CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework:
Basic Blocks

An Algorithm

Two passes:

- Pass One: Find all the leaders
  - Leaders are born (entry point or command line arguments)
  - Leaders are made (target of jump, call, or branch, or fall-through from branch)
- Pass Two: Print the basic blocks
  - Sort the leaders
  - Print the basic block
  - A basic block ends when you hit a return, branch, jump, or halt, or when you hit the start of another basic block

Issues?

- Using RAD?
- Finding leaders?
- Printing basic blocks? In order?

More Registers and Arguments on the Stack

Register Aliases

Setting the 32-bit view *clears* the top 32 bits. Setting the 16- or 8-bit views does *not*.

There is no equivalent to AH, BH, CH, and DH for the other registers.

64-bit Register	Lowest 32 Bits	Lowest 16 Bits	Lowest 8 Bits
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Stack Arguments

Sometimes it makes sense to pass an argument on the stack. The trick is correctly referencing the argument to recover it.

- Push *arg*
  `[RSP]` = *arg*
- Call function
  `[RSP]` = `RIP`, `[RSP+8]` = *arg*
- `push rbp`
  `[RSP]` = `RBP`, `[RSP+8]` = old `RIP`, `[RSP+16]` = *arg*
- `mov rbp, rsp`
  `[RBP+16]` = *arg*

```
; This function takes a single argument pushed
; onto the stack prior to calling.  It is
; retrieved from [rbp+16].

print:
        push rbp
        mov rbp, rsp
        mov rax, 0
        mov rdi, form
        mov rsi, [rbp+16]
        call printf wrt ..plt
        leave
        ret
```

What's on the Stack?

Tiny routine to print the stack, assuming the C library. Can omit the function stuff to simplify.

```
Address                     ebp           Content
0x00007ffeda3b1168          [ebp+56]
              0x0000556f997091c0
0x00007ffeda3b1160          [ebp+48]
              0x00000001f84b9e88
0x00007ffeda3b1158          [ebp+40]
              0x00007ffeda3b1228
0x00007ffeda3b1150          [ebp+32]
              0x00007f10f84eb598
0x00007ffeda3b1148          [ebp+24]
              0x00007f10f832b1e3
0x00007ffeda3b1140          [ebp+16]
```

```
show_stack:
        push rbp
        mov rbp, rsp
        mov rax, 0
        mov rdi, head
        call printf wrt ..plt
        mov rcx, 8
.top:
        mov r15, rcx
        mov rax, 0
        mov rdi, stack
        lea rsi, [rbp+r15*8-8]
        lea rdx, [r15*8-8]
        mov rcx, [rbp+r15*8-8]
        call printf wrt ..plt
        mov rcx, r15
        loop .top
        leave
        ret

head:   db "Address             ",9
        db "ebp      ",9
        db "Content",10,0
stack:  db "0x%016lx ",9
        db "[ebp+%d] ",9
        db "0x%016lx",10,0
```

PIC, PIE, PLT, GOT, and RDI-relative

Position Independent Executable (PIE)

Some instructions require information about the location of resources, including libraries, strings, and instructions.

```
call puts                     mov rsi, [str]                         jmp build
```

This makes it hard to use techniques like address space layout randomization (ASLR), a defensive programming technique. It also makes it hard to write shared libraries that are dynamically loaded, and which might end up anywhere in memory. It's *annoying*. You can't load two programs into the same address space because they might not be address compatible.

Ideally you would write position independent code (PIC).

-no-pie and -static

We've been telling GCC that we are not writing position independent code with `-no-pie` (do not try to create a position independent executable) and `-static` (include libraries we need in the executable so they have fixed addresses).

This is *not practical* in general, because now *every* executable has to have a copy of the libraries. We really want to have the libraries loaded in memory once, and then let everyone reference them. To make that work we need (among other things) to make them *location independent*.

```
$ gcc -o copy copy.o         ; ls -l copy         #  16,640 bytes
$ gcc -o copy copy.o -static ; ls -l copy         # 863,064 bytes
```

The Global Offset Table (GOT)

GOT is a section (`.got`) created during assembly that holds addresses of data and routines. It is "fixed up" during load by computing the memory offsets based on the actual addresses of the code, data, etc. Instead of referencing data or instruction addresses directly, you reference them via the GOT.

Now the only problem you have is finding the GOT. To do that you use a special symbol (`_GLOBAL_OFFSET_TABLE_`) and a hack to get its address into a register (typically `RBX`… which is now no longer useful since you have to use it to reference stuff: `lea [rbx+str]`). 32-bit code is full of this stuff, and 64-bit code still has it (`` objdump -s -j .got `which python3` ``) and you *can* use it.

If you want to read about it you can… but 64-bit code introduced something really useful: `RIP`-relative addressing!

RIP-Relative Addressing

At load time the code is scanned and offsets to locations are "fixed up" by the loader to reference data and instruction addresses relative to `RIP`. This makes the program position independent!

You write: `lea rsi, [rel str]`, the assembler gives you: `lea rsi,[rip+0x0]`, and the linker converts this into `lea rsi,[rip+0xbc465]` once it has laid out the sections, segments, etc. Now our code is position independent!

This is so useful you can just write `default rel` at the top of your NASM file and then the instruction `lea rsi, [str]` is implicitly converted to `lea rsi, [rel str]`. If you don't want that for some reason, write `lea rsi, [abs str]`.

So far, so good

Some instructions require information about the location of resources, including libraries, strings, and instructions.

```
call puts                      mov rsi, [str]                          jmp build
```

We've handled the last two cases.

```
                                         mov rsi, [rel str]            lea rax,
[rel build] ; jmp rax
```

The jump is overkill. If you are jumping to an address within a section, you can just use `jmp build` directly.

