Welcome to Week 8!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 8 folder**.
- Read **Chapter 10** in the latest edition of the textbook (chapter 11 in earlier editions).
- Begin **Quiz 8**. It is due Sunday at midnight.
- Begin **Assignment 7**. 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 8 Discussion Forums** located in the course **Discussion Board**. Please ask as many questions as needed, and don't hesitate to answer one-another's questions, but don't reveal an answer to the final exam.
- Try out the various **code examples** in the lecture notes. Feel free to modify them and conduct your own **"What ifs"**.

"Humans ... I will never understand your fascination with array references. Replace them with pointers and you will achieve the performance you need to achieve Warp Factor 7."

--- Mr. Speck
First Science Officer, USS Enterprise

Pointers

This week we'll learn about one of C's most powerful features, **pointers**. We'll start out simple and then build upon our pointer knowledge to demonstrate advanced concepts. The videos and images in the notes this week will be very important, as being able to visualize what is happening in memory will go a long way in helping you understand exactly what is happening with each statement. This "visualization" is what I find lacking in many books and other classes.

While the chapter on pointers in the book will provide general introductory information, I really feel this week you will benefit more by concentrating on the lecture notes and videos provided, as well as TRYING OUT the sample code.

If you can remember one thing this week, remember that a **pointer** variable **contains** an **ADDRESS** !

Star Trekking - Pointing the Way to Speed

To introduce you to why pointers are so important, especially in terms of speed and execution, click the image below or go back to the lecture notes this week to access the one minute video below and watch how Science Officer Mr. Speck of the USS Enterprise helped Chief Engineer Mr. Scoot (a.k.a., "Scootty") with a performance problem he was having with his Warp Drive Program (that was of course written in C, since it is one of the fastest executing programming languages in the Galaxy).

Video - Introduction to Printers

Watch this video first to get a good introduction on pointers in general. You'll learn how to set up a pointer variable, how to initialize it, how to reference it, and how to integrate them into your program.

Video - Common Pointer Mistakes

Pointers are very powerful as they can provide you with access to specific memory locations on the computer. This can be both good and bad. Saying all that, by following my simple rules, you can avoid common errors.

Contents verses Addresses

Two new operators:

& (the address of)

... and

* (indirection)

Examples:

```
int total;                  /* simple integer  */
char myString [ 80 ];   /* character array */

scanf ( "%d", &total );       /* the address of total */

scanf ( "%s", myString );    /* same as saying & myString [ 0 ], its an address */
```

We have already used **&** and scanf. It said, "pass the address of total to scanf".

The & is not used for strings (char arrays) because strings and arrays are automatically referenced by the compiler as **addresses**.

Elements of arrays are **passed by value**, not by address, so you need the **&** to specify the address of an element.

```
scanf ( "%s", &myString [ 0 ] );
```

You just can't say:

```
scanf ( "%s", myString [ 0 ] );
```

Remember that the compiler sees the array name as the address of the first element. This might be the second most important thing to remember this week :)

myString is the same as & myString [0]

SUMMARY:

Given:

```
int total;  /* single integer variable */

char myString [ 20 ];  /* an array of twenty characters */
```

Code Sample	Compiler will see	Why?
total	The Contents	Not an Array
&total	An Address	Uses the & operator
myString	An Address	An Array Name
&myString [5]	An Address	Uses the & operator
myString [5]	The Contents	Array Element

Simple Pointer Example

A **pointer** is a variable that represents the location (rather than the value) of a data item, such as a variable or an array element. Pointers are used in programs to access memory and manipulate the contents of variables and other items using their addresses. There are two important things to remember about pointer variables:

1) The **contents** of a pointer variable contain an **address** in memory (a location), not a data value
2) Pointer variables are declared to point to a **specific type** (such as an integer, character, float, ...)

A good example of a program that shows how an integer pointer works is shown below. I am going to use simple integer values for addresses (like 1000, 1004, ...) in the lecture notes this week. However, in the real world, address values are very large. I would recommend displaying them in a **long hexadecimal format (*%lx*)** ... for more on this, read the notes at the end of this page. You can see that the IDEOne system is using a 32 bit (4 byte) addressing scheme based on the values it shows. Review how the program worked and what addressing values really existed at: http://ideone.com/4QUsaG

```
#include <stdio.h>
int main ( )
{
    int a = 10;         /* simple integer */

    int *int_ptr;       /* pointer to type integer */


   int_ptr = &a;     /* set pointer equal to the address of a */

   /* Actual addresses in memory are very large, just using %i will  */
   /* not always work, hexadecimal format (%lx) is good choice.       */

   printf ("%i %lx %lx %lx %i \n", a, &a, int_ptr, &int_ptr, *int_ptr);

   return (0);
```

```
}
```

Output

```
10 bffb9578 bffb9578 bffb957c 10
```

The important thing to understand is that there is **direct method** of accessing the variable a, such as:

```
printf ("a = %i \n", a);  /* direct method */
```

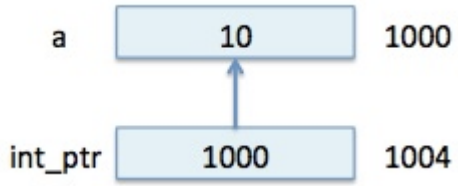
... and an **indirect method** this done through the dereference, or indirection operator, the *.

```
/* indirect method to access the variable a via a pointer */
printf ("Indirectly access with a pointer: %i", * int_ptr);
```

At the end of the program, the variables a and int_ptr would look like this in memory:

Name	Memory Location	Contents
a	1000	10
int_ptr	1004	1000

The pointer relationship with the variables a and int_ptr are more clearly illustrated as:



In reality, as shown in our program above, the computer will decide upon the actual memory location of each variable, but I'll just use 1000 for my examples since it is a nice easy number to work with. The space set aside these days on computers to hold actual addresses depends on the type of computer you are using ... some older ones are 16 bit (2 bytes), some are 32 bits (or 4 bytes), and many now are 64 bits (or 8 bytes). Remember from our initial set of lecture notes that 8 bits is equal to 1 byte of memory.

In my examples this week, I am going to assume that pointers will store address values at 32 bit (4 byte) memory locations. For example, with integers, let's assume it takes up 4 bytes of memory. That is why you see the variable "a" start at location 1000, and then the variable "int_ptr" is 4 bytes past it at location 1004.

What will print out from the code above?

Variable	Value
a	10
&a	1000
int_ptr	1000
&int_ptr	1004
*int_ptr	10

No separate pointer type exists. A pointer is identified by the type it points to.

