Arrays and Pointers. Lecture Plan.

• **Intro into arrays.**
  definition and syntax
  declaration & initialization
  major advantages
  multidimensional arrays
  examples

• **Intro into pointers.**
  address and indirection operators
  definition of pointers
  pointers and arrays – comparison
  pointer arithmetic
Arrays and Pointers

Array is a group of elements that share a common name, and that are different from one another by their positions within the array.

C syntax:  
\[
\begin{align*}
x[1] &= 3.14; \\
x[2] &= 5.2; \\
x[3] &= 6347; \\
\end{align*}
\]

Declaration:  
\[
\begin{align*}
\text{int } x[5]; \\
\end{align*}
\]

Sets aside memory for the array

Type     Name     Size
Sets aside memory for the array
Arrays and Pointers

Initialization:

```c
int grade[]={100, 99, 85};
int grade[3]={100, 99, 85};
int grade[100]={1, 3, 5, 7};

grade[4]-grade[99] will be zeros.

grade[36] = 87;
```

Multidimensionality:

Scalar variable

Vector variable (1D)  
```c
a_0, a_1, a_2, ...
```

Matrix variable (2D)  
```c
a_{00}, a_{01}, a_{02}, ...
```
```c
a_{10}, a_{11}, a_{12}, ...
```
```c
a_{20}, a_{21}, a_{22}, ...
```
```c...
```
Arrays and Pointers

Declaration: \( \text{int } L=100, M=100, N=100; \)
\[
\text{float } a[L][M][N];
\]

Initialization: \( \alpha[2][2] = \{1,2,3,4\}; \)
\[
\alpha[2][2] = \{(1,2), (3,3)\};
\]
\[
\alpha[0][1] = 3;
\]
\[
\alpha[1][1] = 2;
\]

NB: Array size is fixed at declaration.
\[
\#define L 100
\]
\[
\#define M 100
\]
\[
\#define N 100
\]
\[
\ldots
\]
\[
\text{int } a[L][M][N]
\]
Arrays and Pointers

NB: In C numbers of array elements start form zero: x[0], x[1], x[2], x[3], x[4]. There is no x[5].

NB: If x[5] is accessed, no error will result!

Utility: simplify programming of repetitive operations
improve clarity
improve modularity
improve flexibility
Arrays and Pointers

Example: a program to compute the class average of the midterm.

Scalar form:

```c
int main(void) {
    float average;
    int sum=0,grade1,
        grade2,..;
    scanf("%d",&grade1);
    scanf("%d",&grade2);
    ...
    sum += grade1;
    sum += grade2;
    ...
    average = sum/95.0;
}
```

Vector (array) form:

```c
int main(void) {
    float average;
    int i,n,sum=0,grade[100];
    scanf("%d",&n);
    for(i=0;i<n,&n;i++){
        scanf("%d",&grade[i]);
        sum += grade[i];
    }
    ...
    average = (float)sum/n;
}
```
Arrays and Pointers

Example: Integration using Composite Trapezoid Rule

\[ I = \int_{a}^{b} f(x) \, dx \]

Continuous function \( f(x) \), \( x \) belongs to \([a,b]\) 
and a set of discrete values \( f(x_i) \), \( x_i \) belong to \([a,b]\).

\[ I = \sum_{i=1}^{N} \frac{h}{2} [f(x_{i-1}) + f(x_i)] = h \left[ \frac{f(a) + f(b)}{2} + \sum_{i=1}^{N-1} f(x_i) \right] \]
Arrays and Pointers

Given a function \( y = f(x) \) to integrate
form \( x = a \) to \( x = b \):

```c
int main(void) {
    ...
    h = (b - a) / n;
    integral = 0.5 * (func(a) + func(b));
    for (i = 1; i < n; i++)
        integral += func(a + i * h);
    integral *= h;
    ...
    return (0);
}
```
Arrays and Pointers

Given discrete data $y_i = f(x_i)$ integrate form $x=a$ to $x=b$:

```c
int main(void) {
    ...
    for (i=0; i<=n; i++)
        scanf("%f", &y[i]); /*reading $f(x_i)$*/
    integral = 0.5*(y[0]+y[n]);
    for(i=1; i<n; i++){
        scanf("%f", &y); /*summing $f(x[i])$*/
        integral += y;
    }
    scanf("%f", &a)
    scanf("%f", &b)
    integral *= (b-a)/n;
    ...
    return(0);
}
```
Arrays and Pointers

Calculating the average. Version 1. /* No arrays. */

```c
#include <stdio.h>
int main(void)
{
    float ave;
    int sum=0;
    int data1, data2, data3;
    scanf("%d", &data1);
    scanf("%d", &data2);
    scanf("%d", &data3);
    sum == data1;
    sum += data2;
    sum += data3;
    ave = sum/3.0;
    ...
}
```

- inefficient coding
- only works for a fixed number of data points
Array and Pointers

/* no arrays, scalar "for" loop */

#include <stdio.h>
int main(void)
{
    float ave;
    int i, n, datai, sum=0;
    scanf("%d", &n);
    for (i=0; i<n; i++){
        scanf("%d", &datai);
        sum += datai;
    }
    ave = (float) sum/n;
    ...
}
Arrays and Pointers

