Lecture #12 – C – Pointers & Strings

BBM 101
Introduction to Programming I

Aykut Erdem, Fuat Akal & Aydın Kaya // Fall 2018
Last time... Functions, Loops and M-D Arrays in C

- Switch statement
- While/for loop
- Multidimensional Arrays
- Functions
Today

- Pointers
- Strings
Variables Revisited

• What actually happens when we declare variables?
  
  ```c
  char a;
  int b;
  ```

• C reserves a byte in memory to store `a`, four bytes to store `b`.

• Where is that memory? At an `address`.

• Under the hood, C has been keeping track of variables and their addresses.

Slide credit: B. Huang
Pointers

• We can work with memory addresses too. We can use variables called **pointers**.

• A **pointer** is a variable that contains the address of a variable.

• Pointers provide a powerful and flexible method for manipulating data in your programs; but they are difficult to master.

  – Close relationship with arrays and strings
Benefits of Pointers

• Pointers allow you to reference a large data structure in a compact way.
• Pointers facilitate sharing data between different parts of a program.
  – Call-by-Reference
• **Dynamic memory allocation:** Pointers make it possible to reserve new memory during program execution.
Pointer Variable Declarations and Initialization

• Pointer variables
  – Contain memory addresses as their values
  – Normal variables contain a specific value (direct reference)
    \[
    \text{count} = 7
    \]
  – Pointers contain address of a variable that has a specific value (indirect reference)
  – Indirection – referencing a pointer value
    \[
    \text{countPtr} \quad \text{count} = 7
    \]
Pointer Operators

• & (address operator)
  – Returns the address of operand
    ```
    int y = 5;
    int *yPtr;
    yPtr = &y;       // yPtr gets address of y
    ```
  – yPtr “points to” y

  ![Diagram showing the relationship between y, yPtr, and y's value]

  Address of y is value of yPtr
Pointer Operators

• * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - \texttt{*yptr} returns \texttt{y} (because \texttt{yptr} points to \texttt{y})
  - * can be used for assignment
    • Returns alias to an object
      \begin{verbatim}
      *yptr = 7; // changes y to 7
      \end{verbatim}
    - Dereferenced pointer (operand of *) must be an \textit{lvalue} (no constants)

• * and \& are inverses
  - They cancel each other out
int rate;
int *p_rate;

rate = 500;
p_rate = &rate;

/* Print the values */
printf("rate = %d\n", rate);  /* direct access */
printf("rate = %d\n", *p_rate); /* indirect access */
/* Using the & and * operators */

#include <stdio.h>

int main()
{
    int a;     /* a is an integer */
    int *aPtr; /* aPtr is a pointer to an integer */

    a = 7;
    aPtr = &a;     /* aPtr set to address of a */

    printf( "The address of a is %p
    The value of aPtr is %p", &a, aPtr );

    printf( "The value of a is %d
    The value of *aPtr is %d", a, *aPtr );

    printf( "Showing that * and & are inverses of each other.
            &*aPtr = %p
            *&aPtr = %p", &*aPtr, *&aPtr );

    return 0;
}

The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
*aPtr = 0012FF88
*aPtr = 0012FF88
# Operator Precedences – Revisited

<table>
<thead>
<tr>
<th>Operators</th>
<th>( )</th>
<th>[ ]</th>
<th>Precedences</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-</td>
<td>++</td>
<td>--</td>
<td>!</td>
<td>* &amp; (type)</td>
</tr>
<tr>
<td>*</td>
<td>/</td>
<td>%</td>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>==</td>
<td>!=</td>
<td></td>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>right to left</td>
</tr>
<tr>
<td>=</td>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
</tr>
<tr>
<td>,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>left to right</td>
</tr>
</tbody>
</table>

- **Associativity**: left to right, right to left
- **Type**: highest, unary, multiplicative, additive, relational, equality, logical and, logical or, conditional, assignment, comma
Addressing and Dereferencing

```c
int a, b, *p;

a = b = 7;
p = &a;
printf("*p = %d\n",*p);

*p = 3;
printf("a = %d\n",a);

p = &b;
*p = 2 * *p - a;
printf("b = %d \n", b);
```

Program Output

```
*p = 7
a = 3
b = 11
```
float x, y, *p;

x = 5;
y = 7;
p = &x;
y = *p;

