Functions in general

John Lapinskas, University of Bristol

A checklist to implement functions

We have four sub-goals for functions in Hack VM. We'll first discuss how to accomplish them in general, for any language (e.g. C), then discuss Hack VM specifically next video.

A checklist to implement functions

We have four sub-goals for functions in Hack VM. We'll first discuss how to accomplish them in general, for any language (e.g. C), then discuss Hack VM specifically next video.

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

A checklist to implement functions

We have four sub-goals for functions in Hack VM. We'll first discuss how to accomplish them in general, for any language (e.g. C), then discuss Hack VM specifically next video.

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

A checklist to implement functions

We have four sub-goals for functions in Hack VM. We'll first discuss how to accomplish them in general, for any language (e.g. C), then discuss Hack VM specifically next video.

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

A checklist to implement functions

We have four sub-goals for functions in Hack VM. We'll first discuss how to accomplish them in general, for any language (e.g. C), then discuss Hack VM specifically next video.

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

Goal 4: Static variables should be unaffected by function calls and returns.

The ubiquity of the stack

In **any** language and (almost) **any** architecture, the best way to achieve these goals will use a stack.

In Hack, the stack doesn't exist at the assembly level, only at the level of the Hack VM. This is unusual! Both ARM and x86-64 have:

- A register `ESP` which takes the role of the stack pointer `SP`.
- `push` and `pop` assembly instructions to manipulate the stack.
- `call`, `ret`, `enter` and `leave` assembly instructions to do most of what we discuss in this video.

So while these operations are slow and cumbersome in Hack assembly, with a simple `push local 5` operation translating to 10+ instructions, they are very fast in modern CPUs.

(MIPS doesn't have native `push` and `pop` commands, but does have `jal` and `jr` instructions that remove a lot of the burden of function calls.)

Goal 1: Program flow

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

We may assume we have access to a stack.

Goal 1: Program flow

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

We may assume we have access to a stack.

On call: Push the return address onto the stack, then jump to the start of the function code.

Goal 1: Program flow

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

We may assume we have access to a stack.

On call: Push the return address onto the stack, then jump to the start of the function code.

On return: Pop the return address from the stack, then jump to it.

Goal 1: Program flow

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

We may assume we have access to a stack.

On call: Push the return address onto the stack, then jump to the start of the function code.

On return: Pop the return address from the stack, then jump to it.

How do we know what addresses to jump to?

Goal 1: Program flow

Goal 1: Program flow. On function call, we should jump to the start of the function. On function return, we should jump back.

We may assume we have access to a stack.

On call: Push the return address onto the stack, then jump to the start of the function code.

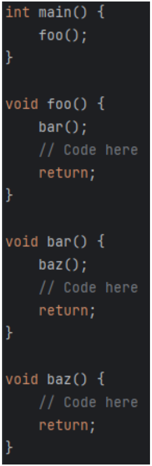
On return: Pop the return address from the stack, then jump to it.

How do we know what addresses to jump to?

We can leave it to the assembler! Labels are a part of any assembly language, not just Hack, and they already solve this problem.

Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

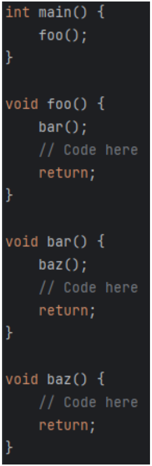
Assembly code (sketch)

```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

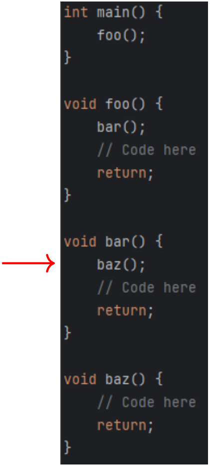
```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

(label0)

Goal 1: Program flow example

C code

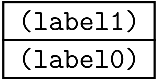


```
int main() {
    foo();
}

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

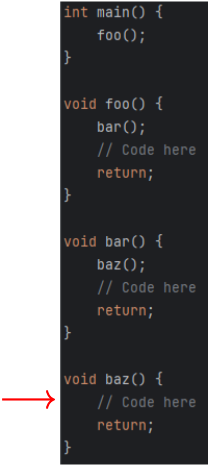
```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack



Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

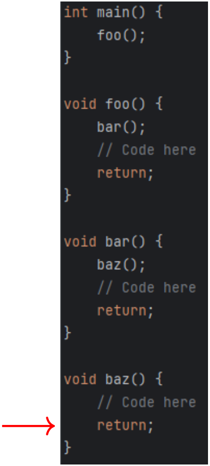
(label2)

(label1)

(label0)

Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

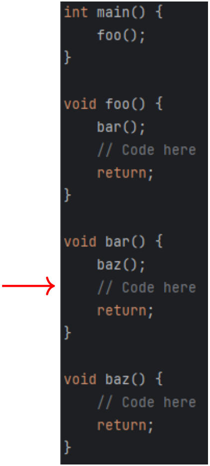
```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

(label2)
(label1)
(label0)

Goal 1: Program flow example

C code

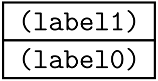


```
int main() {
    foo();
}

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

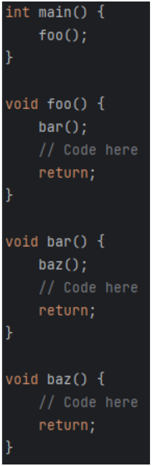


(label1)

(label0)

Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

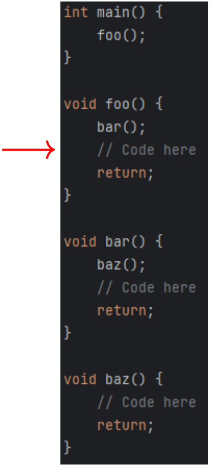
Stack

(label1)

(label0)

Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

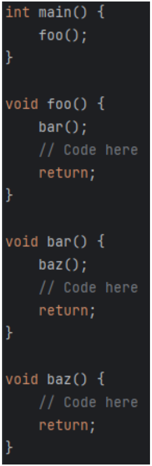
```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

`(label0)`

Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

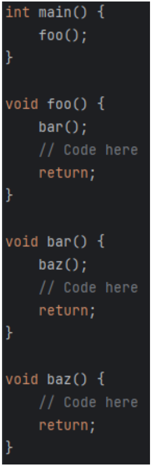
```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

`(label0)`

Goal 1: Program flow example

C code



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

void foo() {
    bar();
    // Code here
    return;
}

void bar() {
    baz();
    // Code here
    return;
}

void baz() {
    // Code here
    return;
}
```

Assembly code (sketch)

```
// Start of main code
@label0
[Push address (label0) onto stack]
[Jump to (foo)]
(label0)
// Halt

(foo)
@label1
[Push address (label1) onto stack]
[Jump to (bar)]
(label1)
// More code here
[Pop return address off stack]
[Jump to return address]

(bar)
@label2
[Push address (label2) onto stack]
[Jump to (baz)]
(label2)
// More code here
[Pop return address off stack]
[Jump to return address]

