

Welcome to Week 7!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 7 folder**.
- Read **Chapter 9** in the latest edition of the textbook (chapter 10 in earlier editions).
- Begin **Quiz 7**. It is due Sunday at midnight.
- Begin **Assignment 6**. It is due Sunday at midnight.

Recommended (optional) activities

- Attend **chat** this Thursday night, from 8:00 pm - 9:00 pm Eastern Time. Although chat participation is optional, it is highly recommended.
- Post any questions you might have on this week's topic in the **Week 7 Discussion Forum** located in the course **Discussion Board**. Please ask as many questions as needed, and don't hesitate to answer one-another's questions.
- Try out the various **code examples** in the lecture notes. Feel free to modify them and conduct your own "**What ifs**".

"No matter what planet we seem to visit, Dr. McKoy always has this to say about one of my crew members who wear a red shirt: --- He's Dead Jim. ---"

--- James T. Kurt
Captain, USS Enterprise

Introduction to Strings

This week we will learn how to work with data contained within a **Character String**.

Remember that a variable can be of type **character**, and that characters are enclosed within single quotes

```
char letter = 'A' ;
```

The most important character we will discuss this week is called the **null terminator character**. Its use will be apparent in the upcoming lectures notes, but for now, I wanted to alert you to its existence and that it is represented as a character with a backslash zero as shown below:

```
char null_value = '\0';
```

The main topic this week is character strings.

Important Definition: A **character string** in C is nothing more than one or more consecutive characters in memory that ends with a null terminator character.

When you think of a data type to store strings that are all characters, and consecutive in memory, your first thought should be that they should be stored in an **Array**, and that is the right way to be thinking! If I wanted to represent the word **Hello** as a string within an array, one way I could do it is by the following statement:

```
char word [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The array could be visualized as:

'H'	'e'	'l'	'l'	'o'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]

In C, there is no **type string**. We have int, float, and char, each of which represents the memory used. Since strings are stored as characters, we use an array of type character. C does not support string manipulation, but has **functions** that can be employed to do the job.

Characters are normally sized at **8 bits or 1 byte**. If the first character location in our array example above was address 1000, this is what it would look like in memory in the diagram below. Note that each memory address is consecutive in memory, and what makes it a string is that there is a null terminator character at the end.

'H'	'e'	'l'	'l'	'o'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]
1000	1001	1002	1003	1004	1005

If you decide to define a string within a array, just make sure you leave enough room for the null terminator character if specifying the size of the array.

```
char word [ 6 ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Star Trekking - Stringing Along a Few Quotes

In the world of Star Trek, there are countless famous quotes attributed to their characters, where each has their own favorite saying. Quotes are just a series of words and phrases that you would store in a character string within the C Programming Language. Seems that Captain Kurt is quite the impressionist ... and while appearing on Larry Cling Live, he shares some of his favorites. Prepared to "entertained" with "the Captain" by watching the Movie which you can access by clicking the image below or from its own link back in our weekly lecture notes.



Video - Strings and Characters

This video concentrates on how strings and characters are represented in memory and the various things you can do with them through the use of various operations and functions. Strings and characters are common to just about any modern programming language and a good understanding of them can really help you get the most out of your program designs and implementations.



Character Operations

Whenever a character constant or variable is used in an expression in C, it is automatically converted to, and subsequently treated as, an **integer** value. This applies when characters are passed as arguments to functions as well: they are automatically converted to integers by the system.

For example: a statement to verify whether the character variable is lower case could be written on a system that uses **ASCII** character representation as:

```
char c; /* character variable */

if ( c > 'a' && c <= 'z' )
    printf ("character %c is lower case\n", c);
```

The **ASCII (American Standard Code for Information Interchange)** was developed as a numerical representation of characters, as computers don't really understand character values, but they definitely understand numbers, so its really just a mapping between characters and numbers that a computer can work with. These characters include letters (a-z, A-Z), digits (0-9), punctuation (periods, semicolons, question marks, etc.), and any character you can represent on your keyboard.

Please take this time to look at the ASCII Table, which can be found in the Appendix of your Kochan textbook, or at the following link:

<http://www.asciitable.com/>

Below is a partial ASCII table to review while we concentrate on examples with lower case letters

Decimal	Hexadecimal	Octal	Binary	Character
0	00	000	00000000	NUL
...
65	41	101	01000001	A
66	42	102	01000010	B
67	43	103	01000011	C
68	44	104	01000100	D
...
87	57	127	01010111	W
88	58	130	01011000	X
89	59	131	01011001	Y
90	5A	132	01011010	Z
91	5B	133	01011011	[
92	5C	134	01011100	\
...
97	61	141	01100001	a
98	62	142	01100010	b
99	63	143	01100011	c
100	64	144	01100100	d
...
121	79	171	01111001	y
122	7A	172	01111010	z
123	7B	173	01111011	{
124	7C	174	01111100	

125	7D	175	01111101	I
126	7E	176	01111110	~
127	7F	177	01111111	DEL

Notice that 'a' has an value a decimal value 97 while 'z' has a value of 122. Therefore, one could also check to see if a character was lower case by using **decimal** values.

```
if ( c >= 97 && c <= 122 ) /* ASCII decimal values to check between 'a' and 'z' */
```

Alternatively, we could also use **hexadecimal** or **octal** values as well (remember that hexadecimal numbers in C start with **0x** and octal numbers start with **0**) to do the same check:

```
if ( c >= 0x61 && c <= 0x7A ) /* using hexadecimal to check between 'a' and 'z' */
```

```
if ( c >= 0141 && c <= 0172 ) /* using octal to check between 'a' and 'z' */
```

Why can't we just say

```
if ( c > 'a' )
```

for checking just for lower case letters? Notice that there are other character values (starting at 123) that are not letters. We can check the upper and lower limits to determine if a character value is a lower case letter, just like we've done in both cases above. However, ask yourself, which statement is better, the first or the second one?

```
if ( c >= 'a' && c <= 'z' )
```

OR

```
if ( c >= 97 && c <= 122 )
```

The answer is the **first** one! If someone had to review or maintain your code, you don't want them to have to consult the ASCII table to figure out what letters or characters you are using. Always make your code **readable** to the next person. You may not always be around to maintain the code in the future. For the rest of the notes going forward, we'll tend to go with the ASCII character representation, for example, using 'a' instead of 97.

Note that there are ASCII values for upper and lower case values as well. An 'A' has a value of 65 and 'Z' is 90. If you **sort** characters, you'll notice that uppercase letters get sorted before lower case letters.

The **printf** command can be used to print out the value used to internally represent the character stored inside the variable **char_val** below on your machine.

```
char_val = 'a';
printf ("%c %d %o %x \n", char_val, char_val, char_val, char_val);
```

will print:

```
a 97 141 61
```

and this statement:

```
printf ("%c \n", char_val = 'a' + 3); /* assigns value of 100 (97 + 3) to the variable named char_val */
```

will print:

```
d
```

Escape Characters

There are various **escape characters** that are also available. Some of these you can see, others like a new line or a tab, are "white spaces" as you can't really see them on a screen, but you know they are there.

Character	Name	Octal
\b	backspace	\010
\f	form feed	\014
\n	new line	
\r	carriage return	
\t	horizontal tab	
\v	vertical tab	
\\	backslash	
\"	double quote	
\'	single quote	
\?	question mark	

Check out the ASCII table value in the book or online (<http://www.asciitable.com>) to find the octal values for **\n**, **\r**, **\t**, and **\v**

The answers are **012**, **015**, **011**, and **013**

****, **\"**, and **\'** are used to include those characters in places where, without the escape, they would have different meaning.

```
printf ("\t in the horizontal tab character. \n");
```

... would print the string and start at a new line:

```
\t is the horizontal tab character
```

**** followed by a carriage return is used to **continue a line**. Note the continuation character shown at the end of the first line. You can do this over as many lines as you wish when setting or printing strings.

```
char digits = {"This is turning out to be a very very \  
very long line."};
```

Interesting Character

Try printing the character **\007** in a program to make your device beep.

