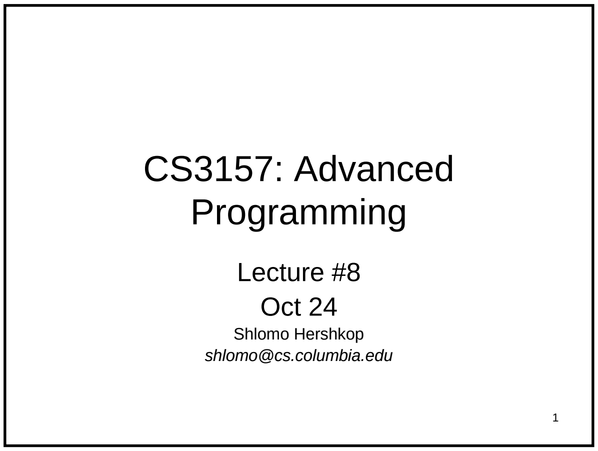
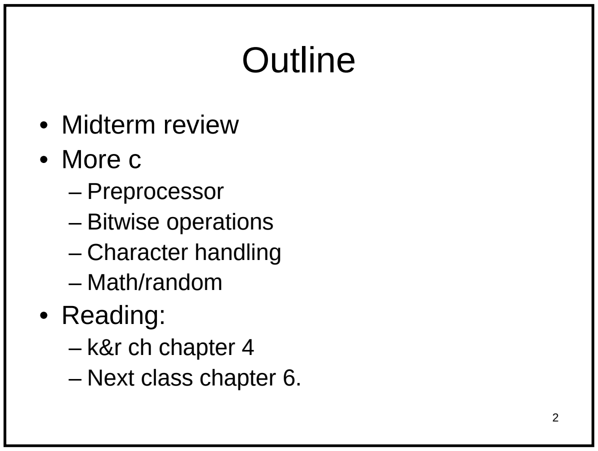


CS3157: Advanced Programming

Lecture #8

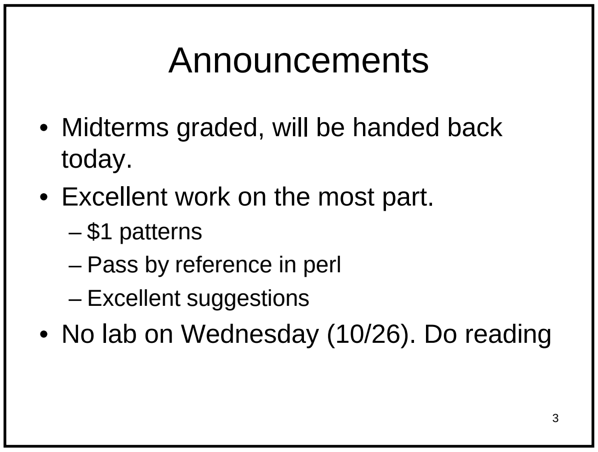
Oct 24

Shlomo Hershkop
shlomo@cs.columbia.edu



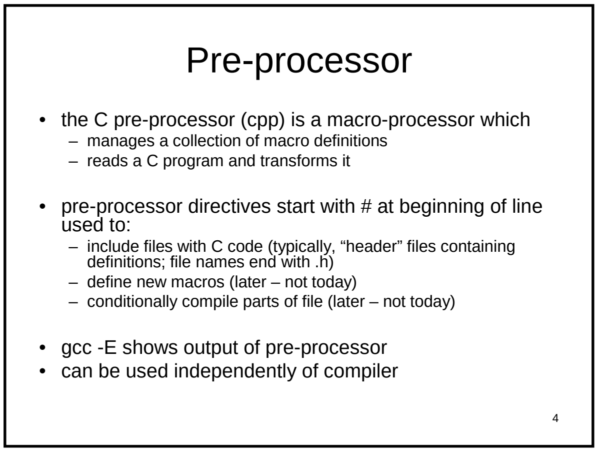
Outline

- Midterm review
- More c
  - Preprocessor
  - Bitwise operations
  - Character handling
  - Math/random
- Reading:
  - k&r ch chapter 4
  - Next class chapter 6.



