211: Computer Architecture
Lecture 2

Topic:

- C Programming

Introduction to C

TAs will cover C in more details in sections

- Will also help you with machine/compilation logistics

Learning C

- Is no big deal; you already know Java
- Start by coding and testing small programs
- Learn how to use a debugger!
  - TAs will help

Why Learn C?

You are learning to be a computer scientist

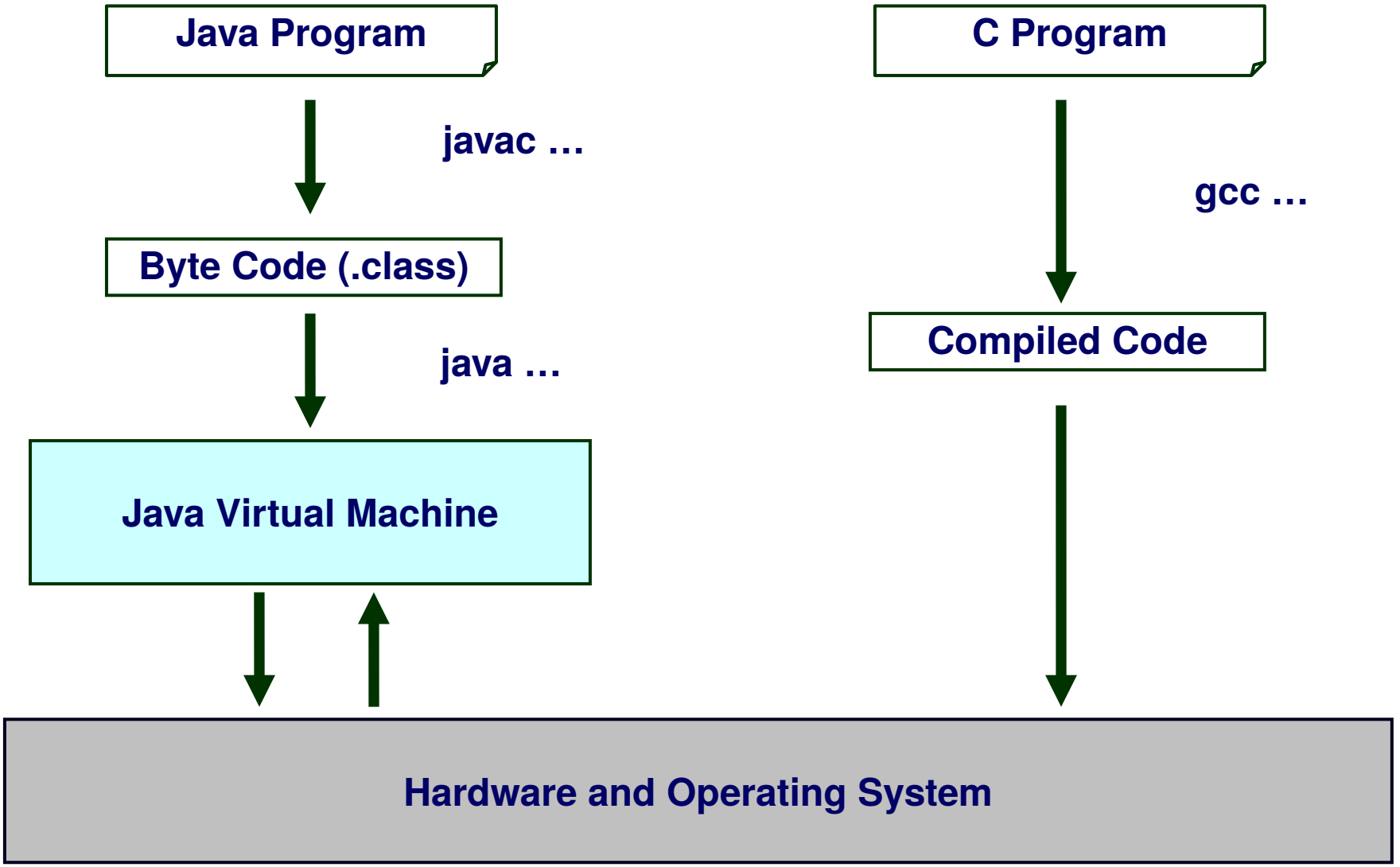
- Languages are just tools
- Choose tool appropriate to the task

