COSC 366
Intro to Computer Security

Lecture 04
Software Security

Dr. Suya

Fall 2024

Today's Class

- Software security overview
- Refresher: function calls, memory layout

Overview

Why Software Security First

- Programs and their code are the basis of computing
- Most people today use off the shelf programs
- Programs are written by humans
- Flaws occur regularly or sporadically despite testing

What We Will Study

- Unintentional programming oversights

  - benign program flaws are often exploited for malicious impact

  - when this happens, which of CIA is compromised?

  - usually a stepping stone to something bigger

- Malicious programs - malware

Unintentional Programming Oversights

- Buffer overflow
- Other programming oversights
- Countermeasures

The Most Infamous: Buffer Overflow

- A buffer overflow is a bug that affects low-level language, typically C and C++
- A program with bug will normally just crash
  - In terms of CIA, what does it compromise?
- If under malicious attack, it can be exploited to
  - steal private information
  - corrupt valuable information
  - inject and execute code of the attacker's choice

What Is Buffer Overflow

- What is buffer

    - contiguous memory associated with a variable or field

    - e.g., when you type in something, it's held in the buffer before being processed

    - common in C: null-terminated strings that are arrays of chars

- What is buffer overflow

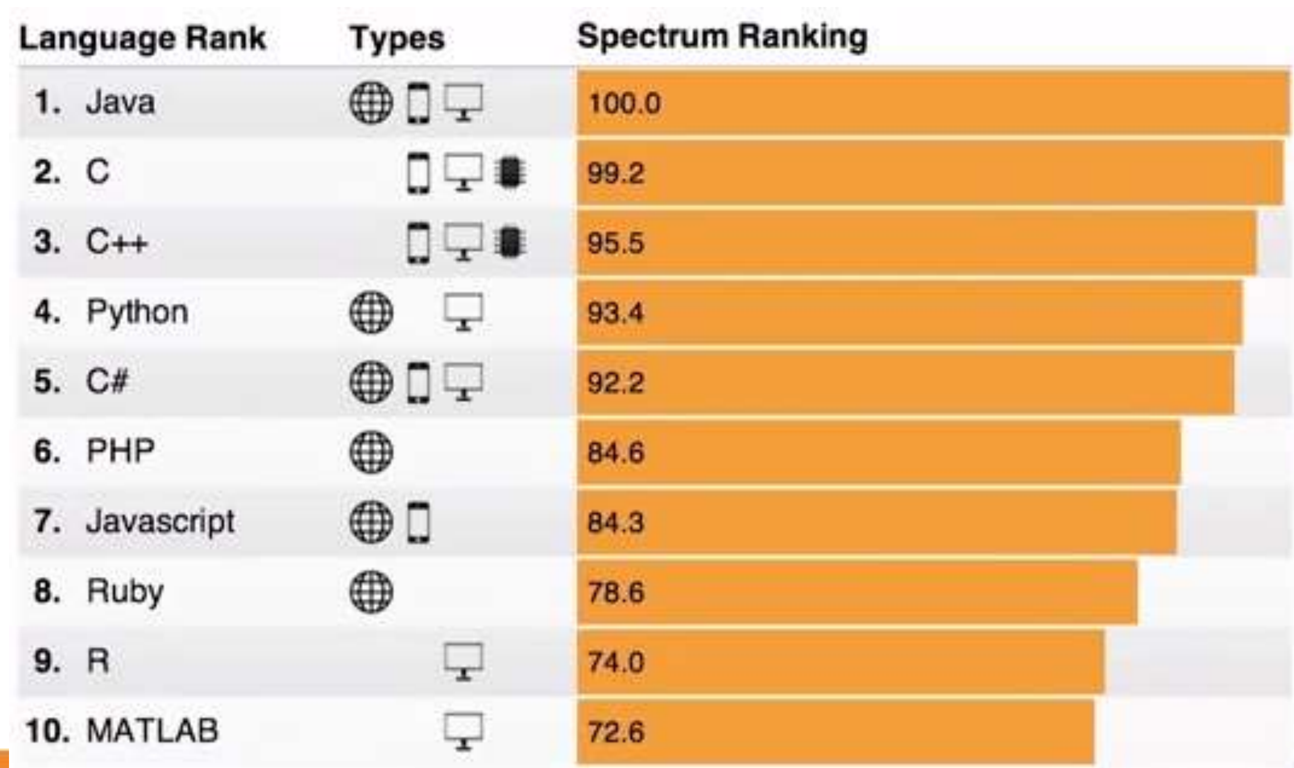
    - read/write more than a buffer can hold

- Where are the extra data go?

    - we will find out

Why Do We Study It

- It has a long history and gives a good lesson
- It is still very relevant today
  - C and C++ are still popular
  - buffer overflows still occur regularly



Critical Systems in C/C++

- Most OS kernels and utilities
  - fingerd, X window server, shell
- Many high-performance servers
  - Microsoft IIS, Apache httpd, nginx
  - Microsoft SQL server, MySQL, redis
- Many embedded systems
  - industrial control systems (e.g., SCADA), automobiles, airplanes, smartphones

History of Buffer Overflows

- 1988: Morris worm
  - 10% of the Internet (6,000 machines) infected
- 2001: CodeRed: exploited MS-IIS server
  - 300,000 machines infected in 14 hours
- 2003: SQL Slammer: exploited MS-SQL server
  - 75,000 machines infected in 10 minutes
- 2014: Heartbleed
  - 17% (half a million) secure web servers infected upon disclosure

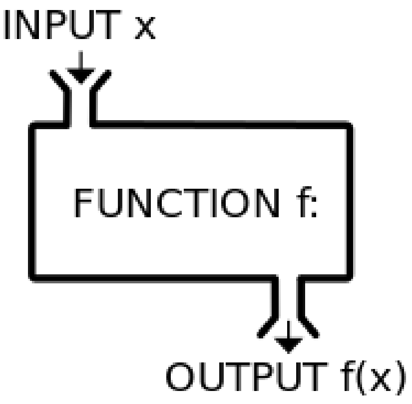
Refresher

- What are function calls?
- How is program data laid out in memory
- What does call stack look like
- What effect does calling (and returning from) a function have on memory?
- We will use x86 32-bit Linux processor model as example

FUNCTION

What's function?

- Assigns to each element of *X* exactly one element of *Y*
- A group of statements that together perform a task.
- Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.



Function

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);
   return 1;
}
```

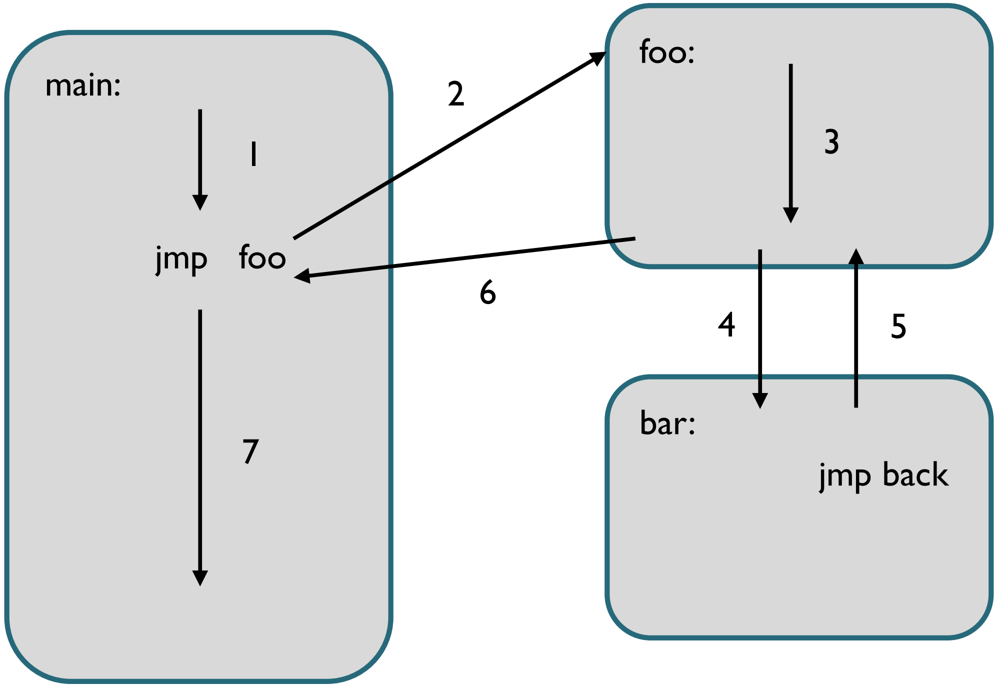
- Function name
  - Main
- Arguments
  - none
- Local variables
  - E.g., a, b
- Return address
  - Invisible
- Return value
  - 1

Function call/return

```
foo(...) {
      …
      bar();
      …
}
```

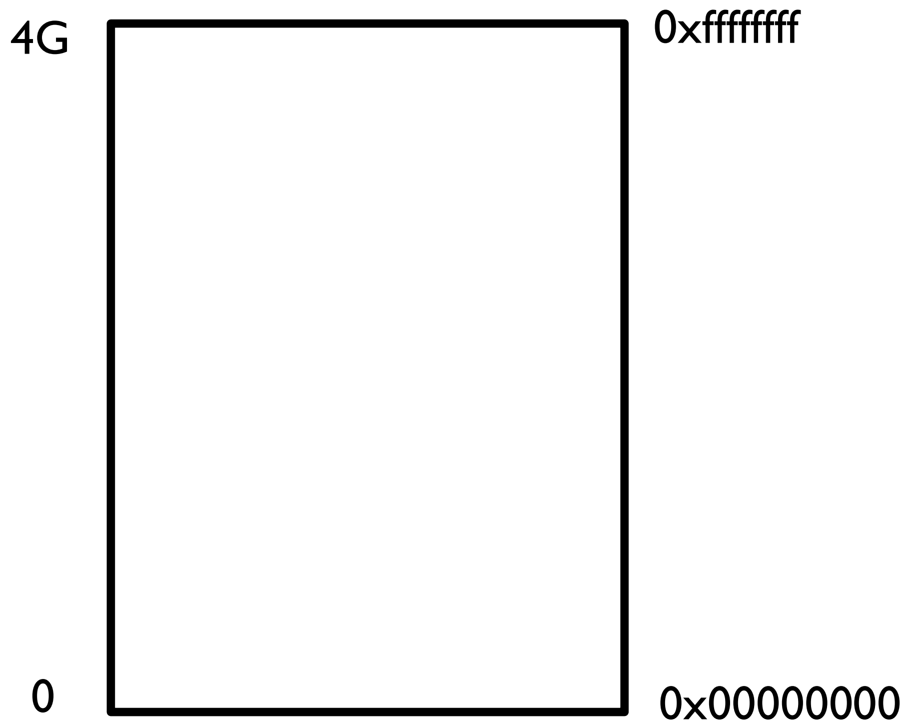
```
bar(...) {
      …
      …
}
```

