Compilers and computer architecture

13. Dynamic memory management

Hsi-Ming Ho

(slides by Niel de Beaudrap)

Email: **hsi-ming.ho@sussex.ac.uk**
Office hours: Online, by email appointment

Autumn 2022

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

In a procedure with recursive subroutines, we generally don't know what subroutine calls we will make

$\Longrightarrow$ managed dynamically

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

In a procedure with recursive subroutines, we generally don't know what subroutine calls we will make

$\Longrightarrow$ managed dynamically

In a more complex program, we also generally don't know what persistent objects we may need to create

$\Longrightarrow$ managed dynamically

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

In a procedure with recursive subroutines, we generally don't know what subroutine calls we will make

$\Longrightarrow$ managed dynamically

In a more complex program, we also generally don't know what persistent objects we may need to create

$\Longrightarrow$ managed dynamically

The instructions for each subroutine (apart from dynamically loaded libraries) are part of the program, once compiled

$\Longrightarrow$ managed statically

A sketch: running a program

How memory is organised at run-time

When a program is invoked:

A sketch: running a program

How memory is organised at run-time

When a program is invoked:

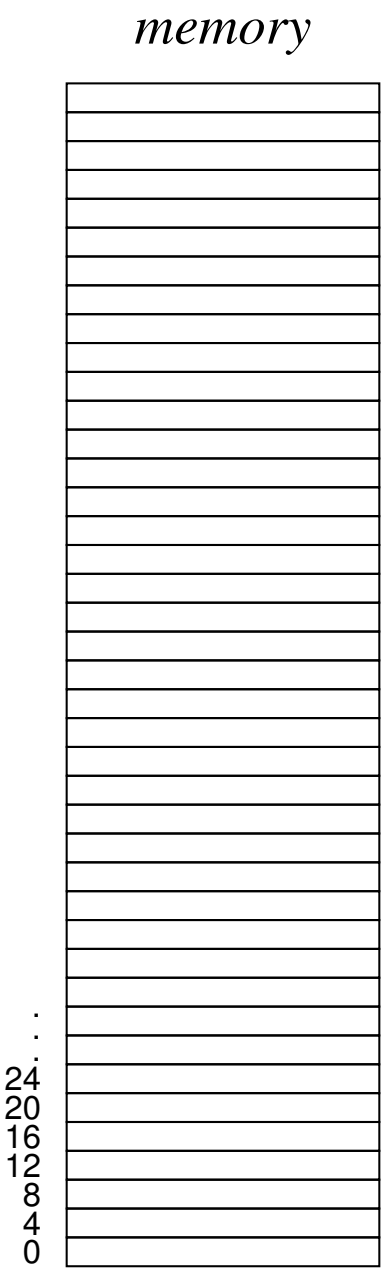
- The OS allocates space for the program
  — a block of *virtual memory*
  (both to store code and data).

A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).

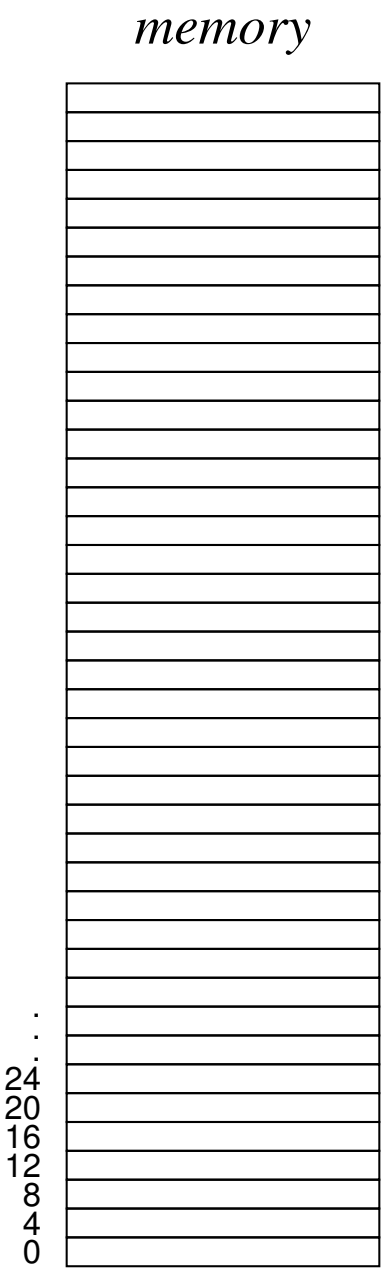


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.

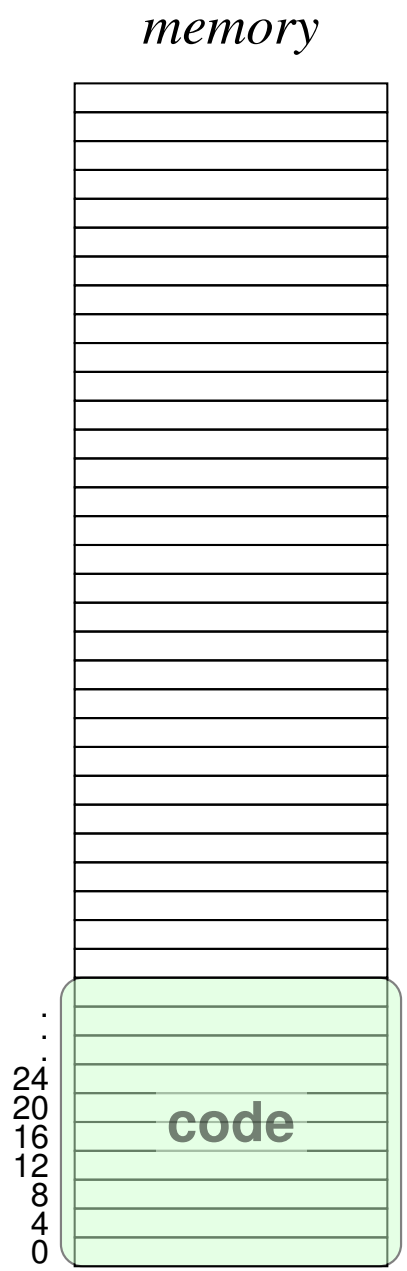


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.

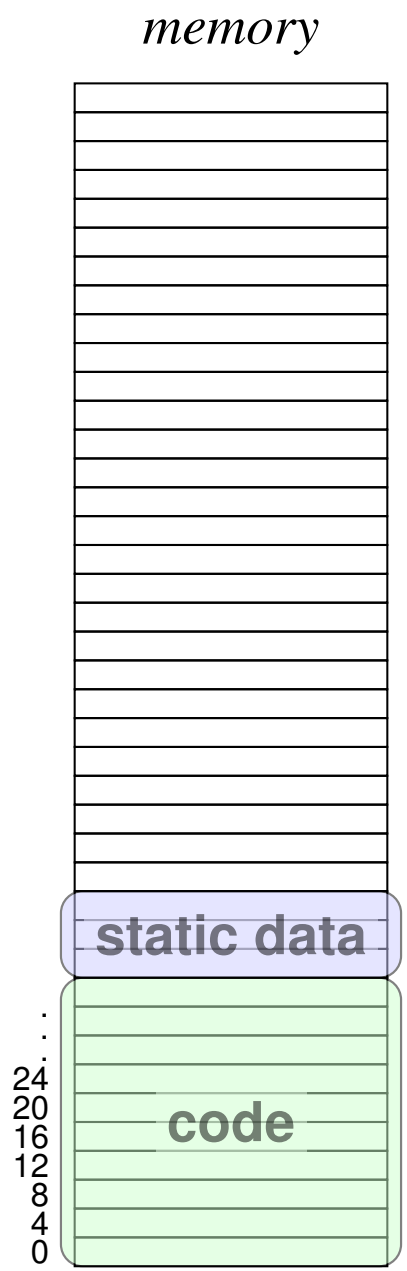


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.

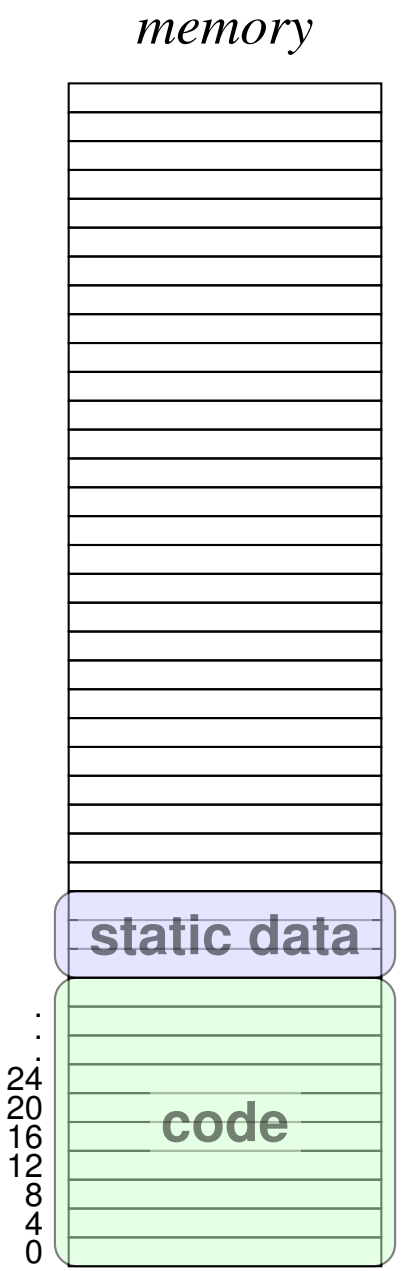


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables (‘static’ data)

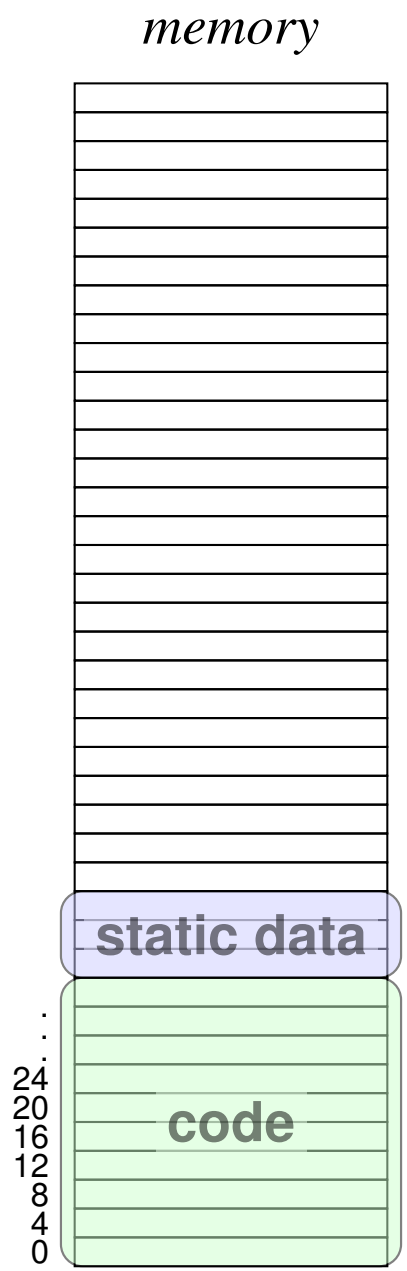


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables (‘static’ data)
  - . . . sometimes, may require linking libraries dynamically as well

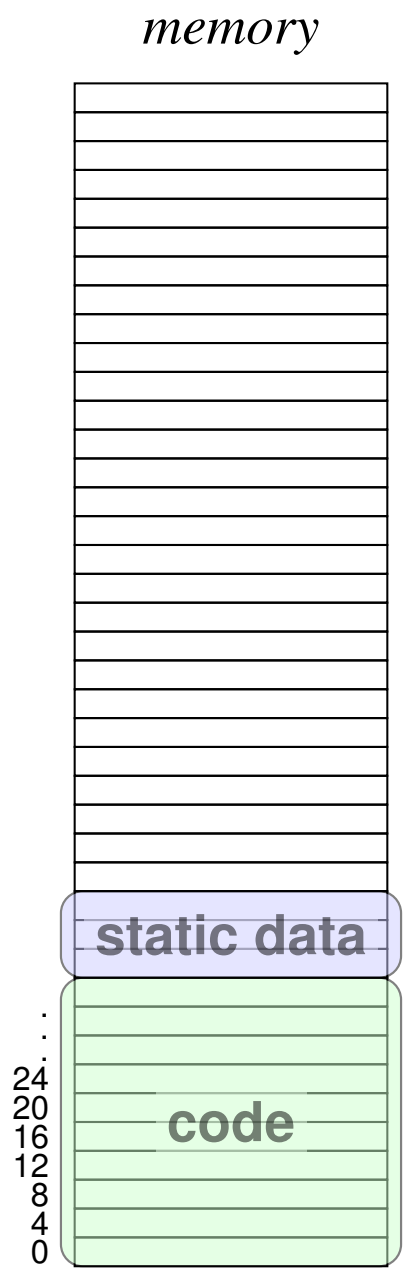


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables (‘static’ data)
  - . . . sometimes, may require linking libraries dynamically as well
  - . . . we generally expect not to modify the loaded code at run-time

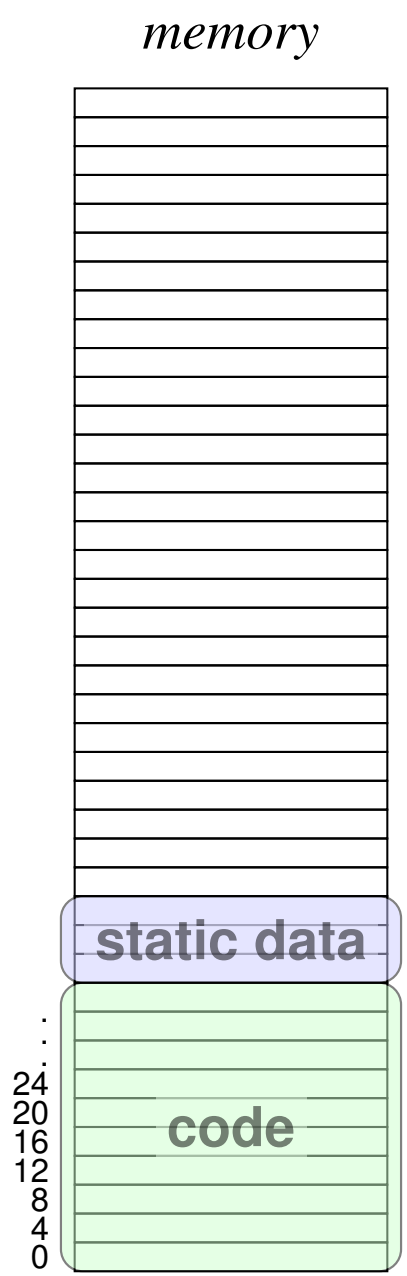


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables ('static' data)
  - . . . sometimes, may require linking libraries dynamically as well
  - . . . we generally expect not to modify the loaded code at run-time
- The OS jumps to the entry point of the program — *i.e.*, the program's entry point (*e.g.*, the `main` method in Java).

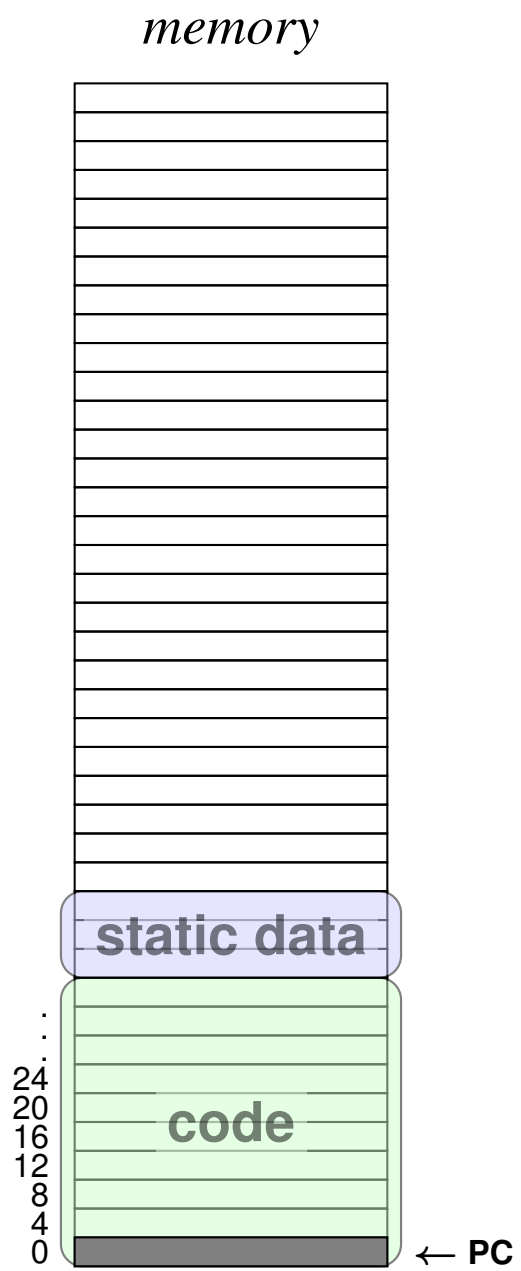


A sketch: running a program

How memory is organised at run-time

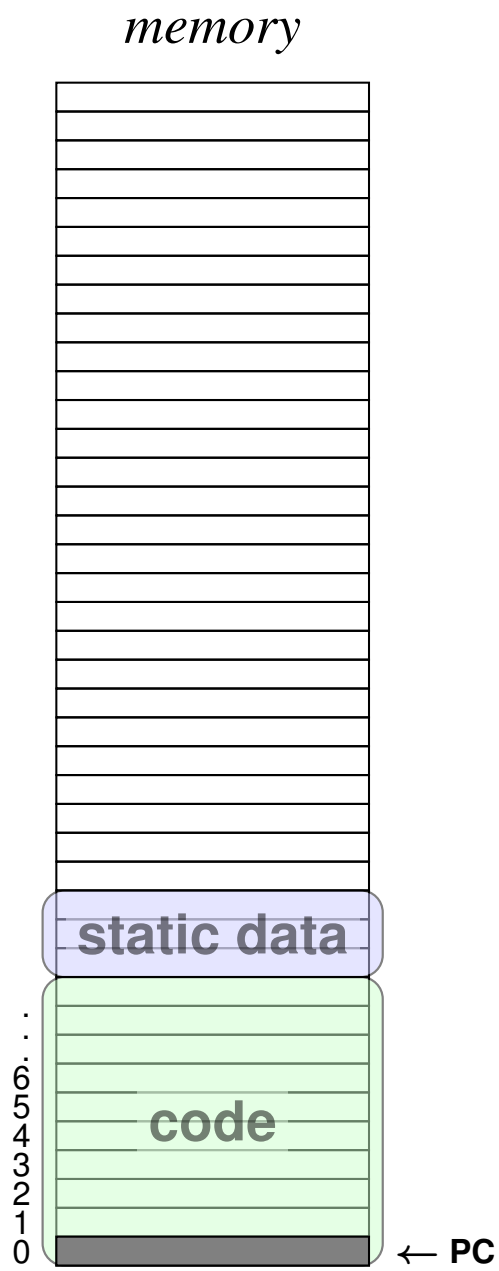
When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables ('static' data)
  - . . . sometimes, may require linking libraries dynamically as well
  - . . . we generally expect not to modify the loaded code at run-time
- The OS jumps to the entry point of the program — *i.e.*, the program's entry point (*e.g.*, the `main` method in Java).



A sketch: running a program

How memory is organised at run-time

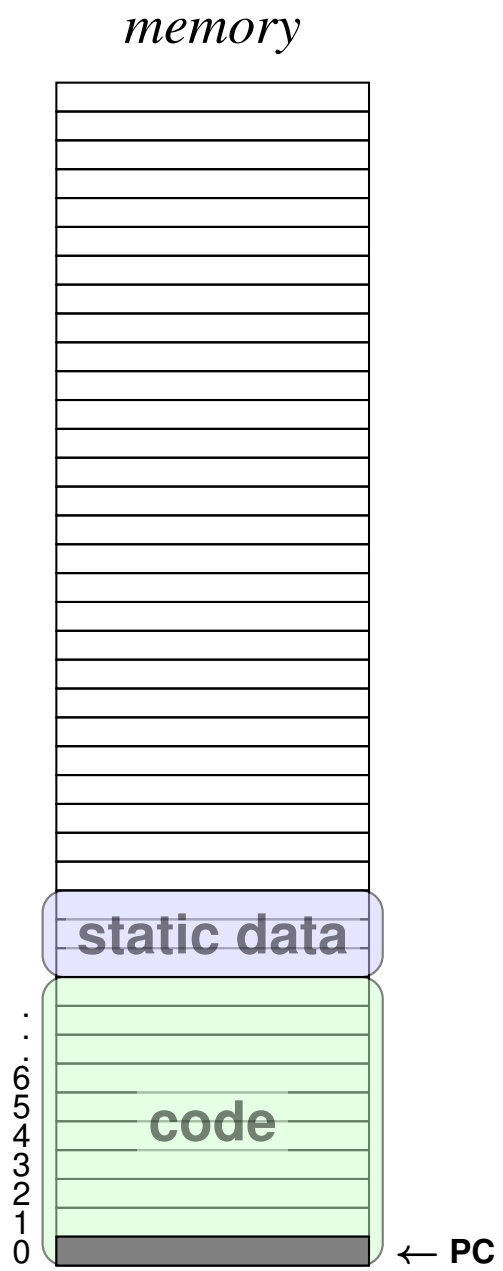


A sketch: running a program

How memory is organised at run-time

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.



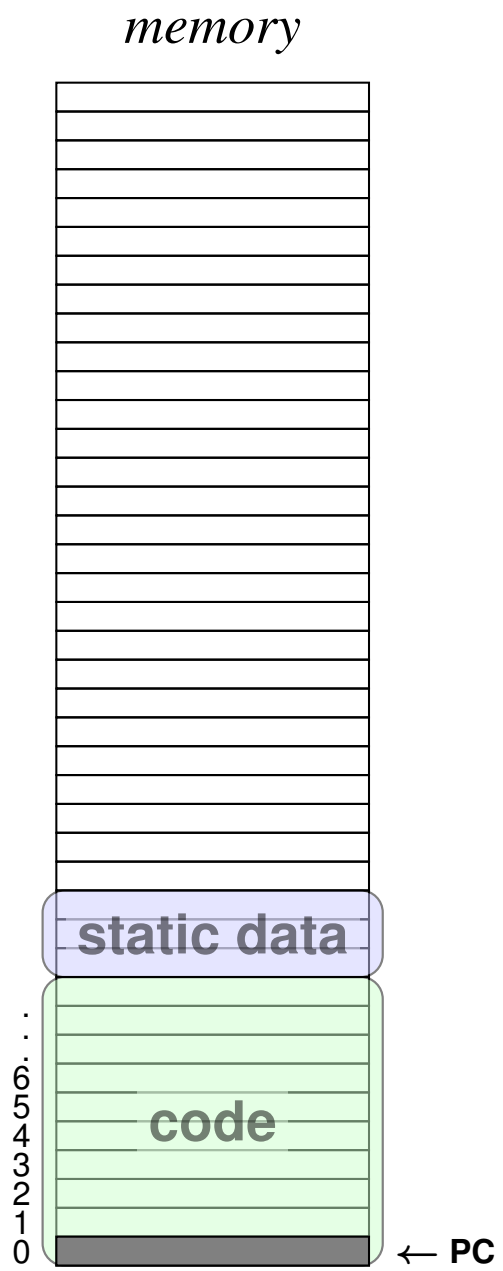
A sketch: running a program

How memory is organised at run-time

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their “lifetime” and scope



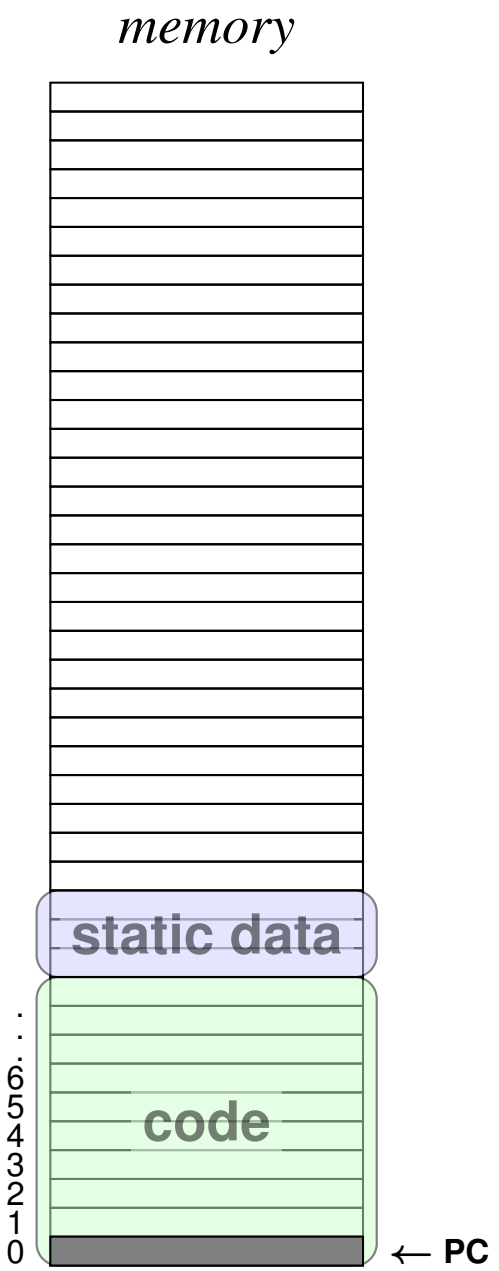
A sketch: running a program

How memory is organised at run-time

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their “lifetime” and scope
- Persistent objects: longer-lived (again by definition), more akin to statically initialised data



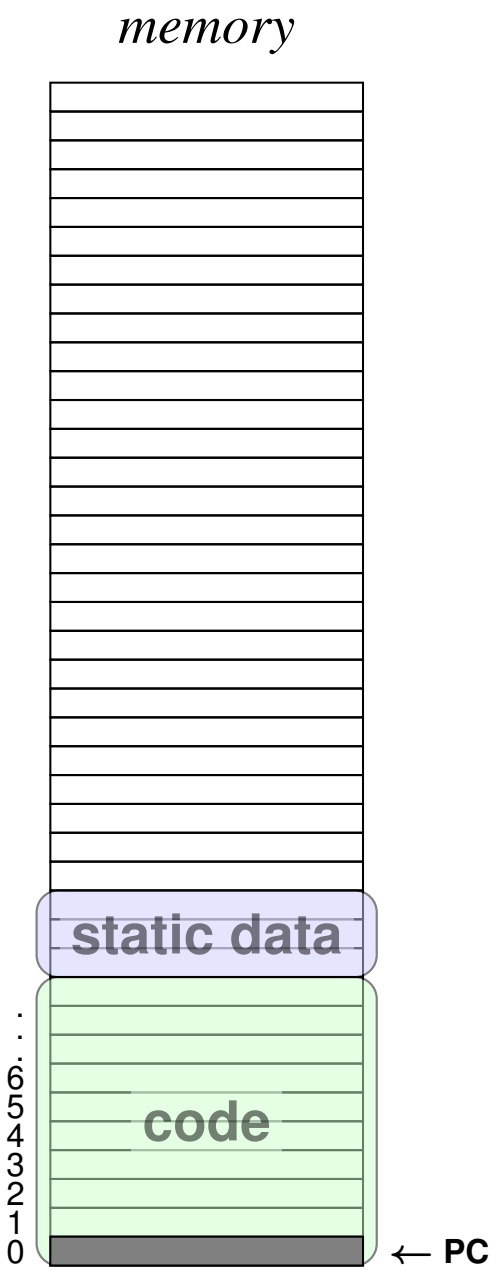
A sketch: running a program

How memory is organised at run-time

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their "lifetime" and scope
- Persistent objects: longer-lived (again by definition), more akin to statically initialised data
- Thanks to OS memory management, we may **pretend** that the *available memory* is allocated statically — but how can we use it?



A sketch: running a program

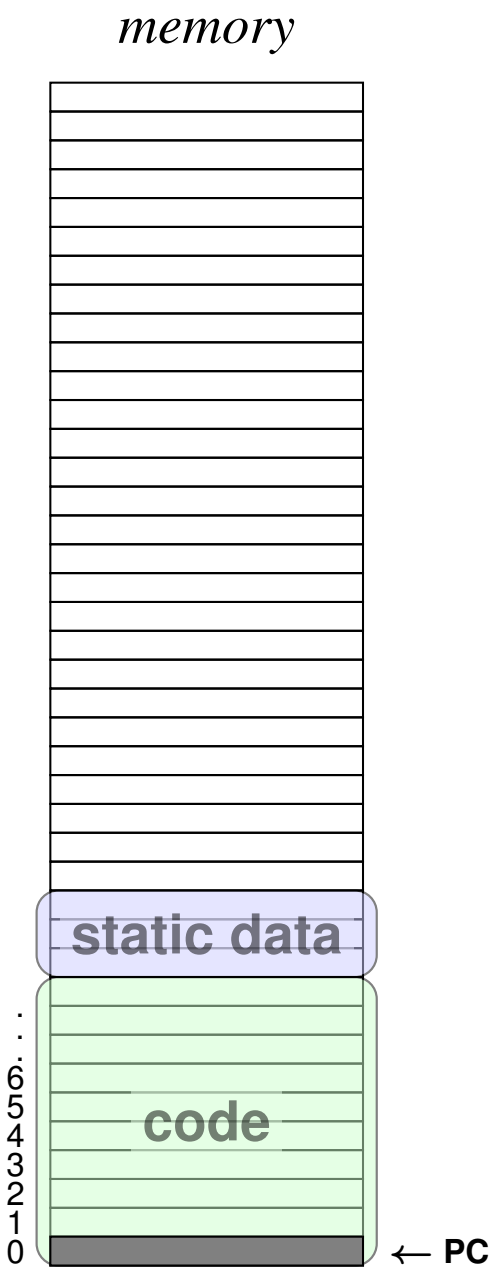
How memory is organised at run-time

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their "lifetime" and scope
- Persistent objects: longer-lived (again by definition), more akin to statically initialised data
- Thanks to OS memory management, we may **pretend** that the *available memory* is allocated statically — but how can we use it?

Code-generation must describe (at compile time) how these will be managed at run-time.



Activation / lifetime of a procedure

Let us first consider how we can store data (*e.g.*, local variables) for the execution of a subroutine `proc`.

Activation / lifetime of a procedure

Let us first consider how we can store data (*e.g.*, local variables) for the execution of a subroutine `proc`.

We introduce some terminology to help us think about procedure calls:

- An **activation** of `proc` is simply an execution of a call to `proc` — *i.e.*, an instance of running `proc`.