Character Functions

There are many C Library character functions that can be used to test or manipulate a single character. Below is a table of most of them. In order to use these functions, you'll need to include the following header file:

```
#include <ctype.h>
```

Function Example	Description
ch = toupper(ch);	convert lower to upper case
ch = tolower(ch);	convert upper to lower case
i = isupper(ch);	returns true for upper case
i = islower(ch);	returns true for lower case
i = isalpha(ch);	returns true for any letter
i = isdigit(ch);	returns true for digits 0-9
i = isprint(ch);	returns true for printable character
i = isalnum(ch);	returns true for letter or digit
i = iscntrl(ch);	returns true for octal 0 thru 037 or 0177
i = isxdigit(ch);	returns true for hex digit, 0-9, A-F
i = ispunct(ch);	returns true for > octal 40 and not letter or digit

In the ASCII table, you will find control characters such as form feed (octal 14, /014) or delete (Octal 177)

For example, toupper returns its argument (ch) as an upper case character. If (ch) is already upper case, it is returned unchanged.

A sample function to set a string to all uppercase would be as follows:

```

/*****
**  Function: str_to_upper
**
**  Description: Converts all lowercase characters in a given
**  string to upper case. It will verify if each
**  character is lower case first.
**
**  Parameters: string - a character string (input)
**
**  Returns: string - converted to upper case
*****/

void string_to_upper (char string [ ] )
{
    int i; /* loop index */

    for (i = 0; string[i] != '\0'; ++i)

```

```
    {
        if ( islower ( string [i] ) )
            string [i] = toupper ( string [i] );
    }
}
```

Working with Strings

Recall that in the previous lecture note, we initialized a string value into an array of characters by individually sequencing a set of characters within single quotes:

```
char word [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

While that can work, its really not the best way to get the job done. An equivalent statement to initialize an array to contain the string Hello would be to enclose the characters within **double quotes**:

```
char word [ ] = { "Hello" };
```

Either way would represent the variable named `word` in memory as an array like so:

'H'	'e'	'l'	'l'	'o'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]
1000	1001	1002	1003	1004	1005

Three major **reasons why using double quotes** for a string are better than individually sequencing characters in an initialization are:

- 1) Any string within double quotes will automatically append the null terminator character to the end of the string
- 2) It can be painful to put together a string with each character being put between single quotes and separated by comma.
- 3) Constructing a string with single characters between quotes only works when you are declaring an array of characters

To **print** a string, you can use the **%s format**, which expects a character array and will print a character at a time from the address provided (normally the array name) until it encounters a null terminator character.

```
printf ("%s \n", word);
```

Remember that in C, an array name is seen by the C compiler as the address of the first element of the array:

```
&word [ 0 ]
```

You could also say:

```
printf ("%s \n", &word[0] );
```

The printf statement will see the **%s format** and know that it will printing out a string. Based on our memory layout above, it will start at location 1000 and continue printing characters one at a time to the screen until it encounters a null terminator character.

You could think of it working like this:

Start at location 1000, is that the null terminator? No, print that character	H
Continue to location 1001, is that the null terminator? No, print that character	He
Continue to location 1002, is that the null terminator? No, print that character	Hel
Continue to location 1003, is that the null terminator? No, print that character	Hell
Continue to location 1004, is that the null terminator? No, print that character	Hello
Continue to location 1005, is that the null terminator? Yes, stop printing	

Common Error:

A common error new C programmers make is confusing single quotes with double quotes. Use single quotes for a single character only. Such as 'a', 'B', '#' and those special characters that start with back slash, like '\n', '\t', and '\0'. Put character strings within double quotes, such as "text", "ABC123", and "Hello There".

Don't use them the wrong way:

```
char value = "ABC"; /* A single char value trying to hold 4 chars, including the null terminator char */
```

value = 'ABC'; */* This is not a character string, its between single quotes, not double quotes */*

Challenge Problem:

Write an array of structures and initialize it to include the names: Connie Cobol, Mary Apl, Frank Fortran, Jeff Ada, and Anton Pascal.

... Answer on the next page

Challenge Problem Solved

Problem: Write an *array of structures* and initialize it to include the names: Connie Cobol, Mary Apl, Frank Fortran, Jeff Ada, and Anton Pascal.

Answer: The code to solve this problem is shown below and can also be tried out at: <http://ideone.com/n9ck3B>

```
#include <stdio.h>
#define SIZE 5

struct employee
{
    char first_name [20]; /* First name */
    char last_name [20]; /* Last name */
};

/*****
**
** Name: printNames
**
** Description: Prints the first and last names in an array
**
** Parameters: myNames - Array of Structures containing names
**
** Returns: Nothing
**
*****/

void printNames (struct employee myNames [ ] )
{
    int i; /* loop index */

    /* Print first name and then last name */
    for (i = 0; i < SIZE; ++i)
    {
        printf ("%s %s \n", myNames [ i ].first_name,
                  myNames [ i ].last_name);
    }

} /* printNames */

int main()
{
    struct employee names [SIZE] =
    {
        { "Connie", "Cobol" },
        { "Mary", "Apl" },
        { "Frank", "Fortran" },
        { "Jeff", "Ada" },
        { "Anton", "Pascal" } /* no comma here */
    };

    printNames ( names );

    return (0);
}
```

Output:

Connie Cobol
Mary Apl
Frank Fortran
Jeff Ada
Anton Pascal

Additional Challenge:

Could you add a function that will print all names starting with last name, then first name?

Initializing and Storing with Large Strings

If you wanted to store a long string within a character array variable and initialize it over multiple lines, you will need to use the **continuation character**. The continuation character is a backslash character (\) that you place at the end of each line. Use it anytime as needed to initialize a string that may need to be placed on multiple lines within your program. An example is shown below, with the continuation character shown in purple.

```
char myString [ ] = {"All we are saying is give \
peace a chance."};
```

If you did not use it in the statement above, you would get an error:

```
char myString [ ] = {"All we are saying is give
peace a chance."};
```

```
prog.cpp:6:22: warning: missing terminating " character [enabled by
default]
prog.cpp:6:5: error: missing terminating " character
```

Note that you can also use **constants** to define a string. Use and pass them any place a string would be accepted. If a string is not going to change, then it makes sense to store it in a meaningful and unique *symbolic constant name*. See the below program which has two quotes from John Lennon, each stored in their own constant, with any new lines and other special characters added to help with printing it in the way it was intended over multiple lines.

You can try it out at: <https://ideone.com/2KkDUr>

```
#include <stdio.h>

#define JOHN_LENNON_PEACE "\nAll we are saying is give peace a chance."

#define JOHN_LENNON_PEACE2 "\nPeace is not something you wish \
for;\nIt's something you make, something you do,\nsomething \
you are, and something you give away."

int main(void) {

    /* print both quotes */
    printf ("%s", JOHN_LENNON_PEACE);
    printf ("\n");
    printf ("%s", JOHN_LENNON_PEACE2);

    return 0;
}
```

Output

All we are saying is give peace a chance

Peace is not something you wish for;
It's something you make, something you do,
something you are, and something you give away.

Video - String Library Functions

This 10 minute video will go over the common string functions available in the C Library, in particular, strlen, strcpy, strcat, and strcmp. Once you understand how these functions work, locating and incorporating any of the available C Library string functions will be a breeze.



The Standard C Library

The **Standard C library** contains a large assortment of functions to help make a programmer's task easier. On most systems, the standard C library is automatically searched by the linker when you link your program.

To use the routines, just go ahead and call them in your program, including the appropriate header file as necessary for your program. One example we have seen so far is **stdio.h**. It is important to remember to include the correct header file for the following reasons:

1. It contains the correct declarations (i.e., **prototypes**) for each library function.
2. Some items, like **Macros** (see Chapter 13/14) are defined in them
3. **Constants** and **Types** are sometimes declared

What is important to note is the C code for the function is not included in the header file, just its prototype and any needed constants, types, and macros in order for it to run correctly.

For string routines, the following include file is needed:

```
#include <string.h>
```

This include statement below is exactly the same of the previous one. In fact, string.h is often linked to strings.h. You only need to provide one, NOT BOTH !!!!! The string.h file is the standard one, and you can't always guarantee that a strings.h file even exists (for example, won't work on IDEOne). I would recommend including only **string.h** when using string library routines.

