Software Security
COSC 466/566
Spring 2023

Dr. Doowon Kim

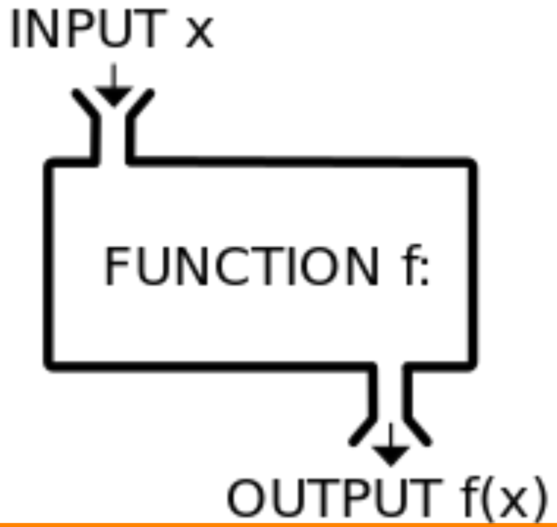
THE UNIVERSITY OF
TENNESSEE

Today’s class

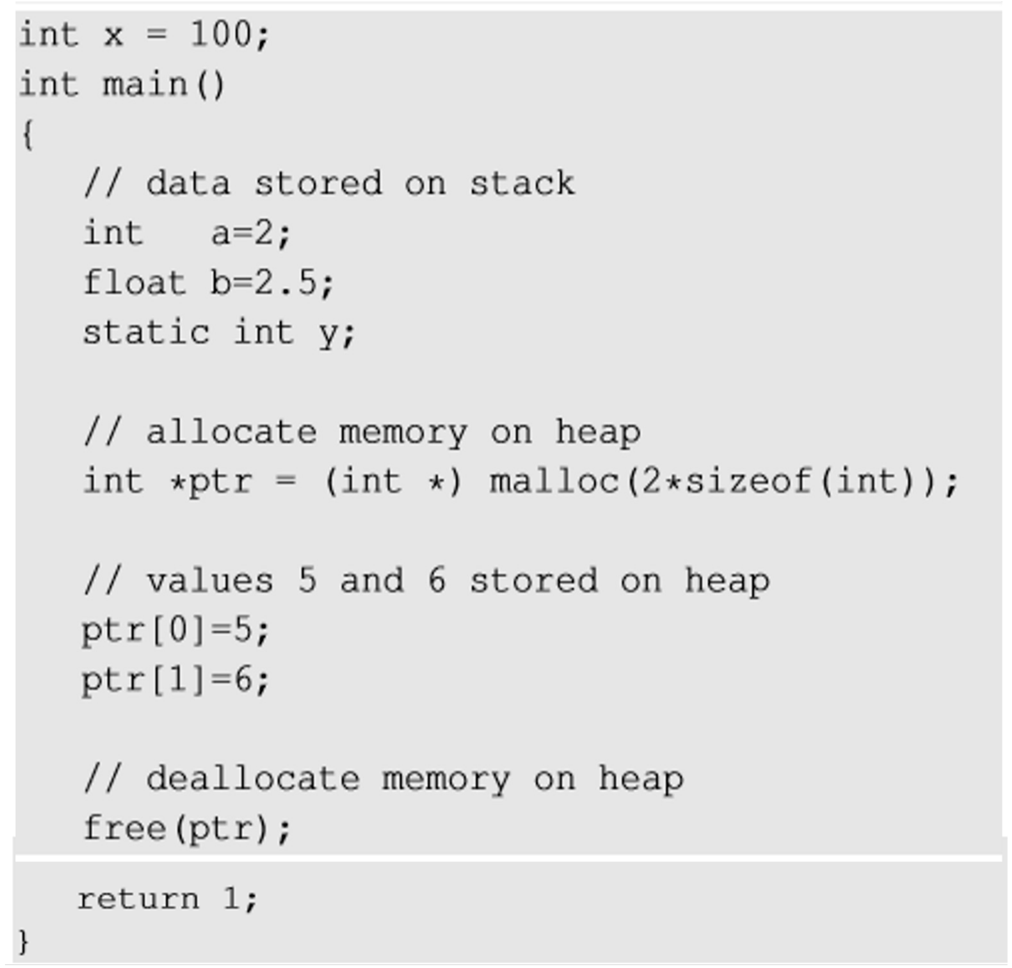
Function

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;
}
```

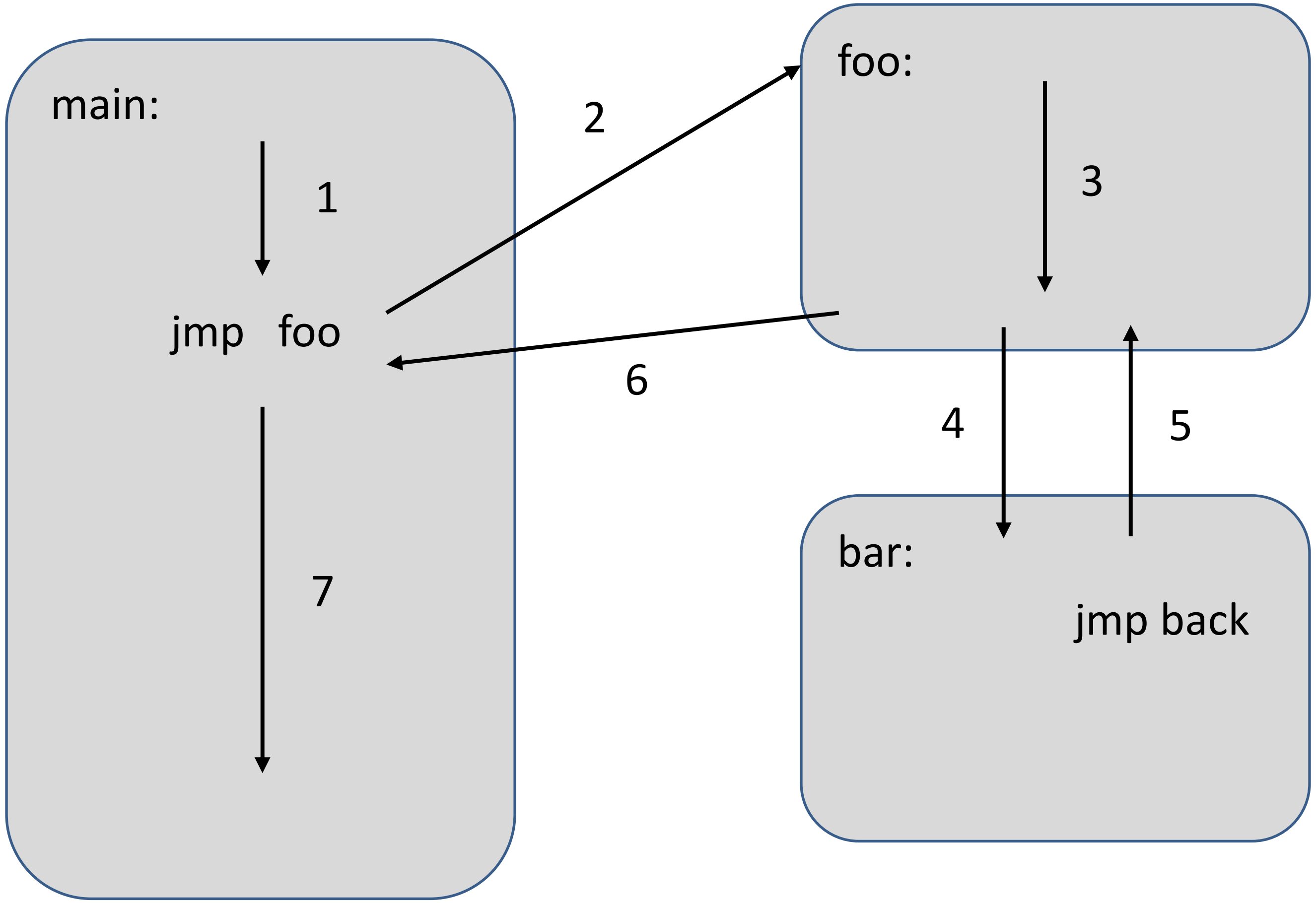
- Function name
  - Main
- Arguments
  - none
- Local variables
  - E.g., a, b
- Return address
  - Invisible
  - this parameter is passed automatically when the function is called
  - the function needs to be able to get back to wherever it was called from
- Return value
  - 1

Function call/return

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

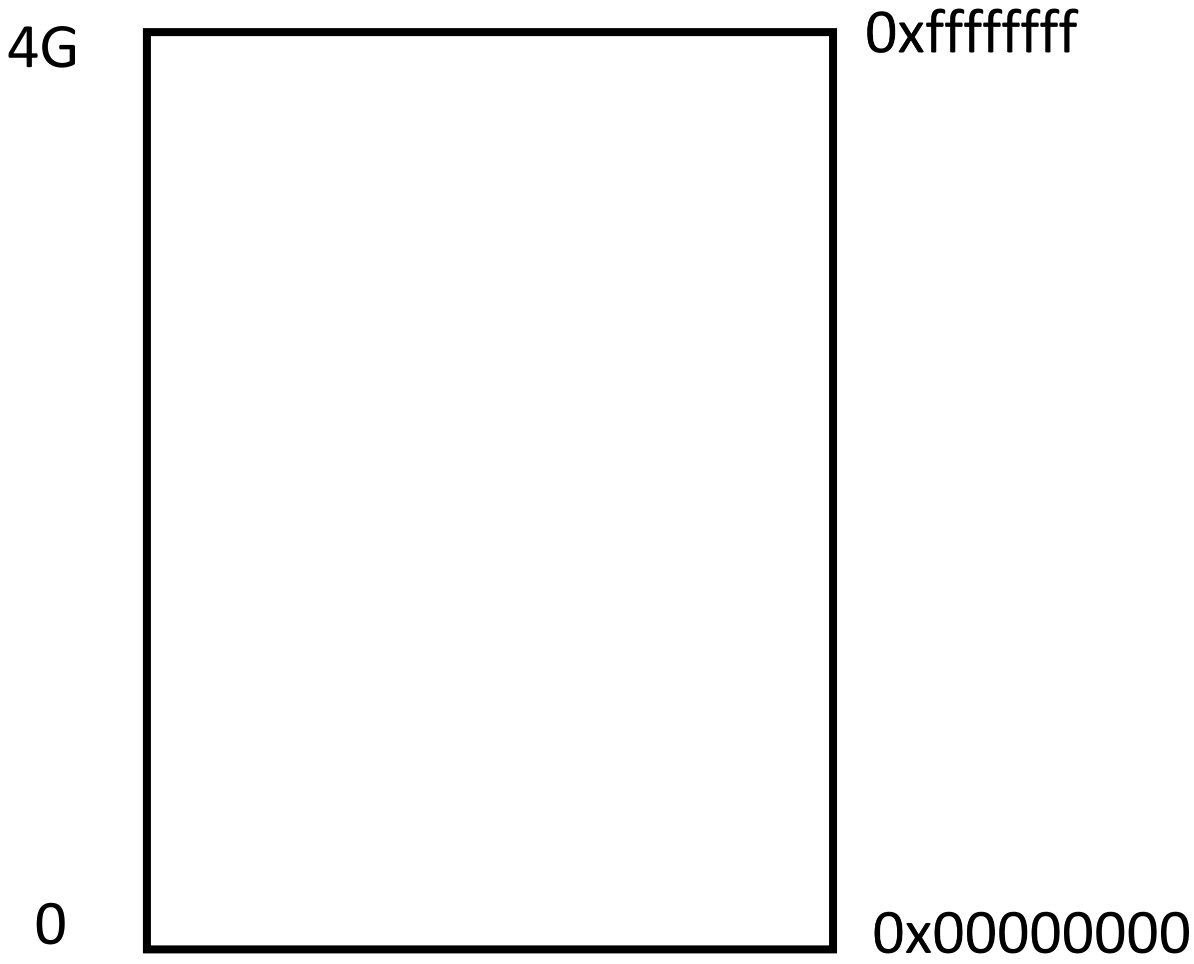
```
bar(…) {
      …
      …
}
```

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



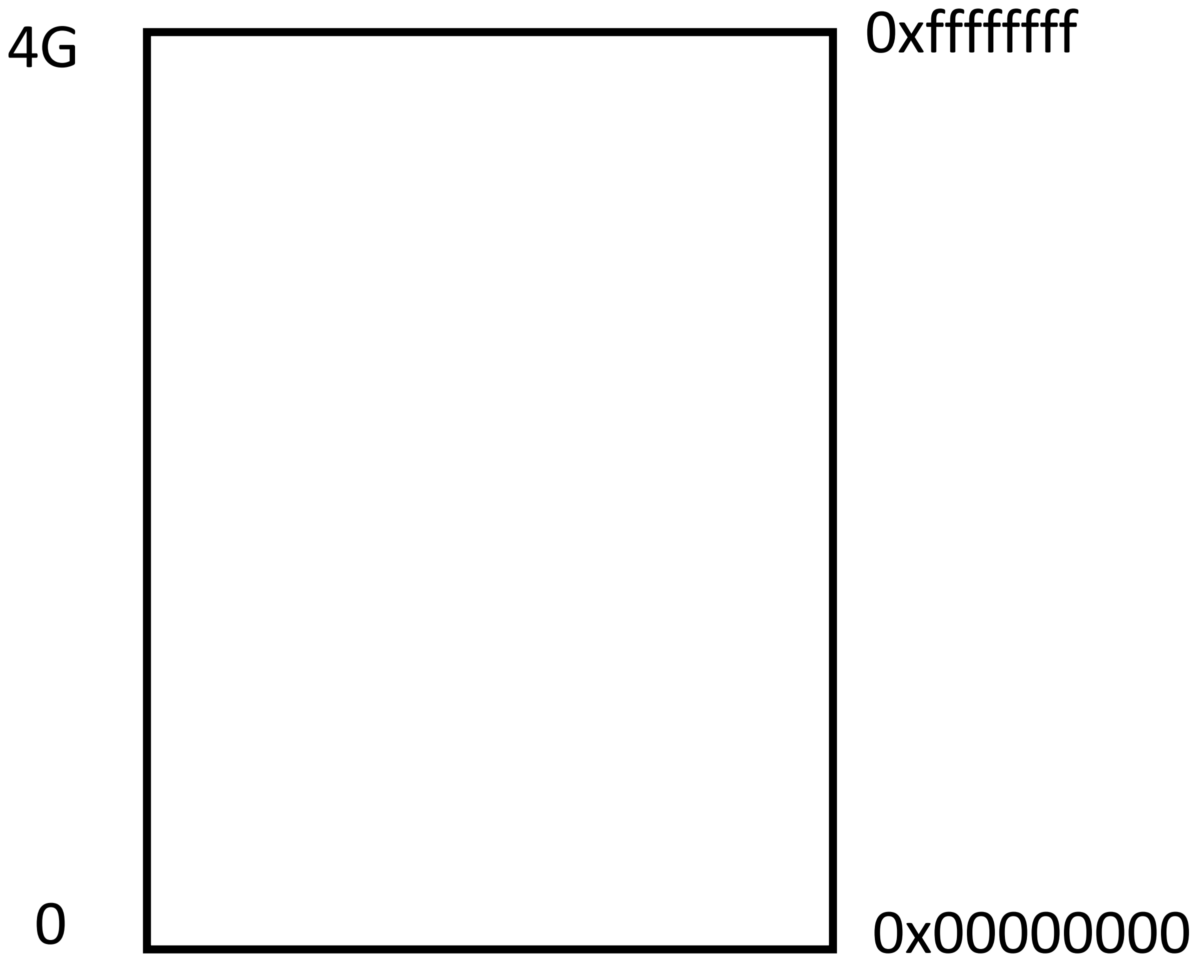
Memory Layout

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



Wait!

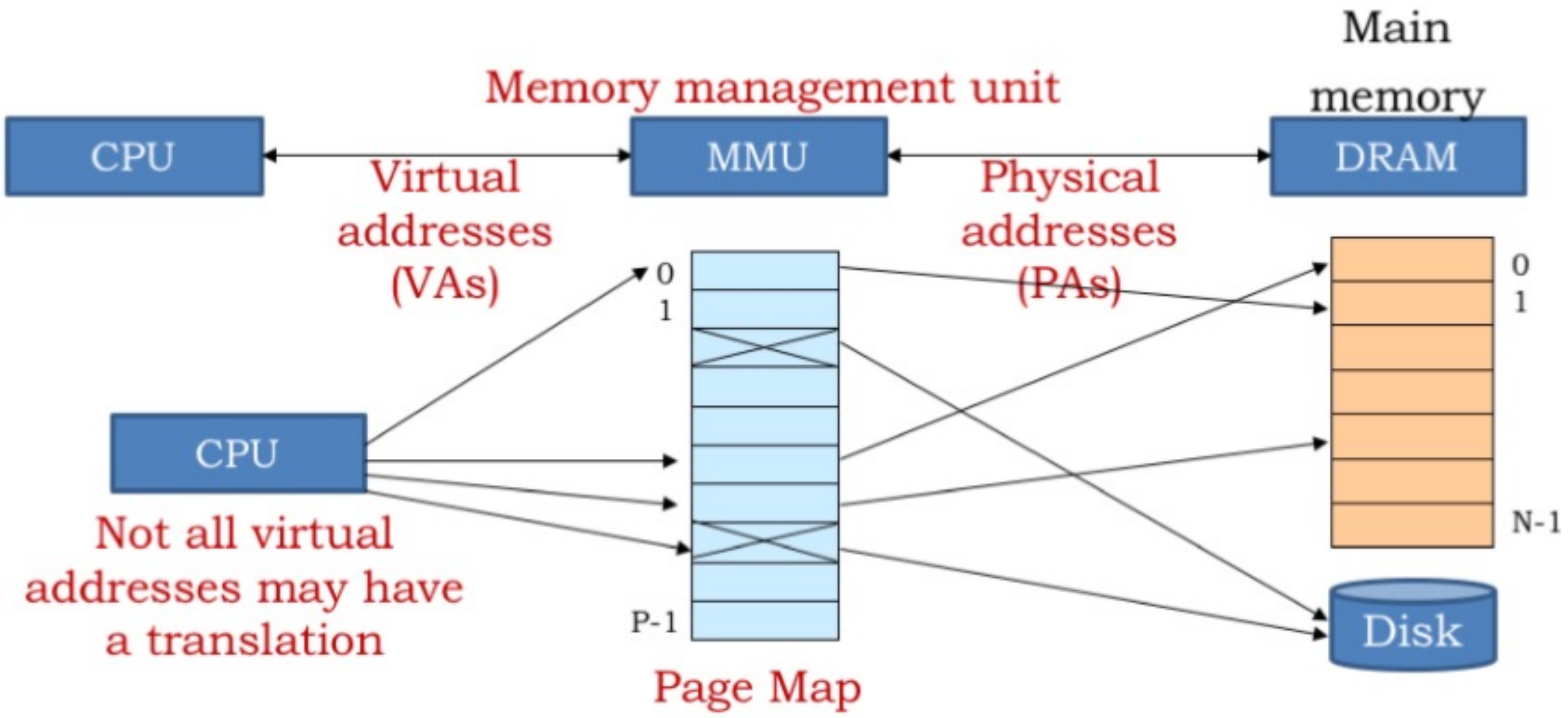