What about calls?

We still haven't addressed the library function call.

```
call puts
```

RIP-relative addressing won't save us here (unless we use `-static`), since we don't know the offset to a *dynamically-loaded* library routine.

Program Linkage Table (PLT)

The program linkage table (PLT) solves the dynamic loading and linkage problem… but it's not as simple as RIP-relative addressing.

We can read about this… but let's take a look at some disassembly.

We will use the `copy.asm` program provided with this lecture.

WRT

Here we reference the library function puts… with respect to the PLT. Magic! NASM knows how to correctly encode this.

```
extern puts
;...

call puts wrt ..plt
```

Tracing the PLT

Now we disassemble.

```
118a:       e8 a1 fe ff ff          call   1030 <puts@plt>
```

Okay. Let's have a look in the PLT. Location 1030 is clearly not the `puts` code.

```
0000000000001030 <puts@plt>:
    1030:       jmp    QWORD PTR [rip+0x2f9a]        # 3fd0 <puts@GLIBC_2.2.5>
```

Tracing the PLT

```
0000000000001030 <puts@plt>:
    1030:       jmp    QWORD PTR [rip+0x2f9a]        # 3fd0 <puts@GLIBC_2.2.5>
```

Okay... what's happening here? What is at location `rip+0x2f9a`? Well...

```
  0x1036
+ 0x2f9a
  0x3fd0
```

Tracing the PLT

```
0000000000001030 <puts@plt>:
    1030:       jmp    QWORD PTR [rip+0x2f9a]        # 3fd0 <puts@GLIBC_2.2.5>
```

The instruction jumps to the address contained in location `0x3fd0`. What is that? It is not in the .text section... or in any code section. It's in GOT!

```
$ objdump -s -j .got copy
Contents of section .got:
 3fb8 c83d0000 00000000 00000000 00000000  .=..............
 3fc8 00000000 00000000 36100000 00000000  ........6.......
 3fd8 00000000 00000000 00000000 00000000  ................
 3fe8 00000000 00000000 00000000 00000000  ................
 3ff8 00000000 00000000                    ........
```

So what happens?

The GOT is fixed up by the loader.

- PLT contains stubs to jump to the targets, and
- GOT contains a table of target addresses.

When we `call puts wrt ..plt` the stub is called and this triggers the loading of the library (if it is not already loaded). Then the GOT table is fixed to contain the addresses of the loaded routines and finally `puts` is called. The next time we call `puts` the GOT contains the correct address and we jump directly there.

How does all this work?

There is not enough time! But there is an excellent article that walks through it using gdb which you need to read. Head here:

https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html

You'll also learn a lot about using gdb. I recommend you follow along.

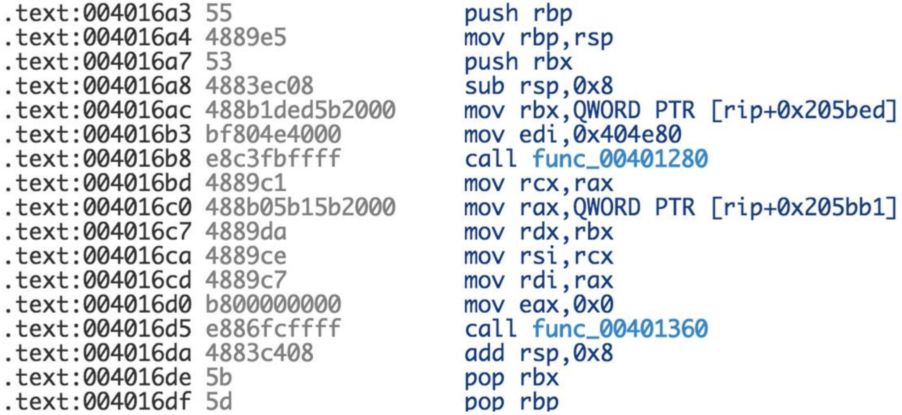
Liveness Analysis

Liveness

A variable is *live* if it contains a value that *may* be needed later in the program.

This is often obvious in high-level programs, where most variables are live, but may not be obvious in assembly.

We can work backward to find out...



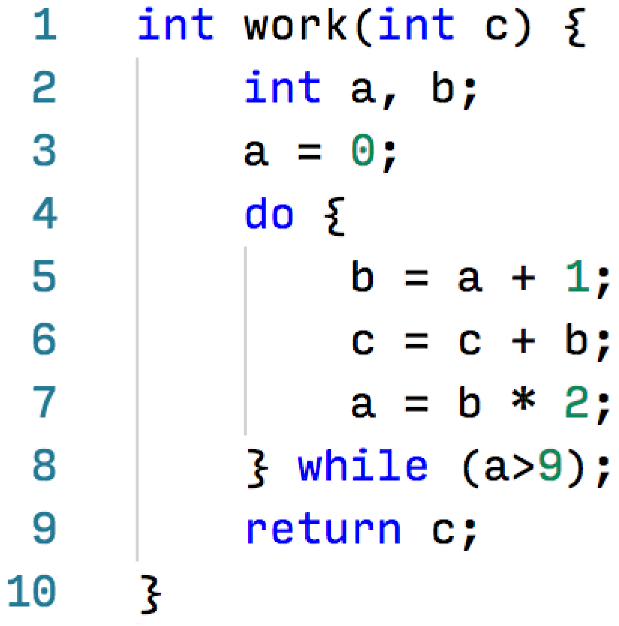
```
.text:004016a3 55                  push rbp
.text:004016a4 4889e5              mov rbp,rsp
.text:004016a7 53                  push rbx
.text:004016a8 4883ec08            sub rsp,0x8
.text:004016ac 488b1ded5b2000      mov rbx,QWORD PTR [rip+0x205bed]
.text:004016b3 bf804e4000          mov edi,0x404e80
.text:004016b8 e8c3fbffff          call func_00401280
.text:004016bd 4889c1              mov rcx,rax
.text:004016c0 488b05b15b2000      mov rax,QWORD PTR [rip+0x205bb1]
.text:004016c7 4889da              mov rdx,rbx
.text:004016ca 4889ce              mov rsi,rcx
.text:004016cd 4889c7              mov rdi,rax
.text:004016d0 b800000000          mov eax,0x0
.text:004016d5 e886fcffff          call func_00401360
.text:004016da 4883c408            add rsp,0x8
.text:004016de 5b                  pop rbx
.text:004016df 5d                  pop rbp
```

