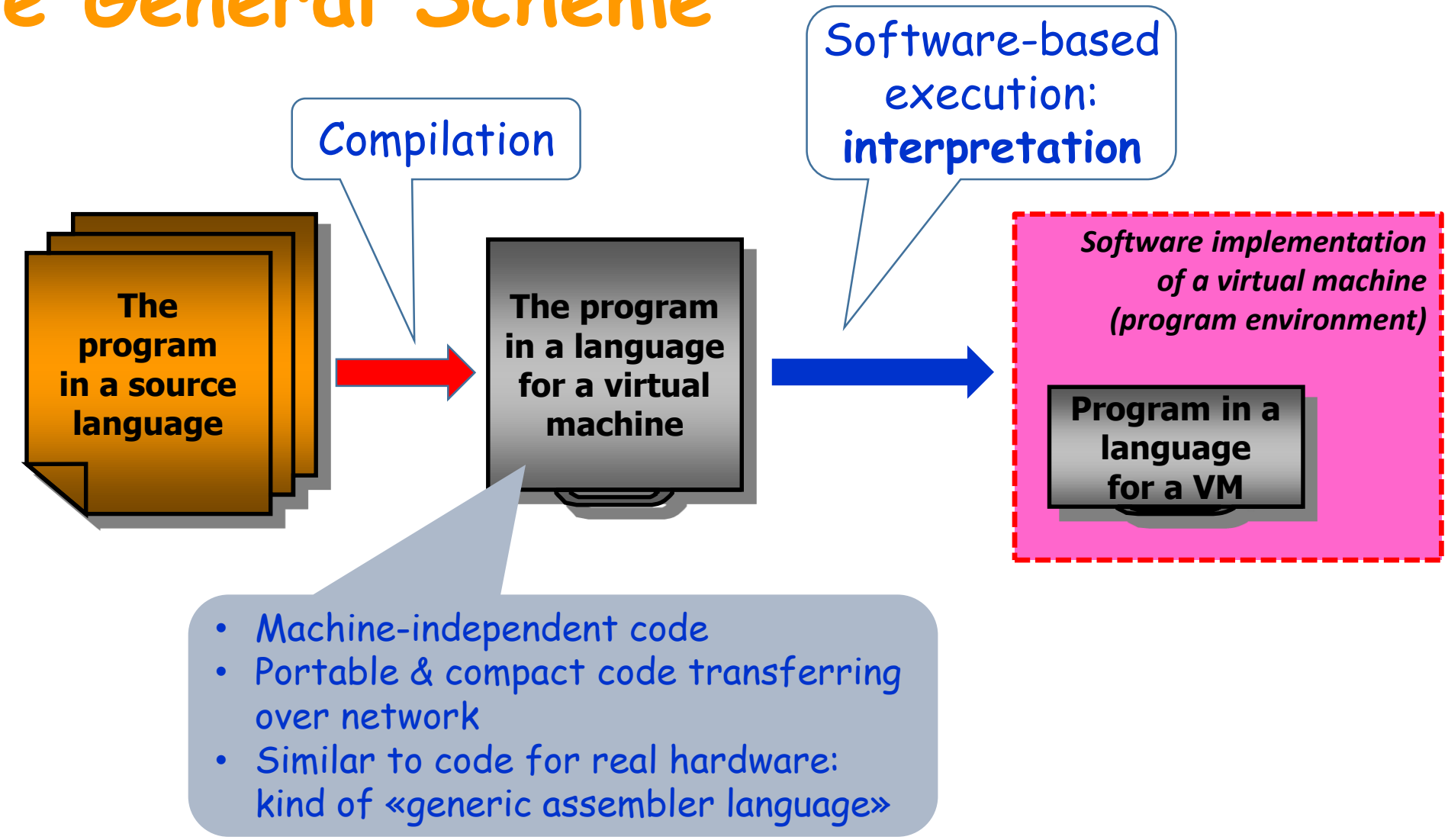
Compiler Construction: Practical Introduction

Lecture 10
Bytecode Interpretation Techinques

Eugene Zouev

Spring Semester 2023
Innopolis University

Compilation & Execution: The General Scheme



The Common Memory Model

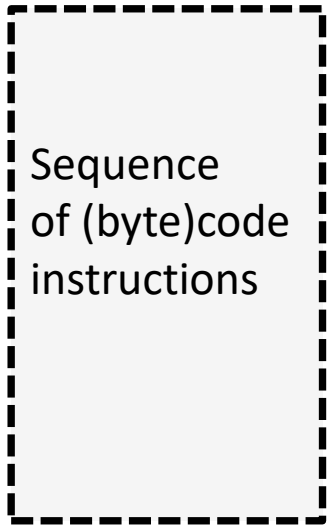
Conceptual View

Each program uses three kinds of memory:

- **Program**
- **Dynamic memory** ("Heap")
- **Stack**

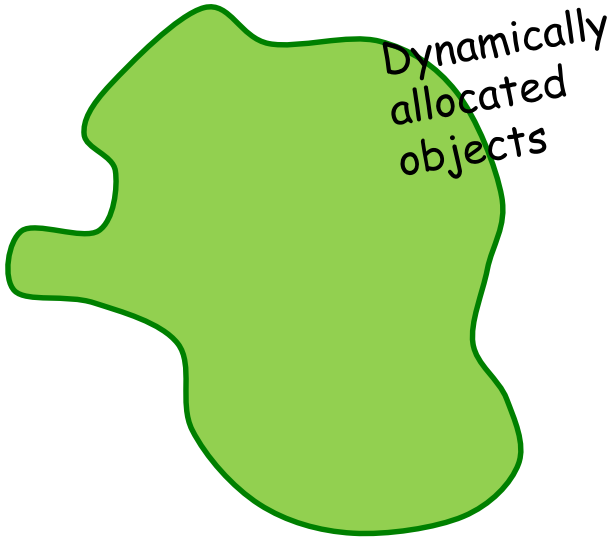
Both for hardware and for a virtual machine

Program

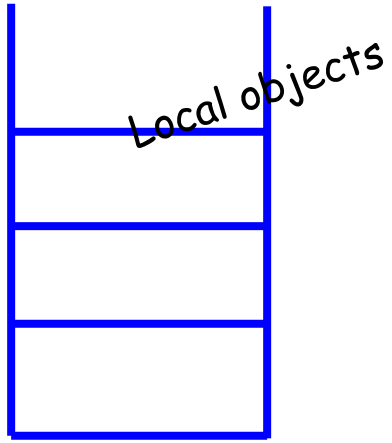


Program cannot modify this memory: self-modified programs are not allowed

Heap



The discipline of using heap is defined by program **dynamic semantics**, i.e., at runtime (while program execution)



The discipline of using stack is defined by the (static) **program structure**

The Stack

LIFO memory: "Last in –First out"

- The most VM implementations are stack-based.
- This means that most operations are performed on top of the stack.

=> The majority of VM bytecodes work on stack.