Announcements

- Midterms graded, will be handed back today.
- Excellent work on the most part.
  - $1 patterns
  - Pass by reference in perl
  - Excellent suggestions
- No lab on Wednesday (10/26). Do reading



Pre-processor

- the C pre-processor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it
- pre-processor directives start with # at beginning of line used to:
  - include files with C code (typically, “header” files containing definitions; file names end with .h)
  - define new macros (later – not today)
  - conditionally compile parts of file (later – not today)
- gcc -E shows output of pre-processor
- can be used independently of compiler

Pre-processor cont.

```
#define name const-expression
#define name (param1,param2,...) expression
#undef symbol
```

- replaces name with constant or expression
- textual substitution
- symbolic names for global constants
- in-line functions (avoid function call overhead)
- type-independent code

```
#define MAXLEN 255
```

Example

```
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i)) { ...}
```

- becomes

```
if ((i) < 100) {...}
```

- Caution: don't treat macros like function calls

```
#define valid(x) ((x) > 0 && (x) < 20)
```

- is called like:

```
if (valid(x++)) {...}
```

- and will become:

```
valid(x++) -> ((x++) > 0 && (x++) < 20)
```

- and may not do what you intended...

- conditional compilation

- pre-processor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- machine or OS-dependent code

- can be used to comment out chunks of code— bad!
- (but can be helpful for quick and dirty debugging :-)

- example:

```
#define OS linux
...
#if OS == linux
puts( “Wow you are running Linux!" );
#else
puts( "why are you running something else???" );
#endif
```

- ifdef
- for boolean flags, easier:

```
#ifdef name
code segment 1
#else
code segment 2
#endif
```

- pre-processor checks if name has been defined, e.g.:

```
#define USEDB
```

- if so, use code segment 1, otherwise 2

Function

- Declaration:
  - `Return-type function-name (parameters if any);`
- Definition:
  - `Return-type function-name (parameters if any){`

```
Return-type function-name (parameters if any){
declarations


statements
}
```

Command Line Args

```
int main( int argc, char *argv[] )
```

- argc is the argument count
- argv is the argument vector
  - array of strings with command-line arguments
- the `int` value is the return value
  - convention: return value of 0 means success,
  - > 0 means there was some kind of error
  - can also declare as `void` (no return value)

- Name of executable followed by space-separated arguments

```
$ a.out 1 23 "third arg"
```

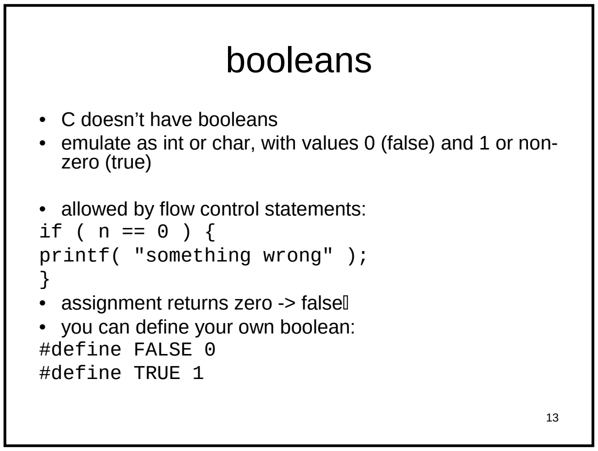
- this is stored like this:

```
1. a.out
2. 1
3. 23
4. “third arg”
•  argc  = 4
```

- If no arguments, simplify:

```
int main() {
printf( "hello world" );
exit( 0 );
}
```

- Uses exit() instead of return() — almost the same thing.



booleans

- C doesn’t have booleans
- emulate as int or char, with values 0 (false) and 1 or non-zero (true)

