Welcome to Week 9!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 9 folder**.
- No textbook readings this week, concentrate on the lecture notes.
- Begin **Quiz 9**. It is due Sunday at midnight.
- Begin **Assignment 8**. 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 9 Discussion Forum** located in the course **Discussion Board**. Please ask as many questions as needed, and don't hesitate to answer one-another's questions.
- Try out the various **code examples** in the lecture notes. Feel free to modify them and conduct your own **"What ifs"**.

"Mr Speck, our new warp drive program seems to have reached a limit in our run time area."

--- Mr. Sili
Navigation Officer, USS Enterprise

Advanced Pointers

In the previous lecture notes, we learned the basics about pointers. Now let's cover some advanced concepts, most of which are not even covered in the book!

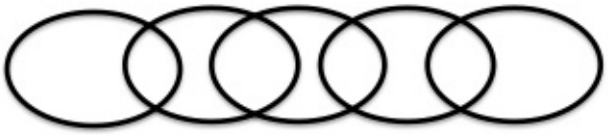
The main focus of this week will be on the topic of **dynamic memory allocation**. In our homework assignments so far, we have set an array at **compile time** to some specific size in the code. The problem with that approach is what would happen if you wrote a payroll program and tried to license it for use by multiple companies? Each company would have a different number of employees, and at times, would also increase or decrease staff. You don't want to have to **recompile** your program every time you have a change in your employee count. Likewise, you don't want to just make up a huge worst-case array size number, like a million, as there is likely not that many **consecutive** memory locations in either the **data** or **function stack frame** area to store information on all those employees.

I've seen how a few students in their homework assignments prompted for a variable to specify the number of employees at **run time**, and then used that variable to declare and specify an array:

```
int size; /* array size */

printf ("Enter array size: ");
scanf ("%i", &size);

int array [size];
```

That approach does not always work with every C compiler. Furthermore, you have the same issue with possibly not having enough **consecutive** space in the **data** or function **stack frame** runtime areas.

A solution that C provides is the ability to allocate space, on demand, when you are actually **running** your compiled program. We'll cover a **data structure** known as a **linked list** this week where we can individually allocate space in the **HEAP** area, and then use pointers to link and connect one employee to another. You could envision a linked list as a "chain" which each "link" stores information about an employee.
A **chain** is defined simply as a series of connected "links". You often see various metal chains at the local hardware or jewelry stores. Each chain link is often the same size and linked together to form a chain of some specific length. Below is my crude picture of a chain, but feel free to Google™ on images of chains if you wish.



Star Trekking - Running Out of Memory

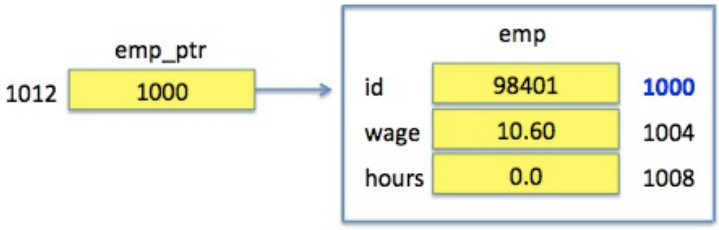
To wrap up the introduction for this week's topic, recall in our Star Trek movie last week where Mr. Speck suggested to Chief Engineer Mr. Scoot that using pointers will help dramatically improve the performance of his Warp Drive C Program. This week, apparently a bit intimidated by Mr. Speck, the chief engineer sends second officer Mr. Sili to discuss another problem the engineering team has encountered. It seems that their C program defined many large fixed arrays, and using them has had the adverse result of running out of space in the function stack frame area. Mr. Speck, not happy that their design did not handle this potential issue, suggests the use of dynamic memory allocation at run time along with using Linked Lists. Click the image below to check out what happens.

Pointers and Structures

Let's review how a pointer works with a structure, as it will be a key concept going forward this week. 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**

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


int main ()
{
    struct employee emp; /* holds employee info */

    /* a pointer to our structure */
    struct employee * emp_ptr;

    emp_ptr = &emp; /* point to address of emp struct */


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

       /* or */

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

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

    return (0);

}
```

Better illustrated, it shows a pointer variable **emp_ptr** 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 * operator. 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 emp in the memory pointed to by ***emp_ptr**. This would be legitimate if ***emp_ptr** had been set to point to anything. You must set a pointer before using it.

Structures containing Pointers

A structure can have members that are themselves pointers. Consider the code below which you can also access and try out at: http://ideone.com/23NlkI

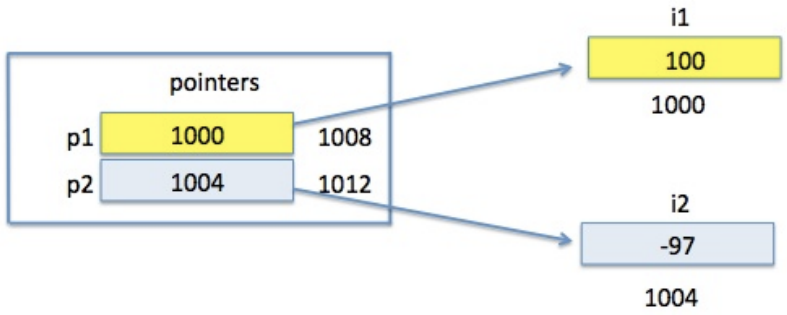
```
/* Declare two integers, initialize one to 100 */
int i1 = 100;
int i2;

/* note:  no tag, and the structure is inside main */
/* structure is not global.  It is available only to main */
struct
{
    int *p1;   /* this member is a pointer */
    int *p2;   /* and so is this one */

} pointers;

pointers.p1 = &i1;
pointers.p2 = &i2;

*pointers.p2 = -97;

printf ("%i %i \n", i2, *pointers.p2);
```

---

pointers is a structure containing 2 pointers, **p1** and **p2**. This example prints

-97 -97

It would look like this in memory:



Contrast:

*** (pointers . p2)** is the same as *** pointers . p2** ... p2 is a pointer member field in a structure

with ...

(* emp_ptr) . id is a pointer to a structure and then, by use of the "**.**" **operator** to a **member** called **id** in that structure.

Video - Introduction to Linked Lists

This video will introduce you to a data structure known as a linked list, which differs from an array in that space is dynamically allocated in the Heap area during RUN time. Linked Lists are great when you don't know how big an item might be up front. Watch this video to learn how to set up and use a linked list.

Video - Using a Linked List

This video is the exciting conclusion to our introduction to linked lists. You'll learn how to add additional nodes to a link list and indicate the end of the list. Once the list is built, you will then learn how to "traverse" each node in the link list to access and update member items.

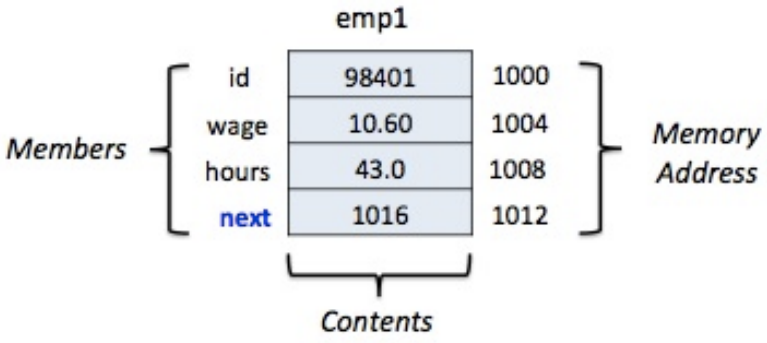
Linked Lists

Linked Lists allow you to put like items together which may or may not be consecutive in memory. You can navigate to **"nodes"** of information by using pointers. Items can be added or deleted from the list. The drawback is that you can only sequentially traverse from the current node to the next node. Therefore, if you were searching a list, you need to start from the beginning and check each node one at a time.

A **node** is essentially a structure containing one or more pointers that can point to other nodes, the sum total of the nodes will make up your "**linked list**". A structure type we will use for our nodes in the code example to follow is shown below. Note the **last** member called **next** is a pointer to a structure employee type itself. This is important as it will allow us to point to the "**next**" node in our linked list.

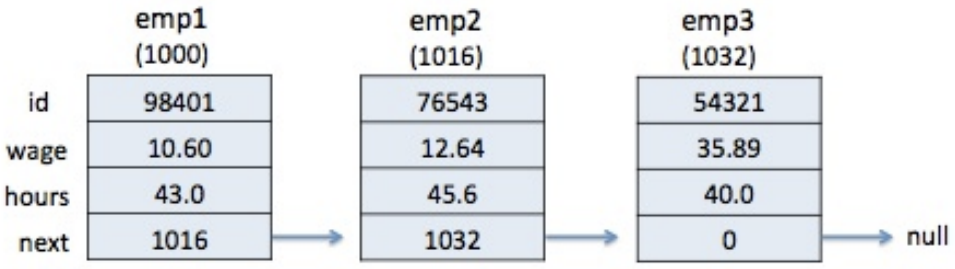
```
struct employee
{
   int id;
   float wage;
   float hours;

