Welcome to Week 6!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 6 folder**.
- Read **Chapter 8** in the latest edition of the textbook (chapter 9 in earlier editions).
- Begin **Quiz 6**. It is due Sunday at midnight.
- Begin **Assignment 5**. It is due Sunday at midnight.
- Begin the **Midterm Exam**. It is due in two weeks (see Class Syllabus for due date).

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 6 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, except if it reveals a midterm exam answer.
- Try out the various **code examples** in the lecture notes. Feel free to modify them and conduct your own **"What ifs"**.

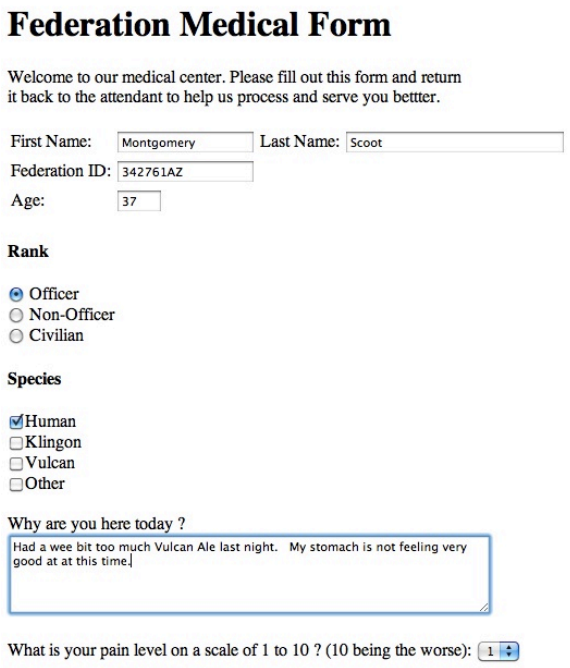
"Thank you, Mr. Sili, for helping me "structure" my data in Dr. McKoy's Medical Records System."

--- Nyota Arugula
Communications Officer, USS Enterprise

Structures

This week we'll learn about another way to represent data which may or may not be of the same type, and show how it can be captured within a container known as a **Structure**. Structures will give us the advantage in our homework assignment this week of simply working with one array (of structures) rather than having to have five arrays for each piece of employee data. We will also learn about how a C Program is executed within the **C Run Time Environment**.

To help introduce you to structures, let's continue to look in yet again on the daily life of the USS Enterprise. **Dr. McKoy** has tasked **Lt. Arugula** to update his C Program that processes the **medical records** for all the Star Fleet personnel aboard the ship. It seems that she is having an issue with how to store into variables all the different items that appear on a medical records form. Some look like floating point numbers, others are integers, and some are one or more characters. For example, here is a section of a medical form that the good doctor might use in his sickbay. As you can see, it has all kinds of data. Ask yourself, what variable types would you use to represent the data on the form? The answer is a combination of many variable types, as well as arrays. A Medical Form as shown below would be a good candidate for a programmer to utilize a **structure** to model the data and figure out a way to store and process information on it.



Federation Medical Form

Welcome to our medical center. Please fill out this form and return it back to the attendant to help us process and serve you bettter.

First Name: Montgomery Last Name: Scoot

Federation ID: 342761AZ

Age: 37

Rank

- (•) Officer
- ( ) Non-Officer
- ( ) Civilian

Species

- [x] Human
- [ ] Klingon
- [ ] Vulcan
- [ ] Other

Why are you here today ?

Had a wee bit too much Vulcan Ale last night. My stomach is not feeling very good at at this time.

What is your pain level on a scale of 1 to 10 ? (10 being the worse): 1

Star Trekking - A Structuring of Data

This week **Lt. Arugula** asks for help on how to "structure her data" from fellow second officer **Mr. Sili**, who has a bit more experience with C Programming. Click the image below or go back to the link in the lecture notes this week and watch the movie to find out what happens.

Array Review

In Chapter 7, we learned about arrays. **Arrays** let us work with a group of variables, all of the **same type**.

```
int array1 [ 10 ];  /* array declaration */

x = array1 [ i ];    /* to use on element, identify it */
```

Structures are like *records* in other languages. They permit working with a group of both **like** and **unlike** variables.

Video - Arrays and Structures

This video will introduce you to two Data Structures: Arrays and Structures. Arrays store like items together and Structures can store related items that may be comprised on many different types. Arrays can be part of Structures, and Structures can be part of Arrays, ... they often work together.

Structure Syntax

A **structure** is a set of values that can be referenced **collectively**.

Differs from an **array**:

1. Items in a structure are referred to as **Members** of a structure rather than elements.
2. Members of a structure **do not necessary have the same type**
3. Members are **referenced differently**

The general format of a structure is:

```
struct struct_name
{
        type member-name1;
        type member-name2;

        ...

        type member-nameN;
};
```

Defines a structure **template** called **struct_name** with indicated members. Each **member declaration** takes on the following general form:

<type> *<member-name> ;*

Suppose you needed to store several dates inside your program.

```
struct date

{

    int month;  /* more than one member */

    int day;     /* members may differ in type */

    int year;    /* defines a template, no space allocated */

} ;
```

This defines a structure called date with three members who happen to be of the same type: an integer called **month**, an integer called **day**, and an integer called **year**. A common error for new programmers is to forget the **semicolon** after the ending **curly brace**. With the above declaration, you are only telling the compiler what a date structure looks like; you're **not** reserving any **memory space**. Basically, you've defined a **template**, not a variable. Even though each member is the same type (integer), they are all **logically related**. You can now can store, reference, and update each part of the date as needed.

Member	Type
month	Integer
day	Integer
year	Integer

They don't all have to be the same type, here is an example of how you could use a structure to store information about the employees we have been processing in our homework assignments:

```
struct employee
{
        long int clock;
        float wage;
        float hours;
        float ot;
        double gross;
};
```

Notice that unlike our date structure, this structure has five members, and not all members have the same type. This is what makes a structure a very powerful feature, and it allows us to better model what most items in the real world would look like.