Current task: learning computer architecture and how programs written in high-level language runs on computers

- C closer to machine so easier to see mapping

It's fun

Comparison with Java



Anatomy of a C Program

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

char cMessage[] = "Hello\n";

/* Execution will start here */
int main (int argc, char **argv)
{
    int i, count;

    count = atoi(argv[1]);
    for (i = 0; i < count; i++) {
        printf("Hello %d\n", i);
    }
}
```

include files

declaration of global variables

comment

one or more function; each program starts execution at “main”

declaration of local variables

code implementing function

Comments

Begins with /* and ends with */

Can span multiple lines.

Cannot have a comment within a comment or string

- Example:

  “my/*don't print this*/string”

- would be printed as:

  my/*don't print this*/string

Comments are critical

- How much and where is an art

Variable Declarations

Variables are used as names for data items

Each variable has a type, which tells the compiler how the data is to be interpreted (and how much space it needs, etc.)

```
int counter;
int startPoint;
```

Variables can be global or local

global: declare outside scope of any function
accessible from anywhere

local: declare inside scope of a function
accessible only from inside of the function

Basic Data Types

Keyword	Data Type	Examples
char	individual characters	‘a’, ‘b’, ‘\t’, ‘\n’
int	integers	-15, 0, 35
float	real numbers	-23.6, 0, 4.56
double	real numbers with double precision	-23.6, 0, 4.56

Modifiers

- short, long: control size/range of numbers
- signed, unsigned: include negative numbers or not

Arithmetic Operators

Symbol	Operation	Usage	Assoc
*	multiply	x * y	l-to-r
/	divide	x / y	l-to-r
%	modulo	x % y	l-to-r
+	addition	x + y	l-to-r
-	subtract	x - y	l-to-r

* / % have higher precedence than + -

Rule of thumb: remember only a few precedence rules

Use () for everything else

Special Operators: ++ and --

Changes value of variable before (or after) its value is used in an expression

Symbol	Operation	Usage
++	postincrement	x++
--	postdecrement	x--
++	preincrement	++x
--	predecrement	--x

Pre: Increment/decrement variable before using its value

Post: Increment/decrement variable after using its value

Be careful when using these operators!

Relational Operators

Symbol	Operation	Usage
>	greater than	x > y
>=	greater than or equal	x >= y
<	less than	x < y
<=	less than or equal	x <= y
==	equal	x == y
!=	not equal	x != y

Result is 1 (TRUE) or 0 (FALSE)

Don't confuse equality (==) with assignment (=)

Logic Operators

Symbol	Operation	Usage	Assoc
!	logical NOT	!x	r-to-l
&&	logical AND	x && y	l-to-r
||	logical OR	x || y	l-to-r

Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero)

Result is 1 (TRUE) or 0 (FALSE)

Bit Operators

Symbol	Operation	Usage	Assoc
~	complement	~x	r-to-l
&	bit AND	x & y	l-to-r
|	bit OR	x | y	l-to-r

Operate on bits of variables or constants

For example:

- ~0101 = 1010
- 0101 & 1010 = 0000
- 0101 | 1010 = 1111

Expressions and Assignments

Expression = “a computation” with a result

- (x + y) * z
- Be careful of type conversion!

  int x, z; float y;

  the result of the expression (x + y) * z will have what type?

Assignment

- x = (x + y) * z;
- The assignment statement itself is an expression and has a value. In this case, it’s the value assigned to x.

Control Statements

Conditional

- if else
- switch

Iteration (loops)

- while
- for
- do while

Specialized “go-to”

- break
- continue

The if Statement

```
if (expression-1)
    {statements-1}
else if (expression-2)
    {statements-2}
// ...
else if (expression-n-1)
    {statements-n-1}
else
    {statements-n}
```

Evaluates expressions until find one with non-zero result

- executes corresponding statements

If all expressions evaluate to zero, executes statements for "else" branch

The switch Statement

```
switch(expression) {
  case const-1: statements-1;
  case const-2: statements-2;
  default: statements-n;
}
```

Evaluates expression; results must be integer

Finds 1st “case” with matching contant

- Executes corresponding statements
- Continue executing until encounter a break or end of switch statement

“default” always matches

The switch Statement (Example)

