A C Primer (Part I)

Ion Mandoiu
Laurent Michel

Overview

- C resources
- C in context, briefly
- Getting started
- Basics
  - Names
  - Types
  - Expressions
- Control flow (briefly)
- Functions (briefly)

Resources

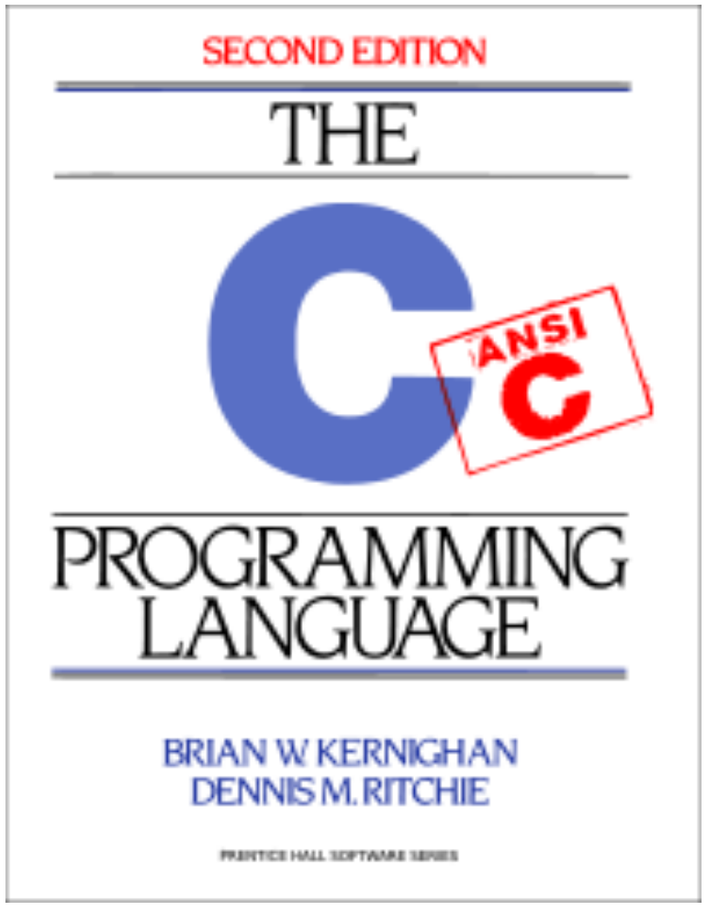
- What you do need….
  - A C compiler
  - A linker
  - A text editor
- We will use the GNU toolchain
  - Compiler: gcc (also known as cc)
  - Linker: gcc (also known as cc)
  - Editor: [I'll use emacs, take your pick! — but ***not*** word—]

C in context

- Pedigree
  - Context: Implementation of UNIX
  - Ancestry: B/BCPL [Ken Thompson]
  - Developed: ’69 to ’73
  - Author: Dennis Ritchie
  - Paradigm: Imperative / Structured Programming
  - Descendents: C++/C#/Objective-C/Java/PHP/Python/D/….

C in Context

- Classic Book
  - Kernighan & Ritchie
  - The “K&R” C
- C is still evolving
  - Different standard versions
    - K&R C 1978
    - ANSI C 1989
    - C99 1999
    - C11 2011
  - We will start with K&R and *very* quickly move to C99



Design principles

- C should be…
  - Simple
  - Easy to compile
  - Typed [weakly]
  - Support low-level memory access
  - Ideal for embedded controller, OS, …
- Yet…
  - C is *powerful*
  - C is *fast*

Paradigm

- C is a purely **procedural** language
  - No object-orientation whatsoever
- Central Dogma
  - Object of interest: Computations
  - Main abstraction tool: Procedures
    - Caller / Callee dichotomy
    - Abstracts away “How things are done”
- Programming means
  - Organizing processes as procedures
  - Composing processes through procedure calls

Paradigms

Procedural

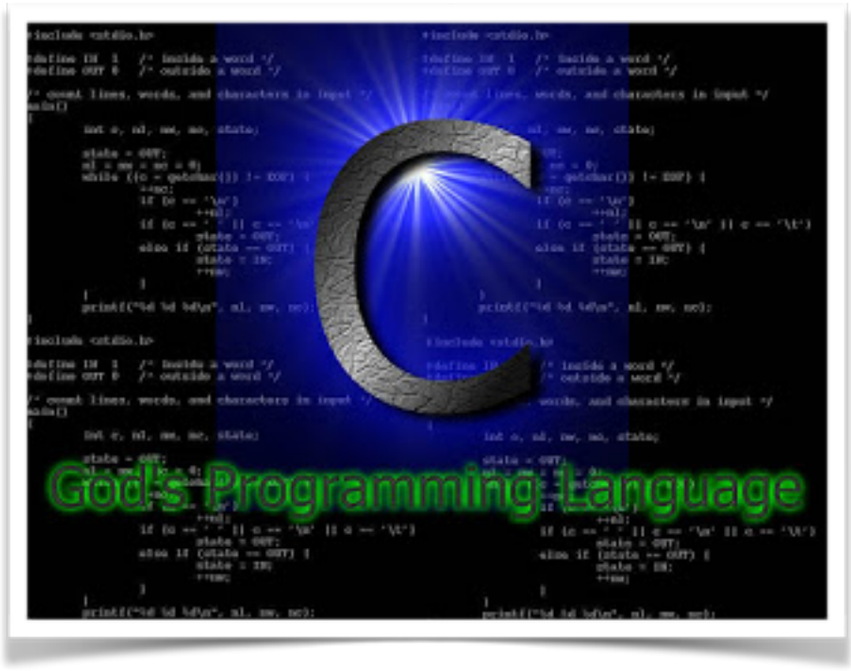
Functional

Object-Oriented

Logic

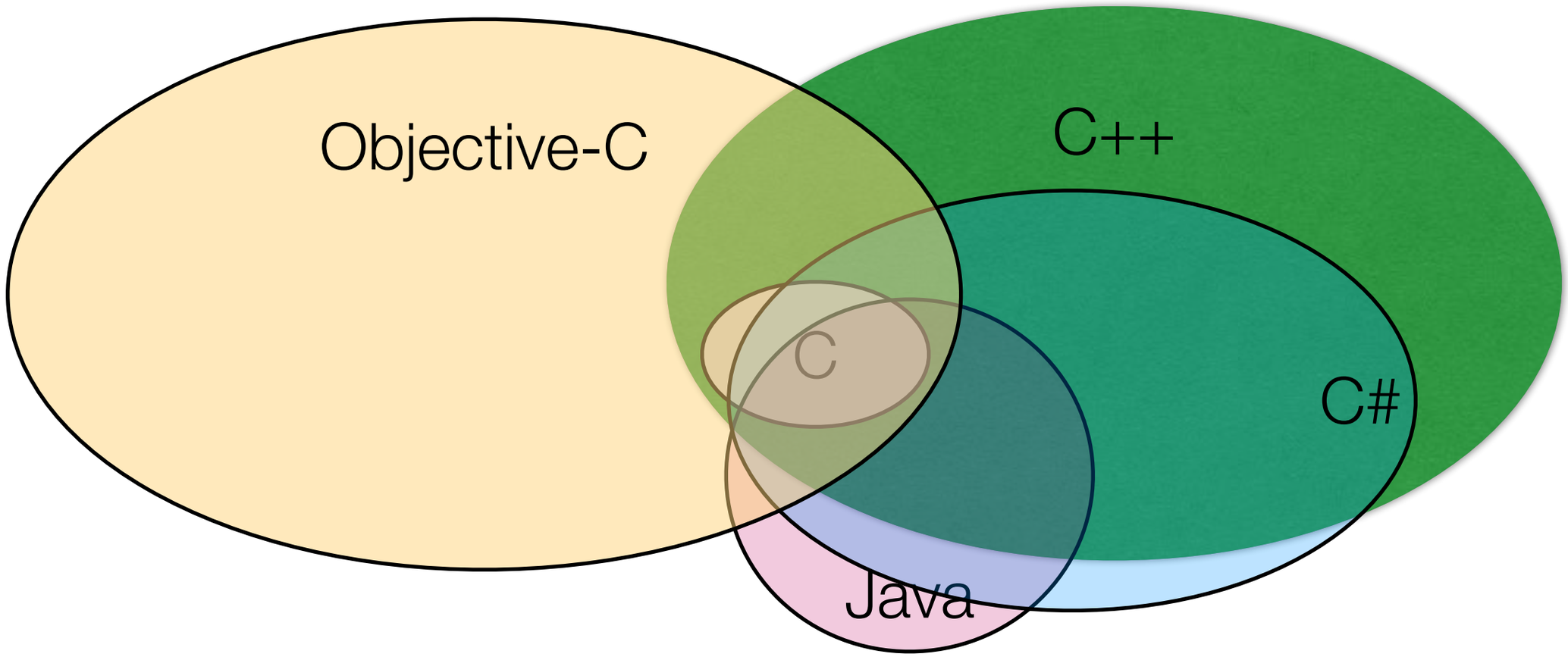
Procedural Programming in C

- Adheres to the philosophy
- Generates very efficient code
- Exposes as many low-level details you wish to see
- Provides full control over memory management (no GC!)
- The Programmer is **fully in charge**



C and “The Others”

- Brief VENN Diagram



Morale

- Syntax of Java / C# / C++
  - Is deeply rooted in C
  - Easy to adapt
  - Mostly need to “forget” O.O. related syntax
- Environment
  - Quite different
    - Compiled
    - Manual build/link process

Getting Started

- Your very first program! A classic!
  - A single hello.c source file
  - a preprocessor directive
  - defines a single function ‘main’
    - calls a function ‘printf’
    - returns a value to the O.S.
- Compile with…
  - A single command ‘cc’
    - Compiles *and* links
  - Output is called ‘a.out’

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c
$ a.out
```

Getting Started

- Word to the wise…
  - The K&R version is even *simpler*
  - It does not specify arguments
  - K&R C does not even care!
  - What identifies the function is just its *name* and nothing else

```c
#include <stdio.h>

int main()
{
   printf("Hello world!\n”);
}
```

Terminal

```
$ cc hello.c
$ a.out
```

Word to the Wise

- K&R C is **very** permissive
  - You can omit the return type (defaults to int)
  - You can omit the argument list (if you do not use them)
  - You can omit return statements (output is then undefined)

- Real Code [C99] ?

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

arguments

array of arguments

Getting Started

- Comments?
  - The compiler ignores everything between “/*” and “*/”
  - Comments are meant to be human readable!
  - Comments can be multi-line

```
/* my first program */

#include <stdio.h>

int main(int argc, char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

```
/**********************
 * my first program *
 **********************/

#include <stdio.h>
…
```

Getting Started

- What's up with #include ?
  - It "imports" a header file with the specification of utility functions that exist in a ***library*** to be linked with the program

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c
$ a.out
```

Getting Started

- What is ‘main’ ?

  - A *special* function that defines the entry point for the program. This is where the Operating System transfers control once the program starts.

- Two inputs:

  - the number of arguments

  - an array of arguments [more on this later]

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c
$ a.out
```

Getting Started

- What is ‘printf’ ?

  - A C *library* function to print on the *standard output* for the process.
  - It takes at least a string
  - ‘\n’ is a newline character

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c
$ a.out
```

Getting Started

- Why do we ‘return’ anything?
  - When the process dies it must tell the O.S. how things went. This is the way to report back.
  - Returning a status of 0 means ‘everything went according to plans’

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c
$ a.out
```

Getting Started

- What is *cc* doing really ?
  - Three steps
    - preprocesses hello.c
    - compiles hello.c to hello.o
    - links hello.o with libc
    - writes executable file a.out
  - You **can** and **often will** separate those steps!

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c
$ ./a.out
```

Getting Started

- The output executable…
  - Is running from the current working directory ./
  - It's name can be changed!

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c  -o hello
$ ./hello
```

Getting Started

- What about debugging ?
  - Two steps
    - Add ‘debug information’
    - Run a debugger
- Demo!

```c
#include <stdio.h>

int main(int argc,char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

Terminal

```
$ cc hello.c -g -o hello
$ gdb hello
```

Second Program

- Purpose
  - Read an integer from the standard input: **n**
  - Compute the sum of all integers between 0 and **n**
  - Print the result on the standard output
- New concepts
  - standard input and output

The Code

A Few Caveats

```c
/*
   Compute the sum of the integers
   from 1 to n, for a given n
*/
#include <stdio.h>

int main(void) {
   int i, n, sum;
   sum = 0;
   printf("Enter n:\n");
   scanf("%d", &n);
   i = 1;
   while (i <= n) {
      sum = sum + i;
      i = i + 1;
   }
   printf("Sum from 1 to %d = %d\n", n, sum);
   return 0;
}
```

main takes nothing

Local Declarations

The Code

A Few Caveats

```c
/*
  Compute the sum of the integers
  from 1 to n, for a given n
*/
#include <stdio.h>

int main(void) {
  int i, n, sum;
  sum = 0;
  printf("Enter n:\n");
  scanf("%d", &n);
  i = 1;
  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }
  printf("Sum from 1 to %d = %d\n", n, sum);
  return 0;
}
```

Reading from standard input

Assignments

The Code

A Few Caveats

```
/*
  Compute the sum of the integers
  from 1 to n, for a given n
*/
#include <stdio.h>

int main(void) {
  int i, n, sum;
  sum = 0;
  printf("Enter n:\n");
  scanf("%d", &n);
  i = 1;
  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }
  printf("Sum from 1 to %d = %d\n", n, sum);
  return 0;
}
```

Formatted output

The Code

```c
/*
  Compute the sum of the integers
  from 1 to n, for a given n
*/
#include <stdio.h>

int main(void) {
  int i, n, sum;
  sum = 0;
  printf("Enter n:\n");
  scanf("%d", &n);
  i = 1;
  while (i <= n) {
    sum = sum + i;
    i = i + 1;
  }
  printf("Sum from 1 to %d = %d\n", n, sum);
  return 0;
}
```

Running It [Demo!]

