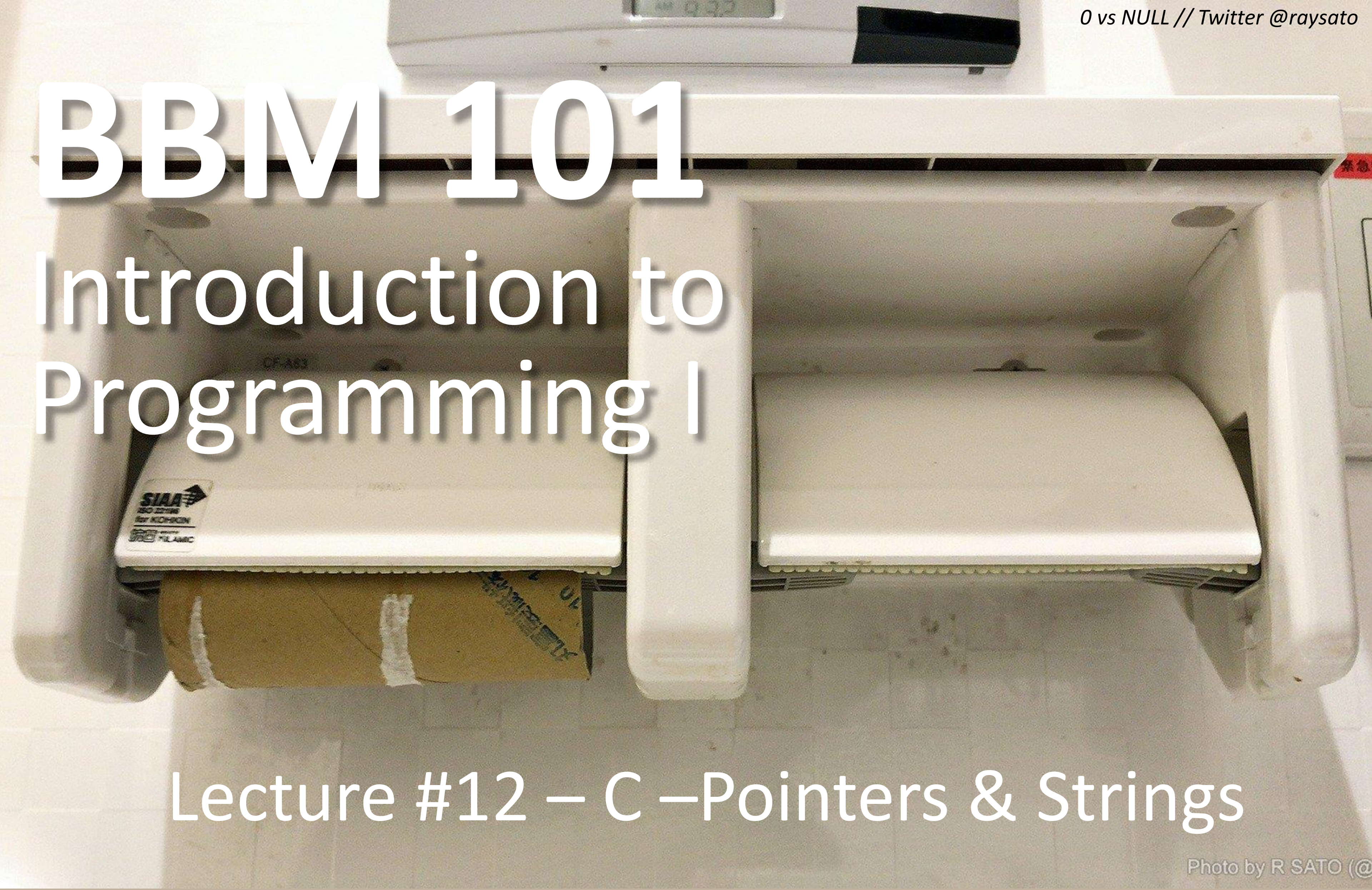
0 vs NULL // Twitter @raysato

BBM 101 Introduction to Programming I



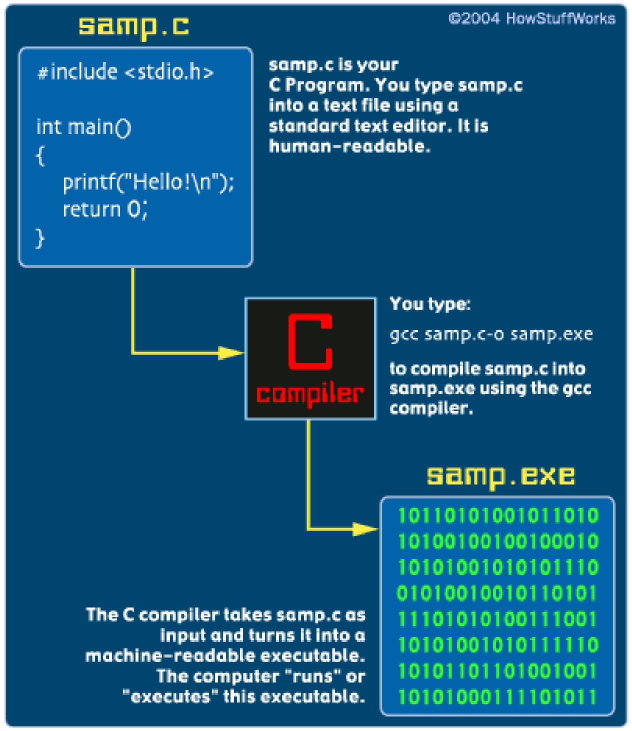
Lecture #12 – C –Pointers & Strings

HACETTEPE UNIVERSITY

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


samp.c
©2004 HowStuffWorks
samp.c is your C Program. You type samp.c into a text file using a standard text editor. It is human-readable.
You type:
to compile samp.c into samp.exe using the gcc compiler.
C compiler
samp.exe
The C compiler takes samp.c as input and turns it into a machine-readable executable. The computer "runs" or "executes" this executable.

```
#include <stdio.h>

int main()
{
   printf("Hello!\n");
   return 0;
}
```

```
gcc samp.c-o samp.exe
```

```
10110101001011010
10100100100100010
10101001010101110
01010010010110101
11101010100111001
10101001010111110
10101101101001001
10101000111101011
```

Today

- Pointers
- Strings

