Welcome to Week 4!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 4 folder**.
- Read **Chapter 6** in the latest edition of the textbook (chapter 7 in earlier editions).
- Begin **Quiz 4**. It is due Sunday at midnight.
- Begin **Assignment 3**. 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 4 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. Sili, How goes the development of your new C program? You know, the one that improves our defensive shields and protects every inch of my precious star ship."

--- Montgomery Scoot
Chief Engineer, U.S.S. Enterprise

Introduction to Arrays

Now that we have learned about implementing the three basic fundamental properties of computer programming in C, let's take a look at a data structure known as an **array**. The notes this week will concentrate on some topics (mainly array addressing) that are very important. It will help us understand how arrays, and later pointers, work in C, as well as their usage in functions (when we start Chapter 8).

The notes are broken up into two sections. There is an initial video that will go over all the basic concepts you'll need to understand for how to set up and use Arrays. It is followed by series of lecture notes that provides lots of detail and examples. They are followed by yet another video that will cover two and three dimensional arrays, followed yet again by a series of detailed lecture notes. View the videos first and then get the detail you'll need in the lecture notes ... and of course, read the chapter in the book on Arrays.

Before we start covering the actual syntax on Arrays, let us go over why they are needed in the first place. We'll cover a good example that clearly illustrates the need for an Array, and it will be followed by yet another, ingenious if I must say so myself, scenario utilizing the Star Trekking Universe. It is all on this page, so read on below ...

Why Arrays are Needed

Suppose you had a problem to solve where you are given a set of five integer numbers and you need to determine which of those numbers are greater than the average. Finding the average would be easy. All you would need to do is add all the numbers together and then divide by five.

When you keep a **running total** in a variable, make sure it is initialized to zero, otherwise, you might be adding the first number to an undefined number. This is resolved below by initializing the sum variable to zero. Consider the code below which is also at http://ideone.com/CFWeDc

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

```
int main ( )
{

    float average;   /*  The average of a set of numbers  */
    int i;           /*  loop index                       */
    int number;      /*  user entered number              */
    float sum;       /*  running total of the numbers     */

    sum = 0;         /* Very important to do this         */

    /* Read in all numbers and keep a running total */
    for (i=0; i < 5; ++i)
    {
         printf ("Enter an integer value: ");
         scanf ("%i", &number);

         sum += number;  /* running total */
    }

    average = sum / 5;    /* get the average */

    /* WHAT DO I DO NOW?  */

    return (0);
}
```

The problem is that the above program reads in a number, and then **reuses** that variable the next time around to read the next number. There is no way after you figured out the average to determine which of those numbers you entered were greater than the average. The **number** variable will only contain the **last** integer value you entered into it. You could set up five variables like below where you could have a place to store all the values you read in:

```
int number1;    /* first number     */
int number2;    /* second number    */
int number3;    /* third number     */
int number4;    /* fourth number    */
int number5;    /* fifth number     */
```

That would work, but ask yourself, what if you had to read in thousands of numbers and process them? It would quite

the coding task since you would have and to create and utilize thousands of variable names. Arrays offer a solution that allow you to **economically** store items of the same type together. These array items can then be accessed anytime during the program to read, modify, or just to print their values. The best thing is you only need to come up with an array name, its type (integer, float, char, ...) and a size, and your code will be much cleaner, readable, faster, and maintainable.

So read on and let us explore how arrays work, and more importantly, how they work in the C programming language. At the end of the lecture notes this week, I'll present an array solution to our program above.

Star Trekking Movie - An Array of Possibilities

Let's have some fun and look in on a conversation about a C program that is being developed on the USS Enterprise. In my movie, Chief Engineer **Montgomery Scoot** helps second officer **Mr. Sili** with his program to improve how the ship's shields protect the Enterprise. In the original Star Trek series, "Shields" provided limited protection against other ship attacks or the hazards of space travel ... also called "deflectors" or "deflector shields", one could also think of them as "force fields". Various geographic areas of the shield around the ship are parceled into individual **sectors**, the total of which protects the entire area of the ship. For you "Trekkies" out there, you might recognize the technology from a scene in *Star Trek II, The Wrath of Khan*.

Watch Video

The Importance of Raising Shields

Duration: 1:06
User: n/a - **Added:** 9/17/09

Believe it or not, I also found a short twenty-two second video that helps to visualize the "Science" behind shields being brought up and down on a Starship. When you watch it, envision each of the sectors lighting up, that in total, constitute an "Array". Since it has no words associated with it, I've also attached an accompanying audio file depicting what is happening. If you don't see the controls, click here for it.

-00:00

Watch Video

Shields Up & Down Video Demo.wmv

Duration: (0:22)
User: tenacontrol - **Added:** 4/8/11

In my movie below, Mr. Scoot's suggestion is to use **Arrays** in Mr. Sili's C Program instead of creating many **individual variables**. For those not familiar with the original Star Trek series, the running gag throughout the show was that everyone thought Mr. Scott loved his ship more than his fellow crew members. You'll find that I've incorporated that gag, along with why using an Array is a great development choice, into the movie below. Click the image below or jump back to the class lecture notes page to watch the movie.


share
xtranormal
00:56
02:10

Answers to Exercises from Last Week

Before we continue on, here are my answers to selected exercises at the end of the book chapter on Conditionals.