- How would it be possible for two programs to run at the same time on your Windows or MacOS?
- May conflict your program with other programs
- You have a limited memory like 4GB, your program needs more memory space than 4GB.
- How can we overcome this challenge?

Virtual Memory

- Freeing applications from having to manage a shared memory space. You don't worry about managing memory when programming. ➔ Process isolation, Simplifying application writing, Simplifying compilation, linking, loading

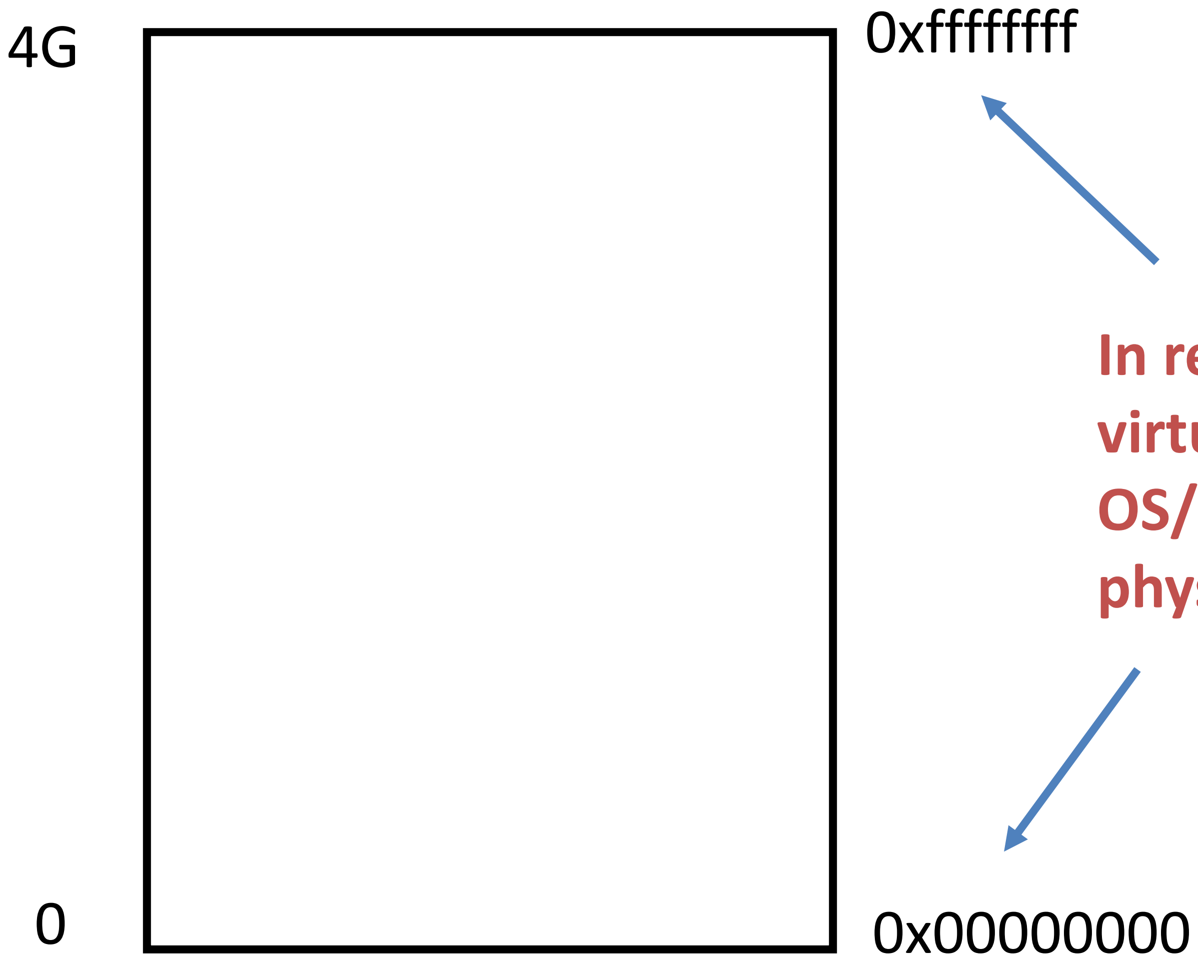
- Able to conceptually use more memory than might be physically available

Virtual Memory



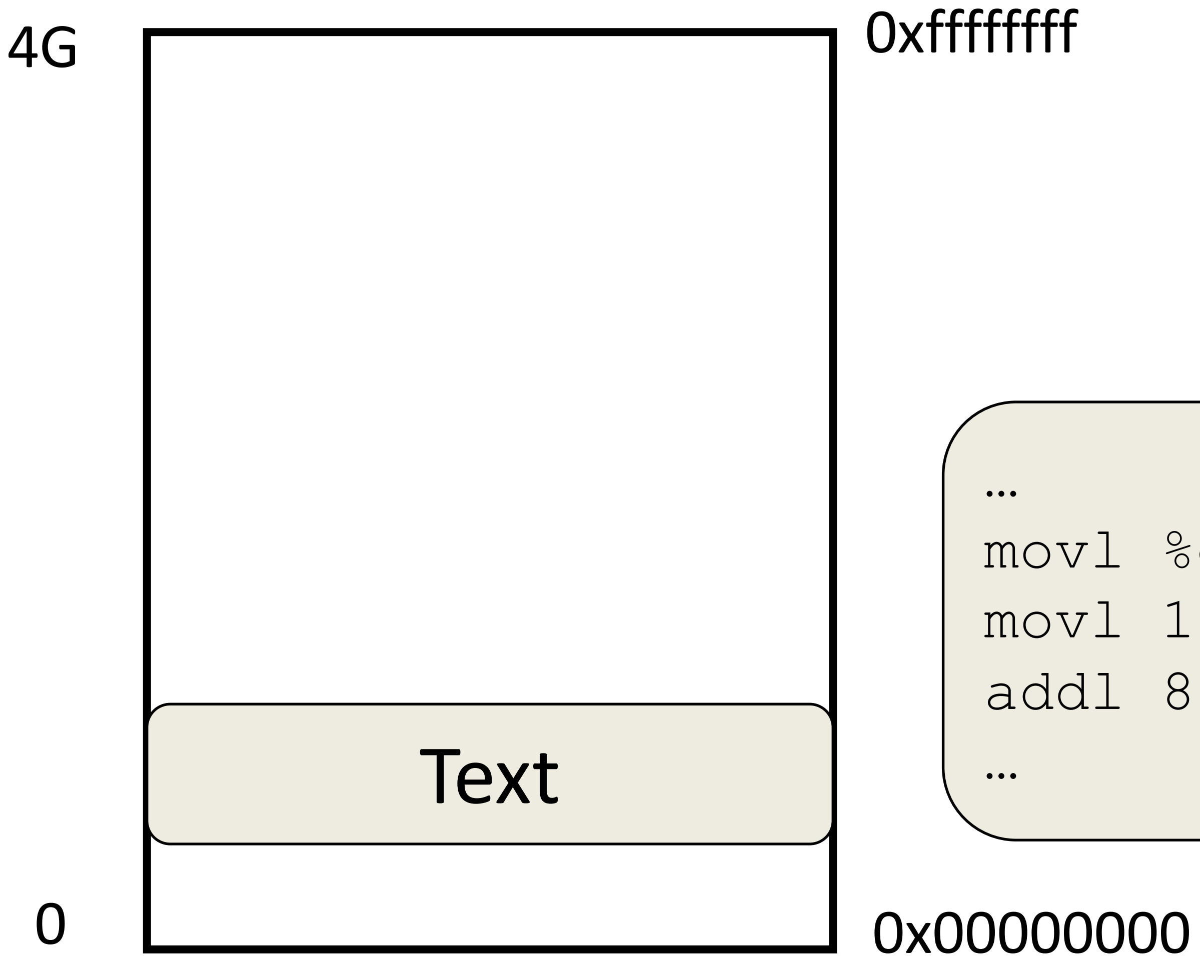
All programs are stored in memory

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



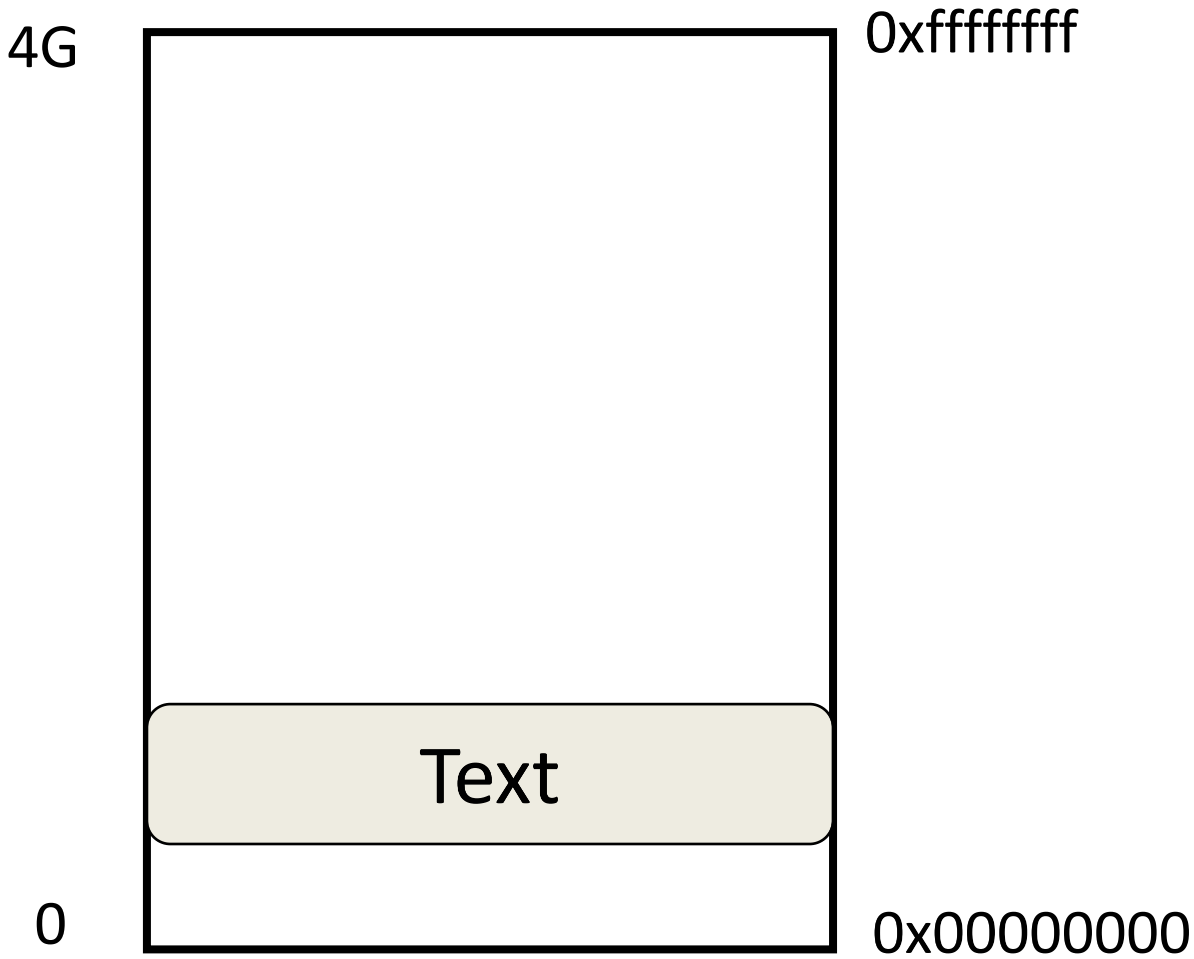
In reality, these are virtual address; the OS/CPU map them to physical addresses.

The instructions are stored in memory

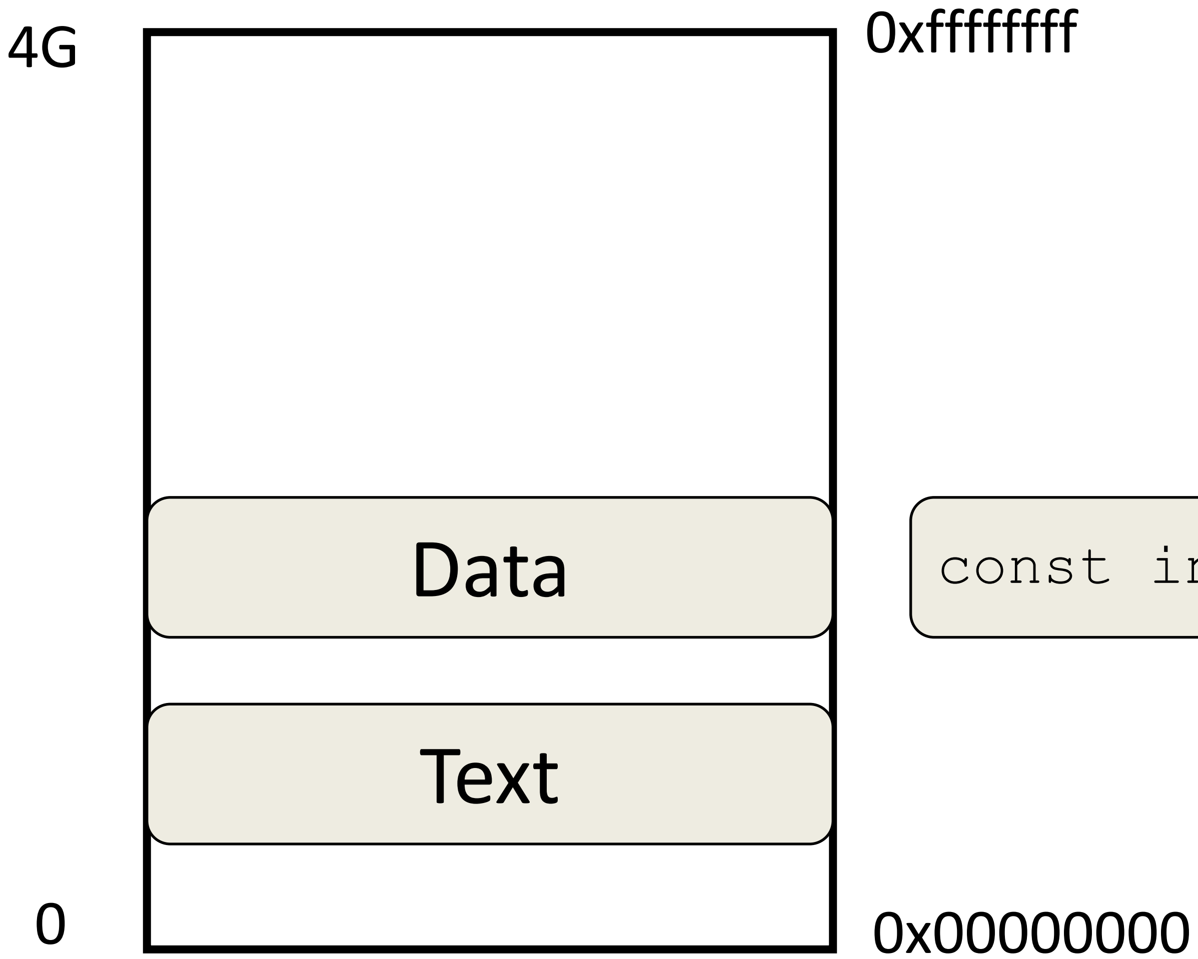


