cs3157: and another C lecture (mon-28-feb-2005)

- today:
  - arrays
  - pointers
  - dynamic memory allocation
  - functions
  - function arguments
  - arrays and pointers as function arguments

arrays overview.

- arrays and pointers are strongly related in C

```
int a[10];   // declare an array of size 10 ints (consecutive in memory)
int *pa;     // declare a pointer to an int
pa = &a[0];  // pa points to the 0th element of a i.e., the address of a[0]
pa = a;      // this has the same effect
```

- pointer arithmetic is meaningful with arrays:
  if we do $pa = \&a[0]$; then $*(pa+1)$ points to $a[1]$
- remember difference between $(*pa)+1$ and $*(pa+1)$ (which == $*pa+1$)
- note that an array name is a pointer, so we can also do $*(a+1)$ and in general:
  $*(a+i) == a[i]$ and so are $a+i == \&a[i]$
- the difference:
  an array name is a constant, and a pointer is not
  so we can do: $pa = a$ and $pa++$
  but we can NOT do: $a = pa$ or $a++$ or $p = \&a$
- when an array name is passed to a function, what is passed is the beginning of the array

arrays (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;`

arrays (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]`

array (3).

character array example

```
#include <stdio.h>
#define MAX 6
int main( void ) {
  char str[MAX] = "ABCDE";
  int i;
  for ( i=0; i<MAX-1; i++ ) {
    printf( "%c", str[i] );
  }
  printf( "\n" );
} /* end of main() */
```


arrays (4).

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() */
```


pointers overview.

- a pointer contains the address of an object (but not in the OOP sense)
- allows one to access object "indirectly"
- & = unary operator that gives address of its argument
- * = unary operator that fetches contents of its argument (i.e., its argument is an address)
- note that & and * bind more tightly than arithmetic operators
- you can print the value of a pointer with the formatting character `%p`
- example: `pointers.c`

pointers (1).

- 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:

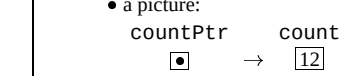
```
int *count;
float *avg;
```

pointers (2).

- 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:



pointers (3).

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
...		

pointers (4).

- 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

pointers (5).

```
#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 );
}
```

pointers (6).

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
```


dynamic memory allocation overview.

- used when you don't know at compile-time how much memory to allocate
- pre-allocated memory comes from the "stack"
- dynamically allocated memory comes from the "heap"
- family of functions in `stdlib`, including:

```
void *malloc( size_t size );
void *realloc( void *ptr, size_t size );
void free( void * );
```

- `malloc` and `realloc` return a generic pointer `(void *)` and you have to "cast" the return to the type of pointer you want
- example: `malloc.c`

dynamic memory allocation (1).

- *malloc()* allocates a block of memory:

```
void *malloc( size_t size );
```

- lifetime of the block is until memory is freed, with *free()*:

```
void free( void *ptr );
```

- example:

```
int *dynvec, num_elements;
printf( "how many elements do you want to enter? " );
scanf( "%d", &num_elements );
dynvec = (int *)malloc( sizeof(int) * num_elements );
```


dynamic memory allocation (2).

- memory leaks — memory allocated that is never freed:

```
char *combine( char *s, char *t ) {
  u = (char *)malloc( strlen(s) + strlen(t) + 1 );
  if ( s != t ) {
    strcpy( u, s );
    strcat( u, t );
    return u;
  }
  else {
    return 0;
  }
} /* end of combine() */
```

- `u` should be freed if `return 0;` is executed
- but you don't need to free it if you are still using it!

dynamic memory allocation (3).

- note: `malloc()` does not initialize data
- you can allocate and initialize with "calloc":

```
void *calloc( size_t nmemb, size_t size );
```

  - calloc allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero.

- you can also change size of allocated memory blocks with "realloc":

```
void *realloc( void *ptr, size_t size );
```

  - realloc changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized.

- these are all functions in `stdlib.h`
- for more information: `unix$ man malloc`

dynamically allocated arrays (1).

