Parameter Passing

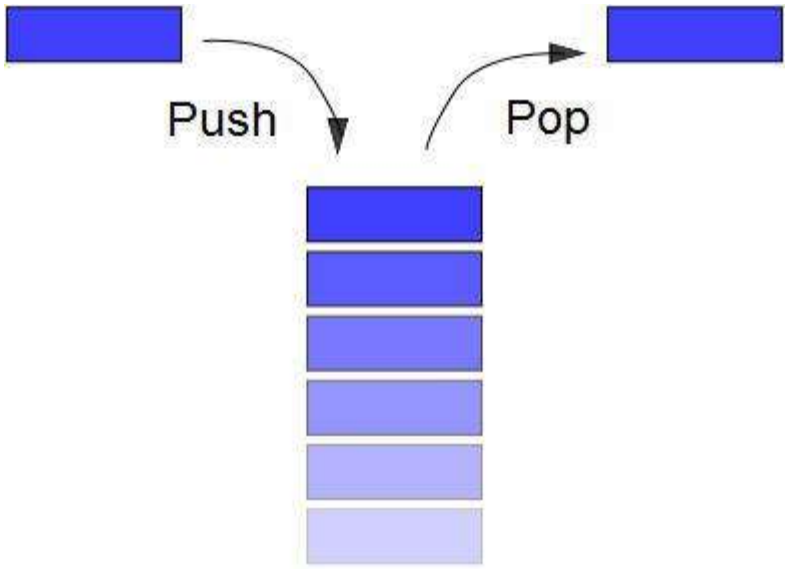
COMP360

“It should be noted that no ethically-trained software engineer would ever consent to write a DestroyBaghdad procedure. Basic professional ethics would instead require him to write a DestroyCity procedure, to which Baghdad could be given as a parameter.”