Live and Dead Variables

A variable is **live** *at a particular point in a program* iff its value at that point *will be used in the future*. Otherwise the variable is **dead**.

- This means you have to know the *future uses* of the variable. Structuring pays off here.
- This analysis is used for *register allocation* when compiling programs. Lots of variables, but only a few registers… multiple variables can use the same register if *at most* one is live at any given time.

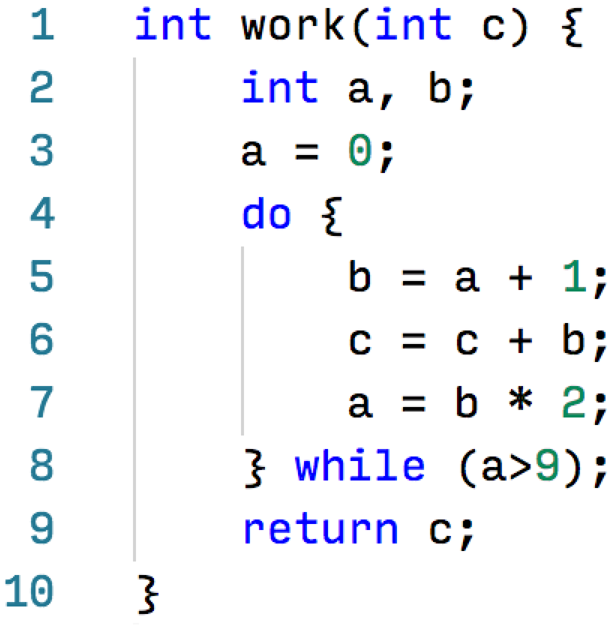
Liveness



```
 1  int work(int c) {
 2      int a, b;
 3      a = 0;
 4      do {
 5          b = a + 1;
 6          c = c + b;
 7          a = b * 2;
 8      } while (a>9);
 9      return c;
10  }
```

We know c is live at the end, since it is returned on line 9. Both a and b are auto variables, and they go out of scope.

Liveness



```
1   int work(int c) {
2       int a, b;
3       a = 0;
4       do {
5           b = a + 1;
6           c = c + b;
7           a = b * 2;
8       } while (a>9);
9       return c;
10  }
```

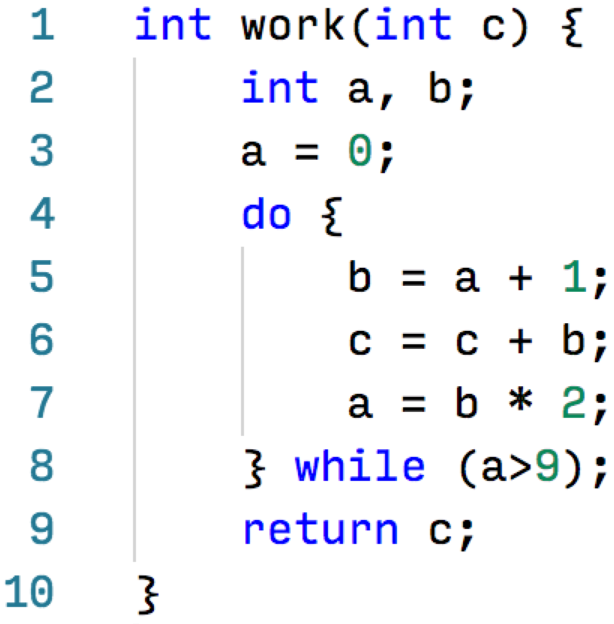
At line 7 we have `a = b * 2`.

The lvalues are *removed* from the live set, and the rvalues are *added to* the live set.

If we start with the set {c}, then we remove a (which is already not in the set) and add b.

The live variable set *immediately prior to* line 7 is {b,c}.

Liveness



```
1    int work(int c) {
2        int a, b;
3        a = 0;
4        do {
5            b = a + 1;
6            c = c + b;
7            a = b * 2;
8        } while (a>9);
9        return c;
10   }
```

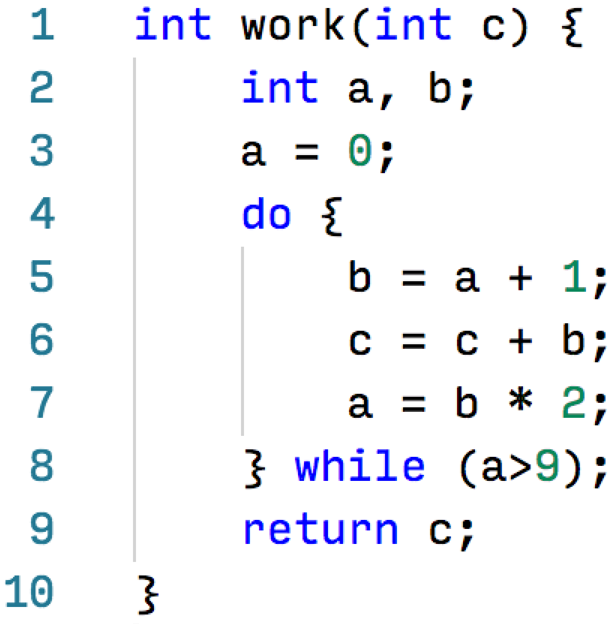
At line 6 we have `c = c + b`.