  struct employee * next;
} ;
```

To visualize our linked list node, consider the illustration below. I've added the **member names**, their **contents**, and **memory addresses**. I am going to assume that each member takes up four bytes of memory.



The diagram below shows how the linked list is set up once our program to follow is executed. Note that each linked list item (emp1, emp2, and emp3) has a **next member** which is **pointer** to an item of type **structure employee**. This allows each **node** to point to **another node** of that same type, thereby creating a list that is joined together by each node's **next** member pointer. The three variables are shown below along with their starting addresses in memory. While a structure contains many members, the starting address of a variable of any structure type is the starting address of the first member.

Notice how in the linked list example below, the **next** member points the starting address of another variable of type structure employee. Also note that in the last node (**emp3**), it has its **next** member pointing to **NULL** (or zero), which is a **FALSE** value. This allows one to traverse through the list, and knowing that when the **next** member value is 0, that it signifies the end of the list.



Let's look at a complete example of three linked list nodes that are strung together:

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

  struct employee * next;
} ;


int main ()
{

    /* initialize three variables of type struct employee */
   struct employee emp1 = { 98401, 10.60, 43.0 };
   struct employee emp2 = { 76543, 12.64, 45.6 };
   struct employee emp3 = { 54321, 35.89, 40.0 };

   /* emp1 points to emp2 */
   emp1.next = &emp2;

   /* emp2 points to emp3 */
   emp2.next = &emp3;

   /* the last one points to 0, to indicate the last list item */
   emp3.next = (struct employee *) 0;

   printf ("%i \n", emp1.next -> id);

   return (0);

}
```

The expression **emp1.next -> id** is equivalent to **(*emp1.next).id**

- It says to get what is in the **id** field of the structure pointed to by **next** pointer in the **emp1** structure.

The book has detailed info on examples like this.

- The problem is that you are limited to 3 employees. If you want to add more, you have to declare more variables.

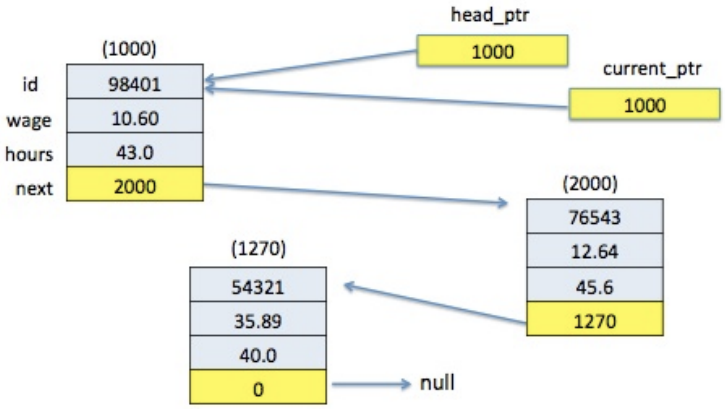
- C offers a better solution with **dynamic allocation** which allows you to allocate memory as needed during run-time.

Dynamically created Linked Lists

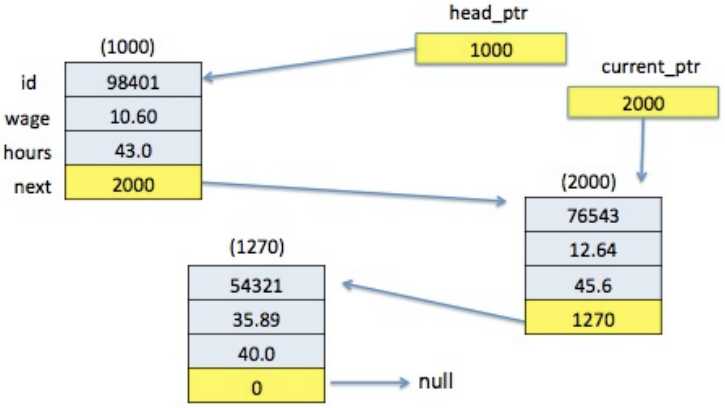
The **linked list** example we previously covered allowed us to connect three structure variables of the same type together. We could link them together in a variety of ways, but in the end, we had to specific three variables (emp1, emp2, and emp3) up front. This could be problematic if we had to process one thousand employees or more, we certainly don't want to specify a thousand different variables. Thankfully, C provides the ability to **allocate** a set of **nodes on demand** through a set of *dynamic memory allocation library functions*.

Before we discuss exactly how in C we implement a linked list with dynamic memory allocation, let us first visually present how it could be set up and accessed. In the diagram below, we have **three linked list nodes** and **two pointers**, a **head pointer** that will always point to the first node in a linked list, and a **current pointer** that will point to the current node being accessed (to update, examine, print, etc.). Unlike the two pointer variables that you specify in your code at "compile time" which will likely be stored in a function stack frame area, the three linked list nodes don't have to be specified up front in your initial program. You can call dynamic memory functions in C when running your program (i.e., at run time) to allocate as many linked list nodes as needed in the **Heap** area.

Notice how the first linked list node starts at location **1000**, then it points to linked list node at location **2000**, which in turn points to a linked list node at location **1270**, and that last node then points to **null**. These nodes don't actually have up front variables names assigned to them, they are simply allocated and referenced with pointers to comprise our linked list.

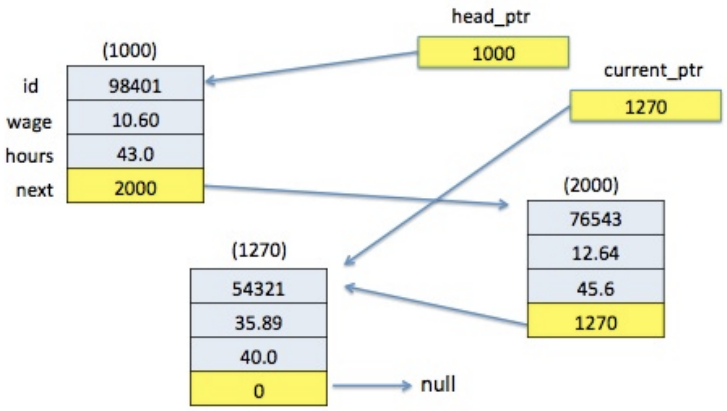


The idea is that you always point the **head pointer** (head_ptr) to the first linked list node, and it does not change unless you want to indicate another linked list node is the first node in the linked list. The **current pointer** (current_ptr) can be updated as needed to allow for traversal through the linked list one node at a time. In the scenario below, we moved the current pointer from the first linked list node to the second linked list node.