Classic stack: three actions:

- Put a value on top of stack ("**push**")
- Remove a value from top of stack ("**pop**")
- Check if stack is **empty**

The Stack: How It Goes

Schematic example

Program

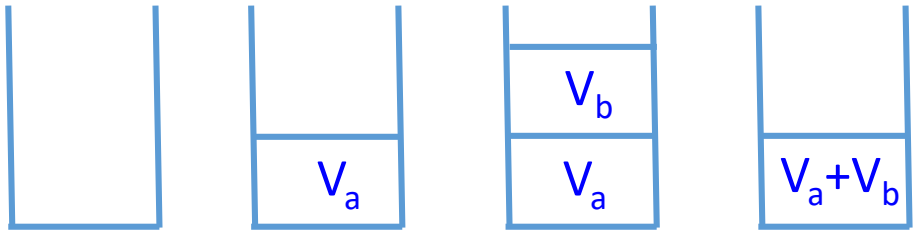
```
...
a + b
...
```

Bytecode instructions

```
...
LOAD a
LOAD b
PLUS
...
```

What does VM do

- Load the value from memory by address a to the top of the stack
- Load the value from memory by address b to the top of the stack
- Take two values from top of the stack.
- **Apply addition operator on them.**
- Remove two topmost values from the stack.
- Put the result of the addition to the top of the stack.



The Stack: How It's Implemented

Schematic example

```
class VMStack
{
  private Array<Object> stack;
  private int top;

  public void Push(Object v) {
    stack[top++] = v;
  }

  public Object Pop {
    return stack[--top];
  }

  public VMStack() {
    stack = new Array<Object>(100);
    top = 0;
}
```

```
var stack = new VMStack();
var memory = new Memory();
...
stack.Push(memory[a]);
stack.Push(memory[b]);

var tmp1 = stack.Pop();
var tmp2 = stack.Pop();
stack.Push(tmp1+tmp2);
```

LOAD a

LOAD b

PLUS

a, b are addresses in memory

To be more precise:

```
stack.Push((T)tmp1+(T)tmp2);
```

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

Stack is for Functions

The rules

- Every time a function is called, **a new frame is allocated** on the stack.

  Activation record, or **Stackframe**

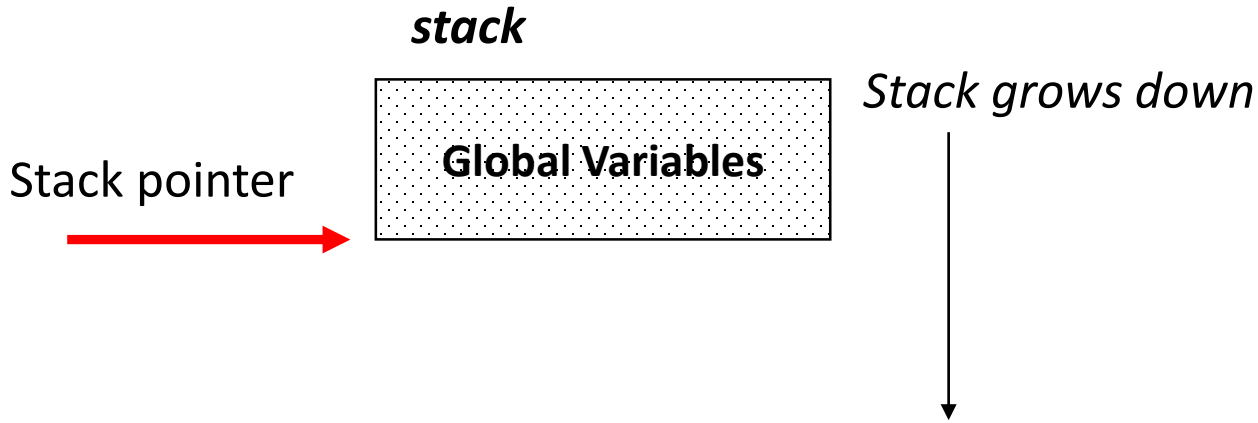
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables

- Stack frames are adjacent blocks of memory; **stack pointer** indicates the start of the stack frame.

- When function ends, the stack frame is popped off the stack; frees memory for future stack frames.

Stack is for Functions

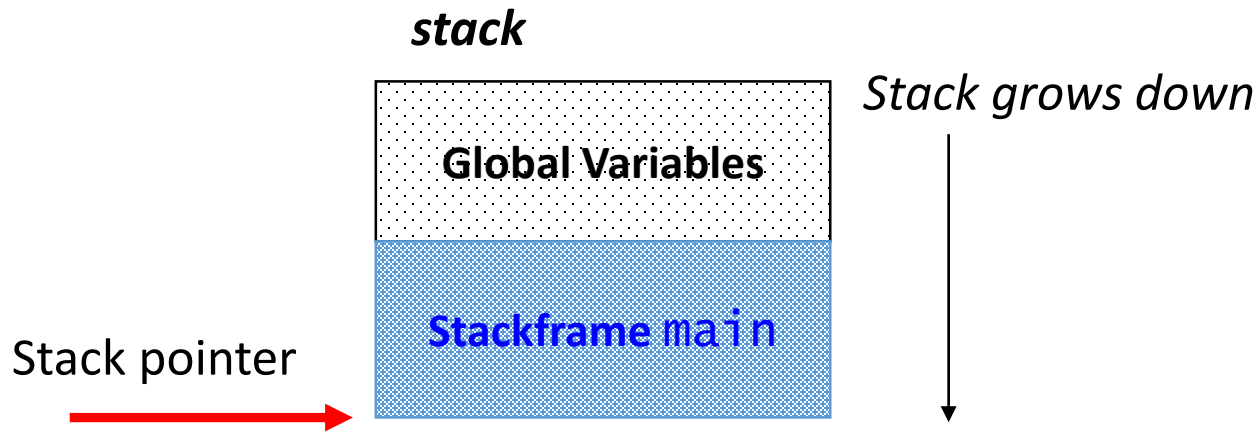
```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```



Stack is for Functions

```
int main()
{
   a();
}
void a (int m)
{
   b(1);
}
void b (int n)
{
   c(2);
}
void c (int o)
{
   d(3);
}
void d (int p)
{
}
```

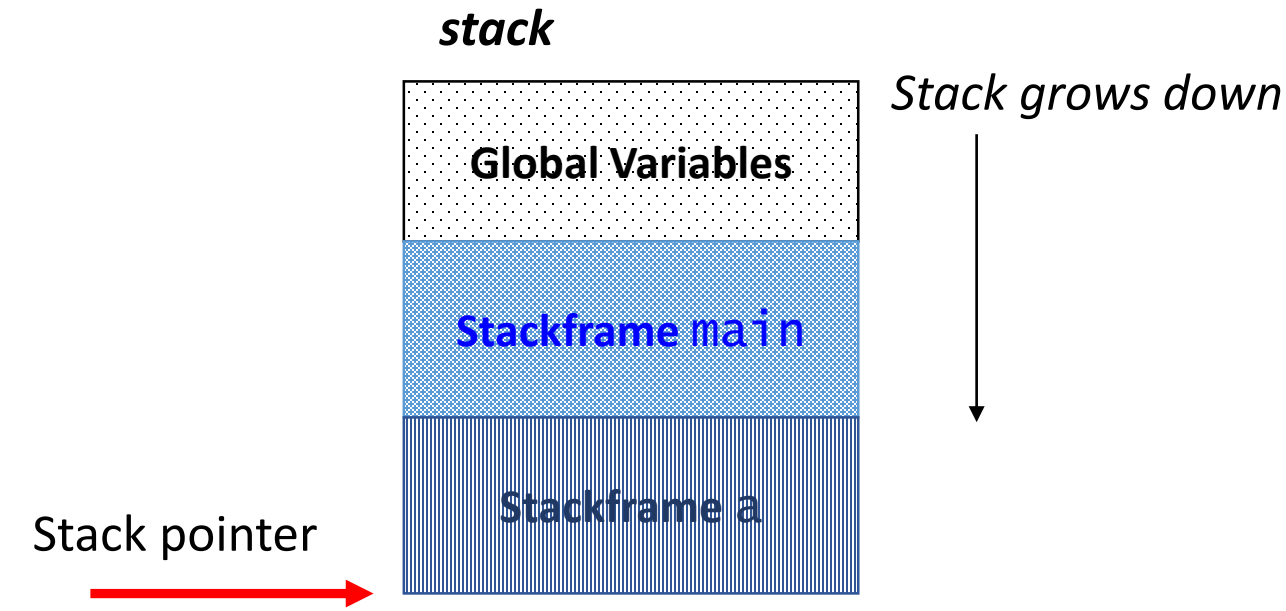
Call chain



Stack is for Functions