The lvalues are *removed* from the live set, and the rvalues are *added to* the live set.

We start with the set *immediately after* line 6, which we know is {b,c}, and then we remove c and add both b and c.

The live variable set *immediately prior to* line 6 is {b,c}.

Liveness



```
1   int work(int c) {
2       int a, b;
3       a = 0;
4       do {
5           b = a + 1;
6           c = c + b;
7           a = b * 2;
8       } while (a>9);
9       return c;
10  }
```

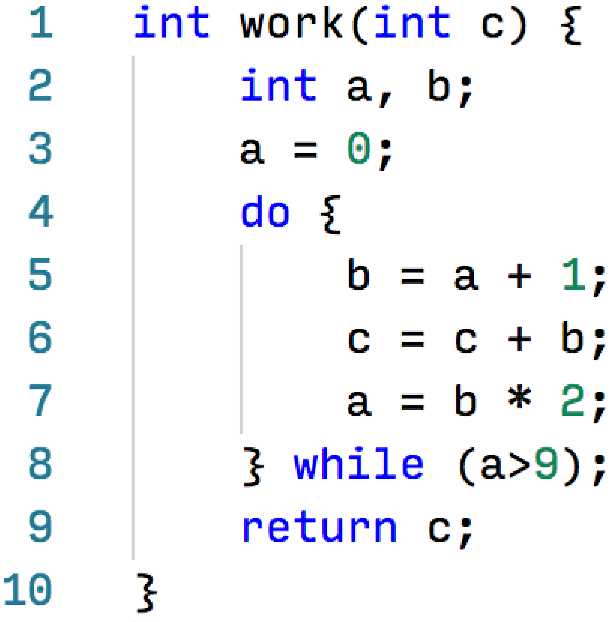
At line 5 we have `b = a + 1`.

The lvalues are *removed* from the live set, and the rvalues are *added to* the live set.

We start with the set *immediately after* line 5, which we know is {b,c}, and then we remove b and add a.

The live variable set *immediately prior to* line 5 is {a,c}.

Liveness

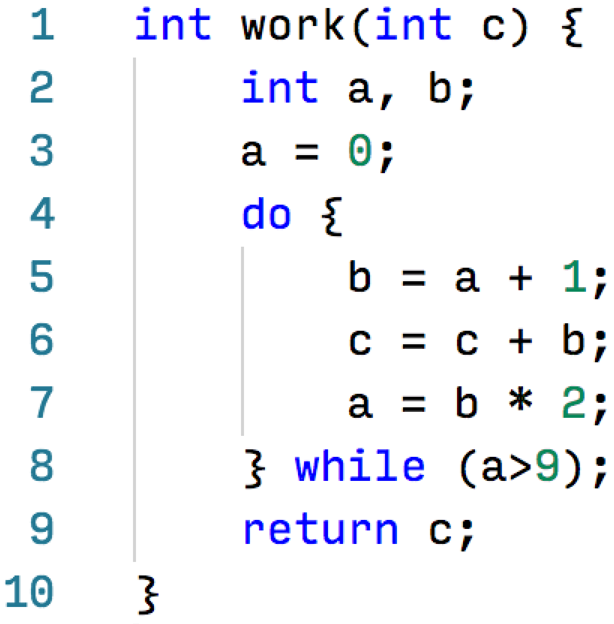


```
1   int work(int c) {
2       int a, b;
3       a = 0;
4       do {
5           b = a + 1;
6           c = c + b;
7           a = b * 2;
8       } while (a>9);
9       return c;
10  }
```

5: {a,c}
6: {b,c}
7: {b,c}

So, on entry to the loop body only a and c are live.

Liveness



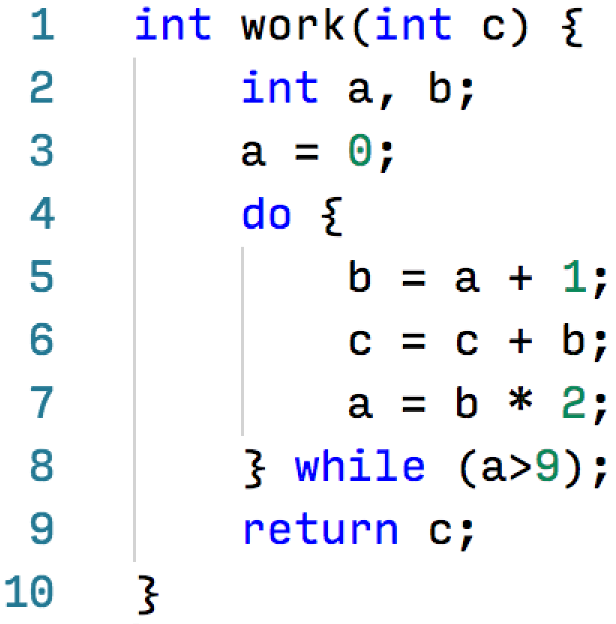
```
1   int work(int c) {
2       int a, b;
3       a = 0;
4       do {
5           b = a + 1;
6           c = c + b;
7           a = b * 2;
8       } while (a>9);
9       return c;
10  }
```

Another way to think of this is to consider *edges* of the flowgraph.

