C PROGRAMMING COURSE E-BOOK

from www.cstutoring.com

C PROGRAMMERS GUIDE LESSON 1

File:	CGuideL1.doc
Date Started:	July 24,1998
Last Update:	Feb 28, 2002
Status:	proof

INTRODUCTION

This manual will introduce C Programming Language techniques and concepts. The C Programming Language was the first common sense practical computer programming language. It is used widely today in industry and system programming. This manual teaches by using the **analogy** approach. This approach allows you to understand concepts by comparing the operation to common known principles. This makes it easier for you to understand and grasp new concepts. We also use the **seeing** approach. The words we use are carefully chosen so that you get a picture of what is happening. **Visualization** is a very powerful concept to understanding programming. Once you get the picture of what is happening then programming is much easier to understand. It is very important for you to visualize and see things as a picture rather than trying to understand. by reading and memorizing textbooks. Pictures are made up of words. This document does not cover all details of C programming but acts as a guide so that you can get started, right away with C programming. The main purpose of this document is to introduce and teach you C programming techniques. The reader is encouraged to have a textbook as a companion to look up important terms for clarification or for more in depth study. Good textbooks are:

Title	Author(s)	Publisher
The C Programming Language	Brian W. Kernighan , Denis M Ritchie	Prentice Hall
Programming in ANSI C	Run Kumar, Rakesh Agrawal	West Publishing

PURPOSE OF PROGRAMMING LANGUAGES

The purpose of a programming language is to instruct a computer what you want it to do. In most cases of beginner programs, the computer tells the programmer what to do. Good programmers are in control of the computer operation. A typical computer program lets the user enter data from the keyboard, performs some calculation and then displays the results on a computer screen. Users can store input and output data on a file for future use. Important C keywords are introduced in **bold**. C language definitions are introduced in ***violet italics***, programming statements are in **blue** and programming comments in **green**.

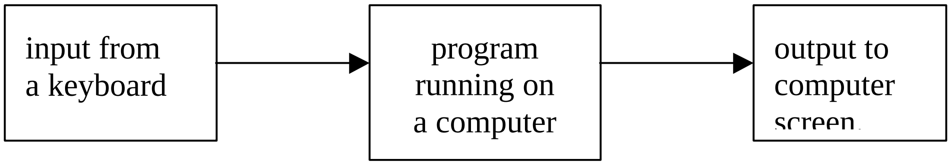
LESSONS AND ASSIGNMENTS

You should understand all the material and do all exercises in each lesson before attempting the next lesson. Course grades are (P) Pass, (G) Good and (E) Excellent. Excellent is awarded to students with outstanding programs that involve creativity and works of genius. Good is awarded to students that have exceptional working programs. Pass is awarded to students that have minimal working programs.

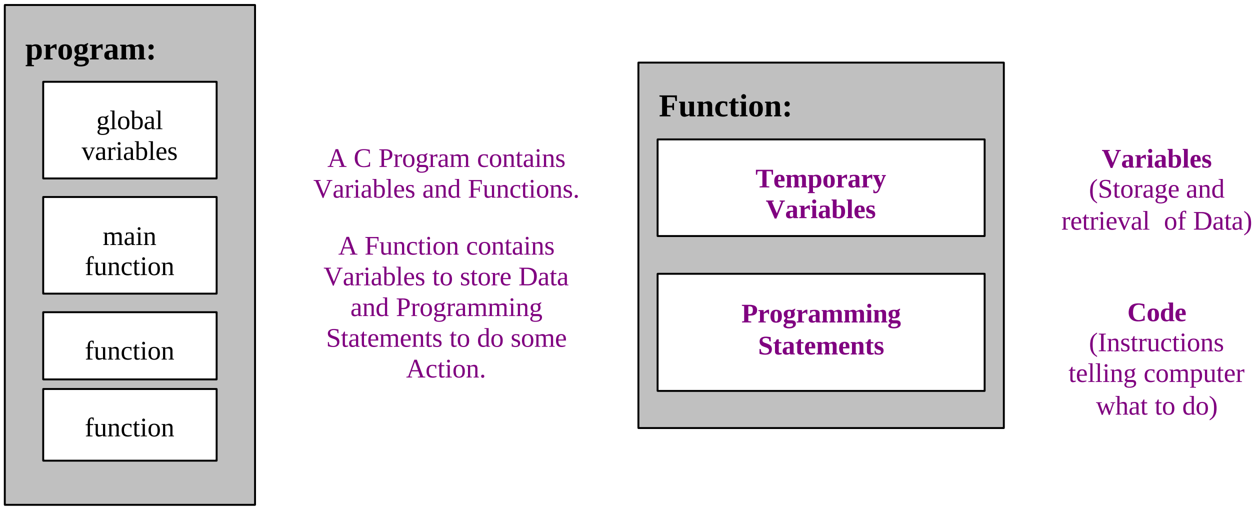
LESSON 1 C PROGRAMMING COMPONENTS AND VARIABLES

C Programs

A program running on a computer is used to calculate some results from input data from a computer keyboard and display the results on a computer screen.



A program must be written in a computer language before it can run on the computer. A C program is written as words representing **variables** and **programming** statements grouped together in a **function**. Variables are used to store and retrieve **data** values and programming statements are **instructions** telling the computer what to do. Programming statements are also known as **source code** or **just code** and allows the program to do something like add two numbers or print a message to the computer screen. Variables and programming statements are grouped together into a **function**. The purpose of using a function is to avoid repeating the same programming statements over and over again to do the same thing. A program will call a function to perform a particular task more than once. A program will have many functions. The first function to execute in a program will be the **main** function. Every C Program must have a main function. The variables declared in a program are called global variables because they are shared between all functions. That are also called permanent variables because they retain their value for the life of the program. The variables declared in a Function are temporary. they only retain their value when the function is being used.



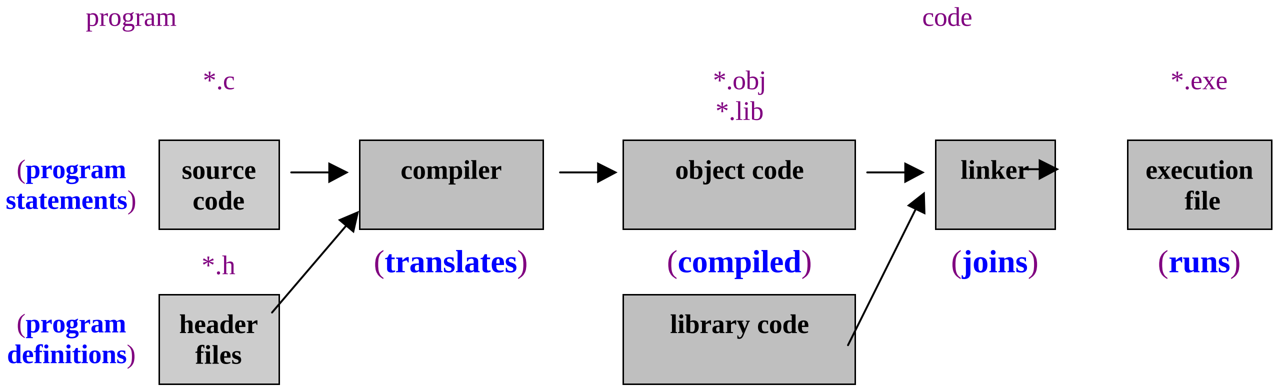
A C Program contains Variables and Functions.

A Function contains Variables to store Data and Programming Statements to do some Action.

You will learn all about programming statements, functions and variables in this course. The idea is to get familiar with the terms, do the questions and exercises and then understand. Once you do something, understanding is much easier. People who do poor at programming try to understand first. They don't understand, so they do not do. By not doing they will never understand !

Compiling and Running C Programs

A compiler is needed to **translate** a program in source code into an intermediate file called an **object file**. The compiler has pre-compiled object files known as **library files**. Library file's contain object code that your program may need to perform tasks like reading a key from the keyboard or writing a message to the computer screen. The compiler may need additional information to compile your program. The additional information is stored in files called a **header file**. The header file contains information about functions contained in other source files or compiled library files. The most famous header file you see at the top of every C program is #include <stdio.h>. The compiler needs to know how to compile your source code when you make references to other functions not contained in your program. Functions that read input from a keyboard or send messages to a computer screen. The object and library files are linked together into an **execution file**. Before you can run the execution file, the contents are loaded into the computer memory. A computer executes machine code. Machine code is instructions at the computer level that tells the computer what to do. When your program runs, the computer is executing the machine code loaded from the execution file.



C PROGRAM FORMAT AND COMPONENTS

A C program is made up programming components usually arranged in a predefined format. By following a proven format, your program will be more organized and easier to read. A C program usually starts with **comments**, **include statements**, **define statements**, **typedef's, enumeration's**, **structure definitions, function prototypes**, **global variables,** the **main function** and **function definitions.** A **compiler** reads a C program and converts it into an **execution code** so that the computer can run it**.** A C program is made up functions. Functions are made up of variables declarations and statements. The first function to execute in a C program is the main function. The other program functions may be implemented before the main function or after. A function is declared before it is defined. A function declaration is known as a **prototype**. When you use function **prototypes,** the functions are usually defined after the main function. It does not matter where you define your functions before or after the main function. It can be your own preference.

C Program format:

comments
#include statements
#define statements
typedef's
enumeration's
structure definitions
function prototypes
global variables
main function
function definitions

Comments

All programs need **comments** to explain how the program works, what the statements and functions do. Comments may be placed anywhere in your program. A comment starts with a /* and ends with a */. The compiler ignores comments.

```
/* this is a comment */
```

Include Statements

Include statements are used to add external files called **header files** into your program. A header file gives the compiler additional information how to compile the program. A header file contains function prototypes of system library functions. An example of a system library header file is the input/output libraries. These libraries allow you to get data from a keyboard or from a file or send information to the computer screen or store data on an output file. The header file name is enclosed by < > triangle brackets to indicate that they are to be located in the **compiler directory**.

```
#include  <header_file_name>
```

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

Your program may need additional header files that you wrote. Your header files will contain function declarations of the functions you will be using in your program. In this case, the header file name is enclosed by quotes to indicate that they are to be located in your **working directory**. This is the directory where your program files are.

```
#include  "user_file_name"
```

```
#include "myheader.h"
```

Many programmers like to put all their program declarations in a header file and their program code statements in an implementation file. Header files have the extension of **".h"** which refers to "header" where the C program code statement files has the extension **".c".**

Constants

Numbers like (0-9) and letters like ('a'- 'Z') are known as **constants**. Constants are **hard coded values** that are assigned to variables. There are many different types of constants.

character: 'A' **numeric:** 9 **decimal:** 10.5 **string:** "hello"

Letters constants have the **single quotes 'a'** around them. Letters are represented by **ASCII** numeric values. This means each letter has an equivalent decimal value representation. For example, the letter 'A' has the decimal value 65. Numbers may be represent by a numeric value 9 or as a letter like '9'. Do you know the difference between '9' and 9 ? Numbers are grouped together to represent larger numbers 1234. You may also group letters together into **character strings** surrounded by **double quotes** like "hello". What is the difference between "1234" and 1234 ? Character strings are made up of individual characters.

representing numbers in a computer

All memory is made up of bits having values 0 and 1. 8 Bits are grouped together into a byte. Each bit has a weight to 2 to the power of N.

1 byte is 8 bits

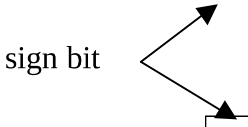
0	1	1	0	1	0	0	1

bits are 1 and 0's

Numbers may use from 1 to many bytes. Groups of memory bytes are used to represent numbers. The smallest to largest number a data type can represent is known as the **range**. Small range numbers use few bytes of memory where large range numbers use many bytes of memory. Numbers may be **unsigned** or **signed**. **Unsigned** meaning positive numbers only where **signed** means negative or positive numbers. The first bit **M**ost **S**ignificant **B**it (**MSB**) in a **signed** number is called the sign bit.

Signed **Positive** numbers start with a 0.

0	1	1	0	1	0	0	1



Signed **Negative** numbers start with a 1

1	1	1	0	1	0	0	1

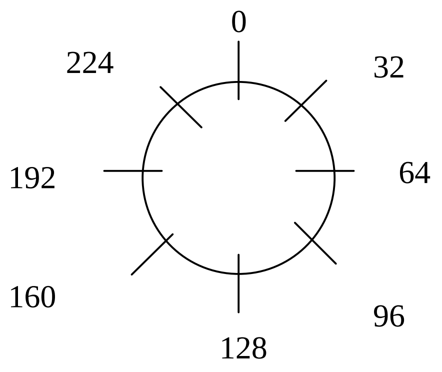
For unsigned numbers, it does not matter since the number will always be positive grater or equal to zero. It is important now to understand how signed and unsigned numbers are represented in computer memory. The same binary numbers are used to represent negative or positive numbers. It is just how they are forced to be represented in your program. This means the same binary number can be a positive number like 192 if **unsigned** or a negative number like -64 if **signed**. The number representation are known as number wheels.

RANGE OF NUMBERS

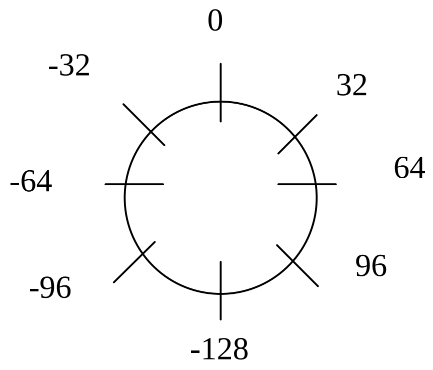
The **range** of a number tells us the smallest number and the largest number a memory can represent. Two types of numbers:

(1)	**signed**	positive and negative numbers
(2)	**unsigned**	positive numbers only

We now show you how to calculate the range of signed and unsigned numbers. The following calculates the range for an un**signed** char data type of 1 byte. 1 byte is 8 bits, therefore N =8. A signed number has positive numbers only.

The range of a **unsigned number** is: 0 to $(2^{N}$ bits$)-1$	
To calculate the range of a unsigned number for N = 8 :	
0 to $(2^{N}$ bits$)-1$ 0 to $(2^{8})-1$ 0 to 256-1 0 to 255	The range of an 8 bit unsigned number is 0 to 255.

The following calculates the range for an **signed** char data type of 1 byte. 1 byte is 8 bits, therefore N =8. A signed number has negative and positive numbers.

The range for signed numbers is calculated as:	
$-2^{(N-1)}$ to $(2^{(N-1)})-1$.	
To calculate the range of a signed number for N = 8 :	
$-2^{(N-1)}$ to $(2^{(N-1)})-1$. -2^{7} to (2^{7}) - 1 -128 to 127	The range of an 8 bit signed number is: -128 o 127.

The negative number is 1 extra because you are splitting a even number into two parts

Data Types

Data types are used to specify how data is to be represented in a computer memory. Data types tell the compiler what kind of data, the memory is to represent. Common C data types are **char, short, int, long, float** and **double.** Why do we need different data types ? We need different data types to specify what kind of data a variable is to represent. By using different data types, we can minimize the amount of data memory space we use. The following table lists the common C data types. The **int** data type is platform dependent, this means the size is dependent on the computer and operating system used. **int** may be 16 or 32 bits.

data type	bytes	bits N	example(s)	unsigned range (pos) 0 to (2**N)-1	signed range (neg/pos) -2**(N-1) to 2**(N-1)-1
char	1	8	'A', 23	0 to 255	-128 to 127
short	2	16	1234	0 to 65,535	-32,768 to 32,767
int	2 4	16 32	12467 12345565	0 to 65,535 0 to 4294967296	-32.768 to 32,767 -2147483648 to 2147483647
long	4	32	123456788	0 to 4294967296	-2147483648 to 2147483647
float	4	32	34.56 454e-23	1.2**e-**38 to 3.4**e**38	
double	8	64	3456.76545 2.3456**e**156	2.2**e**308 to 1.8**e**308	

unsigned and signed numbers

Unsigned numbers are only positive, where signed numbers may be negative or positive. Data types are usually defaulted to signed. If you are unsure then you can force data types to be signed or unsigned. Data types may be forced to be signed with the **signed** keyword or unsigned with the **unsigned** keyword.

unsigned int	force int to be unsigned	(positive only)	unsigned int x:
signed int	force int to be signed	(positive/negative)	signed int x:

float and double number representation

Float and **double** data types are used to represent large numbers known as floating point numbers stored in a small memory space. Floating point numbers have a **fraction** (**mantissa**) part and an **exponent part** and represented in decimal point notation 24.56 or exponent notation 3.456E04. It is the stored exponent that allows the floating point number to represent a large value in a small memory space. The following is a 32 bit floating point format known as a **float**, where the sign, exponent and fractional parts are stored separately.

1	8	23	32 bit floating point number (**float**)
sign bit	exponent	fraction (mantissa)	$(-1)^{sign} * 2 \text{ exp }^{(exponent-127)} * (mantissa)$
-	12	.12365	$1.2365 * 10^{12}$

The following is a 64 bit floating point format known as a **double**, where the sign, exponent and fractional parts are stored separately. Double has greater precision over float because it stores more bits in their exponent and fraction.

1	16	47	64 bit floating point number (**double**)
sign bit	exponent	fraction (mantissa)	$(-1)^{sign} * 2^{(exponent-127)} * (mantissa)$
-	12	.12365	$1.2365 * 10^{12}$

Variables

Variables let you store and retrieve data in a computer memory represented by a **variable name,** like **x.** The variable name lets you identify a particular memory location. Each computer memory location contains 8 bits known as a byte. Each memory location has an address. The addresses lets you access the value at that memory location. All address start at 0 and increment by the number of byres used by the variable. Some variables take 2 byes, so address for these variables will increment by 2. Instead of remembering the address of a value at a particular memory location you use a name like **x**. The compiler assigns the memory address automatically for you when it compiles your program.

The variable x represents the memory location where the value 5 is stored in the computer memory

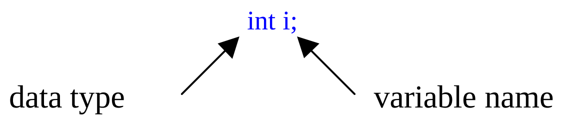
	address	**value**
	1006	
x	1004	5
	1002	
	1000	

A variable is like a bank account. You can put money in and take money out. The particular place where your money is located, is identified by the bank account number. The location where the variable is stored in memory is known as an address. Variables have different **data types**. Common data types used are **char**, **int**, **float** and **double** etc. When you want to use a variable you have to declare it. When you declare a variable you specify the data type and the variable name. Declaring a variable is like opening up a bank account and receiving a bank account number. The currency of the money would be the **data type** and the account number is the place in computer memory where the variable resides.

Declaring a variable

When you declare a variable you specify the data type and the name of the variable and end with a semicolon. All variables in C must be declared at the top of a function.

```
/* main function */
void main()
{
int i;

}
```



The name is used to represent the location in memory that the compiler has **reserved** for the variable. When variables are declared they have undefined values. This means the variables data value is unknown. Each variable is assigned a memory location represented by the variable name. The memory location is also known as an **address**. Each memory byte is one address. The address increases by the data type size and increments sequentially. Every variable get a value stored at the memory location the variable is representing. When a variable is declared the value of the variable is undefined, because at the memory location the variable is representing no new value has been stored there yet. Variable names are also known as **identifiers**.

```
data_type   variable_name;

int i;          /* declare a variable named i having a data type of integer */

char ch;        /* declare a variable named c having a data type of character */
```

address	name	value
1000	i	?
1002	ch	?

Why is the address of the variable **ch** at 1002 ?

More than one variable may be declared at a time in a **variable list**. Each variable declared in the list is separated by a comma. All variables declared are of the same data type.

```
data_type   variable_list;

int j, k, m;          /* declare a variable's i, j, k  having a data type of integer */
```

address	name	value
1003	j	?
1005	k	?
1007	m	?

The data type indicates what kind of data the variable is to represent. The variable name is used to represents the value at a particular address location in memory. Why is the address of the variable **i** at 1003 ? Why does each address increment by 2 ?

Initializing Variables

Variables may be initialized when they are declared. This would be like opening a bank account with an initial deposit. A variable is initialized when it is declared by using an **expression**. An expression represents a calculated value. An expression may be a constant or an another variable. If the expression is another variable, then the value represented by the variable is used for initialization not the name of the variable.

The expression on the left side is assigned to the variable on the right side.

```
data_type   variable_name = expression;

int x = 5;                /* declare and initialize x to 5 */

int y = x;                /* declare and initialize y to the value of x which is 5 */
```

y gets gets value of x which is 5

address	name	value	rep
1009	x	5	
1011	y	5	(x)

You may declare and initialize variables in a list.

```
data_type   initializaton_list;

int a=4, b=6, c=3;                /* declare and initialize a,b,c */
```

address	name	value
1013	a	4
1015	b	6
1017	c	3

Variables must be declared at the top of your program or in a function. When they are declared in a program, they are known as **global** variables. Global, meaning they are known to all functions. Global variables are to be avoided since they are the source of all data corruption. This means many functions will be accessing the global variables and you do not know which function put in the bad data. When a variable is declared in a function, they must be declared at the top of the function, and is known only to that function. Variables declared in a function are also known as **temporary** or **local** variables, because they are local to that function only. Variables in a function do not retain their value after the function is finished executing.

Assigning values to Variables

An assignment statements allow you to assign data values to a variable name by using the assignment operator "**=**". An assignment statement is like putting and taking money out of a bank account that was previously opened. Before an assignment statement can be used, the variable must be declared. You do not need to declare a variable again to assign or retrieve data from it. You only need to declare a variable once, but you can assign and retrieve values to the variable as many times you wish. You do not need to open up a bank account again every time you want to deposit or withdraw money. When you assign a new value to a variable, the old value is lost and is replaced with the new one. This is just the same as when you deposit money into your bank account, the old amount is replaced with the new amount. When you assign a value to a variable you are using the same address that the variable was declared with.

Assignments statements may be used anywhere in a function and have the form:

```
variable_name  =  expression;

i = 5           /* assign variable i the value of  5 */

ch = 'a'        /* assign variable ch the value of 'a' */

x = 3;          /* assign variable x the value of 3 */

y = x;          /* assign variable y the value of x which happens to be 3 */
```

address	name	value	rep
1000	i	5	
1002	ch	'a'	65
1009	x	3	
1011	y	3	(x)

y gets value of x which is 3

When we assign x to y we assign the value that x represents to y, which happens to be 3. The analogy here is we are transferring money from one account to the other.

If all variables need to be assigned the same value, then you can do it all at once. The value is assigned right to left. Assign a value to many variables.

```
variable_list = expression;

a = b = c = 10;               /* declare and initialize a,b,c */

                              /* assign values right to left */
```

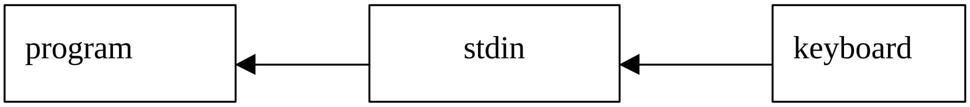
address	name	value
1013	a	10
1015	b	10
1017	c	10

The statement is evaluated **left** to **right** but executes **right** to **left**. **0** is assigned to **c**, **c** is assigned to **b** and **b** is assigned to **a**.

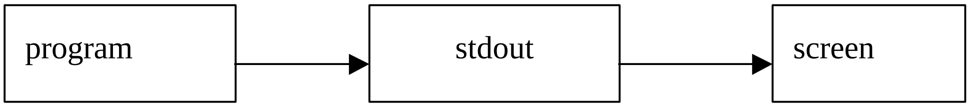
input/output streams

Data flows through data streams. The input data stream is called **stdin** and the output data stream is called **stdout.** The **input** data stream is used to get data from the keyboard and the **output** data stream is used to send data to the computer screen. You need to include the stdio library #include<stdio.h> before you can use the input and output data stream functions.

Input streams allow your program to receive data from the key board or a file.



Output streams allow your program to send data to the computer screen or to a data file.



write a message to the screen

To write a message to the screen the **printf()** function is used. The printf() function takes a character string that represents the message you want to print out on the computer screen.

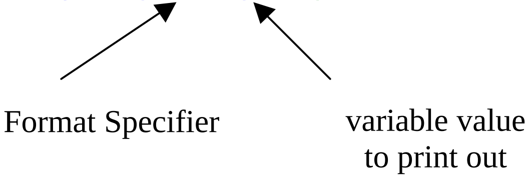
```
printf ( "string" );

printf("hello\n"); /* print hello to screen */
```

```
hello
```

The **'\n'** means start a new line on the computer screen **after** the string is printed out. You can also print out values of variables separately or with messages by using **format specifiers** in a **control string**. Format specifiers start with the percent sign and end with a letter that represents the data type of the variable you want to display on the screen. Format specifiers are listed in a control string , where values are to be **substituted** for the format specifiers. Every format specifier must have a corresponding value represented by a variable or constant.