```
int main()
{
   a();
}
void a (int m)
{
   b(1);
}
void b (int n)
{
   c(2);
}
void c (int o)
{
   d(3);
}
void d (int p)
{
}
```

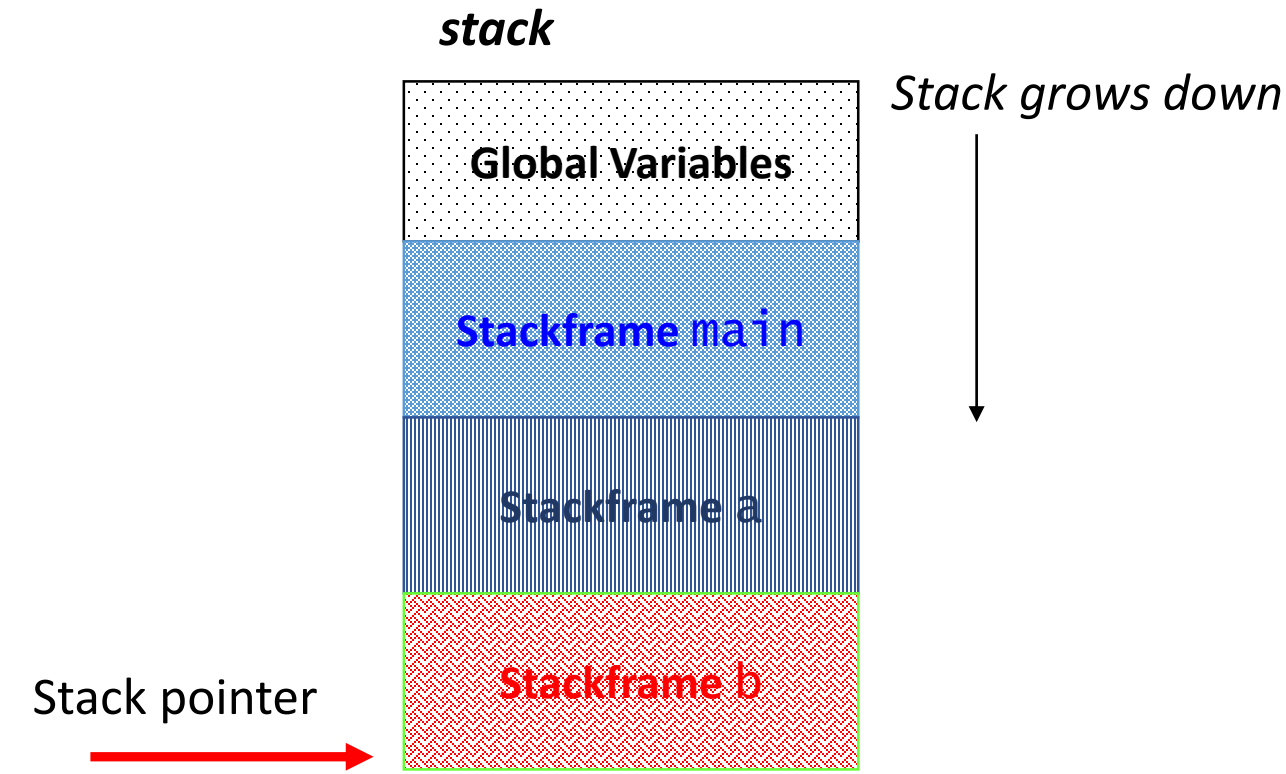
Call chain



Stack is for Functions

```
int main()
{
   a();
}
void a (int m)
{
   b(1);
}
void b (int n)
{
   c(2);
}
void c (int o)
{
   d(3);
}
void d (int p)
{
}
```

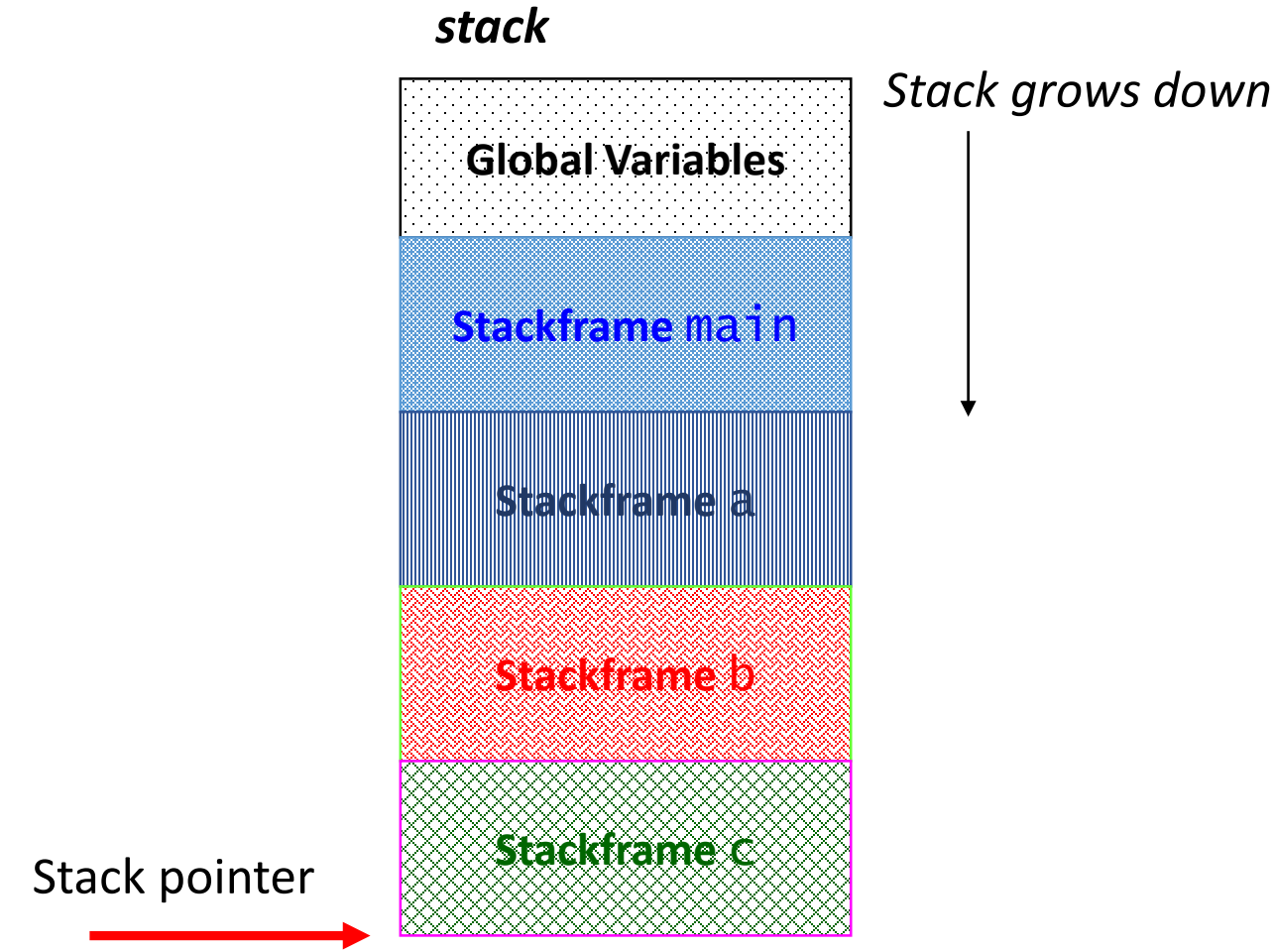
Call chain



Stack is for Functions

