CS61C

Great Ideas in Computer Architecture
(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Dan Garcia

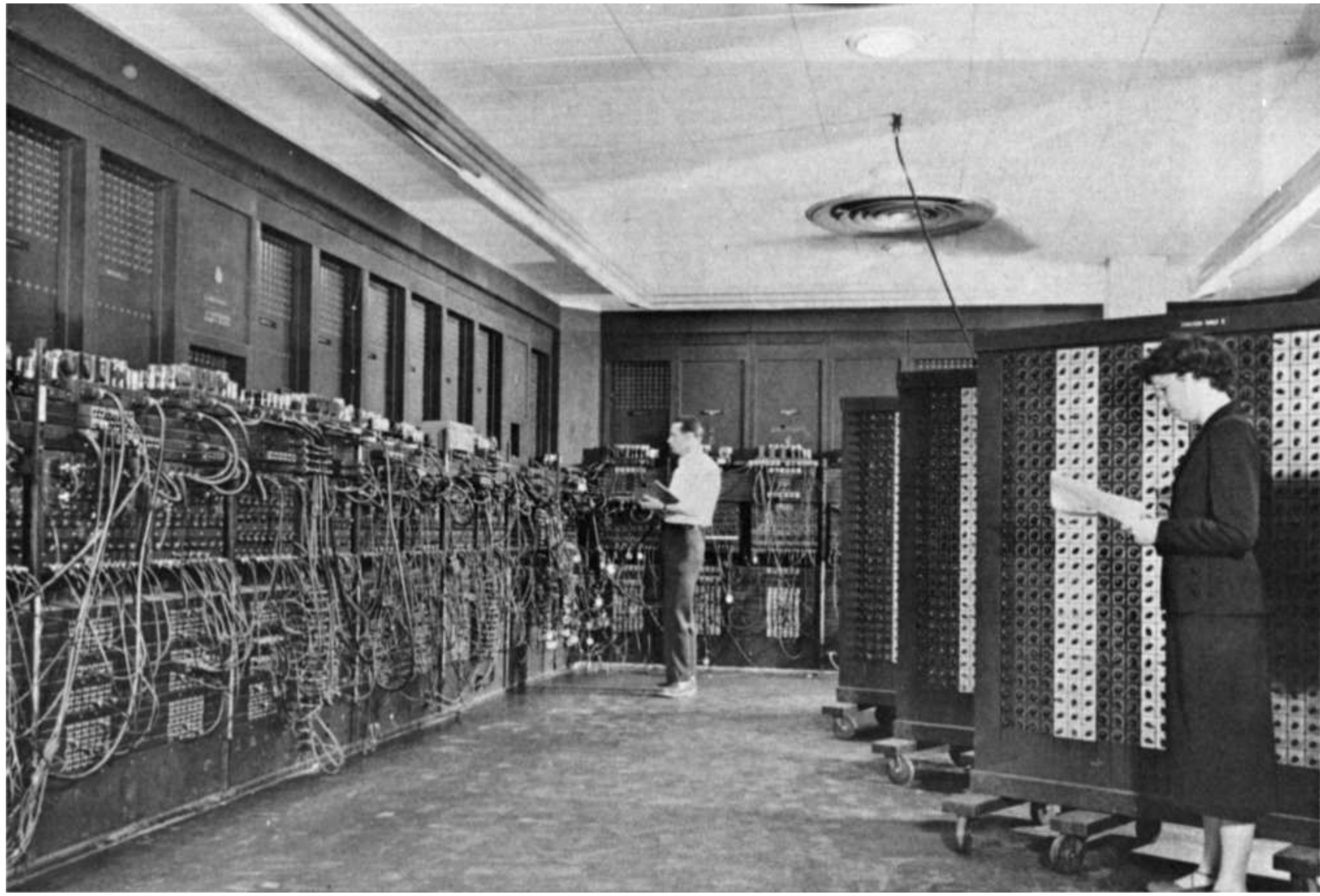
UC Berkeley
Professor
Bora Nikolić

Introduction to the C Programming Language

Computer Organization

ENIAC (U Penn, 1946)

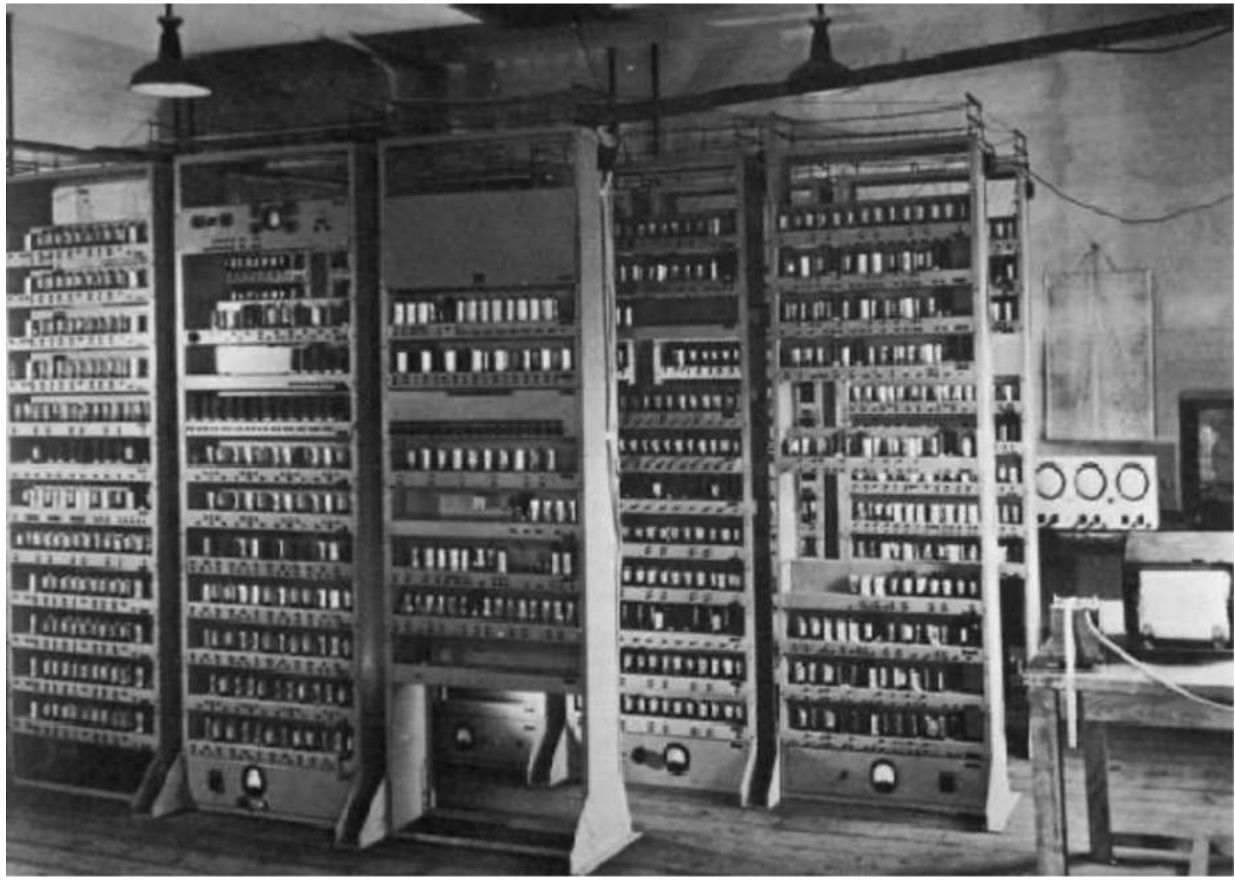
- First Electronic General-Purpose Computer
- Blazingly fast
  - Multiply in 2.8ms!
  - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program
- Programmed with patch cords and switches
  - At that time & before, "computer" mostly referred to people who did calculations



Garcia, Nikolić

EDSAC (Cambridge, 1949)

- First General Stored-Program Computer
- Programs held as numbers in memory
  - This is the revolution:
    It isn't just programmable, but the program is just the same type of data that the computer computes on
  - Bits are not just the numbers being manipulated, but the instructions on how to manipulate the numbers!
- 35-bit binary Twos complement words



Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

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

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

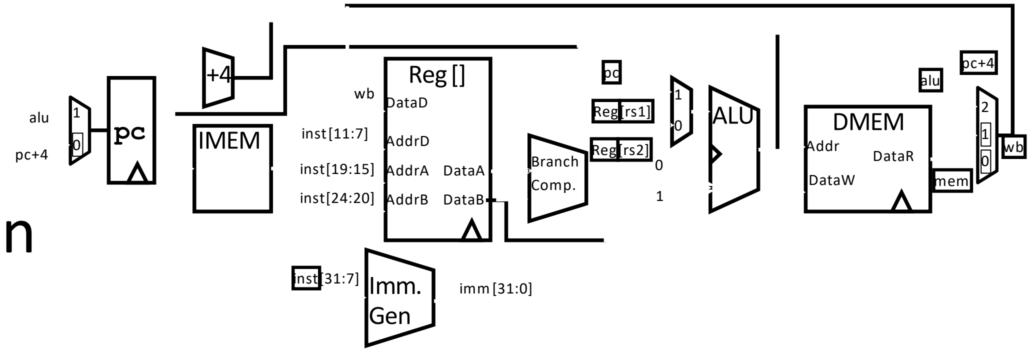
Anything can be represented as a number, i.e., data or instructions

Assembler

Machine Language Program (RISC-V)

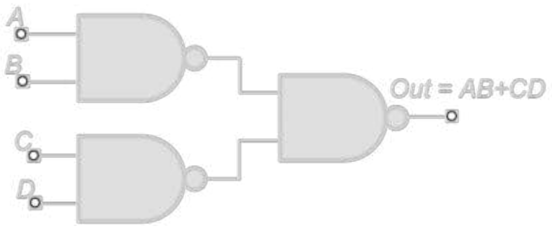
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