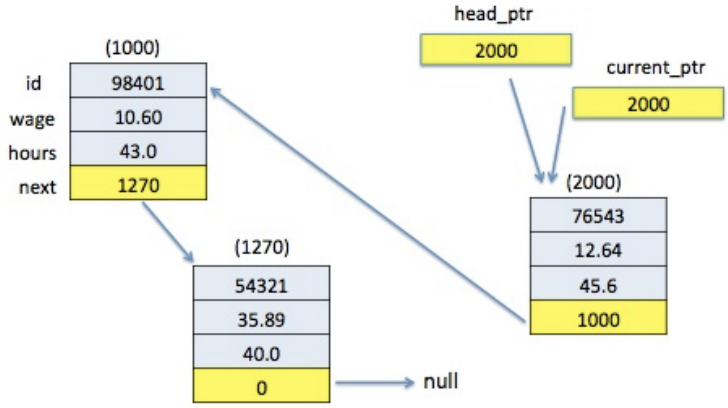


The next diagram shows how you set the current pointer to traverse to the next node in our list, which is the third and

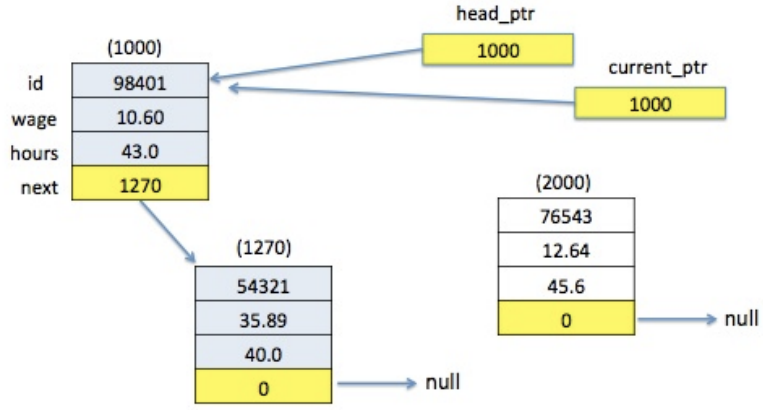
final linked list node. That node has a next member pointer **value of 0 (or null)**, which evaluates to a FALSE value in C. This is useful as it indicates we are at the **end of our linked list**.



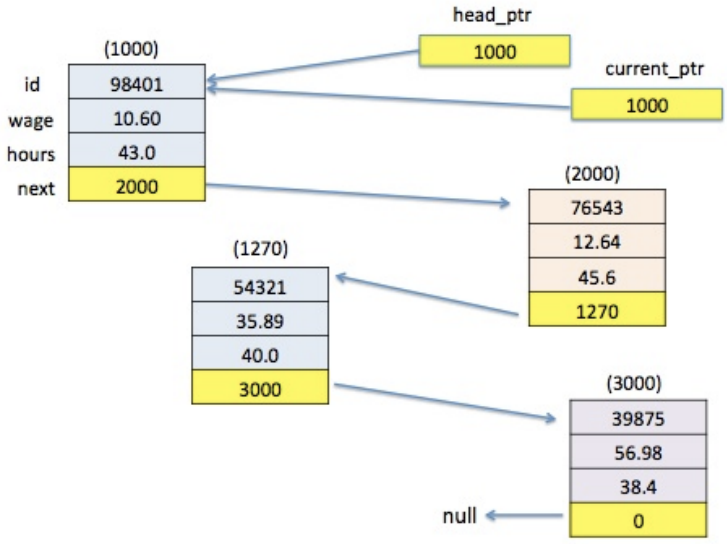
Finally, let's consider another scenario where later on we may want to change the order of our linked list. This is very easy as we can set both the **head** and **current pointer** to the node that we want to be the beginning of our linked list (in this case, the one that starts at location 2000) and then rearrange the next pointers as needed in each node.



We can also **delete** a node from a linked list by just updated the pointer references in the next member variable in various nodes. In the example below, the linked list node that starts at memory location 2000 is no longer part of our linked list.



Of course, at any point in our program, we can decide to bring back previously allocated notes back into our list (such as the one at memory location 2000 shown in orange ... it is still there), or create new nodes to solve a "just in time" need (for example, one at location 3000 shown in purple).



Understand that you can insert or delete nodes in your linked list to order it any way you desire. In most real life applications, programmers will keep a linked list sorted in a particular order to facilitate printing or searching. There is a lot you can do with linked lists. This class will just scratch the surface and the plan here is to provide a general overview with sufficient code detail so we can show you how to better work with pointers and dynamic memory allocation. If you take a **Data Structures** course in the future, you can learn how to **delete** nodes, **insert** nodes (at the beginning, the end, or somewhere in-between), **update** nodes, **search** lists, **sort** linked lists, as well as many other capabilities.

Also note that there is a data structure called a **doubly linked list**, which has two pointers ... a **next pointer** to the next node, and **pred pointer** to the previously accessed node. This allows for traversal in both directions, **forward** and **backward**. We will just cover what is known as a **single linked list** in this class, which allows for only a forward traversal.

Summary

- **Linked lists** are useful in dealing with data where you don't know up front how many nodes of information you will need to allocate.
- Linked lists created **dynamically** are stored in the **Heap** memory area, which we covered in our previous discussion of the C Run Time Environment.
- Each individual node that comprises a linked list does not have to be stored consecutively in memory like an **array element**
- By contrast, the size of an **Array** is fixed at compile time, and each element must be consecutive in memory
- The main **drawback** of a linked list is that you can only traverse and search through the list one node at a time. The worst case is the item you are looking for in at the end of the list, or not in the list at all.

In the lecture notes that follow, we will learn how to **dynamically allocate nodes in the Heap** during **run time** to create and extend a linked list.

Dynamic Memory Allocation

You have the ability to request more storage area for your variables at **run time** rather than having to guess the worst case at **compile time**. The area that you request is found in the heap (refer to our C run time environment notes).

Think about our homework problem where we have to hard code via a constant or a user prompt the array size of our employee structures. No matter how you did it, you always have to put a worst case value (most of you put 5) as the array size.

Let's look at some of the main C library functions to utilize.

```
calloc (n, size)
malloc (size)
sizeof (type)
free (pointer)
cfree (pointer)

char *calloc(number,size)
unsigned number, size;

char *malloc (size)
unsigned size;
```

Here is a summary of 5 new items in C:

Function	Description
calloc	gets n chunks of size area is initialized to 0 returns char pointer to start of area you must cast the returned pointer to the right type returns zero if not able to allocate requested area
malloc	gets requested number of bytes area not initialized returns char pointer to the start of the area you must cast the returned pointer to the right type returns zero if not able to allocate requested area
cfree	gives back the area pointed to by **calloc** pointer
free	gives back the area pointed to by the **malloc** pointer
sizeof	compile time function figures out how many bytes something is

Dynamic Memory Allocation Examples

Shown below are examples using the **calloc**, **malloc** and **sizeof** library functions. In combination, each is critical to working with dynamic memory allocation in the C Programming Language.

calloc example

```
int *int_ptr;         /* pointer to an integer */
char *calloc ( );  /* system default of the calloc library function */
                      /* note the char return type */

int_ptr = ( int * ) calloc ( 5, sizeof ( int ) );
```

malloc example

```
int *int_ptr;           /* pointer to an integer */
char *malloc ( );     /* system default of the calloc library function */
                      /* note the char return type */

int_ptr = ( int * ) malloc ( 5 * sizeof ( int ) );
```

Both functions will allocate the desired amount of space in the heap area. A value of 0 (FALSE) will be returned if the memory can't be allocated (such as the situation where the heap area is full). The space allocated has to be **consecutive** in memory (it's like one big block). A way to check this is:

```
if ( int_ptr = ( int * ) malloc ( 5 * sizeof ( int ) ) == ( int * ) NULL )
   printf ("Unable to allocate memory\n")
   /* otherwise, int_ptr contains a starting address in the heap */
```

OR YOU CAN DO IT WITH MULTIPLE STATEMENTS ...

```
int_ptr = ( int * ) malloc ( 5 * sizeof ( int ) );   /* request memory allocation */

if ( int_ptr  == ( int * ) NULL )     /* unable to allocate 5 integers ? */
   printf ("Unable to allocate memory\n")
```

In the above statement, malloc will return a pointer to **int_ptr**, which will then be check to see if it is 0. The **NULL** value is a **constant** for the value 0 and is found in **stdio.h**. It is casted because i**nt_ptr** expects either a integer pointer value or address.