Note:

If you are really printing address locations, you might want to use **hexadecimal**, as it can handle very large numbers ... a printf format of %x or %lx (the letter l for long) would work best. Hexadecimal works very well with memory locations and you'll find that many software developers like working with it better when printing address locations or setting breakpoint values during debugging. So, you could change the printf statement above to this to print the address locations as hexadecimal:

```
printf ("%i %lx %x %lx %i \n", a, &a, int_ptr, &int_ptr, *int_ptr);
```

Important Point:

If you are going to use the indirection operator on a pointer to get at the value it is pointing to, it is VERY IMPORTANT to make sure that the pointer has been set to an actual address value in your program, and that it is

NOT UNDEFINED. If you try to reference a pointer that does not have a valid address (or is 0 or some other value), you will likely have your program abort and dump **core**. It will like show up on the screen as a **Bus Error** or **Segmentation Error**, meaning you are accessing a protected area of memory. If you get one of these errors, your program (when run) will just halt on the statement where the error was encountered.

One other point, there is no guarantee that an address value used by a variable during one program run **will be the same** the next time the program is run. I can also guarantee the address value **will be different** if you run your program on another computer system.

Pointers and Characters

A pointer can be declared to point to any valid C variable type. Here is an example of a character pointer (a pointer that can point to a character location in memory). In the code below, **char_ptr** is declared to be a pointer that can point to a variable of type character.

Feel free to try it out at http://ideone.com/2q1B4J

```
#include <stdio.h>
int main ()
{
    char c = 'Q'; /* c is a simple char variable */

    char * char_ptr = &c; /* pointer to type char */

    printf (" %c %c \n ", c, *char_ptr);

    return (0);
}
```

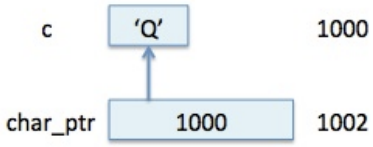
What will print out?

```
Q Q
```

It would be stored like this given that the variable **c** starts at memory location 1000. Note that since a character only takes up one byte of memory, I started the pointer variable, which will take up at least 4 bytes of memory to hold a memory address value, at the next **even** memory location, in this case 1002.

Name	Address Location	Contents
c	1000	'Q'
char_ptr	1002	1000

You can visualize it like this and note that a char only takes up one byte of memory, while an address on our computer takes up 4 bytes ... and addresses are what are stored in the contents of pointer variables:



Let's look deeper into both statements to really understand what is happening:

A variable named c	c
is declared to be of type char	char c
and is assigned a value	char c =
of Q	char c = 'Q';

Now that the variable c has been declared, its address can be used

A variable named char_ptr	char_ptr
is declared to be a pointer	*char_ptr
to type char	char *char_ptr
and is assigned a value	char *char_ptr =
of the address of c	char *char_ptr = &c;

More on Pointer References

In the example below, there is a **direct** and **indirect** reference to the variable named "c". When you use a **pointer** to get to the memory location of the variable named "c", then it is an *indirect* reference.

```
#include <stdio.h>
int main ()
{
    char c = 'Q';
    char *char_ptr = &c;

    printf ("%c ", c); /* direct */
    printf ("%c \n", *char_ptr); /* indirect */

    return (0);
}
```

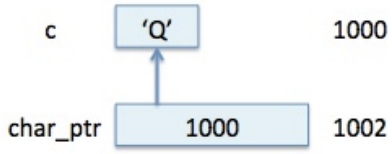
What will print out?

Q Q

Here is what things would look like in memory after the printf statement:

Name	Memory Location	Contents
c	1000	'Q'
char_ptr	1002	1000

... and better illustrated as:



Why did char_ptr start at 1002 instead of 1001?

... on my system, new variables will often start on **even byte boundaries**.

Note that the pointer to a variable of type char was initialized in the declaration statement. This is the same as declaring the pointer and then initializing by assignment.

Thus: char *char_ptr = &c; is equivalent to:

```
char *char_ptr;    /*  in the declaration ,... AND ... */

char_ptr = &c;     /* also with this statement later on ... */
```

However:

```
char * char_ptr;

*char_ptr = &c;   /* don't do this */
```

would put the address of c in whatever *char_ptr pointed to (but not in char_ptr itself). This may or may not be valid.

Assigning values via pointers

You can use a pointer to **update a value** in the memory location it is currently referencing.

```
#include <stdio.h>
int main ()
{
    char c = 'Q';         /* simple char */
    char *char_ptr = &c; /* pointer */

    printf("%c %c \n", c, *char_ptr);

    /* Always make sure a pointer contains */
    /* an address before referencing it.   */
    *char_ptr = 'R';

    printf ("%c %c \n", c, *char_ptr);

    return (0);
}
```

What will print out?

```
Q Q
R R
```

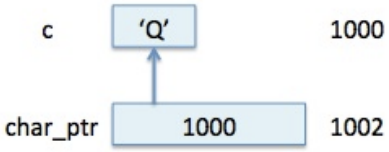
Notice that *char_ptr can be either on the **right or left side** of the assignment operator

Now R will print ... since we replaced Q, which had been in variable c, with R.

As far as how things would be stored, this is what c and char_ptr would look like after they have been declared:

Name	Memory Location	Contents
c	1000	'Q'
char_ptr	1002	1000

And can be visualized as:



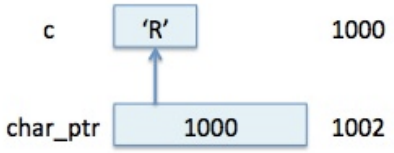
After this statement:

```
*char_ptr = 'R';
```

This is what c and char_ptr would look like:

Name	Memory Location	Contents
c	1000	'R'
char_ptr	1002	1000

... and visualized as:



A pointer is just another way to reference a memory location to read, update, or modify.

- The **direct** method: c = 'R';

- The **indirect** (via a pointer) method: *char_ptr = 'R';

... both methods change the contents of the variable c to the character value 'R'

Pointers and Precedence

Here is an example where a pointer references one integer variable in a program, and a pointer instead of a direct reference is used in a calculation of another integer variable. The order of precedence is very important to understand in this calculation.

```
#include <stdio.h>
int main ()
{
    int i1 = 10;
    int i2;
    int *p1 = &i1;

    /* whatever p1 points to is divided by 2

    i2 = *p1 / 2 + 10; /* ( (*p1) / 2) + 10 */

    printf ("%i", i2);

    return (0);

}
```

What will print out?

The **indirection operator (*) has higher precedence than division**. 15 will print.

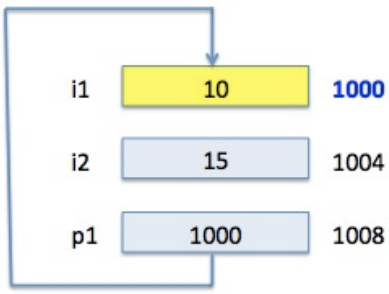
Whatever the contents are that pointer p1 points to, in this case, i1, will be then divided by 2 and its result added to 10.

You can change the order of precedence just by using **parentheses**. Most coding standards will mandate that you do that anyway, so it is clear what you want the precedence to be as well as to anyone reading or maintaining your code.

