CSE 230
Intermediate Programming in C and C++
Arrays, Pointers and Strings
Fall 2017
Stony Brook University
Instructor: Shebuti Rayana
http://www3.cs.stonybrook.edu/~cse230/
Pointer Arithmetic and Element Size

- If \( p \) is a pointer to a particular type, then the expression \( p + 1 \) yields the correct machine address for storing or accessing the next variable of that type.
- Valid operations: \( p + i \), ++\( p \), \( p += 2 \) etc.
- If \( p \) and \( q \) are both pointing to elements of an array, then \( p - q \) yields the \text{int} value representing the number of array elements between them.
Example: Pointer Arithmetic

```c
int i = 7,*p = &i, *r;
double a[2]={0.1,0.2},*q, *s;
r = p + 1;
q = a; //q points to a[0]
s = q + 1; // s = &a[1]
printf("%d\n",(int)r - (int)p);
printf("%d\n",(int)s - (int)q);
Printf("%d\n",s - q);
```
Example: Pointer Arithmetic

printf("%d\n", (int)r - (int)p);
4

printf("%d\n", (int)s - (int)q);
8

Printf("%d\n", s - q);
1

- The difference in terms of array elements is 1, but the difference in memory locations is 8 as size of double is 8.
Arrays as Function Arguments

- In function definition, the parameter that is declared as an array is a pointer.
- When an array is passed to a function the base address (&a[0]) is passed, not the elements of the array are copied.
- Example:

```c
double sum(double a[], int n) //n is the size of a[]
{
    int i;
    double sum = 0.0;

    for(i=0;i<n;i++)
    {
        sum += a[i];
    }
    return sum;
}
```
Arrays as Function Argument

- Following two are same:
  ```
  double sum(double a[], int n)
  double sum(double *a, int n)
  ```

- Array declaration = pointer declaration in parameter list, but not inside the function body

- From the caller: `sum(a, n);` or `sum(&a[0], n);` both are correct

- `sum(&a[7], k - 7) = a[7], a[8], ..., a[k-1]`
An Example: Bubble Sort

```c
void swap(int *, int *);

void bubblesort(int a[], int n)
{
    int i, j;

    for(i = 0; i < n-1; i++)
    {
        for(j = n-1; j>i; j--)
        {
            if(a[j-1] > a[j])
            {
                swap(&a[j-1], &a[j]);
            }
        }
    }
}
```

Bubble sort is expensive takes $O(n^2)$
## Each Pass of Bubble Sort

<table>
<thead>
<tr>
<th>Unsorted Data</th>
<th>7</th>
<th>3</th>
<th>66</th>
<th>3</th>
<th>-5</th>
<th>22</th>
<th>-77</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Pass</td>
<td>-77</td>
<td>7</td>
<td>3</td>
<td>66</td>
<td>3</td>
<td>-5</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>Second Pass</td>
<td>-77</td>
<td>-5</td>
<td>7</td>
<td>3</td>
<td>66</td>
<td>3</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>Third Pass</td>
<td>-77</td>
<td>-5</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>66</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Fourth Pass</td>
<td>-77</td>
<td>-5</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>66</td>
<td>22</td>
</tr>
<tr>
<td>Fifth Pass</td>
<td>-77</td>
<td>-5</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>22</td>
<td>66</td>
</tr>
<tr>
<td>Sixth Pass</td>
<td>-77</td>
<td>-5</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>22</td>
<td>66</td>
</tr>
<tr>
<td>Seventh Pass</td>
<td>-77</td>
<td>-5</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>22</td>
<td>66</td>
</tr>
</tbody>
</table>
Dynamic Memory Allocation

- Two standard library functions in `stdlib.h`
  - `calloc()`: Contiguous memory allocation
  - `malloc()`: Memory allocation

- Example usage of `calloc()`:
  ```
  int *a;
  int n;
  scanf("%d", &n);
  a = calloc(n, sizeof(int));
  ```

- The space is initialized with all bits set to 0
Dynamic Memory Allocation (cont.)

- Example `malloc()`:
  
  ```c
  a = malloc(n*sizeof(int));
  ```

- Unlike `calloc()`, `malloc()` does not initialize the memory locations

- In `malloc()` is faster

- Programmer must call `free()` to free the allocated memory with them

- Example: `free(a);`
Strings

- One-dimensional arrays of type char terminated with end-of-string ‘\0’ or null (byte with all bits off)
- Size must include space for ‘\0’
- String constants are written in double quotes, e.g., “abc” (character array of size 4)
- String constant: “a” (size 2) vs character constant: ‘a’ (size 1)
  - Example: `char *p = “abc”; printf(“%s %s\n”, p, p+1);`
  - output: abc bc
Strings (cont.)

