Memory Use

COMP360

“That's the thing about people who think they hate computers. What they really hate is lousy programmers.”

Larry Niven

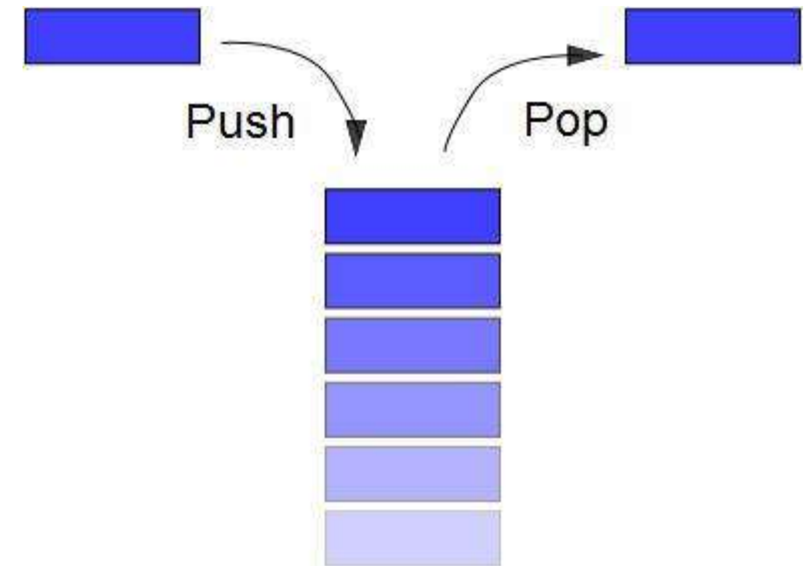
Parameter Passing Paradigms

Call by

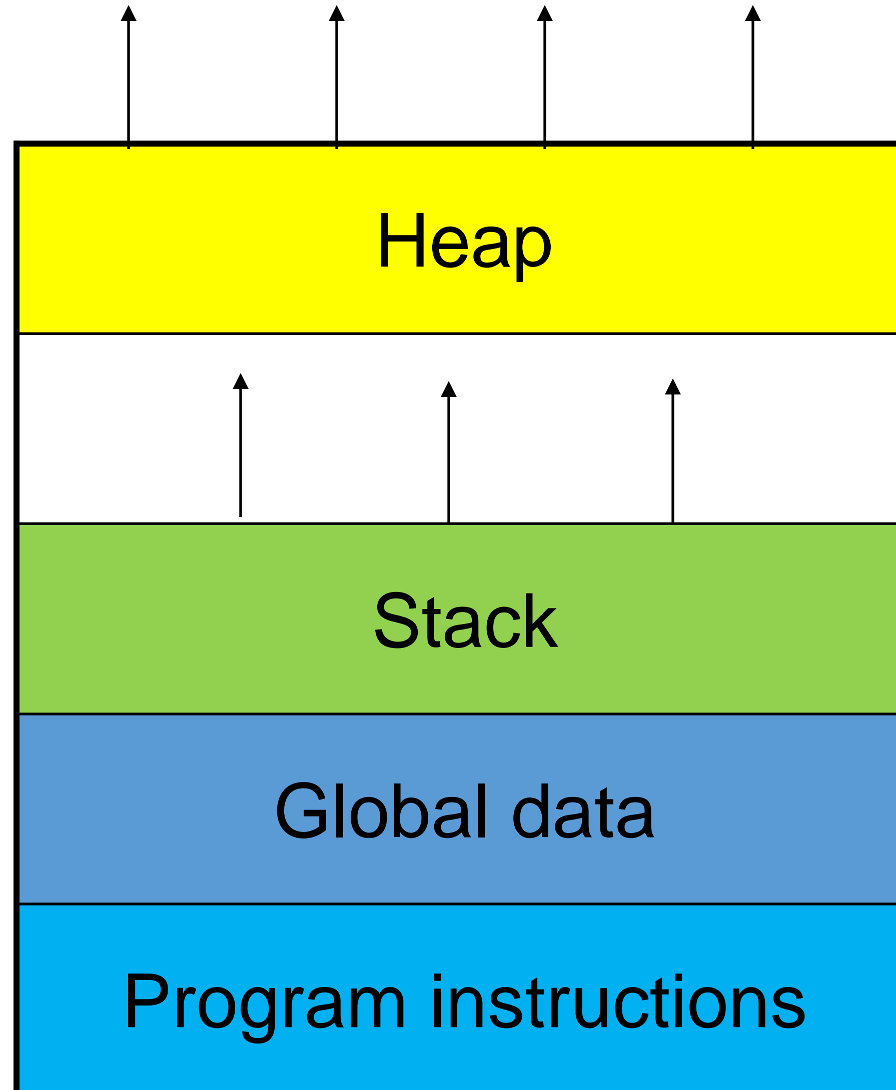
- reference
- value (in)
- value result (in out)
- result (out)
- constant value
- name