After the printf statement above, this is what memory might look like:

Name	Memory Location	Contents
i1	1000	10
i2	1004	15
p1	1008	1000

And visualized as:



Video - Arrays and Pointers

Pointers allow you to access array elements more quickly than by accessing elements with indexes. This key point is that you can make sections of your C code run up to 50% faster in terms of each array access by simply replacing array index references with pointer references.

Pointers to Arrays

Pointers are closely associated with arrays and therefore provide an alternative way to access individual array elements.

```
#include <stdio.h>
int main ()
{
    char char_array [ 5 ] = "John";

    char *char_ptr; /* a pointer to a char */

    char_ptr = char_array; /* char_array is an address */

    char_ptr = &char_array [ 0 ]; /* same as above */

    return (0);
}
```

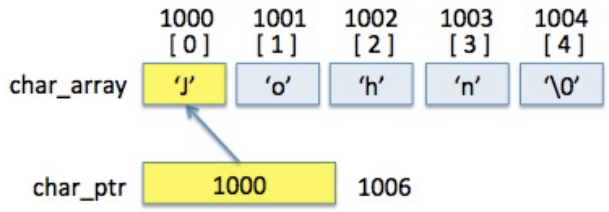
There is no separate pointer type. A pointer needs to know what type (int, char, float, struct) it points to so it can calculate whether the next element is one, two, or N bytes from the current location.

Remember that **arrays are passed to functions by address**. Array elements are **passed by value**. This is why we need the address operator & in the second example, and not the first.

Below is what char_array and char_ptr might now look like in memory. I started char_ptr at an even memory location, using address 1006 instead of 1005 that was available. Also, remember that elements in an array are consecutive in memory. The second element (char_array [1]) immediately follows the first element (char_array [0]), and so on for each array element.

Name	Memory Location	Contents
char_array [0]	1000	'J'
char_array [1]	1001	'o'
char_array [2]	1002	'h'
char_array [3]	1003	'n'
char_array [4]	1004	'\0'
- NOT USED -		
char_ptr	1006	1000

It can be better visualized by the illustration below. It is very important to understand that char_ptr is not a pointer to an array, rather an address to a byte location that is defined as **type char**, as it is a pointer to a character. The char location just happens to be in an array element.



Pointer Arithmetic

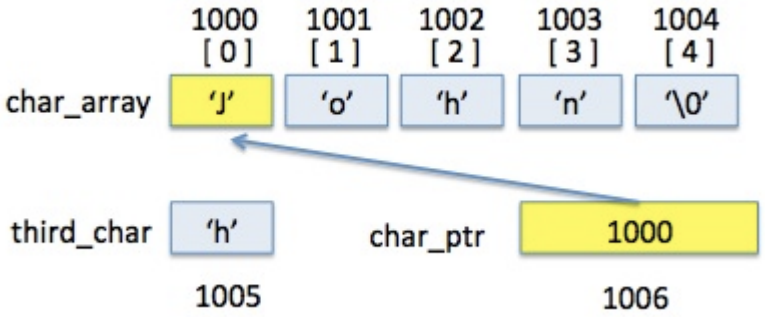
Pointers contain **addresses** and they can only point to memory locations that have been defined to hold a specific type. In the code below, both char_array and third_char hold character values. The char_ptr variable is defined a pointer that can reference a character value location in memory. **Pointer arithmetic** allows us to perform **mathematical operations** on the contents of a pointer, because it contains a memory address. Consider the program below:

```
#include <stdio.h>
int main ()
{
    char char_array [ 5 ] = {"John"}, third_char;

    char *char_ptr; /* a pointer to a character */

    char_ptr = char_array; /* array name = address */

    /* assignment using array index */
    third_char = char_array [ 2 ];

    /* assignment using pointer */
    third_char = *( char_ptr+2 );

    return (0);

}
```

Below is what char_array, and third_char and char_ptr might now look like in memory. With the addition of the third_char variable, char_ptr just now happens to start at an even memory location (1006). See how the third_char variable now has the value of the byte at the memory location two bytes pass the start of the array (char_array[0]), in this case, the letter 'h' was is stored n char_array[2], or memory location 1002. Remember how we calculated array offsets in our array lecture notes? Might be a good time to revisit it.

Name	Memory Location	Contents
char_array [0]	1000	'J'
char_array [1]	1001	'o'
char_array [2]	1002	'h'
char_array [3]	1003	'n'
char_array [4]	1004	'\0'
third_char	1005	'h'
char_ptr	1006	1000

Better illustrated, here is what it looks like:



Because char_ptr was declared to point to char, the compiler knows that char_ptr+2 means 2 char locations (not 2 integers or anything else). Since a char is 1 byte, it means 2 bytes.

Pointers to arrays can be incremented using the unary ++ or decrement with - - operators.

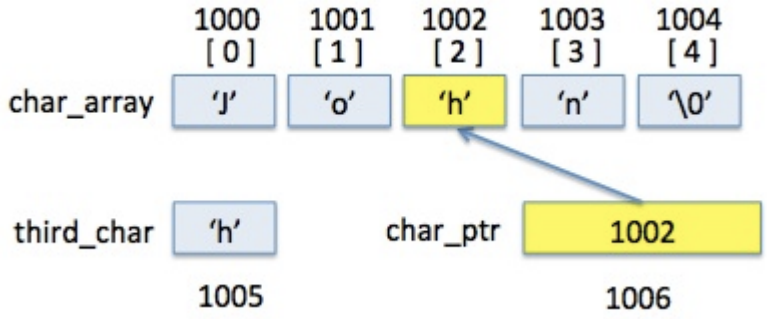
The next array element can also be accessed by incrementing the pointer with:

```
++char_ptr;    /*  increment address by one char  (1 byte) */
```

And two character locations past it would be:

```
char_ptr += 2;  /* increment two char memory locations (two bytes) */
```

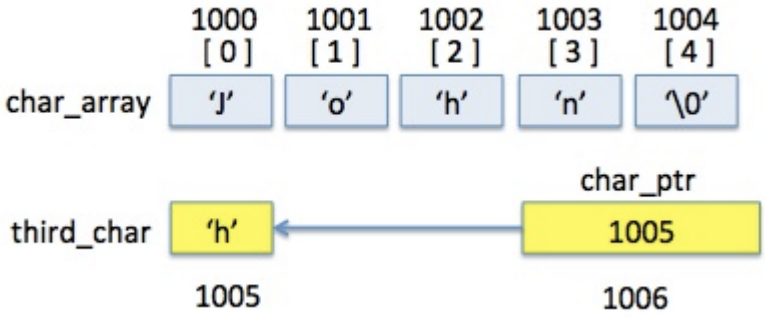
If we implemented the statement above, char_ptr would be changed, and things would now look like:



Finally, char_ptr does not have to point to any array element, you could also make it point to another character location that is not a array, such as the variable third_char.

```
char_ptr = &third_char;
```

Now, things would look a bit different as show below:


1000
[0]
1001
[1]
1002
[2]
1003
[3]
1004
[4]
char_array
'J'
'o'
'h'
'n'
'\0'
char_ptr
third_char
'h'
1005
1005
1006

Why use Pointers?