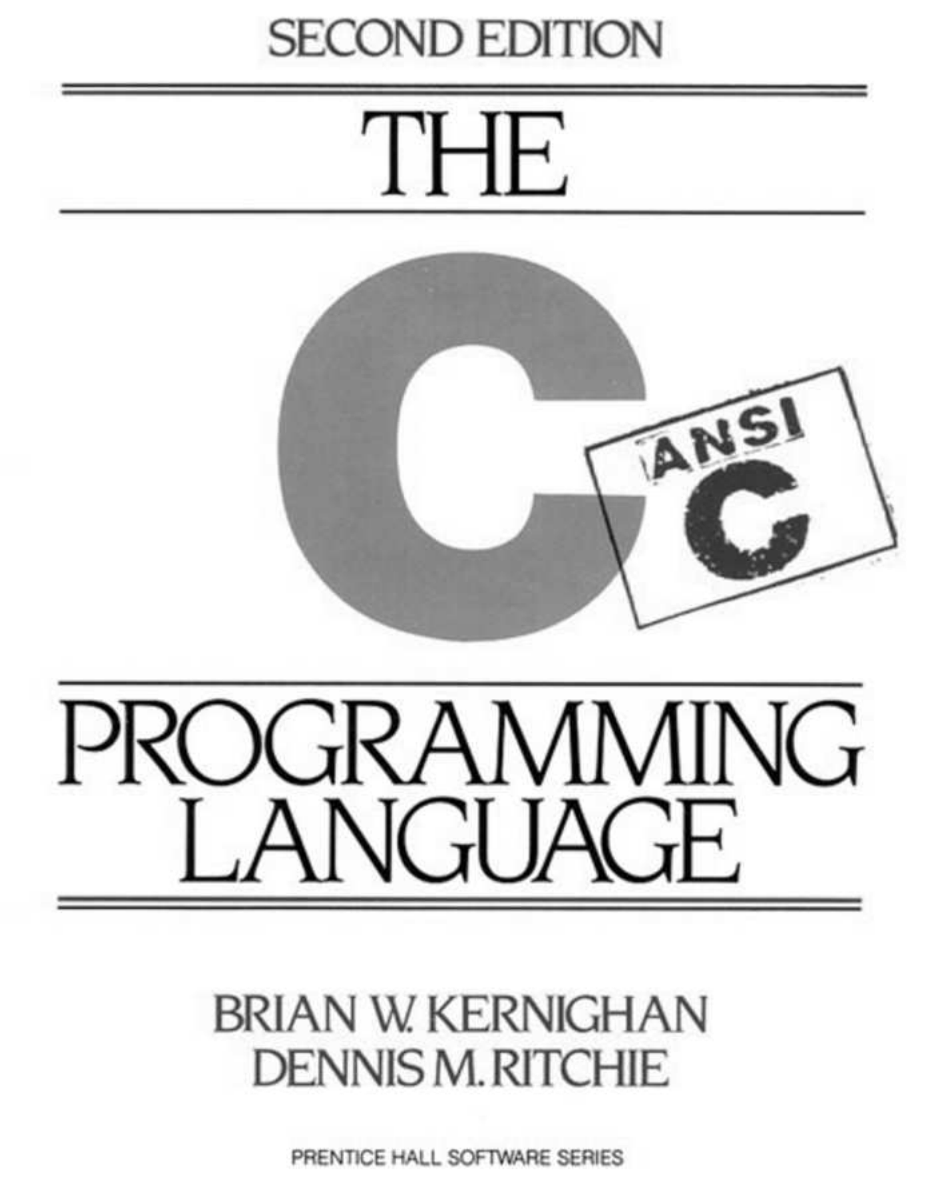
Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to C (1/2)

- Kernighan and Ritchie
  - *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.*
- Enabled first operating system not written in assembly language!
  - UNIX - A portable OS!



Introduction to C (2/2)

- Why C?
  - We can write programs that allow us to exploit underlying features of the architecture
    - memory management, special instructions, parallelism
- C and derivatives (C++/Obj-C/C#) still one of the most popular programming languages after >40 years!
- If you are starting a new project where performance matters use either Go or Rust
  - Rust, "C-but-safe": By the time your C is (theoretically) correct w/all necessary checks it should be no faster than Rust
  - Go, "Concurrency": Practical concurrent programming to take advantage of modern multi-core microprocessors

Disclaimer

- You will not learn how to fully code in C in these lectures! You'll still need your C reference
  - K&R is a *must-have*
  - Useful Reference: "JAVA in a Nutshell," O'Reilly
    - Chapter 2, "How Java Differs from C"
  - Brian Harvey's helpful transition notes
    - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
- Key C concepts: Pointers, Arrays, Implications for Memory management
  - Key security concept: All of the above are *unsafe*: If your program contains an error in these areas it might not crash immediately but instead leave the program in an inconsistent (and often exploitable) state

☑

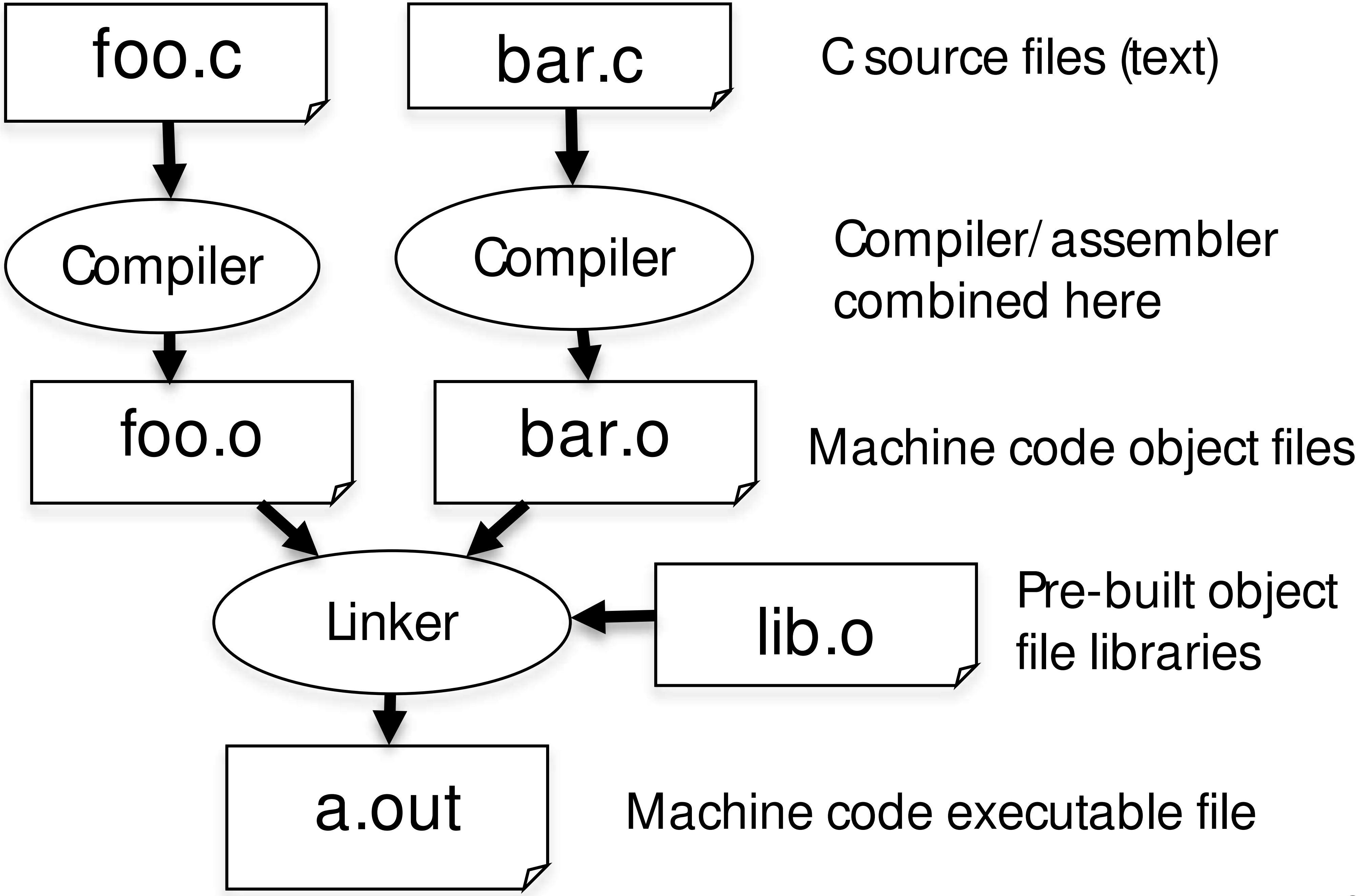
Compile
v.
Interpret

Compilation: Overview

- C compilers map C programs directly into architecture-specific machine code (string of 1s and 0s)
  - Unlike Java, which converts to architecture-independent bytecode that may then be compiled by a just-in-time compiler (JIT)
  - Unlike Python environments, which converts to a byte code at runtime
    - These differ mainly in exactly when your program is converted to low-level machine instructions (“levels of interpretation”)
- For C, generally a two part process of compiling .c files to .o files, then linking the .o files into executables;
  - Assembling is also done (but is hidden, i.e., done automatically, by default); we’ll talk about that later

Garcia, Nikolić

C Compilation Simplified Overview (more later)



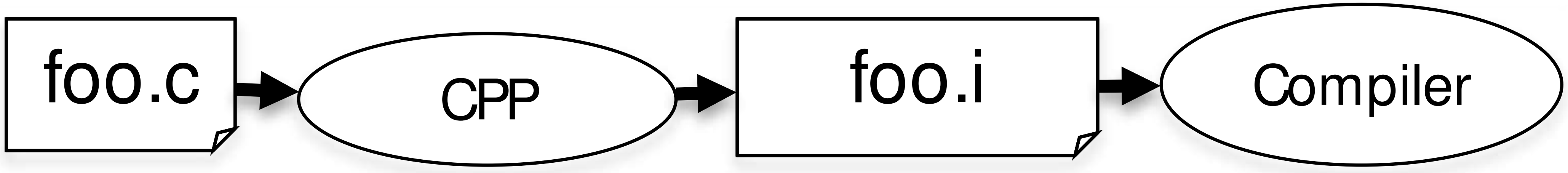
Compilation: Advantages

- Reasonable compilation time: enhancements in compilation procedure (**`Makefile`**s) allow only modified files to be recompiled
- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
  - But these days, a lot of performance is in libraries:
  - Plenty of people do scientific computation in Python!?!
    - they have good libraries for accessing GPU-specific resources
    - Also, many times python allows the ability to drive many other machines very easily … wait for Spark™ lecture
    - Also, Python can call low-level C code to do work: Cython

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., MIPS vs. x86 vs. RISC-V) and the operating system (e.g., Windows vs. Linux vs. MacOS)
- Executable must be rebuilt on each new system
  - I.e., “porting your code” to a new architecture
- “Change → Compile → Run [repeat]” iteration cycle can be slow during development
  - but make only rebuilds changed pieces, and can compile in parallel: **`make -j`**
  - linker is sequential though → Amdahl’s Law

C Pre-Processor (CPP)



- C source files first pass through macro processor, CPP, before compiler sees code
- CPP replaces comments with a single space
- CPP commands begin with "#"
  - **`#include "file.h"`** /* Inserts **`file.h`** into output */
  - **`#include <stdio.h>`** /* Looks for file in standard location, but no actual difference! */
  - **`#define PI (3.14159)`** /* Define constant */
  - **`#if/#endif`** /* Conditionally include text */
- Use **`-save-temps`** option to gcc to see result of preprocessing
  - Full documentation at: **`http://gcc.gnu.org/onlinedocs/cpp/`**

CPP Macros: A Warning...