```
$ cc sum.c
$ ./a.out
Enter n:
100
Sum from 1 to 100 = 5050
$
```

C Primer - Expressions and Types

Basics

- Similar to Java/C++/Python Basics
- Expressions are inductively defined…
  - Leaves
    - Constants (aka Literals)
    - Local variables
    - Arrays
  - Internal nodes
    - For pretty much anything else [operators,field access,….]
- Data Types

Constants

- Very simple idea
  - You cannot change them
  - You cannot name them
  - You can specify constants for different *types*
    - Integers (Decimal / Hexadecimal)
    - Floats / Double
    - Characters
    - Strings

```
200
0x7fffffff
3.1415
2.5L
'h'
'\n'
"Hello world!\n"
```

Local Variables

- Again…. Remember Java! [and to some extent Python]
  - Local variables have **types**
  - Local variables are **declared** inside a function [99% of the time]

```
int main() {
   int celcius,fahr;
   int lower,upper;
   lower = 0;
   upper = 300;
   fahr = 90;
   celcius = 5 * (fahr - 32) / 9;

   printf("From %d Farenheit to %d celcius degrees\n",fahr,celcius);
    return 0;
}
```

declare local variables

initialized to constants

computed….

Printed!

Assignment

- Again, same story as in Java/Python

```
lhs = expression;
```

  - LHS = Left Hand Side
    - Something that can be *written to*
  - Expression
    - Based on arithmetic operators
    - Based on array access [more later]
    - Based on pointer manipulations

Critical Observation

- For expressions and assignments
  - In C, everything has a **TYPE**
  - The LHS
  - The Expression
    - Every “node” inside the expression
- C **checks** that the assignment makes sense
  - Type of LHS is “compatible” with type of Expression

Examples

- What is going on here ?

Once again, C is amazingly permissive….
But it ***warns*** *you* that this looks fishy
Beware!
Read the warnings!
Heed them!

```
    fahr = 91.45;

    return 0;
}
```

```
Admin (master) $ cc type.c -c
type.c:8:11: warning: implicit conversion from 'double' to 'int' changes value from 91.45 to 91 [-Wliteral-conversion]
   fahr = 91.45;
        ~ ^~~~~
1 warning generated.
```

Assignments **ARE NOT** Statements

- Assignments are ***expressions***
  - You can chain them!

```c
#include <stdio.h>

int main()
{
   int a,b,c;

   a = b = c = 10;
   printf("a = %d, b = %d, c = %d\n",a,b,c);

   a = (b = (c = 20));
   printf("a = %d, b = %d, c = %d\n",a,b,c);
}
```

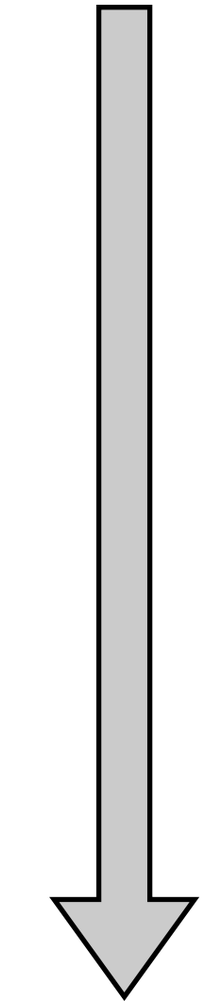
Expressions in a Nutshell [K&R, Chapter 2]

- All the conventional arithmetic operators
- Pre/Post increment/decrement
  - Also available in Java: i++ ++i —j j—
  - Use them wisely
- Conditional assignments….
  - An if-then-else as an expression [COND **?** ifTrue **:** ifFalse]
  - Also exist in Python [ifTrue **if** COND **else** ifFalse]
- Usual precedence and associativity
- Many more later on with arrays, pointers and bits.

Operator Precedence

- Simple conventional semantics

Most



Least

Operators	Associativity
`( )`	left to right
`! ~ ++ -- + - * (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
	left to right

Basic Data Types

- A few ***integer*** types
  - Consider x86_64 (a 64-bit architecture)

size (in bits)	signed	unsigned
8	`char` `-128 .. 127`	`unsigned char` `0..255`
16	`short` `-32768..32767`	`unsigned short` `0..65535`
32	`int` $-2^{31} .. 2^{31} - 1$	`unsigned int` $0..2^{32}-1$
64	`long` $-2^{63} .. 2^{63} - 1$	`unsigned long` $0..2^{64}-1$
64	`long long` $-2^{63} .. 2^{63} - 1$	`unsigned long long` $0..2^{64}-1$

Basic Data Types

- A few ***integer*** types
  - Consider i386 (“old-fashioned” 32-bit architecture)

size (in bits)	signed	unsigned
8	`char` `-128 .. 127`	`unsigned char` `0..255`
16	`short` `-32768..32767`	`unsigned short` `0..65535`
32	`int` $-2^{31} .. 2^{31} - 1$	`unsigned int` $0..2^{32}-1$
32	`long` $-2^{31} .. 2^{31} - 1$	`unsigned long` $0..2^{32}-1$
64	`long long` $-2^{63} .. 2^{63} - 1$	`unsigned long long` $0..2^{64} -1$

Bottom Line

- long and unsigned long….
  - Both correspond to the *native* machine word width
    - 32-bit on 32-bit architectures
    - 64-bit on 64-bit architectures
  - It will matter when we focus on pointers…

Character type

- A quick word…
  - Characters are 8-bit wide.
  - Typically use ASCII code
  - Every integer in 0..127
    - Yields a “symbol”
- Characters in 0..31 range…
  - Are “Control” character (aka, non-printable)

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

So…

- The character ‘H’ is none other than….
  - 72
- Observe how…
  - ‘0’ through ‘9’ are consecutive!

```
char ch = ‘8’;
char z  = ‘0’;

int x   = ch - z;
```

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Non-printable characters?

- These are sometimes useful
  - Showing the constant (literal)

‘\n’	newline
‘\r’	carriage-return
‘\f’	form-feed
‘\t’	tabulation
‘\x7’	audible bell
‘\b’	backspace

Basic Data Types

- What about booleans ?

There are no booleans in C!

- Use integers or even better: characters
  - char is small (1 byte)
  - char can represent 256 values
- Convention
  - 0 “means” FALSE
  - Anything other than 0 “means” TRUE

Convenience trick

- Use two MACROS to define TRUE and FALSE
  - Use “macros” to
    - Alias BOOL to “char” type
    - Alias TRUE to 1
    - Alias FALSE to 0
  - In reality, **anything** != 0 is TRUE
- Then rely on characters for boolean…

```
#define TRUE   1
#define FALSE  0
```

Example

```c
#include <stdio.h>

#define BOOL char
#define TRUE 1
#define FALSE 0

int main(int argc,char* argv[])
{
    BOOL x = TRUE;
    BOOL y = FALSE;
    BOOL w,z;

    z = x || y;
    w = x && y;
    printf("z is = %d\n",z);
    printf("w is = %d\n",w);
    return 0;
}
```

Basic Data Types

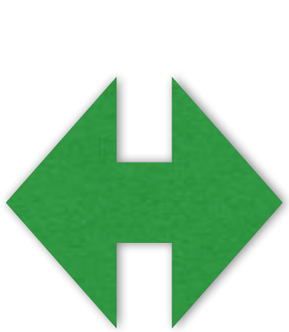
- A few ***floating point*** types
  - Consider x86_64 again

size (in bits)	size (bytes)	Name & Range
32	4	**float** SP=1.17 * 10^{-38} LP=3.40 * 10^{+38}
64	8	**double** SP=2.22 * 10^{-308} LP=1.79 * 10^{+308}
80/128	16	**long double** SP=3.65 * 10^{-4951} LP=1.18 * 10^{+4932}

Type Casting…

- Useful to **convert** an **operand** to another type before doing arithmetic
  - Example: integer to double?

```
int x = 10;
int y = 3;

double z = x / y;
```



```
int x = 10;
int y = 3;

double z = (double)x / y;
```

- Syntax

```
(<Type>)<expression>
```

Be Mindful…

- Are your operand signed or unsigned ?
- What is the size (in bit) of each operand?
- When doing arithmetic
  - C will convert the second operand to the type of the first
  - C may truncate results if the type of assignee is smaller than the type of the result (and the result cannot be represented)
- Example?

```
unsigned int x = 3;
unsigned int y = 7;
unsigned int z = x - y;
```

z holds the representation of -4, but reading it as an “unsigned int” yields a ***very*** different value!

Code Example

```c
#include <stdio.h>
int main()
{
    unsigned int x = 3;
    unsigned int y = 7;

    unsigned int z = x - y;
    printf("%ud result\n",z);
    return 0;
}
```

printf format Flag for “unsigned 32-bit int”

Important note

- Beware of “silly” mistakes with relational operators..


```
x = 42
```

This assigns 42 into variable x (and evaluates to 42)

```
x == 42
```

This tests whether x is equal to 42 and return 0/1 accordingly

C Primer - Control Flow

Control flow [K&R, Chapter 3]

- All the usual suspects from Java/C++/Python
  - blocks
  - for
  - while
  - if-then-else
  - switch
  - break
  - continue

Control

- Control is about
  - Evaluating “boolean” conditions to alter the flow of the program
- Corollary for C
  - Since any integer is a boolean (0 and non-zero) ….
  - Unlike Java, you can use integers in control flow primitives!

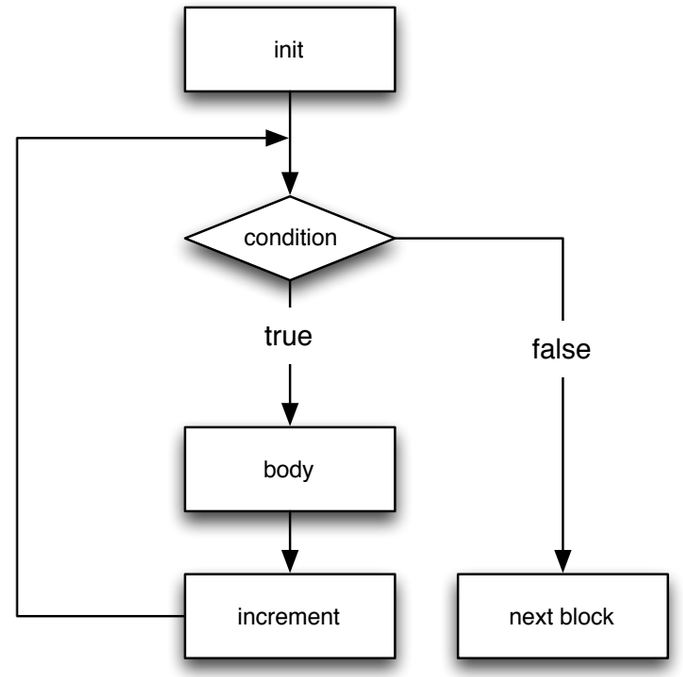
Blocks

- Purpose
  - Group statements together
  - Blocks contain a semicolon separated list of statements
  - Useful for loops / branching / switches
- Extra feature [C99]
  - You can *nest* declarations inside blocks!
  - Examples shortly! [right after we see at least one loop]
- Also known as
  - Compound Statement [K&R, Annex A.9.3]

For Loop

- Just like in Java / C++
  - Called a “bounded” or “counting” loop
  - Syntax

```
for(<expression> ; <expression> ; <expression>)
    <stmt>
```



init can contain a declaration

  - Python refresher

```
for <id> in  <expression>:
    <stmt>
```

For Loop

- As an example
  - A “counting” loop in C

```
for(int i=0 ; i < 10 ; i++)
    <stmt>
```

  - A counting loop in Python

```
for i in  range(0,10):
    <stmt>
```

For Loop example

- Produce a table Fahrenheit to Celcius

```c
#include <stdio.h>

int main() {
   int celcius,fahr;
   int lower,upper;
   int step;

   lower = 0;
   upper = 300;
   step  = 10;
   for(fahr=lower;fahr <= upper;fahr += step) {
      celcius = 5 * (fahr - 32) / 9;
      printf("From %d F to %d C degrees\n",fahr,celcius);
   }
   return 0;
}
```

Relatively simple….

But

We can clean up a lot!

Use “MACROS” for constants

- MACROS are handled by the pre-processor (search & replace)
  - Purely textual trick before the compiler runs
  - Always written in UPPERCASE

More on MACROS later on…