```
main(...) {
      …
      foo(…);
      …
}
```



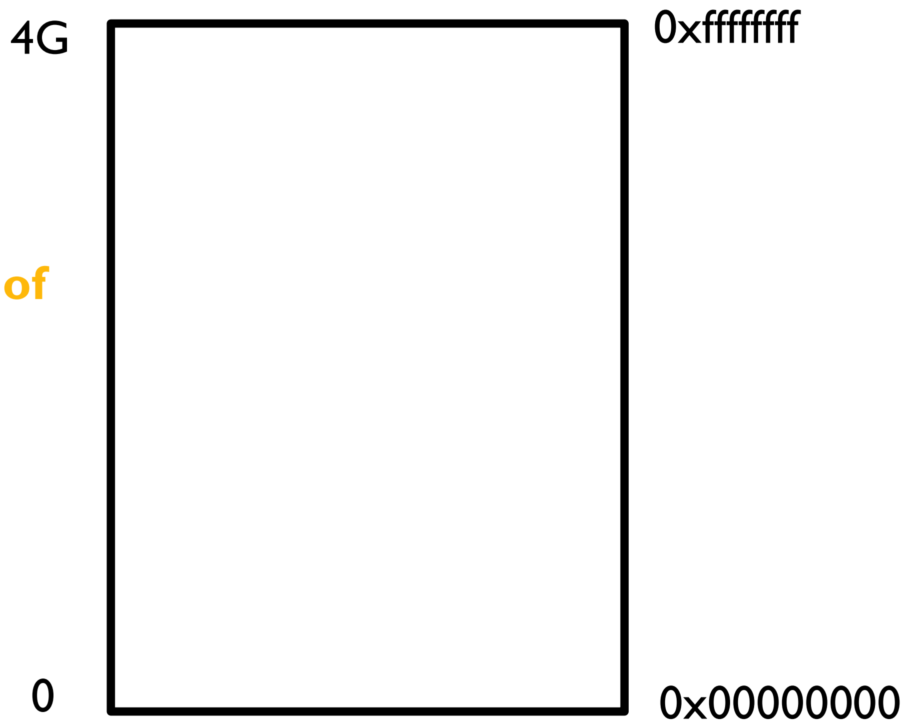
MEMORY LAYOUT

All programs are stored in memory



All programs are stored in memory

The process's view of memory is that it owns all of it



Can the 32-bit system have more than this memory space?

Wait!

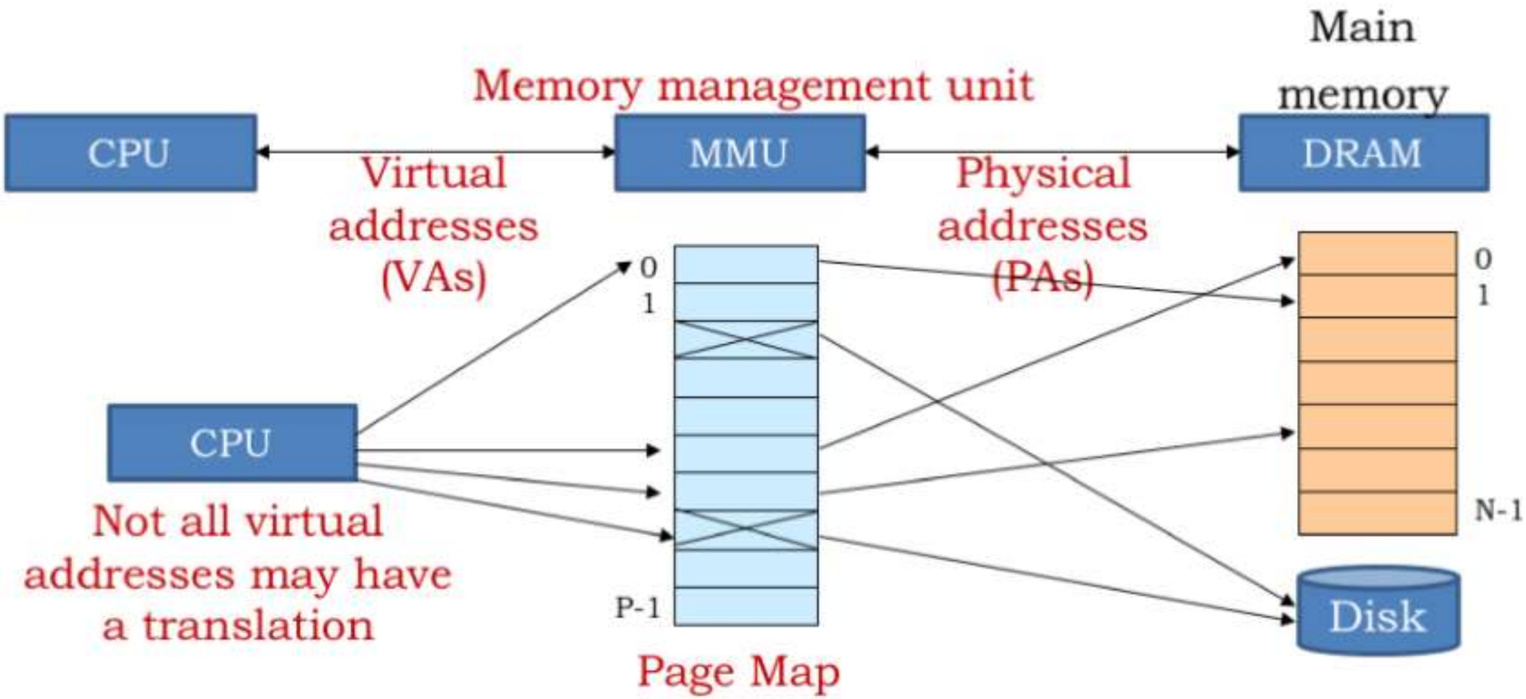
- How would it be possible for two programs to run at the same time on your Windows or MacOS?
  - May conflict your program with other programs
  - You have a limited memory like 4GB, your program needs more memory space than 4GB.
  - How can we overcome this challenge?

Virtual Memory

- Freeing applications from having to manage a shared memory space.
  - You don't worry about managing memory (at low level) when programming → Process isolation, simplifying application writing, simplifying compilation, linking, loading

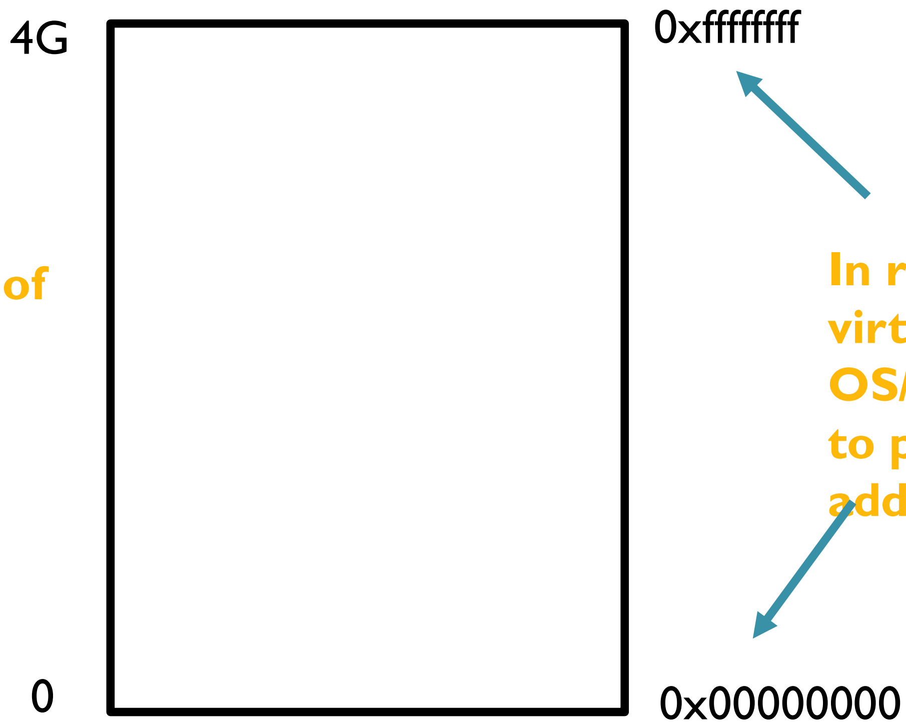
- Able to conceptually use more memory than might be physically available

Virtual Memory



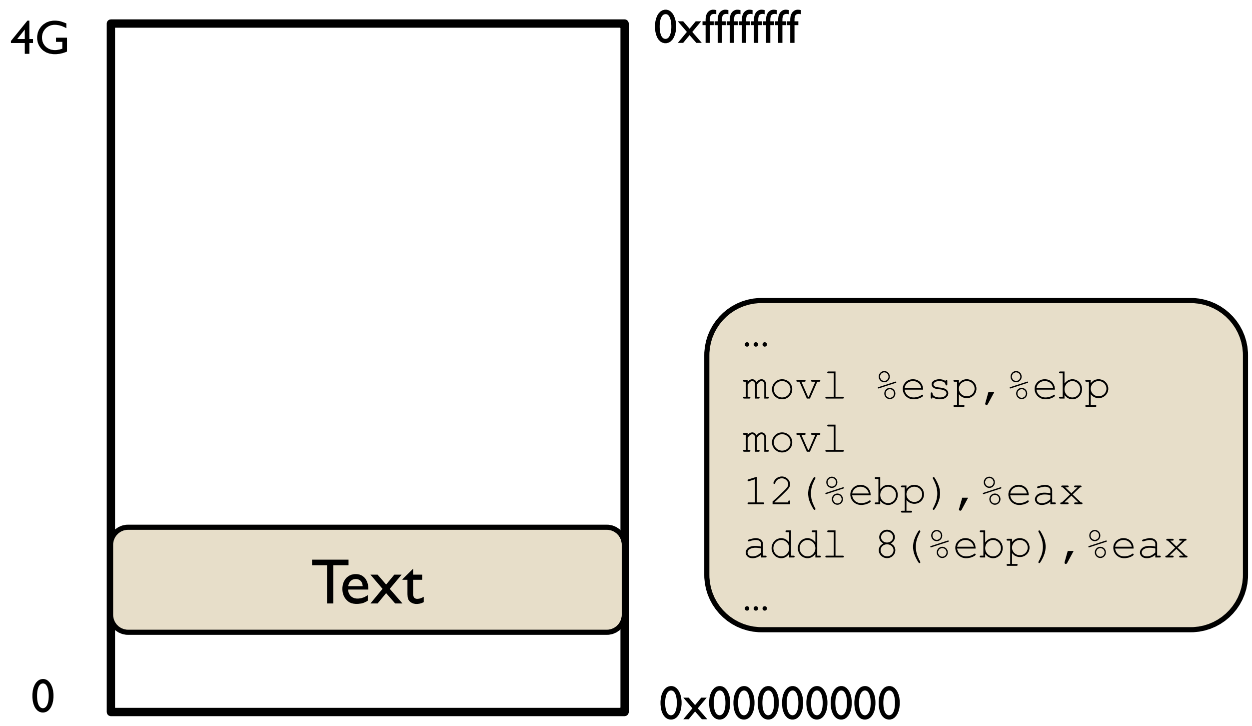
All programs are stored in memory

The process's view of memory is that it owns all of it