Thus,

\[
\begin{align*}
  y &= *p; \\
y &= *&x; \\
y &= x;
\end{align*}
\]

All equivalent
Addressing and Dereferencing

| Declarations and initializations | int k=3, j=5, *p = &k, *q = &j, *r;  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expression</strong></td>
<td><strong>Equivalent Expression</strong></td>
</tr>
<tr>
<td>p == &amp;k</td>
<td>p == (&amp;k)</td>
</tr>
<tr>
<td>p = k + 7</td>
<td>p = (k + 7)</td>
</tr>
<tr>
<td>* * &amp;p</td>
<td>* ( * (&amp;p))</td>
</tr>
<tr>
<td>r = &amp;x</td>
<td>r = (&amp; x)</td>
</tr>
<tr>
<td>7 * * p/ *q +7</td>
<td>(( (7 * (*p) ) ) / (*q) ) + 7</td>
</tr>
<tr>
<td>* (r = &amp;j) *= *p</td>
<td>( * (r = (&amp;j))) *= (*p)</td>
</tr>
</tbody>
</table>

Actually, C doesn’t bother about these assignments. But you may lose the track of the data.
The **NULL Value**

- The value null means that no value exists.
- The null pointer is a pointer that ‘intentionally’ points to nothing.
- If you don't have an address to assign to a pointer, you can use NULL.
- NULL value is actually 0 integer value, if the compiler does not provides any special pattern.
  - Do not use NULL value as integer!

counterPtr = &count;     counterPtr = NULL;

```
countPtr    count
    ● ➔ 7
```

```
countPtr
    ● ➔ NULL
```
#include <stdio.h>

int main()
{
    int *dptr;
    dptr = NULL;

    if(dptr == 0)
        printf("The value of dptr is %p \n",dptr);

    if(dptr == NULL)
        printf("The value of dptr is %p\n",dptr);
    //most likely it will crash.
    printf("Value of *dptr is %d \n", *dptr);

    return 0;
}

Program Output
The value of dptr is 00000000
The value of dptr is 00000000
Pointers to void

- `void *identifier;`
- In C, `void` represents the absence of type.
- `void` pointers are pointers that point to a value that has no specific type.
- This allows void pointers to point to any data type.
- The data pointed by void pointers cannot be directly dereferenced.
- We have to use explicit type casting before dereferencing it.
Pointers to void

```
int main(void)
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

Invalid: error - invalid use of void expression

```
int main(void)
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *(int *)ptr);
    return 0;
}
```

Valid: prints “10”
Calling Functions by Reference

• Call by reference with pointer arguments
  – Pass address of argument using & operator
  – Allows you to change actual location in memory
  – Arrays are not passed with & because the array name is already a pointer

• * operator
  – Used as alias/nickname for variable inside of function
    ```c
    void double_it( int *number )
    {
        *number = 2 * ( *number );
    }
    ```
  – *number used as nickname for the variable passed
void SetToZero (int var)
{
    var = 0;
}

• You would make the following call:
  SetToZero(x);

• This function has no effect whatever. Instead, pass a pointer:
  void SetToZero (int *ip)
  {
      *ip = 0;
  }

• You would make the following call:
  SetToZero(&x);

This is referred to as call-by-reference.
/* An example using call-by-reference */
#include <stdio.h>

void change_arg(int *y);

int main (void)
{
    int x = 5;
    change_arg(&x);
    printf("%d \n", x);
    return 0;
}

void change_arg(int *y)
{
    *y = *y + 2;
}
/* Cube a variable using call-by-reference 
with a pointer argument */

#include <stdio.h>

void cubeByReference( int * ); /* prototype */

int main()
{
    int number = 5;

    printf( "The original value of number is %d", number );
    cubeByReference( &number );
    printf( "The new value of number is %d \n", number );

    return 0;
}

void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr; /* cube number in main */
}

Notice that the function prototype takes a pointer to an integer (int *).

Notice how the address of number is given - cubeByReference expects a pointer (an address of a variable).

Inside cubeByReference, *nPtr is used (*nPtr is number).

Program Output

The original value of number is 5
The new value of number is 125
/* Cube a variable using call by value */
#include <stdio.h>

int CubeByValue (int n);

int main(void)
{
    int number = 5;
    printf("The original value of number is %d\n", number);
    number = CubeByValue(number);
    printf("The new value of number is %d\n", number);
    return 0;
}

int CubeByValue (int n)
{
    return (n*n*n);
}
/* Swapping arguments (incorrect version) */
#include <stdio.h>

void swap (int p, int q);
int main (void)
{
    int a = 3;
    int b = 7;
    printf("%d  %d\n", a,b);
    swap(a,b);
    printf("%d  %d\n", a, b);
    return 0;
}

void swap (int p, int q)
{
    int tmp;

    tmp = p;
    p = q;
    q = tmp;
}
/* Swapping arguments (correct version) */
#include <stdio.h>

void swap (int *p, int *q);
int main (void)
{
    int a = 3;
    int b = 7;
    printf("%d %d\n", a, b);
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}

void swap (int *p, int *q)
{
    int tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
/* 
* This function separates a number into three parts: a sign (+, -, 
* or blank), a whole number magnitude and a fraction part. 
* Preconditions: num is defined; signp, wholep and fracp contain 
* addresses of memory cells where results are to be stored 
* Postconditions: function results are stored in cells pointed to by 
* signp, wholep, and fracp 
*/

void separate(double num, char *signp, int *wholep, double *fracp) 
{
    double magnitude;

    if (num < 0)
        *signp = '-';
    else if (num == 0)
        *signp = ' ';
    else
        *signp = '+';

    magnitude = fabs(num);
    *wholep = floor(magnitude);
    *fracp = magnitude - *wholep;
```c
int main()
{
    double value;
    char sn;
    int whl;
    double fr;