```
int fork;
…
switch(fork) {
     case 1:
         printf(“take left’’);
     case 2:
         printf(“take right”);
         break;
     case 3:
         printf(“make U turn”);
         break;
     default:
         printf(“go straight”);
}
```

Loops

Statement	Repeats set of statements
while (expression) {…}	zero or more times, while expression ≠ 0, compute expression before each iteration
do {...} while (expression)	one or more times, while expression ≠ 0 compute expression after each iteration
for (start-expression; cond-expression; update-expression) {…}	zero or more times while cond-expression ≠ 0 compute start-expression before 1st iteration compute update-expression after each iteration

Specialized Go-to's

break

- Force immediate exit from switch or loop
- Go-to statement immediately following switch/loop

continue

- Skip the rest of the computation in the current iteration of a loop
- Go-to evaluation of conditional expression for execution of next iteration

Specialized Go-to's (Example)

What does the following piece of code do?

```
int index = 0;
int sum = 0;
while ((index >= 0) && (index <= 20))
{
    index += 1;
    if (index == 11) break;
    if ((index % 2) == 1) continue;
    sum = sum + index;
}
```

Functions

Similar to Java methods

Components:

- Name
- Return type
  - void if no return value
- Parameters
  - pass-by-value
- Body
  - Statements to be executed
  - return forces exits from function and resumes execution at statement immediately after function call

```
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

Function Calls

Function call as part of an expression

- x + Factorial(y)
- Arguments evaluated before function call
  - Multiple arguments: no defined order or evaluation
- Returned value is used to compute expression
- Cannot have a void return type

Function call as a statement

- Factorial(y);
- Can have a void return type
- Returned value is discarded (if there is one)

Function Prototypes

Can declare functions without specifying implementation

- int Factorial(int)
  - Can specify parameter names but don't have to
  - This is called a function signature

Declarations allow functions to be "used" without having the implementation until link time (we'll talk about linking later)

- Separate compilation
  - Functions implemented in different files
  - Functions in binary libraries
- Signatures are often given in header files
  - E.g., stdio.h gives the signatures for standard I/O functions

Input and Output

Variety of I/O functions in C Standard Library

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

```
printf("%d\n", counter);
```

- String contains characters to print and formatting directives for variables
- This call says to print the variable counter as a decimal integer, followed by a linefeed (\n)

```
scanf("%d", &startPoint);
```

- String contains formatting directives for parsing input
- This call says to read a decimal integer and assign it to the variable startPoint. (Don't worry about the & yet.)

Input and Output

Variety of I/O functions in C Standard Library

#include <stdio.h>

sprintf(s, "%d\n", counter);

- This call says to produce a string much like printf, but to store the result in s instead of printing it to stdout.

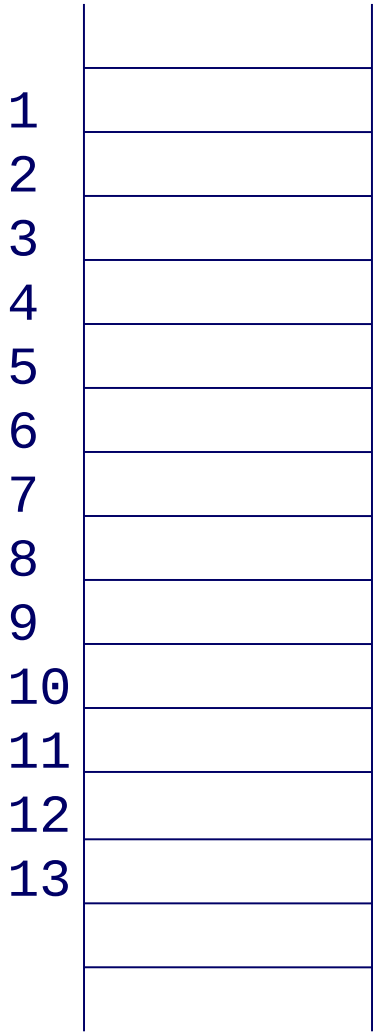
atoi(s);

- This call says to convert a string representation of an integer (e.g., “42”) to the integer value (42).
- What happens if it the string isn’t a valid int?