Member	Type
clock	long integer
wage	float
hours	float
ot	float
gross	double

Summary

Members may be of **different types**: int, float, char, double, long, ... even another struct type!

The **template**, defined before the function(s), can be used to create many instances of variables, all of the same size, but each with its own memory allocation.

Structure Variables

Declaring Structure Variables

Now that you've told the C compiler what the structure looks like, you're ready to go ahead and declare variables to be of this structure type.

```
struct date today;
```

This tells the compiler to reserve space for a variable called **today**, which is of type **struct date**.

Member	Type
month	integer
day	integer
year	integer

Member types can be any valid C types (int, long, short, float, char, double, ...), including other structure types. Compare how we would create a variable of type integer with a variable of type struct date

```
int value;
 struct date today;
```

Don't get hung up on the *today* variable having two words that specifies its type, in this case **struct date**, whereby the *value* variable is just a single word for its type, **int**.

Assigning Variables to Structure Variables

Arrays identify elements with indexing; structures join a member name with its structure using a **period** to separate the names.

A member of a structure is accessed by specifying the variable name, followed by a period, which in turn is followed by the member name. It can be used on the right or left side of an equation just like any other variable.

```
<structure variable name> . <member name>
```

Below are some examples.

```
today.day = 31;
++today.day;
x = today.day;
printf ("%d", today.day);
x = today.day + 7;
```

In the case of our structure variable **today**, members would be **month**, **day**, and **year**. The **value** could be any integer value or expression.

So to store the date "March 23, 1996" inside the **today** variable, you could accomplish this with the following three statements:

```
today.month = 3;
```

```
today.day   = 23;
today.year  = 1996;
```

Member	Type	Value
month	integer	3
day	integer	23
year	integer	1996

Below is an example of a program to print the date stored in the variable **today**. Try it out at:
http://ideone.com/y4TaPk

```
/* user defined structure type for a date */
struct date
{
   int month;
    int day;
   int year;
};

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

   struct date today;   /* defines a variable of type struct date */

   today.month = 3;
   today.day = 23;
   today.year = 1996;

   printf ("%d/%d/%d \n", today.month, today.day, today.year - 1900);  /* Y2K Issue ? */

   return (0);

}
```

Output:

```
3/23/96
```

Structure Initialization and Operations

Structure Initialization

The general format for initializing a structure is:

```
struct  struct_name  struct_variable  =  {val1, val2, ... valN};
```

So to initialize the variable today to March 23, 1996

```
struct date today = {3, 23, 1996};
```

struct is hard-coded and always that value

date is the name of structure type that you need to supply, same *name rules* as variables

today is the variable of the type **struct date**

Notice that the order is based on how the structure was declared, so month is first and has a value of 3, day is second and gets set to 23, while year is last and set to 1996. Any value not initialized would be set to 0. The word **static** is necessary if you are not using an ANSI C compiler. For most compilers these days, you don't need it.

```
static struct date today = {3, 23, 1996};
```

Operations on Structures

One of the few operations supported by structures is the ability to assign one structure variable to another, **provided they are of the same type.**

Therefore, if you wanted to copy the date stored the variable date to another variable of type structure date called tomorrow, you could do the following:

```
/* declare two variables, initialize only the first.  The other, tomorrow, has undefined values */
struct date today  =  {3,23, 1996},  tomorrow ;
/* set the contents of tomorrow to today */
tomorrow = today;
```

Saying **tomorrow = today;** has the same effect as setting the ***corresponding members*** equal to each other:

```
tomorrow.month   =   today.month;
tomorrow.day     =   today.day;
tomorrow.year    =   today.year;
```

You can't do much else with structure as a whole, except to pass them to and return them from functions. Don't try to test two structures for equality like:

```
if (today  ==  tomorrow)
```

it won't work. You need to compare them member by member:

```
if   (tomorrow.month  ==   today.month  &&
```

```
tomorrow.day   ==  today.day   &&
tomorrow.year  ==  today.year)
```

Initializing Structures

```
/* a structure type should be declared outside all functions */

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

/* a structure variable can be declared outside all functions,
   thus it is global.  A global variable may be initialized globally
   without the word "static".  ANSI C does not need the word "static" */

struct employee emp1 = { 98401, 10.60 };  /* global variable */

int main ( )
{

  static struct employee emp2 = { 526488, 9.75 };  /* local variable */

}
```

Like Arrays, put initialization values in curly braces and separate by commas.

Notice that 98401 is used rather than **098401**, which is **octal**

Most Non-ANSI C compilers do not permit initialization of automatic (i.e., local) structure varaibles, in this case, emp2

Global structures or static structures may be **initialized**.

This is what would initially be stored in the two structures:

emp1

Member	Value
id	98401
wage	10.60
hours	0
ot	0
gross	0

emp2

Member	Value

id	526488
wage	9.75
hours	0
ot	0
gross	0

Video - Passing Structures to Functions

This video will show the various ways you can pass a Structure to a Function. Like integer, float, and character variables, Structures as "Passed by Value". This means that only a COPY of each member's contents (i.e., their values) are passed to a called function.