Exercise 4

```
//*********************************************************************
// Exercise 4 - A program which acts as a simple "printing calculator"
//
// The program allows the user to type an expression in the form:
//
//     number operator
//
// The following operators are recognized by the program:
//
//     + - * / S E
//
//*********************************************************************


#include <stdio.h>

int main ()
{
    float number;          /* number value to read from terminal        */
    char op;               /* operator value to read from terminal      */
    float accumulator = 0; /* Contents of a calculator's accumulator    */
    int moredata = 1;      /* Flag - operator has not typed 'E          */
                           /* which signifies that execution is to end. */


    printf ("Begin Calculations \n");

    do /* read until the user types the operator 'E' */
    {

    /* read in a number and operator from the screen */
    printf("Enter an expression (Number Operator format): ");
    scanf ("%f %c", &number, &op);

    switch(op)
    {
        case '+': /* Addition */

            printf("= %f\n", accumulator += number);
            break;

        case '-': /* Subtraction */

            printf("= %f\n", accumulator -= number);
            break;

        case '*': /* Multiplication */

            printf("= %f\n", accumulator *= number);
            break;

        case '/': /* Division */

            if(number == 0)
            printf("Error, Cannot divide by Zero\n");
            else
            printf("= %f\n", accumulator /= number);
            break;

        case 's':
        case 'S': /* Store */

            printf("= %f\n", accumulator = number);
            break;
```

```
        case 'e':
        case 'E': /* Exit */

            printf("= %f\n", accumulator);
            printf("End of Calculations \n");
            moredata = 0;
            break;

        default: /* Unknown Operator */

            printf("Error, Unknown Operator: %c\n", op);
            break;

    } /* switch */

    } while (moredata);

    return(0);

} /* main */
```

```
==========================================
```

```
Begin Calculations
Enter an expression (Number Operator format): 10 S
= 10.000000
Enter an expression (Number Operator format): 2 /
= 5.000000
Enter an expression (Number Operator format): 55 -
= -50.000000
Enter an expression (Number Operator format): 100.25 S
= 100.250000
Enter an expression (Number Operator format): 4 *
= 401.000000
Enter an expression (Number Operator format): 0 E
= 401.000000
End of Calculations
```

Exercise 6

```
/*****************************************************************************/
/* Exercise 6                                                                */
/*                                                                           */
/*    Write a program that takes an integer keyed in from the terminal       */
/*    and extracts and displays each digit of the integer in English.  So,   */
/*    if the user types in 932, then the program should display              */
/*                                                                           */
/*    nine three two                                                         */
/*                                                                           */
/*    Remember to display "zero" if the user types in  just a 0.).  Note     */
/*    this problem is a hard one, but not too difficult for your             */
/*    illustrious professor.                                                 */
/*                                                                           */
/*****************************************************************************/


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

int main ()
{

    int digit_count;       /* the number of digits                      */
    int digit_10;          /* determine digit value of 10, ones,        */
                           /*   tens, hundreds, thousands, etc          */
    int i;                 /* loop counter                              */
    int left_digit;        /* the leftmost digit of the current number  */
    int new_num;           /* temporary number value                    */
    int number;            /* current number                            */
```

```
int orig_number;       /* the number the user typed in             */

/* initialize variables */
digit_count = 0;
digit_10 = 1;

/* Prompt for a number */
printf("Enter your number: ");
scanf("%d", &number);

/* save the original number for later */
orig_number = number;

/* Print zero and exit if the number is 0 */
if(orig_number == 0)
{
    printf ("zero\n");
    exit (0);
}
else if(orig_number < 0)  /* exit if negative number as well */
{
    fprintf (stderr, "Error, negative numbers are not allowed \n");
    exit (EXIT_FAILURE);
}

/* determine the number of digits */
while(number != 0)
{
    number /= 10;
    digit_count++;
    digit_10 *= 10;

}  /* digit count while loop */

/* determine each digit starting from the left, one digit at a time */
i = digit_count;
new_num = orig_number;
digit_10 /= 10;

while(i != 0)
{
    left_digit =  new_num / digit_10;
    new_num %= digit_10;
    digit_10 /= 10;
    i--;

    /* print the English value of the current left digit */
    switch(left_digit)
    {
        case 0:
          printf("zero ");
          break;
        case 1:
          printf("one ");
          break;
        case 2:
          printf("two ");
          break;
        case 3:
          printf("three ");
          break;
        case 4:
          printf("four ");
          break;
        case 5:
          printf("five ");
          break;
        case 6:
          printf("six ");
          break;
        case 7:
          printf("seven ");
          break;
        case 8:
          printf("eight ");
```

```
            break;
          case 9:
            printf("nine ");
            break;
          default:
            printf("invalid ");
            break;
      }  /* end switch */

} /* while loop for left digits */

printf("\n");  /* new line before quitting */

return(0);

}
```

Video - Array Basics

This video will introduce you to the basic concepts in setting up and working with the data structure known as an Array. You will learn how to set up and utilize a single dimensional array, as well as see how C works with addresses to calculate and access array elements. View this first before reading through any lecture notes this week.