- You often see C preprocessor macros defined to create small "functions"
  - But they aren't actual functions, instead it just changes the *text* of the program
  - In fact, all **`#define`** does is *string replacement*
  - **`#define min(X,Y) ((X)<(Y)?(X):(Y))`**
- This can produce, umm, interesting errors with macros, if **`foo(z)`** has a side-effect
  - **`next = min(w, foo(z));`**
  - **`next = ((w)<(foo(z))?(w):(foo(z)));`** ☑

C vs Java

C vs. Java (1/3)

	C	Java
Type of Language	Function Oriented	Object Oriented
Programming Unit	Function	Class = Abstract Data Type
Compilation	**gcc hello.c** creates machine language code	**javac Hello.java** creates Java virtual machine language bytecode
Execution	**a.out** loads and executes program	**java Hello** interprets bytecodes

hello, world	**#include <stdio.h>
	int main(void)
	{
	  printf("Hi\n");
	  return 0;
	}**
		**public class HelloWorld {
		 public static void
		main(String[] args) {
		    System.out.println("Hi");
		} }**

Storage	Manual (**malloc**, **free**)	New allocates & initializes, Automatic (garbage collection) frees

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

C vs. Java (2/3)

	C	Java
Comments (C99 same as Java)	`/* … */`	`/* … */` or `//` … end of line
Constants	**#define**, **const**	**final**
Preprocessor	Yes	No
Variable declaration (C99 same as Java)	At beginning of a block	Before you use it
Variable naming conventions	**sum_of_squares**	**sumOfSquares**
Accessing a library	**#include <stdio.h>**	**import java.io.File;**

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

C vs. Java (3/3) …operators nearly identical

- arithmetic: `+, -, *, /, %`
- assignment: `=`
- augmented assignment: `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
- bitwise logic: `~, &, |, ^`
- bitwise shifts: `<<, >>`
- boolean logic: `!, &&, ||`
- equality testing: `==, !=`
- subexpression grouping: `()`
- order relations: `<, <=, >, >=`
- increment and decrement: `++` and `--`
- member selection: `.`, `->`
  - Slightly different than Java because there are both structures and pointers to structures, more later
- conditional evaluation: `? :`

Has there been an update to ANSI C?

- Yes! It's called the "C99" or "C9x" std
  - To be safe: "**`gcc -std=c99`**" to compile
  - **`printf("%ld\n", __STDC_VERSION__);`** ➔ **`199901`**
- References
  - **`en.wikipedia.org/wiki/C99`**
- Highlights
  - Declarations in **`for`** loops, like Java
  - Java-like **`//`** comments (to end of line)
  - Variable-length non-global arrays
  - **`<inttypes.h>`**: explicit integer types
  - **`<stdbool.h>`** for boolean logic def's

Has there been an update to C99?

- Yes! It's called the "C11" (C18 fixes bugs…)
  - You need "`gcc -std=c11`" (or `c17`) to compile
  - `printf("%ld\n", __STDC_VERSION__);` ➔ `201112L`
  - `printf("%ld\n", __STDC_VERSION__);` ➔ `201710L`
- References
  - `en.wikipedia.org/wiki/C11_(C_standard_revision)`
- Highlights
  - Multi-threading support!
  - Unicode strings and constants
  - Removal of `gets()`
  - Type-generic Macros (dispatch based on type)
  - Support for complex values
  - Static assertions, Exclusive create-and-open, …

C Syntax: main

- To get the **main** function to accept arguments, use this:
  - `int main (int argc, char *argv[])`
- What does this mean?
  - **argc** will contain the number of strings on the command line (the executable counts as one, plus one for each argument). Here **argc** is 2:
    - `unix% sort myFile`
  - **argv** is a pointer to an array containing the arguments as strings (more on pointers later).

☑

Garcia, Nikolić

C Syntax

C Syntax: True or False?

- What evaluates to FALSE in C?
  - **`0`** (integer)
  - **`NULL`** (pointer: more on this later)
  - Boolean types provided by C99's **`stdbool.h`**
- What evaluates to TRUE in C?
  - …everything else…
  - Same idea as in Scheme
    - Only **`#f`** is false, everything else is true!

Garcia, Nikolić

Typed Variables in C

- Must declare the type of data a variable will hold
  - Types can't change. Eg, **`int var = 2;`**

Type	Description	Example
`int`	Integer Numbers (including negatives) At least 16 bits, can be larger	**`0, 78, -217, 0x7337`**
`unsigned int`	Unsigned Integers	**`0, 6, 35102`**
`float`	Floating point decimal	**`0.0, 3.14159, 6.02e23`**
`double`	Equal or higher precision floating point	**`0.0, 3.14159, 6.02e23`**
`char`	Single character	**`'a', 'D', '\n'`**
`long`	Longer **`int`**, Size >= **`sizeof(int)`**, at least 32b	**`0, 78, -217, 301720971`**
`long long`	Even longer **`int`**, size >= **`sizeof(long)`**, at least 64b	**`31705192721092512`**

Integers: Python vs. Java vs. C

- C: **`int`** should be integer type that target processor works with most efficiently
- Only guarantee:
  - **`sizeof(long long)`**
    ≥ `sizeof(long)` ≥ `sizeof(int)` ≥ `sizeof(short)`
  - Also, **`short`** >= 16 bits, **`long`** >= 32 bits
  - All could be 64 bits
  - This is why we encourage you to use **`int`*`N`*`_t`** and **`uint`*`N`*`_t`**!!

Language	**`sizeof(int)`**
Python	>=32 bits (plain **`int`**s), infinite (long **`int`**s)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

Consts and Enums in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float  golden_ratio = 1.618;
const int    days_in_week = 7;
const double the_law      = 2.99792458e8;
```

  - You can have a constant version of any of the standard C variable types

- Enums: a group of related integer constants. Eg.,

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};
```

Typed Functions in C

- You have to declare the type of data you plan to return from a function
- Return type can be any C variable type, and is placed to the left of the function name
- You can also specify the return type as **`void`**
  - Just think of this as saying that no value will be returned
- Also need to declare types for values passed into a function
- Variables and functions MUST be declared before they are used

```
int number_of_people    () {  return 3;     }
float dollars_and_cents () {  return 10.33; }
```

Structs in C

- Typedef allows you to define new types.

```
typedef uint8_t BYTE;
BYTE b1, b2;
```

- Structs are structured groups of variables e.g.,

```
typedef struct {
    int length_in_seconds;
    int year_recorded;
} SONG;

SONG song1;
song1.length_in_seconds  =  213;
song1.year_recorded      = 1994;

SONG song2;
song2.length_in_seconds  =  248;
song2.year_recorded      = 1988;
```

Dot notation: **`x.y = value`**

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs (shows Java's legacy) for control flow
  - A statement can be a `{}` of code or just a standalone statement
- if-else
  - `if (expression) statement`

```
if (x == 0) y++;
if (x == 0) {y++;}
if (x == 0) {y++; j = j + y;}
```

  - `if (expression) statement1 else statement2`
    - There is an ambiguity in a series of if/else if/else if you don't use `{}`s, so use `{}`s to block the code
    - In fact, it is a bad C habit to not always have the statement in `{}`s, it has resulted in some amusing errors...
- while
  - `while (expression) statement`
  - `do statement while (expression);`

C Syntax : Control Flow (2/2)

- for

```
for (initialize; check; update) statement
```

- switch

```
switch (expression){
  case const1:    statements
  case const2:    statements
  default:        statements
}
break;
```

  - Note: until you do a **break** statement things keep executing in the switch statement

- C also has **goto**
  - But it can result in spectacularly bad code if you use it, so don't!

First Big C Program: Compute Sines table

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    int    angle_degree;
    double angle_radian, pi, value;

    printf("Compute a table of the sine function\n\n");
    pi = 4.0*atan(1.0); /* could also just use pi = M_PI */
    printf("Value of PI = %f \n\n", pi);
    printf("Angle\tSine\n");
    angle_degree = 0;/* initial angle value */
    while (angle_degree <= 360) { /* loop til angle_degree > 360 */
        angle_radian = pi * angle_degree / 180.0;
        value = sin(angle_radian);
        printf ("%3d\t%f\n ", angle_degree, value);
        angle_degree += 10; /* increment the loop index */
    }
 return 0;
}
```

```
PI = 3.141593

Angle  Sine
  0    0.000000
  10   0.173648
  20   0.342020
  30   0.500000
  40   0.642788
  50   0.766044
  60   0.866025
  70   0.939693
  80   0.984808
  90   1.000000
… etc …
```

☑

Garcia, Nikolić

Bugs, and Pointers

C Syntax: Variable Declarations

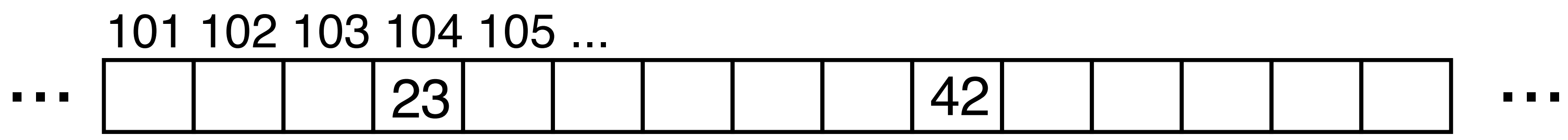
- Similar to Java, but with a few minor but important differences
  - All variable declarations must appear before they are used
  - All must be at the beginning of a block.
  - A variable may be initialized in its declaration; *if not, it holds garbage!*
    - the contents are undefined…
- Examples of declarations:
  - Correct: `{ int a = 0, b = 10;` …
  - Incorrect in ANSI C: `for (int i=0;` …
  - Correct in C99 (and beyond): `for (int i=0;`…

An Important Note: Undefined Behavior…

- A lot of C has "Undefined Behavior"
  - This means it is often unpredictable behavior
    - It will run one way on one computer…
    - But some other way on another
    - Or even just be different each time the program is executed!
- Often characterized as "Heisenbugs"
  - Bugs that seem random/hard to reproduce, and seem to disappear or change when debugging
  - Cf. "Bohrbugs" which are repeatable

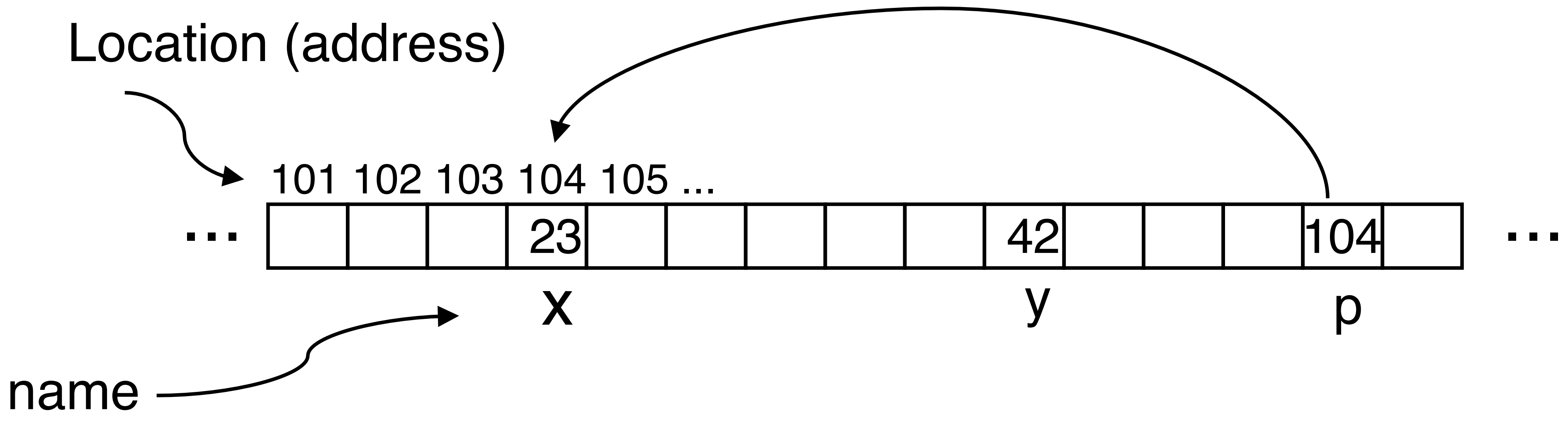
Address vs. Value

- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
  - Do you think they use signed or unsigned numbers? Negative address?!
- Don't confuse the address referring to a memory location with the value stored in that location.
- For now, the abstraction lets us think we have access to ∞ memory, numbered from 0…



Pointers

- An address refers to a particular memory location. In other words, it points to a memory location.
- Pointer: A variable that contains the address of a variable.



Pointer Syntax

- **`int *p;`**
  - Tells compiler that variable **`p`** is address of an **`int`**

- **`p = &y;`**
  - Tells compiler to assign address of **`y`** to **`p`**
  - **`&`** called the “address operator” in this context

- **`z = *p;`**
  - Tells compiler to assign value at address in **`p`** to **`z`**
  - **`*`** called the “dereference operator” in this context

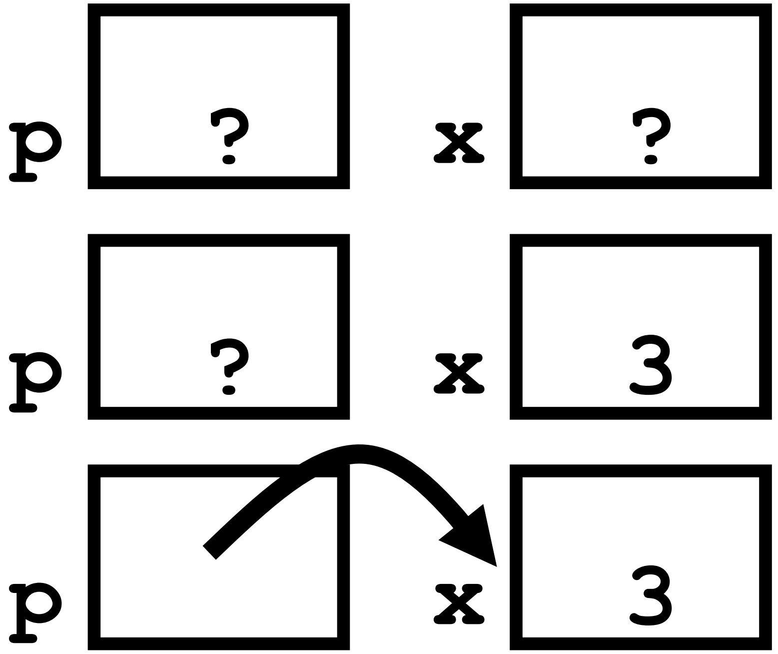
Pointers

- How to create a pointer:
  - & operator: get address of a variable

