

POINTERS IN C

C Arrays and Pointers

- In C, pointers are more challenging
 - You will need to know
 - when to use a pointer
 - when to *dereference* the pointer
 - when to pass an address to a variable rather than the variable itself
 - when to use pointer arithmetic to change the pointer
 - how to use pointers without making your programs unreadable
 - Basically, you have to learn how to not “shoot yourself in the foot” with pointers

The Basics

- A pointer is merely an address of where a datum or structure is stored
 - all pointers are typed based on the type of entity that they point to
 - to declare a pointer, use * preceding the variable name as in `int *x;`
- To set a pointer to a variable's address use & before the variable as in `x = &y;`
 - & means “return the memory address of”
 - in this example, x will now point to y, that is, x stores y's address
- If you access x, you merely get the address
- To get the value that x points to, use * as in `*x`
 - `*x = *x + 1;` will add 1 to y
- * is known as the *indirection* (or dereferencing) operator because it requires a second access
 - that is, this is a form of indirect addressing

Example Code

```
int x = 1, y = 2, z[10];
int *ip;           // ip is a pointer to an int, so it can point to x, y, or an element of z

ip = &x;           // ip now points at the location where x is stored
y = *ip;           // set y equal to the value pointed to by ip, or y = x
*ip = 0;           // now change the value that ip points to to 0, so now x = 0
                  // but notice that y is unchanged
ip = &z[0];        // now ip points at the first location in the array z

*ip = *ip + 1;     // the value that ip points to (z[0]) is incremented
```

```
int x, *y, z, *q;
x = 3;
y = &x;           // y points to x
printf("%d\n", x); // outputs 3
printf("%d\n", y); // outputs x's address, will seem like a random number to us
printf("%d\n", *y); // outputs what y points to, or x (3)
printf("%d\n", *y+1); // outputs 4 (print out what y points to + 1)
printf("%d\n", *(y+1)); // this outputs the item after x in memory – what is it?
z = *(&x);         // z equals 3 (what &x points to, which is x)
q = &*y;           // q points to 3 – note *& and &* cancel out
```

Arrays and Pointers

- We declare an array using [] in our declaration following the variable name
 - `int x[5];` // unlike Java, we can't do `int[] x;`
- You must include the size of the array in the [] when declaring unless you are also initializing the array to its starting values as in:
 - `int x [] = {1, 2, 3, 4, 5};`
 - you can also include the size when initializing as long as the size is \geq the number of items being initialized (in which case the remaining array elements are uninitialized)
- As in Java
 - you access array elements just as in Java as in `x[4]`
 - array indices start at 0
 - arrays can be passed as parameters, the type being received would be denoted as `int x[]`
- Arrays in C are interesting because they are pointed to
 - the variable that you declare for the array is actually a pointer to the first array element
- You can interact with the array elements either through pointers or by using []
- One of the intriguing features of pointers in C is the ability to manipulate the pointers through pointer arithmetic – a pointer is an int value, so we can add or subtract
 - this will be used for stepping through arrays rather than using array indices

Using Pointers with Arrays

- Recall in an earlier example, we did `ip = &z[0];`
- This sets our pointer to point at the first element of the array
 - In fact, `z` is a pointer as well and we can access `z[0]` either using `z[0]`, `*ip`, or `*z`
- What about accessing `z[1]`?
 - We can do `z[1]` as usual, or we can add 1 to the location pointed to by `ip` or `z`, that is `*(ip+1)` or `*(z+1)`
 - While we can reset `ip` to be `ip = ip+1`, we cannot reset `z` to be `z = z+1`
 - adding 1 to `ip` will point to `z[1]`, but if `z = z + 1` were legal, we would lose access to the first array location since `z` is our array variable
- Notice that `ip=ip+1` (or `ip++`) moves the pointer 4 bytes instead of 1 to point at the next array location
 - this is done no matter what size the element is
 - if the array were an array of doubles, the increment would move `ip` to point 8 bytes away, if the array were chars, then `ip` would be 1 byte further
- We could declare our arrays as pointers
 - notably, we might do this for our formal parameters as this better describes what we are dealing with)
- For instance
 - `function1(int *array)` rather than `function1(int[] array)`