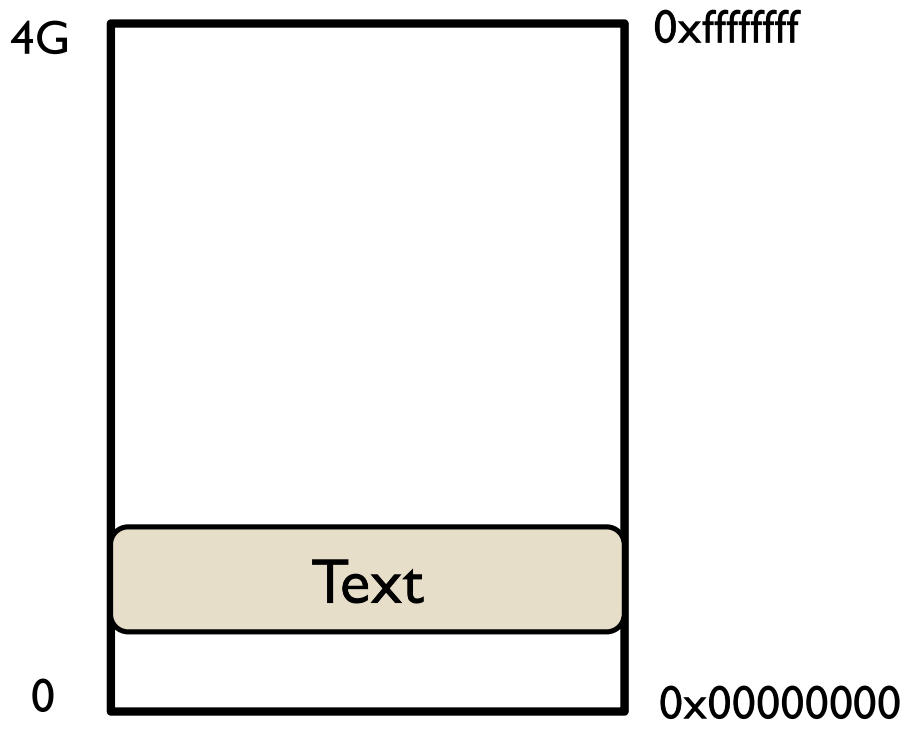


In reality, these are virtual address; the OS/CPU map them to physical addresses.

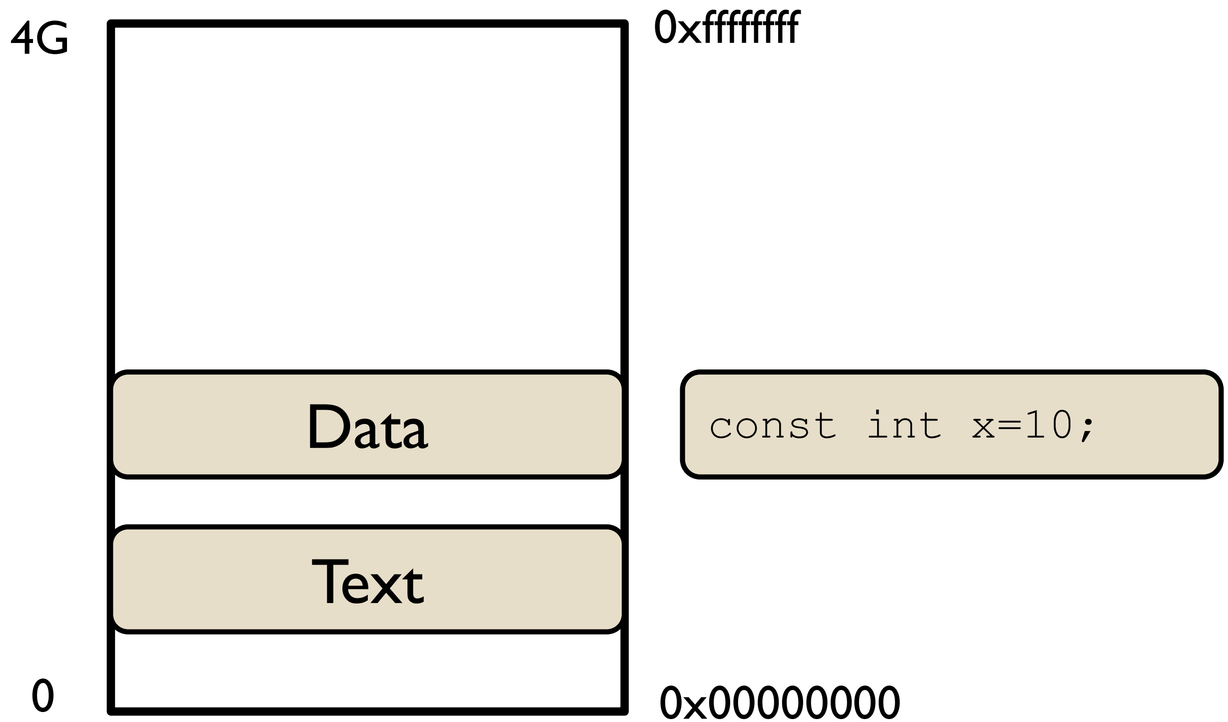
The instructions are stored in memory



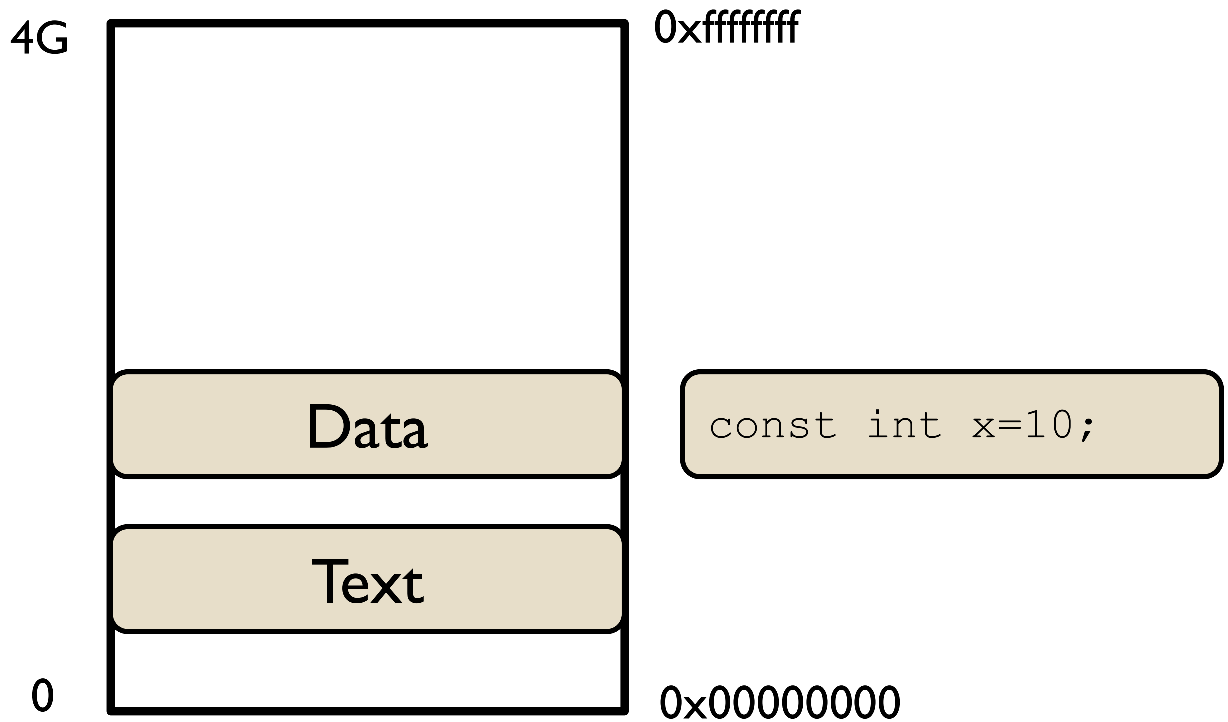
The instructions are stored in memory



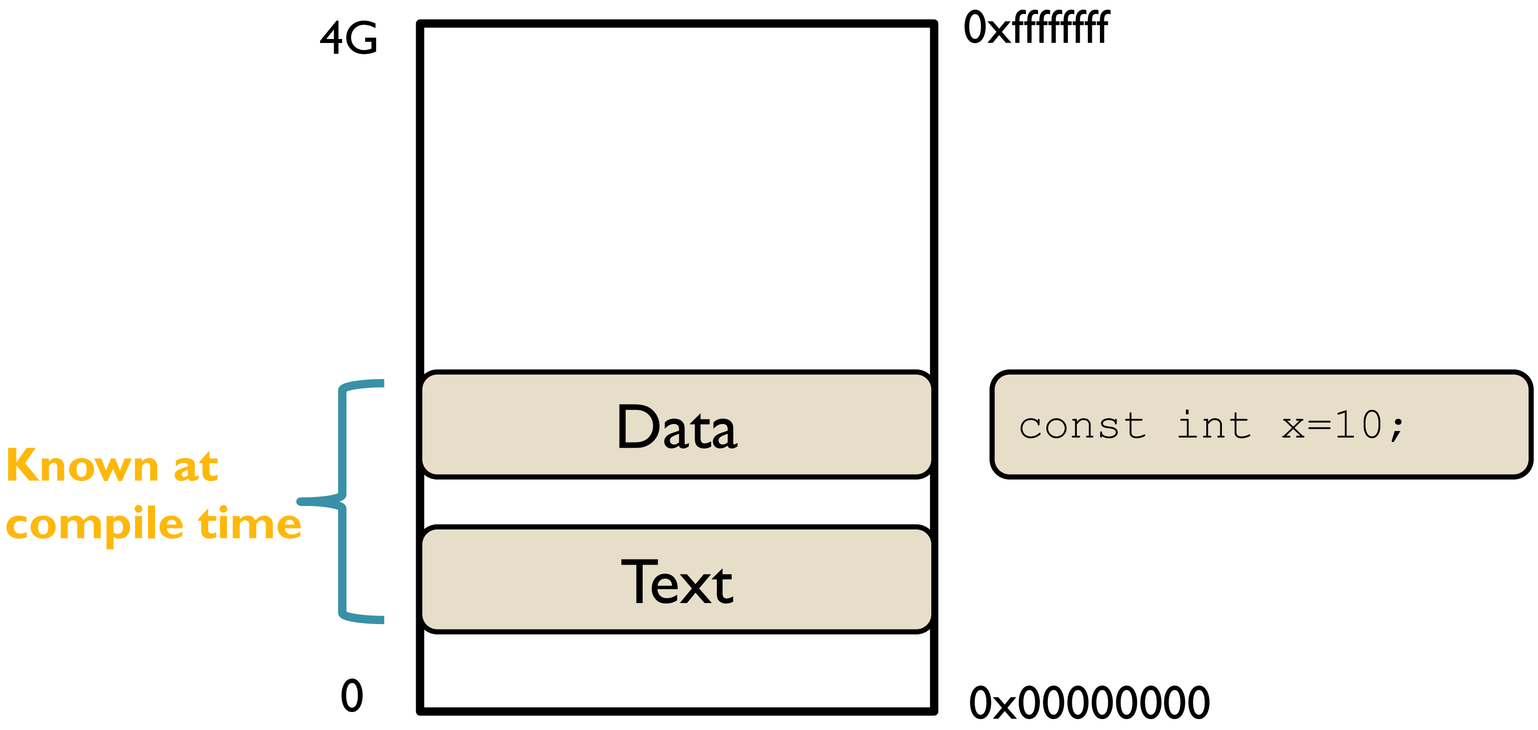
Data are stored in memory



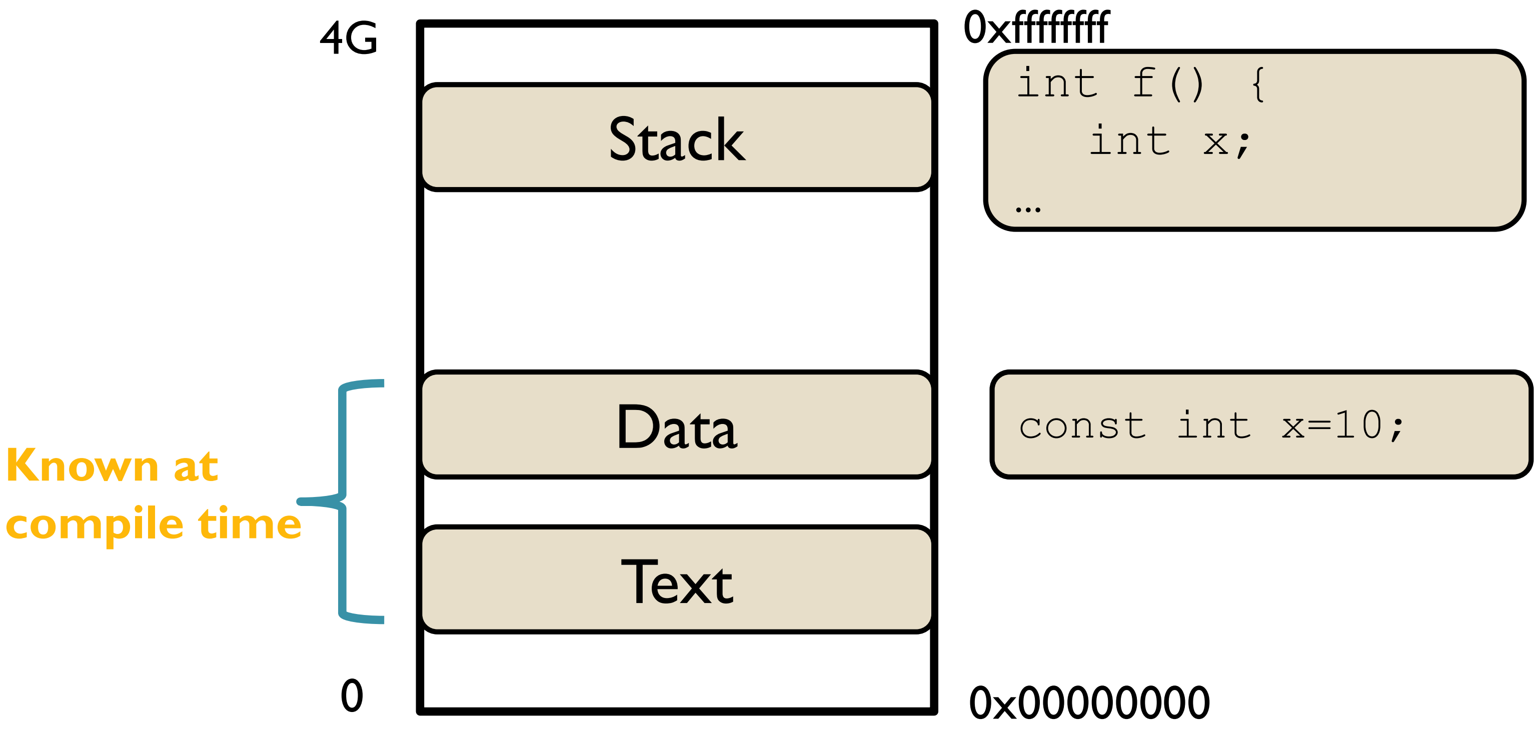
Data are stored in memory



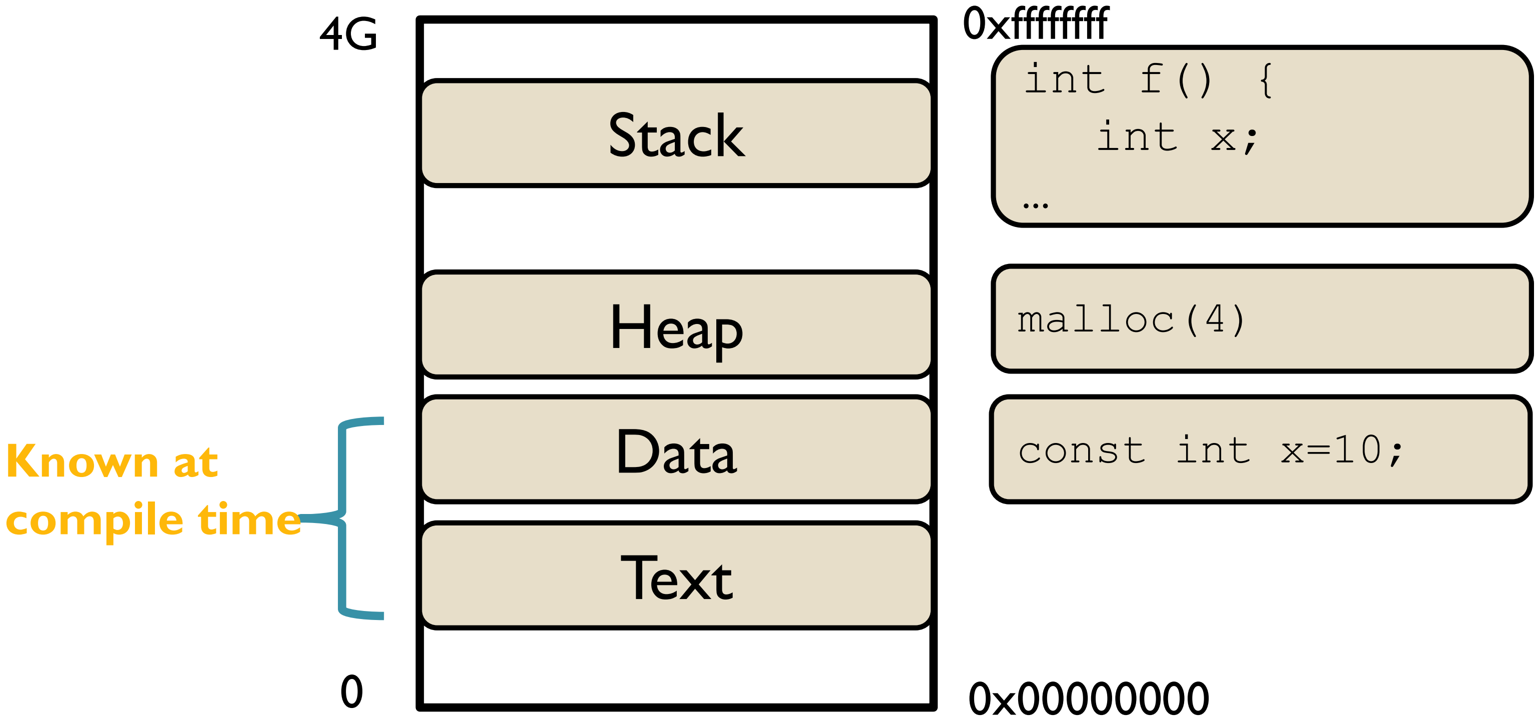
Data are stored in memory



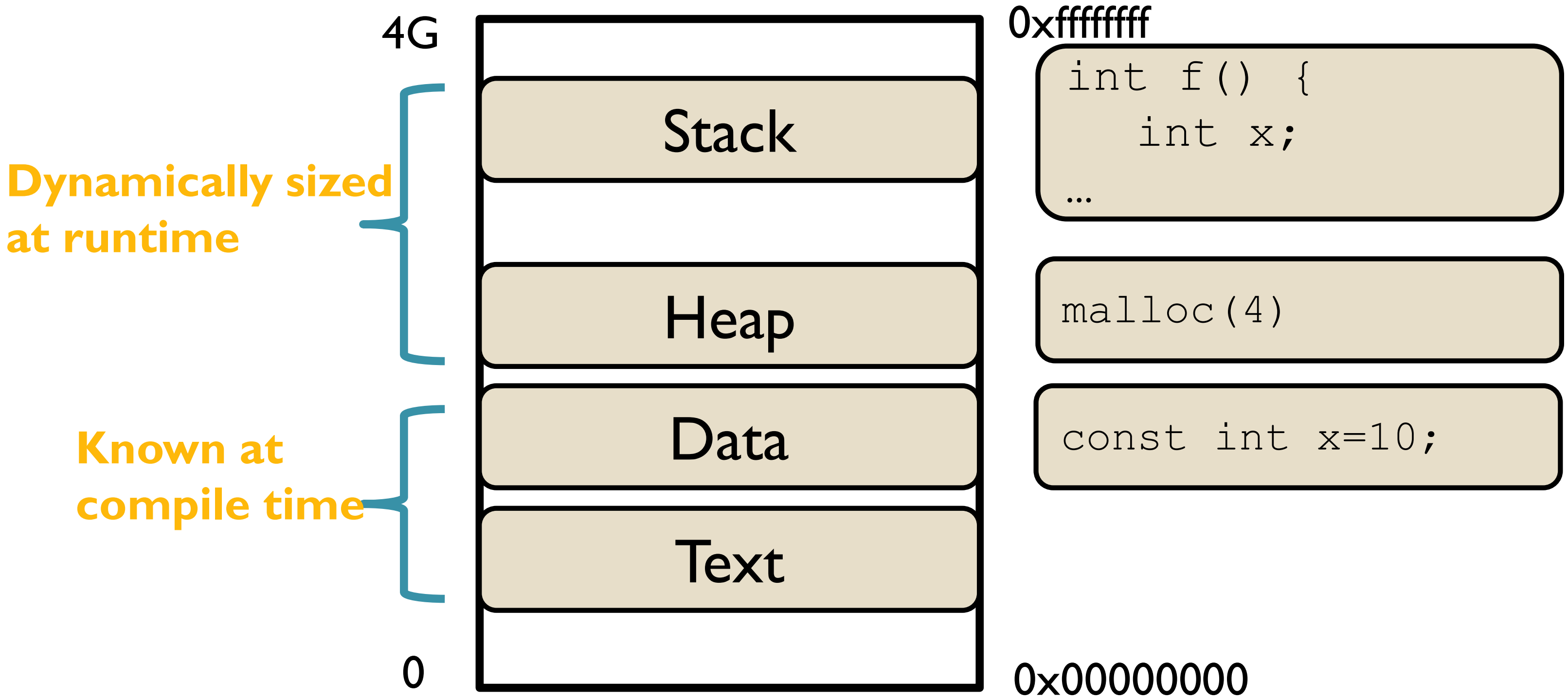
Stack (Local variables)



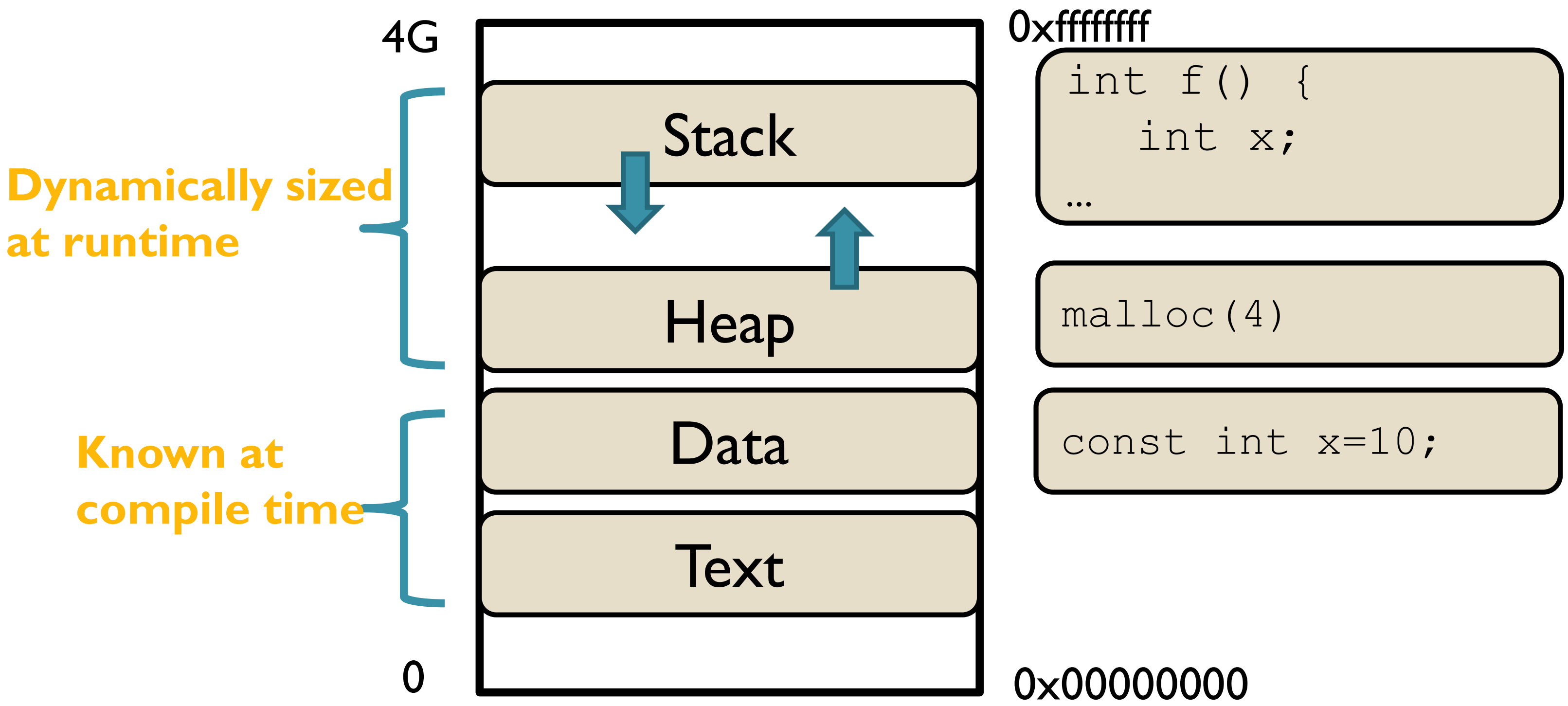
Heap (Dynamic memory)



Heap (Dynamic memory)



Stack & Heap grow in opposite directions



Program Memory Stack

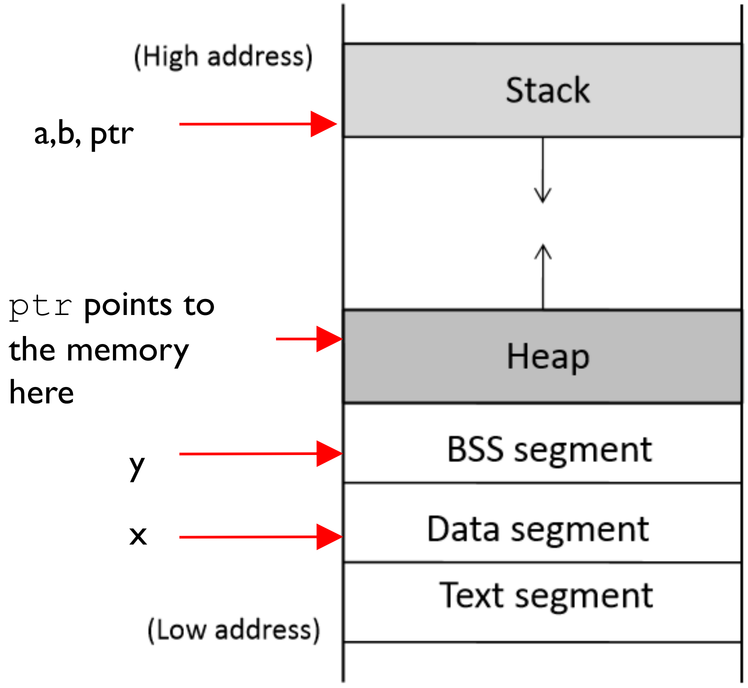
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```

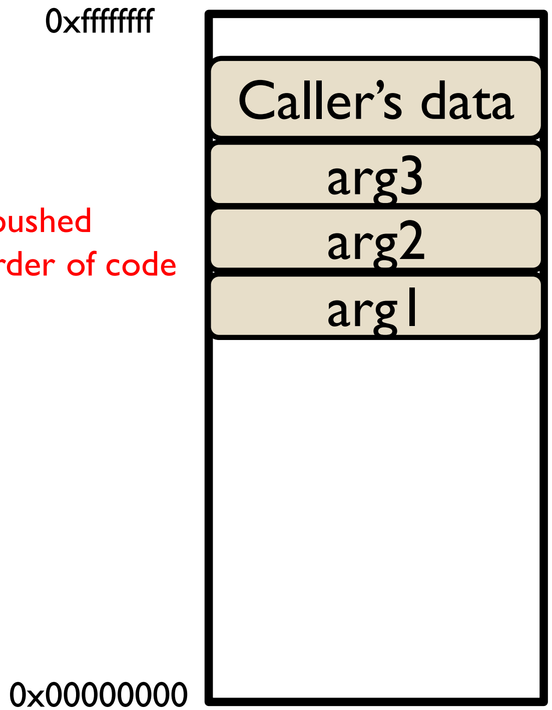


STACK LAYOUT

Stack layout when calling function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

Arguments pushed
in reverse order of code

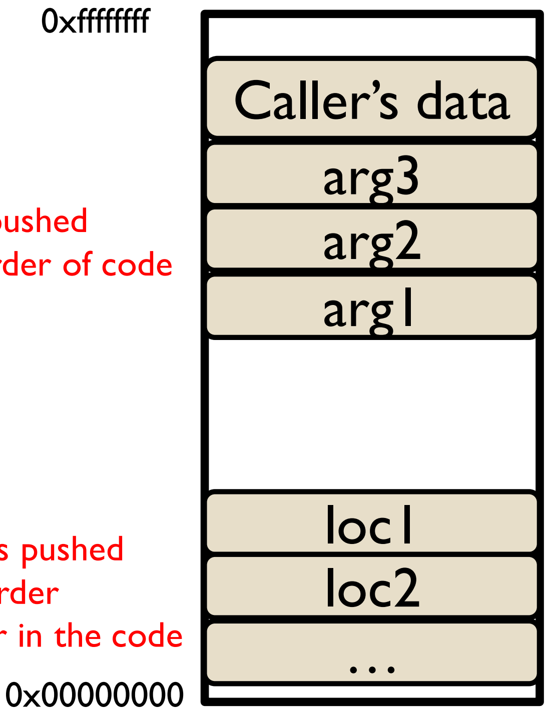


Stack layout when calling function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