```
int main()
{
   a();
}
void a (int m)
{
   b(1);
}
void b (int n)
{
   c(2);
}
void c (int o)
{
   d(3);
}
void d (int p)
{
}
```

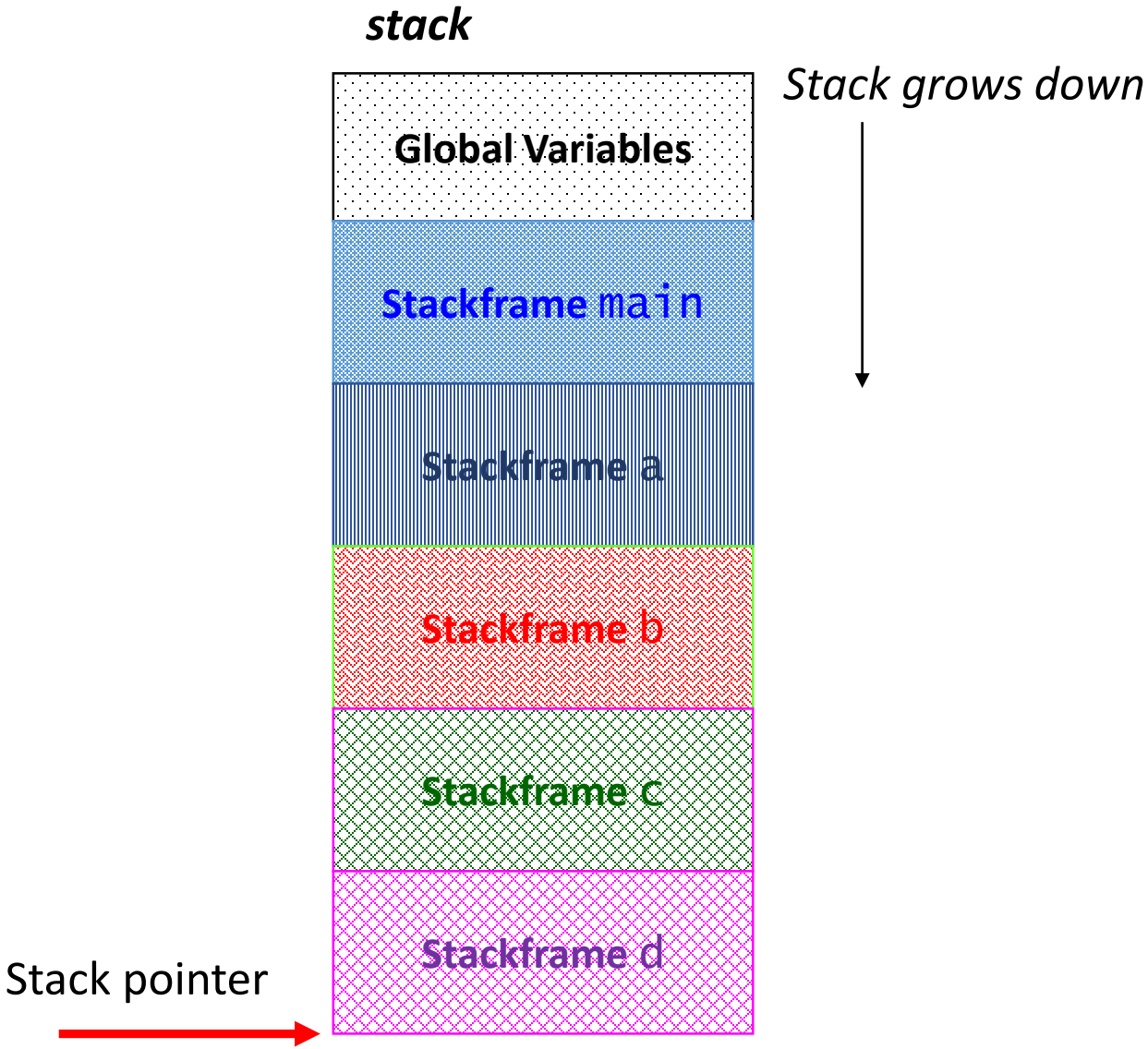
Call chain



Stack is for Functions

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

Call chain

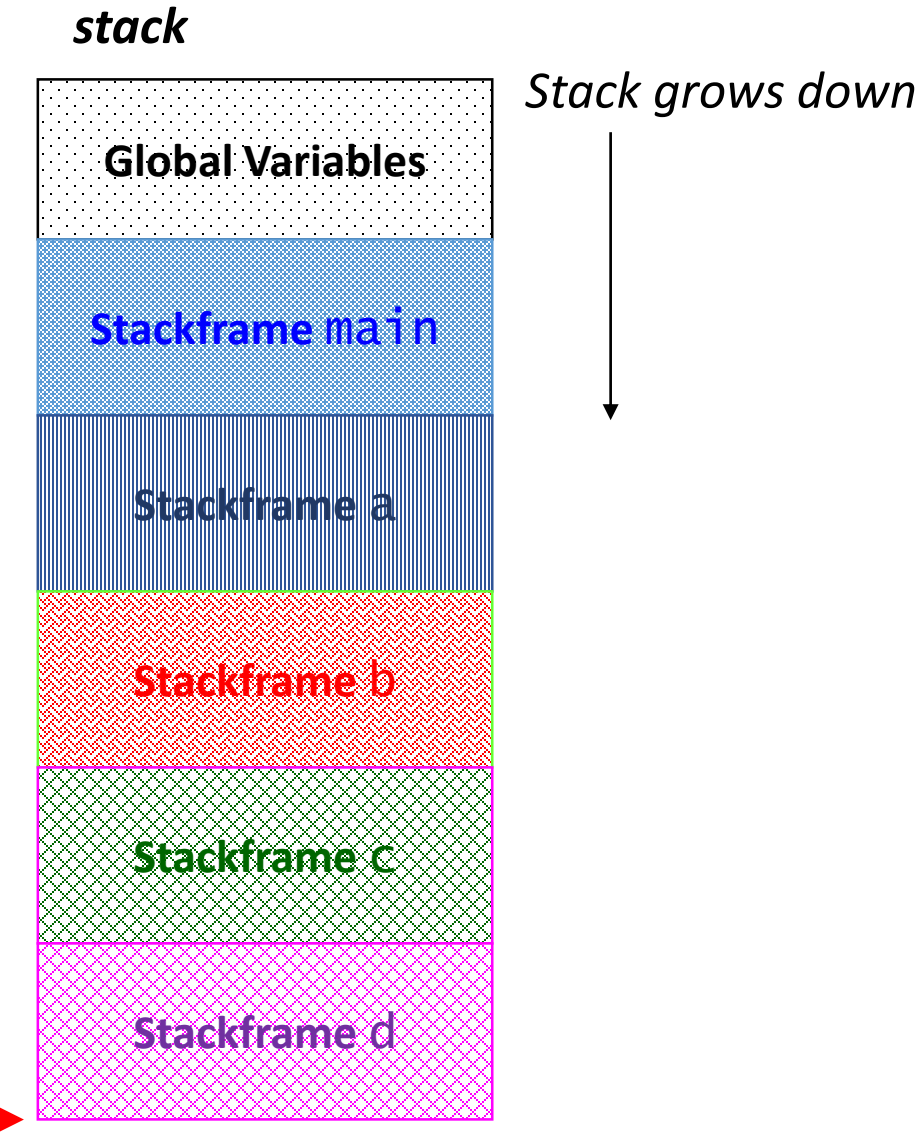


Stack is for Functions

