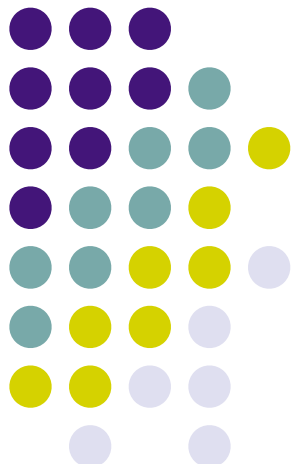
Arrays: Part 2 of 2

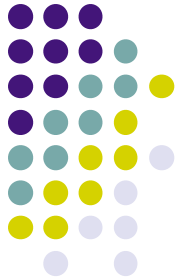
CMSC 104, Spring 2014

Christopher S. Marron

(thanks to John Park for slides)



Arrays, Part 2 of 2



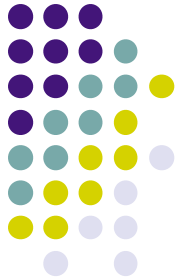
Topics

- Array Names Hold Address
- How Indexing Works
- Call by Value
- Call by Reference
- Grades Program Revised

Reading

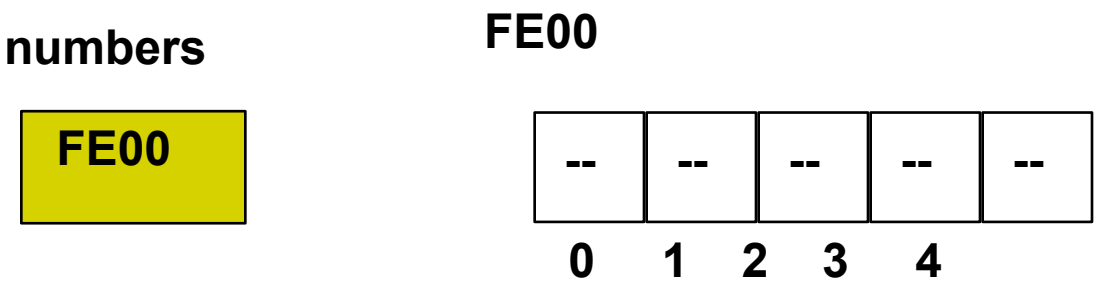
- Section 5.8
- Sections 6.1 - 6.5

Array Declarations Revisited

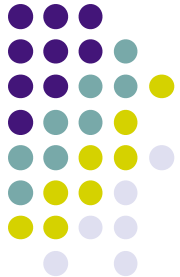


```
int numbers[5] ;
```

- This declaration sets aside a chunk of memory that is big enough to hold 5 integers.
- Besides the space needed for the array, there is also a variable allocated that has the name of the array. This variable holds the address of the beginning (address of the first element) of the array.



Array Name Holds an Address



```
#include <stdio.h>
int main( )
{
    int numbers[5] = {97, 68, 55, 73, 84} ;
    printf (“numbers[0] = %d\n”, numbers[0]) ;

    printf (“numbers = %X\n”, numbers) ;
    printf (“&numbers[0] = %X\n”, &numbers[0]) ;
    return 0 ;
}
```

output

```
numbers[0] = 97
numbers = FE00
&numbers[0] = FE00
```

How Indexing Works

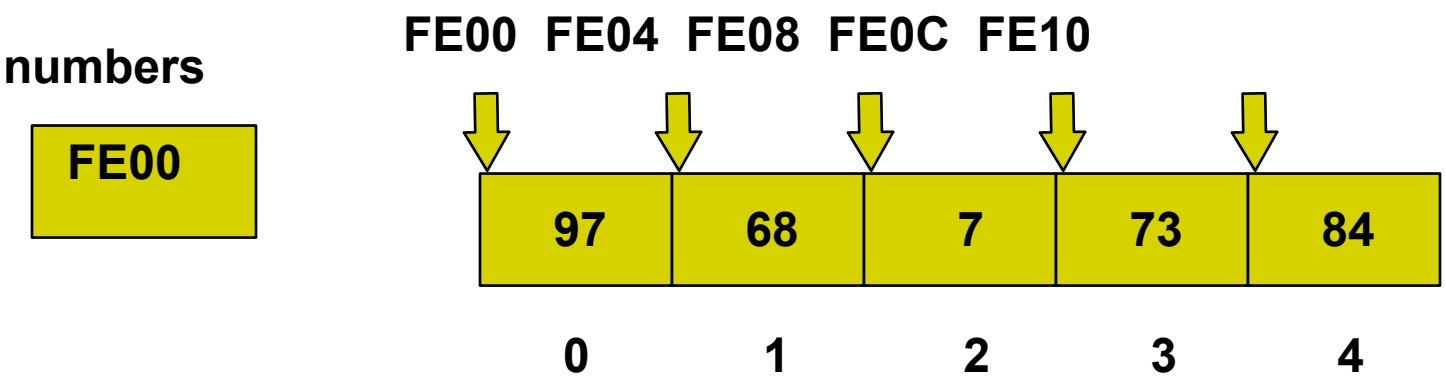
numbers[2] = 7 ;