Functions and Structures

As with ordinary variables, and unlike arrays, any changes made by the function to values contained in the structure argument will have no effect on the original structure. They effect only the **copy of the structure** that is created when the function is called. Review the code at: https://ideone.com/g5l7tp

```
#include <stdio.h>


/* Its OK to make the type global, that way all functions can use it */
/* Note that "struct date" is NOT a global variable */
struct date

{

    int month;

    int day;

    int year;

};

void printNextDay (struct date dateval);  /* function prototype */


int main ()
{
    struct date today; /* local variable to main */

    /* Set up a date to pass to the printNextDay function */
    today.day = 17;
    today.year = 1996;
    today.month = 10;

    /* pass by value the information to our function*/
    printNextDay (today);

    /* The value of today will be unchanged - still the 17th */
    printf ("%d/%d/%d \n", today.month, today.day, today.year-1900);


    return (0);

} /* main */

/*************************************************************************
**
** Function: printNextDay
**
** Description: Simply prints the next day of a given 20th century date
** in MM/DD/YYYY format. Does not check for last day in the month (known
** issue to be addressed in the future)
**
** Parameters: dataval - a structure with month, day, and year
**
** Returns: void
**
*************************************************************************/
```

```
void printNextDay (struct date dateval)
{

    ++dateval.day; /* add a day to the value passed into this function */

    printf ("%d/%d/%d\n", dateval.month, dateval.day, dateval.year-1900);

    return; /* optional, no value returned since it returns void */

} /* printNextDay */
```

Output:

10/18/96

10/17/96

- The **struct date** template is defined prior to the first function; it is **global** and allocates no memory.
- The **local** variable **today** in the main function is declared as type **struct date** and ***passed by value*** as an ***argument*** to **printNextDay**.
- The dateval **parameter** in printNextDay just holds a **copy** of the value passed from main, since nothing is returned, the local variable **today** in main is **unaffected** by any changes made in printNextDay.

Returning Structures from Functions

As with ordinary variables, and unlike arrays, any changes made by the function to values contained in the structure argument will have no effect on the original structure. They effect only the copy of the structure that is created when the function is called. Thus, structures are **passed by value**.

Note that all structures are not created equal: **nextDay** returns a type of **struct date**.

A code example is shown below using date formats and you can try it out at: http://ideone.com/dZ1lWc

```
struct date
{
    int month;
    int day;
    int year;
};

/* function prototype */
struct date nextDay (struct date dateval);

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

    /* two structure variables */
    struct date today, tomorrow;

    /* set today to the proper date */
    today.day = 17;
    today.year = 1996;
    today.month = 10;

    /* This statement illustrates the ability to pass a  */
    /* structure to a function and to return one as well */

    tomorrow = nextDay (today); /* tomorrow updated */

    printf ("%d/%d/%d\n", tomorrow.month,
                          tomorrow.day,
                          tomorrow.year-1900);

    return (0);
}

/*********************************************************
** Function:  nextDay
**
** Description:  Returns the next day given a date
**
** Parameters:  dateval - a given date
**
** Returns:     dateval - updated next day date
**
*********************************************************/

struct date nextDay (struct date dateval)
{

    ++dateval.day; /* add a day */
```

```
    return (dateval); /* return updated structure */
} /* nextDay */
```

Arrays of Structures

A structure variable can be declared as an **array**. In this case, there will be three **elements** of a **structure type** based on the template "employee".

```
struct employee
{
   int id;
   float wage;
   float hours;
   float ot;
   float gross;
};

struct employee  emps [ 3 ] ;

int x ;
```

Each **member** can be set or assigned to another variable that shares the same type. Given that x is a simple variable defined as an integer type:

```
 emps [ 1 ] . id = 98401;

 x = emps [ i ] . id;
```

How would you pass the **third occurrence** of the structure employee in the array emps to a variable x?

The **third occurrence** of the structure employee is passed to the function **check_employee** as:

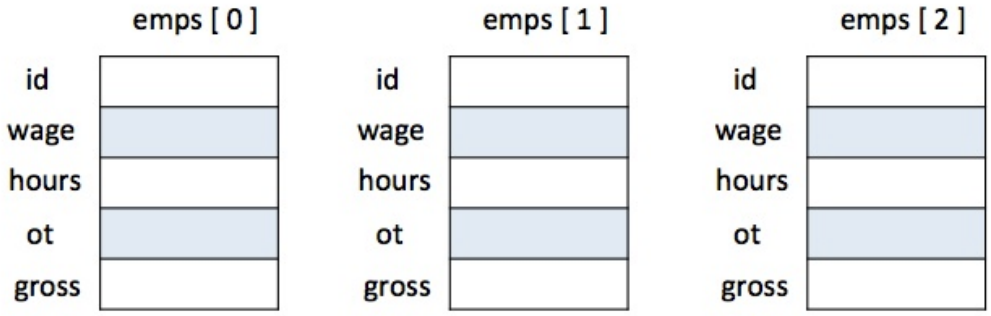
```
check_employee ( emps [ 2 ] );  /* pass by value, just an element in the array */
```

Notice that the index above applies to the structure, not the member. You are referencing an **element** of an array called emps. Each element happens to be of type structure employee.

The id member, an integer value, of the third occurrence of the structure employee could be called as:

```
check_id ( emps [ 2 ] . id );   /* also passed by value, passes in integer value */
```

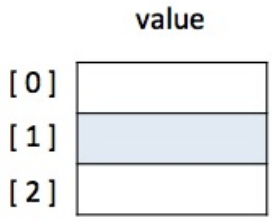
To visualize what would be stored, the **array emps** would contain 3 **elements** which are defined as **type struct employee**.



Compare to a simple array of integers:

```
int value [ 3 ] ;
```

The real difference is that the elements are just less complex, in the case of the variable value ... they are just integers. In both cases, the array elements are consecutive in memory ... all the array rules apply regardless of the type of its elements.



Initializing Arrays of Structures

Initializing an **array of structures** is the same as initializing arrays in general. The code statement below initializes each of the three elements in the emps array.

```
struct employee emps [ 3 ] = { { 98401, 10.60 } , /* other members
default to 0 */
                               { 526488, 9.75 } ,
                               { 765349, 10.5 }   /* no comma here */
                             };
```

Remember that all **array elements** in the emps array are of *type* ***struct employee***. That means each element, such as emps [0] would be a structure and allow access to the **members** id, wage, hours, ot, and gross. The next element, emps [1] , would also be a structure, and allow access to its members of id, wage, hours, ot, and gross. There would be a total of three elements, where each element is the same type, in this case, **struct employee**.

