ECE15: Introduction to Computer Programming Using the C Language

Lecture Unit 9: Two-Dimensional Arrays, Strings, Dynamic Allocation
Outline

❖ Two-dimensional arrays
  ▸ Declaration, initialization, and subscripting
  ▸ Example: Transposing a matrix
  ▸ Example: Blurring an image

❖ Dynamic memory allocation

❖ Strings and string functions

❖ Command-line arguments
What is a two-dimensional array? A two-dimensional array is an array each of whose elements is itself an array. Such arrays are used in C to represent matrices and similar structures.

Declaring a two-dimensional array:

- `double matrix[8][16];`
- `int screen[768][1024];`
- `char a[M][N];`

Referencing a two-dimensional array:

- `matrix[i][j], matrix[7][15], *(&matrix[0][0] + 16*i + j)`

Recall that:

- `salaries[i]` is equivalent to `*(salaries+i)`
- `matrix[i][j]` is equivalent to `*(matrix[i]+j)`
- `matrix[i]` is a one-dimensional array

Elements stored consecutively row-by-row

Last element

Lecture 9 ECE15: Introduction to Computer Programming Using the C Language
Initializing a two-dimensional array:

```
int M[2][3] = {{1,0,2},{3,9,7}};
```

Equivalent:

```
int M[2][3] = {1,0,2,3,9,7};
```

```
int M[][3] = {1,0,2,3,9,7};
```

What about more than two dimensions?

```
int RubicCube[3][3][3];
```

```
double SpaceTime[X][Y][Z][Time];
```

```
char My5Darray[5][5][5][5][5];
```

There is **no limit** on array dimensionality in C, but seldom used.
Outline

❖ Two-dimensional arrays
  ▶ Declaration, initialization, and subscripting
  ▶ **Example**: Transposing a matrix
  ▶ **Example**: Blurring an image

❖ Dynamic memory allocation

❖ Strings and string functions

❖ Command-line arguments
Example: Transposing a Matrix

```
#include <stdio.h>
#define SIZE 3
int main()
{
  int M[SIZE][SIZE] = {{1,3,8},{0,9,4},{2,7,6}};
  int i,j, tmp;

  printf("Matrix:\n");
  for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
      printf("%d%s",M[i][j],(j == SIZE-1)?"\n":"");

  for (i=1; i<SIZE; i++)
    for (j=0; j<i; j++)
    {
      tmp = M[i][j]; M[i][j] = M[j][i]; M[j][i] = tmp;
    }

  printf("Transpose:\n");
  for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
      printf("%d%s",M[i][j],(j == SIZE-1)?"\n":"");

return 0;
}
```

Operation:

\[ M[i,j] \leftrightarrow M[j,i] \]
#include <stdio.h>
#define SIZE 3

void transpose(int A[SIZE][SIZE]);
void print_matrix(int [][SIZE]);

int main()
{
    int M[][SIZE] = {1,2,3,4,5,6,7,8,9};
    printf("Matrix:\n");
    print_matrix(M);
    transpose(M);
    printf("Transpose:\n");
    print_matrix(M);
    return 0;
}

void transpose(int A[][SIZE])
{
    int i,j, tmp;
    for (i=1; i<SIZE; i++)
    for (j=0; j<i; j++)
    {
        tmp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = tmp;
    }
}

void print_matrix(int A[][SIZE])
{
    int i,j;
    for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
        printf(" %d%s", A[i][j], (j == SIZE-1) ? "\n":"");
}
#include <stdio.h>
#define SIZE 3
void swap(int *, int *);
void transpose(int A[SIZE][SIZE]);
void print_matrix(int [] [SIZE]);

int main()
{
  int M[SIZE][SIZE] = {1,2,3,4,5,6,7,8,9};
  printf("Matrix:\n");
  print_matrix(M);
  transpose(M);
  printf("Transpose:\n");
  print_matrix(M);
  return 0;
}

transpose3.c

void transpose(int A[][SIZE])
{
  int i,j;
  for (i=1; i<SIZE; i++)
    for (j=0; j<i; j++)
      swap(&A[i][j],&A[j][i])
}

void swap(int* px, int* py)
{
  int temp = *px;
  *px = *py;
  *py = temp;
}

Alternatives:

// Using pointer arithmetic
swap(A[i]+j, A[j]+i);
swap((int*) (A+i)+j, (int*) (A+j)+i);
swap((int*) A+SIZE*i+j, (int*) A+SIZE*j+i);
Two-dimensional arrays
- Declaration, initialization, and subscripting
- Example: Transposing a matrix
- Example: Blurring an image

Dynamic memory allocation

Strings and string functions

Command-line arguments
How to Blur a Cow?