    /* Gets data */
    printf("Enter a value to analyze:");
    scanf("%lf", &value);

    /* Separates data value in three parts */
    separate(value, &sn, &whl, &fr);

    /* Prints results */
    printf("Parts of %.4f \n sign: %c \n", value, sn);
    printf("whole number magnitude: %d \n", whl);
    printf("fractional part : %.4f \n", fr);

    return 0;
}
```

Program Output

Enter a value to analyze:13.3
Parts of 13.3000
    sign: +
    whole number magnitude: 13
    fractional part : 0.3000

Enter a value to analyze:-24.3
Parts of -24.3000
    sign: -
    whole number magnitude: 24
    fractional part : 0.3000
sizeof function

- **sizeof**
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if `sizeof( int )` equals 4 bytes, then
    ```c
    int myArray[ 10 ];
    printf( "%d", sizeof( myArray ) );
    ```
    will print 40

- **sizeof** can be used with
  - Variable names
  - Type name
  - Constant values
Finding the Length of an Array

• We can find the length of an array by using sizeof function.

```c
int myArray[ 10 ];
printf( "%d", sizeof( myArray )/sizeof(int) );
```

• will print 10
/* Demonstrating the sizeof operator */
#include <stdio.h>

int main()
{
    char c;           /* define c */
    short s;          /* define s */
    int i;            /* define i */
    long l;           /* define l */
    float f;          /* define f */
    double d;         /* define d */
    long double ld;   /* define ld */
    int array[ 20 ];  /* initialize array */
    int *ptr = array; /* create pointer to array */
Example

```c
printf( "sizeof c = %d\tsizeof(char) = %d"
   "\n sizeof s = %d\tsizeof(short) = %d"
   "\n sizeof i = %d\tsizeof(int) = %d"
   "\n sizeof l = %d\tsizeof(long) = %d"
   "\n sizeof f = %d\tsizeof(float) = %d"
   "\n sizeof d = %d\tsizeof(double) = %d"
   "\n sizeof ld = %d\tsizeof(long double) = %d"
   "\n sizeof array = %d"
   "\n sizeof ptr = %d\n",
 sizeof c, sizeof( char ), sizeof s,
 sizeof( short ), sizeof i, sizeof( int ),
 sizeof l, sizeof( long ), sizeof f,
 sizeof( float ), sizeof d, sizeof( double ),
 sizeof ld, sizeof( long double ),
 sizeof array, sizeof ptr );

return 0;
}
```
### Example

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
</tr>
<tr>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
</tr>
<tr>
<td>ld</td>
<td>12</td>
</tr>
<tr>
<td>array</td>
<td>80</td>
</tr>
<tr>
<td>ptr</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Program Output

33
Pointers and Arrays

- Arrays are implemented as pointers.

- Consider:

```c
double list[3];
&list[1] : is the address of the second element
&list[i] : the address of list[i] which is calculated by the formula
base address of the array + i * 8 (sizeof double)
```
The Relationship between Pointers and Arrays

• Arrays and pointers are closely related
  – Array name is like a constant pointer
  – Pointers can do array subscripting operations

• Declare an array \texttt{b[ 5 ]} and a pointer \texttt{bPtr}
  – To set them equal to one another use:
    \begin{verbatim}
    bPtr = b;
    \end{verbatim}
  • The array name (\texttt{b}) is actually the address of first element of the array \texttt{b[ 5 ]}
    \begin{verbatim}
    bPtr = \&b[ 0 ]
    \end{verbatim}
  • Explicitly assigns \texttt{bPtr} to address of first element of \texttt{b}
The Relationship between Pointers and Arrays

- Element $b[3]$
  - Can be accessed by $*(bPtr + 3)$
    - Where $n$ is the offset. Called pointer/offset notation
  - Can be accessed by $bptr[3]$
    - Called pointer/subscript notation
      - $bPtr[3]$ same as $b[3]$
  - Can be accessed by performing pointer arithmetic on the array itself
    - $*(b + 3)$
/* Using subscripting and pointer notations with arrays */
#include <stdio.h>
int main(void)
{
    int i, offset, b[4]={10,20,30,40};
    int *bPtr = b;

    /* Array is printed with array subscript notation */

    for (i=0; i < 4; i++)
        printf("b[%d] = %d\n", i, b[i]);
Example (cont.)

/* Pointer/offset notation where the pointer is the array name */

for (offset=0; offset < 4; offset++)
    printf("*(b + %d) = %d\n",offset,* (b + offset));

/* Pointer subscript notation */
for (i=0; i < 4; i++)
    printf("bPtr[%d] = %d\n", i, bPtr[i]);

/* Pointer offset notation */
for (offset = 0; offset < 4; offset++)
    printf("*(bPtr + %d) = %d\n", offset"
        "*(bPtr + offset)");

    return 0;
}
Example (cont.)

Program Output

b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
Bubble Sort Using Call-by-reference

• Implement bubblesort using pointers
  – Swap two elements
  – swap function must receive address (using &) of array elements
    • Array elements have call-by-value default
  – Using pointers and the * operator, swap can switch array elements

• Pseudocode

  Initialize array
  print data in original order
  Call function bubblesort
  print sorted array
  Define bubblesort
/* This program puts values into an array, sorts the values into ascending order, and prints the resulting array. */

#include <stdio.h>
#define SIZE 10

void bubbleSort( int *array, const int size );
void swap( int *element1Ptr, int *element2Ptr );
int main() {
   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 }; int i;
   printf( "Data items in original order\n" );

   for ( i = 0; i < SIZE; i++ )
      printf( "%4d", a[ i ] );

   bubbleSort( a, SIZE ); /* sort the array */
   printf( "\nData items in ascending order\n" );
/* loop through array a */
for ( i = 0; i < SIZE; i++ )
    printf( "%4d", a[ i ] );
    printf( "\n" );
return 0; /* indicates successful termination */
} /* end main */

/* sort an array of integers using bubble sort algorithm */
void bubbleSort( int *array, const int size )
{
    int pass,j;
    for ( pass = 0; pass < size - 1; pass++ )
        for ( j = 0; j < size - 1; j++ )
            /* swap adjacent elements if they are out of order */
            if ( array[ j ] > array[ j + 1 ] )
                swap( &array[ j ], &array[ j + 1 ] );
} /* end function bubbleSort */
Example

/* swap values at memory locations to which element1Ptr and element2Ptr point */

void swap( int *element1Ptr, int *element2Ptr )
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
} /* end function swap */

Program Output

Data items in original order
2  6  4  8  10  12  89  68  45  37
Data items in ascending order
2  4  6  8  10  12  37  45  68  89
The Data Type char

• Each character is stored in a machine in one byte (8 bits)
  – 1 byte is capable of storing $2^8$ or 256 distinct values.

• When a character is stored in a byte, the contents of that byte can be thought of as either a character or as an integer.
The Data Type char

• A character constant is written between single quotes.
  ‘a’
  ‘b’

• A declaration for a variable of type char is
  char c;

• Character variables can be initialized
  char c1=‘A’, c2=‘B’, c3=‘*’;
In C, a character is considered to have the integer value corresponding to its ASCII encoding.

<table>
<thead>
<tr>
<th></th>
<th>Lowercase</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘a’</td>
<td>‘b’</td>
<td>‘c’</td>
<td>...</td>
<td>‘z’</td>
<td></td>
</tr>
<tr>
<td>ASCII value</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>...</td>
<td>122</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Uppercase</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘A’</td>