```
int main()
{
    a();
}
void a (int m)
{
    b(1);
}
void b (int n)
{
    c(2);
}
void c (int o)
{
    d(3);
}
void d (int p)
{
}
```

Return chain

Call chain

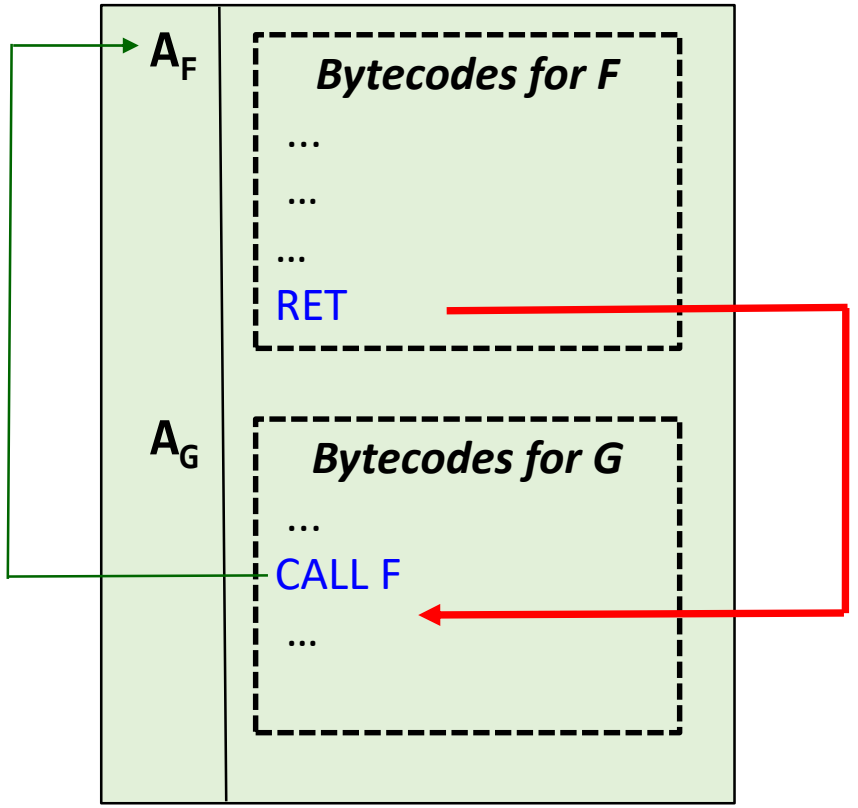


Stackframe: What Is It For?

Suppose a function is called.
What we need to know about the function?

- **Return address** (who called me?)
- …

```
int F(int a,int b)
{
  int c = 7;
  int x = (a-b)*(a+c);
  return x;
}

void G() {
  ...
  int x = F(1,99);
  ...
}
```

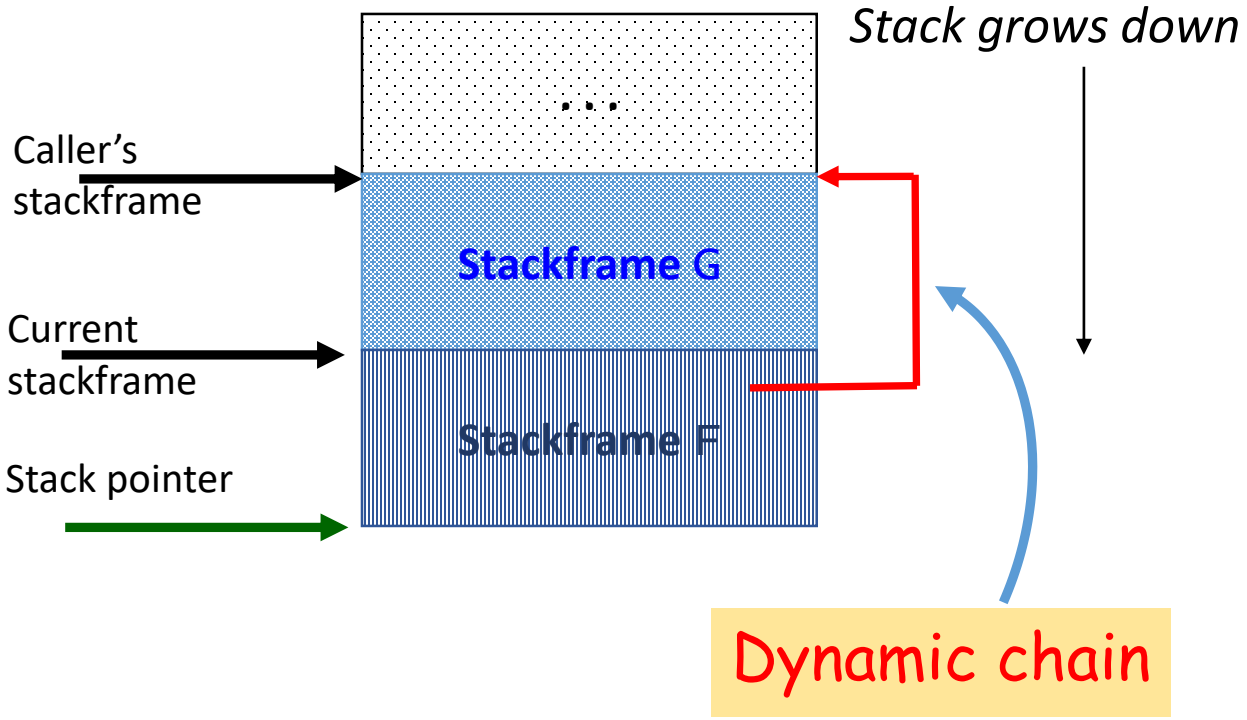


Stackframe: What Is It For?

Suppose a function is called.
What we need to know about the function?

- **Return address** (who called me?)
- **Stackframe of the caller**