```
#include <strings.h>
```

Let's look at some common string functions:

1. **strlen** - string length
2. **strcpy** - string copy
3. **strcat** - string cat
4. **strcmp** - string compare

Important Note:

Like any good author, Stephen Kochan will come up with easy to understand functions that will be covered while discussing Strings in Chapter 10. However, while they are good to review to help you understand what is going on, please do not use those functions in any homework, exam, and programs you develop at school or, worse, at work. You want to always use tried and true functions in the *Standard C Library*. These are standard, fast, and most importantly, reliable. The same can be said with various character functions that we covered earlier.

Interested in reading more about the Standard C Library? Check out what Wikipedia has to say, which provides details and links to each item in the Standard C Library.

http://en.wikipedia.org/wiki/C_standard_library

The strlen function

The **strlen** function returns an **integer** representing the number of characters in a character string given as its argument. It does not include the terminating null character in its count. The function is defined as follows:

```
int strlen (char str [ ] )
```

Its really important to understand that **strlen** does not return the number of elements in an array, rather, the number of characters in a string starting at a specific memory location that is passed to it.

Consider the following declaration:

```
char s1 [20] = { "Some String" };
```

It will store "Some String" in an array called s1. Note that at element 11, a null terminator was automatically added to indicate the end of the string because double quotes were used to initialize the string into array s1 in the declaration. It is also likely since you initialized the s1 array, that the elements s[1] through s[19] are also set to '\0'. Remember that '\0' is equivalent to the 0 in the ASCII character set. You can see that the string "Some String" has been placed into our array below, shown in [Blue](#).

s1

Element Value	'S'	'o'	'm'	'e'	' '	'S'	't'	'r'	'i'	'n'	'g'	'\0'								
Array Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]	[19]
Memory Address	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019

Now, what is the length of the string starting at the beginning of the array s1? Let's use the strlen function to figure that out.

```
printf ("The length of the string at s1 is: %i ", strlen ( s1 ) );
```

Again, an equivalent statement would be passing the address of the first element of the array:

```
printf ("The length of the string at s1 is: %i ", strlen ( &s1 [ 0 ] ) ); /* same as s1 */
```

Given either of the two strlen function calls above, it will start at the memory location of the first element of our array, which in our example, is memory location 1000. It will then start its count and continue to count each consecutive character in memory until the null terminator character is found. In our example, it is array element **s1 [11]** that contains the first null terminator character encountered, and it is located at memory location 1011.

For the purposes of the strlen function, once it reaches the end of the string at location 1011, it does not care what's in the rest of the array, its only concern is determining the string length starting at a given memory location. Thus, either call above will print:

The length of the string at s1 is 11

You don't have to give strlen the starting address of the array, I could give it the address of any element for an array of characters. In the example below, I passed it the address of element s1 [5]

```
printf ("The length of the string at s1 is: %i ", strlen ( &s1 [ 5 ] ) );
```

It will access memory location 1005, and start a count of characters until it encounters the first null terminator character at element s [11] , which is memory location 1011

The length of the string at s1 is 6

If I printed the string that starts at location &s1 [5], using the %s format in a printf statement:

```
printf ("%s", &s [5 ] );
```

It will start printing characters found at that memory location and beyond until it encounters the first null terminator, so it will print:

String

Its important to understand that you have to pass strlen an **address**, you can't simply pass an array element, as a specific array element is not an address, rather it would pass the contents of whatever is stored there, in other words a value.

```
printf ("The length of the string at s1 is: %i ", strlen ( s1 [ 5 ] ) ); /* will not work, passes the letter 'S' */
```

As you can see, there are lots of strings located in the s1 array. Let's conclude by looking at a small program to see what would be printed with each strlen call. Note that in order for this function to work correctly, you have to include the **string.h** header file.

You can watch it work at: <https://ideone.com/ntSQ3t>

```
#include <stdio.h>
#include <string.h>

int main ()
{

    char s1 [20] = {"Some String"}; /* test string */
    int retVal; /* holds return value */

    printf ("The string at the start of array s1 has a size of %d \n", strlen (s1));
    printf ("The empty string has a size of %d \n", strlen (""));
    printf ("String with just a single space character, size is %d \n", strlen (" "));
    printf ("Two special characters in a string, size is %d \n", strlen ("\007\n"));

    retVal = strlen (s1) + 10; /* 11 + 10 */

    printf ("\n retVal = %i \n", retVal );

    return (0);
}
```

Output:

```
The string at the start of array s1 has a size of 11
The empty string has a size of 0
String with just a single space character, size is 1
Two special characters in a string, size is 2
retVal = 21
```


The strcpy function

The **strcpy** function takes two arguments, both character pointers. Don't worry about the term character pointer, we'll cover that next week. Know that character pointers and arrays are similar in that they both work with memory addresses. With this function, like most other string functions, it is expected that the characters arrays contain a null terminator character.

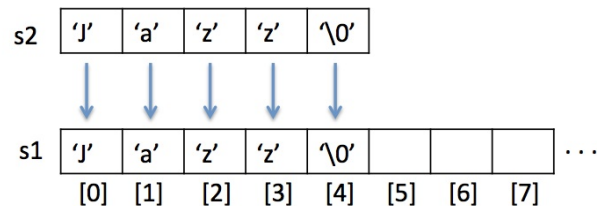
```
strcpy (char s1 [ ], char s2 [ ] )
```

This function copies the character string pointed to by the second argument to that pointed to by the first.

```
char s1 [10]; /* declare two character arrays to hold string values */
char s2 [5];  /* ... this one is only 5 characters long */

strcpy (s2, "Jazz"); /* Put the string Jazz into s2 */

strcpy (s1, s2); /* Copy the string starting at s2 into s1 */
```



Warning: Make sure that there is enough space in the destination character array to accommodate the string to be copied.

But why does C need a function like this???? The reason is that Strings are interpreted by the C compiler as an Address, likewise, Array Names are seen as addresses as well. Say you have these two declarations:

```
char oldQuote [ ] = { "Now is the time" };
char newQuote [ ] = {"What's Up?"};
```

The next two statements are invalid. You can't do any of these, as its trying to set one address to another address:

```
oldQuote = newQuote; /* can't set one array address to another */
```

The compiler sees it as below

```
&oldQuote [0] = &newQuote [0]; /* invalid, same as above */
```

Additionally, you can just set an array value to a literal string value, its the same problem.

```
newQuote = "How bout those Cowboys?"; /* literal string is an address */
```

The answer is that C provides the strcpy function. It will take a String in the second argument (s2), and copy each character into the first argument (s1) until it encounters the null terminator character. Let's go over a few examples of its usage with this example which can also be found at: <http://ideone.com/OvaT9l>

```
#include <stdio.h>
#include <string.h>

int main ()
{
```

```

char s1 [12] = { "String" };
char s2 [7];

printf ("S1: %s \n", s1);

/* contents of s1 copied into s2, and null */
/* terminator will be added as well into s2 */

strcpy (s2, s1);
printf ("S2: %s \n", s2);

/* "New String" will be copied into s1 */

strcpy (s1, "New String");
printf ("S1: %s \n", s1);

return (0);
}

```

Output:

```

S1: String
S2: String
S1: New String

```

S1 would end up as:

'N'	'e'	'w'	' '	'S'	't'	'r'	'i'	'n'	'g'	'\0'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

S2 would end up as:

'S'	't'	'r'	'i'	'n'	'g'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]	[6]

The strcat function

The **strcat** function takes **two arguments**, both character pointers. These must point to **null-terminated** arrays.

