Pointers

Pointers, Arrays, and Strings
Review

• Introduction to C
  ▫ C History
  ▫ Compiling C
  ▫ Variables
  ▫ Logical Operators
  ▫ Control Structures(loops, if/else)
• Functions and Macros
• Separate Compilation
• Arrays
• Strings
Pointers

• Variable which points to a memory location
• Components
  ▫ Name – name of the pointer variable
  ▫ Type – Objective type that the pointer is addressing

• E.g. int * myPointer;
  ▫ Variable size in memory is not based on objective type, but based on potential memory size
  ▫ Why?
Dereferencing Pointers

- Because a pointer stores a memory location, not a value, you will need to get access to the value at that memory location
  - Called Dereferencing
- Multiple ways to dereference a pointer
  - **Unary operator** `*`
    - `*myPtr = 5;  //dereference to value at myPtr`
  - **Bracket offset**
    - `myPtr[0] = 5;  //dereference to value at myPtr + offset * sizeof(objective_type)`
Address of

• Pointers are used to tell the memory location of a value, however you need to be able to access the memory location of that value
• & operator
  ▫ Literally the “address of” the variable
• E.g.
  ▫ int val = 5;
  ▫ int* myPtr = &val; //declare a pointer to an integer and set it equal to the memory location of val
Pointer Example

- Example Code
  ```c
  int val = 10;
  int* myPtr = &val; //declare pointer to val
  *myPtr = 5;        //dereference pointer and set
                     //that location = 5
  printf("val =\%d",val); //print val=(value of val)
  ```

- Output
  ```
  val=5
  ```
Objective Type

• The type of variable which is being pointed to or type of array
  ▫ int* p; //objective type = int
  ▫ int a[10]; //objective type of a is int
  ▫ int ** p; // root objective type is int but objective
    //type of p is int *

• Adding 1 to a pointer variable actually increments by the size of the objective type
  ▫ Incrementing an int* on GL increments the value by 4 (size of int)
  ▫ Incrementing an int** on GL increments the value by 8 (size of int*)
Pointers and Arrays

• Strong relationship between pointers and arrays
• E.g
  ▫ `int a[10];` //creates array of 10 integers
  ▫ `int* p;` //creates a pointer to an int
  ▫ `p=a;` // assigns the memory location of the first
    //element of the array to p, therefore making
    //p an alias for a, reference array using p or a
• `p=a` can also be written `p=&(a[0])` or `p=&a[0]`
Pointers and Arrays

• Name of array is equivalent to pointer to first element of array and vice-versa
• Therefore if \( a \) is the name of an array, \( a[i] \) is equivalent to \( *(a+i) \)
• It follows then that \&a[i] and \((a+i)\) are also equivalent
  ▪ Both represent address of i-th element beyond a
• Additionally, if \( p \) is a pointer, then it may be used with a subscript as if it were the name of an array
  ▪ \( p[i] \) is identical to \( *(p+i) \)
• In short, an array-and-index expression is equivalent to a pointer-and-offset expression
What is the difference?

- If name of array is synonymous with a pointer to the first element of the array, and function parameters defined as arrays are “almost” like pointers, what is the difference between array name and a pointer?
  - Array name can only “point” to the first element of its array, a pointer may be changed to point to any variable or array of the appropriate type
  - E.g.
    - `int vec[3] = {1,2,3};`
    - `Vec = &value; //can’t do this`
Example

```c
int g, grades[ ] = {10, 20, 30, 40}, myGrade = 100, yourGrade = 85, *pGrades;
/* grades can be (and usually is) used as array name */
for (g = 0; g < 4; g++)
    printf("%d\n",grades[g]);
/* grades can be used as a pointer to its array if it doesn’t change*/
for (g = 0; g < 4; g++)
    printf("%d\n", *(grades + g));
/* but grades can’t point anywhere else */
g= &myGrade; /* compiler error */
/* pGrades can be an alias for grades and used like an array name */
pGrades = grades; /* or pGrades = &(grades[0]); */
for( g = 0; g < 4; g++)
    printf("%d\n", pGrades[g]);
/* pGrades can be an alias for grades and be used like a pointer that changes */
for (g = 0; g < 4; g++)
    printf("%d\n", *(pGrades++));
/* BUT, pGrades can point to something else other than the grades array */
pGrades = &myGrade;
printf("%d\n", &pGrades);
pGrades = &yourGrade;
printf("%d\n", &pGrades);
```
Pointer Arithmetic

