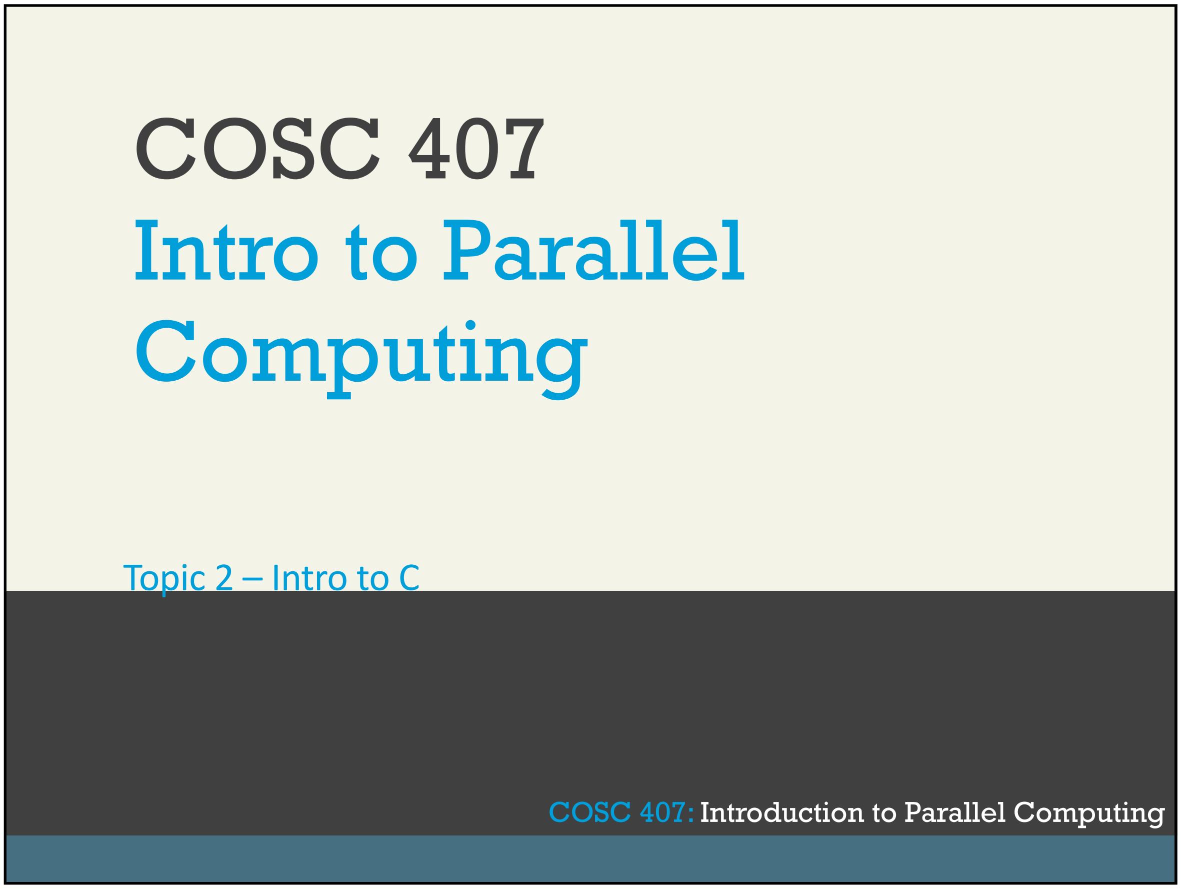
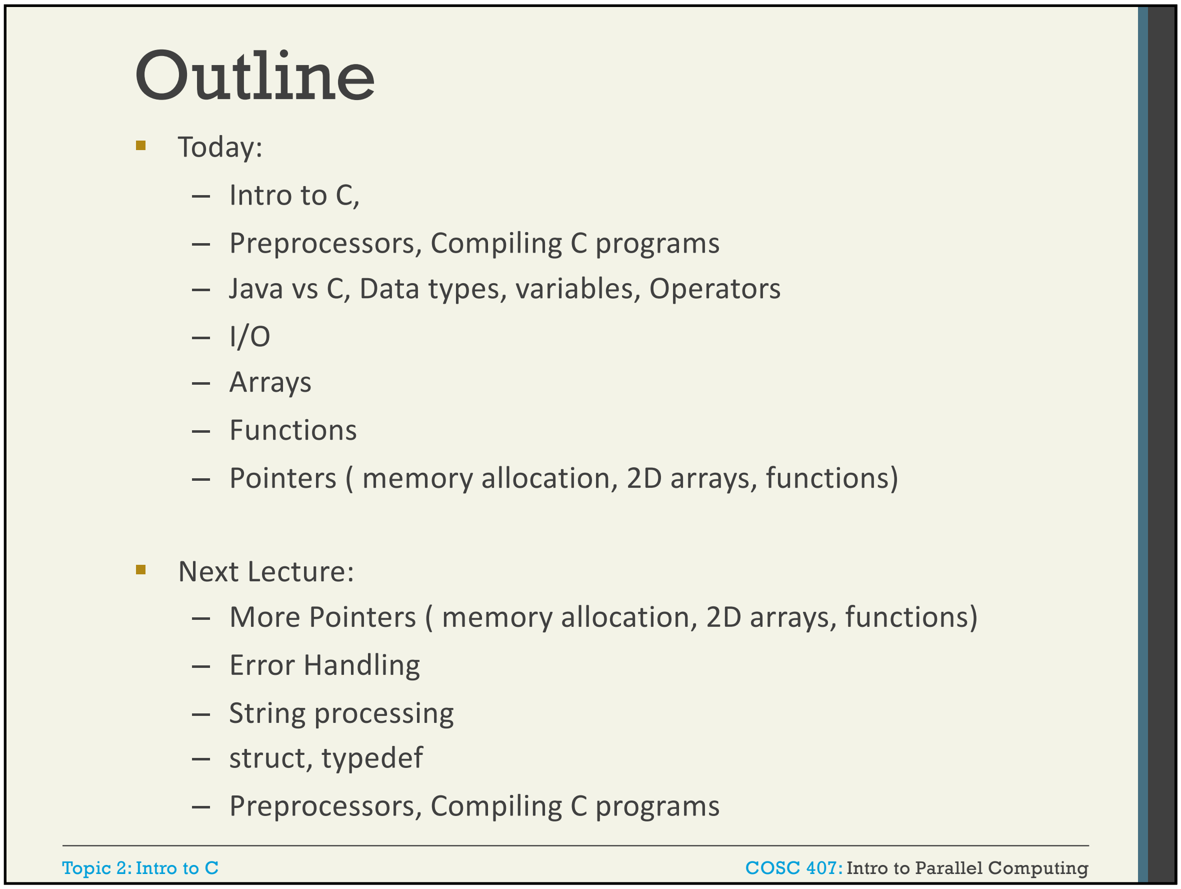


COSC 407
Intro to Parallel Computing

Topic 2 – Intro to C

COSC 407: Introduction to Parallel Computing



Outline

- Today:
  - Intro to C,
  - Preprocessors, Compiling C programs
  - Java vs C, Data types, variables, Operators
  - I/O
  - Arrays
  - Functions
  - Pointers (memory allocation, 2D arrays, functions)
- Next Lecture:
  - More Pointers (memory allocation, 2D arrays, functions)
  - Error Handling
  - String processing
  - struct, typedef
  - Preprocessors, Compiling C programs

Poll – I C….

- How do you feel about C (are you c-faring)?
  - A. I am already fluent with C
  - B. I am ok with with C
  - C. I don’t know much about C
  - D. C?

Objectives

- Getting started
- Intro to C
  - This is not an "Intro to C programming" course
    - Will cover some of the challenging concepts
    - You must know the basics of writing C code
      - Parallel programming will be done in C
  - You will reuse what you already know (Java/Python/something else…).
    - Not start from scratch
- Pointing out where to find information
- Pointing out common mistakes

C & the Impact of Dennis Ritchie

A "*simple*" procedural programming language developed by Dennis Ritchie while at Bell Labs in the 1970's

> *"…without Dennis there would be no C programming language. Without C there would be no C++. That means there won't be any Windows, Unix, Linux, Firefox, Photoshop, After Effects, Fl Studio, VLC Media Player, Microsoft Words, Excel. It also means no PC Games, Xbox Games, Playstation Games. No Black Ops, Halo, Warcraft, Grand Theft Auto. The list is endless. 90% of computer applications in the world are written in C and C++.*
>
> *Without UNIX, there would be no internet. That means no Google, Facebook, Youtube, Wikipedia, etc.*
>
> *Apple, Microsoft, and most other computer companies would be nothing without him."*

Source: http://www.likeyou.me/index.php/tech/steve-jobs-vs-dennis-ritchie/994/

Dennis MacAlistair Ritchie (b. September 9, 1941; d.October 12, 2012)

C

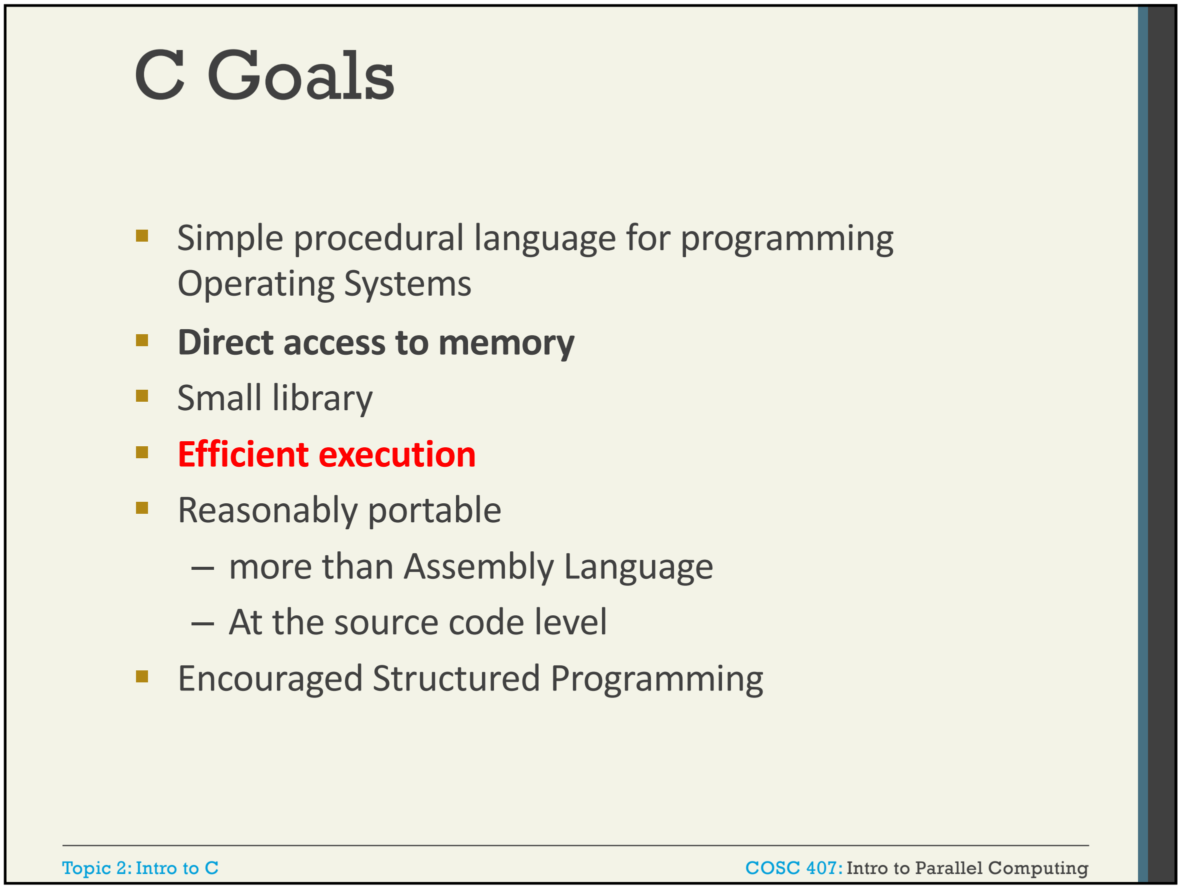
- There are many books and tutorials available,
  - The C Programming language, B. W. Kernighan and Dennis Ritchie, Prentice-Hall (K&R C)
  - C Programming: a modern approach, K.N. King, http://knking.com/books/c/
  - Many online tutorials
    - http://www.tutorialspoint.com/cprogramming
    - http://www.cprogramming.com/tutorial/c-tutorial.html
- IDEs and compilers
  - GCC (GNU Compiler Collection)
  - Eclipse for IDE for C/C++ Developers
  - Some online IDEs and compilers
    - https://www.onlinegdb.com/online_c_compiler
    - https://replit.com/languages/c

Software

- The Labs are equipped with the required software
- If you use your laptop, the following list is required
  - **An editor**
    - Eclipse for Scientific Computing
    - or your preferred IDE
    - or you can even use command line + an editor of some sort
  - **C compiler**
    - Windows - MinGW (Minimalist GNU for Windows)
      - http://www.mingw.org
      - Requires configuration
        - » See Canvas for instructions.
    - macOS – Apple command line tools
    - Linux – ☺
  - More software will be needed as we proceed, but not today…

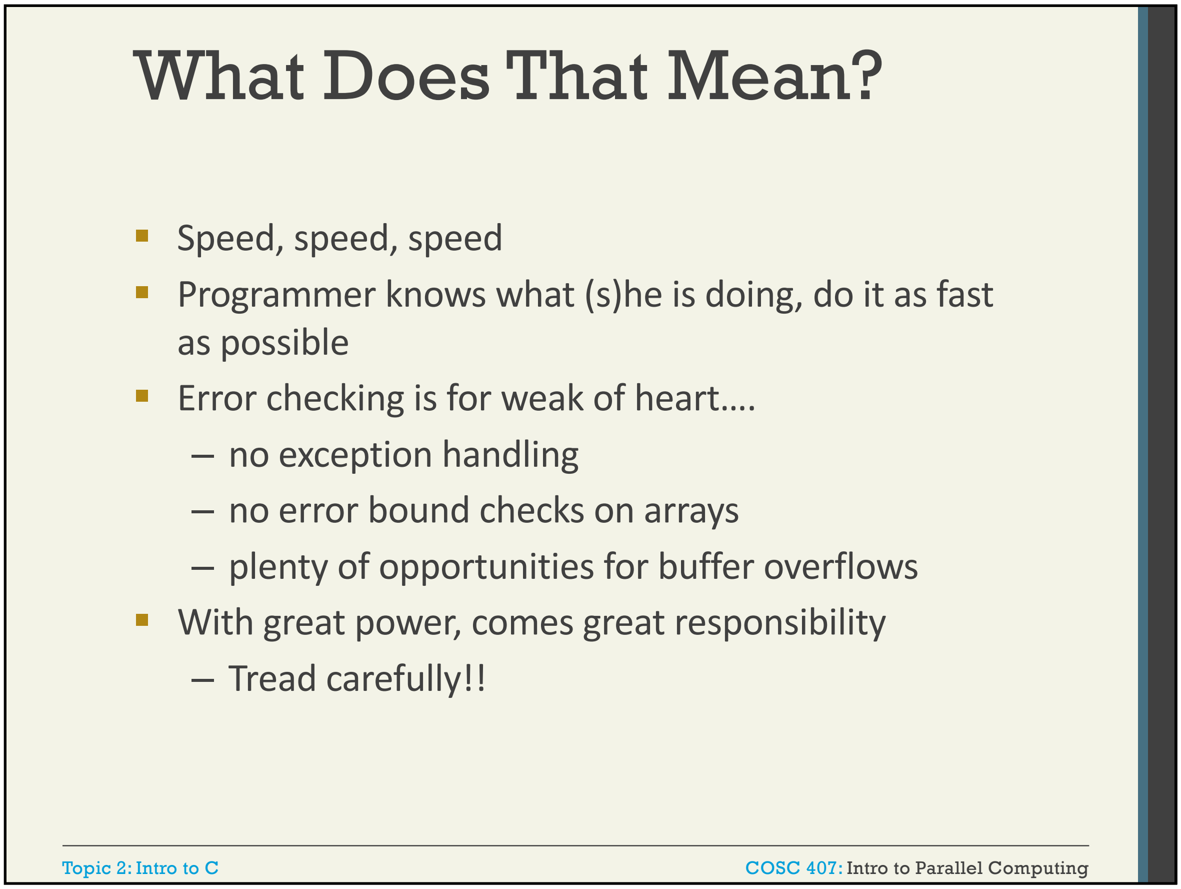
Why C?

