Computer Architecture, Fall 2019

C: Introduction, Pointers

Jun 2019	Jun 2018	Change	Programming Language	Ratings	Change
1	**1**		**Java**	**15.004%**	**-0.36%**
2	2		C	13.300%	-1.64%
3	4	↑	Python	8.530%	+2.77%
4	3	↓	C++	7.384%	-0.95%
5	6	↑	Visual Basic .NET	4.624%	+0.86%
6	5	↓	C#	4.483%	+0.17%
7	8	↑	JavaScript	2.716%	+0.22%
8	7	↓	PHP	2.567%	-0.31%
9	9		SQL	2.224%	-0.12%
10	16	⇈	Assembly language	1.479%	+0.56%

Review

- Bits can be used to represent anything!
- *n* bits can represent up to 2^n things
- Number Representation
  - Bits can represent anything!
  - *n* bits can represent up to 2^n things
  - Unsigned, Bias, 1's, 2's
  - Overflow
  - Sign Extension

Question: Take the 4-bit number `x = 0b1010`.

Which of the following numbers does **x** NOT represent in the schemes discussed last lecture?

- unsigned
- sign and magnitude
- biased notation
- one's complement
- two's complement

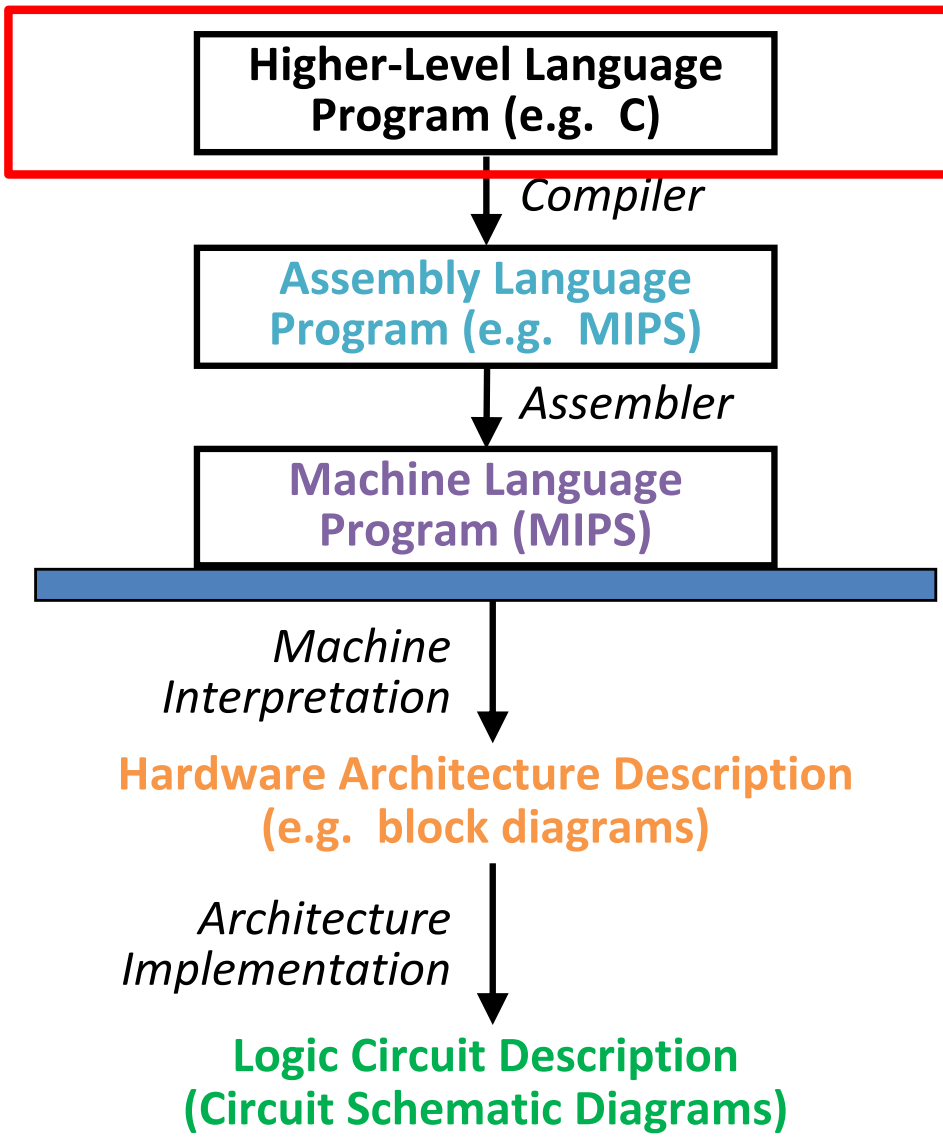
(A) -4

(B) -6 ← Two's Complement

(C) 10 ← Unsigned

(D) -2 ← Sign and Magnitude

Overview

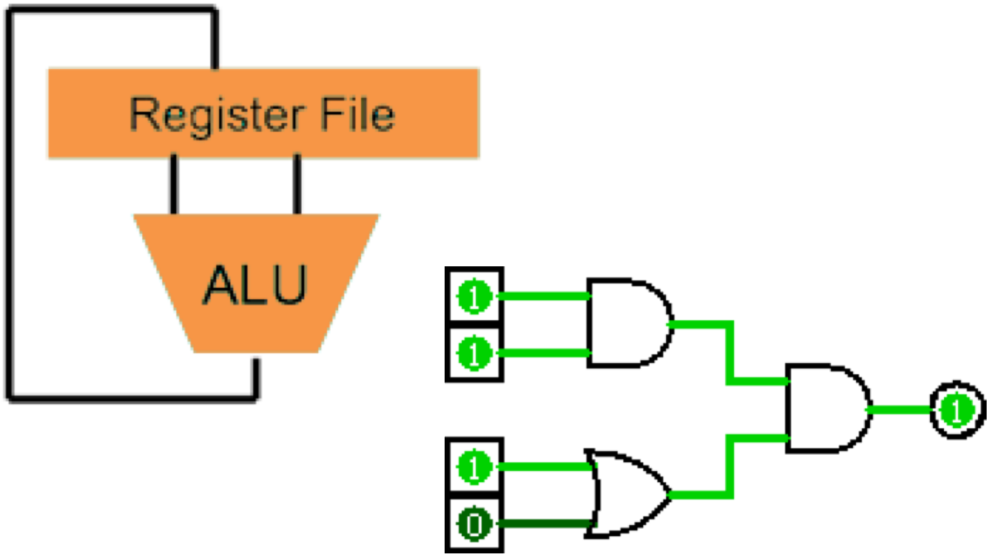



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

This week

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

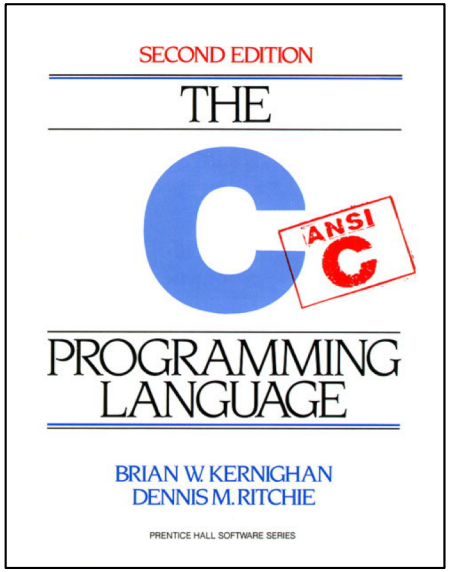


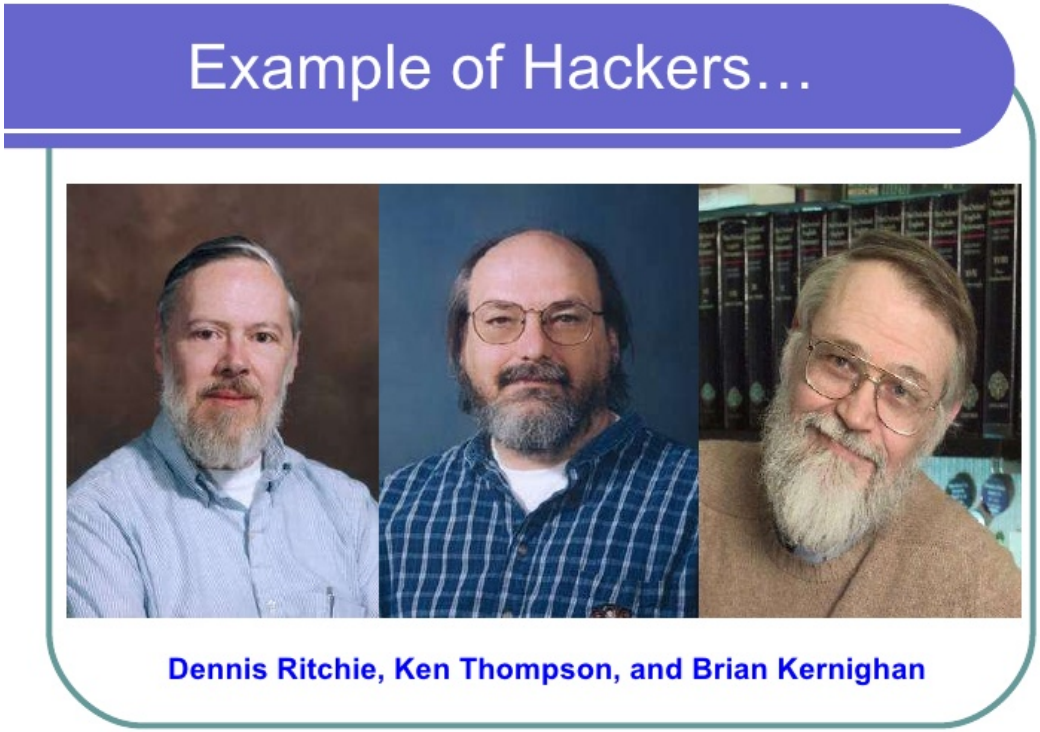
Agenda

- Basic C Concepts
  - Compilation
  - Variable Types
- C Syntax and Control Flow
- Pointers
  - Address vs. Value

Disclaimer