Calculating the average. Version 3. /* with arrays */
#include <stdio.h>
#include <math.h>
#define NMAX 100
int main(void)
{
    float ave;
    int i, n, data[NMAX], sum=0;
    scanf("%d", &n);
    if(n>NMAX) printf("number of pts > NMAX");
    for (i=0; i<n; i++)
        scanf("%d", &data[i]);
    sum += data[i];
}
ave = float(sum)/n;  // array size is fixed at declaration
...  // use #define to have some flexibility
}
Arrays, Summing up

• The name identifies the location in memory, big enough to store the whole array.
• a[k] refers to the k-th element of the array, the indexing starting from 0.
• The memory allocation happens when the array is declared: use # to set the dimensions.
• Advantages: clear and compact coding, better modularity, take advantage of loops for repetitive operations.
Arrays and Pointers

Intro into pointers.

- address operator, unary, right to left precedence

- variable \( v \) – location (address) of \( v \) in the memory

The special type of variable to operate with the address is needed: 

\[ \text{POINTER} \quad \text{pv} = \& v; \]

<table>
<thead>
<tr>
<th>Identifier</th>
<th>( v )</th>
<th>( \text{pv} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory address</td>
<td>1776</td>
<td>1997</td>
</tr>
<tr>
<td>Value</td>
<td>5</td>
<td>1776</td>
</tr>
</tbody>
</table>
Arrays and Pointers

Declaration: \texttt{int \*p; \ p} – pointer to integer variable.
Value range: zero or NULL address and a set of positive integers.

Assignment: \texttt{p=}0; \ p=\texttt{NULL}; \ p=\&i; \ p=(\texttt{int \*})1776;

Indirection (dereferencing) operator \texttt{*} - “inverse” to \&. Gives the value of the variable pointed to by the pointer.
\texttt{p} = \&i; \ i = \*p; \textit{We can access any variable, if we know the variable’s address!}
\texttt{\&i} = \texttt{p}; illegal, addresses are allocated by declarations.
\texttt{p} = \&3; \ p = \&(i+j); illegal: constants and expressions do not have addresses.
Arrays and Pointers

Relationship between arrays and pointers:

• Array name is a pointer constant, its value is the address of the first element of the array.

• Pointers can be subscribed

\[ a[i] = *(a + i) \quad a \text{ – address of } a[0] \]  
(base address or the array)

\[ a[i] = *(p + i) \text{ points to } i\text{-th element of the array} \]

NB: a is a constant pointer, a=p, ++a, &a are illegal.
Arrays and Pointers

Pointer arithmetic is equivalent to array indexing:

\[ p = a + 1 \quad p = \&a[1] \]
\[ p = a + m \quad p = \&a[m] \]

Summing the array using pointers:

\[
\text{for } (p = a; p < \&a[N]; ++p)
\]
\[
\quad \text{sum } += \ *p;
\]
\[
\quad \text{or}
\]
\[
\text{for } (i = 0; i < N; ++i)
\]
\[
\quad \text{sum } += \ *(a + i);
\]
Arrays and Pointers

Pointer arithmetic:

\[ p + 1 \quad ++p \quad p + i \quad p += i \]

However, pointers and numbers are not quite the same:

```c
double a[2], *p, *q;
p = a;
q = p + 1;
printf("\%d\n", q - p);  /* 1 is printed */
printf("\%d\n", (int) q - (int) p);  /* 8 is printed */
```

The difference in terms of array elements is 1, but the difference in memory locations is 8!
Arrays and Pointers

Arrays and pointers as function arguments:

“call by value”  

- Variables themselves are passed as function arguments.
- The variables are copied to be used by the function.
- Dealing directly with variables, which are not changed in calling environment.

“call by reference”

- Pointers are used in the argument list: addresses of variables are passed as arguments.
- Variables are directly accessed by the function.
- The variables may be changed inside the function and returned.
Arrays and Pointers

Passing arrays to functions:
As individual scalars: \( x = \text{sum}(\text{grade}[k], \text{grade}[k+1]) \);
prototype: \[
\text{int sum}(x, y)
\]
{
    int x, y;
    ...
}

Using pointers: \( x = \text{sum}(\text{grade}, n) \);
prototype: \[
\text{int sum}(\text{int } *\text{grade}, \text{int } n);
\]
{
    int res, *p;
    res = 0;
    for (p=grade; p<&grade[N]; ++p)
        res += *p;
    return(res);
}
Arrays and Pointers

The function swaps two variables, using “call by reference”.
void swap(int *p, int *q)
{
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}
Arrays and Pointers

Checking how “swap” works:

```c
#include <stdio.h>
void swap(int *, int *)
{
    int i = 3, j = 5;
    swap(&i, &j);
    printf("%d %d\n", i, j);
    return 0;
}
/* 5  3 is printed */
```
Arrays and Pointers