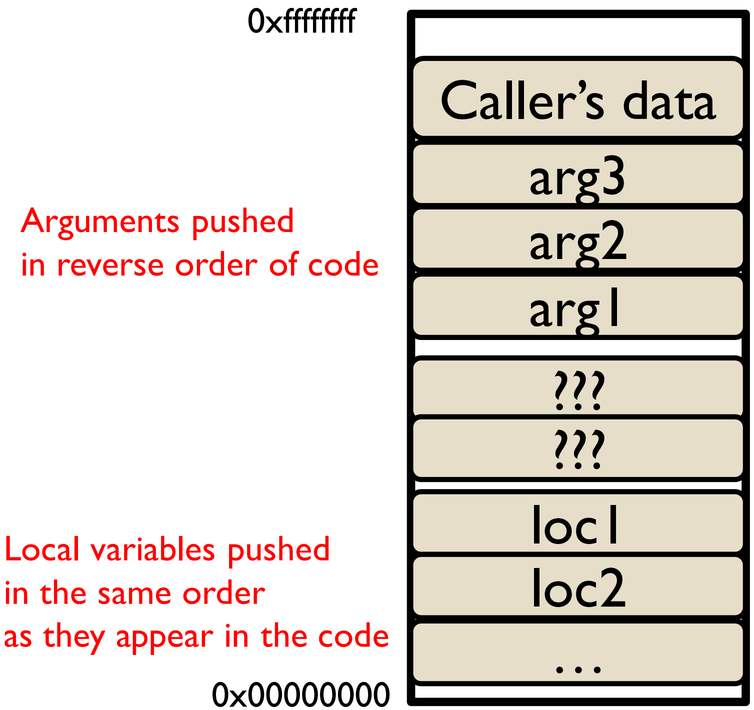
Arguments pushed
in reverse order of code

Local variables pushed
in the same order
as they appear in the code



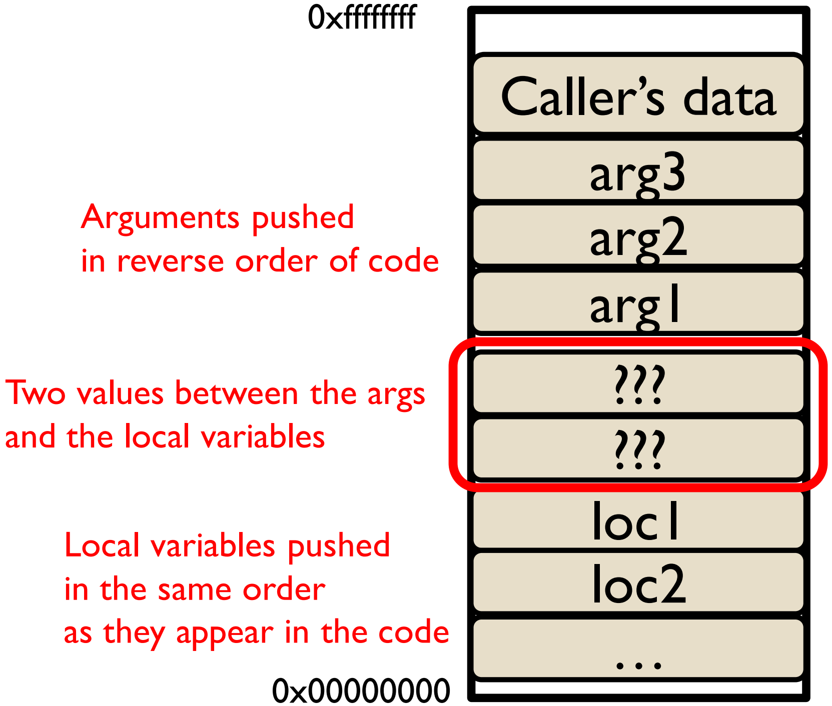
Stack layout when calling function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



Stack layout when calling function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

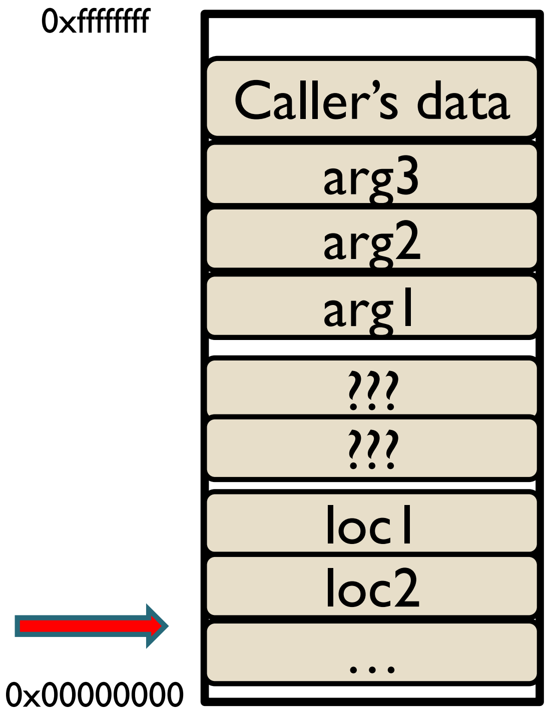


EBP (EXTENDED BASE POINTER)

What's the addr. of loc2?

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

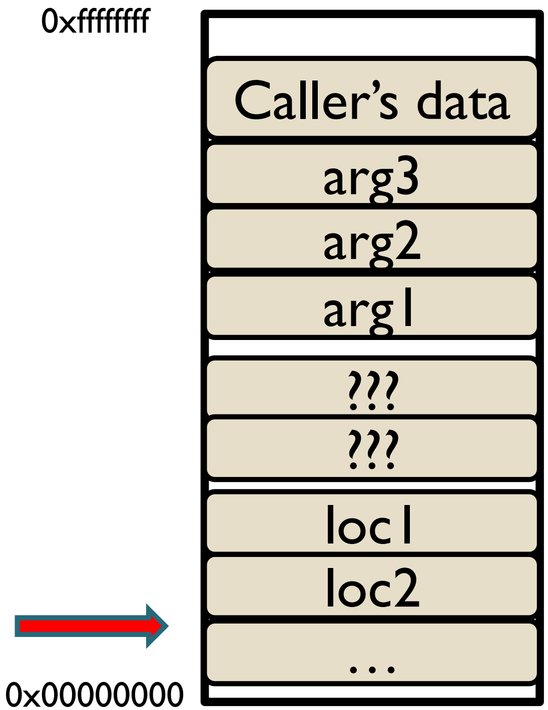
Q) Where is loc2?
What's the specific address?



What’s the addr. of loc2?

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

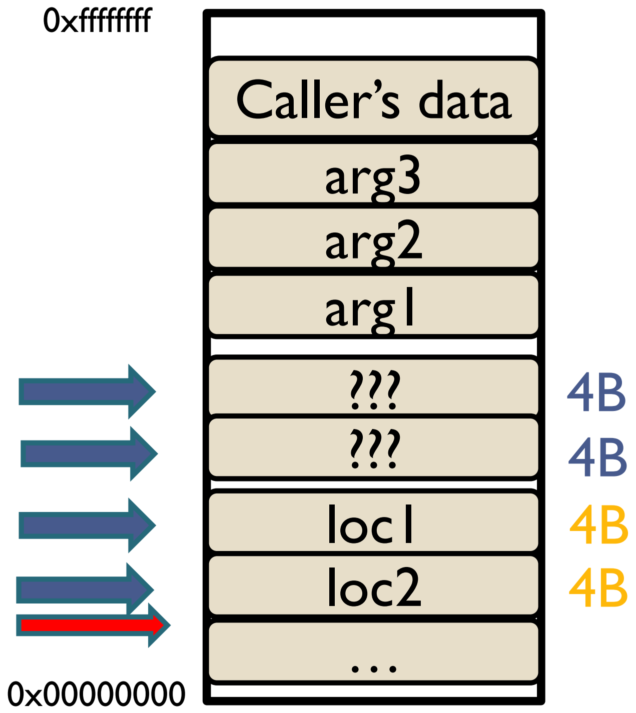
Q) Where is loc2?
What’s the specific address?
A) We don’t know before running
since undecidable at compile time



What's the addr. of loc2?

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

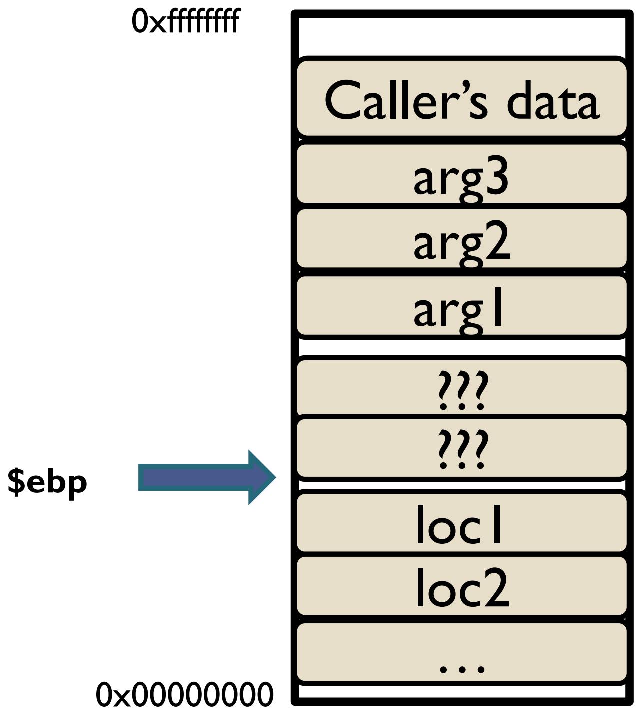
Q) Where is loc2?
What's the specific address?
A) But we can know loc2 is always
8bytes before "???"s ➔ addr of ??? - 8B