- You will not learn how to fully code in C in these lectures!
  - K&R is THE resource
  - MANY MANY online sources





Introduction

- *C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.*

  – Kernighan and Ritchie

- With C we can write programs that allow us to exploit underlying features of the architecture

C Concepts

These concepts distinguish C from other programming languages that you may know:	
Compiler	Creates useable programs from C source code
Typed variables	Must declare the kind of data the variable will contain
Typed functions	Must declare the kind of data returned from the function
Header files (`.h`)	Allows you to declare functions and variables in separate files
Structs	Groups of related values
Enums	Lists of predefined values
Pointers	Aliases to other variables

Compilation

- C is a compiled language
- C *compilers* map C programs into architecture-specific machine code (string of 0s and 1s)
  - Unlike Java, which converts to architecture-independent bytecode (run by Java Virtual Machine, JVM)
  - Unlike Python, which directly *interprets* the code
  - Main difference is when your program is mapped to low-level machine instructions

Compilation Advantages

- **Excellent run-time performance:** Generally much faster than Python or Java for comparable code because it optimizes for the given architecture

- **Fair compilation time:** enhancements in compilation procedure (`Makefile`s) allow us to recompile only the modified files

Compilation Disadvantages

- Compiled files, including the executable, are architecture-specific (CPU type and OS)
  - Executable must be rebuilt on each new system
  - i.e. “porting your code” to a new architecture

- “Edit→ Compile → Run [repeat]” iteration cycle can be slow

Typed Variables in C

declaration assignment

```
int   x = 2;
float y = 1.618;
char  z = 'A';
```

You must declare the type of data a variable will hold

Declaration must come before or simultaneously with assignment

Type	Description	Examples
`int`	signed integer	5,-12,0
`short`	`int` (`short`)	smaller signed integer
`long`	`int` (`long`)	larger signed integer
`char`	single text character or symbol	'a', 'D', '?'
`float`	floating point non-integer numbers	0.0, 1.618, -1.4
`double`	greater precision FP number	

- Integer sizes are machine dependent!
  - Common size is 4 or 8 bytes (32/64-bit), but can't ever assume this
- Can add "unsigned" before `int` or `char`

sizeof()

- If integer sizes are machine dependent, how do we tell?
- Use `sizeof()` function
  - Returns size in bytes of variable or data type name

    Examples: `int x; sizeof(x); sizeof(int);`
- Acts differently with arrays and structs, which we will cover later
  - Arrays: returns size of whole array
  - Structs: returns size of one instance of struct (sum of sizes of all struct variables + padding)