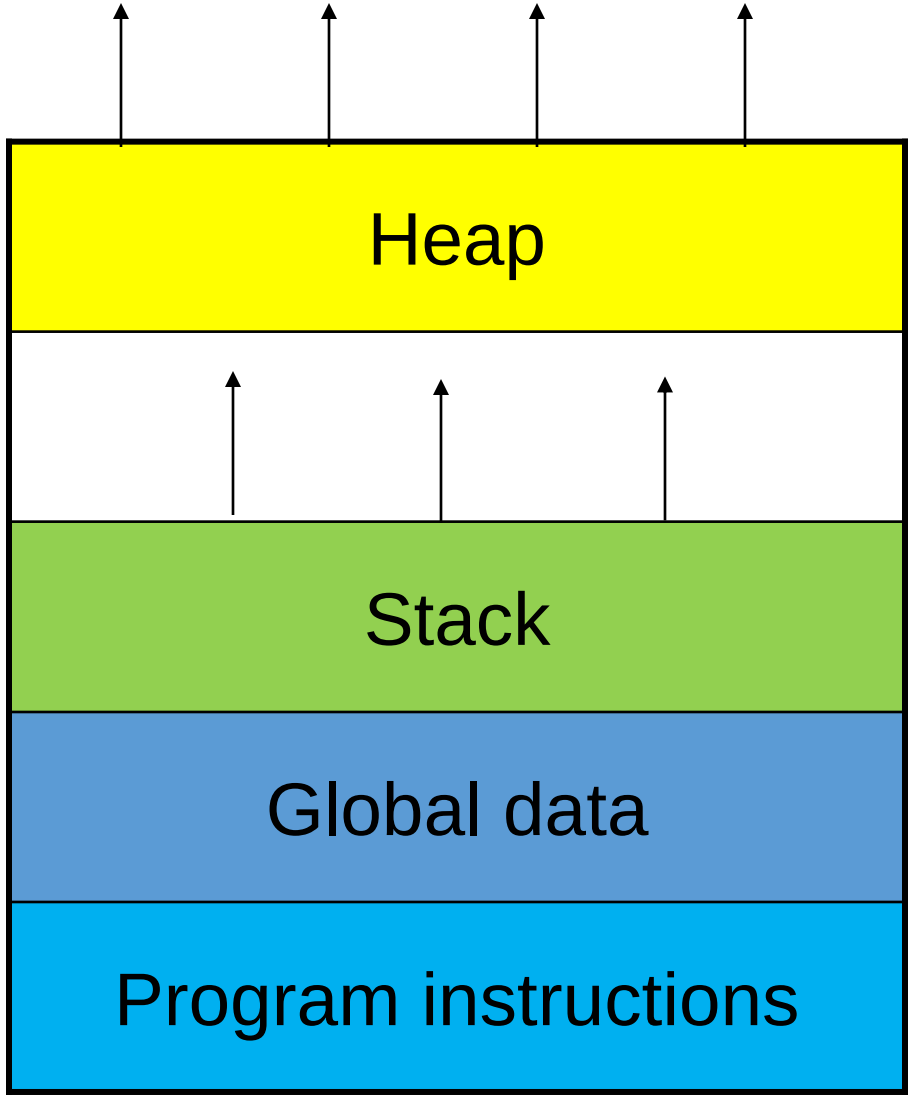
Nathaniel S. Borenstein

Stacks

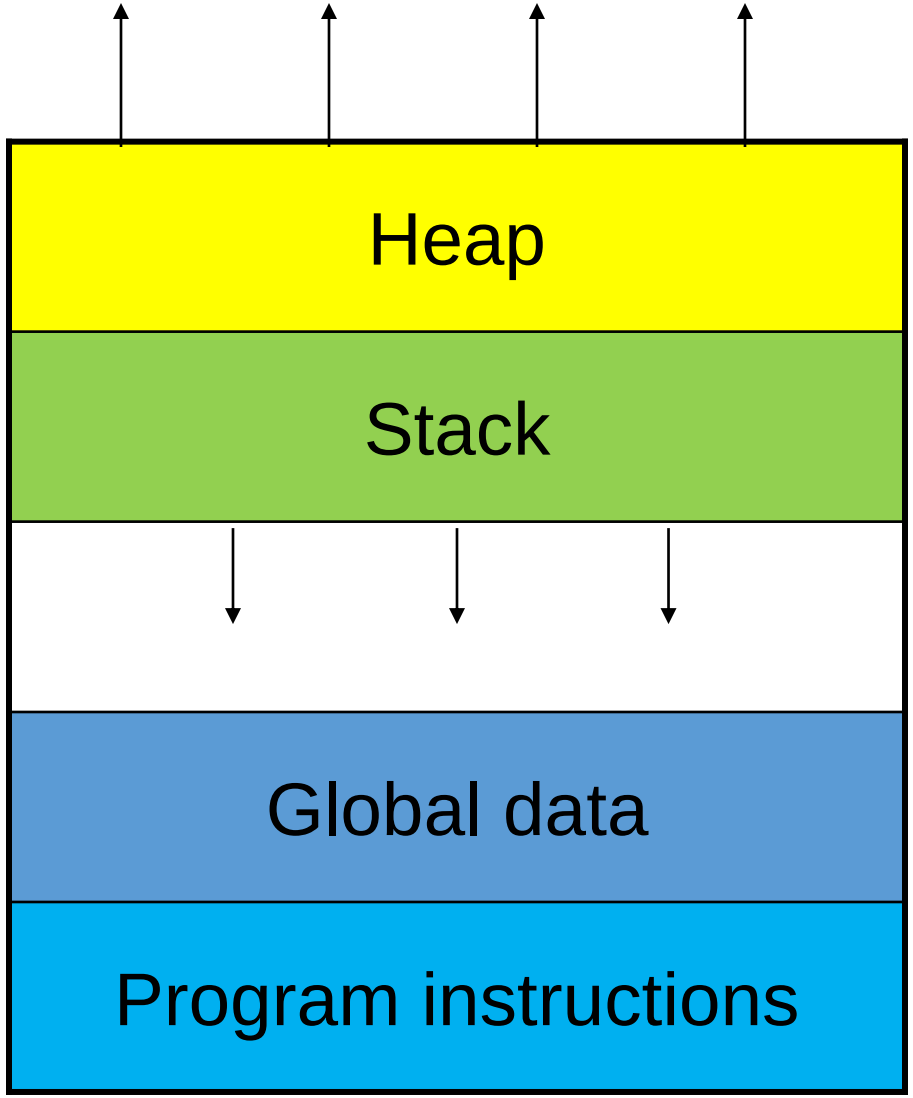
- Many programming languages use stacks to pass parameters
- Many computer architectures have stack instructions to help implement these programming languages
- Most architectures have stack pointer register. The stack pointer always points to the top item on the stack.