```
…
movl  %esp,%ebp
movl  12(%ebp),%eax
addl  8(%ebp),%eax
…
```

The instructions are stored in memory

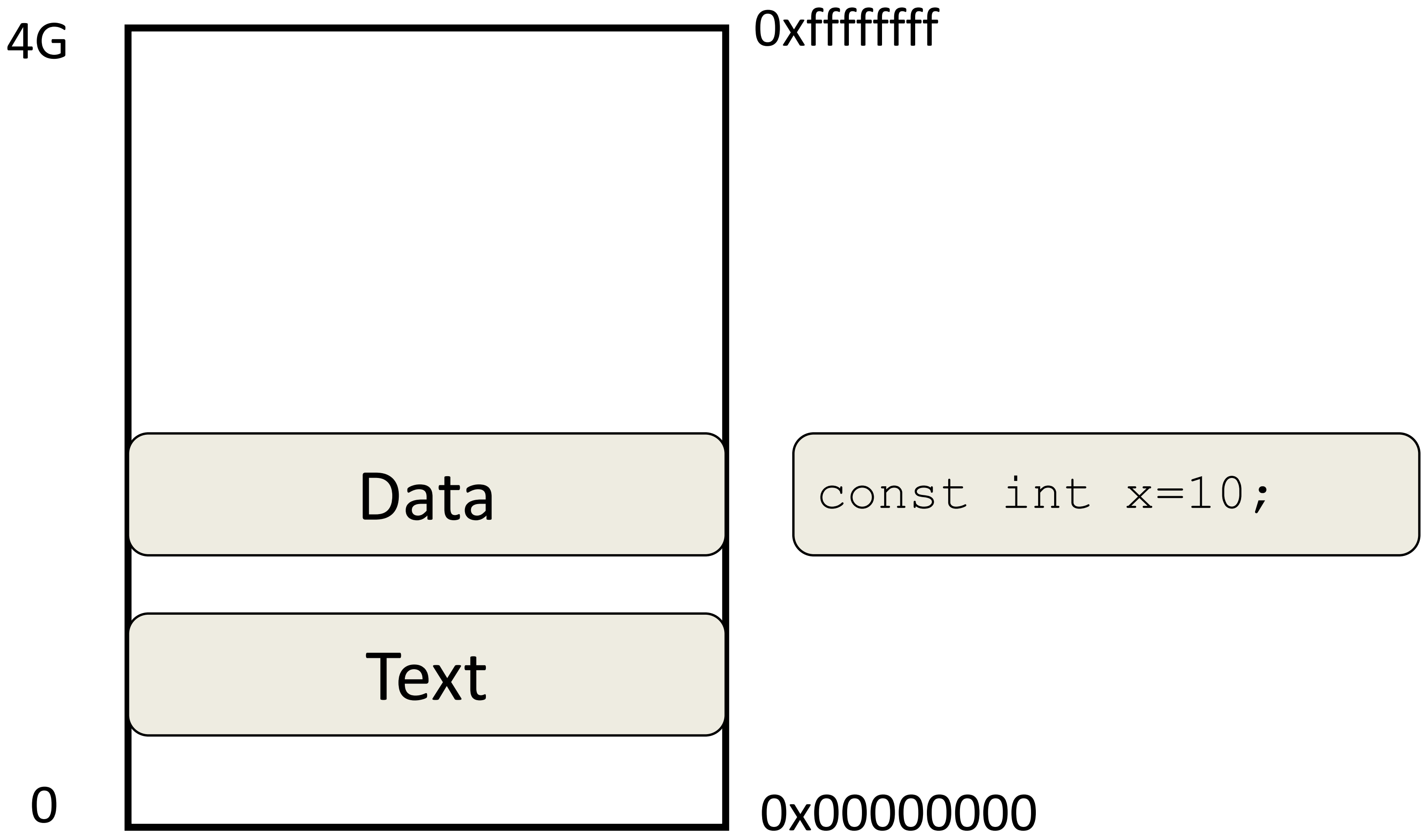


Data are stored in memory



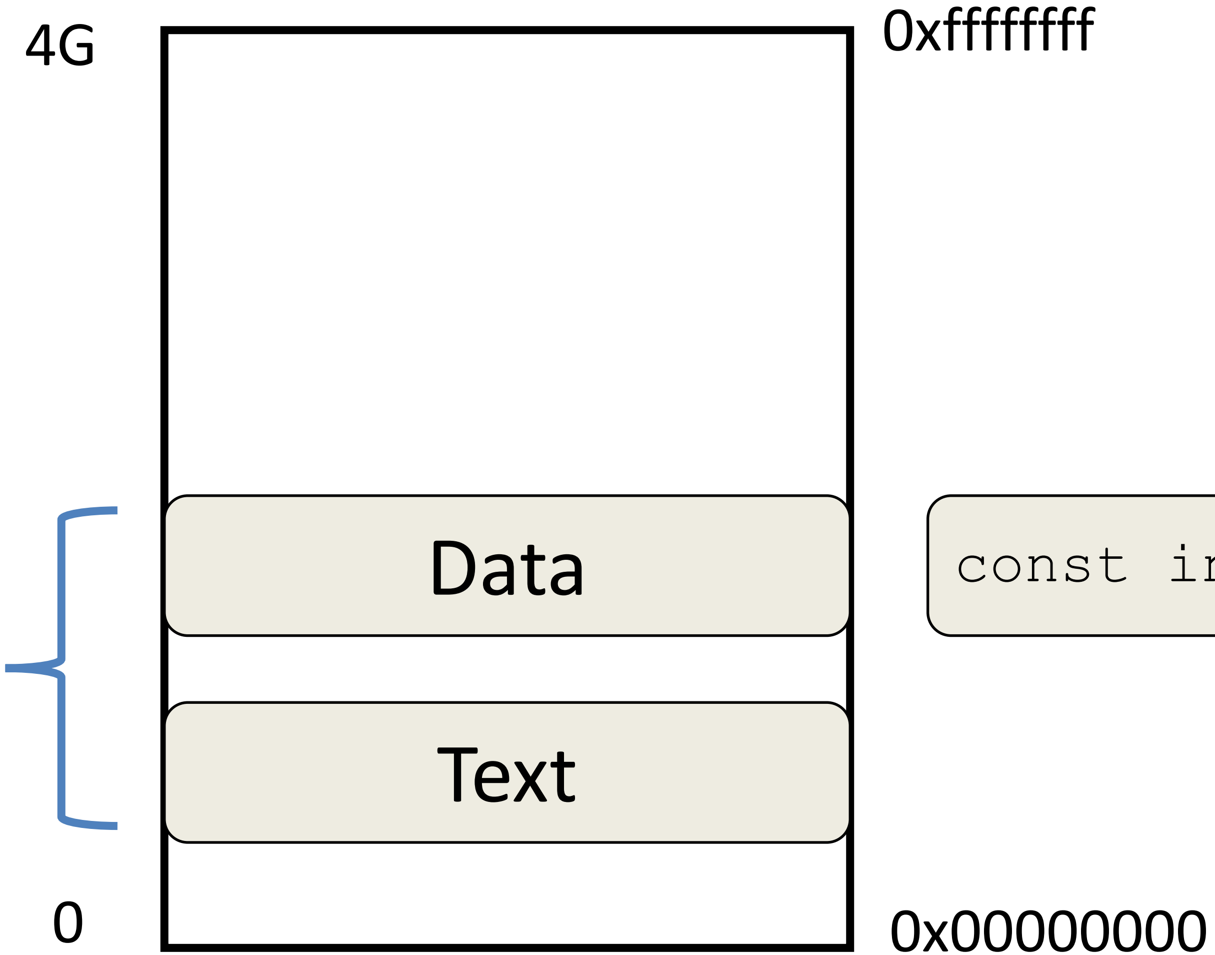
```
const int x=10;
```

Data are stored in memory



```
const int x=10;
```

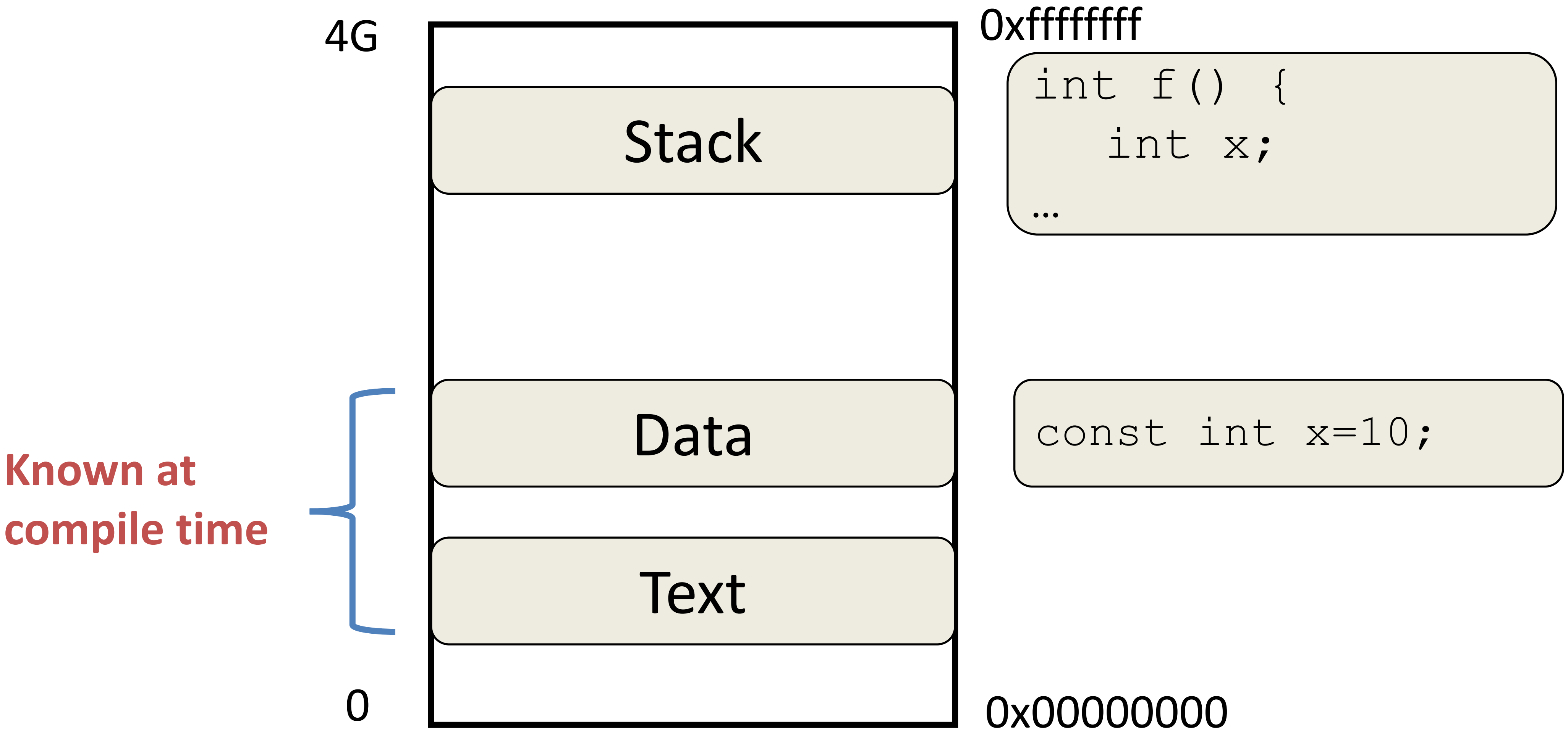
Data are stored in memory



Known at compile time