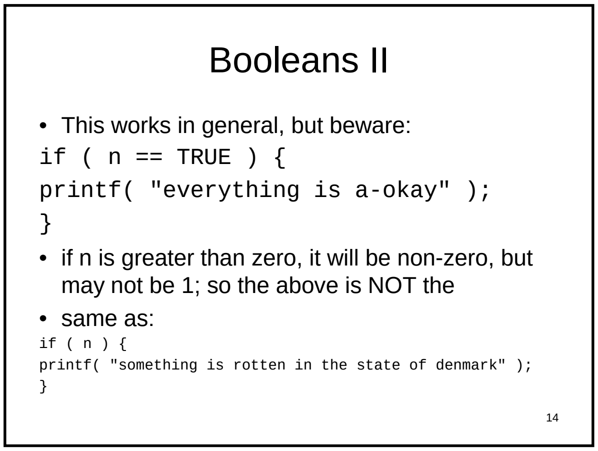
- allowed by flow control statements:

```
if ( n == 0 ) {
printf( "something wrong" );
}
```

- assignment returns zero -> false
- you can define your own boolean:

```
#define FALSE 0
#define TRUE 1
```

13



Booleans II

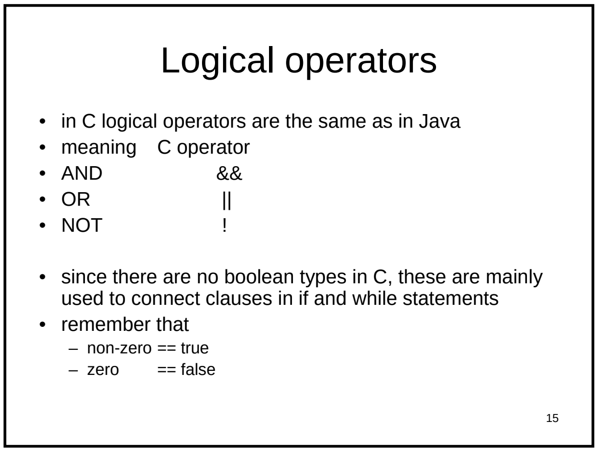
- This works in general, but beware:

```
if ( n == TRUE ) {
printf( "everything is a-okay" );
}
```

- if n is greater than zero, it will be non-zero, but may not be 1; so the above is NOT the
- same as:

```
if ( n ) {
printf( "something is rotten in the state of denmark" );
}
```

14

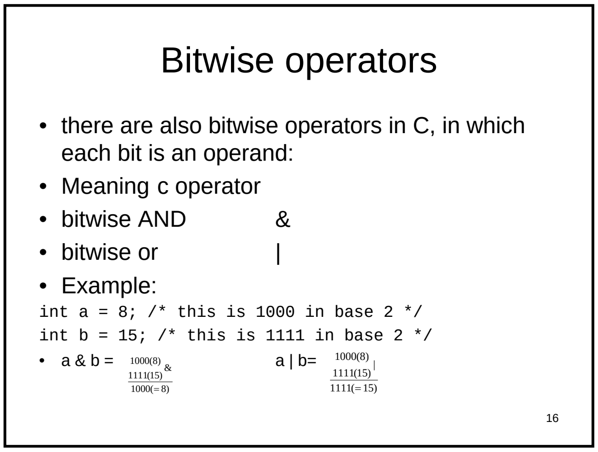


Logical operators

- in C logical operators are the same as in Java
- meaning C operator
- AND &&
- OR ||
- NOT !

- since there are no boolean types in C, these are mainly used to connect clauses in if and while statements
- remember that
  - non-zero == true
  - zero == false