Stacks

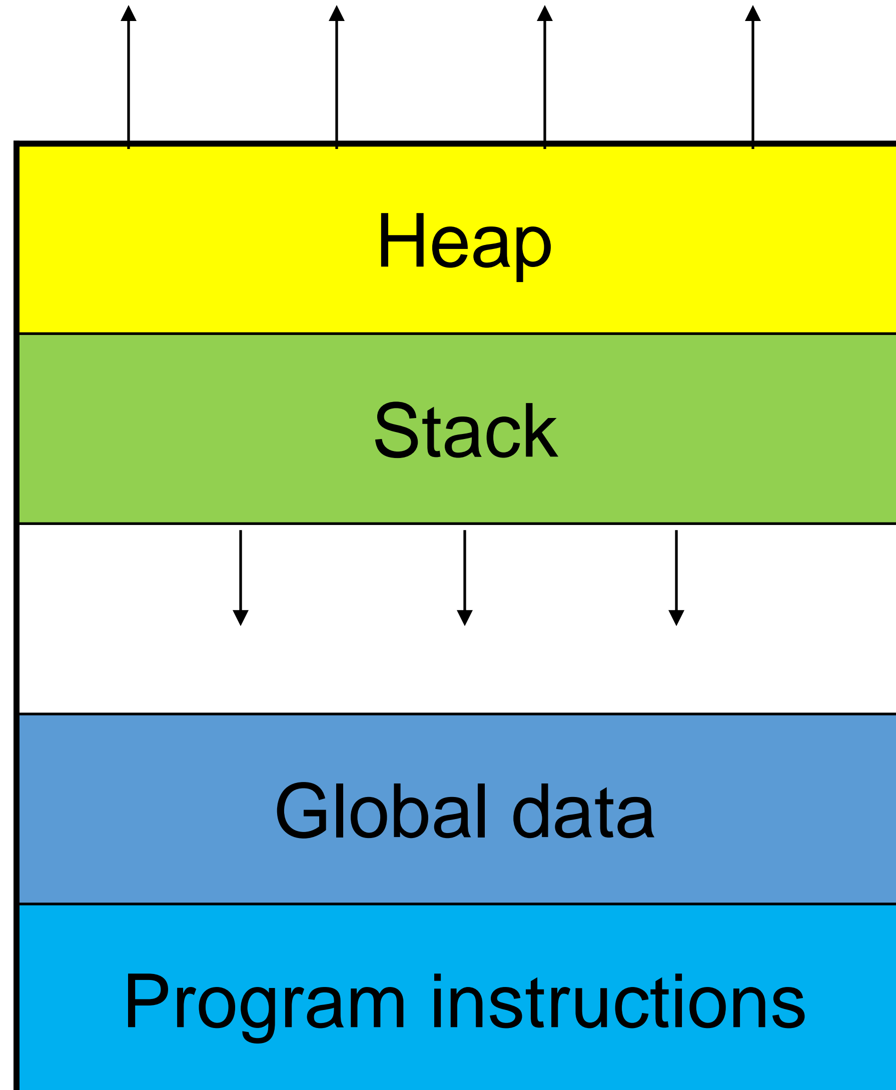
- Many programming languages use stacks to pass parameters
- Many computer architectures have stack instructions to help implement these programming languages
- Most architectures have stack pointer register. The stack pointer always points to the top item on the stack.



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

The return address pushed on the stack points to an address in

A. program instructions
B. global data
C. stack
D. heap
E none of the above

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
- Link the activation records
- 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

When a method is called many times in a program, how does it know where to return?