```
const int x=10;
```

Stack (Local variables)

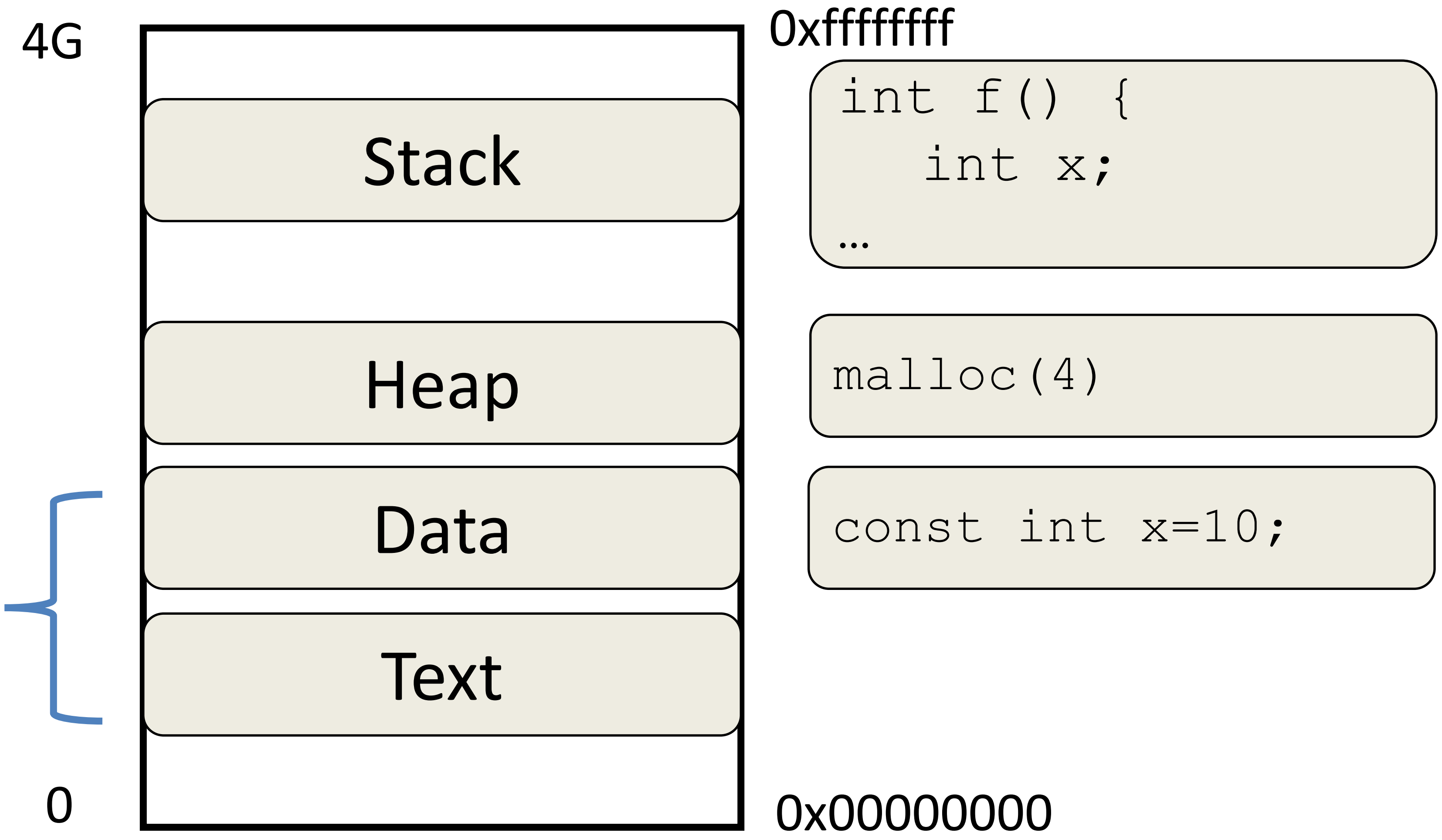


```
int f() {
   int x;
…
```

```
const int x=10;
```

Known at compile time

Heap (Dynamic memory)



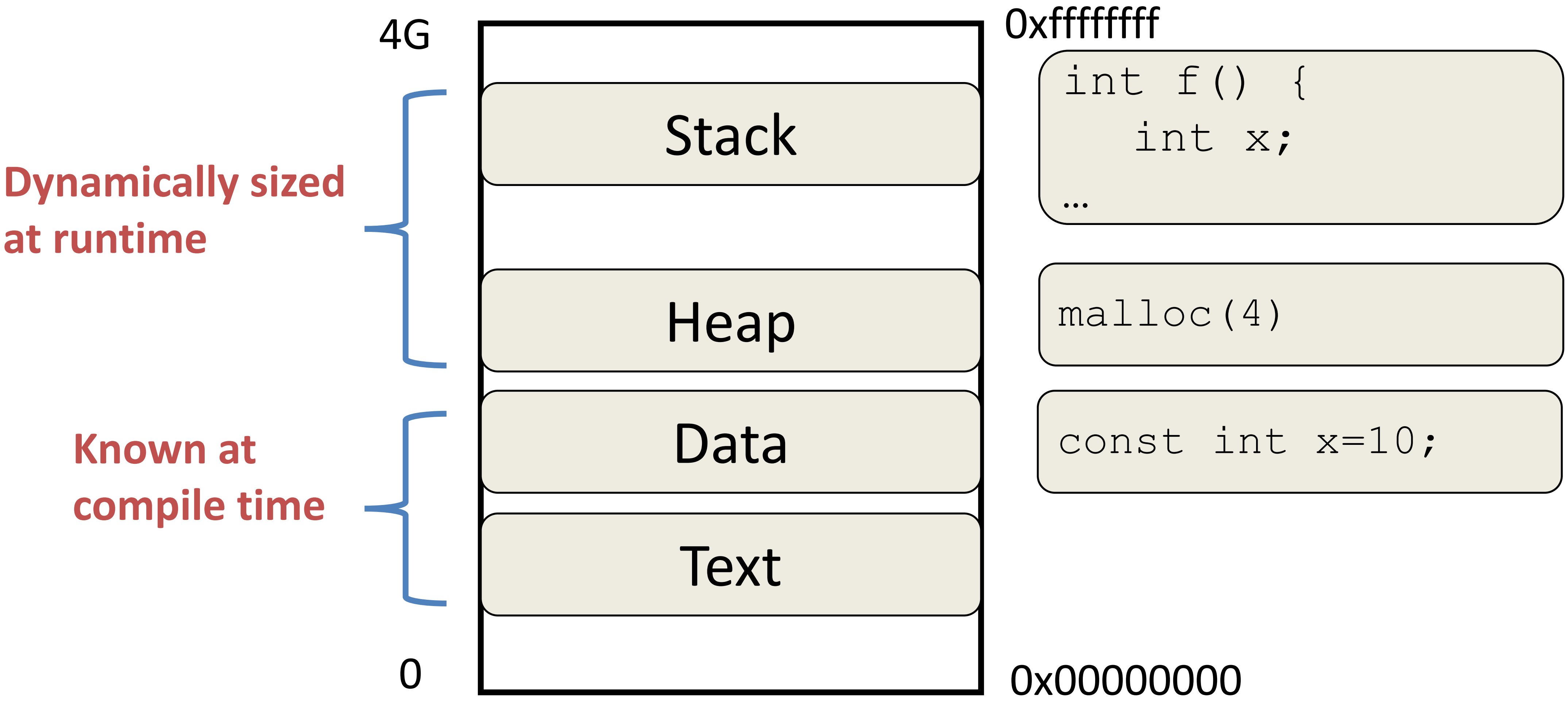
```
int f() {
    int x;
…
```

```
malloc(4)
```

```
const int x=10;
```

Known at compile time

Heap (Dynamic memory)



Dynamically sized at runtime

Known at compile time

4G

0

Stack

Heap

Data

Text

0xffffffff

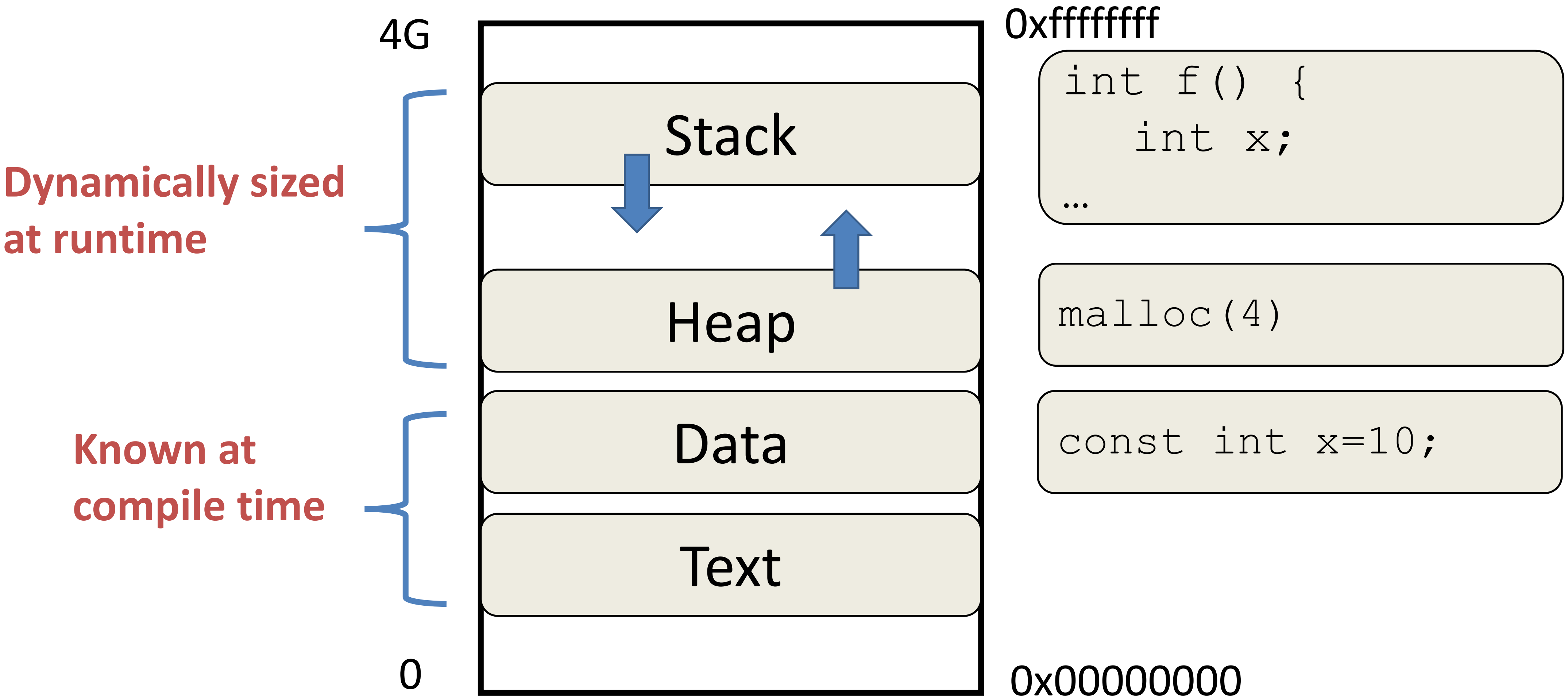
0x00000000

```
int f() {
   int x;
…
```

```
malloc(4)
```

```
const int x=10;
```

Stack & Heap grow in opposite directions



Dynamically sized at runtime

Known at compile time

4G

0

Stack

Heap

Data

Text

0xffffffff

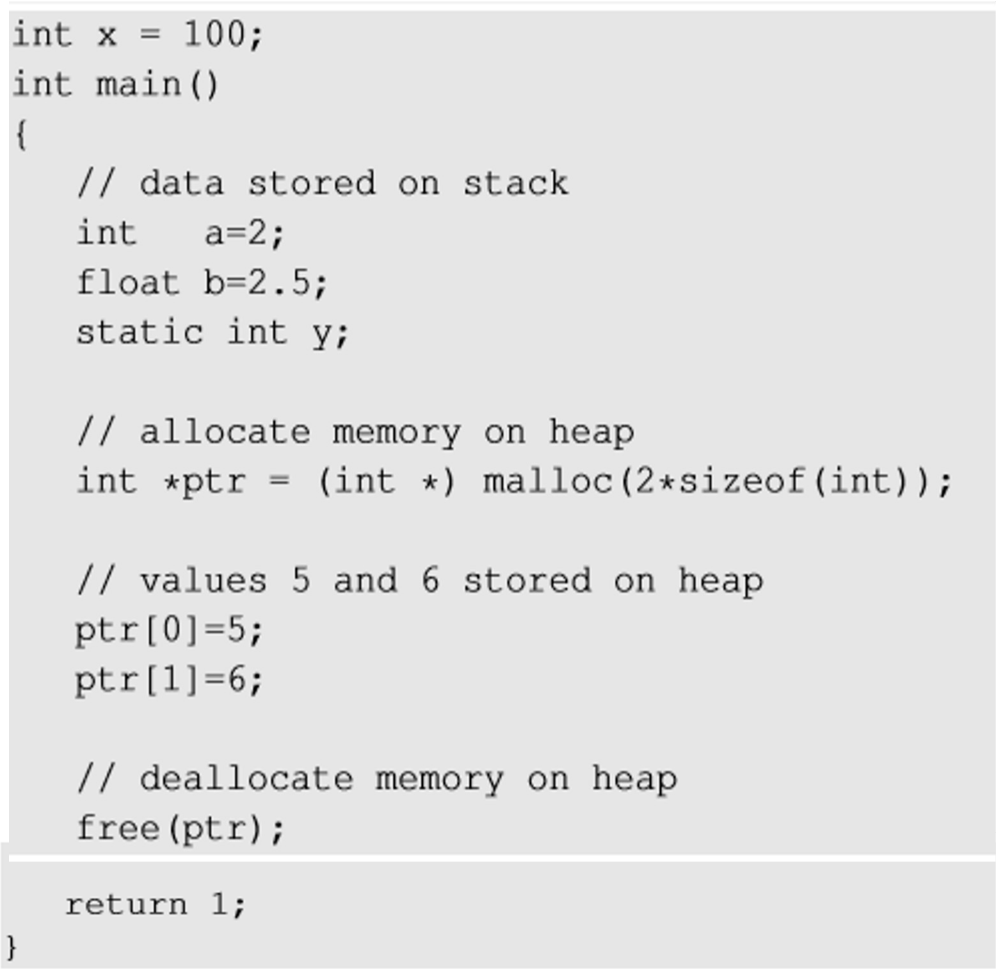
```
int f() {
    int x;
…
```

```
malloc(4)
```

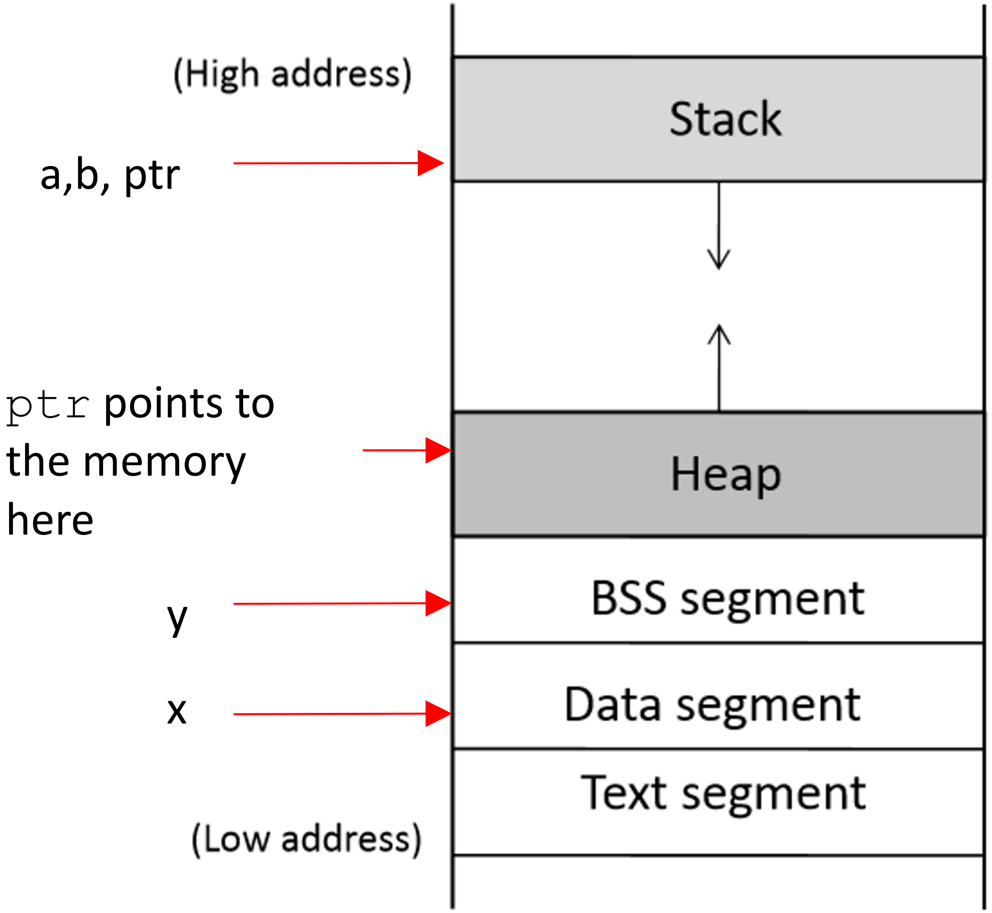
```
const int x=10;
```

0x00000000

Program Memory Stack



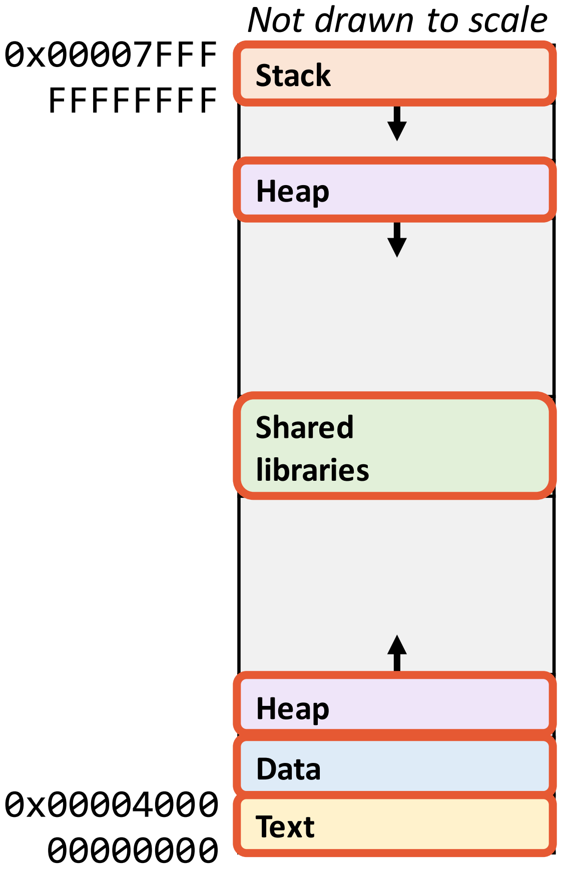
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // 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;
}
```



x86-64 Memory Layout

- Text
  - Executable machine instructions for the currently running program
  - Read-only
- Shared libraries
  - Executable machine instructions from libraries
  - Shared by all processes
  - Read-only