- "arrays" are defined by specifying an element type and number of elements
  - statically:

```
int vec[100];
char str[30];
float m[10][10];
```

  - dynamically:

```
int *dynvec, num_elements;
printf( "how many elements do you want to enter? " );
scanf( "%d", &num_elements );
dynvec = (int *)malloc( sizeof(int) * num_elements );
```

- for an array containing N elements, indeces are 0..N-1
- stored as a linear arrangement of elements
- often similar to pointers

dynamically allocated arrays (2).

- C does not remember how large arrays are (i.e., no length attribute, unlike Java)
- given:

```
int x[10];
x[10] = 5;  /* error! */
```

- ERROR! because you have only defined x[0]..x[9] and the memory location where x[10] is can become something else...
- `sizeof x` gives the number of bytes in the array
- `sizeof x[0]` gives the number of bytes in one array element
- thus you can compute the length of `x` via:

```
int length_x = sizeof x / sizeof x[0];
```


dynamically allocated arrays (3).

- when an array is passed as a parameter to a function:
  - the size information is not available inside the function
  - array size is typically passed as an additional parameter

```
printArray( x, length_x );
```

  - or globally

```
#define VECSIZE 10
int x[VECSIZE];
```

dynamically allocated arrays (4).

- array elements are accessed using the same syntax as in Java: `array[index]`
- C does not check whether array index values are sensible (i.e., no bounds checking)
- e.g., `x[-1]` or `vec[10000]` will not generate a compiler warning!
- if you're lucky, the program crashes with

```
Segmentation fault (core dumped)
```


dynamically allocated arrays (5).

- C references arrays by the address of their first element
- array is equivalent to `&array[0]`
- you can iterate through arrays using pointers as well as indexes:

```
int *v, *last;
int sum = 0;
last = &x[length_x-1];
for ( v = x; v <= last; v++ )
  sum += *v;
```


dynamically allocated arrays (6).

- example:

```
#include <stdio.h>
#define MAX 12
int main( void ) {
  int x[MAX]; /* declare 12-element array */
  int i, sum;
  for ( i=0; i<MAX; i++ ) { x[i] = i; }
  /* here, what is value of i? of x[i]? */
  sum = 0;
  for ( i=0; i<MAX; i++ ) { sum += x[i]; }
  printf( "sum = %d\n",sum );
} /* end of main() */
```


dynamically allocated arrays (7).

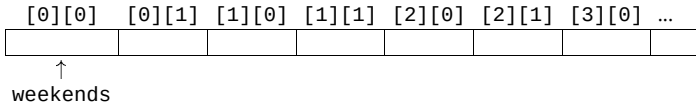
- another example:

```
#include <stdio.h>
#define MAX 10
int main( void ) {
  int x[MAX]; /* declare 10-element array */
  int i, sum, *p;
  p = &x[0];
  for ( i=0; i<MAX; i++ ) { *p = i + 1;  p++; }
  p = &x[0];
  sum = 0;
  for ( i=0; i<MAX; i++ ) {  sum += *p;  p++; }
  printf( "sum = %d\n",sum );
} /* end of main() */
```

2-dimensional arrays.

- 2-dimensional arrays

```
int weekends[52][2];
```



- you can use indices or pointer math to locate elements in the array
  - `weekends[0][1]`
  - `weekends+1`
- `weekends[2][1]` is same as `*(weekends+2*2+1)`, but NOT the same as `*weekends+2*2+1` (which is an integer)!

functions (1).

- similar to methods in java
- but there aren't classes in C
- and functions can't be overloaded
- syntax:

```
<type>  name( argument-list-if-any )
argument-declarations-if-any;
{
  function-body;
  return [<expression>];
}
```

or

```
<type>  name( argument-list-if-any-including-declarations  ) {
  function-body;
  return [<expression>];
}
```


functions (2).

- a program is just a set of individual function definitions
- `char` promotes to `int` in any expression, so k&r says you don't need to define functions that return `char` (only `int`)
- `int` is the default return type
- function arguments are "passed by value"
- the function receives a temporary copy of the value of the argument (not the argument's address)
- functions with a variable number of arguments use the first argument to tell it how many arguments will follow (e.g., `printf`)
- function arguments
  - since function arguments are "passed by value", you can use pointers to have a function change the value of a variable
- example: `swap.c`