Memory

C's memory model matches the underlying (virtual) memory system

- Array of addressable bytes



Memory

C's memory model matches the underlying (virtual) memory system

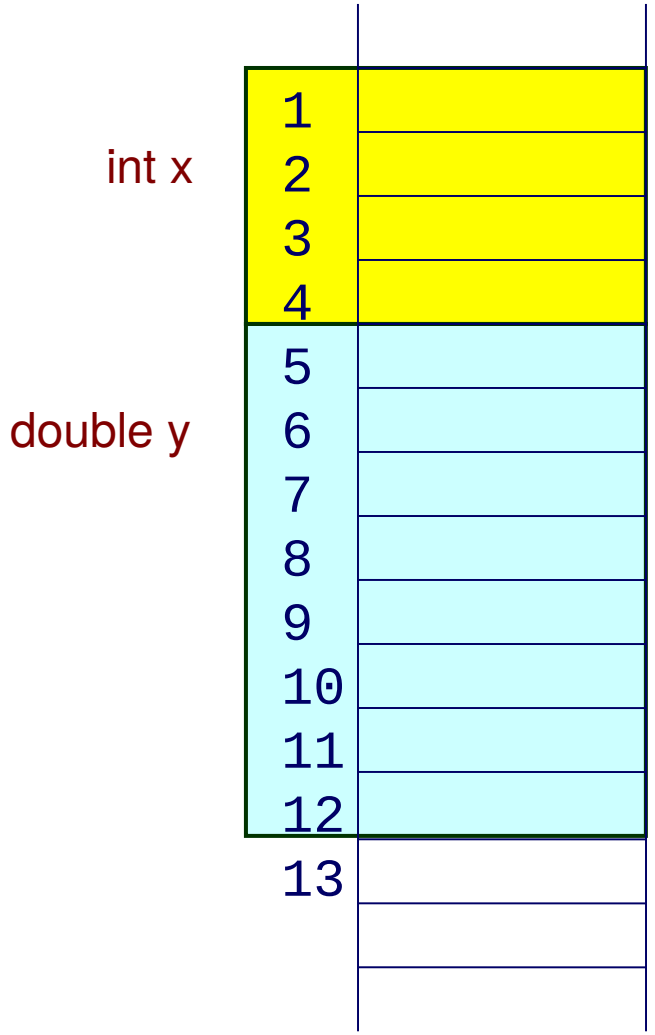
- Array of addressable bytes

Variables are simply names for contiguous sequences of bytes

- Number of bytes given by type of variable

Compiler translates names to addresses

- Typically maps to smallest address
- Will discuss in more detail later



Pointers

A pointer is just an address

Can have variables of type pointer

- Hold addresses as values
- Used for indirection

When declaring a pointer variable, need to declare the type of the data item the pointer will point to

- int *p; /* p will point to a int data item */

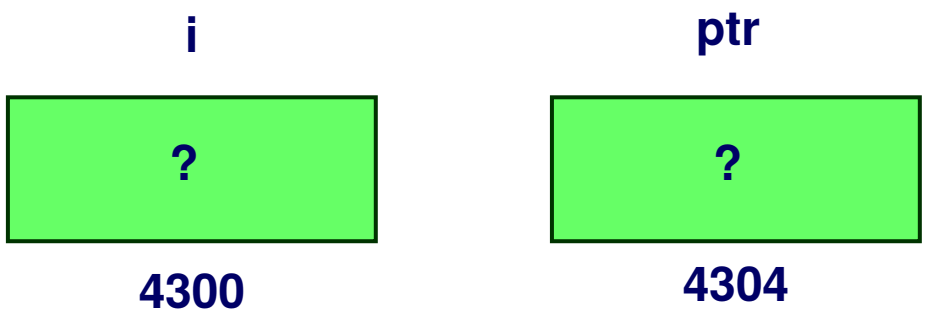
Pointer operators

- De-reference: *
  - *p gives the value stored at the address pointed to by p
- Address: &
  - &v gives the address of the variable v