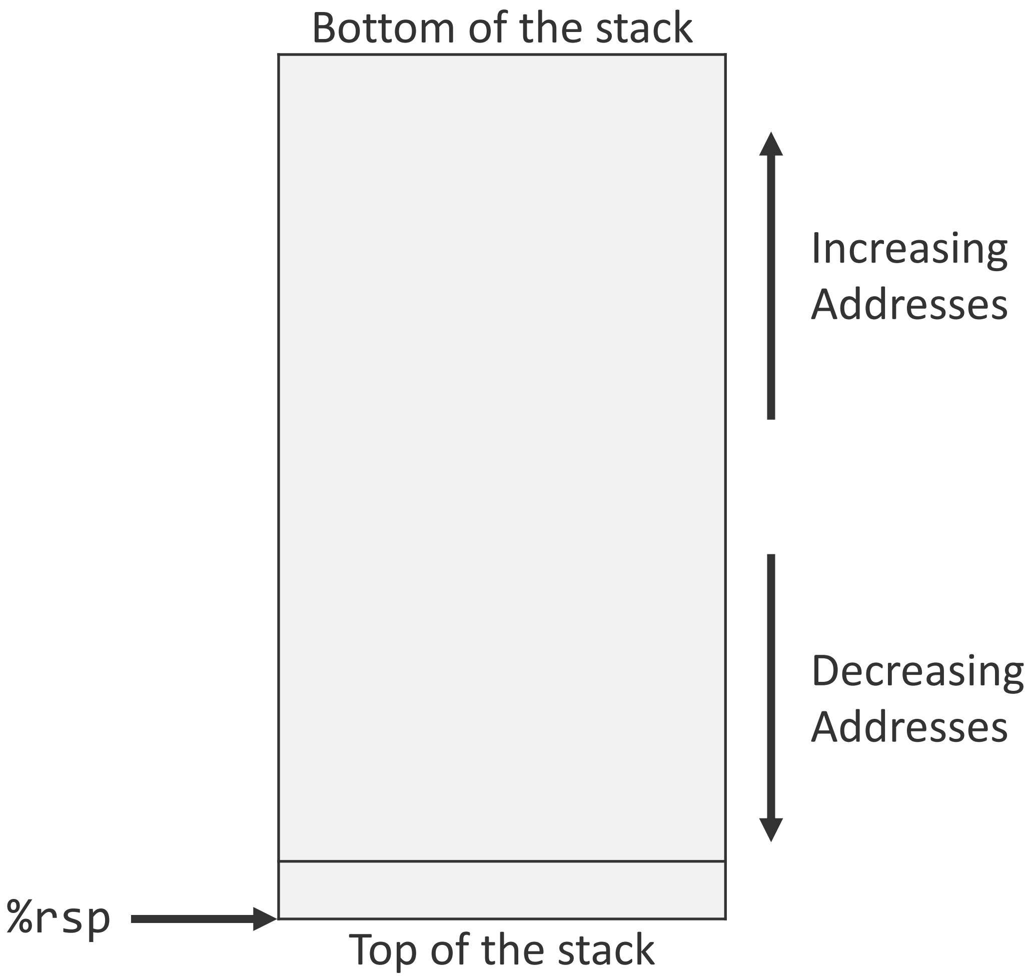
- Data
  - Statically allocated data
  - Global variables, `static` variables, string constants
- Stack
  - Stores stack frames and associated data
  - 8MB by default size limit
- Heap
  - Dynamically allocated objects
  - `malloc()`, `calloc()`, `new`



Stack Management

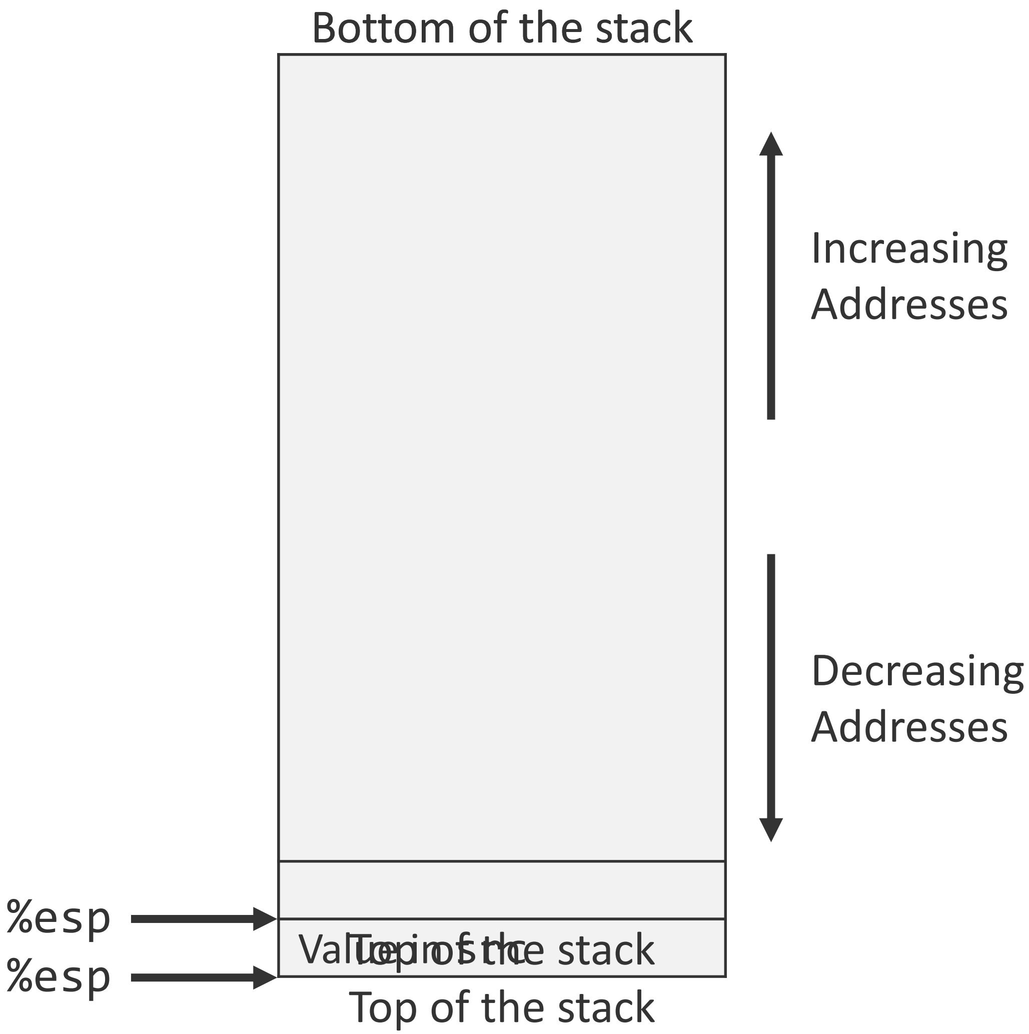
Stack

- Managed programmatically
  - Compiler outputs all necessary management functionality
- Grows towards lower addresses
- %rsp points to the lowest stack address
  - Points to the "top" element



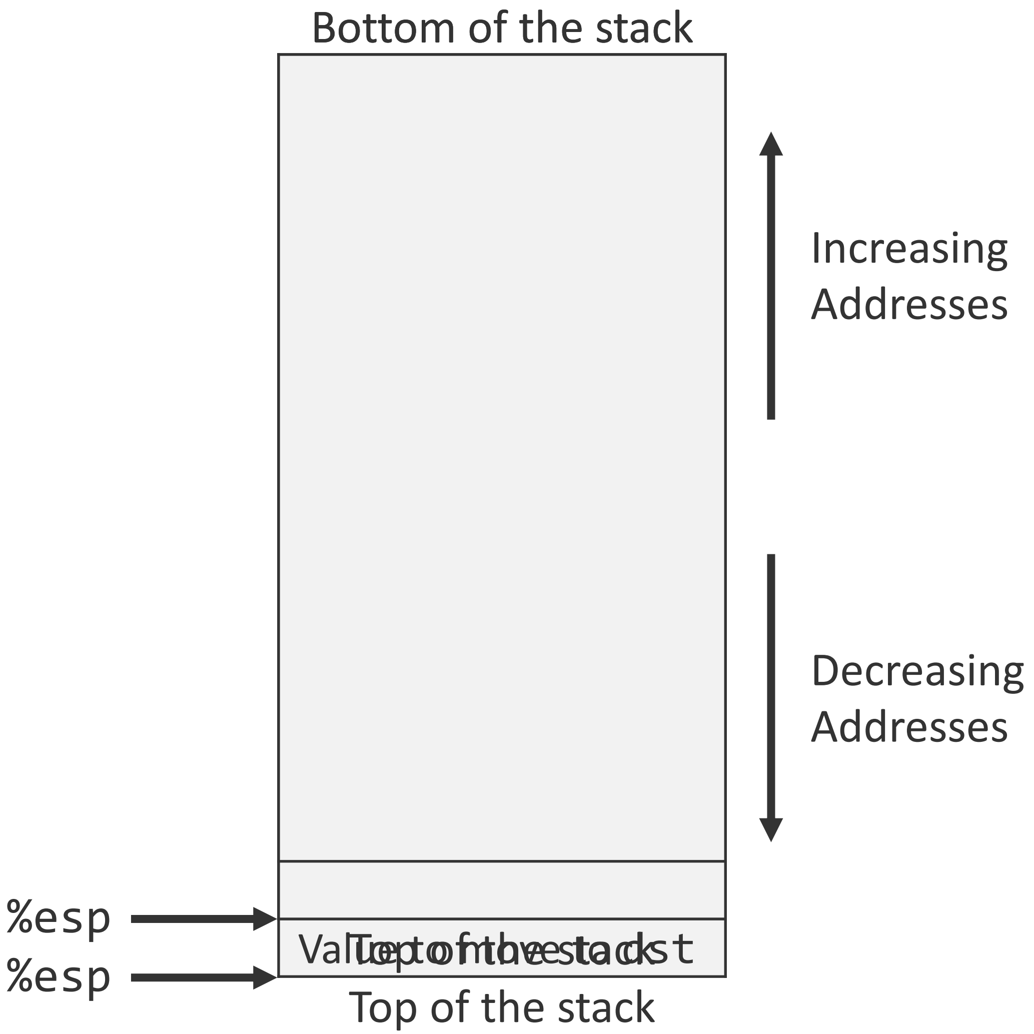
Push Instruction

- **`push src`**
  - Decrement `%rsp` by 4
  - Write operand `src` to memory address stored in `%esp`



Pop Instruction

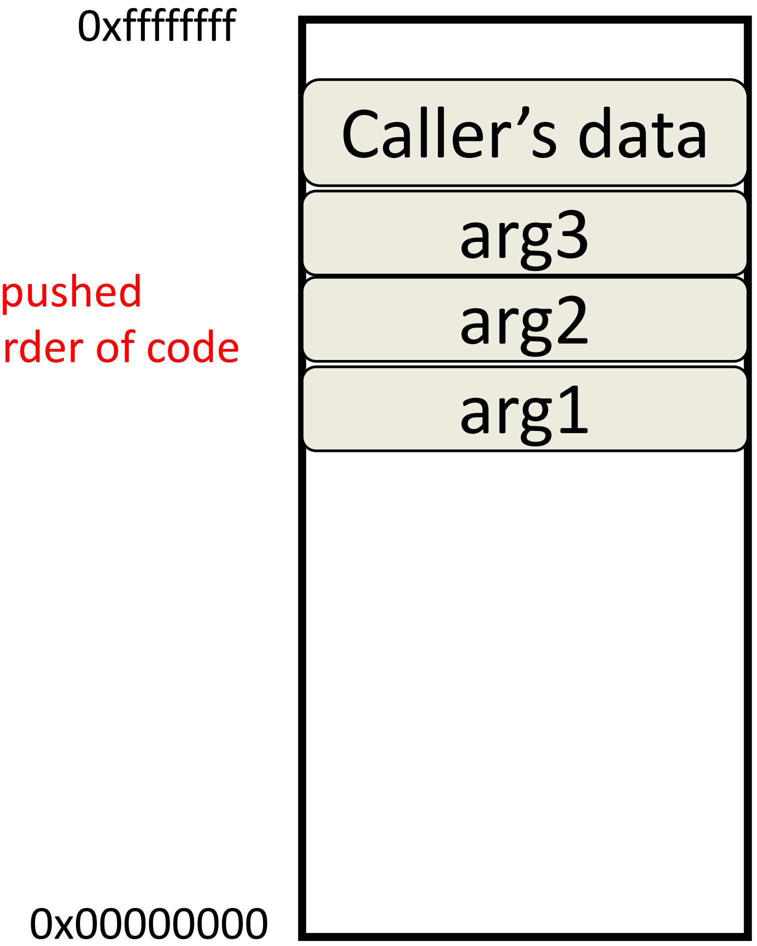
- **`pop dst`**
  - Read value at the memory address stored in `%rsp`
    - – Put that value into `dst`
  - Increment `%esp` by 4