Program Memory Organization



Program Memory Organization



Intel method

Function Call Hardware

- All computers have machine language instructions to support function calls
- The level of hardware support varies with modern computers providing more support

Intel Call instruction

- The **`CALL`** instruction basically pushes the program counter on the stack and branches to a new location
- There are many versions of the Intel **`CALL`** instruction to support different addressing modes and changes in privileges

Intel RET instruction

- The **`RET`** or return instruction pops a value from the stack and places it in the program counter register
- Since the program counter contains the address of the next instruction to execute, this has the effect of branching back to the calling program

Basic Steps to Call a Method

- Compute any equations used in the parameters, such as `x=func(a + b);`
- Push the parameter values on the stack
- Execute a call instruction to push the return address on the stack and start execution at the first address of the function

Upon function entry

- Save the contents of the registers
  - Many systems have the convention that a method should return with the registers just the way they were when called
- Increase the stack pointer to reserve memory for the local variable
- Start executing the function code

Upon function exit

- Reduce the stack by the size of the local variable
- Pop the register values
- Execute the return instruction to pop the address from the stack into the program counter

Example Function Call

- Consider the function

```
void thefunc(Widget b, int a ){
    int r = a;
}
```

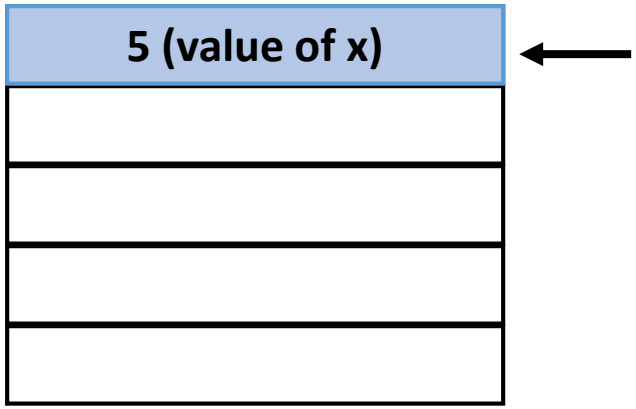
- that is called by the main program

```
int x = 5;
Widget y = new Widget();
thefunc( y, x );
```

- The Widget y is passed by reference. The int x is passed by value.