```
char * strcat ( char s1 [ ], char s2 [ ] )
```

This function takes the character string pointed to by the second argument (s2) and copies it to the end of character string pointed by the first argument (s1). It also puts a **terminating null character** at the end of the destination array. The **return type** of **char *** is something we will learn about more in detail next week, but it will essentially return a character pointer to the beginning of the string s1.

Warning: Make sure that there is enough space in the destination character array (in this case, s1) to accommodate the string to be copied.

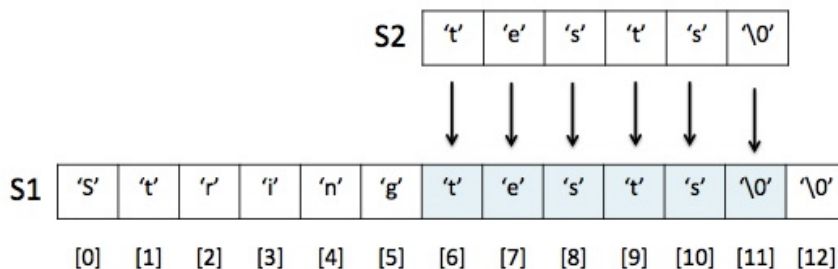
In the sample program below, a character array named S1 is concatenated with a string contained in another array S2. You don't have to call them S1 and S2, they can be any valid C variable name. The array S1 is initialized with the string that contains the word "String". Even though S1 has a size of 13 elements, the first string found in S1 is shown in blue below with the rest of the array elements initialized to the null terminator character. That array S1 is where the two strings will be concatenated using the strcat function.

S1	'S'	't'	'r'	'i'	'n'	'g'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'
----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------

When the **strcat** function is run

```
strcat (S1, S2);
```

... it is essentially performing the following operation:



The complete program (which can also be found at: <http://ideone.com/4lWK4r>) that shows how to call **strcat** to perform the above operation is as follows:

Source:

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char S1 [13] = {"String"}, S2 [ ] = {"tests"};

    printf ("S1: %s \n", s1);
    printf ("S2: %s \n", s2);

    strcat (S1, S2);

    printf("S1: %s \n", S1);
}
```

```
        return(0);  
    }
```

Output:

```
S1: String  
S2: tests  
S1: Stringtests
```

NOTE:

You can print the concatenated string if you wanted since the function returns a pointer to the beginning of S1.

```
printf ( "%s", strcat ( S1, S2) );    /* S2 copied to S1, then prints Stringtests */
```

Alternatively, you can also put the **result** of the **strcat** function (the concatenated string) into yet another string. Below is an example of copying the result into a new string (S3) using the **strcpy** function. Just make sure that S3 is big enough to store that string.

```
strcpy ( S3, strcat ( S1, S2) );    /* strcat done first, then strcpy */
```

Use can also use a **literal string** as your source string, it does not have to be in a named array, such as S2.

```
strcat ( S1, " tests" );    /* this is OK */
```

... but you can't do it the other way, as there is no defined memory area to put the string contents of S1 into ...

```
strcat ( "tests", S1 );    /* this is NOT OK */
```

The strcmp function

The **strcmp** function takes two arguments, both character pointers. These must point to null-terminated arrays.

```
int strcmp ( char s1 [ ], char s2 [ ] )
```

This function will return the following based on the comparison of the two strings:

- 0 -> if the strings are equal
- negative number -> if the first string is less than the second
- positive number -> if the first string is greater than the second

```
if ( !strcmp (s1, s2) )  
    printf ("strings are equal \n");
```

or

```
if ( strcmp (s1, s2) == 0 )  
    printf ("strings are equal \n");
```

or

```
if ( strcmp (s1, s2) )  
    ; /* do nothing */  
else  
{  
    /* code for equal case */  
}
```

PITFALL

Thus, a **strcmp** on equal strings results in a return of **zero**, which is FALSE. You can use the negation to handle this, or an *if else* with null statements for the TRUE case.

Note that in these two arrays, the array size may be different, but the strings contained within the array are the same. They will also print the same value: **String**

Anything after the null terminator in the first string s1 is garbage and not part the the string

S1:

'S'	't'	'r'	'i'	'n'	'g'	'\0'	'e'	's'	't'	's'	'\0'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

S2:

'S'	't'	'r'	'i'	'n'	'g'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]	[6]

null strings

Given the following statement:

```
char word = { "" }; /* it is really just one character: '\0' */
```

Note: There is no space between the quotes, the length is **zero**. The string contains only the **null terminator character**. Setting the first array element in a character array can effectively be used to clear and/or initialize strings.

Given:

```
char s2 [ ] = { "tests" };
```

't'	'e'	's'	't'	's'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]

Setting the first character to null, as in:

```
s2 [ 0 ] = '\0';
```

Would effectively set the string in the array to the **null string**. If printed, it would print nothing. If we run **strlen** on it, it will return zero. Remember, a string is a series of consecutive characters that ends with the null terminator character.

I've seen some programs define a constant for the NULL Terminator that is used to indicate the "End of String", such as:

```
#define EOS '\0'
```

Then use it later to initialize a string

```
s2 [ 0 ] = EOS;
```

... or terminate a string (i.e., indicate the end of a string)

```
s2 [ 5 ] = EOS;
```

Setting the first element of an array to the null terminator would effectively set the string in the array to the **null string**. If printed, it would print nothing. If we run **strlen** on it, it will return zero. Remember, a string is a series of consecutive characters that ends with the null terminator character.

If the first character encountered happens to be a null terminator character, then that will simply be the "string", and in this case, we'll refer to it as the **null string**. This is sometimes referred to as an "empty string".

'\0'	'e'	's'	't'	's'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]

The include statement

Throughout this class we have been including the standard input and output header file that contains most of the types, constants, and function prototypes supported by the C Language:

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

In this statement, the # must be the first character on the line. It directs the preprocessor to handle the statement.

< > tells the compiler where to find the file stdio.h

... in most cases, this is defined when you install the compiler on your system. On a UNIX system, look in `/usr/include` ... on your PC, if you installed Visual C++, it will be installed under a directory structure where it was installed on your computer. If you happen to go there, you'll see most of the include files that end with a *.h extension. Feel free to look them over, especially strings.h (or string.h) and stdio.h.

You can also use double quotes to explicitly tell the compiler where to look:

```
#include "tim.h"                /* look in current directory */
#include "/local/tools/headers/tool.h" /* specific directory */
```

Most of the functions mentioned up to this point require the standard input and output header file (**stdio.h**). This week we saw how we needed to include the **ctype.h** header file for character library functions and the **string.h** header file for string library functions. As you get more comfortable with C Programming, you will likely be creating your own header files to store your project's specific constants, types, and function prototypes. This becomes really important when you break your functions into separate files, and compile and link them all together to create production grade programs (see Chapter 15 for details). Using the include statement causes the **preprocessor** to place some code in the program before compilation. We will take a more detailed look at the preprocessor and include statements in Chapter 13.

Inputting Character Strings

Given the following statements:

```
char string1 [ 12 ];  
char firstname [ 20 ];  
char lastname [ 20 ];  
  
scanf ("%s", string1);  
scanf ("%s %s", firstname, lastname);
```

In the first scanf example, all character input up to the first blank, tab or end of line (carriage return) character goes into the string1 character array. The important thing to remember however is that this array can only handle 12 characters.

The second scanf statement will read in two character string values. If there were three words in the input buffer, the first two words go into the first_name and last_name character arrays, and the third word would not be read until the next scanf.

No & character is used in front of the variable names since the & means "address of" and the array name is already translated by the compiler as an address.

Inputting a Combination of Character Strings and Characters

What if you wanted write a program that would read in a first name, a middle initial, and a last name.

For example: **Connie S Cobol**

We could store the items into two strings and a character, as in:

```
char firstname [20];  
char middle;  
char lastname [20];
```

The first name and last name are easy as we just did that in our previous discussion with a scanf statement. When you are reading in character strings along with single characters using scanf statements, there is one trick. Instead of reading in a character as:

```
scanf ("%c", middle); /* read in the middle initial */
```

Add a space before the %c format to consume any previous white space.

```
scanf (" %c", middle); /* note the space before the %c */
```

Please review the code below and try it out at: <https://ideone.com/1z27pF>

```
#include <stdio.h>
int main ( )
{
    char firstname[20];    /* first name */
    char middle;           /* middle initial */
    char lastname[20];     /* last name */

    /* read in first, middle, and last name */
    printf ("\nEnter first name: ");
    scanf ("%s", firstname);

    printf ("\nEnter middle initial: ");
    scanf (" %c", &middle); /* note the space before the %c */

    printf ("\nEnter last name: ");
    scanf ("%s", lastname);

    /* print out the name to the screen */
    printf ("\n\nThe name read is:  %s %c %s \n", firstname, middle, lastname);

    return 0;
}
```

The output would be:

Outputting Character Strings

There are few tricks to output character strings. With formatting, you can decide how many characters in a string get displayed, how its padded with blanks, and whether the output is left or right justified. Review and try out the code below to give you some ideas.