Alternatively, you can initialize each member in every element of the array, separated by commas. To get at all the members, you'll need to put zeros for any values which do not have an initial value

```
struct employee emps [ 3 ] = { 98401, 10.60, 0, 0, 0, 526488, 9.75, 0,
0, 0, 765349, 10.5, 0, 0, 0 };
```

Below is what each **element** of the emps array would look like after execution of either of the statements above:

	emps [0]
id	98401
wage	10.60
hours	0
ot	0
gross	0

	emps [1]
id	526488
wage	9.75
hours	0
ot	0
gross	0

	emps [2]
id	765349
wage	10.5
hours	0
ot	0
gross	0

Video - Passing Arrays of Structures

An Array of Structures is an array where each member is of some structure type, for example, the ***struct date*** type we covered in our lecture notes. This video will show you various ways to pass an Array of Structures to a function.

Passing Arrays of Structures

QUESTION:

If an element of an array is passed by value:

```
somefunc ( array1 [ 2 ] );
```

How is an element of an array of structures passed?

By value or by reference?

```
check_id ( emps [ i ] );
```

... ANSWER:

It is passed by value.

QUESTION:

If the entire array is passed by reference (i.e., the starting address of the array is passed):

```
somefunc ( array1 );
```

How is the entire array of structures passed?

By value or by reference?

```
check_id ( emps );
```

... ANSWER:

It is passed by reference - array1 is the same as &array1 [0]

Structures Within Structures

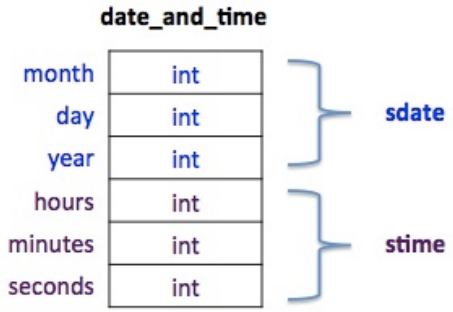
It is possible to define a **structure** that itself **contains another structure** as one or more of its members, as well as define structures that contain arrays. Let's say you have two structures, one to collectively store and work with time values (hours, minutes, and seconds) and other to work with date values (month, day and year). These two structures are shown below:

```
struct time
{
    int hour, minutes, seconds;
};

struct date
{
    int month, day, year;
};
```

Now let's add a new structure that contains two members who are themselves structures. We can have a structure called **date_and_time** that contains all members needed to work with both **date** and **time** values. This structure is shown below:

```
struct date_and_time
{
    struct date sdate;
    struct time stime;
};
```

You could envision the structure template looking something this:



The program below puts all of this together to show you how to set up and work with structures that contain other structures as their members. Feel free to try it out at: https://ideone.com/GYqpu9

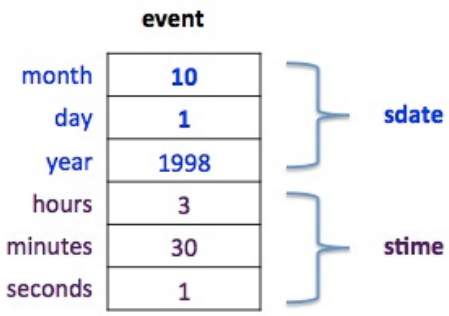
```
struct time
{
    int hour, minutes, seconds;
};

struct date
{
    int month, day, year;
};

struct date_and_time
```

```
{
    struct date sdate;
    struct time stime;
};

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

    struct date_and_time event =
    {
        {2,1,1988}, /* date - month, day, year*/
        {3,30,0}    /* time - hour, minutes, and seconds*/
    };

    event.sdate.month = 10;

    ++event.stime.seconds;

    printf ("\nDate:\t %i/%i/%i\n",
            event.sdate.month,
            event.sdate.day,
            event.sdate.year);

    printf ("Time:\t %i hour(s) %i minute(s) %i second(s)\n",
            event.stime.hour,
            event.stime.minutes,
            event.stime.seconds);

    return (0);

} /* main */
```

Output:

```
Date:    10/1/1988
Time:    3 hour(s) 30 minute(s) 1 second(s)
```

From left to right, you go from **general** to **specific**, separated by periods. Even though both **date** and **time** are simple structures with three members who all have a type of **int**, the member names give an indication of what they are used for, and show why **time** is **different** than **date**. Here is what the variable **event** would look like after the program is executed. Note that **month** was set to 10 and **seconds** was incremented to 1.



Arrays of Structures Containing Structures

As we showed previously, it is possible to define a structure that itself contains another structure as one or more of its members, as well as define structures that contain arrays.

The example code below shows how to declare, update, and print an **array of structures** where each **element** is bit more **complex** than we have previously covered. The array **event** below contains **three elements**, each **element** has **members that are themselves structures** to work with both time and date values.

Try it out at: https://ideone.com/ieQ0S8

```
struct time
{
    int hour, minutes, seconds;
};

struct date
{
    int month, day, year;
};

struct date_and_time
{
    struct date sdate;    /* stores date values */
    struct time stime;    /* stores time values */
};

#include <stdio.h>
int main ( )
{
    struct date_and_time event [3] =
    {
        { {2,1,1988}, {3,39,10} }, /* first date, then time values */
        { {5,6,1989}, {3,56,20} },
        { {8,5,1996}, {6,40,44} }
    };

    event [1].sdate.month = 10;  /* change a month value */

    ++event [1].stime.seconds; /* Add one second */

    for (int i=0; i < 3; ++i)
    {

        printf ("\nDate:\t %i/%i/%i\n",
                event[i].sdate.month,
                event[i].sdate.day,
                event[i].sdate.year);


        printf ("Time:\t %i hour(s) %i minute(s) %i second(s)\n",
                event[i].stime.hour,
                event[i].stime.minutes,
                event[i].stime.seconds);
    }

    return (0);
}
```