Let us look at two functions that accomplish the same thing. Both will take an array of integer values along with its size as parameters, and return the sum of all the elements in that array. One is done without pointers, and the other uses pointers. One is **MUCH faster** than the other. Let's look deeper into both functions:

```
/* Non Pointer solution */

int array_sum (int array [ ], int num_of_elements)
{
      int i;              /* array index */
      int sum = 0;   /* sum of array elements */

      for (i = 0; i < num_of_elements; ++i)

            sum += array [ i ];  /* calculate array address each time */

     return (sum);
}
```

```
/* Pointer solution */

int array_sum (int *array, int num_of_elements)
{
      int sum = 0;   /* sum of array elements */
      int *array_end = array + num_of_elements;    /* end of array */

      for (; array < array_end;  ++array)

           sum += *array;   /* big savings here */

     return (sum);
}
```

In looking at both functions, the only real major difference in a statement that gets executed each time in the loop is between these statements:

```
sum += array [ i ];
```

verses

```
sum += *array;
```

The big savings is that when your program is executing, it sees array [i] and needs to calculate its address. Remember what we learned in our lecture notes on arrays? The following calculation must be made each time in the loop:

```
& array [ i ] = array + ( i * sizeof (int)  )
```

On the other hand, when the compiler sees *array , it just references that memory location found in the array variable, which is a pointer that contains an address.

It seems like a really small difference, doesn't it? However, let's say the difference between the two statements is that figuring out **array [i]** takes one more executed instruction (machine language) to process, as no matter what high order language you use, they all get translated by the compiler and run in

machine language. If your loop executes 1000 times, that means that the non-pointer function will be executing 1000 more machine instructions at run time EACH time that function is called. Over the long run in your program, not using pointers can really add up in terms of how fast your program will execute.

Important Point to Remember:

Pointers are more efficient because they simply reference a memory location directly rather than having to calculate an address each time as in: array [i]

Video - Pointers and Functions

Pointers are special when used with functions. Pointers are passed by value, meaning that only a copy is passed to a function and its value is not returned to the calling function. Watch this video to learn how to work with pointers with functions. It will show you how to work with a simple pointer and a pointer that works with Arrays.

Pointers and Functions

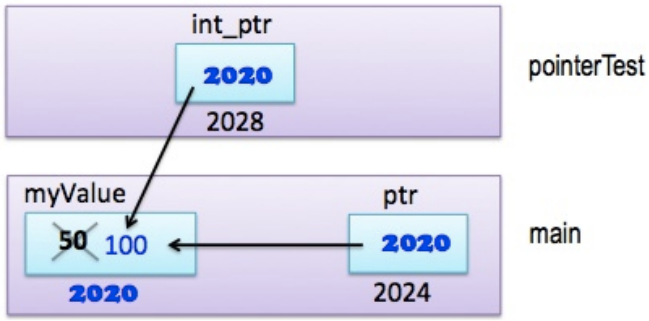
A **pointer** is **passed by value**, however, since it contains an **address**, when passed to a function, a value at a location being referenced by a pointer can be accessed and/or modified, even if that memory location is a ***local variable in another function***.

Take the code example below which sets up a simple integer **local variable** called **myValue** in the main function and sets up a **local variable pointer** to it (called **ptr**), where it then passes the contents of that pointer to our **pointerTest** function. This function will then **update** the value being referenced by this pointer to 100 ... which in this case is the local variable called **myValue** in the calling function (**main**).

Still confused by my wordy explanation? Simply refer to the diagram below which shows what is happening in the function stack frame when the statement

```
*int_ptr = 100;
```

is being executed in the **pointerTest** function. Note that both pointers, **int_ptr** and **ptr**, are pointing to the same location in memory, in this case **memory location 2020**, which happens to be where the local variable **myValue** lives in the **main function** stack frame. Let's now hope that the illustration below makes everything crystal clear once you review both it and the complete program example below.



Also note that at no time is the address of the pointer modified, only the **value** at the location it is pointing to is modified (in this case, the local variable in the main function called **myValue**).

Feel free to try it out at: http://ideone.com/d0vD2n

```
#include <stdio.h>

/*****************************************************************
 * Function:  pointerTest
 *
 * Description:  Demos passing a pointer to a function as a parameter
 *               and changing the value referenced by that pointer will
 *               actually update the same variable value being
 *               referenced by a pointer in the calling function.
 *
 * Parameters:  int_ptr - Pointer to an int (to value in calling funct)
 *
 * Returns:  void (but value in calling function will be set to 100
 *
*****************************************************************/
void pointerTest (int * int_ptr)
{
     *int_ptr = 100; /* set the contents of the address it points
                        to to 100, which in this case, is where
                        variable i in the main routine is located */
}
```

```
int main ()
{

    int myValue = 50;    /* simple integer value to test */

    int *ptr = &myValue; /* ptr is a pointer to an integer */
                         /* set to address of myValue      */

   printf ("Before function call:  myValue = %i and *ptr = %i \n",
            myValue, *ptr);

    /* call the function and pass the contents of the pointer ptr, */
    /* which contains the ADDRESS of the variable myValue          */
    pointerTest ( ptr );

    /* upon return from test, the variable i will now be set to 100 */
    printf ("After function call:  myValue = %i and *ptr = %i \n",
            myValue, *ptr);

    return (0);
}
```

OUTPUT:

```
Before function call:  myValue = 50 and *ptr = 50
After function call:  myValue = 100 and *ptr = 100
```

SUMMARY:

Pointers are passed by value, but allow **access** to variables in the calling function because the **value is an address**.

... Note that the argument and formal parameters do not contain the * operator

Constant Character Strings and Pointers

C treats a constant **character string as a pointer**, an address, therefore it cannot be assigned to a variable that is not a pointer.

An **exception** is made for initializing arrays when they are declared. Given the following three declarations which shows a character pointer and two character array variables:

```
char *char_ptr;
char text [ 80 ];
char string1 [ 80 ] = "This is OK";
```

Let's determine if the following statements are valid:

```
char_ptr = "A String";    /* This is OK as "A String" is seen as an address */

text = "A String";        /* This is NOT OK, sets address to another address  */

&text [ 0 ] = "A String"; /* This is NOT OK ... same as above */
```

In summary, you can set the contents of a pointer to an address and you can set the contents of a variable being declared as a character array to a **literal string** value (like "This is OK").

When the compiler sees a literal string, it will find an address in memory to store it somewhere, often in the **Data area**. When it sees an **array name**, remember the following:

Code Segment	What the Compiler Sees
text	An Address, same as &text [0]
&text [0]	An Address, same as text
text [1]	The contents
text [j]	The contents
&text [2]	An Address

You can not set one address equal to another address. The only exception is that you can set a pointer that contains an address equal to another address.