```c
#include <stdio.h>
#define LOWER 0
#define UPPER 300
#define STEP  10
int main(int argc,char* argv[]) {
   int celcius,fahr;
   for(fahr=LOWER;fahr <= UPPER;fahr += STEP) {
      celcius = 5 * (fahr - 32) / 9;
      printf("From %d F to %d C degrees\n",fahr,celcius);
   }
   return 0;
}
```

Fold **Declarations** inside the blocks

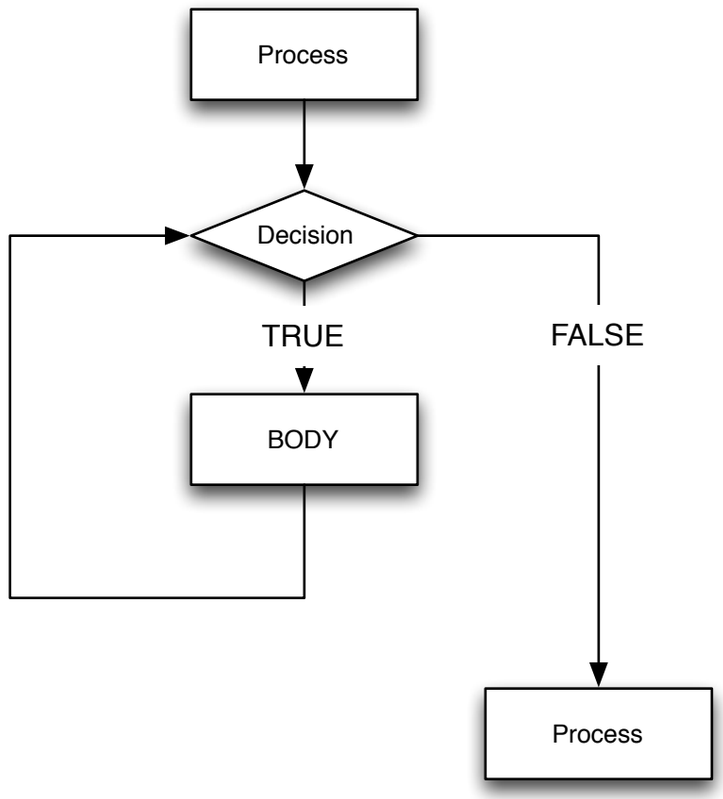
- Two of them…
  - **fahr** in the loop header (it is inside the block)
  - **celcius** inside the loop block

```c
#include <stdio.h>
#define LOWER 0
#define UPPER 300
#define STEP  10
int main(int argc,char* argv[]) {
   for(int fahr=LOWER;fahr <= UPPER;fahr += STEP) {
       int celcius = 5 * (fahr - 32) / 9;
       printf("From %d F to %d C degrees\n",fahr,celcius);
   }
   return 0;
}
```

It keeps the declaration close to the statements using them

While Loops

- Same as Java/C++




```
while (c)
   <body>
```

While Loops

- Just like in Java/C++
  - Called an “unbounded” loop
  - Syntax (indentation *does not* matter. But it makes it clean looking!)

```
while (<expression>)
    <stmt>
```

- Python refresher

```
while <expression>:
    <stmt>
```

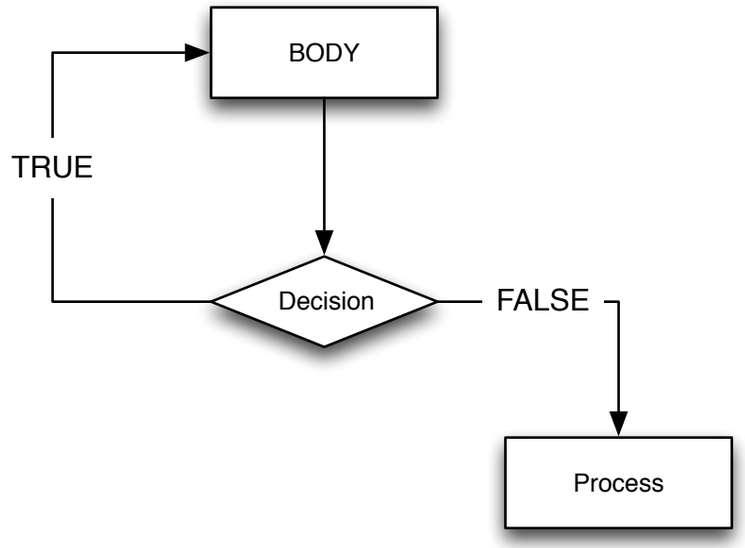
While loop example

- Again, the degree conversion….

```
#include <stdio.h>
#define LOWER 0
#define UPPER 300
#define STEP  10
int main(int argc,char* argv[]) {
   int fahr=LOWER;
   while (fahr <= UPPER) {
      int celcius = 5 * (fahr - 32) / 9;
      printf("From %d F to %d C degrees\n”,fahr,celcius);
      fahr += STEP;
   }
   return 0;
}
```

Do-While Loop

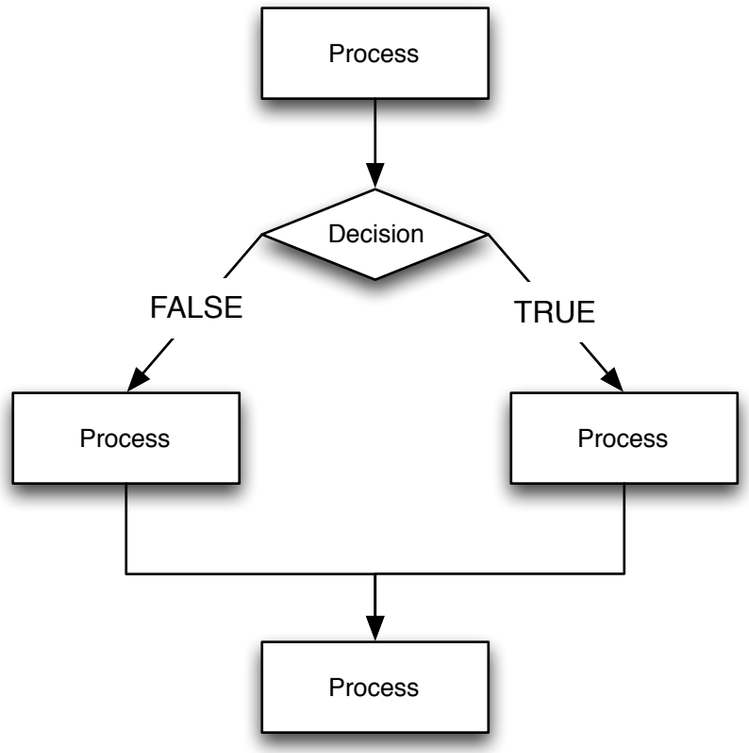
- Same as in Java



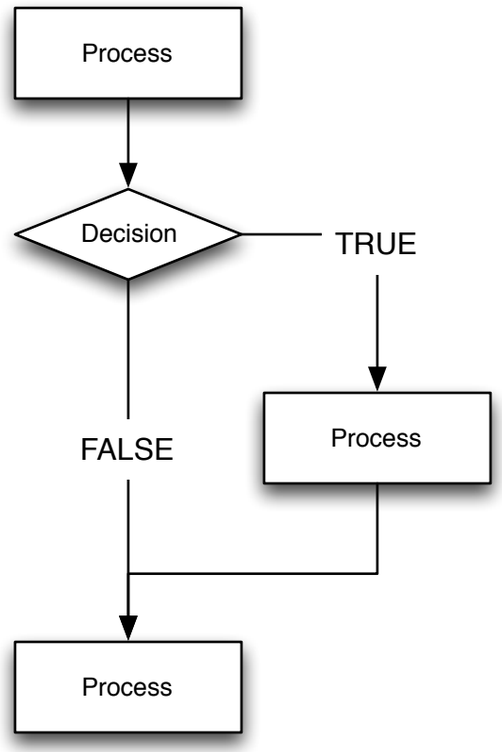

```
do
   <body>
while (c);
```

Branching

- Same as in Java
  - Two variants




```
if (c)
   <block>
else
   <block>
```




```
if (c)
   <block>
```

If-Then-Else

- Syntax

```
if (<expression>) <stmt>
if (<expression>) <stmt> else <stmt>
```

- Beware of the dangling else.
  - Use blocks to disambiguate (again, like in Java)

If-Then-Else

- Relation to Python
  - Again: indentation carries no information in C
  - Again: block structure is *explicit* via a C block
  - Condition **must be** parenthesized
- C version

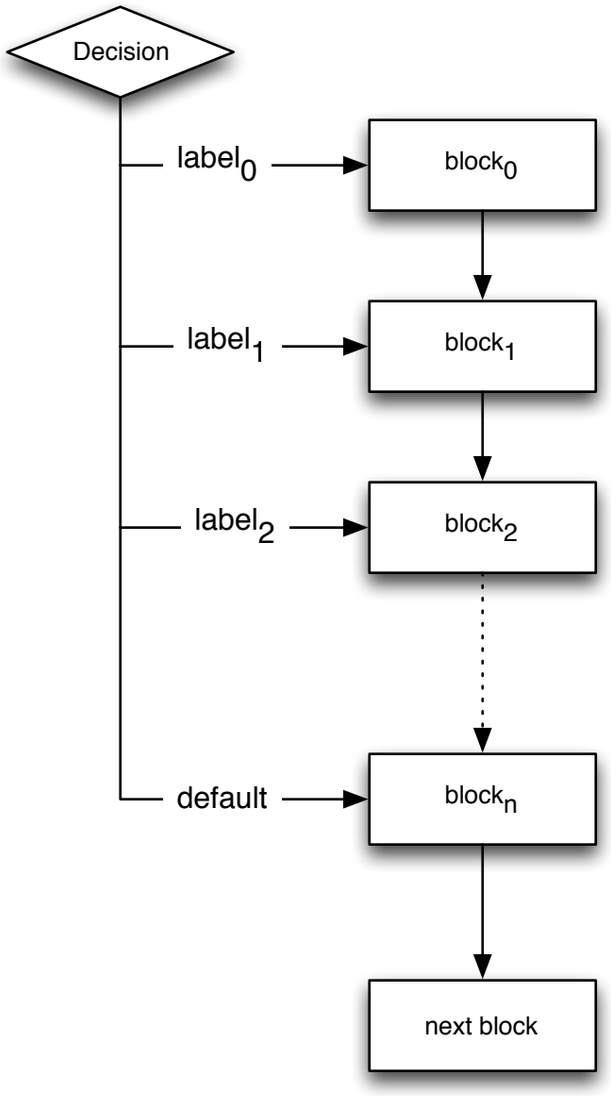
```
if (n==0) return 1 else return n * fact(n-1)
```

- Python version

```
if n==0:
   return 1
else:
   return n * fact(n-1)
```

Switching

- Again, same as Java….




```
switch(expression) {
   case label0: <block0>
   case label1: <block1>
   …
   default: <blockn>
}
```

Switch

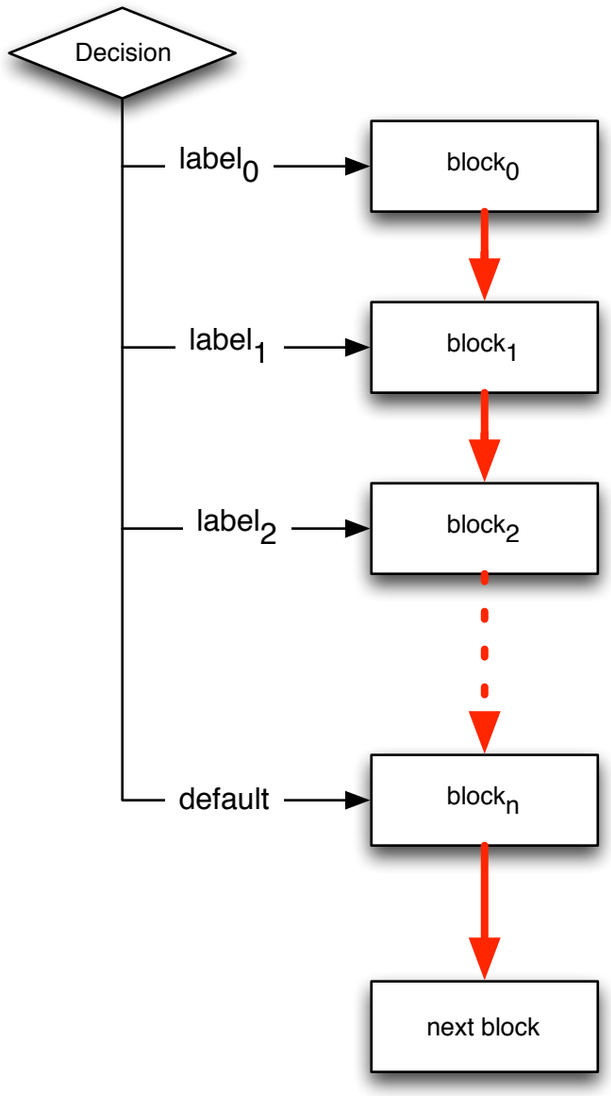
- Syntax
  - Called “selection” statement
  - Pseudo-syntax [detailed, real syntax in A.9]

```
switch (<expression>) {
   <labeledStmts>
}

<labeledStmts> :== <labeledStmt>
               :== <labeledStmt>  <labeledStmts>