```
int *p, x;

x = 3;

p =&x;
```



Note the "*" gets used 2 different ways in this example. In the declaration to indicate that **`p`** is going to be a pointer, and in the **`printf`** to get the value pointed to by **`p`**.

- How get a value pointed to?
  - * "dereference operator": get value pointed to

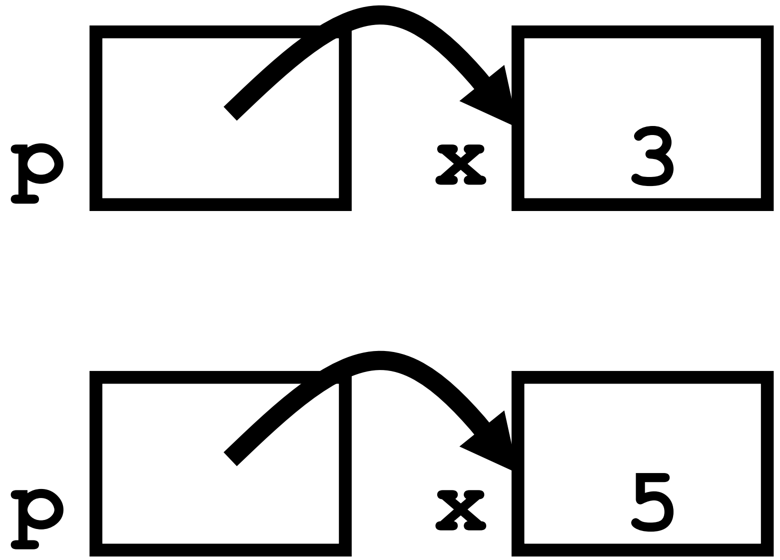
```
printf("p points to %d\n",*p);
```

Garcia, Nikolić

Pointers

- How to change a variable pointed to?
  - Use dereference `*` operator on left of `=`

```
*p = 5;
```



Garcia, Nikolić

Pointers and Parameter Passing (1/2)

- Java and C pass parameters "by value"
  - procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

```
void addOne (int x) {
     x = x + 1;
}
int y = 3;
addOne(y);
```

y is still = **3**

Garcia, Nikolić

Pointers and Parameter Passing (2/2)

- How to get a function to change a value?

```
void addOne (int *p) {
    *p = *p + 1;
}
int y = 3;
addOne(&y);
```

`y` is now = **4**

More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.
- What does the following code do?

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

Pointers in C … The Good, Bad, and the Ugly

- Why use pointers?
  - If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
    - Otherwise we'd need to copy a huge amount of data
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    - Most problematic with dynamic memory management—coming up next time
    - Dangling references and memory leaks

☑

Using Pointers Effectively

Pointers

- Pointers are used to point to any data type (**`int`**, **`char`**, a **`struct`**, etc.).
- Normally a pointer can only point to one type (**`int`**, **`char`**, a **`struct`**, etc.).
  - **`void *`** is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs… and security issues… and a lot of other bad things!
- You can even have pointers to functions…
  - **`int (*fn) (void *, void *) = &foo`**
    - **`fn`** is a function that accepts two **`void *`** pointers and returns an **`int`** and is initially pointing to the function **`foo`**.
  - **`(*fn)(x, y)`** will then call the function

Garcia, Nikolić

Pointers and Structures

```
typedef struct {
    int x;
    int y;
} Point;

Point p1;
Point p2;
Point *paddr;
```

```
/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too */
p1 = p2;
```

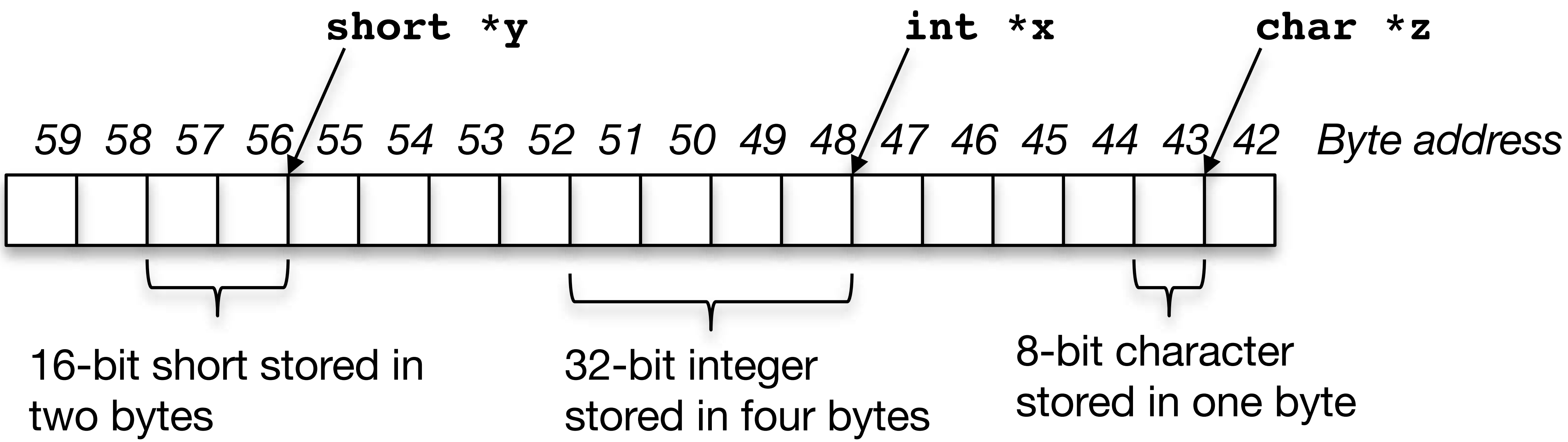
NULL pointers...

- The pointer of all 0s is special
  - The "NULL" pointer, like in Java, python, etc...
- If you write to or read a null pointer, your program should crash
- Since "0 is false", its very easy to do tests for null:
  - `if(!p) { /* P is a null pointer */ }`
  - `if(q) { /* Q is not a null pointer */ }`

Garcia, Nikolić

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
  - Hardware's memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - Eg., 32-bit integer stored in 4 consecutive 8-bit bytes
- But we actually want "word alignment"
  - Some processors will not allow you to address 32b values without being on 4 byte boundaries
  - Others will just be very slow if you try to access "unaligned" memory.



Arrays

Arrays (1/5)

- Declaration:
  - **`int ar[2];`**
  - …declares a 2-element integer array
  - An array is really just a block of memory
- Declaration and initialization
  - **`int ar[] = {795, 635};`**
  - declares and fills a 2-elt integer array
- Accessing elements:
  - **`ar[num]`**
  - returns the numth element.

Arrays (2/5)

- Arrays are (almost) identical to pointers
  - **`char *string`** and **`char string[]`** are nearly identical declarations
  - They differ in very subtle ways: incrementing, declaration of filled arrays
- Key Concept: An array variable is a "pointer" to the first element.

Garcia, Nikolić

Arrays (3/5)

- Consequences:
  - **ar** is an array variable but looks like a pointer in many respects (though not all)
  - **ar[0]** is the same as ***ar**
  - **ar[2]** is the same as ***(ar+2)**
  - We can use pointer arithmetic to access arrays more conveniently.
- Declared arrays are only allocated while the scope is valid

```
char *foo() {
   char string[32]; ...;
   return string;
}
```
is incorrect

Arrays (4/5)

- Array size n; want to access from 0 to n-1, so you should use counter AND utilize a variable for declaration & incr
  - Wrong

```
int i, ar[10];
for(i = 0; i < 10; i++){ ... }
```

  - Right

```
int ARRAY_SIZE = 10;
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Why? SINGLE SOURCE OF TRUTH
  - You're utilizing indirection and <u>avoiding maintaining two copies</u> of the number 10

Garcia, Nikolić

Arrays (5/5)

- Pitfall: An array in C does <u>not</u> know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array <u>and its size</u> to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are VERY difficult to find; be careful!
  - You'll learn how to debug these in lab…

Pointer Arithmetic

- pointer + n
  - Adds **`n*sizeof(`**“whatever pointer is pointing to”**`)`** to the memory address

- pointer – n
  - Adds **`n*sizeof(`**“whatever pointer is pointing to”**`)`** to the memory address

Pointers (1/4) …review…

- Java and C pass parameters "by value"
  - procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

```
void addOne (int x) {
    x = x + 1;
}
int y = 3;
addOne(y);
```

`y` is still = `3`

Pointers (2/4) …review…

- How to get a function to change a value?

```
void addOne (int *p) {
    *p = *p + 1;
}
int y = 3;
addOne(&y);
```

y is now = **4**

Garcia, Nikolić

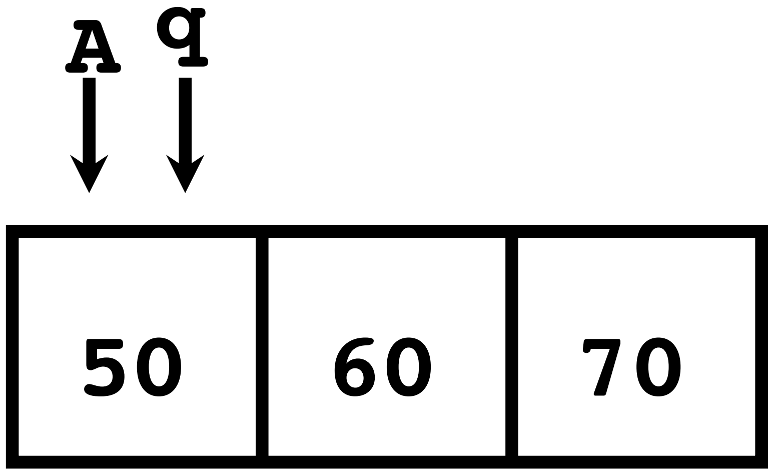
Pointers (3/4)

- But what if you want to change a pointer?
  - What gets printed?

