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
$ ```
About C

- Hardware independent
- Programs portable to most computers
- Case-sensitive
- Four stages
  - **Editing**: Writing the source code by using some IDE or editor
  - **Preprocessing or libraries**: Already available routines
  - **Compiling**: translates or converts source to object code for a specific platform
  - **Linking**: resolves external references and produces the executable module
C Syntax and Hello World

```c
#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 `{ ... }`
- Can your program have more than one .c file?
- What do the <> mean?

Print out a message. '\n' means "new line".

Return '0' from this function.
Preprocessing

Compilation occurs in two steps: “Preprocessing” and “Compiling”

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 are joined ( \
)

The compiler then converts the resulting text into binary code the CPU can run directly.
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**
What is “Memory”? 

Memory is like a big table of numbered slots where bytes can be stored.

<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>

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:

- **char** — a single character (1 slot)
- **char [10]** — an array of 10 characters
- **int** — signed 4 byte integer
- **float** — 4 byte floating point
- **int64_t** — signed 8 byte integer
## 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.

### Example

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

<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>
</tbody>
</table>

The initial value of `x` is undefined. The compiler puts them somewhere in memory.
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

An int consumes 4 bytes

<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>
Lexical Scoping

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

Lexical 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.

(Returns nothing)

```c
void p(char x) {
    char y;
    char z;
}
char z;

void q(char a) {
    char b;
    {
        char c;
    }
    char d;
}
```
Expressions and Evaluation

Expressions combine Values using Operators, according to precedence.

\[ 1 + 2 \times 2 \rightarrow 1 + 4 \rightarrow 5 \]

\[ (1 + 2) \times 2 \rightarrow 3 \times 2 \rightarrow 6 \]

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 <true>
(x < 4) \rightarrow (4 < 4) \rightarrow 0
((x < 5) || (x < 4)) \rightarrow (<true> || (x < 4)) \rightarrow <true>
```

Not evaluated because first clause was true
Precedence

• Highest to lowest
  • ()
  • *, /, %
  • +, -
Comparison and Mathematical Operators

== equal to
< less than
<= less than or equal
> greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
! logical not

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

Beware division:
• If second argument is integer, the result will be integer (rounded):
  5 / 10 → 0 whereas 5 / 10.0 → 0.5
• Division by 0 will cause a FPE

Don’t confuse & and &&..
1 & 2 → 0 whereas 1 && 2 → <true>
Assignment Operators

x = y    assign y to x
x++     post-increment x
++x     pre-increment x
x--     post-decrement x
--x     pre-decrement x

x += y    assign (x+y) to x
x -= y    assign (x-y) to x
x *= y    assign (x*y) to x
x /= y    assign (x/y) to x
x % y    assign (x%y) to x

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

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

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

Don’t confuse = and ==

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

int x=5;
if (x==6)    /* always true */
{
    /* x is now 6 */
}
/* ... */
A More Complex Program: pow

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

float pow(float x, uint32_t exp)
{
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* "recursive" case */
    return x*pow(x, exp-1);
}

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

“if” statement

```
/* 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)?

Tracing “pow()”:
- What does pow(5,0) do?
- What about pow(5,1)?
Recall lexical 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.
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;
}
```

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

Different Loop-constructs
- while
- do-while
- for

Conditions
- if-else
- switch-case
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;
}

{ 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!
## 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: 4
- memory content at 4: 72
This is exactly how “pointers” work.

A “pointer type”: pointer to char

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

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

{  
    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;
}

{ 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!
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
struct: a way to compose existing types into a structure

#include <sys/time.h>

/* declare the struct */
struct my_struct {
    int counter;
    float average;
    struct timeval timestamp;
    uint in_use:1;
    uint8_t data[0];
};

