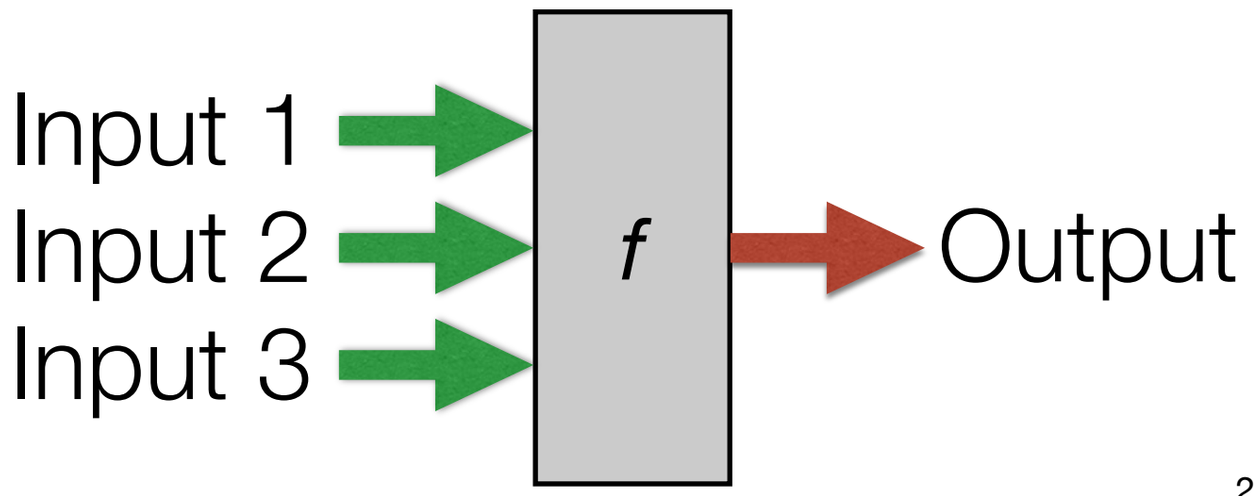
A C Primer (4) : Functions [ABC Chapter 5]

Ion Mandoiu
Laurent Michel
Revised by M. Khan, J. Shi and W. Wei

Building blocks for larger programs

- Effective programming requires problem decomposition into manageable pieces
- A C program is made of functions
  - Starting from main()
- Idealized view: a function is a black box…
  - It SPECIFIES **what** the computation will do
  - It ABSTRACTS AWAY **how** the computational process works
  - It takes INPUTS
  - It produces an OUTPUT



Function definitions

```
type function_name ( parameter_list ) {
  declarations & statements
}
```

- No nesting (cannot define functions in a function)
- Return type can be “void” (no return value expected)
  - If missing, compiler assumes int
- Return statements

```
return;        // terminate execution and return control to caller
return expr;   // terminate and pass value of expr back to caller
```

- Execution also terminates if end of function body reached
  - Returned value undefined

Function declarations (prototypes)

- Functions can be defined in any order
  - Declare a function before first use if definition comes later
  - Function prototypes often placed in header files (and reused)

```
#include <stdio.h>

int fahrToCelsius(int);

int main() {
   for(int fahr=0; fahr <= 300; fahr += 10)
      printf("%d F is %d C degrees\n",fahr,fahrToCelsius(fahr));
   return 0;
}

int fahrToCelsius(int degF) {
   return 5 * (degF - 32) / 9;
}
```

Declaration

Definition

Example: computing b^n

- Power function:
  - Returns an int
  - Two parameters: base b and exponent n, both int
- Functions can declare local variables
  - Parameters and local variables can only be accessed inside the function
  - Storage class "**auto**" by default, i.e., discarded when function returns
- Parameters are passed **by value**
  - n is changed in the power function but i DOES NOT change

```
int power(int b, int n)
{
   int rv = 1;
   while (n>0) {
      rv *= b;
      n--;
   }
   return rv;
}

int foo ()
{
   int i = 10;
   return power(2, i);
}
```

Static and global variables

- Static local variables
  - Not visible outside function but **retain** value across function calls
- Global variables
  - Declared outside functions; retained for entire duration of program
  - Storage class “extern” by default

    ➔Can be accessed from functions in other files
  - Static global variables

    ➔Visible only in functions defined in the same file following variable declaration

Be cautious with static and global variables

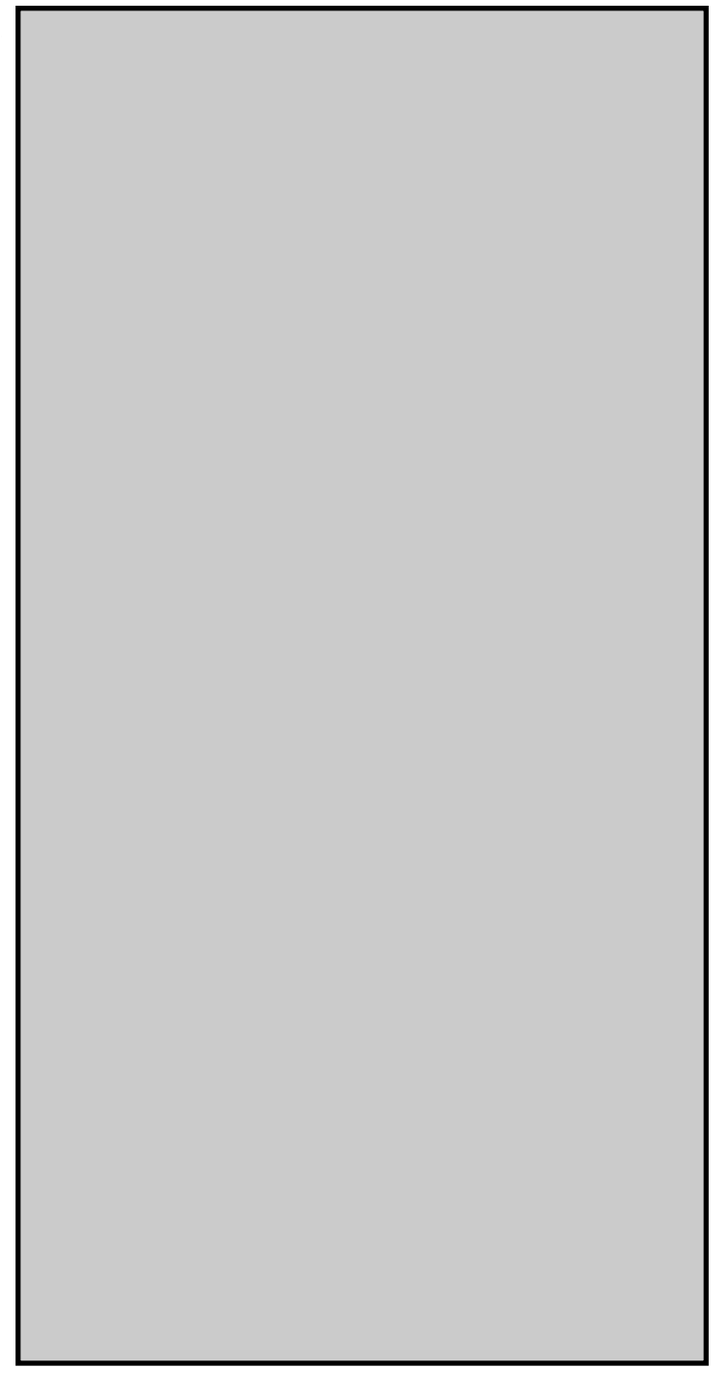
- "*Nice*" functions **only** depend on their inputs
- Static and global variables have side-effects
  - Retain values across function calls
  - Change the meaning of the function at each call!
  - You cannot understand the function without holding **all** the code in your head
- Unless you have really good reasons and really know what you are doing, **do not use static or global variables.**