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

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr( q);
printf("*q = %d\n", *q);
```

*q = 50



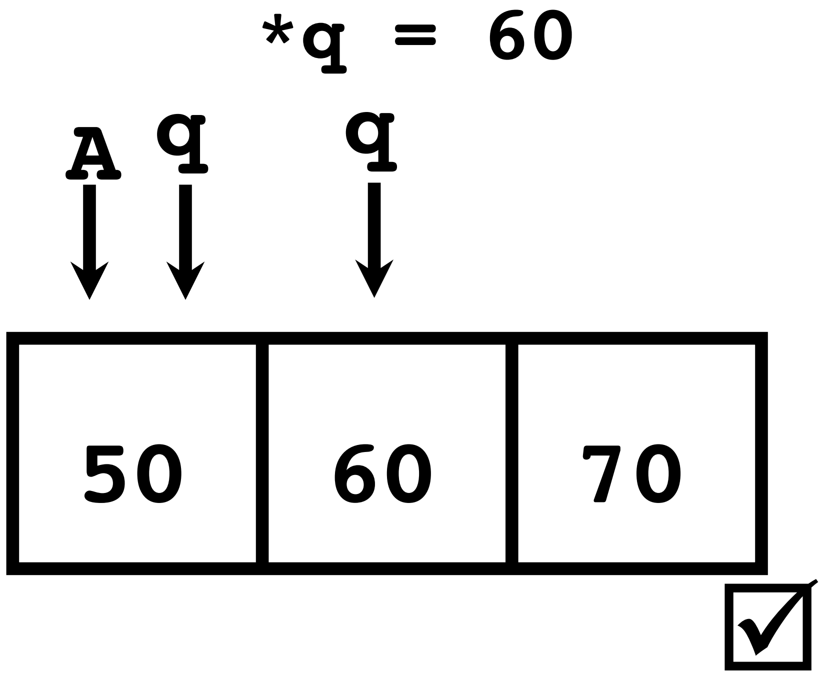
Garcia, Nikolić

Pointers (4/4)

- Idea! Pass a pointer to a pointer!
  - Declared as **`**h`**
  - Now what gets printed?

```
void IncrementPtr(int **h)
{   *h = *h + 1;   }

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);
printf("*q = %d\n", *q);
```



Garcia, Nikolić

Function Pointer Example

`map` (actually `mutate_map` easier)

```c
#include <stdio.h>

int x10(int), x2(int);
void mutate_map(int [], int n, int(*)(int));
void print_array(int [], int n);

int x2 (int n) {    return 2*n;    }
int x10(int n) {    return 10*n;   }

void mutate_map(int A[], int n, int(*fp)(int)) {
    for (int i = 0; i < n; i++)
        A[i] = (*fp)(A[i]);
}

void print_array(int A[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ",A[i]);
    printf("\n");
}
```

```
% ./map
3 1 4
6 2 8
60 20 80
```

```c
int main(void)
{
    int A[] = {3,1,4}, n = 3;
    print_array(A, n);
    mutate_map (A, n, &x2);
    print_array(A, n);
    mutate_map (A, n, &x10);
    print_array(A, n);
}
```

Memory

Dynamic Memory Allocation (1/4)

- C has operator **`sizeof()`** which gives size in bytes (of type or variable)
- Assume size of objects can be misleading and is bad style, so use **`sizeof(type)`**
  - Many years ago an **`int`** was 16 bits, and programs were written with this assumption.
  - What is the size of integers now?
- "**`sizeof`**" knows the size of arrays:

```
int ar[3]; // Or:   int ar[] = {54, 47, 99}
sizeof(ar) → 12
```

  - …as well for arrays whose size is determined at run-time:

```
int n = 3;
int ar[n]; // Or: int ar[fun_that_returns_3()];
sizeof(ar) → 12
```

Dynamic Memory Allocation (2/4)

- To allocate room for something new to point to, use **`malloc()`** (with the help of a typecast and **`sizeof`**):

```
ptr = (int *) malloc (sizeof(int));
```

  - Now, **`ptr`** points to a space somewhere in memory of size (**`sizeof(int)`**) in bytes.
  - **`(int *)`** simply tells the compiler what will go into that space (called a typecast).
- **`malloc`** is almost never used for 1 var
- **`ptr = (int *) malloc (n*sizeof(int));`**
  - This allocates an array of n integers.

Dynamic Memory Allocation (3/4)

- Once **`malloc()`** is called, the memory location contains garbage, so don't use it until you've set its value.
- After dynamically allocating space, we must dynamically free it:
  - **`free(ptr);`**
- Use this command to clean up.
  - Even though the program frees all memory on exit (or when main returns), don't be lazy!
  - You never know when your main will get transformed into a subroutine!

Garcia, Nikolić

Dynamic Memory Allocation (4/4)

- The following two things will cause your program to crash or behave strangely later on, and cause VERY VERY hard to figure out bugs:
  - **`free()`** ing the same piece of memory twice
  - calling **`free()`** on something you didn't get back from **`malloc()`**
- The runtime does not check for these mistakes
  - Memory allocation is so performance-critical that there just isn't time to do this
  - The usual result is that you corrupt the memory allocator's internal structure
  - You won't find out until much later on, in a totally unrelated part of your code!

Garcia, Nikolić

Managing the Heap: **realloc(p, size)**

- Resize a previously allocated block at **p** to a new size
- If **p** is NULL, then **realloc** behaves like **malloc**
- If size is 0, then **realloc** behaves like **free**, deallocating the block from the heap
- Returns new address of the memory block; NOTE: it is likely to have moved!

```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check NULL, contents of first 10
elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

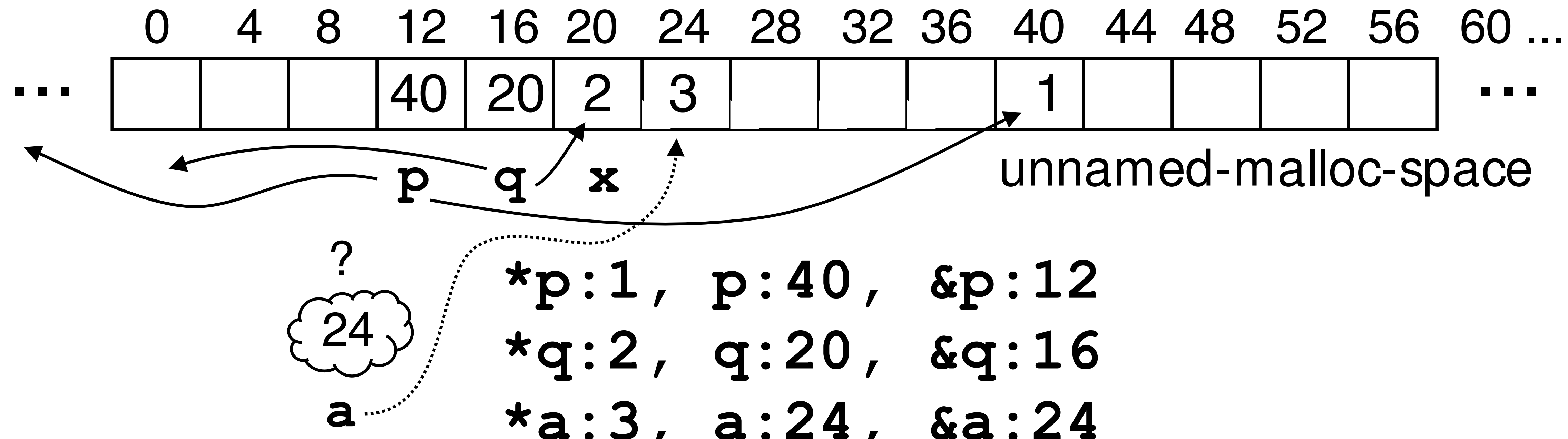
Arrays not implemented as you'd think

```
void foo() {
    int *p, *q, x;
    int a[4];
    p = (int *)
    malloc (sizeof(int));
    q = &x;

    *p = 1; // p[0] would also work here
    printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);

    *q = 2; // q[0] would also work here
    printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);

    *a = 3; // a[0] would also work here
    printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```



***p:1, p:40, &p:12**
***q:2, q:20, &q:16**
***a:3, a:24, &a:24**

K&R: "An array name is not a variable"

Garcia, Nikolić

Mini-summary

- Pointers and arrays are virtually same
- C knows how to increment pointers
- C is an efficient language, with little protection
  - Array bounds not checked
  - Variables not automatically initialized
- Use handles to change pointers
- Dynamically allocated heap memory must be manually deallocated in C.
  - Use **`malloc()`** and **`free()`** to allocate and deallocate memory from heap.
- (Beware) The cost of efficiency is more overhead for the programmer.
  - "C gives you a lot of extra rope, don't hang yourself with it!"

☑

Linked List Example

Linked List Example

- Let's look at an example of using structures, pointers, **malloc()**, and **free()** to implement a linked list of strings.

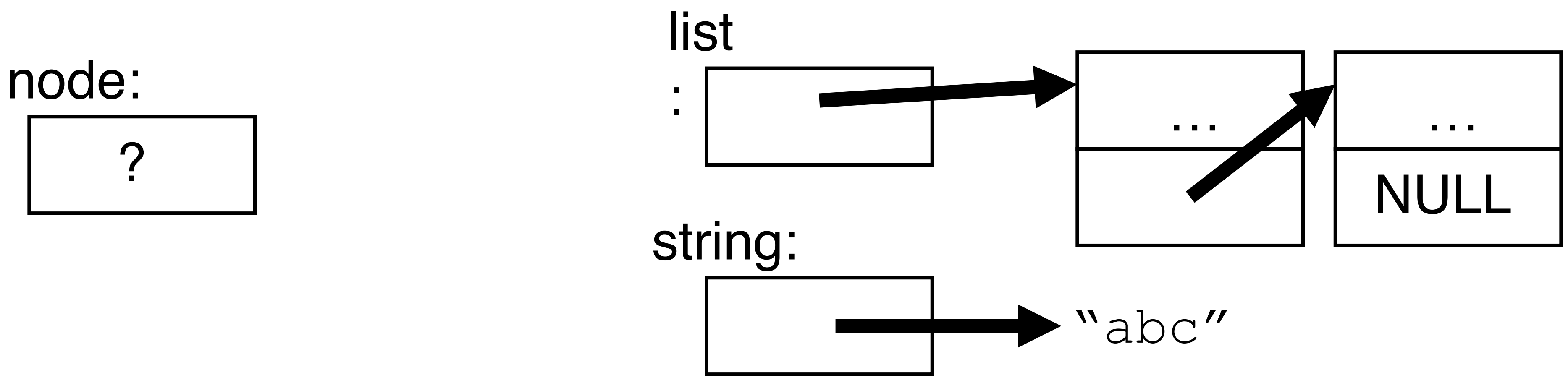
```
struct Node {
    char *value;
    struct Node *next;
};
typedef struct Node *List;