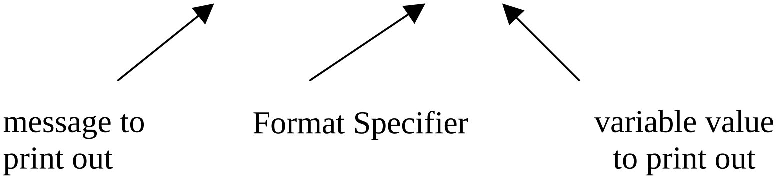
```
printf("%format specifier", variable);

printf("%d",x); /* print out value of variable */
```

```
3
```



In the above example the value of **x** is inserted where the **%d** format specifer is. You can also print out messages and variables with a printf statement. The printf function will now contain a control string that contains a message and format specifiers.

```
printf ( "control string", variable_list);

printf("the value of x is: %d\n",x); /* print out message and value of variable */
```



The output on the screen would look like this:

```
the value of x is: 3
```

The **%d** is a format specifier which means to print a signed integer value at this location in the control string. The value appears where the format specifier is located. The values to be printed are listed after the control string in order to match each format specifier.

Format specifiers that you can use in your control strings:

specifier	use for
%c	character
%d	integer
%h	short
%ld	long
%f	float

specified	use for
%uc	unsigned character
%u	unsigned integer
%uh	unsigned short
%uld	unsigned long
%lf	double

Make sure you match the correct format specifier with the correct data type of your variable.

getting values from the keyboard

To get a value from the keyboard you use **scanf()** function. **scanf()** also has a control string to identify the data types you want to read from the keyboard. The **scanf()** function also prints to the screen what characters you type on the key board this is called "echo to the screen"

```
scanf ( "control string", variable_address_list);
```

```
scanf("%d",&x); /* get number value from keyboard */
```

& specifies location of x (rather than value of x)

Format Specifier

The number entered on the keyboard will be deposited into the variable **x**. Do you know what the "&" (ampersand) means ? It means we want **scanf** to place the value it received at the variables location. You must be careful in using the %c format specifier.. The %c format specifier reads a individual character. When reading in a single character after reading an integer you may need also to read in the "enter key" using a \n,

```
scanf("\n%c",&ch); /* last character to read */
```

\n get enter key first

You can also use a leading space to remove the enter key

```
scanf(" %c",&ch); /* last character to read */
```

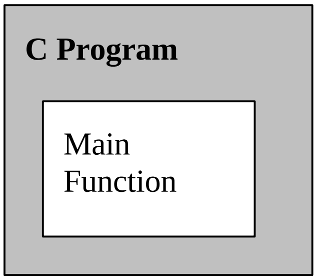
space get enter key first

Do you know why the enter key remain in the input stream ? When you read a number from the keyboard using another format specifier like %d, using scanf the enter key '\n' remains in the input stream . When you read a character using %c you first read in the enter key, and then scanf terminates early and returns the enter key. You don't even get a chance to read a character. The trick is to make your format control string to read in the enter key '\n' first and then and then scanf will wait for the user to type in a character. Scanf now will returned the character typed in.

main function

A minimal C program just only requires a main function. The main function is the first function to executed in your program. The job of the main function is to execute statements and call other functions. We introduce the main functions at this time so that you can run your first program and do the exercises.

```
#includes

void main( )

        {
        variables
        program
        statements
        }
```



```
/* Lesson 1 program 1 */
#include <stdio.h>
void  main ()

        {
        int x = 5; /* declare and initialized x to 5 */
        printf("hello\n"); /* print hello on screen */

        /* print out value of x on computer screen */
        printf("the value of x is: %d\n",x);

        /* ask user to type in a number */
        printf("please enter a number: ");

        /* get number from keyboard place value in x */
        scanf("%d",&x);

        /* print out new value of x on computer screen */
        printf("the value of x is: %d\n",x);
        }
```

program output:
(assume user typed in a 3)

```
hello
the value of x is: 5
please enter a number: 3
the value of x is: 3
```

The main function starts with the keyword **void** which indicates the **main** function does not return a value. Functions may return calculated values. The main function has the name **main. Function names** end with **round brackets** ( ) to distinguish the **function name** from a **variable name**. Functions may receive values from other functions. **The main** function in this example does not receive any values. The round brackets are empty. **Function statements** are introduced by a open curly bracket "{" and the function statement ends with a closing curly bracket "}". The above C program prints out the word "hello" on your computer screen, asks the user to enter a number from the keyboard to print out on the computer screen.

LESSON 1 EXERCISE 1

Type in the above program in your compiler and run it. Print out your name and ask the person to enter a letter instead of a number. Call your program L1Ex1.c.

LESSON 1 EXERCISE 2

Write a program that only has a main function that declares and initializes 5 variables of **different** data types. Print the values out to the screen. Use **printf** function to print the values to the screen. Next ask the user to enter a number for each of the variables. Use the **scanf** function to get numbers from the keyboard to place into each variable. Print out the new values using the **printf** function. Call your program L1ex2.c.

The **sizeof** operator tells you the number of bytes a data type or variable uses.

```
int sz = sizeof(x);

printf("%d\n",sz);
```

You can also use the data type rather tan the variable name.

```
int sz = sizeof(int);
```

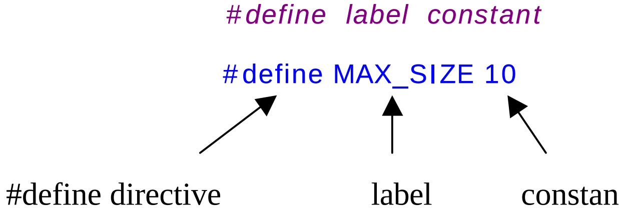
Print out the size in bytes of each variable in the above exercise.

EVOLUTION IN PROGRAMMING LANGUAGES

People think in a higher abstract level than computers do. A program needs to have meaning. People like to work with ideas and representations. Computers like to work with numbers. The job of the compiler is to translate a human ideas into numbers that a computer can work with. C has mechanisms that allow you to work in abstract representations. The C compiler will convert the abstract representation into a number for you automatically.

Define directives

The **define directive** let's you represent a constant by a **label.** A label is also called an **identifier** and made up of letters. Do not confuse a label with a keyword. Keywords are C language reserved words like **int**.



#define label constant

```
#define MAX_SIZE 10
```

Associates a constant with a label

By defining a label to represent a value this means every time the compiler sees MAX_SIZE the numeric value 10 is substituted. You must not put a semi-colon at the end of the #define statement because the compiler would substitute "10;" rather than what you want "10". Why do we need #define statements ? Using define statements is a must. There should be no hardcoded numeric values in your program. The purpose is this, if you change MAX_SIZE to be 12 then you do not need to change all 10's to 12's in your program. Without using a #define directive you may inadvertently change some 10's to 12's that don't need to be changed and then your program may not operate as expected. Instead of putting numbers in your program you put labels representing numbers. Define statements are at the top of your program. Do you know why ?

Here is a sample program using #define statement.

```
        #include <iostream.h>

        #define MaxSize 10      // define label MaxSize to represent constant 10
        void main()

                {
                 int x = MaxSize;
                 printf("max size  is  %d\n" ,"MaxSize) ;
                }
```

Program Output:

```
max size is 10
```

Typedef's

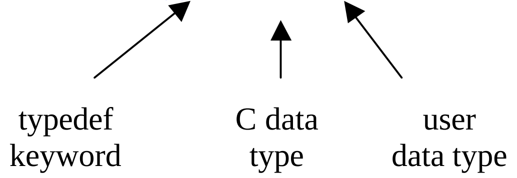
The next evolution in abstract data representation is **typedef** meaning **type definition.** Typedef's allow you to define your own **data types** from common C base data types **int, char, float**, **double** etc. using the **keyword** typedef. **Keywords** are **reserved** words that are command that define the C language. Your own data type must be made up of a known C data type. A common **typedef** most commonly used is **bool**. **Bool** means **true** or **false**. **True** is a non zero number where **false** is a zero number. **Typedefs** allow you to have your own user data type representing a C data type.

We need to use the **#define** directive to define **true** as value 1 and **false** as value 0:

```
                #define TRUE  1    /* true is any non zero value */
                #define FALSE 0    /* false is always zero */
```

To define your own data type you uses the keyword **typedef** followed by a C primitive data type and your own user data type.

```
                typedef    data_type    user_data_type;

                typedef int bool; /* define your own data type bool  meaning true or false */
```



Typedef's Associates a user data type with a C primitive data type

Now you have your own **bool** user data type that is the same as the C data type **int**. The advantage is this, to you **bool** means **true** or **false** but to the compiler bool means **int**. You must think that your data type is being substituted for a C data type. This is a very important concept in programming to grasp. The compiler is always substituting a sophisticated meaning to a simpler representation. Humans must think at a higher abstract level then a computer. **Typedef** allows you to define your own data type that has **meaning**. A computer does not need to do this. The computer has no feelings, a human being does. When you see **bool,** you must make the connection that your user data type has the **meaning true** and **false** and is being substituted for an int data type. Can you see how programming languages work. Soon as we get a new programming concept it uses previous concepts. The **typedef** mechanism is using mechanisms from #define. Before you can use your own data type you must declare a variable to represent your own user data type. The variable will be used to store the values **true** or **false**. We declare the variable by stating the user data type and the variable name.

We declare a variable to represent a **bool** data value.

```
user_data_type  variable_name;

bool flag;   /* declare a variable called flag having user data type bool */
```

When somebody sees the bool user data type they automatically make the assumption that the variable will represent a true or false value. When the compiler sees the user data type bool it substitutes in its own data type int.

```
int flag; /* compiler representation */
```

Now you can assign meanings like TRUE and FALSE to a variable.

```
flag = TRUE;     /* set the value of flag to TRUE */
```

When somebody sees the TRUE label they automatically make the assumption that the variable is a bool user data type. When the compiler sees TRUE it will substitute a 1. When the compiler sees FALSE it will substitute a 0.

```
flag = 1;  /* compiler substitutes a  1 for the label TRUE */
```

Here is an example program defining and using your own data type bool:

```
/* lesson 1 program 2 */
#include<stdio.h>
#define TRUE  1    /* true is any non zero value */
#define FALSE 0     /* false is always zero */
typedef int bool;  /*define own user data type bool to represent true and false */

void main()

        {
        bool flag;   /* declare a variable called flag having user data type bool */
        flag = TRUE;     /* set the value of flag to TRUE */
        printf("the value of flag is: %d",flag);
        }
```

```
the value of flag is: 1
```

Why does the output say 1 rather than "true"? True and false are represent by 1 and 0 respectively. True and false are used in situations where we want to represent a situation where something is either on or off, yes or no. When you see TRUE you know automatically that the variable **flag** will have TRUE and FALSE representation. You can make the assumption that variable flag must be a bool data type. This association concept is quite powerful in programming and has a great psychological impact.. With typedef's everyone is happy. The computer gets to work with its own data types and numbers and the humans get to work with their own user data types, with names and labels that have meaning.

LESSON 1 EXERCISE 3

Write a program that uses your own data type like days of week. Ask the user to enter some data for your days of week data type and display the results. You will need 7 definitions or constant values for days of the week. Start with Sunday equals 0. Declare a week day variable initialized to Monday. Call your file L1ex3.c.

ENUMERATION'S

enumeration's are an excellent way to assign a collection of **sequential values** to a group of **labels** associated with a common name. For example you may need labels to represents values for the days of the week. You need to assign 0 to Sunday and all the way to 6 for Saturday. This is a lot of work to do when you use #define statements:

```
#define SUN    0
#define MON    1
#define TUE    2
#define WED    3
#define THU    4
#define FRI    5
#define SAT    6
```

enum lets you do the same thing automatically, its like having many sequential #define statements.

Associates a sequence of labels with a common name

```
enum   enumeration_name   {  enumeration_list };
```

0	1	2	3	4	5	6

```
enum   DAYS_OF_WEEK   {  SUN,  MON,  TUE,  WED,  THU,  FRI,  SAT  };
```

The **enum** has the default **int** data type and the first label gets the substituted value of 0. All other labels get incremental values. To use an enumeration you declare an integer variable.

```
data_type  variable_name = enumeration_label;

int weekday = SUN;    /* weekday gets value of  label SUN which is 0 */
```

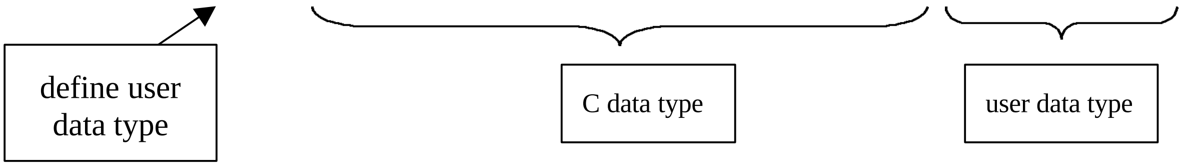
When the compiler sees SUN it substitutes the number 0.

```
int weekday = 0;    /* compiler representation */
```

Now you can use days of the week labels instead of numbers. No body walks around and says "today is 0 " they say "today is Sunday " right ? In programming you should be able to do the same.

By using the **typedef** directive you can also define a weekday data type. Typedef lets you define your own data type for your enums. DAYS_OF_WEEK is now a user data type having data type enum that has default **int** data type

```
typedef  enum  {enumeration_ list}  user_defined_labels;

typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT}DAYS_OF_WEEK;
```



Before you can use your DAYS_OF_WEEK data type you need to declare a variables having your user **enum** data type.

```
user_enum_data_type  variable_name  =  enumeration_label;

DAYS_OF_WEEK weekday = SUN;
```

When the compiler sees the user enumeration data type DAYS_OF_WEEK it substitute* its own data type int.

```
int weekday = 0;    /* compiler representation */
```

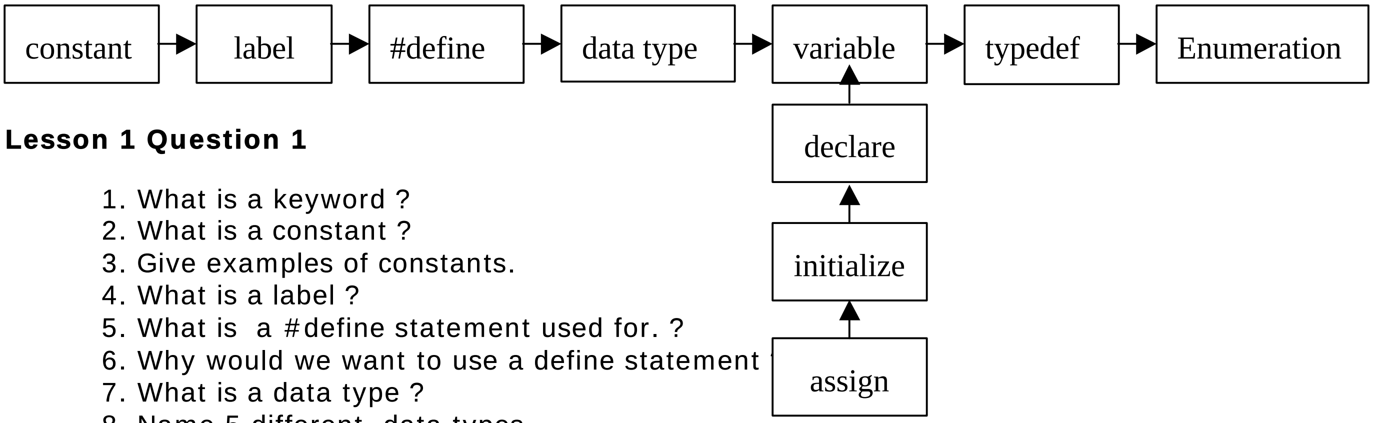
Using typedef is preferred since you can now have your own DAYS_OF_WEEK data type. Now you and the compiler knows that weekdays is a DAYS_OF_WEEK data type. This is extremely powerful in writing and debugging When you trace through your program it will say "MON" rather than "1" for values of weekday. Enumeration names and labels should start with a CAPITAL letter to distinguish them from variable names. Enumeration labels are defaulted to **int** data types. Enumeration's are defined at the top of the program outside of any function. Enumeration's are very powerful and have a big psychological impact in programming. Enumeration's allow the programmer to program with everyday things rather than using numbers. Is it not better to say: days equal MON rather than: days equal 1. Who says today is 1 ? Example program using enumeration's

```
/* lesson 1 program 3 */
#include <stdio.h>

typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT}DAYS_OF_WEEK;

void main()

        {
        DAYS_OF_WEEK day; /* declare a day data type */

        day = TUE; /* assign Tuesday to day */
        prinf("Today is: %d\n",day); /* print out day */
        }
```

program output:

```
Today is: 2
```

Why does the Program print out 2 rather than "Tuesday" ?



Lesson 1 Question 1

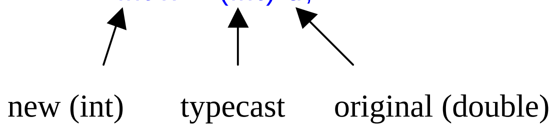
1. What is a keyword ?
2. What is a constant ?
3. Give examples of constants.
4. What is a label ?
5. What is a #define statement used for. ?
6. Why would we want to use a define statement
7. What is a data type ?
8. Name 5 different data types.
9. Why do we need different data types ?
10. What are variables used for ?
11. Why would you want your own data type ?
12. How would you make your own data type ? Give an example.
13. What does an enumeration do ?
14. Give another example of an enumeration**.**

TYPECASTING

C is a moderately typed language, this means every variable must be assigned to the same data type. It also means functions must also be passed data types it knows about. When you use other peoples functions you must supply them with the data type they know about. Nobody knows about your data types. You need to tell the compiler what they are or you may need to force one data type to another data type. This is what type casting is all about forcing an old data type value to a new data type value. The old data type value does not change. The new data type receives the forced converted value from the old data type value. You type cast by enclosing the forcing data type in round brackets proceeding the variable or value you want to force.

```
new_data_type = (typecast) original_data_type.
```

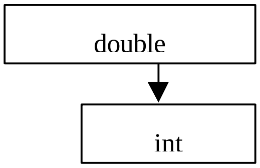
```
int x = (int) d;
```



(typecast)
Force one data type value to represent another data type value

A good example of type casting is trying to force a double to be an integer,

```
double d = 10.5;

int x = (int) d;
```



To be able to force a double data type into an int data type we type cast. When we typecast we loose some of the data. In this case we loose the 0.5 and x gets the value 10. int data types cannot represent fractional data.

```
int x = 5;
DAYS_OF_WEEK days = (DAYS_OF_WEEK)x; // force x to be a DAYS_OF_WEEK
printf("%d", (int)days); // force days to be a int
```

In the first example **x** is assumed to be an int data type that has an integer value that we force to a DAYS_OF_WEEK data type. In the second example we convert a DAYS_OF_WEEK data type to an **int** so that **printf()** can write a value to it. The **printf()** function does not know anything about the DAYS_OF_WEEK data type, but it knows what an int is. Although enumeration's have default of **int** the compiler only thinks **days** is a DAYS_OF_WEEK data type, the data type it was declared with.

LESSON 1 EXERCISE 4

Write a small program using the following questions that just includes a main function. All statements are sequential. All variables must be declared inside your main function at the top. You must use #define statements or enumeration's . Do not put any numbers in your main function statements. Call your program L1ex4.c.

1. Make a color data type using typedef and enumeration having three colors: RED, GREEN and BLUE.
2. In the main function declare a color data type variable and assign the value GREEN to it
3. Use the printf statement to print out the value of your color data variable.
4. Ask the user to enter a number between 0 and 2.
5. use **scanf** to get the number from the keyboard
6. Assign the number to your color data type variable
7. Use the printf statement to print out the value of your color data variable.

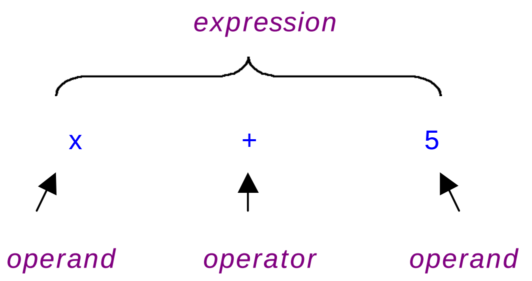
C PROGRAMMERS GUIDE LESSON 2

File:	CguideL2.doc
Date Started:	July 24,1998
Last Update:	Feb 28, 2002
Status:	proof

C LESSON 2 OPERATORS, BINARY NUMBERS AND CONDITIONS

OPERATORS

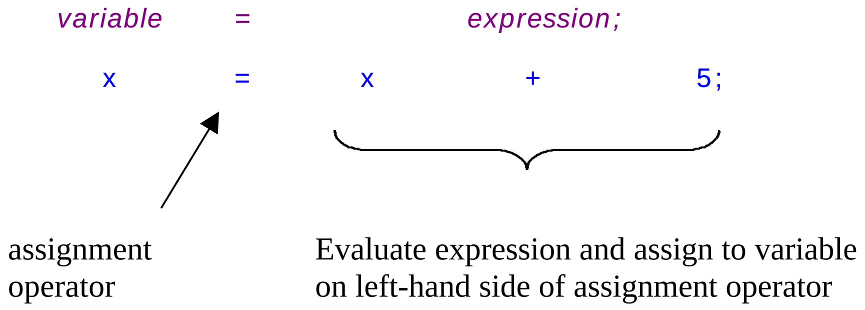
Operators are **symbols** that to do **operations** on variables and constants. The operations may be adding or comparing the values of variable and constants. The variables or constants are called **operands**. Operators with operands are known as **expressions**.



operators do operations on variables and constants

assignment operator

Assignment statements use the assignment operator to **evaluate** and **assign** expressions to a variable. The right hand side of the assignment operator is **evaluated** and **assigned** to the **variable** on the left hand side of the assignment operator. The original value of x is lost, it is overridden by the evaluated value of the expression.

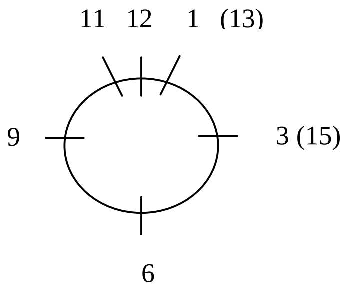

variable
=
expression;
x
=
x
+
5;
assignment operator
Evaluate expression and assign to variable on left-hand side of assignment operator

arithmetic operators

Arithmetic operators let you add, subtract, mutltiply and divide.

Evaluate the following for x = 10; y = 5.

+	addition	x = x + 5;	15
-	subtraction	x = x - 5;	5
*****	multiplication	x = x * y;	50
/	division	x = x / y;	2
%	modulus (remainder)	x = x % y;	0



To find the modulus of a number:

You multiply the remainder after division by the denominator. For example 15 mod 12 : **Divide 15/12 = 1.25** **You** use the remainder to calculate the mod: **0.25 * 12 = 3.** Therefore 15 mod 12 is 3	Or else you can subtract repeatively by 12 until you get a number less than 12 15 - 12 = 3

LESSON 2 EXERCISE 1

Write a C program that uses all the arithmetic operators and prints the results to the screen. Call your program L2Ex1.c

increment and decrement operators

The increment operator **++** adds 1 to a variable and the decrement operator **- -** subtracts 1 from a variable. They save you some typing. They work in a **prefix** mode and a **postfix** mode. **Pre** meaning increment/decrement the variable **before** the assignment and **post** means increment/decrement the variable **after** the assignment It is a 2 step process.

			example	**description**	**step 1 (before)**	**step 2 (after)**
++	increment after	y++	x = y++;	x gets value of y then y increments after assignment	x = y;	y = y + 1;
	increment before	++y	x = ++y;	y increments before assignment, then x gets value of y	y = y + 1;	x = y;
- -	decrement after	y - -	x = y - -;	x gets value of y then y decrements after assignment	x = y;	y = y - 1;
	decrement before	- - y	x = - - y;	y decrements before assignment, then x gets value of y	y = y - 1;	x = y;

LESSON 2 QUESTION 1

Fill in the values for x and y. **Assume all statements <u>continuous</u>.** The next line depends on the previous line.