If you get confused in the future, feel free to come back to this page to refresh your memory on these rules and examples.

Array of Pointers

You can have an array where each element is a pointer. Notice that pines is declared as an array of pointers. It will hold six pointers, each pointing to wherever the C compiler stores character strings (most times in the **data area** ... remember the C run time discussion?). Therefore, each element will contain memory addresses.

```
#include <stdio.h>
int main ()
{

   static char *pines [ ] =
   {
      "white",
      "yellow",
      "black",
      "red",
      "pitch",
      "Austrian"
   };

   printf ( "%s \n", pines [ 1 ] );
   printf ( "%c \n", *( pines [ 1 ] + 1) );

   return (0);

}
```

The first printf statement will print **yellow** ... remember, arrays start at 0 for an index.

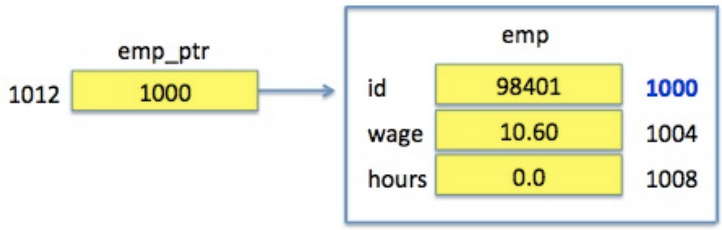
The second printf will print the second character of the second string, the **'e'** from **yellow**.

Pointers and Structures

A pointer can be defined to reference the starting address of a variable that is a structure type. The pointer does not point to individual members of the structure, rather, it points to the starting address of the first member in the structure. In the example below, the address of a structure called **emp** is placed in a variable (**emp_ptr**) which is defined as a pointer to type **struct employee.** The program below shows how to set a pointer to a variable of a structure type and then reference it to access each of variable's members.

```
#include <stdio.h>

struct employee
{
    int id;
    float wage;
    float hours;
};

int main ()
{
    struct employee emp;       /* structure */
    struct employee * emp_ptr; /* pointer to structure */

    emp_ptr = &emp; /* pointer to address of structure */


    (*emp_ptr).id = 98401; /* works */

             /* or */

    emp_ptr -> id = 98401; /* better */

    /* access and set the other two members too */
    emp_ptr->wage = 10.60;
    emp_ptr->hours = 0;

    return (0);
}
```

Better illustrated, it shows a pointer variable that contains the address to the starting location (the first member) of the variable emp, which is of type struct employee:



If we had said ***emp_ptr**.id = 98401, it would be treated as *** (emp_ptr.id)** = 98401

because "." has higher precedence than *. The **id** member of **emp_ptr** does not exist; **emp_ptr** points to a structure but is not itself a structure.

The -> (hyphen, greater than) allows you to write things such as:

```
if  ( emp_ptr -> id = = 98401 )
```

instead of

```
if  ( ( *emp_ptr ).id = = 98401 )
```

The -> produces more readable code. Of course, **emp.id = 98401** also works. But **emp_ptr.id = 98401** does not work.

If however, instead of **emp_ptr = &emp**, we said ***emp_ptr = emp,** we would be trying to put the contents of emp1 in the memory pointed to by ***emp_ptr**. This would be legitimate if ***emp_ptr** had been set to point to anything.

I said it before, and I'll say it again, an important pointer concept is that you must set a pointer to **valid address** before using it.

Pointers and Structures in Functions

When a **pointer** that has been **set** to point to a specific structure variable is passed to a another function, that pointer can be used to access or update the contents of that specific structure variable in that **called function**. Similar to what happens when you pass an entire array to a function (i.e., **Call by Reference**). Let us expand our previous program and design it in a way that calls functions to print and update the contents of an employee variable by passing in a pointer to a structure to it. You can try it out at: https://ideone.com/HdT3OD

```
#include <stdio.h>

struct employee
{
   int id;
   float wage;
   float hours;
};

/*****************************************************************************
* Function:  updateEmp
*
* Description:  Updates an employee using a pointer to a structure
*
* Parameters:  emp_ptr - Pointer to the employee structure variable
*              clock - employee clock number value
*              wage - employee weekly wage value
*              hours - employee hours worked value
*
* Returns:  void (but note the structure pointed to will be updated)
*
*****************************************************************************/
void updateEmp (struct employee * emp_ptr, int clock, float wage, float hours)
{
   /* update employee using the passed  */
   /* pointer to the structure variable */
   emp_ptr -> id = clock;
   emp_ptr->wage = wage;
   emp_ptr->hours = hours;

   return;
}


/*****************************************************************************
* Function:  printEmp
*
* Description:  Prints information contained in an employee structure
*               given a pointer that has been set to it.
*
* Parameters:  emp_ptr - Pointer to the employee structure variable
*
* Returns:  void
*
*****************************************************************************/
void printEmp (struct employee * emp_ptr)
{
```

```
    /* print a nice header */
    printf ("\nClock  Wage   Hours ");

    /* print the contents of the employee structure variable */
    /* using the pointer to the employee structure variable  */
    printf ("\n%06i %5.2f %5.1f\n", emp_ptr->id,
                                    emp_ptr->wage,
                                    emp_ptr->hours);
    return;
}


/* Program using pointer to a structure */
int main ()
{
    struct employee emp;          /* structure */
    struct employee * emp_ptr;  /* pointer to structure */

    emp_ptr = &emp;  /* pointer to address of structure */


    (*emp_ptr).id = 98401;  /* works */

    /* or */

    emp_ptr -> id = 98401;  /* better */

    /* access and set the other two members too */
    emp_ptr->wage = 10.60;
    emp_ptr->hours = 0;

    /* print the employee */
    printf ("\nEmployee changed in main function by our pointer:\n");
    printEmp (emp_ptr);

    /* Update it via a function call */
    updateEmp (emp_ptr, 58734, 14.56, 40.0);

    /* print the updated employee */
    printf ("\nEmployee changed in updateEmp function by our pointer:\n");
    printEmp (emp_ptr);

    return (0);
}
```

Sample Output

```
Employee changed in main function by our pointer:

Clock    Wage   Hours
098401 10.60   0.0

Employee changed in updateEmp function by our pointer:

Clock    Wage   Hours
058734 14.56  40.0
```