Characters

- Encode characters as numbers, same as everything!
- ASCII standard defines 128 different characters and their numeric encodings (http://www.asciitable.com)
  - `char` representing the character 'a' contains the value 97
  - `char c = 'a';` or `char c = 97;` are both valid
- A `char` takes up 1 byte of space
  - 7 bits is enough to store a char ($2^7 = 128$), but we add a bit to round up to 1 byte since computers usually deal with multiples of bytes

Typecasting in C (1/2)

- C is a "weakly" typed language
  - You can explicitly typecast from any type to any other:

```
int i = -1;
if(i < 0)
  printf("This will print\n");
if((unsigned int)i < 0)
  printf("This will not print\n");
```

- This is possible because everything is stored as bits!
  - Can be seen as changing the "programmer's perspective" of the variable

Typecasting in C (2/2)

- C is a "weakly" typed language
  - You can explicitly typecast from any type to any other:

```
int i = -1;
if(i < 0)
  printf("This will print\n");
if((unsigned int)i < 0)
  printf("This will not print\n");
```

- Can typecast *anything*, even if it doesn't make sense:

```
struct node n;   /* structs in a few slides */
int i = (int) n;
```

  - More freedom, but easier to shoot yourself in the foot

Typed Functions in C

```
// function prototypes
int my_func(int,int);
void sayHello();

// function definitions
int my_func(int x,int y)
{
   sayHello();
  return x*y;
}
void sayHello()
{
  printf("Hello\n");
}
```

- You have to declare the type of data you plan to return from a function
- Return type can be any C variable type or `void` for no return value
  - Place on the left of function name
- Also necessary to define types for function arguments
- Declaring the "prototype" of a function allows you to use it before the function's definition

Structs in C

- Way of defining compound data types
- A structured group of variables, possibly including other structs

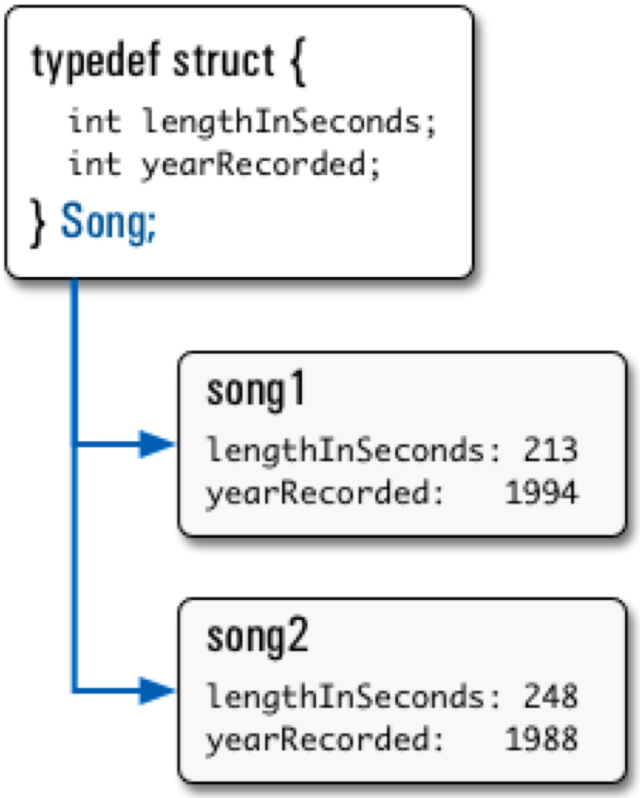
```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

song1.lengthInSeconds =  213;
song1.yearRecorded    = 1994;

Song song2;

song2.lengthInSeconds =  248;
song2.yearRecorded    = 1988;
```



	C	Java
Type of Language	Function Oriented	Object Oriented
Program-ming Unit	Function	Class = Abstract Data Type
Compilation	Creates machine-dependent code	Creates machine-independent bytecode
Execution	*Loads* and *executes* program	JVM *interprets* bytecode

Hello World	#include<stdio.h>
	int main(void) {
	  printf("Hello\n");
	  return 0;
	}
		public class HelloWorld {
		  public static void main(String[] args) {
		  System.out.printl("Hello");
		  }
		}

Memory manage-ment	Manual (`malloc`, `free`)	Automatic (garbage collection)

From http://www.cs.princeton.edu/introcs/faq/c2java.html

Agenda

- Basic C Concepts
  - Compilation
  - Variable Types
- C Syntax and Control Flow
- Pointers
  - Address vs. Value