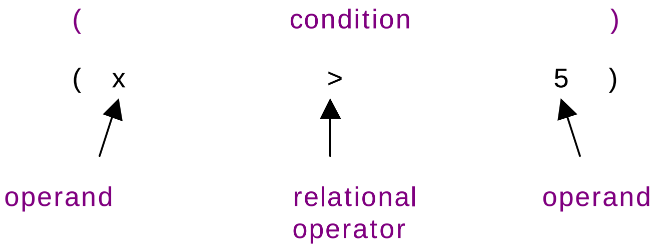
	x	**y**
y = 5;	**??**	**5**
x = y++;		
x = y--;		
x = ++y		
x = --y		

LESSON 2 EXERCISE 2

Write a C program to verify your answer above and print out the results to the screen. Call your program and class L2Ex2.c

Relational operators

Relational operators are used to **compare** the values of variables to see if they are **greater, greater or equal, less than** or **less than or equal**. Testing is done by a **test condition** having the form:



The test **condition** is evaluated as **true (1)** or **false (0)**. Evaluate the following for x = 5; y = 5;

>	greater than	(x > 5)	false	0
>=	greater than equal	(x >= 5)	true	1
<	less than	(x < y)	false	0
<=	less than or equal	(x <= y)	true	1

You can always print out the result of any comparison.

```
printf("%d",(x > 5));
```

```
0
```

LESSON 2 EXERCISE 3

Write a C program that uses all the **relational** operators and prints the results to the screen. Call your C program L2Ex3.c

Equality operators

Equality operators are used to test if a variable is **equal** or **not equal** to another variable or constant. Evaluate the following for x = 5; y = 5;

= =	is equal	(x = = 5)	true	1
! =	is not equal	(x != 5)	false	0

The **equality operator** is different from the **assignment operator** "=" be careful! C will accept the assignment operator or the equality operator in a test condition. Instead of testing if a variable is equal to an expression, the evaluated expression will be assigned to your variable. Now you are really in big trouble. What is the difference between (x = 5) and (x == 5) ? One is an assignment the other one is a test condition.

LESSON 2 EXERCISE 4

Write a C program that uses all the equality operators and prints the results to the screen. Call your C program and L2ex4.c

Logical operators

Logical operators are used to test if **both conditions** are true or if **either conditions** are true in an **expression**. Parenthesis are used around the test conditions to force which condition is to be evaluated first.

(expression   )

((x = = 0)   &&   (y != 0))

↗   ↑   ↖

((condition)   logical operator   (condition))

The logical AND operator && test if **both** conditions are true and the logical OR operator | | test if **either** condition is true. Evaluate the following for x = 5; y = 5;

&&	**both**	logical **AND**	((x != 5) && (x = = y))	test if x is not equal to 5 AND x is equal to y	false
| |	**either**	logical **OR**	((x = = 5) | | (x != y))	test if x is equal to 5 OR x is not equal to y	true

The **logical operators** are said to be **short circuited** or referred to as **lazy evaluation.** As soon as if finds one condition **not true** or **false** it will not test the other condition.

LESSON 2 EXERCISE 5

Write a C program that uses all the logical operators and prints the results to the screen. Call your program L3Ex5.c

LESSON 2 EXERCISE 6

Write a program that asks the user to type in three numbers. Write a logical expression that tests if the **first** number is larger than the **second** number && the **second** number is smaller than the **third**. Write another logical expression that tests if the **first** number is larger than the **second** number || the **second** number is smaller than the third number. Just print out the results of the logical expression. No **if** statements are required. Call your program L2Ex6.c.

bit wise operators and truth table

A truth table lists all the possible combinations of the input to calculate the outputs. You know how to add numbers so we will make a truth table for adding binary numbers.

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ +\ 0 & +\ 1 & +\ 0 & +\ 1 \\ \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1\ 0} \quad \text{(binary 2)} \\ & & & \uparrow \text{ carry bit} \end{array}$$

The bit wise operators work on individual bits. The bit wise operator truth tables should now be easier to understand. Instead of ADDing you will be **AND**ing, **OR**ing and **XOR**ing. The **bitwise operators** operate on individual bits of variables. You need to understand binary and hexadecimal numbers to use bit wise operators. The following table lists all the bit wise operators. where 0 means false and 1 means true.

bit wise operator		purpose	truth table			
&	bitwise AND	set bits to zero (mask bits)	0 & 0 = 0	0 & 1 = 0	1 & 0 = 0	1 & 1 = 1
|	bitwise inclusive OR	set bits to 1's (set bits)	0 | 0 = 0	0 | 1 = 1	1 | 0 = 1	1 | 1 = 1
^	bit wise exclusive XOR	mask/set bits	0 ^ 0 = 0	0 ^ 1 = 1	1 ^ 0 = 1	1 ^ 1 = 0

Did you notice every time you AND & a bit with 0 the answer is 0. Every time you OR | a bit with 1 the result is 1. When you XOR ^ two bits together that are the same you get a 0. When you XOR ^ two bits together that are different you get a 1.

There is another operator called the complement operator ~ that toggles 0's to 1 and 1 to 0's

~	one's complement	toggle bits	~ 0 = 1	~1 = 0

Example using bit wise operators where x = 5 (0101) binary.

<table>
<tr><td>x = x & 1;</td><td>x = x | 1;</td><td>x = x ^ 1;</td><td>x = ~x;</td></tr>
</table>

<table>
<tr><td>0101</td><td>0101</td><td>0101</td><td></td></tr>
<tr><td>& 0001</td><td>| 0001</td><td>^ 0001</td><td>~ 0101</td></tr>
<tr><td>0001</td><td>0101</td><td>0100</td><td>1010</td></tr>
</table>

LESSON 2 QUESTIONS 2

1. Fill in the values in the following chart for each bit wise operation. Convert all binary answers to decimal.

<table>
<tr><td></td><td></td><td>AND</td><td></td><td>OR</td><td></td><td>XOR</td><td>ones</td></tr>
<tr><td></td><td></td><td>1011</td><td></td><td>1011</td><td></td><td>1011</td><td>complement</td></tr>
<tr><td></td><td>&</td><td>0001</td><td>|</td><td>0001</td><td>^</td><td>0001</td><td>~1011</td></tr>
<tr><td>binary:</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>decimal:</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

2. How would you use the AND & bitwise operator to test if a number was even or odd ?
3. What do you think the XOR bit wise operator is used for ? Give an example.
4. What would we need the OR operator for ? Give an example.
5. What other uses would the bitwise AND & operator be used for ? Give an example.
6. How would you convert a **upper case** character like 'A' to a **lowercase** character like 'a' ?
7. How would you convert a **lower case** character like 'a' to a **uppercase** character like 'A' ?

Hint: 'A' is hex 0x41 'a' is hex 0x61

testing if a number is even or odd

Example using **bit wise** & operator to test if a number is even or odd:

```
result = (num & 1); /* test number if odd or even */
```

If the number is odd the result of the test condition is = 0x01 a true condition. Why?
If the number is even the result of the test condition is = 0x00 a false condition. Why ?
A true condition is considered a **non zero** value where a **zero** value is considered false.

You can also test a number if it is even or odd by using the % (mod) operator.

```
result = (num % 2); /* test number if odd or even */
```

When num is even then result is 0 (false), when the num is odd the result is 1 (true).

encryption using xor

Take any char like 'A' **xor** it with another character like 's' called a **key** and you get a secret character like '%'. Take your secret character '%' xor it with your key character 's' and you get your original character back 'A'. Try it !

binary numbers

A **memory cell** in a computer is called a **bit** and can have two possible states **off** or **on** and is represented as a 0 or a 1 respectively. Numbers containing 0's and 1's are known as **binary** numbers having a number base of 2. Since 0's and 1's are very difficult to work with, four bits are grouped together to represent the numbers 0 to 15. The 4 bits grouped together are known as a **hexadecimal** number having base 16 because $2^4 = 16$

0	1	1	0

Hexadecimal numbers use numbers 0 to 9, and letters A to F to represent the decimal numbers from 0 to 15. Letters are used because we want one character to represent a number. Hexadecimal numbers because they are more convenient to work with than binary numbers. The following chart shows the numbers 0 to 15 represented by bases binary, hexadecimal and decimal.

binary (base 2)	hexadecimal (base 16)	decimal (base 10)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

binary (base 2)	hexadecimal (base 16)	decimal (base 10)
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Why do we uses letters A to F ?

converting from binary to hexadecimal numbers

It's easy to convert a binary number top a hexadecimal number. All you have to do is group four binary bits together in groups.

<table>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td colspan="4">E</td><td colspan="4">5</td><td colspan="4">B</td><td colspan="4">9</td></tr>
</table>

convert binary: 0111001010110110 to hexadecimal

converting from hexadecimal to binary numbers

If you want to convert hexadecimal numbers to binary just use the chart ! Look up the hexadecimal number and write down the binary equivalent.

E				5				B				9			
1	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**

convert hexadecimal: 72B6 to binary

converting from binary to decimal

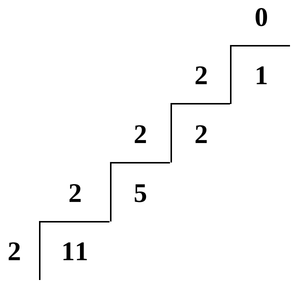
To convert a binary number to decimal you just multiply each bit positional weight and add them up. A positional weight is the power of 2 of the bit position. Bit positions start from the right most significant bit (RMSB) and has bit position 0. 2 to the power of 0 is 1 (2^0 =1). The weight increase by powers of 2 as you move left. Example take the binary number 1011

bit position	3	2	1	0	
each bit has a weight:	2^3	2^2	2^1	2^0	
power of two	8	4	2	1	
binary number	1	0	1	1	
multiply bit by weight	1* 8	0 * 4	1 * 2	1 * 1	
add up result	8	+ 0	+ 2	+ 1	= 11

Convert binary number 1101 to a decimal number.

converting from decimal base 10 to binary base 2

To convert a decimal number to its binary equivalent you continually divide all the quotients by 2 and keep track of the value of each remainder. Each remainder must be multiplied by the base (2). To get the correct binary number you then take the remainders in reverse. Example convert decimal 11 to binary 1011. Start at the bottom of the chart. We divide 11 by 2. The quotient is **5** and the remainder is **.5**. Multiply each remainder by the base 2. Continue with the other quotients. Take the answer from the top of the chart to the bottom: 1011. When you multiply each remainder by the base you are actually taking the mod 2 of the number.



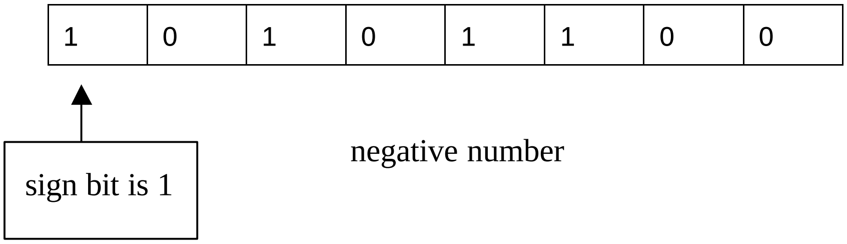
rem		**base**		**ans**
.5	*	**2**	=	**1**
.0	*	**2**	=	**0**
.5	*	**2**	=	**1**
.5	*	**2**	=	**1**

1 0 1 1

Convert decimal number 13 to a binary number.

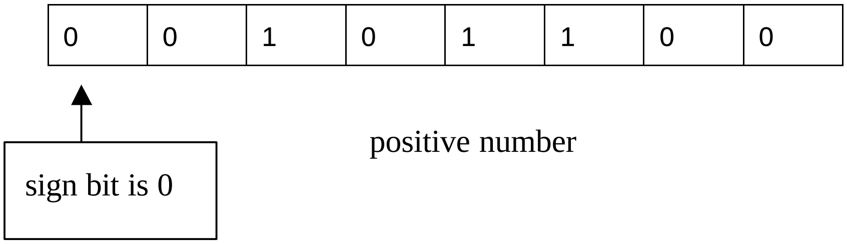
representing signed numbers

The most significant bit (MSB) is called the sign bit and determines if a signed number represents a negative or positive number. When the MSB or most left bit is a 1 then a signed number is considered **negative**.



1	0	1	0	1	1	0	0

When the MSB or most left bit is 0 then a signed number is considered **positive**.



0	0	1	0	1	1	0	0

The following chart lists all the numbers for a signed number using 4 bit resolution. When N is 4 and we use our signed number range formula:

```
-2**(N-1) to 2**(N-1)-1 out range is -2 ** 3 to 2 **3 -1 = -8 to 7
```

signed	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
binary	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
unsigned	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7

calculating negative numbers from positive numbers

How do we calculate negative binary representation for binary numbers ? To find out what binary number is represented by a negative number we use **two's complement** on a positive binary number to find its negative binary representation. You use **1's complement** to complement each bit and then **add 1** (2's complement) to convert from a negative binary number to a positive binary number or from a positive number to a negative number. The following example uses a bit resolution of 4 bits.

Convert	to negative		to positive	
number	0101	(5)	1011	(-5)
1's complement	1010	(-6)	0100	(4)
add 1	0001		0001	
result	1011	(-5)	0101	(5)

complement:

change 1 to 0

change 0 to 1

Convert 3 to -3 then to positive 3

shift operators

The Shift operators let you multiply or divide by powers of 2. There are two shift operators:

operator		definition	description	example
<<	shift left	variable << N	multiply by powers of 2	x << 1
>>	shift right	variable >> N	divide by powers of 2	x >> 1

Left shift operator <<

To multiply by a power of 2 you shift **left** N bits.

```
x = x << N;
```

To shift bits left you put an extra 0 on the right hand side of your binary number. Red represents the left most bit that will be lost when we shift right. Green represents the shifted bits and violet represents the added right bit after we shift right. Example shift 0101 left by 1 bit. There is no difference shifting signed or unsigned numbers left.

shift left

before	after
0101 (5)	1010 (10)

remove left
most bit

add zero
on right

Here is a small program that assigns 5 to x and shifts x left by 1 bit.

```
#include <stdio.h>

/* shift number left */
void main()
{
int x = 5;
printf("%d\n",x);
x = x << 1;
printf("%d\n",x);
}
```

program output:

```
5
10
```

Try shifting 0011 left by 1 bit.

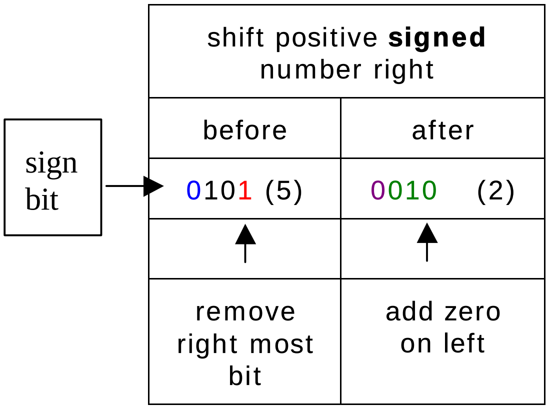
What happens when you shift a signed negative number left ? Try it !

Right shift operator >>

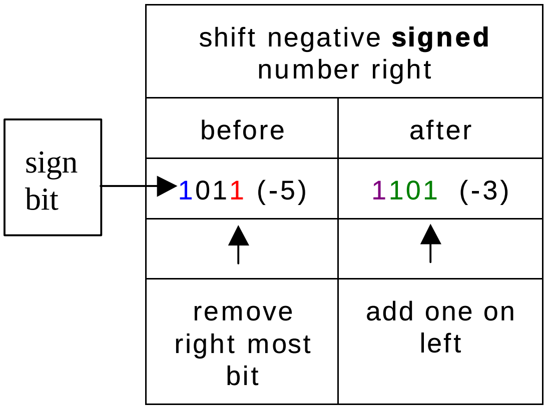
To divide by a power of 2 you shift right N bits.

```
x = x >> N;
```

To shift bits right you put an extra bit on the left hand side of your binary number and remove the right most bit. The extra bit on the left is a 0 if the left most bit is 0 indicating a positive signed number and is 1 if the left most bit is 1 indicating a negative signed number. Shifting numbers 0101 (5) and 1011(-5) right by 1 bit:



shift positive **signed** number right	
before	after
0101 (5)	0010 (2)
↑	↑
remove right most bit	add zero on left



shift negative **signed** number right	
before	after
1011 (-5)	1101 (-3)
↑	↑
remove right most bit	add one on left

When you shift a negative number right the msb but remain a 1 to preserve the negative number. This is called signed extension. This means if you divide a negative number by 2 then the answer must still be negative. When we shift right the right most bit is lost. We cannot represent fractions with integers data types. When we shift right it appears the values are truncated to a smaller value 5/2 = 2.5 = 2. (2 is smaller than 3) -5/2 = -3.(-3 is smaller than =-2). When we shift right the left most shifted bit is a 1 if the shifted number is a negative signed number, otherwise the most left shifted bit is a 0.

Example program to shift right.

```
#include <stdio.h>

/* shift positive number right */
void main()
{
int x = 5;
printf("%d\n",x);
x = x >> 1;
printf("%d\n",x);

/* shift negative number right */
int x = -5;
printf("%d\n",x);
x = x >> 1;
printf("%d\n",x);
}
```

program output:

```
5
2
```

```
-5
-3
```

Try shifting 0110 **signed** right by 1 bit
Try shifting 1110 **signed** right by 1 bit
Try shifting 1111 **signed** right by 1 bit

shifting unsigned numbers right

If you shift right an unsigned number then the left most added bit will always be 0. Example shift 1011 (11) left by 1 bit. Blue represents the sign bit. Red represents the right most bit that will be lost when we shift right. Green represents the shifted bits and violet represents the signed extended bit after we shift right. 11/2 = 5.5 = 5 integer.

unsigned shift right	
before	after
1011 (11)	0101 (5)
↑	↑
remove right most bit	add zero on left

Here is a small program that assigns -5 to x and shifts **unsigned** int x left by 1 bit.

```
#include <stdio.h>

/* shift unsigned number right */
void main()
{
unsigned int x = -5;
printf("%d\n",x);
x = x << 1;
printf("%d\n",x);
}
```

program output:

```
65531  (-5)
32765
```

Try shifting 1110 **unsigned** right by 1 bit
Try shifting 1111 **signed** right by 1 bit

LESSON 2 QUESTIONS 3

1. Fill in the values in the following chart for each bit wise operation. Convert all binary answers to decimal.

	unsigned	**signed**	**signed**	**unsigned**
	shift left	**shift left**	**shift right**	**shift right**
	1011 << 1	**1011 << 1**	**1011 >> 1**	**1011 >> 1**
binary:				
decimal:				

2. When we shift right where does the most right binary bit go ?
3. Why are the left shift and right shift bitwise operators so important ?

shift operator summary:

shift op	operation	purpose	operation	binary
<<	left shift (multiply by 2^N)	shift n bits left by power of 2	5 << 1 = 10	0101 << 1 = 1010
>>	right shift (divide by 2^N)	shift N bits right by power of 2	5 >> 1 = 2	0101 >> 1 = 0010
>>	right shift (divide by 2^N)	shift N bits right by power of 2	-5 >> 1 = -3	1011 >> 1 = 1101

LESSON 2 EXERCISE 7

Write a C program that just has a main function. In your main function initialize a variable to a positive number. Use all the shift operators to shift by 2 and print out the results. Next initialize the variable to a negative number. Use all the shift operators to shift by 2 and print out the results. Test if the shifted numbers are even or odd and print out the result of the test. Call your C program file L2ex7.c.

shortcut operators

Short cut operators allow you to save typing in adding numbers etc.

+=	x += 5;	x = x + 5;
-=	x -= 5;	x = x - 5;
&=	x &= 5;	x = x & 5;
<<=	x <<= 2;	x = x << 2;

*=	x *= 5;	x = x * 5;
/=	x /= 5;	x = x / 5;
|=	x |= 5;	x = x | 5;
>>=	x>>=2	x = x >> 2;

%=	x %= 5;	x = x %5;
^=	x ^= 5;	x = x ^5;

What is the difference between `x += 5;` and `x = +5;` ?

x += 5; **means** **x = x + 5;**

x = +5; **means** **x = 5;**

Precedence

Precedence tells the compiler which operations are to be performed first for expressions. Round brackets are used to force the compiler to evaluate which operations are to be performed first. If round brackets are not used then the compiler must decide for you. The compiler makes it decision by using a **precedence table**. The precedence **level** in the table tells the compiler which operations are to be performed first. Associativity is a term that states in which order operations are to be evaluated from left to right or from right to left. Operators in an expression having the same level precedence are evaluated to the associatively direction.

Precedence Table

The important thing here to realize from this table is that multiplication and division have higher precedence them addition and subtraction.

operator	level	associativity	operator	level	associativity
[ ] . ()(function call)	1	left to right	&	8	left to right
! ~ ++ -- (cast)	2	right to left	^	9	left to right
* / %	3	left to right	|	10	left to right
+ -	4	left to right	&&	11	left to right
<< >>	5	left to right	||	12	left to right
< <= > >=	6	left to right	?:	13	left to right
== !=	7	left to right	,	15	left to right
= += -= *= /= %= &= |= ^= <<= >>= >>>=				14	right to left

Examples forcing precedence and using precedence are as follows.

```
a = 1; b = 2; c = 3; d = 4;

x = a + b * c + d;        /* Which operations are done first ?  */      b * c then + a + b
                          /* What is the value of x ?  */

x = (a + b) * (c + d);    /* Which operations are done first ? */       a + b then c + d then *
                          /* What is the value of x ? */
```

LESSON 2 EXERCISE 8

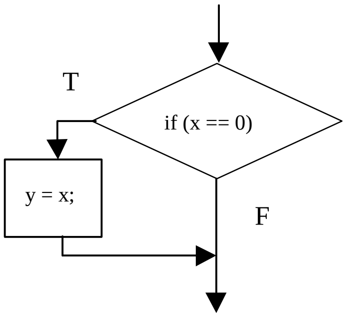
Write a C program that just has a main function. In your main function design your own mathematical equation using all the arithmetic operators. Assign values to all of your variables, don't use any round brackets. Print out the results. Put round brackets in your mathematical equation and print out the results Compare both answers print out if they are the same or not. Call your C program file L2ex8.c.

if statement

The **if statement** is uses **test condition** evaluated as **true** or **false** to make a choice or decision in a program flow. An **if** condtition contains one or more condition that are evaluated as **true** or **false**. When the **if condition** is **true** then the statements belonging to the **if statement** is executed. When the **if expression** is **false** then program flow is directed to the next program statement. If an **if statement** has more than one statement them the statements belonging to the **if** expressions must be enclosed by curly brackets. An if statement allows you to make a **choice** in program execution direction.

definition	one statement	more than one statement

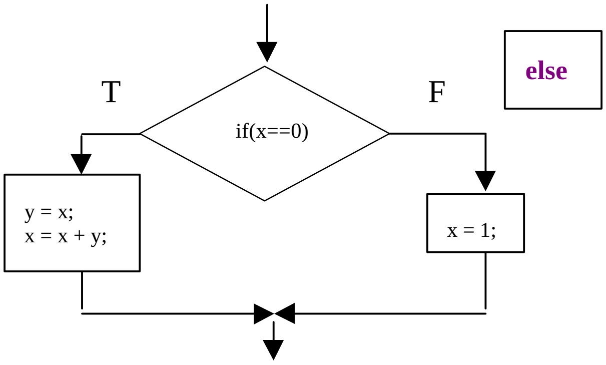
if (condition)
true statement(s)
	if (x == 0)
	y = x;
		if (x == 0)
		{
		y = x;
		x=x+1;
		}



if - else statement

An if statement may have an optional **else** statement. The else statement executes the alternative **false** condition. Curly brackets are also needed if more than one statement belongs to the **else** statements.

if (condition)
true statement(s)
else
false statement(s)
	if (x == 0)
	{
	y = x;
	x=x+1;
	}
	else
	x=1;



Conditional Operators ? :

The conditional operator **?:** evaluates a condition, **if** the condition is **true** it returns the value of the true expression **else** returns the value of the false expression. It's like an **if-else** statement but in a condensed form.

```
        ( condition ) ? true_expression : false_expression;

    y = (x == 5) ? x+1 : x-1;  /* assign result of expression to y */
```

This means if x is equal to 5 assign x + 1 to y else assign x - 1 to y.

```
      if ( x == 5)
            y = x+1;
      else
            y = x-1;
```

?	**if**
:	**else**

LESSON 2 EXERCISE 9

Continue the program from Lesson 1 Exercise 2. All statements are sequential. All variables must be declared at the top inside your main function. You must use #define statements or enumeration's. Do not put any numbers in your main function. You can use **printf** to print out the values of your variables, **scanf** to get numbers from the keyboard and the **if** and **else** statements to make decisions in program execution flow. Call your program L2ex4.c.

1. Make a color data type enumeration using typedef having three colors:
RED, GREEN and BLUE.
2. Declare a color data type variable and assign the value GREEN to it
3. Ask the user to type in a number between 0 and 2.
4. If the number is greater than BLUE tell them number is too high.
5 If the number is less than RED tell them the number is too low.
6 If the number matches the color print out " match ".
7. else change the color variable to the number they choose
8. Print out the color corresponding to the number entered.