Function call context

- Function call context includes
  - Copies of function arguments (call-by-value)
  - (Auto) local variables
  - Return address, …
- Call contexts managed automatically using the **execution stack**
  - A stack frame is created automatically for each function call
  - The frame lasts for the duration of the call
  - Discarded automatically when the function terminates
  - NOTHING in the frame survives the call

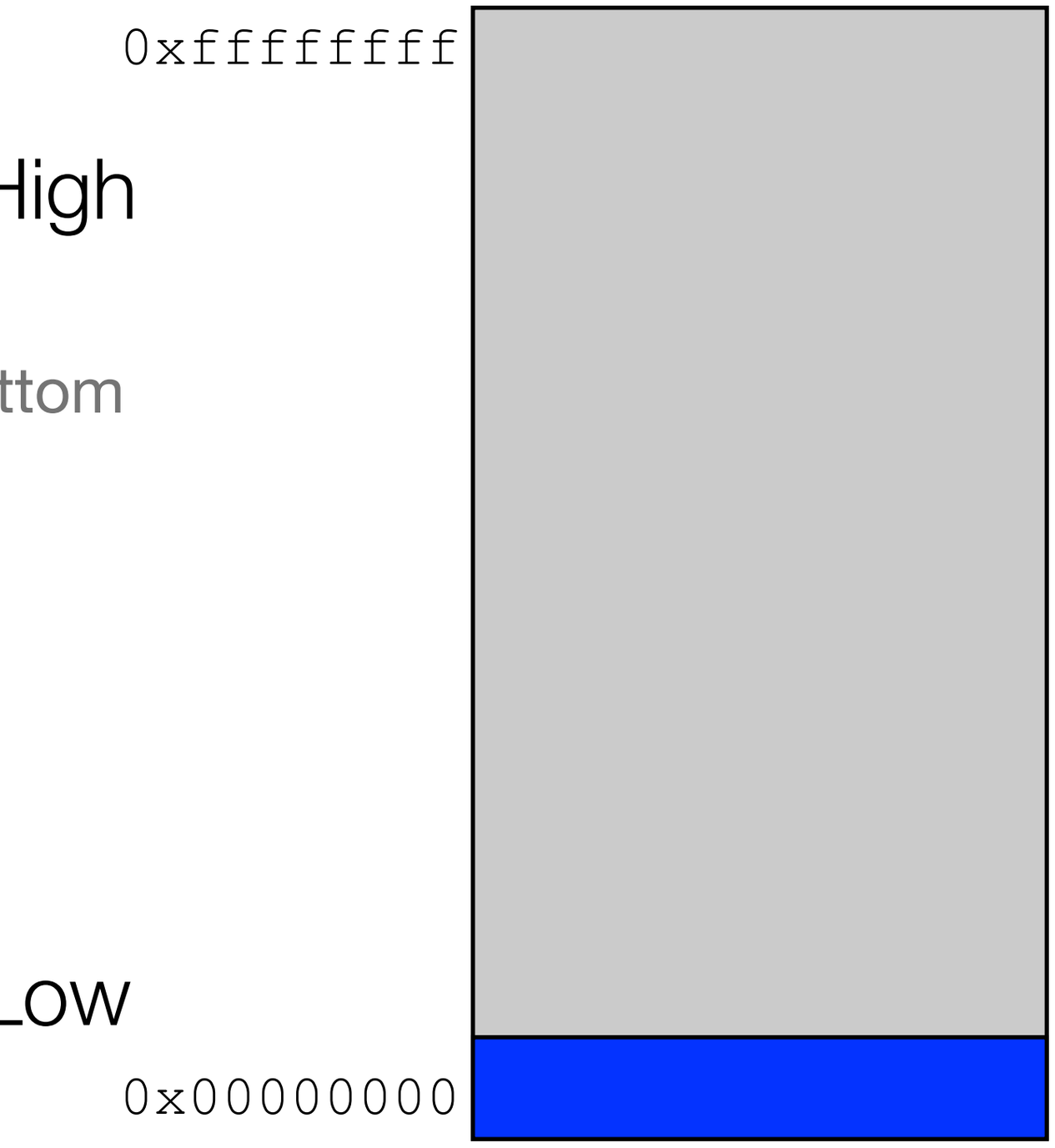
Memory organization

- Memory….
  - Every *Process* has an
    - **Address Space**



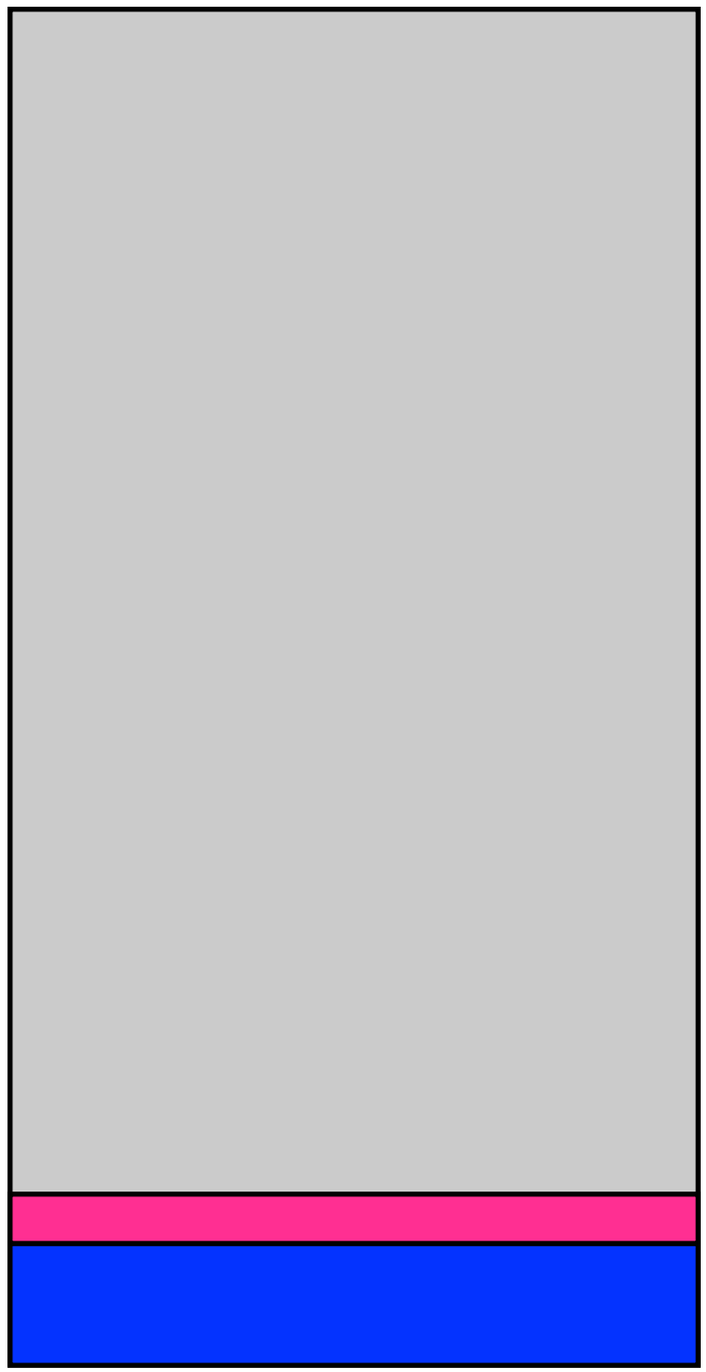
Memory organization

- Memory….
  - Every *Process* has an
    - Address Space
  - **Executable** code is at the bottom
    - Not exactly from address 0