/* Create a new (empty) list */
List ListNew(void)
{ return NULL; }
```

Garcia, Nikolić

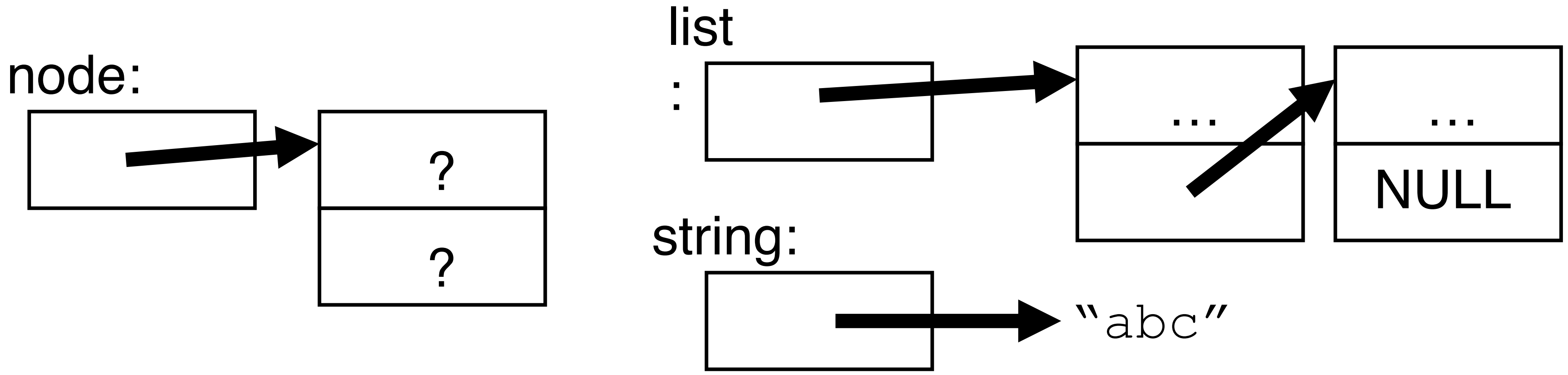
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



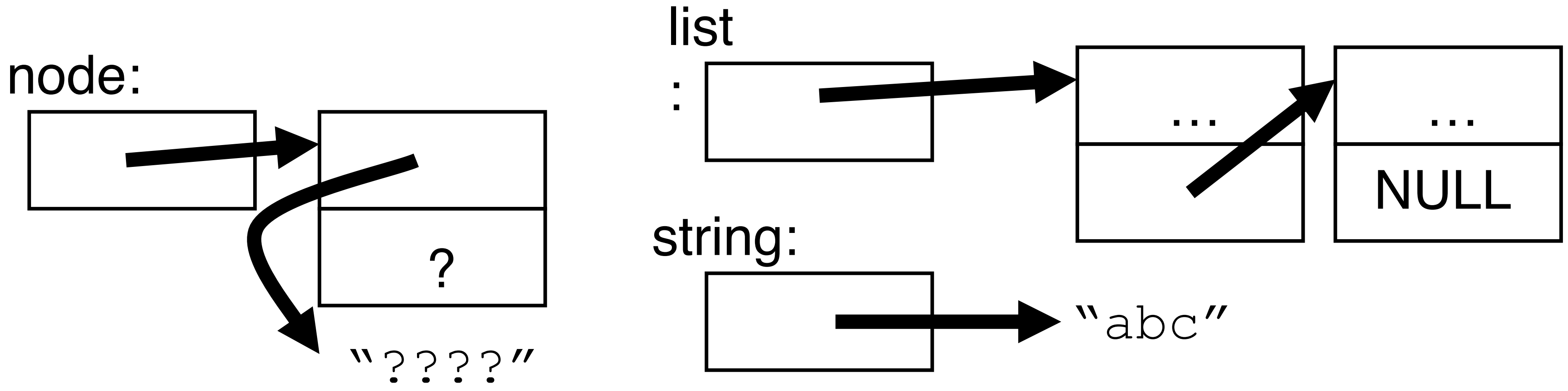
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Linked List Example

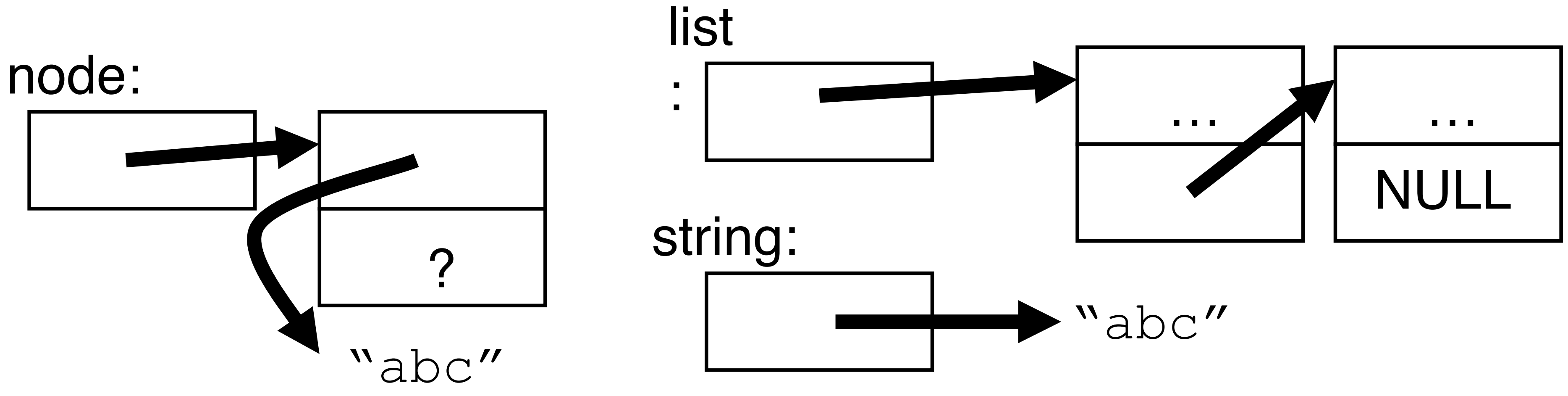
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Garcia, Nikolić

Linked List Example

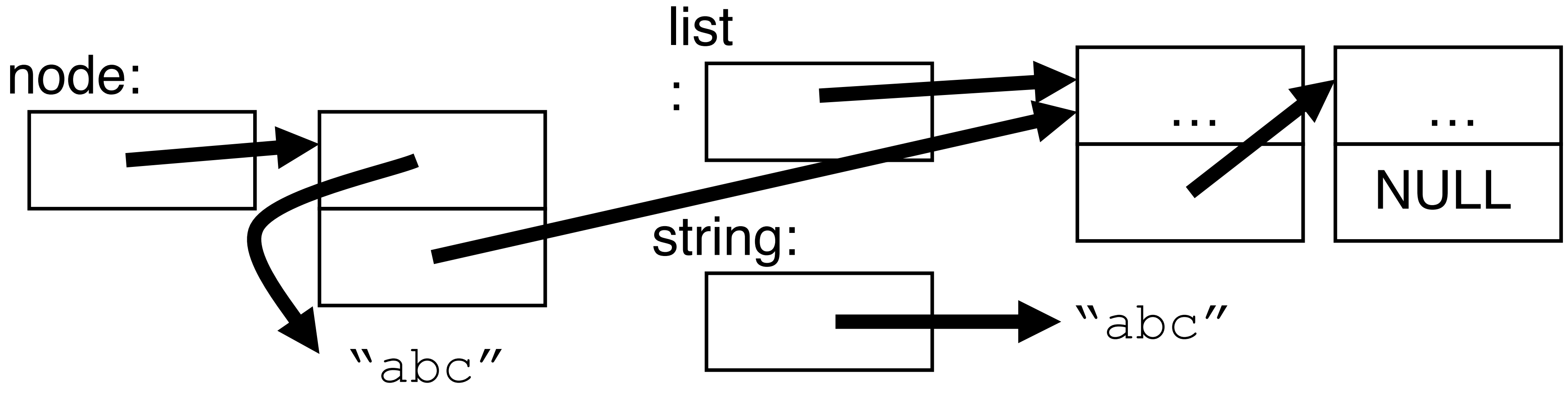
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Garcia, Nikolić

Linked List Example

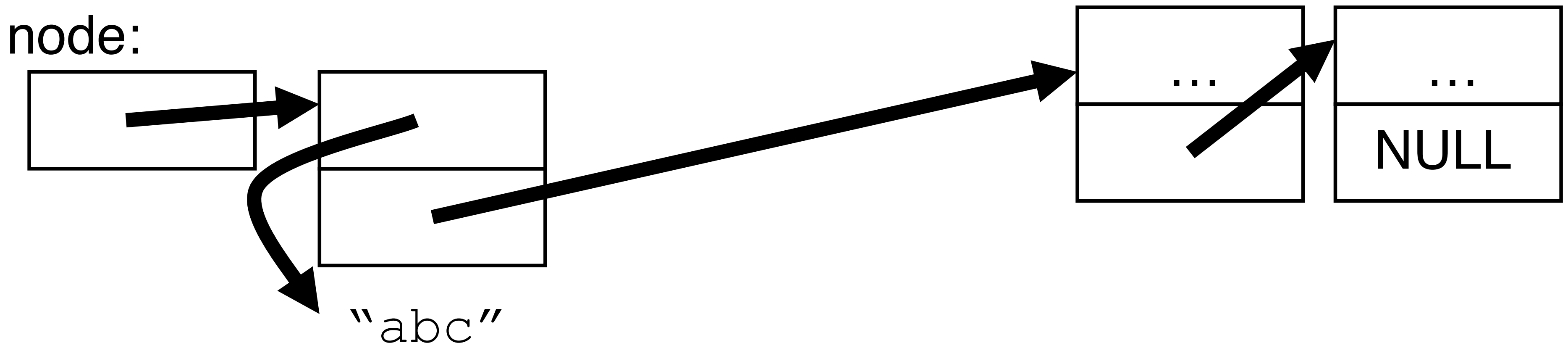
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Garcia, Nikolić

Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



☑

Garcia, Nikolić

Memory Locations

Don't forget the globals!

- What is stored?
  - Structure declaration does not allocate memory
  - Variable declaration does allocate memory
- So far we have talked about several different ways to allocate memory for data:
  - Declaration of a local variable

```
int i; struct Node list; char *string; int ar[n];
```

  - "Dynamic" allocation at runtime by calling allocation function (alloc).