A Pointer to an Array of Structures

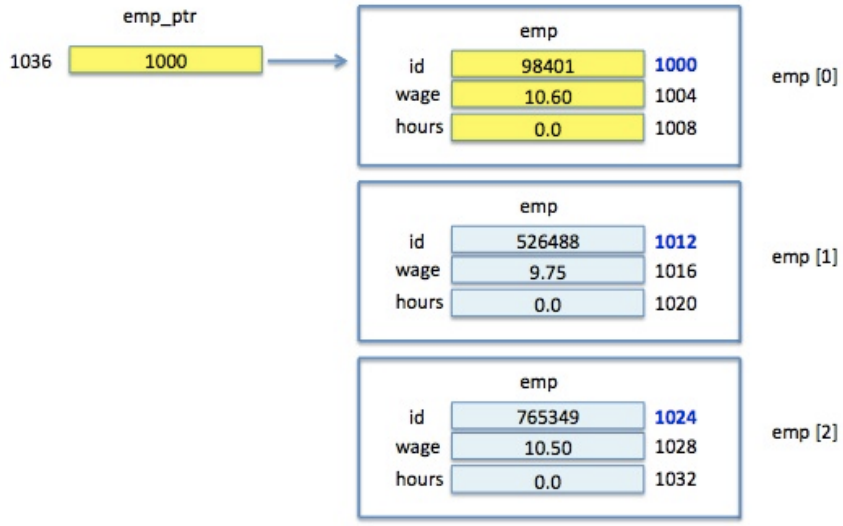
Let's take our structure pointer discussion up a level and show you how it can be very useful when using a pointer to a structure within an **Array of Structures**. In the code below, we have replaced our simple structure declaration with an array of structures which has three elements where the **id** and **wage** members are all initialized leaving the **hours** member in each element to default to zero.

```
/* Array of structures */

struct employee emp [3] =  {    {98401, 10.60},
                                {526488, 9.75},
                                {765349, 10.50}
                           };
```

Setting a pointer to an array of structures is no different to setting a pointer to a simple structure, since a pointer does not point to an entire array of structures, rather, it points to a specific element within the array whose type matches the type the pointer was defined at, in this case, struct employee. You can guarantee with an array, that each **element** is of the **same type,** and every element will follow each other **consecutively** in memory. These two key concepts about arrays will be important with our discussion on pointers, as well as show how similar they are in terms of how the compiler sees the both of them.

```
/* A pointer to variables of type structure employee */

struct employee * emp_ptr;

/* Set pointer equal to starting address of the array */
/* Array name is seen by the compiler as:  &emp [0]   */

emp_ptr = emp; /* same as &emp [0] */
```

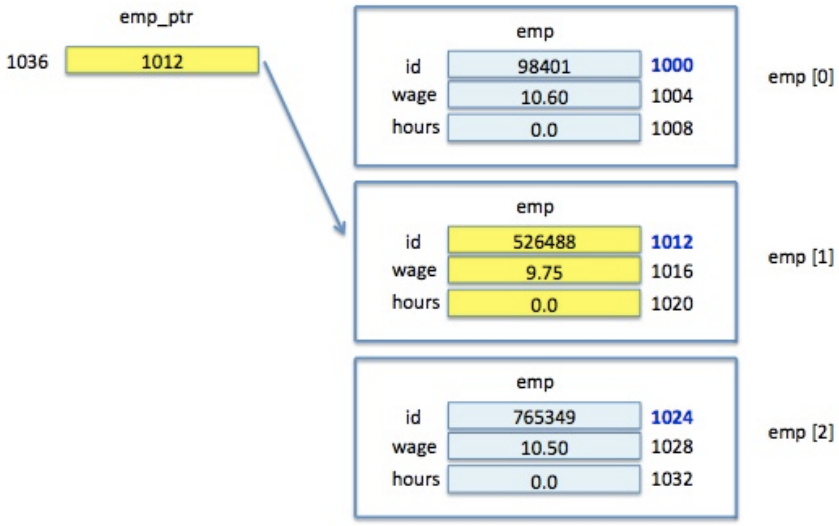
One can now visualize the **emp** and **emp_ptr** variables in **memory** as shown below. I started the emp variable at address location 1000 in my example and assumed both integer and float values on my computer each have a size of 4 bytes. In the real world, it is not that important that you know the exact memory location of an item, but it is very important to illustrate it within this class so you know exactly what is going on. Each computer is different and will have its own memory size that is consistent on that computer for each variable type, along with starting addresses that it will assign at run time.



Whenever emp_ptr is incremented by one, it will increment the address stored in emp_ptr by the size of the type it points to, in this case, struct employee.

```
    /* increment pointer by the size of struct employee */
    ++emp_ptr;
```

Note that structure employee has one int member and two floats, each of which on my computer are 4 bytes each for a total of 12 bytes. Therefore, rather than emp_ptr being incremented by 1 to 1001, it will be set to 1012 (or 1000 + 12). Any reference to the pointer will now reference the second element in the structure since emp_ptr previously pointed to the first element of the array of structures. You can now visualize things in memory looking like this:



To put everything together into a complete program for you to try, consider the code below. It has a function called main that has a local variable called emp which is an array of structures along with a pointer to a structure called emp_ptr that can be used to access and modify the members contained in each array element. There are two functions, for which we will pass the starting address of the array of structures along with the array size. The key difference here is instead of referencing array elements with indexes, we are going to **use pointers** which are **much faster** into terms of accessing the data item in memory.

Instead of using array indexes to locate items, such as:

```
    printf ("Enter hours for employee %06i: ", emp [ i ] . id);
    scanf  ("%f", &emp [ i ] . hours);
```

We can use pointers, which are nearly **twice as fast** in terms of accessing the data item (I base this on the fact that they will generate less machine/assembler instructions that get executed during run time ... this is just me being a software engineer again :) If you get in the habit of using pointers instead of array references, these little tips can really speed up your programs in the long run, especially if these statements are located with loops that are executed many times over.

```
    printf ("Enter hours for employee %06i: ", ptr -> id);
    scanf  ("%f", &ptr  -> hours);
```

The program below provides a complete solution that you can try out in our favorite compiler or within IDEOne. The homework assignment this week is just an expansion upon this code. To try it out, go to: http://ideone.com/wzH4fF

```
#include <stdio.h>

#define SIZE 3

/* struct type for an employee */
struct employee
{
    int id;
    float wage;
    float hours;
};
```

```
/* Function Prototypes */
void getHours (struct employee * ptr, int size);
void printData (struct employee * ptr, int size);

/*******************************************************
* Function - getHours
*
* Description: Will prompt for the number of hours worked
* for each employee stored in the employee
* array being passed
*
* Parameters: ptr - pointer to an array of employees
* size - the number of employees to process
*
* Returns: void (the pointer will update the passed array)
*
*******************************************************/

void getHours (struct employee * ptr, int size)
{
    int i; /* loop index */

    for (i = 0; i < size; ++i)
    {
        printf ("\nEnter hours for employee %06i: ", ptr->id);
        scanf ("%f", &ptr -> hours);

        /* move pointer to next employee */
        ++ptr;
    }
}

/*******************************************************
* Function - printData
*
* Description: For a given set of employees, this function
* will print a one line summary report on each
* employee. The total number of employees
* printed is based on the size parameter.
*
* Parameters: ptr - pointer to an array of employees
* size - the number of employees to process
*
* Returns: void (the pointer will update the passed array)
*
*******************************************************/

void printData (struct employee * ptr, int size)
{
    int i; /* loop index */

    printf ("\n\nID\tWage\tHours\n");

    for (i = 0; i < size; ++i)
    {
        printf ("%06i\t%5.2f\t%5.1f\n",
                ptr->id,
                ptr->wage,
                ptr->hours);

        /* move pointer to next employee */
        ++ptr;
    }
}
```

```
int main ()
{

    /* Array of structures */
    struct employee emp [SIZE] = { {98401, 10.60},
                                   {526488, 9.75},
                                   {765349, 10.50}
                                 };

    /* Read in hours for each employee */
    getHours (emp, SIZE);

    /* Print employee information to the screen */
    printData (emp, SIZE);

    return (0);

}
```