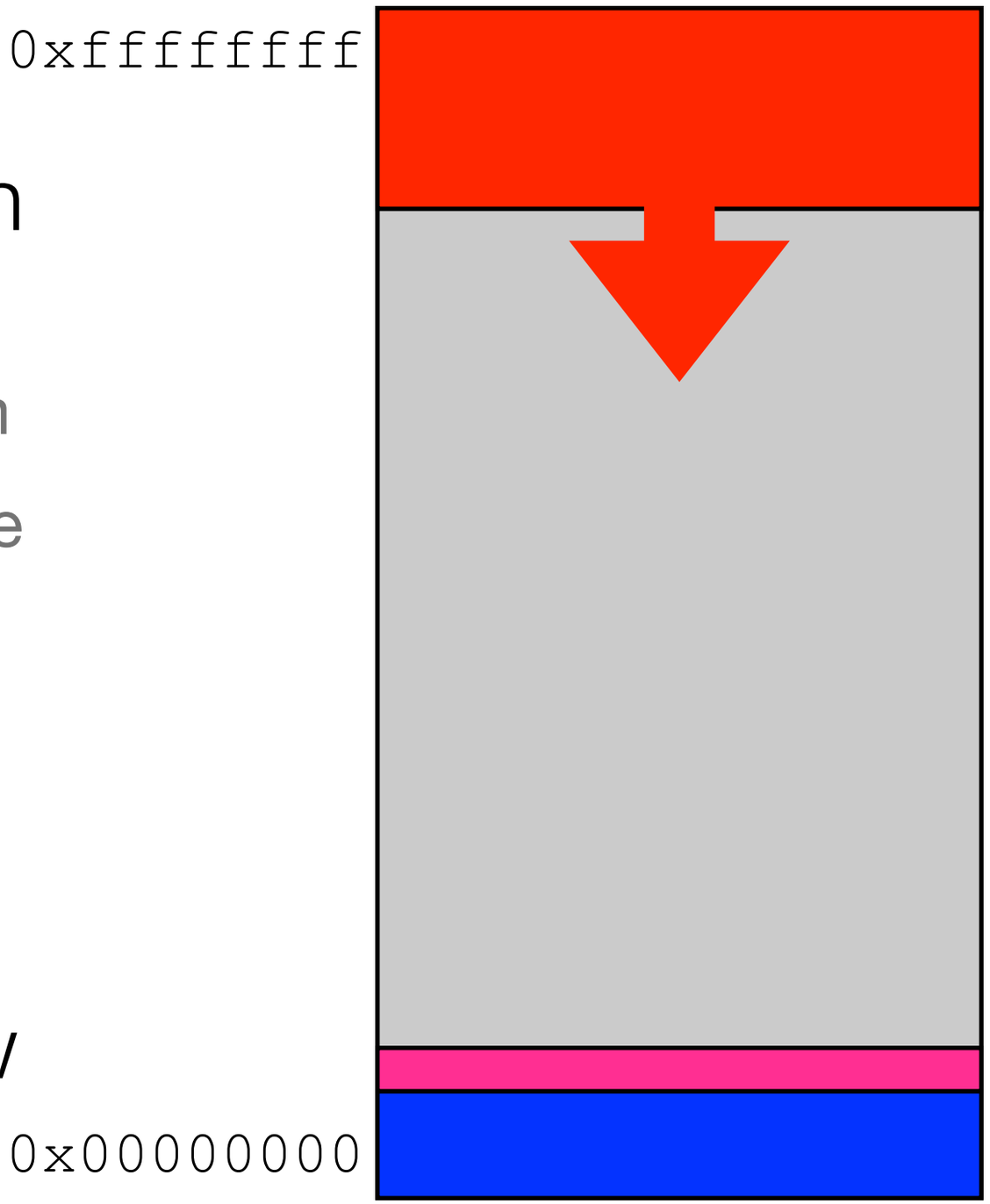
Memory organization

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above



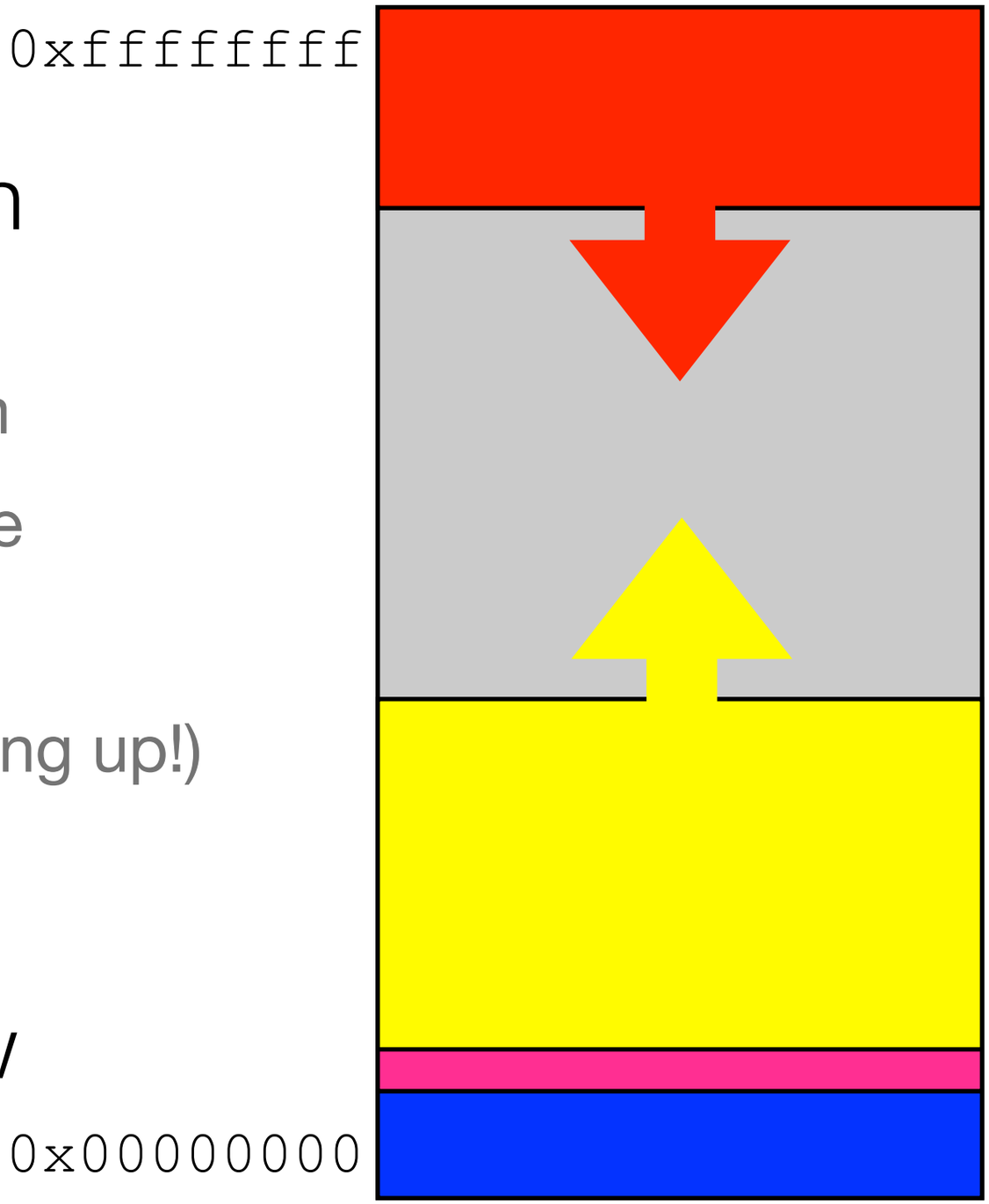
Memory organization

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above
  - **Stack** is at the top (going down!)



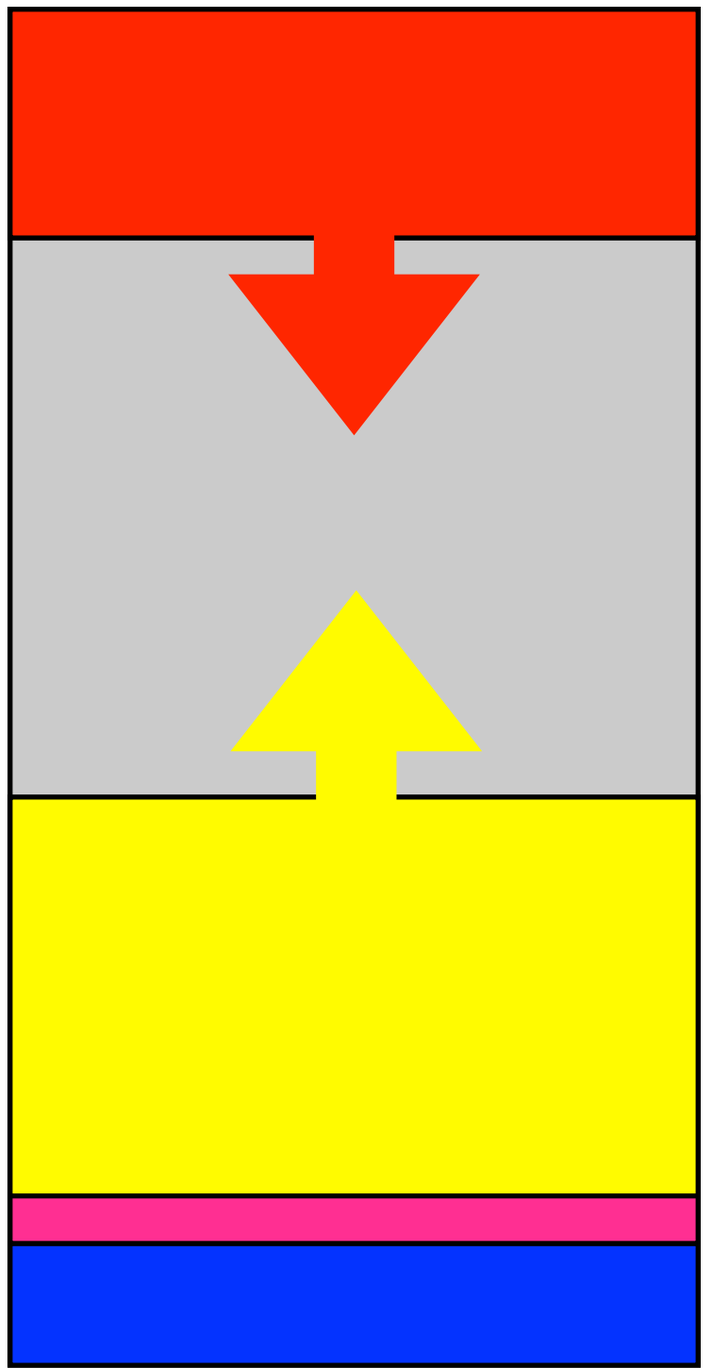
Memory organization

