CS 61C:
Great Ideas in Computer Architecture
Introduction to C, Part I

Instructor:
Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa13

Agenda

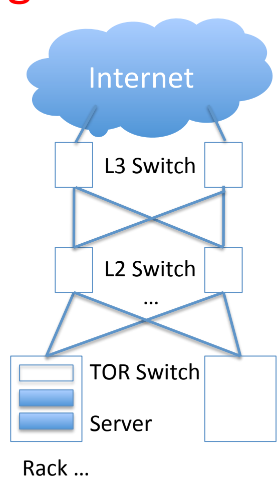
- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, …

Agenda

- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, …

WSC Case Study
Server Provisioning

WSC Power Capacity		8.00 MW
Power Usage Effectiveness (PUE)	1.50	
IT Equipment Power Share	0.67	5.36 MW
Power/Cooling Infrastructure	0.33	2.64 MW
IT Equipment Measured Peak (W)	145.00	
Assume Average Pwr @ 0.8 Peak	116.00	
# of Servers		46207
# of Servers		46000
# of Servers per Rack	40.00	
# of Racks		1150
Top of Rack Switches		1150
# of TOR Switch per L2 Switch	16.00	
# of L2 Switches		72
# of L2 Switches per L3 Switch	24.00	
# of L3 Switches		3



WSC Case Study
Capital Expenditure (Capex)

- Facility cost and total IT cost look about the same

Facility Cost	$88,000,000
Total Server Cost	$66,700,000
Total Network Cost	$12,810,000
Total Cost	$167,510,000

- However, replace servers every 3 years, networking gear every 4 years, and facility every 10 years

WSC Case Study
Operational Expense (Opex)

		Years Amortization		Monthly Cost	
Amortized Capital Expense	Server	3	$66,700,000	$2,000,000	55%
	Network	4	$12,530,000	$295,000	8%
	Facility		$88,000,000		
	Pwr&Cooling	10	$72,160,000	$625,000	17%
	Other	10	$15,840,000	$140,000	4%
Operational Expense	Amortized Cost			$3,060,000	
	Power (8MW)		$0.07	$475,000	13%
	People (3)			$85,000	2%
	Total Monthly	$/kWh		$3,620,000	100%

- $3.6M/46000 servers = ~$80 per month per server in revenue to break even
- ~$80/720 hours per month = $0.11 per hour
- So how does Amazon EC2 make money???

August 2013 AWS Instances & Prices

Instance	Per Hour	Ratio to Small	Compute Units	Virtual Cores	Compute Unit/ Core	Memory (GB)	Disk (GB)	Address
Standard Small	$0.065	1.0	1.0	1	1.00	1.7	160	32 bit
Standard Large	$0.260	4.0	4.0	2	2.00	7.5	840	64 bit
Standard Extra Large	$0.520	8.0	8.0	4	2.00	15.0	1680	64 bit
High-Memory Extra Large	$0.460	7.1	6.5	2	3.25	17.1	420	64 bit
High-Memory Double Extra Large	$0.920	14.2	13.0	4	3.25	34.2	850	64 bit
High-Memory Quadruple Extra Large	$1.840	28.3	26.0	8	3.25	68.4	1690	64 bit
High-CPU Medium	$0.165	2.5	5.0	2	2.50	1.7	350	32 bit
High-CPU Extra Large	$0.660	10.2	20.0	8	2.50	7.0	1690	64 bit
xxxxxxxxxxxxxxx	$X	15.3	33.5	16	2.09	23.0	1690	64 bit

- Closest computer in WSC example is Standard Extra Large
- @$0.11/hr, Amazon EC2 can make money!
  - even if used only 50% of time
  - See http://aws.amazon.com/ec2/pricing and http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html

Which statement is TRUE about Warehouse Scale Computer economics?

- ☐ The dominant operational monthly cost is server replacement.
- ☐ The dominant operational monthly cost is the electric bill.
- ☐ The dominant operational monthly cost is facility replacement.
- ☐ The dominant operational monthly cost is operator salaries.