- The element assigned the value 7 is stored in a memory location that is calculated using the following formula:

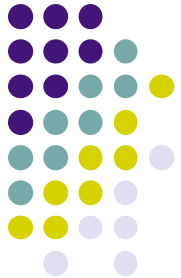
Location = (beginning address) +
(index * sizeof(array type))

Assuming a 4-byte integer,

Location = FE00 + (2 * 4)



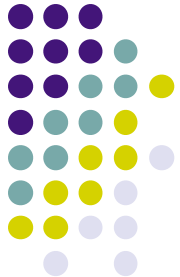
Indexing Arrays



- As long as we know
  - the beginning location of an array,
  - the data type being held in the array, and
  - the size of the array (so that we don't go out of range),

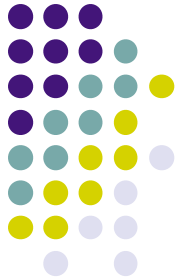
  then we can access or modify any of its elements using indexing.
- The array name alone (without [ ]) is just a variable that contains the starting address of the block of memory where the array is held.

Call (Pass) by Value



- So far, we have passed only values to functions.
- The function has a local variable (a formal parameter) to hold its own copy of the value passed in.
- When we make changes to this copy, the original (the corresponding actual parameter) remains unchanged.
- This is known as **calling (passing) by value**.

Passing Arrays to Functions



- The function prototype:

  void FillArray (int nums[ ], int numElements);

- The function definition header:

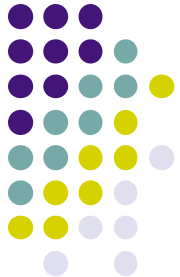
  void FillArray (int nums[ ], int numElements)

- The function call:

  FillArray (ages, SIZE);

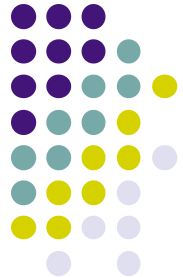
- Notice that we are passing only the name of the array (the address) and that we aren't returning anything (the function is void) because we will be modifying the original array from within the function.

Call (Pass) by Reference



- As demonstrated with arrays, we can pass addresses to functions. This is known as **calling (passing) by reference**.
- When the function is passed an address, it can make changes to the original (the corresponding actual parameter). There is no copy made**.**
- This is great for arrays, because arrays are usually very large. We really don't want to make a copy of an array. It would use too much memory.

Passing an Array to a Function



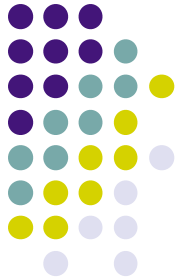
```
#include <stdio.h>
#define SIZE 4
void  FillArray (int intArray[ ], int size) ;
int main ( )
{
    int someArray [SIZE] ;
    FillArray (someArray, SIZE) ;

    /* Print the elements of the array */
    for ( i = 0; i < SIZE; i++ )
    {
        printf (someArray[%d] = %d\n",
                i, someArray[ i ] ) ;
    }
    return 0 ;
}
```