Activation / lifetime of a procedure

Let us first consider how we can store data (*e.g.*, local variables) for the execution of a subroutine `proc`.

We introduce some terminology to help us think about procedure calls:

- An **activation** of `proc` is simply an execution of a call to `proc` — *i.e.*, an instance of running `proc`.
- The **lifetime** of an activation of `proc` is
  - All steps to execute the activation of `proc`, including . . .

Activation / lifetime of a procedure

Let us first consider how we can store data (*e.g.*, local variables) for the execution of a subroutine `proc`.

We introduce some terminology to help us think about procedure calls:

- An **activation** of `proc` is simply an execution of a call to `proc` — *i.e.*, an instance of running `proc`.
- The **lifetime** of an activation of `proc` is
  - All steps to execute the activation of `proc`, including . . .
  - . . . all the steps in procedures that `proc` calls while running (including, possibly, recursive calls to itself).

Observations about procedure lifetimes

Consider the following Java snippet (try it, in a class declaration):

```
public static void main(String[] args){
   f(0); System.out.println();
   g(0); System.out.println();
   h(0); }
```

Observations about procedure lifetimes

Consider the following Java snippet (try it, in a class declaration):

```
public static void main(String[] args){
   f(0); System.out.println();
   g(0); System.out.println();
   h(0); }

static void f (int n) {
   report("entering f", n); g(n+1);
   report("leaving f", n); }
```

Observations about procedure lifetimes

Consider the following Java snippet (try it, in a class declaration):

```
public static void main(String[] args){
   f(0); System.out.println();
   g(0); System.out.println();
   h(0); }

static void f (int n) {
   report("entering f", n); g(n+1);
   report("leaving f", n); }

static void g (int n) {
   report("entering g", n); h(n+1);
   report("leaving g", n); }
```

Observations about procedure lifetimes

Consider the following Java snippet (try it, in a class declaration):

```
public static void main(String[] args){
   f(0); System.out.println();
   g(0); System.out.println();
   h(0); }

static void f (int n) {
   report("entering f", n); g(n+1);
   report("leaving f", n); }

static void g (int n) {
   report("entering g", n); h(n+1);
   report("leaving g", n); }

static void h (int n) {
   report("entering h", n);
   report("leaving h", n); }
```

Observations about procedure lifetimes

Consider the following Java snippet (try it, in a class declaration):

```
public static void main(String[] args){
   f(0); System.out.println();
   g(0); System.out.println();
   h(0); }

static void f (int n) {
   report("entering f", n); g(n+1);
   report("leaving f", n); }

static void g (int n) {
   report("entering g", n); h(n+1);
   report("leaving g", n); }

static void h (int n) {
   report("entering h", n);
   report("leaving h", n); }

static void report (String s, int n) {
   for (int i=0; i<3*n; i++) System.out.print(" ");
   System.out.println(s); }
```

Observations about procedure lifetimes

```
begin f
   begin g
      begin h
      end h
   end g
end f
```

Observations about procedure lifetimes

```
begin f
   begin g
      begin h
      end h
   end g
end f

begin g
   begin h
   end h
end g
```

Observations about procedure lifetimes

```
begin f
   begin g
      begin h
      end h
   end g
end f

begin g
   begin h
   end h
end g

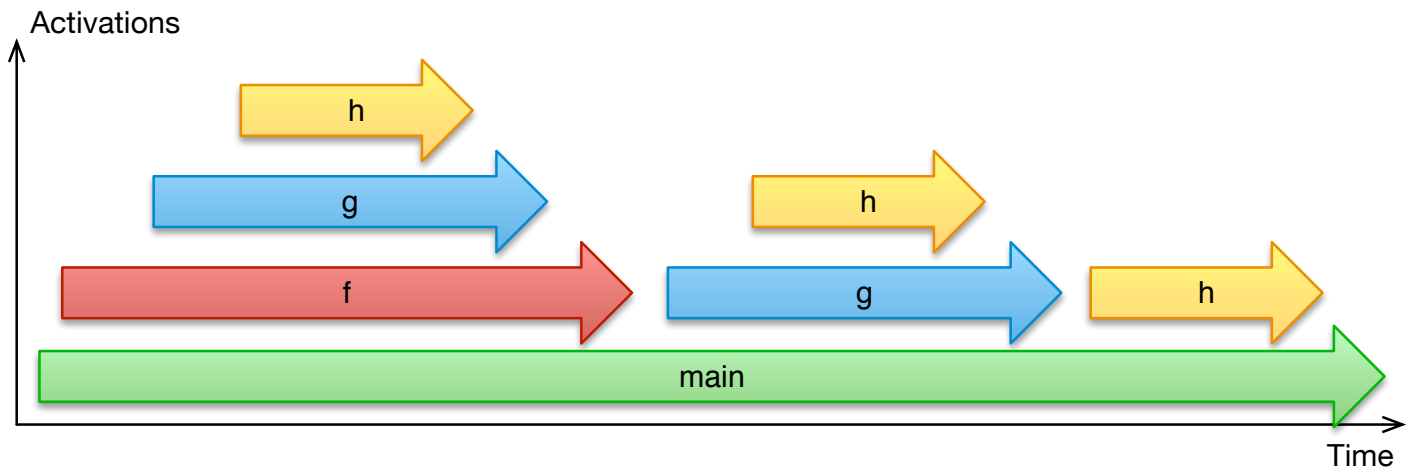
begin h
end h
```

Observations about procedure lifetimes

```
begin f
   begin g
      begin h
      end h
   end g
end f

begin g
   begin h
   end h
end g

begin h
end h
```

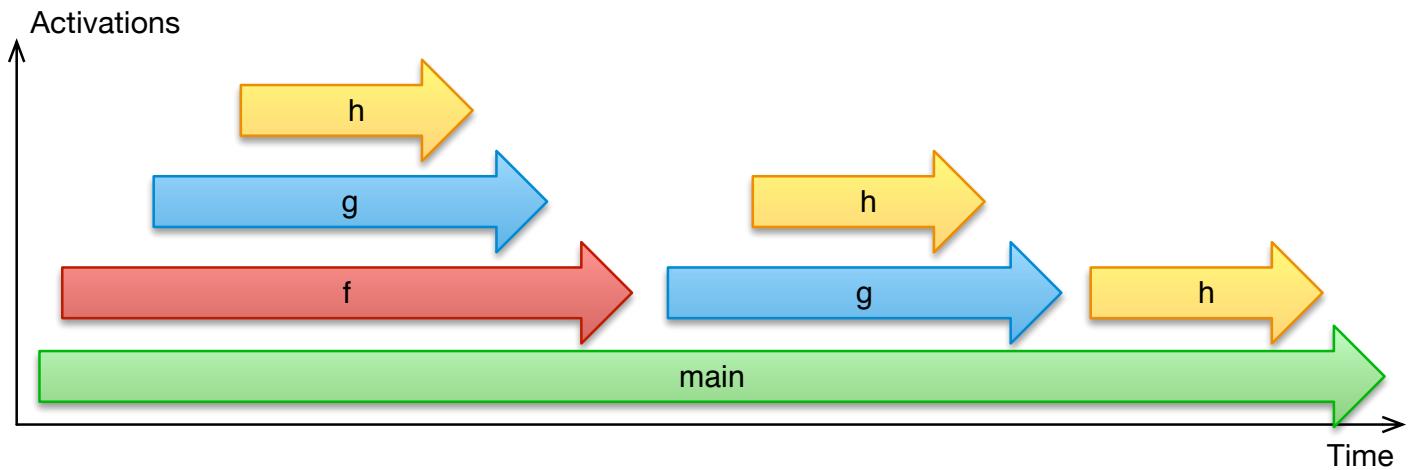


Observations about procedure lifetimes

```
begin f
   begin g
      begin h
      end h
   end g
end f
```

```
begin g
   begin h
   end h
end g
```

```
begin h
end h
```



Lifetimes of procedure activations are properly nested — they correspond to (part of) the syntactic structure of the program.

- Most importantly, there is a unique “stack” of live subroutine calls at all times.

Observations about procedure lifetimes

What about recursive procedures?

Consider a function to compute the factorial of an integer

(with an extra parameter to report recursion depth)

Observations about procedure lifetimes

What about recursive procedures?

Consider a function to compute the factorial of an integer

(with an extra parameter to report recursion depth)

```
public static void main(String[] args){
   int n = 4;
   int result = factorial(4,0);
   System.out.println(n + "! = " + result); }
```

Observations about procedure lifetimes

What about recursive procedures?
Consider a function to compute the factorial of an integer
(with an extra parameter to report recursion depth)

```
public static void main(String[] args){
   int n = 4;
   int result = factorial(4,0);
   System.out.println(n + "! = " + result); }

static int factorial (int n, int d) {
   int result = n;
   report("begin", n, d);
   if (n > 1) result *= factorial(n-1,d+1);
   report("end", n, d);
   return result; }
```

Observations about procedure lifetimes

What about recursive procedures?
Consider a function to compute the factorial of an integer
(with an extra parameter to report recursion depth)

```
public static void main(String[] args){
   int n = 4;
   int result = factorial(4,0);
   System.out.println(n + "! = " + result); }

static int factorial (int n, int d) {
   int result = n;
   report("begin", n, d);
   if (n > 1) result *= factorial(n-1,d+1);
   report("end", n, d);
   return result; }

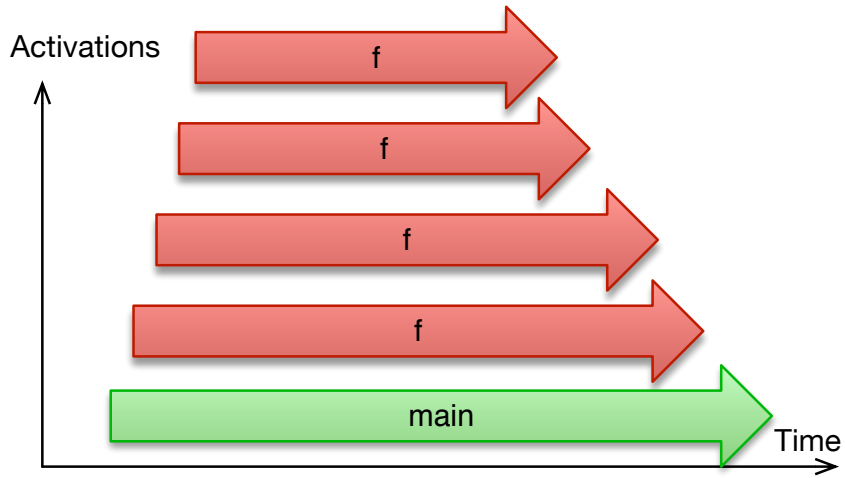
static void report (String s, int n, int d) {
   for (int i=0; i < 3*d; i++) System.out.print(" ");
   System.out.println(s + " factorial(" + n + "," + d + ")" ); }
```

Observations about procedure lifetimes

```
begin factorial(4,0)
   begin factorial(3,1)
      begin factorial(2,2)
         begin factorial(1,3)
         end factorial(1,3)
      end factorial(2,2)
   end factorial(3,1)
end factorial(4,0)
4! = 24
```

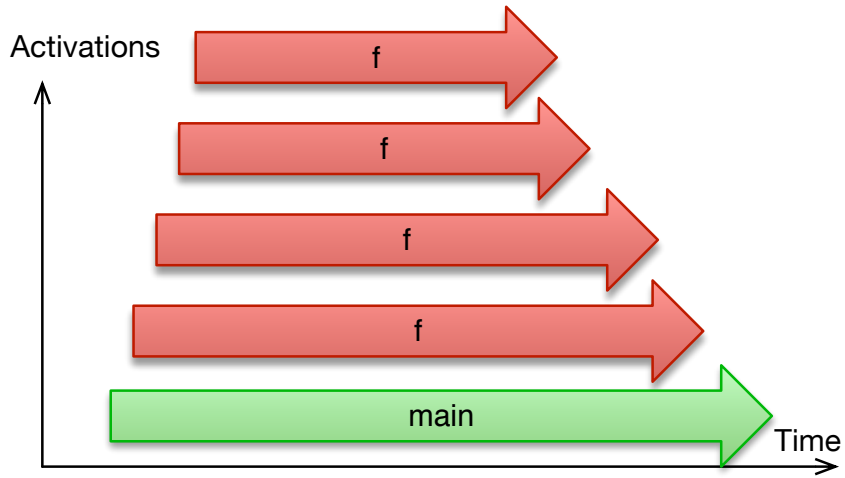
Observations about procedure lifetimes

```
begin factorial(4,0)
   begin factorial(3,1)
      begin factorial(2,2)
         begin factorial(1,3)
         end factorial(1,3)
      end factorial(2,2)
   end factorial(3,1)
end factorial(4,0)
4! = 24
```



Observations about procedure lifetimes

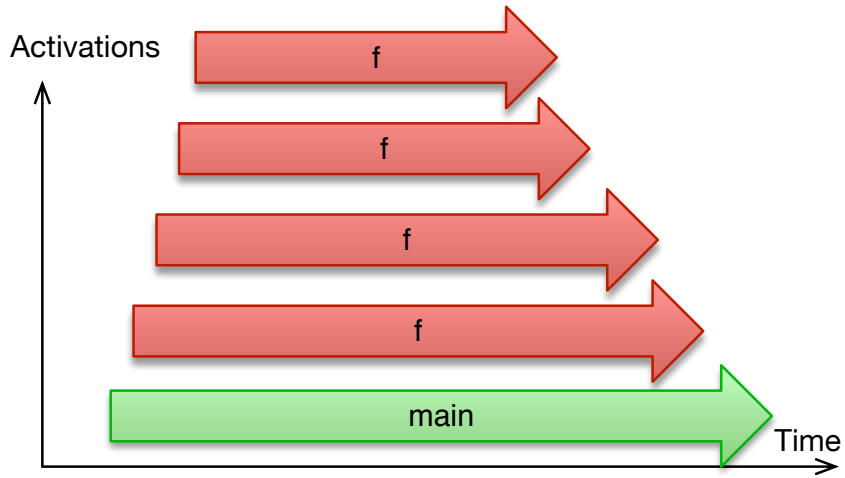
```
begin factorial(4,0)
   begin factorial(3,1)
      begin factorial(2,2)
         begin factorial(1,3)
         end factorial(1,3)
      end factorial(2,2)
   end factorial(3,1)
end factorial(4,0)
4! = 24
```



- Again, a *stack* of live subroutine calls

Observations about procedure lifetimes

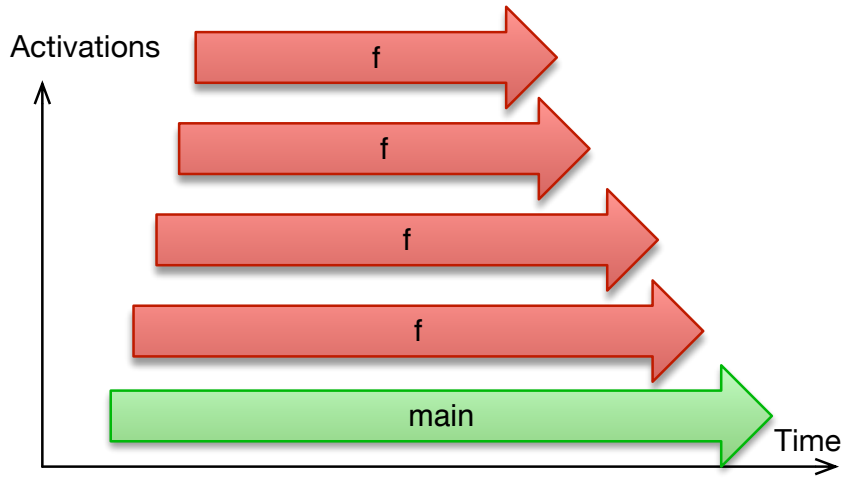
```
begin factorial(4,0)
   begin factorial(3,1)
      begin factorial(2,2)
         begin factorial(1,3)
         end factorial(1,3)
      end factorial(2,2)
   end factorial(3,1)
end factorial(4,0)
4! = 24
```



- Again, a *stack* of live subroutine calls
- More than one activation *of the same procedure* might be active at the same time

Observations about procedure lifetimes

```
begin factorial(4,0)
   begin factorial(3,1)
      begin factorial(2,2)
         begin factorial(1,3)
         end factorial(1,3)
      end factorial(2,2)
   end factorial(3,1)
end factorial(4,0)
4! = 24
```



- Again, a *stack* of live subroutine calls
- More than one activation *of the same procedure* might be active at the same time
- How many? There is no obvious pre-determinable limit (e.g., it may depend on user input)

Activation records

We need to store / pass along some data to orchestrate execution of a procedure call — *e.g.*,

- procedure arguments
- local variables
- return value, and
- return address

  (location of next instruction after subroutine call)

Activation records

We need to store / pass along some data to orchestrate execution of a procedure call — *e.g.*,

- procedure arguments
- local variables
- return value, and
- return address
  (location of next instruction after subroutine call)

We call this information the **activation record**.

Activation records

We need to store / pass along some data to orchestrate execution of a procedure call — *e.g.*,

- procedure arguments
- local variables
- return value, and
- return address

  (location of next instruction after subroutine call)

We call this information the **activation record**.

Our observations above suggests maintaining a **stack** of *currently live* activations, in increasing order of recursive depth.

Activation records

We need to store / pass along some data to orchestrate execution of a procedure call — *e.g.*,

- procedure arguments
- local variables
- return value, and
- return address

  (location of next instruction after subroutine call)

We call this information the **activation record**.

Our observations above suggests maintaining a **stack** of *currently live* activations, in increasing order of recursive depth.

- On calling a procedure:
  - push an activation record on the stack with relevant data

Activation records

We need to store / pass along some data to orchestrate execution of a procedure call — *e.g.*,

- procedure arguments
- local variables
- return value, and
- return address

  (location of next instruction after subroutine call)

We call this information the **activation record**.

Our observations above suggests maintaining a **stack** of *currently live* activations, in increasing order of recursive depth.

- On calling a procedure:
  - push an activation record on the stack with relevant data
- On procedure termination:
  - put return value in an easy-to-find location
  - pop activation record off the stack (i.e. discard it).

Activation records

We need to store / pass along some data to orchestrate execution of a procedure call — *e.g.*,

- procedure arguments
- local variables
- return value, and
- return address

  (location of next instruction after subroutine call)

We call this information the **activation record**.

Our observations above suggests maintaining a **stack** of *currently live* activations, in increasing order of recursive depth.

- On calling a procedure:
  - push an activation record on the stack with relevant data
- On procedure termination:
  - put return value in an easy-to-find location
  - pop activation record off the stack (i.e. discard it).

We sometimes call an activation record, a **stack frame**.

Recap: Why do we need activation records?

The computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers

Recap: Why do we need activation records?

The computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers

What memory addresses does the computer “know about”?

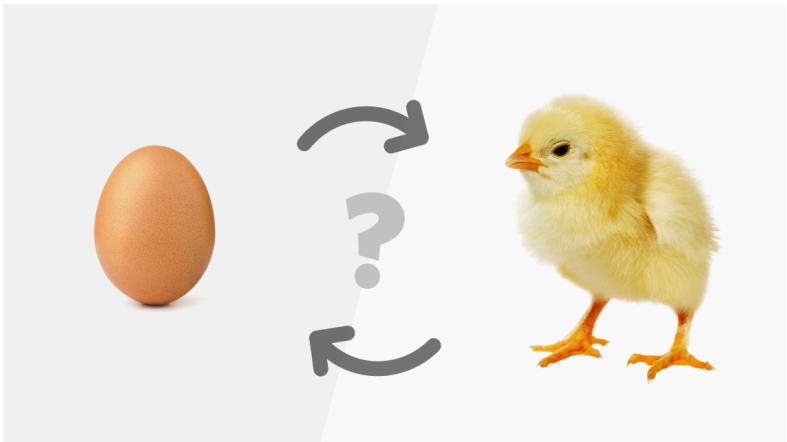
Recap: Why do we need activation records?

The computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers

What memory addresses does the computer “know about”?

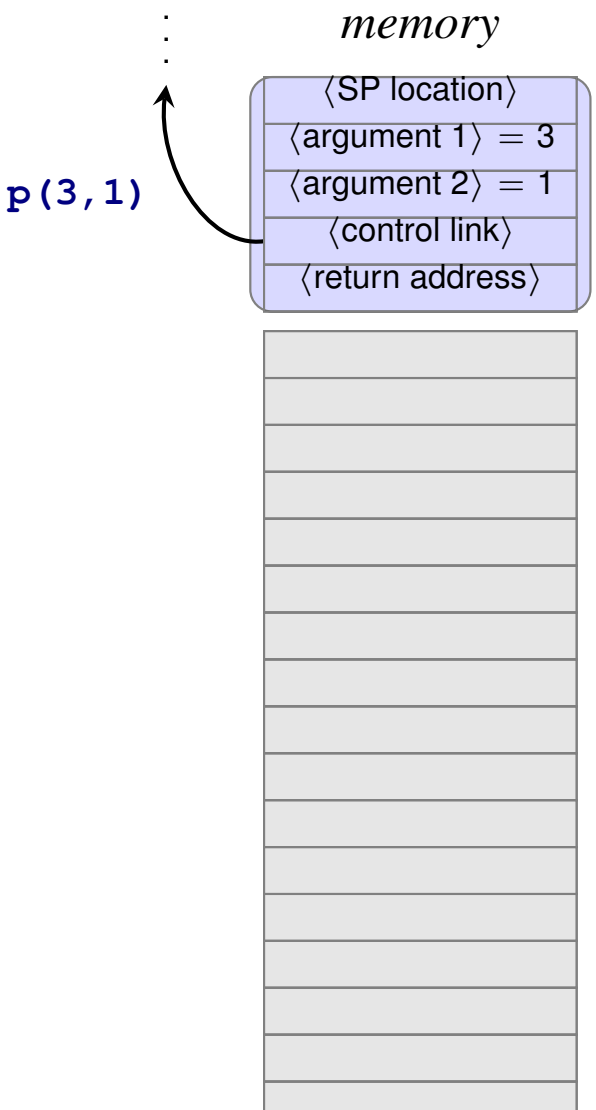
— apart from **static** addresses (known at compile time):
only those stored somewhere the computer knows about!



Is there any place in memory that can we assume that the computer **definitely does** “know about”?

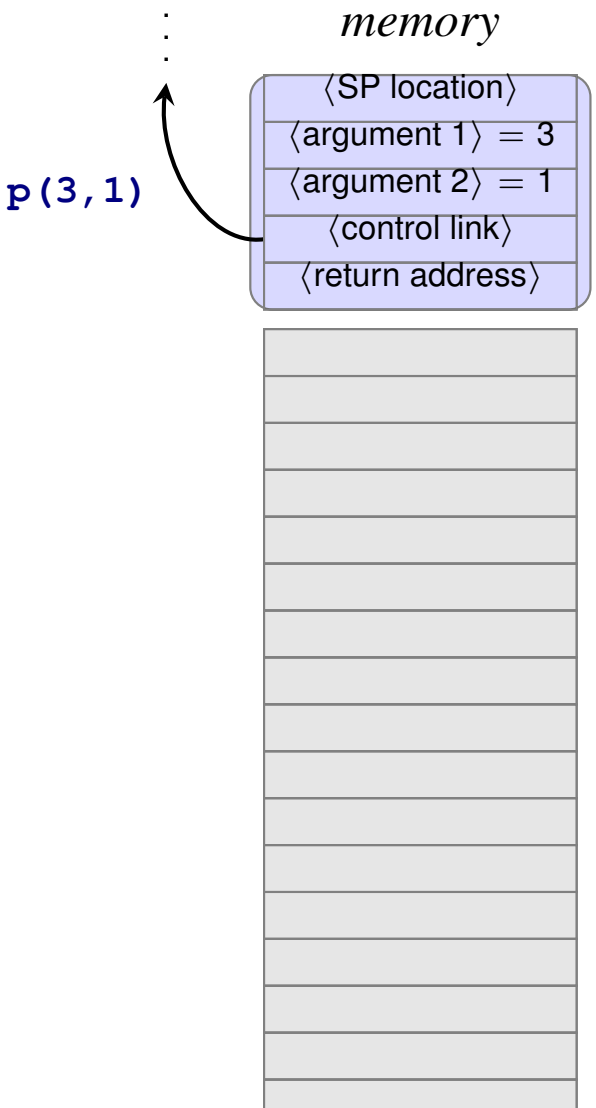
Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.



Pointers to manage activation records

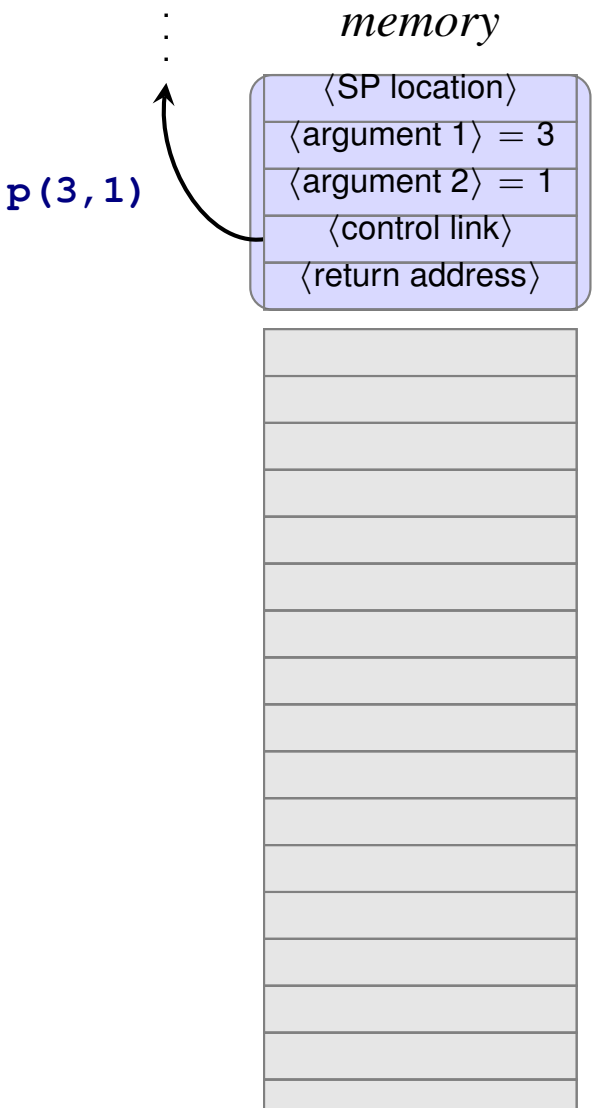
To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.



