`void printdigit(int n)`
`{ if (n<0 || n>9)`
`  cout << "ERROR\n";`
`  if (n==0)`
`  cout << "-0-";`
`  else`
`  cout << "-" << n << "-"; }`
		The compiler notes that printdigit has only one local variable.
		A stack frame for printdigit will only need space for one int, plus the return address, plus one extra bookkeeping int. That is just 3 ints in total.
		'n' will always be at position +2 in the frame, although the frame itself could be anywhere.

`void printbackwards(int num)`
`{ int q = num/10;`
`  int r = num%10;`
`  if (q>0)`
`  printbackwards(q);`
`  printdigit(r);`
`  int z=r*17+q; }`
		The compiler notes that printbackwards has four local variables: num, q, r, z.
		A stack frame for printdigit contain a total of 6 ints
		'q' will always be at position −1 in the frame, 'r' at position −2, etc. although the frame itself could be anywhere.
		'num' will be at position +2.

`void main()`
`{ int ant=372;`
`  printbackwards(ant);`
`  cout << "\n"; }`
		Main's stack frame will be just 3 ints long, with 'ant' at position −1.

The location of the currently active stack frame is always stored in the special purpose CPU register called the Frame Pointer (FP). The exact location of a stack frame in memory can not be predicted until the program is running.

Just suppose that when printbackwards is called, its 6 element stack frame just happens to be allocated to memory locations 1371 to 1376. While printbackwards is running, its area of memory looks like this:

memory		compiler's personal notes
1377: *(not for our use)*		
1376: num is stored here		The address of 'num' is $FP + 2
1375: return address here		
1374: extra special int here	← FP points here	The value of $FP is 1374
1373: q is stored here		The address of 'q' is $FP − 1
1372: r is stored here		The address of 'r' is $FP − 2
1371: z is stored here		The address of 'z' is $FP − 3
1370: *(not for our use)*		

Only the compiler knows these addresses, it generates executable instructions with them built in.

For example, q=num/10; might be translated to this:

```
LOAD    $1    * $FP + 2
DIV     $1    10
STORE   $1    $FP - 1
```

when the program is running, only the Frame Pointer is needed.

A CPU can be expected to have a few more special purpose registers: the Program Counter (PC), and the Stack Pointer (SP). For simplicity, the three special registers may be treated as just different names for ordinary registers, perhaps $PC≡$15, $SP≡$14, $FP≡$13, something like that.

The frame pointer just tells us where the currently active stack frame is. The stack pointer keeps track of how much memory has been used for stack frames, and therefore tells us exactly where the next one should be created. Everything with an address higher than $SP is part of the stack, and won't be used for anything else. Everything with an address lower than $SP is not part of the stack, and is not safe. One day the stack might extend into it.

With this, the scheme for creating new stack frames is easy. To allocate space for 6 ints, simply subtract 6 from the value of $SP. Then the 6 locations immediately above $SP are protected from accidental re-use, and can become the stack frame. To destroy a stack frame (when a function exits/returns) all that is needed is to add 6 back to $SP, and everything will be as it was before.

The compiler knew exactly how big each function's stack frame would need to be, so it easily inserts the appropriate SUB instruction to the beginning of the function's code, and the matching ADD to the end.

As memory would be, just before the program prints the digit 3:

999999	0	main's stack frame	fake return address
999998	0		fake saved FP
999997	372		ant
999996	372	printbackwards' first stack frame	num
999995	xxxxx		return address
999994	999998		saved FP
999993	37		q
999992	2		r
999991	??		z not set yet
999990	37	printbackwards' second stack frame	num
999989	yyyyy		return address
999988	999994		saved FP
999987	3		q
999986	7		r
999985	??		z not set yet
999984	3	printbackwards' third stack frame	num
999983	yyyyy		return address
999982	999988		saved FP
999981	0		q
999980	3		r
999979	??		z not set yet
999978	3	printdigit's stack frame	n
999977	zzzzz		return address
999976	999982		save FP

At this point, the frame pointer register (FP) would contain the number 999976.

The number xxxxx represents the address of the executable instruction in main, immediately following the CALL to printbackwards.

The number yyyyy represents the address of the executable instruction in printbackwards, immediately following the (recursive) CALL to printbackwards.

The number zzzzz represents the address of the executable instruction in printbackwards, immediately following the CALL to printdigit.

Of course, the diagram above is based on the assumption that we have exactly 1,000,000 memory locations.

Now you can try it.

999999			
999998			
999997			
999996			
999995			
999994			
999993			
999992			
999991			
999990			
999989			
999988			
999987			
999986			
999985			
999984			
999983			
999982			
999981			
999980			
999979			
999978			
999977			
999976			
999975			
999974			
999973			
999972			
999971			
999970			
999969			
999968			
999967			
999966			
999965			
999964			
999963			
999962			
999961			
999960			
999959			
999958			
999957			