<labeledStmt>  :==
   case  <label>  : [ <stmt> ; ]
   default        : <stmt>
```

- PS: There are no switches in Python

Switching

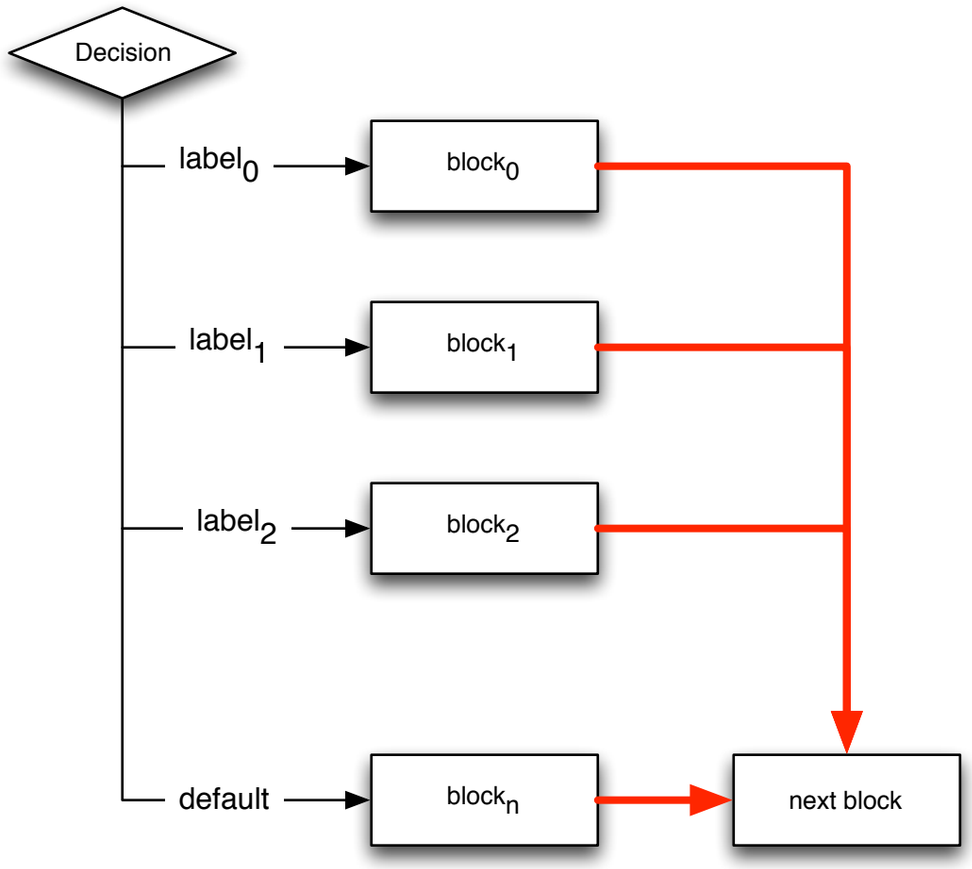
- Again, same as Java….




```
switch(expression) {
   case label0: <block0>
   case label1: <block1>
   …
   default: <blockn>
}
```

Switching

- This is *usually* what you mean




```
switch(expression) {
   case label0: <block0>;break;
   case label1: <block1>;break;
   …
   default: <blockn>
}
```

Don’t forget the breaks

Which brings us to…

- The dangerous ones
  - Two control primitives

`break`

Break out of the innermost control flow primitive. This could be
A **switch**
A **loop**

`continue`

Skip the remainder of this block and go to the next iteration of the **innermost loop** currently running

Break Statement

- Meant primarily to work with switch (again, like Java/C++)
  - Prevents to “fall-through” to the next case.

- Also works with loops
  - Within for-loops and while-loops
  - Jumps to the end of the loop without executing remaining statements in current iterations or even remaining iterations

Continue Statement

- Purpose similar to continue in Java
  - Skip the current iteration and go to the next one
  - Can be used within for / while / do-while loops
  - Can appear in a nested if / then / else
    - It always refers to the enclosing loop.
    - If there are nested loops, it only affect the ‘innermost’ loop

Switch Full Example (silly)

- Simple example

```
#include <stdio.h>

int main(int argc,char* argv[])
{
    char s[7] = "123456";
    int i = 0;
    int n = 0;
    while(s[i] != 0) {
        n = n * 10;
        switch(s[i]) {
            case '0': break;
            case '1': n += 1;break;
            case '2': n += 2;break;
            case '3': n += 3;break;
            case '4': n += 4;break;
            case '5': n += 5;break;
            case '6': n += 6;break;
            case '7': n += 7;break;
            case '8': n += 8;break;
            case '9': n += 9;break;
        }
        i += 1;
    }
    printf("converted: %d\n",n);
}
```

Switch/Break Example

- It works…. but this is an ugly *example* of switches…. It is far easier!

```
#include <stdio.h>

int main(int
{
    char s[7]
    int i = 0;
    int n = 0;
    while(s[i]
        n = n *
        switch(
            case
            case
            case
            case
```

```
#include <stdio.h>

int main(int argc,char* argv[]) {
    char s[7] = "123456";
    int i = 0;
    int n = 0;
    while(s[i] != 0) {
        n = n * 10;
        n = n + s[i] - '0';
        i += 1;
    }
    printf("converted: %d\n”,n);
    return 0;
}
```

```
': n += 4;break;
': n += 5;break;
': n += 6;break;
': n += 7;break;
': n += 8;break;
': n += 9;break;

rted: %d\n",n);
```

Caveat Emptor

- Continue and Break…..
  - Are troublesome fellows.
  - Very easy to misuse.
- Recommendation
  - Use break with switch
  - Avoid break within loops
  - Avoid continue altogether

The forbidden One

- Forget you ever knew they existed
- This is a terrible control primitive
- Don’t ever use it.
- Really.

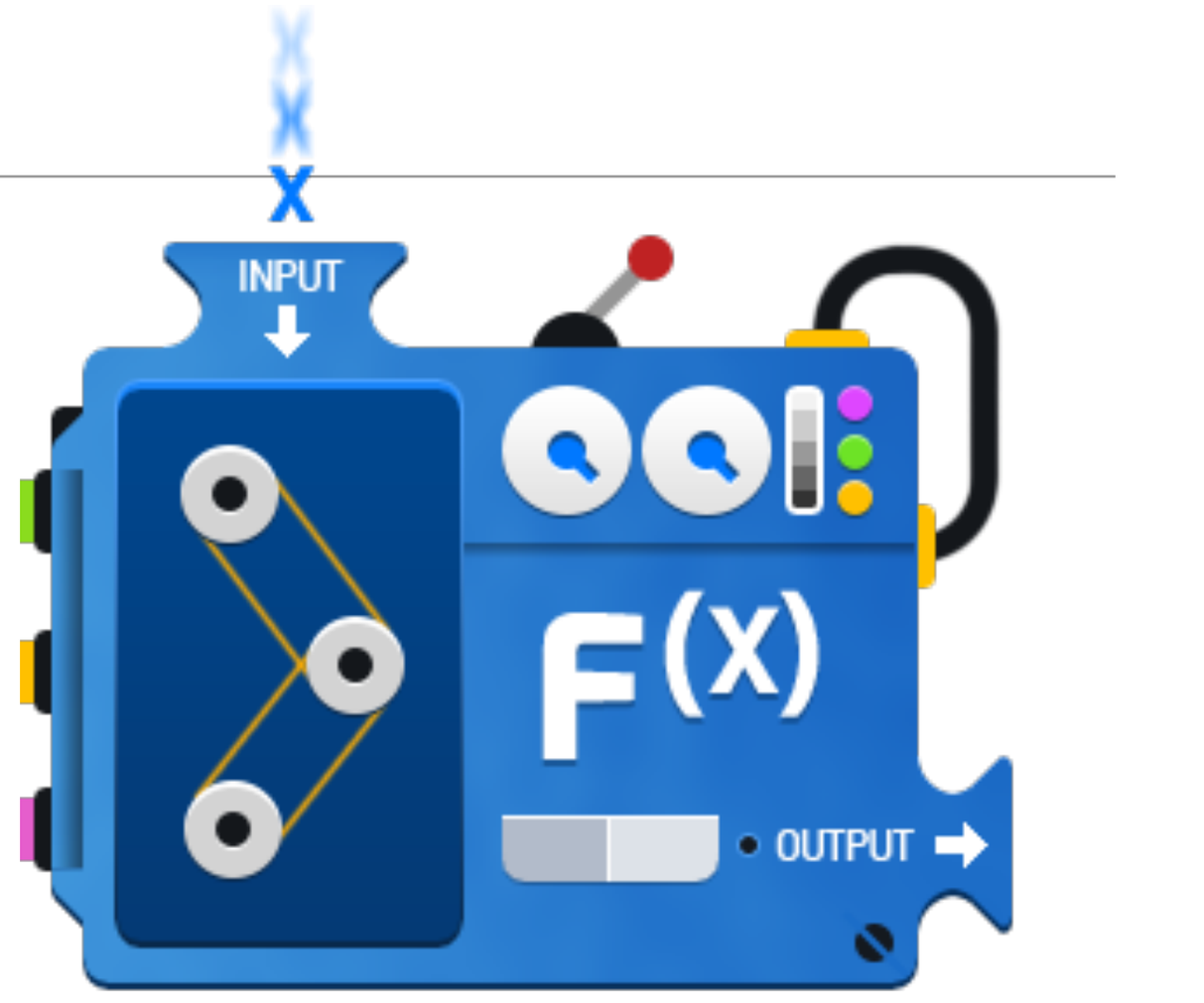
- Seriously.
- Nothing good will come out of it

```
   goto <label>;
…
label:
   <block>
```

Summary

- Common mistakes
  - Confuse assignment and test x=8 vs. x==8
  - Off by 1 errors in counting loops for(int i=0;i <=n ;i++) …
  - Confuse logical and bitwise ops x && y vs. x & y
  - Loop with empty bodies while (condition);
  - Forget the breaks in a switch
  - Dangling else in nested if-then-else

C Primer - Functions



Functions [K&R, Chapter 4]

- Functions are….

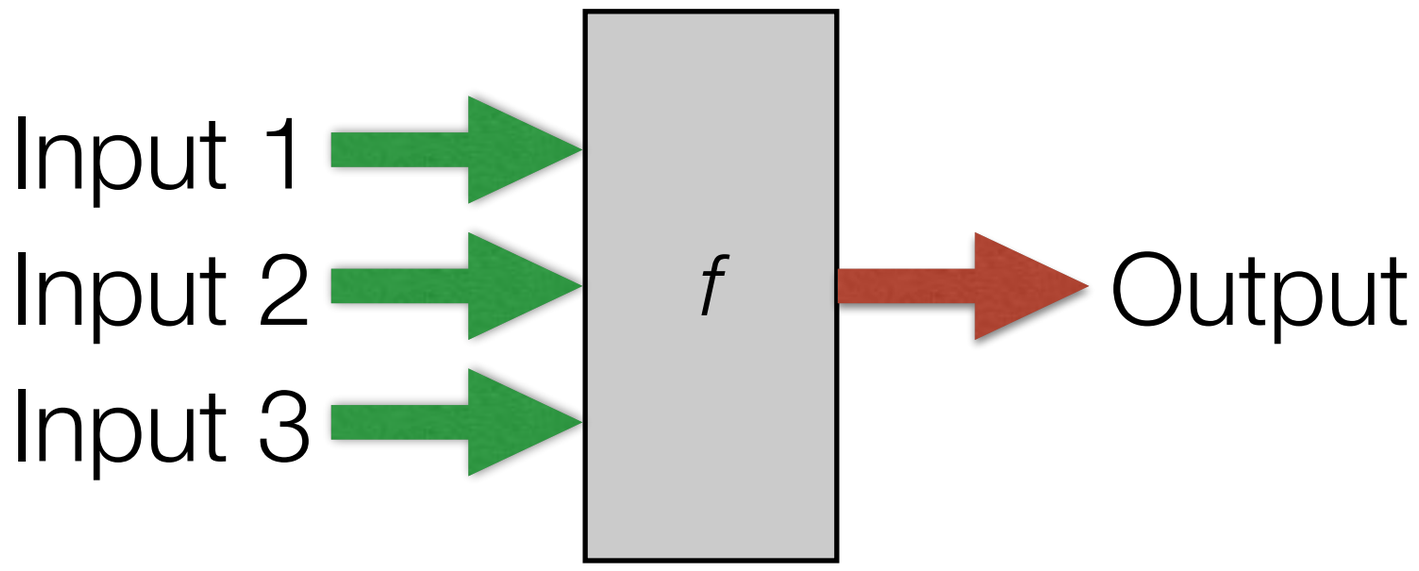
Fundamental Language tool

to build

procedural abstractions

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



Deviations from the Idealized View are **LETHAL**
(to the programmer)

- Good programs should not stray from the idealized view
- Any deviation from the gospel is dangerous
- Let me rephrase…

Back to Sanity…

- "*Nice*" functions **only** depends on their inputs
  - No side-effects
  - No globals / statics
  - Small and manageable
- Usually two steps
  - **Declare** the function "prototype"
  - **Define** the function "implementation"

Example

```c
#include <stdio.h>

#define LOWER 0
#define UPPER 300
#define STEP  10
int fahrToCelcius(int degF);
int main(int argc,char* argv[]) {
   for(int fahr=LOWER;fahr <= UPPER;fahr += STEP) {
      int celcius = fahrToCelcius(fahr);
      printf("From %d F to %d C degrees\n",fahr,celcius);
   }
   return 0;
}

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

Prototype

Implementation

Functions in General

- Simple syntactic form

```
<returnType> <name>(<parameters>) {
   <stmt>
}

<parameters>    ::=
                ::= <parameter> , <parameters>
                ::= …

<parameter>     ::= <type> <name>
```

Example

- Compute x^n

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

Very simple
- start with 1
- multiply by base
- decrease exponent
- stop when n == 0

Argument are passed by value

Improving the Example

- Compute x^n

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

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

Post-decrement….