(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

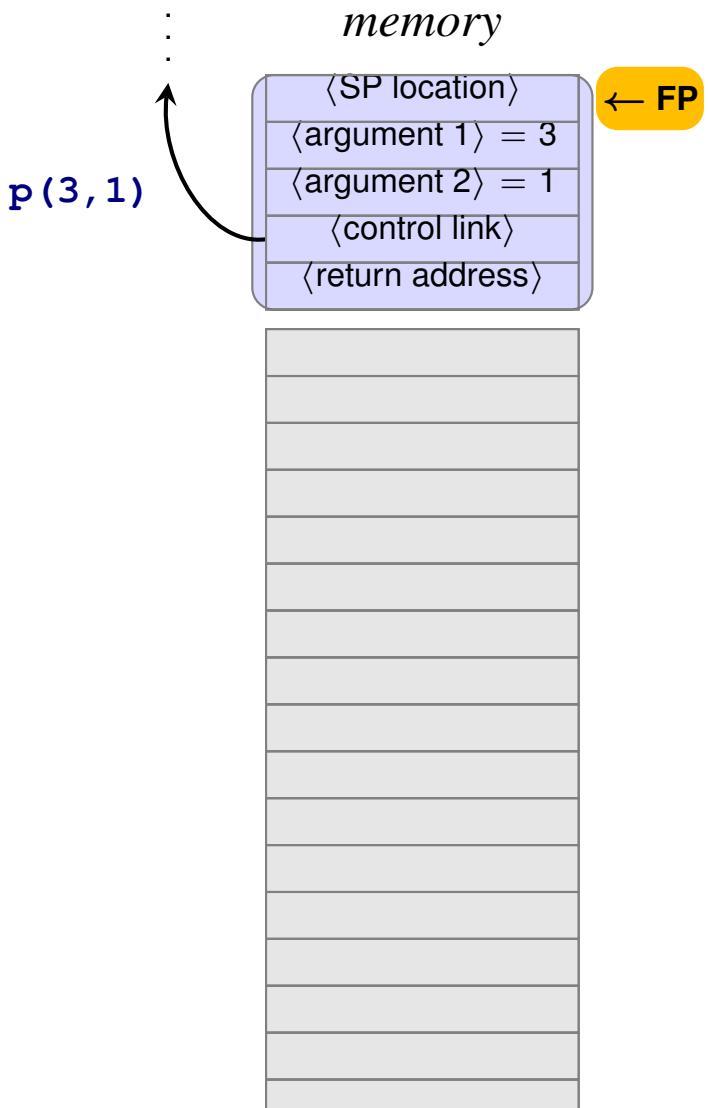


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

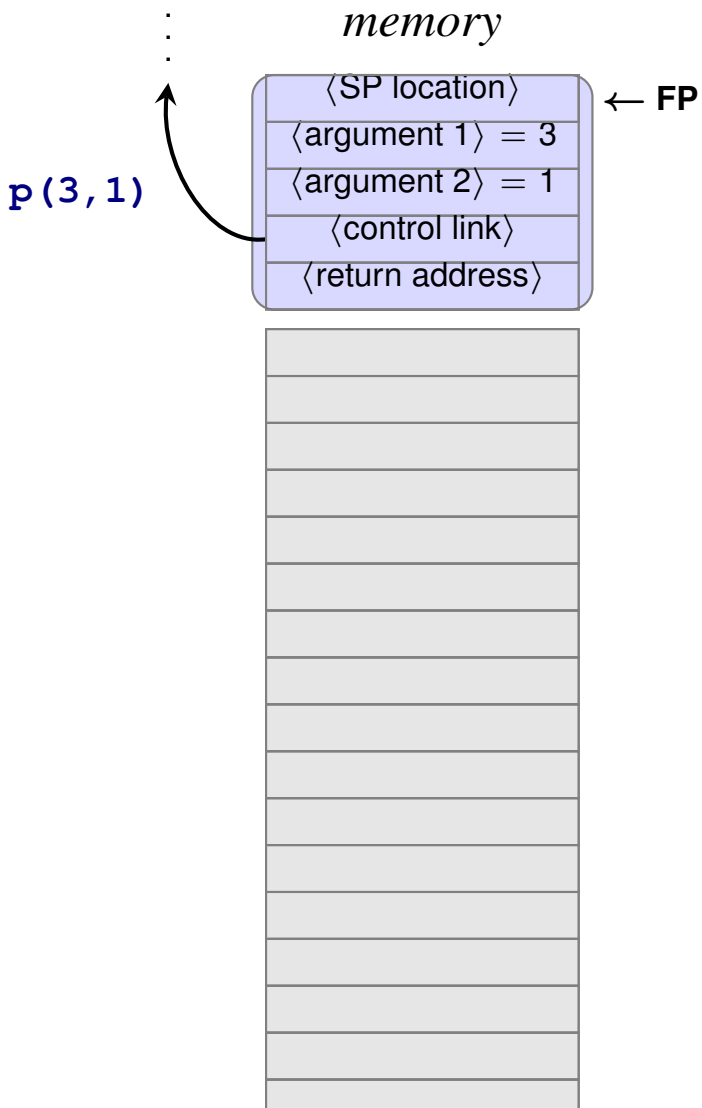


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

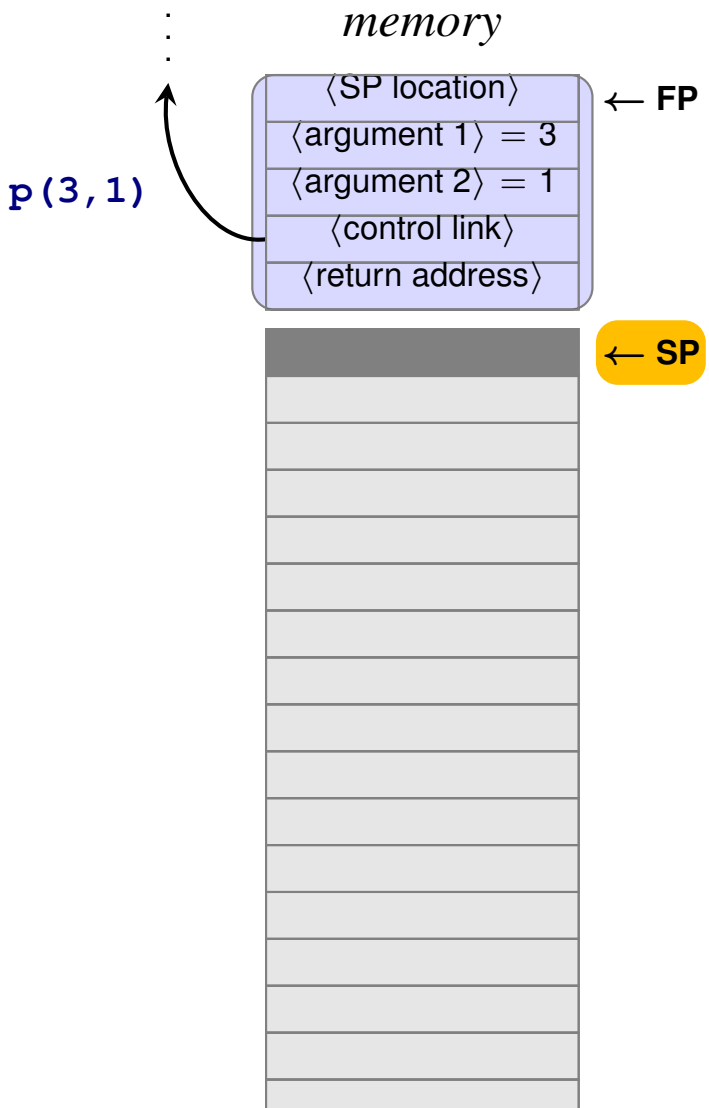


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

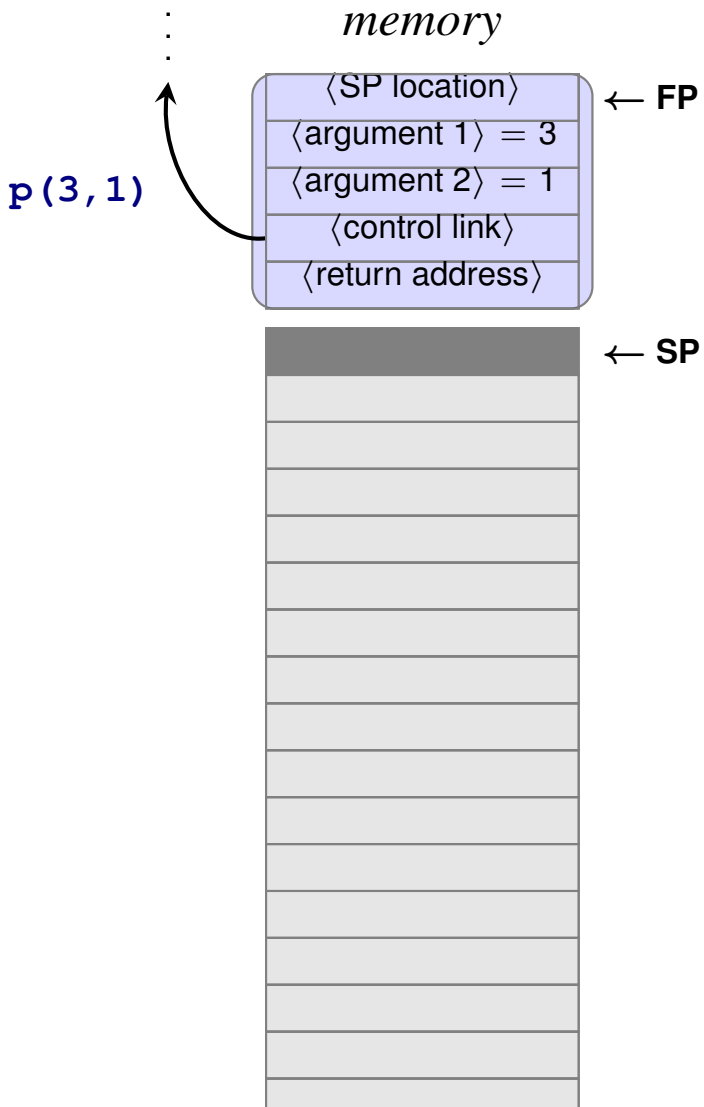


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

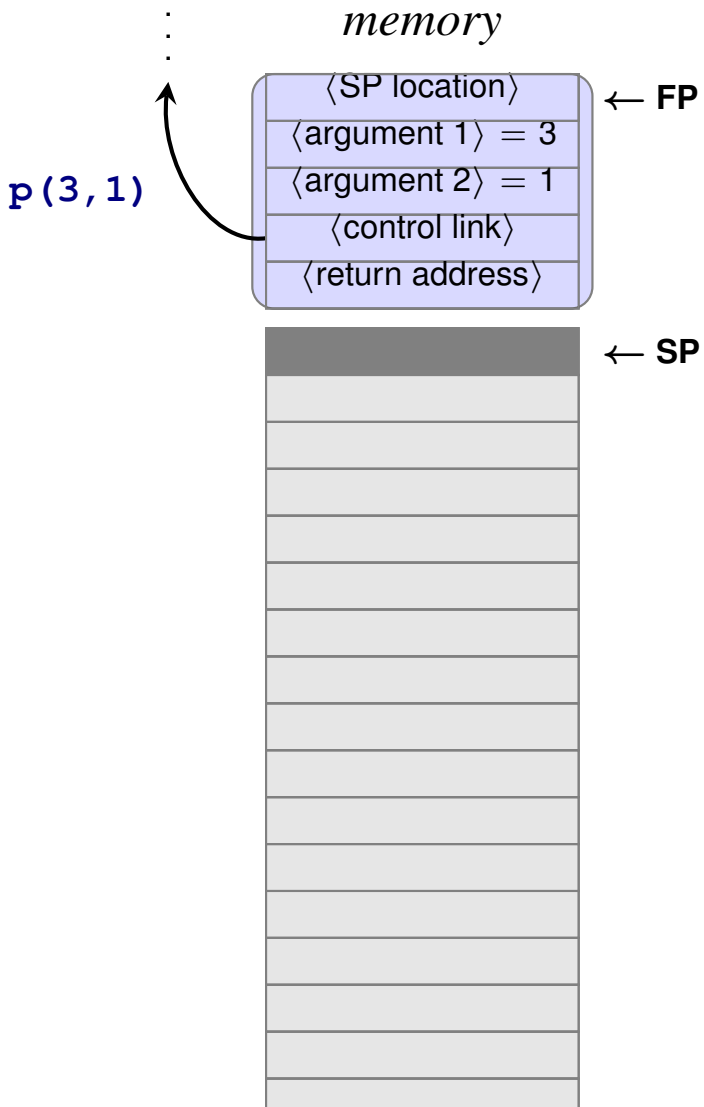


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

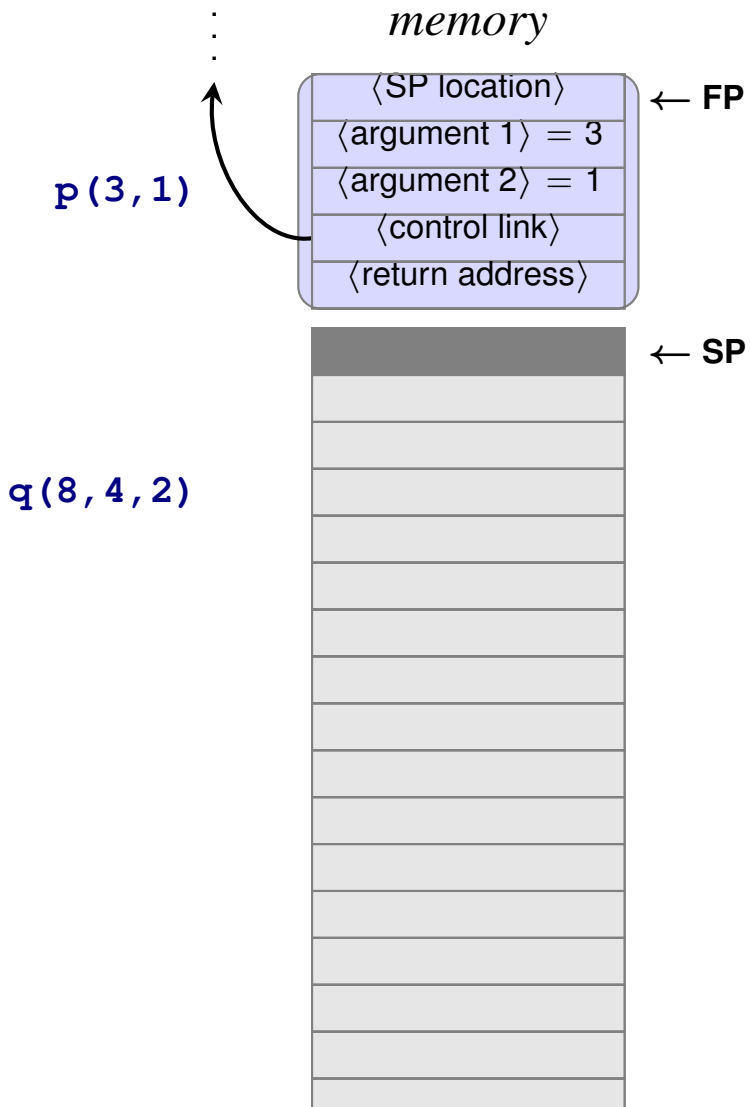


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

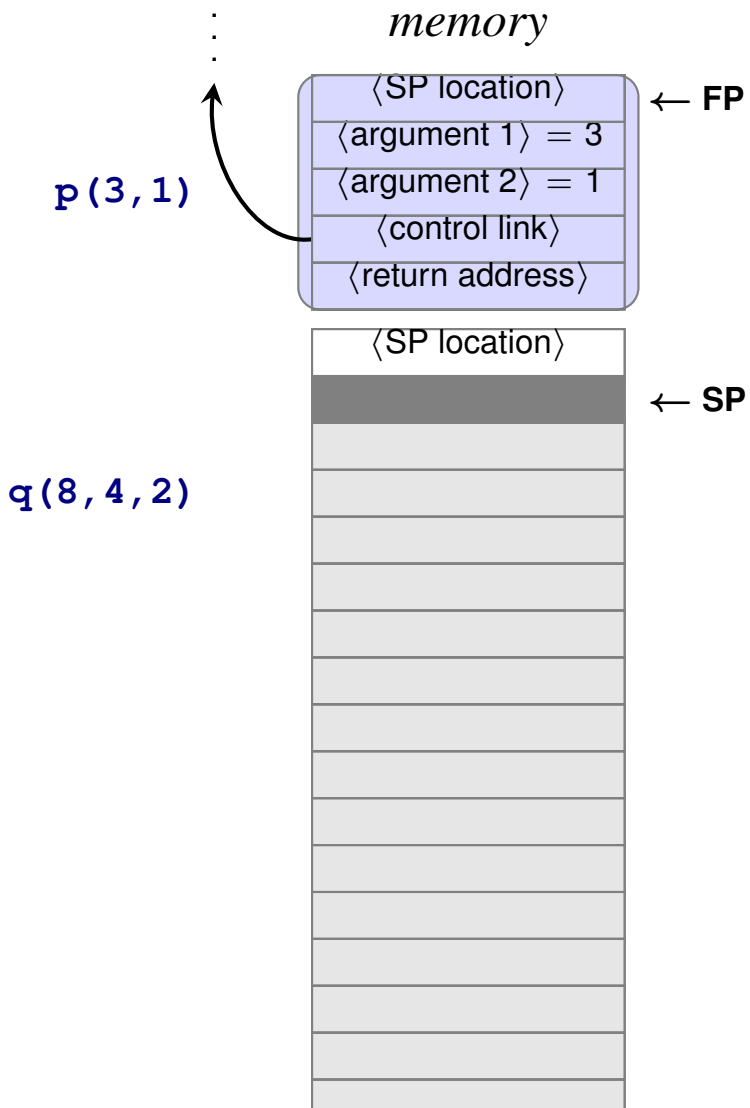


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

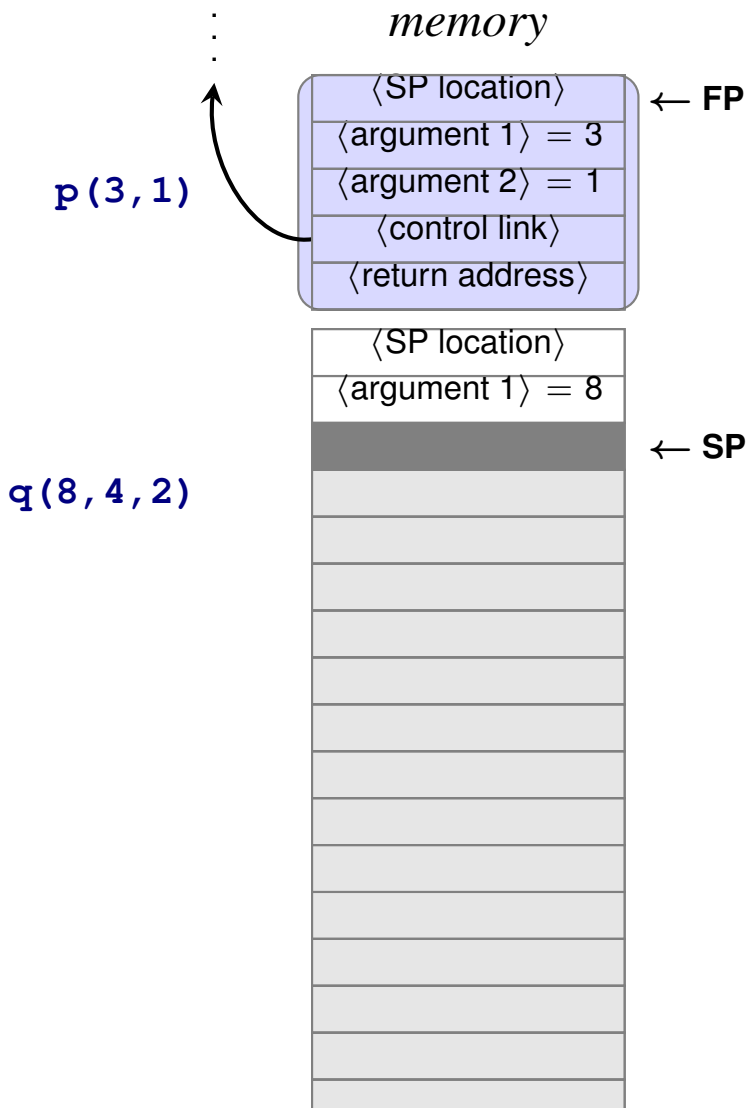


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

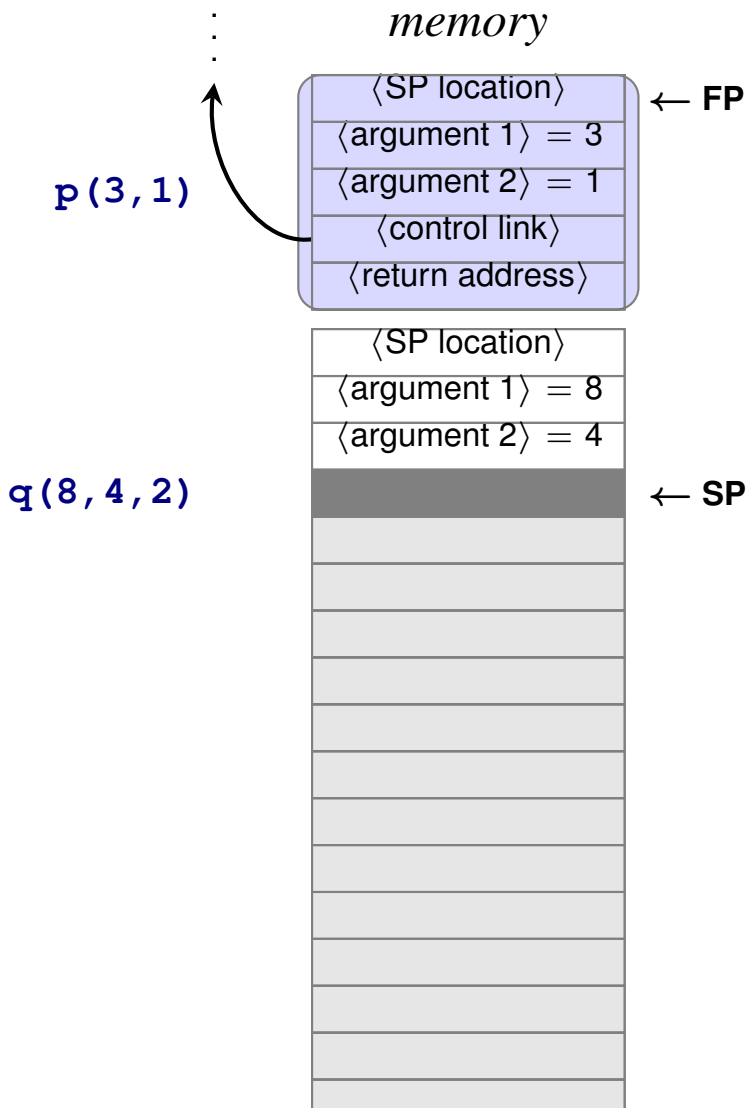


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

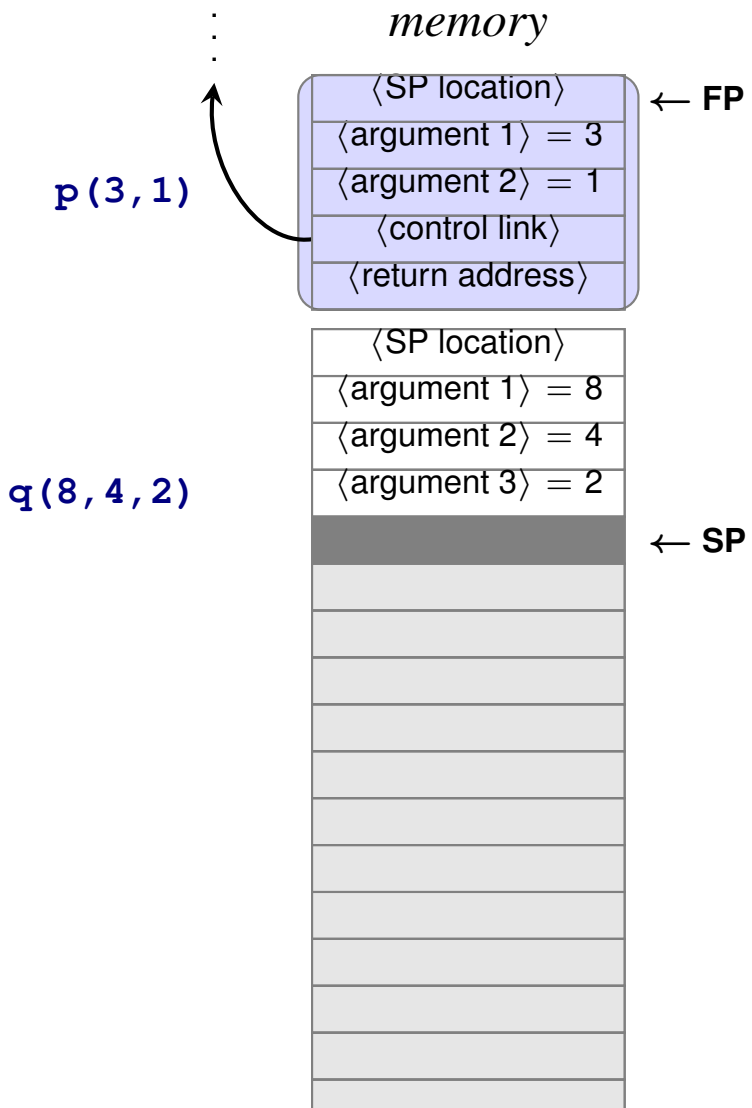


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

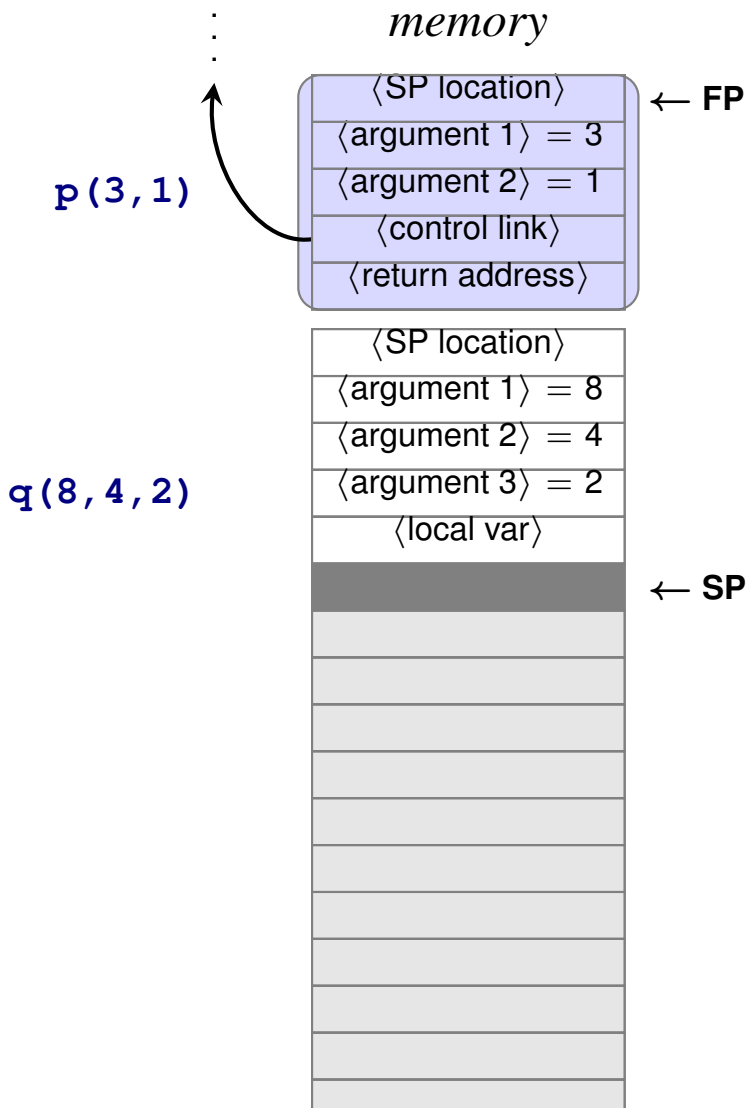


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

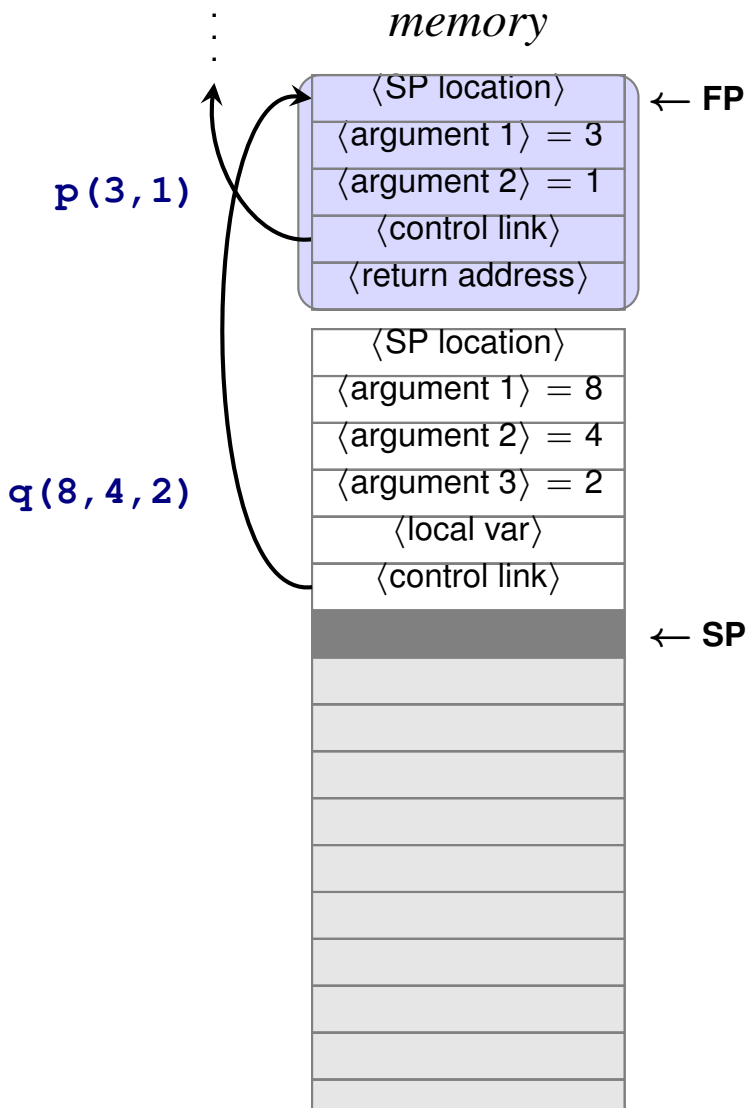


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

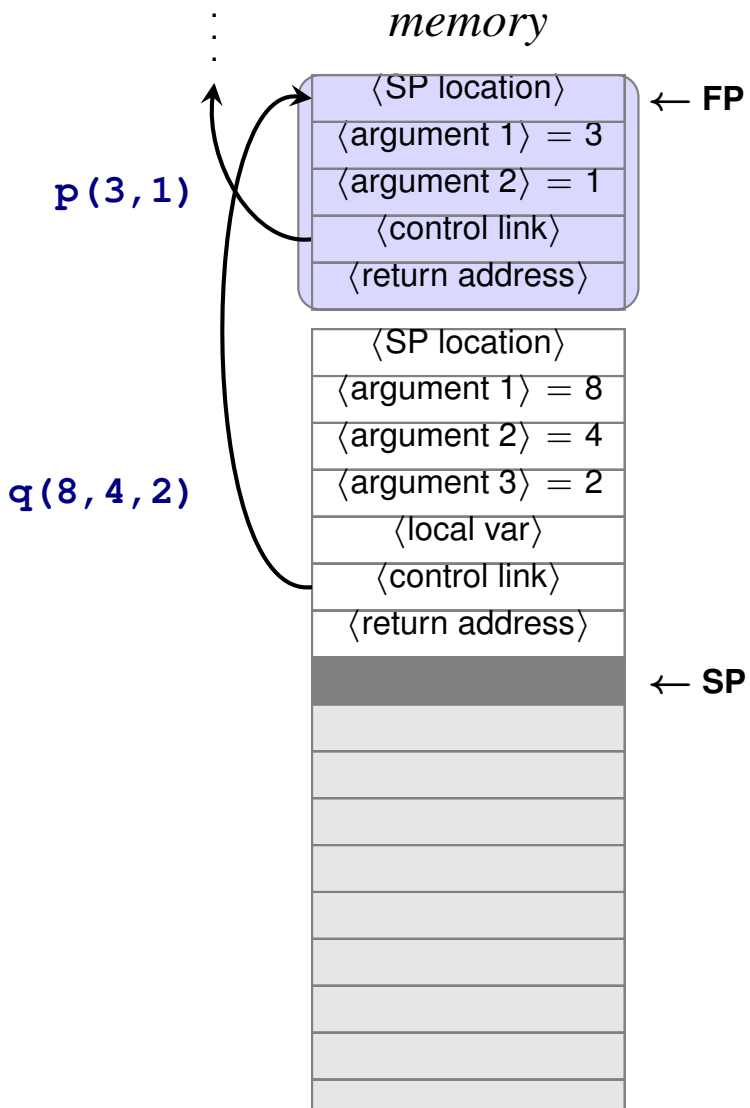


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

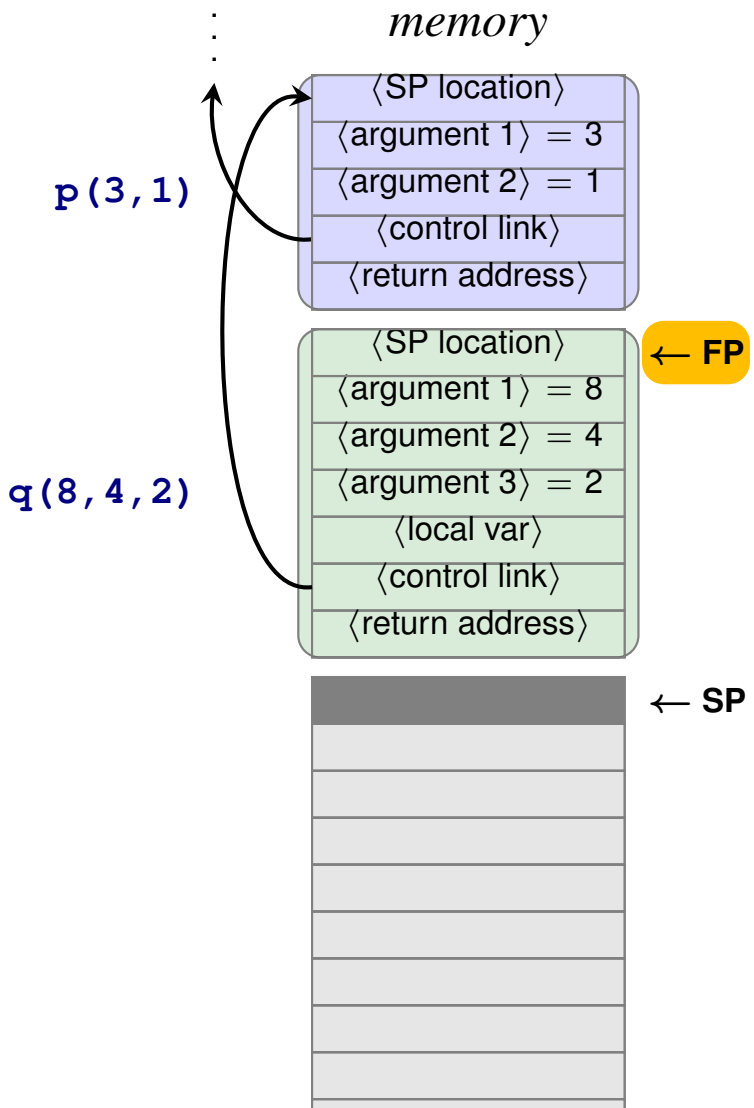


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

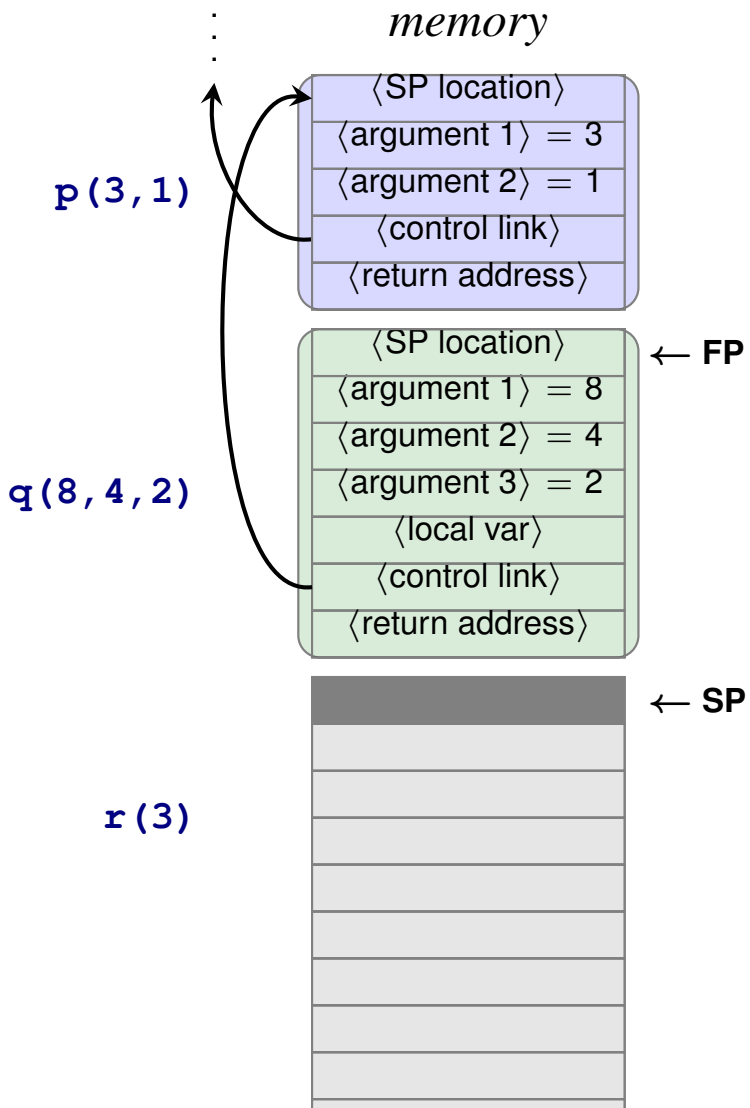


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

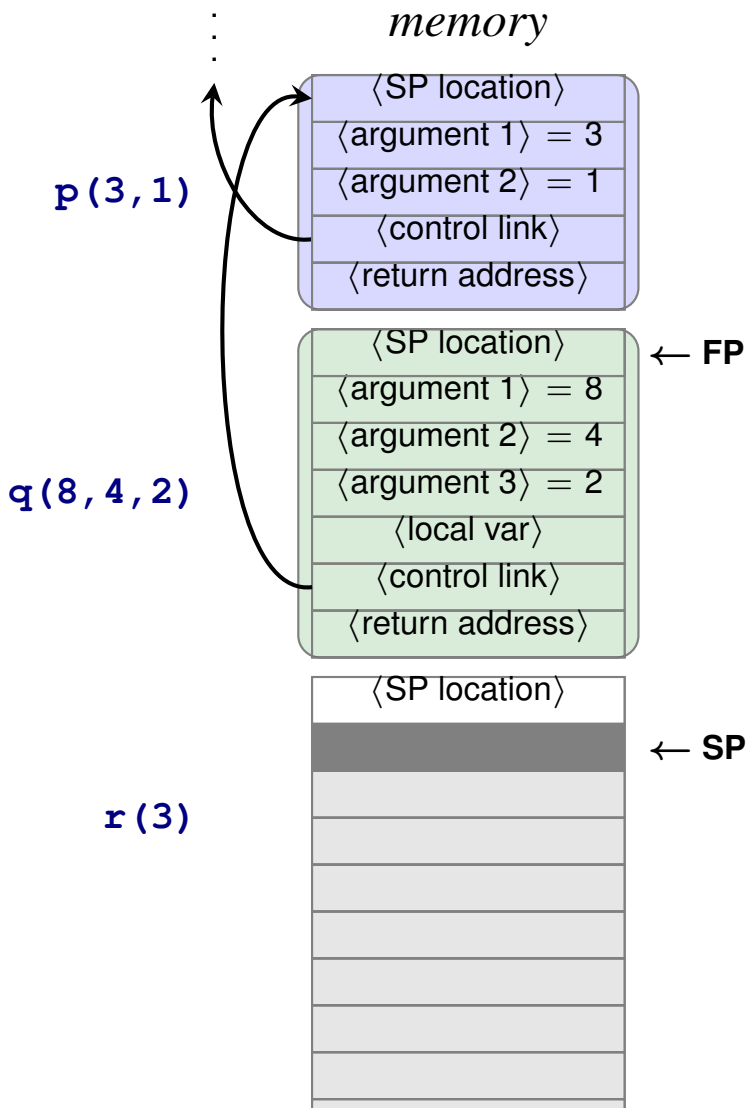


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

