CS3157: Advanced Programming

Lecture #9

Mar 6

Shlomo Hershkop

shlomo@cs.columbia.edu

Outline

- Arrays
- Pointers
- Memory allocation
- functions
- function arguments
- arrays and pointers as function arguments

- Reading
  - Chapter 5,6-6.3

Arrays again

- Arrays and pointers are strongly related in C

```
int a[10];
int *pa;
```

- (remember that &a[0] is the address of the first element in a, that is the beginning of the array

```
pa = &a[0];
pa = a;
```

- pointer arithmetic is meaningful with arrays:
- if we do

```
Pntr = &a[0]
```

- then

```
*(Pntr +1) =
```

- Is whatever is at a[1]

- There is a difference between
  - *(Pntr) + 1
  - and (*Pntr +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: Pntr = a and Pntr ++
- But we can NOT do: a = Pntr or a++ pr or Pntr = &a
- That is you can not reassign it as a pointer

Note

- When an array name is passed to a function, what is passed is the beginning of the array, that is passed by reference
- It is important, since this is an address, any changes to that memory location will stick when you come back from the function

From last time

- 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

code

```
#include <stdio.h>
main() {
  int x, y;      // declare two ints
  int *px;       // declare a pointer to an int
  x = 3;         // initialize x
  px = &x;
  y = *px;
 printf( "x=%d px=%p y=%d\n",x,px,y );
}
```

Memory allocation

- One of the main advantage to c/cpp is that you can manipulate memory yourself (and are responsible to clean up after yourself.
- When you don’t it is called memory leaking…more on this later

Array vs memory allocation

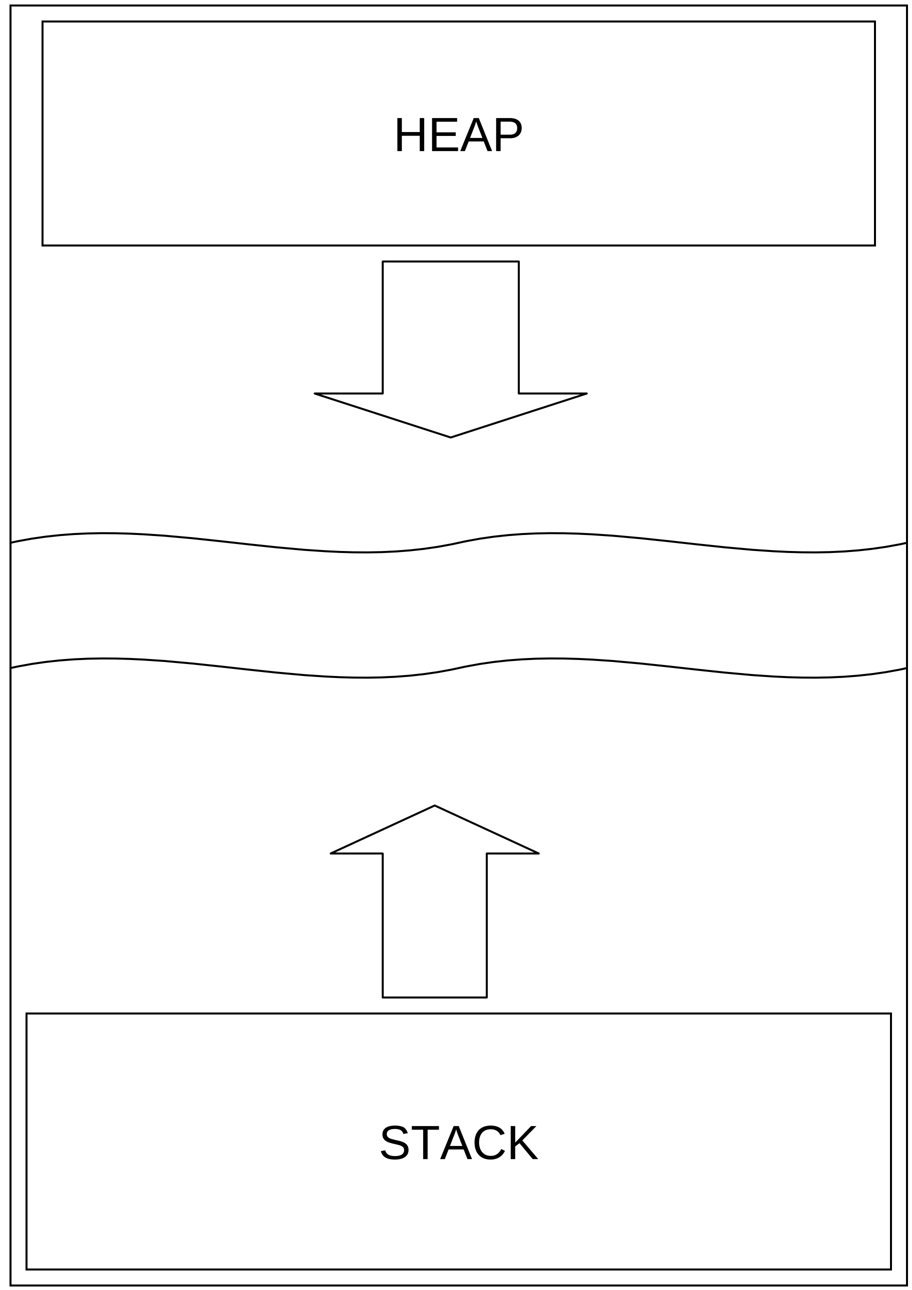
- Arrays are great when you have a rough idea of how many items you will be dealing with
  - 10 numbers
  - 30 students
  - Less than 256 characters of input