Homework Tip: I feel the best method in this example is the last one, that prints and uses all twenty characters in the name array and left justifies it (`%-20.20s`). Using that format on your homework this week would help you line up your output better when printing the rest of the employee information (clock, wage, hours, overtime, and gross) to ensure everything else lines up nicely in a row and column format. Just like using min and max for floats (e.g., `%5.2f`, `%5.1f`, `%8.2f`) helps line up your floating point output (better than using just `%f`), so does using min and max values for printing out string output (i.e., `%-20.20s` is better than `%s` in representing string items in a tabular format).

Try it out at: <https://ideone.com/8GOG58>

```
#include <stdio.h>

#define SIZE 3
#define NAMESIZE 20

struct president
{
    char name [NAMESIZE];
};

int main ()
{
    int i; /* loop index */

    /* These are my three favorite presidents */
    struct president myPresidents [SIZE] = {
        {"Ronald Reagan"},
        {"George Washington"},
        {"Thomas Jefferson"}
    }
```

```

};

printf ("\nPrint whatever size the names are, Left Justified \n");
for (i = 0; i < SIZE; ++i )
{
    printf ("%s\n",
        myPresidents[i].name);
}

printf ("\n\nTake up a minimum of 20 spaces, Right Justified \n");
for (i = 0; i < SIZE; ++i )
{
    printf ("%20s\n",
        myPresidents[i].name);
}

printf ("\n\nTake up a minimum of 10 spaces, pad as needed, Left Justified \n");
printf ("... if name is longer, it will still print all characters\n");
for (i = 0; i < SIZE; ++i )
{
    printf ("% -10s\n",
        myPresidents[i].name);
}

printf ("\n\nPrint a maximum of 15 characters, pad if needed, Left Justified,");
printf ("\n... this is good if you need to limit a field size being displayed
\n");
for (i = 0; i < SIZE; ++i )
{
    printf ("% -15.15s\n",
        myPresidents[i].name);
}

printf ("\n\nPrint 20 characters, pad with spaces if needed, Left Justified,");
printf ("\n... this is good if you need to display all characters, \n");
printf ("\n... and really useful in lining up items in a table format. \n");
for (i = 0; i < SIZE; ++i )
{
    printf ("% -20.20s\n",

```

```
        myPresidents[i].name);  
    }  
    return (0);  
}
```

This would be the output:

Print whatever size the names are, Left Justified

Ronald Reagan
George Washington
Thomas Jefferson

Take up a minimum of 20 spaces, Right Justified

Ronald Reagan
George Washington
Thomas Jefferson

Take up a minimum of 10 spaces, pad as needed, Left Justified

... if name is longer, it will still print all characters
Ronald Reagan
George Washington
Thomas Jefferson

Print a maximum of 15 characters, pad if needed, Left Justified,

... this is good if you need to limit a field size being displayed
Ronald Reagan
George Washingt
Thomas Jefferso

Print 20 characters, pad with spaces if needed, Left Justified,

... this is good if you need to display all characters,

... and really useful in lining up items in a tabular format.

Ronald Reagan

George Washington

Thomas Jefferson

A Musical Example - "We can be Heroes, just for one day"

My high school and college years with music were in the seventies and early eighties, and yes, I did grow up in the **MTV** generation when music videos were first shown on television. For every Van Halen and Bruce Springsteen concert, you could also find me watching punk rockers like the Ramones, Billy Idol, and the Talking Heads. The Ramones were one of our favorite local bands out of NYC, proving that just about anyone could be in a rock band if they really wanted to be.

I put together a quick example of a C program working with large strings using lyrics from one of my favorite musicians, **David Bowie**. The song I present for your entertainment and review below is one of my favorite songs: "**Heroes**". Note how I initialized the first verse in a symbolic constant using one string and a series of *continuation characters*. In the second verse, I just mashed things together into a variable to limit how many physical lines were needed in the code. Note that both of my strings contain many **new line characters** to get the printing effect I wanted, and more importantly, they are automatically **NULL terminated** (by the string initialization) so the string stops printing.

You can try it out at: <https://ideone.com/fstoH7>

```
// Lyrics from David Bowie's Heroes
#include <stdio.h>

// You can store many sentences in one string, use
// the continuation character (the "\") after each line. You
// can add new lines and other special characters as needed
// to get the printing effect you need.

#define HEROES_VERSE1 "\nI, I wish you could swim \
\nLike the dolphins, like dolphins can swim \
\nThough nothing, \
\nnothing will keep us together \
\nWe can beat them, for ever and ever \
\nOh we can be Heroes, \
\njust for one day"
```

```

int main ()
{
    // You can store a quote in a variable as well.
    // It does not matter if you put each sentence on a new line,
    // the C program will just print characters until the null
    // terminator is encountered

    char heroes_verse2 [ ] = {"\nI, I will be king\nAnd you, you \
will be queen\nThough nothing will drive them away \nWe can \
be Heroes, just for one day\nWe can be us, just for one day"};

    char song [5000]; /* store a song */

    printf ("\nThe first two verses of David Bowie's Heroes:\n");
    printf ("%s", HEROES_VERSE1);
    printf ("\n");
    printf ("%s", heroes_verse2);

    /* Let store the two verses in one variable */
    strcpy (song, HEROES_VERSE1); /* input is a constant */
    strcat (song, "\n");          /* input is a new line character */
    strcat (song, heroes_verse2); /* input is a variable */

    /* print the song */
    printf ("\n\n\n*** David Bowie's Heroes ***\n");
    printf ("%s", song);

    /* print number of characters in the song */
    printf ("\n\nNumber of characters in the the song is: %i \n",
            strlen (song));

    return (0);
}

```

The output from the program above is shown below. Note that it came out exactly how I wanted it presented, and perfect if you had to use it to sing along at your next karaoke party :)

The first two verses of David Bowie's Heroes:

I, I wish you could swim
Like the dolphins, like dolphins can swim
Though nothing,
nothing will keep us together
We can beat them, for ever and ever
Oh we can be Heroes,
just for one day

I, I will be king
And you, you will be queen
Though nothing will drive them away
We can be Heroes, just for one day
We can be us, just for one day

*** David Bowie's Heroes ***

I, I wish you could swim
Like the dolphins, like dolphins can swim
Though nothing,
nothing will keep us together
We can beat them, for ever and ever
Oh we can be Heroes,
just for one day

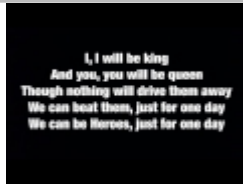
I, I will be king
And you, you will be queen
Though nothing will drive them away
We can be Heroes, just for one day
We can be us, just for one day

Number of characters in the the song is: 342

If you have some time, feel free to create a program that will print out your favorite song. Doesn't have to be the whole song, but a few verses of the song lyrics will do. Sadly, Dave Bowie passed away in 2016 which was a real shock to rock world. I might be showing my age, but I really miss those "Classic Rock" days of the 70's and 80's, with vinyl records and all :) Below is a version of one of his songs called "Heroes" on YouTube™ which always seems to pick me up whenever I am feeling down. As one commenter named Simon Pollard on the video posted:

"The world is 4.543 billion years old and we have somehow managed to exist at the same time as David Bowie."

Well said my friend ... rest in peace David Bowie!