- **Paradigm**: C is not object-oriented. Methods are called functions and procedures (but C++ is OO)
- **Program organization**: A program in C contains a collection of source and header files. The header contains function protocols and should be included if you want to use a specific function defined in that module.
  - Small and efficient libraries
- **Memory Management**: C does not have the garbage-collection mechanism
- **Performance**: C is faster than Java
- **Lower-level language** is needed to get closer to the hardware
- **Java hides many details**
  - Memory management
  - Initializations
  - Buffer management, array bounds



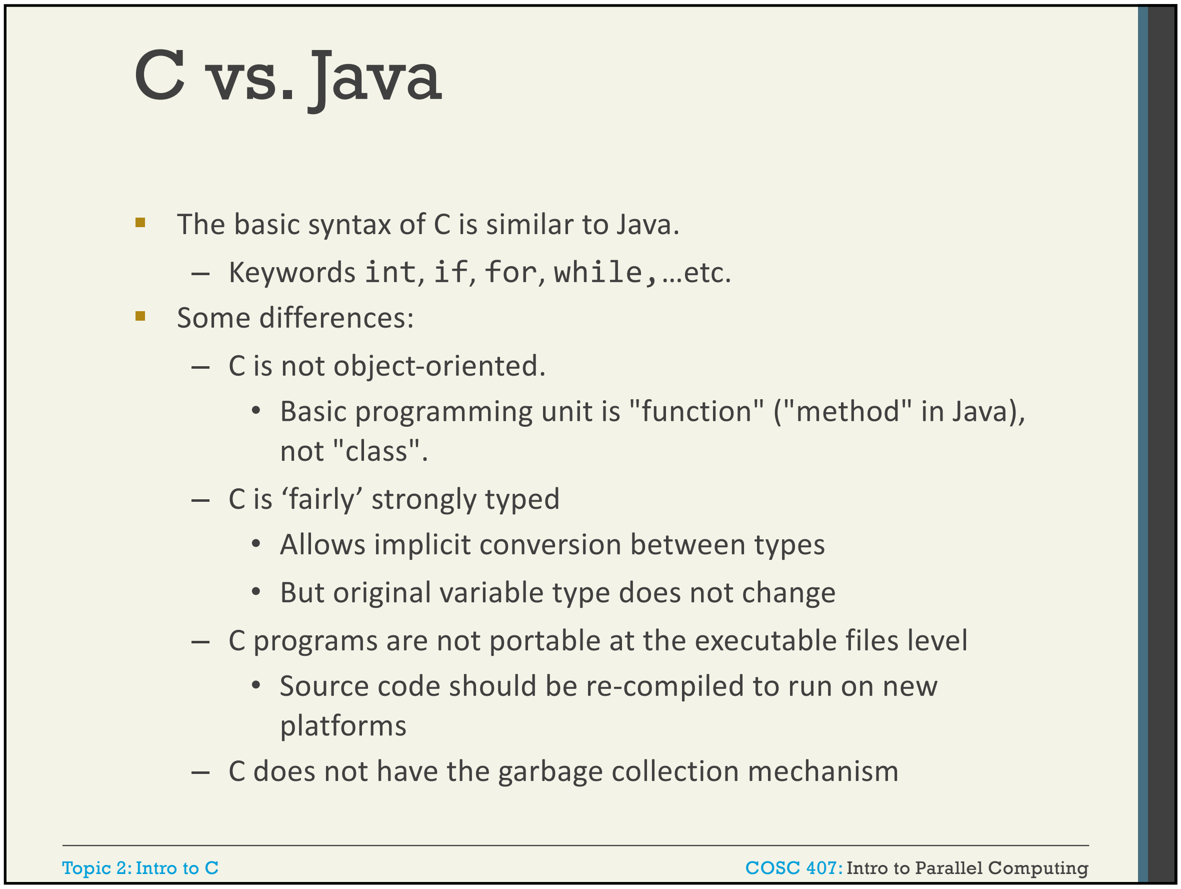
C Goals

- Simple procedural language for programming Operating Systems
- **Direct access to memory**
- Small library
- **Efficient execution**
- Reasonably portable
  - more than Assembly Language
  - At the source code level
- Encouraged Structured Programming



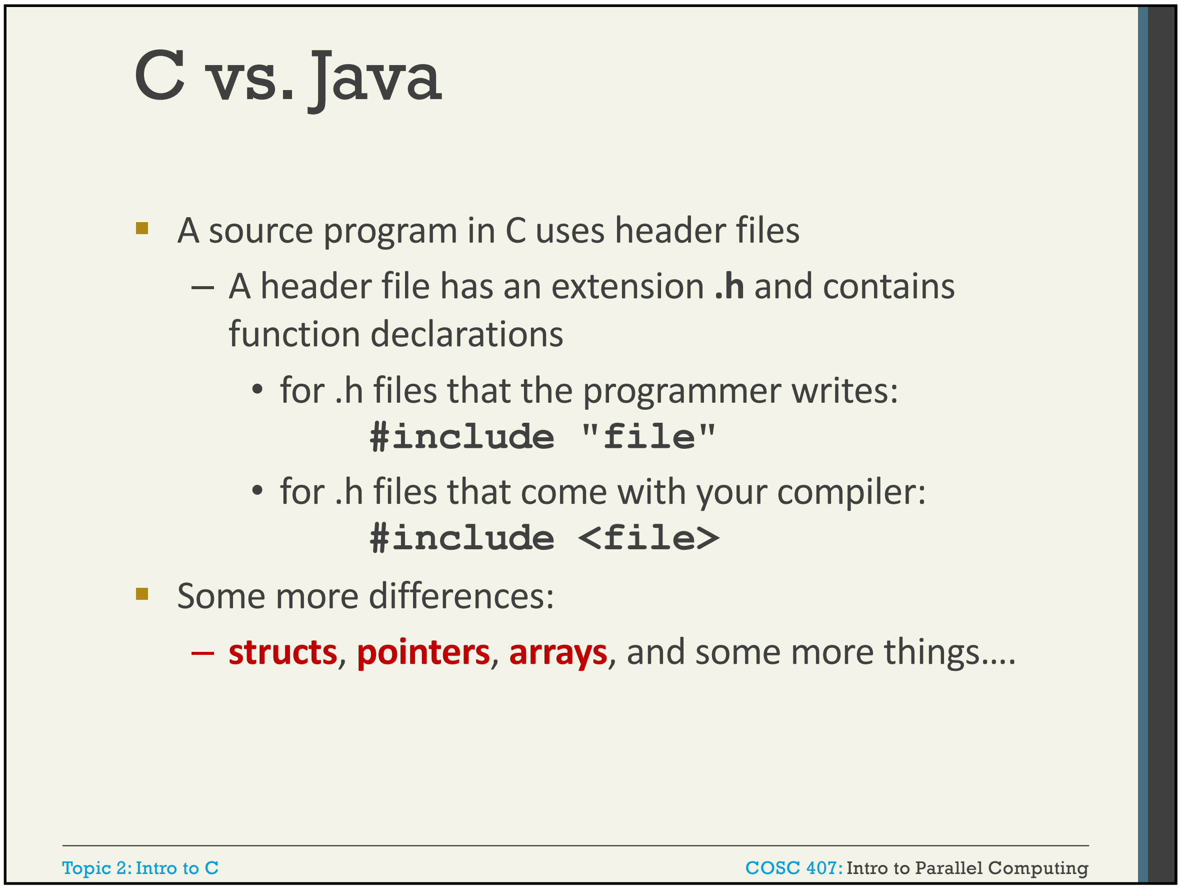
What Does That Mean?

- Speed, speed, speed
- Programmer knows what (s)he is doing, do it as fast as possible
- Error checking is for weak of heart….
  - no exception handling
  - no error bound checks on arrays
  - plenty of opportunities for buffer overflows
- With great power, comes great responsibility
  - Tread carefully!!



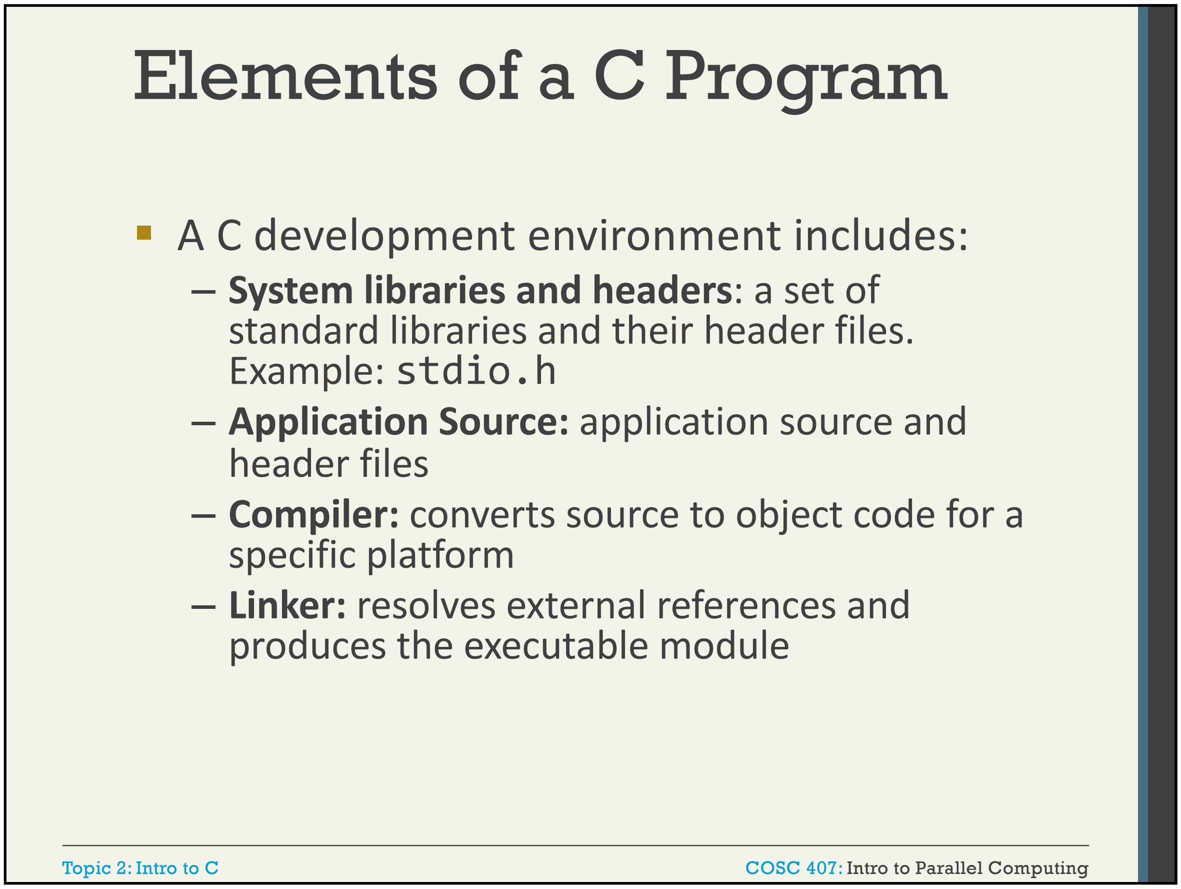
C vs. Java

- The basic syntax of C is similar to Java.
  - Keywords `int`, `if`, `for`, `while`, …etc.
- Some differences:
  - C is not object-oriented.
    - Basic programming unit is "function" ("method" in Java), not "class".
  - C is 'fairly' strongly typed
    - Allows implicit conversion between types
    - But original variable type does not change
  - C programs are not portable at the executable files level
    - Source code should be re-compiled to run on new platforms
  - C does not have the garbage collection mechanism



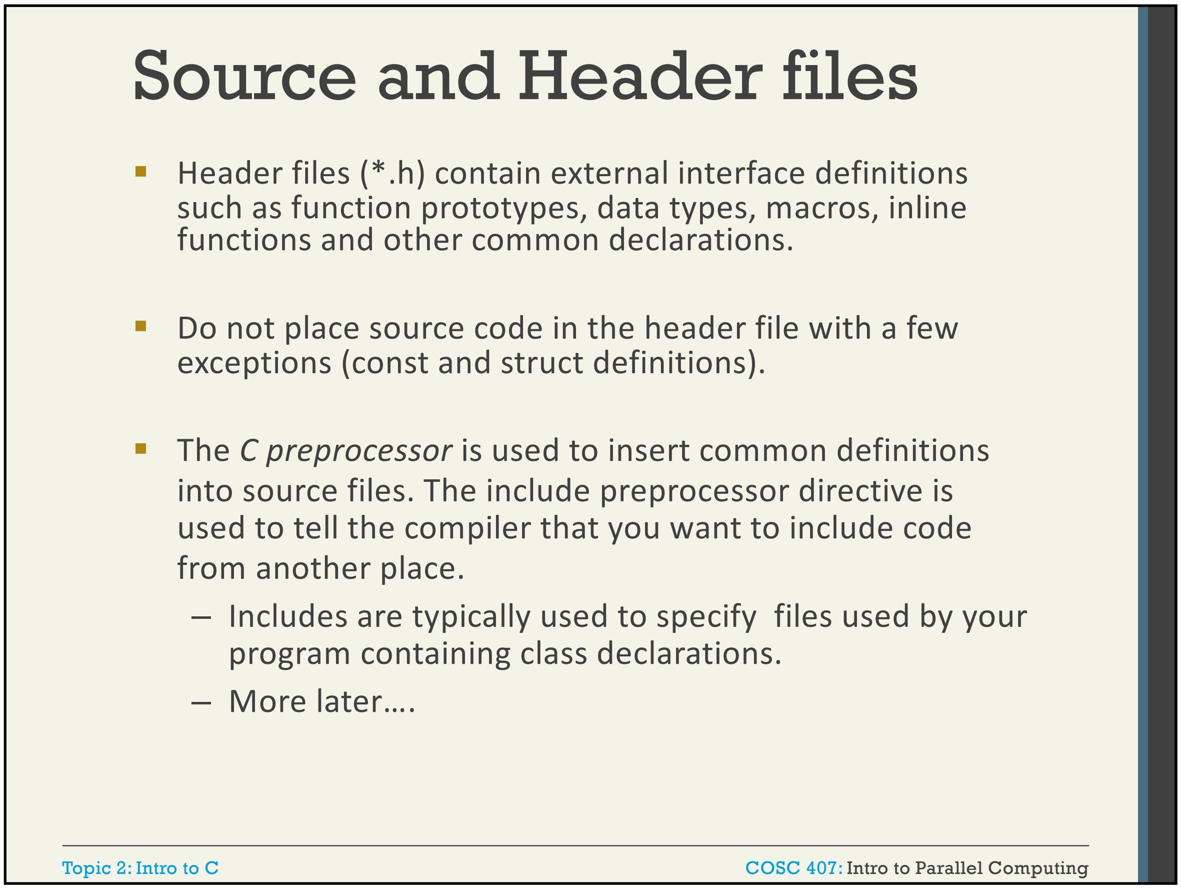
C vs. Java

- A source program in C uses header files
  - A header file has an extension **.h** and contains function declarations
    - for .h files that the programmer writes:
      `#include "file"`
    - for .h files that come with your compiler:
      `#include <file>`
- Some more differences:
  - **structs**, **pointers**, **arrays**, and some more things….