- a is **live** on edges 3➞5, 7➞8, and 8➞5
- a is dead on edge 5➞6 and edge 6➞7
- b is **live** on edge 5➞6 and edge 6➞7
- b is dead on edge 3➞5 and edge 8➞5 because it is clobbered
- c is everywhere **live** (see line 6)

We can summarize this by giving the live edge set for each variable

Liveness



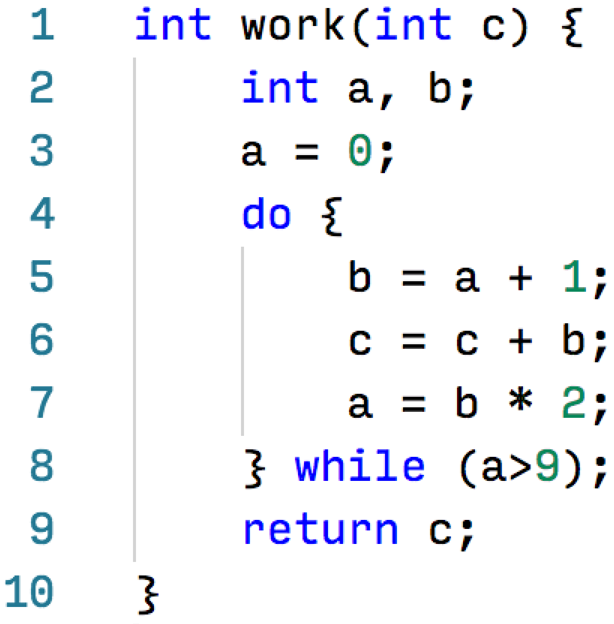
```
1    int work(int c) {
2        int a, b;
3        a = 0;
4        do {
5            b = a + 1;
6            c = c + b;
7            a = b * 2;
8        } while (a>9);
9        return c;
10   }
```

- a {3→5, 7→8, 8→5}
- b {5→6 ,6→7}
- c {3→5, 5→6, 6→7, 7→8, 8→5, 8→9}

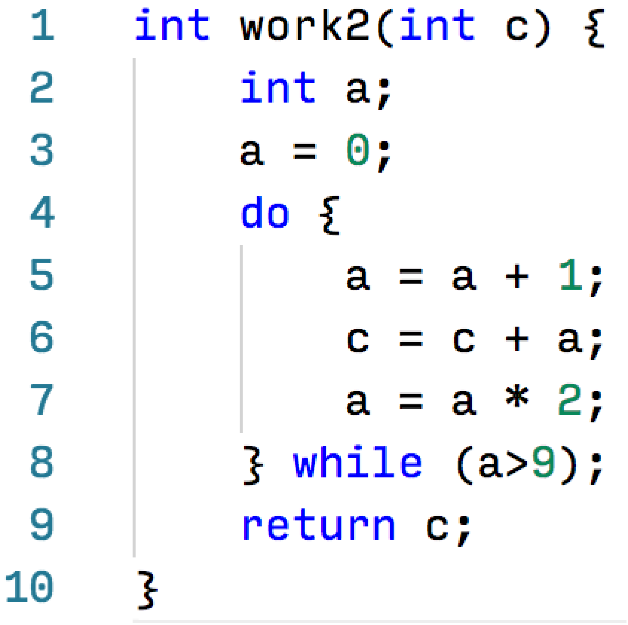
Note that a's and b's live sets are *disjoint*. They could share a register.

If they could share a register… couldn't they also share a variable… and we don't need both?

Liveness



```c
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```



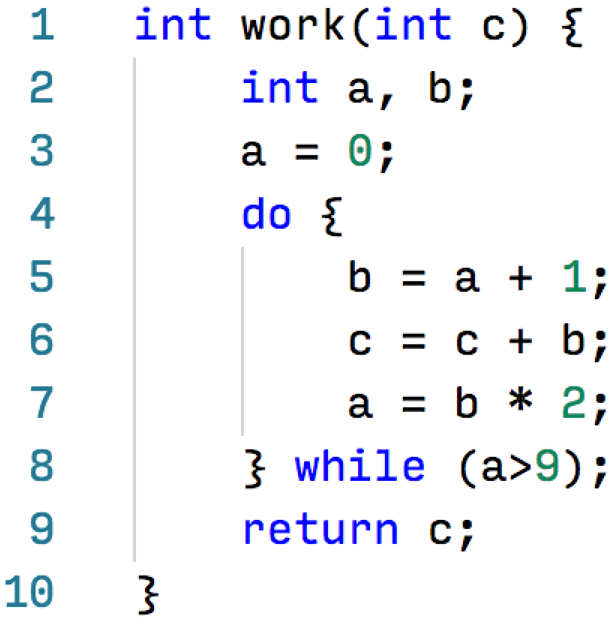
```c
int work2(int c) {
    int a;
    a = 0;
    do {
        a = a + 1;
        c = c + a;
        a = a * 2;
    } while (a>9);
    return c;
}
```

How does this help?

Liveness analysis reduces the number of things you need to care about. That's especially useful in assembly, where there are a *lot* of things you might have to care about (registers, flags, memory).

Trace Tables

Liveness