Variable Sizes

Before we discuss Arrays this week, let's first understand that each type in C implies a consistent size in bytes for any variable declared of that type. You remember from our discussion in Chapter 4 that a byte (http://en.wikipedia.org/wiki/Byte) is 8 bits, and I'll add here that byte boundaries are what memory addresses are based on. So far, we have looked at integer, float, char, and double as our basic types in C. There are also long options, such as *long int,* that will make an integer the biggest it can be on a specific machine, and *short int,* which is the smallest integer size available. On many older computers, most integer sizes were 2 bytes and making it a long integer would increase it to 4 bytes. In the case of a float verses a double, a variable of type double is normally bigger to handle the increased precision of digits past the decimal point.

The C library has a very useful routine called **sizeof** that you can pass a specific type to and it will return how many bytes it is. Its already defined in the stdio.h file that you include in most of your C programs. When in doubt, include the stdio.h file. It has the following syntax:

sizeof (type)

... where type is any valid C variable type, such as int, float, double, char and others that we will cover later on in this class. The important thing is that it tells you the byte size of the type you pass to it, and it's value will be an integer (a whole number), as you won't get fractional values back. We'll cover *sizeof* again later on this class when we cover pointers and dynamic memory allocation.

Since many of you are using IDEOne, I ran the following code there to determine the standard byte sizes of each type on their computer server. Feel free to try this program if you run a compiler on your computer, it may differ from computer to computer as you are starting to see 32 bit and 64 bit type machines rather than 16 bit ones from the past. Some of you may even use UNIX, LINUX, Apple, and other type systems. Try it out at: http://ideone.com/q1gpL.

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

   printf ("Integer is %i, Long Integer is %i,
            Char is %i, Float is %i, and Double is %i \n",
            sizeof(int), sizeof(long int), sizeof(char),
            sizeof(float), sizeof(double) );

   return (0);

}
```

Output:

```
Integer is 4, Long Integer is 4, Char is 1, Float is 4, and Double is 8
```

The important point here is on a specific machine, any variable defined with a specific type will ALWAYS be that byte size. For example, if I declare two integers and run it on IDEOne:

```
int x;
int y;
```

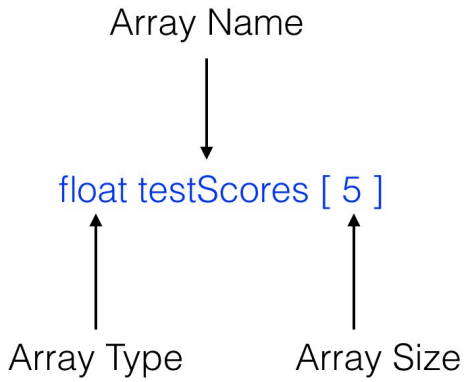
... according to my program above, each integer will be 4 bytes long. There will NOT be one integer that is 2 bytes and another that is 4 bytes, you can count on consistent sizes in bytes of each type in the C Language when it is run on a particular machine, and really for any computer language.

As we start our discussion on Arrays, please keep this in mind, as everything about Arrays is based on this concept. Going forward with this lecture and those that follow each week, let us assume that in my examples, these sizes will apply to the following types:

Type	Size (in Bytes)
int	4
short int	2
long int	4
char	1
float	4
double	8

Declaring an Array

An ***Array*** is an identifier that refers to a collection of data items that all have the same name. The data items must all be the same type (e.g., integers, float, char, double, ...). The individual data items in turn are represented by their corresponding ***array elements***, where the first data item is represented by the first array element, and so on. The individual array elements are distinguished from one another through the use of an ***array index*** (a.k.a., subscript) which must be **an integer** value. Let's look at the basic syntax of declaring an Array.



- The Array Type is any valid C **type**
- The Array Name is any valid C **variable name**
- The Array Size must be an **integer** value
    - It can be part of an **integer expression**:

      ```
      float testScores [23+12];   /* will evaluate to integer value 35 */
      ```

    - ... or a **variable** containing an integer value:

      ```
      float testScores [studentCount];   /* studentCount must an integer value */
      ```

    - ... or a **constant** that is an integer value, such as:

      ```
      #define STUDENT_SIZE 45
      float testScores [STUDENT_SIZE];   /* array size is 45, an int value */
      ```

The **array size** determines how many **array elements** are stored consecutively in memory. The examples below show how each array is set up in memory based on the **array type** used in the variable declaration. On the left, I created addresses which are **offset** by the size of the array type. You do not need to know the exact addresses as this is done by the computer. However, it is important to understand how C calculates addresses to find an array element, and knowing the size of the array type helps determine the offset from one array element to the next. To simplify things in my

example, I started with a nice round number of 1000 for an address, an item that is 4 bytes past it would be 1004 (1000 + 4 bytes). Memory locations are referenced on **byte boundaries**.

Consider the following program:

```
#include <stdio.h>
int main ( )
{
        int clock [5];
        float hours [5];
        double gross [5];
        char letters [6];

        return (0);
}
```

Let's look at the following declarations so that you can better visualize what an array of each type might look like in memory. Note how each array element in a given array variable has the **same type and byte size**, and that the byte size is used to determine the location of the next array element. Also note that each array element is distinguished from another through the use of an **array index** (e.g., clock [**0**] and clock [**3**]). As for the actual values stored in these memory locations, we'll discuss that in the lecture notes that follow, for now, let us concentrate on how arrays are stored in memory.

int clock [5]; */* an integer is 4 bytes on IDEOne */*

Address	Size (bytes)	Type	Array Element
1000	4	Integer	clock [0]
1004	4	Integer	clock [1]
1008	4	Integer	clock [2]
1012	4	Integer	clock [3]
1016	4	Integer	clock [4]

float hours [5]; */* a float is 4 bytes on IDEOne */*

Address	Size (bytes)	Type	Array Element
1020	4	float	hours [0]
1024	4	float	hours [1]

1028	4	float	hours [2]
1032	4	float	hours [3]
1036	4	float	hours [4]

double gross [5]; /* *a double is 8 bytes on IDEOne* */

Address	Size (bytes)	Type	Array Element
1040	8	double	gross [0]
1048	8	double	gross [1]
1056	8	double	gross [2]
1064	8	double	gross [3]
1072	8	double	gross [4]

char letters [6]; /* *char is 1 byte on all machines* */

Address	Size (bytes)	Type	Array Element
1080	1	char	letter [0]
1081	1	char	letter [1]
1082	1	char	letter [2]
1083	1	char	letter [3]
1084	1	char	letter [4]
1085	1	char	letter [5]

PITFALL

C does NOT do **BOUNDS CHECKING** (that is up to you!). If your array contains 5 elements, and you reference an index of 10, it will try to access that address in memory. The compiler is not going to give you an error, but you may receive one when running your program.

C does **not have any limits on the array size**. However, you are constrained by program stack and other memory constraints. Don't make an array a huge size like 1 million, but do think worst case for now. I'll show you have to dynamically allocate additional space as needed later on in the class with our last homework when running your program in real time.

Don't declare an array with zero or a negative size, that will be invalid.

```
float gross [0];  /* arrays SHOULD have a positive size */
float hours [-2]; /* definitely NOT a negative size */
```

It makes no sense to declare an array with only one element:

```
int clock [1]; /* DO NOT make an array with only 1 element */
int clock;     /* Just make it a simple integer variable */
```

Array Elements

Array **elements** can be used like any variable, on the left side or right side of an assignment, here are a few examples.

```
g = values [0];

values [i] = h;

h = values [i+8];

++values [i];       /* better than:  values[i] = values[i] + 1     */

values [i] += 10;   /* better than:  values [i] = values [i] + 10; */
```

Array Index (or subscript)

The array **index,** or subscript, must be an **integer constant**, integer **variable**, or integer valued **expression**.

```
int values [12];     /* integer constant */
int values [x];      /* assuming x declared, a positive integer value */
int values [x + 5];          /* integer expression with a variable */
int values [4 + STD_HOURS]; /* integer expression + simple constants*/
```

These **would** be **invalid:**

```
int values [4.5];    /* its float ... not integer */
int values [5a];     /* invalid variable name */
```

This **may** be invalid

```
int values [abc]; /* if abc is a defined variable of type integer, contains */
                  /* a positive integer value > 1, then valid, otherwise not */
```

A ***constant*** is a great way to specify and utilize an **array size**

```
#define SIZE 5

int gross [SIZE];  /* like saying: int gross [5];   */
```

... then use the **constant** later in the code, like in a loop, after you have assigned values to each element:

```
for (i = 0; i < SIZE; ++i)
{
    printf ("gross [ %i ] = %8.2f \n", i, gross [ i ];
}
```

... The **constant** can also be used as an **index**, but use care:

```
gross [SIZE-1] = 456.78;  /* OK, the index would be 5-1 which is 4     */
                          /* its like saying:  gross [4] = 456.78;     */
```

... but **do not** do this:

```
gross [SIZE] = 345.67;  /* Invalid, since array indexes for this array */
                        /* only go from 0 to 4, 5 is out of bounds.    */
```

Do not declare an array to have just **one element**

```
float gross [1] = 345.67;  /* one element? no array needed here */

/* ... just make it a simple float variable */
float gross = 345.67;
```

... finally, **do not** use a **negative number** for the array size when declaring it, or when initializing an element

```
gross [-2] = 453.95; /* It will set the location two locations before the */
                     /* starting address of the array to 453.95.   You    */
                     /* better hope that it an integer location and the   */
                     /* it does not clobber some other variable.          */
```

Initializing Array Elements

In C, if you just create an array, but don't initialize it, all values are **undefined**. Just like if you create a variable of type integer, float, double, and char.

```
float hours;
```

Address	Variable Name	Contents
1000	hours	Undefined

```
float gross [3];
```

Address	Array Element	Contents
1004	gross [0]	Undefined
1008	gross [1]	Undefined
1012	gross [2]	Undefined

The **exception** to this rule is if your variables are defined as *static* or *global*. In those cases, they are initialized to **zero**. We'll talk more about static and global variables when we get to our discussion on functions in Chapter 8. For now, don't worry about it. When you declare arrays or any other variable, you have the option of initializing the initial contents. With arrays, the initial values are enclosed within **curly braces.** If your array has 5 elements and you initialize less than 5 elements, the leftovers will be set to zero because once you initialize some elements, the other elements are just set to zero in this case.