Calling Procedures

Stack layout when calling function

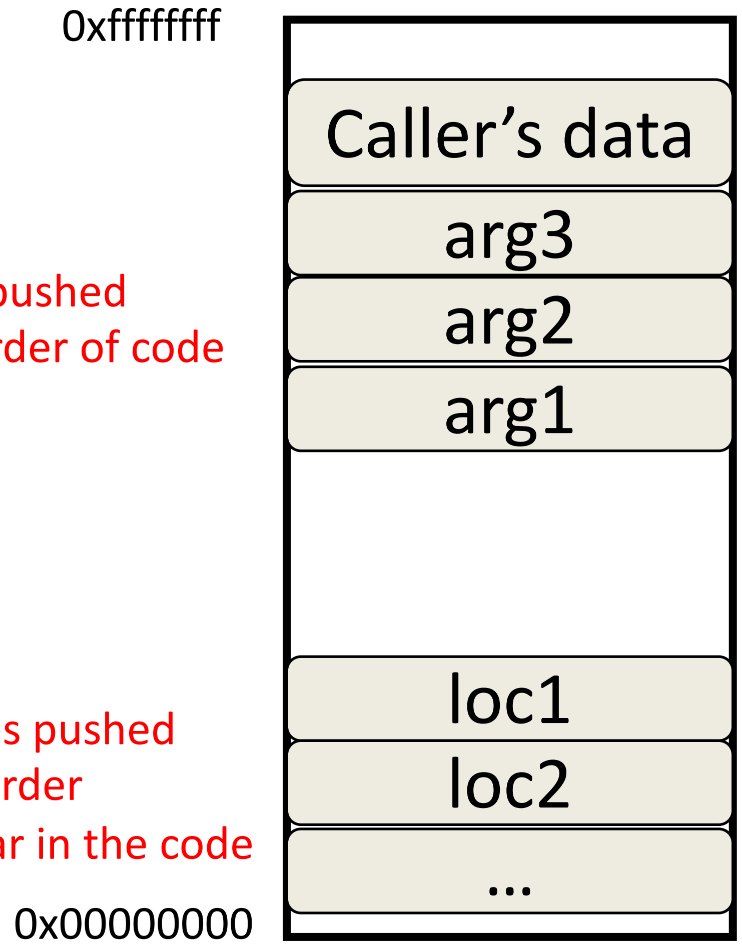
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



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;
    ...
}
```

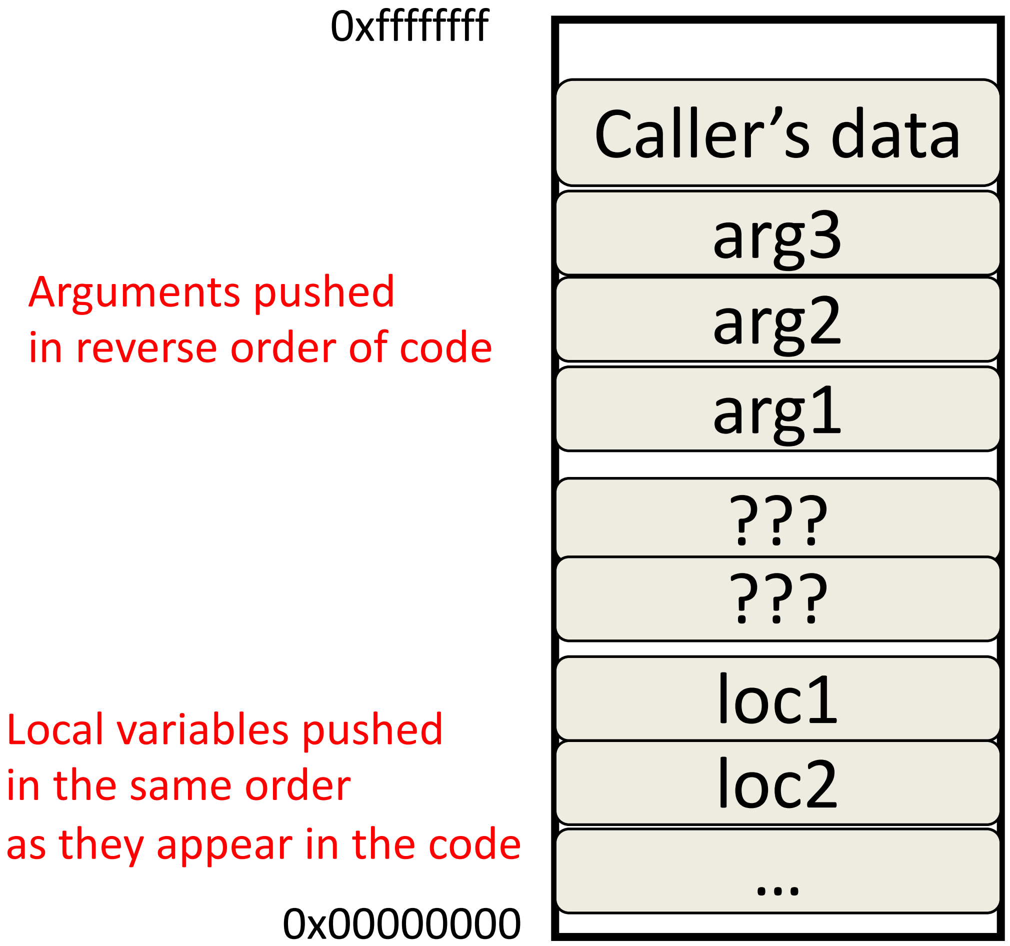


Arguments pushed
in reverse order of code

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;
    ...
}
```

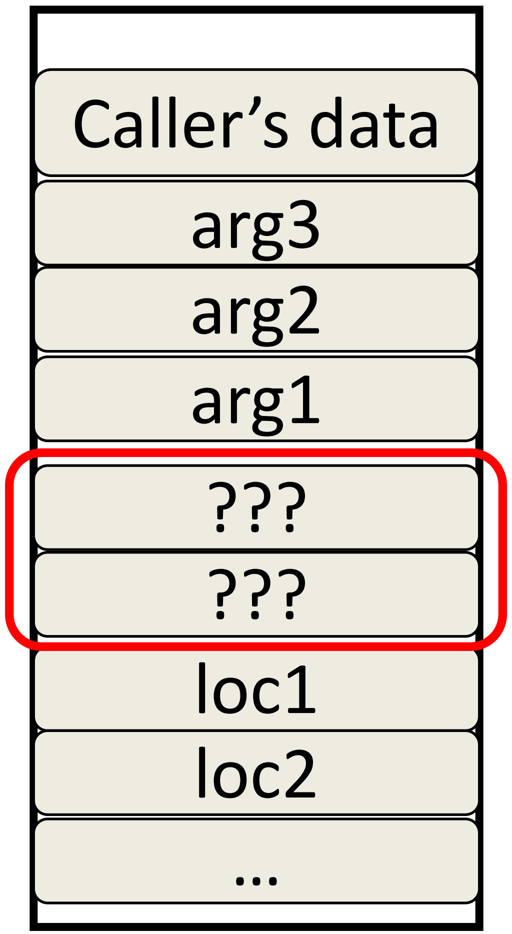


Arguments pushed
in reverse order of code

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;
    ...
}
```



Arguments pushed
in reverse order of code

Two values between the args
and the local variables

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

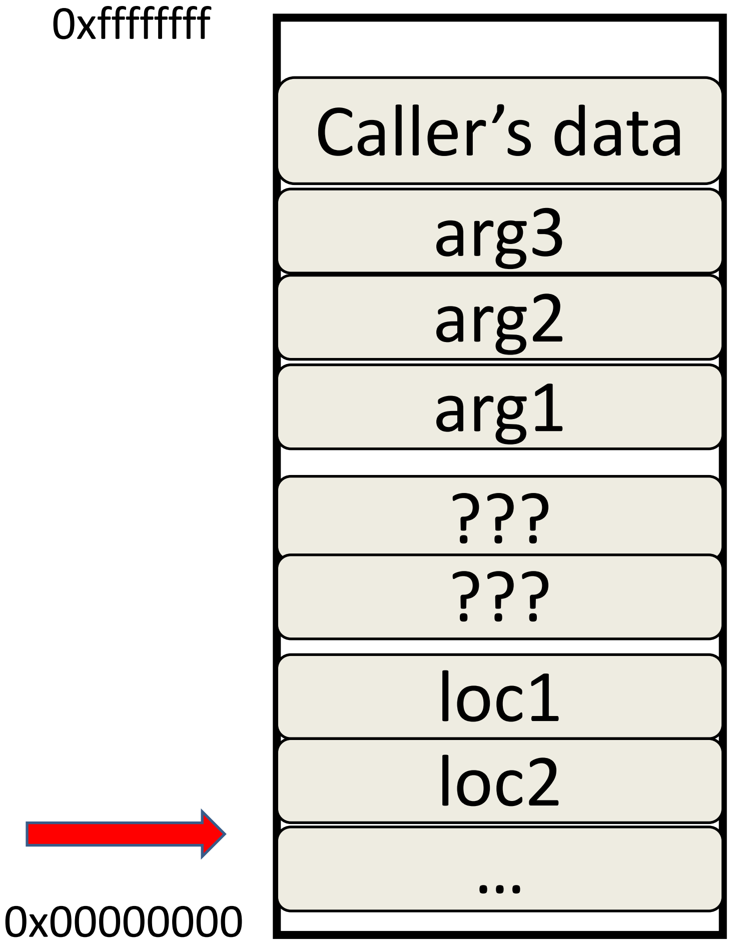
EBP (Base Pointer)

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?



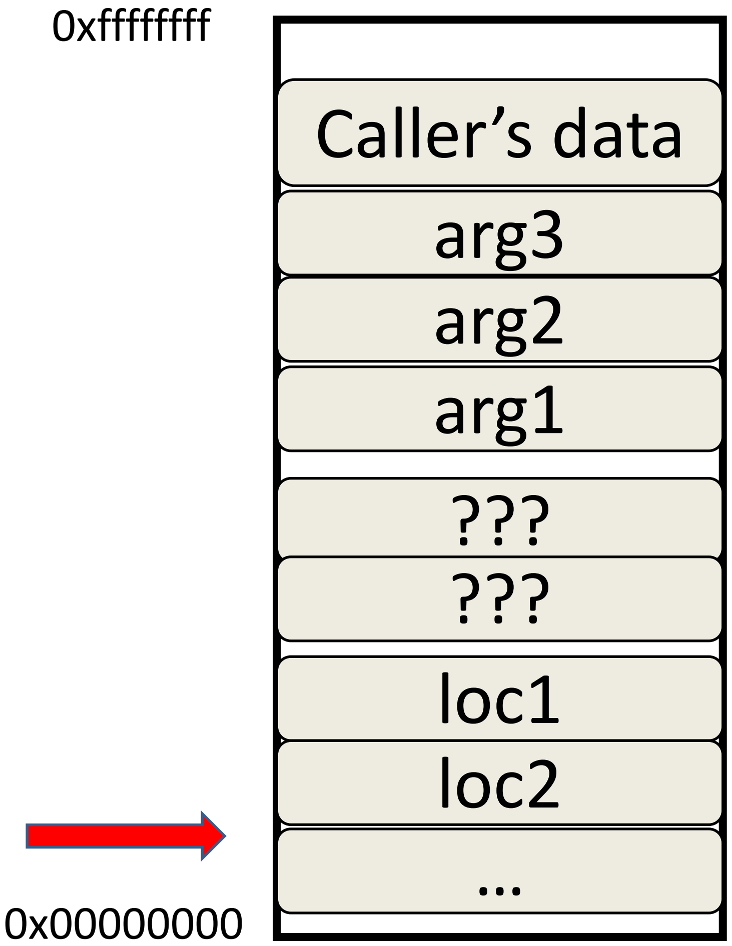
What’s the addr. of loc2?

```
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?

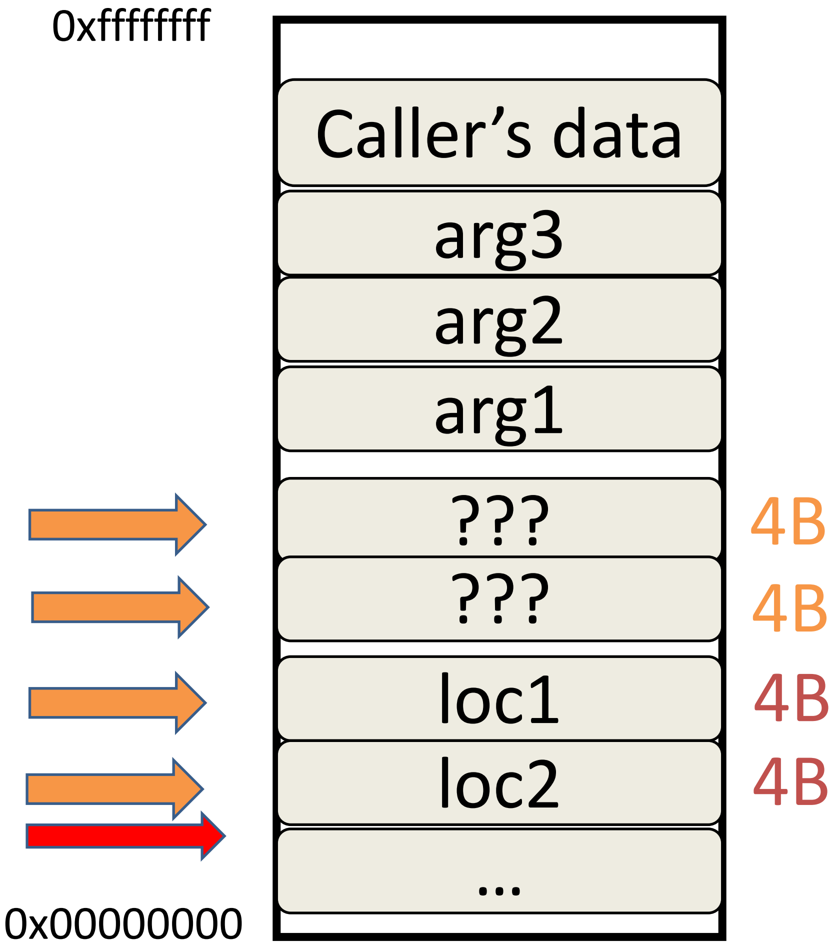
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

Q) Where is loc2?

What’s the specific address?