Pointer Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

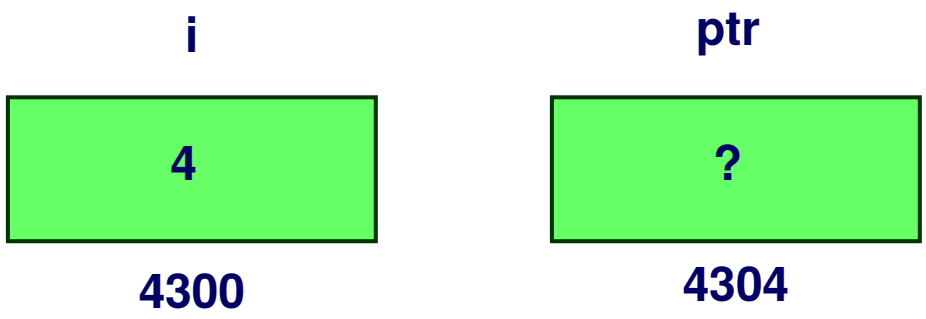


Pointer Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the value 4 into the memory location associated with i

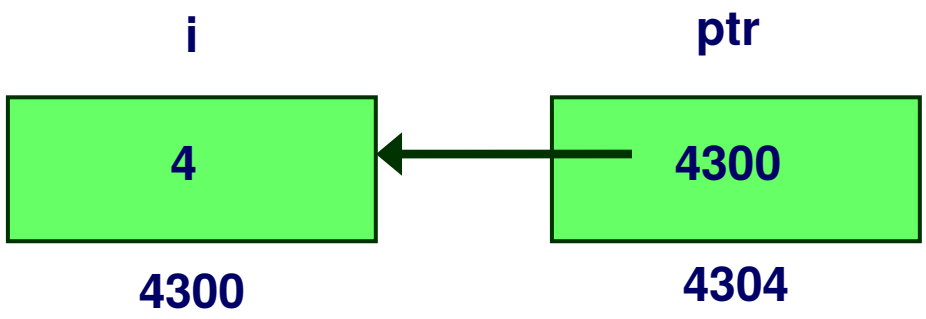


Pointer Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the address of i into the memory location associated with ptr



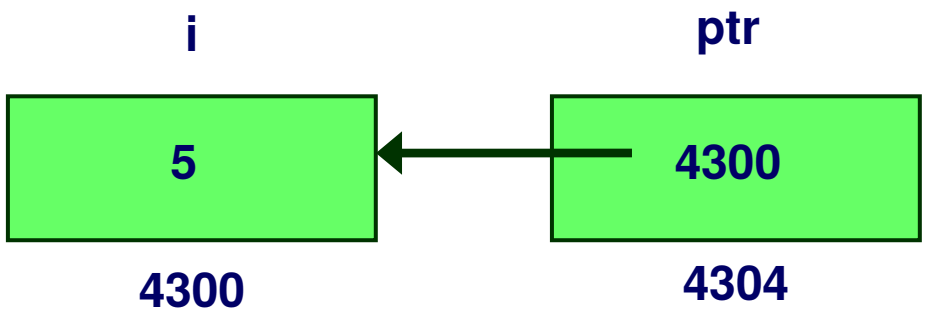
Pointer Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the result into memory at the address stored in ptr

read the contents of memory at the address stored in ptr



Example Use of Pointers

What does the following code produce? Why?

```
void Swap(int firstVal, int secondVal)
{
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(fv, sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Parameter Pass-by-Reference

Now what does the code produce? Why?

```
void Swap(int *firstVal, int *secondVal)
{
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(&fv, &sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Null Pointer

Sometimes we want a pointer that points to nothing

In other words, we declare a pointer, but we're not ready to actually point to something yet

```
int *p;
p = NULL;  /* p is a null pointer */
```

NULL is a predefined constant that contains a value that a non-null pointer should never hold

- Often, NULL = 0, because address 0 is not a legal address for most programs on most platforms

Type Casting

C is NOT strongly typed

Type casting allows programmers to dynamically change the type of a data item

Arrays

Arrays are contiguous sequences of data items

- All data items are of the same type
- Declaration of an array of integers: “int a[20];”
- Access of an array item: “a[15]”

Array index always start at 0

The C compiler and runtime system do not check array boundaries

- The compiler will happily let you do the following:
  - int a[10]; a[11] = 5;