Variables Revisited

- What actually happens when we declare variables?

```
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.

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.

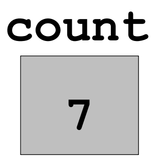
– Close relationship with arrays and strings

Benefits of Pointers

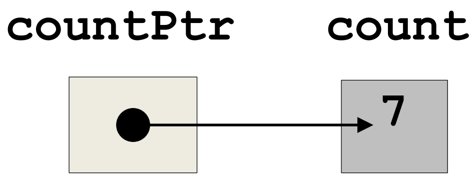
- Pointers allow you to reference a large data structure in a compact way.
- Pointers facilitate sharing data between different parts of a program.
  - Call-by-Reference
- **Dynamic memory allocation:** Pointers make it possible to reserve new memory during program execution.

Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value

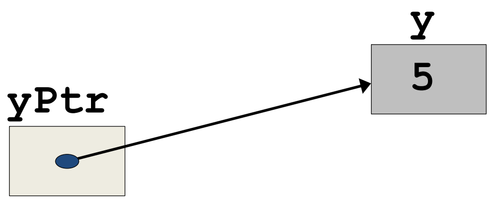


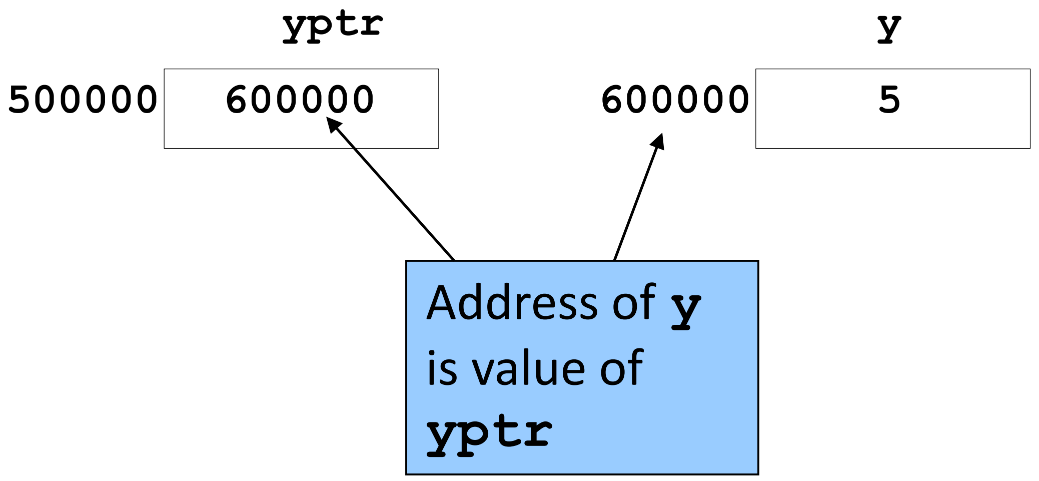
Pointer Operators

- **`&`** (address operator)
  - Returns the address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;      // yPtr gets address of y
```