Decrement n after testing

One statement left… no braces!

Only **local** copy of n is changed!

Improving the Example

- Compute x^n

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

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

Recursive version….

Induction on n

Base case returns 1

Inductive case multiplies by base

Bottom line: code is simple

What Brings Impurity in Functions…

- Side-effects
  - Usually via a so-called “static” or “global” variable.
- Bottom-line
  - The static changes the meaning of the function at each call!
  - You cannot understand the function locally
  - You must understand and hold **all** the code in your head to know what the function does.
- Always remember…

> Deviations from the Idealized View are **LETHAL**
> (so do not use static —ever—)

For Completeness Sake…

- Silly function contains a “static” declaration
  - hidden variable
    - It survives every invocation
    - Its state does not disappear from calls to calls
    - The function output no longer depends on its inputs alone!
    - Is hidden in the bowels of silly

```
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
…	…	…	…

For Completeness Sake…

- It could be worse still!
  - hidden variable
    - Declared outside of silly!
    - Its state does not disappear from calls to calls
    - 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;
}
```

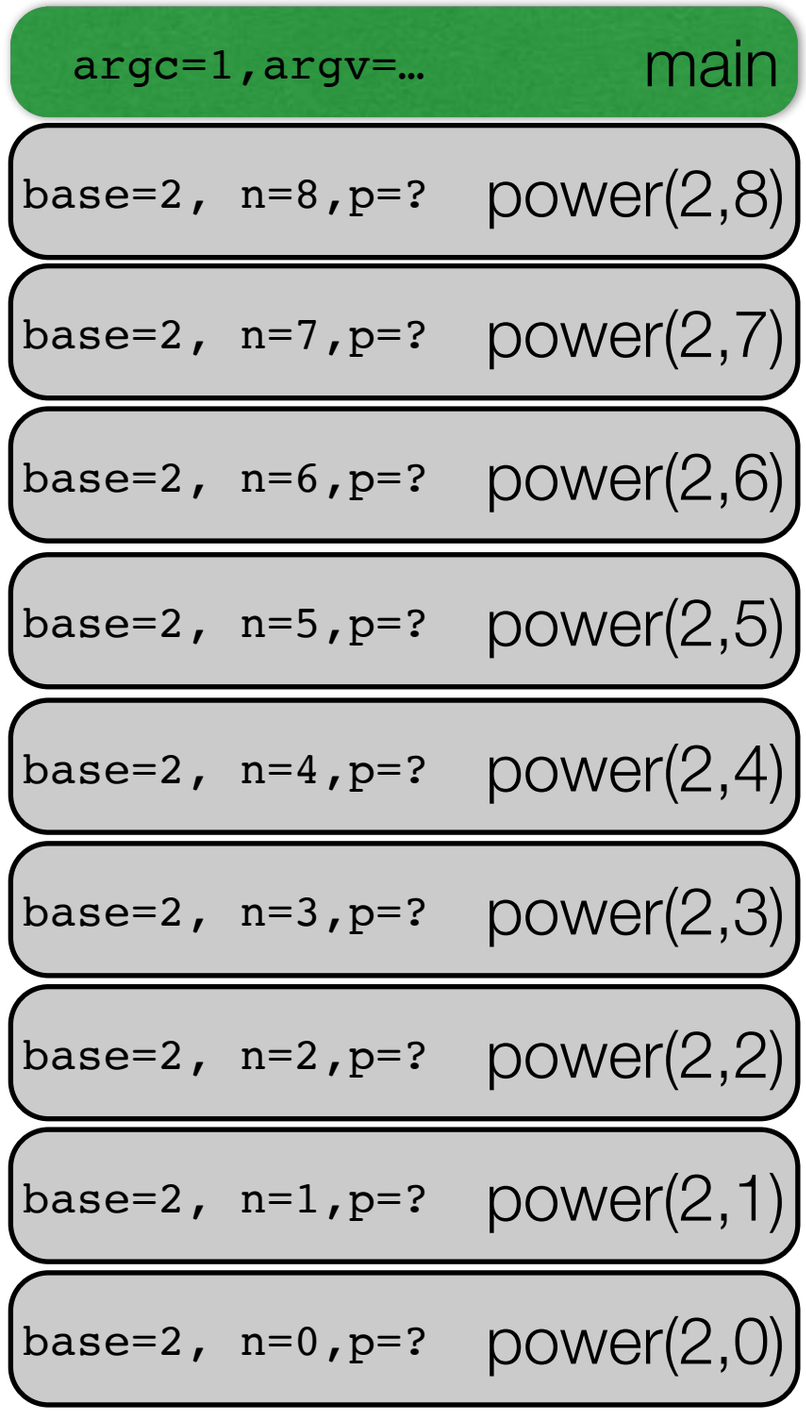
IN	OUT	hidden	hidden
1	2	0	1
1	3	1	2
1	4	2	3
2	7	3	4
…	…	…	…

Context

- Function “State”
  - Created when the function is called
  - Last for the duration of the call
  - Discarded when the function terminates
- Function “State” includes
  - Function arguments “Formals”
  - Function locals

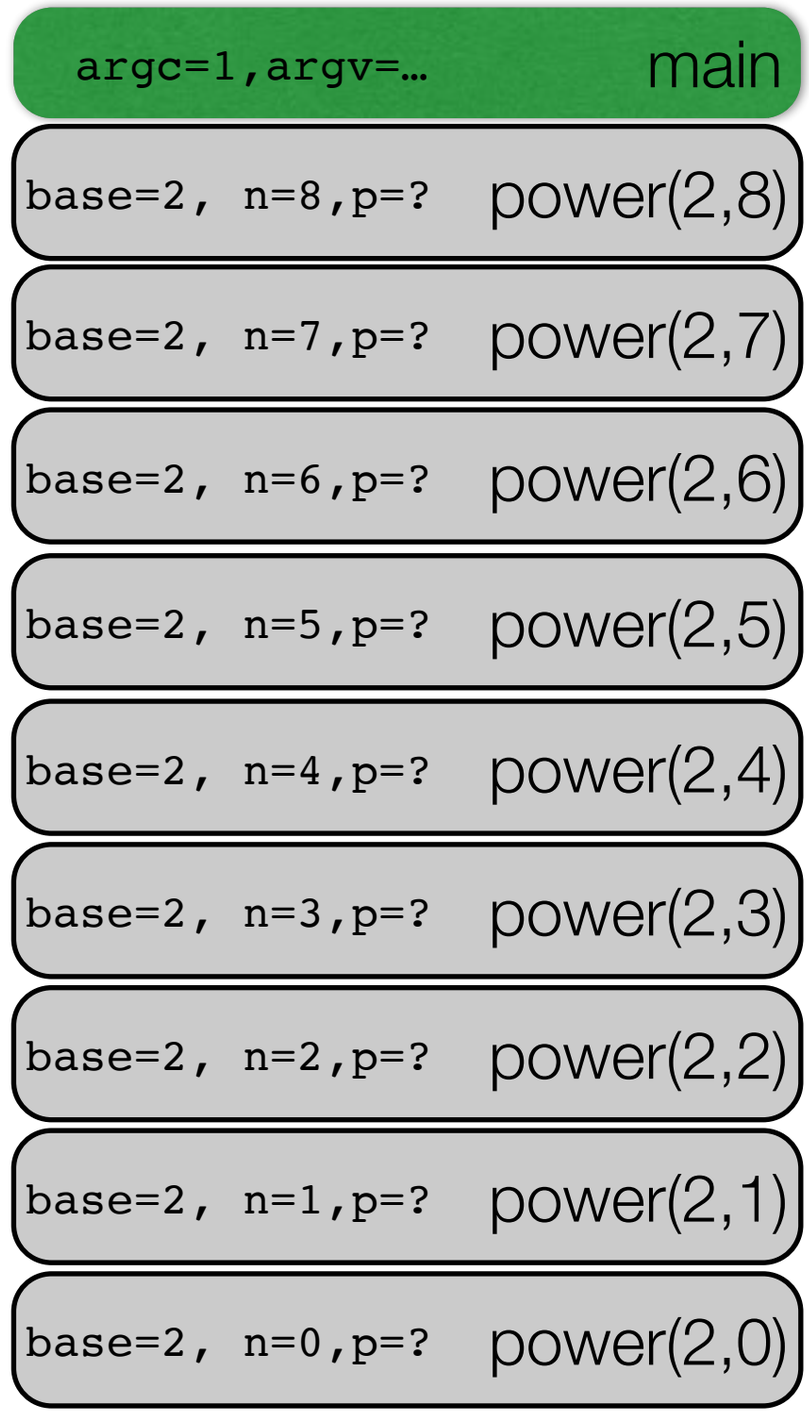
Context In Picture

```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;
}
```



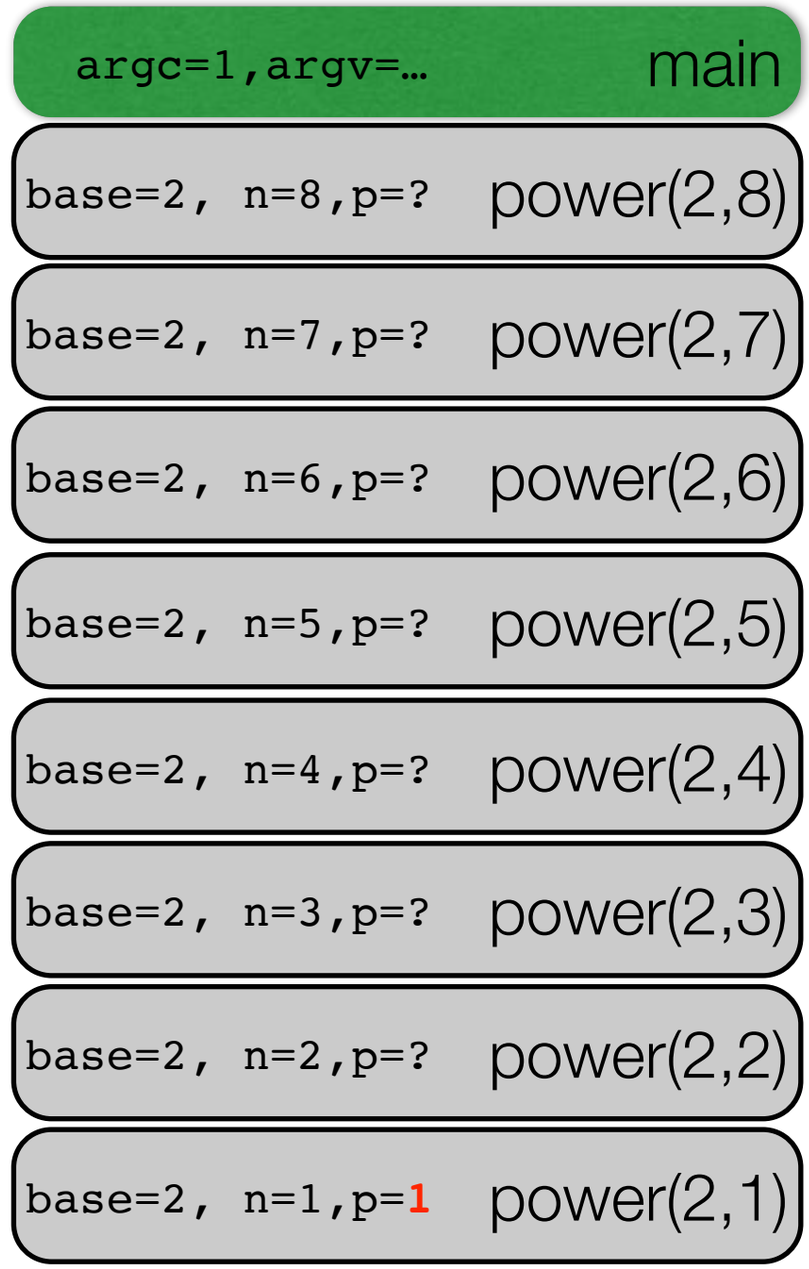
Context In Picture

```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;
}
```