What's the connection to two-dimensional arrays?

❖ Images are usually represented as two-dimensional arrays
❖ Each pixel in the image is represented by an array element
❖ The value of the pixel indicates its color: 0 for black to 255 for white

How to blur an image?

*Replace each pixel by the average of its surrounding pixels!*
Blurring: An Example

This pixel value will be replaced by the average of 9 surrounding pixels (here the average is 112)

At the edges, average over fewer neighbors
A Sequence of Blurs

Here is an example of what happens when blurring is applied iteratively:

That's how the cow got out of focus...
Implementation in C

```c
#include <stdio.h>

#define M 10  // # rows
#define N 10  // # columns
#define FALSE 0
#define TRUE 1

int inbounds(int i, int j);
int mean_around_pixel(int picture[][N], int i, int j);
void copy2Darray(int from[][N], int to[][N]);
void blur_image(int picture[][N]);

int main()
{
    int picture[M][N] = {...};
    ...
    blur_image(picture);
    ...
}
```
Implementation of Functions

```c
int inbounds(int i, int j)
{
    if (i < 0 || i >= M || j < 0 || j >= N)
        return FALSE;
    else return TRUE;
}

int mean_around_pixel(int picture[][N], int i, int j)
{
    int di, dj, sum = 0, neighbors = 0;
    for (di = -1; di <= 1; di++)
        for (dj = -1; dj <= 1; dj++)
            if (inbounds(i+di, j+dj))
            {
                sum += picture[i+di][j+dj];
                neighbors++;
            }
    return (int)((double)sum/neighbors + 0.5);
}
```
void copy2Darray(int from[][N], int to[][N])
{
    int i,j;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            to[i][j] = from[i][j];
}

void blur_image(int picture[][N])
{
    int i,j;
    int blurred[M][N];
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            blurred[i][j] = mean_around_pixel(picture,i,j);
    copy2Darray(blurred,picture);
}
Dynamic memory allocation

Strings and string functions

Command-line arguments

Two-dimensional arrays
- Declaration, initialization, and subscripting
- Example: Transposing a matrix
- Example: Blurring an image
**Why Dynamic Memory Allocation?**

Recall differences between arrays and pointers:

- `double salaries[800];`  
  Memory allocated for **800 variables** of type `double`
- `double *p;`  
  Memory allocated for **one variable** of type `double`

**Therefore:**

- `*salaries = 3.5;`  
  `x = salaries[42];`  
  ✔️
- `*p = 3.5;`  
  `x = p[42];`  
  ✗

**But arrays are often inadequate!**

- Many C programs require memory whose **size** is **not known** until run-time.
- C99 standard allows to declare `salaries[N]` where `N` is a variable, but C89 does not. *Don't rely on this!*
- What if we need a block of memory which is **not** an array? For example, mixes `double` and `int`?

Allocate memory during run-time: **Dynamic memory allocation!**
The malloc and calloc Functions

The `malloc` and `calloc` standard library functions, declared in `<stdlib.h>`, make it possible for a C program to request additional memory from the operating system at run-time.

```c
void *malloc(int size);
void *calloc(int n_objects, int object_size);
```

❖ The amount of memory requested (always in bytes) is passed as a parameter to both `malloc` and `calloc`.

❖ Both `malloc` and `calloc` return a pointer to a contiguous block of memory of the requested size. The pointer is of type `void *`.

❖ **What if there is not enough memory?** If the allocation fails, then both `malloc` and `calloc` will return `NULL`.

**Example:**

```c
double *p; int n;
   // ...read n from the user
p = (double *) malloc(n * sizeof(double));
if (p == NULL) return 1;
```
The malloc and calloc Functions

What is the difference between malloc and calloc?

void *malloc(int size);

void *calloc(int n_objects, int object_size);

❖ malloc returns the allocated memory uninitialized, it contains garbage values. calloc returns the memory initialized to zeros.

Example:

Allocate an array of \( n \) integers, initialized to zeros.

```c
int *p; int n,i;
   // ...read n from the user
p = (int *) malloc(n*sizeof(int));
if (p == NULL) return 1;
for (i = 0; i < n; i++)
  p[i] = 0;
```

Not much!

```c
int *p; int n;
   // ...read n from the user
p = (int *) calloc(n,sizeof(int));
if (p == NULL) 
{    printf("Allocation error!");  
   return 1;
}
```
Memory Release and Leaks

Often, memory needs to be dynamically allocated in a loop or within a function. In such cases, **beware of leaks!**

```c
int foo(int n)
{
    double *p; int foovalue;
    p = (double *) calloc(n, sizeof(double));
    if (p == NULL) exit(1);
    // work with p, compute foovalue...
    return foovalue;
}
```

It is your responsibility to release memory you allocated!