A. Call address in the machine language
B. Return address on the stack
C. Returns to earliest call in the source code
D. Depends on count in Program Counter

Activation Records

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

Stack Activation Frame 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 1	The first parameter to the method
parameter 2	The second parameter to the method

Example Function Call

- Consider the function

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

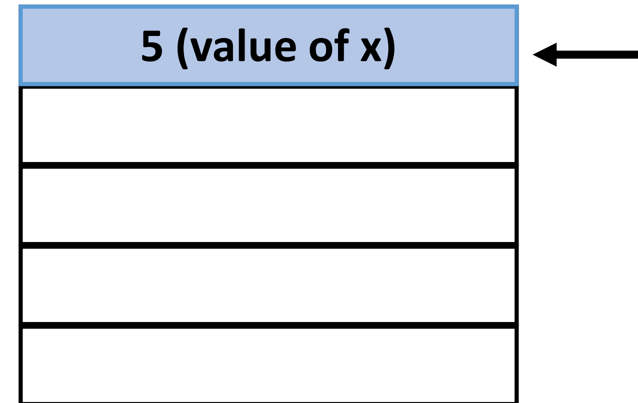
- that is called by the main program

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

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

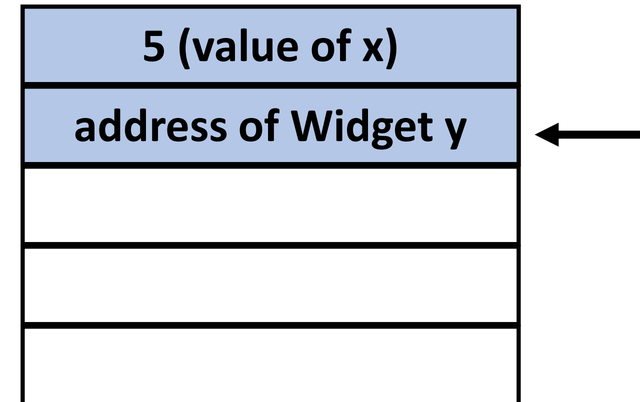
Stack for Call Parameters

- push x



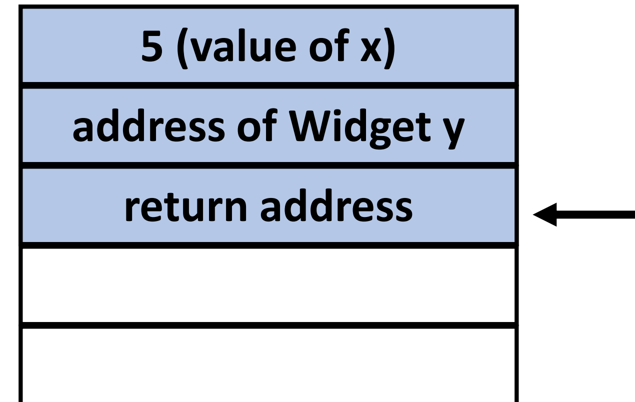
Stack for Call Parameters

- push x
- push address of y