Agenda

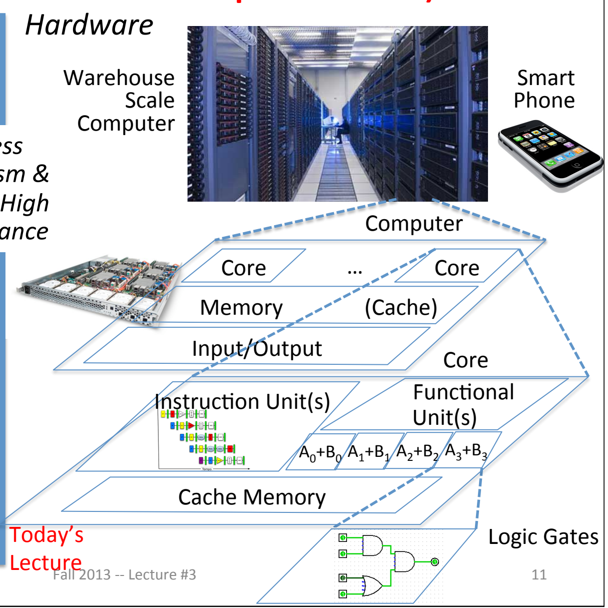
- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, …

New-School Machine Structures (It's a bit more complicated!)

Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



Great Idea: Levels of Representation/ Interpretation

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

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

We are here!

Compiler

Assembly Language Program (e.g., MIPS)

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

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

Assembler

Machine Language Program (MIPS)

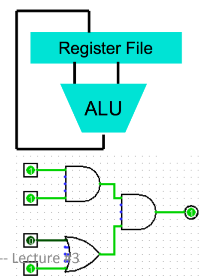
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to C "The Universal Assembly Language"

- "Some" experience is required before CS61C
  C++ or Java OK

- Class pre-req included classes teaching Java
- Java used in two labs and one project
- C used for everything else

Language Poll!

Please raise card for *first* one of following you can say yes to

- ☐ `I have programmed in C,C++,C#, or Objective-C`
- ☐ I have programmed in Java
- ☐ I have programmed in FORTRAN, Cobol, Algol-68, Ada, Pascal, or Basic
- ☐ None of the above

Disclaimer

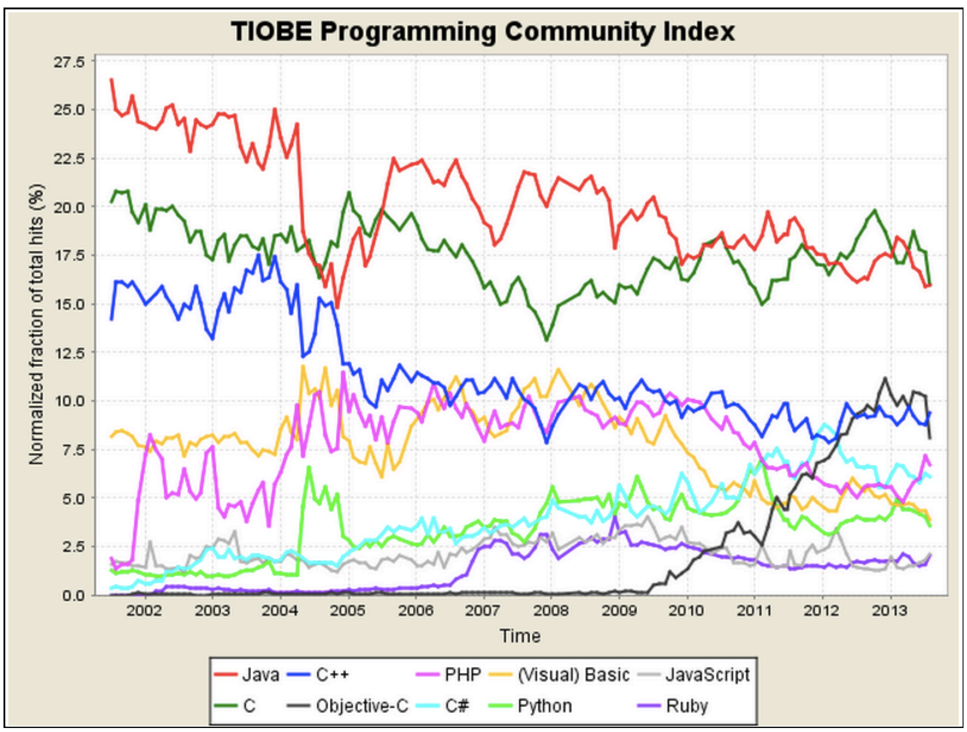
- You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course
  - K&R is a must-have
    - Check online for more sources
  - "JAVA in a Nutshell," O'Reilly
    - Chapter 2, "How Java Differs from C"
    - http://oreilly.com/catalog/javanut/excerpt/index.html
  - Brian Harvey's helpful transition notes
    - On CS61C class website: pages 3-19
    - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
- Key C concepts: Pointers, Arrays, Implications for Memory management