Output:

```
Date: 2/1/1988
Time: 3 hour(s) 39 minute(s) 10 second(s)

Date: 10/6/1989
Time: 3 hour(s) 56 minute(s) 21 second(s)

Date: 8/5/1996
Time: 6 hour(s) 40 minute(s) 44 second(s)
```

Here is what our array would look like just before the end of the program after it has been updated.

	event [0]
month	2
day	1
year	1998
hours	3
minutes	39
seconds	10

	event [1]
month	**10**
day	6
year	1989
hours	3
minutes	56
seconds	**21**

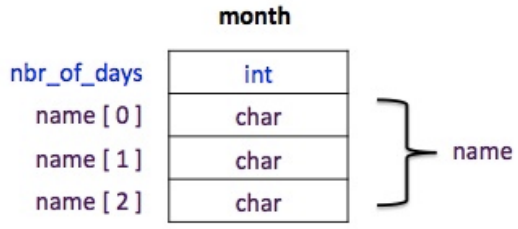
	event [2]
month	8
day	5
year	1996
hours	6
minutes	40
seconds	44

Structures Containing Arrays

Structures can also have **arrays** as one or more **members**. In the structure below, the **name** member is an array of characters.

```
struct month
{
    int nbr_of_days;
    char name [ 3 ];
};
```

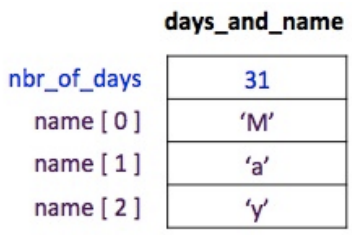
The **template** would look something like this (note how space is allocated for each array element in the name array):



A code example is shown below:

```
struct month
{
    int nbr_of_days;
    char name [ 3 ];
};

int main ( )
{
    struct month days_and_name = {31, 'M', 'a', 'y'};
    char one_char;

    one_char = days_and_name.name [ 2 ]; /* the 'y' value */

    return (0);
}
```

You can envision the structure **variable** called **days_and_name** looking like this after it is initialized in the code above.



QUESTION?

How is a structure containing an array passed?

By **value** or by **reference**?

ANSWER:

It would be passed **by value** since you are still passing a structure, even though it contains an array.

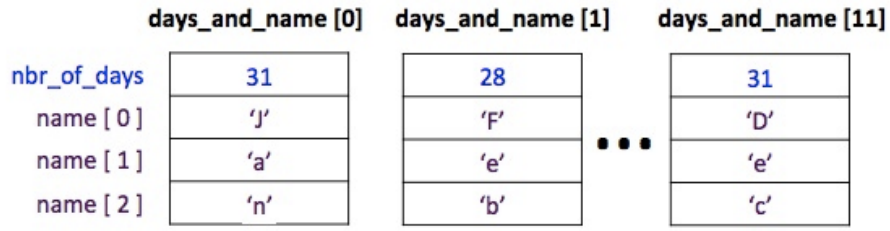
```
somefunc ( days_and_name );
```

Arrays of Structures Containing Arrays

Structures can also have **arrays** as **one or more members,** and the **structure** itself can be declared as an **array** if desired.

```
struct month
{
    int nbr_of_days;
    char name [ 3 ]; /* array who is a member of this structure */
};

int main()
{

    struct month days_and_name [ 12 ]; /* an array that contains month
info */

}
```

You can envision the above **Array of Structures** containing an **Array** as a **member** looking like this (note: I don't have room to show all 12 elements, but you get the idea):



What **character** does the following statement refer to:

```
months [ 1 ] . name [ 2 ]
```

The **third character** in the **second month** (remember that array indexes start at **zero**) is: **'b'**

Notice the consistency in the use of the index to identify elements in both arrays:

```
months [ 1 ] or name [ 2 ]
```

and the use of the period to identify specific members of the structure.

Structure Variants

A **structure variant** is a structure that does not have a t**emplate name** associated with it.

```
struct /* template not named */
{
    int month;
    int day;
    int year;
} todays_date, purchase_dates = {9, 25, 1987};
/* two structures created, one initialized */
```

The **template** does not have a **name**, thus it can not be declared later in the program. If it had a template name, you could declare it both here, as in **todays_date**, and later.

If the above template is global, then todays_date and purchase_date are global.

Structures containing Structures

As you recall, a structure can combine like and unlike types. To illustrate this, let's look at modeling a typical **credit card**. I'm sure most of you have a credit card. If you have one handy, pull one out and take a look at the information on it. If you had to capture the information on the front of the credit card using C variables, in particular structures, how would you go about it? Here is a sample public domain image of a credit card I pulled off the Internet.

Source: Pixabay.com (Public Domain, Free to reuse)

1) Step 1 - Identify and Analyze the values (let's call them attributes) on the Credit Card

Credit Card Number - 16 digit number, digits range from 0-9, can start with a series of zeros
Start Date - The initial date the credit card is valid
Expiration Date - The date the credit card is no longer valid, it has expired
Customer Name - First, Middle, and Last Name

... I've seen some that also have a ***Member Since*** date, but let's go with the four items above.

2) Step 2 - Determine what type best fits each Attribute Value

The **credit card number** is broken up into 4 parts, one could group them together as they have some meaning (if you are really interested, check out: http://money.howstuffworks.com/personal-finance/debt-management/credit-card1.htm), but let's just store them together within one type in C. It is not a **float** number, but seems to be an **integer** value (whole numbers, digits 0-9). The tricky part is that numbers could start with zero, which in C would make them **Octal** (http://en.wikipedia.org/wiki/Octal) numbers, and the digits 8 and 9 are not valid octal numbers (0-7). The best type would just be a **character string** who is at least 16 characters long, plus room for the *extra null terminator character* ... let's just make it 20 characters. In reality, there are no mathematical functions (add, multiply, divide, etc.) that will probably need to be performed on the credit card number, so a character string should be just fine.

```
char number [20];
```