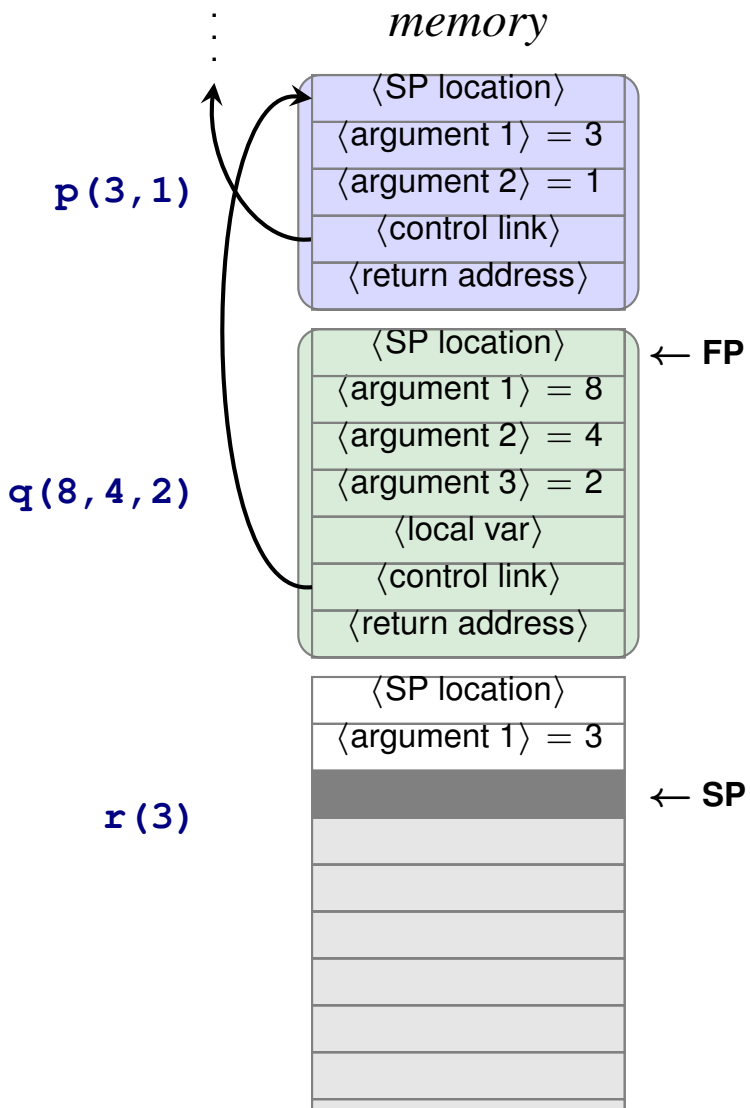


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

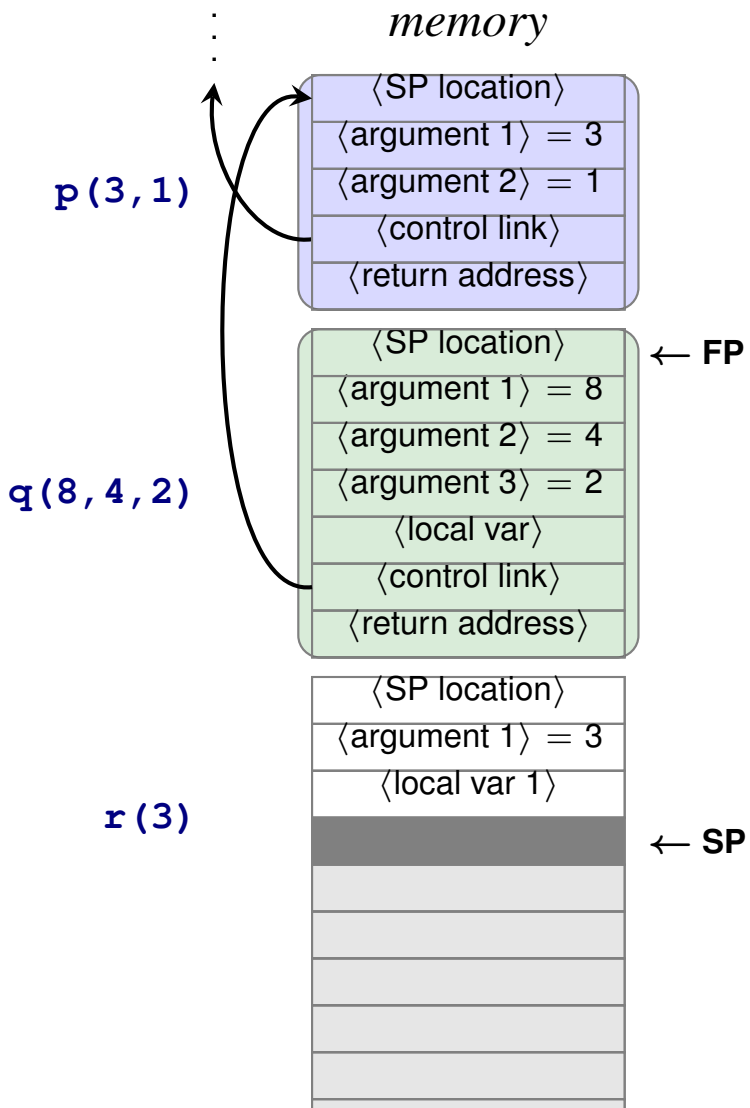


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

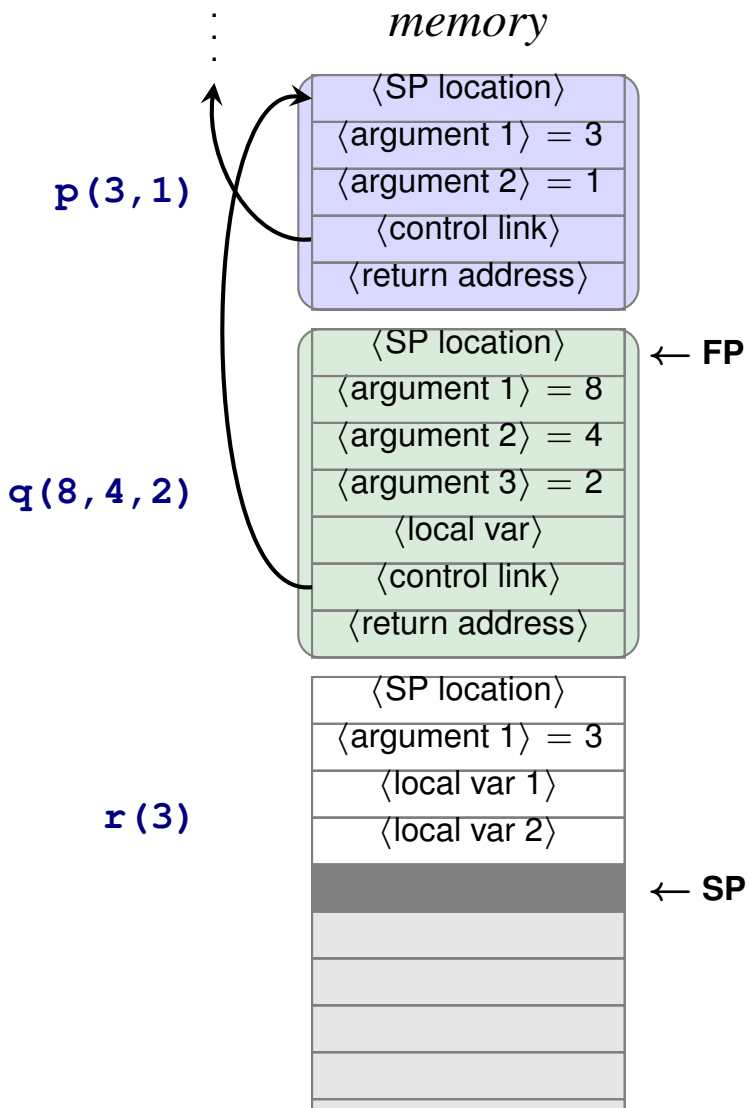


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

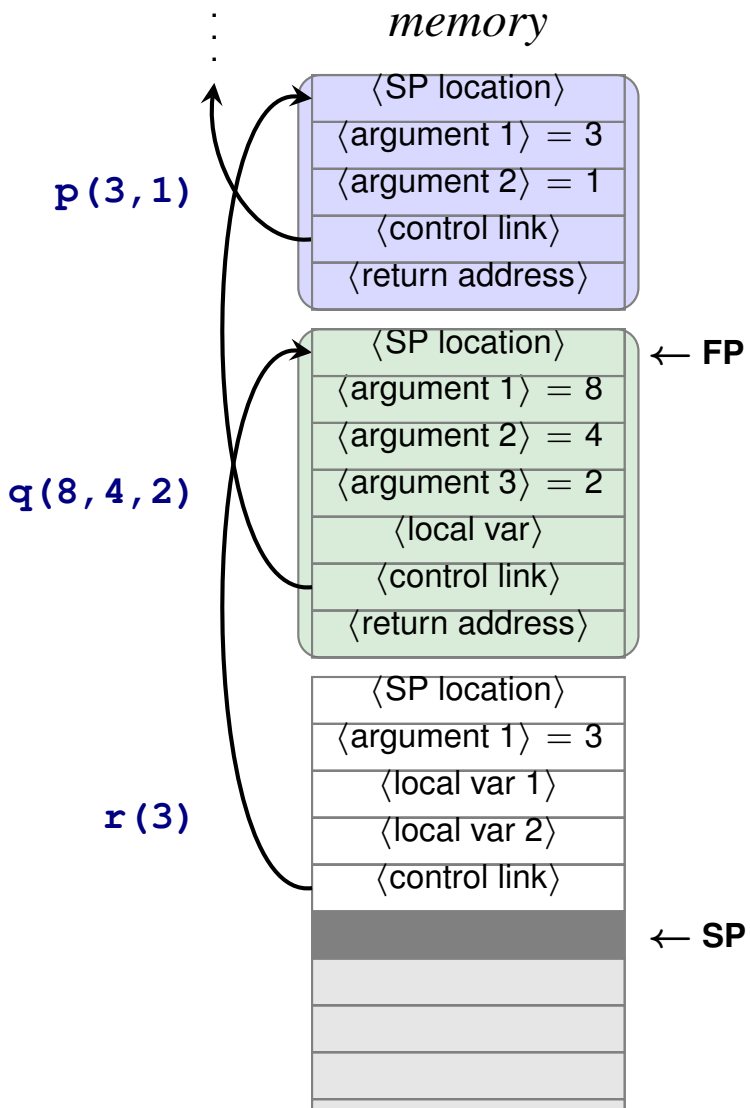


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

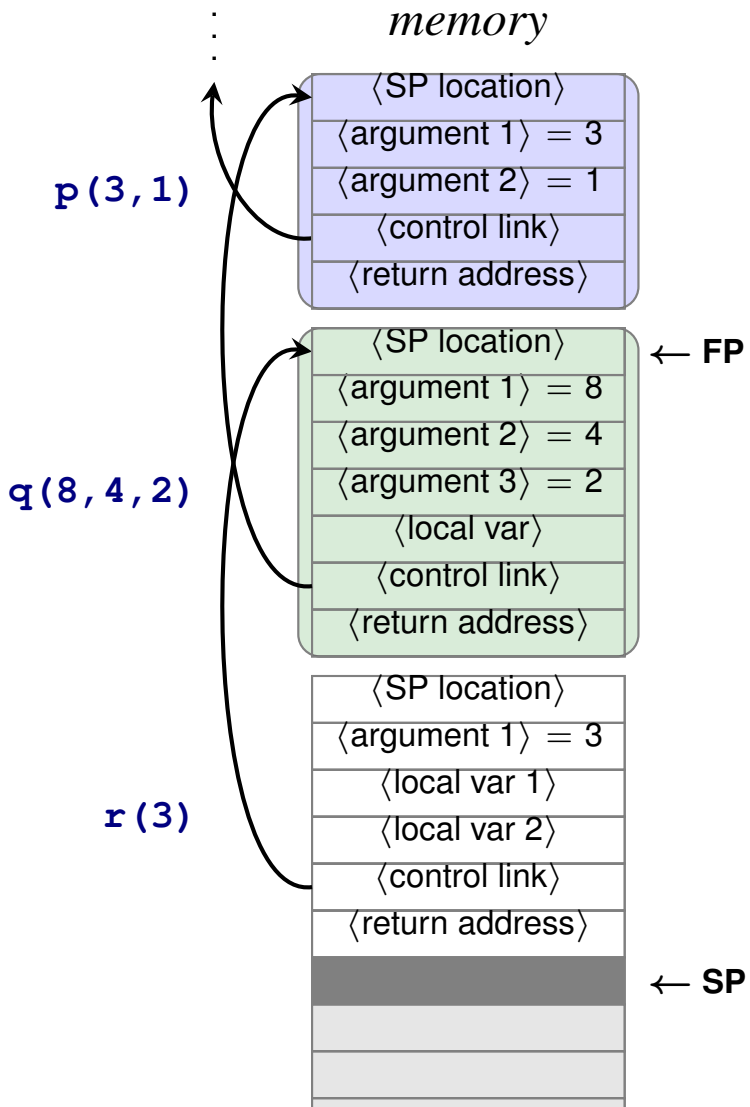


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

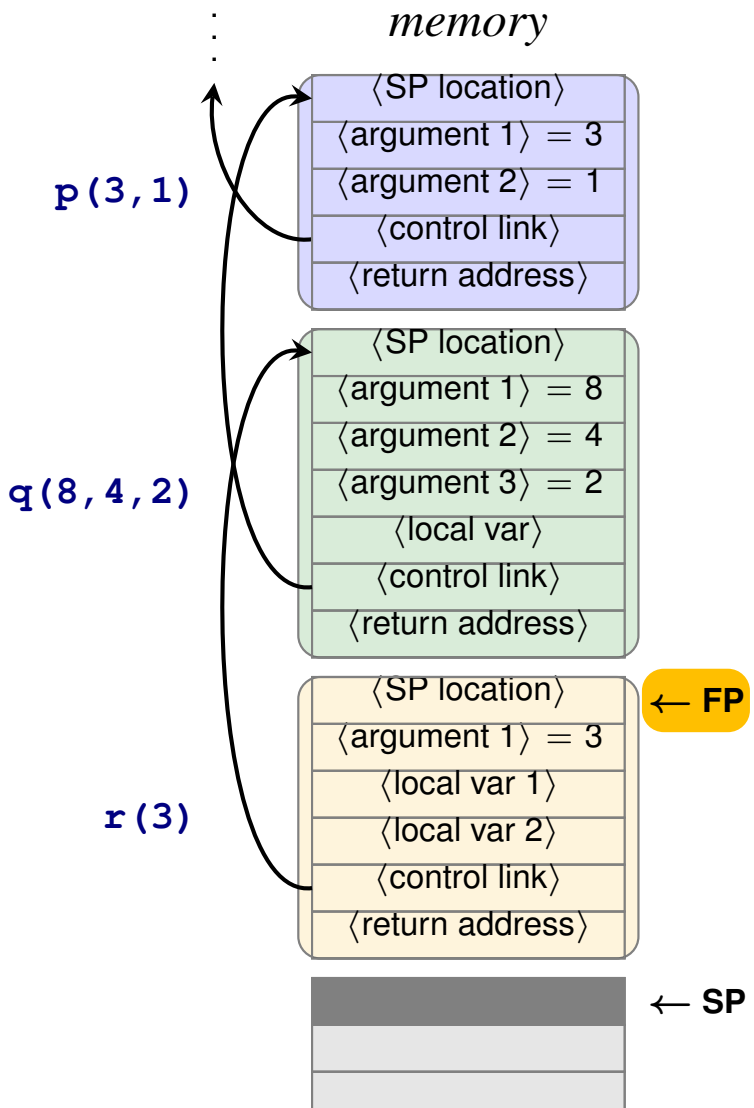


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

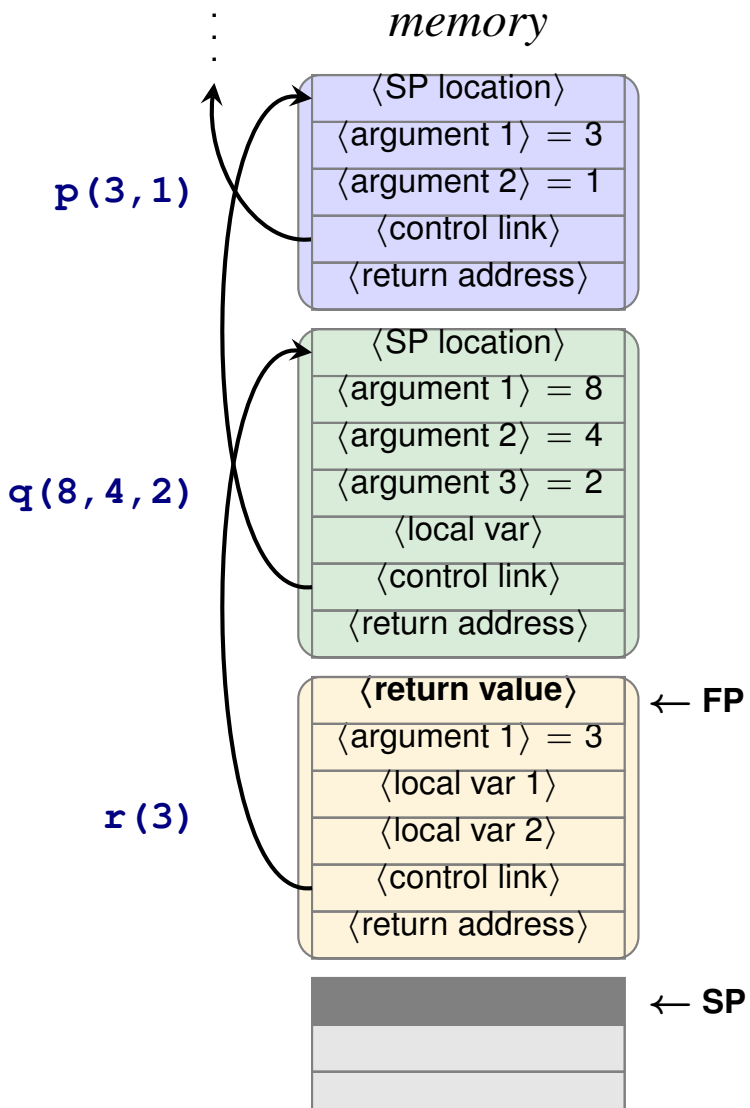


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

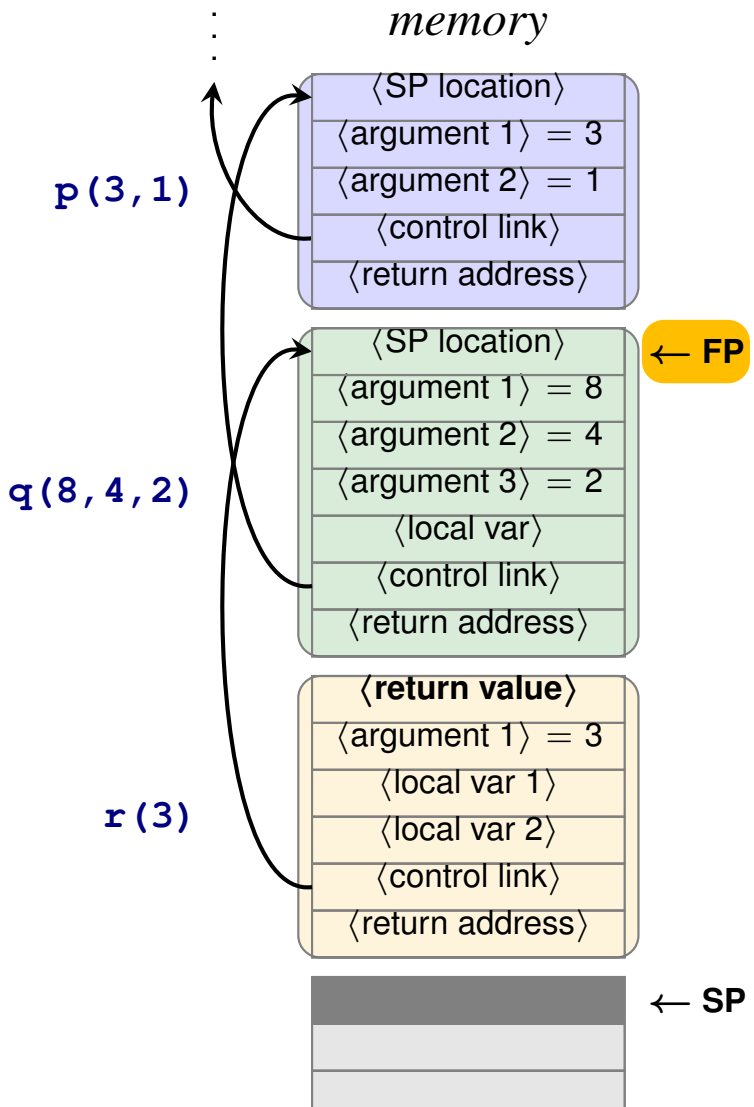


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.

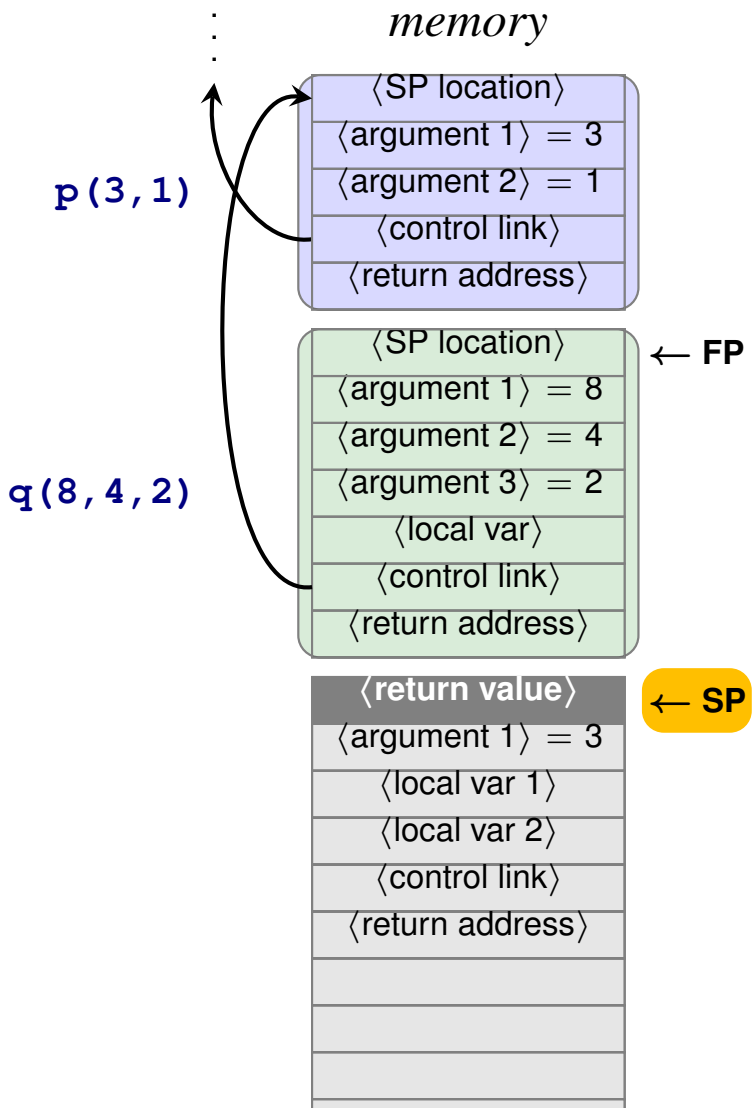


(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Pointers to manage activation records

To solve the issue of being able to keep track of information in memory (particularly for subroutine calls), processors usually have dedicated **frame pointer** and **stack pointer** registers.



(**N.B.** addresses count upwards: we often grow stacks downwards from higher addresses)

- **frame pointer (FP):** points to a fixed location (*e.g.*, the beginning) of the activation record
- **stack pointer (SP):** points to next free address in front of the stack
- Local arguments / variables can be easily found relative to these locations in memory
- We manage subroutine calls by storing/retrieving data on the stack

Design of activation records / stack frames

Question. What information do we keep in an activation record?