Elements of a C Program

- A C development environment includes:
  - **System libraries and headers**: a set of standard libraries and their header files. Example: `stdio.h`
  - **Application Source:** application source and header files
  - **Compiler:** converts source to object code for a specific platform
  - **Linker:** resolves external references and produces the executable module



Source and Header files

- Header files (*.h) contain external interface definitions such as function prototypes, data types, macros, inline functions and other common declarations.

- Do not place source code in the header file with a few exceptions (const and struct definitions).

- The *C preprocessor* is used to insert common definitions into source files. The include preprocessor directive is used to tell the compiler that you want to include code from another place.
  - Includes are typically used to specify files used by your program containing class declarations.
  - More later….

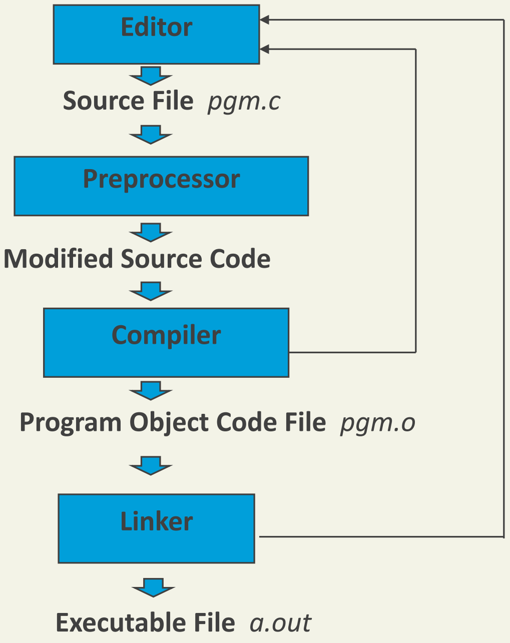
C Compilation

Preprocessor executes commands in code beginning with #.

Compiler translates source code into **object (machine) code.** Checks for **syntax errors** and **warnings**.

Compiler translates source code into **object (machine) code** but not functions from header files. Checks for **syntax errors** and **warnings**.

Linker replaces missing object files with appropriate code (links things together)



C vs Java Example

"Hello World" program!

In Java:

```
import java.lang.*  //not necessary
public class HelloWorld {
  public static void main(String[] args){
      System.out.println("Hello UBC");
  }
}
```

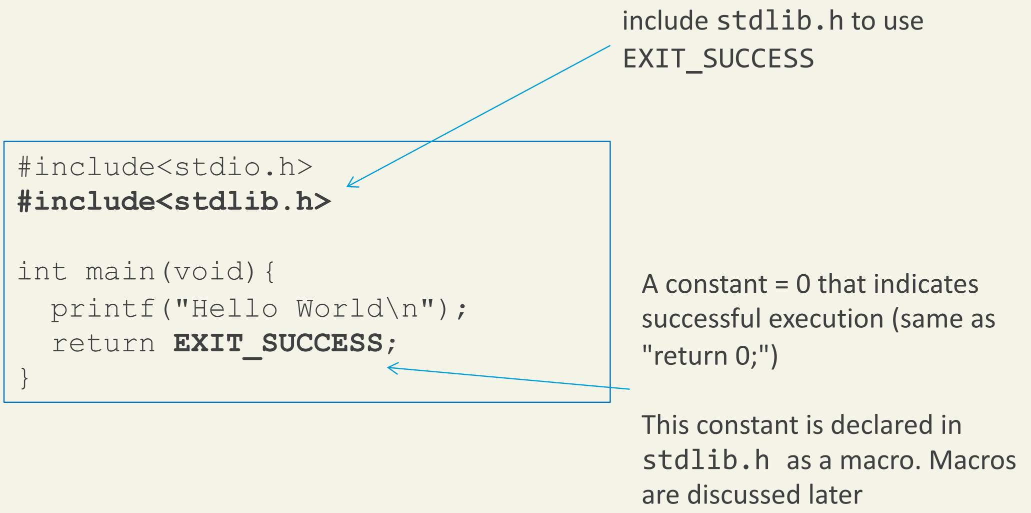
include `stdio`. header to compile the `printf` command

In C:

```
#include<stdio.h>
int main(void){
  printf("Hello UBC\n");
  return 0;
}
```

`main` is the first procedure to be executed

Example, cont'd



include stdlib.h to use EXIT_SUCCESS

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

int main(void){
  printf("Hello World\n");
  return EXIT_SUCCESS;
}
```

A constant = 0 that indicates successful execution (same as "return 0;")

This constant is declared in stdlib.h as a macro. Macros are discussed later

Data Types - Integer

- Integer types (can be signed (default) or unsinged)

Type	Storage size (machine dependent)	Range	
		signed	**unsigned**
char	1 byte	-128 to 127	0 to 255
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295

- To use unsigned types, put the keyword 'unsigned' before the type.
- size depends on particular system
  - Use sizeof(type) to find the value on your platform

Data Types Floating-point/void

- Floating-point types

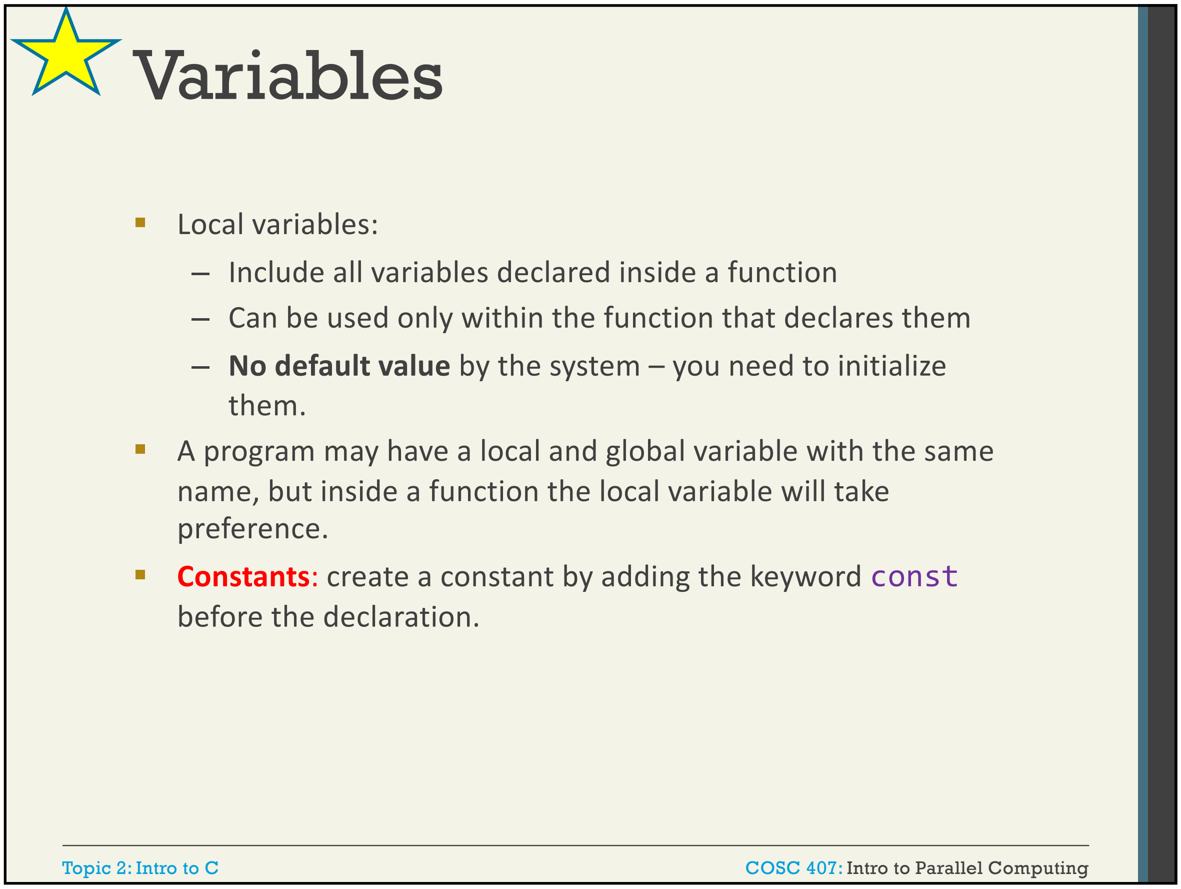
Type	Storage size	Range	Precision
`float`	4 byte	1.2E-38 to 3.4E+38	6 decimal places
`double`	8 bytes	2.3E-308 to 1.7E+308	15 decimal places
`long double`	10 bytes	3.4E-4932 to 1.1E+4932	19 decimal places

- `void` Represents the absence of type.

Variables

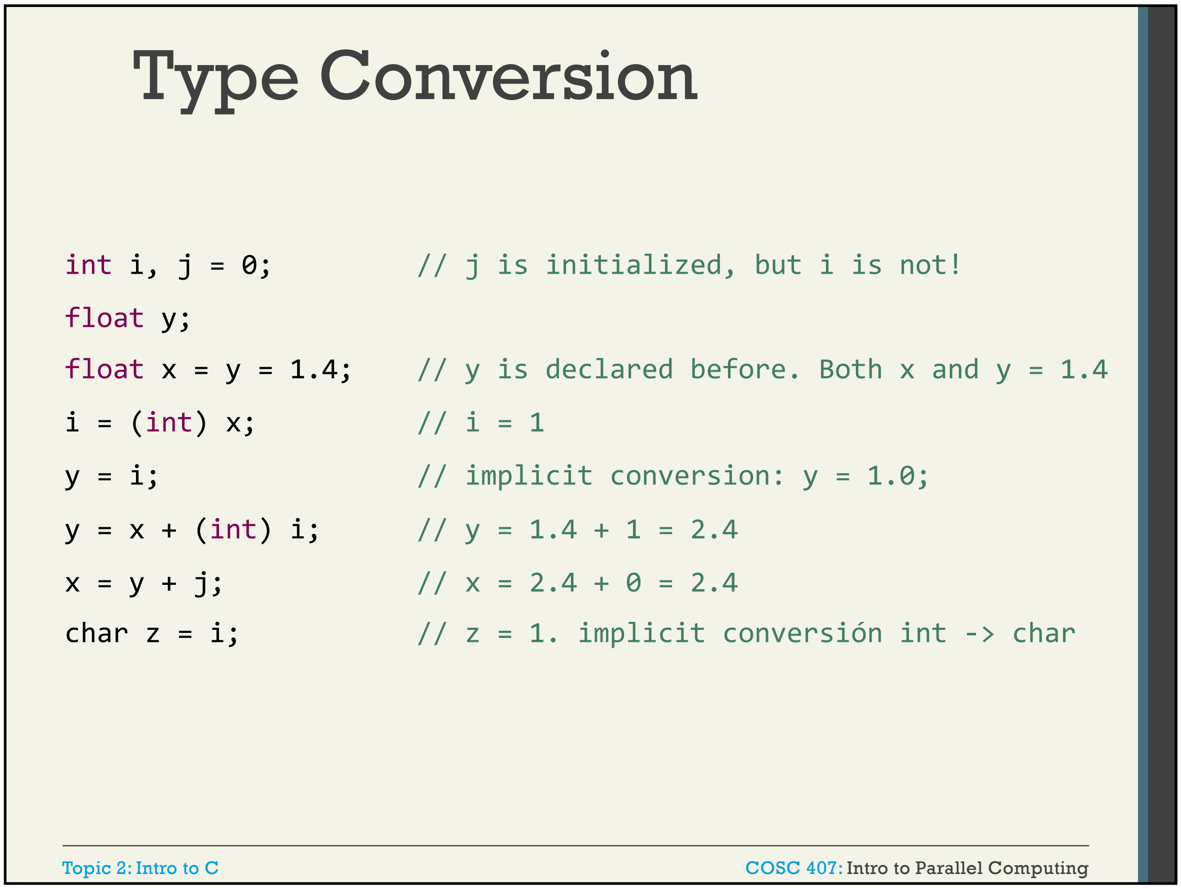
```
int    n;
unsigned short sh;
char   ch;
float  f;
double d;
```

- A variable is a name given to a memory storage location that we can manipulate in our programs.
  - A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
  - The first letter of a variable should be either a letter or an underscore.
  - There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.
- Global variables:
  - Include all variables declared outside any function (and outside `main` too).
  - Can be seen by all functions
  - **Default values**: 0 for numbers, '\0' for char, NULL for pointers
    - Still just initialize them



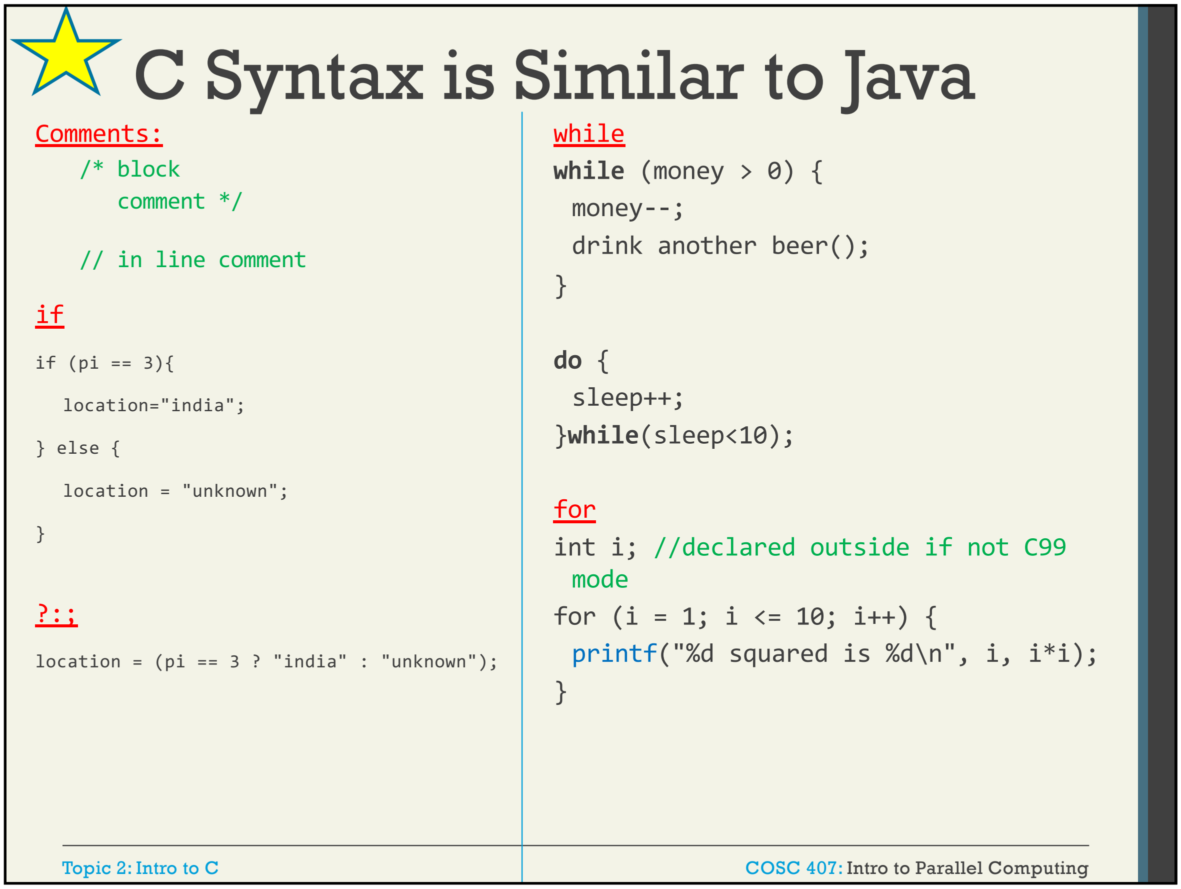
Variables

- Local variables:
  - Include all variables declared inside a function
  - Can be used only within the function that declares them
  - **No default value** by the system – you need to initialize them.
- A program may have a local and global variable with the same name, but inside a function the local variable will take preference.
- **Constants:** create a constant by adding the keyword `const` before the declaration.



Type Conversion