output

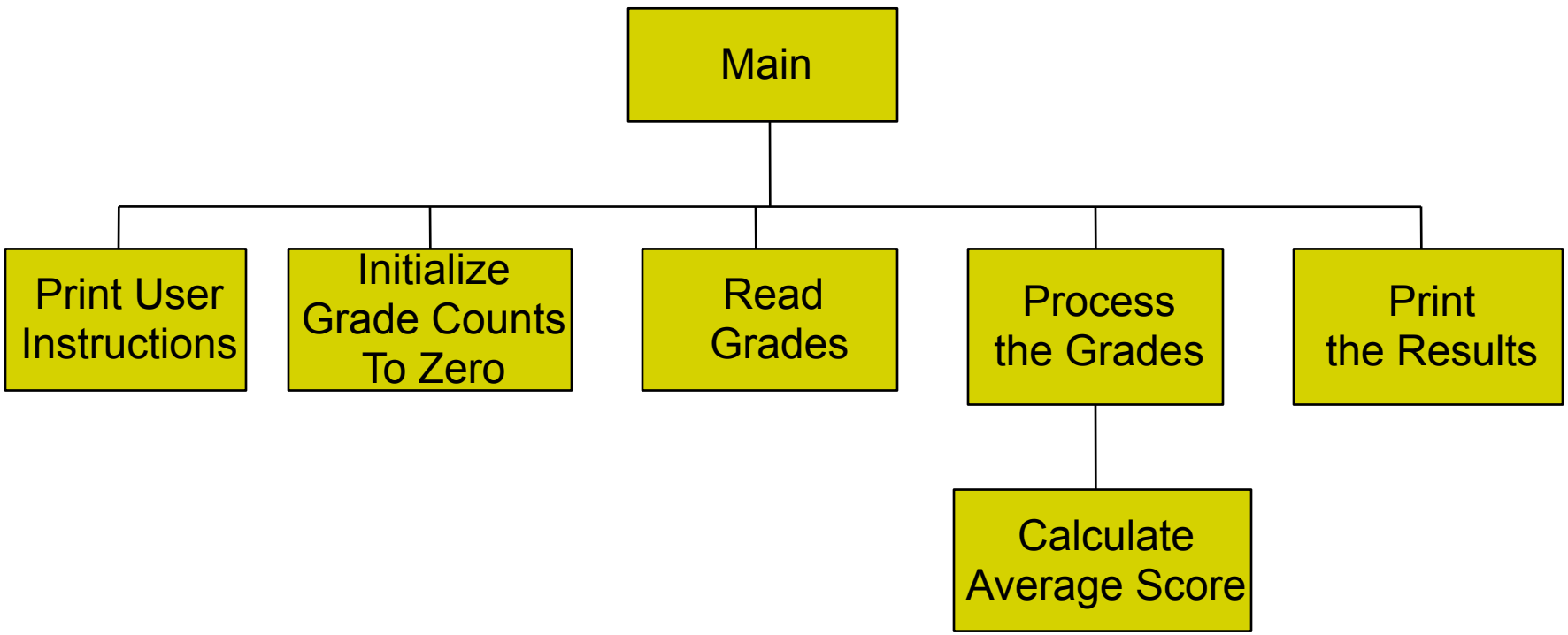
```
someArray[0] = 0
someArray[1] = 1
someArray[2] = 2
someArray[3] = 3
```

```
/*****************************************
FillArray is a function that will fill each
element of any integer array passed to
it with a value that is the same as that
element's subscript.
*****************************************/

void  FillArray (int anArray[ ],
                 int numElements)
{
    int i ;
    for ( i = 0; i < numElements; i++ )

    {
        anArray [i] = i ;
    }
}
```

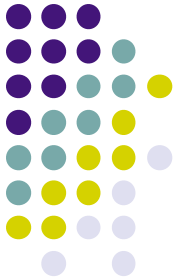
Grades Program Using Pass by Reference



- Problem: Find the average test score and the number of A's, B's, C's, D's, and F's for a particular class.
- New Design:

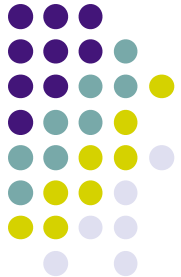


“Clean” Grades Program (con’t)



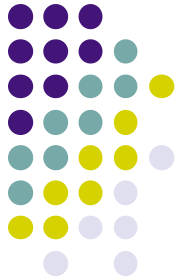
```
#include <stdio.h>
#define  SIZE        39
#define  GRADES       5
#define  A            4
#define  B            3
#define  C            2
#define  D            1
#define  F            0
#define  MAX        100
#define  MIN          0
void     PrintInstructions ( ) ;
void     InitArray (int anArray[ ], int size) ;
void     FillArray (int anArray[ ], int size) ;
double ProcessGrades (int score[ ], int size, int gradeCount[ ] ) ;
double FindAverage (double sum, int num) ;
void     PrintResults (double average, int gradeCount[ ] ) ;
```

“Clean” Grades Program (con’t)



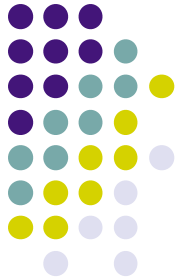
```
int main ( )
{
   int score [SIZE];                  /* student scores                    */
   int gradeCount [GRADES] ;  /* count of A’s, B’s, C’s, D’s, F’s */
   double average;                  /* average score                    */

   PrintInstructions ( ) ;
   InitArray (gradeCount, GRADES) ;
   FillArray (score, SIZE) ;
   average = ProcessGrades (score, SIZE, gradeCount ) ;
   PrintResults (average, gradeCount) ;

   return 0 ;
}
```

“Clean” Grades Program (con’t)



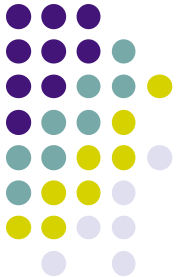
```
/*****************************************************************
** PrintInstructions - prints the user instructions
** Inputs:  None
** Outputs:  None
*****************************************************************/
void PrintInstructions ( )
{
    printf (“This program calculates the average score\n”) ;
    printf (“for a class of 39 students.  It also reports the\n”) ;
    printf (“number of A’s, B’s, C’s, D’s, and F’s.  You will\n”) ;
    printf (“be asked to enter the individual scores.\n”) ;
}
```