A) But we can know loc2 is always

8bytes before “???”s ➔ addr of ??? - 8B



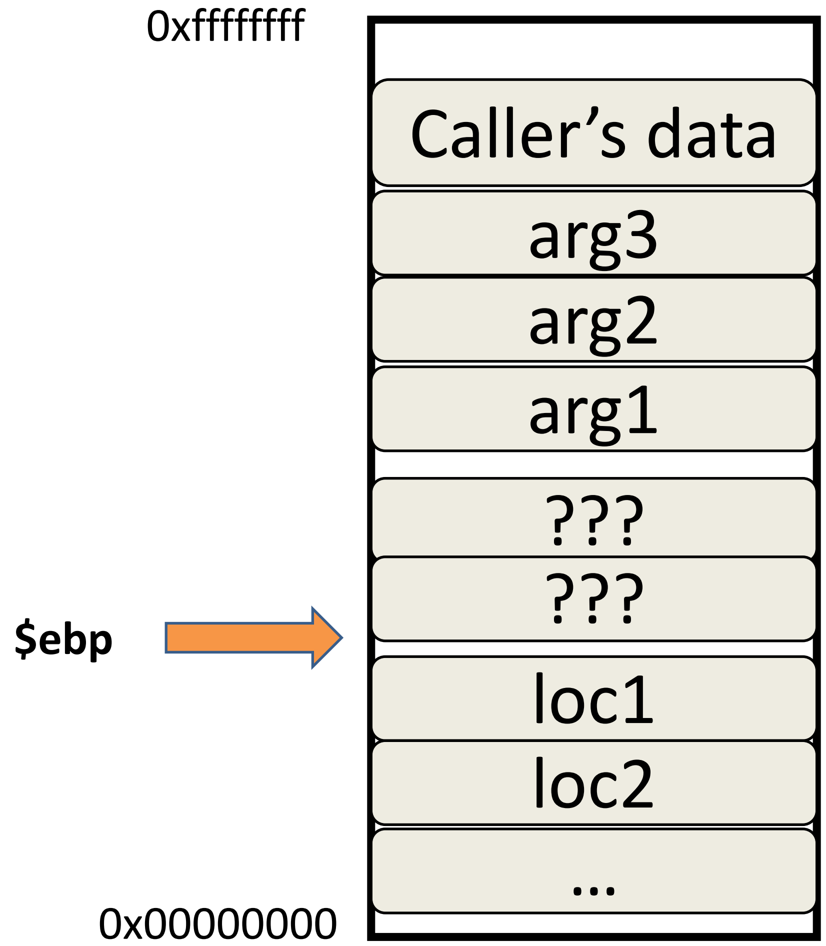
EBP (Base Pointer)

```
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)

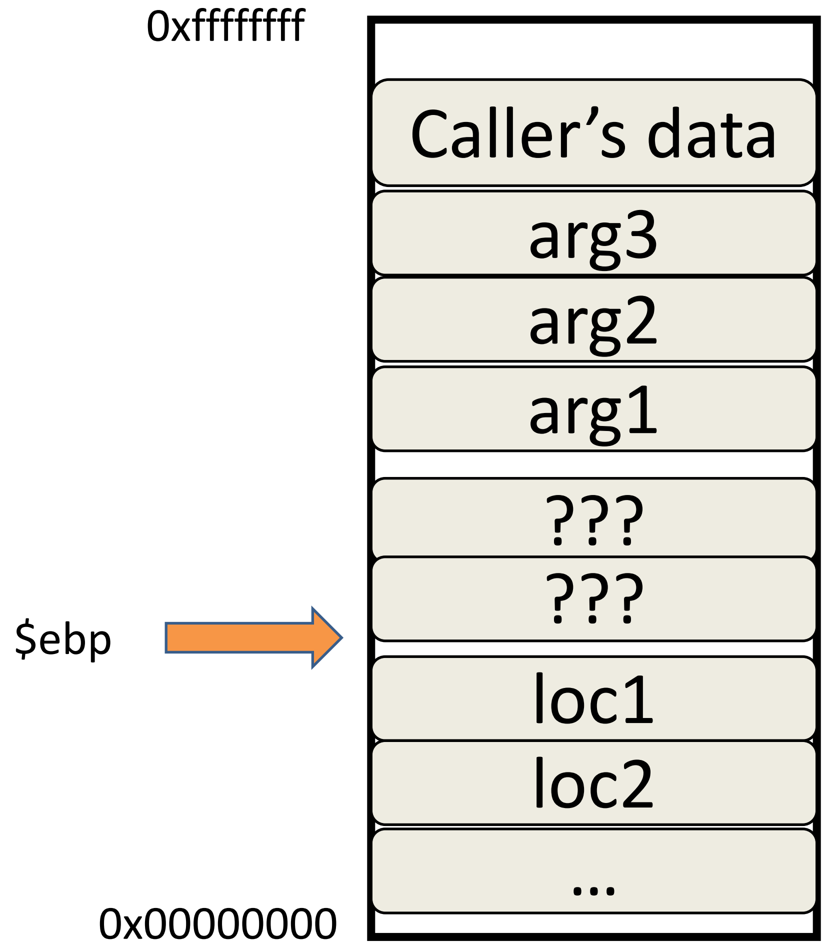
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

Q) where is loc2?

What's the specific address?

Quiz: 8 (%ebp) or -8 (%ebp)

A) -8 (%ebp)

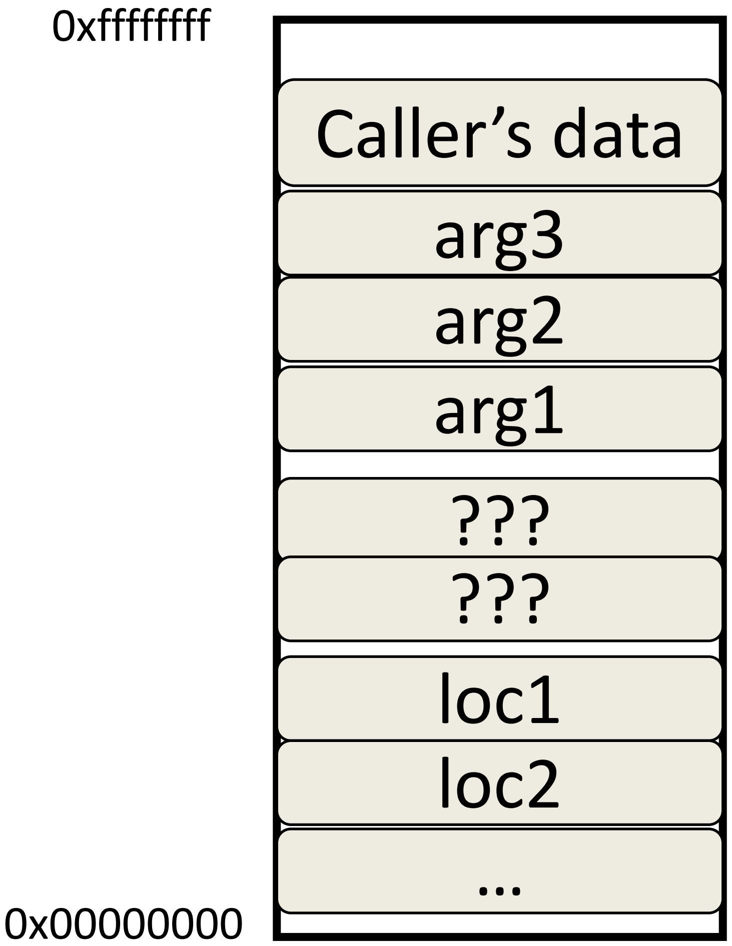


Stack layout when calling function

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

Q) What are "???"?

First, we need $ebp



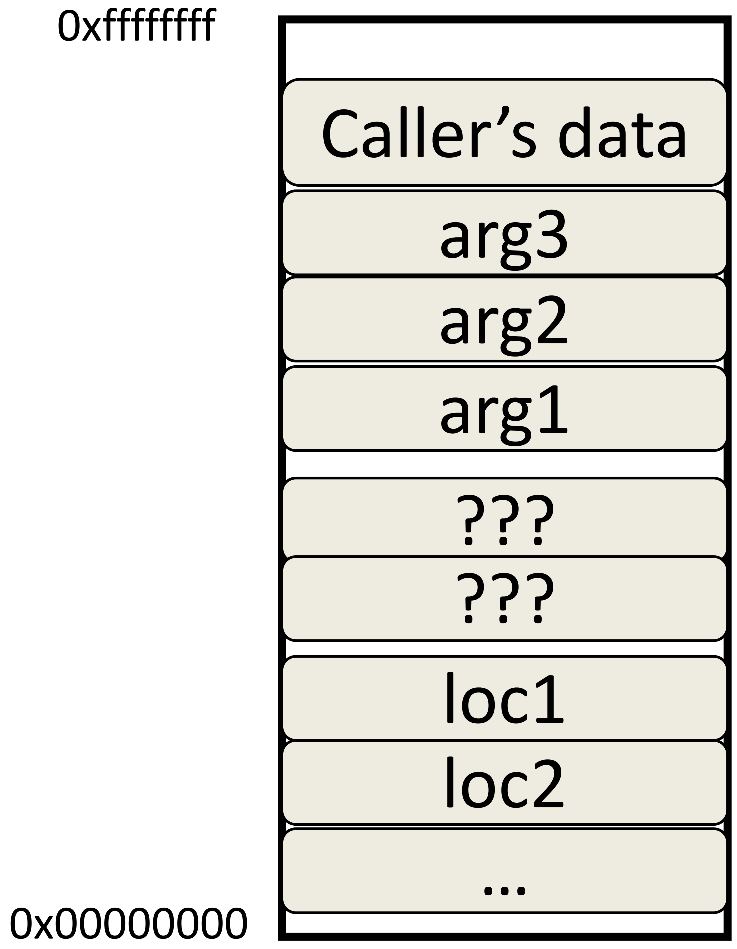
Stack layout when calling function

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

Q) What are “???” ?

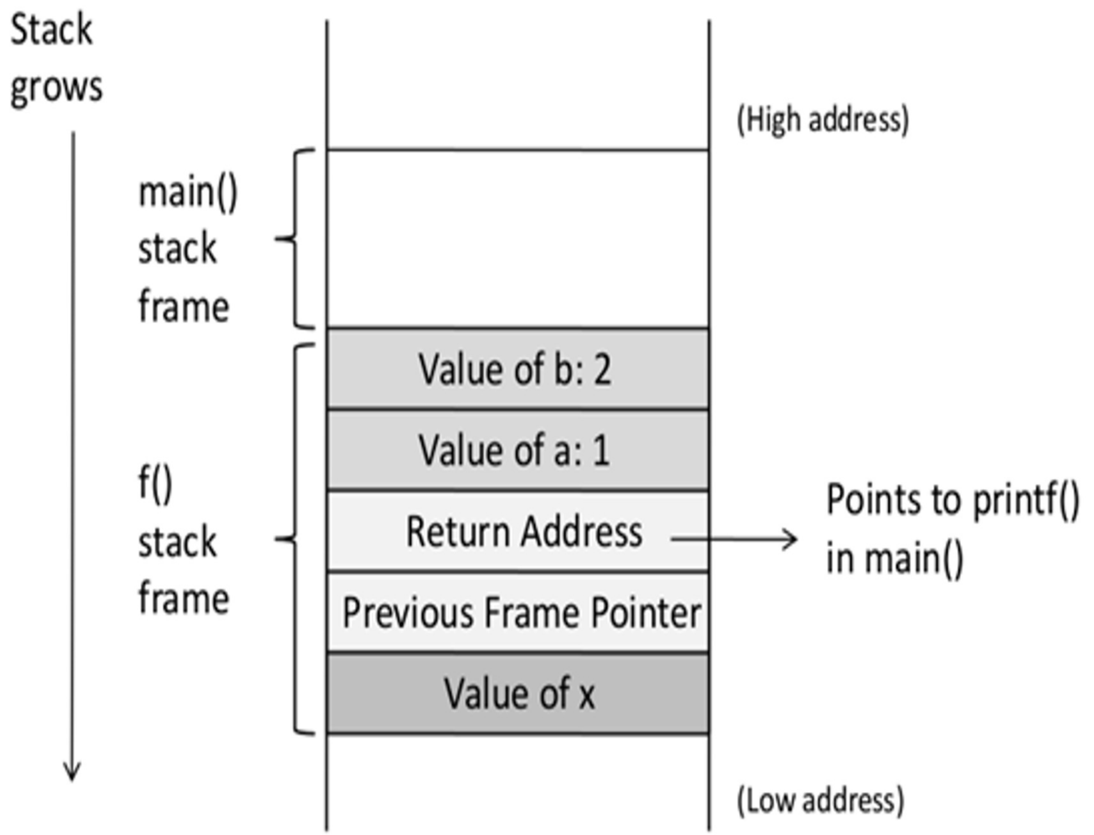
First, we need $ebp

Second, we need a return address

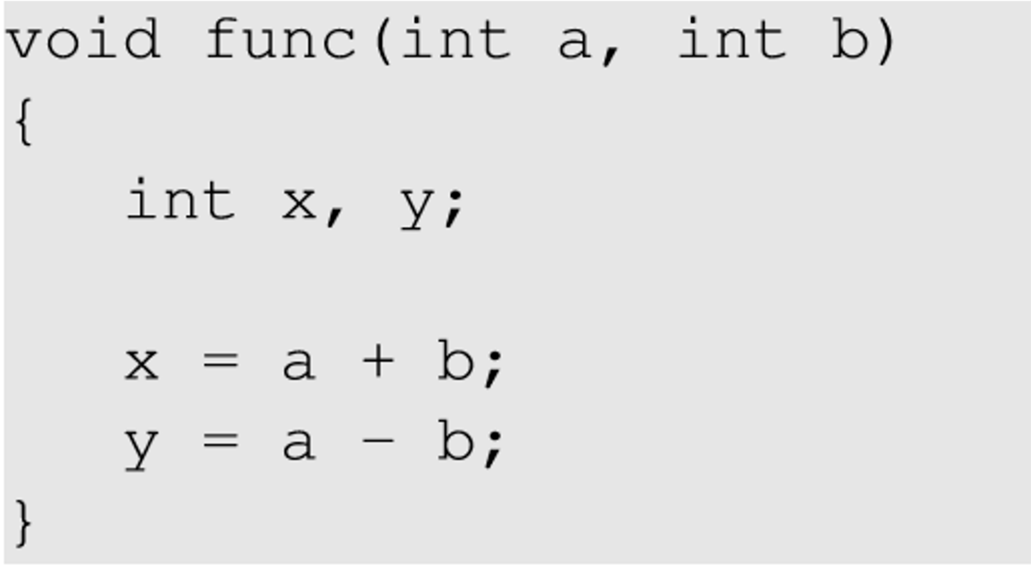


Function Call Stack

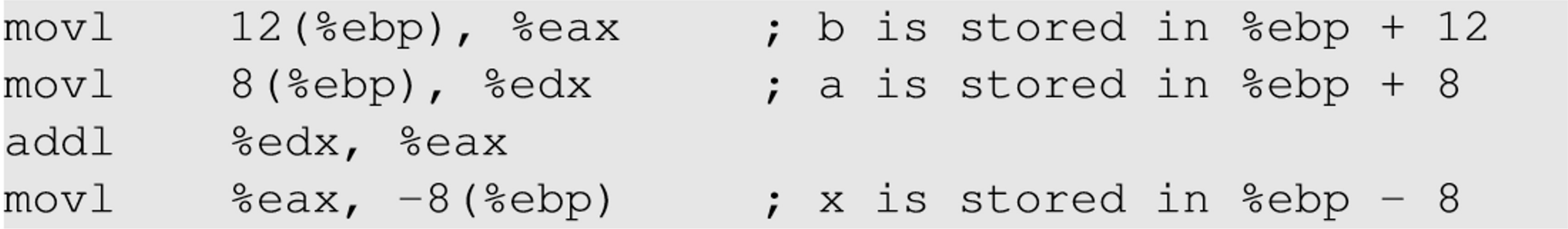
```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



Order of the function arguments in stack

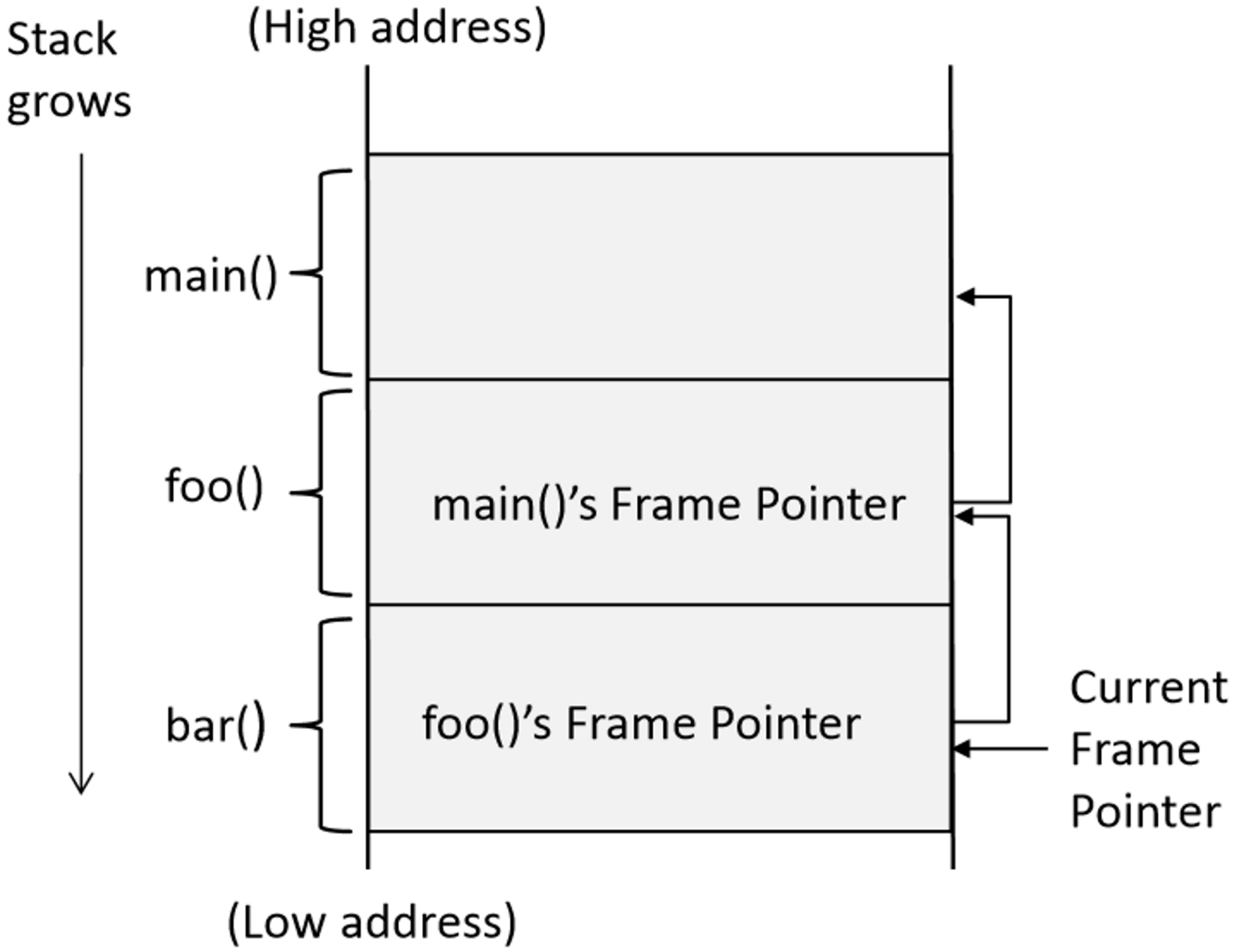


```
void func(int a, int b)
{
   int x, y;

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