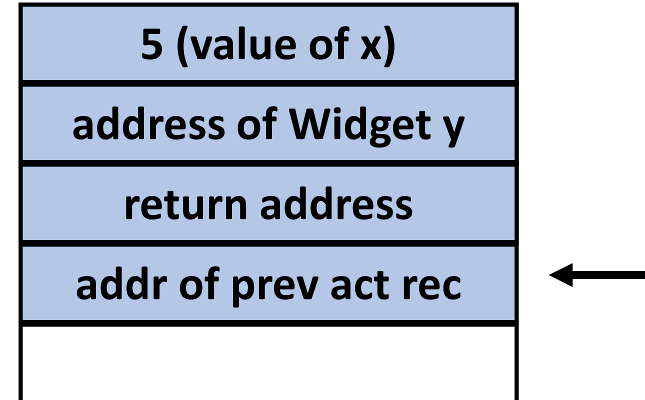
Stack for Call

- push x
- push address of y
- call thefunc



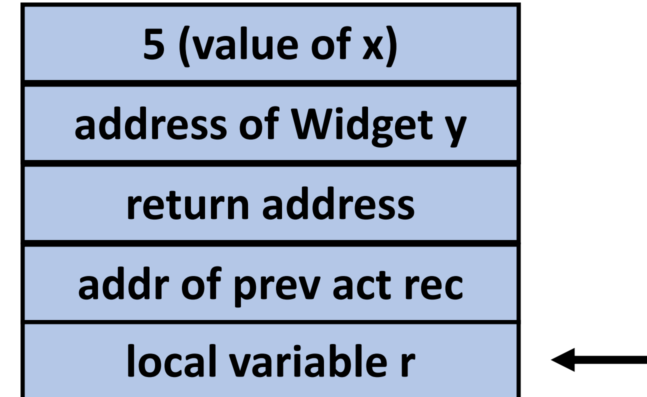
Stack with Activation Records

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



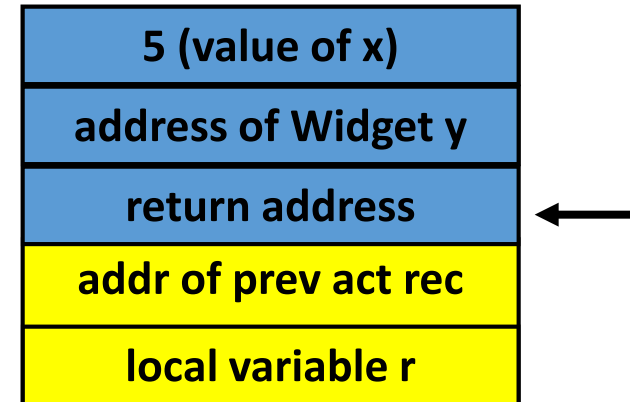
Stack Use by Function

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



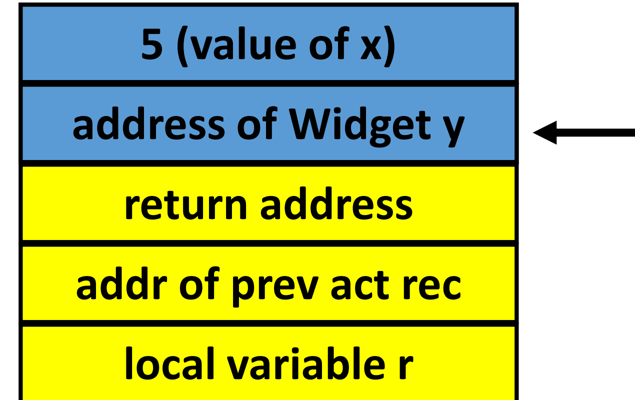
Stack for Return

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



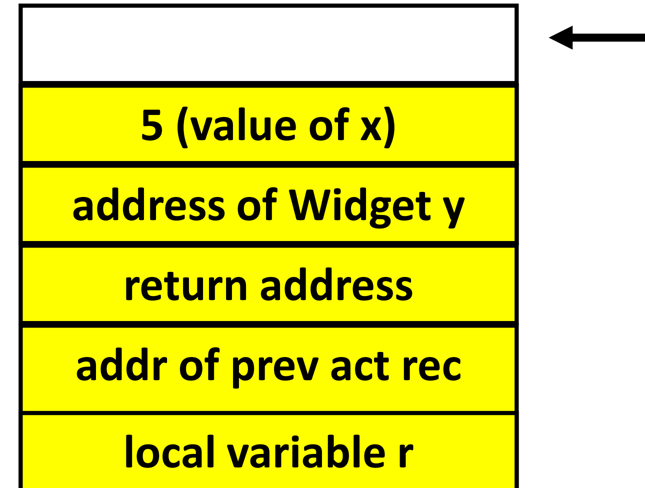
Stack for Return

- 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



Explain the Implementation

Working in teams of students, explain how the following parameter passing paradigms can be implemented (value or address on the stack)

- reference
- value (in)
- value result (in out)
- result (out)

What will this C++ program do?