```
int i, j = 0;          // j is initialized, but i is not!
float y;
float x = y = 1.4;     // y is declared before. Both x and y = 1.4
i = (int) x;           // i = 1
y = i;                 // implicit conversion: y = 1.0;
y = x + (int) i;       // y = 1.4 + 1 = 2.4
x = y + j;             // x = 2.4 + 0 = 2.4
char z = i;            // z = 1. implicit conversión int -> char
```



C Syntax is Similar to Java

Comments:

```
  /* block
     comment */

  // in line comment
```

if

```
if (pi == 3){
   location="india";
} else {
   location = "unknown";
}
```

?::

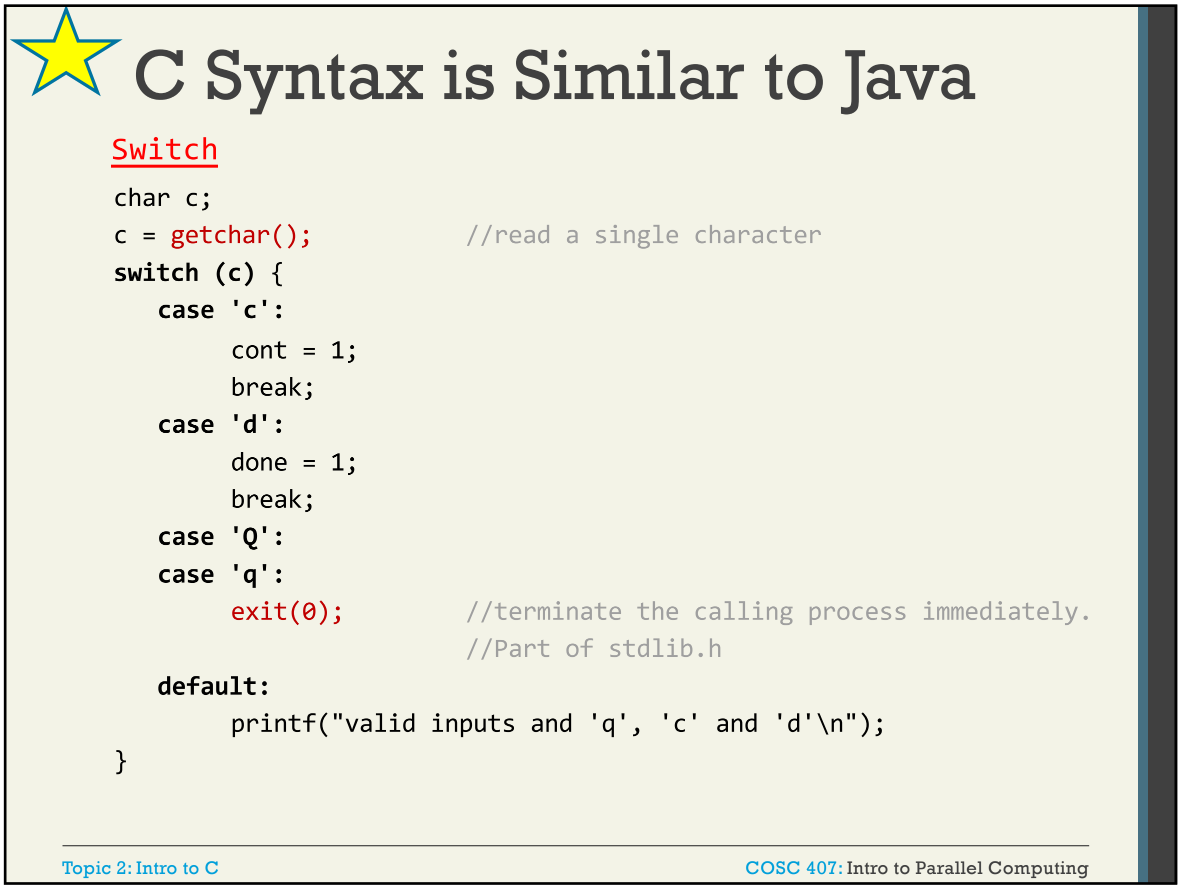
```
location = (pi == 3 ? "india" : "unknown");
```

while

```
while (money > 0) {
  money--;
  drink another beer();
}

do {
  sleep++;
}while(sleep<10);
```

for

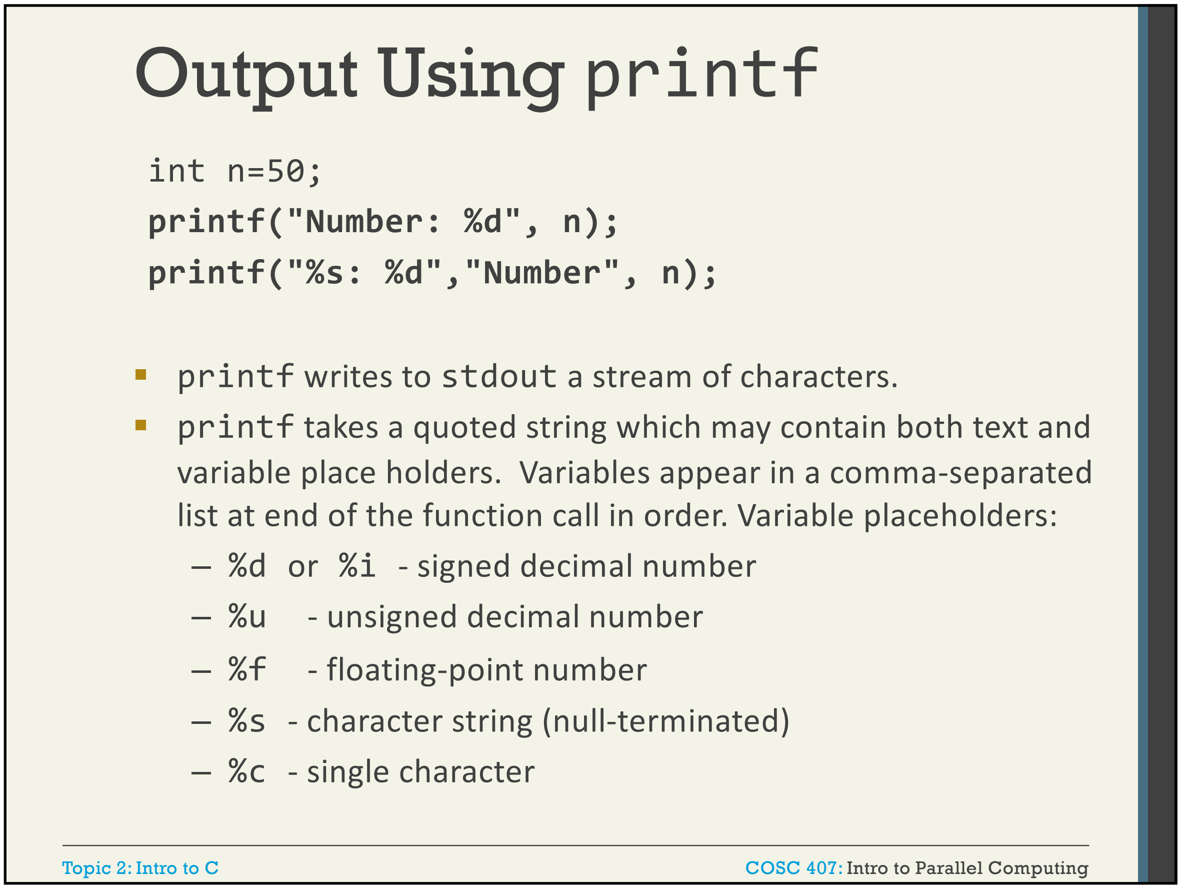
```
int i; //declared outside if not C99
  mode
for (i = 1; i <= 10; i++) {
  printf("%d squared is %d\n", i, i*i);
}
```



C Syntax is Similar to Java

Switch

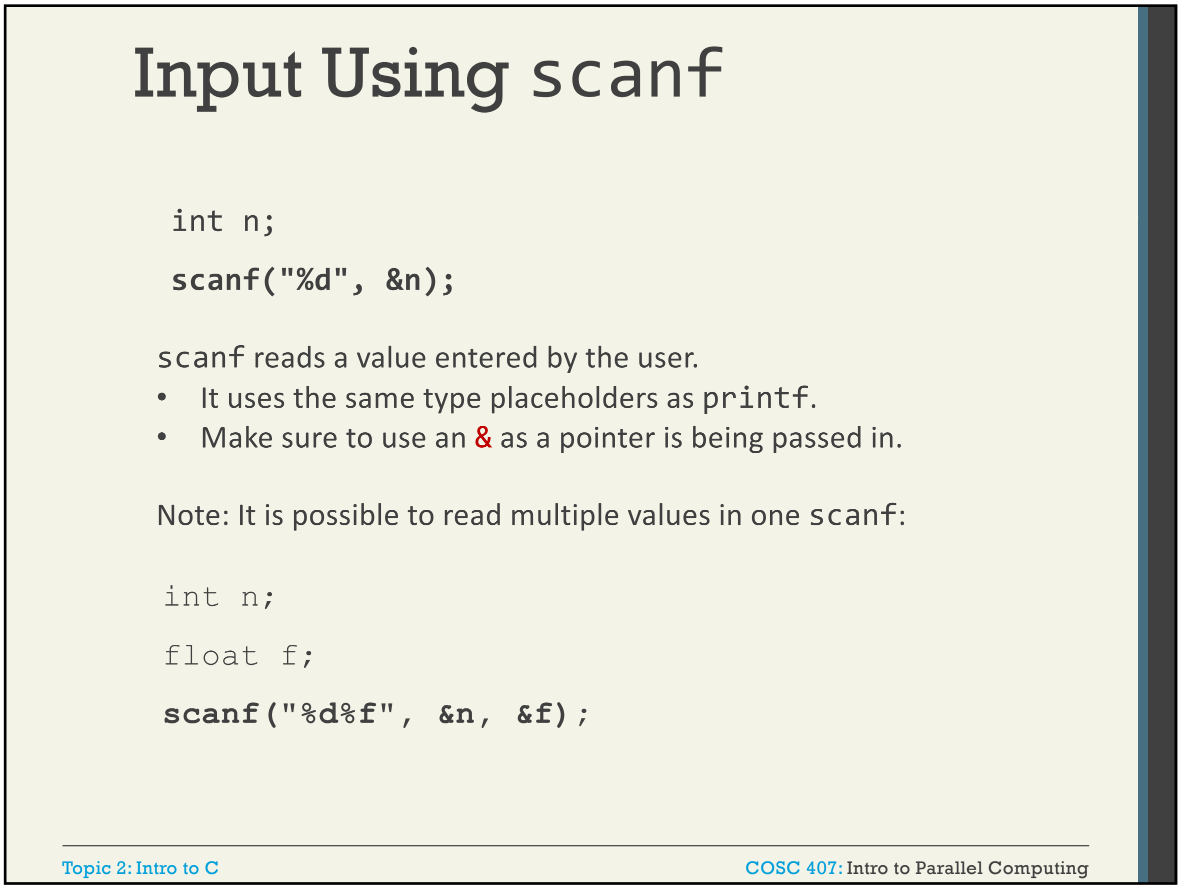
```
char c;
c = getchar();           //read a single character
switch (c) {
   case 'c':
         cont = 1;
         break;
   case 'd':
         done = 1;
         break;
   case 'Q':
   case 'q':
         exit(0);        //terminate the calling process immediately.
                         //Part of stdlib.h
   default:
         printf("valid inputs and 'q', 'c' and 'd'\n");
}
```



Output Using printf