15



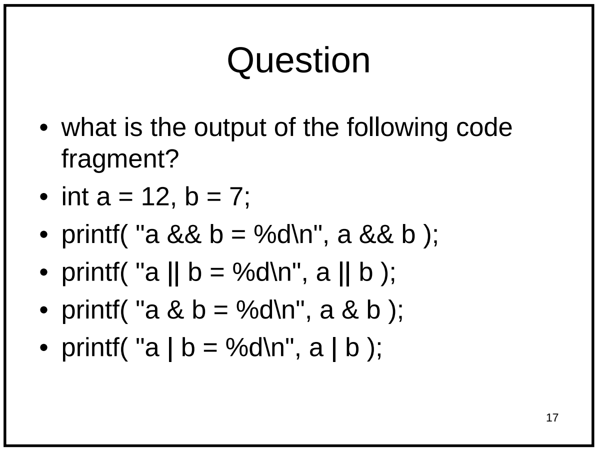
Bitwise operators

- there are also bitwise operators in C, in which each bit is an operand:
- Meaning c operator
- bitwise AND &
- bitwise or |
- Example:

```
int a = 8; /* this is 1000 in base 2 */
int b = 15; /* this is 1111 in base 2 */
```

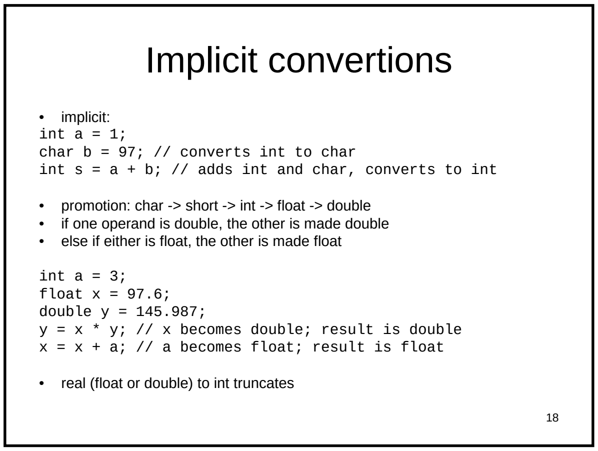
- a & b = $\begin{array}{r} 1000(8) \\ 1111(15)\ \& \\ \hline 1000(=8) \end{array}$ a | b= $\begin{array}{r} 1000(8) \\ 1111(15)\ | \\ \hline 1111(=15) \end{array}$

16



Question

- what is the output of the following code fragment?
- int a = 12, b = 7;
- printf("a && b = %d\n", a && b);
- printf("a || b = %d\n", a || b);
- printf("a & b = %d\n", a & b);
- printf("a | b = %d\n", a | b);



Implicit convertions

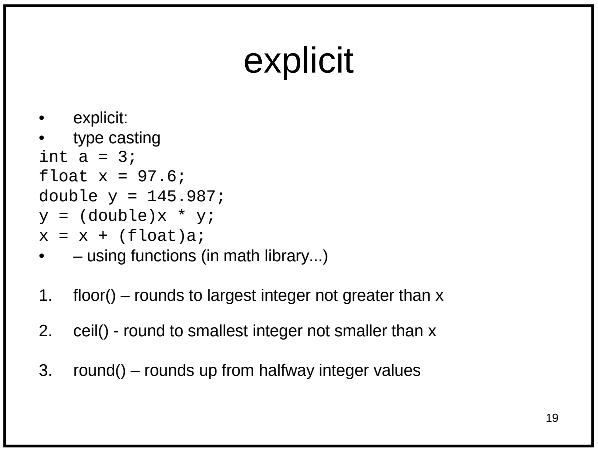
- implicit:

```
int a = 1;
char b = 97; // converts int to char
int s = a + b; // adds int and char, converts to int
```

- promotion: char -> short -> int -> float -> double
- if one operand is double, the other is made double
- else if either is float, the other is made float

```
int a = 3;
float x = 97.6;
double y = 145.987;
y = x * y; // x becomes double; result is double
x = x + a; // a becomes float; result is float
```