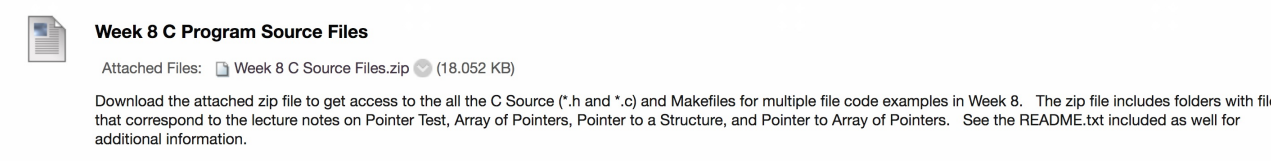
Output:

```
Enter hours for employee 098401: 45.6
Enter hours for employee 526488: 40.0
Enter hours for employee 765349: 34.4

ID     Wage  Hours
098401 10.60 45.6
526488 9.75  40.0
765349 10.50 34.4
```

Week 8 C Program Source Files

We covered many different programs with pointers, strings, structures, and functions 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 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:



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

We just completed probably the most complex topic in C. Yet, this is the most important topic. Using pointers will allow you to really get the full benefits in terms of speed and execution that make C one of the fastest executing languages in the world.

Here are few important things about pointers that I hope you picked up this week:

- There are **no specific pointer types** in C, instead, a pointer is defined to reference a specific type within C
- The **contents** of a pointer variable should be a **valid address** within memory - Pointer variables contain addresses, not values!
- Always make sure a pointer points to some place in memory **before referencing it** .. don't reference location 0 for example
- The main reason to use pointers is that they are faster than array references - **SPEED!**

Test your knowledge this week with our **Quiz** on pointers.

The **homework assignment** this week will allow you to **substitute** the use of **pointers** in all places within your code where you previously used **array references**. There is a template you are welcome to use and it is based on one of our lecture notes this week that dealt with pointers to an array of structures. Feel free to improvise and add your own look and feel to homework assignment, but make sure you use pointers instead of array references. If you don't feel you have pointers understood well at this time, continue to re-read the lecture notes and re-watch the videos, and of course, try out the examples in the lecture notes. It goes without saying that starting with the homework template will serve you well and make your job much easier. However, you may find that it might be even easier to reference the templates as needed and simply update your homework from last week to utilize pointers instead of array references.

For Next Week

The most complex topic of the course will be covered next week. We'll take you very deep into the discussion of pointers and way past what the book has to offer. The concept to be covered is a data structure known as **Linked Lists**, where we will be able to ask for additional memory when the program is running. This is often referred to in the computer science world as **Dynamic Memory Allocation**. This will allow our homework to then work for **ANY** number of employees, which if you think about, would be a very important feature if you were to someday sell and license a program that provided this capability. In the real world, you don't always know how many data items you will be processing up front, it is something that is a variable.

Yet that is a topic for next week. This week we will know how many data items we will be processing up front in our homework assignment. Good Luck!

HOMEWORK 7
POINTERS

Write a C program that will calculate the gross pay of a set of employees utilizing pointers instead of array references. You program should continue to use all the features from past homeworks (functions, structures, constants, strings).

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.

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;
};
```

Set a pointer to it and then use that pointer going forward to access elements (and their associated members) in your array of structures. Again, do not use array references with indexes (use emp_ptr->hours ... not ... emp [i].hours as the latter is not a fast).

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.

Do not use any array references with indexes. For example:

```
emp[i].wage /* this is bad, it uses an array reference with an index,
in this case, i */

emp_ptr->wage; /* this is good, it uses a pointer to reference the wage
value */
```

Intermediate Optional Challenge

Add two more rows to calculate the minimum and maximum values but remember to use pointers instead of array references in your calculations.

a) Calculate and print the **minimum** wage, hours, ot, and gross values
b) 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 not 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 will now look like something shown below. Make sure however that you utilize pointer references instead of arrays references to make all this happen.

```
-------------------------------------------------------------
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.

Homework 7 Code Template

Call by Reference

Here is a **template** to use for homework 7 that calls every function ***by reference*** (i.e., passes the address of the array of structures which is in effect a pointer). If you want to deviate from it, that is fine, as long as you follow the rules specified in the homework 7 assignment. There are many ways to design and implement this program. The template will compile and start printing values right out of the box. It is your job to fill in the missing code and provide input so that it displays the correct information.

It can be accessed at: http://ideone.com/ek7PV0 or you can simply cut/paste it into IDEOne or your favorite compiler/text editor.

```
/*************************************************************************/
/*                                                                       */
/*  HOMEWORK:   7                                                        */
/*                                                                       */
/*  Name:   Joe Student                                                  */
/*                                                                       */
/*  Class:  C Programming, Cybercourse                                   */
/*                                                                       */
/*  Date:                                                 */
/*                                                                       */
/*  Description:  Program which determines gross pay based on overtime   */
/*                and outputs a formatted answer.  Employee information  */
/*                is stored in an array of structures and referenced     */
/*                through the use of pointers.                           */
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>


/* define all constants here */
#define SIZE       5

/* type to hold employee information */
struct employee
{
     char name [20];  /* Employee first and last name */
     int id;          /* unique employee identifier */
     float wage;      /* hourly wage rate */
     float hours;     /* hours worked in a given week */
     float overtime;  /* hours worked after the standard work week */
     float gross;     /* total gross pay, standard pay + overtime pay */
};


/* add function prototypes here if you wish */


/* Remember to add function comment header block for each function */

/* like shown below for printData, a stub for getHours is below.   */
void getHours ( struct employee * emp_ptr, int size )
{

}
```

```
    /*****************************************************************/
    /*                      Function: printData                      */
    /*                                                               */
    /*  Purpose:    Outputs to screen in a table format the following:  */
    /*                  - Employee First and Last Name               */
    /*                  - Employee clock number                      */
    /*                  - Wage rate for an employee.                 */
    /*                  - Total hours worked by employee.            */
    /*                  - Overtime Hours.                            */
    /*                  - Gross Pay.                                 */
    /*                                                               */
    /*  Parameters: emp_ptr - pointer to array of structures         */
    /*              size - number of employees to process            */
    /*                                                               */
    /*  Returns:    Nothing, since emp_ptr is passed by reference    */
    /*****************************************************************/

void printData ( struct employee * emp_ptr, int size )
{
    int n;  /* counter used in for loop to keep track of iterations */

    /* prints the output for all employees to the screen */
    for (n = 0; n < SIZE; n++)
    {
        printf ("%-20.20s %06i   $%5.2f     %4.1f      %4.1f    $%7.2f \n",
                emp_ptr->name,
                emp_ptr->id,
                emp_ptr->wage,
                emp_ptr->hours,
                emp_ptr->overtime,
                emp_ptr->gross);

        ++emp_ptr;  /* move to next employee */
    }
    printf ("\n");
}

    /*****************************************************************/
    /*                           Function: Main                      */
    /*****************************************************************/

int main()
{

    /* A structure array to hold information on employees */
    struct employee emp [SIZE]  =  { {"Connie Cobol", 98401, 10.60},
                                     {"Frank Fortran", 526488, 9.75},
                                     {"Mary Apl", 765349, 10.50},
                                     {"Jeff Ada", 34645, 12.25},
                                     {"Anton Pascal", 127615, 10.0}
                                   };

    /* Get user input for the hours worked for each employee */
    getHours ( emp, SIZE );

    /* Calculate overtime for each employee */

    /* Calculate gross pay for each employee */

    /* Print column headers for each employee */

    /* Print employee data to the screen */
    printData ( emp, SIZE );

    return 0;
}
```