  - **`yPtr` “points to” `y`**





Address of **`y`** is value of **`yptr`**

Pointer Operators

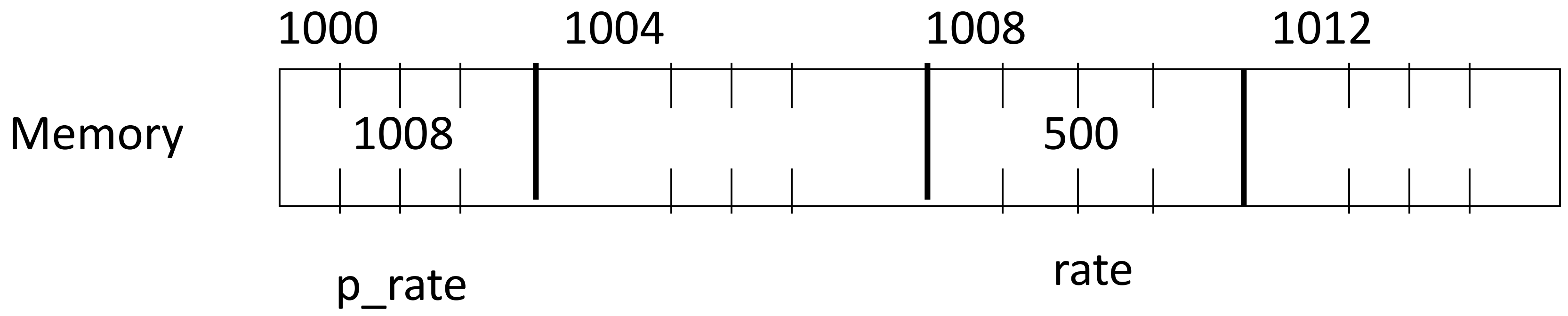
- **`*`** (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - **`*yptr`** returns **`y`** (because **`yptr`** points to **`y`**)
  - **`*`** can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  // changes y to 7
```

  - Dereferenced pointer (operand of *) must be an *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 */
```

```c
/* 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\nThe value of aPtr is %p", &a, aPtr );


   printf( "\n\nThe value of a is %d\nThe value of *aPtr is %d", a, *aPtr );


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



   return 0;
}
```

The address of **a** is the value of **aPtr.**

The * operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how * and **&** are inverses

Program Output

```
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

Operators								Associativity	Type
()	[]							left to right	highest
+	-	++	--	!	*	&	(type)	right to left	unary
*	/	%						left to right	multiplicative
+	-							left to right	additive
<	<=	>	>=					left to right	relational
==	!=							left to right	equality
&&								left to right	logical and
||								left to right	logical or
?:								right to left	conditional
=	+=	-=	*=	/=	%=			right to left	assignment
,								left to right	comma

Addressing and Dereferencing

```
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
```

Addressing and Dereferencing

```
float x, y, *p;

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

Thus,

```
      y = *p;
      y = *&x;
      y = x;
```

} All equivalent

Addressing and Dereferencing

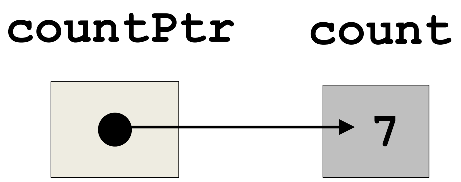
Declarations and Initializations		
`int k=3, j=5, *p = &k, *q = &j, *r;`		
`double x;`		
Expression	**Equivalent Expression**	**Value**
`p == &k`	`p == (&k)`	1
`p = k + 7`	`p = (k + 7)`	illegal
`* * &p`	`* ( * (&p))`	3
`r = &x`	`r = (& x)`	illegal
`7 * * p/ *q +7`	`(( (7 * (*p) )) / (*q)) + 7`	11
`* (r = &j) *= *p`	`( * (r = (&j))) *= (*p)`	15

Actually, C doesn't bother about these assignments. But you may lose the track of the data.

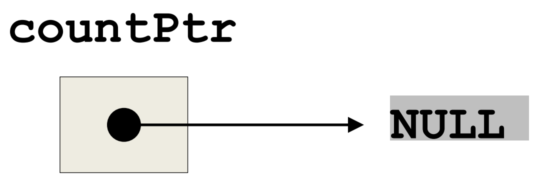
The **NULL** Value

- 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;
```



NULL pointer Example

```c
#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

```
void double_it( int *number )
 {
   *number = 2 * ( *number );
 }
```

  - ***number** used as nickname for the variable passed

Passing parameters by reference

```
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( "\nThe 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`**).

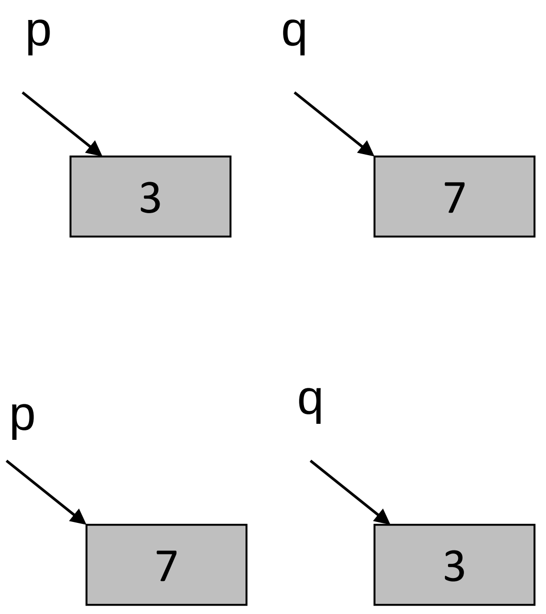
Program Output