- real (float or double) to int truncates



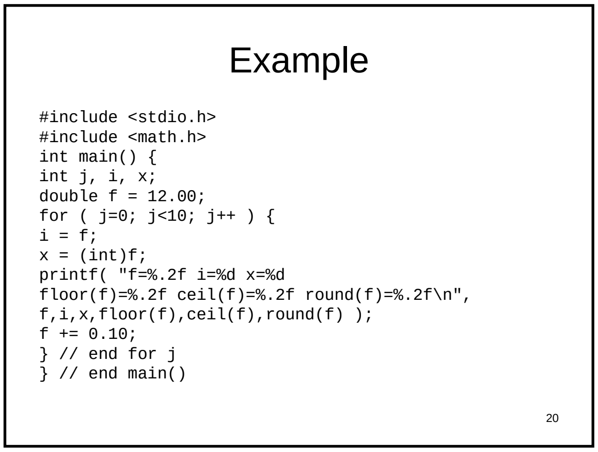
explicit

- explicit:
- type casting

```
int a = 3;
float x = 97.6;
double y = 145.987;
y = (double)x * y;
x = x + (float)a;
```

- – using functions (in math library...)

1. floor() – rounds to largest integer not greater than x
2. ceil() - round to smallest integer not smaller than x
3. round() – rounds up from halfway integer values



Example

```
#include <stdio.h>
#include <math.h>
int main() {
int j, i, x;
double f = 12.00;
for ( j=0; j<10; j++ ) {
i = f;
x = (int)f;
printf( "f=%.2f i=%d x=%d
floor(f)=%.2f ceil(f)=%.2f round(f)=%.2f\n",
f,i,x,floor(f),ceil(f),round(f) );
f += 0.10;
} // end for j
} // end main()
```

Output

- f=12.00 i=12 x=12 floor(f)=12.00 ceil(f)=12.00 round(f)=12.00
- f=12.10 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.20 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.30 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.40 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.50 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.60 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00
- f=12.70 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00
- f=12.80 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00
- f=12.90 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00

Be aware

- almost any conversion does something— but not necessarily what you intended!!
- – example:

```
int x = 100000;
short s = x;
printf("%d %d\n", x, s);
```

- – output is:

```
100000 -31072
```

- WHY?

math library

- Functions ceil() and floor() come from the math library
- definitions:
  - ceil(x): returns the smallest integer not less than x, as a double
  - floor(x): returns the largest integer not greater than x, as a double
- in order to use these functions, you need to do two things:

1. include the prototypes (i.e., function definitions) in the source code:
   #include <math.h>
2. include the library (i.e., functions' object code) at link time:
   unix$ gcc abcd.c -lm

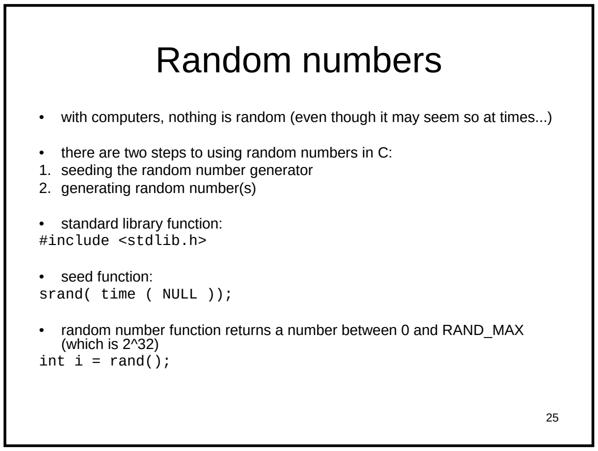
- exercise: can you write a program that rounds a floating point?

math

- some other functions from the math library (these are function prototypes):
  - double sqrt(double x);
  - double pow(double x, double y);
  - double exp(double x);
  - double log(double x);
  - double sin(double x);
  - double cos(double x);

- exercise: write a program that calls each of these functions

- questions:
  - can you make sense of /usr/include/math.h?
  - where are the definitions of the above functions?
  - what are other math library functions?



Random numbers

- with computers, nothing is random (even though it may seem so at times...)

