Lab 2

Arrays and Functions

CPSC 275
Introduction to Computer Systems

One-Dimensional Arrays

- An ***array*** is a data structure containing a number of data values, all of which have the same type.
- These values, known as ***elements,*** can be individually selected by their position within the array.
- The elements of a one-dimensional array `a` are conceptually arranged one after another in a single row (or column):

One-Dimensional Arrays

- To declare an array, we must specify the *type* of the array's elements and the *number* of elements:

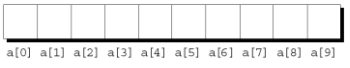
```
int a[10];
```

- The elements may be of any type; the length of the array can be any (integer) constant expression.
- Using a macro to define the length of an array is an excellent practice:

```
#define N 10
…
int a[N];
```


Array Subscripting

- To access an array element, write the array name followed by an integer value in square brackets.
- This is referred to as ***subscripting*** or ***indexing*** the array.
- The elements of an array of length *n* are indexed from 0 to *n* – 1.
- If `a` is an array of length 10, its elements are designated by `a[0]`, `a[1]`, …, `a[9]`:



Array Subscripting

- Expressions of the form `a[i]` are lvalues, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```


Array Subscripting

- Examples of typical operations on an array `a` of length `N`:

```
for (i = 0; i < N; i++)
  a[i] = 0;             /* clears a */

for (i = 0; i < N; i++)
  scanf("%d", &a[i]);   /* reads data into a */

for (i = 0; i < N; i++)
  sum += a[i];          /* sums the elements of a */
```

Array Subscripting

- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined.
- A common mistake: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*:

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```


Array Subscripting

- An array subscript may be any integer expression:

```
a[i+j*10] = 0;
```

- The expression can even have side effects:

```
i = 0;
while (i < N)
  a[i++] = 0;
```


Array Initialization

- An array, like any other variable, can be given an initial value at the time it's declared.
- The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```


Array Initialization

- If the initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

- Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

- It's illegal for an initializer to be longer than the array it initializes.

Array Initialization

- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- The compiler uses the length of the initializer to determine how long the array is.

Using the `sizeof` Operator with Arrays

- The `sizeof` operator can determine the size of an array (in bytes).
- If `a` is an array of 10 integers, then `sizeof(a)` is typically 40 (assuming that each integer requires four bytes).
- We can also use `sizeof` to measure the size of an array element, such as `a[0]`.
- Dividing the array size by the element size gives the length of the array:

```
sizeof(a) / sizeof(a[0])
```

Program: Dot Products

Write a C program (**dotprod.c**) which will compute the dot product of two *N*-vectors, *x* and *y,* that is,

$$x \bullet y = x_0 y_0 + \cdots + x_{N-1} y_{N-1}$$

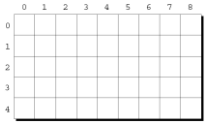
Test your program using $x_i = y_i = 1.0$, $i = 0, \ldots N-1$, for different values of *N*. Measure the running time of your program when *N* is large, using Linux **time** command.

Multidimensional Arrays

- An array may have any number of dimensions.
- The following declaration creates a two-dimensional array (a *matrix,* in mathematical terminology):

```
int m[5][9];
```

- `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:

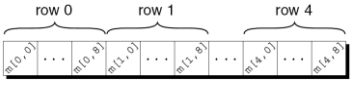


Multidimensional Arrays

- To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`.
- The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.

Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



Multidimensional Arrays

- Nested `for` loops are ideal for processing multidimensional arrays.
- Consider the problem of initializing an array for use as an *identity matrix*. A pair of nested `for` loops is perfect:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```


Initializing a Multidimensional Array

- We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- Initializers for higher-dimensional arrays are constructed in a similar fashion.

Program: Matrix-Vector Multiplication

Write a C program (**matvecmul.c**) which will compute the product of an *N*-by-*N* matrix *A* by an *N*-vector *x*. Test your program using

$a_{ij} = i+1$, for $i, j = 0, \ldots N-1$
$x_j = 1.0$ for $j = 0, \ldots N-1$

for different values of *N*. Measure the running time of your program when *N* is large, using Linux **time** command.

Functions

Function Definitions

- General form of a ***function definition:***

return-type function-name (*parameters*)
{
  declarations
  statements
}

- Functions may not return arrays.
- Specifying that the return type is `void` indicates that the function doesn't return a value.

Function Definitions

```
#include <stdio.h>

double average(double a, double b) /* function definition */
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```


Function Declarations

- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

return-type function-name (*parameters*) ;

- The declaration of a function must be consistent with the function's definition.
- Here's the `average.c` program with a declaration of `average` added.

Function Declarations

```
#include <stdio.h>

double average(double a, double b);   /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)    /* DEFINITION */
{
  return (a + b) / 2;
}
```

Function Declarations

- Function declarations of the kind we're discussing are known as ***function prototypes.***
- A function prototype doesn't have to specify the names of the function's parameters, as long as their types are present:

```
double average(double, double);
```


Arguments

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument.

Array Arguments

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified:

```
int f(int a[])  /* no length specified */
{
  …
}
```

- C doesn't provide any easy way for a function to determine the length of an array passed to it.
- Instead, we'll have to supply the length—if the function needs it—as an additional argument.

Array Arguments

- Example:

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

Array Arguments

- The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

- As usual, we can omit the parameter names if we wish:

```
int sum_array(int [], int);
```


Array Arguments

- When `sum_array` is called, the first argument will be the name of an array, and the second will be its length:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

Array Arguments, cont'd

- Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);    /*** WRONG ***/
total = sum_array(b, LEN);    /*** CORRECT ***/
```

But what are we really passing?

The address of the first element of the array.

```
total = sum_array(&b[0], LEN);
```

31

Passing Multidimensional Arrays

```
#define LEN 100

int main(void)
{
  int b[LEN][LEN];
  …
  setToZero(b, LEN);
  …
}

void setToZero(int b[LEN][LEN], int n)    OR
 { … }

void setToZero(int b[][LEN], int n)
 { … }
```

32

The `return` Statement

- A non-`void` function must use the `return` statement to specify what value it will return.
- The `return` statement has the form

```
return expression ;
```

- The expression is often just a constant or variable:

```
return 0;
return status;
```

- More complex expressions are possible:

```
return n >= 0 ? n : 0;
```

33

dotprod1.c

Rewrite **dotprod.c** which will compute the dot product of two *N*-vectors using a function. Your main() function should define the two vectors, *x* and *y*, and call a function which will accept two vectors and their length and return their dot product.

matvecmul1.c

Rewrite **matvecmul.c** which will compute the product of an *N*-by-*N* matrix *A* by an *N*-vector *x* using a function, that is, the function should place the product in a vector *b*, where $b = Ax$.

Your main() function should define *A, b,* and *x* and call a function which will accept a matrix A, two vectors, *x* and *y*, and their length and place the product in *b*.