- The `free` library function, declared in `<stdlib.h>`, releases dynamically allocated memory, like this: `free(p)`. It does not need to know the size of the memory to be released (known to the OS), only a pointer to it.

- When the program terminates, **all** its memory is released by the operating system. But, do **not** try this at home: `while(1) malloc(100);`
Salaries with Dynamic Allocation

Your Task: Read the **number** of employees in a company, then read their salaries and process this data (average, median, etc).

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

int main()
{
    double *salaries; int n,i;
    scanf("%d", &n);  // read number of employees
    salaries = (double *) calloc(n, sizeof(double));
    if (salaries == NULL)
    {
        printf("Cannot allocate array of %d salaries!", n);
        return 1;
    }
    for (i = 0; i < n; i++)
    {
        scanf("%lf", salaries+i);
        // compute average, stdev, median, etc...
    }
    free(salaries);
    return 0;
}
```

salaries_alloc.c
### Allocating an Integer Matrix

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

int **alloc_matrix(int M, int N);

int main()
{
  int m,n; // m rows, n columns
  int **A;
  scanf("%d%d", &m,&n);
  A = alloc_matrix(m,n);
  // work with the matrix...
  int i;
  for (i = 0; i < m; i++)
     free(A[i]);
  free(A);
  return 0;
}
```

```c
int **alloc_matrix(int M, int N)
{
  int **p;
  int *aux;
  int i,j;
  aux = (int *) malloc(M*N*sizeof(int));
  if (aux == NULL) exit(1);
  p = (int **) malloc(M*sizeof(int *));
  if (p == NULL) exit(1);
  for (i = j = 0; i < M; i++, j += N)
     p[i] = &aux[j];
  return p;
}
```

The matrix A is an array of m arrays, each consisting of n integers.
Outline

❖ Two-dimensional arrays
  ▸ Declaration, initialization, and subscripting
  ▸ **Example**: Transposing a matrix
  ▸ **Example**: Blurring an image

❖ Dynamic memory allocation

❖ Strings and string functions

❖ Command-line arguments
What is a string? A string is simply a sequence of characters.

Examples: 

```
char class1[] = "ece15";    // array
char *class2 = "ece15";     // pointer
char *class3 = "ece" "15";  // concatenation
char *class4; class4 = "ece15"; // assignment
```

The C language does not have a special type dedicated to representing strings.

Strings are stored simply as arrays of `char`:

```
char class1[] = "ece15";    // array
char *class2 = "ece15";     // pointer
char *class3 = "ece" "15";  // concatenation
char *class4; class4 = "ece15"; // assignment

// Possible but risky:
char class5[] = {'e','c','e','1','5'};
...
```

string_def.c
Null Character as String Delimiter

- The **null** character '\0' is a special character whose ASCII code is zero. Thus it is represented in memory as 00000000.

- The character '\0' is used in C to denote the end of a string. For example, the string "ece15" is represented in memory as:

```
 e c e 1 5 \0 Y # ...
```

- This makes it possible to find where the string ends without knowing a priori how many characters it contains:

```
int stringlength(char* string)
{
  int i = 0;
  while (string[i] != '\0') i++;
  return i;
}
```

Correct declaration: `char class5[] = {'e','c','e','1','5','\0'};`
Accessing String Elements

Since a string \texttt{s[]} is just an array, individual characters in a string can be accessed using subscription \texttt{s[i]} as usual.

**Example:** Count the number of blanks in a given string.

```c
char *s = "just a string";
i = 0;
counter = 0;
while (s[i] != '\0')
  if (s[i++] == ' ')
    ++counter;
```

```c
char *s = "just a string";
i = 0;
counter = 0;
while (*(s+i))
  if (*(s + i++) == ' ')
    ++counter;
```

```
char s[] = "just a string";
counter = 0;
while (*s)
  if (*(s++) == ' ')
    ++counter;
```

```c
char empty_string[1] = {'\0'};
```
String Input and Output

Read and Write: Using `scanf` and `printf` standard library functions from `<stdio.h>` with the `%s` conversion specification

```c
#include <stdio.h>

int main()
{
    printf("My fav class: %s\n", "ece 15");

    char word1[80], word2[80];

    printf("Two words: ");
    scanf("%s %s", word1, word2);

    printf("word1: %s\n", word1);
    printf("word2: %s\n", word2);

    return 0;
}
```

---

String Input and Output

Read and Write: Using `scanf` and `printf` standard library functions from `<stdio.h>` with the `%s` conversion specification

```c
#include <stdio.h>

int main()
{
    printf("My fav class: %s\n", "ece 15");

    char word1[80], word2[80];

    printf("Two words: ");
    scanf("%s %s", word1, word2);

    printf("word1: %s\n", word1);
    printf("word2: %s\n", word2);

    return 0;
}
```
### Useful String Functions

The C standard library contains many functions for manipulating strings!

