CCPROG2 Notes [Arrays to Algorithms]

Data Types

```
DATA TYPES
char      char *      char []    char [][]   ...   char *[]
int       int *       int []     int [][]    ...   int *[]
float     float *     float []   float [][]  ...   float *[]
double    double *    double []  double [][] ...   double *[]
void      void *
          (generic pointer data type)
```

Determining the data type of an expression

Example:

Given the ff declaration:

```
double M[2][3];
```

What is the data type of

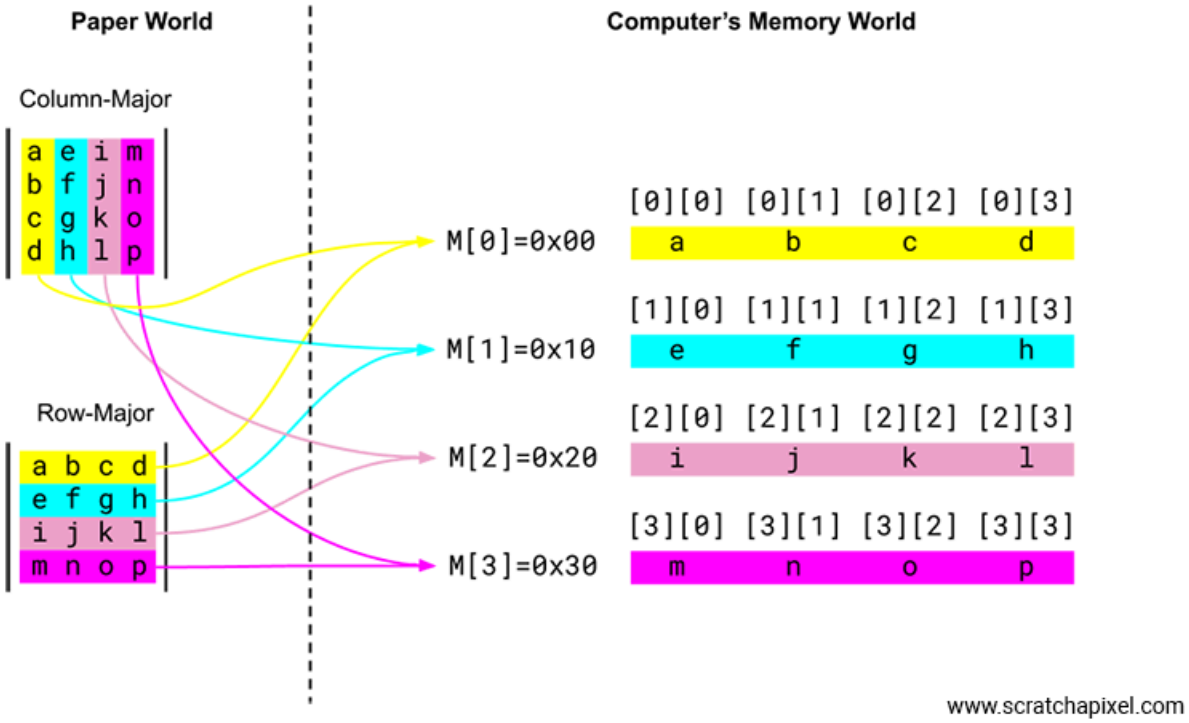
a. M[i][j]
   double

b. M[1] // the 2nd row of the 2D array
   double [] // 1D double array data type

c. &M[1][2]
   double *

d. M
double [][] // 2D array data type

Row & Col Major Order



Array Indexing to Pointer Notation

Terms:

- dereference (*)
- address of (&)
- array name (A)

Examples:

```
// 1D Arrays
A[0] == *(A)
A[i] == *(A + i)
&A[0] == A
&A[i] == A + i

// 2D Arrays
A[2][3] == *(*(A + 2) + 3)
A[1][4] == *(* A + 9) //  row major order
// it's the same as saying:
*(n[0] + 9) // ← the 9th element NOT EQUIVALENT to n[0][9]

// refer to this visualization for the last example:
Row 0 → A[0][0], A[0][1], A[0][2], A[0][3], A[0][4]
//        0       1       2       3       4
Row 1 → A[1][0], A[1][1], A[1][2], A[1][3], A[1][4]
//        5       6       7       8      **9**
Row 2 → A[2][0], A[2][1], A[2][2], A[2][3], A[2][4]
```

The General Formula

For any
2D array `A[m][n]`, the element `A[i][j]` is stored in memory at:

```
A[i][j] == *(*A + (i * n + j))
```

Where:

- `i * n` jumps to row `i`
- `j` moves `j` columns forward

More Examples

```
board[row][col]
*(*(board + row) + col)
*(board + row)[col]
```

