Pointers and Arrays

- A **pointer** is a variable that stores the address of another variable.
- Pointers are similar to **reference variables** in Java.
- May be used to produce more compact and efficient code (but can be tricky to understand and debug if not used carefully!)
- Pointers allow for complex “linked” data structures (e.g. linked lists, binary trees)
- Pointers allow for passing function parameters by reference instead of by value.
- Pointers and arrays are closely related.
Memory Organization

- Memory is an array of consecutively addressed memory cells.
- Typical size of each cell is 1 byte.
- A `char` takes one byte whereas other types use multiple cells depending on their size.
Memory Organization

Example:

```java
int a = 7;
char b = 'b';
```
**Pointer Syntax**

- **Address operator (&):** gives the address in memory of an object.
  
  ```
  p = &c; // p points to c
  // (address of c is assigned to p)
  ```

- **Indirection or dereferencing operator (*):** Gives access to the object a pointer points to.

- **How to declare a pointer variable?** Declare using the type of object the pointer will point to and the * operator.
  
  ```
  int *pa; // a pointer that points to an int object
debug double *pb; // a pointer that points to a double object
  ```
int a = 7;
char b = 'b';

int *pa = &a;  // pa points to a
Pointers Illustrated (2)

```c
int a = 7;
char b = 'b';

int *pa = &a; // pa points to a
int c = *pa;   // c = 7
```
Pointers Example 1

- Consider the following declaration
  
  ```c
  int x = 1, y = 2;
  int *ip;  // ip is a pointer to an int
  ```

- Now we use the address `&` and dereferencing `*` operators:
  
  ```c
  ip = &x;  /* ip now points to x */
y = *ip;  /* y is now 1 */
*ip = 10;  /* x is now 10 */
  ```

- Note that *ip can be used any place x can be used. Continuing the example:
  
  ```c
  *ip = *ip + 1;  /* x is now 11 */
*ip += 1;  /* x is now 12 */
++*ip;  /* x is now 13 */
(*ip)++;  /* x is now 14, parentheses required */
  ```

- See full example at C-examples/pointers-and-arrays/pointers1.c.
Pointers as Function Arguments

/* WRONG! */
void swap(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}

/* swap *px and *py */
void swap(int *px, int *py) {
    int tmp;
    tmp = *px;
    *px = *py;
    *py = tmp;
}

- The function on the left is called as `swap(a, b)` but doesn’t work since C passes arguments to functions by value (same as Java).
- The one on the right will be called as `swap(&a, &b)` and works.
- See full example at C-examples/pointers-and-arrays/swap.c.
- Demo using the GDB
Pointers and Arrays

- The name of an array is a pointer to the element zero of the array. So we can write
  
  ```
  int a[10];
  int *pa = &a[0]; // or int *pa = a;
  ```

- Accessing the $i$th element of an array, $a[i]$, can be written as $*(a + i)$. Similarly $*(pa + i)$ can be written as $pa[i]$.

- Note that $pa + i$ points to the $i$th object beyond $pa$ (advancing by appropriate number of bytes for the underlying type).

- Note that the name of an array isn’t a variable, so we can’t change it’s value. So $a = pa$ or $a++$ aren’t legal.
In Class Exercise

```c
int a[10];
int *pa = &a[0]; // or int *pa = a;

What happens when we run the following code?
pa += 1;
for (i = 0; i < 10; i++)
    printf("%d ", pa[i]);

What happens when we run the following code (after the above statements)?
pa = a;
pa--;
printf("%d ", *pa);
```
1-dimensional Arrays

Arrays can be statically declared in C, such as:

```c
int A[100];
```

The space for this array is declared on the stack segment of the memory for the program (if `A` is a local variable) or in the data segment if `A` is a global variable.
However, dynamically declared arrays are more flexible. These are similar to arrays declared using the `new` operator in Java.

```java
/* Java array declaration */
int n = 100;
int[] A = new int[n];
for (i = 0; i < n; i++)
    A[i] = i; //example initialization
```

The equivalent in C is the standard library call `malloc` and its variants.

```c
int n = 100;
int * A = (int *) malloc (sizeof(int) * n);
for (i = 0; i < n; i++)
    A[i] = i; //example initialization
```

The space allocated by malloc is in the heap segment of the memory for the program. To free the space, use the `free()` library call.

```c
free(A);
```
Dynamically Allocating Memory (2)

- Use `malloc()` and `free()` to allocate and free memory dynamically.

```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nmemb, size_t size);
```