Map of memory

- Think of memory as a box
- Main is placed on the bottom and any variable on top of that
- Any function call gets placed on top of that
- This part of memory grows upward
- It is called the stack
- Your program is over when the stack is empty

heap

- The heap is the other side of memory
- Global variables, and allocated memory is created on the heap
- It grows downwards


HEAP
STACK

Dynamic Memory Allocation

- pre-allocated memory comes from the “stack”
- dynamically allocated memory comes from the “heap”
- To get memory you allocated (malloc) memory, and to let it go, you free it (free)
- 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
- That is if you are allocation a bunch of characters, you say
- Ptr = (char*) malloc….

Malloc.c

```
#include <stdio.h>
#include <stdlib.h>
#define BLKSIZ 10
main() {
  FILE *fp;
  char *buf, k;
  int  bufsiz, i;
  // open file for reading
  if (( fp = fopen( "myfile.dat","r" )) == NULL ) {
    perror( "error opening myfile.dat" );
    exit( 1 );
  }
  // allocate memory for input buffer
  bufsiz = BLKSIZ;
  buf = (char *)malloc( sizeof(char)*bufsiz );
```

II

```
// read contents of file
  i = 0;
  while (( k = fgetc( fp )) != EOF ) {
    buf[i++] = k;
    if ( i == bufsiz ) {
      bufsiz += BLKSIZ;
      buf = (char *)realloc( buf,sizeof(char)*bufsiz );
    }
  }
  if ( i >= bufsiz-1 ) {
    bufsiz += BLKSIZ;
    buf = (char *)realloc( buf,sizeof(char)*bufsiz );
  }
  buf[i] = '\0';
  // output file contents to the screen
  printf( "buf=[%s]\n",buf );
  // close file
  fclose( fp );
} // end main()
```

Dynamic memory

- 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 );
```

Memory leaking

- 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!

Example 2

```
int main(void) {

  char *string1 = (char*)malloc(sizeof(char)*50);
  char *string2 = (char*)malloc(sizeof(char)*50);
  scanf("%s",string2);
  string1 = strong2;  //MISTAKE THIS IS NOT A COPY

  ...
  free(string2);
  free(string1); ///????

  return 0
  }
```

Memory leak tools

- Purify
- Valgrind
- Insure++
- Memwatch (will use it in lab)
- Memtrace
- Dmalloc

Dynamic memory

- note: malloc() does not initialize data, that is you have garbage there with whatever was there in memory
- 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: `man malloc`

Dynamic arrays

- “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

Dynamic arrays II

- 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
- You can compute the length of x via:

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

Arrays cont.

- when an array is passed as a parameter to a function:
  - The size information is not available inside the function, since you are only passing in a start memory location
  - array size is typically passed as an additional parameter

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

  - or globally

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

arrays

- 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

- 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;
```

Code

```
#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() */
```

Code 2

```
#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)!

swap

```
void swapNot( int a,int b ) {
  int tmp = a;
  a = b;
  b = tmp;
} // end swapNot()


void swap( int *a,int *b ) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
} // end swap()
```

swap

```
int x, y;          // declare two ints
int *px, *py;      // declare two pointers to ints
x = 3;   // initialize x
y = 5;   // initialize y

printf( "before: x=%d y=%d\n",x,y );

swapNot( x,y );
printf( "after swapNot: x=%d y=%d\n",x,y );

px = &x; // set px to point to x (i.e., x's address)
py = &y; // set py to point to y (i.e., y's address)

printf( "the pointers: px=%p py=%p\n",px,py );

swap( px,py );
printf( "after swap with pointers: x=%d y=%d px=%p py=%p\n",x,y,px,py );

// you can also do this directly, without px and py:
swap( &x,&y );
printf( "after swap without pointers: x=%d y=%d\n",x,y );
```

Next time

- Do reading on memory allocation and structs
- See you in lab Wednesday.