(baz)
// More code here
[Pop return address off stack]
[Jump to return address]
```

Stack

Goal 2: Memory allocation

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

Goal 2: Memory allocation

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

In C, the size of every single variable can be worked out at compile time.

Things that look like their length is decided at run-time (like strings and arrays) are really pointers, which are always 64 bits long.

The memory they point to either has size fixed at compile time (e.g. `char *out = "Hello world!";` or `int myArray[100];`) or assigned explicitly by the programmer with `malloc` and `free`.

Goal 2: Memory allocation

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

In C, the size of every single variable can be worked out at compile time.

Things that look like their length is decided at run-time (like strings and arrays) are really pointers, which are always 64 bits long.

The memory they point to either has size fixed at compile time (e.g. `char *out = "Hello world!";` or `int myArray[100];`) or assigned explicitly by the programmer with `malloc` and `free`.

A C compiler that creates a symbol table for the local variables in a function during semantic analysis (i.e. after parsing) will therefore know in advance exactly how much space to allocate every time the function is called. Ditto arguments.

So the compiler needs to:

- Find space for a known number of local/argument variables of known size each time the function is called.
- Implement `malloc` and `free`. This memory is then independent of function calls, so no further action is needed. (See later video!)

Combining goals 2 and 3

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

We know when generating assembly code exactly what variables we need to allocate.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

Combining goals 2 and 3

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

We know when generating assembly code exactly what variables we need to allocate.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

We can use the stack! Let's say a call to a function f will need 10 words of local variables, 7 words of argument variables, and 5 words of state (including relevant registers and the return address). We can have these variables stored in a symbol table.

Combining goals 2 and 3

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

We know when generating assembly code exactly what variables we need to allocate.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

We can use the stack! Let's say a call to a function f will need 10 words of local variables, 7 words of argument variables, and 5 words of state (including relevant registers and the return address). We can have these variables stored in a symbol table.

On function call:

- Store the current stack pointer in a register/memory as `OSP` (Old Stack Pointer).
- Add 10 to the stack pointer `SP` to leave room for local variables.
- Push our program state and arguments onto the stack (adding 12 to `SP`).
- Jump to the function label.

Combining goals 2 and 3

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

We know when generating assembly code exactly what variables we need to allocate.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

We can use the stack! Let's say a call to a function f will need 10 words of local variables, 7 words of argument variables, and 5 words of state (including relevant registers and the return address). We can have these variables stored in a symbol table.

During function execution:

- Say each variable takes one word of storage, and we stored arguments at the bottom, then local variables, then program state. (This doesn't really matter.)
- References to the i'th argument variable become references to RAM[$\mathtt{OSP} + i$].
- Reference to the i'th local variable become references to RAM[$\mathtt{OSP} + 7 + i$].
- We can use symbol tables to store the offset for each variable (which also handles variables of sizes other than one word).

Combining goals 2 and 3

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

We know when generating assembly code exactly what variables we need to allocate.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

We can use the stack! Let's say a call to a function f will need 10 words of local variables, 7 words of argument variables, and 5 words of state (including relevant registers and the return address). We can have these variables stored in a symbol table.

On function return:

- Optionally, store a return value.
- Reset our stack pointer `SP` back to `OSP`, effectively freeing the memory we used for the old local variables, arguments, and program state.
- Copy our old program state back into registers.
- Jump to the return address (from the stack).
- Optionally, do something with the return value.

Combining goals 2 and 3

Goal 2: Memory allocation. On function call, we should allocate memory for the new local and argument variables. On function return, we should free that memory.

We know when generating assembly code exactly what variables we need to allocate.

Goal 3: Program state. On function call, we should set aside all existing local/argument variables and most register values, and replace them with new ones. On function return, we should pick them back up unchanged.

We can use the stack! Let's say a call to a function *f* will need 10 words of local variables, 7 words of argument variables, and 5 words of state (including relevant registers and the return address). We can have these variables stored in a symbol table.

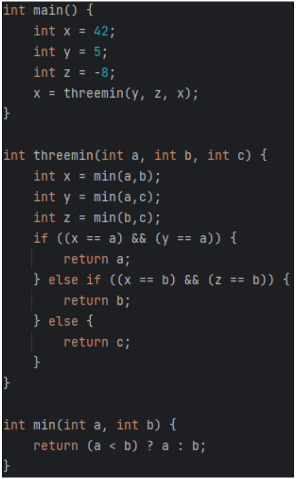
In the example on the next slide, we assume the stack starts at `0SP`, that all variables are one word long, and that the program state is 10 words long.

We also completely ignore what happens when we "call" or "return from" `main`.

Finally, note that this is a *possible* way of implementing function calls from C. The actual implementation would set up the stack slightly differently for optimisation reasons (or maybe optimise out the function calls altogether).

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

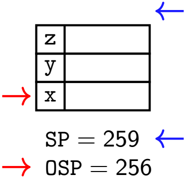
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)

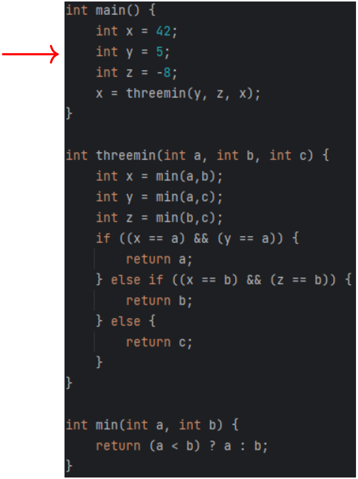


$\mathtt{SP} = 259$

$\mathtt{OSP} = 256$

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables
(stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

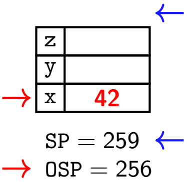
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

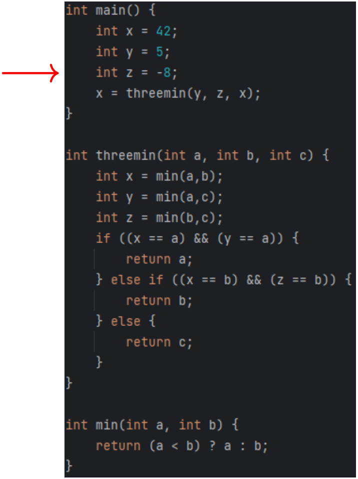
Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored
by program)



Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

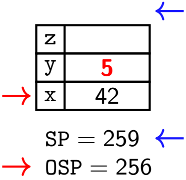
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

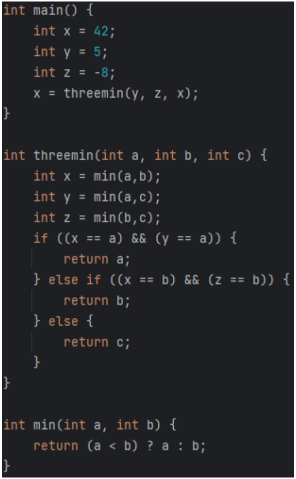
Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

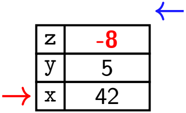
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)

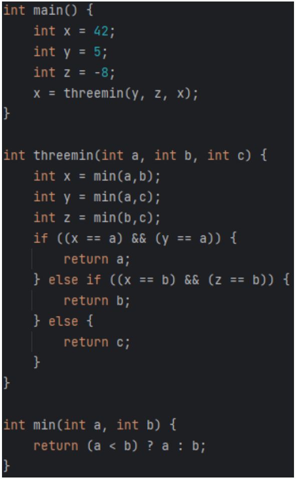


SP = 259

OSP = 256

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables
(stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

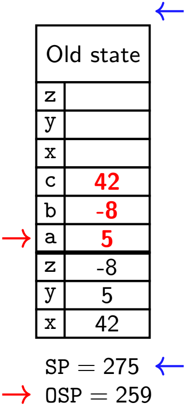
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored
by program)

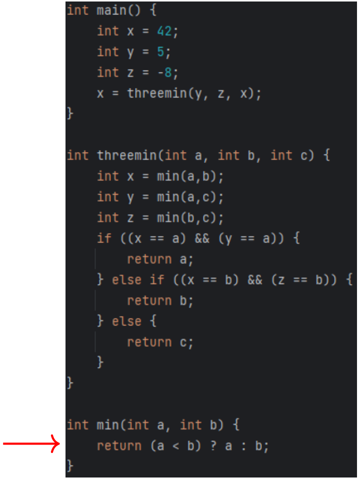


Old state
z
y
x
c **42**
b **-8**
a **5**
z -8
y 5
x 42

SP = 275
OSP = 259

Extended example

C code