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above
  - **Stack** is at the top (going down!)
  - **Heap** grows from the bottom (going up!)



Memory organization

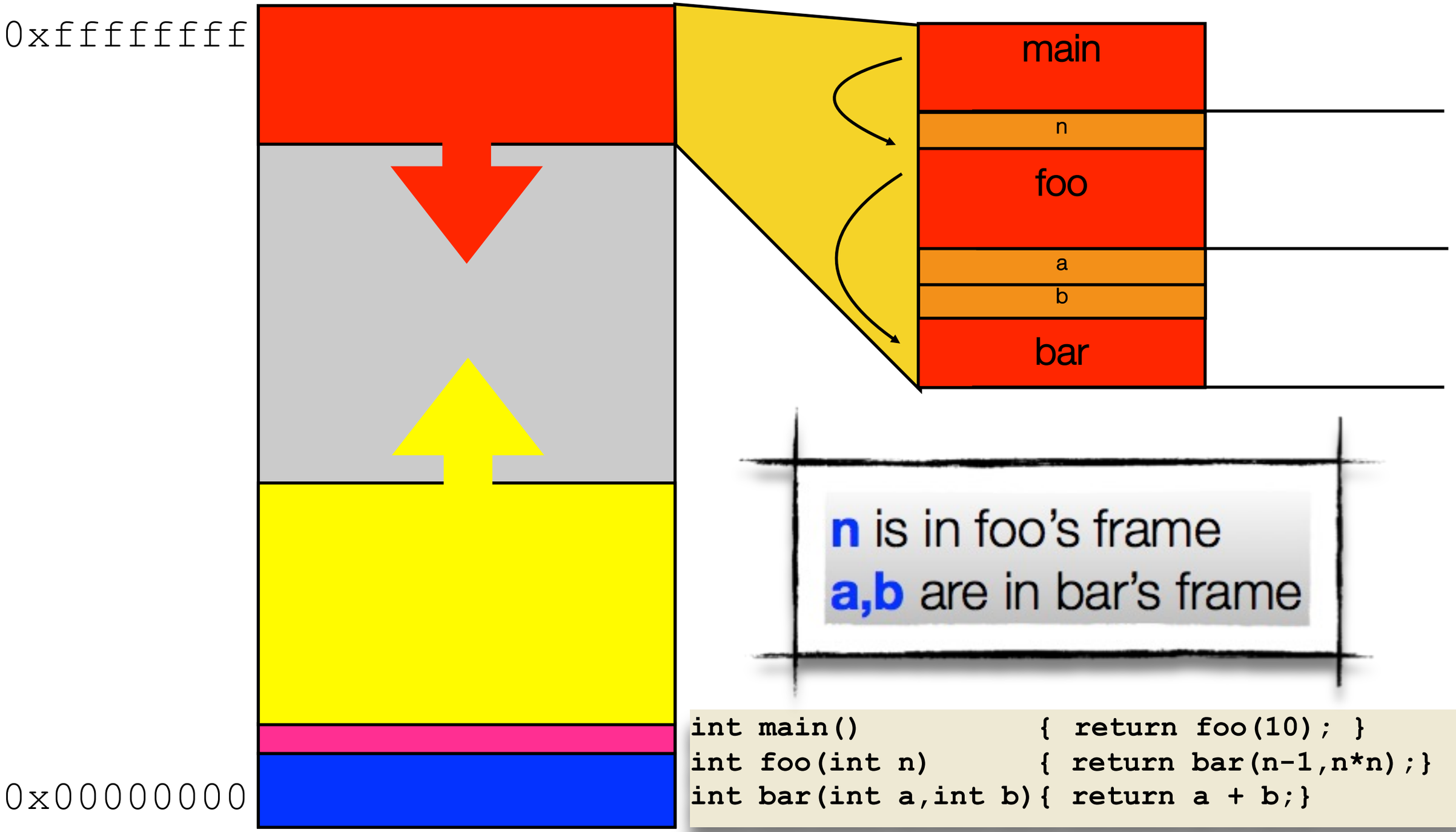
- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above
  - **Stack** is at the top (going down!)
  - **Heap** grows from the bottom (going up!)
  - **Gray** no-man's land is up for grab



