-return*: formal is bound to value of actual; upon return from routine, actual gets copy of formal
- *by name*: formal is bound to expression for actual; expression evaluated whenever needed; writes to parameter are allowed (and can affect other parameters!)
- *by need*: formal is bound to expression for actual; expression evaluated the first time its value is needed; cannot write to parameters

Parameter passing

Modes

What does a reference to a formal parameter in the execution of a subroutine mean in terms of the actual parameters?

It depends on the parameter *mode*.

- *by value*: formal is bound to value of actual
- *by reference*: formal is bound to location of actual
- *by copy-return*: formal is bound to value of actual; upon return from routine, actual gets copy of formal
- *by name*: formal is bound to expression for actual; expression evaluated whenever needed; writes to parameter are allowed (and can affect other parameters!)
- *by need*: formal is bound to expression for actual; expression evaluated the first time its value is needed; cannot write to parameters

What are the advantages of passing by need?

Parameter passing

Modes

What does a reference to a formal parameter in the execution of a subroutine mean in terms of the actual parameters?

It depends on the parameter *mode*.

- *by value*: formal is bound to value of actual
- *by reference*: formal is bound to location of actual
- *by copy-return*: formal is bound to value of actual; upon return from routine, actual gets copy of formal
- *by name*: formal is bound to expression for actual; expression evaluated whenever needed; writes to parameter are allowed (and can affect other parameters!)
- *by need*: formal is bound to expression for actual; expression evaluated the first time its value is needed; cannot write to parameters

How does reference differ from copy-return?

Reference vs Copy-Return

Consider the following PASCAL program:

```
  var
    global: integer := 10;
    another: integer := 2;
  procedure confuse (var first, second: integer);
  begin
    first := first + global;
    second := first * global;
  end;
begin
  confuse(global, another); /* first and global */
                            /* are aliased */
end
```

- different results if by reference or by copy-return
- such programs are considered erroneous in ADA
- passing by value with copy-return is less error-prone

Parameter Passing in C

- C: parameter passing is always by value: assignment to formal is assignment to local copy
- passing by reference can be simulated by using pointers

```
void incr (int *x) {
  (*x)++;
}
incr(&counter); /* pointer to counter */
```

- no need to distinguish between functions and procedures: `void` indicates side-effects only

Parameter Passing in C++

- default is by-value (same semantics as C)
- explicit reference parameters also allowed:

```
void incr (int& y) {
  y++;
}

// compiler knows declaration of incr,
// builds reference
incr(counter);
```

- semantic intent can be indicated by qualifier:

```
// passed by reference, but call cannot
// modify it
void f (const double& val);
```

Parameter Passing in JAVA

- semantics of assignment to parameter differs for primitive types and for classes:
  - primitive types have value semantics
  - objects have reference semantics
- consequence: methods can modify objects
- for formals of primitive types: assignment allowed, only affects local copy
- for objects: `final` means that formal is read-only

Parameter Passing in Ada

- goal: separate semantic intent from implementation
- parameter modes:
  - `in` : read-only in subprogram
  - `out` : write in subprogram
  - `in out` : read-write in subprogram
- independent of whether binding by value, by reference, or by copy-return
  - `in` : bind by value or reference
  - `out` : bind by reference or copy-return
  - `in out` : bind by reference or by value/copy-return
- functions can only have `in` parameters

Passing Subroutines as Parameters

C and C++ allow parameters which are pointers to subroutines:

```
void (*pf) (int);
// pf is a pointer to a function that takes
// an int argument and returns void

typedef void (*PROC)(int);
// type abbreviation clarifies syntax

void do_it (int d) { ... }

void use_it (PROC);

PROC ptr = &do_it;

use_it(ptr);
use_it(&do_it);
```

Are there any implementation challenges for this kind of subroutine call?

Passing Subroutines as Parameters

Not really: can be implemented in the same way as a usual subroutine call: in particular the *referencing environment* can stay the same.

Passing Subroutines as Parameters

Not really: can be implemented in the same way as a usual subroutine call: in particular the *referencing environment* can stay the same.

What if a nested *subroutine is passed as a parameter?*

Passing Subroutines as Parameters

```
procedure A(I : integer; procedure P);
  procedure B;
  begin
    writeln(I);
  end;
begin
  if I > 1 then P
  else A(2, B);
end;

procedure C; begin end;

begin
  A(1, C);
end.
```

What does this program print?

Passing Subroutines as Parameters

What if a nested *subroutine is passed as a parameter?*

Deep Binding

A closure must be created and passed in place of the subroutine.