<td>‘B’</td>
<td>‘C’</td>
<td>...</td>
<td>‘Z’</td>
</tr>
<tr>
<td>ASCII value</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>...</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Digit</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘0’</td>
<td>‘1’</td>
<td>‘2’</td>
<td>...</td>
<td>‘9’</td>
<td></td>
</tr>
<tr>
<td>ASCII value</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>...</td>
<td>57</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Other</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘&amp;’</td>
<td>‘*’</td>
<td>‘+’</td>
<td>...</td>
</tr>
<tr>
<td>ASCII value</td>
<td>38</td>
<td>42</td>
<td>43</td>
<td>...</td>
</tr>
</tbody>
</table>
Characters and Integers

• There is no relationship between the character ‘2’ (which has the ASCII value 50) and the constant number 2.
• ‘2’ is not 2.
• ‘A’ to ‘Z’ 65 to 90
• ‘a’ to ‘z’ 97 to 112

• Examples:
  – printf(“%c’,a’);
  – printf(“%c”,97); have similar output.
  – Printf(“%d’,a’);
  – printf(“%d”,97); have also similar output.
The Data Type char

- Some nonprinting and hard-to-print characters require an escape sequence.
- For example, the newline character is written as `\n` and it represents a single ASCII character.

<table>
<thead>
<tr>
<th>Name of character</th>
<th>Written in C</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert</td>
<td>\a</td>
<td>7</td>
</tr>
<tr>
<td>backslash</td>
<td>\</td>
<td>92</td>
</tr>
<tr>
<td>double quote</td>
<td>&quot;</td>
<td>34</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>\t</td>
<td>9</td>
</tr>
</tbody>
</table>
Input and Output of Characters

- `getchar()` reads a character from the keyboard.
  
  \[
  c = \text{getchar}(); /* variable } c \text{ contains the next character of input */}
  
- `putchar()` prints a character to the screen.
  
  \[
  \text{putchar}(c); /* prints the contents of the variable } c \text{ as a character */}
/* Illustrating the use of getchar( ) and putchar( ) */

#include <stdio.h>
int main (void)
{
    char c;
    while ((c=getchar()) != EOF) {
        putchar(c);
        putchar(c);
    }

    abcdef
    aabbccddeeff

    EOF : It is control-d in Unix; control-z in DOS.
/* Capitalize lowercase letters and * double space */

int main(void)
{
    int c;
    while ((c=getchar()) != EOF){
        if ('a' <= c && c <= 'z')
            putchar(c+'A'-'a'); /*convert to uppercase*/
        else if (c == '\n'){
            putchar ('\n');
            putchar ('\n');
        }
        else putchar (c);
    }
}
Character Functions (ctype.h)

<table>
<thead>
<tr>
<th>Function</th>
<th>Nonzero (true) is returned if</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalpha(c)</td>
<td>c is a letter</td>
</tr>
<tr>
<td>isupper(c)</td>
<td>c is an uppercase letter</td>
</tr>
<tr>
<td>islower(c)</td>
<td>c is a lowercase letter</td>
</tr>
<tr>
<td>isdigit(c)</td>
<td>c is a digit</td>
</tr>
<tr>
<td>isalnum(c)</td>
<td>c is a letter or digit</td>
</tr>
<tr>
<td>isspace(c)</td>
<td>c is a white space character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>toupper(c)</td>
<td>changes c to uppercase</td>
</tr>
<tr>
<td>tolower(c)</td>
<td>changes c to lowercase</td>
</tr>
<tr>
<td>toascii(c)</td>
<td>changes c to ASCII code</td>
</tr>
</tbody>
</table>
/* Capitalize lowercase letters and double space */
#include <stdio.h>
#include<ctype.h>

int main(void)
{
    int c;
    while (((c=getchar()) != EOF)){
        if (islower(c))
            putchar(toupper(c)); /*convert to uppercase */
        else if (c == 'n'){
            putchar ('n');
            putchar ('n');
        }
        else  putchar (c);
    }
}
Fundamentals of Strings and Characters

• **Characters**
  – Building blocks of programs
    • Every program is a sequence of meaningfully grouped characters
  – Character constant
    • An `int` value represented as a character in single quotes
    • `'z'` represents the integer value of `z`

• **Strings**
  – Series of characters treated as a single unit
    • Can include letters, digits and special characters (`*, /*, $`) 
  – String literal (string constant) - written in double quotes 
    • "Hello"
  – Strings are arrays of characters in C
    • String is a pointer to first character
    • Value of string is the address of first character
Strings

• A string constant such as “a string” is an array of characters.
• Each element of the array stores a character of the string.
• In its internal representation, the array is terminated with the null character ‘\0’ so that the end of the string can be found easily.
• Thus, the length of the array is defined one more than the number of characters between the double quotes.
Declaring Strings