- `malloc()` takes as an argument the number of bytes of memory to allocate. `Malloc` does not initialize the allocated memory (why?)
- `calloc()` allocates and initializes the memory to zero. See man page for `realloc()`.
- These functions return a pointer to `void`. What does that mean?
The void * pointer (1)

- A void * pointer is a pointer that can contain the address of data without knowing its type. This allows pointers of any type to be assigned to it. This supports generic programming in C (similar to Object class in Java).
- Normally, we cannot assign a pointer of one type to a pointer to another type.
- A void * pointer cannot be dereferenced. It must be cast to the proper type of the data it points to before it can be used.
- Note that malloc() returns the void * pointer as it doesn’t know what type of data we intend to store in the allocated memory. Hence, we need to cast the return value from malloc() to the appropriate type.
Swapping arrays using pointers

How to swap two arrays? Here is the naive way.

```c
int *A = (int *) malloc(sizeof(int) * n);
int *B = (int *) malloc(sizeof(int) * n);
// initialize A and B (code not shown here)
for (i = 0; i < n; i++) {
}
```

Using pointers we can merely swap the values of pointers.

```c
int *A = (int *) malloc(sizeof(int) * n);
int *B = (int *) malloc(sizeof(int) * n);
// initialize A and B (code not shown here)
int *tmp = A; A = B; B = tmp;
```

Simpler and far more efficient!

Example C-examples/pointers-and-arrays/1d-arrays.c
Pointers of the same type can be assigned to each other. Pointers of any type can be assigned to a pointer of type `void *` as an exception.

```c
int *pa = &x;
int *pi = pa;
double *pd = pa; // ILLEGAL
void *pv = pa;
```

The constant zero represents the null pointer, written as the constant `NULL` (defined in `<stdio.h>`). Assigning or comparing a pointer to zero is valid.

```c
int *pa = NULL;
if(pa == NULL) {
    // don't try to dereference pa.
}
```
Pointers can be compared or subtracted if they point to the members of the same array.

Adding/subtracting a pointer and an integer is valid

All other pointer arithmetic is illegal.

The C standard implies that we cannot do arithmetic with a `void *` pointer but most compilers implement it. For the purposes of arithmetic, they treat `void *` as a `char *`. For example, incrementing a `void *` pointer will increment it by one byte.

Pointer and address arithmetic is one of the strengths of C.

Example: C-examples/pointers-and-arrays/pointer-types.c
Consider the following code:

```c
int * A;
void *ptr;

A = (int *) malloc(sizeof(int) * n);
ptr = A;
```

Based on the above code, mark all of the following expressions that correctly access the value stored in `A[i].`

1. *(A + i)
2. *(ptr + i)
3. *(int *)(ptr + i)
4. *((int *)ptr + i)
5. *(ptr + sizeof(int)*i)
Two-dimensional Arrays

- A 2-dimensional array can be statically allocated in C as shown in the following example:

  ```c
  int Z[4][10];
  ```

- This array is laid out in memory in row major order, that is, it is laid out as a 1d array with row 0 first followed by row 1, row 2 and then row 3.

- Some languages use a column major layout for 2-d arrays, which is column 0, then column 1, ..., and finally column 9 in the above example.
Two-dimensional Arrays

- A 2-dimensional array can be dynamically allocated in C as shown in the following example:

  ```c
  int **X;
  X = (int **) malloc(n * sizeof(int *));
  for (i=0; i<n; i++)
      X[i] = (int *) malloc(n * sizeof(int));
  ```

- Now `X` can be used with the normal array syntax, as below

  ```c
  for (i=0; i<n; i++)
      for (j=0; j<n; j++)
          X[i][j] = i;
  ```

- To free the 2-dimensional array, we need to reverse all the calls to `malloc()`, as shown below:

  ```c
  for (i=0; i<n; i++)
      free(X[i]);
  free(X);
  ```

- See full example at

  C-examples/pointers-and-arrays/2d-arrays.c
Layout of a Dynamically Allocated 2d Array

```c
int **X
int *X
int 0 1 9
```
3-dimensional Arrays

- **In-class team exercise.** Write code to dynamically allocate a 3-dimensional array of characters named X with size $p \times q \times r$ (*Hint: Mind your p’s and q’s :-)*

- **In-class team exercise.** Come up with some application examples that could benefit from a 3-dimensional array

- Draw a picture of the memory layout for the 3-dimensional array allocated above.