Design of activation records / stack frames

Question. What information do we keep in an activation record?

- Depends on the details of the programming language, compilation strategy, and target architecture. . .

Design of activation records / stack frames

Question. What information do we keep in an activation record?

- Depends on the details of the programming language, compilation strategy, and target architecture… trade-offs:
  - number of instructions required to perform a subroutine call

Design of activation records / stack frames

Question. What information do we keep in an activation record?

- Depends on the details of the programming language, compilation strategy, and target architecture… trade-offs:
  - number of instructions required to perform a subroutine call
  - amount of data saved on the stack
    (*e.g.*, old register values, exception handler address, …)

Design of activation records / stack frames

Question. What information do we keep in an activation record?

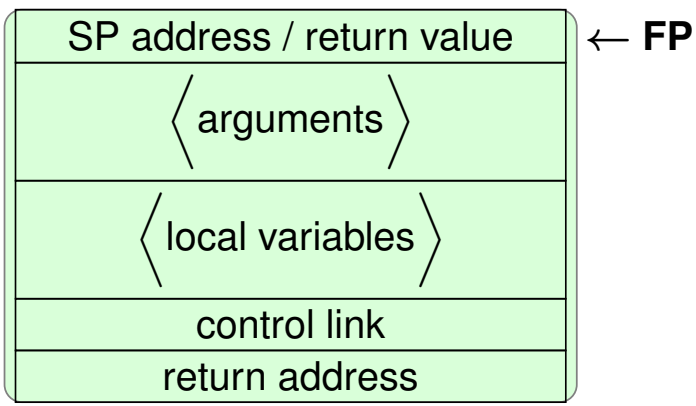
- Depends on the details of the programming language, compilation strategy, and target architecture… trade-offs:
  - number of instructions required to perform a subroutine call
  - amount of data saved on the stack
    (*e.g.*, old register values, exception handler address, …)
- But, it usually includes: the subroutine arguments, local variables, and the return address.

Design of activation records / stack frames

Question. What information do we keep in an activation record?

- Depends on the details of the programming language, compilation strategy, and target architecture… trade-offs:
  - number of instructions required to perform a subroutine call
  - amount of data saved on the stack
    (*e.g.*, old register values, exception handler address, …)
- But, it usually includes: the subroutine arguments, local variables, and the return address.

Structure used above:

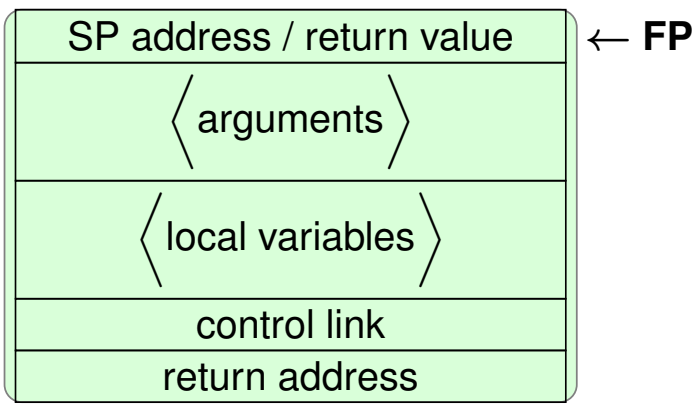


Design of activation records / stack frames

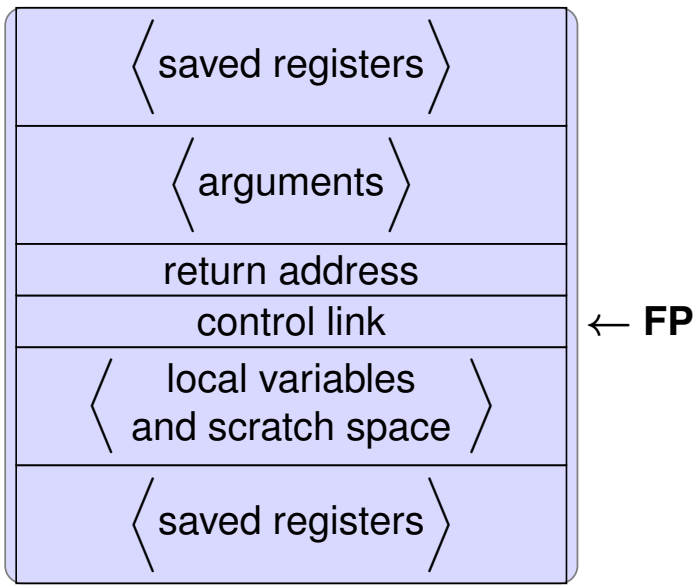
Question. What information do we keep in an activation record?

- Depends on the details of the programming language, compilation strategy, and target architecture… trade-offs:
  - number of instructions required to perform a subroutine call
  - amount of data saved on the stack
    (*e.g.*, old register values, exception handler address, …)
- But, it usually includes: the subroutine arguments, local variables, and the return address.

Structure used above:



…used by C, on x86-32:

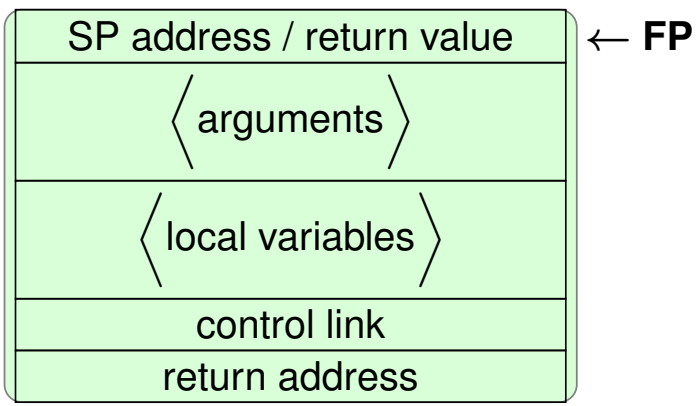


Design of activation records / stack frames

Question. What information do we keep in an activation record?

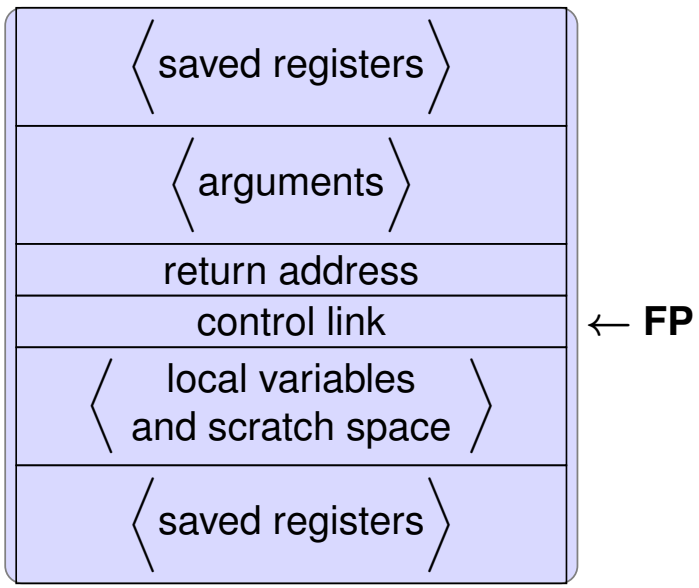
- Depends on the details of the programming language, compilation strategy, and target architecture. . . trade-offs:
  - number of instructions required to perform a subroutine call
  - amount of data saved on the stack
    (*e.g.*, old register values, exception handler address, . . .)
- But, it usually includes: the subroutine arguments, local variables, and the return address.

Structure used above:



- Size can be determined at compile-time.

. . . used by C, on x86-32:



Stacks as arrays

The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

Stacks as arrays

The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

Stacks as arrays

The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

- Usually we think of "stacks" as a data structure, with access only to the front
- With assembly, we operate at such a low level that there is no reason we cannot read **any** element of the stack.

Stacks as arrays

The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

- Usually we think of "stacks" as a data structure, with access only to the front
- With assembly, we operate at such a low level that there is no reason we cannot read **any** element of the stack.

The "stack" is essentially just one very large array!

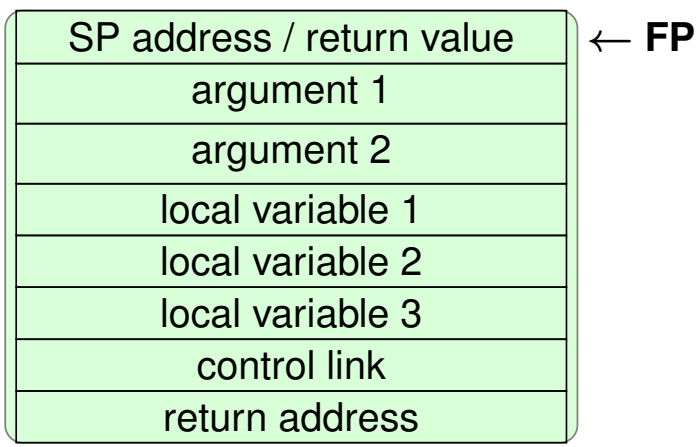
Stacks as arrays

The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

- Usually we think of "stacks" as a data structure, with access only to the front
- With assembly, we operate at such a low level that there is no reason we cannot read **any** element of the stack.

The "stack" is essentially just one very large array!



Stacks as arrays

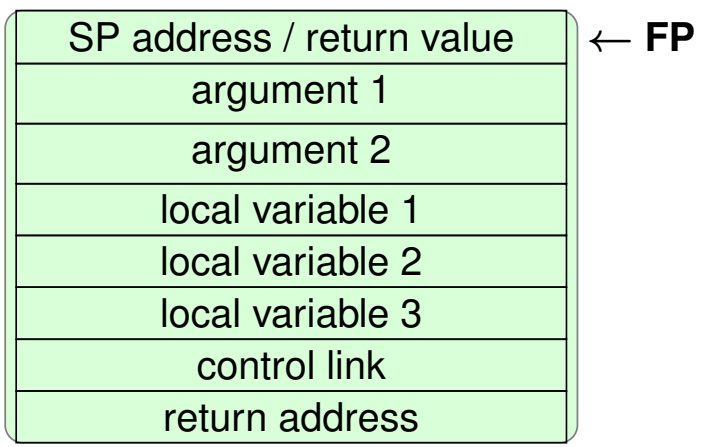
The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

- Usually we think of "stacks" as a data structure, with access only to the front
- With assembly, we operate at such a low level that there is no reason we cannot read **any** element of the stack.

The "stack" is essentially just one very large array!

- Supposing a 4-byte 'word' for each item on the stack:



Stacks as arrays

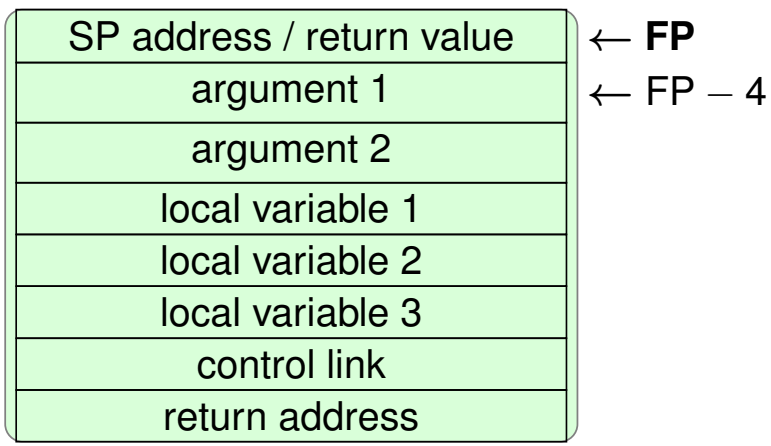
The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

- Usually we think of "stacks" as a data structure, with access only to the front
- With assembly, we operate at such a low level that there is no reason we cannot read **any** element of the stack.

The "stack" is essentially just one very large array!

- Supposing a 4-byte 'word' for each item on the stack:



Stacks as arrays

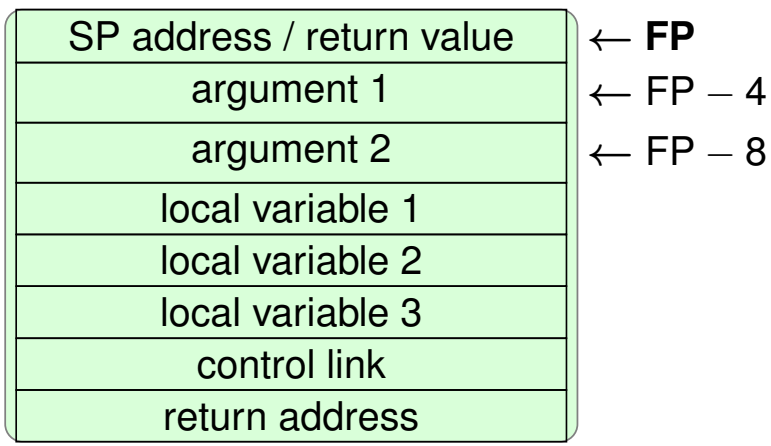
The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

- Usually we think of "stacks" as a data structure, with access only to the front
- With assembly, we operate at such a low level that there is no reason we cannot read **any** element of the stack.

The "stack" is essentially just one very large array!

- Supposing a 4-byte 'word' for each item on the stack:



Stacks as arrays

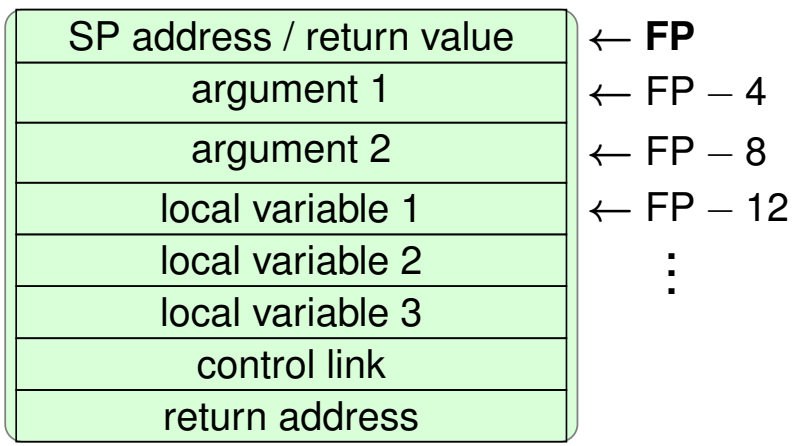
The size and "shape" (*e.g.*, which field is at what offset) can be determined at **compile-time**

$\Longrightarrow$ the compiler can emit code that accesses any field in an activation record, using an offset from the frame pointer

- Usually we think of "stacks" as a data structure, with access only to the front
- With assembly, we operate at such a low level that there is no reason we cannot read **any** element of the stack.

The "stack" is essentially just one very large array!

- Supposing a 4-byte 'word' for each item on the stack:



Example subroutines: revisited

```
public static
  void main(String []args){
    f(0); System.out.println();
    g(0); System.out.println();
    h(0); }

static void f (int n) {
    report("entering f", n); g(n+1);
    report("leaving f", n); }

static void g (int n) {
    report("entering g", n); h(n+1);
    report("leaving g", n); }

static void h (int n) {
    report("entering h", n);
    report("leaving h", n); }

static void
  report (String s, int n) {
    for (int i=0; i<3*n; i++)
      System.out.print(" ");
    System.out.println(s); }
```

Example subroutines: revisited

```
public static
  void main(String []args){
    f(0); System.out.println();
    g(0); System.out.println();
    h(0); }

static void f (int n) {
    report("entering f", n); g(n+1);
    report("leaving f", n); }

static void g (int n) {
    report("entering g", n); h(n+1);
    report("leaving g", n); }

static void h (int n) {
    report("entering h", n);
    report("leaving h", n); }

static void
  report (String s, int n) {
    for (int i=0; i<3*n; i++)
      System.out.print(" ");
    System.out.println(s); }
```

```
begin f
   begin g
      begin h
      end h
   end g
end f

begin g
   begin h
   end h
end g

begin h
end h
```

Example subroutines: revisited

```
public static
  void main(String []args){
    f(0); System.out.println();
    g(0); System.out.println();
    h(0); }

static void f (int n) {
    report("entering f", n); g(n+1);
    report("leaving f", n); }

static void g (int n) {
    report("entering g", n); h(n+1);
    report("leaving g", n); }

static void h (int n) {
    report("entering h", n);
    report("leaving h", n); }

static void
  report (String s, int n) {
    for (int i=0; i<3*n; i++)
      System.out.print(" ");
    System.out.println(s); }
```

```
begin f
   begin g
      begin h
      end h
   end g
end f

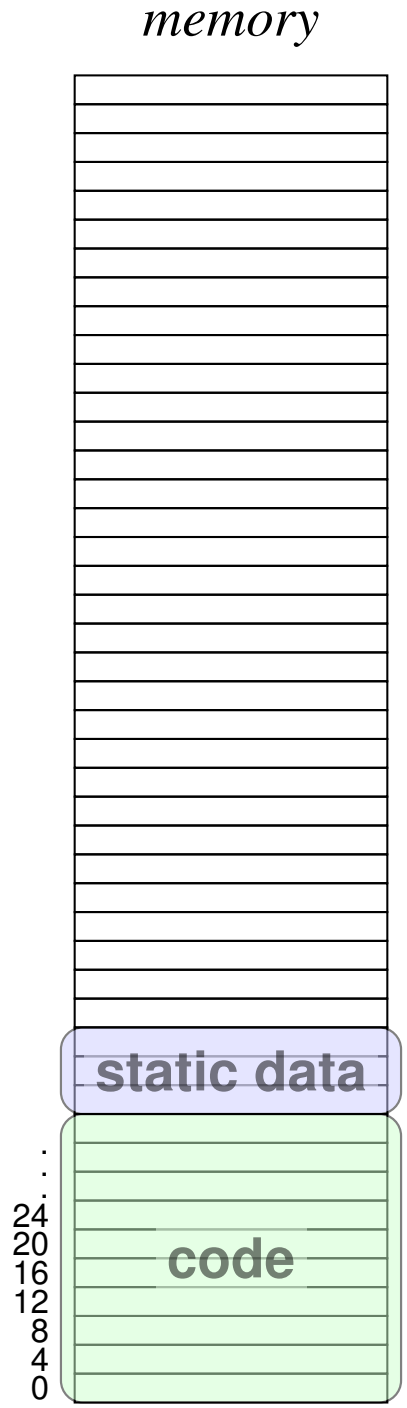
begin g
   begin h
   end h
end g

begin h
end h
```

. . . how does the stack behave for this program (using the simple frame structure above)?

