Contents

1 Intro to array **4**
1.1 array on integer . . . 4
1.2 Declaring arrays . . . 4
1.2.1 int A[5]; where we get garbage values . . . 4
1.2.2 int A[5] = {2,4,6,8,10}; all values initialized . . . 4
1.2.3 int A[5] = {2.4}; only first 2 are initialized, rest is 0 . . . 4
1.2.4 int A[5] = {0}; all zero . . . 4
1.2.5 int A[] = {2,4,6,8,10}; automatically create A[5] . . . 4
1.3 Traverse using for loop . . . 4
1.4 To print an element at position 2 . . . 4
1.4.1 A[2]; . . . 4
1.4.2 2[A]; . . . 4
1.4.3 *(A+2); . . . 4

2 Static vs Dynamic array **4**
2.1 Static . . . 4
2.1.1 size cannot be modified. Memories created on STACK . . . 4
2.1.2 C: size decided at compilation time . . . 4
2.1.3 C++: size at run time. Eg. cin » n; int A[n]; . . . 4
2.2 Dynamic . . . 4
2.2.1 on HEAP . . . 4
2.2.2 Note: remember to free memory . . . 4
2.2.3 Access on heap; . . . 4

3 Demo static dynamic array **5**

4 Increase array size **5**
4.1 int *p = new int[5] . . . 5
4.2 Take another pointer: int *q = new int[10] => Create larger array separately . . . 5
4.3 Copy p[i] onto q[i] . . . 5
4.4 delete/free memory in p . . . 5
4.5 tells p to to point to q => both p and q points to the same larger array . . . 5
4.6 free q . . . 5
4.7 demo . . . 5

5 2D array **8**
5.1 Method 1: int A[3][4] => 3 row, 4 col on STACK . . . 8
5.1.1 Memory allocates like a 1D array of 12 memory blocks . . . 8
5.2 Method 2: int *A[3] => array of int pointers of size 3 on STACK, actual array on HEAP . . . 8
5.2.1 block 0 [ ] -> want array of size 4 here | | | | | . . . 8
5.2.2 block 1 [ ] -> want array of size 4 here | | | | | . . . 8

5.2.3 block 2 [ ] -> want array of size 4 here | | | | | 8
5.2.4 A[0] = new int[4] => create array of size 4 for block 0 .. 8
5.2.5 A[1] = new int[4] and A[2] = new int[4] 8
5.3 Method 3: int **A; everything on HEAP 8
5.4 A = new int*[3] create array of int pointers (like above) on HEAP 8
5.5 A[0] = new int[4] on HEAP 8
5.6 A[1] = new int[4] on HEAP 8
5.7 A[2] = new int[4] on HEAP 8
5.8 Demo : ‘`2darray.c`‘ 8

6 1D Array in compilers **8**
6.1 int x = 10; compiler allocates address for x and store 10 at that address 8
6.2 Compiler memory to address 8
6.3 int A[5] = {.....}; 8
6.4 A[i] = Base index + index * sizeof (data type) 8
6.5 A[3] = L0 + 3 * 2 8
6.6 If index starts at 1: A[i] = Base index + (index-1)*sizeof(data type) 8

7 2D Array in compilers **8**
7.1 ROW MAJOR MAPPING 8
7.1.1 Elements store row by row in A[m x n] 8
7.1.2 A = a00 a01 a02 a03 | a10 a11 a12 a13 | a20 a21 a22 a23 | 8
7.1.3 Say we access A[1][2] and say a00 has address 200 8
7.1.4 In general A[i][j] = **L0 + [i*n+j]*sizeof(data type)** .. 9
7.1.5 If index starts at 1: A[i][j] = L0 + [(i-1)*n+(j-1)]*sizeof(data type) 9
7.2 COL MAJOR MAPPING 9
7.2.1 Map colum by colum 9
7.2.2 A = a00 a10 a20 | a01 a11 a21 | a02 a12 a21 | a03 a13 a23 | 9
7.2.3 Say we want A[1][2] 9
7.2.4 In general, **A[i][j] = L0 + [j*m + i]*sizeof)(data type)** 11

8 4D Array **11**
8.1 Type A[d1][d2][d3][d4] 11
8.2 Row major Add(A[i][i2][i3][i4]) = L0 + [i1*d2*d3*d4 + i2*d3*d4 + i3*d4 +i4]*sizeof(data) 11
8.3 Col major Add(A[i1][i2][i3][i4]) = L0 + [i4*d1*d2*d3 + i3*d1*d2 + i2*d1 + i1]*sizeof(data) 11

9 For nD array **11**
9.1 Row major mapping: L0 + $\mathrm{SUM_p}$ from 1 to n [$(\mathrm{i_p})$ * $\mathrm{product_q}$ = p + 1 to n of dq] * sizeof(datetype) 11
9.1.1 $O(n^2)$ 11

9.1.2 If rewrite by taking commons => O(n) -> **HOMER'S RULE** . . . 11

10 3D Array **11**
10.1 int A[l][m][n] . . . 11
10.2 Row major Addr(A[i][j][k]) = L0 + [i*m*n + j*n + k] + sizeof(datatype) 11
10.3 Colum major Addr(A[i][j][k]) = L0 + [k*m*l + j*l + i] + sizeof(datatype) 11