```c
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

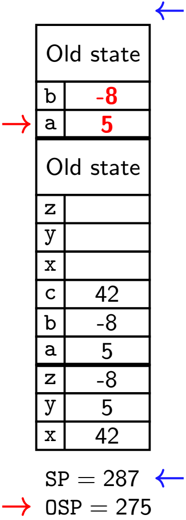
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



Old state

b **-8**

a **5**

Old state

z

y

x

c 42

b -8

a 5

z -8

y 5

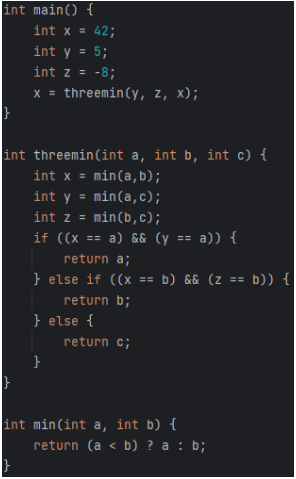
x 42

SP = 287

OSP = 275

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

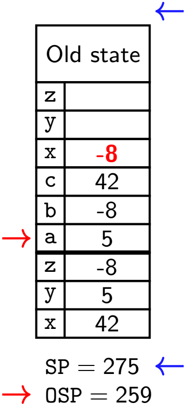
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