```
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;
}
```

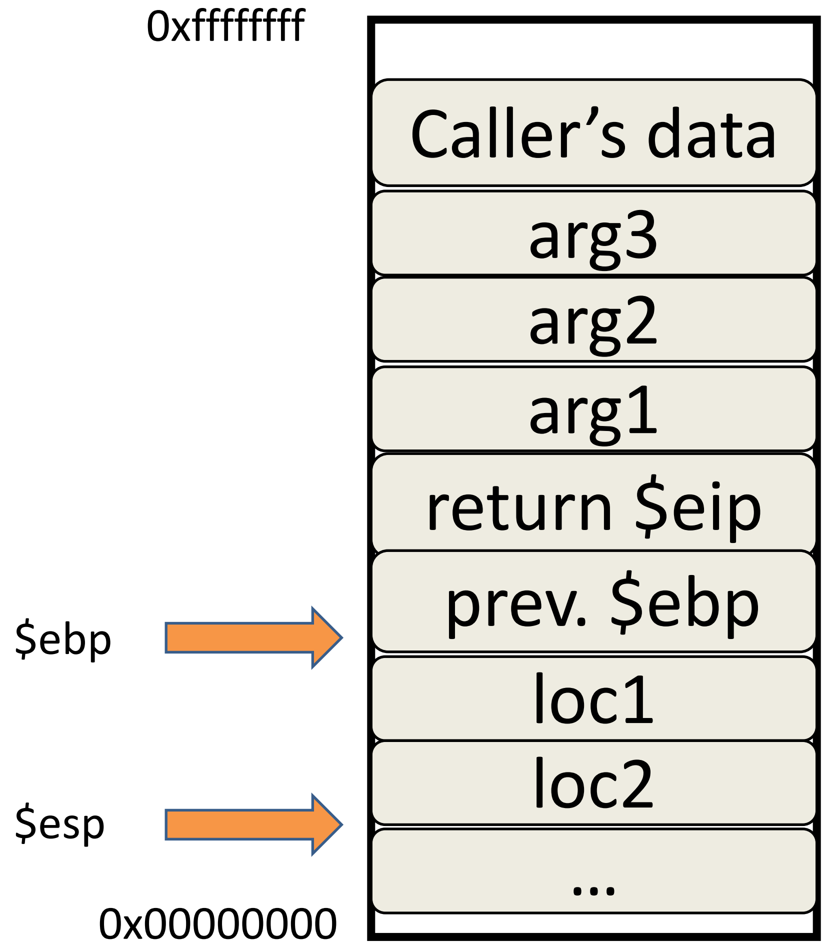
Returning from functions

In C

```
return;
```

In compiled assembly

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



0xffffffff

Caller’s data

arg3

arg2

arg1

return $eip

prev. $ebp

loc1

loc2

…

$ebp

$esp

0x00000000

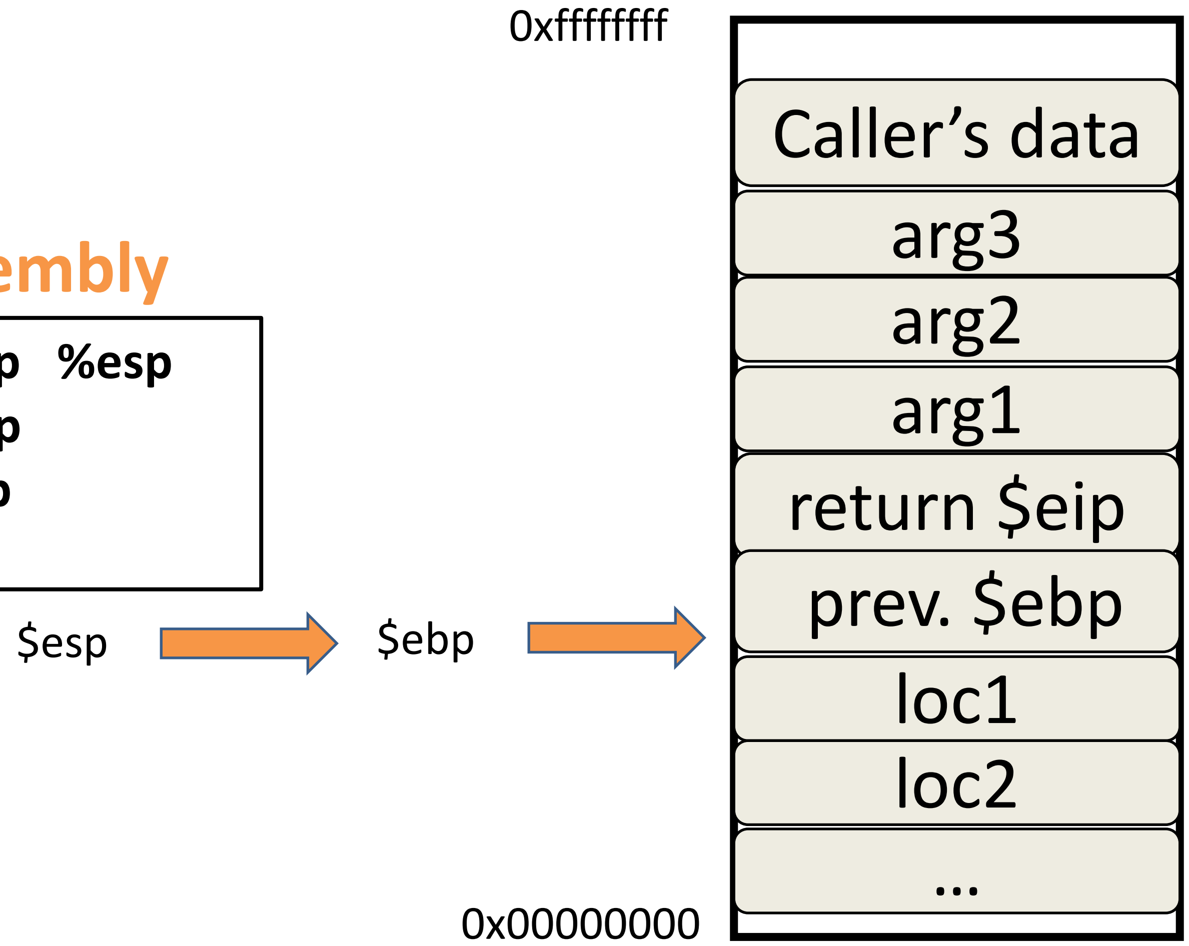
Returning from functions

In C

```
return;
```

In compiled assembly

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



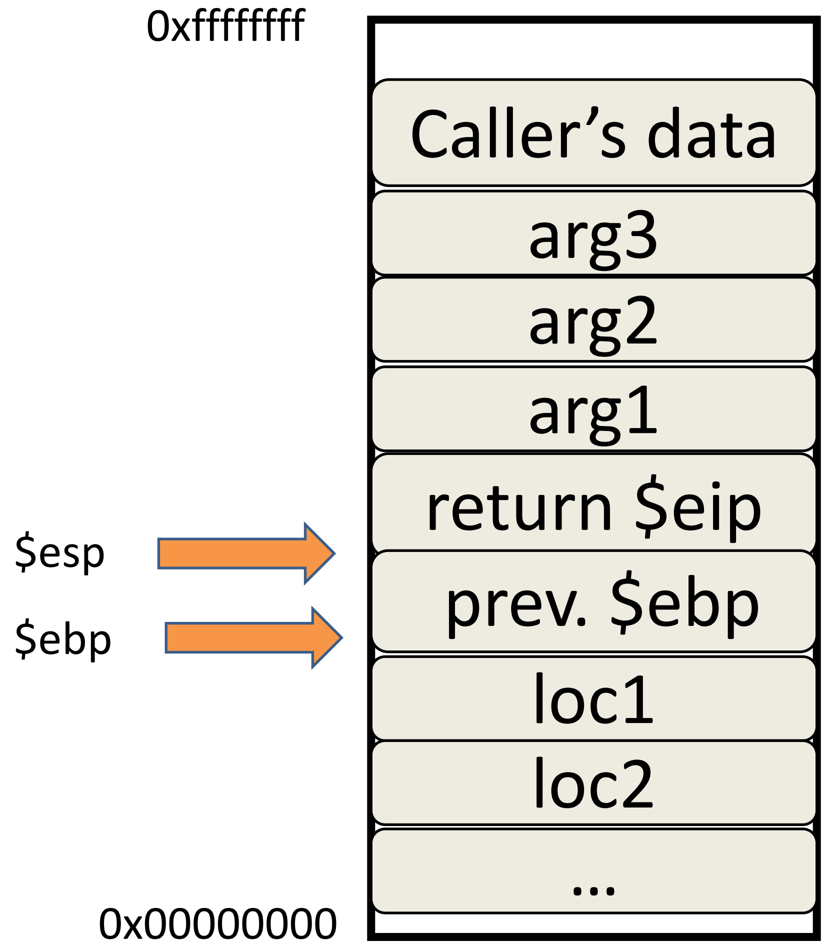
Returning from functions

In C

```
return;
```

In compiled assembly

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



0xffffffff

Caller’s data

arg3

arg2

arg1

return $eip

prev. $ebp

loc1

loc2

…

0x00000000

$esp

$ebp

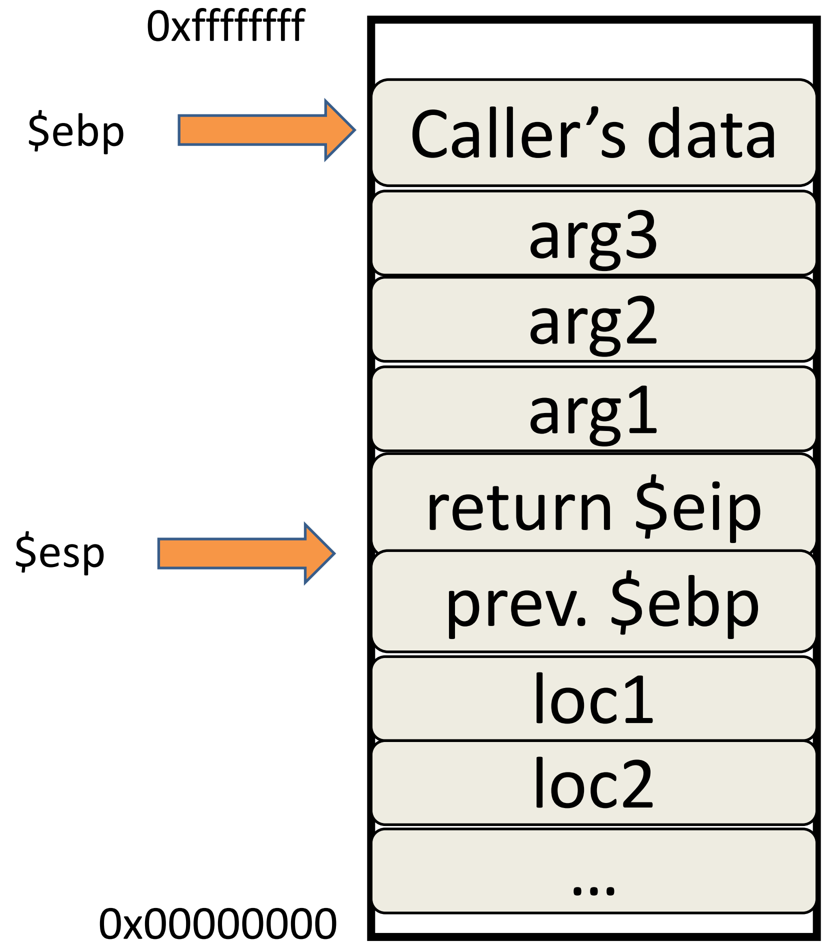
Returning from functions

In C

```
return;
```

In compiled assembly

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



Returning from functions

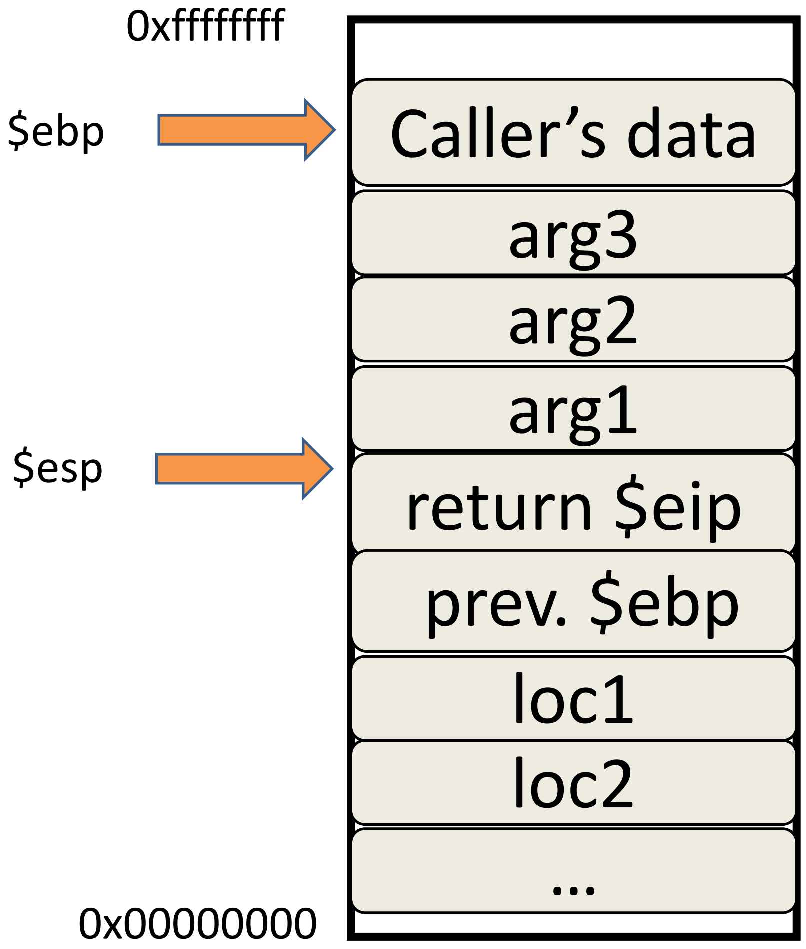
In C

```
return;
```

In compiled assembly

```
leave:      mov     %ebp   %esp
            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