C and Java operators nearly identical

For precedence/order of execution, see Table 2-1 on p. 53 of K&R

- arithmetic: `+`, `-`, `*`, `/`, `%`
- assignment: `=`
- augmented assignment: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- bitwise logic: `~`, `&`, `|`, `^`
- bitwise shifts: `<<`, `>>`
- boolean logic: `!`, `&&`, `||`
- equality testing: `==`, `!=`
- subexpression grouping: `(` `)`
- order relations: `<`, `<=`, `>`, `>=`
- increment and decrement: `++` and `--`
- member selection: `.`, `->`
- conditional evaluation: `?` `:`

Generic C Program Layout

Handled by Preprocessor

```
#include <system_files>
#include "local_files"

#define macro_name macro_expr
```

Dumps other files here (`.h` and `.o`)

← Macro substitutions

```
/* declare functions */
/* declare external variables and structs */
```

Remember rules of scope! (internal vs. external)

```
int main(int argc, char *argv[]) {
    /* the innards */
}

/* define other functions */
```

Programs start at `main()`

`main()` must return `int`

Sample C Code

```
#include <stdio.h>
#define REPEAT 5

int main(int argc, char *argv[]) {
    int i;
    int n = 5;
    for (i = 0; i < REPEAT; i = i + 1) {
        printf("hello, world\n");
    }
    return 0;
}
```

C Syntax: `main`

- To get arguments to the main function, use:
  - `int main(int argc, char *argv[])`
- What does this mean?
  - `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument).
  - `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

`main` Example

- Unix (or Linux) shell prompt (command line)

```
$ foo hello 87
```

- Here `argc = 3` and the array `argv` contains pointers to the following strings:

```
argv[0] = "foo"
argv[1] = "hello"
argv[2] = "87"
```

- We will cover pointers and strings later

C Syntax: Variable Declarations

- *All* variable declarations must appear before they are used (e.g. at the beginning of a block of code)
- A variable may be initialized in its declaration; if not, it holds garbage!
- Variables of the same type may be declared on the same line
- Examples of declarations:
  - Correct:
    ```
    int x;
    int a, b=10, c;
    ```
  - Incorrect:
    ```
    for(int i=0; i<10; i++);
    short x=1, float y=1.0;
    ```

C Syntax: True or False

- No explicit Boolean type in C (unlike Java)
- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of *pointer*: more on this later)
- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Same idea as in Scheme: only #if is false, anything else is true!

C Syntax: Control Flow

- Should be similar to what you've seen before
  - `if-else`
    - `if (expression) statement`
    - `if (expression) statement1 else statement2`
  - `while`
    - `while (expression) statement`
    - `do statement while (expression);`

C Syntax: Control Flow

- Should be similar to what you've seen before
  - `for`
    - `for (initialize; check; update) statement`
  - `switch`
    - `switch (expression) {`

```
        case const1:    statements
        case const2:    statements
        default:        statements
    }
```

    - `break`

`switch` and `break`

- Case statement (`switch`) requires proper placement of `break` to work properly
  - "Fall through" effect: will execute all cases until a `break` is found

```
switch(ch){
  case '+': … /* does + and - */
  case '-': … break;
  case '*': … break;
  default: …
}
```

  - In certain cases, can take advantage of this!

Has there been an update to ANSI C?

- Yes! It's called the "C99" or "C9x" std
  - Use option "`gcc -std=c99`" at compilation
- References

  `http://en.wikipedia.org/wiki/C99`

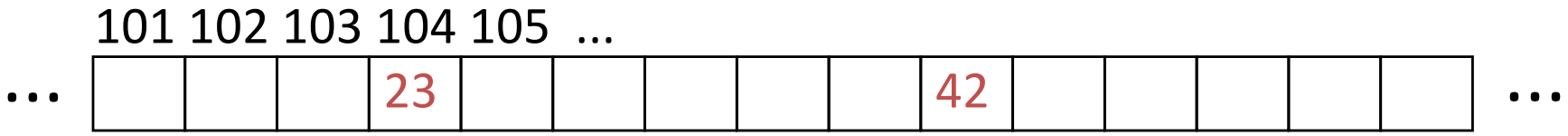
  `http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html`
- **Highlights:**
  - Declarations in `for` loops, like Java (#15)
  - Java-like `//` comments (to end of line) (#10)
  - Variable-length non-global arrays (#33)
  - `<inttypes.h>` for explicit integer types (#38)
  - `<stdbool.h>` for boolean logic definitions (#35)