EBP (Base Pointer)

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

Q) Where is loc2?
What's the specific address?
A) But we can know loc2 is always 8bytes before "???"s ➔ addr of ??? - 8B

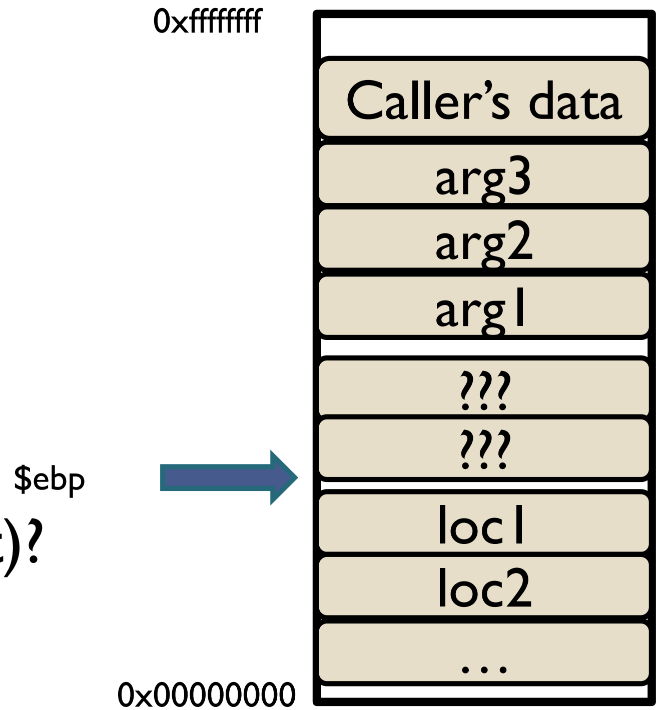


EBP (Base Pointer): Notation

- %ebp: A memory address
- (%ebp): The value at memory address %ebp (like dereferencing a pointer)

EBP (Base Pointer)

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

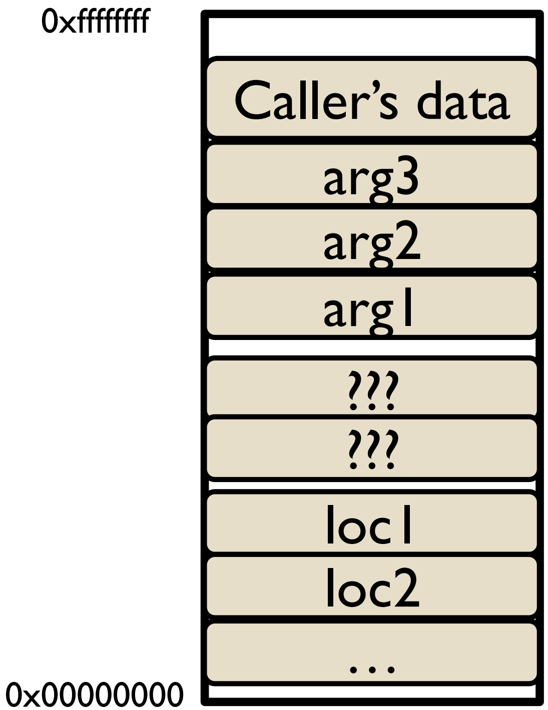


Q) where is loc2?
What's the specific address (and content)?

A) -8 (%ebp)

Stack layout when calling function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

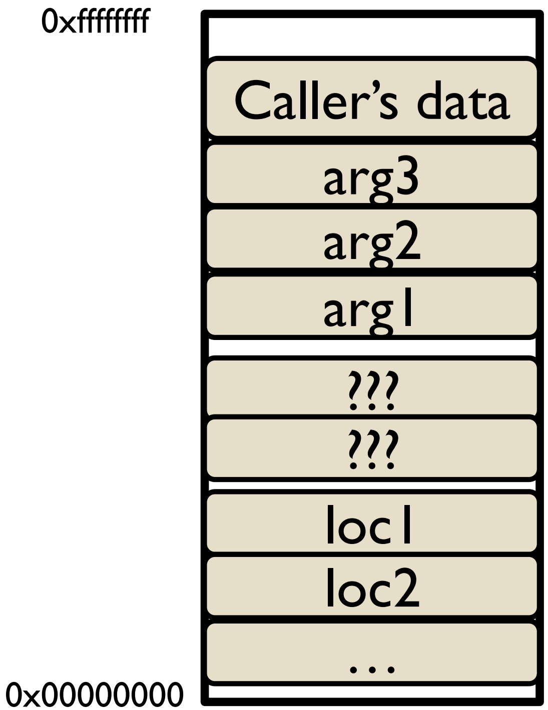


Q) What are "???"?

First, we need $ebp

Stack layout when calling function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



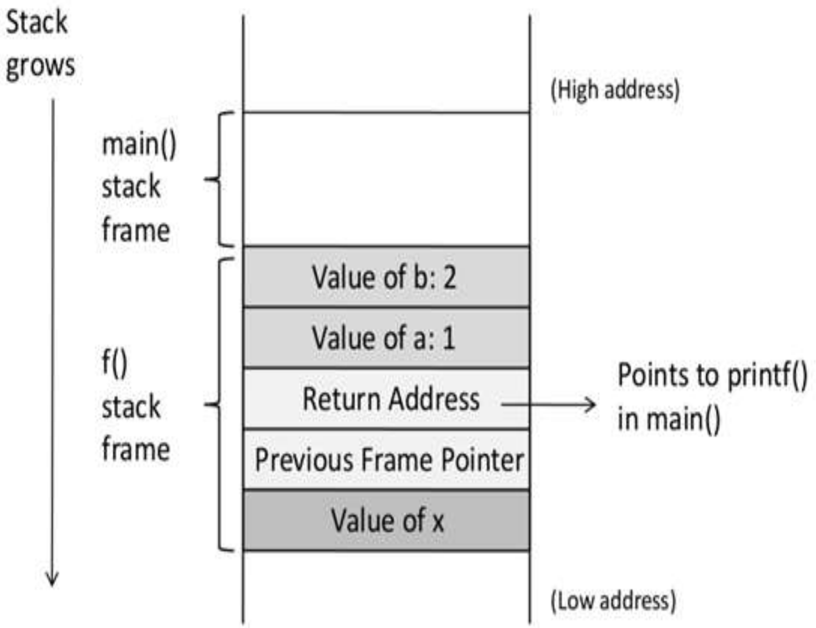
Q) What are "???" ?

First, we need $ebp

Second, we need a return address

Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



Order of the function arguments in stack

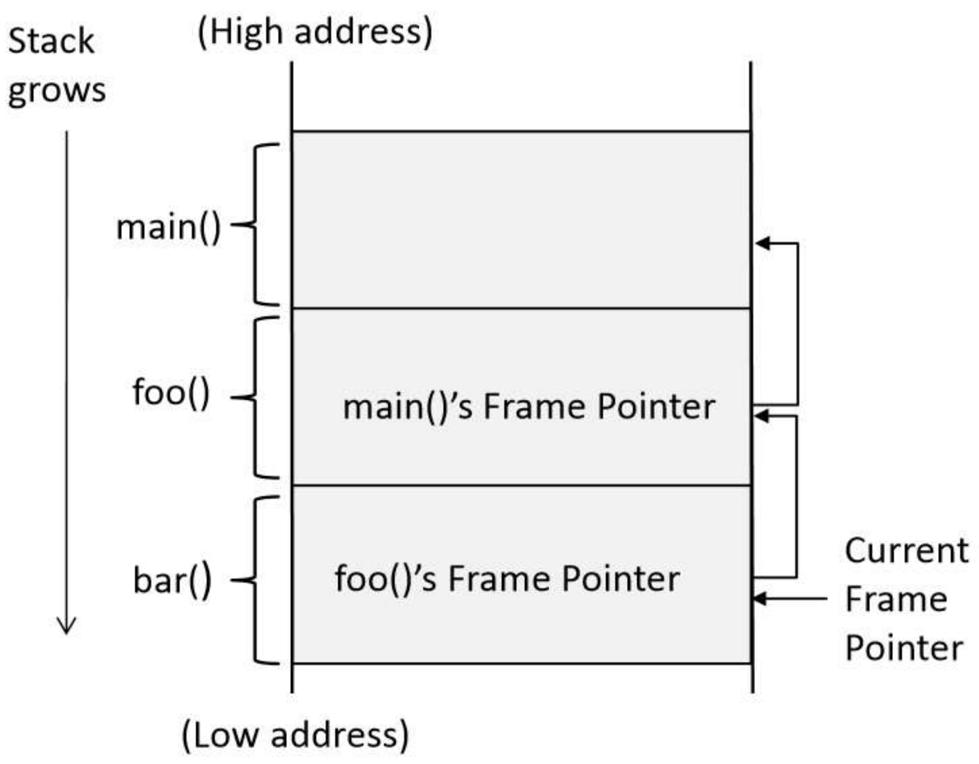
```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