C PROGRAMMERS GUIDE LESSON 3

File:	CGuideL3.doc
Date Started:	July 12,1998
Last Update:	Feb 27, 2002
Status:	proof

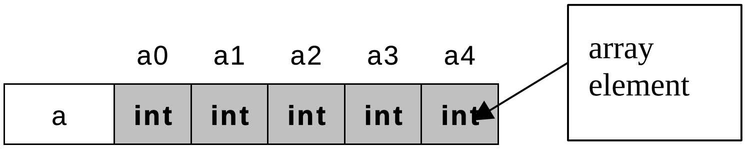
LESSON 3 ARRAYS AND POINTERS

ARRAYS

An array is like declaring many variables after each other all having the same data type.

```
int a0;
int a1;
int a2;
int a3:
int a4;
```

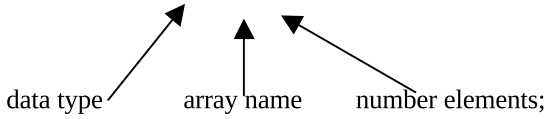
The **data type** represents the kind of data the memory is supposed to represent. Data types are **int, char, long, float**, **double** etc. Instead of declaring many separate data types an array lets you declare a group of declared variables having the same data arranged consecutively in memory referred to by a common name like **a**. Each consecutive variable is known as an array **element**. Each array element can store a value.



To declare an array we specify the desired data type, the array name and the number of elements we need enclosed in square brackets. Memory for the array is reserved in compile time.

data_type array_name [number_of_elements];

```
int a[5]; /* declare an array of size 5 of data type integer */
```



Why do we need arrays ?

We need arrays because we want to access a group of numbers that have been stored under a common name. For example, if we want to record the temperature of a day for each hour. Each recorded temperature will be an element of the array.

int temperatures[5];	**34**	**35**	**37**	**34**	**32**

1 dimensional arrays

An array with only a single **row** of data it is known as a **1 dimensional array**. The row has the numbers of columns equal to the array size. Array **a** will have 1 row and 5 columns. Each column in a 1 dimensional array is known as an array element. When an array is declared the value of the array element is unknown.

```
int a[5];
```

a	**int**	**int**	**int**	**int**	**int**
	??	??	??	??	??

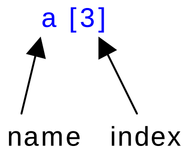
Each element in an array is accessed by an index. In C arrays indexes start a 0. When you have an array of 5 elements the first element always has an index of 0 where the last element has an index of 4. Don't forget this. It is easy to get mixed up when accessing array elements. To avoid confusion we will always refer to array rows and columns as indexes that start at 0.

	0	**1**	**2**	**3**	**4**
a	**int**	**int**	**int**	**int**	**int**

accessing array elements

To access an array element we specify which array element we want by enclosing an array index in square brackets. All array elements start at index 0.

array_name[array index]



array elements are accessed by an index

assign a value to an array element:

The value of the expression is assigned to the array at the specified index.

array_name[index] = expression;

```
a[3] = 5;  /* set the array element at index 3 to value 5 */
```

a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**
??	??	??	5	??

It is a three step process to assign a value to an element of an array: `a[3] = 5;`

step 1:	go to array	a
step 2:	go to array element specified by index	a[3]
step 3:	assign value to this array element	a[3] = 5;

read a value stored in an array element:

Assign the value read from the array to a variable.

variable = array name[index];

`x = a[3];` /* assign the value at array index 3 to x */

a[0]	a[1]	a[2]	a[3]	a[4]
??	??	??	5	??

It is a four step process to read a value to an element of an array: `x = a[3];`

step 1	go to array	a
step 2	go to array element specified by index	a[3]
step 3:	get value at this array element	value = a[3]
step 4:	assign value to variable	x = value

What is the value of x ?

LESSON 3 EXERCISE 1

Write a small program that just has a main function by answering the following questions. Call your program file L3ex1.c. **Do not use loops**.

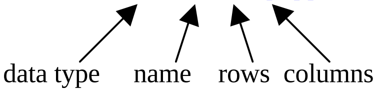
1. Declare a 1 dimensional array of size 5 columns called **a** of data type int
2. Assign the value 0 to 4 to each column index.
3 Print out the values of each array element using printf

2 dimensional arrays

Arrays that have more than 1 row are called **2 dimensional** arrays and are declared with two square brackets [ ] [ ]. The first square bracket indicates how many **rows** and the second square bracket indicates how many **columns.** A 2 dimensional array is like a grid.

data_type   array_name [number_of_rows] [number_of_ columns];

`int b [3][4];`   /* declare a 3 row by 4 column two dimensional array */



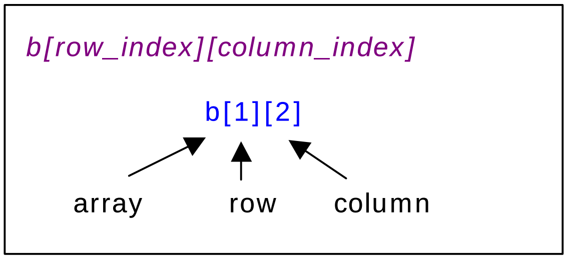
3 rows by 4 columns 2 dimensional array of 12 elements

	col 0	col 1	col 2	col 3
row 0	int	int	int	int
row 1	int	int	int	int
row 2	int	int	int	int

Two dimensional arrays in C are arranged in **row major order**. This means the first dimension is the number of **rows** and the second dimension is the number of **columns**. All array indexes start at zero. There are 3 rows having **row indexes** 0 to 2 and 4 columns having **column indexes** 0 to 3. Why do the rows have indexes 0 to 2 and the columns have indexes 0 to 3 if it is a 3 by 4 two dimensional array ? Because indexes start at zero !

accessing 2 dimensional array elements

To access individual elements of the 2 dimensional array you specify which array row and column index you want to access in square brackets. The first bracket specifies which **row index** and the second bracket specifies which **column index**.



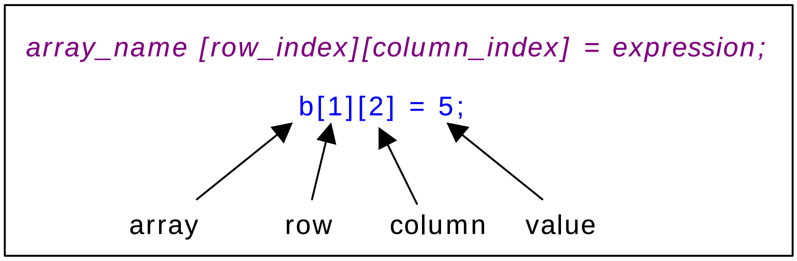
b[row_index][column_index]

```
b[1][2]
```

	col 0	col 1	col 2	col 3
row 0	b[0][0]	b[0][1]	b[0][2]	b[0][3]
row 1	b[1][0]	b[1][1]	b[1][2]	b[1][3]
row 2	b[2][0]	b[2][1]	b[2][2]	b[2][3]

assigning values to a 2 dimensional array

To assign a value to an element location inside a two dimensional array you specify the row index and the column index in square brackets and the value you want to assign. The expression can be a constant another variable or an arithmetic expression. Assign 5 to array element at row index 1 and column index 3



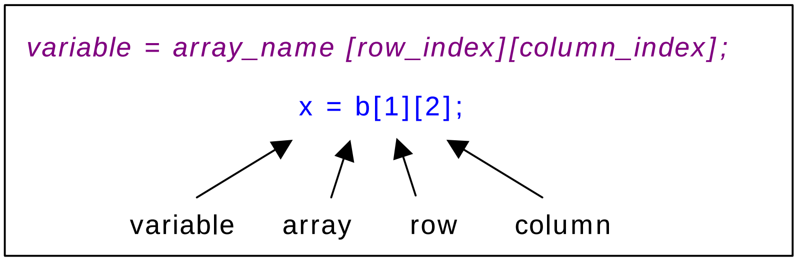
array_name [row_index][column_index] = expression;

```
b[1][2] = 5;
```

	col 0	col 1	col 2	col 3
row 0	??	??	??	??
row 1	??	??	5	??
row 2	??	??	??	??

reading a 2-dimensional array

To read a value from an element stored in a two dimensional array, the array element specified at the row and column index is assigned to the variable on the left of the assignment statement. Read the array element value at row index 1 and column index 2 :



variable = array_name [row_index][column_index];

```
x = b[1][2];
```

	col 0	col 1	col 2	col 3
row 0	??	??	??	??
row 1	??	??	5	??
row 2	??	??	??	??

What is the value of x ?

LESSON 3 EXERCISE 2

Write a small program that just has a main function by answering the following questions. Call your program file L3ex2.c.

1. Declare a 3 by 3 two dimensional array called **b** of data type int..
2. Assign to each array element in the array the value of its row and column.
   (example row 1 column 2 would get the value 12);
3. print out the values of each array element using printf

Declaring and Initializing Arrays

Arrays may be **initialized** with constants when they are declared. Initializing means you can **set** each array element to the value you want when you declare the array. An initializing **list** is used to initialize an array when it is declared. The initialization list, lists all the values in sequence you want to initialize the array with. An initializing list starts with an open curly bracket, lists each array element value separated by a comma and ends with a closed curly bracket.

{value0,value1,value2,value3,value4}

To declare and initialize a 1 dimensional array of size 5:

data_type array_name[size] = { initialization_list};

```
int a[5] = {10,12,34,21,24};
```

	0	1	2	3	4
a	**10**	**12**	**34**	**21**	**24**

initializing a two dimensional array

To initialize a 2 dimensional array, you list all the element values for the 2 dimensional array, in **row major order**. Row major order means you list all the column elements for each row first. Each row of column elements is grouped together with separate curly brackets. If you do not include the extra curly brackets for each row you will get the partially bracketed warning message from the compiler. Each row has its own sets of brackets

data_type array_name[rows][columns] = { initialization_list}

```
int b[3][4] = {{45,46,23,29},{56,23,76,52},{12,45,32,26}};

/* initialize a 2 dimensional array  3 rows by 4 columns */
```

45	**46**	**23**	**29**
56	**23**	**76**	**52**
12	**45**	**32**	**26**

setting all the array elements to zero

You can uses the initialization list to set all the elements of one and two dimensional array to zero,

```
int a[5] = {0};  // set all one dimensional array elements to zero

int b[3][4] = {0}; // set all two dimensional array elements to zero
```

This works for most compilers.

Character string arrays

Arrays of characters are known as character strings. The last character of a character string must be a string terminator character represented by '\0'. The C compiler expects all character strings to end with a terminator. You need to always include 1 extra character when you declare the size of the character string array. To declare a character string with room for 5 characters and the end of string terminator character:

```
data_type array_name [ size];

char sa[6];       /* declare character string */
```

	0	1	2	3	4	5
sa	**char**	**char**	**char**	**char**	**char**	**'\0'**

end of string terminator ↑

initializing character strings

You can also initialize character string arrays. You need to count how many characters you need and maybe leave a couple of extra characters for safety. The compiler automatically inserts the string terminator character at the end.

```
char s[10] = "hello";  /* extra characters are set to '\ 0' */
```

h	e	l	l	o	'\0'				

If you cannot count, then let the compiler count for you automatically. You indicate to the compiler to count for you by not specifying the number of characters you want [ ]. The compiler will count all the characters needed in your character string automatically for you and also include 1 extra count for the character string terminator character.

```
char s[ ] = "hello"; /* compiler counts characters for you */
```

h	e	l	l	o	'\0'

Letting the compiler count for you will save lots of computer memory. In the first situation we will have lots of wasted memory. In the second situation we have no wasted space. In either case an end of string terminator '\0' is assign to the end of each character string. When you specify the size you must include 1 extra character for the end of string terminator or else your program will crash. To print out a character string to the screen you use the **"%s"** format specifier.

```
printf("%s", s);  /* print out character string s */
```

You do not need to specify any index. because it will print out from the start of the character string and print out each character until the end of string terminator is found.

initializing 2-dimensional array of characters

You can also initialize a two dimensional arrays of character strings. For character strings the curly brackets for each row can be omitted, because each character string is already treated like a row. Notice each row of columns has 10 characters.

```
/* initialize a 2 dim array of character strings */

char s1[2][10] = {"hello", "goodbye"};
```

h	e	l	l	o	'\0'				
g	o	o	d	b	y	e	'\0'		

You must always specify the number of columns when declaring two dimensional arrays of character strings. The row size is optional. If you cannot count then the compiler can count the number of rows for you. You must always specify the number of columns. Why ? The columns need to be the same fixed length so that the start of each row can be found.

```
char s2[ ][20] =  {"hello", "goodbye"};   // omit number of row specified
```

two dimensional character string arrays

Before you can have a 2 dimensional array of character strings you need to make a user data type character string, of the largest number of characters you are going to use.

```
typedef char ta[20];  // declare character string data type
```

You can declare and initialize a 2 dimensional array of your user data type of 20 characters

```
ta b[2][3] = {{"hello","goodbye","tomorrow"}, {"one","two","three"}};
```

hello	goodbye	tomorrow
one	two	three

The complete program is as follows:

```
#include <stdio.h>

typedef char ta[20];  // declare character string data type

void main()
{
// declare and initialize a 2 dimensional array of character strings
ta sb[2][3] = {{"hello","goodbye","tomorrow"}, {"one","two","three"}};

// print out values
printf("%d %d %d\n, sb[0][0], sb[0][1],sb[0][2]);
printf("%d %d %d\n, sb[1][0], sb[1][1],sb[1][2]);
}
```

LESSON 3 EXERCISE 3

Write a program that initializes a 1 dimensional array of character strings of 5 different words. You need to make a character string data type. Ask the users to type in a number between 1 and 5. Print out the word from the 1 dimensional array corresponding to the word stored in the array. Repeat this 5 times. Do not use loops. Call your program file L3ex3.c.

Array index incrementers and decrementers

An array index specifies which element of an array you want to access. The **incrementor operator ++** automatically adds 1 to an array index. The **decrementor operator --** automatically subtracts 1 from an array index. They operate in a **prefix** mode and a **postfix** mode. **Pre** means before the assignment and **Post** means after the assignment. When using array indexes the contents of the array element selected by the array index is accessed. The array index is incremented or decremented before or after the assignment. It is a two step process the value is a**ccessed** from the array and the index is incremented or the index is incremented and the value **accessed** from the array.

statement	description	step 1	step 2
x = a[i++];	post **increment** array index i by 1 **after** the assignment	**x = a[i];**	**i = i+1;**
x = a[++i];	pre **increment** array index i by 1 **before** the assignment	**i = i+1;**	**x = a[i];**
x = a[i--];	post **decrement** array index i by 1 **after** the assignment	**x = a[i];**	**i = i-1;**
x = a[--i];	pre **decrement** array index i by 1 **before** the assignment	**i=i-1;**	**x = a[i];**

LESSON 3 QUESTION 1

A 1 dimensional array is pre-initialized with the following values and i is initialized to 2.

```
int a[5] = {23,18,35,42,10};
i = 2;
```

23	18	35	42	10

Fill in the columns for each value of x and i. before the statement is executed and after the statement is executed. Assume each statement is **continuous**. Each line depends on the last, the next statement relies on the previous statement. Hint: write the array index offsets on the top of the array box before proceeding.

statement	i before assignment	x	i after assignment
x = a[i++]	2		
x = a[++i]			
x = a[i--]			
x = a[--i]			

POINTERS

Pointers are variables that point to a block of memory. A Pointer is simply a variable that holds an **address value** rather than a **data value**. What is an address ? When you declare a variable the compiler automatically assigns a location in the computer memory for it. The address specifies a location in the computer memory where a value is stored. When you declare a variable the name of the variable represents the memory address location. When you assign a value to a variable, the value is placed in the computer memory address location represented by the variable name. When you read the value of a variable the value is taken from the memory address location. When you declare a pointer variable the compiler also assigns a memory location for the pointer variable, represented by the pointer variable name. The contents at that memory location is another address location not a data value. The only difference between a pointer variable and a non-pointer variable, is that the contents of the pointer variable represents an **address** location rather than a **data** value.

pointer variable

p	address location

non-pointer variable

x	data value

You declare a pointer just like as you would an ordinary variable.

data_type variable_name;

```
int x;  /* declare variable x of data type int*  */
```

address	name	value
1000	**x**	**??**

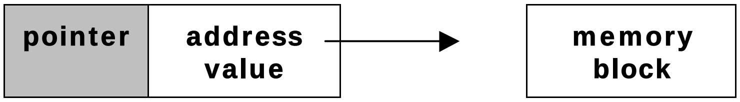
When you declare a variable the value or contents is unknown. x is at an address location and contains an unknown data value. You declare a pointer variable with the data type followed by a * (star) and the pointer's name. The * (star) preceding the pointer variable name is used to indicate this variable is a pointer variable. The pointer variable holds an address location rather than a data value. In C the * star goes with the variable name.

data_type pointer_variable_name;*

```
int* p;   /* declare p having data type int*  */
```

address	name	contents
2000	p	??

p is a pointer variable whose value or contents contain an address location. The contents is an address and is what the pointer points to. The contents has the address of a memory location holding a data value. Notice p is at an address location 2000 and its contents will contain an address location.



A pointer is a variable that stores an address value rather than a data value

The only difference between a variable and a pointer is that you declare a pointer with a * The * indicates the variable will hold an address value rather than a data value. Pointers may point to memory blocks from 1 byte to many bytes. When we say **point** we mean p contains the **address** of the first byte in the memory block. Memory blocks are made up of consecutive bytes of data. **Data types** are made up of 1 bytes or many bytes. Data type **char** is 1 byte, **int** is 2 bytes and **float** is 4 bytes. The pointer must know what kind of data type it is pointing to. The pointer stores an address value. It is this address value that lets the pointer, read and write data values to and from memory locations. The data type of the pointer specifies what kind of data the pointer is representing.

When a pointer is first declared it does not contain or point to an address. You must assign an address to it or allocate memory to the pointer. If you declare a pointer variable and you do not assign an address to it then you should assign **NULL** to it. NULL is used to represent no known address. If you do not assign an address to a pointer then the pointer will contain **garbage**. Garbage is what crashes your program and may even lock up your computer. On some compilers you may have to use 0 rather than NULL.

data_type variable_name = NULL;*

```
int* p = NULL;     /* declare pointer p with no address assigned */
```

address	name	contents
2000	p	NULL

pointer analogy

An analogy to a pointer is a mail man who delivers mail. For each letter delivered he knows who to deliver the letter to from the address written on the letter. You can consider the letter is a pointer variable with a stated address written on it. The mailman can be considered the executing program, which reads the address and delivers the letter to the house. The letter is pointing to the house. The house can be considered a memory block containing data contents. The letter is considered the pointer.

letter ----> house   pointer ------> memory

Why do we need pointers ?

We need pointers to access portions of blocks of memory that were declared in compile time or to memory allocated in run time. In compile time memory has been reserved for your program before it is ran. A pointer can point to individual bytes of the reserved memory. When your program is running your program may need additional memory. You may not know how much additional memory you may need before your program runs. A pointer must be used to allocate the additional memory in run time. When you declare a variable, the variable name represents a memory location. A pointer points to a memory location that may or may not have a name. The only way to access memory not represented by a name is with a pointer.

We need pointers :

(1) allocate memory when program is running
(2) point to existing memory

allocating memory using a pointer

A pointer may point to existing memory or to allocated memory. Memory is allocated in C by the **malloc** or **calloc** functions defined in <stdlib.h>.

function	prototype	description
malloc	void *malloc (size_t size);	allocates a block of memory **size** in bytes
calloc	void *calloc (size_t nitems, size_t size);	allocates a block of memory **size** in bytes and how many items **nitems** The memory block is cleared to 0.

Memory is allocated in **run time** when the program is running. **Malloc** or **calloc** can allocate memory from 1 byte to many bytes. **Malloc** is used more than **calloc**. **Calloc** is like **malloc** except it clears the allocated memory to zero and allows you to allocate many item blocks of memory. To use **malloc** or **calloc** you need to include the header file <stdlib.h> in your program or else your program will not operate properly. When you allocate memory using **malloc** or **calloc** the starting address of the allocated memory is assigned to the pointer. The pointer contains the starting address of the allocated memory.

p	address of a memory location	--->	starting address of allocated memory

typecasting

Malloc or calloc needs to know how many bytes of memory you want to allocate. The pointer must be **type casted** to the data type requested. **Type casting** is used to **force** the allocated memory data type to the pointer variable data type. **Type casting** is very important in C. The C compiler always needs to know what kind of data type it is dealing with. You type cast by enclosing the forcing data type in round brackets

```
(type_cast_data type)

        (int*)                  /*  type cast to a int pointer data type */
```

Type casting is needed for malloc or calloc, because malloc or calloc returns a pointer with no specific data type implied (void *). Type casting tells the compiler what kind of data the allocated memory is supposed to represent. Malloc or calloc allocates memory and assigns the starting address of the newly created memory block to the pointer. You need to use #include <stdlib.h> at the top of your program to use malloc or calloc. The **sizeof** operator is used to get the size in bytes of the data type used. For calloc the 1 means 1 item.

using malloc: (allocates bytes)

```
data_type  variable_name = (data_type_cast)  malloc ( sizeof ( data_type ) );
```

```
int* p = (int*) malloc (sizeof(int)); /* declare and allocate memory using malloc */
```

using calloc: (clears memory, allocates blocks)

```
data_type  variable_name = (data_type_cast)  calloc (nItems,  sizeof ( data_type ) );
```

```
int* p = (int*) calloc (1, sizeof(int)); /* declare and allocate memory using calloc */
```

Memory is allocated and the starting address is assigned to p. Assume we have allocated an **int** data block of memory using malloc or calloc at address 3000. p is assigned the address value 3000. p at address 2000 points to the memory block located at address 3000. The value of the allocated memory allocated with malloc is unitialized and contains any value. The value of the allocated memory allocated with calloc is initialized with value 0.

address	name	contents
2000	p	3000

--->

address	value
3000	??

What is the difference between malloc or calloc? What is difference between int and int* ? What does (int*) do ?

accessing values to memory pointed to by a pointer

When we allocate memory the starting address of the memory block is assigned to the pointer. After we allocate memory we have to assign values to the allocated memory that the pointer points to. Our pointer p points to a memory location representing an **int** data type. To access the memory address pointed to by a pointer we use the * (star) operator on the pointer variable. The * (star) operator means access the value pointed to by a pointer.

To assign a value to a memory block pointed to by a pointer:

We assign a value to what the pointer is pointing to by placing a * (star) operator in front of the pointer variable.

```
*p = 5;     /* assign the value 5 to the address represented by p */
```

address	name	contents
2000	p	3000

--->

address	value
3000	5

To assign a value to the memory location pointed to by a pointer p it's a 4 step process.

steps	assigning a value *p = 5
(1)	go to pointer at address 2000
(2)	get contents of pointer p (location 3000)
(3)	go to address location 3000 specified in pointer
(4)	assign the value 5 to that address location

	address	contents
p	2000	3000
	3000	5

To read a value stored in memory:

We get a value pointed to by the pointer by using the * (star) operator placed in front of the pointer variable.

```
x = *p;    /* read value pointed to by p and assign to x */
```

address	name	contents
2000	p	3000

--->

address	value
3000	5

Assume variable x is at address location 1000.

```
/* x will now have the value 5 */
```

address	name	value
1000	x	5

What is the value of x ?

To read a value from the memory location pointed to by a pointer p it's a 5 step process.

steps	reading a value x = *p;
(1)	go to pointer at address 2000
(2)	get contents of pointer (location 3000)
(3)	go to address location (3000) specified in pointer
(4)	get the value at that address location (5)
(5)	assign the value 5 to the variable on the left hand side of the assignment statement

	address	contents
x	1000	5
p	2000	3000
	3000	5

LESSON3 EXERCISE 4

Write a program that declares a pointer called **p** and allocates an int data block. Assign to the data block the value 5. Print out the value of the data block using the pointer. Call your program L3EX4.c.

getting the address of an existing variable

Pointers point to existing variables like x and y. The address location of the variables was reserved at compile time by the compiler and linker.

Declare two variables x and y, assign 5 to x;

```
int x  = 5;      /* declare int variable x with initial value 5 */
int y;          /* declare int variable y  */
```

address	name	data value
1000	x	5
1002	y	??

To get the address of an existing variable we use the & (ampersand) operator

pointer = & variable name;

```
p = &x;      /* assign address of x to p */
```

address	name	contents
2000	p	&x (1000)

-->

address	name	value
1000	x	5

p now points to x and contains the address 1000.

What is the contents of p ? What is the value that p points to? What is the address of x ? What is the value of x ?

The contents of p is now the address of x. p points to x. If you change x to 3 then the memory contents that p points also has the value of 3. Why ? because p points to x.

```
x = 3;       /* assign 3 to x */
```

address	name	contents
2000	p	&x (1000)

-->

address	name	value
1000	x	3

What is p ? What is *p ? What is &x ? What x ?

Assign to y the value pointed to by p.

```
y = *p;          /* assign the contents of what p points to  variable y  */
```

address	name	value
1002	y	3

What is the value of y ? What is the value of x ?