Intro to C

- *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
- Enabled first operating system not written in assembly language: *UNIX* - A portable OS!
- C and derivatives (C++/Obj-C/C#) still one of the most popular application programming languages after >40 years!

9/3/13 Fall 2013 -- Lecture #3 16



Basic C Concepts

Compiler	Creates useable programs from C source
Typed variables	Kind of data that a variable contains
Typed functions	The kind of data returned from a function
Header files (.h)	Declare functions and variables in a separate file
Structs	Groups of related values
Enums	Lists of predefined values
Pointers	Aliases to other variables

These concepts distinguish C from other languages you may know

Integers: Python vs. Java vs. C

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

- C: `int` should be integer type that target processor works with most efficiently
- Only guarantee: sizeof(`long long`) ≥ sizeof(`long`) ≥ sizeof(`int`) ≥ sizeof(`short`)
  – All could be 64 bits

C vs. Java

	C	Java
Type of Language	Function Oriented	Object Oriented
Program-ming 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("Hello\n");`
	`   return 0;`
	`}`
		`public class HelloWorld {`
		`   public static void main(String[] args) {`
		`      System.out.printl("Hello");`
		`   }`
		`}`

Storage	Manual (**`malloc`**, **`free`**)	Automatic (garbage collection)

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

C vs. Java

	C	Java
Comments	`/* … */`	`/* … */` or `// …` end of line
Constants	`const, #define`	`final`
Preprocessor	Yes	No
Variable declaration	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

Compilation: Overview

- C *compilers* map C programs into architecture-specific machine code (string of 1s and 0s)
  – Unlike *Java*, which converts to architecture-independent *bytecode*
  – Unlike *Python* environments, which *interpret* the code
  – 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

Compilation: Advantages

- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- Fair compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
- Why C?: *we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism*

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type and the operating system
- 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 the development cycle

Typed Variables in C

```
int   variable1   = 2;
float variable2   = 1.618;
char  variable3   = 'A';
```

- Must declare the type of data a variable will hold
  - Types can't change

Type	Description	Examples
int	integer numbers, including negatives	0, 78, -1400
unsigned int	integer numbers (no negatives)	0, 46, 900
float	floating point decimal numbers	0.0, 1.618, -1.4
char	single text character or symbol	'a', 'D', '?'
double	greater precision/big FP number	10E100
long	larger signed integer	6,000,000,000

Typed Functions in C

```
int number_of_people ()
{
  return 3;
}

float dollars_and_cents ()
{
  return 10.33;
}

char first_letter ()
{
  return 'A';
}
```

- 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 necessary to declare types for values passed into a function
- Variables and functions MUST be declared before they are used

Structs in C

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

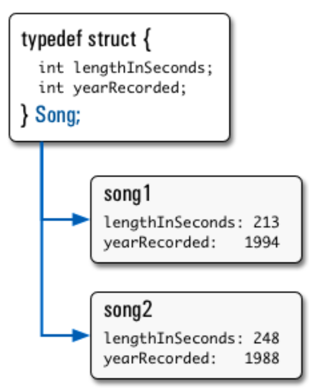
```
typedef struct {
  int length_in_seconds;
  int yearRecorded;
} Song;

Song song1;

Song1.length_in_seconds =  213;
song1.yearRecorded       = 1994;

Song song2;

Song2.length_in_seconds =  248;
song2.yearRecorded       = 1988;
```



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

Consts and Enums in C

- Constant is assigned a 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;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants used to parameterize libraries:

```
    enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
```


Which statement is TRUE regarding C and Java?