Can you tell why are +12, +8 and -8 listed here?

```
movl     12(%ebp), %eax      ; b is stored in %ebp + 12
movl     8(%ebp), %edx       ; a is stored in %ebp + 8
addl     %edx, %eax
movl     %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Stack Layout for Function Call Chain



main()

foo()

bar()

Heap

```
int x = 100; // In Data segment
int main() {
    int a = 2; // In Stack
    float b = 2.5;// In Stack
    static int y;// In BSS

   // Allocate memory on Heap
    int *ptr = (int *) malloc(2*sizeof(int));
   // values 5 and 6 stored on heap
   ptr[0] = 5;// In Heap
   ptr[1] = 6;// In Heap
   free(ptr);
   return 1;
}
```

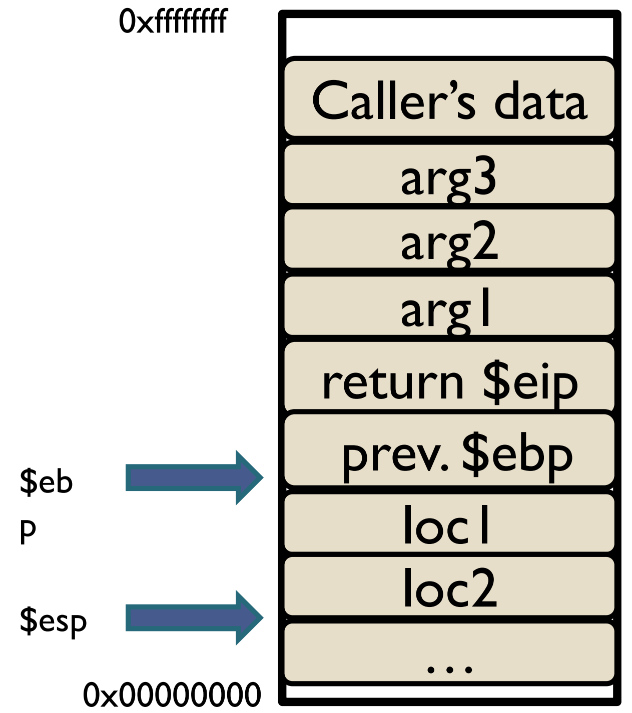
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave:  mov %esp %ebp
        pop %ebp
ret:    pop %eip
```