Both the **Start Date** and the **Expiration Dates** contain the same information, the month and year, where we can both assume they are whole numbers ... which is perfect for an integer type. However, its best to store the month and year individually in case we wanted to sort or search for data, or simply print it in a variety of

format. We could probably store the day as well just to have it. Remember we previously covered a **struct date** type? It would be perfect

```
/* Here is the structure type */
struct date
{
      int month;
      int day;
      int year;
};

/* Two variables of that type */
struct date startDate;
struct date expirationDate;
```

The final data piece is the **customer name**. It seems to have a first name, middle name, and last name. Much of this information is probably some character value, which is most likely letters. Let's group this information together into a structure. You might want to **sort** or **search** by *first* or **last name**, or any *combination*, and **print** in a variety of ways (First Middle Last ... or Last, First Middle or Last, First, ...). Make sure you have enough space to hold a name ... last names these days can be long ... and some people combine maiden names into last names. The last name is a very important piece to help uniquely define a person. If you wanted to, you can add a **prefix** to it (Dr., Mr. Ms., ...) and a **suffix** (like Jr., III, ...).

```
struct name
{
      char first [60];     /* first name */
      char middle;         /* middle initial */
      char last [100];     /* last name */
};

struct name customer;
```

3) Now put all the code together into a Structure Type that can store all the information about a credit card

Note how the structures **date** and **name** provide *supporting structures* for members within the **credit_card** structure. Not all members of the credit care structure need to be a structure type, note that I made the *credit card number* just a character string. The benefit of the supporting structures data and name is that if you need to store another date or name in some other structure or variable, you can now **reuse** these structures, include the credit card structure type itself.

```
struct date
{
      int month;
      int day;
      int year;
};

struct name
{
      char first [60];
      char middle;
      char last [100];
};

struct credit_card
{
      char number [20];
      struct name customer;
      struct date startDate;
      struct date expirationDate;
```

```
};
```

A **common mistake** is to create a *separate* structure for the starting date and one for the expiration date, but they both would have the same members: month, day, and year. Later on, you might store the birthday for a customer, and you can continue to use the struct date type. Another **mistake** is the naming all members by preceding it with the struct name, like credit_card_number, credit_card_customer, credit_care_startDate, and credit_card_expirationDate. For member names in a structure, just provide a good name for what type of data they are holding, and like variable names, make the name as descriptive as possible.

4) Create a variable of that Type to store multiple items

The best way to do this is to create an **array of structures**. Let's create one to store at least 1000 credit cards.

```
struct credit_card mycreditcards [1000];   /* can hold info on up to 1000 credit cards */
```

Of course, if you were setting up a database, you probably want to store much more information, such as the customers address, phone number, etc. In fact, the design of a structure is very similar to the design of a class in C++ or Java, or the actual logical design of a database. However, this exercise was limited to the information that is displayed on the front of a typical credit card. Feel free to expand upon this on your own time if you wish.

Video - C Run Time Environment

This video will give you a good introduction to the 4 areas of the C Run Time Environment - Text, Data, Stack, and Help. It will also discuss the differences between a global, static, and local variable.

C Run-Time Environment

C is special is that you tell the compiler where to place a variable in the run-time environment. The compiler doesn't decide, YOU DO! This is a bonus because it gives you more control of your execution (Of course, it is bad if you don't know what you are doing, but ...)

Understanding your Run Time Environment can help you to:

1. Reduce Run Time Overhead => **Faster Execution**
2. Estimate Memory Requirements => **Require less memory**

In whatever Operating System (OS) you are running under, the system first allocates a chunk of memory for your program.

The executing code and associated system structures make up what's called a process (UNIX point of view), and the OS copies the executable from disk to memory. This is more than just executable statements, as the system has to allocate memory for other parts of the program as well.

The **four program areas** of the run time environment are the **text**, the **stack**, the **data**, and the **heap**. It is possible that a program will not use the data or heap area

The Fixed Sized Program Areas

Text Area

Area reserved for the *executable instructions* of the program and not the ASCII characters that make up the source file. The memory area is considered read only by the system and is often called program text. This size is fixed since the size is known at load time. It is read-only by the system.

Data Area

Area reserved for C *static* variables and *global* variables. There's nothing dynamic about these variables. The system knows exactly how much memory to allocate when it is time to run your program. Constant strings are generally here but some compilers put them in the text area.

This area is divided into two areas: *initialized* data and *un-initialized* data. Where your variable goes depends on whether you initialized it. The un-initialized area is sometimes called the BBS area (Block Started by Symbol). The System fills the area with 0's.

The Dynamic Memory Areas

The Stack

The stack (First in/Last out) is dynamic and grows according to how your program executes. C stores the following items on the stack:

1. Automatic (i.e., local) variables
2. Parameters
3. Return addresses of parameters
4. Temporary variables during expression evaluation

There is a separate stack frame (i.e., items 1-4 above) for each active function. The stack program data area is simply a series of stack frames of active functions as the program executes.

The Heap

The heap is dynamic and allows a program to specify additional storage areas at run time. You can allocate storage from the heap and later return it back when it is no longer needed. It is possible to use all the heap space available, so a programmer must check for overflow errors (no memory left to allocate).

The heap and stack sometimes share the same area; hence, a large amount of heap space may decrease the amount of available stack space, and vice versa.

Variable Types

We covered global, static, and local variables somewhat in last week's lecture notes. The main difference you should be aware of is their **scope**, in terms of how they can be accessed and modified in a C Program. A **global variable** can be accessed by any function in a program, while **static** and **local variables** can only be accessed within the function they have been declared. The major difference between a static and local variable is that a **static variable** is stored in the data area, thus it maintains it previous value each time the function is called. By contrast, a **local variable** lives in the function stack frame, so its value will not be maintained the next time its function is called.

There are a few other differences ... I posted a nice diagram that shows how each variable type stacks up against each other. It is something you might want to reference from time to time.