Both **calloc** and **malloc** return a pointer to **char**, so the function must be declared prior to use:

```
char *calloc ( );
char *malloc ( );
```

Notice the casting: **(int *)** malloc of the return type. int_ptr is a pointer to an **integer value** and can not point to a character location. By casting the return type to an integer pointer rather than a character pointer, the statement will compile and execute properly.

In both cases, a block of memory will be allocated and typed correctly into the heap area. Note that **int_ptr** will be set to the first memory location. The pointer then can be moved by simply modifying the pointer (increment, decrement, etc.).

int_ptr will look like

address	contents

8000	1000

... If I **incremented** the pointer by one (++int_ptr), it would now point to the **next integer location**, 8004 (assuming an integer is 4 bytes).

If space was allocated using **calloc** (note that values are initialized to 0):

address	contents
8000	0
8004	0
8008	0
8012	0
8016	0

If space was allocated using **malloc** (note the undefined values):

address	contents
8000	undefined
8004	undefined
8008	undefined
8012	undefined
8016	undefined

Dynamic Memory Allocation Example

Below is a great example that will help you get started with Dynamic Memory Allocation using a Linked List in C. Please use it as a guide to your last homework assignment 8. A template is also provided in the homework assignment itself, but it is very similar to what is shown below. Relax, its OK if you use this code as is and then expand from it ... you have my permission to use any and all of it. For now, just review the code below to understand how it works given the sample input values and output.

Feel free to try it out in IDEOne at: https://ideone.com/N7pymq

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

struct employee
{
    int id;
    float wage;
    struct employee *next;
};

/*--------------------------------------------------------------
**
** FUNCTION: print_list
**
** DESCRIPTION: This function will print the contents of a linked
**    list. It will traverse the list from beginning to the
**    end, printing the contents at each node.
**
** PARAMETERS: empl - pointer to a linked list
**
** OUTPUTS: None
**
** CALLS: None
**
**------------------------------------------------------------*/

void print_list(struct employee *empl)
{
    struct employee *tmp; /* tmp pointer */
    int i = 0;            /* counts the nodes printed */

    /* Start a beginning of list and print out each value */
    /* loop until tmp points to null (0 or false) */
    for(tmp = empl; tmp ; tmp = tmp->next)
    {
        i++;
        printf("\nEmployee ID: %6i, Wage: %8.2f\n",tmp->id, tmp->wage);

    }

    printf("\n\nTotal Number of Employees = %d\n", i);

}

/*--------------------------------------------------------
**
** FUNCTION: main
**
** DESCRIPTION: This function will prompt the user for an
**    employee id and wage until the user indicates they are
**    finished. At that point, a list of id and wages will be
**    generated.
**
** PARAMETERS: None
**
** OUTPUTS: None
**
** CALLS: print_list
**
**------------------------------------------------------*/
```

```
int main ()
{

    char answer[80];   /* holds user reply to prompt */
    int more_data = 1; /* flag to continue processing */
    char value;        /* holds first character of user reply */

    struct employee *current_ptr; /* pointer to current node*/
    struct employee *head_ptr;    /* points to first node */

    /* Set up storage for first node */
    head_ptr = (struct employee *) malloc (sizeof(struct employee));
    current_ptr = head_ptr;

    while (more_data)
    {

        /* Read in Employee ID and Hourly Wage */
        printf("\n\nEnter employee ID: ");
        scanf("%i", & current_ptr -> id);

        printf("\nEnter employee weekly wage: ");
        scanf("%f", & current_ptr -> wage);

        printf("\n\nWould you like to add another employee? (y/n): ");
        scanf("%s", answer);

        /* add another employee? */
        if ((value = toupper(answer[0])) != 'Y')
        {
            current_ptr->next = (struct employee *) NULL;
            more_data = 0;
        }
        else
        {
            /* create new node, set next pointer to it */
            current_ptr->next = (struct employee *) malloc (sizeof(struct employee));

            /* point to the new node */
            current_ptr = current_ptr->next;
        }
    } /* while */

    /* print out current contents of linked list */
    print_list(head_ptr);

    printf("\n\nEnd of program\n");

    return (0);
}
```

Sample Input:

```
98401
10.60
Y
23123
9.75
N
```

Sample Output:

```
Enter employee ID: 98401
Enter employee weekly wage: 10.60

Would you like to add another employee? (y/n): Y

Enter employee ID: 23123
Enter employee weekly wage: 9.75

Would you like to add another employee? (y/n): N

Employee ID: 98401, Wage: 10.60
```

```
Employee ID: 23123, Wage: 9.75

Total Number of Employees = 2

End of program
```

Returning Pointers for a Linked List

The code below shows a design for how you could traverse a linked list searching for a specific value in one of the linked list node's members. If found, a pointer will be returned indicating the linked list node that has that value, otherwise a **null value** pointer would be returned indicating that the item was not found. Note that a **null value pointer** would be considered FALSE, and that is something the calling function can key off and process as needed. The NULL symbolic constant can be found in the stdio.h header file.

You can watch it integrated with our previous program at: https://ideone.com/kBdqcv

```
/*****************************************************************************
**
** Function: findEntry
**
** Description: Using list_ptr as a starting point, examine every entry in
** the list until an entry is found that has an id_number which
** matches the one passed to the function. If found, return a pointer
** to the node, otherwise, return NULL.
**
** Parameters:
**
**     list_ptr - pointer to the beginning of the link list
**     id_number - employee identifier to search
**
** Returns: Pointer to linked list node with the ID (otherwise NULL returned)
**
*****************************************************************************/
struct employee * findEntry (struct employee * list_ptr, int id_number)
{

    struct employee * found_it_ptr = list_ptr;  /* set working pointer to beginning of list */


   /* search through linked list, return a pointer to the node if the ID is found in it */

    for ( ; found_it_ptr; found_it_ptr = found_it_ptr -> next)
    {

        if ( found_it_ptr -> id == id_number )
            return ( found_it_ptr );   /* found */

    }


    return ( (struct employee *) NULL );   /* not found, return NULL pointer */

};   /* find entry */
```

You might call it this way, assuming **id_value** is a valid integer variable in your program. Also, notice how "**type casting**" for **NULL** is used in the the conditional statement below which I highlighted in **bold**.

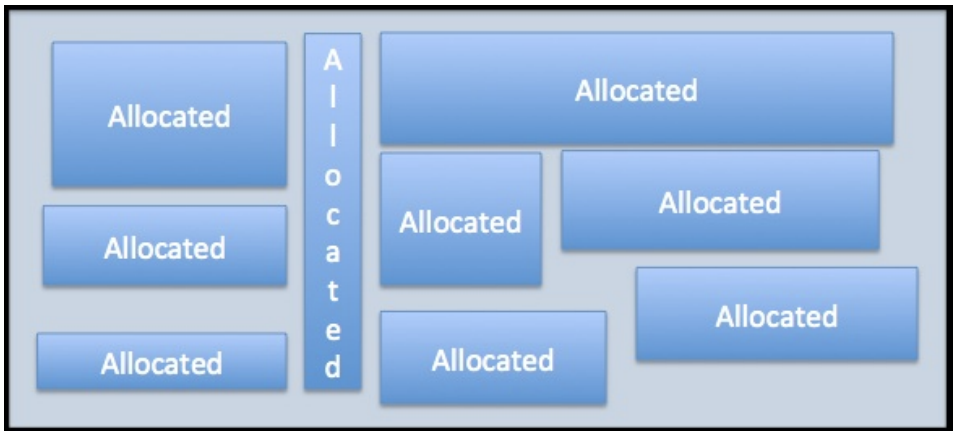
```
/* Enter an ID to search for in our list */

printf ("Enter an ID to search for in our list: ");
scanf ("%i", &id_value);