If you assign a value to what p points to:

```
*p = 7;      /* assign 7 to what p points to */
```

address	name	contents
2000	p	&x (1000)

-->

address	name	value
1000	x	7

What is x, p, *p, y ?

What is the difference between the address of p and the contents of p and the value pointed to by p ? A pointer is located at an address in memory and contains an address to a memory location.

& means address of
* means the value pointed to by the pointer

LESSON3 EXERCISE 5

Write a program that declares a pointer called **p** to a int data variable called **x**. Assign to the data block the value 5 using the pointer p. Print out the value of the data variable x using the pointer p. Call your program L3Ex5.cpp.

LESSON 3 QUESTION 2

What do the following mean ?

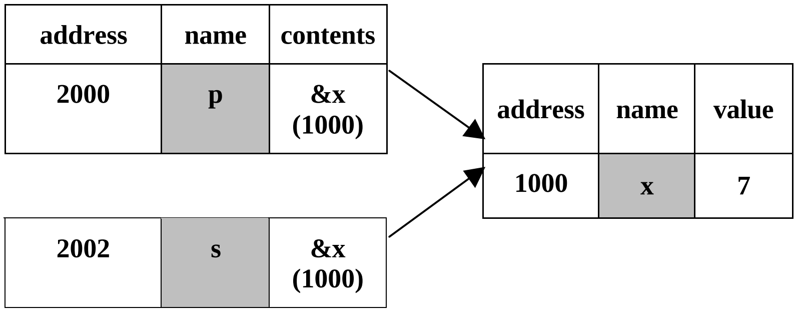
(1) the value pointed to by p	
(2) the value of what p points to	
(3) the contents of p	
(4) the address of p	
(5) assign to p the address of x	

assigning a pointer to another pointer

You can also assign a pointer to another pointer.

```
int* s = p;  /* both p and s point to the same address which happens to be x */
```

The contents of p is assigned to s. In this case both pointers will point to the same memory location. Both pointers point to x. s gets the contents of p which happens to be &x. x happens to be 7 from previous examples.



address	name	contents
2000	p	&x (1000)
2002	s	&x (1000)

address	name	value
1000	x	7

What is &p, p, *p ,&s, s , *s, &x and x ?

The following two statements will change the value of x to 5. Why ?

address	name	value
1000	x	5

```
*s = 5;                 /* change the value of x to 5 */

*p = 5;                 /* change the value of x to 5 */
```

What is x, &x, &p, p,*p, &s, s, and *s ?

LESSON3 EXERCISE 6

Write a program that declares a pointer called **p** and a int data variable called **x**. Assign to the data block the value 5 using the pointer p. Declare another pointer called **s** ad assign **p** to it. Print out the value of the data variable using the pointer **s**. Call your program L3Ex6.cpp.

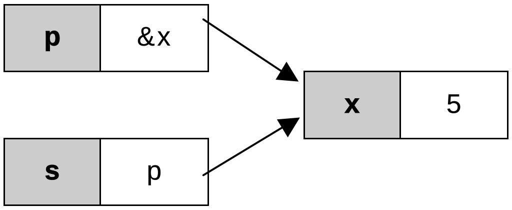
LESSON 3 QUESTION 3

The variable x is assigned the value 5 . The pointer p is assigned the address of x.

```
int x = 5;          /* declare a variable x and assign the value 5 to it */

int* p = &x;        /* declare a pointer p and assign the address of x to it */

int* s = p;         /* declare a pointer s and assign the pointer p to it */
```

Answer the following questions.

1. What is x ?		
2. What is p ?		
3. What is *p ?		
4. What is s ?		
5. What is *s ?		



LESSON 3 EXERCISE 7

Write a small program by answering the following questions that only contains a main function . Call your program L3ex7.c To print out a pointer using printf use the "%p" format specifier: printf("%p",p);

1.declare an integer variable called **x** and a integer pointer variable called **p**
2. allocate memory for your pointer variable using malloc or calloc
3. assign a value to your allocated memory by using the pointer
4. assign the value of your allocated memory to your declared variable.
5. print out all the values and addresses of your variables using printf

POINTERS TO POINTERS

Pointers may also point to other pointers. **Pointers to pointers** are declared with two stars. You can assume a star has been included for each pointer. A **pointer to pointer** points to another pointer where as a pointer just points to a memory location representing a data value. To declare a pointer to pointer:

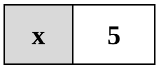
*data type** pointer variable name;*

```
int** q; /* q is a pointer variable pointing to an another pointer */
```

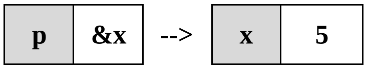
address	pointer to pointer	pointer address value
2004	**q**	**??**

When a pointer to pointer is declared it is unitialized and points to nobody and is said to contain garbage. A pointer to pointer must be assigned an address of a known pointer. The pointer is assigned the address of a variable.

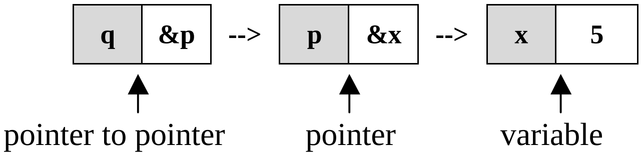
```
int x = 5; /* declare variable x and initialize to the value of 5 */
```



```
int * p = &x; /* declare a pointer p initializes to the address of x */
```



```
q = &p;  /* assign to q the address of p  (q points to p) */
```



The contents of a pointer to pointer is another pointer. The contents of a pointer is a memory location. The contents of a memory location is a data value. What is the difference between a pointer to pointer and a pointer ? The contents of **q** points to a pointer **p** that points to a memory block **x** of type integer. **q** must get the address of **p** and **p** must get the address of **x** . Each pointer contains the memory address that it is pointing to. What is the difference between a pointer to pointer a pointer and a data variable ?

pointer to pointer

address	name	contents
2004	**q**	address of pointer &p (2000)

--->

pointer

address	name	contents
2000	**p**	address of memory location &x (1000)

-->

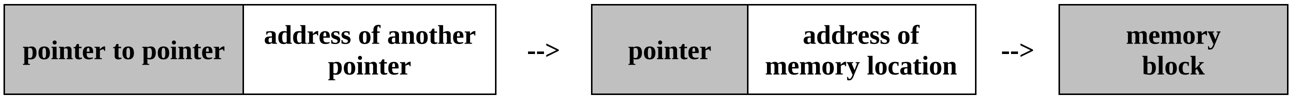
variable

address	name	value
1000	x	5

A pointer to pointer points to another pointer

Pointer to pointer analogy

An analogy to a pointer to a pointer is the mailman who reads the address of a letter goes to that address, picks up another letter reads that address and then delivers the letter to the location of the second letter. The contents of a pointer to pointer is another pointer that points to a memory block.



accessing the pointer that a pointer to pointer points to

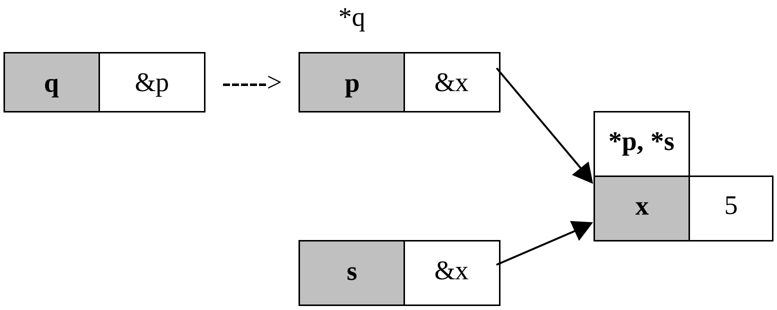
To an access what a **pointer to pointer** points to you use the * (star) operator. The contents of a pointer to pointer must be another pointer. We first declare another pointer called s

```
int * s;  /* declare an int pointer called s */
```

We next assign the contents of the pointer to pointer to s:

```
s = *q ;  /* assign tom pointer s the contents of pointer to pointer q */
```

s must point to x because *q is p or &x.



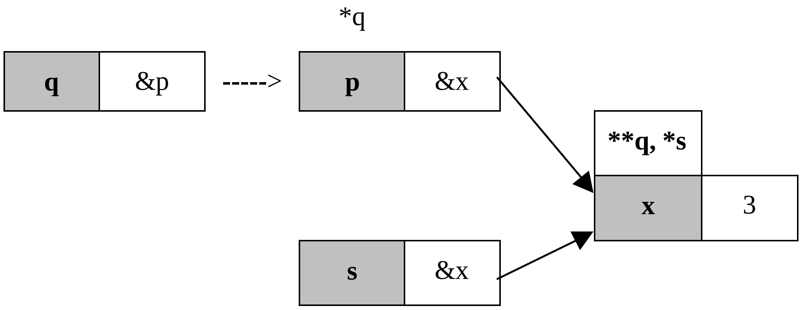
assign *q to s
(1) go to q
(2) get contents of q (&p)
(3) go to p
(4) get contents of p (&x)
(5) assign contents of p to s

What is q, *q, s, *s ?

accessing a data value to a pointer to pointer

To assign a data value to what the pointer to pointer points to we use the two ** operators

```
**q  = 3;  /* assign a data value to what the pointer to pointer  points to */
```



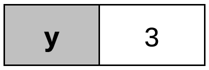
assign 3 to **q
(1) go to q
(2) get contents of q (&p)
(3) go to p
(4) get contents of p (&x)
(5) go to x
(6) assign 3 to x

What is q, *q, **q, s, *s, x ?

To retrieve the data value to what the pointer to pointer points to use: also use **

```
int y = **q; /* get data */
```

What is the value of y ?



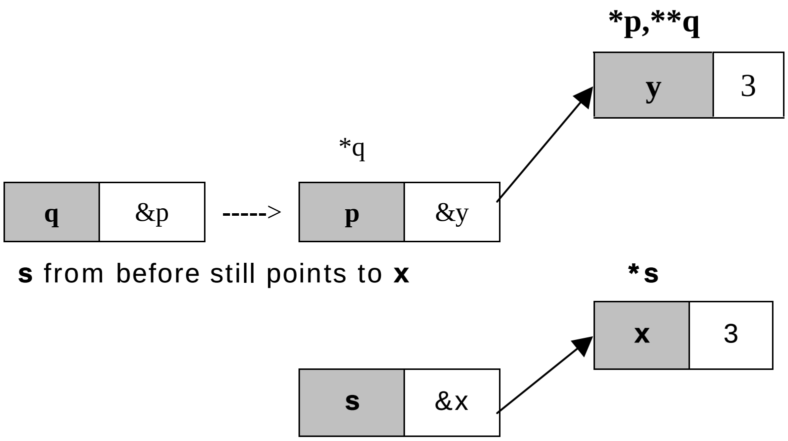
assign **q to y
(1) go to q
(2) get contents of q (&p)
(3) go to p
(4) get contents of p (&x)
(5) go to x
(6) assign 3 to y

changing what a pointer to pointer points to

We can also change what a **pointer to pointer** points to. Assign the address of y to the **pointers to pointers** pointer.

```
*q = &y; /* assign the address of y to what the pointer to pointer points to */
```

Now p points to y.



s from before still points to **x**

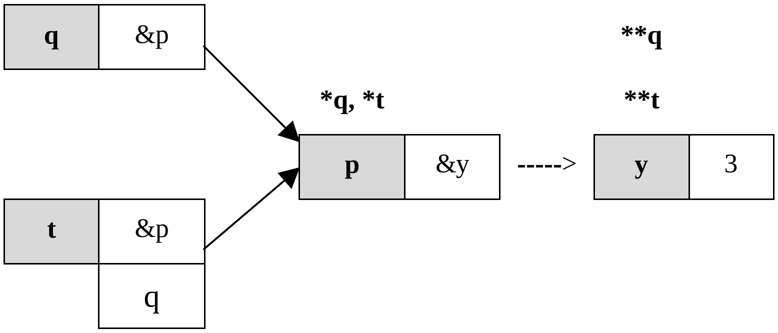
assign &y to *q
(1) go to q
(2) get contents of q (&p)
(3) assign address of y to p

What is q, *q, **q, p, *p, s, *s, x and y ?

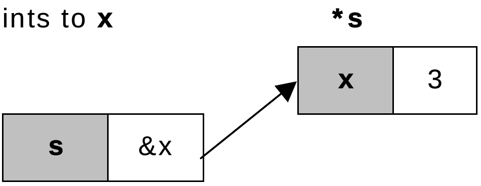
assigning pointers to pointers to other pointers to pointers

You can also assign a pointer to pointer to another pointer to pointer. We declare another pointer to pointer **t**. The value of **q** is assigned to **t**. In this case both pointers will point to the same pointer **p**.

```
int** t = q; /* both p and q point to the same pointer p */
```



s from before still points to **x**



Since q points to p and if you assign q to t then t must also point to p.

Remember a pointer to pointer contents contain a pointer's address location where a pointer contains a memory location.

What is q, *q, **q, t, *t,**t ?

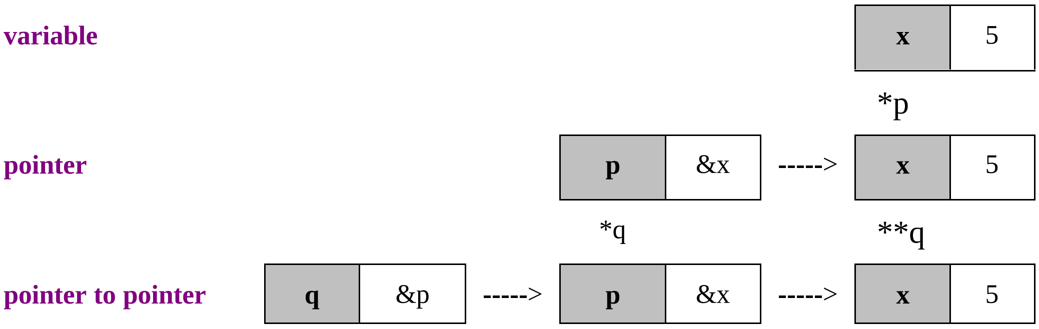
What does *q = s; do ? Draw the new pictures.

pointer summary

The following chart shows the relationships between **variables**, **pointers and pointer to pointers**:

```
int x = 5;

int * p = &x;

int**q = &p;
```

Notice as we go left to right the * (stars) increase.



Evaluate the following questions:

1. What is **q** ? **q** is the contents of q which happens to be &p
2. What does ***q** mean ? *q means what q points to which happens to be p or &x
3. What does **p** mean ? You get the contents of **p** which happens to be &x.
4. What does ***p** mean ? The data value pointed to by the pointer p which happens to be x.
5. What is the difference between **q** and ***q** ?

The contents of **q** is the address of **p (&p)** where ***q** is the value of what **q** point to which is **p** having contents **&x**.

6. What is the difference between **p** and ***p** ?

p is a pointer that has the value of the address of **x** (&x) where ***p** is the value of what the pointer **p** is pointing to which happens to be **x**.

LESSON 3 QUESTION 4

A variable x is assigned the variable 5; the address of x is assigned to the pointer p. The address of pointer p is assigned to a pointer to pointer q. Write the statements, draw the pictures and fill in the values for each variable:

x			
p			
*p			
q			
*q			
**q			

LESSON 3 EXERCISE 8

Write a small program using the following questions that just includes a main function. You should draw diagrams for each question. When it asks you " What is the value of " Write down on paper what you think it is then you may use the printf statement to print out the values to see if you are correct. The following table shows how to use printf for different data types. Call your program file L3ex8.c

int	float	string	char	pointer

int x = 5;
printf("%d",x);
	float flt=10.5;
	fprintf("%f",flt);
		char str[ ] = "hello"
		printf("%s",str);
			char c = 'A';
			printf("%c",ch);
				int* p = NULL;
				printf("%p",p);

1. Declare an integer variable **x** and assign the value **5** to it.
2. Declare a integer pointer variable called **p.**
3. Assign the address of **x** to **p** .
4. What does **p** point to? What is the value pointed to by **p** ?
5. Assign 3 to what **p** point to.
6. What is the value of **x** ?
7. Allocate a integer memory block using malloc and assign to a pointer called   **s**.
8. Assign the value of **2** to the memory location pointed to by **s**.
9. Assign p to s.
10. If **y = *s;** What is the value of **y** ? What is the value of **x** ?
11. If **x = *p;** What is the value of **x** ?

12 Declare an integer pointer to pointer **q** and assign the pointer **p** to it.
13 Use **q** to change the value of the memory location pointed to by **p** to **7**.
14 What is the value of **x** ?
15 Assign the address of **s** to **q**.
16 Assign **p** to **s**;
17 What is **q** pointing to ?

C PROGRAMMERS GUIDE LESSON 4

File:	CguideL4.doc
Date Started:	July 12,1998
Last Update:	Feb 27, 2002
Status:	proof

LESSON 4 USING POINTERS

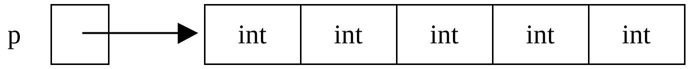
ALLOCATING MEMORY FOR ARRAYS

You allocate memory for arrays in run time when we do not know the size of the array we need before the programs runs. We need to allocate memory for arrays when we need large arrays. The **malloc** or **calloc** functions located in <stdlib.h> are used for allocating memory for arrays in run time. The difference between **malloc** and **calloc** is that **calloc** clears all the allocated memory to zero. **Calloc** also lets you allocate memory in blocks and number of bytes where in **malloc**, you can only allocate by the number of bytes. The result of both operations is the same. When allocating memory with malloc or calloc the starting address of the memory allocated for the array is assigned to a pointer.

allocating memory for 1 dimensional an array's using malloc

```
data_type  pointer_variable_name = (type_cast) malloc ( sizeof ( data_type ) * size_of_array);

int *p = (int* ) malloc (sizeof ( int ) * 5 ); /* 5 integers allocated start address assigned to p. */
```

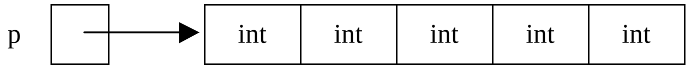
p points to the starting address of the first element in the allocated array.



allocating memory for 1 dimensional arrays using calloc

```
data_type  pointer_variable_name = (type_cast) calloc (number_of_ items , sizeof ( data_type ) );

int *p = (int* ) calloc (5,sizeof (int ) ); /* 5 integers allocated start address assigned to p. */
```

p points to the starting address of the first element in the allocated array.



In both situations we need to **type cast** to **(int*)** because both malloc and calloc return **(void*)** which means a pointer to a memory block with no implied data type.

where:

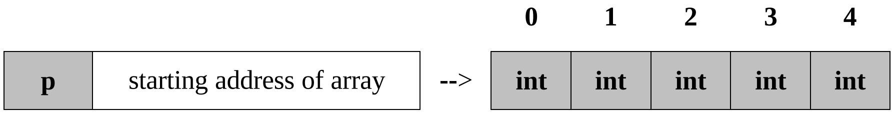
`(int *)`	type cast to an array of integers.
`sizeof (int)`	get the size of the data type int in bytes
`(sizeof ( int ) * 5 );`	multiply data type size by 5 (malloc)
`(5,sizeof (int))`	allocate 5 blocks of size data type int (calloc)

Why do we use **sizeof** ? Why do we typecast ? Why do we multiply by 5 ? Why do we need 5 items ?

sizeof is used to get the size of the data types in bytes, we typecast because we have to force the memory block to the data type we are going to use, we multiply by 5 because we need 5 integers in our array. We need 5 items because we are making an array of 5 int elements.

1 dimensional array of 5 integers is allocated

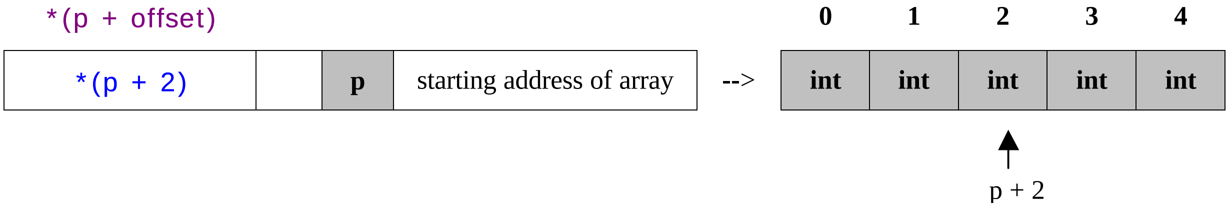
5 sequential integers are allocated and the starting address is assigned to **p**.



p points to the first location of a memory block of consecutive data types of integer, representing a one dimensional array of 1 row by 5 columns. p is of data type int *. All array index's in C start at 0 to avoid confusion and misery we will label all rows and columns indexes starting at 0. A pointer is used to point to the memory block represented a one dimensional array. Without the pointer you would not be able to locate the allocated memory.

accessing array elements pointed to by a pointer

To get the value of an array at a particular index we use an **offset** from p.



An offset is like an array index but is used with pointers to select individual elements of an array. An offset is just like adding or subtracting a address from the original base address. The address is automatically adjusted for the data type size.

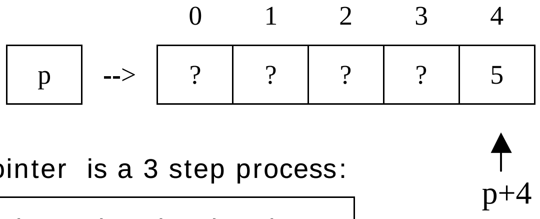
evaluated_address = base_address + offset * data_type_size

We need the * on (p+2) because we want to get the value at the evaluated address. *p means *(p+0) having an offset of 0. To point to a element at the 4th array index we use an offset of 4. To access the 4th index we would write *(p+4)

writing a value to an array element pointed to by a pointer

To write to an array element we use an offset to a pointer, a round bracket and a star *

```
*( p+4 ) = 5; /* write a  value to index 4 of the array pointed to by p */
```

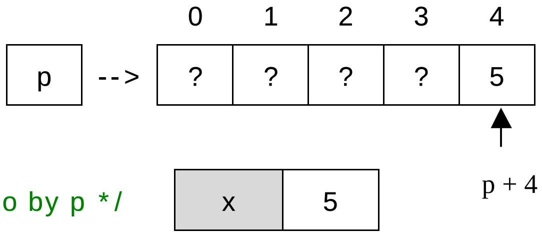


Writing a value to an array element using an offset to a pointer is a 3 step process:

(1)	go to p, get contents of p (the address location pointed to by p)
(2)	add offset to the address location pointed to by p
(3)	go to calculated address and assign value

reading a value from an array pointed to by a pointer

To read a value from the array we use an offset to a pointer, round brackets and a star *



```
x = *(p+4); /* get value of index 4 of array pointed to by p */
```

What is the value of x ?

Reading a value from an array element using an offset to a pointer is a 4 step process:

(1)	go to p, get contents of p (the address location pointed to by p)
(2)	add offset to the address location pointed to by p
(3)	go to calculated address location and get value 5
(4)	assign value 5 to variable x on left hand side of assignment statement

Notice that the brackets are around (p+4), in `x = *(p+4);` if it were not for the brackets the statement would be

```
x = *p + 4; /* get the value pointed to by p and then add 4 to the value */
```

Which means take the value at p and add 4. Which is quite different from take the element at index 4 of p and get the value. Brackets force **precedence**. Precedence states what operation gets done first. If in doubt always use brackets. What is the difference between

`x = *(p+4)` and `x = *p + 4` ?

using array indexes

We can also use an array index enclosed by square brackets to access individual elements of an array. To assign a value to an array element using an array index::

```
        p[4] = 5;   /* assign 5 to the 4th array element */
```

To read a value from an array element using an array index:

```
        x = p[4];  / * get value of index 4 of array pointed to by p */
```

p[index] = *(p+offset)

p[4] = *(p + 4)

p[index] is sort of equivalent to *(p+offset). " sort of" means not all compilers would recognize that these operations are the same. It should be obvious that *p and *(p+0) and p[0] are equivalent accessing the same memory location. What is the difference between *(p+4) , *p + 4 and p[4] ?

De-allocating memory for 1 dimensional array

Memory is de-allocated using the **free()** function found in <stdlib.h>

```
        p = (int* ) malloc (sizeof (int) );  /* allocated memory */
        free ( p );  /* deallocate memory pointed to by p*/
```

Memory must be de-allocated when it is no longer needed or else your computer may run out of memory then your program and computer will stop working.

LESSON 4 EXERCISE 1

Write a small program by answering the following questions. that just includes a main function. You should draw diagrams for each question.

1. Declare and allocate a 1 dimensional array of size 5 columns called **a**.
2. Assign the value 0 to 4 to each column index using offsets to pointers
3. print out the values using array indexes
4. deallocate any allocated memory.