```
1   int work(int c) {
2       int a, b;
3       a = 0;
4       do {
5           b = a + 1;
6           c = c + b;
7           a = b * 2;
8       } while (a>9);
9       return c;
10  }
```

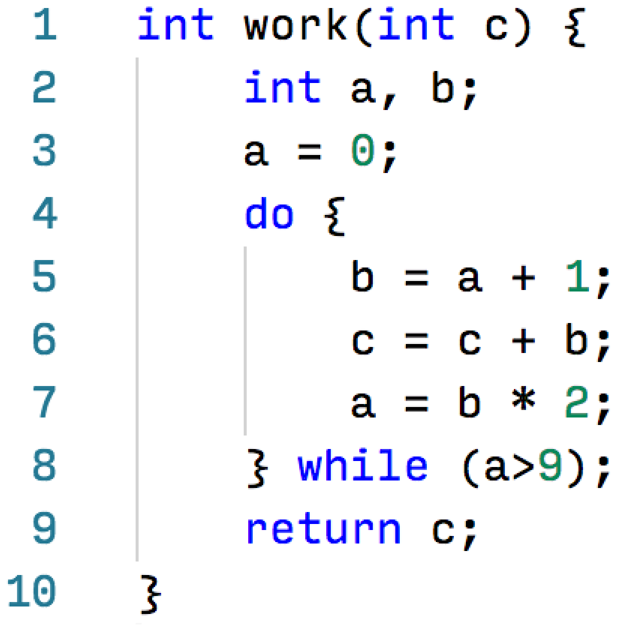
What does this actually compute?

Let’s look at the loop body and build a **trace table**.

This is a table that traces the value of each variable, and give the *new* value in terms of the *old* value.

For example, line 6 computes $c_{i+1} = c_i + b_i$, and we have $b_{i+1} = b_i$.

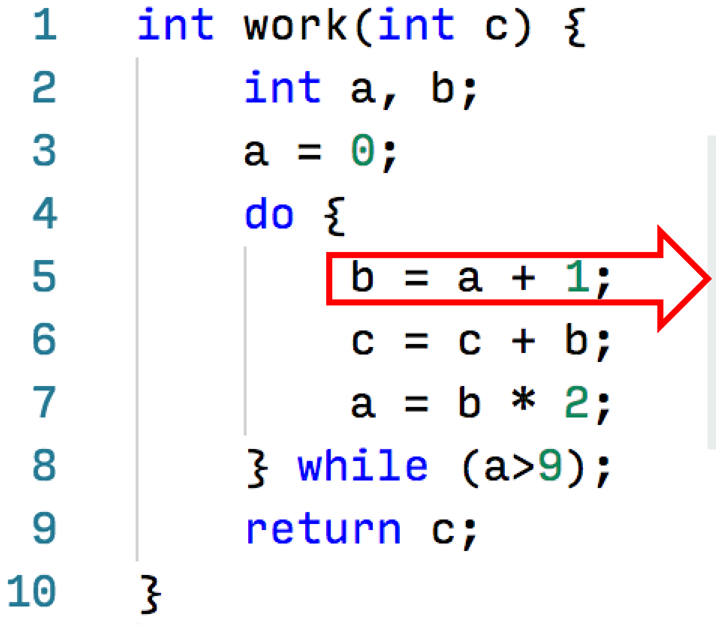
Trace Table



```
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

	a	b	c
0	a_0	b_0	c_0
1			
2			
3			

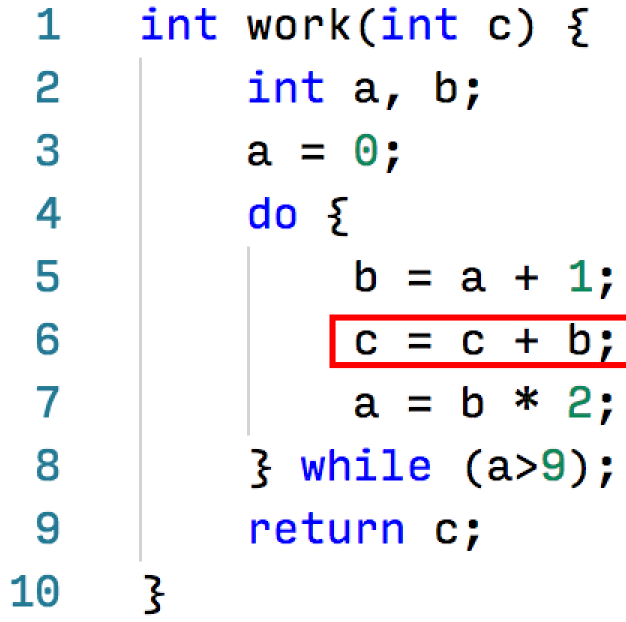
Trace Table



```
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

	a	b	c
0	a_0	b_0	c_0
1	$a_1 = a_0$	$b_1 = a_0+1$	$c_1 = c_0$
2			
3			

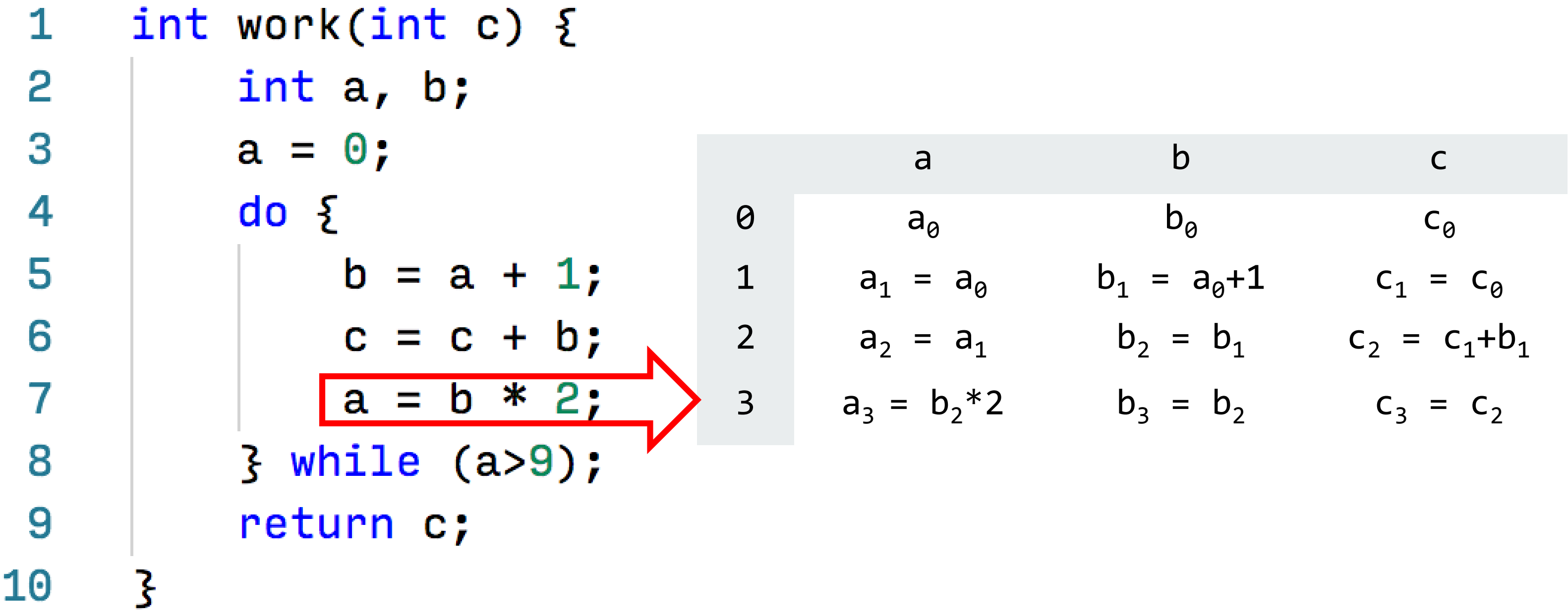
Trace Table