/* Search for by passing a pointer to beginning of the list */

/* which is typically found in our examples in "head_ptr" */

if ( findEntry (head_ptr, id_value) != (struct employee *) NULL )
{
   printf ("ID is in the list \n");
}
else /* NULL Pointer returned */
{
   printf ("ID is NOT in the list \n");
}
```

Video - Linked List Memory Management

Linked Lists are dynamically allocated and stored in the Heap Data Area which does have a limit in terms of how much memory can be allocated into it. It is important to be a "green programmer" and free up space in the Heap whenever possible so that space is available in the future throughout the life of your program.

Freeing Memory

One common error with both new and experienced programmers is forgetting to release memory allocated in the heap area when it is no longer needed. While the heap is a dynamic memory area, it does have an upper limit and can be filled up. When you allocate 100 bytes of memory with a malloc or calloc statement, there must be 100 consecutive bytes available. If not, the memory allocation will not occur, and your program will almost certainly stop working correctly. If your program completes execution, the resources are automatically freed, but many times issues arise in longer running programs with many functions allocating memory using the malloc or calloc library calls. Below is a simplistic picture of what a heap memory area might look like as it is getting full.



There are two library functions in C that will release allocated memory and make it available for use to be reallocated in the future. If you used **malloc** to allocate your memory, then you must use the **free** function to release it. Likewise, if you use **calloc** for memory allocation, it must be released with the **cfree** function.

```
free (int_ptr); /* pointer returned by malloc */
cfree (int_ptr); /* pointer returned by calloc */
```

NOTE: It is very important that the pointer address argument used for the free function (or cfree) contains the same pointer address value returned by malloc (or calloc).

REALITY NOTE: For your small homework in our class, you are not going to run out of memory. However, in the real world, it's important to realize that there is a limit with heap space when you work with dynamic memory allocation at run time. Many systems work in real time, meaning they could be running constantly over many days. In these situations, it is extremely important to properly manage your unused or un-needed memory allocation to keep things running and executing smoothly. You might hear the word "**Garbage Collection**" in your travels and this alludes to the need for programs to periodically clean up resources (like memory allocation areas in the heap that are no longer needed).

Pointers and Functions

Pointers are passed by value, but allow access to variables in the calling function because the value is an address. Note that the argument does not contain the indirection (*) operator. You are passing the contents of the pointer, which contains an address to the location it is referencing. Try it out at: http://ideone.com/rr6YWV

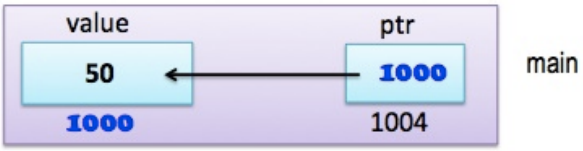
```
#include <stdio.h>

/*****************************************
**
** Function:  test
**
** Description - Demonstrates how a pointer is passed
**               to a function.  The pointer will update
**               a value it is referencing in the Calling
**               Function.
**
** Parameters:  int_ptr - a pointer to an integer
**
** Returns:  void
**
*****************************************/

void test (int * int_ptr)   /* address passed into the pointer parameter */
{
    *int_ptr = 100;   /*  2 */
}

int main ()
{

    int value = 50, * ptr = & value;   /*  1 */

    test ( ptr );   /* pass the contents of the pointer */

    printf ("value = %i \n", value );  /* value will now be 100 */

    return (0);
}
```

A good way to understand exactly what is happening here is by looking at contents of each **local variable** and **parameter** within the **function stack frame**.

Let's look at the stack frame

1) Initial State:

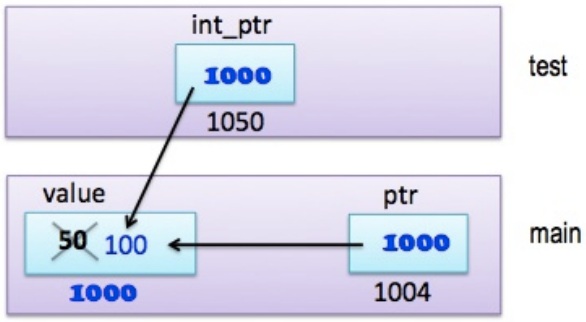
Only the main function is active, so only the stack frame for it is shown below. Note that **value** is initialized to 50 and the pointer variable named **ptr** has been set to the address of the variable **value**.



---

2) In call to test function, after *int_ptr = 100 statement:

Note how both pointers, the **parameter** in the test function (**int_ptr**) and the **local variable** in main (**ptr**) are both pointing to address location 1000, which happen to be where the l**ocal variable** name **value** resides in the **main function**. Using either pointer will change the contents of the local variable **value**.



Pointers to Pointers

A **pointer** is special type of variable in that it can store the **address** of another variable. So far, we have shown how it can point to integers, floats, characters, and even structures. A pointer itself is a variable, so by that logic, a pointer can also hold the address and point to another variable which happens to be a pointer. In this instance, we refer to that variable as a **pointer to a pointer**.

Let us suppose we have a pointer named **ptr_ptr** that points to yet another pointer called **int_ptr** that points to an integer variable that goes by the name of **int1**. Let's also add another simple integer value called **int2**.

```
int int1 = 25;          /* simple integer, initialized to 25 */
int int2;               /* another simple integer, undefined */
int * int_ptr;          /* pointer to an integer */
int ** ptr_ptr;         /* pointer to pointer that points to an integer */
```

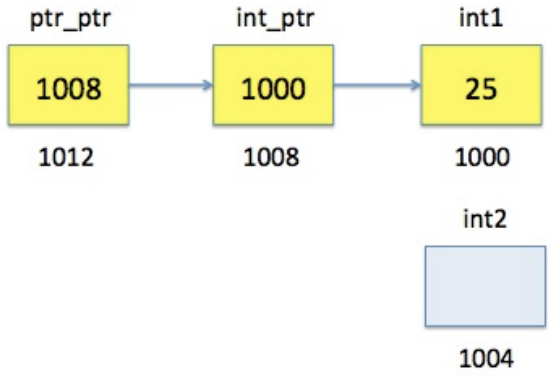
To make i**nt_ptr** point to the integer address location of **int1**:

```
int_ptr = &int1;
```

To make **ptr_ptr** point to the pointer address location of **int_ptr**:

```
ptr_ptr = &int_ptr;
```

In memory, the four variables can be visualized as:



Below is a complete program that shows how a **pointer** and a **pointer to pointer** can be set up and utilized. Feel free to try it out at: http://ideone.com/b4uxYY

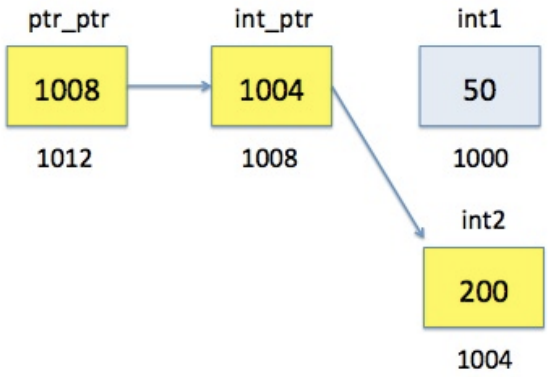
```
/* A pointer can be used to point to other pointers */
#include <stdio.h>
int main ()
{

    int *int_ptr;           /* pointer to an integer */
    int **ptr_ptr;          /* pointer to a pointer that points to an integer   */
    int int2, int1 = 25;    /* simple integer variables which can be pointed to */

    int_ptr = &int1;        /* sets int_ptr to point to an integer */
    ptr_ptr = &int_ptr;     /* sets ptr_ptr to point to the pointer int_ptr */

    printf (" int1 = %d \n", int1);    /* access the variable int1 directly */
    printf (" *int_ptr = %d \n", *int_ptr);     /* access value indirectly through int_ptr */
    printf (" **ptr_ptr = %d \n", **ptr_ptr);   /* access the same value through ptr_ptr */

    /* ptr_ptr is of type pointer to pointer to int. Applying the indirection operator  */
```

```
        /* to this expression (*ptr_ptr) results in one of type pointer to int.  */

        /* Applying it once again (**ptr_ptr) yields an expression of type int. */

        **ptr_ptr = 50;     /* same as saying: *int_ptr = 50; or int1 = 50; */

        int_ptr = &int2;    /* if you wanted int_ptr to point to int2  */
        *ptr_ptr = &int2;  /* change value of int_ptr indirectly though ptr_ptr. */

        *int_ptr = 100;     /* change int2 to 100 indirectly using a pointer   */
        **ptr_ptr = 200;   /* change int2 to 200 indirectly by ptr_ptr          */

        printf ("\nThe final values of all variables: \n\n");

        printf (" int1 = %d \n", int1);              /* access variable int1 directly */
        printf (" int2 = %d \n", int2);              /* access variable int2 directly  */
        printf (" *int_ptr = %d \n", *int_ptr);      /* access int2 indirectly through int_ptr  */
        printf (" **ptr_ptr = %d \n", **ptr_ptr);  /* access value indirectly through ptr_ptr */

        return (0);
    }
```

The output of the sample program would be:

```
int1 = 25
*int_ptr = 25
**ptr_ptr = 25

The final values of all variables:

int1 = 50
int2 = 200
*int_ptr = 200
**ptr_ptr = 200
```

Just before the program ends, you could visualize the final values of the variables:



There is one other way to use a pointer to a pointer, if you tried to reference:

```
    * ptr_ptr
```

It would go only one level and access the **contents** of **int_ptr**, which is a pointer to an integer, and currently contains the address of **int2**, which happens to be the address **1004** in our diagram. Of course, it you just referenced:

ptr_ptr

It would reference the **contents** of **ptr_ptr**, which contains the address of **int_ptr**, which happens to be address **1008** in our diagram.

You can get very crazy here with pointer to pointers. Pointers can go as **many levels** as you like. Just make sure you use the correct number of asterisks in the declaration, and when accessing the value that the pointer ultimately points to. In an upcoming lecture note, we'll cover how they are most commonly utilized, to be passed as **arguments** to a function where they can access and modify the contents of the pointer they are referencing that resides in the calling function. Don't be alarmed if you don't understand what was just stated, all will be explained in great detail going forward.

Pointer to Pointer Examples

The most common way to use a **pointer to a pointer** is by passing it as an argument to function. Remember that pointers are **passed by value**. If you just pass a pointer to a function and update it the **called function**, its contents will **only be updated** within that **called function.** Don't confuse this with changing values that a pointer actually points to, we are talking about the pointer itself. When the called function completes, its memory is wiped off the function stack frame. With a **pointer to a pointer**, it will actually update the pointer value in the **calling function**. Below are two functions, one that is passed a pointer and another that is passed a pointer to that pointer. Review closely how each works and note that only one function actually updates the pointer in the **calling function** (in this case, main).

Try out the code below at: http://ideone.com/1agxc7

```
#include <stdio.h>

/* global variables */
int x = 10;
int y = 20;

/*****************************************************************************
**
**  Function:  no_update_ptr
**
**  Description:  Receives a pointer and updates it.   However, the
**                    contents of the pointer while changed in this
**                    function remain unchanged in the calling function.
**                    This shows that pointers are passed BY VALUE.
**
** Parameters:  ptr - a pointer to an integer
**
** Returns:  void
**
*****************************************************************************/

void no_update_ptr (int * ptr)
{
     ptr = &y;  /* no change to ptr in main */
}

/*****************************************************************************
**
**  Function:  update_ptr
**
**  Description:  Receives a pointer to pointer and uses it to
**                    update the contents of the pointer it is referencing.
**                    The result is that the pointer will be updated in the
**                    calling function.
**
**                     Pass pointers to pointers to functions to change
**                    the contents of the pointers they are pointing to
**                    in both the called AND calling functions.
**
**  Parameters:  ptr - a pointer to a pointer that points to an integer
**
**  Returns:  void
**
```

```
*****************************************************************************/

void update_ptr (int ** ptr)
{
     *ptr = &y;  /* this will actually change ptr in main */
}

/*****************************************************************************
**
**  Function:  somefunc
**
**  Description:  Receives a simple integer, a float pointer,
**                and an integer pointer.  Shows that only
**                pointer variables will update corresponding items
**                they are pointing to in the Calling Function.
**
**  Parameters:  a - simple integer value
**                b - pointer to a float
**                c - pointer to an integer
**
**  Returns:  void
**
*****************************************************************************/
void somefunc (int a, float *b, int *c)
{
    a = 1;
    *b = 2.2;
    *c = 3;
}

int main ()
{
     int a, c, *ptr;  /* two integers and an integer pointer */
     float b;         /* floating point variable */
     ptr = &x;        /* point to global integer variable called x */

     a = 0;
     b = 0;
     c = 0;

     somefunc(a, &b, &c);    /* call to see what changes upon return */

     /* Note that only b and c were updated, not a */
     printf ("a = %d, b = %f, c = %d \n", a, b, c);

     printf ("before calls, *ptr = %d\n", *ptr);

     no_update_ptr (ptr);  /* nothing changes to what ptr points to */

     printf ("*ptr = %d after call to no_update_ptr \n", *ptr);

     update_ptr (&ptr);   /* passing address of the ptr can change ptr upon return */

     printf ("*ptr = %d after call to update_ptr \n", *ptr);

     return (0);

}
```

OUTPUT:

```
a = 0, b = 2.200000, c = 3
before calls, *ptr = 10 *ptr = 10
after call to no_update_ptr *ptr = 20
after call to update_ptr
```

Pointers to a Function

The most common use of **function pointers** is to pass them as arguments to other functions. The example below shows how to set up and use a function pointer. This topic is quite advanced, but it is presented in case you encounter function pointers in the development of future projects. We will not be using them on any assignments or expect you to work with them on exams. Below is a sample program that defines a function, sets a pointer to it, and then use that pointer to call the function. Feel free to try it out at: http://ideone.com/lWrVUm

```
#include <stdio.h>

/* First, define the function to be pointed at ... */

// ********************************************
// Function:  sum
//
// Description:  Returns the sum of two integers
//
// Parameters:
//
//   num1 - an integer number
//   num2 - another integer number
//
// Returns:  The sum of the two parameters
// ********************************************

int sum (int num1, int num2)
{
    return (num1 + num2);  /* return the sum */
}

int main ()
{
    /* Second, define a pointer to a function which */
    /* receives 2 integer values and returns an int */
    int ( *functionPtr ) (int, int);

    /* point to address (note the &) of our sum function */
    functionPtr = &sum;

    /* Finally, let's actually use our function pointer */
    int mySum = (*functionPtr)(5, 10); /* the sum is 15 */

    printf ("mySum is %i \n", mySum);

    return (0);
}
```

The Output would show:

```
mySum is 15
```

Here are some common examples of how **function pointers** are utilized:

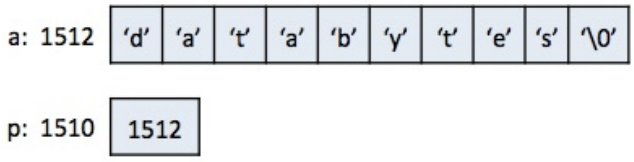
Example	Means
int *somefunct ();	declares a function that returns a pointer to int
int (*funct_ptr) ();	declares a pointer to a function that returns int
funct_ptr = &name_of_funct;	assigns address of the function to pointer
result = (*funct_ptr) (arg1, arg2, ... argn);	parenthesises required to apply * before call

Advanced Pointer Concepts

The following is adapted from the book "*Advanced C: Tips and Techniques*" by Paul and Gail Anderson.

There are many concepts here that will probably hurt your head, but it is good to know where all these crazy derivations that I've been talking all semester originally came from ... Enjoy :)

```
char a [  ] = "databytes";    /* array of characters */
char *p = a;                  /* pointer to the array */
```



The compiler treats array names as constants. This means a may be used anywhere a character pointer is legal, except on the left hand side of an assignment statement or with an "databytes" on separate lines:

```
/* C converts the array name to a pointer */

printf ("%s \n", a);

/* p is already a pointer. The compiler fetches the */
/* same address (1512) from the contents of p's memory */
/* memory location (1510) */

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

Pointer offsets use array names or pointers. These statements display the string "bytes" on separate lines using pointer offsets:

```
/* a + 4 points to the fifth character of the string. */
/* The compiler computes address 1516 from the constant address */
/* address 1512 and the constant 4. This is the address passed */
/* to printf( ) */

printf ("%s \n", a + 4)
```

Likewise, the statements display the fifth byte of the string ('b'):

```
/* direct access to array */
putchar ( a [ 4 ] );

/* indirect access using a pointer */
putchar ( p+4 );
```

The examples suggest a close relationship between arrays and pointers. In fact, the compiler treats arrays names as pointers most of the time. To fully understand this relationship, we present the following:

THE BASIC RULE

```
a [ i ] = ( * ( a + i ) )
```

a is an array of elements
i is an integer expression

Simply stated:

The basic rule says referencing the i'th element of an array is equivalent to applying the indirection operator (i.e., '*') to a pointer offset from the beginning of the array. This implies that anywhere an array reference is legal, we may substitute an equivalent pointer expression. The basic rule is independent of *a's* pointer type.

EXAMPLE:

```
/* twist.c - strange array reference */

main ()
{
   static char a [ ] = "0123456789";  /* test string */

   int i = 5;  /* simple initialized local variable */

   printf ("%c\n", a [ i ] );    /* sixth element */
   printf ("%c\n", i [ a ] );   /* What is this? */
}
```

OUTPUT:

```
5
5
```

The basic rule explains what's going on. C converts the array reference

```
a [ i ] to ( * ( a + i ))
```

which is equivalent to **(* (i + a))** that implies the array reference **i [a]**

This proves that array offsets really don't exist because C converts array references to pointers offsets.

The basic rule also helps to derive equivalent pointer expressions from array references.

```
a [ i ] = ( * ( a + i) )
&a [ i ] = & ( * ( a + i)   /* apply the & operator to both sides */
```

C precedence rules **evaluate** & and * from **right to left**, so remove the outer parentheses to leave up with:

```
&a [ i ] = & * (a + i)
```

which reduces to

```
&a [ i ] = a + i
```

This proves that it doesn't matter whether you use an array name or a pointer with the pointer offset.
The following program proves this. It shows that it doesn't matter whether you use an array name or pointer to display the same address.

```
/* a1.c - shows addresses and indirection operators */

#include <stdio.h>
 char a [ ] = "databytes";
 char *p = a;    /* notice I omitted the & operator: &a */
 int main ( )
 {
   /* using the array */
    printf ("%x %x\n", a + 3, &* (a + 3));
```

```
    /* using the pointer */
    printf ("%x %x\n", p + 3, &* (p + 3));

    return (0);

}
```

OUTPUT:

```
142d 142d
142d 142d
```

Here's another derivation from the basic rule that applies to the first element of any array:

a [i] = (* (a + i))

a [0] = (* (a + 0))

a [0] = * (a + 0)

a [0] = *a

This says that ***a*** points to the first element in the array

Using the previous technique, let's apply the address operator to both sides of the last line:

& a [0] = & * a

& a [0] = a

The final derivation is a combination of previous ones:

a [i] = (* (a + i))

&a [i] = & (* (a + i))

&a [i] = a + i

&a [i] = & a [0] + i

The last two lines show that equivalence. In a C program, you may the expressions

a + i

and

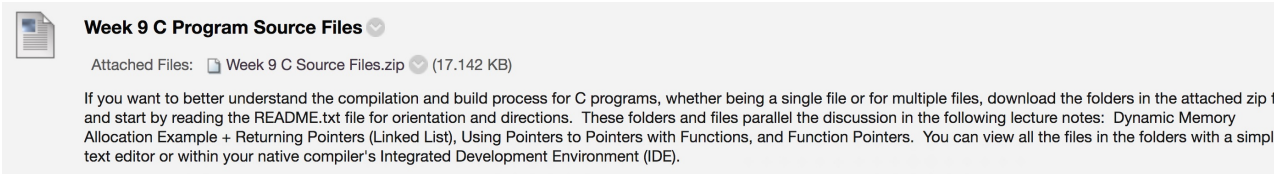
&a [0] + i

interchangeably. The compiler generates the same assembly code in both cases.

Week 9 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.

Summary

That's all for this week, and it is probably our toughest class week to comprehend. Read the pointers chapter again and also quickly look at some items from the last couple of chapters (on the topic of dynamic memory allocation). The best way to learn is to try out the examples and start doing the last homework assignment with linked lists and pointers.

To help with Assignment 8, use the **template provided** in the assignment area. Feel free to use my code as a starting point and add functions and code as needed, including adding members to the structure. Of course, the videos this week should also be useful to help you visualize what is happening both with linked lists and dynamic memory allocation.

As always, there is also a **Quiz** for you to complete by next week as well.

For Next Week

The next and final class will selectively complete the rest of the chapters in the textbook. We concentrate for the most part on the C Preprocessor, in particular *macros*, *enumerated types*, *conditional compilation, and static* and *dynamic libraries*. Other topics to be covered include *typedefs*, working with *multiple files*, *unions*, and a few other surprises.

HOMEWORK 8
DYNAMIC LINKED LISTS WITH POINTERS

Write a C program that will calculate the gross pay of a set of employees. Note that there are two items that will help you with this assignment.

1) Please use the template provide in the Homework 8 folder. Use this template "as is" to jump start your success in completing this homework. It should compile and run right out the box.

2) Watch the videos this week to help you better visualize and understand how the template code works. Also, read through and make sure you understand the concepts in the lecture notes.

WHAT YOU NEED TO DO:

The program should prompt the user to enter the number of hours each employee worked. The output should be similar to your previous homework.

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.

DO continue to use features such as functions and constants, and continue to follow our coding standards.

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

```
struct employee
{
   char first_name [10];
   char last_name [10];
   int id_number;          /* use can use long int if you wish */
   float wage;
   float hours;
   float overtime;
   float gross;

   struct employee *next;
};
```

Create a linked list of structures using the following data:

```
Connie Cobol 98401 10.60
Mary Apl 526488 9.75
Frank Fortran 765349 10.50
Jeff Ada 34645 12.25
Anton Pascal 127615 8.35
```

Unlike previous homework, you need to prompt the user for all of the above information, ... and you still need to prompt for the hours worked for each employee.