[Watch Video](#)

David Bowie - Heroes Lyrics

User: n/a - Added: 1/13/16

To Static or Not to Static

In your coding travels, you may notice the word "**static**" is sometimes used to initialize a variable (simple, array, ...) when declaring it. In older versions of C, there was a time that you had to use "**static**" to initialize a variable when declaring it ... that is no longer so in the most recent versions of C. You can do any of the following below, just realize you can not have a **local variable** and a **static variable** with the same variable name within a given function (you will get a compilation error).

```
char letter = 'A'; /* or */
static char letter = 'A';
```

```
int number = 2; /* or */
static int digit = 2;
```

```
char letters [3] = {'A', 'B', 'C'}; /* or */
static char letters [3] = {'A', 'B', 'C'};
```

```
int numbers [3] = {1,2,3}; /* or */
static int numbers [3] = {1,2,3};
```

As for the differences between a local variable, a static variable will **initialize to 0** by default and will **maintain its value in memory** the next time a function is called as the value is stored in the **data area** instead of the **local function stack frame**.

See example below and/or try it out at: <https://ideone.com/RKHOM5>

```
#include <stdio.h>
#define SIZE 3

int main (void) {

    int i; /* loop and array index */
    int numbers [SIZE] = {1,2,3}; /* a few integers in an array */
    char letters [SIZE] = {'A', 'B', 'C'}; /* a few characters in an array */

    static int s_numbers [SIZE] = {1,2,3}; /* a few integers in an array stored in
a static variable */
    static char s_letters [SIZE] = {'A', 'B', 'C'}; /* a few characters in an array stored
in a static variable */

    /* these are the local variable address on the stack */
    printf ("\nAddresses on the local stack frame: ");
    printf ("\nnumbers: %x, letters %x", &numbers[0], &letters[0]);
```

```

    /* static variables are stored in the data area */
    printf ("\n\nAddresses of static variables in the data area: ");
    printf ("\ns_numbers: %x, s_letters %x\n", &s_numbers[0], &s_letters[0]);

    /* print all the array element values */
    printf ("\n\nShowing element contents of each array: ");
    for (i = 0; i < SIZE; ++i)
    {
        printf ("\n %i, %c, %i, %c",
                numbers[i], letters[i],
                s_numbers[i], s_letters[i]);
    } /* for */

    return 0;
}

```

It would print the following ... note the static addresses of those variables are much higher numbers showing they are stored in the data area ... instead of the other two in the local function stack frame (for main).

Addresses on the local stack frame:
 numbers: **108c9a78**, letters **108c9a85**



Addresses of static variables in the data area:
 s_numbers: **8a80a018**, s_letters **8a80a010**



Showing element contents of each array:
 1, A, 1, A
 2, B, 2, B
 3, C, 3, C

Week 7 C Program Source Files

We covered many different programs with strings, functions and/or structures this week. Some of our programs contained just a main function, while others implemented a few other functions as well. In the real world, most C programs are implemented using multiple files where generally each function is stored in its own C source file. This helps to better isolate issues for debugging as well as speed up the compilation process, as you only need to recompile functions/files that have changed. In programming classes you take going forward, you may very well be creating programs with multiple source files.

I've converted some of the programs we reviewed in the lecture notes this week to work with multiple source files. To access the files, download the folders stored in the zip file available back in our current week's notes. You will find the item right near the end of the list of lecture notes near the Quiz area, and it looks like this:

**Week 7 C Program Source Files** 

Attached Files:  Week 7 C Source Files.zip  (11.397 KB)

If you want to better understand the compilation and build process for C programs, whether being a single file or for multiple files, download the folders in the attached zip file and start by reading the README.txt file for orientation and directions. These folders and files parallel the discussion in the Challenge Problem Solved (printing names), String Input and Output (President Names), and Rock Song Example. It greatly expands the scope of the latter two items, so it is worth a look. You can view all the files in the folders with a simple text editor or within your native compiler's Integrated Development Environment (IDE).

Most folders have the following file which you should read first:

- **README.txt** - general information on how to compile, build, and use the template files

Use the **Makefile** provided if you wish to compile and build your program from the command line, such as on a UNIX or LINUX system. **Alternatively**, you can load files into your **native compiler's** Integrated Development Environment (IDE) and have it build and run from there. In every case, the files will compile and build correctly out of the box. You are welcome to expand upon them by updating files as needed and adding new C source files for any additional functions you plan to design into it.

At the very least, feel free to just download the zip file and associated folders and take a look at all the files within your favorite text editor.

Summary

There are few things that are important for you to remember this week. As a starting point, make sure you include the right **header files** in your program if you use the C Library character and string functions.

```
#include <ctype.h> /* for character library functions */
#include <string.h> /* for string library functions, strings.h also works */
```

Here are **five common errors** new C programmers make with using strings and characters.

1) Not using single quotes to initialize character variables

```
char value; /* simple variable of type char */
```

These are valid ways to initialize a character variable

```
value = 'a'; /* a single character ... a letter */
value = '\n'; /* special character, its one character in the ASCII set */
value = '?'; /* any character from your keyboard can be represented */
```

These are common errors to watch out for:

```
value = 'ab'; /* BAD ... two characters ... can only be one character max */
value = "abc"; /* value can only store one character, not a 4 char string */
```

2) Not using double quotes for Character Strings

```
char phrase [ ] = { "Hello" }; /* OK, initialize in array declaration */
```

This is a common error to watch out for:

```
char phrase [ ] = { 'Hello' }; /* BAD, use double quotes, not single quotes */
```

3) Setting a string to variable with the assignment (=) operator

The only time you can use the assignment operator with a string is to initialize it to a variable when its being declared:

```
char phrase [ ] = { "Hello" } ; /* this is OK, we did this above already */
```

However, later on in code, you can't simply assign it like this:

```
phrase = "Hello"; /* can't set one address equal to another address */
```

Instead, you need to use the strcpy function:

```
strcpy (phrase, "Hello");
```

4) Comparing one string to another with the == operator

```
if (phrase == "Hello") /* Invalid, need to use the strcmp function */
```

```
Try: if ( strcmp (phrase, "Hello") == 0 )
```

5) Mixing up the return value for the strcmp function

```
if (strcmp (phrase, "Hello"))
    printf ("Strings are equal \n"); /* return value logic is backwards */
```

A common error above with using the strcmp function is not checking a return value of zero and assuming a TRUE (non-zero) value means they are equal. It is just the opposite, as a FALSE value indicates there are ZERO differences. Of course, if you do want to use the statement above, just reverse the logic if you don't want to check for zero.

```
if (strcmp (phrase, "Hello"))  
    printf ("Strings are NOT equal \n");
```

For Next Week

Next week we'll be covering the hardest topic in this class, which is **pointers**. Read the next chapter in the book and with my posted lecture notes that week, we'll solve the mystery of pointers together.

As always, there is a **Quiz** this week to test your knowledge of strings.

The **homework** this week should be fun. You'll need to add a character array member to the structure you used in the last homework to store the employee's name. Most everything else will still be the same for the most part, with the exception of adding functions to help calculate the total and average values of the numeric members. See the description of this week's homework assignment that spells out everything you need to do.

You can initialize your array of structures with the employee name, just like you have been initializing the clock number and wage value.

For those that want more of a challenge (and its optional), I added one to figure out the minimum and maximum hours, overtime, and gross values. You can either write functions for these (best), or just figure out and store the values in the main function (will work for me too).

HOMEWORK 6

CHARACTER STRINGS

Write a C program that will calculate the gross pay of a set of employees. Continue to build upon what you have already designed from previous homeworks: Functions, Structures, and Constants. I would highly suggest you just use your previous homework as your code template as much of it can be reused.

The program should prompt the user to enter the number of hours each employee worked. When prompted, key in the hours shown below.

The program determines the overtime hours (anything over 40 hours), the gross pay and then outputs a table in the following format.

Column alignment, leading zeros in Clock#, and zero suppression in float fields is important.

Continue to incorporate functions into your program design.

Use 1.5 as the overtime pay factor.

