cs3157: another C lecture (mon-21-feb-2005)

- today:
  - C pre-processor
  - command-line arguments
  - more on data types and operators:
    - "booleans" in C
    - logical and bitwise operators
    - type conversion
  - more libraries:
    - math library (`math.h`)
    - random numbers (`stdlib.h`)
    - character handling (`char.h`)
  - intro to arrays and pointers

C pre-processor (1).

- 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

C pre-processor (2).

```
#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
- example: `#define MAXLEN 255`

C pre-processor (3).

- 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...

C pre-processor (4).

- 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( "good for you for running Linux!" );
#else
  puts( "why are you running something else???" );
#endif
```


C pre-processor (5).

- `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

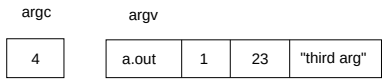
command-line arguments (1).

```
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)

command-line arguments (2).

- Name of executable followed by space-separated arguments
- `unix$ a.out 1 23 "third arg"`
- this is stored like this:



command-line arguments (3).

- If no arguments, simplify:

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

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

more data types and operators: "booleans" in C (1).

- 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
```


more data types and operators: "booleans" in C (2).

- 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" );
}
```


more data types and operators: logical operators.

- in C 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 $\Rightarrow true$
  - zero $\Rightarrow false$

more data types and operators: 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 */
```

```
                1000  (=8)                    1000  (=8)
a & b ⇒      & 1111  (=15)      a | b ⇒     | 1111  (=15)
                1000  (=8)                    1111  (=15)
```


more data types and operators: logical vs bitwise operators.

- 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 );
```

- output is:

```
a && b = 1
a || b = 1
a & b = 4
a | b = 15
```


more data types and operators: implicit type conversion.

- 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

more data types and operators: explicit type conversion.

- explicit:
  - type casting

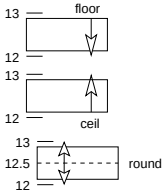
```
int    a = 3;
float  x = 97.6;
double y = 145.987;
y = (double)x * y;
x = x + (float)a;
```

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

`floor()`: rounds to largest integer not greater than x

`ceil()`: rounds to smallest integer not less than x

`round()`: rounds up from halfway between integer values



more data types and operators: example.

- 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()
```


more data types and operators: example (continued).

- 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
```


more data types and operators: caution.

- 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
```


more libraries: the math library (1).

- in the earlier slide, the 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?

more libraries: the math library (2).

- 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?

more libraries: random numbers (stdlib) (1).

- 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();
```


more libraries: random numbers (stdlib) (2).

```
#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 );
}
```


more libraries: character handling (1).

- 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 intepreted 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)

more libraries: character handling (2).

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

more libraries: character handling (3).

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

more libraries: character handling (4).

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)

more libraries: character handling (5).

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

intro to arrays and pointers (1).

- a string is an *array* of characters
- an array is a "regular grouping or ordering"
- a data structure consisting of related elements of the same data type
- in C, an array has a length associated with it
- arrays need:
  - data type
  - name
  - length
- length can be determined:
  - *statically* — at compile time
    e.g., `char str1[10];`
  - *dynamically* — at run time
    e.g., `char *str2;`

intro to arrays and pointers (2).

- defining a variable is called "allocating memory" to store that variable
- defining an array means allocating memory for a group of bytes, i.e., assigning a label to the first byte in the group
- individual array elements are *indexed*
  - starting with 0
  - ending with $length - 1$
- indeces follow array name, enclosed in square brackets (`[ ]`)
  e.g., `arr[25]`

intro to arrays and pointers (3).

integer array example

```
#include <stdio.h>
#define MAX 6
int main( void ) {
  int arr[MAX] = { -45, 6, 0, 72, 1543, 62 };
  int i;
  for ( i=0; i<MAX; i++ ) {
    printf( "%d", arr[i] );
  }
  printf( "\n" );
} /* end of main() */
```


intro to arrays and pointers (4).

- variables that contain memory addresses as their values
- other data types we've learned about in C use *direct* addressing
- pointers facilitate *indirect* addressing
- declaring pointers:
  - pointers indirectly address memory where data of the types we've already discussed is stored (e.g., int, char, float, etc.)
  - declaration uses asterisks (*) to indicate a pointer to a memory location storing a particular data type
- example:

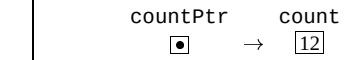
```
int *count;
float *avg;
```

intro to arrays and pointers (5).

- ampersand & is used to get the address of a variable
- example:

```
int count = 12;
int *countPtr = &count;
```

- `&count` returns the *address* of `count` and stores it in the pointer variable `countPtr`
- a picture:



intro to arrays and pointers (6).

here's another example:

```
int i = 3, j = -99;
int count = 12;
int *countPtr = &count;
```

and here's what the memory looks like:

variable name	memory location	value
`count`	0xbffff4f0	12
`i`	0xbffff4f4	3
`j`	0xbffff4f8	-99
...		
`countPtr`	0xbffff600	0xbffff4f0
...		

intro to arrays and pointers (7).

- an array is some number of contiguous memory locations
- an array definition is really a pointer to the starting memory location of the array
- and pointers are really integers
- so you can perform integer arithmetic on them
- e.g., +1 increments a pointer, -1 decrements
- you can use this to move from one array element to another

intro to arrays and pointers (8).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
  int i, *j, arr[5];
  srand( time ( NULL ));
  for ( i=0; i<5; i++ )
    arr[i] = rand() % 100;
  printf( "arr=%p\n",arr );
  for ( i=0; i<5; i++ ) {
    printf( "i=%d arr[i]=%d &arr[i]=%p\n",i,arr[i],&arr[i] );
  }
  j = &arr[0];
  printf( "\nj=%p *j=%d\n",j,*j );
  j++;
  printf( "after adding 1 to j:\n j=%p *j=%d\n",j,*j );
}
```

intro to arrays and pointers (9).

and the output is...

```
arr=0xbffff4f0
i=0 arr[i]=29 &arr[i]=0xbffff4f0
i=1 arr[i]=8 &arr[i]=0xbffff4f4
i=2 arr[i]=18 &arr[i]=0xbffff4f8
i=3 arr[i]=95 &arr[i]=0xbffff4fc
i=4 arr[i]=48 &arr[i]=0xbffff500

j=0xbffff4f0 *j=29
after adding 1 to j:
j=0xbffff4f4 *j=8
```