ECE15: Introduction to Computer Programming Using the C Language

Lecture Unit 5: Arrays
Outline of This Lecture

❖ Introduction to Arrays
  ▶ Array Initialization and Referencing
  ▶ Computing Average Value using Arrays

❖ Example: Computing the Median

❖ Example: Sieve of Eratosthenes
Your Task: There are 800 salaried employees in a company. You need to read the 800 salaries from the input, compute the average salary, and determine how many employees are paid above average.

Observation: You cannot possibly compute all this while reading the salaries from the input. The 800 salaries must be stored somewhere.

Maybe you can do this?

```c
double salary1, salary2, ..., salary800;
```

❌ But these variables cannot be processed in a loop!

Solution: Arrays!
What's an array? An array is an ordered list of variables of the same type, that can be referenced by subscripting a single identifier with a variable index, for example like this:

Declaring an array:

```c
double salaries[800];
```

Referencing an array:

```c
salaries[0], salaries[i], salaries[2*i+j], salaries[exp]
```

Notes on Arrays in C:

- The array elements are always stored in consecutive memory cells.
- The index of the first element in an array is always 0.
- The number of elements in an array must be a constant. For example, you cannot do this:

```c
int array_size; scanf("%d", &array_size);
int my_array[array_size];
```
Arrays can be initialized **in a loop**, like this:

```c
for (i = 0; i < 800; i++)
    scanf("%lf", &salaries[i]);
```

```c
for (i = 0; i < 800; i++)
    salaries[i] = 3.5 * i;
```

Arrays can be also initialized **at declaration time**, like this:

```c
int grades[5] = {100, 97, 79, 0, 0};
int grades[5] = {100, 97, 79};
```

Compiler can determine array size from declaration-time initialization:

```c
int grades[5] = {100, 97, 79, 0, 0};
int grades[] = {100, 97, 79, 0, 0};
```
Example: Working with Salaries

```c
#include<stdio.h>
define NUM_EMPLOYEES 800

int main()
{
    double salaries[NUM_EMPLOYEES], sum = 0.0, average;
    int i, number_above_average = 0;

    for (i = 0; i < NUM_EMPLOYEES; i++)
    {
        scanf("%lf", &salaries[i]);
        sum += salaries[i];
    }
    average = sum/NUM_EMPLOYEES;

    for (i = 0; i < NUM_EMPLOYEES; ++i)
    {
        number_above_average += (salaries[i] > average);
    }

    printf("The average is: %f\n", average);
    printf("There are %d salaries above the average\n", number_above_average);
    return 0;
}
```
Introduction to Arrays

- Array Initialization and Referencing
- Computing Average Value using Arrays

Example: Computing the Median

Example: Sieve of Eratosthenes
What Is the Median?

Given a set of data, the median is the number that **cuts this set in half**: half the data are above the median, and half are below it.

**Example:**

Given the test scores: 85, 90, 86, 92, 86, 87, 88, 0, 0

What is the median?

**Definition:** Let $X$ be a set of $n$ numbers. Arrange the elements of $X$ in increasing order, like this:

$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_{n-1} \leq x_n$$

If $n$ is odd, then the **median of $X$** is the middle element of the ordered sequence, namely $x_{(n+1)/2}$. If $n$ is even, then the **median of $X$** is the average of the two middle elements, namely $(x_{n/2} + x_{n/2+1}) / 2$.

**Observation:** In statistics, median is often more useful than average. For example, if the test scores are 85, 90, 86, 92, 86, 87, 88, 0, 0, then:

- **×** average $= 68.22$
- **✓** median $= 86$
Computing the Median

**Method A:** Sort the $n$ numbers in increasing order, then take the middle one (or average of two) in the ordered sequence. This is inefficient! Sorting $n$ numbers requires about $n \log_2 n$ operations.