/* define an instance of my_struct */
struct my_struct x = {
    in_use: 1,
    timestamp: {
        tv_sec: 200
    }
};

x.counter = 1;
x.average = sum / (float)(x.counter);

struct my_struct * ptr = &x;
ptr->counter = 2;
(*ptr).counter = 3;  /* equiv. */
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.

```c
/* 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 Data Structure

- Strings
- Arrays
- Linked lists
- ...
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 true if <code>c</code> is a digit and false otherwise.</td>
</tr>
<tr>
<td>int isalpha( int c )</td>
<td>Returns true if <code>c</code> is a letter and false otherwise.</td>
</tr>
<tr>
<td>int isalnum( int c )</td>
<td>Returns true if <code>c</code> is a digit or a letter and false otherwise.</td>
</tr>
<tr>
<td>int isxdigit( int c )</td>
<td>Returns true if <code>c</code> is a hexadecimal digit character and false otherwise.</td>
</tr>
<tr>
<td>int islower( int c )</td>
<td>Returns true if <code>c</code> is a lowercase letter and false otherwise.</td>
</tr>
<tr>
<td>int isupper( int c )</td>
<td>Returns true if <code>c</code> is an uppercase letter; false 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 true if <code>c</code> is a white-space character—newline (\n), space ( \ ), form feed (\f), carriage return (\r), horizontal tab (\t), or vertical tab (\v)—and false otherwise.</td>
</tr>
<tr>
<td>int iscntrl( int c )</td>
<td>Returns true if <code>c</code> is a control character and false otherwise.</td>
</tr>
<tr>
<td>int ispunct( int c )</td>
<td>Returns true if <code>c</code> is a printing character other than a space, a digit, or a letter and false otherwise.</td>
</tr>
<tr>
<td>int isprint( int c )</td>
<td>Returns true value if <code>c</code> is a printing character including space (\ ) and false otherwise.</td>
</tr>
<tr>
<td>int isgraph( int c )</td>
<td>Returns true if <code>c</code> is a printing character other than space (\ ) and false 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 <code>nPtr</code> to double.</td>
</tr>
<tr>
<td>int atoi( const char *nPtr )</td>
<td>Converts the string <code>nPtr</code> to int.</td>
</tr>
<tr>
<td>long atol( const char *nPtr )</td>
<td>Converts the string <code>nPtr</code> to long int.</td>
</tr>
<tr>
<td>double strtod( const char *nPtr, char **endPtr )</td>
<td>Converts the string <code>nPtr</code> to double.</td>
</tr>
<tr>
<td>long strtol( const char *nPtr, char **endPtr, int base )</td>
<td>Converts the string <code>nPtr</code> to long.</td>
</tr>
<tr>
<td>unsigned long strtoul( const char *nPtr, char **endPtr, int base )</td>
<td>Converts the string <code>nPtr</code> 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><code>char *strcpy( char *s1, const char *s2 )</code></td>
<td>Copies string <code>s2</code> into array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncpy( char *s1, const char *s2, size_t n )</code></td>
<td>Copies at most <code>n</code> characters of string <code>s2</code> into array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strcat( char *s1, const char *s2 )</code></td>
<td>Appends string <code>s2</code> to array <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncat( char *s1, const char *s2, size_t n )</code></td>
<td>Appends at most <code>n</code> characters of string <code>s2</code> to array <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
</tbody>
</table>
String Manipulation Functions

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

The `strcmp` function compares two character strings `str1` and `str2`. The return value indicates:

- `<0` the first character that does not match has a lower value in `ptr1` than in `ptr2`
- `0` the contents of both strings are equal
- `>0` the first character that does not match has a greater value in `ptr1` than in `ptr2`
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).
  
  – &num[0] is the same as num
Pointers and Arrays in C

• Suppose we define the following array and pointer:

    ```c
    int a[100], *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 a 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)
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 *x;       /* pointer to int;           */  typedef int *T;
int x[10];    /* array of ints;            */  typedef int T[10];
int *x[10];   /* array of pointers to int; */  typedef int *T[10];
int (*x)[10]; /* pointer to array of ints; */  typedef int (*T)[10];
```

typedef defines a new type

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

- `int *x[10];`  
  - `x` is an array of pointers to `int`

- `int (*x)[10];`  
  - `x` is a pointer to an array of `int`

Arrays are the primary source of confusion. When in doubt, use extra parens to clarify the expression.
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;
}
```

**sizeof()** reports the size of a type in bytes

**calloc()** allocates memory for N elements of size k

Returns NULL if can’t alloc

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
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.

Macros and **static inline** functions must be included in any file that uses them, usually via a header file. Common uses for macros:

```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