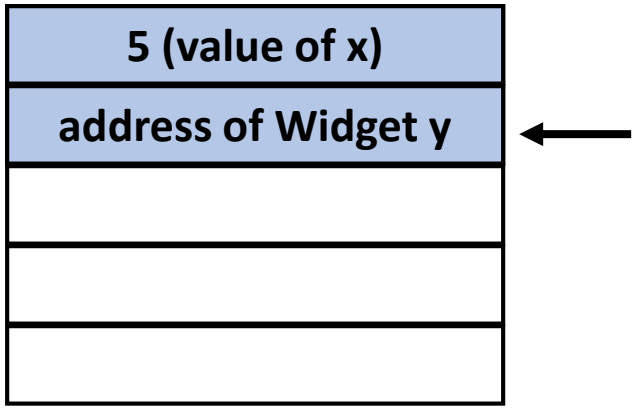
Stack for Call Parameters

- push x



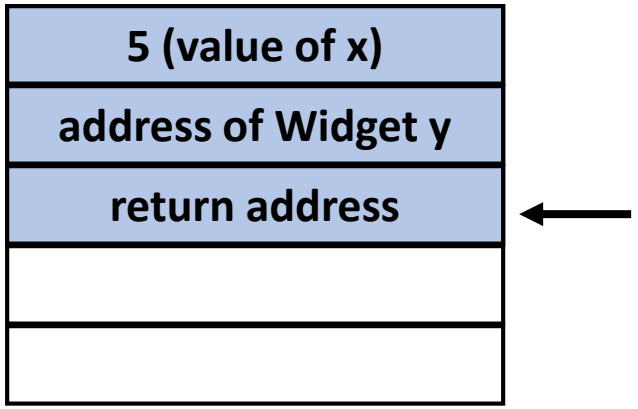
Stack for Call Parameters

- push x
- push address of y



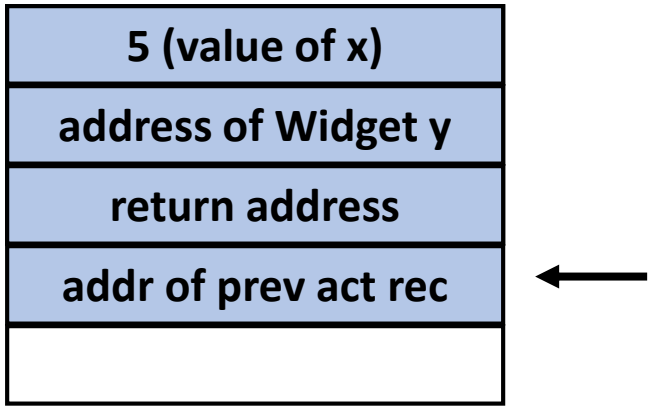
Stack for Call

- push x
- push address of y
- call thefunc



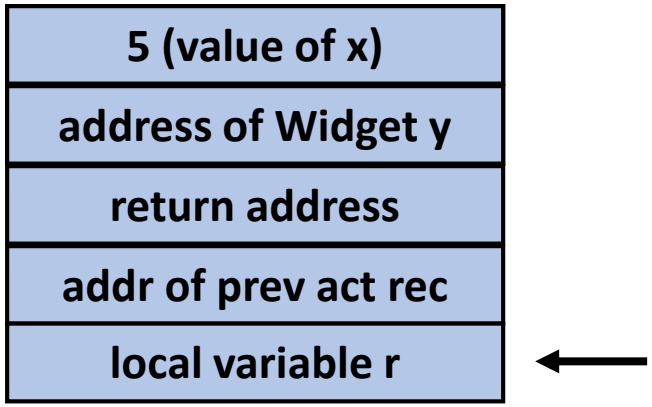
Stack with Activation Records

- push x
- push address of y
- call thefunc
- Link to previous activation record



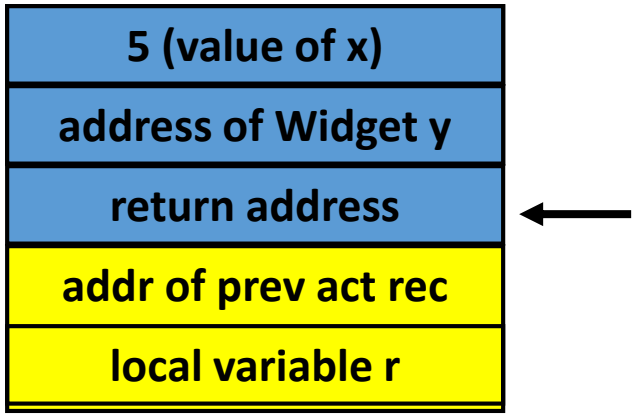
Stack Use by Function

- push x
- push address of y
- call **`thefunc`**
- Link to previous activation record
- increment stack