11 Quiz **11**
11.1 1. A[1. . . 10][1. . . 15] = A[m][n] . . . 11
11.1.1 L0 = 100 . . . 11
11.1.2 Row major Addr(A[i][j]) = L0 + [(i-1)*n+(j-1)]*sizeof(data type) . . . 11
11.1.3 100 + [(i-1)*15+(j-1)]*1 . . . 11
11.1.4 100 + (15i- 15 + j - 1)*1 . . . 11
11.1.5 100 + 15i - 15 + j - 1 . . . 11
11.1.6 84 + 15 i + j . . . 11
11.2 2. unsigned int x[4][3] = {. . . . . . }. Printf("%u, %u, %u", x + 3, *(x+3), *(x+2)+3) . . . 11
11.2.1 1 2 3 . . . 11
11.2.2 **4** 5 6 . . . 11
11.2.3 7 8 9 . . . 11
11.2.4 10 11 12 . . . 11
11.2.5 A = a00 a01 a02 | a03 a04 a05 | a06 a07 a08 | a09 a10 a11 | 11
11.2.6 %u, x + 3 => 2000 + (3*int) = > **2012 address** . . . 11
11.2.7 %u, *(x+3) => Gets value of address 2012 = 4 . . . 11
11.2.8 %u, **(x+2) + 3 => *6** . . . 11
11.2.9 testing: '`quizarray2.c`' . . . 11
11.3 4. ?X[?][?][?] . . . 11
11.3.1 t0 = i*1024 . . . 11
11.3.2 t1 = j + 32 . . . 11

12 Array ADT **11**
12.1 Perform various operations on an array . . . 11
12.2 Data . . . 11
12.2.1 Array space: say 10 . . . 11
12.2.2 Size . . . 11
12.2.3 length . . . 12
12.3 Operations . . . 12
12.3.1 display: printf ("%d", A[i]) in for loop . . . 12
12.3.2 add/append . . . 12
12.3.3 insert . . . 12
12.3.4 delete . . . 12
12.3.5 Linear search . . . 13
12.4 Demo: '`arrayADT.c`' . . . 13

1 Intro to array

1.1 array on integer

1.2 Declaring arrays

1.2.1 int A[5]; where we get garbage values

1.2.2 int A[5] = {2,4,6,8,10}; all values initialized

1.2.3 int A[5] = {2.4}; only first 2 are initialized, rest is 0

1.2.4 int A[5] = {0}; all zero

1.2.5 int A[] = {2,4,6,8,10}; automatically create A[5]

1.3 Traverse using for loop

1.4 To print an element at position 2

1.4.1 A[2];

1.4.2 2[A];

1.4.3 *(A+2);

2 Static vs Dynamic array

2.1 Static

2.1.1 size cannot be modified. Memories created on STACK

2.1.2 C: size decided at compilation time

2.1.3 C++: size at run time. Eg. cin » n; int A[n];

2.2 Dynamic

2.2.1 on HEAP

1. Create pointer int *p on STACK
2. C++: p = new int[5]; create 5 integer array on HEAP
3. C: p = (int*)malloc(sizeof(int)*5);

2.2.2 Note: remember to free memory

1. C++: delete []p; if p is used for an array we use []
2. C: free(p)

2.2.3 Access on heap;

1. p[0] = 5;

3 Demo static dynamic array

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

int main(){
  int A[5] = {2,4,6,8,10};
  int *p;
  int i;

  p=(int*)malloc(sizeof(int) * 5);
  p[0] = 3;
  p[1] = 5;
  p[2] = 7;
  p[3] = 9;
  p[4] = 11;

  for (int i = 0; i < 5 ; i++) {
    printf("%d\t%d\n", A[i], p[i]);
  }


  return 0;
}
```

4 Increase array size

4.1 int *p = new int[5]

4.2 Take another pointer: int *q = new int[10] => Create larger array separately

4.3 Copy p[i] onto q[i]

4.4 delete/free memory in p

4.5 tells p to to point to q => both p and q points to the same larger array

4.6 free q

4.7 demo

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

```
int main(){
  int *p, *q;

  p = (int*)malloc(sizeof(int) * 5);
  p[0] = 3;
  p[1] = 5;
  p[2] = 7;
  p[3] = 9;
  p[4] = 11;