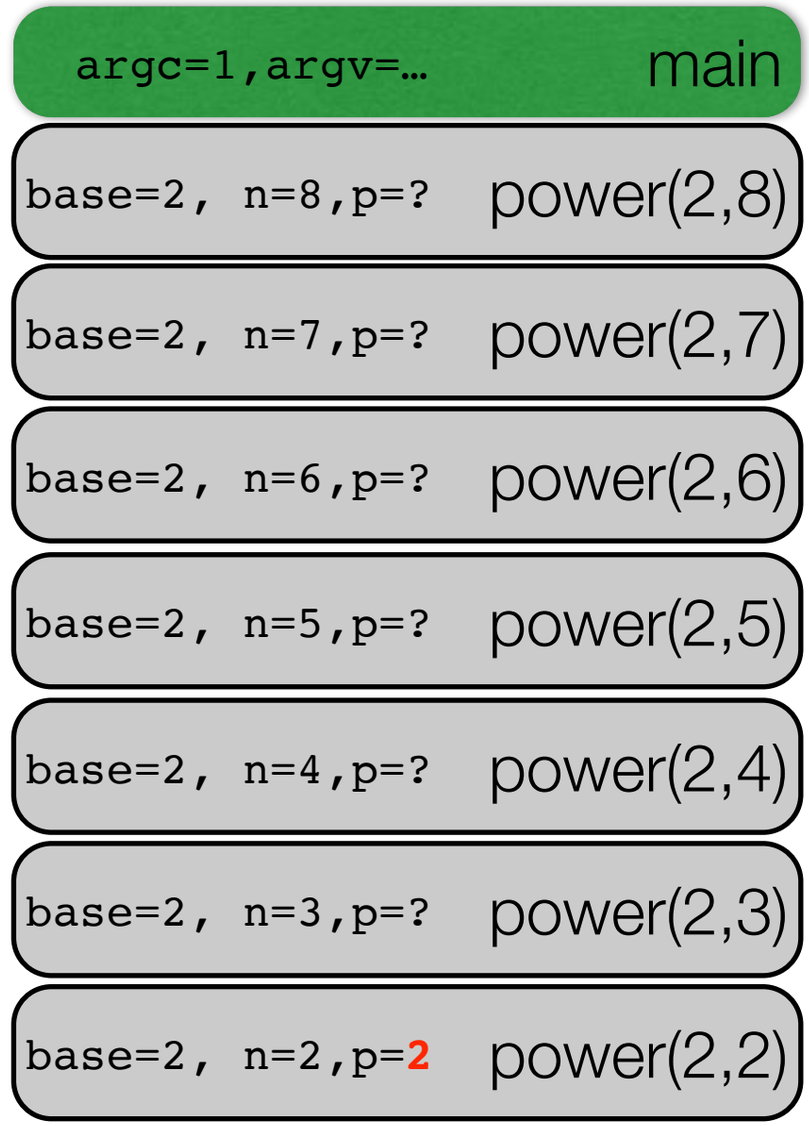
Context In Picture

```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;
}
```



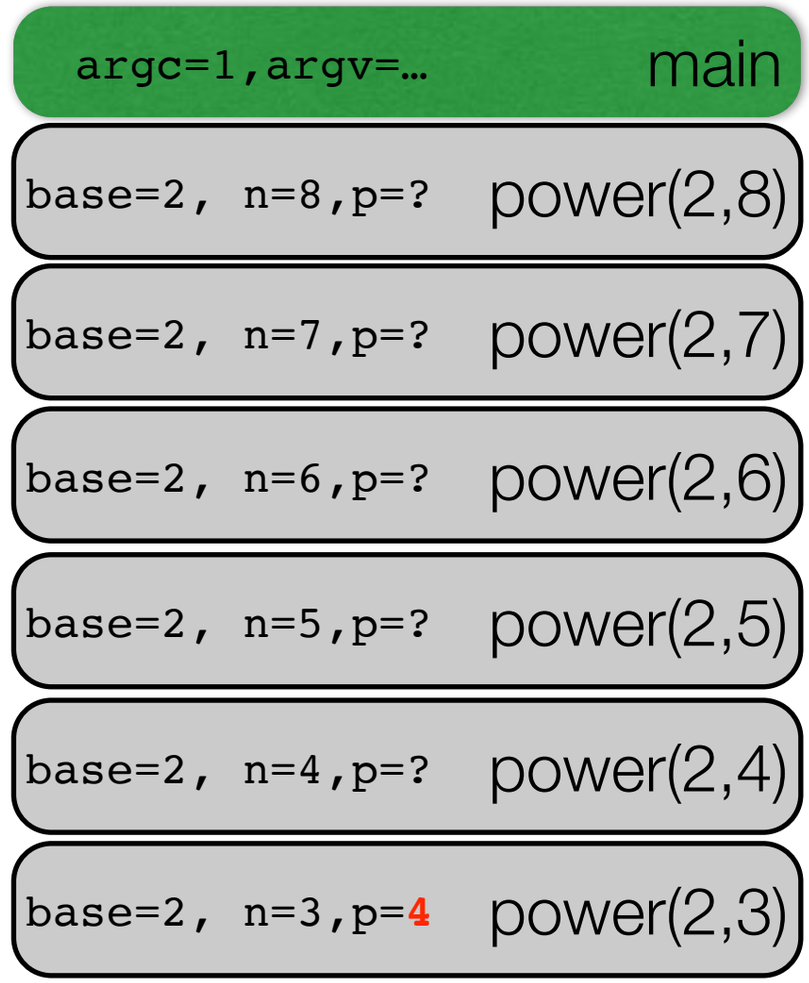
Context In Picture

```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;
}
```



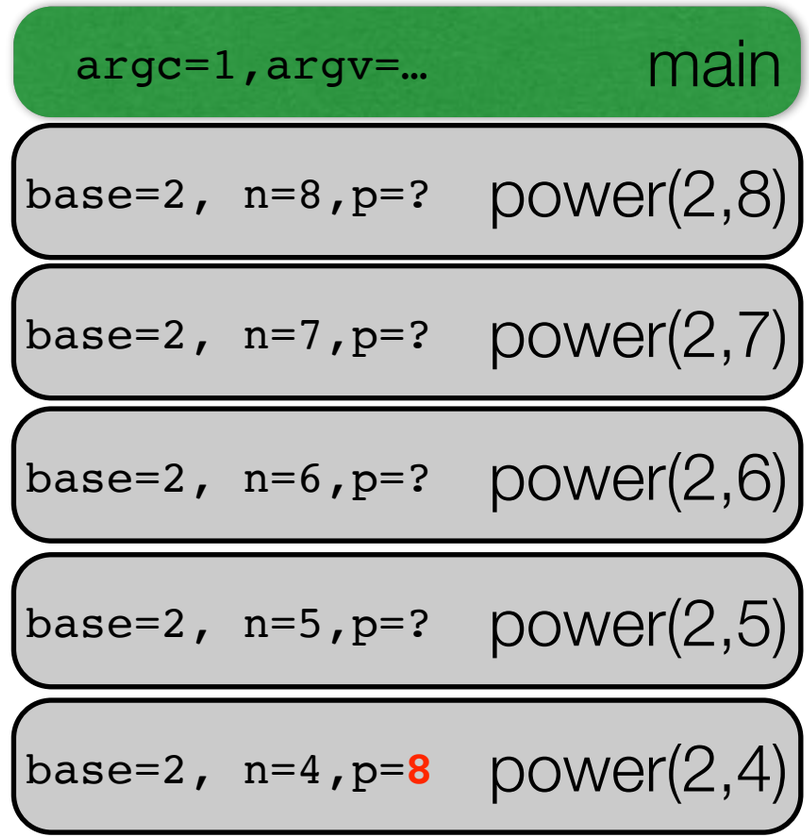
Context In Picture

```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;
}
```



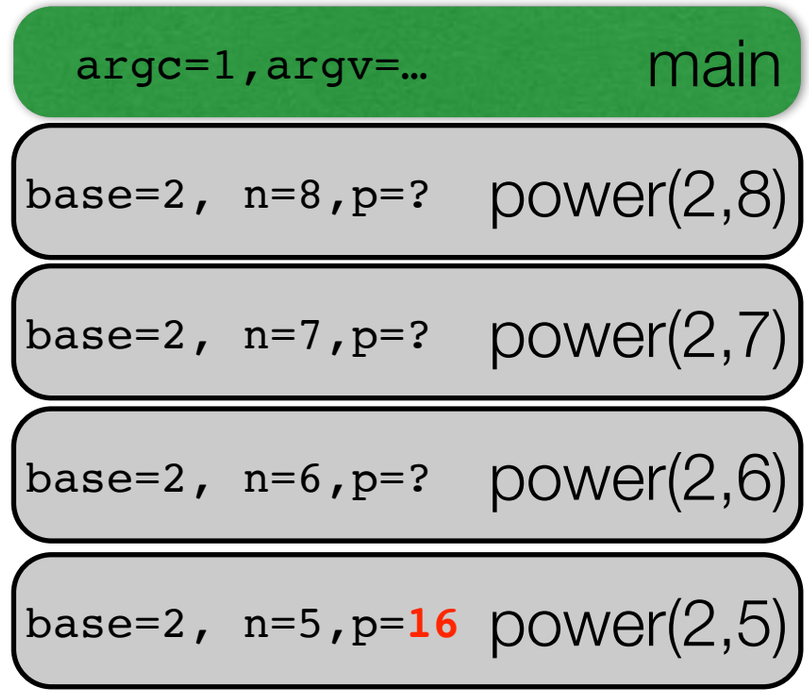
Context In Picture

```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;
}
```



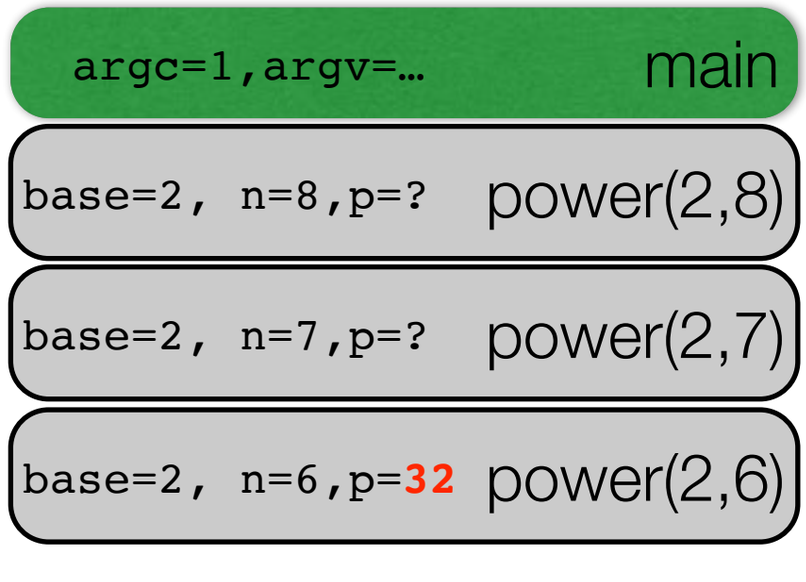
Context In Picture

```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;
}
```



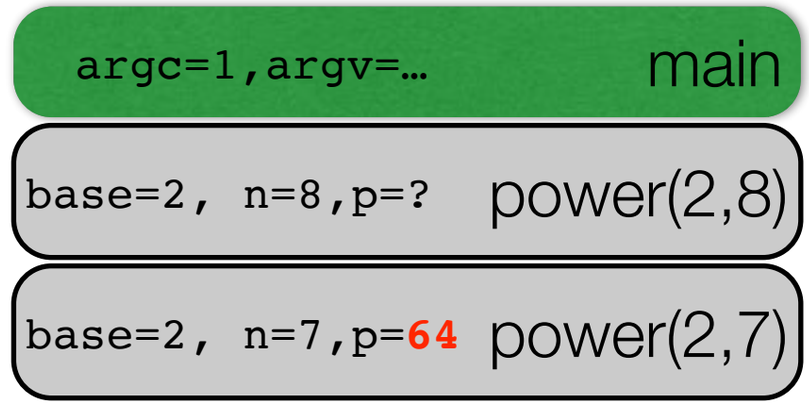
Context In Picture

```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;
}
```



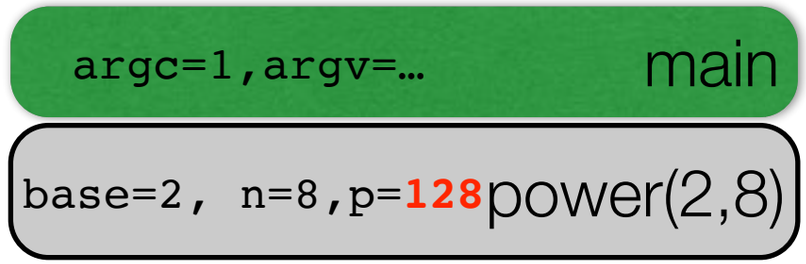
Context In Picture

```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;
}
```



Context In Picture

```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;
}
```



Context In Picture

```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=…

   returned
  Value=256
```

main

Output

```
2^8 = 256
```

Summary

- Context **Frames**
  - Created automatically when function called
  - Contain a “copy” of the function arguments [call by value!]
  - Contains a copy of the function local names
  - Discarded automatically when leaving function
  - NOTHING in the frame survives the call.
- The stack of frames is called: The Execution Stack

Memory management on the stack is **automatic!**

Macros for “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?