Sketch: growth of the stack

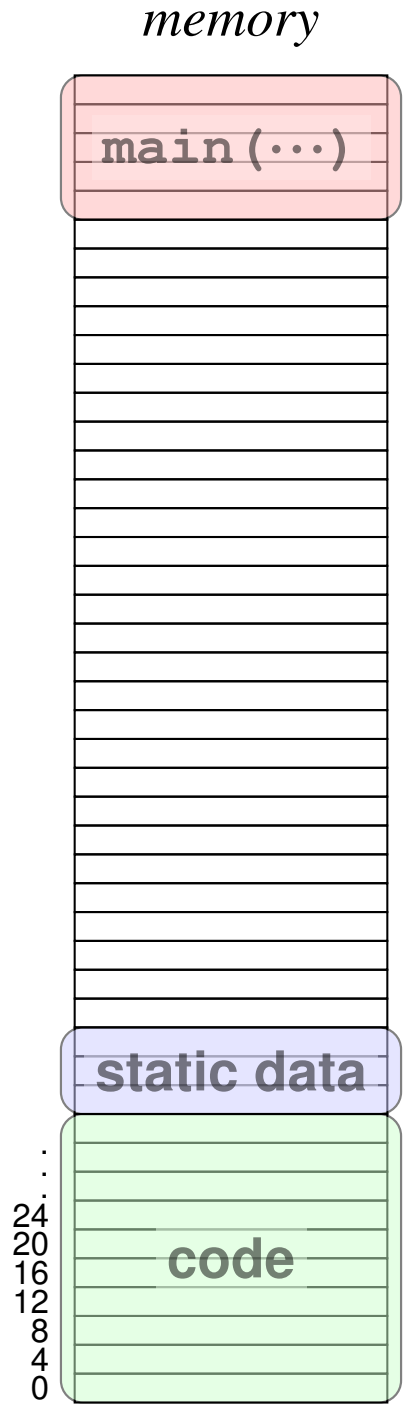
```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

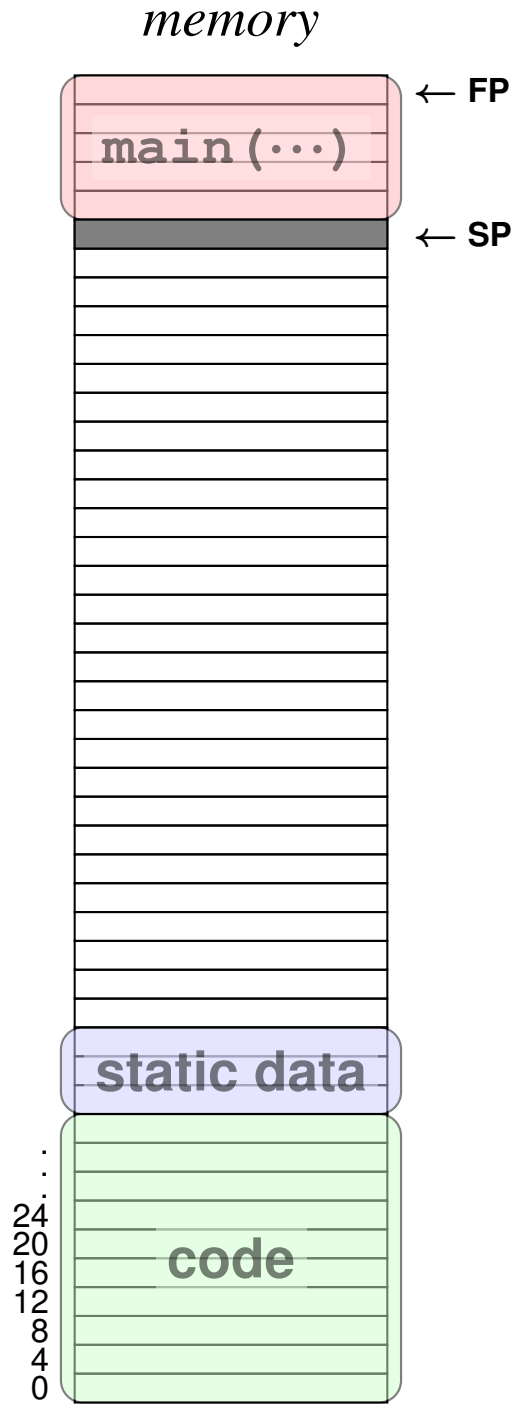
- The stack grows as procedures are called and shrinks when they terminate.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

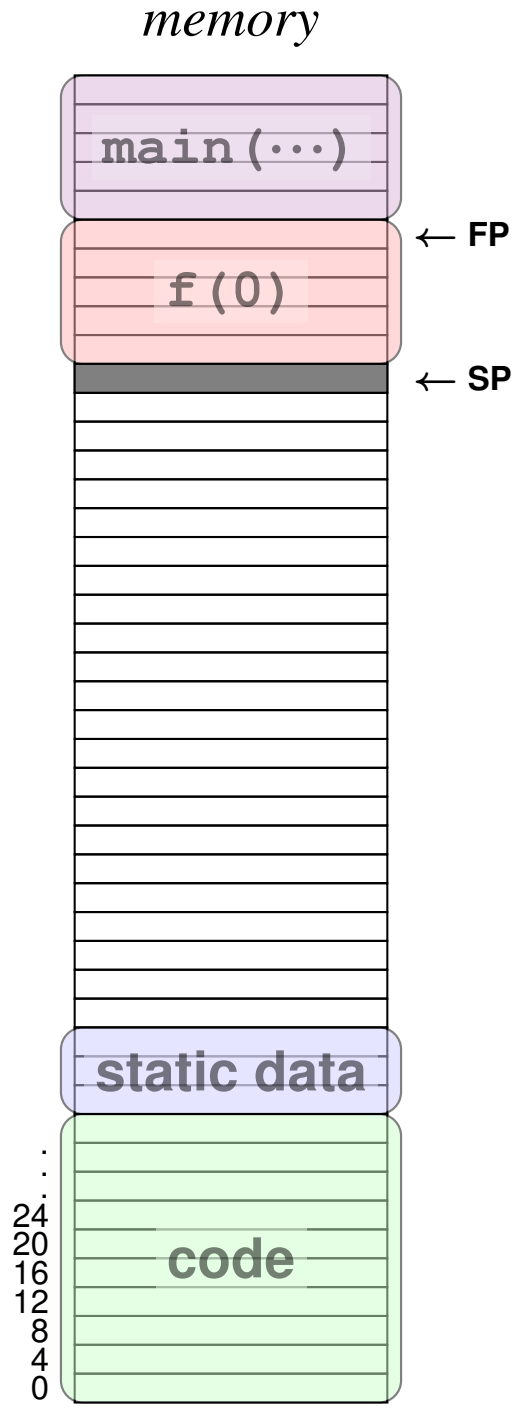
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

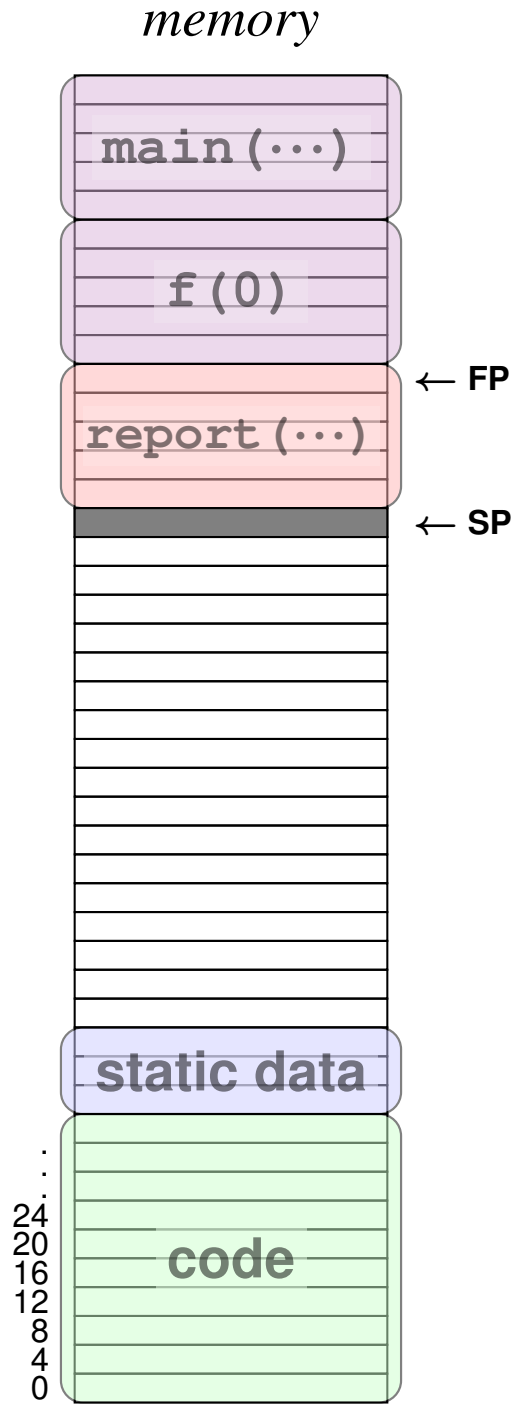
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

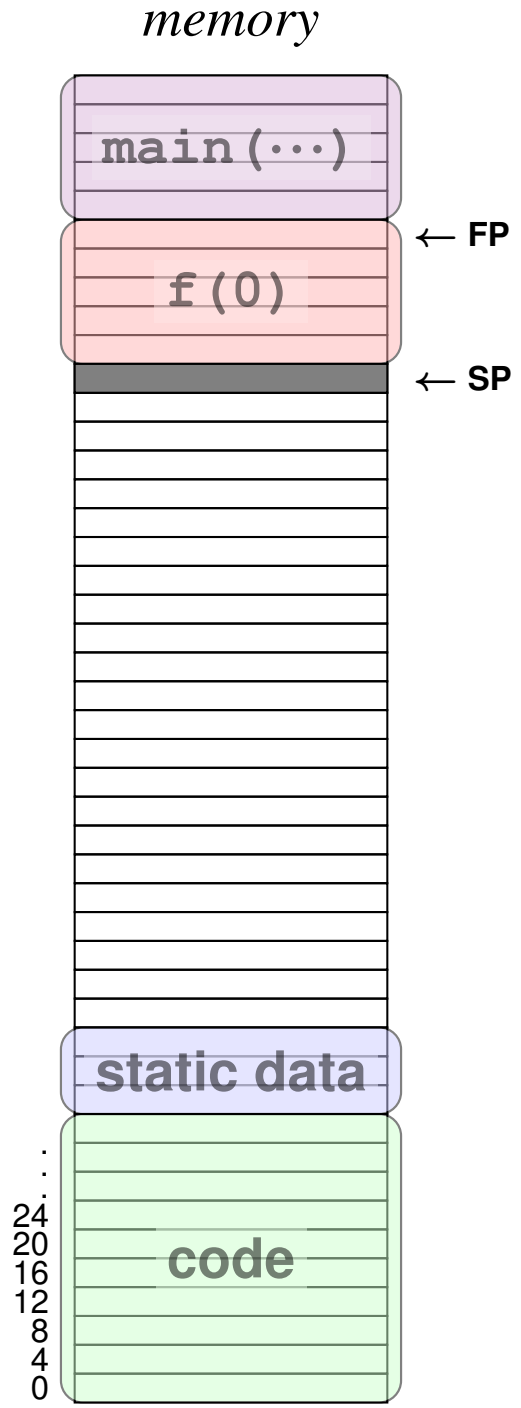
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

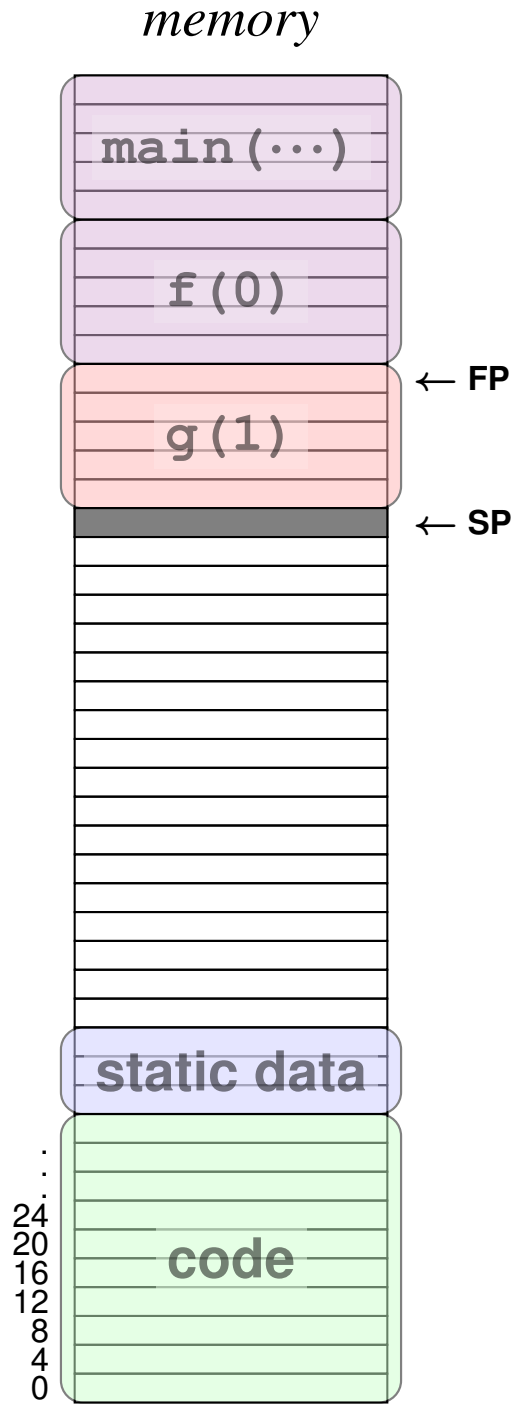
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

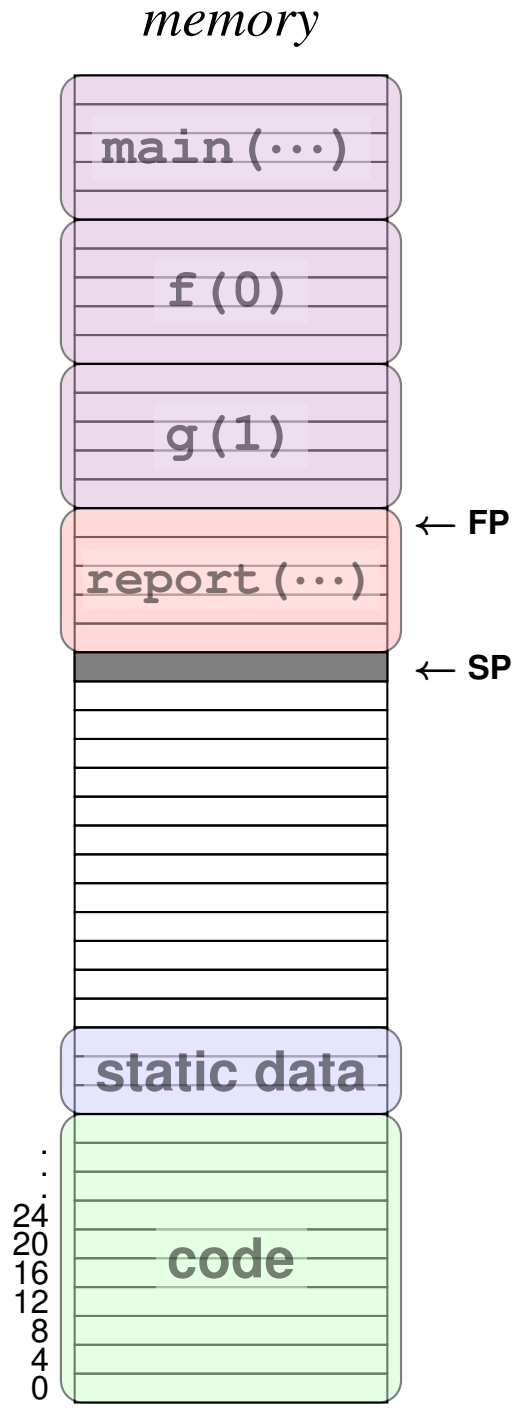
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

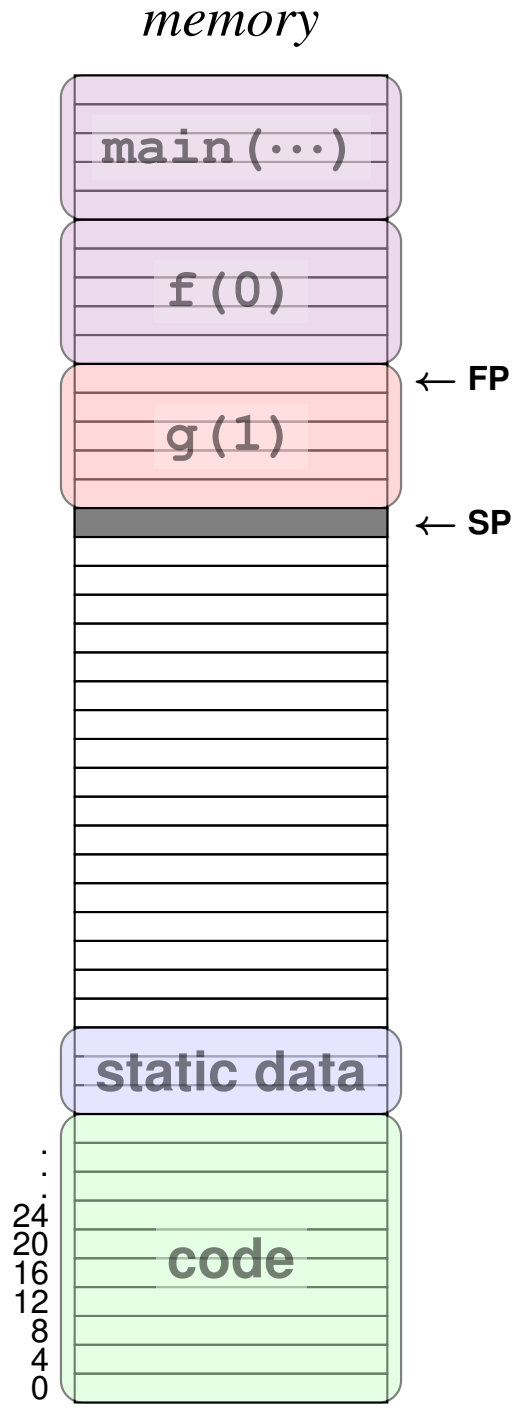
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

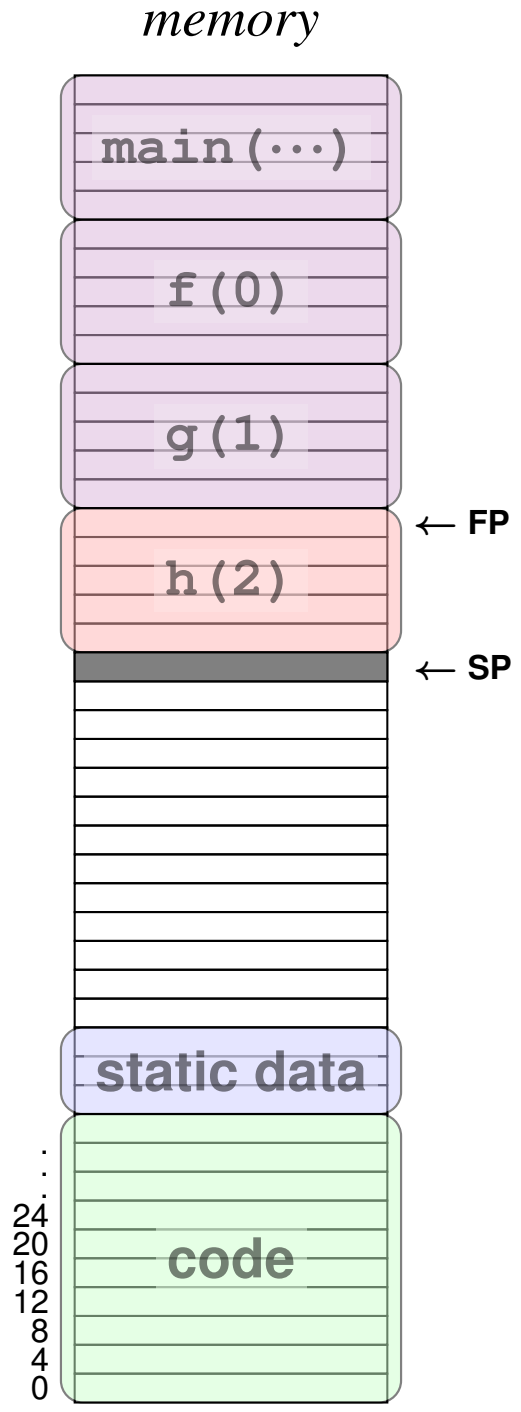
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

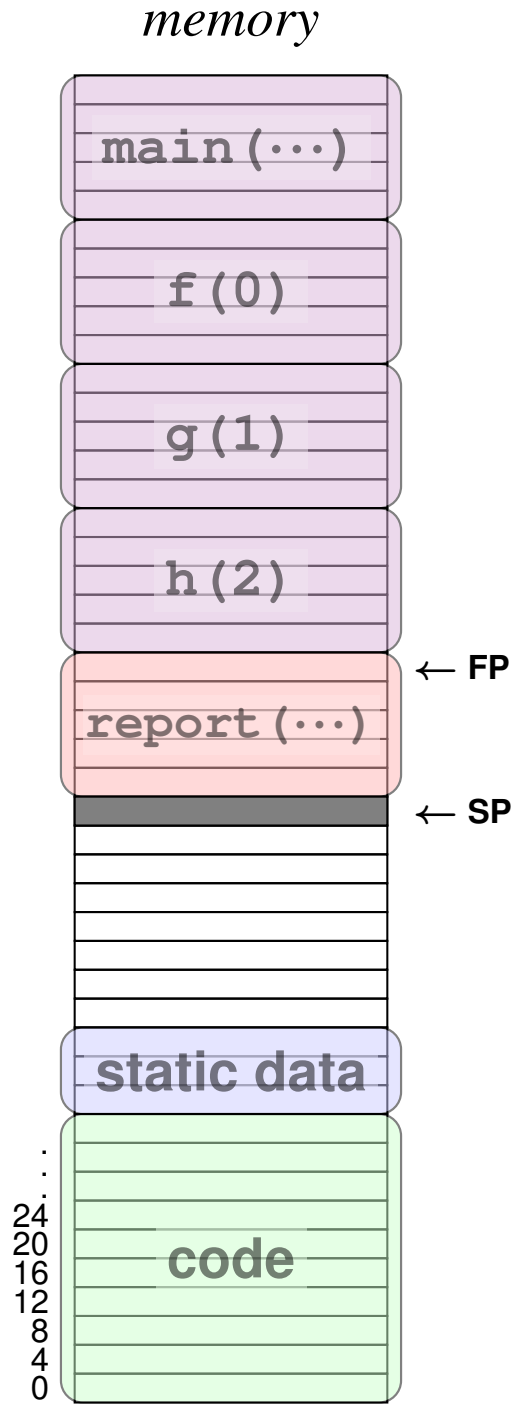
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

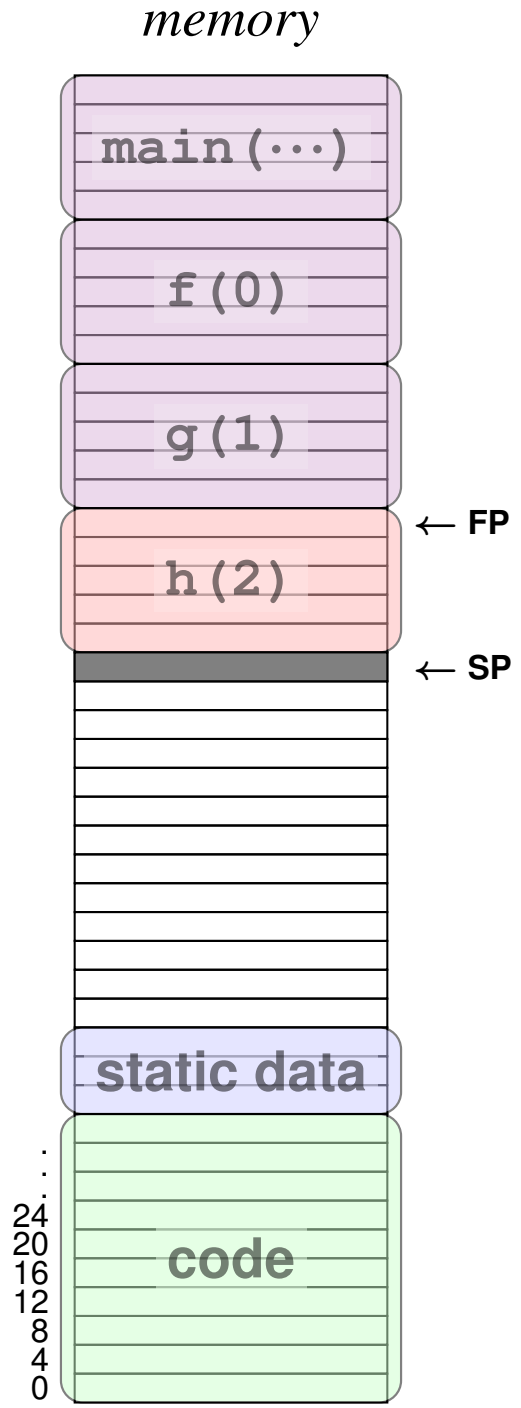
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

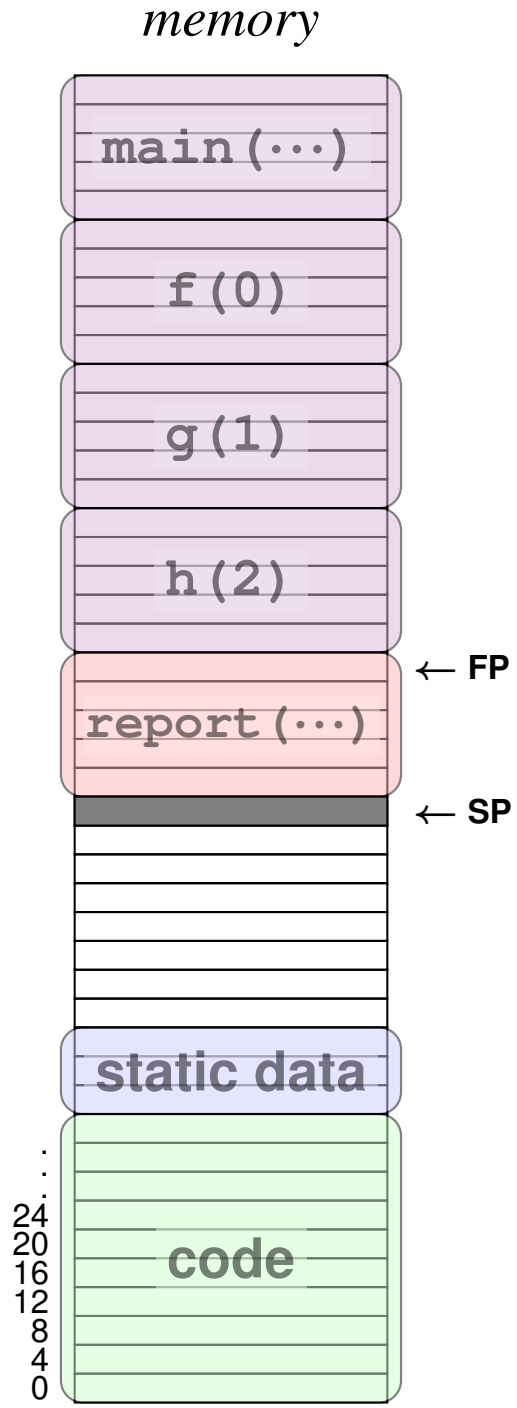
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

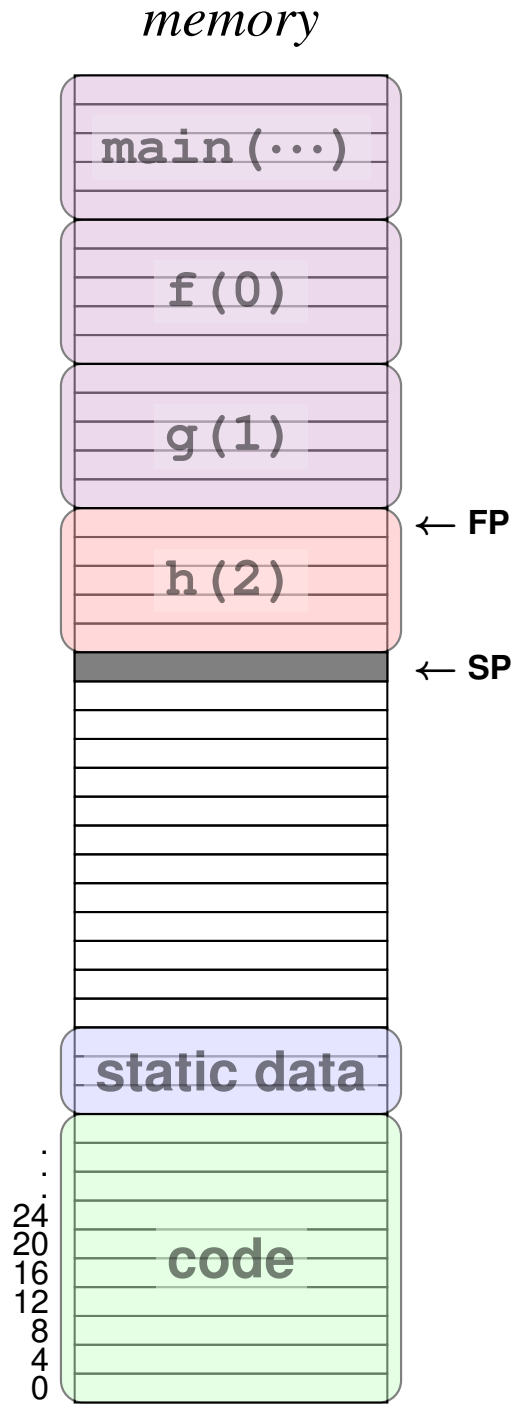
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

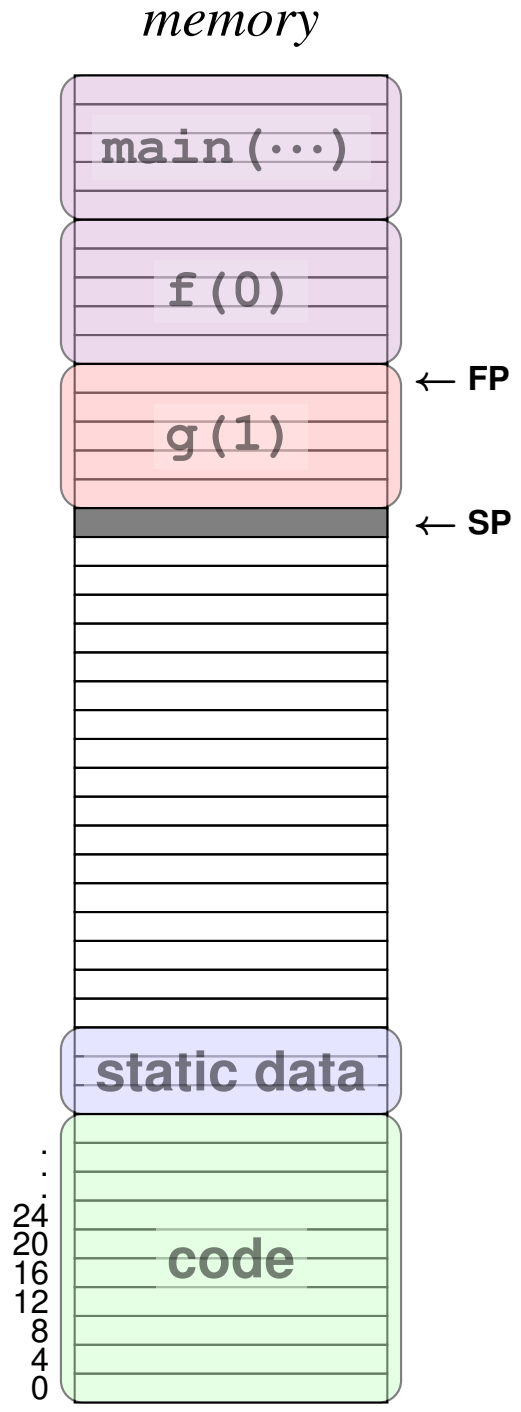
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

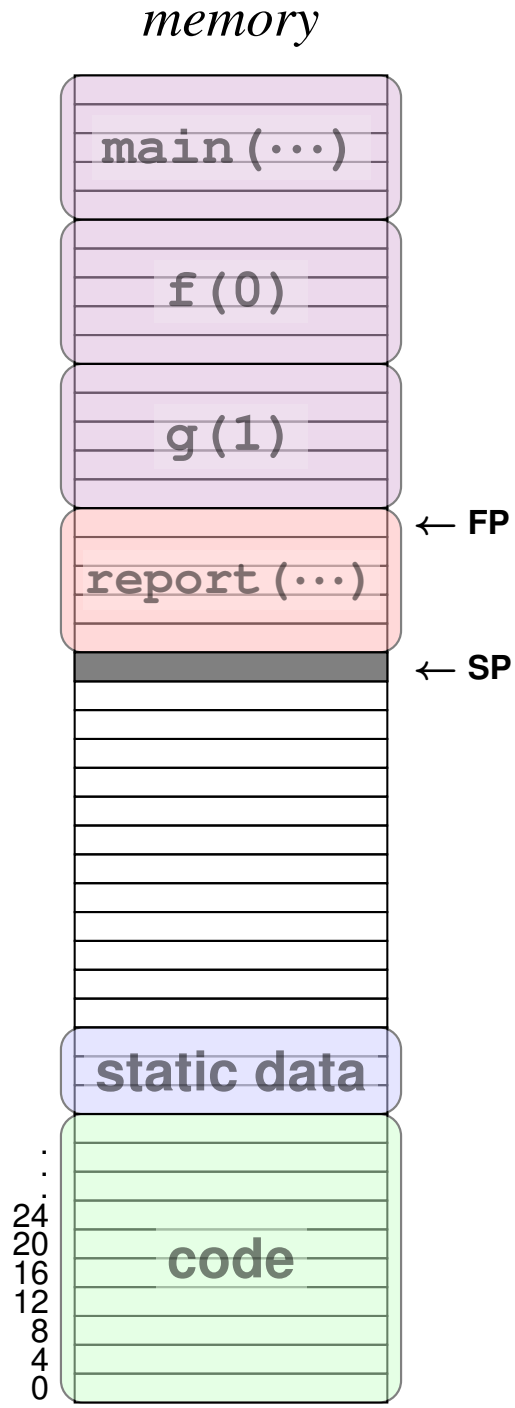
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

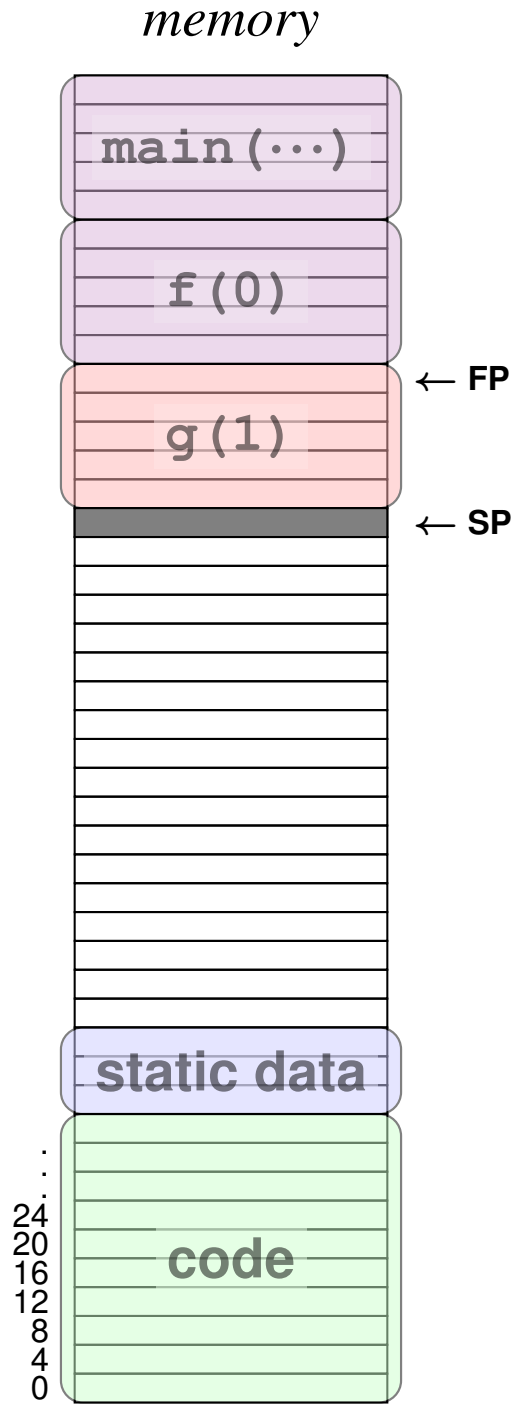
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

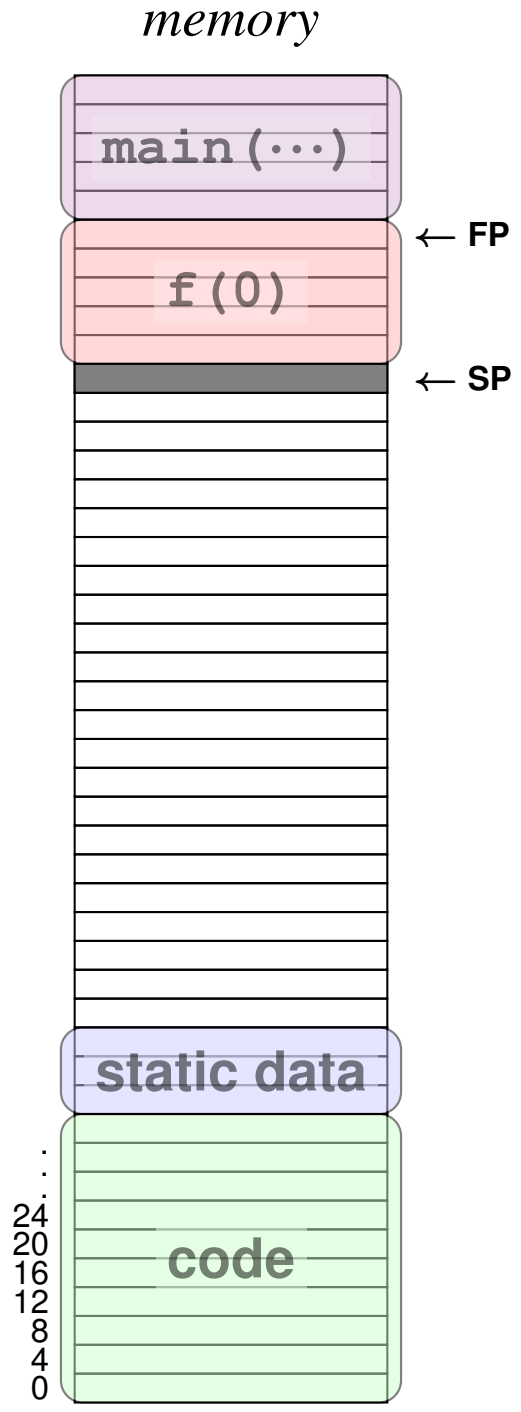
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

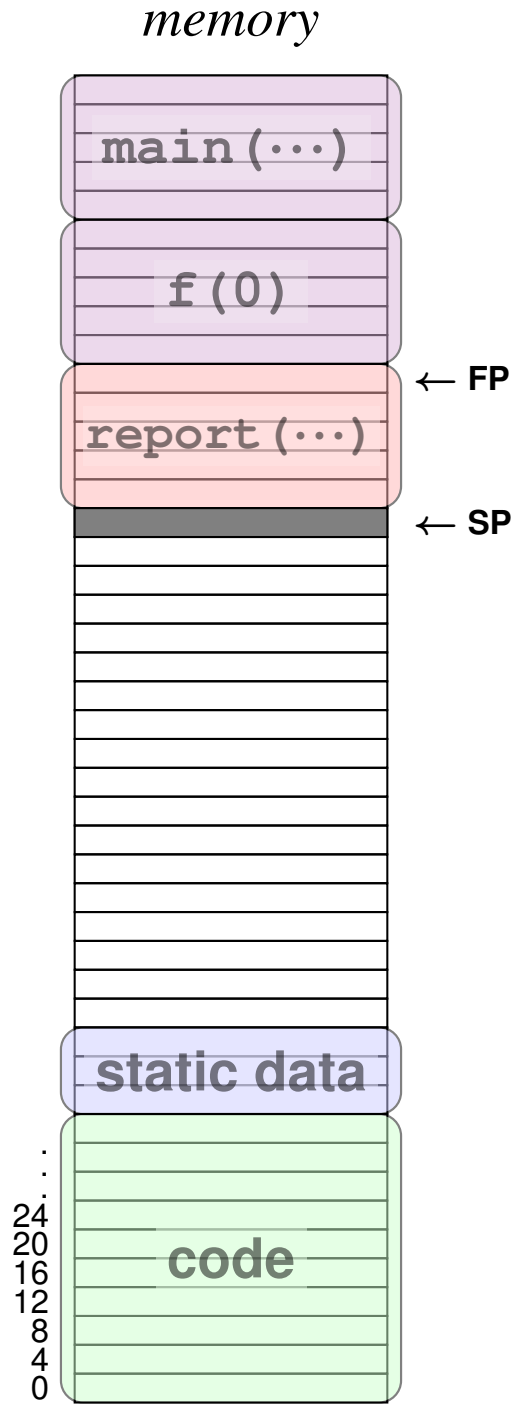
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

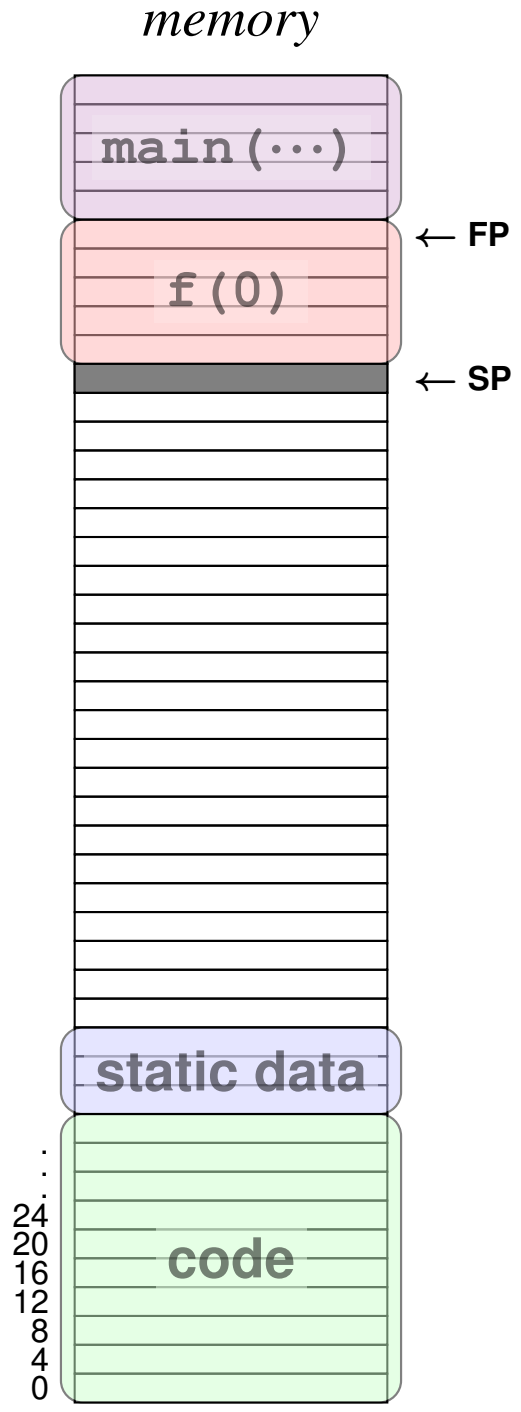