This week, adding a Total and Average row is no longer optional, its required for this assignment:

a) Add a **Total** row at the end to sum up the wage, hours, ot, and gross columns

b) Add an **Average** row to print out the average of the wage, hours, ot, and gross columns

Name	Clock#	Wage	Hours	OT	Gross
Connie Cobol	098401	10.60	51.0	11.0	598.90
Mary Apl	526488	9.75	42.5	2.5	426.56
Frank Fortran	765349	10.50	37.0	0.0	388.50
Jeff Ada	034645	12.25	45.0	5.0	581.88
Anton Pascal	127615	10.00	40.0	0.0	400.00
Total:		53.10	215.5	18.5	2395.84
Average:		10.62	43.1	3.7	479.17

You should implement this program using a structure similar to the suggested one below to store the information for each employee. Feel free to tweak it if you wish. For example, its OK to have a first and last name member instead of just a name member, and if you want to use different

types, that is OK as well.

```
struct employee
{
    char name [20];
    long id_number;
    float wage;
    float hours;
    float overtime;
    float gross;
};
```

Use the following information to initialize your data.

Connie Cobol	98401	10.60
Mary Apl	526488	9.75
Frank Fortran	765349	10.50
Jeff Ada	34645	12.25
Anton Pascal	127615	10.00

Create an array of structures with 5 elements, each being of type struct employee.

Initialize the array with the data provided and reference the elements of the array with the appropriate subscripts.

Intermediate Optional Challenge

Add two more rows to calculate the minimum and maximum values:

- Calculate and print the **minimum** wage, hours, ot, and gross values
- Calculate and print the **maximum** wage, hours, ot, and gross values

Name	Clock#	Wage	Hours	OT	Gross
Connie Cobol	098401	10.60	51.0	11.0	598.90
Mary Apl	526488	9.75	42.5	2.5	426.56
Frank Fortran	765349	10.50	37.0	0.0	388.50
Jeff Ada	034645	12.25	45.0	5.0	581.88
Anton Pascal	127615	10.00	40.0	0.0	400.00
Total:		53.10	215.5	18.5	2395.84
Average:		10.62	43.1	3.7	479.17
Minimum:		9.75	37.0	0.0	388.50
Maximum:		12.25	51.0	11.0	598.90

Advanced Optional Challenge

For those of you that want the ultimate challenge this week, do the intermediate challenge and add the following advanced challenge below:

Define a structure type called struct name to store the first name, middle initial, and last name of each employee instead of storing it all in one string. Your *employee structure* will now look something like this:

```
struct employee
{
    struct name employee_name;
    long id_number;
    float wage;
    float hours;
    float overtime;
    float gross;
};
```

You are welcome to add additional members to your employee structure type, such as those suggested in homework assignment 5. Do not initialize your array of structures, instead prompt for the following information as input to your program:

```
Connie J Cobol      98401      10.60
Mary P Apl          526488      9.75
Frank K Fortran     765349     10.50
Jeff B Ada          34645      12.25
Anton T Pascal      127615     10.00
```

When printing, print just like you did with the intermediate challenge, but display employee names in the following format:

<Last Name>, <First Name> <Middle Initial>

Your output would now look like this:

```
-----
Name                Clock#    Wage    Hours    OT      Gross
-----
Cobol, Connie J    098401    10.60    51.0     11.0     598.90
Apl, Mary P        526488     9.75    42.5      2.5     426.56
Fortran, Frank K   765349    10.50    37.0      0.0     388.50
Ada, Jeff B        034645    12.25    45.0      5.0     581.88
Pascal, Anton T   127615    10.00    40.0      0.0     400.00
-----
```


Total:	53.10	215.5	18.5	2395.84
Average:	10.62	43.1	3.7	479.17
Minimum:	9.75	37.0	0.0	388.50
Maximum:	12.25	51.0	11.0	598.90

Additional Optional Challenge

If you want an additional challenge, a code template has also been provided that starts you in the right direction if you want to implement the homework using separate C source files for each function as well as including a header file as needed. You can load the template files provided into your native compiler and Integrated Development Environment (IDE) to get started.

Tips for Assignment 6

The hardest part of this assignment is figuring out how to calculate and print the sum total and average values of wage, hours, overtime, and gross. Here are few tips to get you started ... both methods below are acceptable.

1) Easiest way

In your print function, just add *local variables* for total wage, total hours, total overtime, and total gross ... and keep a *running total* as you print each employee ... at the end of the loop after you have printed all employees, you will have all three totals calculated. Then you can print the totals, calculate the average (now that you know the four totals and number of employees you processed), and print the average after the loop ... I sketched out a design below:

Initialize the three local variables for holding the totals to 0

```
for (i=0; i < size; ++i) /* any loop is fine */
{

    /* print each employee ... just like we did in the last assignment, add the name member now as well and
    keep a running total of four local variables */

    total_wage += emp[i].wage;
    total_hours += emp[i].hours;
    total_ot += emp[i].ot;
    total_gross += emp[i].gross;

}

/* Print the final values in the four local variables holding the totals */

/* Use the four local variables with the totals to calculate and print the average of hours, overtime, and gross
*/
```

If you decide to do the optional challenge with min and max values, you could also set up local variables for min wage, max wage, min hours, max hours, min ot, max ot, min gross and max gross, and check within the loop so at the end, the right values are in each variable to print.

2) Better Way - Functions

Create a single function that can be called or a series of functions ... you'll need something to pass and/or return from your function(s). Your initial challenge is how do I pass and return three totals? The answer is that you can pass and return a structure with functions ... and you will have access to all the members of that structure. I would suggest the following structure type:

```
struct stats
{

    float total_wage;
    float total_hours;
    float total_ot;
    float total_gross;

};
```

... in a calling function (like main), you could initialize the structure:

```
struct stats my_totals = {0, 0, 0, 0};
```

... then pass it as needed along with your emp structure array to calculate totals for each employee. You could also return it if you wish. There are lots of different ways to do this. Figure out what way works best for your program ... don't just pick my suggested way if you want to go a different direction. There are lots of ways to design your functions and I'll be open to all of them.

```
struct stats print_employees (struct employee emp [ ], struct stats my_totals, int size);
```

For those interested in the challenge, you could add all kinds of members to your stats structure ... totals, min, max, average ... then everything is all in one place. Easy to pass, update, and return between functions. I've also seen some students in the past create four separate structures ... one to hold just the totals, another for averages, a third for min values, and a fourth for max values. As I said before, there are many ways to do this assignment.

```
struct stats
{
    /* totals */
    float total_wage;
    float total_hours;
    float total_ot;
    float total_gross;
    /* averages */
    float avg_wage;
    float avg_hours;
    float avg_ot;
    float avg_gross;
    /* If doing the optional challenge, add the min and max values */
};
```

Assignment 6 Code Templates

If you are happy with the way your homework (Assignment 5 - Structures) turned out from last week, I would highly recommend that you just start with that program as your template for this week. Simply add the employee name to your structure and then work that name into any functions that need it. See the "helpful hints" file that was included with the strings assignment this week for how to tackle the challenge of adding sums and averages to your assignment.

If you want to start from scratch, I've provided two templates that you are welcome to use for this week's assignment. In fact, they are basically the same templates I provided last week. One template is passing the *address* of a single array of structures along with the array size to each function to perform a specific set of tasks.

An *alternate approach* is to pass individual structure member values within each array element of your array of structures. In this case, you pass the *values* you need to each function and have that function return a result, such as a the gross pay or overtime of a particular employee.

Whatever way you do it, **don't use global variables** ... I don't want all your functions looking like: `void foo ()` with no parameters passed or used. There are benefits and drawbacks to each approach. Good luck with these templates, or feel free to create your own from scratch. Of course, if you have lots of time on your hands, try them both :)

Option 1: Call by Reference Code Template

There are many ways to do the assignment, below are just some ideas you are welcome to use for your design. The **template** below provides you with a basic starting point that you can modify, and includes a complete function to print all the **array elements** and **members** in the **array of structures** ... which could prove useful as you add new functions and debug your code. The template should compile and run in whatever compiler you use (including IDEOne).

I always found that creating a function to display the data first is a good initial step. Then you can call it any time to check that your variables are initializing and updating correctly. Don't feel you have to create all your functions right way. Develop and test the ones you need one at a time. This **Divide and Conquer** approach will serve you well and you won't be worried about initially debugging lots of errors.