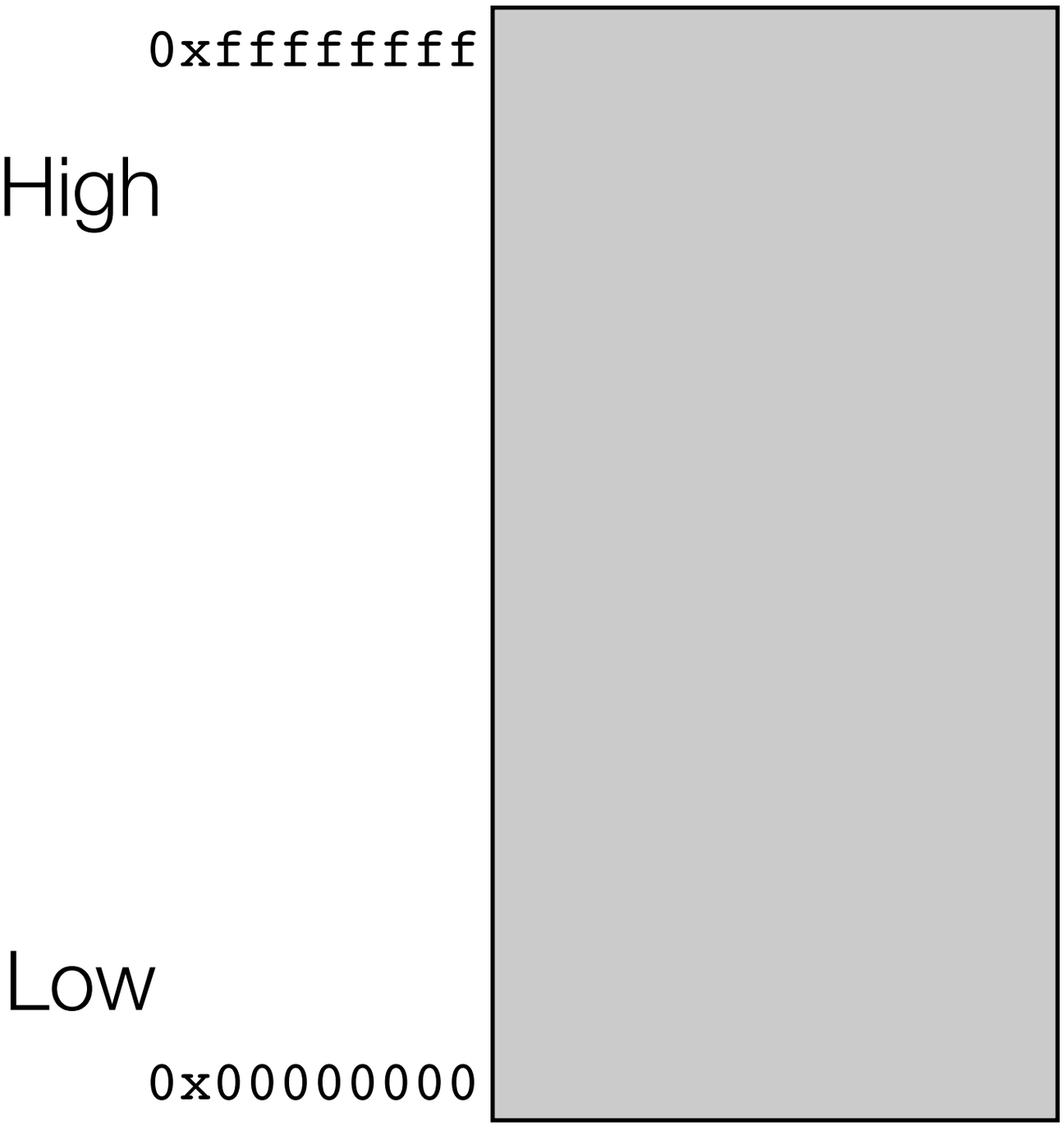
Example

```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
```