```c
char myString[10];

myString[0] = 'H';
myString[1] = 'e';
myString[2] = 'l';
myString[3] = 'l';
myString[4] = 'o';
myString[5] = '\0';
```

<table>
<thead>
<tr>
<th>'H'</th>
<th>'e'</th>
<th>'l'</th>
<th>'l'</th>
<th>'o'</th>
<th>'\0'</th>
<th>?</th>
<th>?</th>
<th>?</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Initializing Strings

- Character arrays can be initialized when they are declared:

```c
char name[4] ={'B','B','M','\0'};
char name[4] = "BBM"; /*compiler automatically adds '\0' */
char name[] = "BBM"; /*compiler calculates the size of the array */
```
Strings and Pointers

• We can declare and initialize a string as a variable of type `char *`

```c
char *color = "blue";
```

• But the interpretation is different. “blue” is stored in memory as a string constant. The variable `color` is assigned the address of the constant string in memory.

• If we declare it as:

```c
char c[] = "blue";
```
the array `c` contains the individual characters followed by the null character.
Inputting Strings

• Using subscripts:

```c
char c, name[20];
int i;
for (i = 0; (c = getchar())!='\n'; i ++)
    name[i] = c;
name[i] = '\0';
```

• Using `scanf` and `%s` format:

```c
scanf("%s", name);
```
  – no need to use `&` operator (array name is a pointer too!)
  – it will skip the leading blanks in the input, then characters will be read in. The process stops when a white space or EOF is encountered.
  – Remember to leave room in the array for '\0'
Printing Strings

• Using %s format:
  printf("%s %s\n", "Nice to meet you", name);

• Using subscripts: e.g. printing "name" backwards
  for (--i; i>=0; --i)
    putchar(name[i]);
  putchar(\n');
Examples

- `printf("***Name:%8s*Lastname:%3s*** \n","John", "Smith");`
  - Output:
    ```
    ***Name:   John*Lastname:Smith***
    ```

- `printf("***%-10s*** \n", "John");`
  - Output
    ```
    ***John     ***
    ```

- `scanf("%d%s%d%s", &day,month,&year,day_name);`
  - Example input:
    ```
    5 November 2001 Monday
    ```
String Handling Functions (string.h)

- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Tokenize strings

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strcpy( char *s1, char *s2 )</code></td>
<td>Copies string <code>s2</code> into array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncpy( char *s1, char *s2, int n )</code></td>
<td>Copies at most <code>n</code> characters of string <code>s2</code> into array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strcat( char *s1, char *s2 )</code></td>
<td>Appends string <code>s2</code> to array <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncat( char *s1, char *s2, int n )</code></td>
<td>Appends at most <code>n</code> characters of string <code>s2</code> to array <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
</tbody>
</table>
String Handling Functions (cont.)

- **unsigned strlen(char *s);**
  - A count of the number of characters before \0 is returned.

- **int strcmp(char *s1, char *s2 );**
  - Compares string s1 to s2
  - Returns a negative number if s1 < s2, zero if s1 == s2 or a positive number if s1 > s2

- **int strncmp(char *s1, char *s2, int n );**
  - Compares up to n characters of string s1 to s2
  - Returns values as above
strcpy() and strncpy()  

• We cannot change the contents of a string by an assignment statement.
  ```c
  char str[10];
  str = “test”; /*Error! Attempt to change the base address*/
  ```

• Thus, we need to use string copy functions
  ```c
  - strcpy(str, “test”); /*contents of str changed*/
  - strncpy(str, “testing”, 5);
    str[5] = ‘\0’; /* str contains “testi” only */
  - strcpy(str, “a very long string”); /*overflow of array boundary */
  ```
**strcat() and strncat()**