**Method B:** Suppose that the numbers in the data set can take at most $m$ possible values. Make a frequency histogram of the values actually encountered in the data set ($n$ operations). Then sum-up bars in the histogram, from left to right, until the sum exceeds $n/2$ (at most $m$ operations).
Example: The input consists of $n = 85$ exam grades, which are integers in the range 0,1,...,100. How many times does each grade occur?

```c
#include <stdio.h>
#define GRADES_NUM 85
#define MAX_GRADE 100
int main()
{
   int histogram[MAX_GRADE + 1] = {0};
   int i, current_grade;

   for (i = 0; i < GRADES_NUM; i++)
   {
      scanf("%d", &current_grade);
      histogram[current_grade]++;
   }

   ...
   return 0;
}
The Median Grade Program

```c
#include<stdio.h>
#define GRADES_NUM  85  //this is assumed to be odd
#define MAX_GRADE  100

int main()
{
   int histogram[MAX_GRADE + 1] = {0};
   int i, current_grade, bar_sum;

   for (i = 0; i < GRADES_NUM; i++)
   {
      scanf("%d", &current_grade);
      histogram[current_grade]++;
   }

   bar_sum = 0; current_grade = 0;
   while (bar_sum <= GRADES_NUM/2)
   {
      bar_sum += histogram[current_grade++];
   }

   printf("The median is: %d\n", current_grade-1);
   return 0;
}
```
Outline of This Lecture

❖ Introduction to Arrays
  ‣ Array Initialization and Referencing
  ‣ Computing Average Value using Arrays

❖ Example: Computing the Median

❖ Example: Sieve of Eratosthenes
Computing All the Primes up to $N$

**Definition of the Problem:** Given a large positive integer $N$, compute all the primes in the range 2,3,...,$N$.

**Possible Solution:** Test all integers in the range 2,3,...,$N$ for primality. This is extremely inefficient even with the best primality testing methods.

**The Eratosthenes Sieve:** First, assume that all the integers in the range 2,3,...,$N$ are prime. Then proceed as follows:

- 2 is prime, mark as not prime all the multiples of 2
- 3 is prime, mark as not prime all the multiples of 3
- 4 was already marked as not prime, just skip it
- 5 is prime, mark as not prime all the multiples of 5
- 6 was already marked as not prime, just skip it

and so on ...

After at most square root of $N$ steps like this, all the numbers in the range 2,3,...,$N$ that remain unmarked must be prime.
**Example:** The following animation demonstrates the sieve of Eratosthenes iteratively computing all the primes up to $N = 120$.

We will next implement this in C, using the array `sieve[N+1]` to mark the numbers as prime ($sieve[i] = 1$) or not prime ($sieve[i] = 0$).
#include <stdio.h>
#include <math.h>

#define N 120

int main()
{
    int i, p, sqrt_N; char sieve[N+1];

    sqrt_N = (int) sqrt(N);
    for (i = 1; i < N+1; i++) sieve[i] = 1;

    for (p = 2; p <= sqrt_N; p++)
        if (sieve[p])
            for (i = 2; i <= N/p; i++) sieve[i*p] = 0;

    printf("The prime numbers between 2 and %d are:\n", N);
    for (i = 2; i <= N; i++) if (sieve[i]) printf("%d\n", i);

    return 0;
}
Comparison with Primality Testing

We wrote a short C program that compares the **Sieve of Eratosthenes** with testing every integer in the range for primality using the **enhanced primality testing** algorithm developed earlier.

**Parameters:**

\[ N = 1,000,000 \]

Repeated 100 times in a loop.

**Results:**

- **Eratosthenes Sieve:** 1.5 sec
- **Primality testing:** 16.0 sec

The Sieve of Eratosthenes is at least ten times more efficient!

```c
#include <math.h>
...
  int is_prime = 1, n, i, sqrt_n;
  ...
  if (n == 1 || (n != 2 && n%2 == 0))
      is_prime = 0;
  else
  {
      sqrt_n = (int) sqrt(n);
      for (i = 3; i <= sqrt_n; i += 2)
          if (n%i == 0) {is_prime = 0; break;}
  }
  if (is_prime)
      printf("%d is a prime\n", n);
  else
      printf("%d is not a prime\n", n);
...```