```
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

	a	b	c
0	a_0	b_0	c_0
1	$a_1 = a_0$	$b_1 = a_0+1$	$c_1 = c_0$
2	$a_2 = a_1$	$b_2 = b_1$	$c_2 = c_1+b_1$
3			

Trace Table



```
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

	a	b	c
0	a_0	b_0	c_0
1	$a_1 = a_0$	$b_1 = a_0+1$	$c_1 = c_0$
2	$a_2 = a_1$	$b_2 = b_1$	$c_2 = c_1+b_1$
3	$a_3 = b_2*2$	$b_3 = b_2$	$c_3 = c_2$

Trace Table

	a	b	c
0	a_0	b_0	c_0
1	$a_1 = a_0$	$b_1 = a_0+1$	$c_1 = c_0$
2	$a_2 = a_1$	$b_2 = b_1$	$c_2 = c_1+b_1$
3	$a_3 = b_2*2$	$b_3 = b_2$	$c_3 = c_2$

$a_3 = b_2*2$

$b_3 = b_2$

$c_3 = c_2$

Trace Table

	a	b	c
0	a_0	b_0	c_0
1	$a_1 = a_0$	$b_1 = a_0+1$	$c_1 = c_0$
2	$a_2 = a_1$	$b_2 = b_1$	$c_2 = c_1+b_1$
3	$a_3 = b_2*2$	$b_3 = b_2$	$c_3 = c_2$

$$\begin{aligned} a_3 &= b_2*2 \\ &= b_1*2 \end{aligned}$$

$$\begin{aligned} b_3 &= b_2 \\ &= b_1 \end{aligned}$$

$$\begin{aligned} c_3 &= c_2 \\ &= c_1 + b_1 \end{aligned}$$

Trace Table

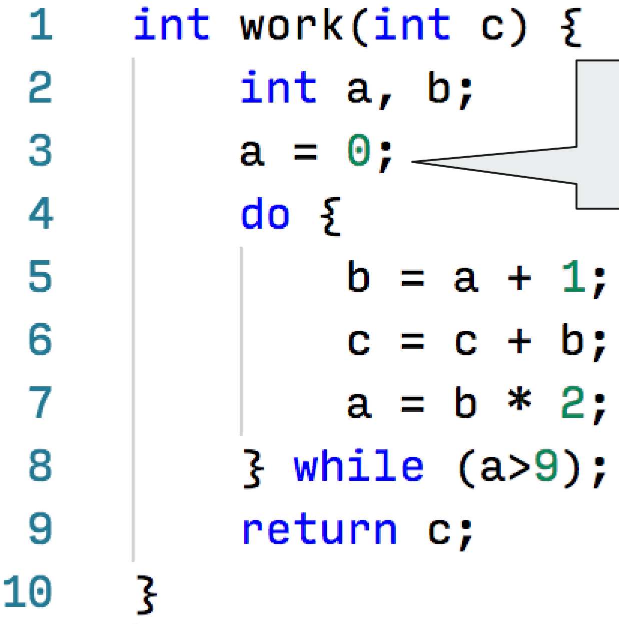
	a	b	c
0	a_0	b_0	c_0
1	$a_1 = a_0$	$b_1 = a_0+1$	$c_1 = c_0$
2	$a_2 = a_1$	$b_2 = b_1$	$c_2 = c_1+b_1$
3	$a_3 = b_2*2$	$b_3 = b_2$	$c_3 = c_2$

$$\begin{aligned} a_3 &= b_2*2 \\ &= b_1*2 \\ &= (a_0 + 1)*2 \end{aligned}$$

$$\begin{aligned} b_3 &= b_2 \\ &= b_1 \\ &= a_0 + 1 \end{aligned}$$

$$\begin{aligned} c_3 &= c_2 \\ &= c_1 + b_1 \\ &= c_0 + a_0 + 1 \end{aligned}$$

Liveness