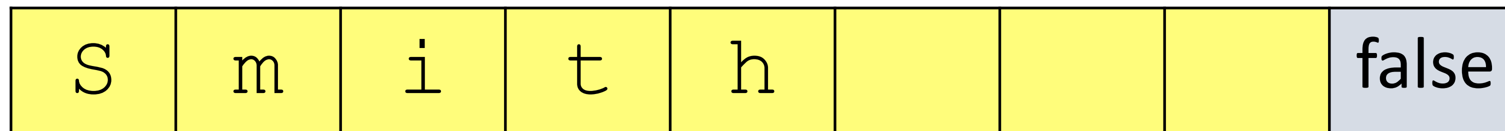
```
void examplefunc() {
   int stuff = 0;
   char info[4];
   int i;
   for (i = 0; i < 7; i++){
        info[i] = stuff++;
    }
}
```

A. Compiler Error
B Run time buffer overflow error
C. Corrupt data
D. Data execute exception

Basic Buffer Overflow

```
boolean rootPriv = false;
char name[8];
cin >> name;
```

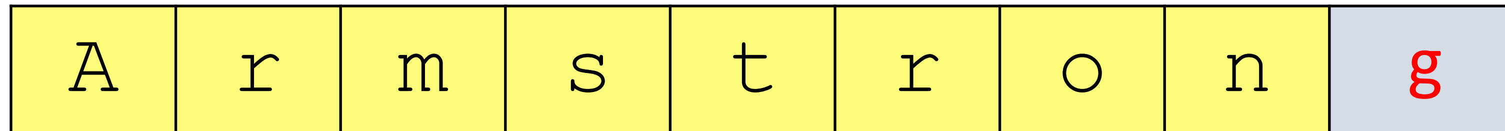
- When the program reads the name “Smith”



Basic Buffer Overflow

```
boolean rootPriv = false;
char name[8];
cin >> name;
```

- When the program reads the name "Armstrong"

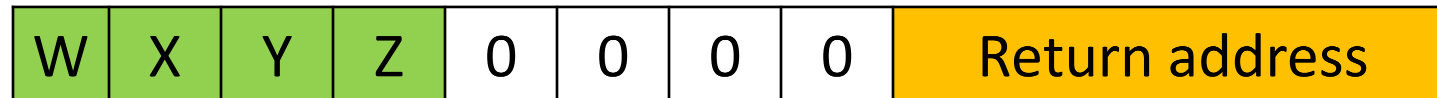


char name[8] rootPriv

Stack Overflow

- A stack overflow exploit occurs when a user enters data that exceeds the memory reserved for the input
- The input can change adjacent data or the return address on the stack

```
char myStuff[4];
```



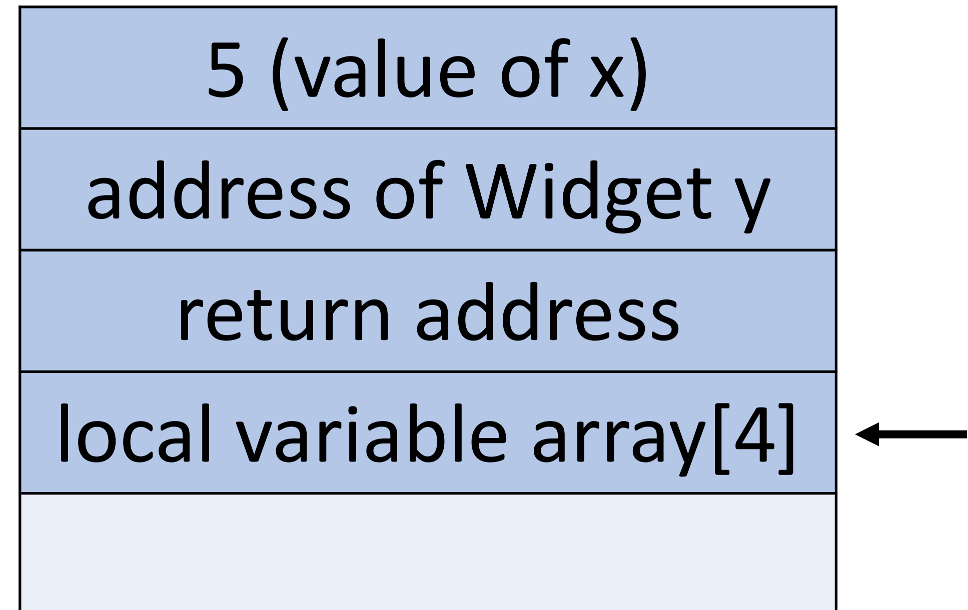
Program Stack

Exceeding Array Bounds

- In many languages, including C and C++, the system will not detect that a method has indexed beyond the end of an array
- If a program stores data in an array using an index bigger than the size of the array, the data will be stored in whatever memory follows the array