Iterating Through the Array

- Here we see two ways to iterate through an array, the usual way, but also a method using pointer arithmetic

```
int j;  
for(j = 0; j < n; j++)  
    a[j]++;
```

```
int *pj;  
for(pj = a; pj < a + n; pj++)  
    (*pj)++;
```

- Let's consider the code on the right:
 - pj is a pointer to an int
 - We start with pj pointing at a, that is, pj points to a[0]
 - The loop iterates while $pj < a + n$
 - pj is a pointer, so it is an address
 - a is a pointer to the beginning of an array of n elements so $a + n$ is the size of the array
 - $pj++$ increments the pointer to point at the next element in the array
 - The instruction $(*pj)++$ says “take what pj points to and increment it”
 - NOTE: $(*pj)++$; increments what pj points to, $*(pj++)$; increments the pointer to point at the next array element
 - what do each of these do? $*pj++$; $++*pj$;

Array Example Using a Pointer

```
int x[4] = { 12, 20, 39, 43}, *y;  
y = &x[0];           // y points to the beginning of the array  
printf("%d\n", x[0]); // outputs 12  
printf("%d\n", *y);   // also outputs 12  
printf("%d\n", *y+1); // outputs 13 (12 + 1)  
printf("%d\n", (*y)+1); // also outputs 13  
printf("%d\n", *(y+1)); // outputs x[1] or 20  
y+=2;                // y now points to x[2]  
printf("%d\n", *y);   // prints out 39  
*y = 38;              // changes x[2] to 38  
printf("%d\n", *y-1); // prints out x[2] - 1 or 37  
*y++;                // sets y to point at the next array element  
printf("%d\n", *y);   // outputs x[3] (43)  
(*y)++;              // sets what y points to to be 1 greater  
printf("%d\n", *y);   // outputs the new value of x[3] (44)
```


Strings

- There is no string type, we implement strings as arrays of chars
 - `char str[10];` // str is an array of 10 chars or a string
 - `char *str;` // str points to the beginning of a string of unspecified length
- There is a `string.h` library with numerous string functions
 - they all operate on arrays of chars and include:
 - `strcpy(s1, s2)` – copies s2 into s1 (including ‘\0’ as last char)
 - `strncpy(s1, s2, n)` – same but only copies up to n chars of s2
 - `strcmp(s1, s2)` – returns a negative int if $s1 < s2$, 0 if $s1 = s2$ and a positive int if $s1 > s2$
 - `strncmp(s1, s2, n)` – same but only compares up to n chars
 - `strcat(s1, s2)` – concatenates s2 onto s1 (this changes s1, but not s2)
 - `strncat(s1, s2, n)` – same but only concatenates up to n chars
 - `strlen(s1)` – returns the integer length of s1
 - `strchr(s1, ch)` – return a pointer to the first occurrence of ch in s1 (or NULL if ch is not present)
 - `strrchr(s1, ch)` – same but the pointer points to the last occurrence of ch
 - `strpbrk(s1, s2)` – return a pointer to the first occurrence of any character in s1 that matches a character in s2 (or NULL if none are present)
 - `strstr(s1, s2)` – substring, return a pointer to the char in s1 that starts a substring that matches s2, or NULL if the substring is not present

Implementing Some of These

```
int strlen(char *s)
{
    int n;
    for(n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

```
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];
}
```

```
void strcpy(char *s, char *t)
{
    int i = 0;
    while((s[i] = t[i]) != '\0')
        i++;
}
```

```
void strcpy(char *s, char *t)
{
    while((*s = *t) != '\0')
    {
        s++; t++;
    }
}
```

```
int strcmp(char *s, char *t)
{
    for(; *s == *t; s++, t++)
        if(*s == '\0') return 0;
    return *s - *t;
}
```

```
void strcpy(char *s, char *t)
{
    while((*s++ = *t++) != '\0');
}
```

Notice in the second strcmp and second and third strcpy the use of pointers to iterate through the strings

The conciseness of the last strcmp and strcpy make them hard to understand

More On Pointer Arithmetic

- We can also perform subtraction on pointers

```
int a[10] = { ... };  
int *ip;  
for(ip = &a[9]; ip >= a; ip--)
```

- Here, we pass to a function the address of the third element of an array (&a[2]) and use pointer subtraction to get to a[0] and a[1])