A *closure* is a reference to a subroutine together with its referencing environment.

When a subroutine is called through a closure, the referencing environment from when the closure was created is restored as part of the calling sequence.

Passing Subroutines as Parameters

What if a nested *subroutine is passed as a parameter?*

Deep Binding

A closure must be created and passed in place of the subroutine.

A *closure* is a reference to a subroutine together with its referencing environment.

When a subroutine is called through a closure, the referencing environment from when the closure was created is restored as part of the calling sequence.

Shallow Binding

When a subroutine is called, it uses the current referencing environment.

Shallow binding is typically the default in languages with dynamic scoping.

Passing Subroutines as Parameters

```
procedure A(I : integer; procedure P);
  procedure B;
  begin
    writeln(I);
  end;
begin
  if I > 1 then P
  else A(2, B);
end;

procedure C; begin end;

begin
  A(1, C);
end.
```

Deep Binding: Since the value of I is 1 when the closure for B is created, the program prints 1.

Syntactic sugar

- Default values for in-parameters (Ada)

```
function Incr (Base: Integer;
               Inc: Integer := 1)
              return Integer;
```

- `Incr(A(J))` equivalent to `Incr(A(J), 1)`
- also available in C++

```
int f (int first,
       int second = 0,
       char *handle = 0);
```

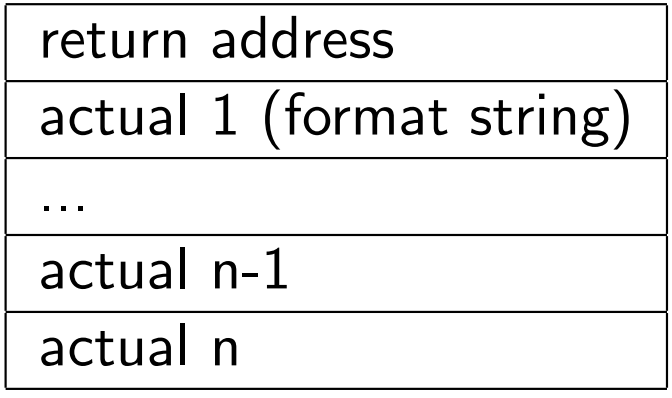
- named associations (Ada):

```
Incr(Inc => 17, Base => A(I));
```

Variable number of parameters

```
printf("this␣is␣%d␣a␣format␣%d␣string", x, y);
```

- within body of `printf`, need to locate as many actuals as placeholders in the format string
- solution: place parameters on stack in *reverse* order



First-class functions: implementation implications

Allowing functions as first-class values forces heap allocation of activation record.

- environment of function definition must be preserved until the point of call: activation record cannot be reclaimed if it creates functions
- functional languages require more complex run-time management
- higher-order functions: functions that take (other) functions as arguments and/or return functions
  - powerful
  - complex to implement efficiently
  - imperative languages restrict their use
  - (a function that takes/returns pointers to functions can be considered a higher-order function)

Recursion

In order to understand recursion, you must first understand recursion.

Recursion

In order to understand recursion, you must first understand recursion.

Recursion is when a subroutine is called from within itself.

Recursion

In order to understand recursion, you must first understand recursion.

Recursion is when a subroutine is called from within itself.

Example

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

Recursion

In order to understand recursion, you must first understand recursion.

Recursion is when a subroutine is called from within itself.

Example

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

Note that recursion requires a stack-based subroutine calling protocol.

Recursion

What are some advantages and disadvantages of using recursion?

Recursion

What are some advantages and disadvantages of using recursion?

- *Advantages*: often conceptually easier, and easier to understand code
- *Disadvantages*: usually slower, can lead to stack overflow

Recursion

What are some advantages and disadvantages of using recursion?

- *Advantages*: often conceptually easier, and easier to understand code
- *Disadvantages*: usually slower, can lead to stack overflow

There is one case when recursion can be implemented without using a stack frame for every call:

A *tail recursive* subroutine is one in which no additional computation ever follows a recursive call.

Recursion

What are some advantages and disadvantages of using recursion?

- *Advantages*: often conceptually easier, and easier to understand code
- *Disadvantages*: usually slower, can lead to stack overflow

There is one case when recursion can be implemented without using a stack frame for every call:

A *tail recursive* subroutine is one in which no additional computation ever follows a recursive call.

For tail recursive subroutines, the compiler can *reuse* the current activation record at the time of the recursive call, eliminating the need to allocate a new one.