Low end may not start from 0.

High end may not be the 0xff…ff.

Zooming in on the stack!



n is in foo's frame
a,b are in bar's frame

```
int main()              { return foo(10); }
int foo(int n)          { return bar(n-1,n*n);}
int bar(int a,int b){ return a + b;}
```

Recursion

- Recursive calls are supported

```
int power(int base,int n) {
   int rv = 1;
   while (n>0) {
      rv *= base;
      n--;
   }
   return rv;
}
```

```
int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}
```

Recursive version….

Induction on n

Base case returns 1

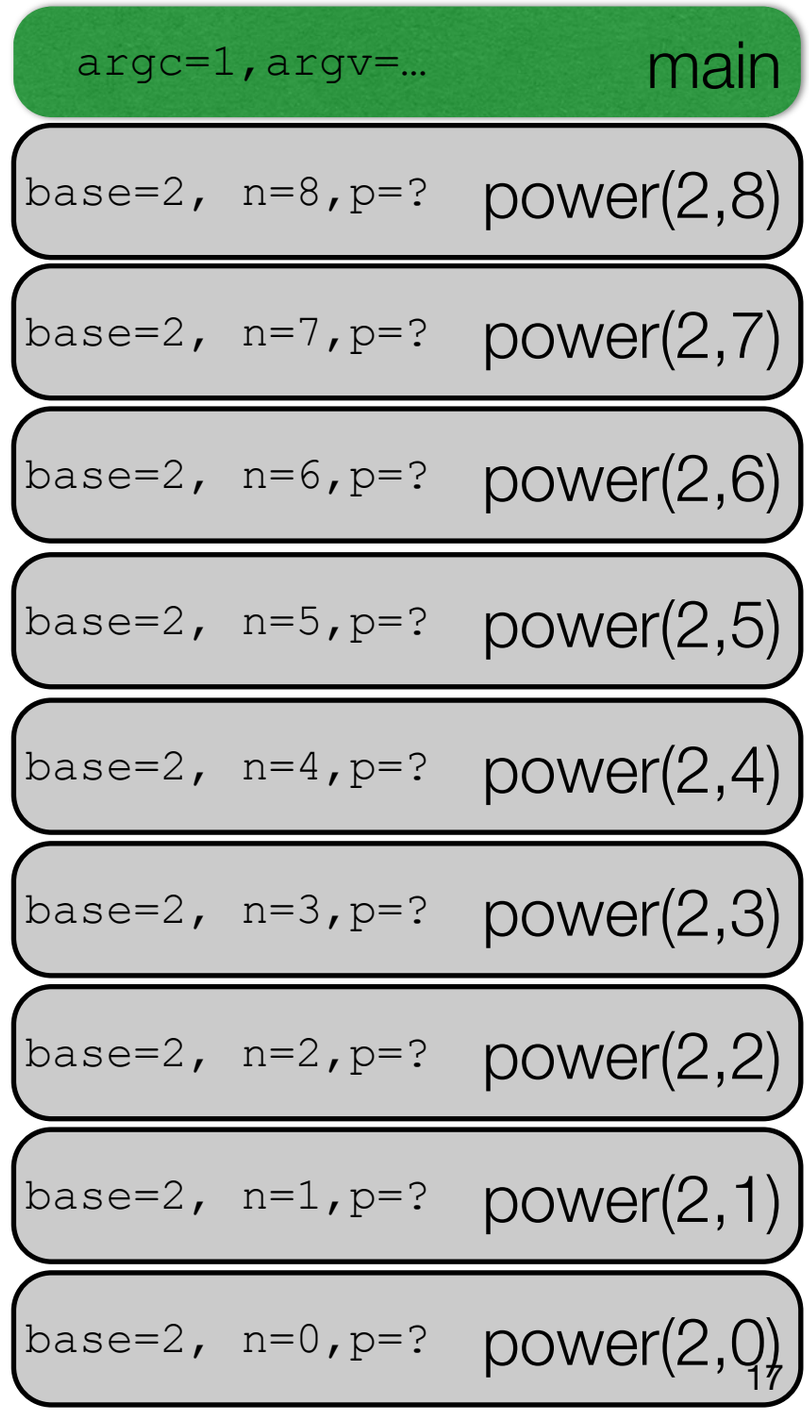
Inductive case multiplies by base

Example – Stack Frame and Recursive Call

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

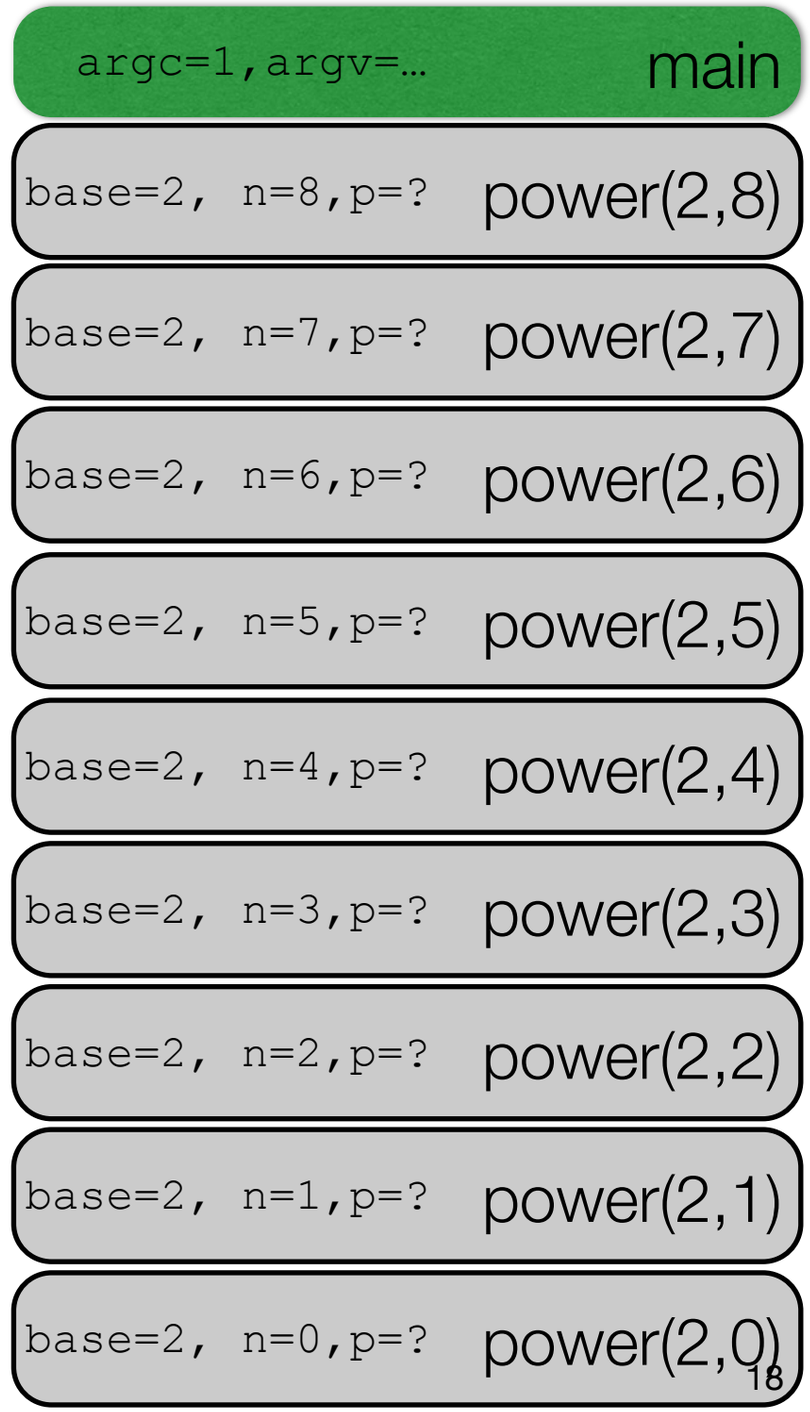


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

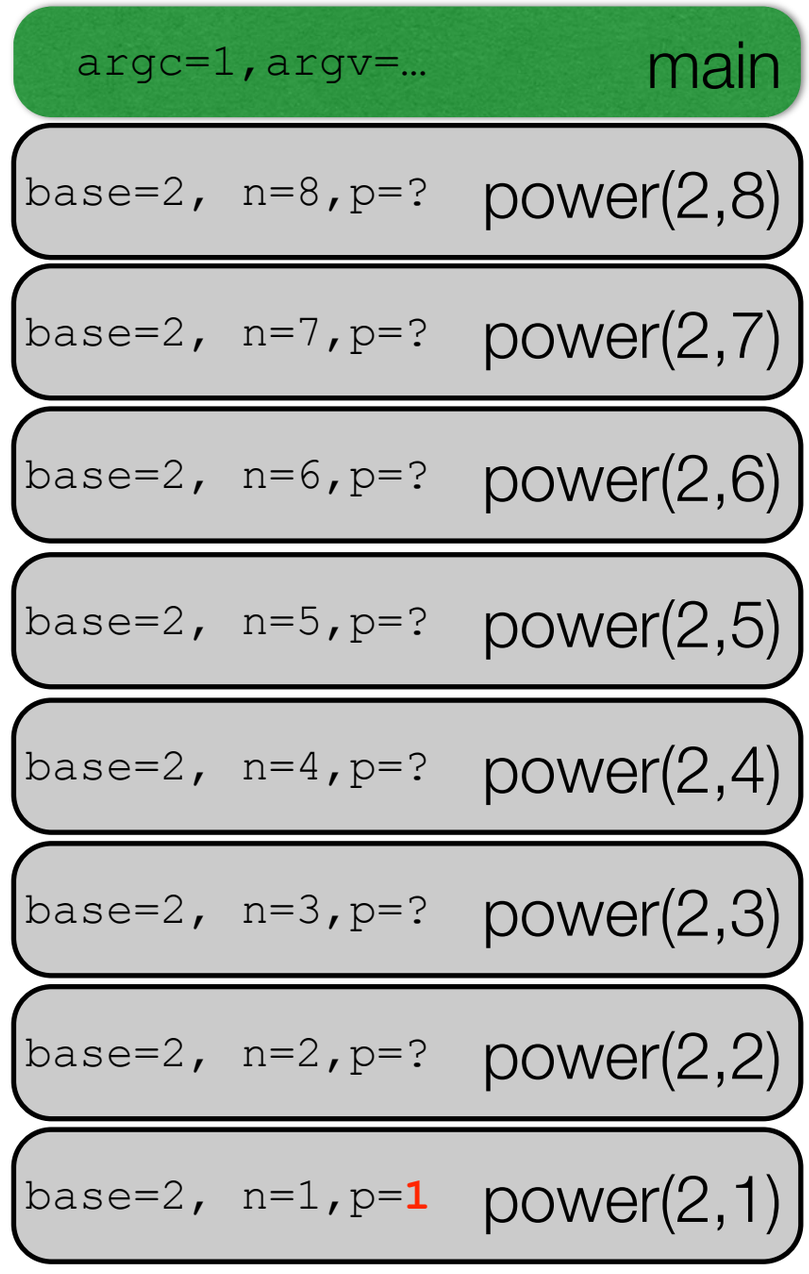


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

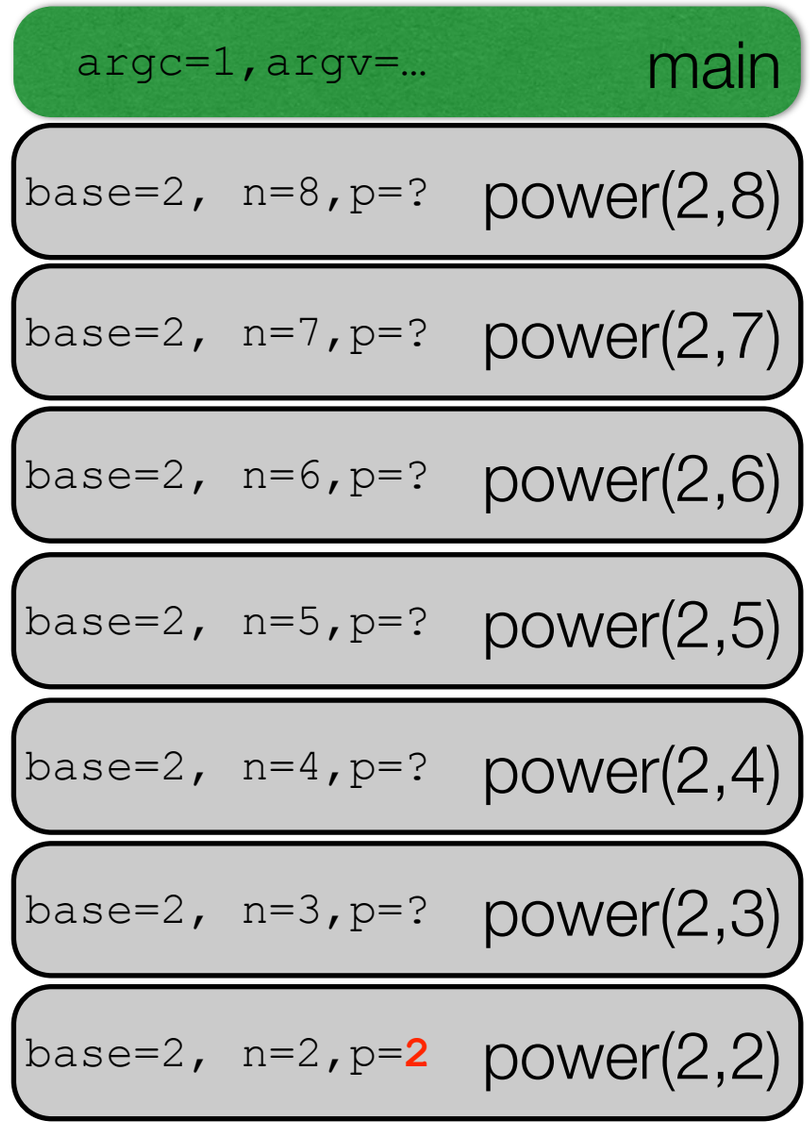


Example

```
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

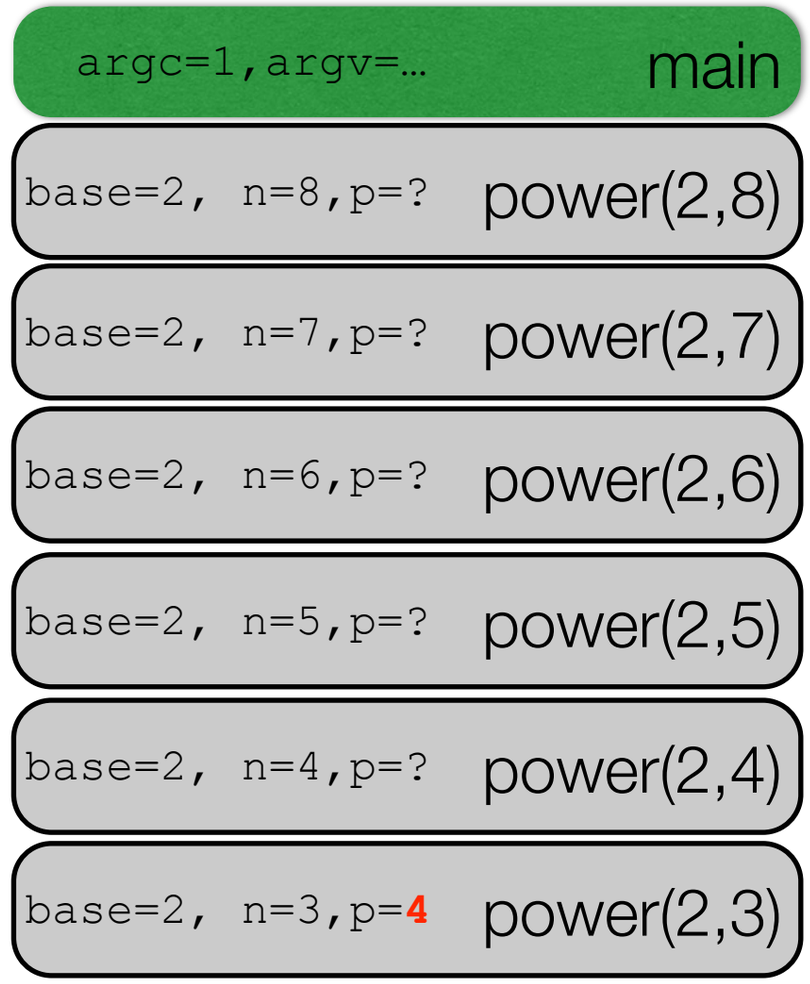