  /* for (int i = 0; i < 5 ; i++) { */
  /*   printf("%d\n", p[i]); */
  /* } */

  q = (int*)malloc(sizeof(int) * 10);

  for (int i = 0; i < 5 ; i++) {
    q[i] = p[i];
  }

  free(p);
  p = q;
  q = NULL;

  for (int i = 0; i < 5 ; i++) {
    printf("%d\n", p[i]);
  }

  return 0;
}
```

5 2D array

5.1 Method 1: int A[3][4] => 3 row, 4 col on STACK

5.1.1 Memory allocates like a 1D array of 12 memory blocks

5.2 Method 2: int *A[3] => array of int pointers of size 3 on STACK, actual array on HEAP

5.2.1 block 0 [ ] -> want array of size 4 here | | | | |

5.2.2 block 1 [ ] -> want array of size 4 here | | | | |

5.2.3 block 2 [ ] -> want array of size 4 here | | | | |

5.2.4 A[0] = new int[4] => create array of size 4 for block 0

5.2.5 A[1] = new int[4] and A[2] = new int[4]

5.3 Method 3: int **A; everything on HEAP

5.4 A = new int*[3] create array of int pointers (like above) on HEAP

5.5 A[0] = new int[4] on HEAP

5.6 A[1] = new int[4] on HEAP

5.7 A[2] = new int[4] on HEAP

5.8 Demo : ‘`2darray.c`‘

6 1D Array in compilers

6.1 int x = 10; compiler allocates address for x and store 10 at that address

6.2 Compiler memory to address

6.3 int A[5] = {. . . ..};

6.4 A[i] = Base index + index * sizeof (data type)

6.5 A[3] = L0 + 3 * 2

6.6 If index starts at 1: A[i] = Base index + (index-1)*sizeof(data type)

7 2D Array in compilers

7.1 ROW MAJOR MAPPING

7.1.1 Elements store row by row in A[m x n]

7.1.2 A = a00 a01 a02 a03 | a10 &a11 a12 a13 | a20 a21 a22 a23 |

7.1.3 Say we access A[1][2] and say a00 has address 200

1. A[1][2] = 200 + [4 + 2]*sizeof(int)

7.1.4 In general A[i][j] = L0 + [i*n+j]*sizeof(data type)

7.1.5 If index starts at 1: A[i][j] = L0 + [(i-1)*n+(j-1)]*sizeof(data type)

7.2 COL MAJOR MAPPING

7.2.1 Map colum by colum

7.2.2 A = a00 a10 a20 | a01 a11 a21 | a02 a12 a21 | a03 a13 a23 |

7.2.3 Say we want A[1][2]

1. A[1][2] = 200 + [2 * 3 + 1]*sizeof(int)

7.2.4 In general, A[i][j] = L0 + [j*m + i]*sizeof)(data type)

8 4D Array

8.1 Type A[d1][d2][d3][d4]

8.2 Row major Add(A[i][i2][i3][i4]) = L0 + [i1*d2*d3*d4 + i2*d3*d4 + i3*d4 +i4]*sizeof(data)

8.3 Col major Add(A[i1][i2][i3][i4]) = L0 + [i4*d1*d2*d3 + i3*d1*d2 + i2*d1 + i1]*sizeof(data)

9 For nD array

9.1 Row major mapping: L0 + SUM_p from 1 to n [(i_p) * product_q = p + 1 to n of dq] * sizeof(datetype)

9.1.1 $O(n^2)$

9.1.2 If rewrite by taking commons => O(n) –> HOMER'S RULE

10 3D Array

10.1 int A[l][m][n]

10.2 Row major Addr(A[i][j][k]) = L0 + [i*m*n + j*n + k] + sizeof(datatype)

10.3 Colum major Addr(A[i][j][k]) = L0 + [k*m*l + j*l + i] + sizeof(datatype)

11 Quiz

11.1 1. A[1. . . .10][1. . . 15] = A[m][n]

11.1.1 L0 = 100

11.1.2 Row major Addr(A[i][j]) = L0 + [(i-1)*n+(j-1)]*sizeof(data type)

11.1.3 100 + [(i-1)*15+(j-1)]*1

11.1.4 100 + (15i- 15 + j - 1)*1

11.1.5 100 + 15i - 15 + j - 1

11.1.6 84 + 15 i + j

11.2 2. unsigned int x[4][3] = {. . . . . .}. Printf("%u, %u, %u", x + 3, *(x+3), *(x+2)+3)

11.2.1 1 2 3

11.2.2 4 5 6

11.2.3 7 8 9

11.2.4 10 11 12

11.2.5 A = a00 a01 a02 | a03 a04 a05 | a06 a07 a08 | a09 a10 a11 |

11.2.6 %u, x + 3 => 2000 + (3*int) = > 2012 address

   (a) int A[10]

2. dynamic

   (a) int * A

   (b) A = new int[size]

12.2.3 length

12.3 Operations

12.3.1 display: printf ("%d", A[i]) in for loop

12.3.2 add/append

1. Add new element at **END** of the array

2. A[Length] = x; length++;

12.3.3 insert

1. shifted forward to allow space

2. start from last, copy prev last and **STOP** until reach insertion point

3. pseudocode

```
for (i = length; i > index ; i--) {
  A[i] = A[i-1];
}

A[index] = x;
length++;
```

12.3.4 delete

1. delete(index)

2. x = A[index]

3. shift to occupy blank space

4. pseudocode

```
for (i = index; i < Length-1 ; i++) {
  A[i] = A[i+1];
 }
Length--;
```

5. Min time: 2 constant, Max time: n+2

12.3.5 Linear search

1. assume unique
2. Use a key

12.4 Demo: ‘`arrayADT.c`‘