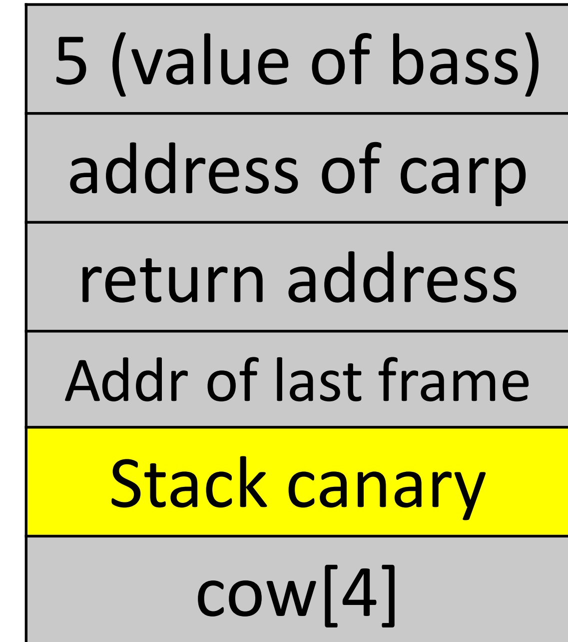
Stack Overflow Attack

- A common security attack is to cause a program to overflow the stack
- If the program stores a value into array[4], it will right in the data past array, the return address
- Instructions might be loaded in the rest of the stack



Stack Canaries

- A stack canary is a random number 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



Data Execution Prevention

- Most newer processors have a bit in the page table that inhibits instruction fetches from that page
- Operating systems can set data execution prevention for stacks
- This prevents the program from executing machine language loaded on the stack by an exploit
- This does **not** prevent programs from overwriting the return address

Random Stack Location

- Microsoft Windows locates a programs stack at a random address since Windows Vista
- Each time the program is executed, the stack is at a different address
- Hackers cannot learn stack addresses from a previous execution of the program

Stack Protection

- Good programs should check all indexes to ensure they are within range
- Avoid functions that do not check limits, such as **`cin`**
- Java always checks array indexes

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	}
```

Stack Activation Frame 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 1	The first parameter to the method
parameter 2	The second parameter to the method

Analyzing Memory

The EBP register, points to the frame pointer in the current activation frame, contains 00012360

methodB	00801000
methodA	00801400
main	00801800

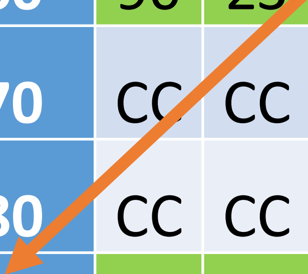
addr	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00012350	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
00012360	90	23	01	00	60	14	80	00	08	00	00	00	CC	CC	CC	CC
00012370	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
00012380	CC	CC	CC	CC	CC	CC	CC	CC	08	00	00	00	CC	CC	CC	CC
00012390	C0	23	01	00	70	18	80	00	03	00	00	00	05	00	00	00
000123A0	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC

Analyzing Memory

Frame pointer
Return address
parameter 2
parameter 1

methodB	00801000
methodA	00801400
main	00801800

addr	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00012350	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
00012360	90	23	01	00	60	14	80	00	08	00	00	00	CC	CC	CC	CC
00012370	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
00012380	CC	CC	CC	CC	CC	CC	CC	CC	08	00	00	00	CC	CC	CC	CC
00012390	C0	23	01	00	70	18	80	00	03	00	00	00	05	00	00	00
000123A0	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC



Method Calls without a Stack

- Some systems do not have stack hardware
- Method calls are generally done with an instruction that loads the return address into a register and jumps to the method

IBM 360 Function Calls

- IBM 360 and its descendants used a **B**ranch **A**nd **L**ink instruction

```
LA    R1, parameter list
BAL   R14, myFunc
```

- The registers were saved in a linked save area upon entry to the function

Univac 1100 Function Calls

- The Univac 1100 series had 36 bit registers and 18 address space
- Functions were called with a Load Modifier and Jump instruction that saved the return address in the upper half of a register

```
LMJ   X11, myFunc
```

address of parameter 1

address of parameter 2

next line of the program