```
The original value of number is 5
The new value of number is 125
```

```c
/* 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;
}
```

```c
/* 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;
}
```

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
```

```
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;
}
```

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

```
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.

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

  - will print `10`

Example

```c
/* 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

```
  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

Program Output

```
sizeof c = 1         sizeof(char)  = 1
sizeof s = 2         sizeof(short) = 2
sizeof i = 4         sizeof(int) = 4
sizeof l = 4         sizeof(long) = 4
sizeof f = 4         sizeof(float) = 4
sizeof d = 8         sizeof(double) = 8
sizeof ld = 8        sizeof(long double) = 12
sizeof array = 80
sizeof ptr = 4
```

Pointers and Arrays

- Arrays are implemented as pointers.

- Consider:

  `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 **`b[ 5 ]`** and a pointer **`bPtr`**
  - To set them equal to one another use:

    ```
    bPtr = b;
    ```

    - The array name (**`b`**) is actually the address of first element of the array **`b[ 5 ]`**

    ```
    bPtr = &b[ 0 ]
    ```

    - Explicitly assigns **`bPtr`** to address of first element of **`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 )`

Example

```
/* 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
```

Example

```
/* 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" );
```

Example

```
/* 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.

lowercase	`'a'`	`'b'`	`'c'`	...	`'z'`
ASCII value	97	98	99	...	122
uppercase	`'A'`	`'B'`	`'C'`	...	`'Z'`
ASCII value	65	66	67		90
digit	`'0'`	`'1'`	`'2'`	...	`'9'`
ASCII value	48	49	50	...	57
other	`'&'`	`'*'`	`'+'`	...	
ASCII value	38	42	43		

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.

Name of character	**Written in C**	**Integer Value**
alert	\a	7
backslash	\\	92
double quote	\”	34
horizontal tab	\t	9

Input and Output of Characters

- **getchar ( )** reads a character from the keyboard.

```
c = getchar();  /* variable c contains the
                   next character of input */
```

- **putchar ( ):** prints a character to the screen.

```
putchar(c);    /* prints the contents of the
                  variable c as a character */
```

```c
/* 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);
   }
}
```

```
cop3223!c C
```

Character Functions (ctype.h)

Function	**Nonzero (true) is returned if**
`isalpha(c)`	c is a letter
`isupper(c)`	c is an uppercase letter
`islower(c)`	c is a lowercase letter
`isdigit(c)`	c is a digit
`isalnum(c)`	c is a letter or digit
`isspace(c)`	c is a white space character

Function	**Effect**
`toupper(c)`	changes c to uppercase
`tolower(c)`	changes c to lowercase
`toascii(c)`	changes c to ASCII code

```
/* 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

```
char myString[10];
myString[0] = 'H';
myString[1] = 'e';
myString[2] = 'l';
myString[3] = 'l';
myString[4] = 'o';
myString[5] = '\0';
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Initializing Strings

- Character arrays can be initialized when they are declared :

```
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 *`**

```
    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:

```
    char c[] = “blue”;
```

  the array `c` contains the individual characters followed by the null character.

Inputting Strings

- Using subscripts:

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

- Using scanf and %s format:

```
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

Function prototype	Function description
`char *strcpy( char *s1, char *s2 )`	Copies string **`s2`** into array **`s1`**. The value of **`s1`** is returned.
`char *strncpy( char *s1, char *s2, int n )`	Copies at most **`n`** characters of string **`s2`** into array **`s1`**. The value of **`s1`** is returned.
`char *strcat( char *s1, char *s2 )`	Appends string **`s2`** to array **`s1`**. The first character of **`s2`** overwrites the terminating null character of **`s1`**. The value of **`s1`** is returned.
`char *strncat( char *s1, char *s2, int n )`	Appends at most **`n`** characters of string **`s2`** to array **`s1`**. The first character of **`s2`** overwrites the terminating null character of **`s1`**. The value of **`s1`** is returned.

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.

```
char str[10];
str = “test”;  /*Error! Attempt to change the base
                address*/
```

- Thus, we need to use string copy functions
  - `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()

```
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”;
```

Expression	**Value**
strlen(s1)	25
strlen(s2+8)	9

Statements	**What is printed**
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 programs 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);
 }
```

```
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;
}
```