```
int F(int a,int b)
{
   int c = 7;
   int x = (a-b)*(a+c);
   return x;
}

void G() {
   ...
   int x = F(1,99);
   ...
}
```

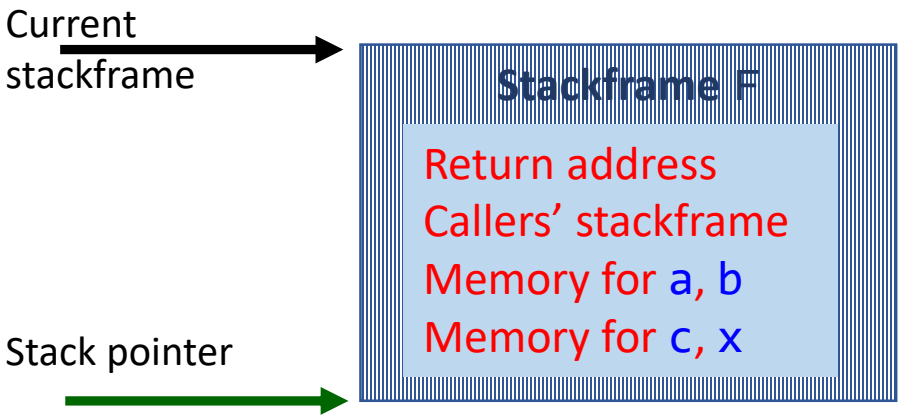


Stackframe: What Is It For?

Suppose a function is called.
What we need to know about the function?

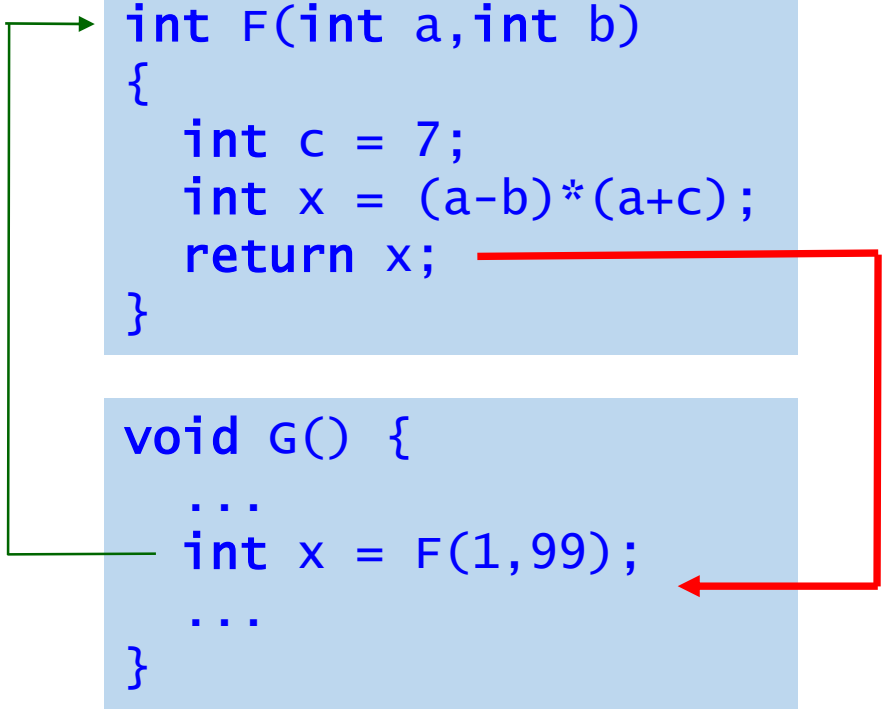
- **Return address** (who called me?)
- **Stackframe of the caller**
- ***Arguments and locals***

```
int F(int a,int b)
{
  int c = 7;
  int x = (a-b)*(a+c);
  return x;
}

void G() {
  ...
  int x = F(1,99);
  ...
}
```



Calling a Function

```
int F(int a,int b)
{
  int c = 7;
  int x = (a-b)*(a+c);
  return x;
}

void G() {
  ...
  int x = F(1,99);
  ...
}
```



FRAME

Reserves a place for the new stackframe for the function being called.

The size of the stackframe is known statically: memory for ret.addres + memory for dynamic chain + N of arguments + number of locals

LOADs

Bytecodes load **values of arguments** to the top of stack (i.e., to the newly created stackframe).

LOADs

Bytecodes initialize local variables of the function.

CALL

Stores the return address (the address of the bytecode immediately followed this one) in the stackframe, and transfers the control to the function body.

At least two improvements possible:

- VCALL for calling virtual functions
- LCALL for calling functions by addresses (for lambda functiuons)

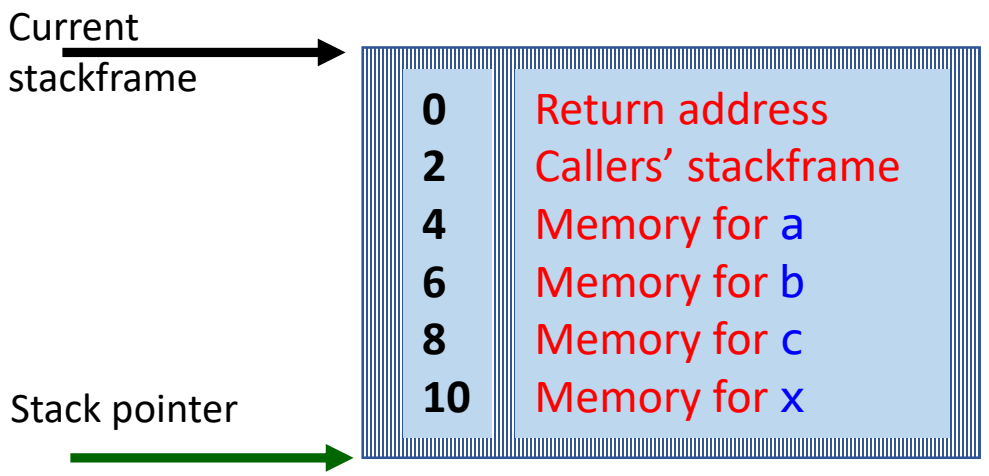
Loading locals