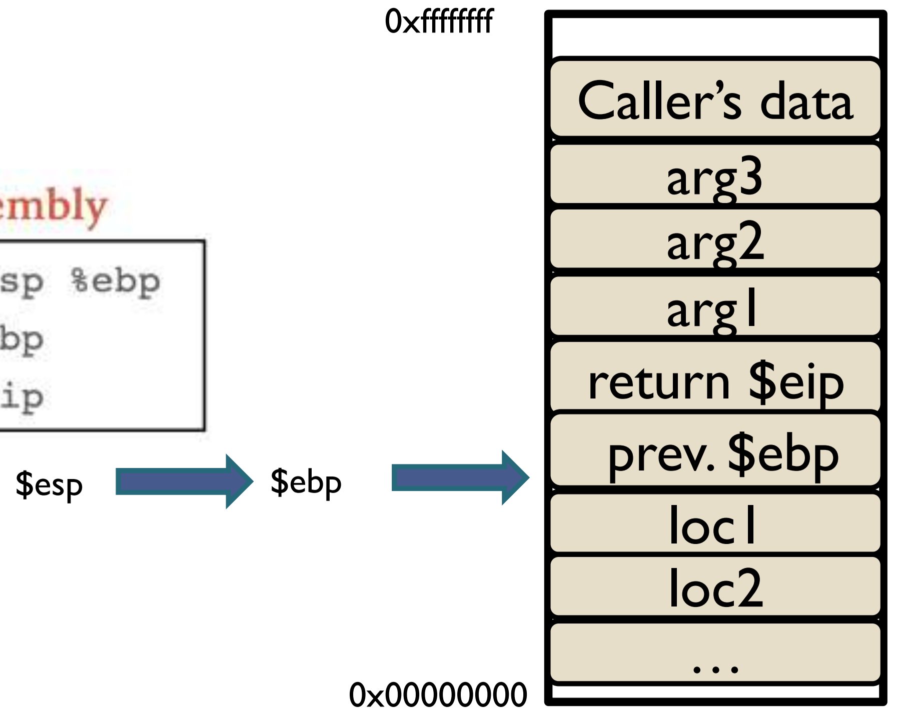
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave: mov %esp %ebp
       pop %ebp
ret:   pop %eip
```



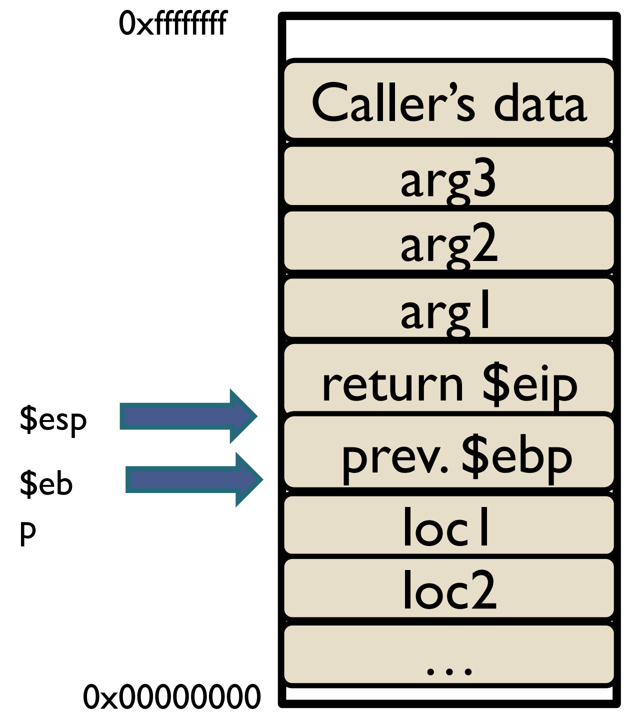
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave:    mov %esp %ebp
        ➜ pop %ebp
ret:      pop %eip
```