- there are two steps to using random numbers in C:

1. seeding the random number generator
2. generating random number(s)

- standard library function:

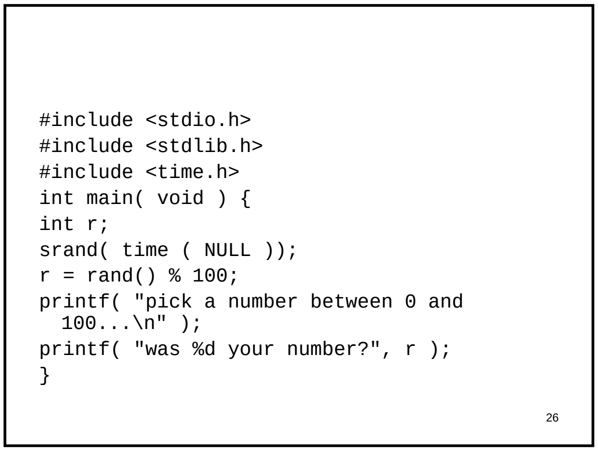
```
#include <stdlib.h>
```

- seed function:

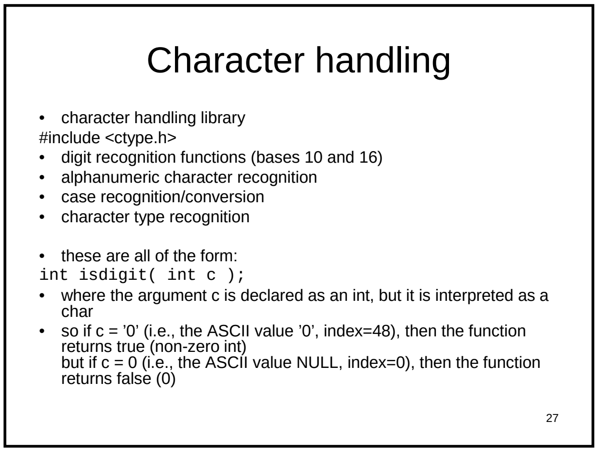
```
srand( time ( NULL ));
```

- random number function returns a number between 0 and RAND_MAX (which is 2^32)

```
int i = rand();
```



```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main( void ) {
int r;
srand( time ( NULL ));
r = rand() % 100;
printf( "pick a number between 0 and
  100...\n" );
printf( "was %d your number?", r );
}
```



Character handling

- character handling library

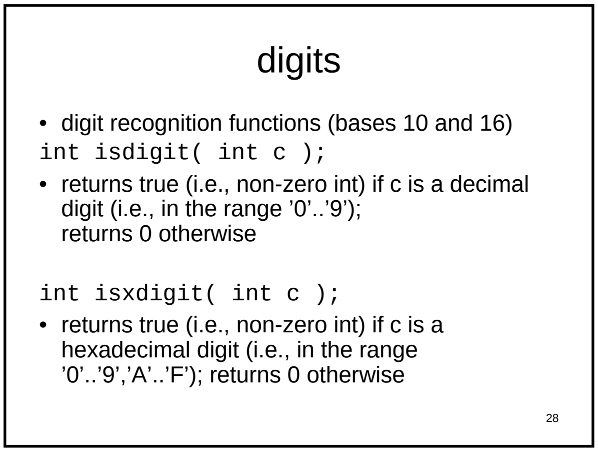
#include <ctype.h>

- digit recognition functions (bases 10 and 16)
- alphanumeric character recognition
- case recognition/conversion
- character type recognition

- these are all of the form:

```
int isdigit( int c );
```

- where the argument c is declared as an int, but it is interpreted as a char
- so if c = '0' (i.e., the ASCII value '0', index=48), then the function returns true (non-zero int)
  but if c = 0 (i.e., the ASCII value NULL, index=0), then the function returns false (0)



digits

- digit recognition functions (bases 10 and 16)

```
int isdigit( int c );
```

- returns true (i.e., non-zero int) if c is a decimal digit (i.e., in the range '0'..'9');
  returns 0 otherwise

```
int isxdigit( int c );
```

- returns true (i.e., non-zero int) if c is a hexadecimal digit (i.e., in the range '0'..'9','A'..'F'); returns 0 otherwise

Alpha numeric

- alphanumeric character recognition

`int isalpha( int c );`

- returns true (i.e., non-zero int) if c is a letter (i.e., in the range ’A’..’Z’,’a’..’z’); returns 0 otherwise

`int isalnum( int c );`

- returns true (i.e., non-zero int) if c is an alphanumeric character (i.e., in the range ’A’..’Z’,’a’..’z’,’0’..’9’); returns 0 otherwise

Case

- case recognition

`int islower( int c );`

- returns true (i.e., non-zero int) if c is a lowercase letter (i.e., in the range ’a’..’z’); returns 0 otherwise

`int isupper( int c );`

- returns true (i.e., non-zero int) if c is an uppercase letter (i.e., in the range ’A’..’Z’); returns 0 otherwise

- case conversion

`int tolower( int c );`

- returns the value of c converted to a lowercase letter (does nothing if c is not a letter or if c is already lowercase)

`int toupper( int c );`

- returns the value of c converted to an uppercase letter (does nothing if c is not a letter or if c is already uppercase)

types

- character type recognition

`int isspace( int c );`

- returns true (i.e., non-zero int) if c is a space; returns 0 otherwise

`int iscntrl( int c );`

- returns true (i.e., non-zero int) if c is a control character; returns 0 otherwise

`int ispunct( int c );`

- returns true (i.e., non-zero int) if c is a punctuation mark; returns 0 otherwise

`int isprint( int c );`

- returns true (i.e., non-zero int) if c is a printable character; returns 0 otherwise

`int isgraph( int c );`

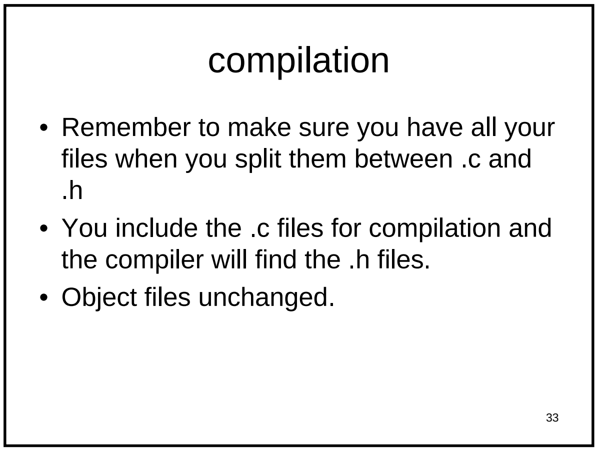
- returns true (i.e., non-zero int) if c is a graphics character; returns 0 otherwise

Header files

- .h files usually used to define methods or centralize defintions

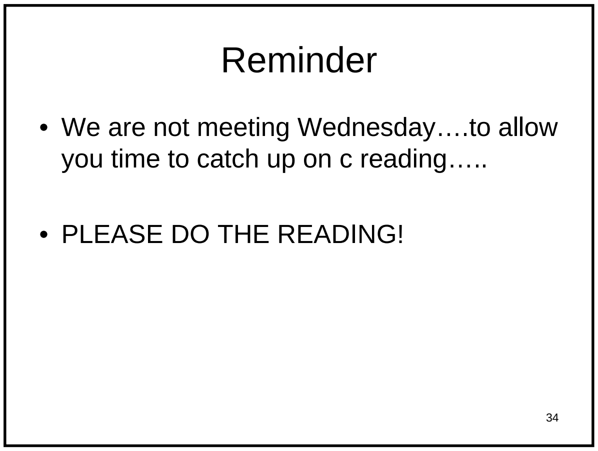
- public int calculateSomething(int []);

- Can either name the variables or not
- int[] vs int ar[]
- In .c file use; #include “something.h”



compilation

- Remember to make sure you have all your files when you split them between .c and .h
- You include the .c files for compilation and the compiler will find the .h files.
- Object files unchanged.



Reminder

- We are not meeting Wednesday….to allow you time to catch up on c reading…..
- PLEASE DO THE READING!