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

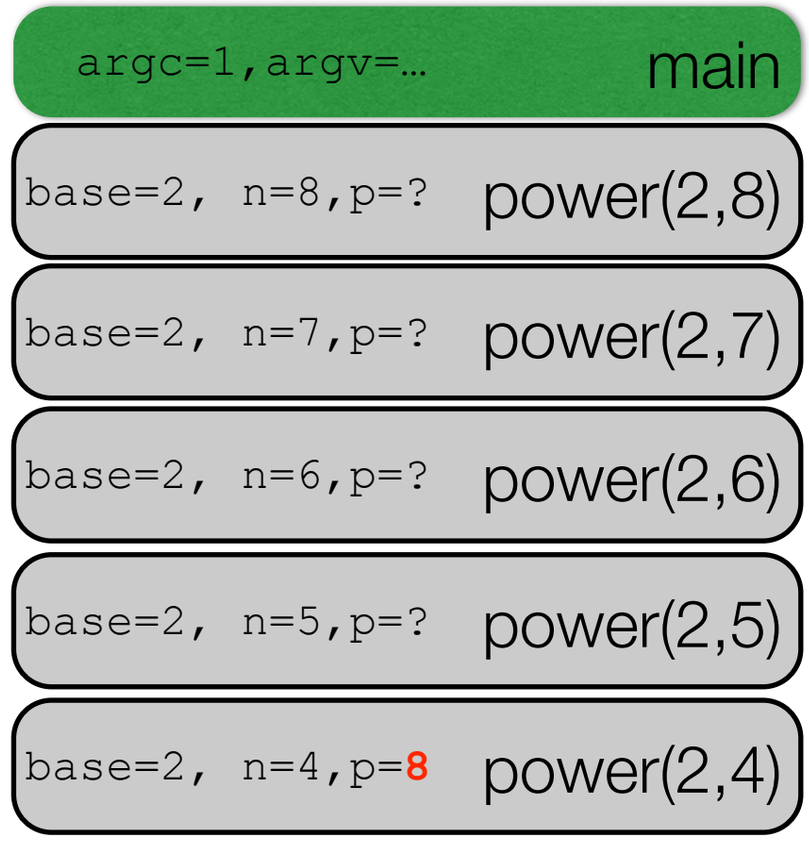


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

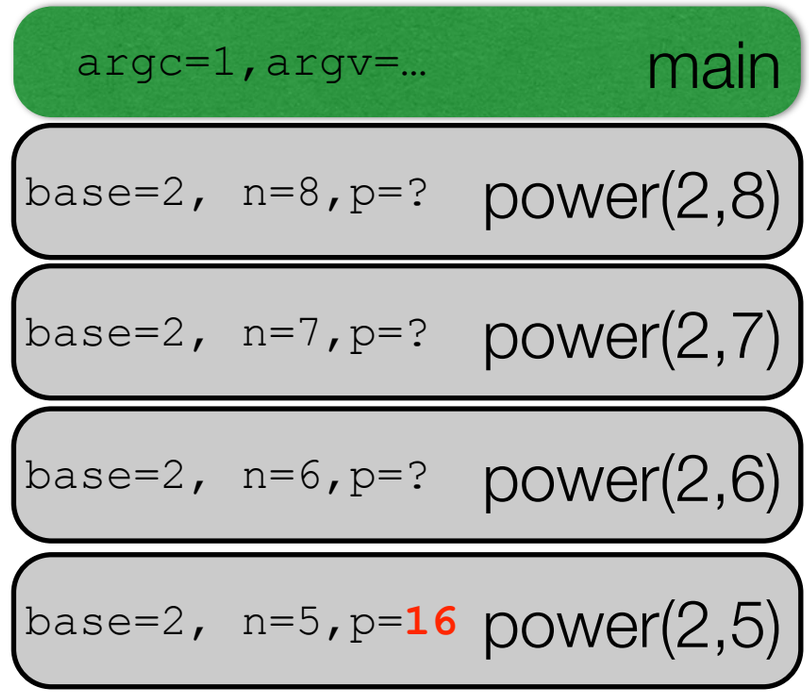


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

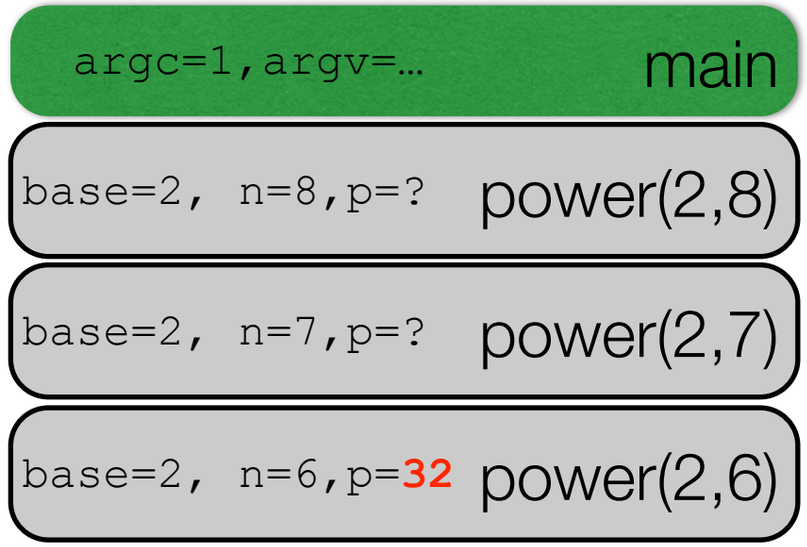


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

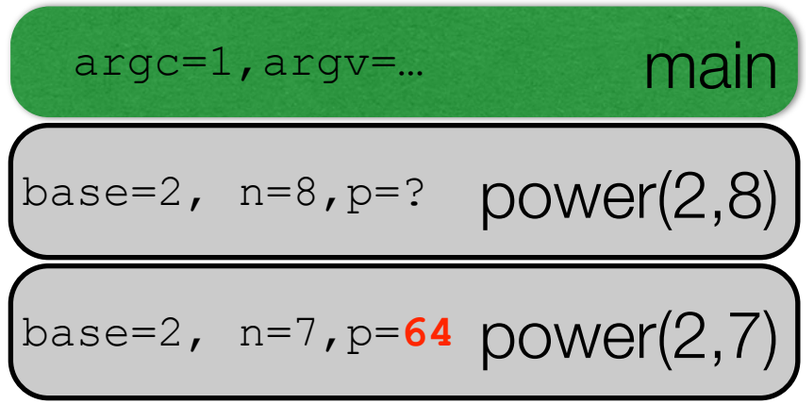


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

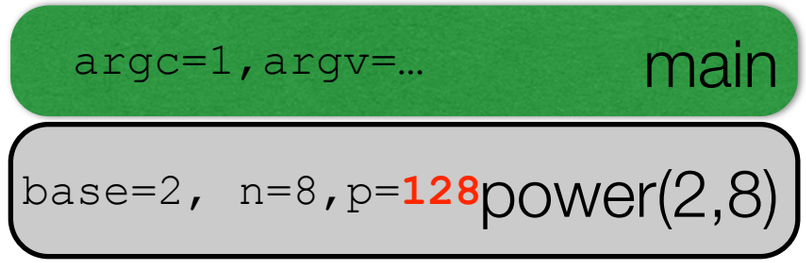


Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```



Example