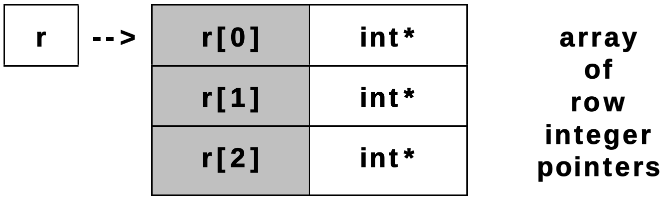
ALLOCATING MEMORY FOR 2 DIMENSIONAL ARRAYS

Pointers to pointers are used to allocate memory for 2 dimensional arrays in run time. In this case the first pointer points to an array of row pointers. Each row pointer points to a row of column integers. We first allocate an array of integer pointers (int*). Each element of the array of integer pointers will represent a row pointer. Each row pointer will point to another array representing a row of columns.

```
        data_type  pointer_name = (data_type_ cast) malloc ( sizeof (data_type) * number_of_rows);
        int** r = (int** ) malloc (sizeof ( int* ) * 3); /* r points to an array of pointers */
```

sizeof (int*) * 3	means get the size of data type int* in bytes and multiply by 3 .
int **	an array of integer pointers
int *	an array of integers

We allocate memory of data type integer pointer (int*) and type cast it to data type int**. The reason we type cast to a data type integer **pointer to pointer** (int **) is that an array of integers pointers is data type int**. **r** points to the starting address a 1 dimensional array having data type integer pointer (int*). We are pointing to an array of pointers so we need two stars **.



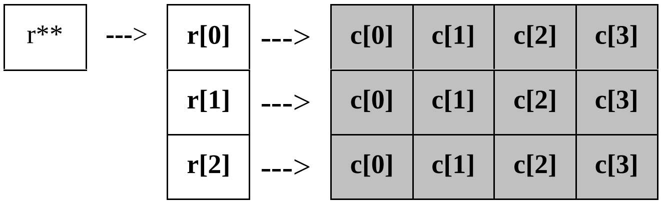
What is the difference between an array of integer and an array of integer pointers ? An array of integer pointers is an array that has array elements represented int pointers rather than int data values. What is the difference between (int**) (int*) and (int) ?

allocating memory for each row

The next thing we must do is to allocate an array of 4 columns of integers per row. Each row will point to an array of data type integer. Allocate memory for each row:

```
r[0] = ( int* ) malloc (sizeof (int) * 4) ; /* make 4 columns for row 0 (r + 0) */
r[1] = ( int* ) malloc (sizeof (int) * 4);  /* make 4 columns for row 1 (r + 1) */
r[2] = ( int* ) malloc (sizeof (int) * 4 ); /* make 4 columns for row 2 (r + 2) */
```

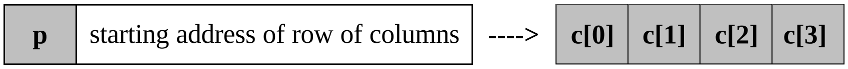
We allocate memory of data type integer and type cast it to data type int *. The reason we type cast it to data type integer pointer (int *) is that an array of integers is data type int*. Now we have a 3 * 4 array of 2 dimensions allocated by using pointers. We first allocated memory for the array of row pointers, then we allocated memory for each row of columns. Each row is an array of integers. Each index in the array is a column. Each row pointer points to an allocated row of columns. The starting address of each row of columns is stored in the array of rows pointers.



accessing array individual elements of an allocated 2-dimensional array

How do I access the columns of each array ? Easy ! Each pointer of r points to an array of columns ! This means all you have to do is to point to the address of each row index that points to the columns you want to access. You can access the elements by :

```
int* p = r[0];   / * p points to row index 0 */
```



r[0] points to an row of columns, therefore **p** now points to an row of columns. Once you get a pointer to a particular row of columns of the array then it is easy to get the value by using pointer offsets. **p** points to the starting address of a particular row of columns.

To assign a value to one of the column elements:

```
*(p+2) = 5;  /* assign 5 to row array index 0 column index 2
```

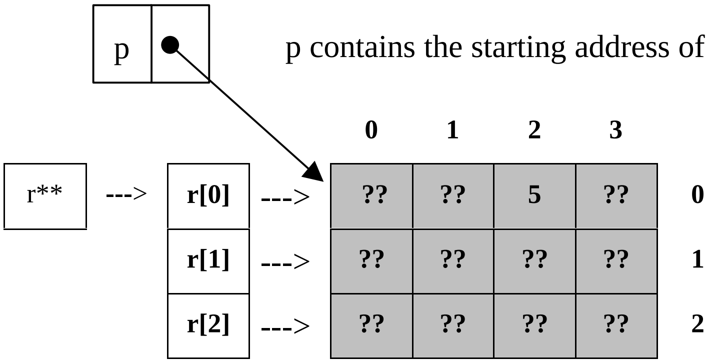
p	--->	??	??	5	??

To read a value from one of the column elements:

```
x = *(p+2);  /* get the value at row index 0 column index 2
```

x	5

What is the value of x ?



p contains the starting address of row index 0

We can combine the above two steps in one line:

```
x = **r ;  /* get the value at row index 0 col index 0   *(*(r+0)+0) */
```

Take the value of *r which happens to be row index 0 and *(*r) which happens to be col index 0 (*p) happens to. r** means *((r+0)+0). Can you figure out what the rest represent ?

```
x = ** ( r+1 );  /* value of row index  1 column index 0 */

x = * ( ( *r ) + 2 );  /* value of row index 0 column index 2 */

x = * ( * ( r+2 ) +1 );  /* value of row index 2 column index 1 */
```

The trick is that the inner star gets a pointer to a row at a particular column and the outer star gets you the value. The first offset to r gets you a particular row and the offset after the first star gets you a particular column. There must be an easier way to access array elements ? Yes there is ! An easier way is to use array indexes on r: The first index is the row and the second index is the column.

```
x = r[1][3];  /* same as x = * ( ( *r + 1 ) + 3 ) */
```

This works because the compiler sets up a 2 dimensional array as a pointer to pointer when allocated. You can say that **r = r [ ][ ]. Beware not all compilers will do this for you;

using variables for index pointers

You can also use variables for index pointers.

```
i = 2;    /* row index */
j = 3;    /* column index */
x = r [ 2 ][ j ];    /* using column index j */
x = r [ 2 ][ j ];    /* using column index j */
x = * ( p [ i ] + 1 );    /* using row index i */
x = p [ i ] [ j ] ;    /* using both row and column index */
```

De-allocating memory for 2 dimensional arrays

Memory is de-allocated using the **free()** function found in <stdlib.h>

```
p = (int* ) malloc (sizeof (int) );  /* allocated memory */
free ( p );  /* deallocate memory pointed to by p*/
```

Memory must be de-allocated when it is no longer needed or else your computer may run out of memory then your program and computer will stop working. When you no longer need the memory allocated for a 2 dimensional array also you have to delete it. You have to free each row separately. and then free the array of row pointers. remember rows are made up of columns, and row pointers point to rows not columns. You deallocate in the **opposite way** you allocate.

```
free (r[0]);  /* free memory allocated for row index 0 */
free (r[1]);  /* free memory allocated for row index 1 */
free (r[2]);  /* free memory allocated for row index 2 */
free (r );    /* free memory allocated for array of row pointers */
```

LESSON 4 EXERCISE 2

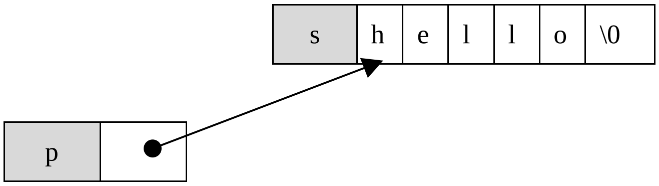
Write a small program by answering the following questions. that just includes a main function. You should draw diagrams for each question. You may use the printf statement to print out the values.

1. Declare and allocate a 3 by 3 two dimensional array by using pointers to pointers called **b**.
2. Assign to each array element the value of its row and column using offsets to pointers or array indexes.
(example row 1 column 2 would get the value 12);
3 Print out the 2 dimensional array as it would actually appear. Assigning a pointer to each of the rows of the 2 dimensional array, and use it to print out all the column values for that row.
4. Make sure you free all allocated memory.

POINTERS TO CHARACTER STRINGS

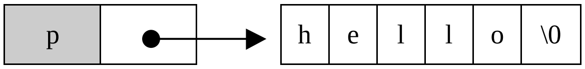
A character string is an array of characters terminated by a the end of string terminator '\0'. When you declare a character string memory is reserved in compile time. The variable name represents the memory location of the character string. You can use a pointer to point to an existing string:

```
char s[] = "hello";

char *p = s;
```



You can also point directly to a character string constant which is the combination of the above two steps. Be careful some compilers may put the character string in read only memory and then you will get an error writing to it !

```
char *p = "hello";
```



In both cases the compiler has reserved memory for the string "hello" in memory at compile time. It is easy to use pointers to character strings. You can read an individual character using an array index:

```
char c = p[2];
```

Or you can read an individual character using a pointer to the character string and a offset.

```
char c = *(p+2);
```

Using the last example, what is the value of c ?

You can also change individual characters in a character string by using pointers:

```
*p = 'j';
```

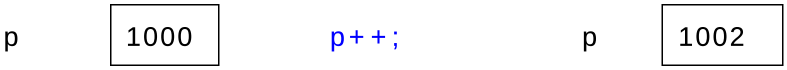
Using the last example, what is the string pointed to by p ? What is the difference between p and *p ?

Using incrementers and decrementers with pointers

Incrementors ++ and decrementer -- operators can also be used with pointers to change the contents of a pointer or values pointed to by the pointer.

p++ or p--

The contents of a pointer contains the address location it is pointing to. The pointer contents is incremented or decremented by the pointers data type size. For example if the pointer points to memory representing a int data type the contents address will increment or decrement by 2 assuming a int data type is 2 bytes in length.



Incrementers and decrementers with pointers come in handy when comparing and copying arrays and strings. When incrementing data values the incrementers increase the data value by 1 and the decrementer decrease the data value by 1.

address Incrementers increment by the data type size

address Decrementers decrement by the data type size

data Incrementers increment by 1

data Decrementers decrement by 1

operation	description	before	after
p++;	post increment the contents of p **after**		p = p+1;
++p;	pre increment the contents of p **before**	p = p+1;	
p - -;	post decrement the contents of p **after**		p = p-1;
- - p;	pre decrement the contents of p **before**	p = p-1;	

x = *(p++);
x = *p++;
	get the value of what p points to then increment the contents of p **after**	x = *p;	p = p+1;

x = *(++p) ; **x = *++p;**	increment the contents of p **before** then get value of what p points to	p=p+1;	x = *p;
x = (*p)++;	increment the value of what p points to **after**	x = *p;	*p = *p+1;
x = ++(*p);	increment the value of what p points to **before**	*p = *p+1;	x = *p;
x = *(p--) ; **x = *p--;**	take the value of what p points to, then decrement contents of p **after**	x = *p;	p = p-1;

x = *(--p);
x = *--p;
	decrement the contents of p **before** then get value of what p points to	p = p-1;	x = *p;

x = (*p)--;	decrement the value pointed to by p **after**	x = *p;	*p = *p-1;
x = --(*p)	decrement the value pointed to by p **before**	*p = *p-1	x = *p;

USING POINTERS WITH INCREMENTERS AND DECREMENTERS

***p** means take the value of what the pointer p ponts to**. *(p++)** means take the value what p points to and increment p. Brackets are used to force precedence. Precedence decides which operation gets done first. If in doubt always use bracket to force what you want to do first. The brackets around *(p++) means take the value of p then increment the contents of p. Since this is a postfix operation the contents address will be incremented after regardless if the brackets are there or not.

***(p++) is the same as *p++**

The brackets around *(++p) means increment the contents address of p then take the value pointed to by p Since this is a prefix operation the contents address will be incremented after regardless if the brackets are there or not.

***(++p) is the same as *++p**

The contents address is automatically increments the data type sizes. This means pointers to **int** data types the contents are automatically incremented by 2 pointers to character data types contents are incremented by 1 etc. Data values are always incremented or decremented by 1.

incrementing/decrementing contents of pointers

Accessing memory locations using incrementers and decrementers with pointers is a two step process.

	step 1		**step 2**	
x=*(p++)	get value pointed to by p	**x = *p;**	increments contents of p after assignment p++	**p=p+1;**
x=*(++ p)	increments contents of p before assignment	**p=p+1;**	get value pointed to by p	**x = *p;**
x=*(p - -)	get value pointed to by p	**x = *p;**	decrement contents of p after assignment	**p=p-1;**
x=*(- - p)	decrement contents of p before assignment	**p=p-1;**	get value pointed to by p	**x=*p;**

incrementing and decrementing memory values pointed to by pointers

Incrementing and decrementing memory locations pointed to by pointers is a three step process.
Memory values are incremented or decremented by 1, not by there data type size.
Do you know why ?

statement	step 1	step 2	step 3
x=(*p) ++	get value pointed to by p *p	assign value x = *p;	increment value pointed to by p *p=*p+1
x=++ (*p)	increment value pointed to by p *p=*p+1	get value pointed to by p *p	assign value x=*p
x=(*p) - -	get value pointed to by p *p	assign value x = *p;	decrement value pointed to by p *p=*p-1;
x=- - (*p)	decrement value pointed to by p *p=*p-1	get value pointed to by p *p	assign value x=*p;

Address locations increment and decrement by the data type size

Memory values only increment and decrement by 1

*** means get value of memory location pointed to by pointer**

What is the difference between x = *(p++) and x = *(++ p) ?

What is the difference between x = *(p++) and x = *p++ ?

What is the difference between x = *(p++) and x = (*p)++ ?

What is the difference between x = (*p)++ and x = ++(*p) ?

What is the difference between x = (*p)++ and x = *(p++) ?

LESSON 4 QUESTION 1

p points to the starting address of a 1 dimensional array of data type **int** that has been pre-initialized with the following values: The starting address is 1000.

p	---->	23	18	35	42	10

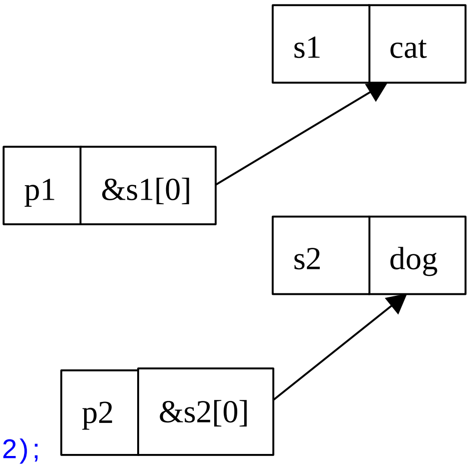
Fill in the columns for each value of x and p before the statement is executed and after the statement is executed. Assume each statement is executed as a **continuos** program. Since the array is of data type **int** the address will change by 2, because we assume are integer data type is 2 bytes long. Hint write down the array offsets on top of the array and the address on the bottom. Assume the variable x is preinitialized to zero.

statement	p before assignment	x	p after assignment
p++	1000	---------	
++p		---------	
p--		---------	
--p		----------	
x = *(p++)			
x = *p++			
x = *(++p)			
x = *++p			
x = (*p)++			
x = *(p--)			
x = *p--			
x = *(--p)			
x = *--p			
x = --(*p)			

using incrementers and decrementers with character strings

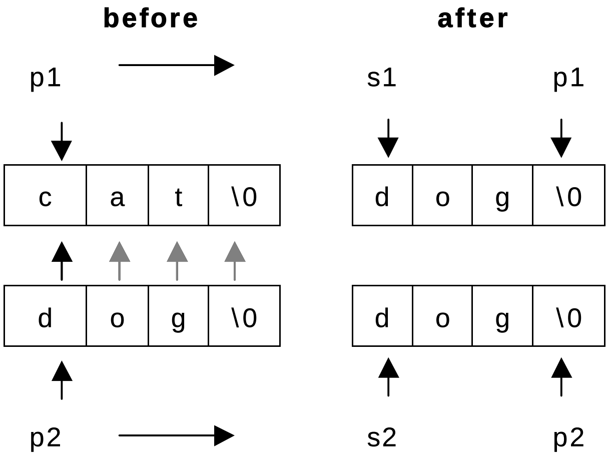
Incrementers and decrementers are great for copying and comparing character strings.

```
/* lesson 4 program 1 */
#include <stdio.h>

void main()

        {
        /* make 2 character strings */
        char  s1[ ] = "cat";
        char  s2[ ] = "dog";
        char *p1 = s1;  /* point to string s1 */
        char *p2 = s2;   /*point to string s2 */

        /* print out strings */
        printf("string s1: %s string s2: %s\n",s1,s2);

        /* copy string s2 to string s1 */
        *p1++ = *p2++;
        *p1++ = *p2++;
        *p1++ = *p2++;
        *p1 = *p2;

        /* print out strings */
        printf("string s1: %s string s2: %s\n",s1,s2);
        }
```



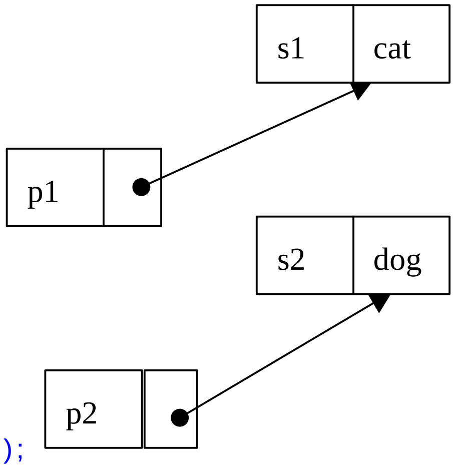
```
s1: cat   s2: dog
```

What is the value of s1 ?   dog
 What is the value of s2 ?   dog
Why do we copy 4 times rather than 3 times ?   Copy end of string terminator.



What would the following program output?

```
        /* lesson 4 program 2 */
        #include <stdio.h>

        void main()

                {
                /* make 2 character strings */
                char  s1[ ] = "cat";
                char  s2[ ] = "dog";
                char *p1 = s1;  /* point to string s1 */
                char *p2 = s2;   /*point to string s2 */

                /* print out strings */
                printf("string s1: %s string s2: %s\n",s1,s2);

                /* copy string s2 to string s1 */
                *++p1 = *++p2;
                *++p1 = *++p2;
                *++p1 = *++p2;
                *p1 = *p2;

/* print out strings */
printf("string s1: %s string s2: %s\n",s1,s2);
}
```



```
s1: cat   s2: dog
```

C PROGRAMMERS GUIDE LESSON 5

File:	CGuideL5.doc
Date Started:	July 12,1998
Last Update:	Feb 27, 2002
Status:	proof

LESSON 5 STRUCTURES

First you define a structure then you declare it to use it

STRUCTURES

Structures allow you to group many variables of **different data types** together under one common name. Each variable is known as a **member** of the structure. Structures are different from arrays, in that they can have **different** data type members. Structures may even include arrays and other structures. Structures must be **defined** before they can be used. When you define a structure you list all the data types you need. Structures are defined at the top of your program before the main function or in a separate header file.

```
struct structure_tag

        {
        data type variable name;
        data type variable name;
        data type variable name;
        data type variable name;
        };
```

```
struct record_type

{
int x; /* int */
float y; /* float */
char c; /* char */
int a[5]; /* int
array */
};
```

x					
y					
c					
a					

Why do we need structures ?

We need structures so that we can store many different data types under a common name. For example a structure may be used to store information about an employee. An employee needs a name an address a social insurance number, salary etc.

```
struct EMPLOYEE
    {
    char name[32];
    char address[32];
    char SIN[10];
    float salary;
   };
```

name	address	SIN	salary

Before you can use a structure you need to:

(1) define

(2) declare

(3) access

(1) defining your own structure data type

Structures are defined at the top of your program before the main function or in a separate header file. You simply list the variables and data types they represent. These are known as **members.**

```
struct structure_tag

        {
        data type variable name;
        data type variable name;
        data type variable name;
        data type variable name;
        };
```

```
struct record_type

{
int x; /* int */
float y; /* float */
char c; /* char */
int a[5]; /* int
array */
};
```

You can also use **typedef** to define a structure as your own user data type. In this case the struct structure_tag will be represented by your user data type: typedef *C_data_type* *user_data_type*

```
typedef  struct structure_tag

        {
        data type variable name;
        data type variable name;
        data type variable name;
        data type variable name;
        }user _data_type_name;
```

```
typedef struct record_type

        {
        int x;
        float y;
        char c;
        int a[5];
        }RECORD, *RECORD_PTR;
```

In the example two user data types are declared a structure "RECORD" and a pointer to a structure "RECORD_PTR", the comma separates the two user data type names. The RECORD is used to declare a structure user data type and RECORD_PTR is used to declare a structure user data type pointer. Typdef's allow the compiler to substitute a simpler user data type with a more complicated primitive data type. Every time the compiler sees RECORD it substitutes struct record_type Every time the compiler sees RECORD_PTR it substitutes struct record_type*. User data types are much easier and convenient to use then language definition data types. When using **typedef** the structure tag name is optional and is not needed in most cases and can be omitted.

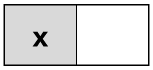
```
typedef  struct

        {
        data type variable name;
        data type variable name;
        data type variable name;
        data type variable name;
        }user _data_type_name;
```

```
typedef struct

        {
        int x;
        float y;
        char c;
        int a[5];
        }RECORD, *RECORD_PTR;
```

(2) Declaring Structures

Without the structure **definition** the compiler would not know how to make the structure, in computer memory. You have to define the structure before you can declare it. After a structure is defined, memory has to be reserved or allocated to it before it can be used. You reserve or allocate memory to a structure by declaring it. You declare a structure in the same way as you would declare a variable.

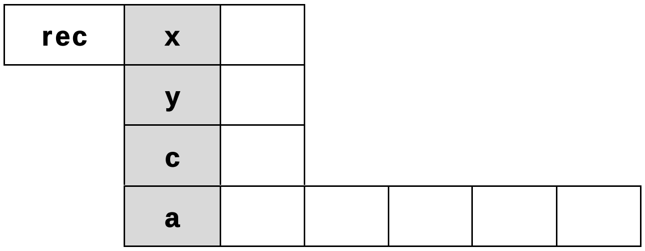
```
int x;   /* declaring a variable */
```



When you declare a structure as a variable, the compiler automatically **reserves** memory for the structure at compile time. The structure definition tells the compiler how much memory to reserve. With structures you can declare many variables at once that have different data types.

struct structure_definition structure_name;

```
struct record_type rec;   /* declares a structure rec of data type struct record_type */
```



structure **rec**
declared as
variable

Declaring structures defined with typedef lets the programmer avoid re-typing **struct**.

user_structure_data_type structure_name;

```
RECORD rec;     /* declare structure */
```

What is the difference between a variable and a structure declared as a variable?

A variable has room only for 1 value where a structure has room for many variable values each represented by a name.

(3) accessing structure members declared as a variable

Now that we know how to define, declare and allocate memory for structures, we must now know how to access the individual data members of the structure. When the structure is declared as a **variable** then the "**.**" **dot operator** is used to access individual members of the structure. The dot '**.**' operator tells you which member you want from the structure. The member variables are like a **subset** of the structure. The dot operator tells you which one you want to access.

assigning a value to a member of a structure

```
RECORD rec;  /* declare a structure variable rec */
```

structure_name.member_name = expression.

```
rec.x = 5; /* assign 5 to the x element of structure rec. */
   ↑
```

dot operator

rec	x	5				
	y					
	c					
	a					

reading a value from a member of a structure

```
int v = rec.x;      /* assign the x element of structure rec to v */
```

What is the value of v ?

example using a structure declared as a variable

Notice we define the structure at the top of our program.

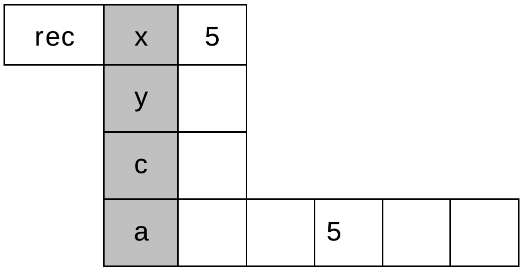
rec	x	5				
	y	10.5				
	c	'A'				
	a				5	

```
/* lesson 5 program 1 */
#include <stdio.h>

/* define structure */
typedef struct
       {
       int x;
       float y;
       char c;
       int a[5]
      }RECORD;

/* declaring structure as a variable */
void main()

         {
         RECORD rec; /* declare structure */
         rec.x = 5;    /* initialize values */
         rec.y = 10.5;
         rec.c = 'A';
         rec.a[2] = 5;
         printf("x is: %d y is: %f c is: %c a[2] is: %d \n",rec.x,rec.y,rec.c,rec.a[2]);
         }
```

```
x is: 5 y is: 10.5 c is: 'A'
```

Notice if a variable member of a structure is an array then its index must be included

```
rec.a[2]= 5; /* assign 5  to the second element of array a of  the structure rec */
```



```
int v = rec.a[2];  /* assign a element index 2 of structure rec to v */
```

What is the value of v ?