```
int a[3] = { ... };  
printf("%d", addem(&a[2]));
```

```
int addem(int *ip)  
{  
    int temp;  
    temp = *ip + *(ip - 1) + *(ip - 2);  
    return temp;  
}
```

Recall:

a[0] = *a and
a[i] = *(a + i)

If a is an array, and p = &a[0] then we can reference array elements as a[i], *(p+i), but we can also reference them as p[i] and *(a+i) – that is, a and p are both pointers to the array And can be dereferenced by * or by []

Multidimensional Arrays

- As in Java, C allows multidimensional arrays by using more []
 - Example: `int matrix[5][10];`
- Some differences:
 - Because functions can be compiled separately, we must denote all but one dimension of a multiple dimensional array in a function's parameter list
 - `void afunction(int amatrix[][10]);`
 - Because arrays are referenced through pointers, there are multiple ways to declare and access 2+ dimensional arrays
 - This will be more relevant when dealing with an array of strings (which is a 2-D array)

```
int a[10][20];  
int *a[10];  
int **a;
```

`*a[4]` –first element of 5th array element
`*a[9]` –first element of 10th array element
`**a` –first element of `a[0]`

```
int *a[3];           // array of 3 pointers  
int x[2] = { 1, 2};  
int y[3] = { 3, 4, 5};  
int z[4] = { 6, 7, 8, 9};  
*a = &x[0];          // a[0] points to x[0]  
*(a+1) = &y[0];       // a[1] points to y[0]  
*(a+2) = &z[0];       // a[2] points to z[0]  
// array a is a jagged array, it is not  
// rectangular, or of equal dimensions
```

Pointers to Pointers

- As indicated in the last slide, we can have an array of arrays which is really an array of pointers or pointers to pointers
 - We may wish to use pointers to pointers outside of arrays as well, although it is more common that pointers to pointers represent array of pointers
 - Consider the following:

```
int a;  
int *p;  
int **q;  
a = 10;  
p = &a;  
q = &p;  
printf("%d", **q);
```

We dereference our pointer p with *p but we dereference our pointer to a pointer q with **q

*q is actually p, so **q is a

// outputs 10

Arrays of Strings Implementation

- We could implement an array of strings as a 2-D array of chars
 - `char array[10][10];`
- This has two disadvantages
 - All strings will be 10 chars long
 - Requires 2 nested for-loops for most operations such as string comparison or string copying, which can become complicated
- Instead, we will implement our array of strings as an array of pointers
 - `char *array[10];`
- Each pointer points to one string
 - Follow the string through the pointer
 - Go to the next string using a for-loop
 - Because `strcpy`, `strcmp`, `strlen` all expect pointers, we can use these by passing an array element (since each array element is a pointer to a string)

Example

```
char *x[ ] = {"hello\0", "goodbye\0", "so long\0", "thanks for all the fish\0"};
    // our array of strings x is a set of 4 pointers
char *y; // let y be a pointer to a char so it can be used to move through a single string
int i;
for(i=0;i<4;i++)    // iterate for each string in x
{
    y = x[i]; // x[i] is an array, x is really a pointer, so this sets y to x's starting addr.
    while(*y!='\0')    // while the thing y points to is not the end of a string
    {
        printf("%c", *y);    // print what y points to
        y++;                // and go on to the next char in x
    }
    printf("\n");           // separate strings in output with \n
}
```

- Notice that if we had used `char x[][] = { ... };` then the storage space would have been 4 strings of length 23 (the length of the longest string) or 92 bytes instead of 42 bytes as it is above

Passing Arrays

- When an array is passed to a function, what is being passed is a pointer to the array
 - In the formal parameter list, you can either specify the parameter as an array or a pointer
- Because you can compile functions separately, the compiler must be able to “know” about an array being passed in to a function, so you must specify all (or most) of the definition:
 - The type and all dimensions except for the first

```
int array[100];  
...  
afunction(array);  
...
```

```
void afunction(int *a) {...}  
or  
void afunction(int a[ ]) {...}
```

```
int array[5][10][15];  
...  
afunction(array);  
...  
void afunction(int a[ ][10][15]) {...} or  
void afunction(int *a[10][15]) {...} or  
void afunction(int a[5][10][15]) {...} or  
void afunction(int **a[15]) {...} etc
```


Assignment

- What do you mean by pointers? What are the disadvantages of using pointers?