Pointer arithmetic summed up:
1. Assignment: `ptr = &a;`
2. Value finding: `*ptr = a;`
3. Taking pointer address: `&ptr – address of ptr in the memory (pointer to pointer).`
4. Addition/subtraction: `ptr2 = ptr1 +1; ptr2–ptr2;`
5. Increment: `ptr1++ ptr1 + 1`
   NB Increment does not work for pointer constants.
6. Indexing – like arrays: `ptr[i] = a[i];`
   NB Pointers and arrays are almost the same:
   
   `....[i] *(....+i)`
Arrays and Pointers

Automatic memory allocation happens when the array is declared:

```c
int data[100];
```

Dynamic memory allocation:
- function `calloc()` takes 2 unsigned integers: number of elements in the array and number of bytes in each element, returns a pointer to the base element of the array and sets all the array elements to zero:

```c
a = calloc(n, sizeof(int));
```

To clear (return) the allocated space the “free” command is used:

```c
free(a);
```
Arrays and Pointers

The other option is function malloc(): it takes one unsigned integer - required number of bytes of memory desired. Both calloc and malloc return pointer to void and the result will be casted automatically.

```c
int main(void) {
    float *a;
    int k;
    scanf("%d, &k);
    a = (float *)malloc(k*sizeof(float);
    ...
    a[0] = ...
    ...
    free(a);
}
```
Arrays and Pointers

Offsetting the pointer for the array to start form the element 1 instead of 0.

```c
int n;
double *a;
a = calloc(n+1, sizeof(double));
or
a = calloc(n, sizeof(double));
--a; /* offset the pointer */
```

a[1] is the first accessible storage element.

![Diagram showing array and pointer]
Multidimensional Arrays and Pointers

```c
int a[3][5]; /* 3 rows, 5 columns */
```

Some differences from vector arrays:
- `a` - pointer to the base address `&a[0][0]` (not to `a[0][0]`)
- `a + i` - pointer to the address of the `i`th row `&a[i][0]`

Both `a` and `a+i` are pointers to pointers.
- `*a` - row address for `a` (1st row), `**a` - value of `a[0][0]`.

We need to dereference twice to get from `a` to the values.

- `a[i]` - pointer to the `i`th row
- `a[i][j]`  `*(&a[0][0] + 5*i + j)`
Multidimensional Arrays and Pointers

Prove that each of the following four expressions is equal to \( a[i][j] \):
\[
\begin{align*}
\&\text{*(a[i] + j)} \\
\&\text{(*(a + i))[j]} \\
\&\text{(*((*(a + i)) + j) /* NOTE 2 dereferencing operations */} \\
\&\text{(*(&a[0][0] + 5*i + j)} \\
\end{align*}
\]

Some more pointer arithmetic:
\[
\begin{align*}
\&\text{*(a + 1)} & \text{address of the second row} \\
\&\text{*(a + j) + k} & \text{address of } a[j][k] \\
\&\text{(*((a + j) + k)} & \text{value of } a[j][k] \\
\&\text{(*((a + j) + k)} & a[j][k] + m
\end{align*}
\]

Storage mapping - finding the array element using a pointer:
\[
a[i][j] = \text{*(&a[0][0] + 5*i + j)}
\]

NB need the number of columns (5), not just pointer to \( a[0][0] \)!
Multidimensional Arrays and Pointers

To pass an nD array to the function we need to set (n-1) dimensions of the array outside the function. For n>1 programming becomes much less flexible: no dynamic memory allocation, etc.

It may be avoided by using arrays of pointers. Let’s build a matrix of an arbitrary size starting form pointer to pointer to double:

```c
int i, n;
double **a, det; /* NB **a declared, not an array */
...... /* getting n */
a = calloc(n, sizeof(double *)); /* a-array of pointers to double */
for (i = 0; i < n; ++i)
    a[i] = calloc(n, sizeof(double));
......
```
Multidimensional Arrays and Pointers

a – 1D pointer array

nxn matrix in computer memory

0 1 2

0 1 2

0 1 2

0 1 2

0 1 2
Multidimensional Arrays and Pointers

Now we can easily pass a to a function, say one summing diagonal elements of the matrix:

double trace(double **a, int n)
{
    int i;
    double sum = 0.0;
    for (i = 0; i < n; ++i);
        sum += a[i][j];
    return sum;
}
Pointers to Functions

What if we need to do the same calculation for several functions?

Example: \[ \sum_{k=m}^{n} f^2(k) \]

The summing routine:

```c
double sum_square(double (*f)(double), int m, int n) {
    int k;
    double sum = 0.0;
    for (k = m; k <= n; ++k)
        sum += f(k) * f(k);
    return sum;
}
```

The first argument is a pointer to function \( f \), which takes double and returns double.
Pointers to Functions

f can either be treated as a function or as a pointer with dereferencing:

\[ \text{sum } += (\*f)(k) \times (\*f)(k); \quad \text{sum } += f(k) \times f(k) \]

- \( f \) the pointer to function
- \( \*f \) the function itself
- \((\*f)(k)\) the call to the function

**Pointer to array**: points to the first memory cell containing the element of the array in the data segment of computer memory.

**Pointer to function**: points to the first memory cell containing the function in the code segment of computer memory.