```
int F(int a,int b)
{
   int c = 7;
   int x = (a-b)*(a+c);
   return x;
}
```

LOAD_LOCAL *offset*

Loads the value of a local variable or an argument

The offset is known statically.



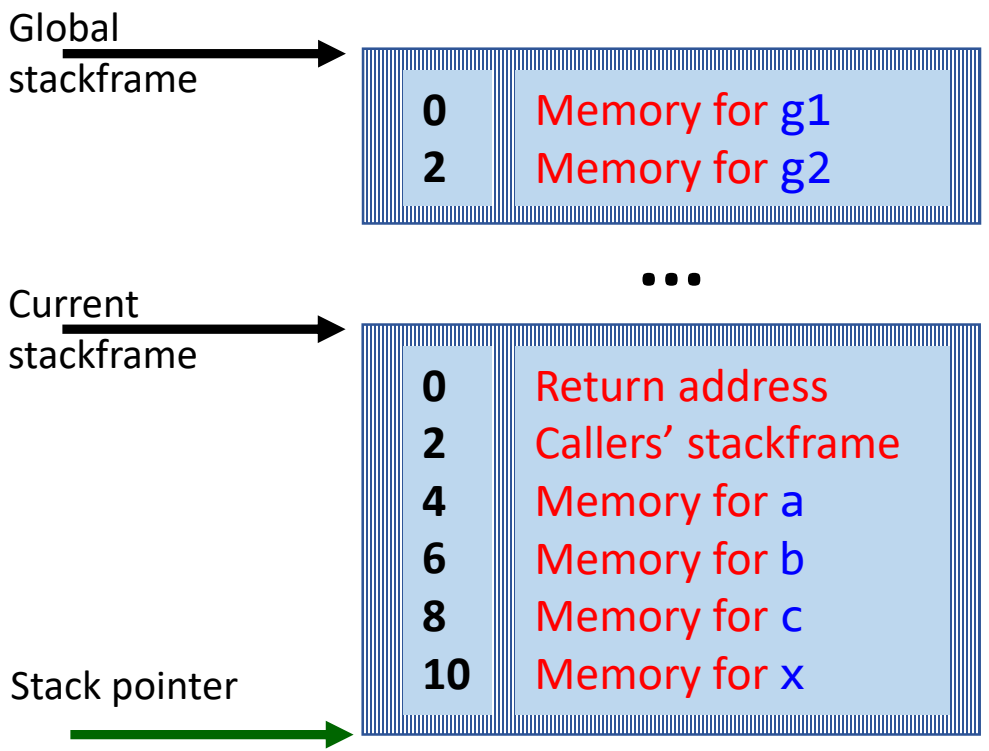
Loading globals

```
int g1, g2;
...
int F(int a,int b)
{
  int c = 7;
  int x = (a-g1)*(b+g2);
  return x;
}
```

LOAD_GLOBAL *offset*

Loads the value of a global variable

Globals are in the topmost stackframe, and their offsets are known statically.



Loading Constants

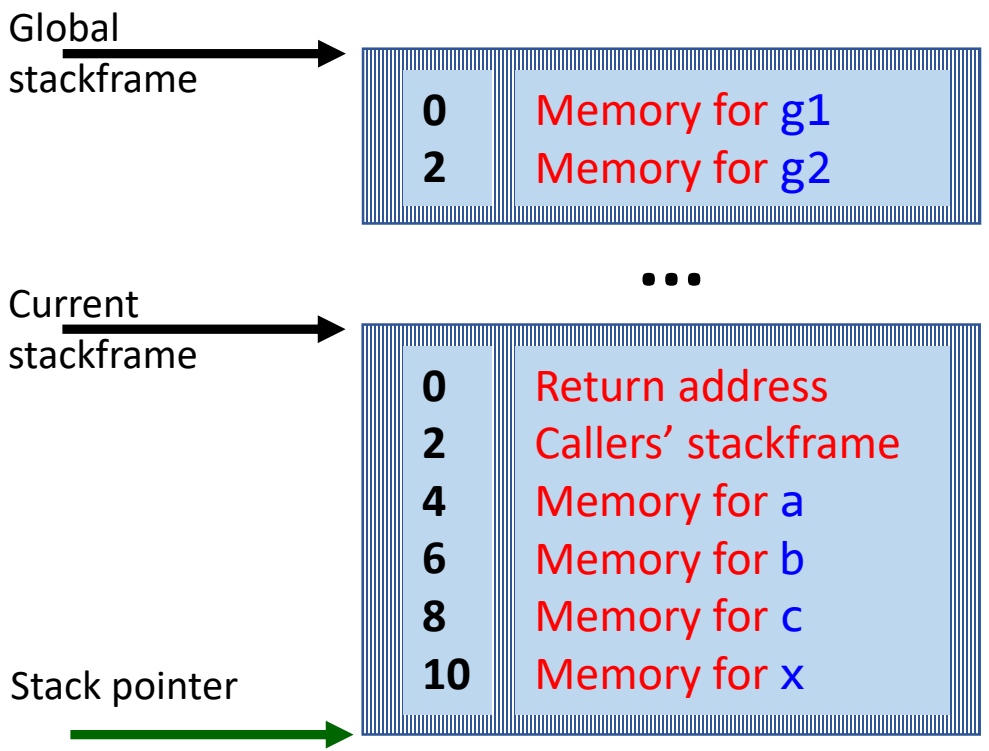
```
int g1, g2;
...
int F(int a,int b)
{
  int c = 7;
  int x = (a-5)*g2;
  return x;
}
```

```
LOAD_LOCAL 4
LOAD_CONST 5
MINUS
LOAD_GLOBAL 2
MULT
```

LOAD_CONST *constant*

Loads the value from the bytecode

Constant is statically specified directly as a bytecode parameter.



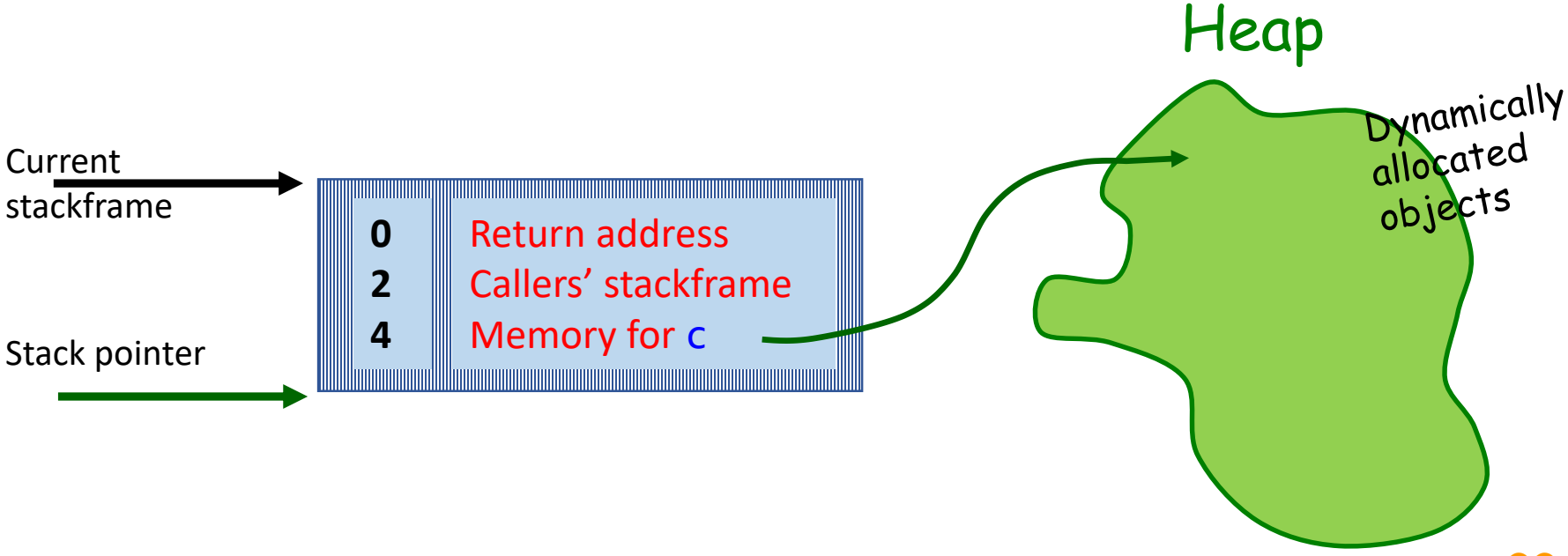
Loading from memory