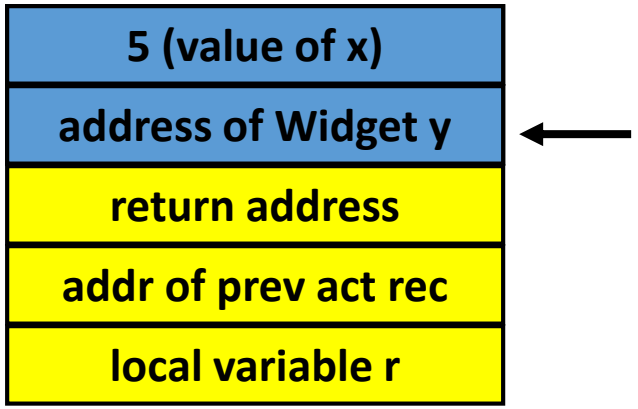
Stack for Return

- push x
- push address of y
- call **`thefunc`**
- Link to previous activation record
- increment stack
- decrement stack



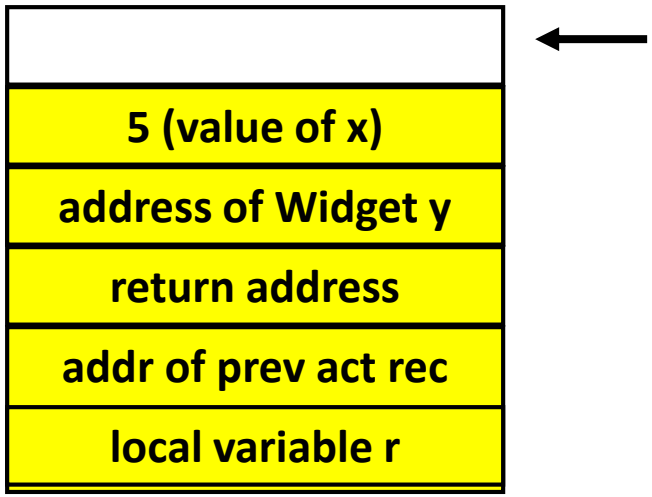
Stack for Return

- push x
- push address of y
- call **`thefunc`**
- Link to previous activation record
- increment stack
- decrement stack
- return



Cleanup Stack

- push x
- push address of y
- call **`thefunc`**
- increment stack
- decrement stack
- return
- decrement stack by 2



Linked Stacks

- Some systems use a doubly linked list to simulate a stack
- Upon entry to a method, a block of memory is acquired which is linked to the previous block
- This block of memory contains the register save area
- Upon exit, the registers are restored and the block released

Activation Records

- An activation record or frame contains the stack information for a method call
- The activation records are linked together

Activation Record Format

locals	The local variables of the method. This can vary in size.
Frame pointer	The address of the previous activation frame.
Return address	The address of the instruction after the method call in the calling program.
parameter 2	The second parameter to the method
parameter 1	The first parameter to the method

Finding the Activation Record

- In the Windows / Intel world, the EBP register points to the activation record
- Local variables are located on the stack and accessed using the EBP register as an index

Byte Ordering

- Some systems store the least significant byte first (**Little Endian**). Others store the most significant byte first (**Big Endian**)
- The decimal number 258 (0100000010_2) would be stored in as a 32 bit binary number

Big Endian

00000000 00000000 00000001 00000010

byte 0 byte 1 byte 2 byte 3

Little Endian

00000010 00000001 00000000 00000000

byte 0 byte 1 byte 2 byte 3

Intel is Little Endian

- The Intel and AMD processors are little endian
- Integers and addresses are stored in reverse byte order

Program Addresses in Visual Studio

- By mousing over the name of a method, you can learn the address of the beginning of the method
- You can look at the register values in the Watch
- The memory can be view at

  Debug / Window / Memory / Memory 1

Microsoft Security

- To reduce the vulnerability to buffer overflow attacks, Microsoft Windows loads the stack at a new random address each time the program is run
- Visual Studio surrounds data values with a stack canary
- The Visual Studio canary is `0xCCCCCCCC`