	Global Variables	Static Variables	Local Variables
Memory Location	Data Area	Data Area	Function Stack Frame
Initialized	Once	Once	Every Time
Default Value	Zero	Zero	Undefined
Scope	Any Function	The function it was declared in	The function it was declared in

Run Time Example

Let us look at an example of how C would store variables in the function stack and program data areas. Note that *staticvar* and *globalvar* are always stored in the program **data area**. A few things I want to you to note and take away from this example:

- **globalvar** (our global variable) can be accessed and modified from any function
- **staticvar** (our static variable) and **autovar** (our local variable) can only be accessed from the functions where they were declared
- **globalvar** is only initialized once to a value of 2
- **staticvar** in the printit function will only be initialize to 2 the first time the function is called. It will then maintain its value in the *data area* throughout the life of the program
- **autovar** is initialized each time in the printit function

Feel free to try it out at: http://ideone.com/bNtlm9

```
/*    GLOBAL, AUTO, and STATIC VARIABLE DECLARATION EXAMPLE */

#include <stdio.h>
int globalvar = 2;  /* global variable initialized only once */
                    /* and its value is always held in memory*/

/************************************************************
**
** Function - printit
**
** Description - Just increments and prints a set of local,
**               static, and global variables
**
** Parameters:  none
**
** Returns:  void
**
************************************************************/

void printit()
{
        static int staticvar = 2; /* static initialized only once */
                                  /* value is always held in memory */

        int autovar = 2; /* local variable within printit */

        globalvar++;
        staticvar++;
        autovar++;

        /* Time line 2 */

        printf("globalvar = %d \n",   globalvar);
        printf("staticvar = %d \n",   staticvar);
        printf("autovar   = %d \n\n", autovar);

} /* printit */

 int main()
{

        int x;    /* local variable within main */

       /* Time line 1 */
```

```
        /* Call printit function three times */
         printit();
         printit();
        printit();

         x = 5;

         /* Time Line 3 */
        return(0);

} /* main */
```

Output:

```
globalvar = 3
staticvar = 3
autovar = 3

globalvar = 4
staticvar = 4
autovar = 3

globalvar = 5
staticvar = 5
autovar = 3
```

Program Area Illustration

Here is what the program data areas look like within the **data area**, which is where *static* and *global* variables are stored, and the function **stack frame**, that evolves as the functions are called and completed. At Time Line #1, only the main function is active as we have not yet called the printit function. The items in the **data area** in terms of storage space remain the same throughout the life of the program, only their values may change.

In Time Line #2, the printit function is called and placed on the function **stack frame**. It has no *parameter*, but does have a *local variable* called autovar which is put on the **stack frame**. The main function is still active and have not yet terminated, and will not do so until the end of the program as main is the first function added to the **stack frame** and the last function deleted from it. The *static* and *global* variables in the program **data area** are updated by one each time the function printit is called.

In Time Line #3, only the main function is active and the globalvar and staticvar still exist in the **data area**. The main function can access globalvar since it is a *global variable* accessible by any function, but only the printif function can access the *static variable* staticvar since it is accessible from the printit function only.

Program Area at Time Line #1

Function Stack Frame

Function	Variable	Value
main	x	Undefined

Data Area

Type	Variable	Value
Global	globalvar	2
Static (printit Function)	staticvar	2

Program Area at Time Line #2 (First time printit function is called)

Function Stack Frame

Function	Variable	Value
printit	autovar	3
main	x	Undefined

Data Area

Type	Variable	Value
Global	globalvar	3
Static (printit Function)	staticvar	3

Program Area at Time Line #2 (second time printit function is called)

Function Stack Frame

Function	Variable	Value
printit	autovar	3
main	x	Undefined

Data Area

Type	Variable	Value
Global	globalvar	4
Static (printit Function)	staticvar	4

Program Area at Time Line #2 (third time printit function is called)

Function Stack Frame

Function	Variable	Value
printit	autovar	3
main	x	Undefined

Data Area

Type	Variable	Value
Global	globalvar	5
Static (printit Function)	staticvar	5

Program Area at Time Line #3

Function Stack Frame

Function	Variable	Value
main	x	5

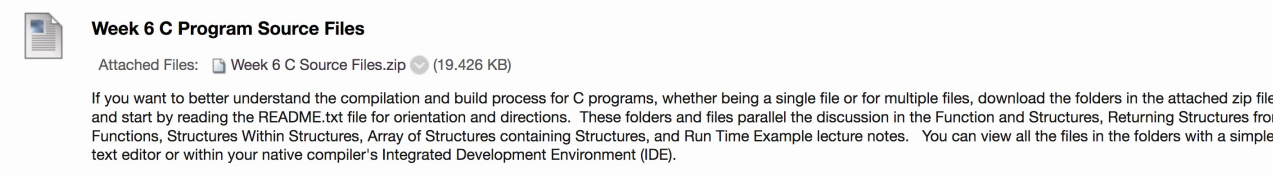
Data Area

Type	Variable	Value
Global	globalvar	5
Static (printit Function)	staticvar	5

Week 6 C Program Source Files

We covered many different programs with 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 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.

For Next Week

Think you know structures? There was quite of bit covered this week. Test your knowledge with our **Quiz** this week.

Finish up **Assignment 4** (Functions) if you are still working on it, and then start **Assignment 5** (Structures). It will help to have a working version of Assignment 4 under your belt before you start Assignment 5, as I want you to continue to use functions and constants. With Assignment 5 this week, you will need to pass **either** an array of structures to each function (where you can then access and update each array element), or you can pass member values to it and have it return a result back to the calling function.

Finally, yes, I did post the **Midterm** this week. You will have two weeks to work on it. It will cover everything up through this week. So what are you waiting for ??? ... get working on it :)

What about our topic next week? We'll cover more than you would ever want to know about the concept of **characters and strings**.

HOMEWORK 5 - STRUCTURES

Write a C program that will calculate the gross pay of a set of employees. Continue to use all the features from your past assignments. In particular, expand upon your previous assignment and continue to use multiple functions and constants to help with various tasks called upon by your program.