```
int F(int a,int b)
{
  int* c = new int(1);
  ...
  return *c;
}
```

```
LOAD_BY_ADDR 4
RET
```

LOAD_BY_ADDR *offset*

- Takes the value of a local variable.
- Treating the value as an address, get **the value by that address**.



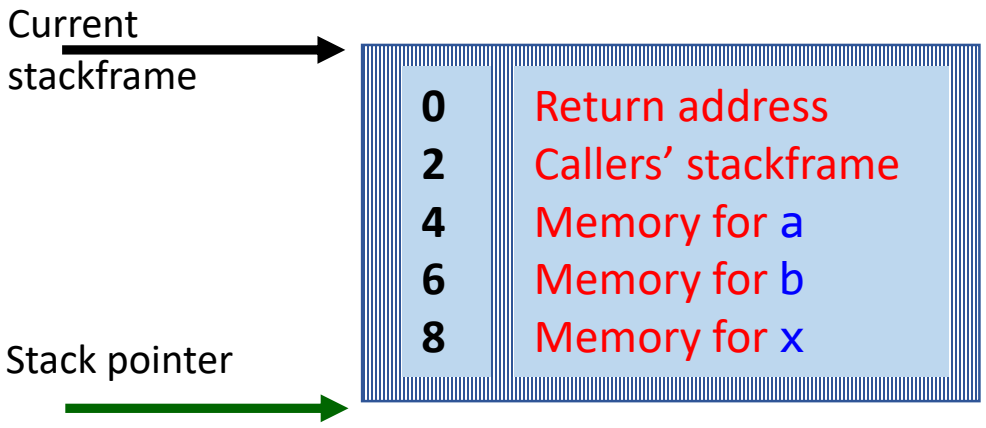
Storing to memory

```
void F(int a,int b)
{
   int x;
   ...
   x = a + b;
}
```

```
LOAD_LOCAL 4
LOAD_LOCAL 6
PLUS
STORE_LOCAL 8
```

STORE_LOCAL *offset*

- Takes the value from the top of stack.
- Stores it to the local variable with the given offset.
- Removes the value from the stack.



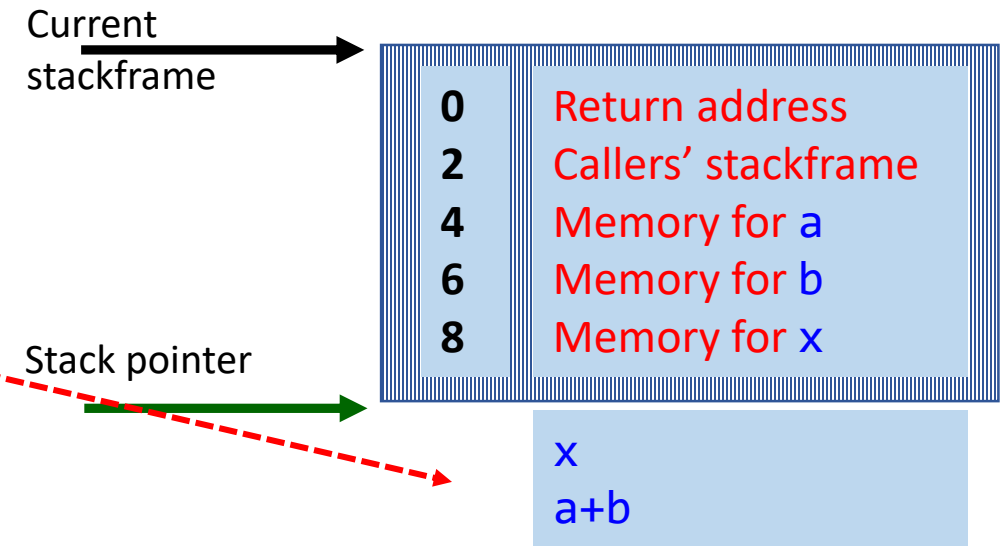
Storing to memory

```
void F(int a,int b)
{
   int* x = new int(1);
   ...
   *x = a + b;
}
```

```
LOAD_LOCAL 8
LOAD_LOCAL 4
LOAD_LOCAL 6
PLUS
STORE_BY_ADDR
```

STORE_BY_ADDR

- Takes the value from the top of stack.
- Takes the value underneath the topmost one treating it as an address.
- Stores the value by the address.
- Removes both values from the stack.



Arithmetics

PLUS
MINUS
MULT
DIVIDE
REM

- Take two values **from the top of stack.**
- **Perform operation.**
- Remove values from the stack.
- Load the result of operation to the stack.

Reality:

I_PLUS
F_PLUS

I_MINUS
F_MINUS

I_MULT
F_MULT

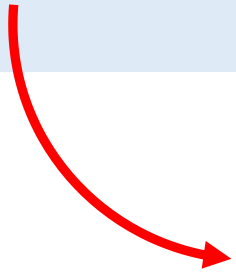
...

Similarly:

LOAD_I_LOCAL *offset*
LOAD_F_LOCAL *offset*
STORE_I_LOCAL *offset*
STORE_F_LOCAL *offset*
...

Comparisons & Jumps

```
void F(int a,int b)
{
  if (a > b )
    Statement1
  else
    Statement2
}
```



```
A1
A2
A3
A4

...

AN
...

AM
```

```
LOAD_LOCAL 4
LOAD_LOCAL 6
COMP_LESS
JUMP_IF_FALSE AN

  Code for Statement1

JUMP AM

  Code for Statement2

...
```

```
COMP_LESS offset
COMP_GREATER offset
COMP_EQUAL offset
COMP_NONEQ offset
```

- Take two values **from the top of stack**.
- **Perform comparison**.
- Remove values from the stack.
- Load the result of comparison (0 or 1) to the stack.

```
JUMP_IF_FALSE offset
JUMP_IF_TRUE offset
```

- If the value on top of the stack is 0 (1) then transfer control to the bytecode with the given offset.
- Remove the value from the stack.

```
JUMP offset
```

- Unconditionally transfers control