```
/* initialize the first three elements to 2, 4, and 6 and
   the other two remaining elements to 0 */

    int values[5] = { 2,4,6 };
```

Address	Array Element	Contents
1000	values [0]	2
1004	values [1]	4
1008	values [2]	6
1012	values [3]	**0**
1016	values [4]	**0**

To confirm, feel free to run something like this on your compiler. I ran my version on IDEOne at:
http://ideone.com/tSQ04

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

      int i;                        /* loop index                    */
      int values [5] = {2,4,6};     /* partially initialized array */

     /* loop to access and print each array element */
      for (i=0; i < 5; ++i)
      {
          printf("values [%i] = %i\n", i, values [i]);

      }

      return (0);
}
```

Output:

```
values [0] = 2
values [1] = 4
values [2] = 6
values [3] = 0
values [4] = 0
```

Other examples include:

```
float data[3] = {0.23456, 987.456, 99.0};
```

Address	Array Element	Contents
1020	data [0]	**0.23456**
1024	data [1]	**987.456**
1028	data [2]	**99.0**

```
char  letter [4] = {'a', 'B', 'r', 'u'};
```

Address	Array Element	Contents
1032	letter [0]	**'a'**
1033	letter [1]	**'B'**
1034	letter [2]	**'r'**
1035	letter [3]	**'u'**

```
int   clock [2]  = {};  /* initialize all elements to zero */
```

Address	Array Element	Contents
1036	clock [0]	**0**
1040	clock [1]	**0**

The type (e.g., **int** or **char)** is used so that the compiler knows how much storage to allocate. For instance, an element of an **int** type array is typically given **2 or 4 bytes of storage** depending on the machine type (remember what were learned in class 1 and 2). An element of a **char** type array is given a **byte** of storage. We'll be experts on char types when we cover strings.

One question you might be having is what a **character** value **defaults** to if you don't provide a value for it in the initialization list for an Array. The answer is that it would be the **null terminator character** which has the value of **zero**. It is referenced as **'\0'** in it's **character format**. We will learn all about characters and strings in a few weeks when we cover **Chapter 10** in the book and my notes.

Initialization can imply Array Size

One can leave out the size of the array, only if the compiler can determine its size, such as the number of elements being initialized. An example is shown below:

```
int values[ ] = { 12,11,223,34,10 };
```

implies an **array size** of **5 elements**, since there are 5 items being initialized above

Address	Array Element	Contents
1000	values [0]	**12**
1004	values [1]	**11**
1008	values [2]	**223**
1012	values [3]	**34**
1016	values [4]	**10**

The **initialization** can determine the **number of elements** in the array if no size is provided.

However, you can't just do this:

```
int values [ ];  /* compiler can't figure out the size */
```

... the compiler would not know how many elements are in this array. With C, the **compiler** needs to know **how many elements** are in the array at **compile time** (i.e., when the code is being compiled by a compiler).

Calculated Array Offsets

The compiler will interpret the **array name** as the starting address of the array. This is **equivalent** to **&a [0]**, where **&** is read **"address of"**. As long as the first element's address is known, any given array element's address can be **calculated** and given an offset.

The **address of an array element** can be calculated as follows given that a is the **array name** and i is the **index**:

*&a [i] = a + (i * (size of array element in bytes))*

Remember that a is the same as &a[0] here, so you could also write it as:

&a [i] = &a [0] + (i * (size of array element in bytes))

Given the array **values** below, the **address** of **values [3]** is calculated as follows given that on our machine, an integer is 4 bytes long.

```
&values[3] = values + (3 * (size of an int))
           = &values[0] + (3 * 4);
           = 1000 + (3 * 4)
           = 1012
```

Address	Array Element	Contents
1000	values [0]	10
1004	values [1]	567
1008	values [2]	56
1012	**values [3]**	**23**
1016	values [4]	78

Important:

This lesson is perhaps the most important thing to understand. When we cover topics like **pointers** later on, it will make things MUCH easier to understand. If you want to print an address of your array, it can be done as follows using **%li** for a long int, as addresses could be large numbers. In reality, you might want to print actual **addresses in hexadecimal** with the %x or long %lx format. They are easier to work with.

```
printf("%lx \n", values);     /* 1000 in our example, same as &value[0] */
printf("%lx \n", &values[2]); /* 1008 in our example                    */
printf("%li \n", values[2]);  /* prints contents, not address, => 56    */
```

If you want to display the values and addresses of each array element on your computer, feel free to try this program. Here is what was presented to me from IDEOne ... I printed my addresses in **hexadecimal** (%lx format), as it looks like they are stored at 32 bit (4 byte) addresses on that system. Hexadecimal is a much better choice for displaying addresses.