```
ptr = (struct Node *) malloc (sizeof(struct Node)*n);
```

- One more possibility exists…
  - Data declared outside of any procedure (i.e., before **main**).
  - Similar to #1 above, but has "global" scope.

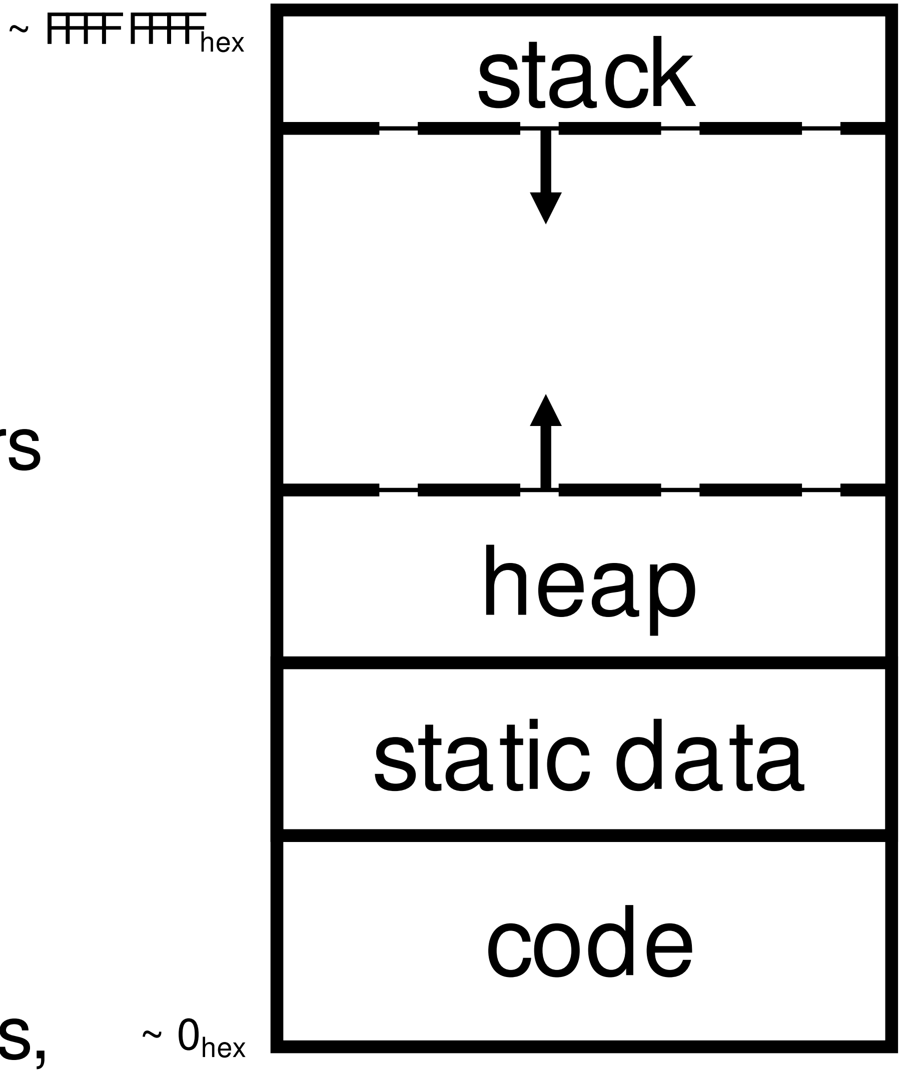
```
int myGlobal;
main() {
}
```

C Memory Management

- C has 3 pools of memory
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address (location of “activation records” in Java or “stack frame” in C)
  - The Heap (dynamic malloc storage): data lives until deallocated by programmer
- C requires knowing where objects are in memory, otherwise things don’t work as expected
  - Java hides location of objects

Normal C Memory Management

- A program's address space contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via **`malloc()`** ; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

Where are variables allocated?

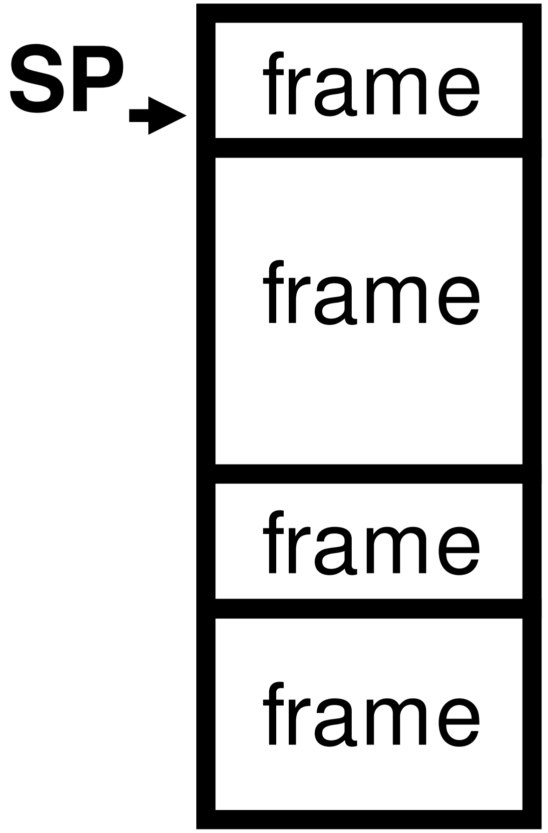
- If declared outside a procedure (global), allocated in “static” storage
- If declared inside procedure (local), allocated on the “stack”
  and freed when procedure returns.
  - NB: **`main()`** is a procedure

```
int myGlobal;
main() {
  int myTemp;
}
```


Garcia, Nikolić

The Stack

- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where top stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames

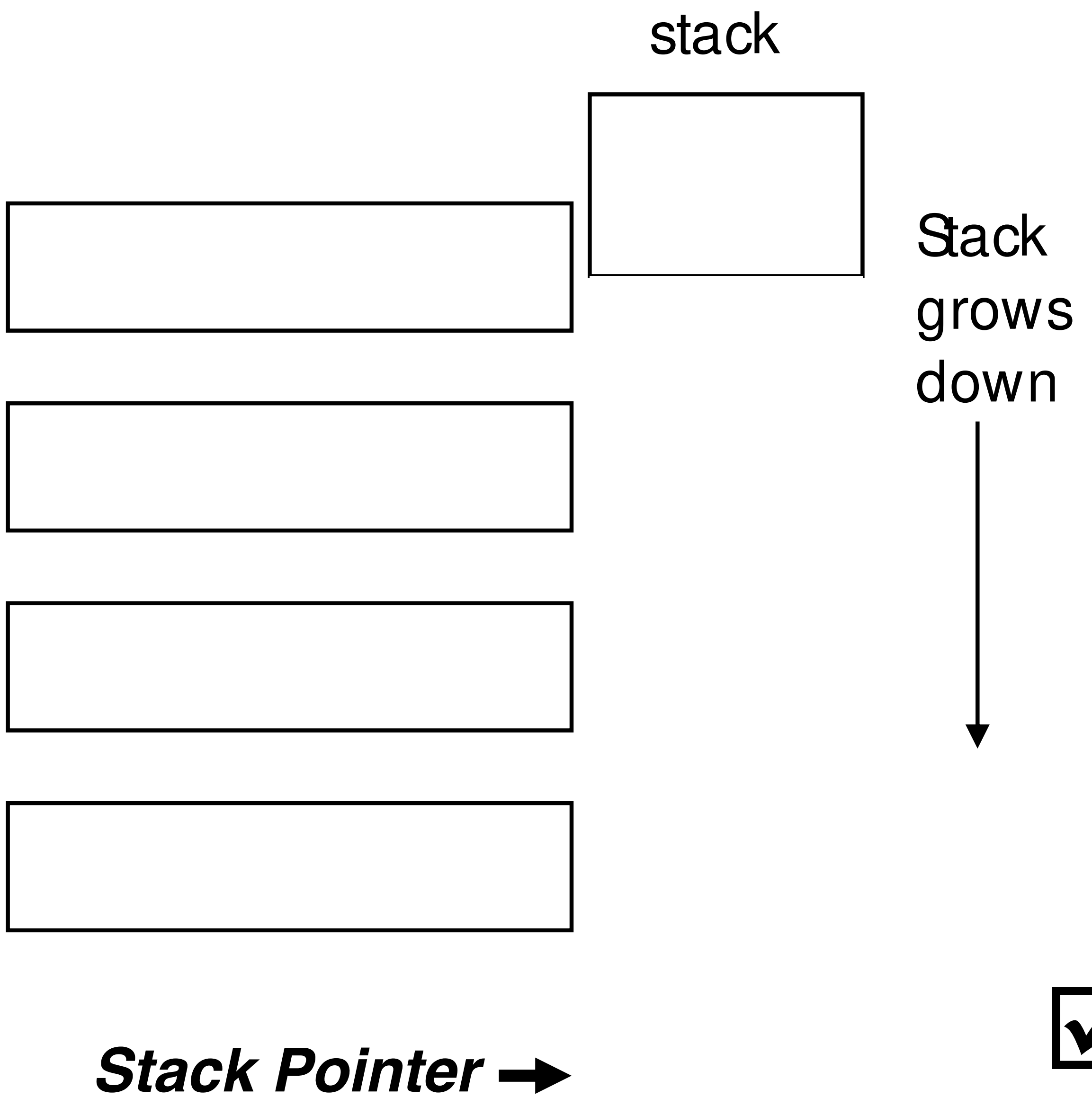


Garcia, Nikolić

Stack

- Last In, First Out (LIFO) data structure

