Many slides of this lecture are adapted from Lewis Girod, CENS Systems Lab
http://lecs.cs.ucla.edu/~girod/talks/c-tutorial.ppt
and Clark Barrett
In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of *The C Programming Language* by Kernighan & Ritchie caused a revolution in the computing world
Why C?

• Mainly because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:
  – Operating Systems
  – Language Compilers
  – Assemblers
  – Text Editors
  – Print Spoolers
  – Network Drivers
  – Language Interpreters
  – Utilities
Your first goal: Learn C!

• Resources
  – These lectures
  – Additional online resources (some links on the website)

• Learning a Programming Language
  – The best way to learn is to write programs
Writing and Running Programs

```c
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

1. Write text of program (source code) using an editor such as emacs, save as file e.g. my_program.c

2. Run the compiler to convert program from source to an “executable” or “binary”:
   ```bash
   $ gcc -Wall -g -o my_program my_program.c
   ```

3. Compiler gives errors and warnings; edit source file, fix it, and re-compile

   Run it and see if it works 😊
   ```bash
   $ ./my_program
   Hello World
   $```

$ gcc -Wall -g -o my_program my_program.c

- generate all warnings
- keep debugging information
- name the generated executable (default: a.out)
- one or more C files
About C

• Procedural language
  – Functions calling each other, starting with main().
• Case-sensitive
• Four stages
  – Editing: Writing the source code by using some IDE or editor
  – Preprocessing
  – compiling: translates or converts source to object code for a specific platform source code -> object code
  – linking: resolves external references and produces the executable module
#include <stdio.h>

/* The simplest C Program */

int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}

#include inserts another file. ".h" files are called "header" files. They contain stuff needed to interface to libraries and code in other "c" files.

This is a comment. The compiler ignores this.

The main() function is always where your program starts running.

Blocks of code are marked by 
{
... 
}

Return ‘0’ from this function

Print out a message. ‘\n’ means “new line”.

Get started with C programming!
The simplest C Program

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

Preprocessed C code:

```c
__extension__ typedef unsigned long long int __dev_t;
__extension__ typedef unsigned int __uid_t;
__extension__ typedef unsigned int __gid_t;
__extension__ typedef unsigned long int __ino_t;
__extension__ typedef unsigned long long int __ino64_t;
__extension__ typedef unsigned int __nlink_t;
__extension__ typedef long int __off_t;
__extension__ typedef long long int __off64_t;
extern void flockfile (FILE *__stream) ;
extern int ftrylockfile (FILE *__stream) ;
extern void funlockfile (FILE *__stream) ;
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```
In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out ( /* */ , // )
- Continued lines (i.e. very long lines ) are joined ( \ )
#include <stdio.h>

/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}

my_program

__extension__ typedef  unsigned long long int   __dev_t;
__extension__ typedef  unsigned int   __uid_t;
__extension__ typedef  unsigned int   __gid_t;
__extension__ typedef  unsigned long int   __ino_t;
__extension__ typedef  unsigned long long int   __ino64_t;
__extension__ typedef  unsigned int   __nlink_t;
__extension__ typedef  long int   __off_t;
__extension__ typedef  long long int   __off64_t;
extern void flockfile (FILE *__stream)  ;
extern int ftrylockfile (FILE *__stream)  ;
extern void funlockfile (FILE *__stream)  ;
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}

• The compiler then converts the resulting text into binary code the CPU can run directly.
• The compilation process involves really several steps:
  • **Compiler**: high level language \(\rightarrow\) assembly
  • **Assembler**: assembly \(\rightarrow\) machine code
  • **Linker**: links all machine code files and needed libraries into one executable file.
• When you type `gcc` you really invoke the compiler, assembler, and linker.
What is “Memory”?

- Is like a big table of numbered slots.
- Each slot stores a byte.

- The number of a slot is its Address.
- One byte Value can be stored in each slot.

Some “logical” data values span more than one slot, like the character string “Hello\n”

A Type names a logical meaning to a span of memory. Some simple types are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>a single character (1 slot)</td>
</tr>
<tr>
<td>char [10]</td>
<td>an array of 10 characters</td>
</tr>
<tr>
<td>int</td>
<td>signed 4 byte integer</td>
</tr>
<tr>
<td>float</td>
<td>4 byte floating point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Define a variable by giving it a name and specifying the type, and optionally an initial value.

```
char x;
char y = 'e';
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>'e' (101)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Initial value of x is undefined
Initial value of y is 'e'

Type is single character (char)
The compiler puts them somewhere in memory.
Multi-byte Variables

Different types consume different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char)

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

An int consumes 4 bytes

padding
Every Variable is Defined within some scope. A Variable cannot be referenced from outside of that scope.

Scopes are defined with curly braces {}.

The scope of Function Arguments is the complete body of the function.

The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block.

The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called Global Vars.

```c
void p(char x) {
    char y;
    char z;
}
char z;
void q(char a) {
    char b;
    {
        char c;
    }
    char d;
}
```
Now that we know about variables, let's combine them to form expressions!
**Expressions and Evaluation**

Expressions combine Values using Operators, according to precedence.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluation 1</th>
<th>Evaluation 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 + 2 \times 2$</td>
<td>$1 + 4$</td>
<td>$5$</td>
<td></td>
</tr>
<tr>
<td>$(1 + 2) \times 2$</td>
<td>$3 \times 2$</td>
<td>$6$</td>
<td></td>
</tr>
</tbody>
</table>

Comparison operators are used to compare values.
In C, $0$ means “false”, and *any other value* means “true”.

```c
int x=4;
(x < 5) \rightarrow (4 < 5) \rightarrow \text{<true>}
(x < 4) \rightarrow (4 < 4) \rightarrow 0
((x < 5) || (x < 4)) \rightarrow (\text{<true>} || (x < 4)) \rightarrow \text{<true>}
```

Not evaluated because first clause was true
Precedence

• **Highest to lowest**
  • ()
  • *, /, %
  • +, -

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. *When in doubt, add parentheses to make it explicit.*
### Comparison and Mathematical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical not</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
</tr>
<tr>
<td>*</td>
<td>mult</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise xor</td>
</tr>
<tr>
<td>~</td>
<td>bitwise not</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
</tbody>
</table>

### Beware in division:

- If second argument is integer, the result will be integer (rounded):
  
  \[
  5 / 10 \rightarrow 0 \text{ whereas } 5 / 10.0 \rightarrow 0.5
  \]

### Don’t confuse & and &&:

\[
1 \& 2 \rightarrow 0 \text{ whereas } 1 \&\& 2 \rightarrow \text{<true>}
\]
Assignment Operators

\[ x = y \quad \text{assign y to x} \]
\[ x++ \quad \text{post-increment x} \]
\[ ++x \quad \text{pre-increment x} \]
\[ x-- \quad \text{post-decrement x} \]
\[ --x \quad \text{pre-decrement x} \]

x += y \quad \text{assign (x+y) to x}

Note the difference between ++x and x++:

```plaintext
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

Don’t confuse = and ==

```plaintext
int x=5;
if (x==6) /* false */
{
    /* ... */
}
/* x is still 5 */
```

```plaintext
int x=5;
if (x=6) /* always true */
{
    /* x is now 6 */
}
/* ... */
```
Functions
What is a Function?

A **Function** is a series of instructions to run. You pass **Arguments** to a function and it returns a **Value**.

“**main()**” is a Function. It’s only special because it always gets called first when you run your program.

```
#include <stdio.h>

/* The simplest C Program */

int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

“**printf()**” is just another function, like **main()**. It’s defined for you in a “**library**”, a collection of functions you can call from your program.

**Return type, or void**

**Function Arguments**

**Returning a value**
A More Complex Program: pow

“if” statement

```c
/* if evaluated expression is not 0 */
if (expression) {
    /* then execute this block */
} else {
    /* otherwise execute this block */
}
```

Tracing “pow()”:
- What does pow(5,0) do?
- What about pow(5,1)?
Recall scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

Yes. Each function call allocates a “stack frame” where Variables within that function’s scope will reside.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>5.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>0</td>
</tr>
<tr>
<td>float x</td>
<td>5.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>1</td>
</tr>
<tr>
<td>int argc</td>
<td>1</td>
</tr>
<tr>
<td>char **argv</td>
<td>0x2342</td>
</tr>
<tr>
<td>float p</td>
<td>5.0</td>
</tr>
</tbody>
</table>
The “for” loop

The “for” loop is just shorthand for this “while” loop structure.

```c
float pow(float x, uint exp)
{
    float result=1.0;
    int i;
    i=0;
    while (i < exp) {
        result = result * x;
        i++;
    }
    return result;
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```
When to Use

Different Loop-constructs

• while
• do-while
• for
When to Use

Conditions

• if-else
• switch-case
Very strong but dangerous concept!
Can a function modify its arguments?

What if we wanted to implement a function `pow_assign()` that modified its argument, so that these are equivalent:

```c
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```c
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```

Would this work?

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}

main()
{
  float p=2.0;
  pow_assign(p, 5);
}

In C, all arguments are passed as values
But, what if the argument is the address of a variable?

Remember the stack!

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x</td>
<td>32.0</td>
</tr>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>32.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
## Passing Addresses

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char x</td>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>char y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

address of \( x \to 4 \)
memory content at address \( 4 \to 72 \)
"Pointers"

This is exactly how “pointers” work.

- address of x: &x
- if y is an address, the content of the memory at that address \( \rightarrow \) *y

```c
void f(char * p)
{
    *p = *p - 32;
}
```

```c
char y = 101; /* y is 101 */
f(&y); /* i.e. f(5) */ /* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:
- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions
A **Valid** pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:
- Program errors that set the pointer value to a strange number
- Use of a pointer that was at one time valid, but later became invalid

**Will ptr be valid or invalid?**

```c
cchar * get_pointer()
{
    char x=0;
    return &x;
}

main()
{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
}
```
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}
main()
{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!
Now that we know pointers (I hope!),
let’s go back to types.
More on Types

We’ve seen a few types at this point: char, int, float, char *

Types are important because:
• They allow your program to impose logical structure on memory
• They help the compiler tell when you’re making a mistake

In the next slides we will discuss:
• How to create logical layouts of different types (structs)
• How to use arrays
• How to parse C type names (there is a logic to it!)
• How to create new types using typedef
Structures

• a collection of related data items
• possibly of different types
• defined using the keyword `struct`
• The members of a struct type variable are accessed with the dot (.) operator:
  – `<struct-variable>.<member_name>`;
struct basics

• Definition of a structure:

```c
struct <struct-type>{
    <type> <identifier_list>;
    <type> <identifier_list>;
    ...
} ;
```

Each identifier defines a member of the structure.
struct basics

- Example:

```c
struct Address {
    int zip;
    char street[50];
    char city[20];
} ;

main()
{
    struct Address adrs;
    ...
    adrs.zip = 10012;
}
```

Example of initializing a structure:

```c
struct Address adrs = {10012, "Mercer", "New York"};
```
Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* accessing element 0 */
x[0] = 'T';

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3); /* x[3] */

/* x[0] evaluates to the first element; 
* x evaluates to the address of the 
* first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
    /* process y[i] */
    printf("%c\n", y[i]);
}

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0..9)

x[3] == *(x+3) == ‘t’  (NOT ‘s’!)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char x [0]</td>
<td>100</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [1]</td>
<td>101</td>
<td>‘e’</td>
</tr>
<tr>
<td>char x [2]</td>
<td>102</td>
<td>‘s’</td>
</tr>
<tr>
<td>char x [3]</td>
<td>103</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [4]</td>
<td>104</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>

For loop that iterates from 0 to COUNT-1.
Memorize it!
Pointers and Arrays in C

• An array name by itself is an address, or pointer in C.

• When an array is declared, the compiler allocates sufficient space beginning with some base address to accommodate every element in the array.

• The base address of the array is the address of the first element in the array (index position 0).
  
  – Example: int num[10];
    
    &num[0] is the same as num
Pointers and Arrays in C

• Suppose we define the following array and pointer:

```c
int a[100]; int *ptr;
```

Assume that the system allocates memory bytes 400, 404, 408, ..., 796 to the array. Recall that integers are allocated 32 bits = 4 bytes.

- The two statements: `ptr = a;` and `ptr = &a[0];` are equivalent and would assign the value of 400 to `ptr`.

• Pointer arithmetic provides an alternative to array indexing in C.

- The two statements: `ptr = a + 1;` and `ptr = &a[1];` are equivalent and would assign the value of 404 to `ptr`.
Pointers and Arrays in C

• Assuming the elements of the array have been assigned values, the following code would sum the elements of the array:

```c
sum = 0;
for (ptr = a; ptr < &a[100]; ++ptr)
    sum += *ptr;
```

• Here is another way to sum the array:

```c
sum = 0;
for (i = 0; i < 100; ++i)
    sum += *(a + i);
```

*a[b]* in C is just syntactic sugar for 
*(*(a + b)*
Strings

• Series of characters treated as a single unit
• Can include letters, digits, and certain special characters (*, /, $)
• String literal (string constant) - written in double quotes
  – "Hello"
• Strings are arrays of characters
• Example:
  – char name[] = “test”;
  – address of the above string can be expressed in two ways:
    • &name[0]
    • name
Strings

• String declarations
  – Declare as a character array or a variable of type `char *`
    ```c
    char color[] = "blue";
    char *colorPtr = "blue";
    ```
  – Remember that strings represented as character arrays end with `'\0'`
    • `color` has 5 elements

• Inputting strings
  – Use `scanf`
    ```c
    scanf("%s", word);
    ```
    • Copies input into `word[]`, which does not need `&` (because a string is a pointer)
  – Remember to leave space for `'\0'`
## Character Handling Library

- In `<ctype.h>`

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int isdigit( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a digit and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int isalpha( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a letter and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int isalnum( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a digit or a letter and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int isxdigit( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a hexadecimal digit character and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int islower( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a lowercase letter and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int isupper( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is an uppercase letter; <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int tolower( int c )</td>
<td>If <code>c</code> is an uppercase letter, <code>tolower</code> returns <code>c</code> as a lowercase letter. Otherwise, <code>tolower</code> returns the argument unchanged.</td>
</tr>
<tr>
<td>int toupper( int c )</td>
<td>If <code>c</code> is a lowercase letter, <code>toupper</code> returns <code>c</code> as an uppercase letter. Otherwise, <code>toupper</code> returns the argument unchanged.</td>
</tr>
<tr>
<td>int isspace( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a white-space character—newline (<code>\n</code>), space (<code> </code>), form feed (<code>\f</code>), carriage return (<code>\r</code>), horizontal tab (<code>\t</code>), or vertical tab (<code>\v</code>)—and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int iscntrl( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a control character and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int ispunct( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a printing character other than a space, a digit, or a letter and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int isprint( int c )</td>
<td>Returns <code>true</code> value if <code>c</code> is a printing character including space (<code> </code>) and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>int isgraph( int c )</td>
<td>Returns <code>true</code> if <code>c</code> is a printing character other than space (<code> </code>) and <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>

Each function receives a character (an `int`) or `EOF` as an argument.
String Conversion Functions

• in `<string.h>`

• Conversion functions
  – In `<stdlib.h>` (general utilities library)
  – Convert strings of digits to integer and floating-point values

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double atof( const char *nPtr )</td>
<td>Converts the string nPtr to double.</td>
</tr>
<tr>
<td>int atoi( const char *nPtr )</td>
<td>Converts the string nPtr to int.</td>
</tr>
<tr>
<td>long atol( const char *nPtr )</td>
<td>Converts the string nPtr to long int.</td>
</tr>
<tr>
<td>double strtod( const char *nPtr, char **endPtr )</td>
<td>Converts the string nPtr to double.</td>
</tr>
<tr>
<td>long strtol( const char *nPtr, char **endPtr, int base )</td>
<td>Converts the string nPtr to long.</td>
</tr>
<tr>
<td>unsigned long strtoul( const char *nPtr, char **endPtr, int base )</td>
<td>Converts the string nPtr to unsigned long.</td>
</tr>
</tbody>
</table>
String Manipulation Functions

- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Determine string length

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *strcpy( char *s1, const char *s2 )</td>
<td>Copies string s2 into array s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strncpy( char *s1, const char *s2, size_t n )</td>
<td>Copies at most n characters of string s2 into array s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strcat( char *s1, const char *s2 )</td>
<td>Appends string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strncat( char *s1, const char *s2, size_t n )</td>
<td>Appends at most n characters of string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.</td>
</tr>
</tbody>
</table>
String Manipulation Functions

```c
int strcmp ( const char * str1, const char * str2 )
```

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;0</td>
<td>The first character that does not match has a lower value in <code>ptr1</code> than in <code>ptr2</code></td>
</tr>
<tr>
<td>0</td>
<td>The contents of both strings are equal</td>
</tr>
<tr>
<td>&gt;0</td>
<td>The first character that does not match has a greater value in <code>ptr1</code> than in <code>ptr2</code></td>
</tr>
</tbody>
</table>
How to Parse C Types

C type names are parsed by **starting at the name** and working outwards according to the rules of precedence:

```c
int (*x)[10];
```

- `x` is a pointer to an array of `int`

```c
int *x[10];
```

- `x` is an array of pointers to `int`
Using typedef

At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

```c
int x;       /* int; */ typedef int T;
int *y;      /* pointer to int; */ typedef int *U;
int z[10];   /* array of ints; */ typedef int V[10];
int *k[10];  /* array of pointers to int; */ typedef int *W[10];
int (*m)[10]; /* pointer to array of ints; */ typedef int (*N)[10];
```

typedef defines a new type

Now:

```c
T x; is the same as int x;
U y; is the same as int * y;
```
and so on ...
What if you want to allocate an array of N elements, and you don’t know N beforehand?
Dynamic Memory Allocation

So far all of our examples have allocated variables statically by defining them in our program. This allocates them in the stack.

But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires dynamic allocation.

```c
int * alloc_ints(size_t requested_count) {
    int * big_array;
    big_array = (int *)calloc(requested_count, sizeof(int));
    if (big_array == NULL) {
        printf("can't allocate %d ints: %m\n", requested_count);
        return NULL;
    }
    /* now big_array[0] .. big_array[requested_count-1] are * valid and zeroed. */
    return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can’t alloc

sizeof() reports the size of a type in bytes

It’s OK to return this pointer. It will remain valid until it is freed with free()
Dynamic Memory Allocation

• `void *malloc (size_t size);`
• `void* calloc (size_t num, size_t size);`
• `void free (void* ptr);`
• Unary operator `sizeof` is used to determine the size in bytes of any data type. Examples:
  – `sizeof(double)`
  – `sizeof(int)`
Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()’d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a “memory leak”.

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic
I/O in C
I/O

• reading from:
  – standard input (usually the keyboard)
  – file

• writing to:
  – standard output (usually the screen)
  – file

• A library of functions is supplied to perform these operations.

• The I/O library functions are listed in the header file <stdio.h>.
Writing to stdout

`printf();`

- This function provides for formatted output to the screen. The syntax is:
  
  ```c
  printf ( "format", var1, var2, ... );
  ```

- The "format" includes a listing of the data types of the variables to be output and, optionally, some text and control character(s).

- Example:

  ```c
  float a; int b;
  scanf ( "%f%d", &a, &b );
  printf ( "You entered %f and %d \n", a, b );
  ```
Formatted Output with printf

Format Conversion Specifiers (This list is not exhaustive):

**d** -- displays a decimal (base 10) **integer**

**l** -- used with other specifiers to indicate a **long**

**f** -- displays a **floating point value**

**x** -- displays a number **hexadecimal** format

**c** -- displays a **single character**

**s** -- displays a **string** of characters
scanf ();

• This function provides for formatted input from the keyboard. The syntax is:
  
  `scanf ( "format" , &var1, &var2, ...);`

• The “format” is a listing of the data types of the variables to be input and the & in front of each variable name tells the system WHERE to store the value that is input. It provides the address for the variable.

• Example:
  
  ```c
  float a; int b;
  scanf ("%f%d", &a, &b);
  ```
Files

• In C, each file is simply a sequential stream of bytes.
• C imposes no structure on a file.
• Steps to deal with files
  – open a file
  – check that the open was successful
  – read/write to a file
  – close a file
First step

• Declaration:

    FILE *fptr1, *fptr2 ;
Opening Files

• The statement:
  
  \[
  \text{fptr1} = \text{fopen} \left( \text{"filename"}, \text{"r"} \right);
  \]
  
  would open the file filename for input (reading).

  – r: read
  – w: write
  – a: append
  – ... there are some more
Testing for Successful Open

• If the file was not able to be opened, then the value returned by the `fopen` routine is NULL.

• For example, let's assume that the file *mydata* does not exist. Then:

```c
FILE *fptr1;
fptr1 = fopen("myfile", "r");
if (fptr1 == NULL)
{
    printf("File 'mydata' did not open.\n");
}
```
Reading From Files

• In the following segment of C language code:

```c
int a, b;
FILE *fptr1, *fptr2;
fptr1 = fopen( "mydata", "r" );
fscanf( fptr1, "%d%d", &a, &b );
```

the `fscanf` function would read values from the file "pointed" to by `fptr1` and assign those values to `a` and `b`. 
End of File

- The end-of-file indicator informs the program when there are no more data (no more bytes) to be processed.
- There are a number of ways to test for the end-of-file condition. One is to use the `feof` function which returns a true or false condition:
  ```c
  fscanf (fptr1, "%d", &var) ;
  if ( feof (fptr1) )
  {
    printf ("End-of-file encountered.\n");
  }
  ```
- Another (better) way of testing EOF:
  ```c
  while(fscanf(fp,"%d ", &current) == 1) 
  {
  }
  ```
Writing To Files

int a = 5, b = 30;
FILE *fptr2;
fptr2 = fopen ( "filename", "w" );
fprintf ( fptr2, "%d %d\n", a, b );

the `fprintf` functions would write the values stored in \textit{a} and \textit{b} to the file "pointed" to by \textit{fptr2}. 
Closing Files

fclose ( fptr1 );

Once the files are open, they stay open until you close them or end the program (which will close all files.)
One last concept ...
Macros can be a useful way to customize your interface to C and make your code easier to read and less redundant. However, when possible, use a static inline function instead.

```c
/* Macros are used to define constants */
#define FUDGE_FACTOR 45.6
#define MSEC_PER_SEC 1000
#define INPUT_FILENAME "my_input_file"

/* Macros are used to do constant arithmetic */
#define TIMER_VAL (2*MSEC_PER_SEC)

/* Macros are used to capture information from the compiler */
#define DBG(args...) do { 
  fprintf(stderr, "%s:%s:%d: ", __FUNCTION__, __FILE__, __LINENO__); 
  fprintf(stderr, args...); 
} while (0)

/* ex. DBG("error: %d", errno); */
```

Float constants must have a decimal point, else they are type int

Put expressions in parens.

Multi-line macros need `\`

args... grabs rest of args

Enclose multi-statement macros in do{while(0)
Conclusions

• We took a quick look at the different features of C
• To get deeper look: check online tutorials. You will find some links at the course webpage
• To become an expert: write code ... write code ... write code