Programming in C

Pointers and Arrays

Pointers and Arrays

- In C, there is a strong relationship between pointers and arrays.
- The declaration `int a[10];` defines an array of 10 integers.
- The declaration `int *p;` defines `p` as a "`pointer to an int`".
- The assignment `p = a;` makes `p` an alias for the array and sets `p` to point to the first element of the array. (We could also write `p = &a[0];`)
- We can now reference members of the array using either `a` or `p`

```
a[4] =9;
p[3] = 7;
int x = p[6] + a[4] * 2;
```

Pointer Arithmetic

- If `p` points to a particular element of an array, then `p + 1` points to the next element of the array and `p + n` points n elements after `p`.

- The meaning a "adding 1 to a pointer" is that `p + 1` points to the next element in the array, REGARDLESS of the type of the array.

Pointer Arithmetic (cont'd)

- If `p` is an alias for an array of ints, then `p[ k ]` is the `k`-th int and so is `*(p + k)`.
- If `p` is an alias for an array of doubles, then `p[ k ]` is the `k`-th double and so is `*(p + k)`.
- Adding a constant, `k`, to a pointer (or array name) actually adds `k * sizeof(pointer type)` to the value of the pointer.
- This is one important reason why the type of a pointer must be specified when it's defined.

Pointers and Arrays Summary

- The name of an array is equivalent to a pointer to the first element of the array and vice-versa.
- Therefore, if `a` is the name of an array, the expression `a[ i ]` is equivalent to `*(a + i)`.
- It follows then that `&a[ i ]` and `(a + i)` are also equivalent. Both represent the address of the `i-th` element beyond `a`.
- On the other hand, 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 and vice-versa.

So, what's the difference?

- If the name of an array is synonymous with a pointer to the first element of the array, then what's the difference between an array name and a pointer?

An array name can only "point" to the first element of its array. It can never point to anything else.

A pointer may be changed to point to any variable or array of the appropriate type

Array Name vs Pointer

```
int g, grades[ ] = {10, 20, 30, 40 }, myGrade = 100, yourGrade = 85,
*pGrade;

/* 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 */
grades = &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);
```

pointerGotcha.c

- But what if `p` doesn't point to an element in an array?

```
int a = 42;
int *p = &a;

printf( "%d\n", *p);          // prints 42
++p;                          // to what does p point now?
printf( "%d\n", *p);          // what gets printed?
printf("%d\n", *(p + 5));     // what gets printed?
```

Printing an Array

- The code below shows how to use a parameter array name as a pointer.
- Although this is not common and is more complex than it needs to be, it illustrates the important relationship between pointers and array names.

```
void printGrades( int grades[ ], int size )
{
  int i;
  for (i = 0; i < size; i++)
   printf( "%d\n", *grades );
   ++grades;
}
```

- What about this prototype?

```
void printGrades( int *grades, int size );
```

Pointers and 2D Arrays

- A 2D array is essentially an array of pointers (each row is a 1D array).
- The name of the 2D array is a pointer to its first row.

Example:

```
int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };
int (*ptr)[3] = matrix; // Pointer to the first row
```

Syntax 2D array

***(*(array_name + row_index) + column_index);**

Explanation:

- **array_name + row_index**
  gives the address of the row_index-th row.
- ***(array_name + row_index)**
  gives the address of the first element in the row_index-th row.
- ***(array_name + row_index) + column_index**
  gives the address of the column_index-th element in the row_index-th row.
- ***(*(array_name + row_index) + column_index)**
  gives the value at that address.

Example

```
int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };

// Access element at row 1, column 2
int value = *(*(matrix + 1) + 2); // value = 6

// Modify element at row 0, column 1
*(*(matrix + 0) + 1) = 10; // matrix[0][1] is now 10
```

Equivalence Between Index and Pointer Methods

Index Method	Pointer Method
matrix[i][j]	*(*(matrix + i) + j)
&matrix[i][j]	*(matrix + i) + j
matrix[i]	*(matrix + i)
matrix	matrix (pointer to the first row)

Index Method Example

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

Pointer Method Example

```
    for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", *(*(matrix + i) + j));
    }
    printf("\n");
}
```