Stack Canaries

- A stack canary is a random or constant value placed on the stack between the user data and the return address.
- Overflowing the local variable and changing the return address will also change the stack canary
- Before returning, the program checks the canary value

5 (value of bass)
address of carp
return address
Addr of last frame
Stack canary
cow[4]

Example Program

```
1   // Stack question in the original Exam 1
2   #include <iostream>
3   using namespace std;
4   int crow = 3, raven = 5, robin;
5
6   int methodA(int x, int y);
7   int methodB(int z);
8
9   int methodA(int dog, int cat) {
10      int goat, cow;
11      goat = dog + cat;
12      cow = methodB(goat);
13      return cow;
14  }
15  int methodB(int bull) {
16      int horse;
17      // What is on the stack at this point?
18      horse = bull * bull;
19      cout << horse;
20      return horse;
21  }
22  int main() {
23      robin = methodA(crow, raven);
24  }
```

Generated Code for line 23
robin = methodA(crow, raven);

```
mov   eax, DWORD PTR ?raven@@3HA          ; raven
push  eax
mov   ecx, DWORD PTR ?crow@@3HA           ; crow
push  ecx
call  ?methodA@@YAHHH@Z                   ; methodA
add   esp, 8
mov   DWORD PTR ?robin@@3HA, eax          ; robin
```

methodA Entry Code

```
push  ebp
mov   ebp, esp
sub   esp, 216                          ; 000000d8H
push  ebx
push  esi
push  edi
lea   edi, DWORD PTR [ebp-216]
mov   ecx, 54                           ; 00000036H
mov   eax, -858993460                   ; ccccccccH
rep   stosd
```

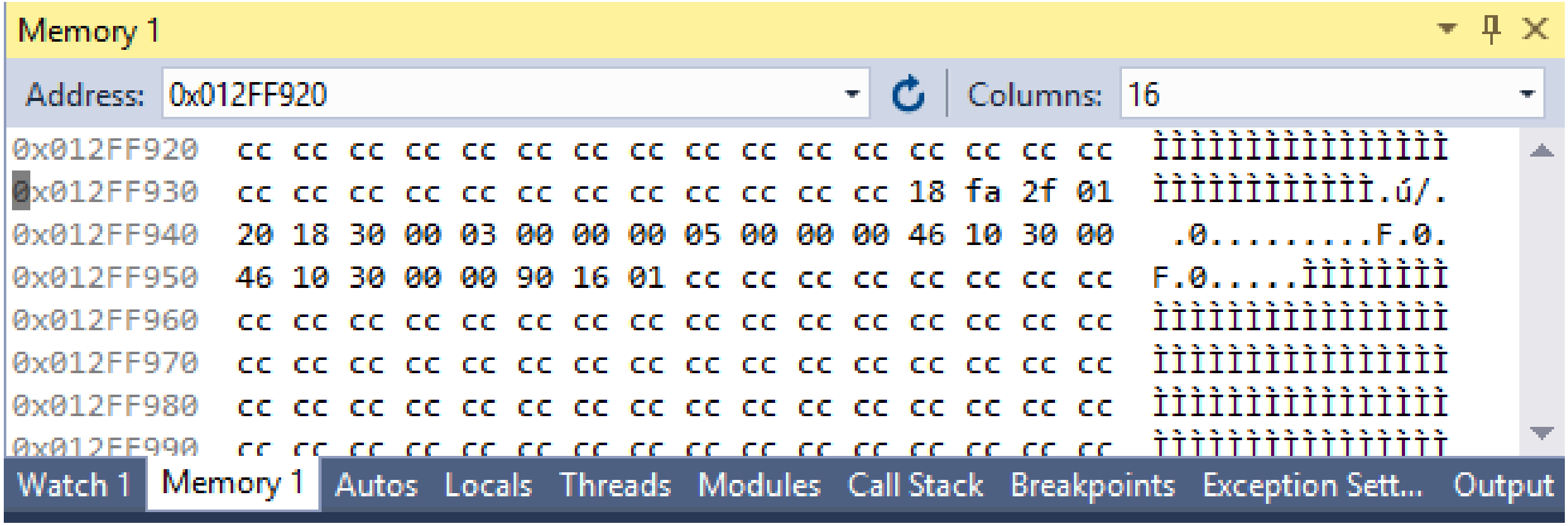
methodA Code

```
11  :       goat = dog + cat;
        mov   eax, DWORD PTR _dog$[ebp]
        add   eax, DWORD PTR _cat$[ebp]
        mov   DWORD PTR _goat$[ebp], eax
12  :       cow = methodB(goat);
        mov   eax, DWORD PTR _goat$[ebp]
        push  eax
        call  ?methodB@@YAHH@Z
        add   esp, 4
        mov   DWORD PTR _cow$[ebp], eax
```