You can watch it run at: http://ideone.com/sKc3A. Note how in the **Output** that the address of each array element is **four bytes** from the next element address. Try it out on your computer and see what you get if you use something other than IDEOne.

```
#include <stdio.h>

int main ()
{
      int i;                                    /* loop index       */
      int values [5] = {10, 567, 56, 23, 78}; /* initialize array */

     /* loop and print the contents and address of each array element */

      printf("Address\t\tArray Element\tContents\n\n");
      for (i=0; i < 5; ++i)
      {
             /* print addresses in hex */
             printf("%lx \tValues [%i] \t%i \n", &values[i], i, values[i] );
```

```
        }

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

OUTPUT:

```
Address     Array Element   Contents

bffc8d9c    Values [0]      10
bffc8da0    Values [1]      567
bffc8da4    Values [2]      56
bffc8da8    Values [3]      23
bffc8dac    Values [4]      78
```

One final Point:

When programming, it is not that important that you know the actual addresses of the variables, it more important to know their contents and how to access them. However, having this knowledge will help you to better understand how C stores and works with variables, and will help you understand the more complex topics such as pointers, structures, and unions later on. I've found that if I can read things back to myself in English, and also have the ability to draw out how things are stored in memory ... then I can really begin to understand how things are working.

Dimensional Concepts

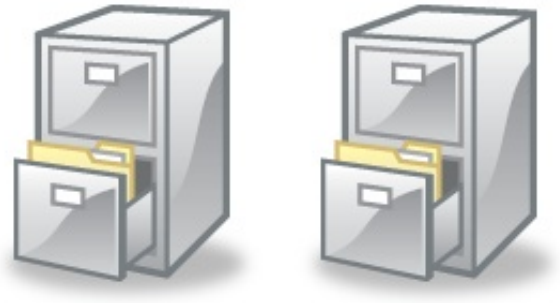
Let's think about dimensions for a second and try to visualize a real world example. Let's say I have a **sheet of paper** which is **2 dimensional,** it has **rows** and **columns**.

Now let's expand that to a 3D concept, which would be a group of papers (the **third dimension** is the **thickness** that the papers provide).

We need a place to store a group of papers together, how about adding a **4th dimension** by putting **papers** into **folders**. We could store them in a single draw of a file cabinet.

The **fifth dimension** could be represented as a file cabinet that contains multiple **draws** of **folders**.

The **sixth dimension** may be a **room** with **file cabinets** in it.



The **seventh dimension** may be **rooms** on a **single floor,** each with the same amount of **file cabinets**.

The **eighth dimension** may be **floors** containing **rooms** full of **file cabinets** containing **draws** containing **folders** containing **papers**.

... I'll leave it as a **mental exercise** for the next levels of dimensions.

Video - Multidimensional Arrays

This video will introduce with how to set up and use multi-dimensional arrays, in particular, those with two and three dimensions.

Initializing Multidimensional Arrays

Multidimensional arrays can be visualized as **rows** and **columns**. There are no actual row and column **attributes** for two dimensional arrays in C, but you can work like they exist within the language, and its much easier for you to think of them that way. Here are some examples of defining and **initializing two dimensional** Arrays, as well as how you can visualize and access each **element**.

```
static int matrix [2][4] = {
                                {10, 5, -3, 17},  /* row 0 */
                                {9,  0,  1, 14}   /* row 1 */
                            };
```

rows \ columns	0	1	2	3
0	**10**	**5**	**-3**	**17**
1	**9**	**0**	**1**	**14**

```
static int matrix [2][4] = {5, 10, 15, 20, 25};  /* this will row fill as needed */
```

rows \ columns	0	1	2	3
0	**5**	**10**	**15**	**20**

```
rows
      1     25      0       0       0
          --------------------------
```

- Use a separate set of braces for **each row**
- Use a **comma** after each set of braces except the last
- Use **outer braces ending with semicolon** around the sets of row braces
- **static** not needed for ANSI C to initialize array items
- If you simply put in values **without specifying rows**, then the array will be initialized one row at a time and will zero filled if necessary. This is shown in the second example.
- Any **values not initialized will default to 0**. If nothing is initialized, then the values are **undefined**, e.g.,

```
int matrix[2][4];
```

Here is how you can specify and initialize values within specific rows:

```
static int matrix [ 2 ] [ 4] = {
                                  {10, 5},             /* Row 0 */
                                  {9, 0, 1, 14}        /* Row 1 */
                                };
```

```
                              columns

                     0         1         2         3
                  -------------------------------------
              0     10         5         0         0
rows
              1      9         0         1        14
                  -------------------------------------
```

Using const with Arrays

With any variable you set up, you have the option of not allowing any changes once it has been declared and initialized by using the "**const"** keyword in the declaration. For example, in a calendar year, each month has a total number of days. Some months have 30, others have 31. One month, February, is normally 28 days, but has 29 days every four years during a leap year. One could construct a "look-up" table using a two-dimensional array of months and years. The two-dimensional **const variable** below shows the summer months of June, July, and August. I'll leave it to you as an exercise to expand this to all months in a year.