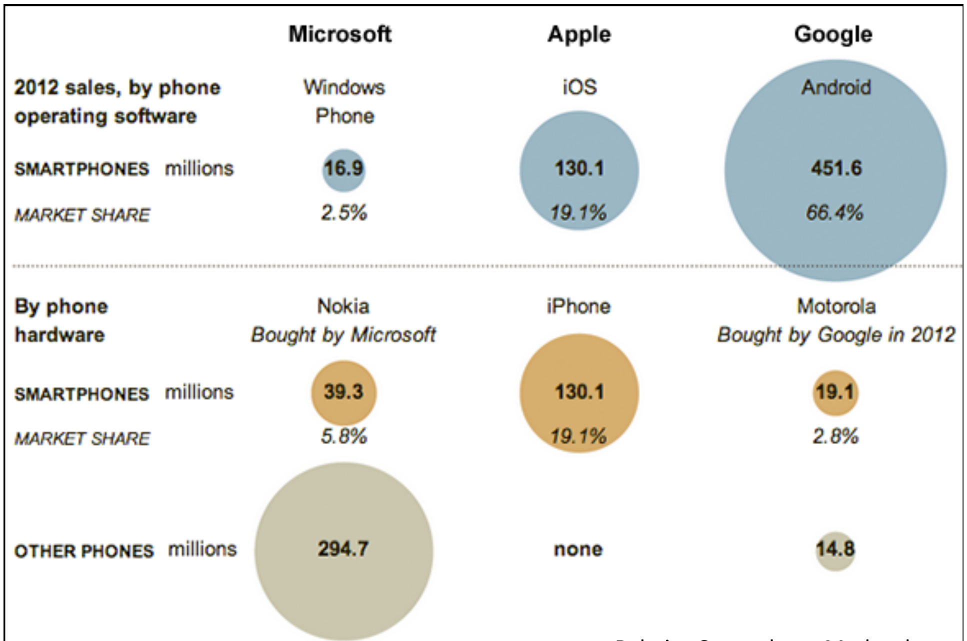
- ☐ `short`, `int`, and `long` are in both languages and they have the same meaning
- ☐ As Java was derived from C, it has the same names of data types
- ☐ C programs use compilers to produce executable code but Java does not
- ☐ C has a preprocessor that allows conditional compilation, but Java does not

Agenda

- WSC Economics (Highlights)
- Compile vs. Interpret
- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, …

Administrivia

- CS61c is relentless!
  – Next week: Lab #2, HW #2
  – Lab #2, Amazon EC2
  – HW #2 will soon be posted
- Your professor requests:
  – Need to leave early? Sit near aisles please …
  – Want to use computer, cell phone? Sit near back of auditorium …



Relative Smartphone Marketshare

9/4/13 Fall 2013 -- Lecture #3 33

Agenda

- WSC Economics (Highlights)
- Compile vs. Interpret
- Scheme vs. Java vs. C
- Administrivia
- **Quick Start Introduction to C**
- Technology Break
- Pointers
- Arrays
- And in Conclusion, …

9/3/13 Fall 2013 -- Lecture #3 34

A First C Program: Hello World

Original C:

```
main()
{
  printf("\nHello World\n");
}
```

ANSI Standard C:

```
#include <stdio.h>

int main(void)
{
  printf("\nHello World\n");
  return (0);
}
```


C Syntax: `main`

- When C program starts,
  - 1st runs job to set up computer
  - Then calls your procedure names 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 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)

Example

- `foo hello 87`
- `argc = 3 /* number arguments */`
- `argv[0] = "foo",`
  `argv[1] = "hello",`
  `argv[2] = "87"`
  - Array of pointers to strings (cover later)

A Second C Program: Compute Table of Sines

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    int    angle_degree;
    double angle_radian, pi, value;
    /* Print a header */
    printf("\nCompute a table of the
   sine function\n\n");

    /* obtain pi once for all        */
    /* or just use pi = M_PI, where */
    /* M_PI is defined in math.h    */
    pi = 4.0*atan(1.0);
    printf("Value of PI = %f \n\n",
   pi);
```