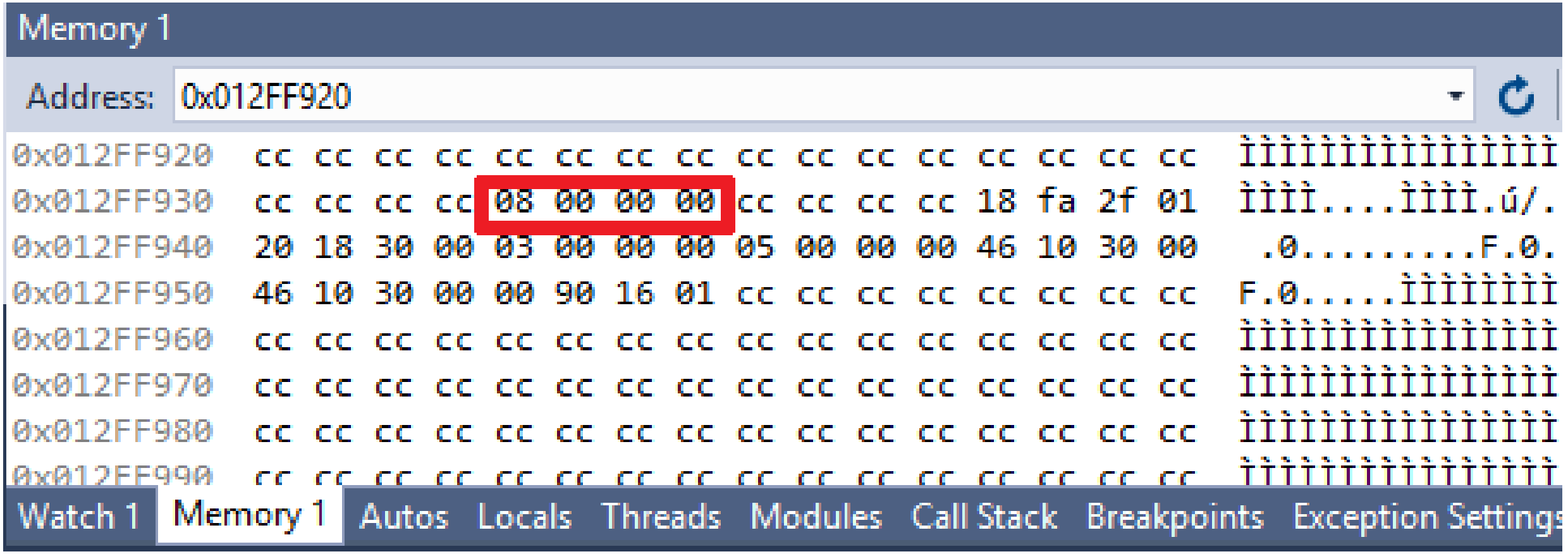
Addresses in Example

- MethodA 0x00301720
- MethodB 0x00301780
- Main 0x003017F0
- EBP in methodA 0x012FF93C
- EBP in methodB 0x012FF84C

Stack in methodA



After `goat = dog + cat`



Stack in methodB