Finding nth Element/Byte Formulas

nth element = Adress at first elem + (n - 1) * (bytes per element)

last byte = starting address + (nth element (0 indexed) * data type size - 1

<string.h> Functions

strcat(dest, src)

- Concatenates two strings (appends the string from the src to the dest).

strcpy(dest, src)

- Copy one string to another.

strlen(str)

- Returns the length of the string.

strcmp(str1, str2)

- Return values:
  - **Zero (0)**: It returns zero when **all of the characters at given indexes in both strings are the same.**
  - **Positive (> 0)**: Returns a value greater than zero is returned when the first not-matching character in **s1** has a greater ASCII value than the corresponding character in **s2.**
  - **Negative (< 0)**: Returns a value less than zero is returned when the first not-matching character in **s1** has a lesser ASCII value than the corresponding character in **s2**.

❗ **NOTE:** Most string.h functions only read/copy until the NULL BYTE ('\0')

Algorithms

- Note that the provided sample algorithms are for 1D Arrays.
- If you want to apply a 1D Array Algorithm to a 2D array for example, you:
  - **Introduce a Nested Loop:**
    - **Outer loop**: Handles the **rows** of the 2D array.
    - **Inner loop**: Handles the **columns** of the 2D array.
  - **Adjust the Indexing:**
    - In a **2D array**, you use two indices: `a[i][j]`, where `i` represents the row index, and `j` represents the column index.

▼ Common Array/String Algorithms

Copy Array

```
void copyArray(int A[], int B[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      B[i] = A[i];
}
```

Copy String from Source to Dest

```
void copyString(char *dest, char *source) // pointer notation in formal parame
{
   int len = strlen(source);
   int i;

   for (i = 0; i <= len; i++)
```

```
        dest[i] = src[i]
}
```

Reverse Array (General)

```
void reverseArray(int A[], int B[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        B[i] = A[n - 1 - i];
}
```

Copy String in Reverse Order

```
void reverseString(char dest[], const char source[]) {
    int len = strlen(source);
    int i;

    // Reverse the string
    for (i = 0; i < len; i++) {
        dest[len - i - 1] = source[i];
    }

    dest[len] = '\0';  // Null-terminate the reversed string
}
```

Swap

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void swapStrings(char a[], char b[])
{
   char temp[strlen(a) + 1];
   strcpy(temp, a);
   strcpy(a, b);
   strcpy(b, temp);
}
```

Minimum

```
// Find the smallest value in the group. Assume that the values are unique.
// Return the index of the minimum element.
int
Minimum(int A[], int n)
{
    int i;
    int min = 0; // assume that 1st element is the smallest
    for (i = 1; i < n; i++) // note: start with index 1 not 0
          if (A[min] > A[i])
                min = i; // update the minimum index

    return min;
}
```

Maximum

```
int Maximum(int A[], int n)
{
   int i;
   int max = 0;  // Assume that the first element is the largest
   for (i = 1; i < n; i++) // Start with index 1, not 0
      if (A[max] < A[i])
         max = i;  // Update the maximum index if a larger element is found
```

```
    return max;  // Return the index of the maximum element
}
```

Average

```
float Average(int A[], int n)
{
    int i;
    int sum = 0;
    for (i = 0; i < n; i++)
        sum += A[i];

    return (float)sum / n;
}
```

▼ Search Algorithms

Linear Search

```
int
LinearSearch(int key, int A[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (key == A[i]) // found
            return i;
    return -1; // not found if we reach this point
}
```

String Ver.

```
int
SearchString(int key, int A[], int n)
```

```
{
   int i;
   for (i = 0; i < n; i++)
      if (strcmp(key, A[i]) == 0) // found
         return i;
   return -1; // not found if we reach this point
}
```

Binary Search

```
int BinarySearch(int key, int A[], int n)
{
   int low = 0, mid;
   int high = n - 1;

   while (low <= high)
   {
      mid = low + (high - low) / 2;

      if (key == A[mid])  // Direct integer comparison
         return mid;  // Found
      else if (key < A[mid])
         high = mid - 1;  // Search lower half
      else
         low = mid + 1;   // Search upper half
   }

   return -1;  // Not found
}
```

String Ver.

```
int
SearchString(int key, int A[], int n)
{
```

```
    int low = 0, mid;
    int high = n - 1;

    while (low <= high)
    {
        mid = low + (high - low) / 2;

        if (strcmp(key, A[mid]) == 0)
            return mid;  // Found
        else if (strcmp(key, A[mid]) < 0)
            high = mid - 1;  // Search lower half
        else
            low = mid + 1;   // Search upper half
    }

    return -1;  // Not found
}
```

▼ Sorting Algorithms

Selection Sort

- involves "selecting" the smallest element again and again
- keeps track of one element at a time
- **Applies `n - 1` iterations/passes**

```
void
SelectionSort(int A[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++) // performs n-1 passes ONLY
    {
        min = i; // min is the index of the lowest element
        for (j = i + 1; j < n; j++)
            if (A[min] > A[j])
```

```
            min = j;
    // swap A[i] with A[min]
        if (i != min)
        {
            temp = A[i];
            A[i] = A[min];
            A[min] = temp;
        }
    }
}
```

String Ver.