Hint: Use one or two **scanf** statements to read in the first and last names with the %s format.

Get the data above from the terminal, and for each one:

- get dynamic memory, using **malloc**, for an employee node
- put the employee data in the dynamic memory node
- link the nodes with pointers in the above order

After the list pointers are in place you must get at the second and later instances of the structure by going from the first structure down the chain of list pointers.

Then, for each employee, read in the hours worked from the terminal. Do all appropriate computations, and write out the table.

You do not need an array of structures like you used in homework 6 and 7. Use the template and dynamically allocate linked list nodes as needed. Similar to the previous homework assignment:

a) Add a **Total** row at the end to sum up the hours, overtime, and gross columns
b) Add an **Average** row to print out the average of the hours, overtime, and gross columns.

Your code should work for any number of employees, and that is how the template is designed.

Tip: Use left justification to line up character array name values ... for example: %-10.10s or %-10s

Remember: Use the Template!

Optional Challenges

1) Basic Optional Challenges (Bronze Level):

a) Calculate and print the ***minimum*** wage, hours, overtime, and gross values
b) Calculate and print the ***maximum*** wage hours, overtime, and gross values

2) Intermediate Optional Challenge (Silver Level):

Instead of building the linked list in the main function, call another function from main that prompts for the various input (name, clock, wage) and then returns back the pointer.

3) Advanced Optional Challenge (Gold Level):

Read the *Input/Output* chapter in the Kochan book (Ch 15 in the 4th edition, or Ch 16 in the third edition). See if you can read the information from a text file instead of the screen. You can skip reading data from the screen if you decide to do the challenge. You will need to use file pointers here for opening up a file on your computer to read the data and optionally, an output file on your computer to print a report of your employee data. Remember that IDEOne will not support reading files from your computer, but it should work well with any compiler you have installed locally on your computer.

You can also try to incorporate any of the previous challenges from the past homeworks as well, and it would be an alternative if you are using IDEOne.

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 multiple template files provided into your native compiler and Integrated Development Environment (IDE) to get started.

Assignment 8 Code Template

Try this example and use it as a guide to your last homework (number 8). You just really need to expand upon it. Add the missing members to the ***struct employee*** type, and add functions as needed (such as ones to calculate overtime, gross pay, and determine totals and average).

I added comments in **bold** and purple below that start with **TODO** to indicate what needs to change.

Relax, it is OK if you use this code as is and then expand from it ... you have my permission to use all of it. The template can also be found at: http://ideone.com/UJclrz

```
#include <stdio.h>
#include <stdlib.h>   /* for malloc */
#include <ctype.h>    /* for toupper */

struct employee
     {
     int id_number;
     float wage;

     /* TODO - Add other members */

     struct employee *next;
     };

/* TODO - Add Function Prototypes as needed */


/* TODO - Add Functions here as needed */


/* Optional TODO - Add Challenge Functions here as needed */


/*--------------------------------------------------------------------------*/
/*                                                                          */
/* FUNCTION:  print_list                                                    */
/*                                                                          */
/* DESCRIPTION:  This function will print the contents of a linked          */
/*               list.  It will traverse the list from beginning to the     */
/*               end, printing the contents at each node.                   */
/*                                                                          */
/* PARAMETERS:   emp1 - pointer to a linked list                            */
/*                                                                          */
/* OUTPUTS:      None                                                       */
/*                                                                          */
/* CALLS:        None                                                       */
```

```
/*                                                                          */
/*--------------------------------------------------------------------------*/
void print_list(struct employee *emp1)
{
        struct employee *tmp;   /* tmp pointer value to current node */
        int i = 0;              /* counts the nodes printed          */

        /* Start a beginning of list and print out each value               */
        /* loop until tmp points to null (remember null is 0 or false)     */
        for(tmp = emp1; tmp ; tmp = tmp->next)
        {
            i++;

            /* TODO - print other members as well */
            printf("\nEmployee ID: %6d, Wage: %8.2f\n",tmp->id_number,
                                                    tmp->wage);
        }

        printf("\n\nTotal Number of Employees = %d\n", i);

}

/*--------------------------------------------------------------------------*/
/*                                                                          */
/* FUNCTION:  main                                                          */
/*                                                                          */
/* DESCRIPTION:  This function will prompt the user for an employee         */
/*               id and wage until the user indicates they are finished.    */
/*               At that point, a list of id and wages will be              */
/*               generated.                                                 */
/*                                                                          */
/* PARAMETERS:   None                                                       */
/*                                                                          */
/* OUTPUTS:      None                                                       */
/*                                                                          */
/* CALLS:        print_list                                                 */
/*                                                                          */
/*--------------------------------------------------------------------------*/
int main ()
{

    char   answer[80];       /* to see if the user wants to add more employees */
    int    more_data = 1;    /* flag to check if another employee is to be processed */
    char   value;            /* gets the first character of answer */

    struct employee *current_ptr,   /* pointer to current node */
                    *head_ptr;      /* always points to first node */

   /* Set up storage for first node */
```

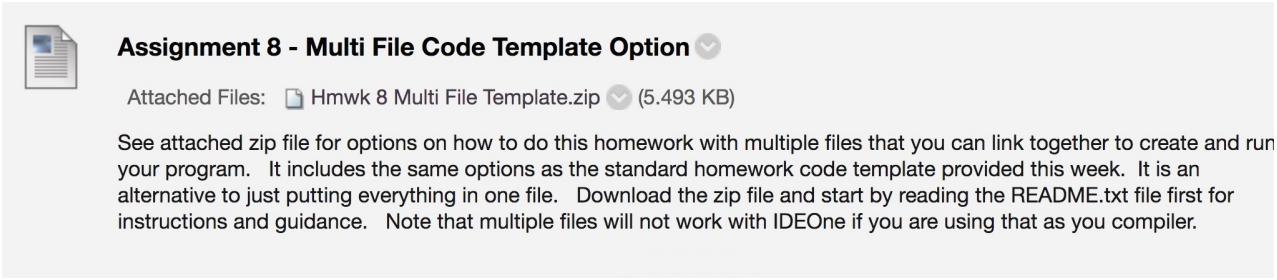
```
head_ptr = (struct employee *) malloc (sizeof(struct employee));
current_ptr = head_ptr;

while (more_data)
{


    /* TODO - Prompt for Employee Name and Hours as well here */

   /* Read in Employee ID and Hourly Wage */
    printf("\nEnter employee ID: ");
    scanf("%i", & current_ptr -> id_number);

    printf("\nEnter employee hourly wage: ");
    scanf("%f", & current_ptr -> wage);


    /* TODO - Call Function(s) to calculate Overtime and Gross */

    printf("Would you like to add another employee? (y/n): ");
    scanf("%s", answer);


    /* Ask user if they want to add another employee */
    if ((value = toupper(answer[0])) != 'Y')
    {
       current_ptr->next = (struct employee *) NULL;
       more_data = 0;
    }
    else
    {
       /* set the next pointer of the current node to point to the new node */
       current_ptr->next = (struct employee *) malloc (sizeof(struct employee));
      /* move the current node pointer to the new node */
       current_ptr = current_ptr->next;
     }
} /* while */


/* TODO: Call Function(s) to determine totals and averages */
/* Optional TODO: Call Challenge Functions to determine min and max values */


/* print out listing of all employee id's and wages that were entered */
print_list(head_ptr);


printf ("\n\nEnd of program\n");
```

```
    return (0);

}
```

Homework 8 - Multi File Template Option

Take the optional challenge this week and implement Homework Assignment 8 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 includes the following files

- **employees.h** - header file with common constants, types, and prototypes
- **main.c** - the main function to start the program
- **makefile** - a file you can use if you want to compile and build from the command line
- **print_list.c** - a function that will print out the current items in a linked list given a pointer to the beginning of it
- **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 8 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.