One thing you'll notice if you decide to do it this way, is that you can just pass a **single array of structures** to any function, and you will have access to any combination of the **array elements** (in this case, each employee) along with **members** inside that structure. This is an example of **Call by Reference** (i.e., *address*). Remember that in homework 4 (*functions*) you had to pass one or more arrays as needed to your functions. This can be quite a pain if you had lots of different attribute information about each employee (for example, if in the future I added their name, hire date, salary grade, ...).

Review the **template** below and feel free to access it at: <https://ideone.com/Gy7nMg>

```

/*****
**
**  HOMEWORK: #6 Structures and Strings (Call by Reference)
**
**  Name: [Enter your Name]
**
**  Class: C Programming
**
**  Date: [enter the date]
**
**  Description: This program prompts the user for the number of hours
**  worked for each employee. It then calculates gross pay
**  including overtime and displays the results in table. Functions
**  and structures are used.
**
*****/

/* Define and Includes */

```

```

#include <stdio.h>

/* Define Constants */
#define NUM_EMPL 5

/* Define a global structure to pass employee data between functions */
/* Note that the structure type is global, but you don't want a variable */
/* of that type to be global. Best to declare a variable of that type */
/* in a function like main or another function and pass as needed. */

struct employee
{
    long id_number;
    float wage;
    float hours;
    float overtime;
    float gross;
};

/* define prototypes here for each function except main */

void Output_Results_Screen (struct employee emp [ ], int size);

/* add your functions here */

/*****
** Function: Output_Results_Screen
**
** Purpose: Outputs to screen in a table format the following
** information about an employee: Name, Clock, Wage,
** Hours, Overtime, and Gross Pay.
**
** Parameters:
**
**     employeeData - an array of structures containing
**     employee information
**     size - number of employees to process
**
** Returns: Nothing (void)
**
*****/

void Output_Results_Screen ( struct employee employeeData[], int size )
{
    int i; /* loop index */

    /* print information about each employee */
    for (i = 0; i < size ; ++i)
    {
        printf(" %06li %5.2f %4.1f %4.1f %8.2f \n",
            employeeData[i].id_number, employeeData[i].wage,
            employeeData[i].hours,
            employeeData[i].overtime, employeeData[i].gross);
    } /* for */
} /* Output_results_screen */

int main ()
{
    /* Set up a local variable and initialize the clock and wages of my
    employees */
    struct employee employeeData[NUM_EMPL] = {
        { 98401, 10.60 },
        { 526488, 9.75 },

```

```

        { 765349, 10.50 },
        { 34645, 12.25 },
        { 127615, 8.35 }
    };

    /* Call various functions needed to read and calculate info */

    /* Print the column headers */

    /* Function call to output results to the screen in table format. */
    Output_Results_Screen (employeeData, NUM_EMPL);

    return(0); /* success */
} /* main */

```

Option 2: Combination of Call by Value and Call by Reference Code Template

Another way to do this assignment is to **pass member values** of specific **elements** into the **array of structures** from a function like main to **other functions** that will return a **value** in the **result**. For example, passing the unique clock number for an employee to a function and returning back the hours they worked in a week, and having this function call in a loop that will process each employee. If you do it this way, you have to pass the right information about the employee as needed by each function, i.e., expect some functions to have **multiple parameters** (for example, to figure out gross pay, you'll need to pass in at least wage, hours, and overtime).

Review the **template** below and feel free to access it at: <https://ideone.com/HAirQ2>

```

/*****
**
** HOMEWORK: #6 Structures and Strings (Call by Value and Reference)
**
** Name: [Enter your Name]
**
** Class: C Programming
**
** Date: [enter the date]
**
** Description: This program prompts the user for the number of hours
** worked for each employee. It then calculates gross pay
** including overtime and displays the results in table. Functions
** and structures are used.
**
*****/

/*Define and Includes */

#include <stdio.h>

/* Define Constants */
#define NUM_EMPL 5

/* Define a global structure to pass employee data between functions */
/* Note that the structure type is global, but you don't want a variable */
/* of that type to be global. Best to declare a variable of that type */
/* in a function like main or another function and pass as needed. */

struct employee
{
    long int id_number;
    float wage;
    float hours;
    float overtime;
}

```

```

    float gross;
};

/* define prototypes here for each function except main */

float Get_Hours (long int id_number);
void Output_Results_Screen (struct employee emp [ ], int size);

/* Add your functions here */

/* Add Function Comment Block Header for Get_Hours */
float Get_Hours (long int id_number)
{
    float hours = 0; /* hours employee worked in a given week */

    /* add code to prompt for and read in employee hours */

    return (hours);
}

/*****
** Function: Output_Results_Screen
**
** Purpose: Outputs to screen in a table format the following
** information about an employee: Name, Clock, Wage,
** Hours, Overtime, and Gross Pay.
**
** Parameters:
**
**     employeeData - an array of structures containing
**     employee information
**     size - number of employees to process
**
** Returns: Nothing (void)
**
*****/

void Output_Results_Screen ( struct employee employeeData[], int size )
{
    int i; /* loop index */

    /* print information about each employee */
    for (i = 0; i < size ; ++i)
    {
        printf(" %06li %5.2f %4.1f %4.1f %8.2f \n",
            employeeData[i].id_number, employeeData[i].wage,
            employeeData[i].hours,
            employeeData[i].overtime, employeeData[i].gross);
    } /* for */
} /* Output_results_screen */

int main ()
{
    /* Set up a local variable to store the employee information */
    struct employee employeeData[NUM_EMPL] = {
        { 98401, 10.60 },
        { 526488, 9.75 },
        { 765349, 10.50 },
        { 34645, 12.25 },
        { 127615, 8.35 }
    };

```

```
int i; /* loop and array index */

/* Call functions as needed to read and calculate information */
for (i = 0; i < NUM_EMPL; ++i)
{
    /* Prompt for the number of hours worked by the employee */
    employeeData[i].hours = Get_Hours (employeeData[i].id_number);

    /* Add other function calls as needed to calculate overtime and gross */

} /* for */

/* Print the column headers */

/* Function call to output results to the screen in table format. */
Output_Results_Screen (employeeData, NUM_EMPL);

return(0); /* success */

} /* main */
```


Homework 6 - Multi File Template Option

Take the optional challenge this week and implement Homework Assignment 6 using multiple files.

To access the files, download the folder stored in the zip file available back our current week's notes. You will find the item right near the end of the list of lecture notes, and it looks like this:



Assignment 6 - Multi File Code Template Option

Attached Files:  Hmwk 6 Multi File Template.zip (13.509 KB)

See attached zip file for options on how to do this homework with multiple files that you can link together to create and run your program. It includes the same options as the standard homework code templates provided this week. It is an alternative to just putting everything in one file. Download the zip file and start by reading the README.txt file first for instructions and guidance. Note that multiple files will not work with IDEOne if you are using that as your compiler.

It contains two folders, one for a design using *Call by Value*, and the other with a design using *Call by Reference*. The same set of files are tailored and implemented into each folder based on the design:

- **employees.h** - header file with common constants, types, and prototypes
- **main.c** - the main function to start the program
- **getHours.c** - a function that will read in the number of hours an employee worked
- **makefile** - a file you can use if you want to compile and build from the command line
- **printEmp.c** - a function that will print out the current items in our array of structures
- **README.txt** - read this first! ... general information on how to compile, build, and use the template files

Use the **makefile** provided if you wish to compile and build your program from the command line, such as on a UNIX or LINUX system. **Alternatively**, you can load the *employee.h* file and the two source files (*main.c* and *print_list.c*) into your **native compiler** Integrated Development Environment (IDE) and have it build the template from there. In either case, the files will compile and build correctly out of the box. Your job will be to expand upon them by updating files as needed and adding new C source files for any additional functions you plan to design into it.

Of course, even if you decide to just implement your homework 6 assignment within a single file using the other homework code template(s) provided, feel free to just download the Multi File Code Template Option folder and take a look at all the files within your favorite text editor.