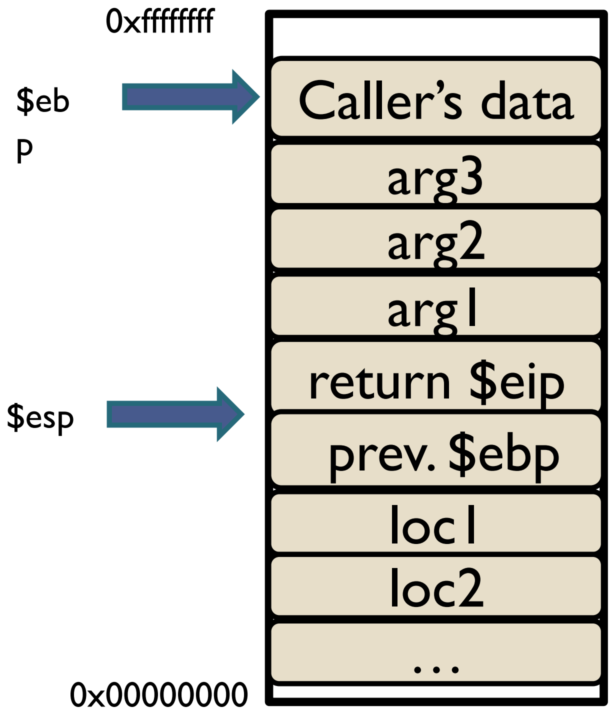
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave:   mov %esp %ebp
       ➜ pop %ebp
ret:     pop %eip
```



Returning from functions

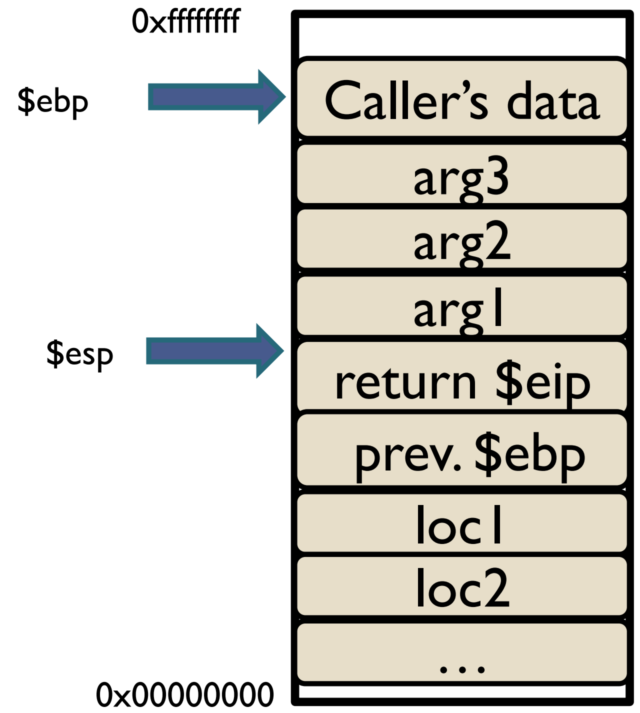
In C

```
return;
```

In compiled assembly

```
leave:  mov %esp %ebp
        pop %ebp
ret:  ➔ pop %eip
```

1. The next instruction is to “remove” the arguments off the stack
2. And now we’re back where we started



Stack & functions: Summary

Calling function (before calling):

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Stack & functions: Summary

Calling function (before calling):

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

1. **Push the old frame pointer** onto the stack: push %ebp
2. **Set frame pointer** %ebp to where the end of the stack is right now: %ebp=%esp
3. **Push local variables** onto the stack; access them as offsets from %ebp

Stack & functions: Summary

Calling function (before calling):

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

1. **Push the old frame pointer** onto the stack: push %ebp
2. **Set frame pointer** %ebp to where the end of the stack is right now: %ebp=%esp
3. **Push local variables** onto the stack; access them as offsets from %ebp

Called function (when returning)

1. **Reset the previous stack frame**: %esp = $ebp; pop %ebp
2. **Jump back to return address**: pop %eip