```c
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```

```
   argc=1,argv=…
                                  main
returned Value=256
```

Output

```
2^8 = 256
```

Study the rest of the slides yourself

Example – Variable Scope

a.c

```
static int i;
int j;

int foo(int n)
{
  int rv;
  static int k;
  // can access i, j, k
}

int bar(int n)
{
   // can access i, j

}
```

b.c

```
// declare j is external
// note the keyword extern
extern int j;

// Cannot define another j

// Can define another i
// this is a different i
static int i;

int f1(void)
{
  // can access j (in a.c)
  j++;

}
```

Order of evaluation of arguments

```
i = 1;
foo(i++, i++);        // calling foo
```

Is it foo(1, 2) or foo(2, 1)?

Do not assume one way or another.

Macros as “fast functions”

- Macros can take arguments!
  - Typical for implementing MIN / MAX

```
#define MIN(x,y) ((x) <  (y) ? (x) : (y))
#define MAX(x,y) ((x) >= (y) ? (x) : (y))

int main()
{
   int a = 10, b = 20;
   int x = MIN(a,b);
}
```

- Notice how
  - Arguments are always put in parentheses
  - Why?

Example

```c
#include <stdio.h>
#define MULG(x,y) ((x)*(y))
#define MULB(x,y) (x*y)

int main()
{
   int x = MULG(99+1,2);
   int y = MULB(99+1,2);
   printf("x is %d\n",x);
   printf("y is %d\n",y);
   return 0;
}
```

```
src (master) $ cc macros.c ; ./a.out
x is 200
y is 101
```

Variable numbers of arguments


```
printf(“Hello”);
printf(“Hello, %s!”, name);
```

How can you pass different numbers of arguments?

```
int foo(int n, ... );   // prototype
int foo(int n, ... )      // implementation
{
// use functions in <stdarg.h> to access the arguments
// va_start(), va_arg()
}
```

Example: static variable in a function

- The “silly” function contains a static declaration
  - "hidden" is set to 0 **only** the first time silly is executed
  - Its value is retained from call to call
  - The function output no longer depends on its inputs alone!

```
int silly(int x) {
    static int hidden = 0;
    return x * 2 + hidden++;
}
```

IN	OUT	hidden	hidden
1	2	0	1
1	3	1	2
1	4	2	3
2	7	3	4
…	…	…	…

It could be worse

- "hidden" variable declared outside silly!
  - Its value is retained from call to call
  - The function output no longer depends on its inputs alone!
  - It (hidden) can be changed by other functions!

```
static int hidden = 0;

int silly(int x) {
  return x * 2 + hidden++;
}

void innocuous(int z) {
   hidden = hidden * z + 10;
}
```

IN	OUT	hidden	hidden
1	2	0	1
1	3	1	2
1	4	2	3
2	7	3	4
…	…	…	…