```
printf("angle     Sine \n");

angle_degree = 0;
/* initial angle value */
/* scan over angle     */
while (angle_degree <= 360)
/* loop until angle_degree > 360 */
   {
      angle_radian = pi*angle_degree/180.0;
      value = sin(angle_radian);
      printf (" %3d      %f \n ",
               angle_degree, value);
      angle_degree = angle_degree + 10;
      /* increment the loop index */
   }
}
```

Second C Program Sample Output

```
Compute a table of the sine
   function

Value of 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
 100      0.984808
 110      0.939693
 120      0.866025
 130      0.766044
 140      0.642788
 150      0.500000
 160      0.342020
 170      0.173648
 180      0.000000
 190     -0.173648
 200     -0.342020
 210     -0.500000
 220     -0.642788
 230     -0.766044
 240     -0.866025
 250     -0.939693
 260     -0.984808
 270     -1.000000
 280     -0.984808
 290     -0.939693
 300     -0.866025
 310     -0.766044
 320     -0.642788
 330     -0.500000
 340     -0.342020
 350     -0.173648
 360     -0.000000
```


C Syntax: Variable Declarations

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

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs (shows Java's legacy) in terms of control flow
  - **`if-else`**
    - `if (expression) statement`
    - `if (expression) statement1`
      `else statement2`
  - **`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`
    ```

C Syntax: True or False

- What evaluates to FALSE in C?
  – 0 (integer)
  – NULL (a special kind of *pointer*: more on this later)
  – *No explicit Boolean type*
- What evaluates to TRUE in C?
  – Anything that isn't false is true
  – Same idea as in Python: only 0s or empty sequences are false, anything else is true!

C and Java 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: ., ->
- conditional evaluation: ? :

Agenda

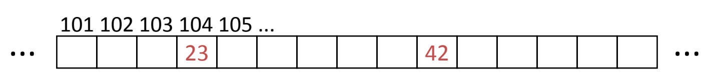
- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, …

Agenda

- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, …

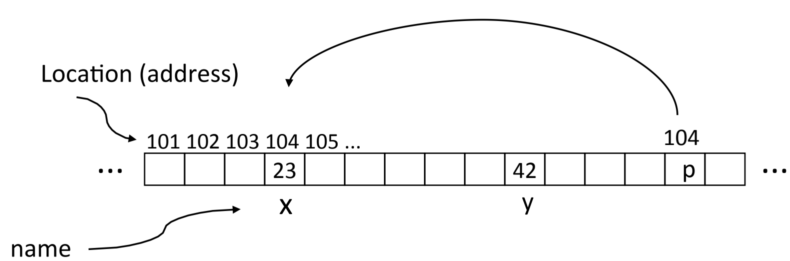
Address vs. Value

- 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 there



Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable



Pointer Syntax

- `int *x;`
  - Tells compiler that variable x is address of an `int`
- `x = &y;`
  - Tells compiler to assign address of `y` to `x`
  - `&` called the "address operator" in this context
- `z = *x;`
  - Tells compiler to assign value at address in `x` to `z`
  - `*` called the "dereference operator" in this context

Creating and Using Pointers

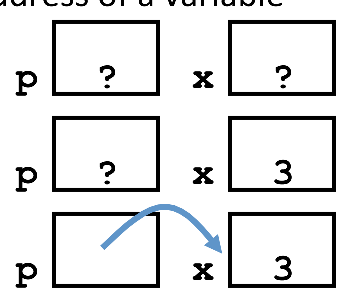
- How to create a pointer:

& 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**.

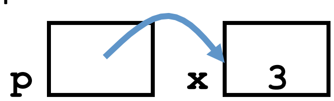
- How get a value pointed to?

"*" (dereference operator): get the value that the pointer points to

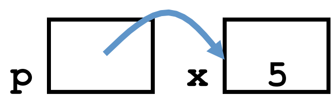
```
printf("p points to %d\n",*p);
```

Using Pointer for Writes

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



***p = 5;**



Pointers and Parameter Passing

- 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 remains equal to 3

Pointers and Parameter Passing

- How can we get a function to change the value held in a variable?

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

addOne(&y);
```

y is now equal to 4

Types of Pointers

- Pointers are used to point to any kind of data (**`int`**, **`char`**, a **`struct`**, etc.)
- Normally a pointer only points 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 other bad things!

More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!
- Local variables in C are not initialized, they may contain anything (aka "garbage")
- What does the following code do?