```
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

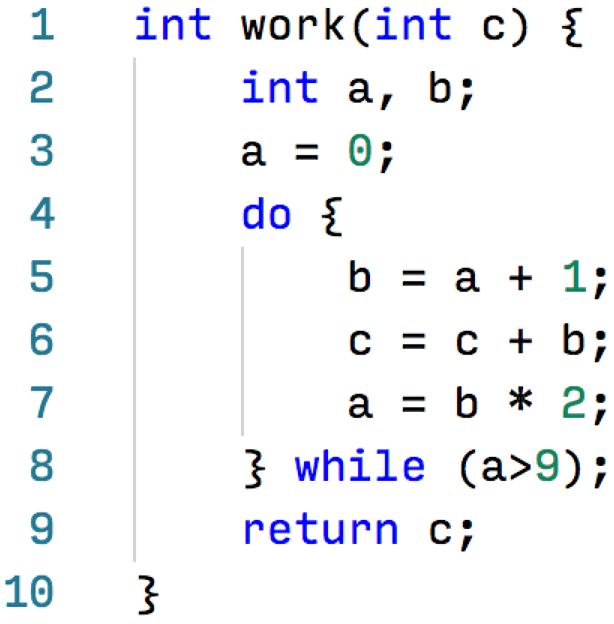
But we know $a_0 = 0$.

$$\begin{aligned} a_3 &= b_2*2 \\ &= b_1*2 \\ &= (a_0 + 1)*2 \end{aligned}$$

$$\begin{aligned} b_3 &= b_2 \\ &= b_1 \\ &= a_0 + 1 \end{aligned}$$

$$\begin{aligned} c_3 &= c_2 \\ &= c_1 + b_1 \\ &= c_0 + a_0 + 1 \end{aligned}$$

Liveness



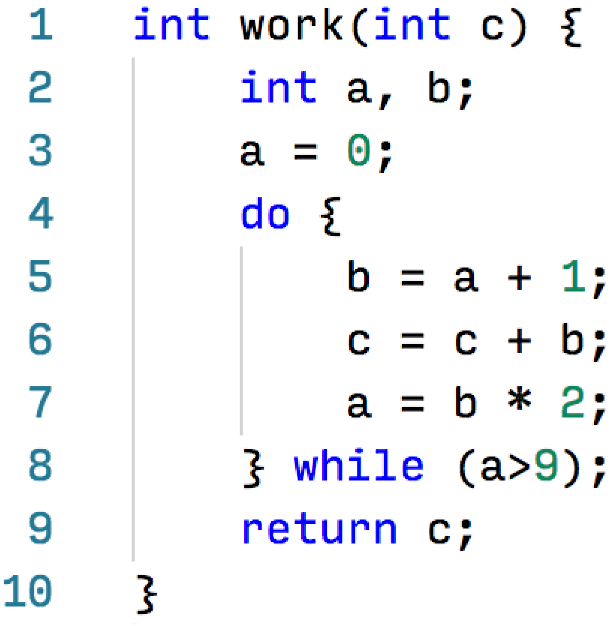
```
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

$$\begin{aligned} a_3 &= b_2*2 \\ &= b_1*2 \\ &= (a_0 + 1)*2 \\ &= 2 \end{aligned}$$

$$\begin{aligned} b_3 &= b_2 \\ &= b_1 \\ &= a_0 + 1 \\ &= 1 \end{aligned}$$

$$\begin{aligned} c_3 &= c_2 \\ &= c_1 + b_1 \\ &= c_0 + a_0 + 1 \\ &= c_0 + 1 \end{aligned}$$

Liveness



```
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

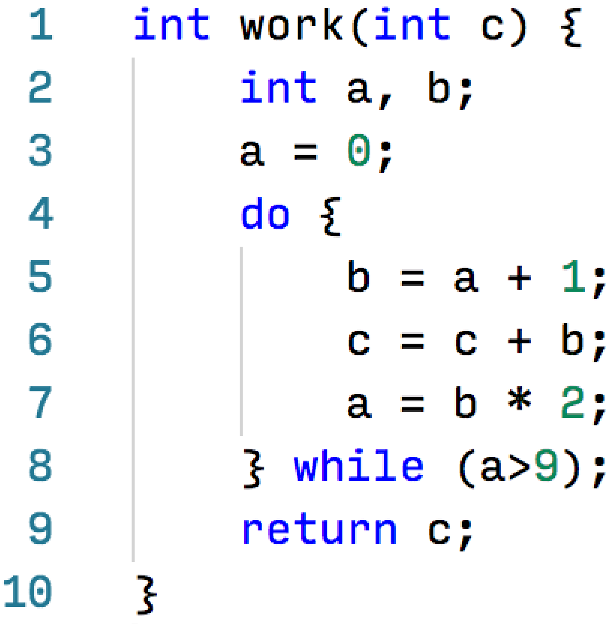
The loop body only runs once

$$\begin{aligned} a_3 &= b_2*2 \\ &= b_1*2 \\ &= (a_0 + 1)*2 \\ &= 2 \end{aligned}$$

$$\begin{aligned} b_3 &= b_2 \\ &= b_1 \\ &= a_0 + 1 \\ &= 1 \end{aligned}$$

$$\begin{aligned} c_3 &= c_2 \\ &= c_1 + b_1 \\ &= c_0 + a_0 + 1 \\ &= c_0 + 1 \end{aligned}$$

Liveness



```c
int work(int c) {
    int a, b;
    a = 0;
    do {
        b = a + 1;
        c = c + b;
        a = b * 2;
    } while (a>9);
    return c;
}
```

$c_3 = c_0 + 1$

The net effect is just to return the argument plus one.

Next Time:
Slicing