Agenda

- Basic C Concepts
  - Compilation
  - Variable Types
- C Syntax and Control Flow
- Pointers
  - Address vs. Value

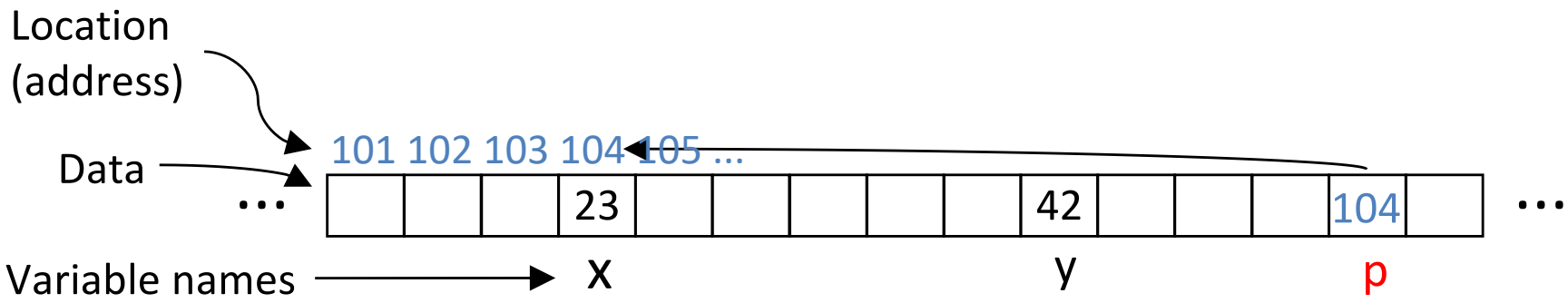
Address vs. Value

- Consider memory to be a single huge array
  - Each cell/entry of the array has an address
  - Each cell also stores some value
- Don't confuse the address referring to a memory location with the value stored there



Pointers

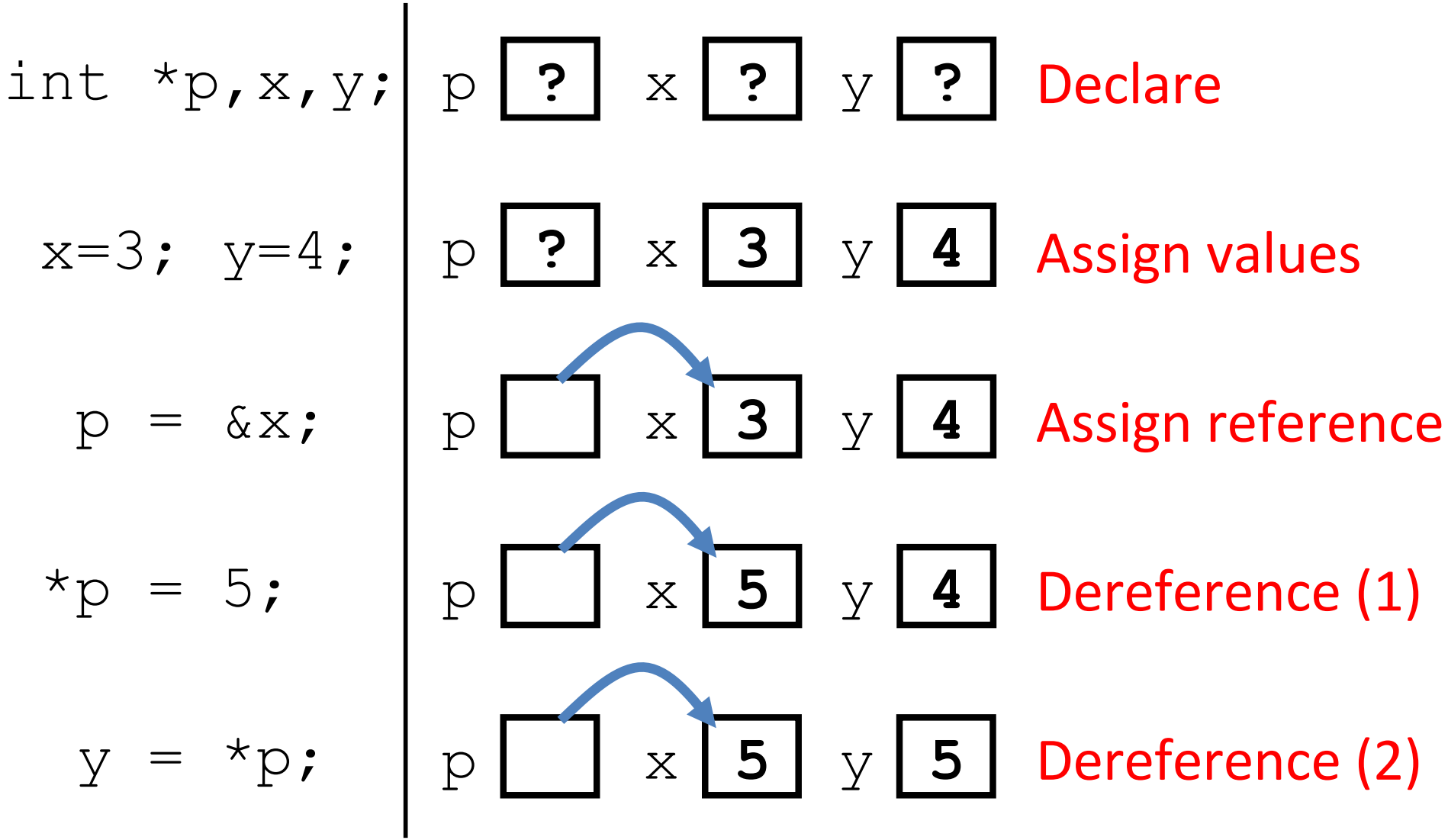
- A *pointer* is a variable that contains an address
  - An address refers to a particular memory location, usually also associated with a variable name
  - Name comes from the fact that you can say that it *points* to a memory location