- A string constant can be treated as a pointer
  - “abc”[1] and *(“abc” + 2) are legal

- Arrays and pointers differences:
  - char *p = “abc”; char s[] = “abc”;

```
4 bytes
abcb\0
```

```
4 bytes
abcb\0
```

```
4 bytes
abcb\0
```

```
4 bytes
```
Example: String

```c
/* count the number of words in a string */

#include <ctype.h>

int word_cnt(const char *s)
{
    int cnt = 0;
    while(*s != '\0')
    {
        while(ispace(*s)) //skip white space
            ++s;
        if(*s != '\0') //found a word
        { 
            ++cnt;
            while(!ispace(*s) && *s != '\0') //skip the word
                ++s;
        }
    }
    return cnt;
}
```
Library Functions for Strings

- C provide numerous string handling functions in standard library with header `string.h`
- `char *strcat(char *s1, const char *s2);`
- `int strcmp(const char *s1, const char *s2);`
  - `S1` is lexicographically greater, equal or less than `s2`
- `char *strcpy(char *s1, const char *s2);`
- `size_t strlen(const char *s);`
  - 4 bytes machine `size_t` is unsigned int
Implementation: `strlen()`

```c
size_t strlen(const char *s) {
    for (n = 0; *s != '\0'; ++s) ++n;
    return n;
}
```
Implementation: `strcpy()`

```c
char *strcpy(char *s1, register const char *s2) {
    register char *p = s1;
    while(*p++ = *s2++)
        ;
    return s1;
}
```
Implementation: `strcat()`

```c
char *strcat(char *s1, register const char *s2)
{
    register char *p = s1;
    while(*p)
        ++p;
    while(*p++ = *s2++)
        ;
    return s1;
}
```
### String: Declaration and Initialization

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Statements</th>
<th>What gets printed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strlen(s1)</code></td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>strlen(s2+8)</code></td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>strcmp(s1, s2)</code></td>
<td>Negative integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>printf(“%s”, s1+10)</code></td>
<td>Big sky country</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>strcpy(s1+10, s2+8)</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>strcat(s1,”s!”)</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>printf(“%s”, s1)</code></td>
<td>Beautiful brown cows!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two Dimensional Arrays

```c
int a[3][5];
```

Expression Equivalent to `a[i][j]`

- `*(a[i]+j)`
- `(*(a+i))[j]`
- `(*(*(a+i))+j)`
- `*(&a[0][0]+5*i+j)`

Three Dimensional Arrays

```c
int a[7][9][2];
```

Expression Equivalent to `a[i][j][k]`

- `*(&a[0][0][0] + 9*2*i + 2*j + k)`
Arrays of Pointers

- Arrays of pointers have many use
- An array of `char *` is considered as array of strings

**Example:**
```c
char *car_make[9];
char *car_make[9] = {
"Suzuki","Toyota","Nissan","Tata","BMW","Audi","Chevrolet","Honda","Mahindra"};
```

- Sort the strings in lexicographic order
Sort in Lexicographic: Example

Void sort_word(char *w[], int n) {
    int i, j;
    for(i=0;i<n;++i) {
        for(j=i+1;j<n;++j) {
            if(strcmp(w[i],w[j])>0) {
                swap(&w[i],&w[j]);
            }
        }
    }

    void swap(char **p, char **q){
        char *temp;
        temp = *p;
        *p = *q;
        *q = temp;
    }
Arguments to main()

- Two arguments named \texttt{argc} and \texttt{argv} can be used with \texttt{main()} to communicate with the OS.

- Example:

  ```c
  int main(int argc, char *argv[])
  ```

- \texttt{argc} provides a count of the number of command line arguments.

- Array \texttt{argv} is an array of pointers that are the words that make up the command line. Because the element \texttt{argv [0]} contains the name of the command itself, the value of \texttt{argc} is at least 1.
Ragged Arrays

- An array of pointers whose elements are used to point to arrays of varying sizes is called a **ragged array**.

```
char a[2][15] = {"abc:", "a is for apple"};

char *p[2] = {"abc:", "a is for apple"};
```
Functions as Arguments

- In C, pointers to functions can be passed as arguments, used in arrays, returned from function.

- **Example:** you want to do an operation with a variety of functions like $\sum_{k=m}^{n} f^2(k)$.

- In one instance $f(k) = \sin(k)$, in another instance $f(k) = \frac{1}{k}$. 
Implementation: Function as Argument

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

double f(double x) {
    return 1/x;
}

sum_square(f, 1, 100)

sum_square(sin, 1, 100)

Equivalent

double sum_square(double (*f)(double x), int m, int n)
Type Qualifier `const` and `volatile`

- If a variable is declared with a `const` type it can not be changed
  ```
  const int k = 3;
  ```

- The `volatile` variables are modified with some unspecified ways by the hardware. Used seldom.