```
/* Constant look up for days in each summer month */
/* ... does not need to change ... ever           */

const int summer_months[3][2] = {  {6,30},   /* June   */
                                   {7,31},   /* July   */
                                   {8,30}    /* August */
                                };
```

The **const** feature when added to a variable will take up space in memory, but the advantage is you can use that variable like any other variable, such as printing specific values, accessing individual areas, passing it to functions, etc. Again, the key point here is once you declare and initialize a variable using **const**, it cannot be changed later on in your program. Trying to update it anywhere in your code will generate an error that will be caught by your compiler.

Tip:

To initialize **each element** in the **row** to a non-default value, you must initialize each row **separately**.

Three Dimensional Arrays

The compiler looks at a three-dimensional array as a one-dimensional array whose elements are two-dimensional arrays.

For example

```
int D3 [ 2 ] [ 3 ] [ 5 ] ;
```

Allocates storage for 30 integers, arranged as two grids of 3 rows by 5 columns each. Its like stacking one two-dimensional array on top of another.

	0	1	2	3	4
0					
1					
2					

D3 [0]

	0	1	2	3	4
0					
1					
2					

D3 [1]

Three-dimensional array references have the following format

type **array_name** [**grid size**] [**row size**] [**column size**]

The order in memory would be the following

```
first Grid 0            ... then Grid 1

  [0][0][0]                 [1][0][0]
        [1]                       [1]
        [2]                       [2]
        [3]                       [3]
        [4]                       [4]
  [0][1][0]                 [1][1][0]
        [1]                       [1]
        [2]                       [2]
        [3]                       [3]
        [4]                       [4]
  [0][2][0]                 [1][2][0]
        [1]                       [1]
        [2]                       [2]
        [3]                       [3]
        [4]                       [4]
```

A sample program that declares, initializes, and prints a three dimensional array is shown below. Note the use of the three "for loops" below (a loop within a loop within a loop). You can try it out with IDEOne at: http://ideone.com/V5l6eF

```
#include <stdio.h>

int main ()
{

   int grid, row, column;   /* you can use any variable name here */

   int D3 [2] [3] [5] = {   /* remember, 3D is an invalid variable name */

      { { 1, 2, 3, 4, 5 },
        { 6, 7, 8, 9, 10 },          /* grid 0 */
        { 11, 12, 13, 14, 15 }
      },

      { { 16, 17, 18, 19, 20 },
        { 21, 22, 23, 24, 25 },     /* grid 1 */
        { 26, 27, 28, 29, 30 }
      }
   };

   for (grid = 0; grid < 2; ++grid)
   {
      printf ("\n");    /* new grid */

      for (row = 0; row < 3; ++row)
      {
         printf ("\n");     /* new row */

         for (column = 0; column < 5; ++ column)
            printf ("%2i ", D3 [grid] [row] [column]);

         printf ("\n");    /* new column */
      }
   }

   return (0);
}
```

The output of this program would look like this:

stdout

```
 1  2  3  4  5

 6  7  8  9 10

11 12 13 14 15


16 17 18 19 20

21 22 23 24 25

26 27 28 29 30
```

You can visualize the three dimensional array after it has been initialized as:

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15

D3 [0]

	0	1	2	3	4
0	16	17	18	19	20
1	21	22	23	24	25
2	26	27	28	29	30

D3 [1]

NOTE: You can have arrays of **any dimensions**, just simply apply the principles learned for three dimensional arrays to 4, 5, 6 and beyond.

Initial Problem Solved with Arrays

Recall in our first lecture note on Arrays, we posed the following problem:

> ***Read in a set of 5 integer values and print all those values greater than the average.***

Feel free to figure this out yourself, but here is my solution using arrays.

```
/* https://ideone.com/DfGxml */

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

     float average;       /*  The average of a set of numbers  */
     int i;               /*  loop index                       */
     int number[5];       /*  store  user entered numbers      */
     float sum;           /*  running total of the numbers     */

     sum = 0;             /* initialize the sum */

     /* Read in all numbers and keep a running total */
     for (i=0; i < 5; ++i)
     {
         printf ("\n Enter an integer value: ");
         scanf ("%i", &number [i] );

         sum += number [i];  /* running total */

     } /* for */

     average = sum / 5;    /* get the average */

     /* print all numbers greater than the average */
```

```
    printf ("\n\n All Numbers greater than the average %.2f: \n\n", average);

    for (i= 0; i < 5; ++i)
    {
        if (number [i]  >  average)
        {
              printf("%i \n", number [i]);
        }
    }  /* for */

    return (0);

} /* main */
```

Alternative Solution using Constants

Now to make the code just a bit better, add a **constant** called SIZE ... if the amount of numbers that needs to be read changes, just modify one line of code:

```
#define SIZE 50
```

... and then recompile. The rest of the code need not change. Also, in the future, we'll learn that the input does not necessarily have to come from entering values at the screen, rather, it could also come from reading values out of a file or database. In that case, you have to think what are the most values that could be read in and adjust your array size to the worst case. In our final homework assignment, I'll show you how acquire more space in your array in a dynamic fashion so you don't have to worry about the size, storage will increase as needed during run time. I ran it with 50 data sets at: http://www.ideone.com/KLmoR2

```
#include <stdio.h>

 #define SIZE 50

 int main ( )
 {

     float average;          /*  The average of a set of numbers  */
     int i;                  /*  loop index                       */
```

```
    int number [SIZE];       /*  store user entered numbers       */
    float sum;               /*  running total of the numbers     */

    sum = 0;                 /* initialize the sum */

    /* Read in all numbers and keep a running total */
    for (i=0; i < SIZE; ++i)
    {
         printf ("\nEnter an integer value: ");
         scanf ("%i", &number [i] );

         sum += number [i];  /* running total */

    } /* for */

    average = sum / SIZE ;  /* get the average */

    /* print all numbers greater than the average */

    printf ("\n\nAll Numbers greater than the average %.2f: \n\n", average);

    for (i= 0; i < SIZE; ++i)
    {
        if (number [i]  >  average)
        {
              printf("%i \n", number [i]);
        }