Pointer Syntax

- `int *x;`
  - Declare variable `x` the address of an `int`
- `x = &y;`
  - Assigns address of `y` to `x`
  - `&` called the “address operator” in this context
- `z = *x;`
  - Assigns the value at address in `x` to `z`
  - `*` called the “dereference operator” in this context

Pointer Example



Pointer Types (1/2)

- Pointers are used to point to one kind of data (`int`, `char`, a `struct`, etc.)
  - Pointers to pointers? Oh yes! (e.g. `int **pp`)

- Exception is the type `void *`, which can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs and other bad things!

Pointer Types (2/2)

- Functions can return pointers

```
char *foo(char data) {
   return &data;
}
```

- Placement of `*` does not matter to compiler, but might to you
  - `int* x` is the same as `int *x`
  - `int *x,y,z;` is the same as `int* x,y,z;` but NOT the same as `int *x,*y,*z;`

Pointers and Parameter Passing

- Java and C pass parameters “by value”
  - Procedure/function/method gets a copy of the parameter, *so changing the copy does not change the original*

Function:

```
void addOne (int  x) {
        x =  x + 1;
}
```

Code:

```
int y = 3;
addOne(y);
```

← y remains equal to 3

Pointers and Parameter Passing

- How do we get a function to change a value?
  - Pass "by reference": function accepts a pointer and then modifies value by dereferencing it

Function:

```
void addOne (int *p) {
        *p = *p + 1;
}
```

Code:

```
int y = 3;
addOne(&y);
```

⟵ y is now equal to 4

Pointers in C

- Why use pointers?
  - When passing a large struct or array, it's easier/faster to pass a pointer than a copy of the whole thing
  - In general, pointers allow cleaner, more compact code

- Careful: Pointers are likely the single largest source of bugs in C
  - Most problematic with dynamic memory management, which we will cover later
  - *Dangling references* and *memory leaks*

Pointer Bugs

- Local variables in C are not initialized, they may contain anything (a.k.a. “garbage”)
- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!

```
void f() {
    int *p,x;
    x = *p;
}
```