HOMEWORK 7 Code Template

Combination of Call by Value and Call by Reference

Show below is an ***alternate code template*** at http://ideone.com/ug82Px that has a one loop in the main function instead of each called function. You can process each employee one at a time with this design. It ***passes and returns values***, rather than ***passing by address*** to other functions. Saying that, in this template, the last function, **printData**, still passes the structure array ***by reference*** as it has a loop inside that function. Likewise from the original template, I have provided the **printData** function code if you wish to use it.

```
/*************************************************************************/
/*                                                                       */
/*  HOMEWORK:   7                                                        */
/*                                                                       */
/*  Name:   Joe Student                                                  */
/*                                                                       */
/*  Class:  C Programming, Cybercourse                                   */
/*                                                                       */
/*  Date:                                                                */
/*                                                                       */
/*  Description:  Program which determines gross pay based on overtime   */
/*                and outputs a formatted answer.  Employee information  */
/*                is stored in an array of structures and referenced     */
/*                through the use of pointers.                           */
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>


/* define all constants here */
#define SIZE       5

/* type to hold employee information */
struct employee
{
     char name [20];  /* Employee first and last name */
     int id;          /* unique employee identifier */
     float wage;      /* hourly wage rate */
     float hours;     /* hours worked in a given week */
     float overtime;  /* hours worked after the standard work week */
     float gross;     /* total gross pay, standard pay + overtime pay */
};


/* add function prototypes here if you wish */


/* Remember to add function comment header block for each function */

/* like shown below for printEmp, a hint for getHours is below.   */
float getHours ( int id )
{
     float hours;  /* hours worked for the employee */


     /* prompt with id to read in a value for hours */
```

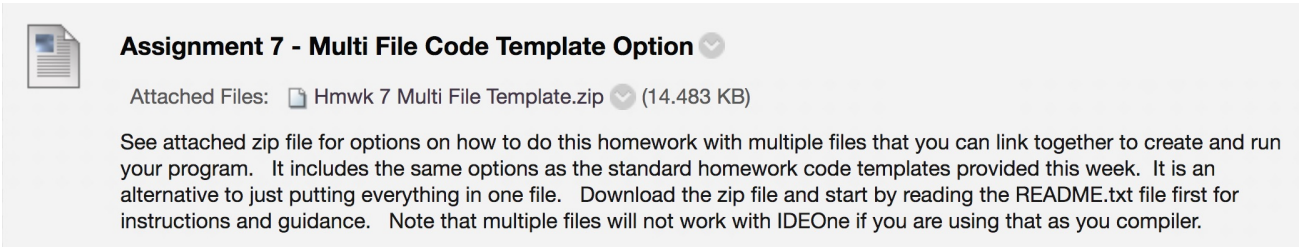
```
     return (hours);
}


/*********************************************************************/
/*  Function: printEmp                                               */
/*                                                                   */
/*  Purpose:    Outputs to screen in a table format the following:   */
/*                  - Employee First and Last Name                   */
/*                  - Employee clock number                          */
/*                  - Wage rate for an employee.                     */
/*                  - Total hours worked by employee.                */
/*                  - Overtime Hours.                                */
/*                  - Gross Pay.                                     */
/*                                                                   */
/*  Parameters: emp_ptr - pointer to array of structures             */
/*              size - number of employees to process                */
/*                                                                   */
/*  Returns:    Nothing, since emp_ptr is passed by reference        */
/*********************************************************************/

void printEmp ( struct employee * emp_ptr, int size )
{
    int n;  /* counter used in for loop to keep track of iterations */

    /* prints the output for all employees to the screen */
    for (n = 0; n < SIZE; n++)
    {
        printf ("%-20.20s %06i   $%5.2f     %4.1f      %4.1f    $%7.2f \n",
                emp_ptr->name,
                emp_ptr->id,
                emp_ptr->wage,
                emp_ptr->hours,
                emp_ptr->overtime,
                emp_ptr->gross);

        ++emp_ptr;  /* move to next employee */
    }
    printf ("\n");
}


/*********************************************************************/
/*                          Function: Main                           */
/*********************************************************************/

int main()
{

    /* A structure array to hold information on employees */
    struct employee emp [SIZE]  =  { {"Connie Cobol", 98401, 10.60},
                                     {"Frank Fortran", 526488, 9.75},
                                     {"Mary Apl", 765349, 10.50},
                                     {"Jeff Ada", 34645, 12.25},
                                     {"Anton Pascal", 127615, 10.0}
                                   };


    int i;                        /* loop and array index */

    struct employee * emp_ptr;    /* pointer to an employee structure */
```

```
    emp_ptr = emp;    /* point to beginning of emp array */


    /* Read in hours, and calculate overtime and gross - Call by Value */
    for (i=0; i < SIZE; ++i)

    {

        /* Get user input for the hours worked for each employee */
        emp_ptr->hours = getHours ( emp_ptr->id );

        /* Add function calls to calculate employee overtime and gross */

        ++emp_ptr;  /* point to next employee */


    } /* end for */

    /* Print column headers for employees */

    /* Print employee data to the screen - Call by Reference */

    printEmp (emp, SIZE);

    return (0);

}
```

Homework 7 - Multi File Template Option

Take the optional challenge this week and implement Homework Assignment 7 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:



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 7 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.