```
int n=50;
printf("Number: %d", n);
printf("%s: %d","Number", n);
```

- printf writes to stdout a stream of characters.
- printf takes a quoted string which may contain both text and variable place holders. Variables appear in a comma-separated list at end of the function call in order. Variable placeholders:
  - %d or %i - signed decimal number
  - %u - unsigned decimal number
  - %f - floating-point number
  - %s - character string (null-terminated)
  - %c - single character



Input Using scanf

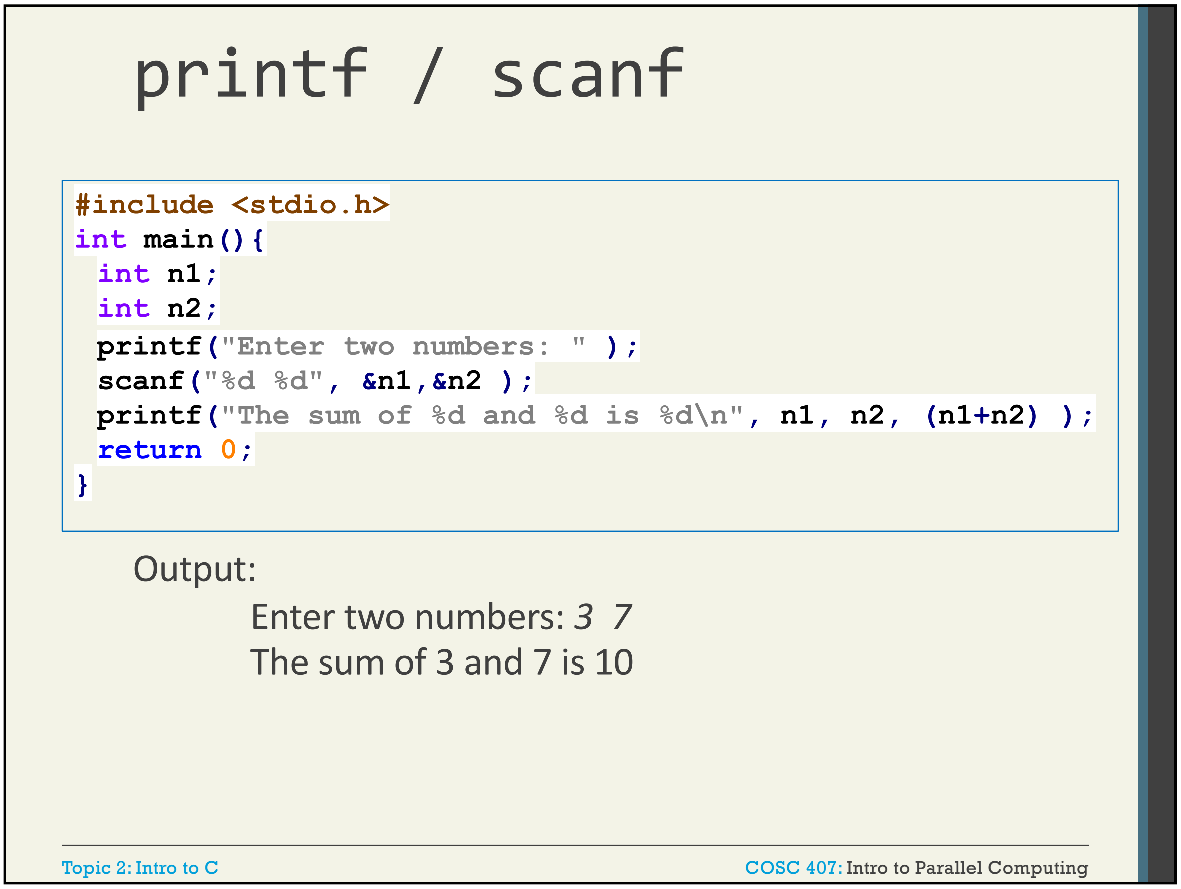
```
int n;
scanf("%d", &n);
```

scanf reads a value entered by the user.

- It uses the same type placeholders as printf.
- Make sure to use an & as a pointer is being passed in.

Note: It is possible to read multiple values in one scanf:

```
int n;
float f;
scanf("%d%f", &n, &f);
```

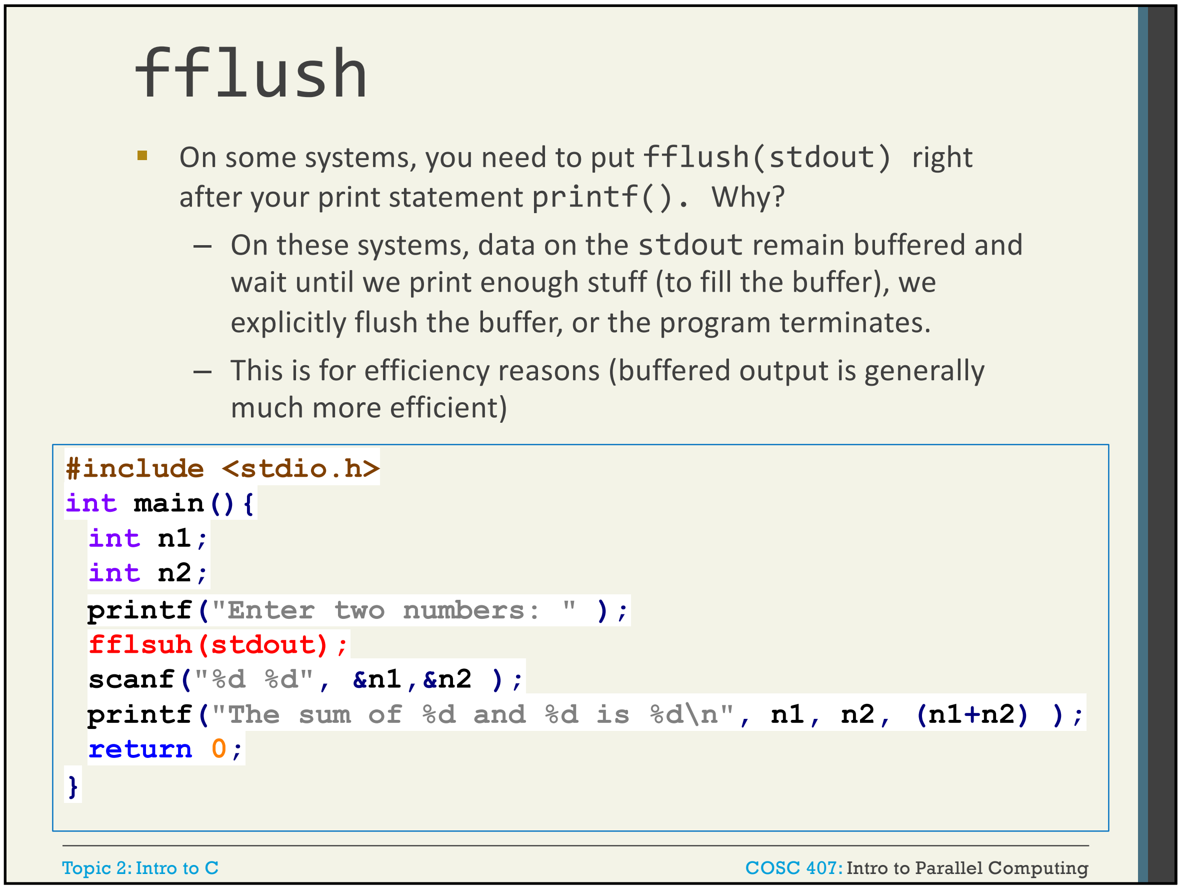


printf / scanf

```
#include <stdio.h>
int main(){
  int n1;
  int n2;
  printf("Enter two numbers:  " );
  scanf("%d %d", &n1,&n2 );
  printf("The sum of %d and %d is %d\n", n1, n2, (n1+n2) );
  return 0;
}
```

Output:

```
Enter two numbers: 3  7
The sum of 3 and 7 is 10
```



fflush

- On some systems, you need to put `fflush(stdout)` right after your print statement `printf()`. Why?
  - On these systems, data on the `stdout` remain buffered and wait until we print enough stuff (to fill the buffer), we explicitly flush the buffer, or the program terminates.
  - This is for efficiency reasons (buffered output is generally much more efficient)

```
#include <stdio.h>
int main(){
  int n1;
  int n2;
  printf("Enter two numbers:  " );
  fflsuh(stdout);
  scanf("%d %d", &n1,&n2 );
  printf("The sum of %d and %d is %d\n", n1, n2, (n1+n2) );
  return 0;
}
```

More Input / Output functions

- Reading/writing a single character

```
char c = getchar();
putchar(c);
```

- Reading a string

```
char str[20];
fgets(str, 20, stdin); //or gets(str)
or scanf("%s", str);
```

  - `fgets` reads a line from the `stdin` and stores it into the string pointed to by `str`. The input is stopped when reading 19 characters or when Enter is pressed.
    - `gets(str)` provides no protection against overflow

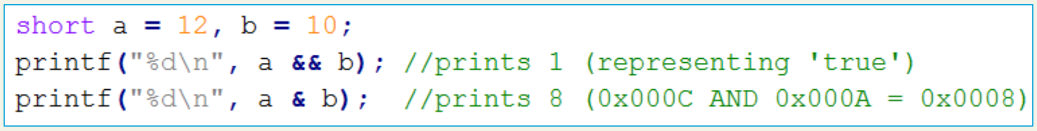
- Writing a string to standard output

```
puts("hello");
```

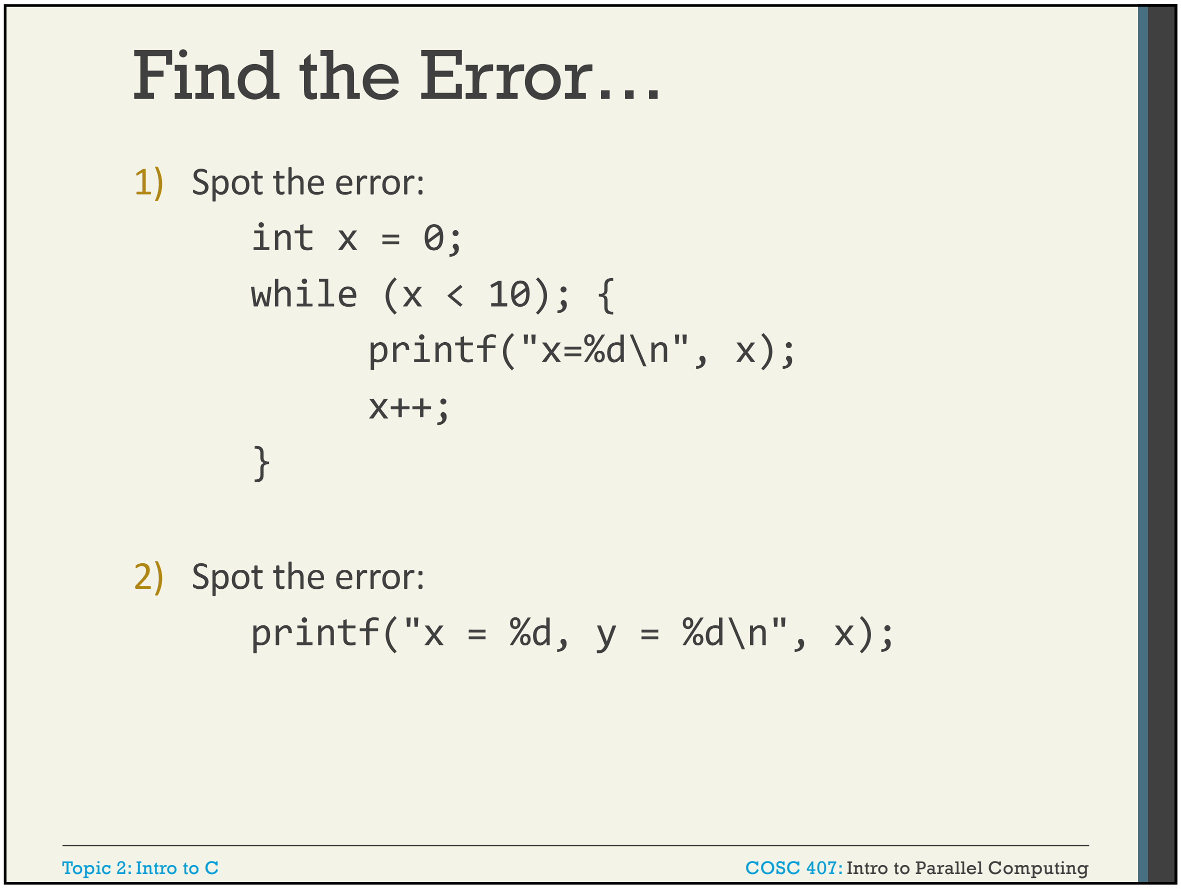
Operators

- Most operators are the same as Java.
  - Arithmetic +, -, *, /, %
  - Relational >=, <=, ==, !=, >, <
  - Logic &&, ||, !
    - 0 is considered false and non-zero values are true
  - Assignment: =, +=, -=, *=, /=, %=, |=, etc.
  - Increment/decrement: ++, --
- **Bitwise** operators (working on data bit-by-bit):

<<	Shift Left	&	bitwise AND
>>	Shift Right	|	bitwise OR
~	NOT	^	bitwise XOR



```
short a = 12, b = 10;
printf("%d\n", a && b); //prints 1 (representing 'true')
printf("%d\n", a & b);  //prints 8 (0x000C AND 0x000A = 0x0008)
```



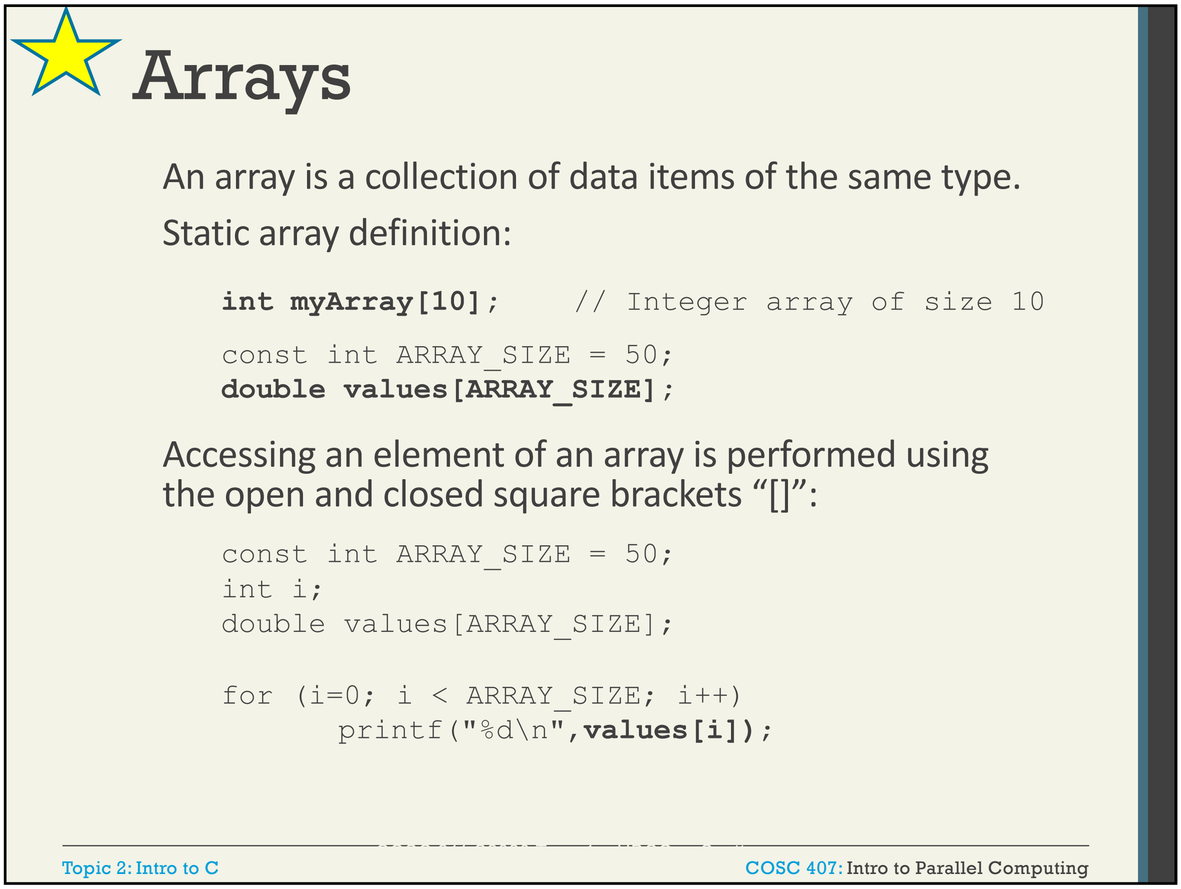
Find the Error…

1) Spot the error:

```
int x = 0;
while (x < 10); {
      printf("x=%d\n", x);
      x++;
}
```

2) Spot the error:

```
printf("x = %d, y = %d\n", x);
```



Arrays

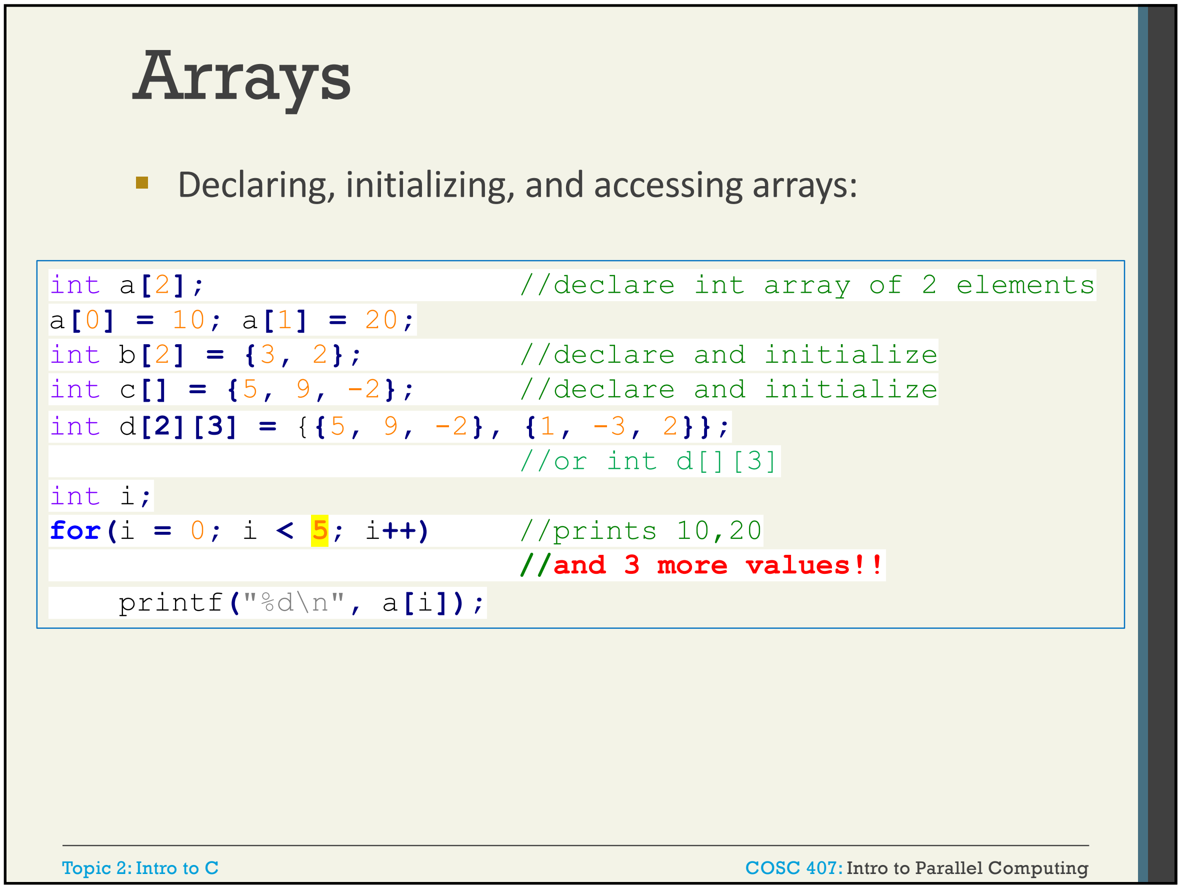
An array is a collection of data items of the same type.

Static array definition:

```
int myArray[10];    // Integer array of size 10

const int ARRAY_SIZE = 50;
double values[ARRAY_SIZE];
```

Accessing an element of an array is performed using the open and closed square brackets “[]”:

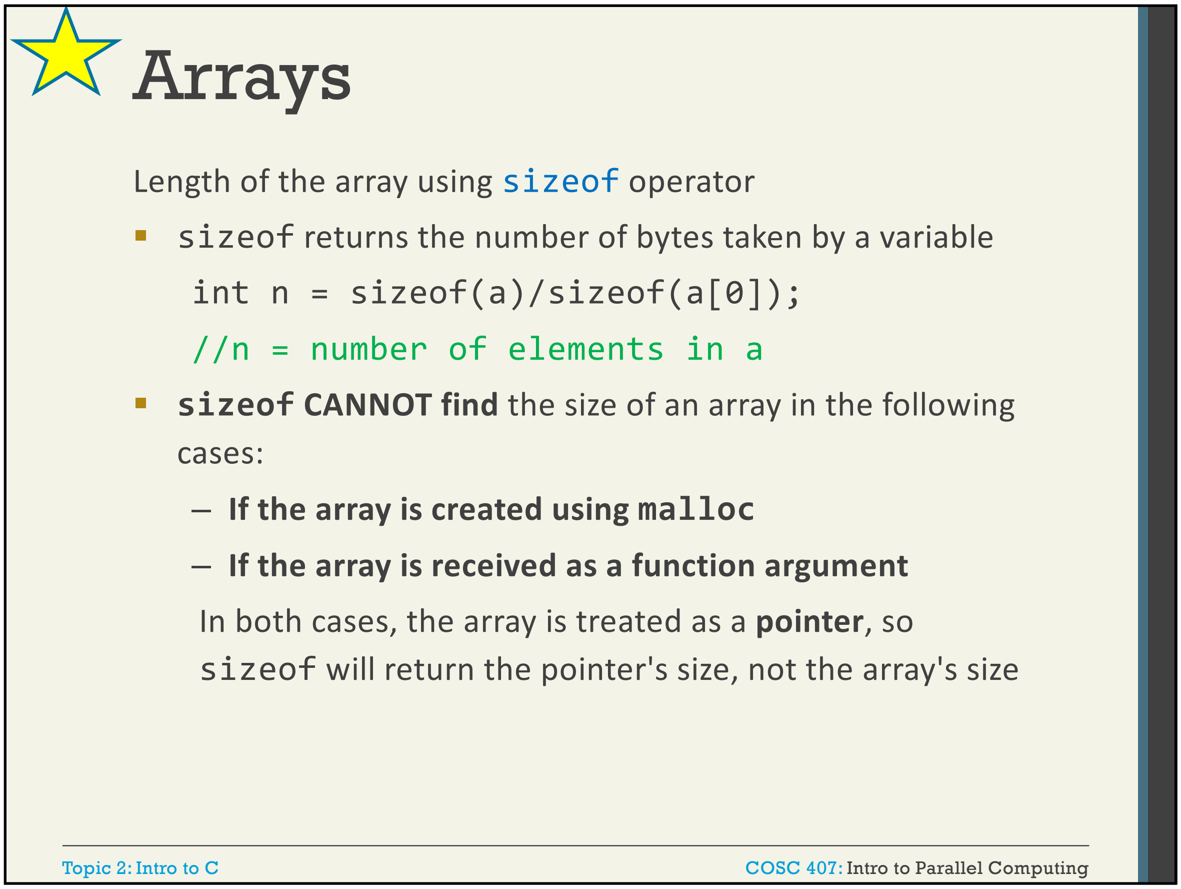
```
const int ARRAY_SIZE = 50;
int i;
double values[ARRAY_SIZE];

for (i=0; i < ARRAY_SIZE; i++)
      printf("%d\n",values[i]);
```



Arrays

- Declaring, initializing, and accessing arrays:

```
int a[2];                  //declare int array of 2 elements
a[0] = 10; a[1] = 20;
int b[2] = {3, 2};         //declare and initialize
int c[] = {5, 9, -2};      //declare and initialize
int d[2][3] = {{5, 9, -2}, {1, -3, 2}};
                           //or int d[][3]
int i;
for(i = 0; i < 5; i++)     //prints 10,20
                           //and 3 more values!!
    printf("%d\n", a[i]);
```



Arrays

Length of the array using `sizeof` operator

- `sizeof` returns the number of bytes taken by a variable

```
int n = sizeof(a)/sizeof(a[0]);
//n = number of elements in a
```

- **`sizeof` CANNOT find** the size of an array in the following cases:
  - **If the array is created using `malloc`**
  - **If the array is received as a function argument**

  In both cases, the array is treated as a **pointer**, so `sizeof` will return the pointer's size, not the array's size

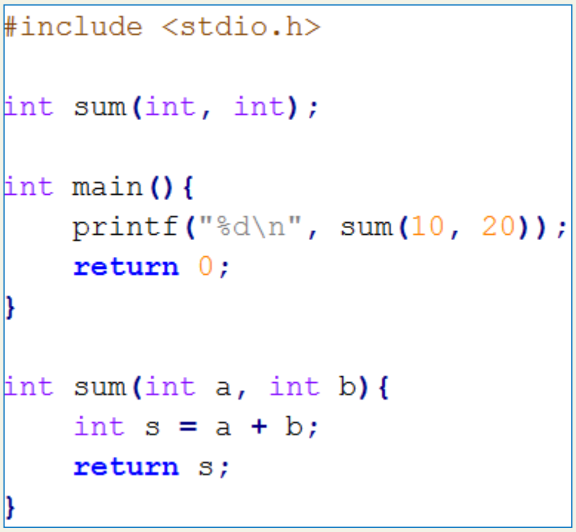
Functions

Functions have **prototypes** that must be declared before referenced.

A prototype is the declaration line without the function body.

- declaration of the function name, return type, and parameters.

Function definition provides the actual body of the function

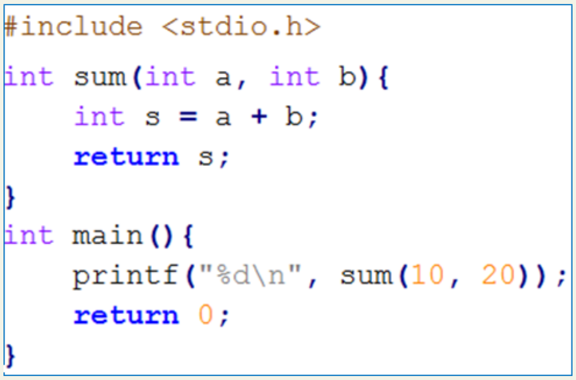


```
#include <stdio.h>

int sum(int, int);

int main(){
    printf("%d\n", sum(10, 20));
    return 0;
}

int sum(int a, int b){
    int s = a + b;
    return s;
}
```

- C is ***call by value*** meaning that a copy of the parameter is passed to the function.
- Many built-in functions that are provided in the C libraries: e.g., `strcat()`

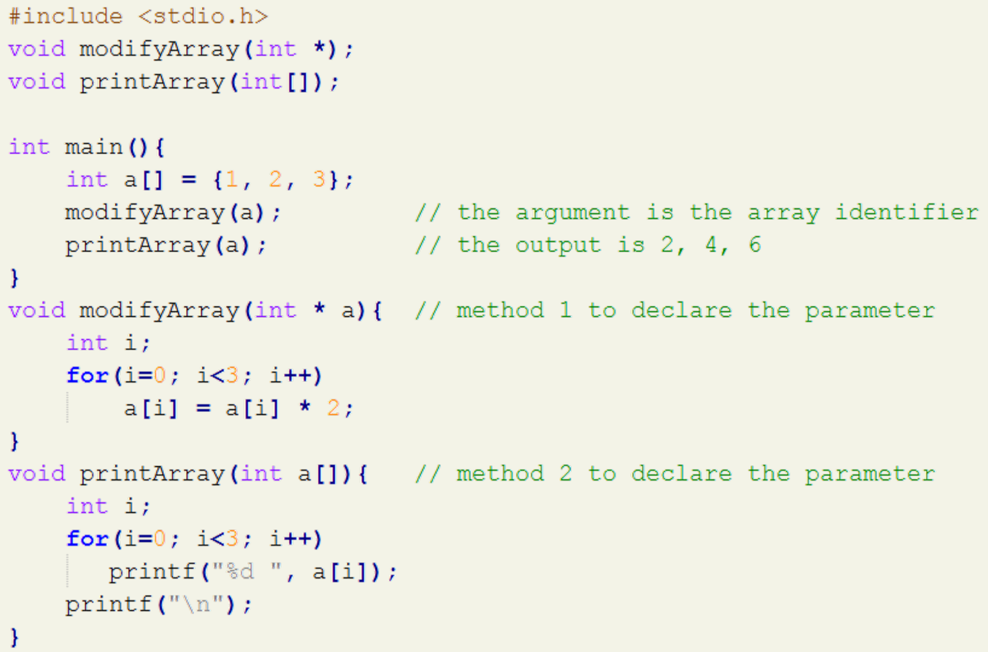
Functions

Function definition before the main function



```
#include <stdio.h>
int sum(int a, int b){
    int s = a + b;
    return s;
}
int main(){
    printf("%d\n", sum(10, 20));
    return 0;
}
```

Passing an Array to a Function



```
#include <stdio.h>
void modifyArray(int *);
void printArray(int[]);

int main(){
    int a[] = {1, 2, 3};
    modifyArray(a);         // the argument is the array identifier
    printArray(a);          // the output is 2, 4, 6
}
void modifyArray(int * a){  // method 1 to declare the parameter
    int i;
    for(i=0; i<3; i++)
        a[i] = a[i] * 2;
}
void printArray(int a[]){   // method 2 to declare the parameter
    int i;
    for(i=0; i<3; i++)
       printf("%d ", a[i]);
    printf("\n");
}
```

Declaring Pointer Variables

A ***pointer variable*** stores an address to a particular location in memory. Pointers point to an object of a particular data type.

- A pointer variable is declared by putting a "*" in front of the variable name:

  *variableType *variableName;*

- For example:

```
int    *intPtr;
double *d;
char   *strPtr;
```

All pointer variables store an ***address*** not the actual type.

- The type is important to know how many consecutive bytes to retrieve starting at the given address.

Pointers

- Each memory location has an address.
  A variable **address** can be **accessed** using the **&** operator
- Example: Print the address of a variable x

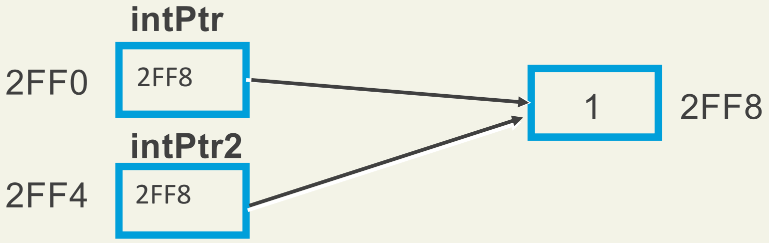
```
#include <stdio.h>
int main(void) {
        int z = 7;
        printf("Address of z: %p\n", &z);
        return 0;
}
```

Manipulating Pointers

- The value of a pointer variable can be changed like any other variable using assignment. Example:

```
int    *intPtr, *intPtr2, i = 1;
intPtr = &i;          // intPtr stores address of i
intPtr2 = intPtr;     // intPtr2=intPtr1 (address of i)
```

- Memory diagram:



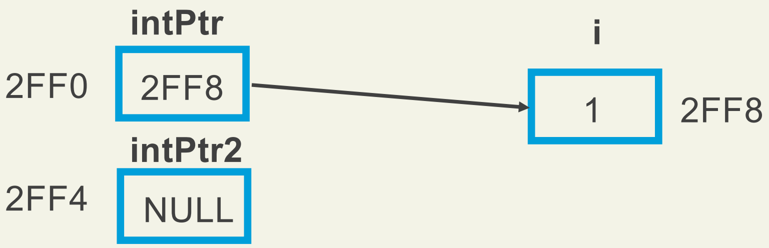
  – Note that the ampersand "**&**" denotes the address of a variable. (***address operator***)

NULL Pointer Values

If you do not want a pointer to point to anything, you set its value to **NULL**. Example:

```
int    *intPtr, *intPtr2, i = 1;
intPtr = &i;         // intPtr stores address of i
intPtr2 = NULL;      // intPtr2=NULL (points to nothing)
```

Memory diagram:



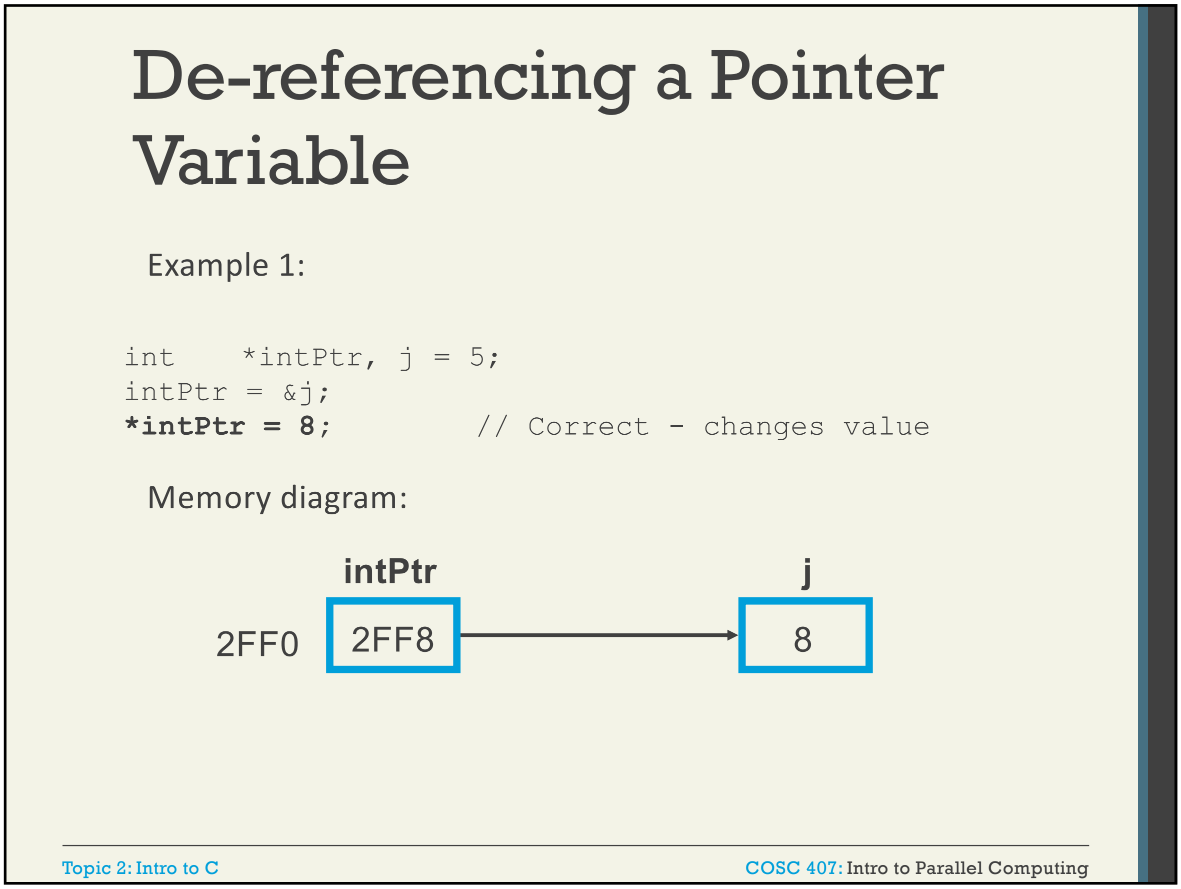
◆ The value of NULL is 0.

De-referencing a Pointer Variable

To change the ***value*** of a variable when you have its pointer (address), you must de-reference the address to operate on the data, not the pointer itself.

To deference a pointer variable, put a "*****" in front of the pointer (think ‘value at’.

– This means that you are interested in the value pointed to, not the value of the address pointer.



De-referencing a Pointer Variable

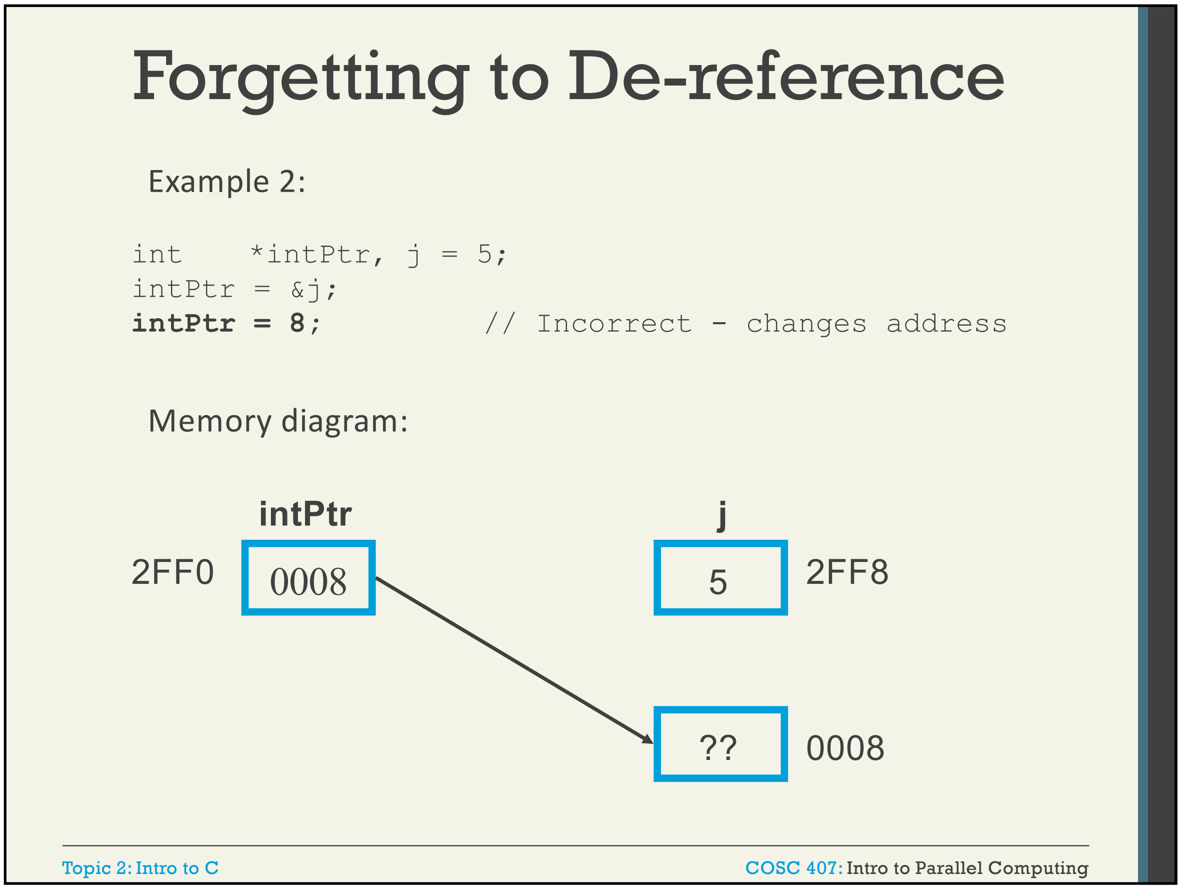
Example 1:

```
int    *intPtr, j = 5;
intPtr = &j;
*intPtr = 8;         // Correct - changes value
```

Memory diagram:

intPtr
j
2FF0
2FF8
8

44



Forgetting to De-reference

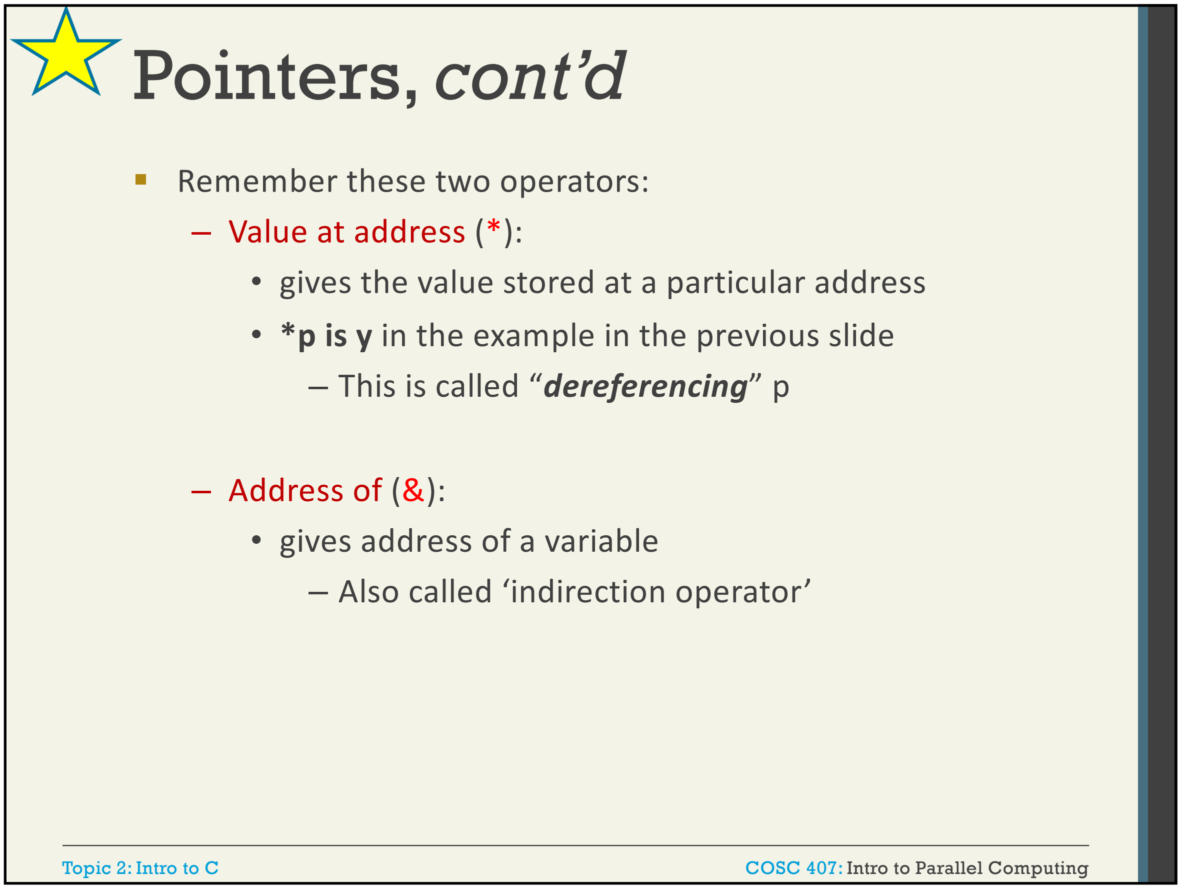
Example 2:

```
int    *intPtr, j = 5;
intPtr = &j;
intPtr = 8;          // Incorrect - changes address
```

Memory diagram:

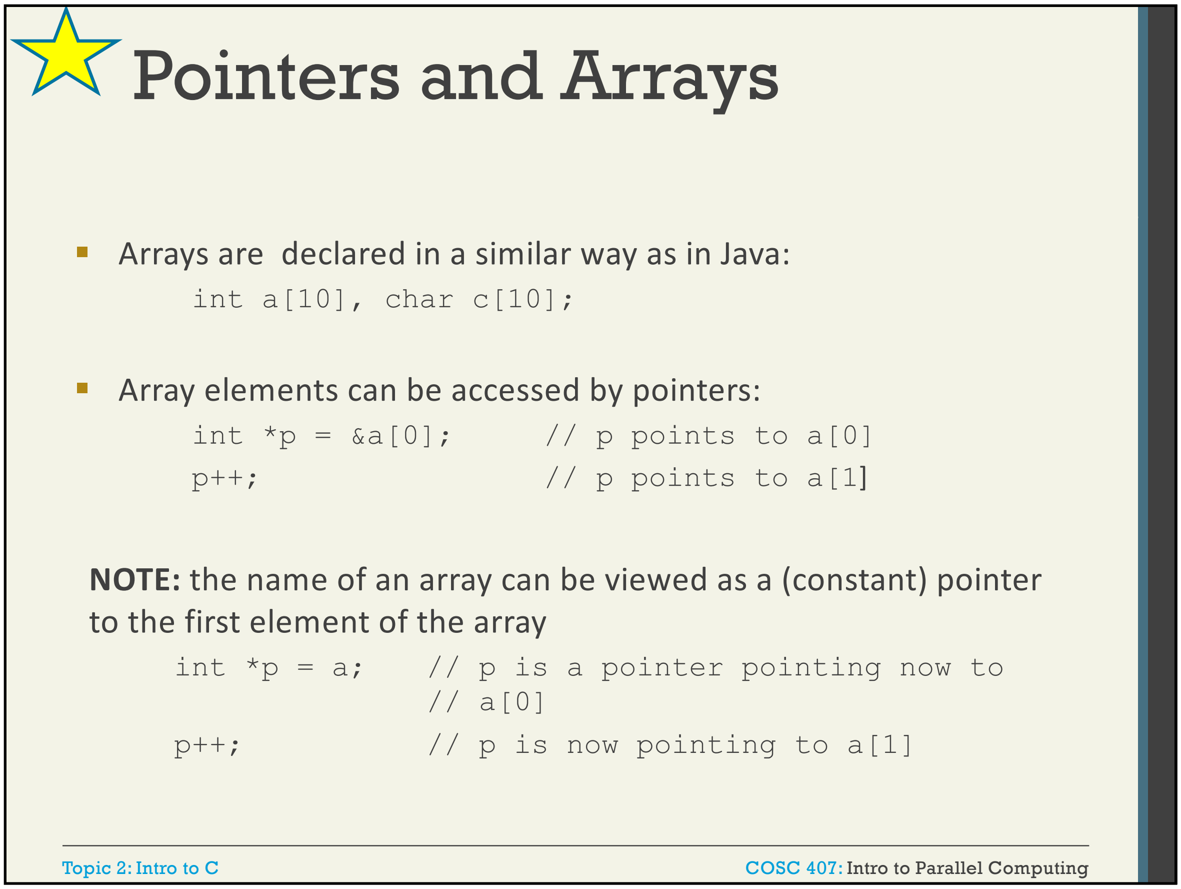
intPtr
j
2FF0
0008
5
2FF8
??
0008

45



Pointers, *cont'd*

- Remember these two operators:
  - Value at address (*):
    - gives the value stored at a particular address
    - ***p is y** in the example in the previous slide
      - This is called "***dereferencing***" p
  - Address of (&):
    - gives address of a variable
      - Also called 'indirection operator'



Pointers and Arrays

- Arrays are declared in a similar way as in Java:

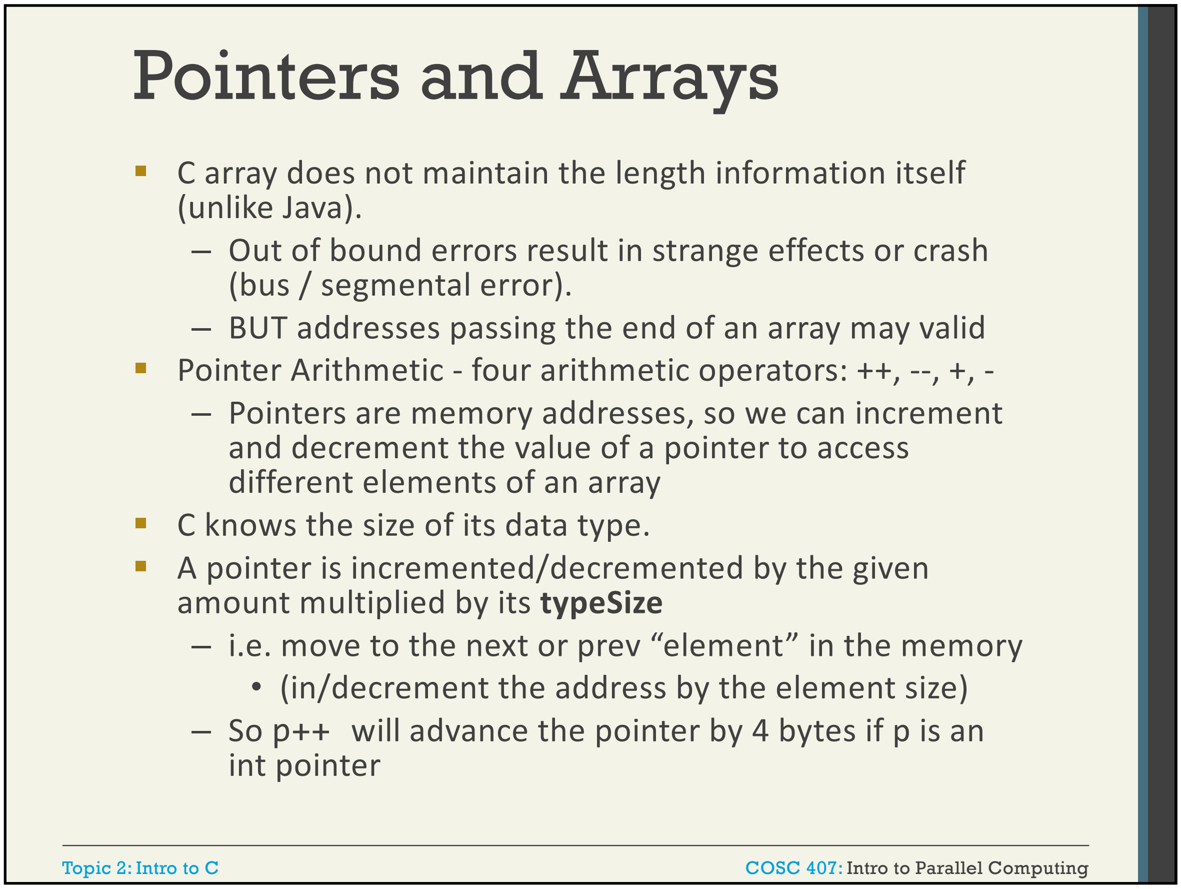
```
int a[10], char c[10];
```

- Array elements can be accessed by pointers:

```
int *p = &a[0];      // p points to a[0]
p++;                 // p points to a[1]
```

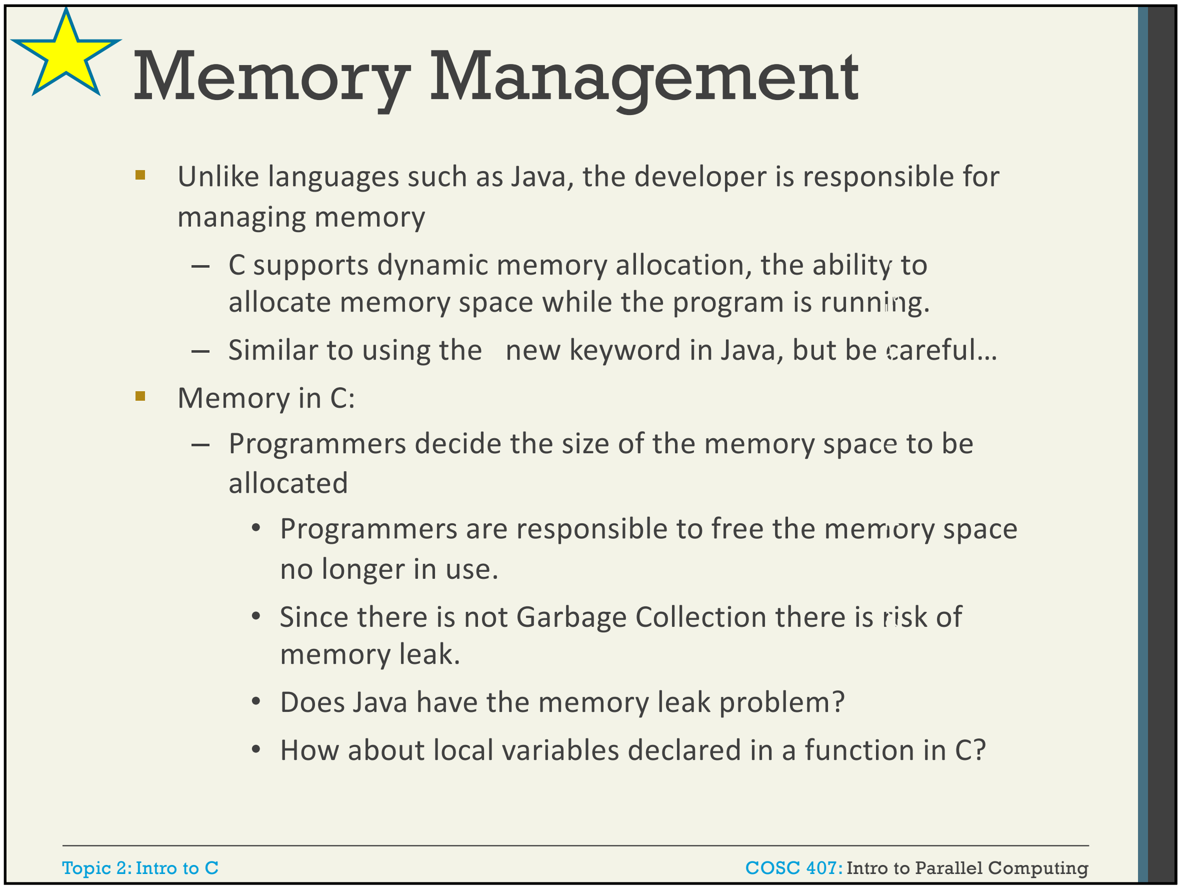
NOTE: the name of an array can be viewed as a (constant) pointer to the first element of the array

```
int *p = a;    // p is a pointer pointing now to
               // a[0]
p++;           // p is now pointing to a[1]
```



Pointers and Arrays

- C array does not maintain the length information itself (unlike Java).
  - Out of bound errors result in strange effects or crash (bus / segmental error).
  - BUT addresses passing the end of an array may valid
- Pointer Arithmetic - four arithmetic operators: ++, --, +, -
  - Pointers are memory addresses, so we can increment and decrement the value of a pointer to access different elements of an array
- C knows the size of its data type.
- A pointer is incremented/decremented by the given amount multiplied by its **typeSize**
  - i.e. move to the next or prev "element" in the memory
    - (in/decrement the address by the element size)
  - So p++ will advance the pointer by 4 bytes if p is an int pointer



Memory Management

- Unlike languages such as Java, the developer is responsible for managing memory
  - C supports dynamic memory allocation, the ability to allocate memory space while the program is running.
  - Similar to using the new keyword in Java, but be careful…
- Memory in C:
  - Programmers decide the size of the memory space to be allocated
    - Programmers are responsible to free the memory space no longer in use.
    - Since there is not Garbage Collection there is risk of memory leak.
    - Does Java have the memory leak problem?
    - How about local variables declared in a function in C?

Outline

- Next Lecture:
  - More Pointers (memory allocation, 2D arrays, functions)
  - Error Handling
  - String processing
  - struct, typedef
  - Preprocessors, Compiling C programs

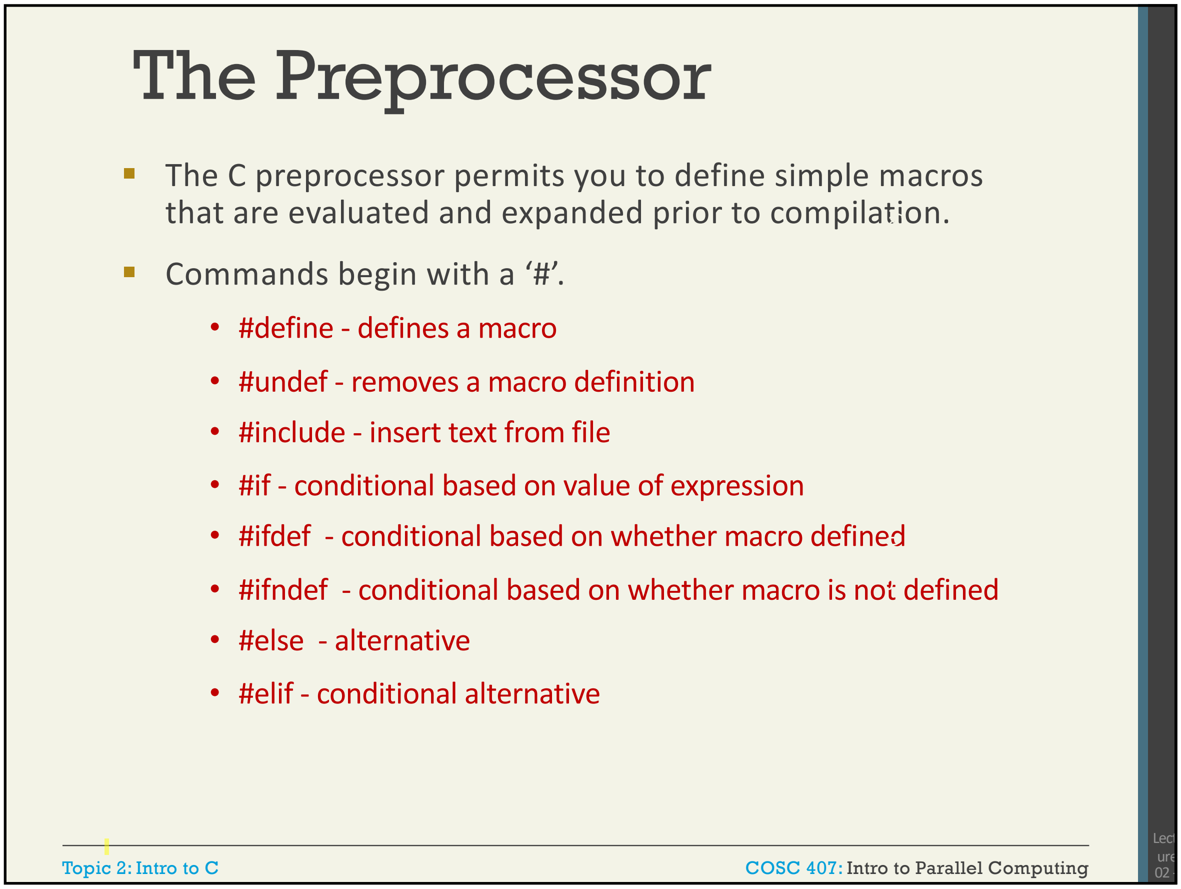
Homework

- Quiz on this material will be released (due by next Thursday)

- Read An introduction to C Programming for Java Programmer, Handley. (AGAIN)
  - Available on Canvas

- Find a C reference for later use (library, online)
  - e.g., http://www.tutorialspoint.com

- Practice on C!!!
  - Review exercises on Canvas and write ON YOUR OWN (you can use online complier, but save your results)
  - Find some online quizzes and practice



The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.
- Commands begin with a '#'.
  - #define - defines a macro
  - #undef - removes a macro definition
  - #include - insert text from file
  - #if - conditional based on value of expression
  - #ifdef - conditional based on whether macro defined
  - #ifndef - conditional based on whether macro is not defined
  - #else - alternative
  - #elif - conditional alternative