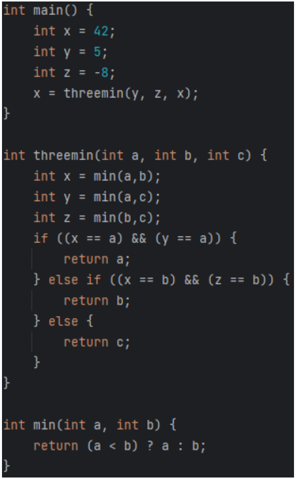
Old state	
z	
y	
x	**-8**
c	42
b	-8
a	5
z	-8
y	5
x	42

SP = 275

OSP = 259

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

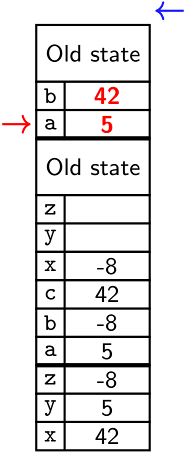
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



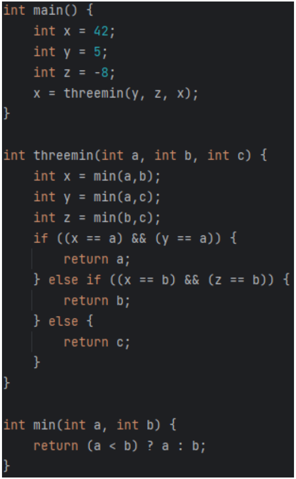
Old state	
b	**42**
a	**5**
Old state	
z	
y	
x	-8
c	42
b	-8
a	5
z	-8
y	5
x	42

SP = 287

OSP = 275

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

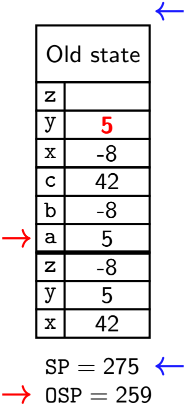
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



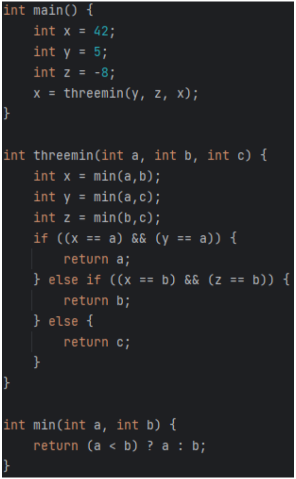
Old state	
z	
y	**5**
x	-8
c	42
b	-8
a	5
z	-8
y	5
x	42

`SP` = 275

`OSP` = 259

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

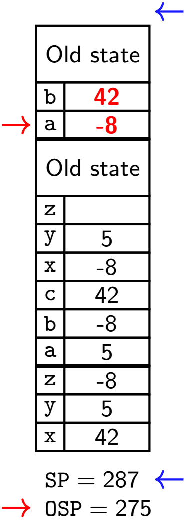
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



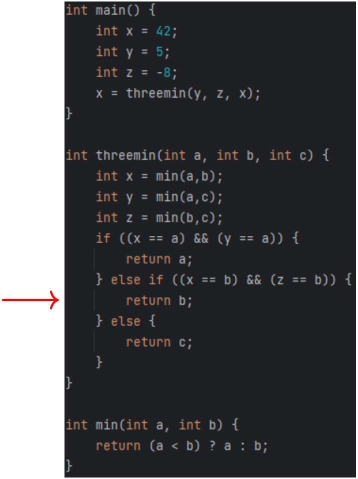
Old state	
b	**42**
a	**-8**
Old state	
z	
y	5
x	-8
c	42
b	-8
a	5
z	-8
y	5
x	42

SP = 287

OSP = 275

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

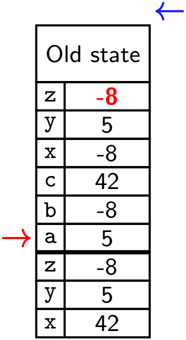
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



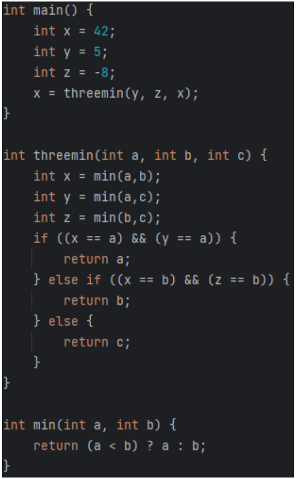
Old state	
z	**-8**
y	5
x	-8
c	42
b	-8
a	5
z	-8
y	5
x	42