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


Pointers and Structures

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

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;
```

Pointers in C

- 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
  - 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—which you will to know by the end of the semester, but not for the projects (there will be a lab later in the semester)
    - *Dangling references* and *memory leaks*

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn't produce efficient code
  - Computers 25,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today's compilers produce much better code, so may not need to use pointers
  - Compilers even ignore hints since they do it better!

How many logic and syntax errors?

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ ≥4

```
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);
}
flip-sign(int *n){*n = -(*n)}
```

9/3/13 Fall 2013 -- Lecture #3 59

What is output after correct errors?

- ☐ x=5,y=6,p=-5
- ☐ x=-5,y=6,p=-5
- ☐ x=-5,y=4,p=-5
- ☐ x=-5,y=-6,p=-5

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

9/3/13 Fall 2013 -- Lecture #3 61

Arrays (1/5)

- Declaration:
  `int ar[2];`
  declares a 2-element integer array: just a block of memory

  `int ar[] = {795, 635};`
  declares and initializes a 2-element integer array

- Accessing elements:
  `ar[num]`
  returns the numth element

Arrays (2/5)

- C arrays are (almost) identical to pointers
  - **`char *string`** and **`char string[]`** are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
  - End of C string marked by 0 in last character
- *Key Concept*: Array variable is a "pointer" to the first (0th) element

C Strings

- String in C is just an array of characters

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (aka "null terminator")

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```


Arrays (3/5)

- Consequences:
  - **ar** is an array variable, but looks like a pointer
  - **ar[0]** is the same as ***ar**
  - **ar[2]** is the same as ***(ar+2)**
  - We can use pointer arithmetic to conveniently access arrays
- Declared arrays are only allocated while the scope is valid

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

is incorrect ***and very very bad***

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 & incrementation
  - Bad pattern
    ```
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  - Better pattern
    ```
    int ARRAY_SIZE = 10
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```
- SINGLE SOURCE OF TRUTH
  - You're utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: "Don't Repeat Yourself"

Arrays (5/5)

- Pitfall: An array in C does not know its own length, and its bounds are not checked!
  - Consequence: We can accidentally access off the end of an array
  - Consequence: We must pass the array *and its size* to any procedure that is going to manipulate it
- Segmentation faults and bus errors:
  - These are VERY difficult to find;
    be careful! (You'll learn how to debug these in lab)

Array Summary

- Array indexing is syntactic sugar for pointers
- **a[i]** is treated as ***(a+i)**
- E.g., three equivalent ways to zero an array:
  - `for (i=0; i < size; i++) a[i] = 0;`
  - `for (i=0; i < size; i++) *(a+i) = 0;`
  - `for (p=a; p < a+size; p++) *p = 0;`

Incrementing a pointer makes it point to the next variable in memory (type of pointer says how big each variable is)

What is TRUE about this function?

```
void foo(char *s, char *t)
{ while (*s)
      s++;
  while (*s++ = *t++)
      ;
}
```

- ☐ It has syntax errors
- ☐ No syntax errors; it changes characters in string `t` to next character in the string `s`
- ☐ No syntax errors; it copies a string at address `t` to the string at address `s`
- ☐ No syntax errors; it appends the string at address `t` to the end of the string at address `s`

Question: Which statement is FALSE regarding C and Java?

- ☐ Arrays in C are just pointers to the 0-th element
- ☐ As Java was derived from C, it has the same control flow constructs
- ☐ Like Java, in C you can check the length of an array (`a.length` gives no. elements in `a`)
- ☐ C has pointers but Java does not allow you to manipulate pointers or memory addresses of any kind

FYI—Update to ANSI C

- "C99" or "C9X" standard
  - `gcc -std=c99` to compile
- 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
  - Java-like // comments (to end of line)
  - Variable-length non-global arrays
  - `<inttypes.h>`: explicit integer types
  - `<stdbool.h>`: for boolean logic types and definitions

And In Conclusion, …

- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it
- Pointer is a C version (abstraction) of a data address
  - `*` "follows" a pointer to its value
  - `&` gets the address of a value
  - Arrays and strings are implemented as variations on pointers
- C is an efficient language, but leaves safety to the programmer
  - Array bounds not checked
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs