Review

- Pointers and Memory Addresses
  - Physical and Virtual Memory
  - Addressing and Indirection
  - Functions with Multiple Outputs

- Arrays and Pointer Arithmetic

- Strings
  - String Utility Functions

- Searching and Sorting Algorithms
  - Linear Search
  - A Simple Sort
  - Faster Sorting
  - Binary Search
Review: Unconditional jumps

- **goto** keyword: jump somewhere else in the same function
- Position identified using labels
- Example (for loop) using **goto**:

```
{ 
    int i = 0, n = 20; /* initialization */
    goto loop_cond;

    loop_body:
    /* body of loop here */
    i++;

    loop_cond:
    if (i < n) /* loop condition */
        goto loop_body;
}
```

- Excessive use of **goto** results in “spaghetti” code
Review: I/O Functions

• I/O provided by `stdio.h`, not language itself
• Character I/O: `putchar()`, `getchar()`, `getc()`, `putc()`, etc.
• String I/O: `puts()`, `gets()`, `fgets()`, `fputs()`, etc.
• Formatted I/O: `fprintf()`, `fscanf()`, etc.
• Open and close files: `fopen()`, `fclose()`
• File read/write position: `feof()`, `fseek()`, `ftell()`, etc.
• ...

Review: `printf()` and `scanf()`

- Formatted output:
  ```c
  int printf (char format[], arg1, arg2, ...)
  ```
- Takes variable number of arguments
- Format specification:
  ```c
  %[flags][width][.precision][length]<type>
  ```
  - types: `d`, `i` (int), `u`, `o`, `x`, `X` (unsigned int), `e`, `E`, `f`, `F`, `g`, `G` (double), `c` (char), `s` (string)
  - flags, width, precision, length - modify meaning and number of characters printed
- Formatted input: `scanf()` - similar form, takes pointers to arguments (except strings), ignores whitespace in input
Strings represented in C as an array of characters (`char []`) 
String must be null-terminated (`'\0'` at end) 
Declaration: 
  ```c
  char str[] = "I am a string."; or 
  char str[20] = "I am a string.";
  ``` 
`strcpy()` - function for copying one string to another 
More about strings and string functions today...
Review

Pointers and Memory Addresses
- Physical and Virtual Memory
- Addressing and Indirection
- Functions with Multiple Outputs

Arrays and Pointer Arithmetic

Strings
- String Utility Functions

Searching and Sorting Algorithms
- Linear Search
- A Simple Sort
- Faster Sorting
- Binary Search
Pointers and addresses

- Pointer: memory address of a variable
- Address can be used to access/modify a variable from anywhere
- Extremely useful, especially for data structures
- Well known for obfuscating code
Physical and virtual memory

- Physical memory: physical resources where data can be stored and accessed by your computer
  - cache
  - RAM
  - hard disk
  - removable storage

- Virtual memory: abstraction by OS, addressable space accessible by your code
Physical memory considerations

- Different sizes and access speeds
- Memory management – major function of OS
- Optimization – to ensure your code makes the best use of physical memory available
- OS moves around data in physical memory during execution
- Embedded processors – may be very limited
Virtual memory

• How much physical memory do I have?
  Answer: 2 MB (cache) + 2 GB (RAM) + 100 GB (hard drive) + …

• How much virtual memory do I have?
  Answer: <4 GB (32-bit OS), typically 2 GB for Windows, 3-4 GB for linux

• Virtual memory maps to different parts of physical memory

• Usable parts of virtual memory: stack and heap
  • stack: where declared variables go
  • heap: where dynamic memory goes
Addressing variables

- Every variable residing in memory has an address!
- What doesn’t have an address?
  - register variables
  - constants/literals/preprocessor defines
  - expressions (unless result is a variable)
- How to find an address of a variable? The \& operator

```c
int n = 4;
double pi = 3.14159;
int *pn = &n; /* address of integer n */
double *ppi = &pi; /* address of double pi */
```

- Address of a variable of type \( t \) has type \( t \ast \)
Dereferencing pointers

- I have a pointer – now what?
- Accessing/modifying addressed variable: dereferencing/indirection operator *
  
  ```c
  /* prints "pi = 3.14159\n" */
  printf("pi = %g\n",*ppi);
  
  /* pi now equals 7.14159 */
  *ppi = *ppi + *pn;
  ```

- Dereferenced pointer like any other variable
- null pointer, *i.e.* 0 (**NULL**): pointer that does not reference anything
Casting pointers

• Can explicitly cast any pointer type to any other pointer type
  ppi = (\texttt{double \ast})pn; // pn originally of type \texttt{(int \ast) \ast}/

• Implicit cast to/from \texttt{void \ast} also possible (more next week...)

• Dereferenced pointer has new type, regardless of real type of data

• Possible to cause segmentation faults, other difficult-to-identify errors
  • What happens if we dereference \texttt{ppi} now?
Functions with multiple outputs

• Consider the Extended Euclidean algorithm
  \texttt{ext\_euclid}(a, b) function from Wednesday’s lecture

• Returns \( \gcd(a, b) \), \( x \) and \( y \) s.t. \( ax + by = \gcd(a, b) \)

• Used global variables for \( x \) and \( y \)

• Can use pointers to pass back multiple outputs:
  \begin{verbatim}
  int ext\_euclid(int a, int b, int *x, int *y);
  \end{verbatim}

• Calling \texttt{ext\_euclid()} , pass pointers to variables to receive \( x \) and \( y \):
  \begin{verbatim}
  int x, y, g;
  /* assume a, b declared previously */
  g = ext\_euclid(a, b, &x, &y);
  \end{verbatim}

• Warning about \( x \) and \( y \) being used before initialized
Accessing caller’s variables

- Want to write function to swap two integers
- Need to modify variables in caller to swap them
- Pointers to variables as arguments

```c
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

- Calling `swap()` function:

```c
int a = 5, b = 7;
swap(&a, &b);
/* now, a = 7, b = 5 */
```
• What is wrong with this code?

```c
#include <stdio.h>

char * get_message() {
    char msg[] = "Aren’t pointers fun?";
    return msg;
}

int main(void) {
    char * string = get_message();
    puts(string);
    return 0;
}
```
• What is wrong with this code?

```c
#include <stdio.h>

char * get_message() {
    char msg[] = "Aren’t pointers fun?";
    return msg;
}

int main(void) {
    char * string = get_message();
    puts(string);
    return 0;
}
```

• Pointer invalid after variable passes out of scope
Review

Pointers and Memory Addresses
- Physical and Virtual Memory
- Addressing and Indirection
- Functions with Multiple Outputs

Arrays and Pointer Arithmetic

Strings
- String Utility Functions

Searching and Sorting Algorithms
- Linear Search
- A Simple Sort
- Faster Sorting
- Binary Search
Arrays and pointers

• Primitive arrays implemented in C using pointer to block of contiguous memory

• Consider array of 8 ints:

```c
int arr[8];
```

• Accessing `arr` using array entry operator:

```c
int a = arr[0];
```

• `arr` is like a pointer to element 0 of the array:

```c
int *pa = arr; ⇔ int *pa = &arr[0];
```

• Not modifiable/reassignable like a pointer
The `sizeof()` operator

- For primitive types/variables, size of type in bytes:
  ```c
  int s = sizeof(char); /* == 1 */
  double f; /* sizeof(f) == 8 */ (64-bit OS)
  ```
- For primitive arrays, size of array in bytes:
  ```c
  int arr[8]; /* sizeof(arr) == 32 */ (64-bit OS)
  long arr[5]; /* sizeof(arr) == 40 */ (64-bit OS)
  ```
- Array length:
  ```c
  /* needs to be on one line when implemented */
  #define array_length(arr) (sizeof(arr) == 0 ? 0 : sizeof(arr)/sizeof((arr)[0]))
  ```
- More about `sizeof()` next week...
• Suppose int *pa = arr;

• Pointer not an int, but can add or subtract an int from a pointer:
  \( \text{pa} + i \) points to \( \text{arr}[i] \)

• Address value increments by \( i \) times size of data type
  Suppose \( \text{arr}[0] \) has address 100. Then \( \text{arr}[3] \) has address 112.

• Suppose char * pc = (char *)pa; What value of \( i \) satisfies
  \( (\text{int} \ *) (\text{pc} + i) == \text{pa} + 3? \)
Pointer arithmetic

• Suppose `int *pa = arr;`
• Pointer not an `int`, but can add or subtract an `int` from a pointer:
  `pa + i` points to `arr[i]`
• Address value increments by `i` times size of data type
  Suppose `arr[0]` has address 100. Then `arr[3]` has address 112.
• Suppose `char *pc = (char *)pa;` What value of `i` satisfies
  `(int *)(pc+i) == pa + 3`?
  • `i = 12`
Review

Pointers and Memory Addresses
  - Physical and Virtual Memory
  - Addressing and Indirection
  - Functions with Multiple Outputs

Arrays and Pointer Arithmetic

Strings
  - String Utility Functions

Searching and Sorting Algorithms
  - Linear Search
  - A Simple Sort
  - Faster Sorting
  - Binary Search
Strings as arrays

• Strings stored as null-terminated character arrays (last character == ' 0')
• Suppose `char str[] = "This is a string.";` and
  ```c
  char * pc = str;
  ```
• Manipulate string as you would an array
  ```c
  *(pc+10) = ' S';
  puts(str); /* prints "This is a String." */
  ```
String utility functions

- String functions in standard header `string.h`
- Copy functions: `strcpy()`, `strncpy()`
  ```
  char * strcpy( strto ,strfrom ); – copy `strfrom` to `strto`
  char * strncpy( strto ,strfrom,n); – copy `n` chars from `strfrom` to `strto`
  ```
- Comparison functions: `strcmp()`, `strncmp()`
  ```
  int  strcmp(str1,str2 ); – compare `str1`, `str2`; return 0 if equal, positive if `str1>`str2, negative if `str1<`str2
  int  strncmp(str1,str2 ,n); – compare first `n` chars of `str1` and `str2`
  ```
- String length: `strlen()`
  ```
  int  strlen( str ); – get length of `str`
  ```
More string utility functions

- Concatenation functions: \texttt{strcat()}, \texttt{strncat()}
  \begin{verbatim}
  char * strcat ( strto ,strfrom ); – add \texttt{strfrom} to end of \texttt{strto}
  char * strncat( strto ,strfrom,n); – add \texttt{n} chars from \texttt{strfrom} to end of \texttt{strto}
  \end{verbatim}

- Search functions: \texttt{strchr()}, \texttt{strrchr()}
  \begin{verbatim}
  char * strchr ( str ,c); – find char \texttt{c} in \texttt{str}, return pointer to first occurrence, or NULL if not found
  char * strrchr ( str ,c); – find char \texttt{c} in \texttt{str}, return pointer to last occurrence, or NULL if not found
  \end{verbatim}

- Many other utility functions exist...
Review

Pointers and Memory Addresses
  • Physical and Virtual Memory
  • Addressing and Indirection
  • Functions with Multiple Outputs

Arrays and Pointer Arithmetic

Strings
  • String Utility Functions

Searching and Sorting Algorithms
  • Linear Search
  • A Simple Sort
  • Faster Sorting
  • Binary Search
Searching and sorting

- Basic algorithms
- Can make good use of pointers
- Just a few examples; not a course in algorithms
- Big-O notation
Searching an array

• Suppose we have an array of int’s
  int arr[100]; /* array to search */
• Let’s write a simple search function:

```c
int * linear_search(int val) {
    int * parr, * parrend = arr + array_length(arr);
    for (parr = arr; parr < parrend; parr++) {
        if (*parr == val)
            return parr;
    }
    return NULL;
}
```
A simple sort

• A simple insertion sort: $O(n^2)$
  • iterate through array until an out-of-order element found
  • insert out-of-order element into correct location
  • repeat until end of array reached

• Split into two functions for ease-of-use

```c
int arr[100]; /* array to sort */

void shift_element(unsigned int i) {
   /* do insertion of out-of-order element */
}

void insertion_sort() {
   /* main insertion sort loop */
   /* call shift_element() for each out-of-order element */
}
```
Shifting out-of-order elements

• Code for shifting the element

```c
/* move previous elements down until insertion point reached */
void shift_element(unsigned int i) {
    int ivalue;
    /* guard against going outside array */
    for (ivalue = arr[i]; i && arr[i-1] > ivalue; i--)
        arr[i] = arr[i-1]; /* move element down */
    arr[i] = ivalue; /* insert element */
}
```
Insertion sort

- Main insertion sort loop

```c
void insertion_sort(void) {
    unsigned int i, len = array_length(arr);
    for (i = 1; i < len; i++)
        if (arr[i] < arr[i - 1])
            shift_element(i);
}
```

- Can you rewrite using pointer arithmetic instead of indexing?
Many faster sorts available (shellsort, mergesort, quicksort, ...)

Quicksort: $O(n \log n)$ average; $O(n^2)$ worst case
- choose a pivot element
- move all elements less than pivot to one side, all elements greater than pivot to other
- sort sides individually (recursive algorithm)

Implemented in C standard library as `qsort()` in `stdlib.h`
Quicksort implementation

- Select the pivot; separate the sides:

```c
void quick_sort(unsigned int left,
                unsigned int right) {
    unsigned int i, mid;
    int pivot;
    if (left >= right)
        return; /* nothing to sort */
    /* pivot is midpoint; move to left side */
    swap(arr+left, arr + (left+right)/2);
    pivot = arr[mid = left];
    /* separate into side < pivot (left+1 to mid)
    and side >= pivot (mid+1 to right) */
    for (i = left+1; i <= right; i++)
        if (arr[i] < pivot)
            swap(arr + ++mid, arr + i);
```

[Kernighan and Ritchie. The C Programming Language. 2nd ed. Prentice Hall, 1988.]

© Prentice Hall. All rights reserved. This content is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/fairuse.
Quicksort implementation

• Restore the pivot; sort the sides separately:

```c
/* restore pivot position */
swap(arr+left, arr+mid);
/* sort two sides */
if (mid > left)
    quick_sort(left, mid-1);
if (mid < right)
    quick_sort(mid+1, right);
```

• Starting the recursion:

```c```
quick_sort(0, array_length(arr) - 1);
```

[Kernighan and Ritchie. The C Programming Language. 2nd ed. Prentice Hall, 1988.]
Discussion of quicksort

• Not stable (equal-valued elements can get switched) in present form
• Can sort *in-place* – especially desirable for low-memory environments
• Choice of pivot influences performance; can use random pivot
• Divide and conquer algorithm; easily parallelizeable
• Recursive; in worst case, can cause stack overflow on large array
Searching a sorted array

• Searching an arbitrary list requires visiting half the elements on average

• Suppose list is sorted; can make use of sorting information:
  • if desired value greater than value and current index, only need to search after index
  • each comparison can split list into two pieces
  • solution: compare against middle of current piece; then new piece guaranteed to be half the size
  • divide and conquer!

• More searching next week...
Binary search

- Binary search: $O(\log n)$ average, worst case:

```c
int * binary_search(int val) {
    unsigned int L = 0, R = array_length(arr), M;
    while (L < R) {
        M = (L+R-1)/2;
        if (val == arr[M])
            return arr+M; /* found */
        else if (val < arr[M])
            R = M; /* in first half */
        else
            L = M+1; /* in second half */
    }
    return NULL; /* not found */
}
```
Binary search

- Worst case: logarithmic time
- Requires random access to array memory
  - on sequential data, like hard drive, can be slow
  - seeking back and forth in sequential memory is wasteful
  - better off doing linear search in some cases
- Implemented in C standard library as `bsearch()` in `stdlib.h`
Topics covered:

- Pointers: addresses to memory
  - physical and virtual memory
  - arrays and strings
  - pointer arithmetic

- Algorithms
  - searching: linear, binary
  - sorting: insertion, quick