BAD

```
void f2() {
    int *ptr;
    *ptr = 5;
}
```

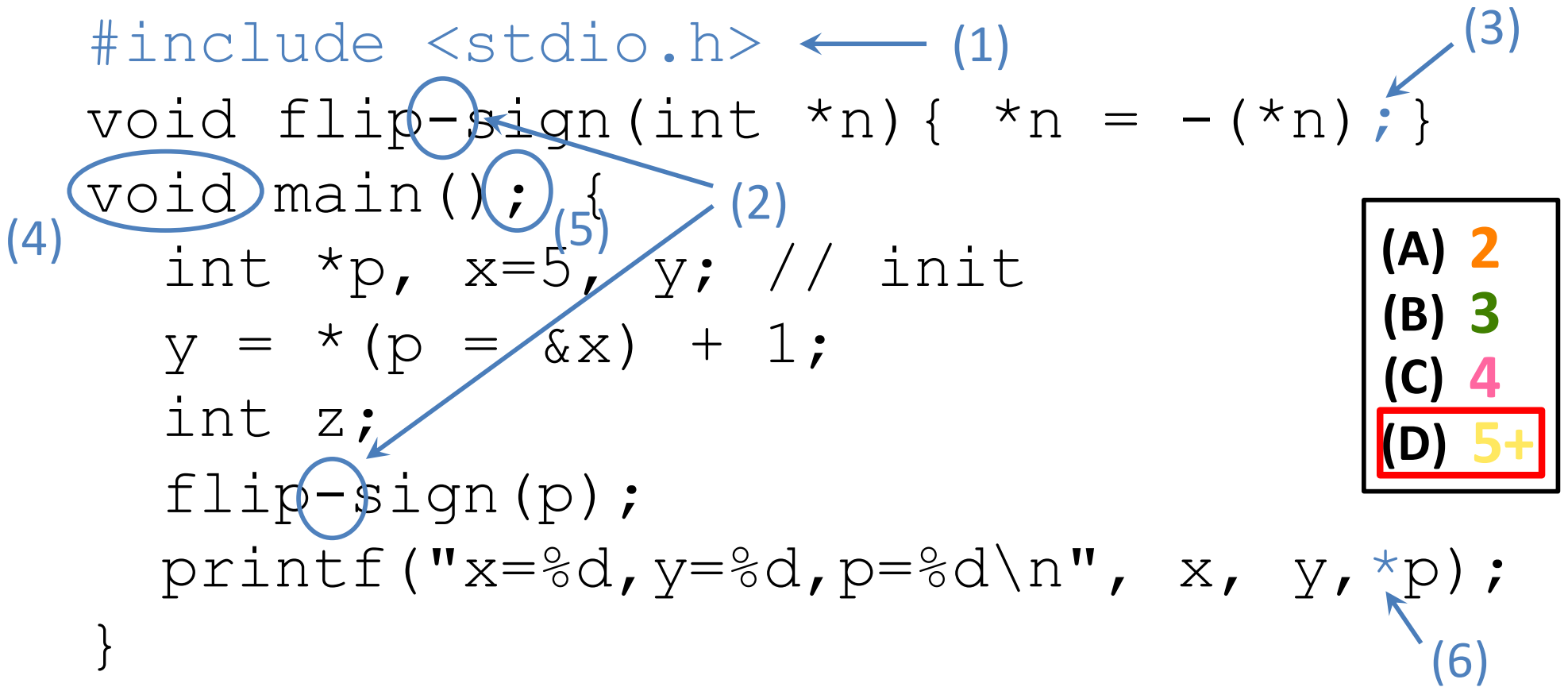
BAD

Question: How many errors (syntax and logic) in this C code (assume C99)?

```
void flip-sign(int *n){ *n = -(*n) }
void main(); {
   int *p, x=5, y; // init
   y = *(p = &x) + 1;
   int z;
   flip-sign(p);
   printf("x=%d,y=%d,p=%d\n", x, y, p);
}
```

(A) 2
(B) 3
(C) 4
(D) 5+

Answer: How many errors (syntax and logic) in this C code (assume C99)?



Question: What is output from the corrected code below?

```
#include <stdio.h>
void flip_sign(int *n){*n = -(*n);}
int main() {
   int *p, x=5, y; // init
   y = *(p = &x) + 1;
   int z;
   flip_sign(p);
   printf("x=%d,y=%d,*p=%d\n",x,y,*p);
}
```

	x,	y,	*p
(A)	5,	6,	-5
(B)	-5,	6,	-5
(C)	-5,	4,	-5
(D)	-5,	-6,	-5

Summary

- C is an efficient (compiled) language, but leaves safety to the programmer
  - Weak type safety, variables not auto-initialized
  - Use pointers with care: common source of bugs!
- Pointer is a C version (abstraction) of a data address
  - Each memory location has an address and a value stored in it
  - `*` "follows" a pointer to its value
  - `&` gets the address of a value
- C functions "pass by value"