“Clean” Grades Program (con’t)



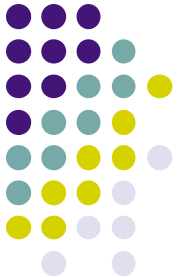
```
/*****************************************************************
/* InitArray - initializes an integer array to all zeros
/* Inputs:  anArray - array to be initialized
/*              size - size of the array
/* Outputs:  None
/*****************************************************************/
void InitArray (int anArray [ ], int size)
{
    for ( i = 0; i < size; i++ )
    {
        anArray [ i ] = 0 ;
    }
}
```

“Clean” Grades Program (con’t)



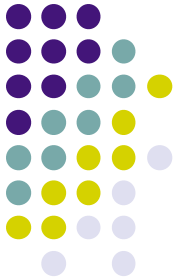
```
/*****************************************************************************
** FillArray - fills an integer array with valid values that are entered by the user.
**             Assures the values are between MIN and MAX.
** Inputs:  anArray - array to fill
** Outputs:  size - size of the array
** Side Effect - MIN and MAX must be #defined in this file
*****************************************************************************/
void FillArray (int anArray [ ], int size)
{
    int i ;   /* loop counter */
    for ( i = 0; i < size; i++ ) {
        printf (“Enter next value : ”) ;
        scanf (“%d “, &anArray [ i ] ) ;
        while ( (anArray [ i ] < MIN)  ||  (anArray [ i ] > MAX) ) {
            printf (“Values must be between %d and %d\n ”, MIN, MAX) ;
            printf (“Enter next value : ”) ;
            scanf (“%d “, &anArray[ i ] ) ;
        }
    }
}
```

“Clean” Grades Program (con’t)



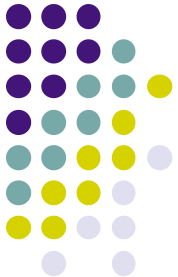
```
/*****************************************************************************
** ProcessGrades - counts the number of A's, B's, C's, D's, and F's, and
**                 computes the average score
** Inputs:  score - array of student scores
            size - size of the array
            gradeCount - grade counts all initialized to zero
** Outputs:  gradeCount - number of A's, B's, C's, D's, and F's
** Side Effect:  A, B, C, D, and F must be #defined in this file
*****************************************************************************/
double ProcessGrades (int score [ ], int size, int gradeCount [ ] )
{
   int total = 0;          /* total of all scores */
   double average;   /* average score      */
   for ( i = 0 ; i < size ; i++) {
      total += score [ i ] ;
      switch ( score [ i ] / 10 )
      {
         case 10 :
         case   9 :  gradeCount [A]++  ;
                     break ;
```

“Clean” Grades Program (con’t)



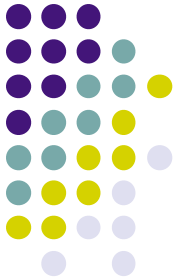
```
            case   8 :  gradeCount [B]++ ;
                        break ;
            case   7 :  gradeCount [C]++ ;
                        break ;
             case   6 :  gradeCount [D]++ ;
                        break ;
             case   5 :
             case   4 :
             case   3 :
             case   2 :
             case   1 :
             case   0 :gradeCount [F]++ ;
                       break ;
             default  : printf (“Error in score.\n”) ;
        }
    }
    average = findAverage (total, size) ;
    return average ;
}
```

“Clean” Grades Program (con’t)



```
/*************************************************************
** FindAverage - calculates an average
** Inputs:  sum - the sum of all values
**              num - the number of values
** Outputs:  the computed average
*************************************************************/
double FindAverage (double sum, int num)
{
   double average ;   /* computed average */

    if ( num != 0 ) {
        average = sum / num ;
    }
    else {
        average = 0 ;
   }

   return average ;
}
```

“Clean” Grades Program (con’t)



```
/*****************************************************************************
** PrintResults - prints the class average and the grade distribution for
**                the class.
** Inputs:  average - class average
**          gradeCount - number of A’s, B’s, C’s, D’s, and F’s
** Outputs:  None
** Side Effect:   A, B, C, D, and F must be #defined in this file
*****************************************************************************/
void  PrintResults (double average, int gradeCount [ ] )
{
   printf (“The class average is %.2f\n”, average ) ;
   printf (“There were %2d As\n”, gradeCount [A] ) ;
   printf (“                %2d Bs\n”, gradeCount [B] ) ;
   printf (“                %2d Cs\n”, gradeCount [C] ) ;
   printf (“                %2d Ds\n”, gradeCount [D] ) ;
   printf (“                %2d Fs\n”, gradeCount [F] ) ;
}
```