LESSON 5 EXERCISE 1

Define a structure of your favorite data type using **typedef**, for example a structure to represent an employee. In your main function declare the structure as a variable. Ask the user of your program to enter data for each member of the structure from the keyboard using **scanf.** Print out the values of the structure by using **printf.** Call your program L5ex1.c

initializing structures

If you know what values you want initialized in your structure you can include them in an initialization list when you declare your structure as a variable.

```
record rec = {5,10,'A',{0,1,2,3,4,5}};
```

Notice we needed separate curly brackets for the array located inside the structure.

Some compilers let you initialize all member values to zero:

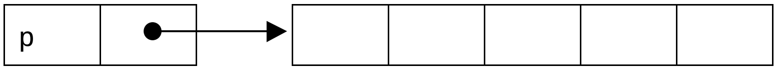
```
record rec = {0};
```

ALLOCATING MEMORY FOR A STRUCTURE

Memory for a structure can be allocated in **run time**. You can use **malloc** or **calloc** to allocate memory. The starting address of the allocated memory is assigned to a pointer. That's right you can also have pointers to structures. Allocating memory for a structure is just like allocating memory for a memory block.

data_type pointer_variable_name = (data_type_cast) malloc (sizeof (data_type));

```
int *p = (int* ) malloc (sizeof ( int ) *5);  /* allocated memory for 5 int assign to p. */
```

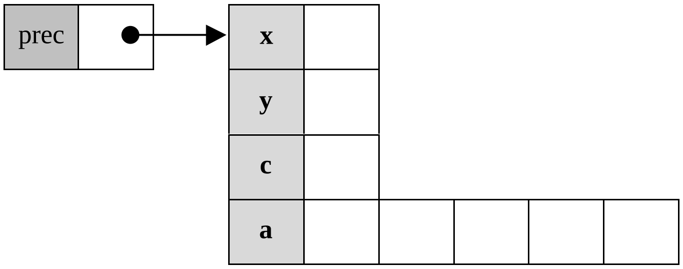


The only difference now is the data type is a structure definition. We allocate memory for a structure definition using **malloc.** Don't forget to **free** your memory after you use it.

```
struct_structure_definition* structure_name = (structure_data_type*)) malloc(sizeof(structure_definition));

        struct record_type* prec = (struct record_type* ) malloc ( sizeof ( record_type ) );
```

where the structure was previously defined as:

```
struct record_type
      {
      int x;
      float y;
      char c;
      int a[5]
     } ;
```



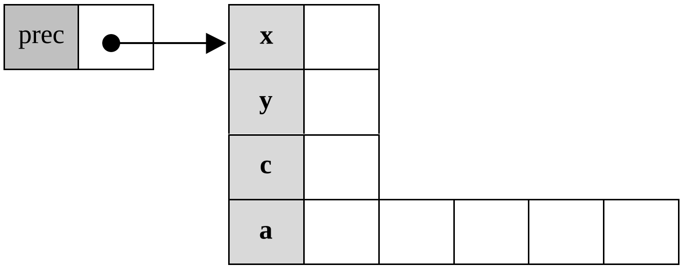
We can also allocate memory using **calloc**, in this case the allocated memory for the structure will cleared to zero. Don't forget to **free** your memory after you use it.

```
struct_structure_definition* structure_name =
         (structure_data_type*)) calloc(number of item, sizeof(structure_definition));

        struct record_type* prec = (record_type* ) calloc (1, sizeof ( record_type ) );
```

Allocating memory for structures defined with **typedef** lets the programmer avoid re-typing **struct** all the time. Again examples using malloc and calloc.

```
        user_ structure_pointer_data_type  pointer_structure_name =
                (structure_data_type*) malloc ( sizeof ( structure_data_type ) );

        RECORD_PTR prec = RECORD_PTR) malloc ( sizeof ( RECORD ) );

        structure_pointer_data_type structure_name =
                (structure_data_type*) calloc ( number of items, sizeof ( structure_data_type ) );

        RECORD_PTR prec = (RECORD_PTR) calloc (1, sizeof ( RECORD ) );
```

```
typedef struct
{
int x;
float y;
char c;
int a[5];
}RECORD, *RECORD_PTR;
```



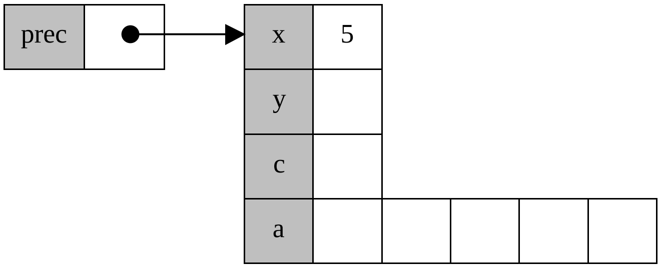
accessing structure members declared as a pointer

If the structure is declared as a **pointer** then the "->" **arrow operator** is used to access individual data type members of the structure. We first allocate memory for the structure,

```
RECORD_PTR prec = ( RECORD_PTR) malloc (sizeof ( RECORD )  );  /* allocate memory for structure */
```

pointer_structure_name->member_name = expression.

The starting address of the memory allocated for he structure is assigned to prec.



assigning a value to a structure pointed to by a pointer

```
prec->x = 5; /* write  value to the x element of structure pointed to by prec */
```

arrow operator

step 1	go to p, get address of p
step 2	calculate address to offset of member variable x
step 3	assign value 5 to that address

reading a value from a structure pointed to by a pointer

```
v = prec->x; /* read the value  of  the structure pointed to by prec  */
```

step 1	go to p, get address of p
step 2	calculate address to offset of member variable x
step 3	read value at that address
step 4	assign value to variable v

What is the value of v ?

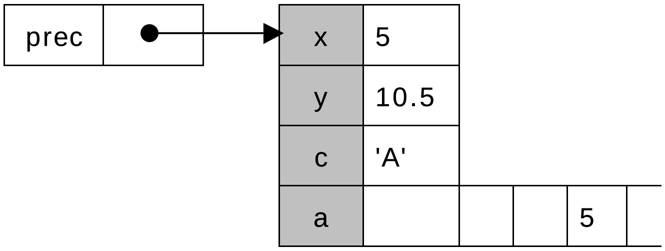
What is the difference between p.x and p->x ?

The arrow operator `prec->x` is equivalent to `(*rec).x`

`p->x =(*p).x`

example using a structure declared as a pointer

```
/* lesson 5 program 2 */
#include <stdio.h>

/* define structure */
typedef struct record_type

        {
        int x;
        float y;
        char c;
        int a[5];
        }RECORD,*RECORD_PTR;

/* declaring structure as a pointer  */
void main()

        {
        /* declare and allocate memory for structure */
        RECORD_PTR prec =(RECORD_PTR)malloc(sizeof(RECORD));
        prec->x = 5;    /* initialize values */
        prec->y = 10.5;
        prec->c = 'A';
        prec->a[2] = 5;
        /* print out values */
        printf("x is: %d y is: %f c is %c a[2] is: %d\n",prec->x,prec->y,prec->c, prec->a[5]);
        free prec; // free memory for structure */
        }
```

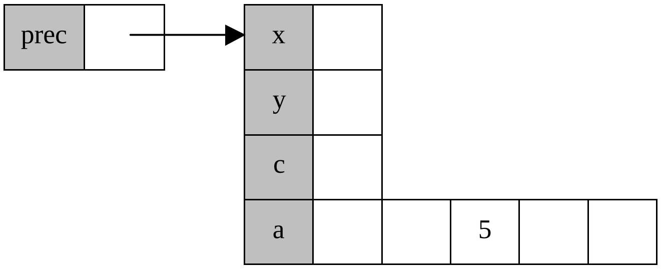


```
x is: 5 y is: 10.5 c is: 'A'
```

accessing arrays in structures pointed to by a pointer

Notice when a variable member of a structure is an array then its index must be included

```
prec->a[2] = 5; /* assign 5  to array a of  the structure pointed to by prec */
```



```
v = prec->a[2];  /* assign a element index 2 of structure rec to v */
```

What is the value of v ?
What is the difference between declaring a structure as a variable or as a pointer ?

Declaring a structure as a variable means the memory has been reserved at compile time and is identified by the structure variable name. Declaring a structure as a pointer means the memory has been allocated at run time and pointed to by the structure pointer.

LESSON 5 EXERCISE 2

Define a structure of your favorite data type using **typedef**, for example a structure to represent an employee. In your main function declare the structure as a variable. Declare a pointer variable for your structure. Initialize each member of the structure from the keyboard using **scanf.** Set the structure pointer to your structure. Print out the values of the structure using **printf** and the pointer to your structure. Call your program L5ex2.c

DECLARING ARRAYS OF STRUCTURES

From our structure definition:

```
typedef struct
{
int x;
float y;
char c;
int a[5];
}RECORD, *RECORD_PTR;
```

You can declare arrays of structures as a variable, the memory for the arrays of structures are **reserved** at compile time.

structure_data_type variable_name[column size];

RECORD arec[4];

arec[0]	arec[1]	arec[2]	arec[3]
x	x	x	x
y	y	y	y
c	c	c	c
a[5]	a[5]	a[5]	a[5]

You can also declare a pointer to allocated array of structures using **malloc** or **calloc**. The memory for the array of structures are **allocated** in **run time**. Don't forget to **free** your memory after you use it.

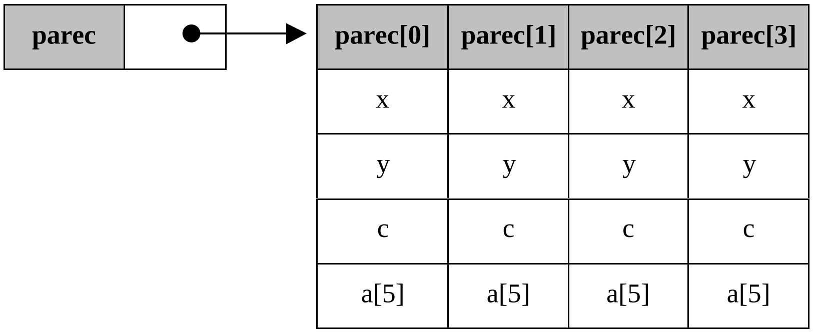
structure_pointer data_type structure_name=
(structure_data_type) malloc (sizeof (structure_definition) * size_of_columns);*

RECORD_PTR parec = (RECORD_PTR) malloc (sizeof (RECORD) * 4);

structure_pointer data_type structure_name=
(structure_data_type) calloc (number of items, sizeof (structure_definition) * size_of_columns);*

RECORD_PTR parec = (RECORD_PTR) calloc (4, sizeof (RECORD));

parec contains the starting address of the first record. When you allocate memory be sure to take the **sizeof (RECORD)** not the **sizeof(RECORD_PTR)** times by how many records you want. If you take the **sizeof (RECORD_PTR)** will not get all the required memory you need.



parec is a pointer to an array of structures

accessing values in arrays of structures

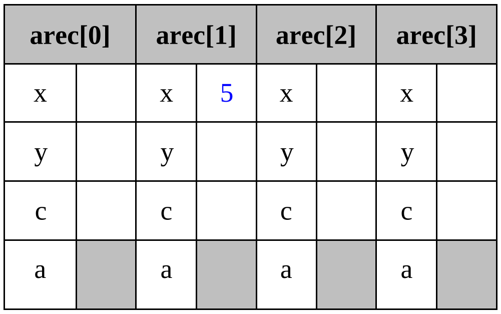
When you have an array of structures, accessing member variables of each structure is identical regardless if the arrays of structures were declared as a variable or as a pointer to allocated memory. The dot "**.**" operator is used to access individual members of each structure located in the array of structures. The arrays do not contain pointers to structure but actual structures therefore the dot operator has to be used and p[ ] means *p.

accessing a structure in an reserved array of structures

The dot operator is used to access each member of the structure located in the array:

```
RECORD arec[4];

arec[1].x = 5; /* assign 5 to the x member */
v = arec[1].x; /* get value of x member  */
```

What is the value of v ?

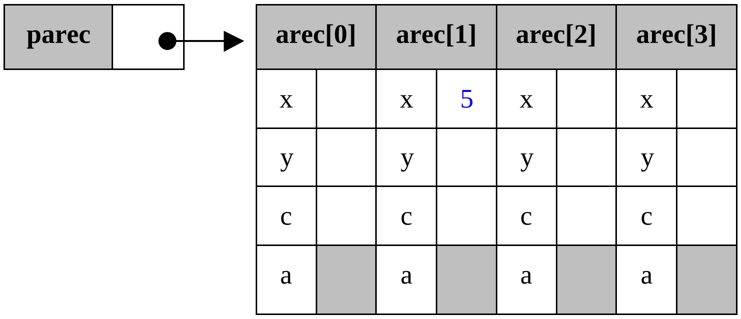


accessing a structure in an allocated array of structures

The dot operator is also used to access each member of the structure located in the array:

```
 RECORD_PTR parec = ( RECORD_PTR) malloc
(sizeof ( RECORD ) * 4 );
parec[1].x = 5; /* assign 5 to x  */
v = parec[1].x; /* get value of x  */
```

What is the value of v ?

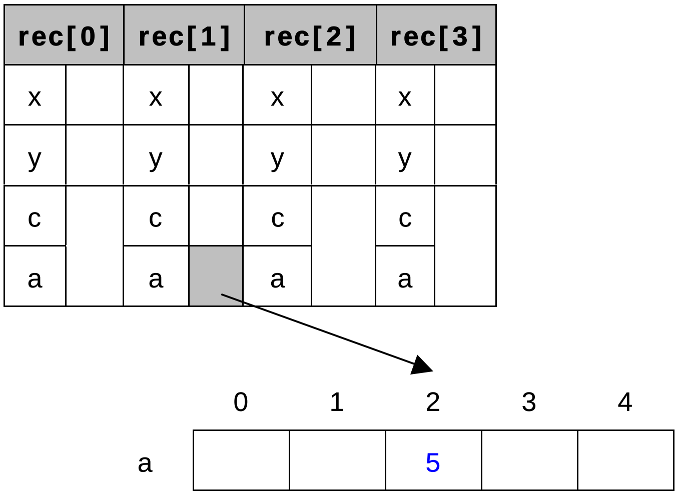


We use the dot '**.**' operator to access the members because we are accessing memory locations that contain data. Not memory locations that contain addresses. That is why we do not use the arrow '->' operator. Why do we not use the arrow operator when accessing member variables of array of structures ?

accessing array members of structures in an reserved array structure

If the structure is an array and member variable is an array then an index is needed for the structure array and for the array member of the structure.

```
arec[1].a[2] = 5; /* assign  5 to element index 2 of structure rec index 1 */
```



rec[0]		rec[1]		rec[2]		rec[3]	
x		x		x		x	
y		y		y		y	
c		c		c		c	
a		a	▨	a		a	

	0	1	2	3	4
a			5		

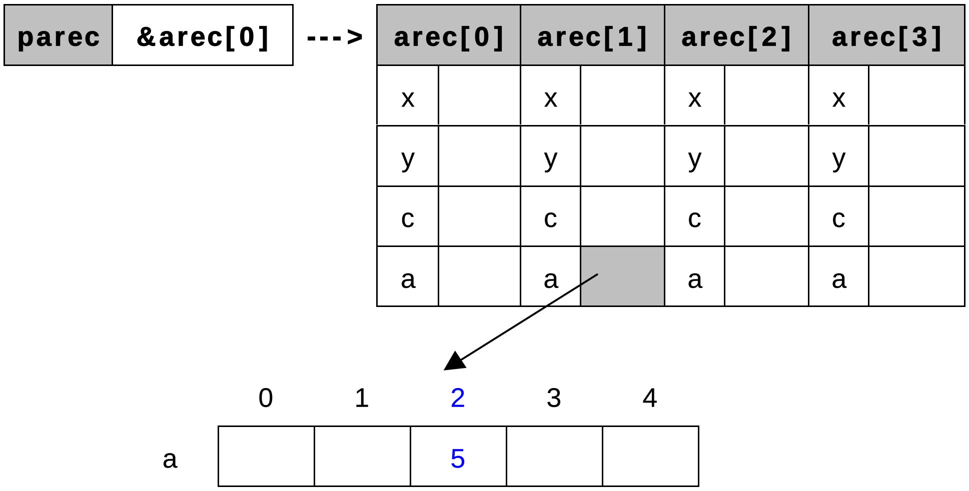
```
v = arec[1].a[2]; /* assign a element index 2 of structure rec index 1 to v */
```

What is the value of v ?

accessing array members in an allocated array structures

If the structure is an allocated array and member variable is an array then an index is needed for the structure array and for the array member of the structure.

```
parec[1].a[2] = 5; /* assign 5 to  element index 2 of structure array index 1 to v */
```



parec	&arec[0]

--->

arec[0]		arec[1]		arec[2]		arec[3]	
x		x		x		x	
y		y		y		y	
c		c		c		c	
a		a	▨	a		a	

	0	1	2	3	4
a			5		

```
v = parec[1].a[2]; /* assign a element index 2 of structure array index 1 to v */
```

What is the value of v ?

using variables to access array structures and member of arrays of structures

You may use variable names as indexes to access array structures;

i = 0;	int v = arec[i].x;	int v = parec[i]->x;	int v = arec[i].x[j];
j = 1;	int v = rec.x[j];	int v = prec->x[j];	int v = parec[i]->x[j];

The following program demonstrates declaring and accessing arrays structures as a variable and as a pointer.

```
/* Lesson 5 program 3 */
#include <stdio.h>

/* define a structure called record */
typedef struct

        {
        int x;
        float y;
        char c;
        int a[5];
        }RECORD,*RECORD_PTR;

/* program to demonstrate using arrays of structures */
void main()

        {
        RECORD arec[4];  /* declare array of structures as a variable  */
        arec[1].x = 5;  /* assign 5 to member x of array index 1 */
        printf("x = %d\n",  arec[1].x ); /* print out value to screen */
        /* allocate array of records */
        RECORD_PTR parec = (RECORD_PTR)calloc(4,sizeof(RECORD));
        parec[1].x = 5; /* assign 5 to member x  */
        printf("x = %d\n",  parec[1].x );     /*print out value to screen */
        free parec; // free memory for array */
        }
```

LESSON 5 EXERCISE 3

Define a structure with 3 members of int, float and char data type. Define your structure using **typedef.** Write a program with only a main function that declares an array of 4 of your structures as a **variable.** Pick an index of the array of structures and initialize the structure elements for that array index with your favorite values. Declare a pointer variable to this structure. Print out the values of the structure using **printf** and your structure pointer. Call your program L5ex3.c

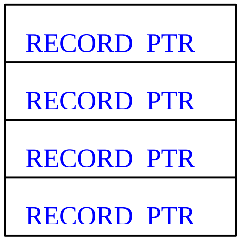
ARRAY OF POINTER STRUCTURES

Sometimes you need an array of pointers to structures. Do you know what an array of pointers to structure is ? An array of pointers to structures is simply an array of pointers. Each pointer will contain the address to a memory block representing a structure. You can declare an array of record pointers as a variable in compile time or as a pointer to an allocated array in run time.

When working with an array of pointers it's always a 2-step process.

(1) reserve or allocate memory for an array of pointers

(2) reserve or allocate memory for the structure.



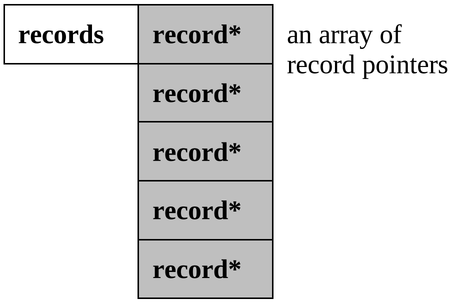
(1) reserve memory for an array of pointers

To declare an array of structures pointers as a variable you use a data type pointer.

```
RECORD_PTR  records[5];  /* array of pointers  declared as a variable */
```

↑

structute pointer data type

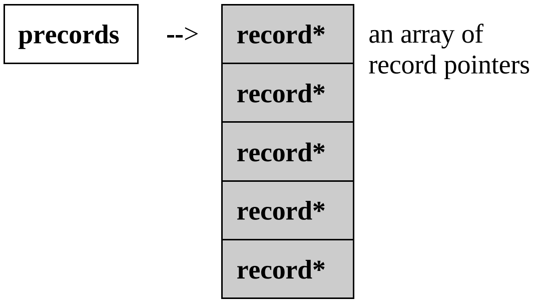


(1) allocate memory for an array of pointers

To allocate an array of structure pointers you need a data type with 2 stars. A RECORD_PTR is already 1 star so we just need 1 additional star.

```
RECORD_PTR* precords = (RECORD_PTR*)calloc(5,sizeof( RECORD_PTR*)); /* array of pointers allocated */
```

One star is for the array and the other star for the contents of the array "a pointer to a structure". Its your old friend a pointer to a pointer. Notice **sizeof uses** RECORD_PTR* to allocate memory for a structure pointer.

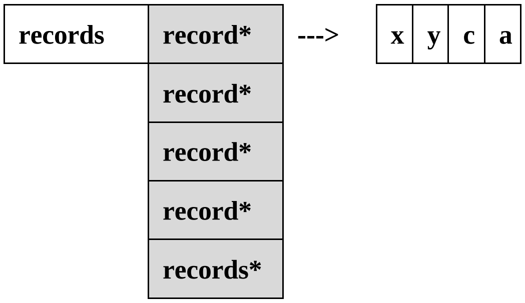


What is the difference between arrays of structures and an array of structure pointers ?

(2) allocating memory for a structure pointed to by a pointer in a reserved array

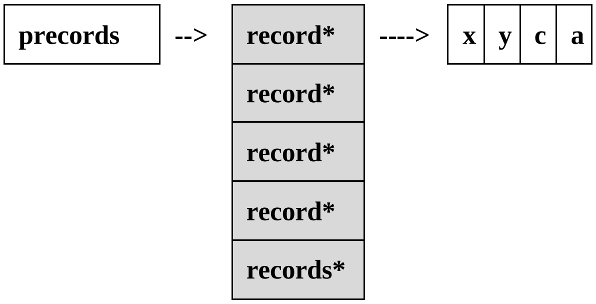
Before you can use your array of structure pointers you need to allocate memory for a structure for each pointer. You allocate a structure to one of the elements of your array of structure pointers using malloc or calloc.

```
records[0] = (RECORD_PTR)malloc(sizeof(RECORD)); /* allocate memory */
```



(2) allocating memory for a structure pointed to by a pointer in a allocated array

```
precords[0] = (RECORD_PTR)malloc(sizeof(RECORD)); /* allocate memory */
```



Accessing structures in an array of structure pointers

To access an individual structure in an array structure pointers you first need to get a pointer to the structure. You can use the reserved or allocated array of structure pointers.

```
RECORD_PTR prec = records[0]; // get pointer to structure from array of pointers declared as a variable

RECORD_PTR prec = precords[0]; // get pointer to structure from array of pointers declared as a pointer
```

In either situation you get a pointer to a record. Once you get a pointer to the structure you want to access its individual members. We use the **arrow operator** on the member element because we have a pointer to the structure.

```
int x = prec->x; // access x member of structure pointed to by prec
```

If you are really a hot shot programmer than you can access the structure members directly.

```
int x = records[0]->x; // access x member of structure pointed to array of record pointers
int x = precords[0]->x; // access x member of structure pointed to array of record pointers
```

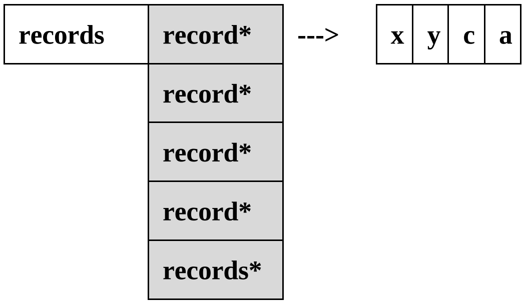
Notice we still use the arrow operator on the member element because the array value is a pointer.

When accessing members of the structure pointed to by the array of structure pointers why do we use the arrow operator and not the dot operator ?