Passing Arrays

- Arrays are passed "by reference".
- When an array is passed to a function, the address of the array is copied onto the function parameter. Since an array address is a pointer, the function parameter may be declared in either fashion. E. g.

```
int sumArray( int a[ ], int size)
```

  is equivalent to

```
int sumArray( int *a, int size)
```

- The code in the function is free to use "`a`" as an array name or as a pointer as it sees fit.
- The compiler always sees "`a`" as a pointer. In fact, any error messages produced will refer to "`a`" as an `int *`

sumArray

```
int sumArray( int a[ ], int size)
{
   int k, sum = 0;
   for (k = 0; k < size; k++)
        sum += a[ k ];
   return sum;
}
```

sumArray (2)

```
int sumArray( int a[ ], int size)
{
   int k, sum = 0;
   for (k = 0; k < size; k++)
        sum += *(a + k);
   return sum;
}

int sumArray( int a[ ], int size)
{
   int k, sum = 0;
   for (k = 0; k < size; k++)
   }
        sum += *a;
        ++a;
   }
   return sum;
}
```

Strings revisited

Recall that a string is represented as an array of characters terminated with a null (\0) character.

As we've seen, arrays and pointers are closely related. A string constant may be declared as either

```
char[ ] or  char*
```

as in

```
char hello[ ] = "Hello Bobby";
```

or (almost) equivalently

```
char *hi = "Hello Bob";
```

A `typedef` could also be used to simplify coding

```
typedef char* STRING;
STRING hi = "Hello Bob";
```

Arrays of Pointers

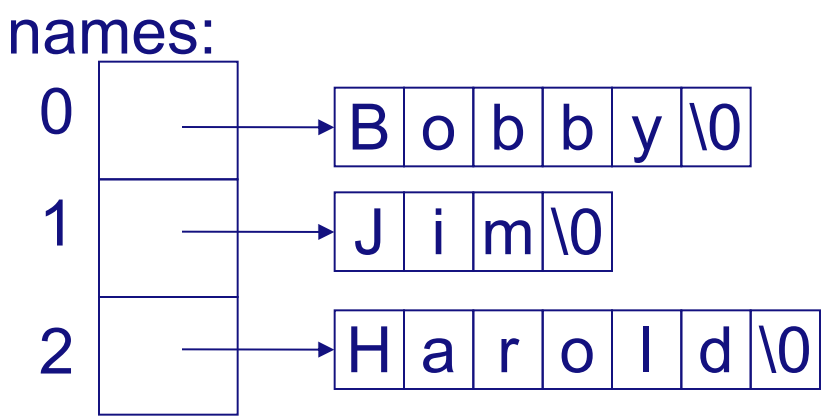
Since a pointer is a variable type, we can create an array of pointers just like we can create any array of any other type.

Although the pointers may point to any type, the most common use of an array of pointers is an array of `char*` to create an array of strings.

Boy's Names

A common use of an array of pointers is to create an array of strings. The declaration below creates an initialized array of strings `(char *)` for some boys names. The diagram below illustrates the memory configuration.

```
char *name[] = { "Bobby", "Jim", Harold" };
```



Command Line Arguments

- **Command line arguments are passed to your program as parameters to main.**

```
int main( int argc, char *argv[ ] )
```

  - `argc` **is the number of command line arguments (and hence the size of** `argv`**)**
  - `argv` **is an array of strings which are the command line arguments. Note that** `argv[0]` **is always the name of your executable program.**

- **For example, typing** `myprog hello world 24` **at the linux prompt results in**
  - **argc = 4**
  - **argv[0] = "myprog"**
  - **argv[1] = "hello"**
  - **argv[2] = "world"**
  - **argv[3] = "24"**

- **Note that to use argv[3] as an integer, you must convert if from a string to an int using the library function** `atoi( )`**.**
  E.g. `int age = atoi( argv[3] );`

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    // Print the number of command line arguments
    printf("argc = %d\n", argc);

    // Print each command line argument
    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    // Example of converting argv[3] to an integer
    if (argc > 3) {
        int age = atoi(argv[3]);
        printf("Converted argv[3] to integer: %d\n", age);
    } else {
        printf("Not enough arguments to convert argv[3] to an integer.\n");
    }

    return 0;
}
```