```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }void d (int p)
        {
        }
```



Memory Management

The Heap (Dynamic memory)

- Large pool of memory,
  not allocated in contiguous order
  - back-to-back requests for heap memory could result blocks very far apart
  - where Java new command allocates memory
- In C, specify number of bytes of memory explicitly to allocate item

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

  - **malloc()**: Allocates raw, uninitialized memory from heap

Memory Management

- How do we manage memory?
- Code, Static storage are easy:
  - they never grow or shrink
- Stack space is also easy:
  - stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky:
  - memory can be allocated / deallocated at any time

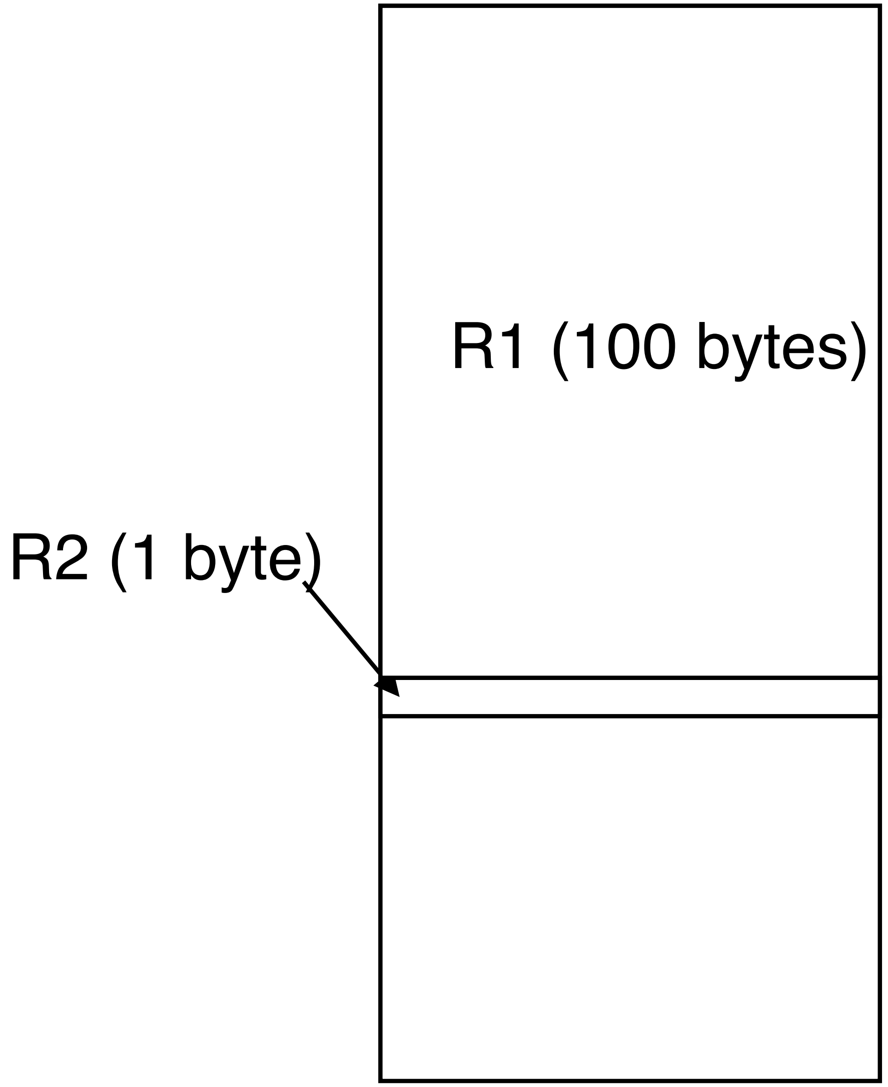
Heap Management Requirements

- Want `malloc()` and `free()` to run quickly
- Want minimal memory overhead
- Want to avoid fragmentation* – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

* This is technically called external fragmention

Heap Management

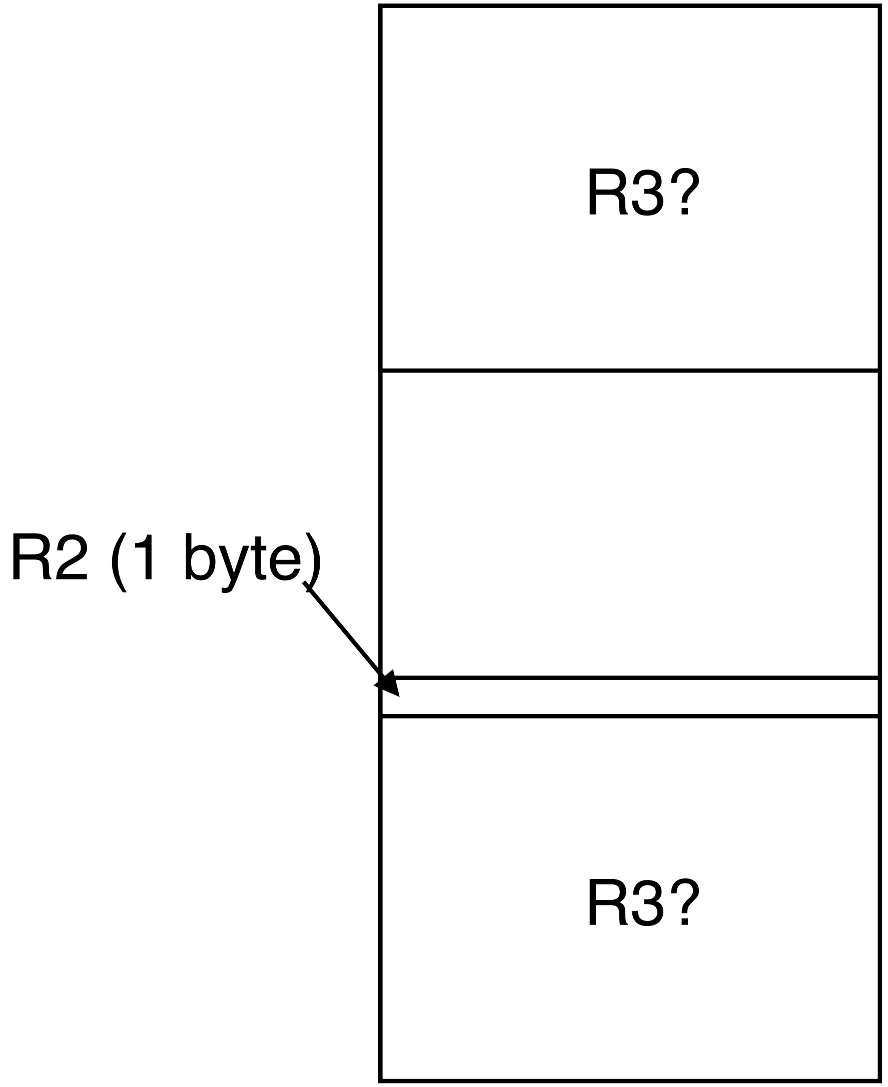
- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



Garcia, Nikolić

Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



Garcia, Nikolić

K&R Malloc/Free Implementation

- From Section 8.7 of K&R
  - Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code
- Each block of memory is preceded by a header that has two fields:
  size of the block and
  a pointer to the next block
- All free blocks are kept in a circular linked list, the pointer field is unused in an allocated block

K&R Implementation

- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can't satisfy the request, it fails.
- `free()` checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced) into a single, larger free block
  - Otherwise, freed block is just added to the free list

Garcia, Nikolić

Choosing a block in `malloc()`

- If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
  - best-fit: choose the smallest block that is big enough for the request
  - first-fit: choose the first block we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there

Garcia, Nikolić

And in conclusion…

- C has 3 pools of memory
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - The Heap (dynamic storage): malloc() grabs space from here, free() returns it.
- **`malloc()`** handles free space with freelist
- Three ways to find free space when given a request:
  - First fit (find first one that's free)
  - Next fit (same as first, but remembers where left off)
  - Best fit (finds most "snug" free space)

☑

When Memory Goes Bad

Pointers in C

- Why use pointers?
  - If we want to pass a huge struct or array, it's easier / faster / etc to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
  - Dangling reference (use ptr before malloc)
  - Memory leaks (tardy free, lose the ptr)

Writing off the end of arrays...

```
int *foo = (int *) malloc(sizeof(int) * 100);
int i;
....
for(i = 0; i <= 100; ++i) {
   foo[i] = 0;
}
```

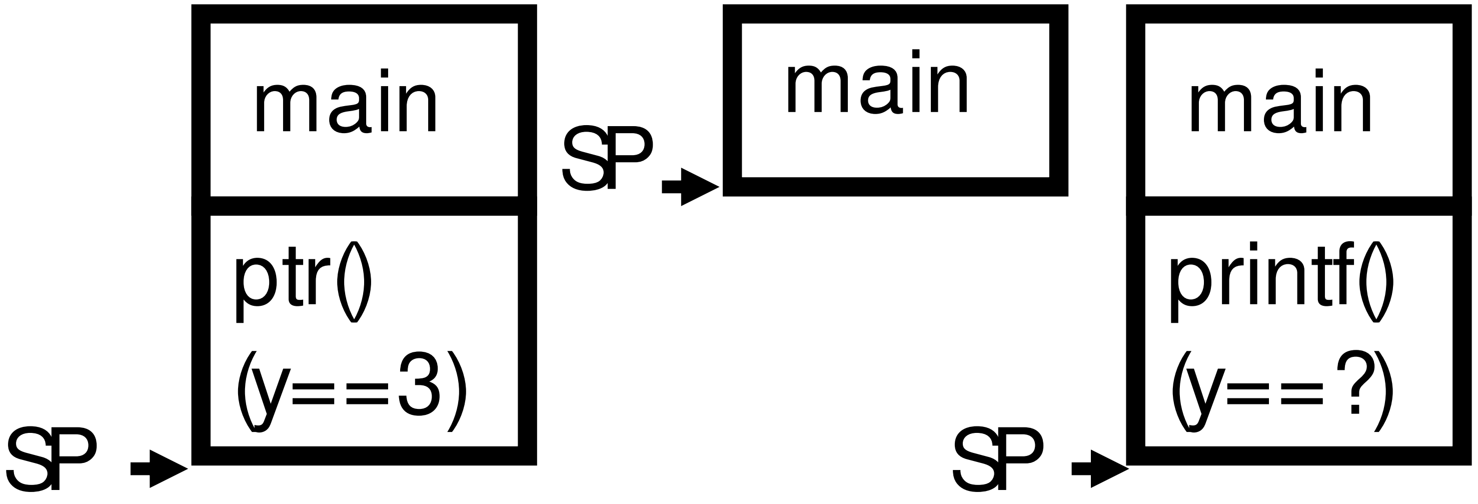
- Corrupts other parts of the program...
  - Including internal C data
- May cause crashes later

Returning Pointers into the Stack

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs !

```
int *ptr () {
    int y;
    y = 3;
    return &y;
};

main () {
    int *stackAddr, content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /*13451514 */
};
```



Garcia, Nikolić

Use After Free

- When you keep using a pointer..

```
struct foo *f
....
f = malloc(sizeof(struct foo));
....
free(f)
....
bar(f->a);
```

- Reads after the free may be corrupted
  - As something else takes over that memory. Your program will probably get wrong info!
- Writes corrupt other data!
  - Uh oh... Your program crashes later!

Garcia, Nikolić

Forgetting **realloc** Can Move Data...

- When you **realloc** it can copy data...
  - ```
    struct foo *f = malloc(sizeof(struct foo) * 10);
    ...
    struct foo *g = f;
    ....
    f = realloc(sizeof(struct foo) * 20);
    ```
- Result is **g** *may* now point to invalid memory
  - So reads may be corrupted and writes may corrupt other pieces of memory
    ```

Freeing the Wrong Stuff...

- If you **`free()`** something never **`malloc`**'ed()
  - Including things like

```
struct foo *f = malloc(sizeof(struct foo) * 10)
...
f++;
...
free(f)
```

- **`malloc`** or **`free`** may get confused..
  - Corrupt its internal storage or erase other data...

Double-Free...

- Eg.,

```
struct foo *f = (struct foo *)
   malloc(sizeof(struct foo) * 10);
...
free(f);
...
free(f);
```

- May cause either a use after **`free`** (because something else called **`malloc()`** and got that data) or corrupt **`malloc`**'s data (because you are no longer freeing a pointer called by **`malloc`**)

Garcia, Nikolić

Losing the initial pointer! (Memory Leak)

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
    plk++;
}
```

This MAY be a memory leak if we don't keep somewhere else a copy of the original malloc'ed pointer

Garcia, Nikolić

Valgrind to the rescue…

- Valgrind slows down your program by an order of magnitude, but...
  - It adds a tons of checks designed to catch most (but not all) memory errors
    - Memory leaks
    - Misuse of free
    - Writing over the end of arrays
- Tools like Valgrind are absolutely essential for debugging C code

And In Conclusion, …

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code

☑