SP = 275 ←

→ OSP = 259

Extended example

C code



```
int main() {
    int x = 42;
    int y = 5;
    int z = -8;
    x = threemin(y, z, x);
}

int threemin(int a, int b, int c) {
    int x = min(a,b);
    int y = min(a,c);
    int z = min(b,c);
    if ((x == a) && (y == a)) {
        return a;
    } else if ((x == b) && (z == b)) {
        return b;
    } else {
        return c;
    }
}

int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Symbol tables (stored by compiler)

`main`

Name	Type	Offset
x	Local, `int`	0
y	Local, `int`	1
z	Local, `int`	2

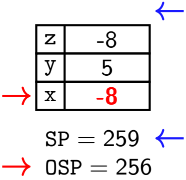
`threemin`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1
c	Argument, `int`	2
x	Local, `int`	3
y	Local, `int`	4
z	Local, `int`	5

`min`

Name	Type	Offset
a	Argument, `int`	0
b	Argument, `int`	1

Stack (stored by program)



z	-8
y	5
x	**-8**

SP = 259

OSP = 256

Terminology

- We call the parts of the program state that need to be pushed onto the stack (e.g. the return address, the old `OSP` value, old register values) the **call frame** or just **frame** of the calling function.

  Last slide, we just called this “Old state”.

Terminology

- We call the parts of the program state that need to be pushed onto the stack (e.g. the return address, the old `OSP` value, old register values) the **call frame** or just **frame** of the calling function.

  Last slide, we just called this "Old state".

- The program can still use the very top of the stack for working storage for arithmetic operations while inside a function.

  We call this sub-stack the **working stack**, and the entire stack (including all past call frames) the **global stack**.

  During a function call, the working stack will be preserved along with the rest of the old state. (It's not part of the call frame, though.)

Terminology

- We call the parts of the program state that need to be pushed onto the stack (e.g. the return address, the old `OSP` value, old register values) the **call frame** or just **frame** of the calling function.

  Last slide, we just called this “Old state”.

- The program can still use the very top of the stack for working storage for arithmetic operations while inside a function.

  We call this sub-stack the **working stack**, and the entire stack (including all past call frames) the **global stack**.

  During a function call, the working stack will be preserved along with the rest of the old state. (It’s not part of the call frame, though.)

- Memory allocated by `malloc` is often said to be allocated **on the heap**, to distinguish it from memory allocated on the stack.

  Surprisingly, there’s no relation at all to a heap data structure!
  It’s just a phrase with no deeper meaning.

Goal 4: Static variables

Goal 4: Static variables should be unaffected by function calls and returns.

Goal 4: Static variables

Goal 4: Static variables should be unaffected by function calls and returns.

This is easy — we just assign each static variable its own area of memory separate from the stack (in Hack VM this is the `static` segment).

We have a symbol table mapping each variable name to its memory.

Last, we refrain from messing with it in the function call/return process.

Goal 4: Static variables

Goal 4: Static variables should be unaffected by function calls and returns.

This is easy — we just assign each static variable its own area of memory separate from the stack (in Hack VM this is the `static` segment).

We have a symbol table mapping each variable name to its memory.

Last, we refrain from messing with it in the function call/return process.

Likewise, we don't have to do anything special to account for any working storage on the stack used for e.g. arithmetic operations — this will naturally be preserved on function call and restored on return.

Goal 4: Static variables

Goal 4: Static variables should be unaffected by function calls and returns.

This is easy — we just assign each static variable its own area of memory separate from the stack (in Hack VM this is the `static` segment).

We have a symbol table mapping each variable name to its memory.

Last, we refrain from messing with it in the function call/return process.

Likewise, we don't have to do anything special to account for any working storage on the stack used for e.g. arithmetic operations — this will naturally be preserved on function call and restored on return.

Non-C languages often blur the line between stack and heap on the surface, e.g. allowing the programmer to define variable-length arrays or freeing memory automatically.

But under the hood, they generally work like C — they store some variables on the stack and others on the heap, and call analogues of `malloc` and `free` to manage heap memory.