• Remember, incrementing a pointer by i actually increments the memory address by (i*(objective_type_size))
• E.g.
  ▫ char c, *cPtr = &c;
  ▫ int i, *iPtr = &I;
  ▫ double d, *dPtr = &d;
  ▫ printf("%p,%p,%p",cPtr++,iPtr++,dPtr++);
  ▫ printf("%p,%p,%p",cPtr,iPtr,dPtr);
• Output
  ▫ 0x01,0x02,0x06
  ▫ 0x02,0x06,0x0E
Array as a Parameter

- With respect to a function’s formal parameters only, C treats an array just like a pointer unlike other arrays
  - Therefore, you can change the value of the array name passed as parameter
    - Generally a bad idea, it serves no particular purpose
- E.g.

```c
void testFunction(int array[]){
    int i;
    array = &i; //does not throw error
}
```
Arrays as a Parameter

- When array is passed to a function, address of the array is copied onto the function parameter
  - i.e. pointer
- Therefore, function parameter may be declared in either fashion
  - `int sumArray(int a[], int size);`
  - `int sumArray(int *a, int size); //equivalent`
  - Code in function is free to use “a” as an array name or a pointer as it sees fit
- Compiler will always see array parameter as a pointer and error messages produced will refer to it as `int*` instead of array
Example

```c
Int sumArray(int a[], int size){
    Int k, sum = 0;
    For (k=0;k<size,k++)
        sum+= a[k];
    Return sum;
}
```

• Note that the size needs to be passed as a parameter which isn’t typically required in high-level languages
  ▫ Compiler does not know size of array, only knows address and type of first component
Array Sizes

- Managing array sizes in C is not a minor issue
- Going outside bounds of an array is not automatically checked, and can lead to serious program or system crashes
- Basic approaches for design of functions using arrays:
  - Use extra parameter to convey number of elements in array
  - Use termination value in array itself that can be discovered
    - Similar to null termination character in string
  - Use predetermined size for the array or some other predetermined method for determining it
    - Global constants
Strings and Pointers

• Recall that a string is represented as an array of characters terminated with null character
• A string constant may be declared as either char[] or char*
  ▫ E.g. char hello[] = “Hello!”; char* hello = “Hello!”
    • Almost equivalent
• Using a typedef could also be used to simplify coding
  ▫ typedef char* STRING; STRING hello = “Hello!”;
Example

• What does the following code do?
char hello[ ] = “Hello!”;
char * ptrChar;
ptrChar = &(hello[3]);
//What is printed from each of the following?
printf("%s
",hello);
printf("%s
",ptrChar);
printf("%s
",&(hello[3]));
printf("%s
",hello + 3);
printf("%s
",hello[3]); //x
Arrays of Pointers

- Since a pointer is a variable type, we can create an array of pointers just like we can create an array of other types.
- Common to use an array of pointers of type char*
  - Used to create an array of strings
Array of Pointers example

- `char *ravens[] = {“Flacco”,”Smith”,”SmithSR”}`
  Almost equivalent to
- `char **ravens= {“Flacco”,”Smith”,”SmithSR”}`
  - As a parameter `*ravens[]` produces `**ravens`

- Often seen for parameters for main functions
  - `Int main(int argc, char* argv[])`
  - `Int main(int argc, char ** argv)`
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char * name[3]={"Flacco","Smith","SmtihSR"}; //may be //stored in 
       read-only memory
    printf("%s",name[1]);
    fflush(stdout); //needed to ensure output displayed before // seg fault 
        (useful note for projects)
    name[1][2]='r'; // here
    printf("%s",name[1]);
    return 0;
}
Command Line Arguments

- Command Line Arguments are passed to your program as parameters to main
  - `int main(int argc, char* argv[])`
    - `argc` is # of arguments (size of `argv`)
    - `argv` is an array of strings which are command line arguments
      - `argv[0]` is always name of your executable program
  - E.g. Typing `myprog hello world 42` at linux prompt results in
    - `argc=4`