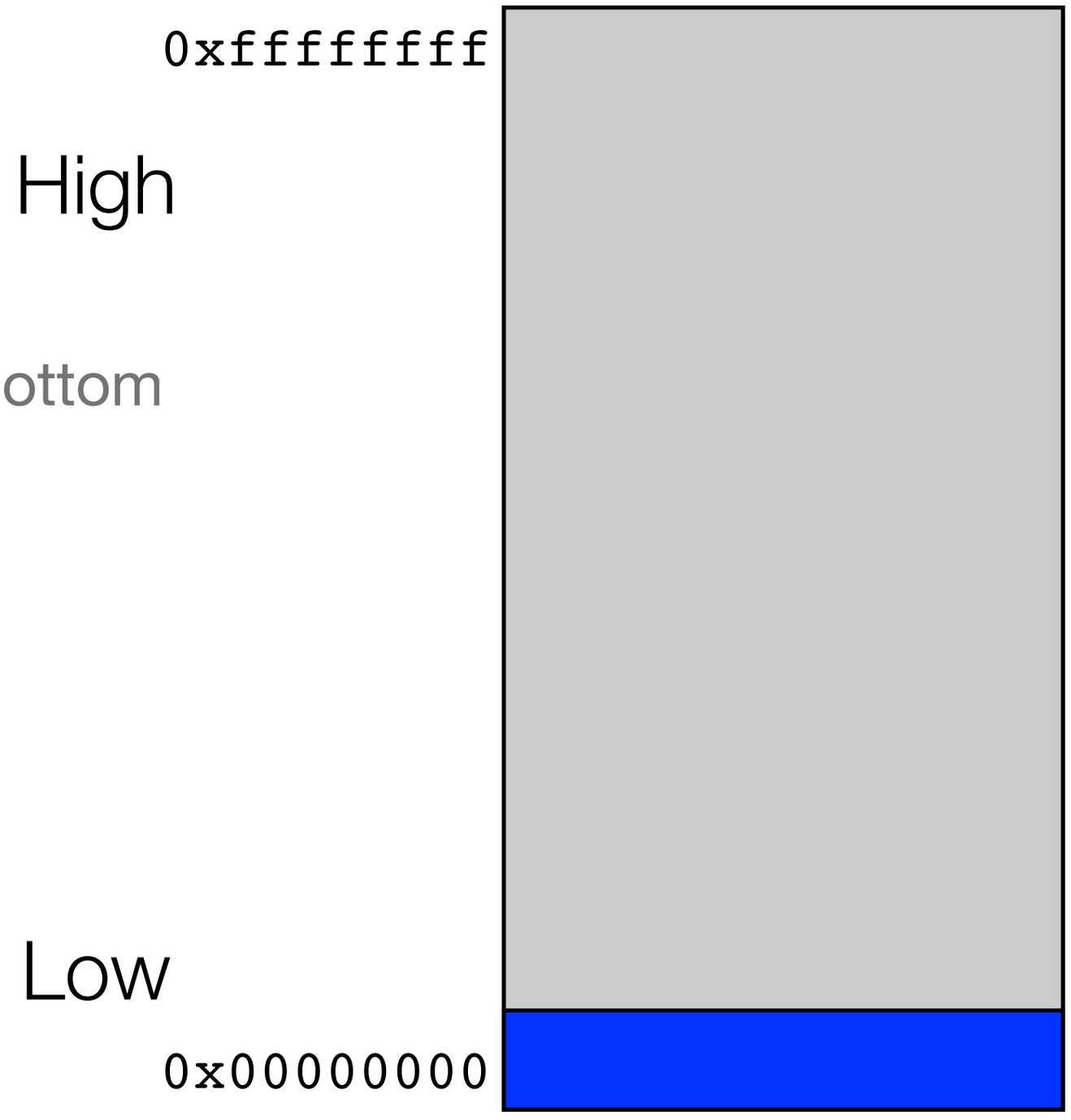
Memory Organization

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



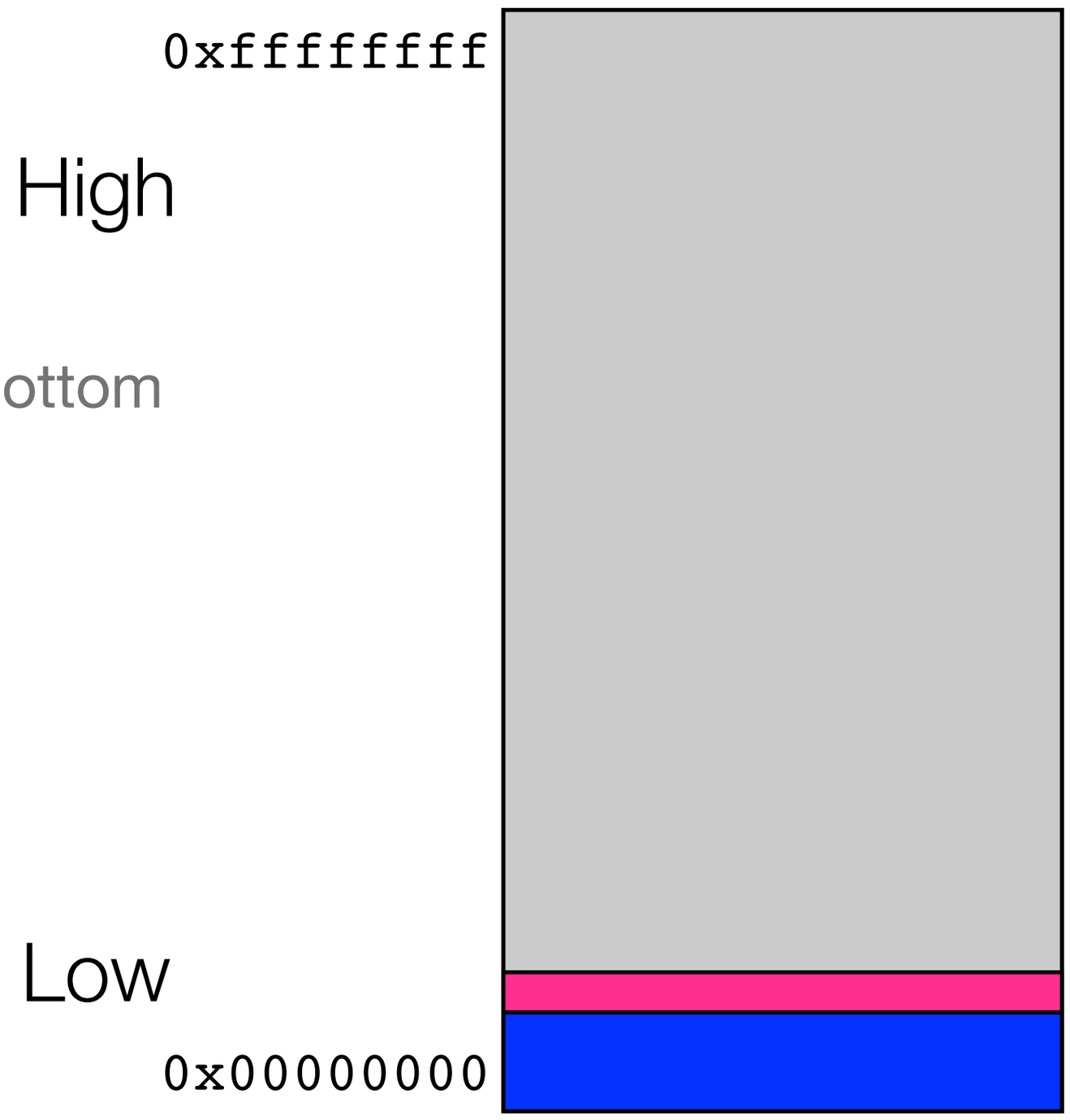
Memory Organization

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom



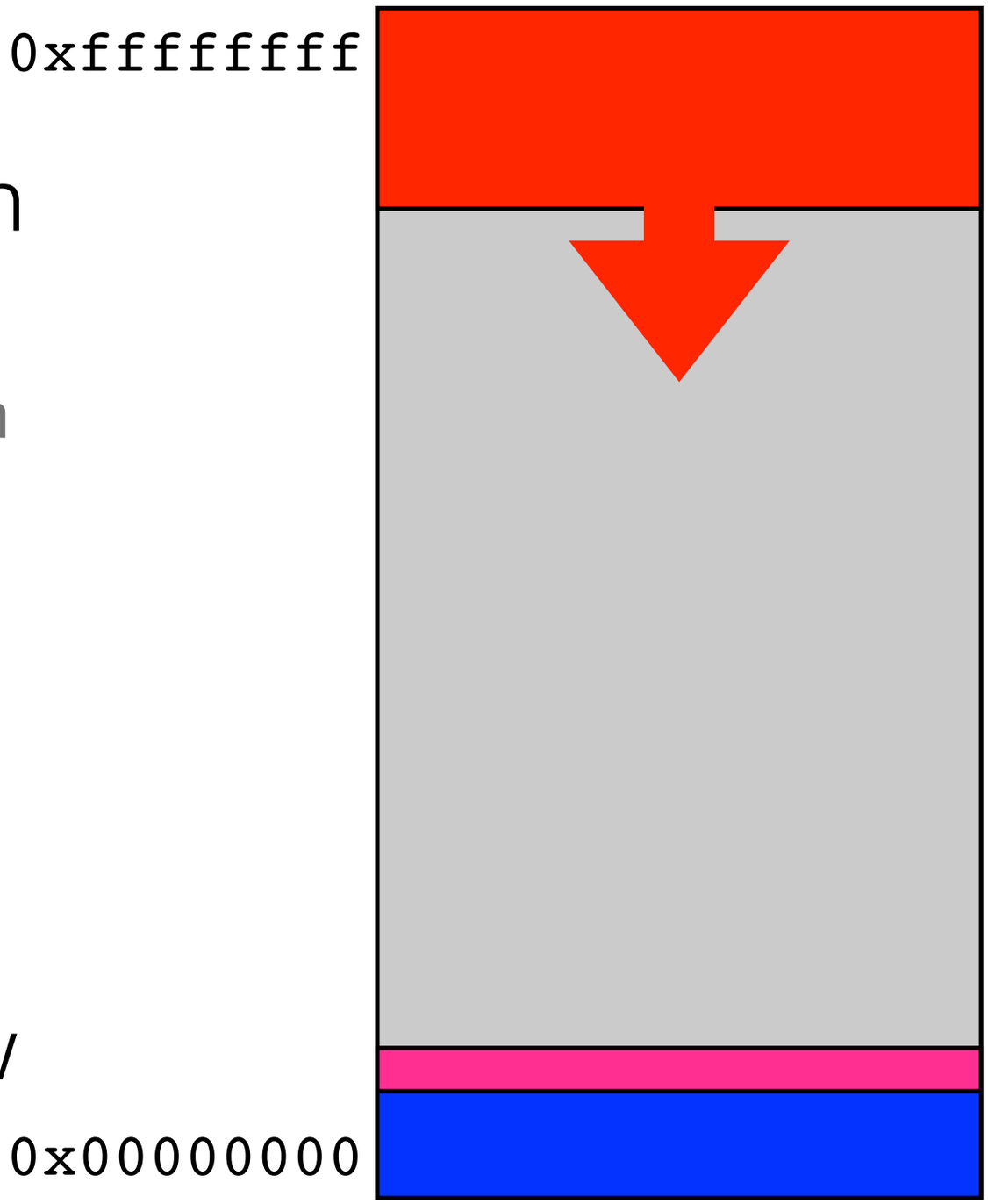
Memory Organization

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



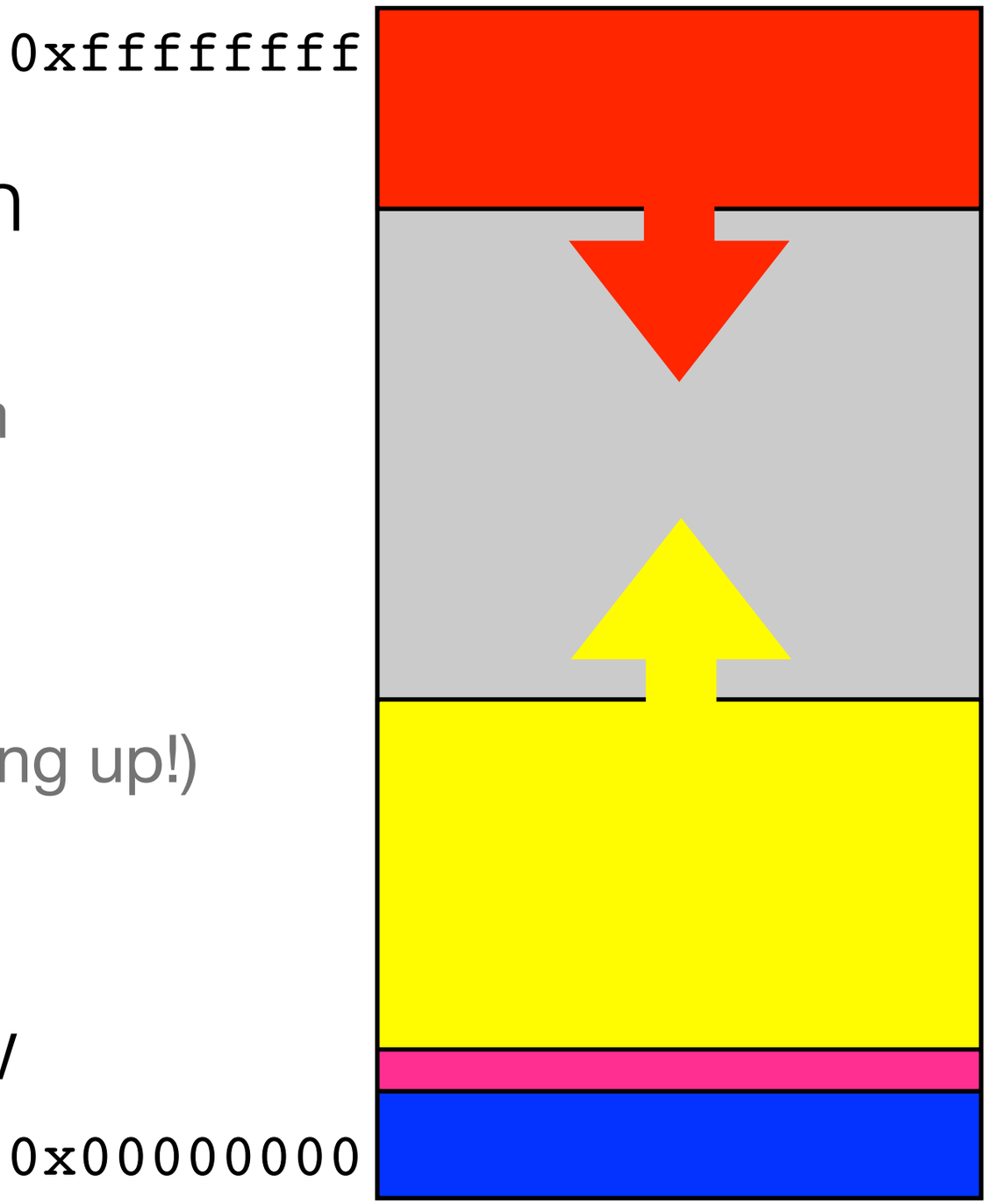
Memory Organization

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



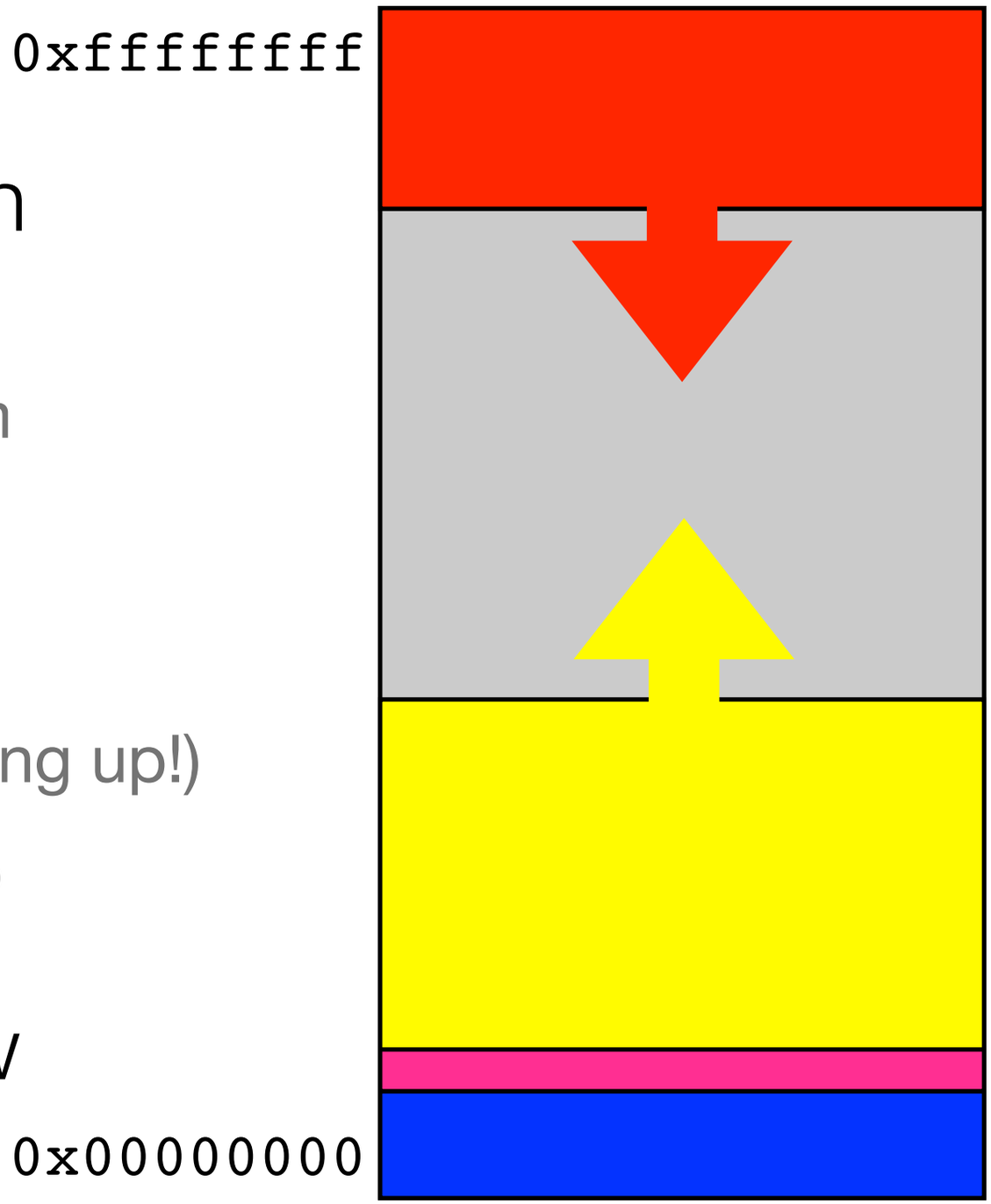
Memory Organization

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

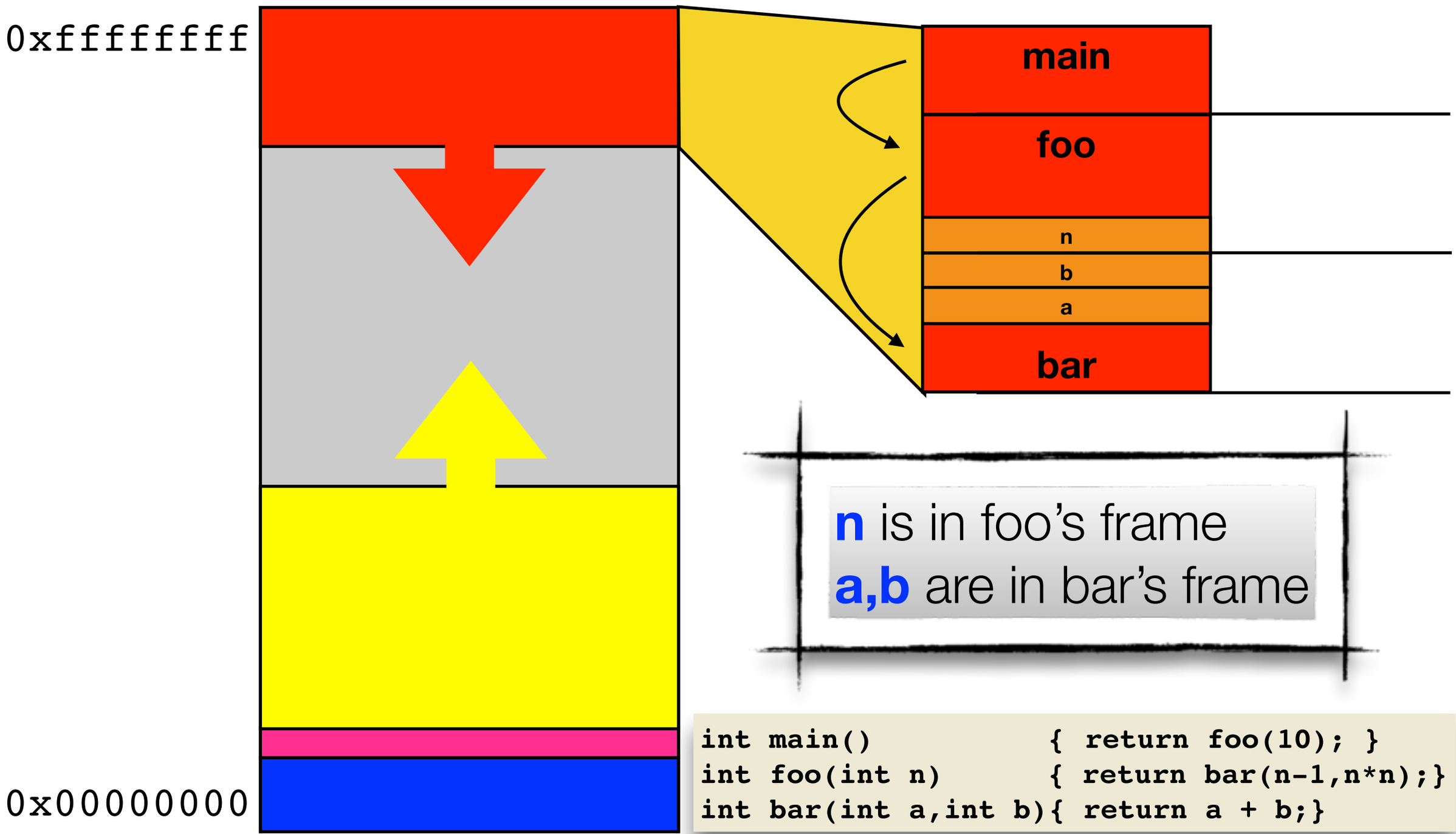


Memory Organization

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** 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



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;}
```

The Road Ahead…

- Compound Types
  - Structures
- Pointers
- Arrays
- Pointer arithmetic
- Memory layout and alignment
- Dynamic Memory management
- Basic I/O