```
void
SelectionSort(int A[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++)
    {
        min = i; // min is the index of the lowest element
        for (j = i + 1; j < n; j++)
            if (strcmp(A[min], A[j]) > 0)
                min = j;
    // swap A[i] with A[min]
        if (i != min)
        {
            temp = A[i];
            A[i] = A[min];
            A[min] = temp;
        }
    }
}
```

Bubble Sort

- repeatedly steps through the list, compares adjacent pairs of elements, and swaps them if they are in the wrong order.
- works by repeatedly swapping elements to “bubble” larger elements to the end.

```
void swap(int* arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {

        // Last i elements are already in place, so the loop
        // will only num n - i - 1 times
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr, j, j + 1);
        }
    }
}
```

TYPEDEF Declaration w/o Using an Alias

General Rule for Expanding Typedefs

- Always replace the alias with its full expanded type.
- When dealing with multi-dimensional arrays:
    - Start from the innermost dimension and work outwards.
    - Use the typedef alias to uncover the original type structure.

Example:

```
typedef int alpha[10];  // equivalent to: int alpha[10];
typedef alpha beta[5];   // equivalent to: int beta[5][10];

int main()
{
   beta x[3][2];  // equivalent to: int x[3][2][5][10];
}
```

- The dimensions of `x` automatically goes to the leftmost, then is followed by the actual dimensions in the typedef declaration
  - Giving us: `x[3][2][5][10]`

    `// x, beta, alpha`
- Basically, append the dimensions of the typedef aliases to the **right ^^**

Another Example (By Sir Flo):

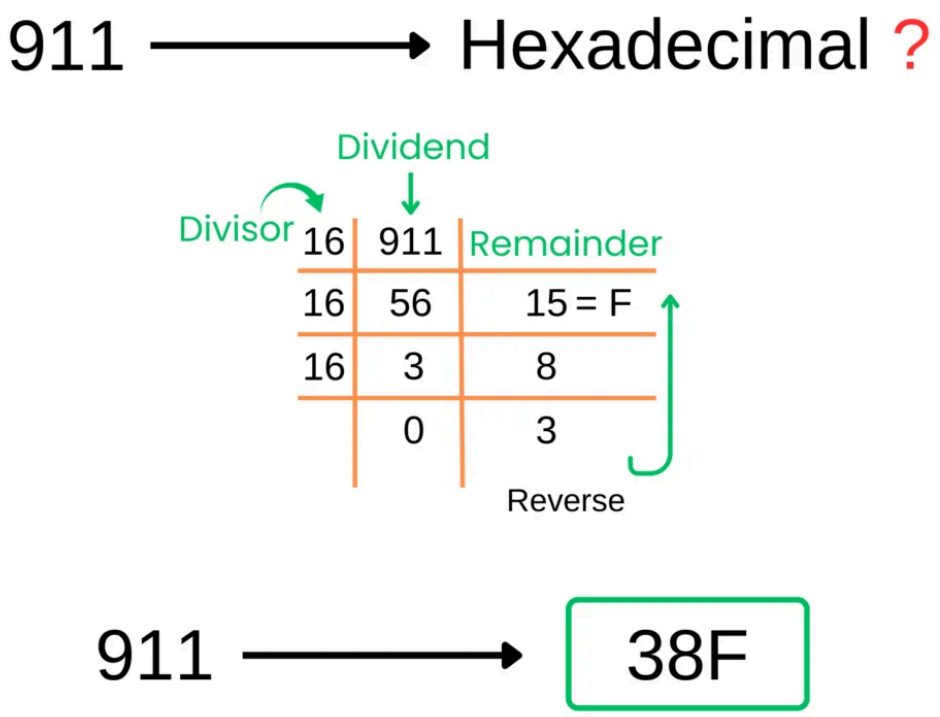
```
typedef char Str10[11];

Str10 Z[2][3][4];

// Declaration w/o Using an Alias
char Z[2][3][4][11];
```

Decimal to Hexadecimal

- just keep dividing by 16 until you reach a dividend of zero, then read the remainders from bottom to top.
  - recall: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.



I/O Redirection

Input redirection only	exe < inputfile
Output redirection only	exe > outputfile
Both 1/O redirection	exe < inputfile > outputfile