    } /* for */

    return (0);

} /* main */
```

Video - Common Array Mistakes

Think you know Arrays now? Think again! Check out this video which will show you the common mistakes novice AND professional programmers make when using arrays in C. Tips will also be provided to help you eliminate these types of errors from your C programs.

Summary

An ***Array*** is an identifier that refers to a collection of data items that all have the same name. The data items must all be the same type (e.g., integers, float, char, double, ...). The individual data items in turn are represented by their corresponding ***array elements***, where the first data item is identified by the first array element, and so on. The individual array elements are distinguished from one another through the use of an ***array index*** (a.k.a., subscript) which must be **an integer** value.

The homework this week will build upon your previous homework. I would recommend that you at least declare arrays to hold the clock and wage values. Feel free to initialize the clock and wage values if you wish to simplify things. You will need to prompt for the hours worked for each employee, and then calculate their overtime and gross pay.

I would suggest making arrays to hold the values for hours, overtime and gross pay as well, but this is your call. There are many ways to do this assignment.

Next Week

Read the next chapter in the book on Functions (it's a long one and very important!) and start your next homework. What you want to do is make each variable an array (hours, wage, clock ...).

You have both a **homework assignment** and **Quiz** to work on this week.

If you have time, try some of the chapter exercises from this week and maybe I'll post a few answers next week.

HOMEWORK 3
ARRAYS

Write a C program that will calculate the gross pay and overtime hours for a set of employees using **arrays**.

- Do not use concepts we have not yet covered in the lecture notes or book
- Do continue to build upon concepts from prior homeworks, such as using **constants** and following all **class coding standards**

The program should **prompt** the user to enter the number of **hours** for each employee worked. When prompted, key in the hours shown below from our test data. The *overtime* and **gross pay** values should be **calculated** based on the hours worked for each employee. Don't initialize your hours, overtime, and gross pay to the test data.

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

Unlike your previous assignments, DO NOT prompt for how many employees to process up front. Instead, define a **constant** and use it to specify your **array size** and later on within your code (such as loop tests), for example:

```
#define SIZE 5
```

Make sure you continue to use other constants as well.

You can also **initialize** the **clock** and **wage** arrays to the 5 test values so you don't
have to enter them all in by hand each time you run your program

```
/* unique employee identifier */
long int clockNumber [SIZE] = {98401, 526488,
                               765349, 34645, 127615};
/* hourly wage for each employee */
float hourlyWage [SIZE] = {10.6, 9.75, 10.5,
                           12.25, 8.35};
```

A few other things to consider:

- With arrays, you now have the ability to show all the data together in a single report (using two or more loops) as shown in the test data example above.

- You should implement this program using one array for clock number, one array for wages, and one for hours. You can optionally create arrays for overtime and gross values as well (I would recommend that you do). It will make your transition to next week's homework on functions much easier.

- Column alignment, leading zeros in Clock#, and zero suppression in float fields are important when printing your output. For example, you don't want wage showing up on your output as 42.340000 ... instead, it should display 43.34 as it represents an hourly pay (i.e., money).

- Use time and a half (1.5) as the overtime pay factor (which should be defined as a constant).

- A code template has been provided following this lecture note to help you get started in the right direction. The template is only a suggestion, feel free to craft your own unique solution if desired.

Optional Challenges

For those of you more experienced programmers, here are a few optional challenges you are welcome to try:

- 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

Assignment 3 Code Template

Your next homework will add the concept of **Arrays**. Use a **constant** to specify the array size at compile time. Yes, you can theoretically can prompt for and use a variable at run time to specify the size of the array, but it does not work on every compiler, so don't do it, instead, use a **constant** as shown in the template below. I'll show you how to create dynamic arrays (i.e., linked lists) in the last homework assignment for this class.

Shown below is a template to help you with your homework. With arrays, NOW you can print that nice output that you could only do previously with file pointers. If you do use file pointers, you could probably do everything in one loop, or you could use the template below to print to a file in the second loop. My template is just one way to do it, feel free to use your own style.

```
/* Add File Header from our class Coding Standards here */

#include <stdio.h>

/* constants to use */
#define SIZE 5 /* number of employees to process */

/* add other constants here ... */

int main()
{
    /* Declare variables needed for program.      */
    /* Recommend an array for clock, wage, hours, */
    /* ... and overtime and gross.                */
    /* Its OK to pre-fill clock and wage values   */
    /* ... or you can prompt for them.            */

    /* unique employee identifier */
    long int clock_number [SIZE] = {98401, 526488,
                                    765349, 34645, 127615};
    int count; /* loop index */

    /* hourly page for each employee */
    float hourly_pay [SIZE] = {10.6, 9.75, 10.5,
                               12.25, 8.35};

    /* Process each employee one at a time */
    for (count = 0; count < SIZE; count++)
    {

        /* Prompt and Read in hours worked for employee */

        /* Calculate overtime and gross pay for employee */
```

```
    }

    /* Print a nice table header */

    /* Access each employee and print to screen or file */
    for(count = 0; count < SIZE; count++)
    {
        /* Print employee information from your arrays */
    }

    return(0);
}
```