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

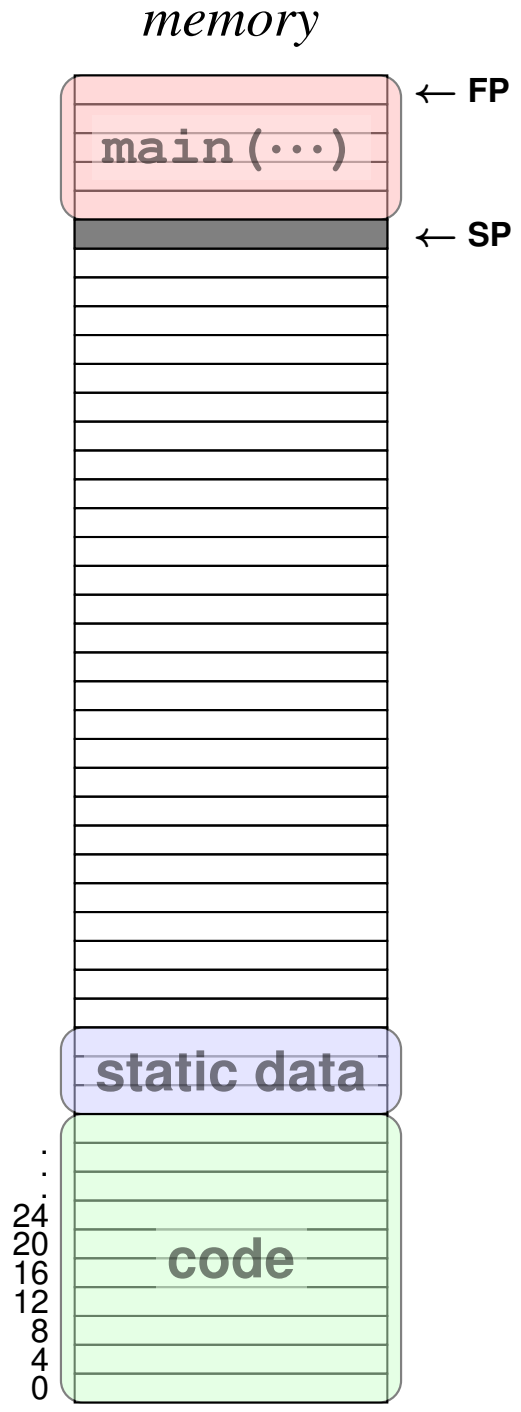
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

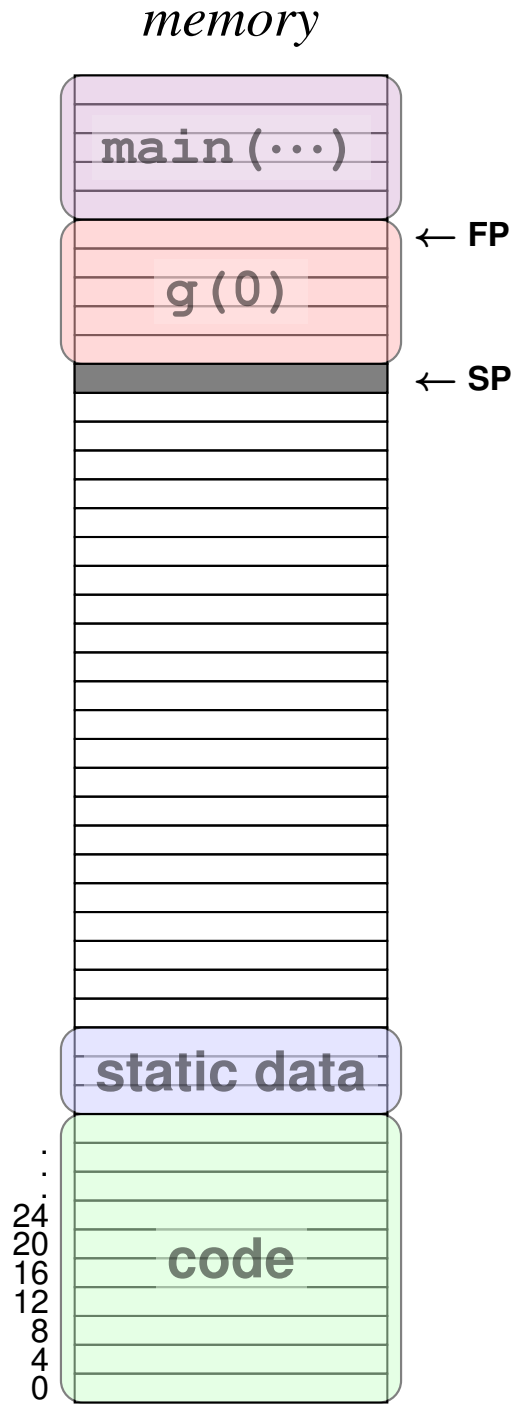
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

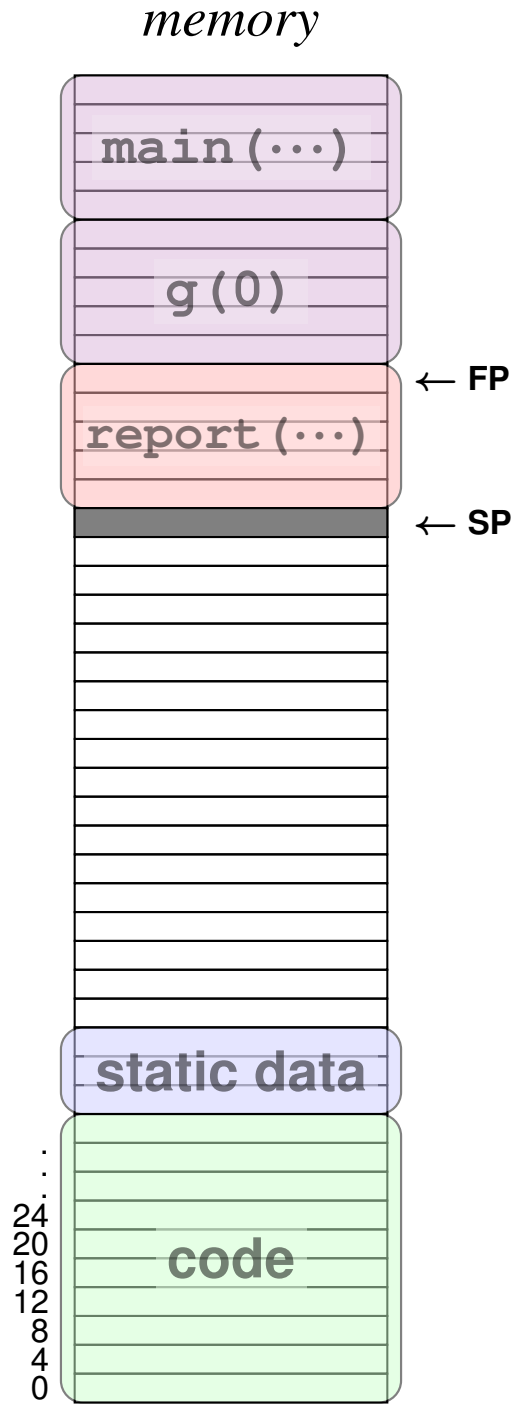
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

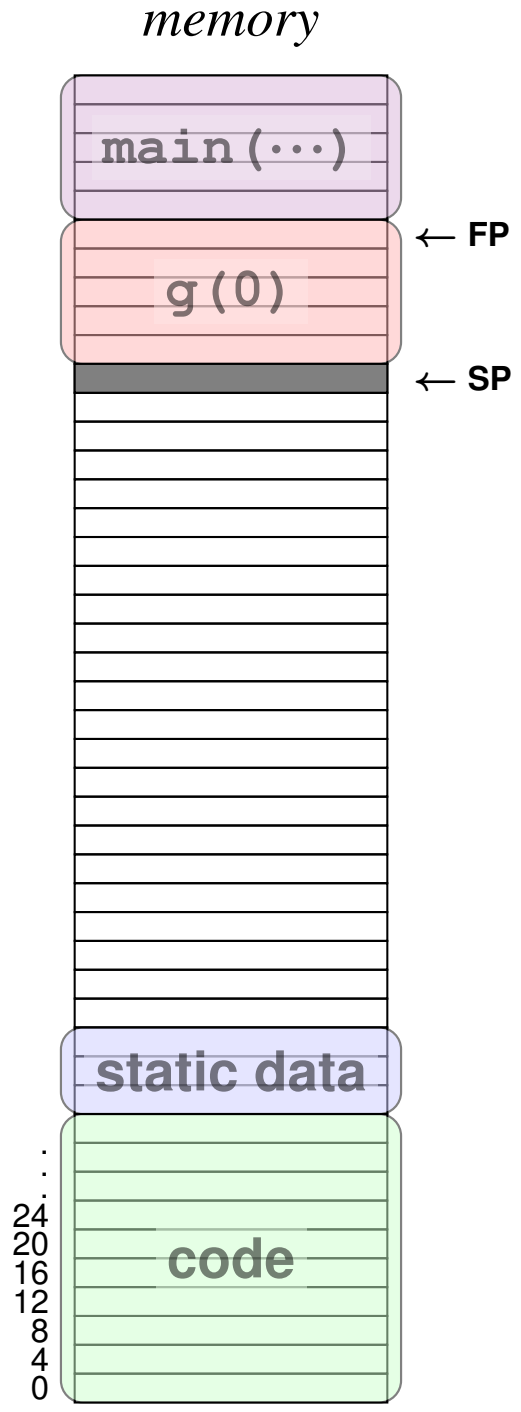
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

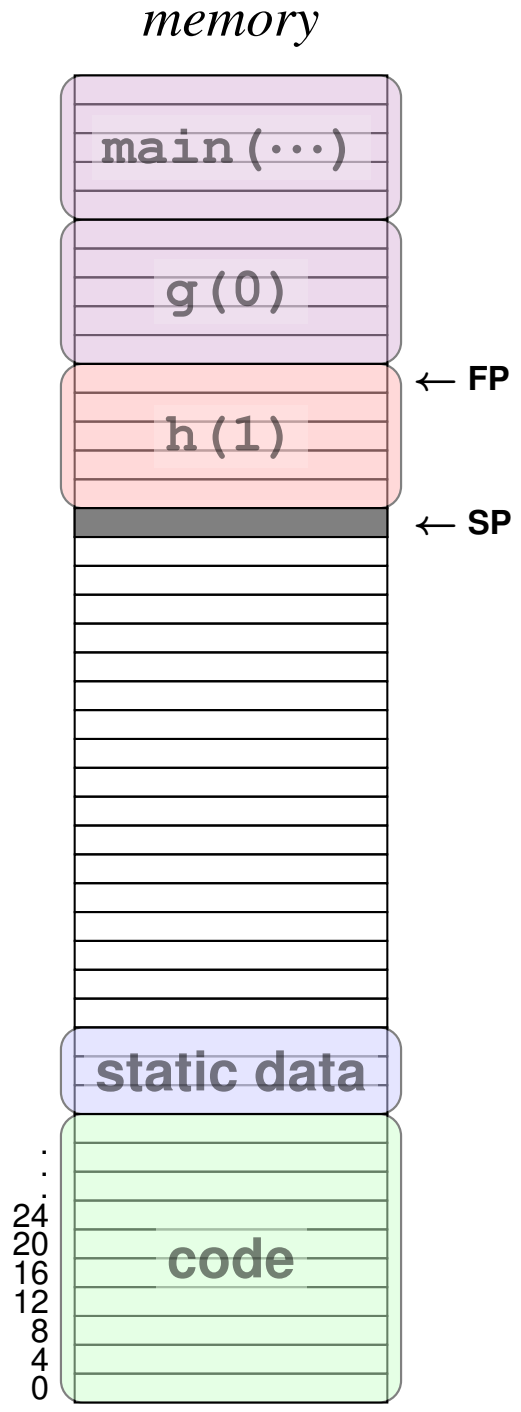
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

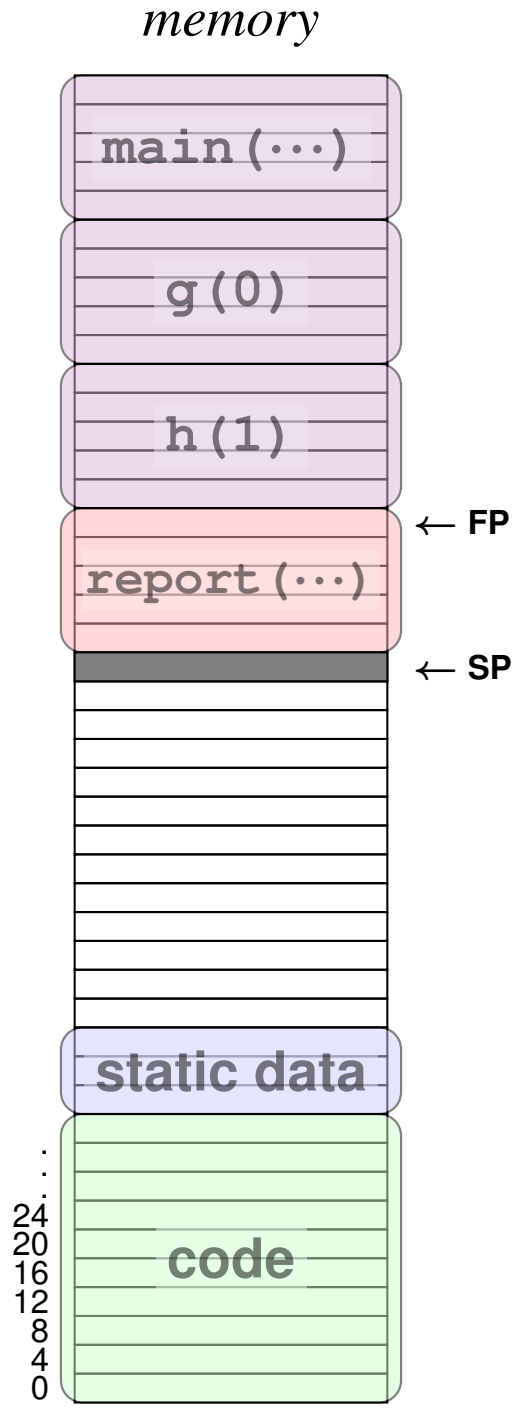
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

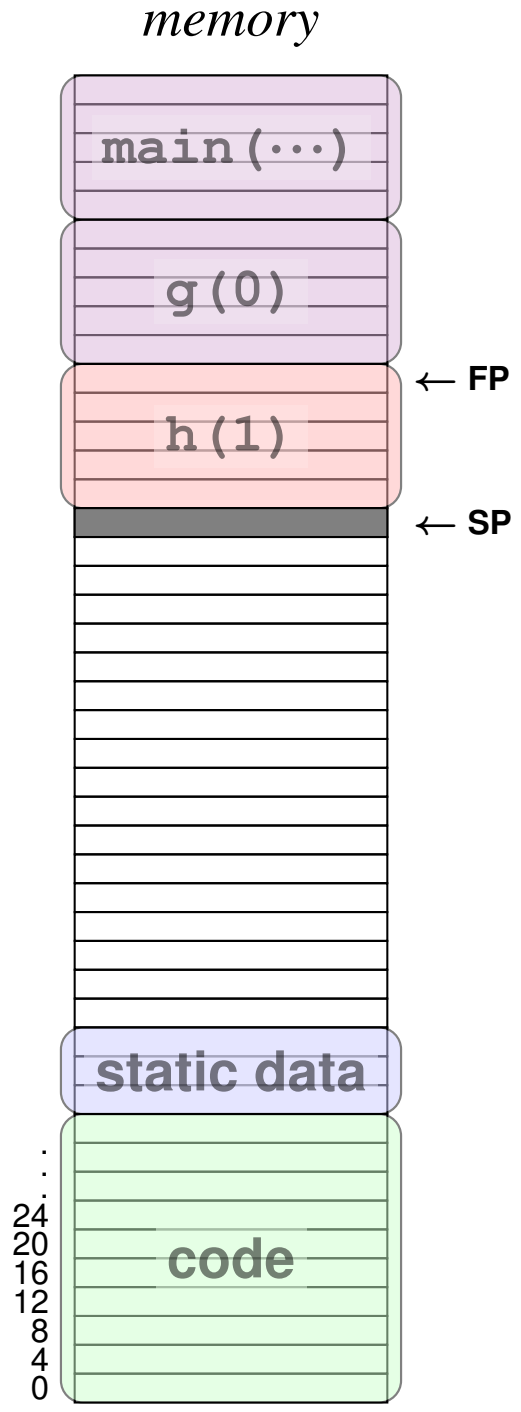
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

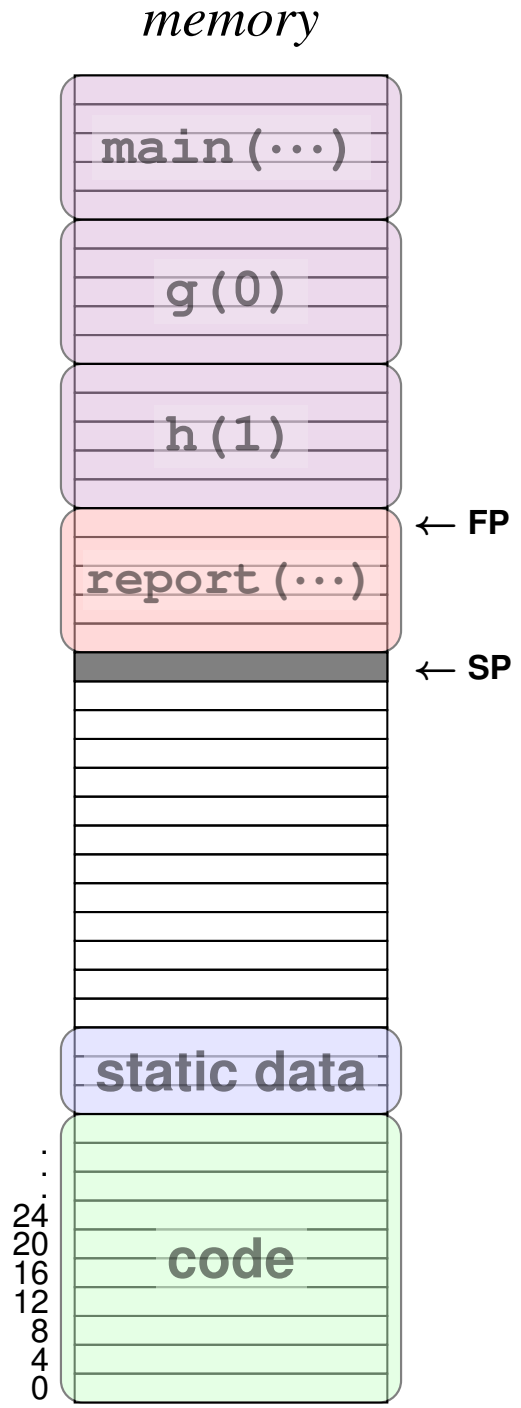
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

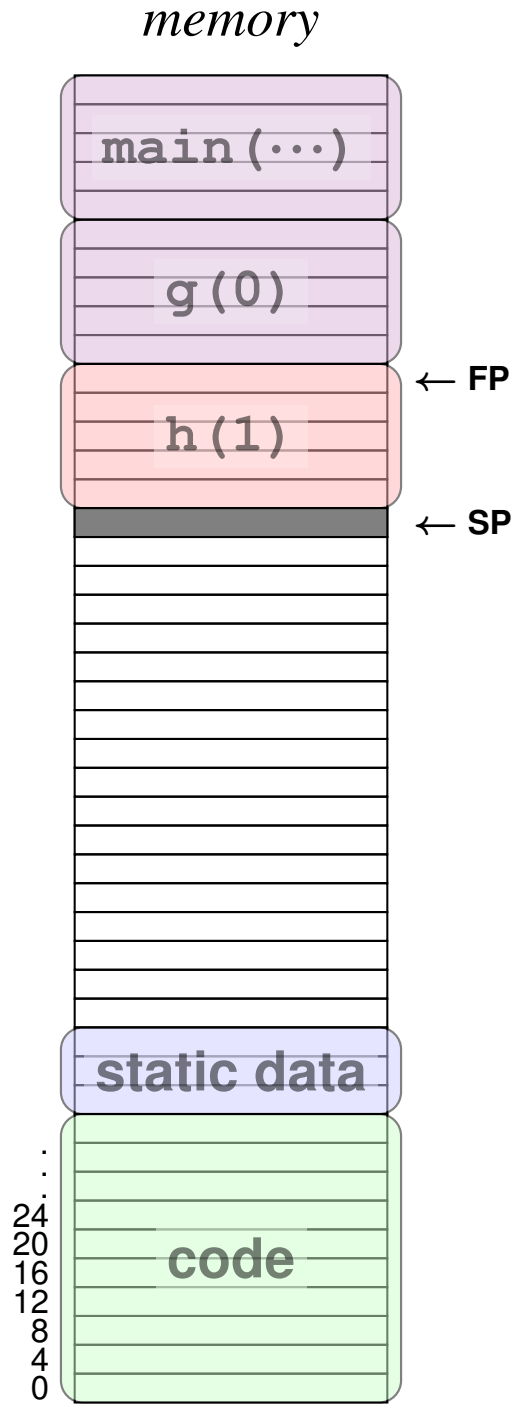
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

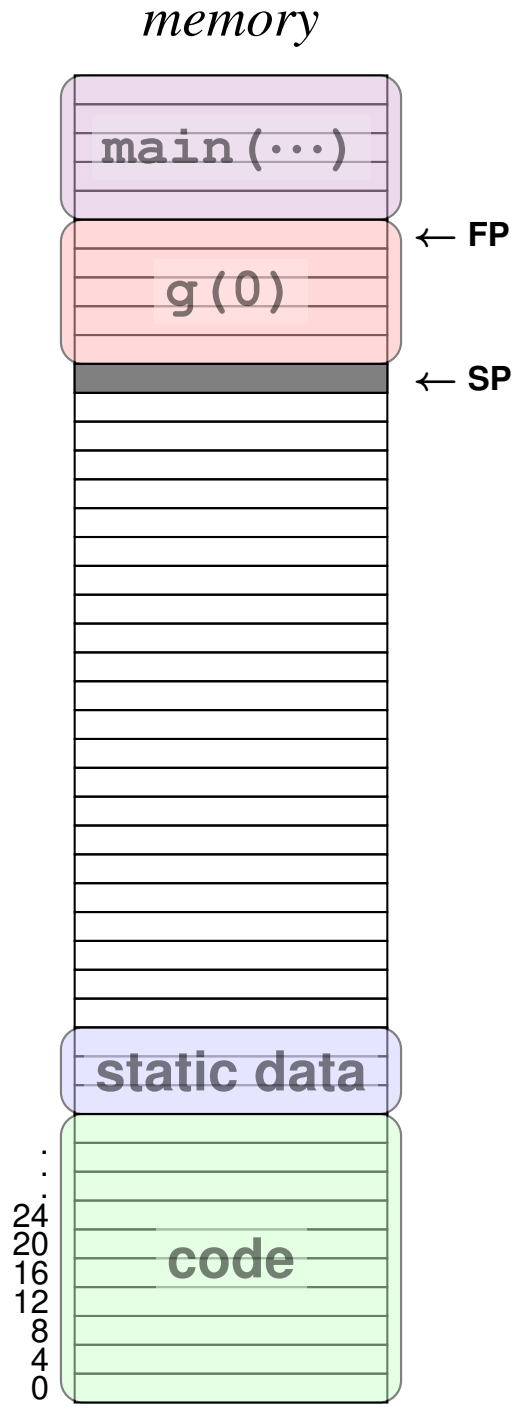
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

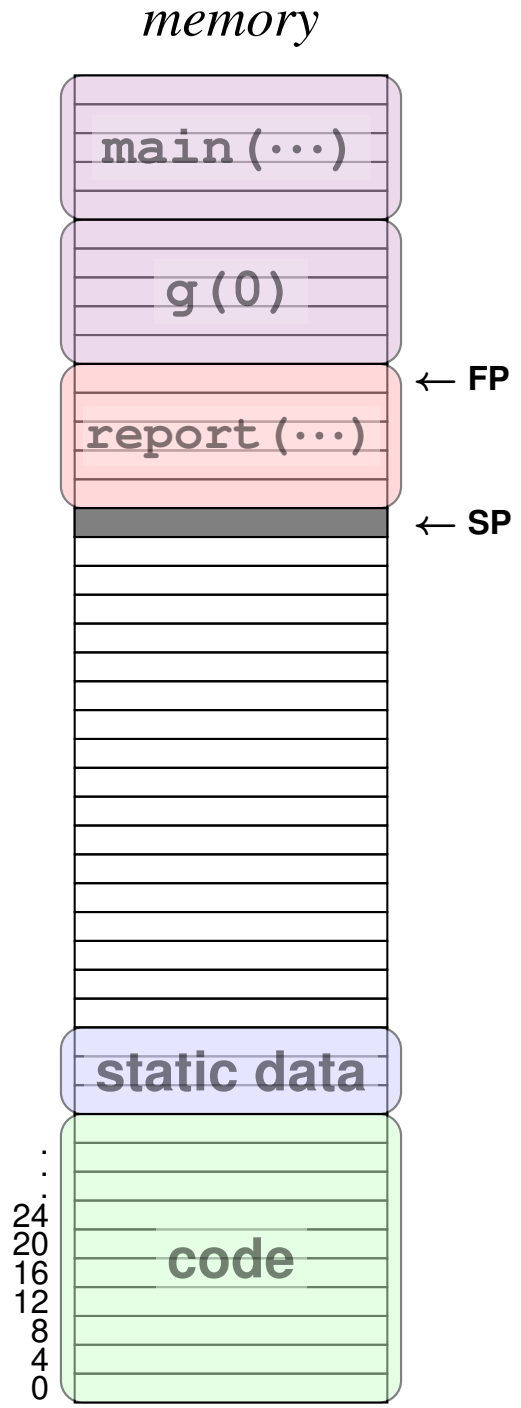
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

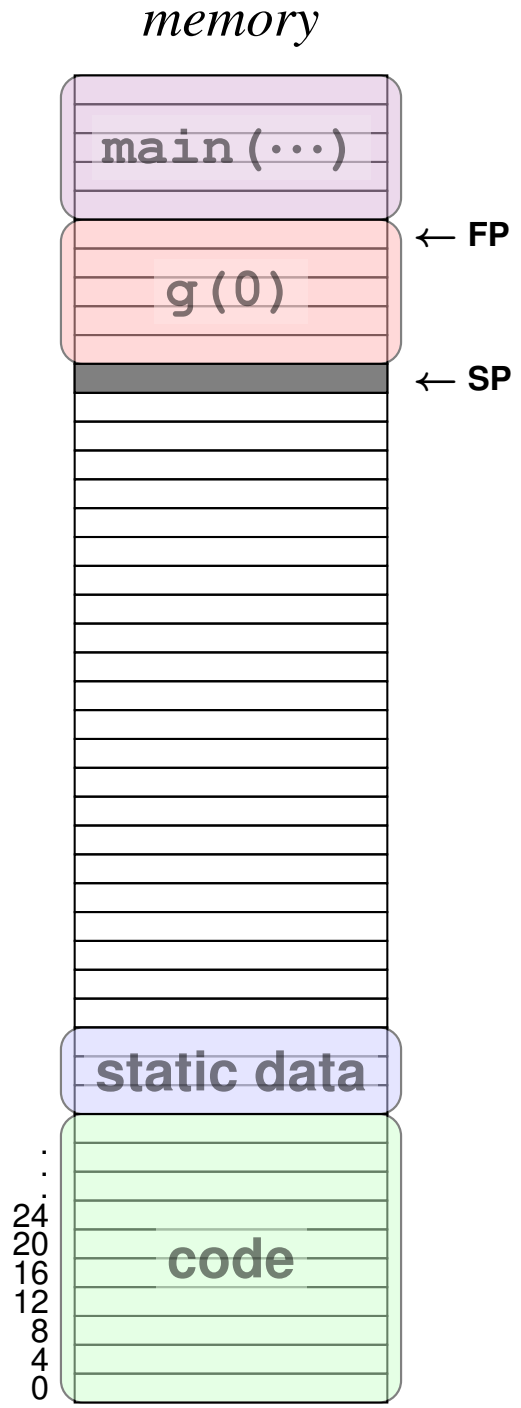
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

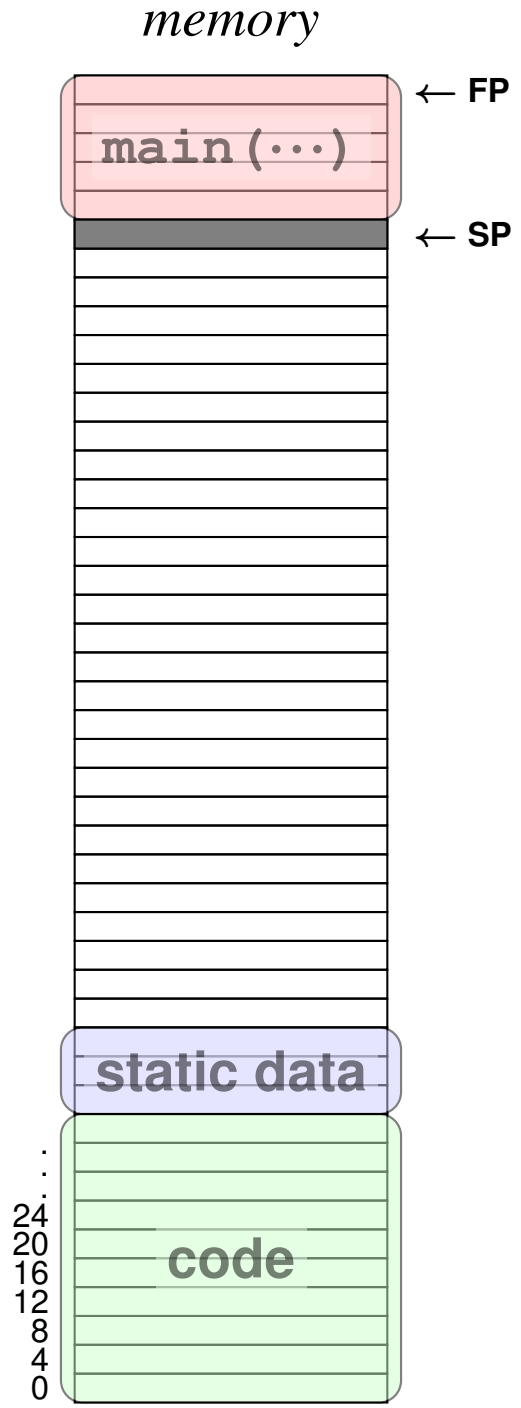
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

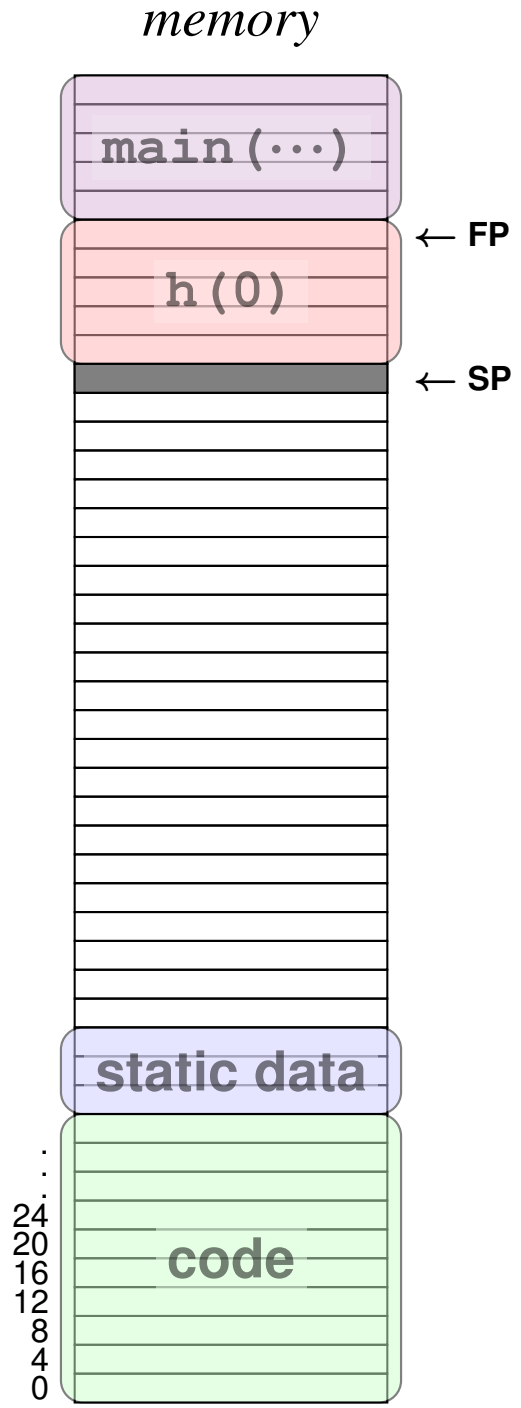
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

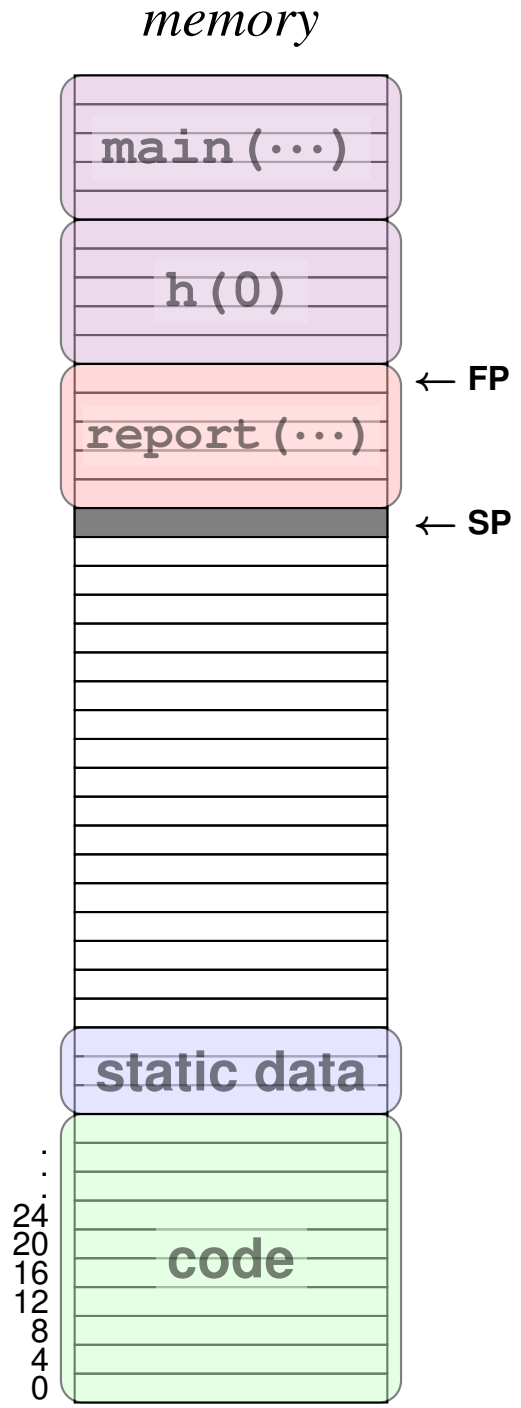
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

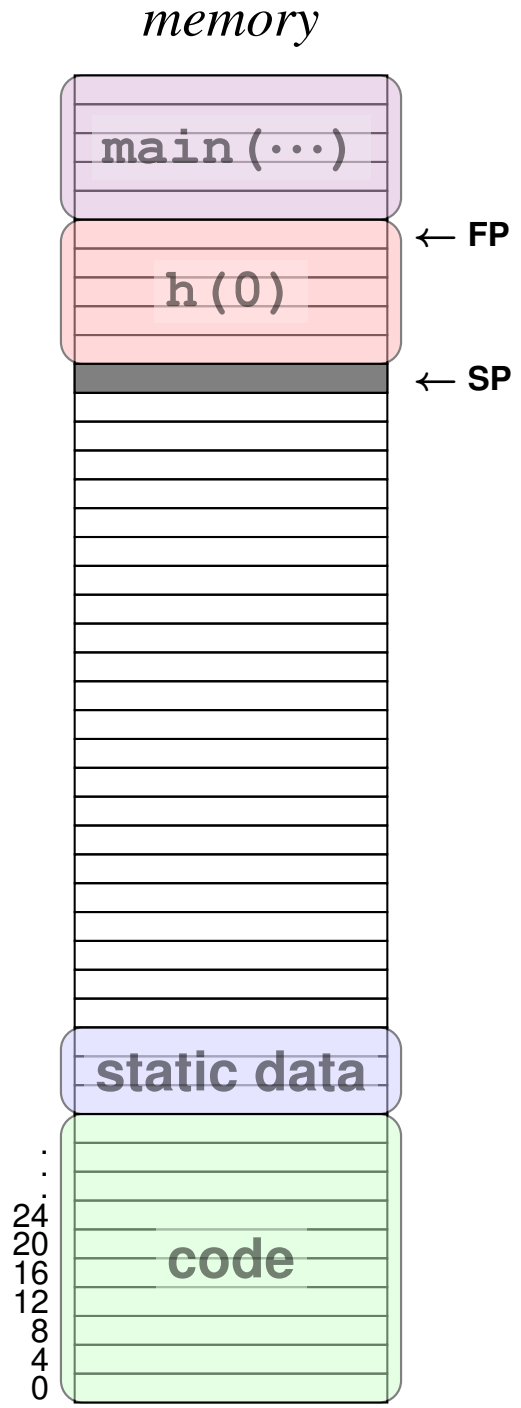
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

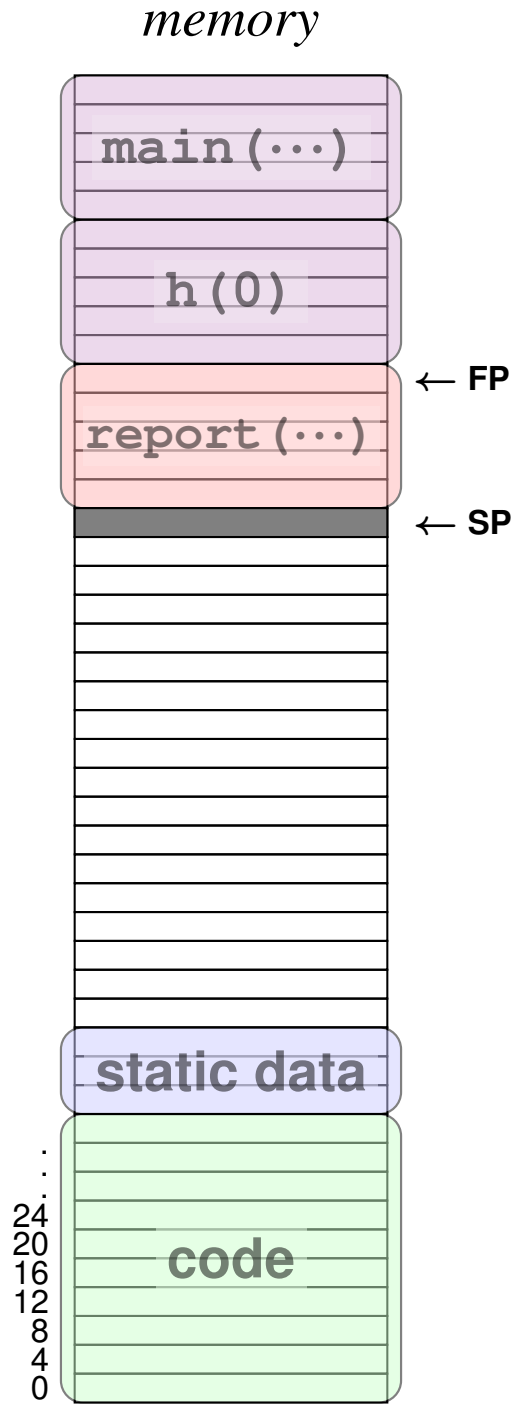
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

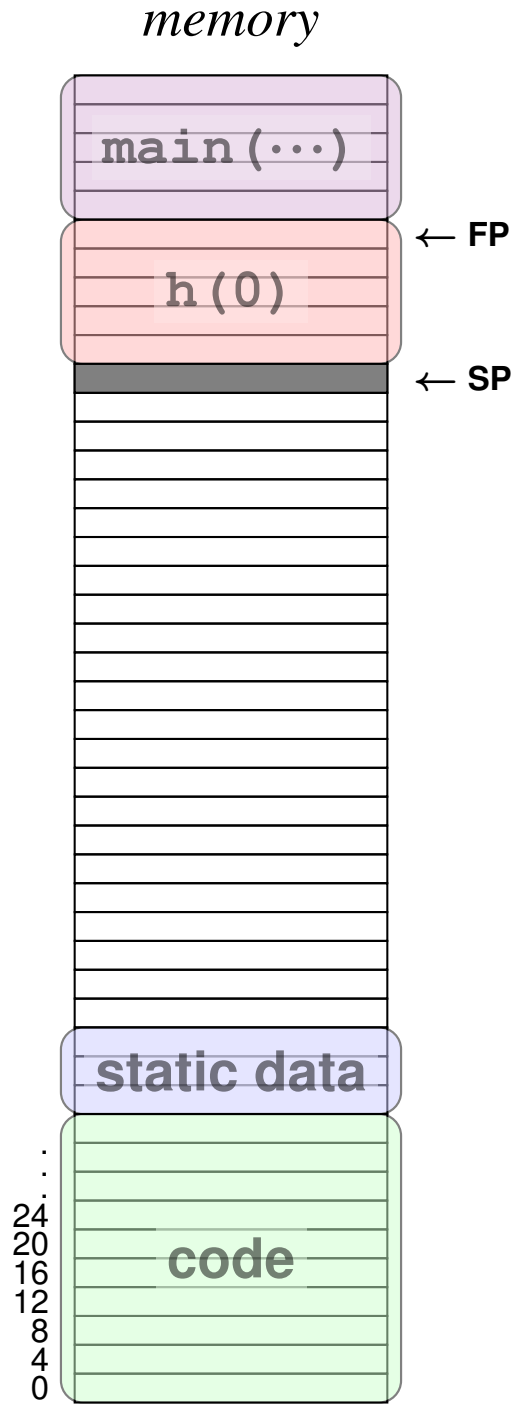
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

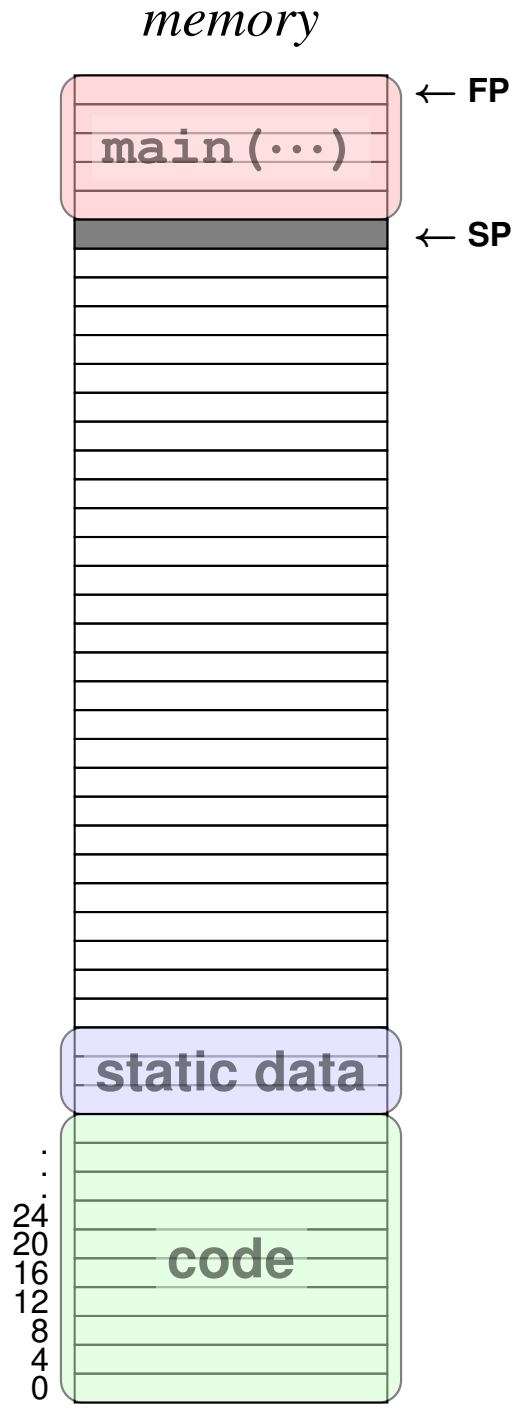
- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Sketch: growth of the stack