#### Functions in `<string.h>`

<table>
<thead>
<tr>
<th>Function</th>
<th>What the function does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int strlen(char *s)</code></td>
<td>Returns the length of the string</td>
</tr>
<tr>
<td><code>char *strcat(char *s1, char *s2)</code></td>
<td>Concatenates <code>s2</code> to the end of <code>s1</code></td>
</tr>
<tr>
<td><code>int strcmp(char *s1, char *s2)</code></td>
<td>Compares two strings in dictionary order</td>
</tr>
</tbody>
</table>

#### Functions in `<stdlib.h>`

<table>
<thead>
<tr>
<th>Function</th>
<th>What the function does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int atoi(char *s)</code></td>
<td>Converts the given string <code>s[]</code> to <code>int</code></td>
</tr>
<tr>
<td><code>double atof(char *s)</code></td>
<td>Converts the given string <code>s[]</code> to <code>double</code></td>
</tr>
</tbody>
</table>

#### Functions in `<stdio.h>`

<table>
<thead>
<tr>
<th>Function</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int sprintf(char *s, char *format, arg1, ...)</code></td>
<td>Writes to a string <code>s[]</code></td>
</tr>
<tr>
<td><code>int sscanf(char *s, char *format, arg1, ...)</code></td>
<td>Reads from a string <code>s[]</code></td>
</tr>
</tbody>
</table>

* Works just like `scanf` and `printf`. **Note:** `s[]` must be big enough!
Outline

❖ Two-dimensional arrays
  ▸ Declaration, initialization, and subscripting
  ▸ **Example**: Transposing a matrix
  ▸ **Example**: Blurring an image

❖ Dynamic memory allocation

❖ Strings and string functions

❖ Command-line arguments
We have seen arrays of `int`, `double`, `char`, ... **everything**

How about an array of... **pointers**? `a[0]`, `a[1]`,... pointers to integers

<table>
<thead>
<tr>
<th>array of <code>int</code></th>
<th><code>int a[];</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer to <code>int</code></td>
<td><code>int *p;</code></td>
</tr>
<tr>
<td>array of pointers to <code>int</code></td>
<td><code>int *a[];</code></td>
</tr>
</tbody>
</table>

```c
#include <stdio.h>

int main()
{
    char *stringarray[] = {
        "First string",
        "Second string",
        "Third string"};

    int i;

    for (i = 0; i < 3; i++)
        printf("%s\n", stringarray[i]);

    return 0;
}
```

Many applications

One next
Command-Line Arguments

Convenient way to run frequent commands

Task: Convert Celsius to Fahrenheit

> gcc c2f.c
> a.out
> Celsius: 100
> Fahrenheit: 212

If run frequently, skip prompt

> gcc c2f2.c
> a.out 100
> Fahrenheit: 212

Provide the program command line arguments

Almost all programs use command-line arguments

> gcc c2f.c
> gcc -o c2f.out c2f.c
> ls *.c
Command-Line Arguments

C provides the program two values

`argc` - integer containing # stings in command line (incl. filename)

`argv` - array of pointers, each pointing to one of the strings

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf("%d arguments\n", argc-1);
    for (i=1; i<argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```plaintext
> a.out Hello 1 world
argc: 4
```

What if all values integers, wanted to increment all?

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf("%d arguments\n", argc-1);
    for (i=1; i<argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```c
increment.c
```

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf("%d arguments\n", argc-1);
    for (i=1; i<argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```c
print_arguments.c
```
#include <stdio.h>

int main() {
  double cel;

  printf("Celsius: ");
  scanf("%lf", &cel);
  printf("Fahrenheit: %.1lf\n", cel*1.8+32);

  return 0;
}

#include <stdio.h>

int main(int argc, char *argv[]) {
  double cel = atoi(argv[1]);

  printf("Fahrenheit: %.1lf\n", cel*1.8+32);

  return 0;
}