```c
char s[8]=“abcd”;
strcat(s,”FGH”);  // s keeps abcdFGH

char t[10]=“abcdef”;
strcat(t,”GHIJKLM”);  //exceeds string length!

strncat(t, "GHIJKLM",3);
t[9] = '\0';  // t keeps abcdefGHI
```
**strcmp() and strncmp()**

- We can compare characters with <,>,<= etc.
  
  e.g. ‘A’ < ‘B’

- But we cannot compare strings with the relational operators.
  
  e.g. `str1 < str2` will compare the memory addresses pointed by `str1` and `str2`.

- Therefore we need to use string comparison functions.

  ```
  strcmp("abcd", "abcde") -> returns a negative number
  ```

  ```
  strcmp("xyz", "xyz") -> returns zero
  ```

  ```
  strcmp("xyz", "abc") -> positive number
  ```

  ```
  strncmp("abcde", "abcDEF", 3) -> zero
  ```

  ```
  strncmp("abcde", "abcDEF", 4) -> positive number
  ```
Examples

char s1[] = "beautiful big sky country";
char s2[] = "how now brown cow";

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen(s1)</td>
<td>25</td>
</tr>
<tr>
<td>strlen(s2+8)</td>
<td>9</td>
</tr>
</tbody>
</table>

Statements

printf("%s", s1+10);   // big sky country
strcpy(s1+10, s2+8)
strcat(s1, "s!");
printf("%s", s1);     // beautiful brown cows!
#include <stdio.h>
#include <string.h>
#define LENGTH 20

/* A string is a palindrome if it reads the same backwards and forwards. e.g. abba, mum, radar. This program checks whether a given string is palindrome or not.

int isPalindrome(char s[]);  // function prototype

int main()
{
    char str[LENGTH];

    // read the string
    printf("Enter a string ");
    scanf("%s", str);

    // Check if it is a palindrome.
    if (isPalindrome(str))
        printf("%s is a palindrome.\n", str);
    else
        printf("%s is not a palindrome.\n", str);
}

```c
int isPalindrome(char str[])
{
    int i, j, flag;

    i = 0;                   // index of the first character
    j = strlen(str) - 1;     // index of the last character
    flag = 1;                // assume it is a palindrome
    while (((i<j) && flag)){
        // compare the ith and jth. characters
        if (str[i] != str[j])
            flag = 0;  // if not same then string cannot be a palindrome.
        else {
            i++;
            j--;
        }
    }                // advance to next characters
    return flag;
} 
```
#include <stdio.h>
#include <string.h>

#define LENGTH 20

// This program converts a positive integer to a binary number which is represented as a string. For instance decimal number 12 is 1100 in binary system.

void toBinary(int decVal, char *); //function prototype

int main()
{
    int num;
    char bin[LENGTH];

    // read a positive integer
    printf("Enter a number: ");
    scanf("%d", &num);

    // Convert the number and print it.
    toBinary(num, bin);
    printf("Binary equivalent of %d is : %s", num, bin);
}

void toBinary(int decVal, char *sb) {

    char s0[LENGTH], s1[LENGTH];

    // create an empty string.
    strcpy(sb,"" );
    if (decVal == 0)
        strcat(sb,"0"); // if number is zero result is 0
    else                // otherwise convert it to binary
        while (decVal != 0) {
            strcpy(s0,"0");
            strcpy(s1,"1");
            if (decVal%2 == 0)
                strcpy(sb,strcat(s0,sb)); //last character is 0
            else
                strcpy(sb,strcat(s1,sb)); //last character is 1
            decVal = decVal / 2; /* advance to find the next digit */
        }
    return sb;
}