- **Takeaway!** Write code to properly free the 3-dimensional array allocated above
A **C-style string** is an array of characters terminated by the null character (ASCII code = 0).

- The header file is `string.h`
- See man page for `string` for a list of all C string functions.
- Four ways of declaring a string.

```c
char s0[] = "magma"; /* declares a fixed size array */
char *s1 = "volcano";
char s2[MAXLEN];
char *s3; /* uninitialized */
```

```c
strcpy(s2, s1);
/* strncpy(s2, s1, MAXLEN); */ /* safer */
s3 = (char *) malloc(sizeof(char)*(strlen(s1)+1)); /* even safer */
strcpy(s3, s1);
```

- C doesn’t provide strings as a basic type so we use functions to do operations with strings...
- Common string manipulation functions: `strlen()`, `strcat()`, `strncat()`, `strcpy()`, `strncpy()`, `strcmp()`, `strncmp()`, `strtok()`, `strsep()`, `strfry()` etc.
/* strcpy; copy t to s, array version */
void strcpy(char *s, char *t) {
    int i=0;
    while ((s[i] = t[i]) != '\0')
        i++;
}

/* strcpy; copy t to s, pointer version 1 */
void strcpy(char *s, char *t) {
    int i=0;
    while ((*s = *t) != '\0'){
        s++;
        t++;
    }
}
/* strcpy; copy t to s, pointer version 2 */
void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0') ;
}

/* strcpy; copy t to s, pointer version 3 */
void strcpy(char *s, char *t) {
    while ((*s++ = *t++)) ;
}

See page 51 of K&R for why you can use an assignment expression as a boolean.
//recommend the use of parentheses around assignment used as a boolean
String Comparison

- `strcmp(s, t)` returns negative, zero or positive if `s` is lexicographically less, equal or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` differ.

```c
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```
The pointer version of `strcmp`:

```c
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```
String Examples

- A String Example:  C-examples/strings/strings-ex1.c
- A String Tokenizing Example:  
  C-examples/strings/strings-ex2.c
- A Better String Tokenizing Example:  
  C-examples/strings/strings-ex3.c
Other String Tokenizing Functions

- Use `strsep()` in cases where there are empty fields between delimiters that `strtok()` cannot handle.
Command line arguments

- Recommended prototype for the main function:
  ```c
  int main(int argc, char *argv[])
  ```
  or
  ```c
  int main(int argc, char **argv)
  ```

- **argc** C-style strings are being passed to the main function from `argv[0]` through `argv[argc-1]`. The name of the executable is `argv[0]`, the first command line argument is `argv[1]` and so on. Thus `argv` is an array of pointers to `char`.

- Draw the memory layout of the `argv` array.
C allows a function call to have a variable number of arguments with the variable argument list mechanism.

Use *ellipsis* ... to denote a variable number of arguments to the compiler. The ellipsis can only occur at the end of an argument list.

Here are some standard function calls that use variable argument lists.

```c
int printf(const char *format, ...);
int scanf(const char *format, ...);
int execlp(const char *file, const char *arg, ...);
```

See `man stdarg` for documentation on using variable argument lists. In particular, the header file contains a set of macros that define how to step through the argument list.

See Section 7.3 in the K&R C book.
Useful macros from `stdarg` header file.

- `va_list argptr;` is used to declare a variable that will refer to each argument in turn.
- `void va_start(va_list argptr, last);` must be called once before `argptr` can be used. `last` is the name of the last variable before the variable argument list.
- `type va_arg(va_list ap, type);` Each call of `va_arg` returns one argument and steps `ap` to the next; `va_arg` uses a type name to determine what type to return and how big a step to take.
- `void va_end(va_list ap);` Must be called before program returns. Does whatever cleanup is necessary.
- It is possible to walk through the variable arguments more than once by calling `va_start` after `va_end`. 
/* C-examples/varargs/test-varargs.c */
#include <stdio.h>
#include <stdarg.h>

void strlist(int n, ...)
{
    va_list ap;
    char *s;

    va_start(ap, n);
    while (1) {
        s = va_arg(ap, char *);
        printf("%s\n", s);
        n--;
        if (n==0) break;
    }
    va_end(ap);
}

int main()
{
    strlist(3, "string1", "string2", "string3");
    strlist(2, "string1", "string3");
}
Exercises

- Reading Assignment: Section 5.1, 5.2, 5.3, 5.5, 5.6, 5.7, 5.8, 5.9, and 5.10. Skip or skim Section 5.4. Skip Sections 5.11 and 5.12 for now.