```
public static
void main(String []args){
   f(0); println();
   g(0); println();
   h(0); }
static void f (int n) {
   /* ... */ }
static void g (int n) {
   /* ... */ }
static void h (int n) {
   /* ... */ }
static void report (String s, int n) {
   /* ... */ }
```

- The stack grows as procedures are called and shrinks when they terminate.
- We maintain **SP** and **FP** to provide access to local variables, and to manage further procedure calls.



Activation records in practice

Activation records are designed with the capabilities of the machine (and experience of software design) in mind.

Activation records in practice

Activation records are designed with the capabilities of the machine (and experience of software design) in mind.

- On machines with many registers: much of the AR (= activation record) is actually stored in those registers
  - **Especially:** the first few arguments, the first few local variables, and the result — to speed up access

Activation records in practice

Activation records are designed with the capabilities of the machine (and experience of software design) in mind.

- On machines with many registers: much of the AR (= activation record) is actually stored in those registers
  - **Especially:** the first few arguments, the first few local variables, and the result — to speed up access
  - This also means the AR may have to save the register values of the calling procedure — procedure invocations are more complicated

Activation records in practice

Activation records are designed with the capabilities of the machine (and experience of software design) in mind.

- On machines with many registers: much of the AR (= activation record) is actually stored in those registers
  - **Especially:** the first few arguments, the first few local variables, and the result — to speed up access
  - This also means the AR may have to save the register values of the calling procedure — procedure invocations are more complicated
- Different solutions will lead to differences in performance (speed v. memory)

Activation records in practice

Activation records are designed with the capabilities of the machine (and experience of software design) in mind.

- On machines with many registers: much of the AR (= activation record) is actually stored in those registers
  - **Especially:** the first few arguments, the first few local variables, and the result — to speed up access
  - This also means the AR may have to save the register values of the calling procedure — procedure invocations are more complicated
- Different solutions will lead to differences in performance (speed v. memory)
- Standardising on a given format can allow software to use libraries written in a different language entirely

Activation records in practice

Activation records are designed with the capabilities of the machine (and experience of software design) in mind.

- On machines with many registers: much of the AR (= activation record) is actually stored in those registers
  - **Especially:** the first few arguments, the first few local variables, and the result — to speed up access
  - This also means the AR may have to save the register values of the calling procedure — procedure invocations are more complicated
- Different solutions will lead to differences in performance (speed v. memory)
- Standardising on a given format can allow software to use libraries written in a different language entirely

There is a lot of freedom to experiment — but good software engineering will consider priorities (speed vs. memory) and existing standards

Heap data

What about persistent objects that emerge at run-time?
Consider the following Java program.

```
public class A { ... }
public class B {
   A f () { return new A (); }
   void g () { A a = f(); ... } }
```

Heap data

What about persistent objects that emerge at run-time?
Consider the following Java program.

```
public class A { ... }
public class B {
   A f () { return new A (); }
   void g () { A a = f(); ... } }
```

The object in **red** cannot only be stored in an AR for an invocation of `f`: it will outlive the invocation.

(The AR will have to store a *reference* to it, at least
— but a reference to where? Not a static address, or the stack. . .)

Heap data

What about persistent objects that emerge at run-time?
Consider the following Java program.

```
public class A { ... }
public class B {
   A f () { return new A (); }
   void g () { A a = f(); ... } }
```

The object in **red** cannot only be stored in an AR for an invocation of `f`: it will outlive the invocation.

(The AR will have to store a *reference* to it, at least
— but a reference to where? Not a static address, or the stack. . .)

We need an additional memory region,
for persistent objects which endure

- potentially much longer than the activation creating them;
- potentially much less time than the program run-time.

This region is called the **heap**.

Multiple meanings of “Heap”

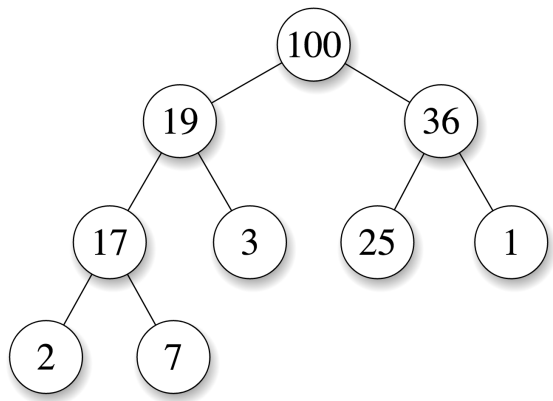
The meaning of the term “heap” here
is different in other parts of CS.

In particular: in the study of algorithms / data structures, a heap
is a tree-based data structure for ranking items

Multiple meanings of “Heap”

The meaning of the term “heap” here
is different in other parts of CS.

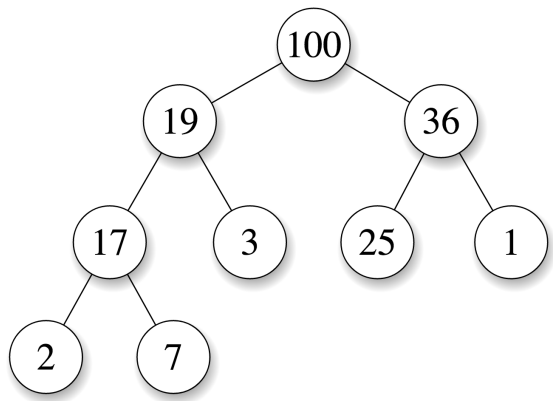
In particular: in the study of algorithms / data structures, a heap
is a tree-based data structure for ranking items — for example:



Multiple meanings of “Heap”

The meaning of the term “heap” here
is different in other parts of CS.

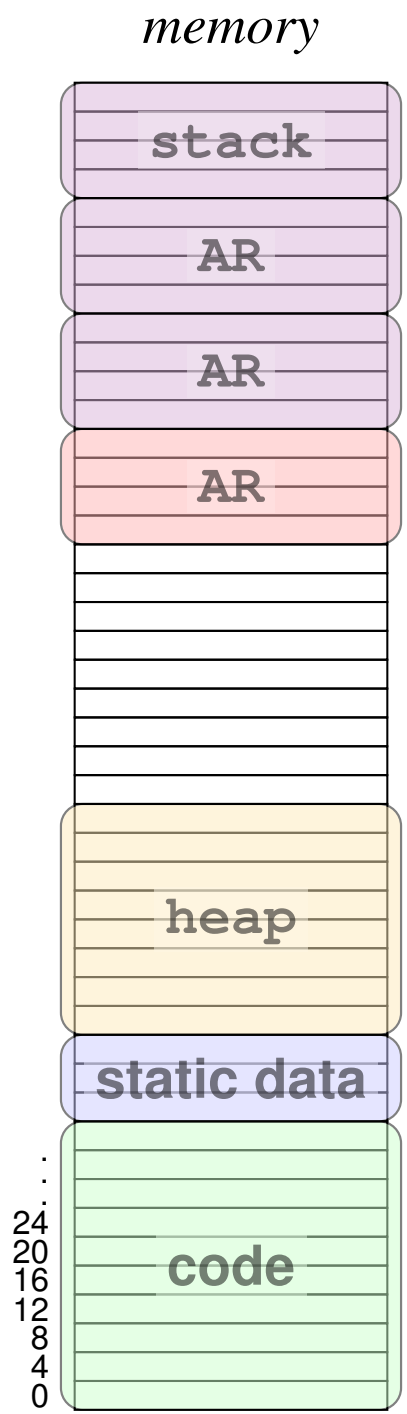
In particular: in the study of algorithms / data structures, a heap
is a tree-based data structure for ranking items — for example:



But our use of the term “heap” is a completely unrelated
concept.

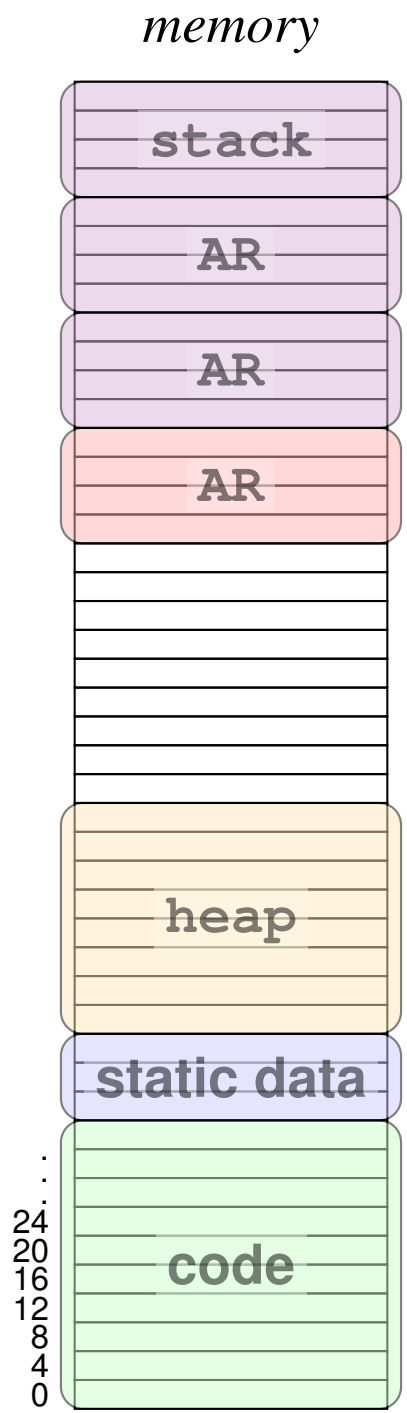
Sketch: memory layout

- **Code:** fixed size.
- **Static data:** fixed size, contains static variables.
- **Stack:** dynamic, grows and shrinks with addition/removals of ARs.
- **Heap:** dynamic, contains all other data.



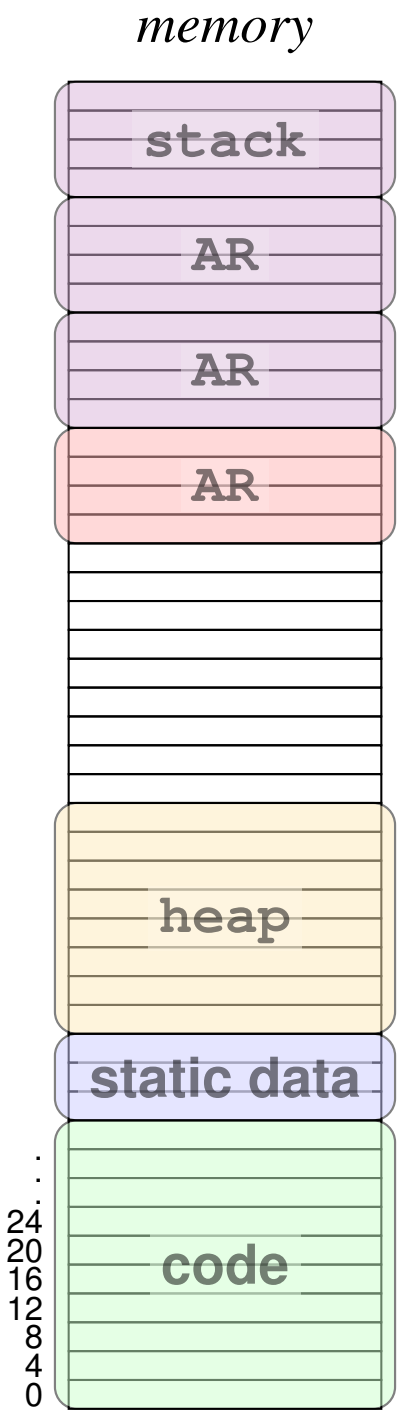
Sketch: memory layout

- **Code:** fixed size.
- **Static data:** fixed size, contains static variables.
- **Stack:** dynamic, grows and shrinks with addition/removals of ARs.
- **Heap:** dynamic, contains all other data. This can also grow/shrink to meet the storage needs of the program



Sketch: memory layout

- **Code:** fixed size.
- **Static data:** fixed size, contains static variables.
- **Stack:** dynamic, grows and shrinks with addition/removals of ARs.
- **Heap:** dynamic, contains all other data. This can also grow/shrink to meet the storage needs of the program —
  - **Manually:** *e.g.*, in C (`malloc`/`free`) and C++ (`new`/`delete`)
  - **Automatically:** through allocation and garbage collection (*e.g.*, in Java, Scala, Haskell, Python, Javascript)



Memory management

Both heap and stack have dynamically changing sizes.
We must make sure they don't grow into each other.

- If using a single block of memory:
  start the stack and heap at opposite ends of the memory.
- What if we run out?
  - terminate the program; cry
  - ask the OS if we can have more memory;
    remap memory usage to fit new allocated address space

The compiler must generate code that checks for (and deals with) this somehow (details are beyond to scope of this module).

What's next?

The above is a strategy to realise basic imperative / functional programs. To produce a code generator, we will need to choose a target platform — whose architecture will force us to fix some specific details of that strategy.

What's next?

The above is a strategy to realise basic imperative / functional programs. To produce a code generator, we will need to choose a target platform — whose architecture will force us to fix some specific details of that strategy.

Target architectures which we consider:

What's next?

The above is a strategy to realise basic imperative / functional programs. To produce a code generator, we will need to choose a target platform — whose architecture will force us to fix some specific details of that strategy.

Target architectures which we consider:

- **Stack machines** — *e.g.*, JVM, WebAssembly, ZPU
  - No general-purpose registers
  - Almost all registers are dedicated to program execution

What's next?

The above is a strategy to realise basic imperative / functional programs. To produce a code generator, we will need to choose a target platform — whose architecture will force us to fix some specific details of that strategy.

Target architectures which we consider:

- **Stack machines** — *e.g.*, JVM, WebAssembly, ZPU
  - No general-purpose registers
  - Almost all registers are dedicated to program execution
- **Register machines** — *e.g.*, x86, RISC-V, many others
  - Several general-purpose registers

What's next?

The above is a strategy to realise basic imperative / functional programs. To produce a code generator, we will need to choose a target platform — whose architecture will force us to fix some specific details of that strategy.

Target architectures which we consider:

- **Stack machines** — *e.g.*, JVM, WebAssembly, ZPU
  - No general-purpose registers
  - Almost all registers are dedicated to program execution
- **Register machines** — *e.g.*, x86, RISC-V, many others
  - Several general-purpose registers
- **Accumulator machines** — *e.g.*, ENIAC, IBM-701, PDP-8
  - A general purpose register which is the main (implicit) operand of most instructions

What's next?

The above is a strategy to realise basic imperative / functional programs. To produce a code generator, we will need to choose a target platform — whose architecture will force us to fix some specific details of that strategy.

Target architectures which we consider:

- **Stack machines** — *e.g.*, JVM, WebAssembly, ZPU
  - No general-purpose registers
  - Almost all registers are dedicated to program execution
- **Register machines** — *e.g.*, x86, RISC-V, many others
  - Several general-purpose registers
- **Accumulator machines** — *e.g.*, ENIAC, IBM-701, PDP-8
  - A general purpose register which is the main (implicit) operand of most instructions

All of these have a bank of *random access memory* which we can use for static and dynamic memory, but motivates using it in different ways.

Getting ready to tackle code generation

This lecture sets up the ideas needed to approach code generation on a number of architectures.

Getting ready to tackle code generation

This lecture sets up the ideas needed to approach code generation on a number of architectures.

In the next few lectures, we consider what we need to carry out Task 2 of the coursework:

1. RISC-V assembly code;
2. Code generation for stack machines;
3. Code generation for register machines / accumulator machines (optional).