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.

```
--------------------------------------------------------------------------
  Clock#   Wage  Hours     OT     Gross
--------------------------------------------------------------------------
  098401  10.60   51.0    11.0    598.90
  526488   9.75   42.5     2.5    426.56
  765349  10.50   37.0     0.0    388.50
  034645  12.25   45.0     5.0    581.88
  127615   8.35    0.0     0.0      0.00
```

You should implement this program using the following structure to store the information for each employee.

```
/* This is the structure you will need, feel free to modify as needed */
struct employee
{
      long  int id_number;  /* or just int id_number; */
      float wage;
      float hours;
      float overtime;
      float gross;
};
```

In your main function, define an array of structures, and feel free to initialize the clock and wage valu

Use the following information to initialize your data.

```
98401  10.60
526488  9.75
765349  10.50
34645  12.25
127615  8.35
```

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. Like the previous homework, use multiple functions in your answer and continue to use constants as needed. The only array you need is the array of structures, you don't need to create a separate array for clock, wage, hours, overtime, and gross. You can either pass the entire array of structures to each function, or pass values that work on one array

element (and its associated structure members) at a time ... either will work.

Templates have been provided this week if you would like to use them. As always, the templates are simply suggestions to help you get started. Feel free to improvise and seek out your own design and solutions as long as they meet the requirements specified in the assignment. Whatever way you do it, **don't use global variables** (unless for file pointers should you decide to utilize them) ... I don't want all your functions looking like:

```
void foo ()
```

... with no parameters passed or values returned. There are benefits and drawbacks to each of my provided code template approaches. Good luck with these templates, or feel free to create your own from scratch.

Intermediate Optional Challenge

For those of you more experienced programmers, continue with the challenge below that was offered in last week's assignment. The ***additional challenge*** this week would be referencing the information from *structures and/or array of structures*, as well as continuing to encapsulate the logic into *functions* that can be called to do this work.

- Calculate and print the ***total sum*** of Wage, Hours, Overtime, and Gross values
- Calculate and print the ***average*** of the Wage, Hours, Overtime, and Gross Values

Clock#	Wage#	Hours	OT	Gross
098401	10.60	51.0	11.0	598.90
526488	9.75	42.5	2.5	426.56
765349	10.50	37.0	0.0	388.50
034645	12.25	45.0	5.0	581.88
127615	8.35	0.0	0.0	0.00
Total	51.45	175.5	18.5	1995.84
Average	10.29	35.1	3.7	399.17

Advanced Optional Challenge

For those of you desiring even more of a challenge, and to really see the power and beauty of structures, complete the intermediate challenge AND add 5 more attributes (i.e., members) about an employee to your employee structure. Experiment with some different types for some of the attributes and look to make at least one of your member(s) a structure of another type. Use the three employee attributes below and feel free to add two more of your own choice:

- **Bonus** - A yearly dollar bonus, either $0 or some particular value, such as $1000.00, $5000.00, etc.
- **Start Date** - The date the employee started with the company. I would make this of type ***struct date***, like we did in the lecture notes.
- **Job Type** - An employee can be classified as either "Exempt" or "Non-exempt". Use a value of 1 for exempt and 0 for non-exempt. Employees whose jobs are governed by the Fair Labor Standards Act (FLSA) are either "exempt" or "nonexempt." Nonexempt employees are entitled to overtime pay, while Exempt employees are not.

  Given that, continue to determine overtime hours for all employees, but you will need to add code to determine whether a given employee is actually *paid* overtime or not. Next week we will change this member variable

type to utilize a character string (but just make it an *int*, *short in*t type for this week).

- ... I'll let you think of two attributes for an employee to give you a grand total of 5.

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.

Assignment 5 Code Templates

I've provided two templates that you are welcome to use for this week's assignment. One is passing the *address* of a single array of structures along with the array size to each function to peform a specific set of tasks. This is much easier than last week in that you only need to pass one array, an array of structures, rather than figuring out which set of arrays you need to pass to each function.

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: http://ideone.com/wjrONU

```
/*****************************************************************
**
** HOMEWORK: #5 Structures
**
** 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 */
```

```c
#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: 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: http://ideone.com/XXyaJ6

```
/*****************************************************************
**
** HOMEWORK: #5 Structures
**
** 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 */
```

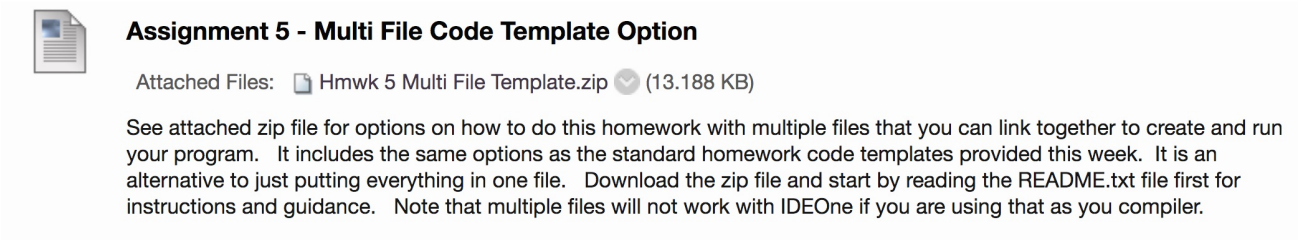
```
float Get_Hours (long int id_number);
void Output_Results_Screen (struct employee emp [ ], int size);

/* Add your functions here */

/* Add Function Comment Header for Get_Hours */
float Get_Hours (long int id_number)
{

    /* Add code as needed ... this is an example of a stub */

}


/*****************************************************************
** Function: Output_Results_Screen
**
** Purpose: Outputs to screen in a table format the following
** information about an employee: 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 }, /* initialize clock and wage values */
        { 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 5 - Multi File Template Option

Take the optional challenge this week and implement Homework Assignment 5 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 5 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.