Can you still remember what the -> arrow means ?

p->x = (*p).x

The following program demonstrates using an array of structure pointers declared as a variable and declared as a pointer.



```
// Lesson 5 program 4
#include <stdio.h>
#include <stdlib.h>

// define a structure called record
typedef struct

        {
        int x;
        float y;
        char c;
        int a[5];
        }RECORD, *RECORD_PTR;

// program to demonstrate using an array of structure pointers
void main()

        {
        RECORD_PTR records[5]; /* reserve an array of 10 structure pointers */
        RECORD_PTR * precords;  /* declare a pointer to a structure */
        records[0] = (RECORD_PTR)malloc(sizeof(RECORD));  /* allocate a structure */
        records[0]->x=5;  /*assign a value to one of the members of the structure */
        printf("%d\n",records[0]->x );  /* print out value to the screen */
        /* declare an array of record pointers as a pointer */
        precords = (RECORD_PTR*)calloc(5,sizeof( RECORD_PTR));
        precords[0] = (RECORD_PTR)malloc(sizeof(RECORD));  /* allocate a structure */
         precords[0]->x = 5;  /* assign a value to one of the members of the structure */
        printf("%d\n",records[0]->x );  /* print out value to the screen */
        free precords[0]; /* free memory  for structure */
        free precords; /* free memory  for array */
        }
```

LESSON 5 EXERCISE 4

Make an array of structure pointers. Allocate memory for a structure and assign to one of the structure pointer in your array. Declare a pointer to your allocated structure using the array of structure pointers. Initialize your allocated structure with values from the keyboard using the pointer. Print out the values of your allocated structure to the screen using the pointer. Call your program L5ex4.c

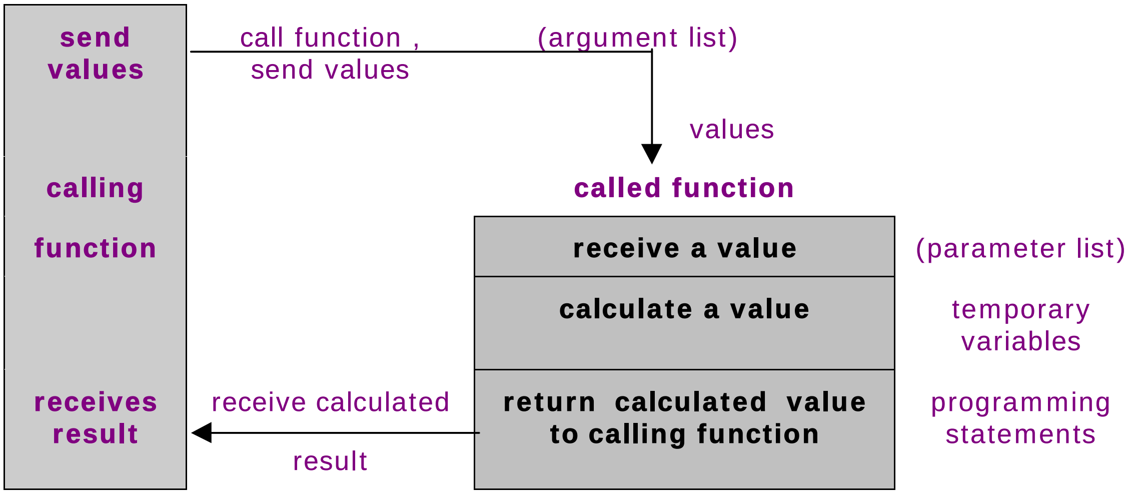
C PROGRAMMERS GUIDE LESSON 6

File:	CGuideL6.doc
Date Started:	July 12,1998
Last Update:	Feb 29, 2002
Status:	proof

C LESSON 6 FUNCTIONS

FUNCTIONS

A program is made up of **functions**. Functions are made up of **temporary variables** and **programming statements.** Variables in a function are temporary meaning once the function finishes executing the variables value disappears. Variables declared in a function are only known to that function. Programming statements in a function execute sequentially one after another. For one function to use the services of another function it **calls** a function by its **name** and may or may not pass values to that function. The **called** functions **receives values** from the calling function through its **parameters** contain in a **parameter list.** The **calling** function sends values to a function by way of **arguments.** Arguments may be variables, constants or returned values from other functions. Functions **calculate values** and may or may not **return** a calculated result.



functions:
(1) receive values
(2) calculate results
(3) return values

Why do we need functions ?

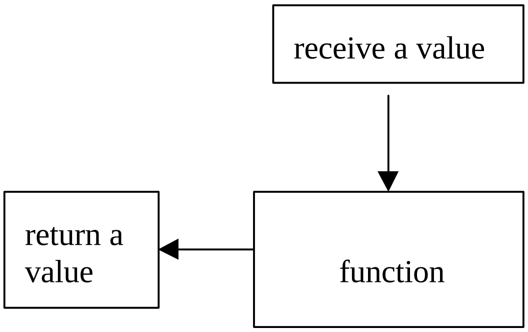
Functions are needed to **avoid repeating** the same program statements, over and over again every time we need to calculate the same thing with different values. For example, the square function is used over and over again to calculate the square of a number for different values. Without functions there would be lots of repetitious typing. Functions contain programming statements that are called to do the same calculation but are passed different values.

function analogy

A function is like a chemical tank. It receives fresh liquid (the arguments) from the chemical plant (the calling function). The liquid is received from an opening at the top of the tank (the parameter list). The liquid is mixed in the tank (calculates a value) and then returns a mixture (the return value) to the chemical plant (the calling function). The variables are the chemicals being mixed and the function programming statements is the chemical equation to do the chemical mixing in the right sequence.

purpose of functions

A function receives a value through its parameter list, do a calculation and returns a result. A function may or not receive a value. a function may or may not return a value. When a function does not receive or return a value the data type is called **void.**



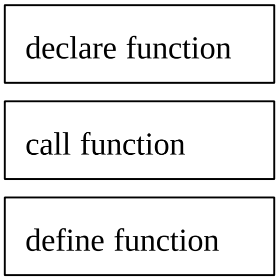
using functions

Before a function can be used it needs to be

(1) declared

(2) called

(3) defined

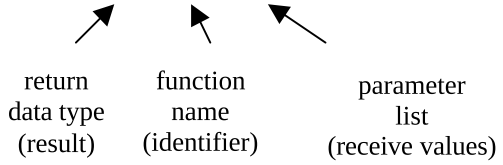


(1) declaring a function

A function is declared by stating what **data type it returns,** the **name of the function** and what **values** it receives. If the function does not return a value then the keyword **void** is used to state no return data type. Declaring a function is also known as a **prototype.** We declare functions as prototypes before we define and use them. To declare a function prototype:

return_data_type function_name (parameter_list);

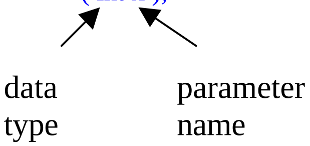
```
int square ( int x ); /* declare  function square that receives a value and returns a  value */
```



parameters receive values

A parameter list contains parameters that receive values. A parameter has a parameter type and a name. Parameters just work like temporary variables that have been **preinitialized** with a value..

```
( parameter_list );

( parameter_type  paramater_name );

( int x );
```



The parameter type tells what kind of data the parameter is receiving. The parameter name is used to store the received parameter value. The parameter list is just a list of parameters in a function prototype.

```
/* declare  function square that receives a value and returns a  value */
int square ( int x );
```

function declaration examples

function declaration	receive data type	return data type
int square (int x);	int	int
put_value (int x);	int	void
int get_value ( );	void	int

void means not to receive or return a value

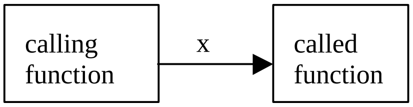
calling a function

To use a function you call it by its name and pass values to that function as arguments through an **argument list**.

```
function_name ( argument_list);

put_value (x ); /* call the print function and pass  argument x to function */
```

An argument list is just values separated by commas. arguments are just values: constants, variables or return valises from other functions.

```
(aregumet1, argument2, argumentn)

(5,x,get_value())
```



The **calling** function calls another function the **called** function. The function receives the argument values through its parameters. The arguments get mapped into the parameters of the function in a sequential order. An analogy is the arguments are the delivery trucks and the parameters are the receiving dock. The function is the factory that will produce products from the received raw material. The person who orders the goods is the calling function that calls the factory (the called function) to produce the goods.

calling a function that does not return a value

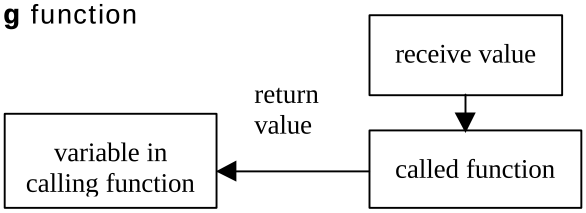
The following function **put_value** is called, it receives a value x but does not return a value. This function may write a message and value to the computer screen.

```
put_value (x ); /* call the print function and pass  argument x to function */
```

calling a function that returns a value

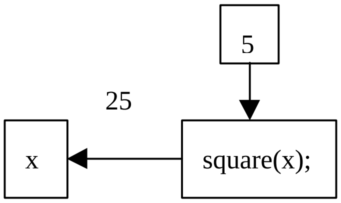
When a function returns a value an **assignment statement** is used to **assign the return** value from the **called** function to the **calling** function. The assignment statement **assigns** the return value of the **called** function to a variable in the **calling** function

```
variable_name = function_name ( argument_list );

x = square ( 5 ); /* call square function with argument 5 */
```



The assignment statement is like the highway, that that **directs** the delivery truck to which customer to deliver the finished products to. The return statement is the shipping department that sends out the good. Obviously, the variable on the left of the assignment statement is the customer who gets the finished goods.

```
x = square ( 5 ); /* call square function with argument 5 */
```

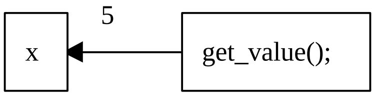


When we call the square function it calculates the square of 5 and returns the result of the calculation to the calling function. The return value of the called function is assigned to the variable **x** of the calling function. The calling function may be the main function or another function.

calling a function that does not receive a value but returns a value

A function may also return value but not receive any values. An example would be a function that calculates values internally or a function that get values from the keyboard.

```
variable_name = function_name ( );

x = get_value ( ); /* call square get_value with no arguments */
```



DEFINING A FUNCTION

After a function is declared it must be defined with variables and programming statements. The purpose of declaring a function lets you call it before you define it. You need to define your function so that the compiler can make code. When you define a function all variables must be declared at the top of each function, before the function statements. An example of a function definition is the square function.

```
return_type function_ name ( paramater_list )

        {
        variable declarations
        statements
        optional return statement
        }
```

```
/* return square of number */
int square (int x )

{
int y  = x * x; /* calculate square of a number */
return y; /* return square of number */
}
```

The square function receives an integer value through parameter x . A temporary variable y is declared used to hold the calculated results of the square of x. The value held in y is returned when the function terminates. We can also define another functions **put_value**. The put_value function receives a value and print it to the computer screen with a message

```
/* print out value */
void put_value (int x )
{
printf("the value is: %d",x);
}
```

using prototypes

Functions maybe declared as **a definition** or as a **prototype**. When they are declared as a definition, then the body of the function is specified with the declaration. When a function is declared as a prototype then only the return type, function name and parameter list is declared. The prototype function declaration is terminated by a semi-colon. The **semi-colon** distinguishes function declarations from a function definition.

```
int square (int x );  // prototype for square function
```

Prototypes allow the body of the function to be defined some where else in the program. Prototypes are declared at the top of a program before all function definitions. All of the function declarations we use will be prototypes. It makes good sense to use prototypes. If you do not use prototypes then you must declare your functions in the order that they are called. This is a nuisance and difficult to do. Function prototypes are listed at the top of a program before the functions are used and defined or listed in header files. What is the difference between a function declaration and a function definition ?

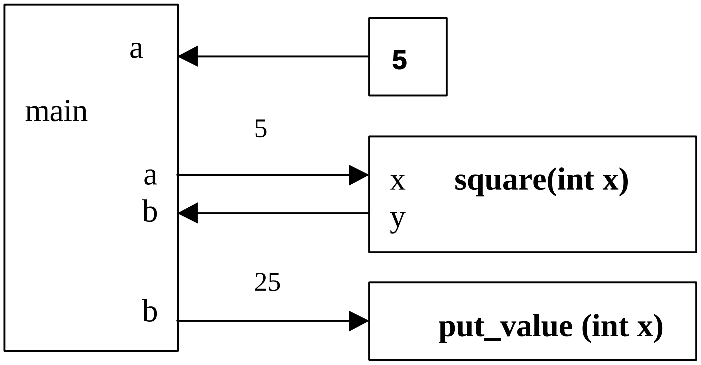
C program format for functions and prototypes	**function prototypes**
	main function
	function definitions

The purpose of a prototype is to let you use functions before you define them.

example program using functions

The following is an example of program that calls the square function to calculate the square of a number. The main function is the **calling** function and the square function is the **called** function. The first function to executed in a C program is the **main** function. All functions are declared as a prototype before the main function. You will notice that arguments and parameters can have the same name or different names. It does not matter they can have the same name or different names. They are not related in any way. You call a function by its name and pass arguments to it. The function receives the arguments through its parameters. The parameters work like temporary variables in a function and are initialized by the passed arguments. The function does a calculation and returns the calculated value to the calling function.

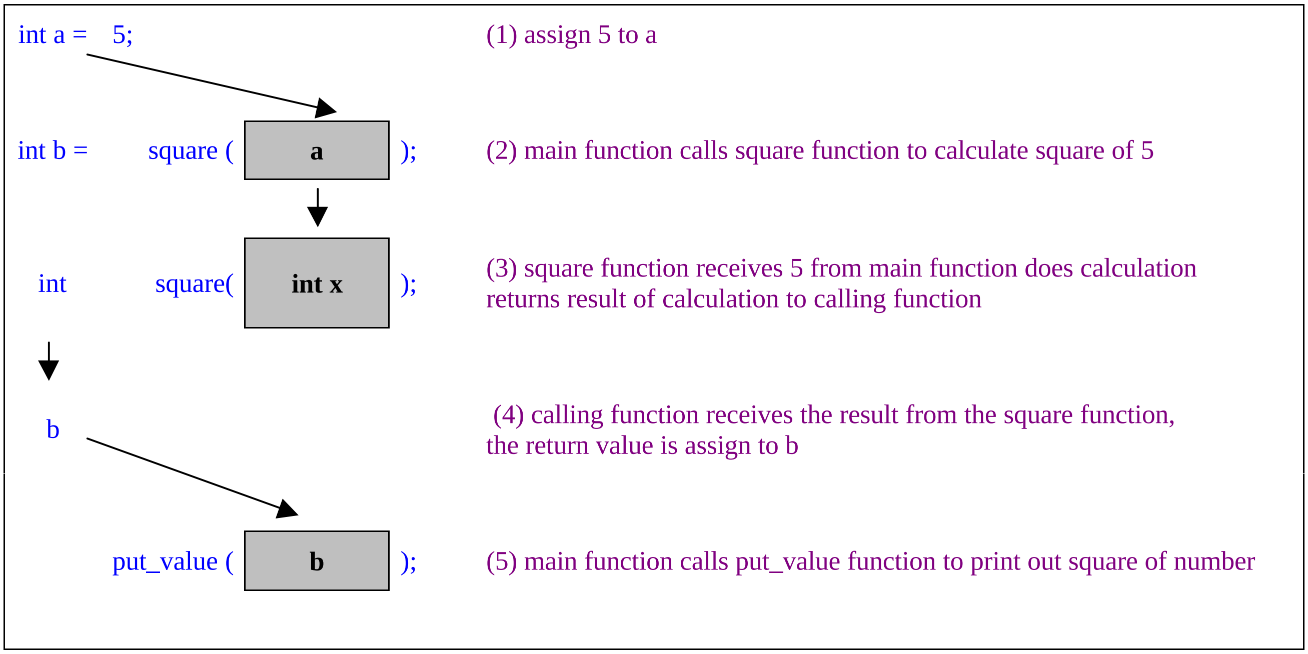
```
/* program to calculate the square of a number */
/* lesson 6 program 1 */

#include <stdio.h>

/* function prototypes */.
int square (int x);
void put_value (int x);

/* main function */
void main()

        {
        int a=5;
        int b = square( a); /* call square to calculate the square of 5 */
        put_value(b); /* print out the square of the number */
        }

 /* function definitions */

/* calculate square of number */
int square (int x )

        {
        int y;
        y  = x * x; /* calculate square of a number */
        return y;  /* return square of number */
        }

/* print out result */
void put_value ( int x )

        {
        /* use printf function to print number to screen */
        printf("the square is: %d\n", x); /* print value to screen */
        }
```



program output:

```
the value is: 25
```

program flow



int a = 5;

(1) assign 5 to a

int b = square (a);

(2) main function calls square function to calculate square of 5

int square(int x);

(3) square function receives 5 from main function does calculation returns result of calculation to calling function

b

(4) calling function receives the result from the square function, the return value is assign to b

put_value (b);

(5) main function calls put_value function to print out square of number

LESSON 6 EXERCISE 1

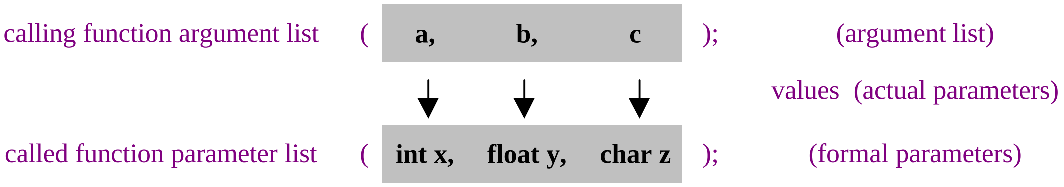
Write an int getValue() function for the above example. It prompts the user to enter a number and from the keyboard. The function returns the entered number. Call you're finished program L6ex1.c

Passing values to functions

Values are passed to the function from the **calling** function to the **called** function by an **argument** list. The function receives the values through its **parameter list**. The parameters hold the values received from the calling function. There is a sequential one to one correspondence between the calling functions **argument list** and the called functions **parameter list**. The names maybe the same or different but the data types must be identical. The parameters x,y,z are known as the **formal** parameters and the received values are called the **actual** parameters. Arguments a,b,c must be of data type int, float and char respectively.

```
int a; float b; char c;
```

You can change the value of the formal parameter in a function, but you cannot change the value of the actual parameter.



calling function argument list (a, b, c); (argument list)

values (actual parameters)

called function parameter list (int x, float y, char z); (formal parameters)

It's a one to sequential mapping between arguments and parameters. Parameters are like temporary variables that are initializes by the arguments.

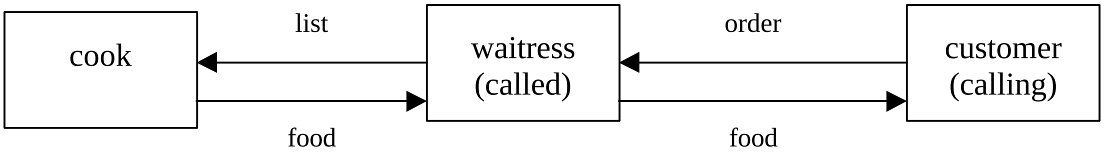
The calling functions supplies values called arguments to the called function. The called function receives the arguments from the calling function. The actual parameters are the received values. The values of the arguments are assigned to the parameters.

```
x = a;  y = b;  z = c;
```

The formal parameters are the declared parameters and contain the values of the arguments. The formal parameters act just like a variable in the function. Variables in a function are only temporary and do not retain their values between function calls.

parameter passing analogy

Think that the calling function is a customer in a restaurant that calls the waitress (the called function) to order some food. The order given to the waitress is the argument list. The waitress receives the order through a parameter list as spoken words (the actual parameters). The waitress writes down the order in her order book. The written words become the formal parameters. The order is given to the cook to cook the food. The food and the order are the variables of the waitress function. The statements of the function are the action taken by the waitress to get the order and give it to the cook. When the food is ready the waitress gives the food (the return value) to the customer (the calling function)



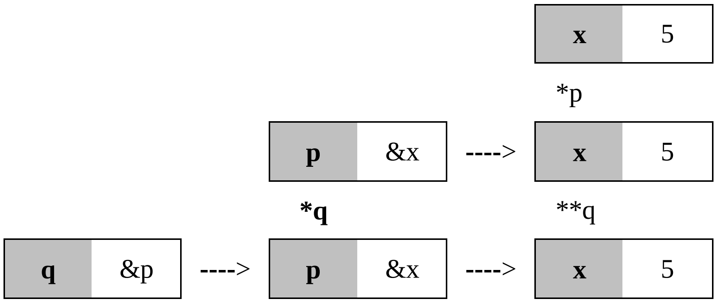
review of variables, pointers and pointers to pointers

A **variable** is just a memory location holding a value. A **pointer** is a variable whose value is an address that points to a memory location storing a value. A star is used to access the value of the memory location pointed to by the pointer. A **pointer to pointer** is one pointer pointing to another pointer. The second pointer points to a memory location storing a value. The inner star is used to access the value of the memory location pointed to by the pointer. The outer star is used to access the contents of the second pointer, which is the address of the memory location pointed to by the second pointer. A star proceeding a pointer (*p) means to access the value of what the pointer points to. A star * in front of a pointer to pointer (*q) gets the contents of what the pointer is pointing to. Two stars proceeding a pointer to pointer (**q) means get the value of what the second pointer is pointing to.

variable: `int x = 5;`

pointer: `int*p = &x;`

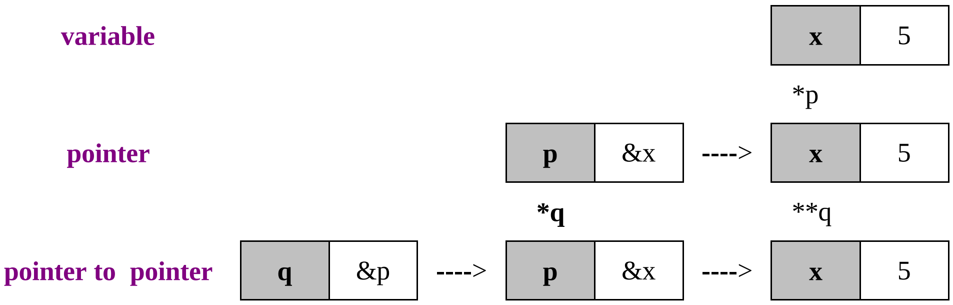
pointer to pointer: `int **q = &p;`



EXERCISE 6 QUESTION 1

A small program containing only a main program declares some variables and assigns values to them.

```
/* lesson 6 question 1 */
void main()

        {
        /* declare variables */
        int x ;     /* memory location */
        int *p ;   /* pointer */
        int **q ; /* pointer to pointer */

        /* assign values to variables */
        x = 5;     /* the value 5 is assigned to x */
        p = &x;  /* p get address of x */
        q = &p;  /* q gets address of p */
        }
```



You go right to get data values

→

You go left to get address values

←

_	*	* *
&	_	*
	&	_
		&

From the above program answer the following questions:

1 What is the value of x ?

2. What is the contents of p?

3. What is the value of *p ?

4. What is the contents of q ?

5. What is the value of *q ?

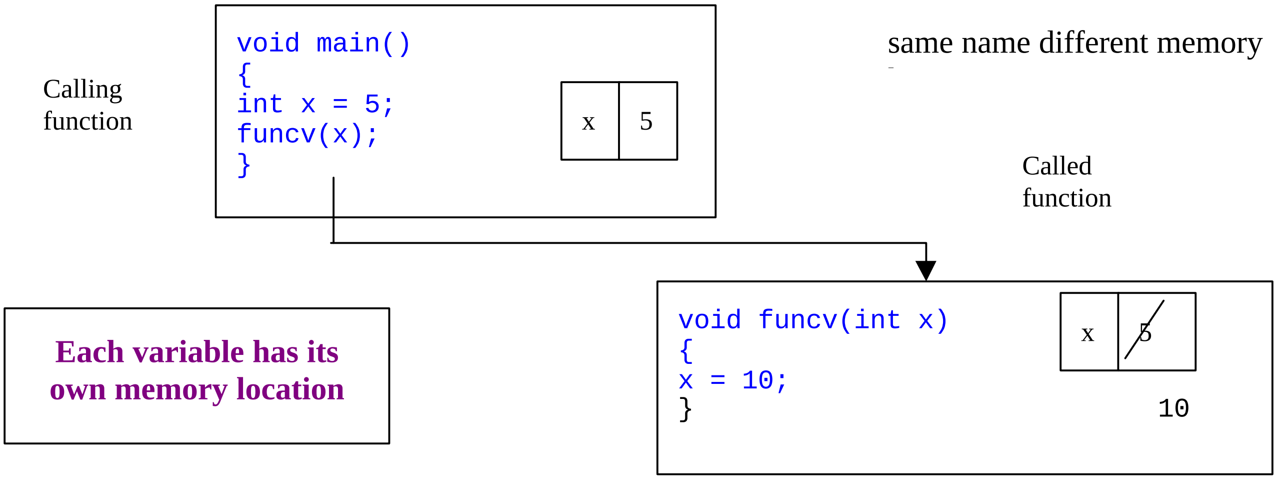
6. What is the value of **q ?

PASS BY VALUE, POINTER AND POINTER TO POINTER

The functions parameter list, lists all the **input, output** or **input/output** values needed by the function. **Input** means the function is receiving an input value through the parameter list. **Output** means the function is allowed to send a value back to the calling function through its parameter list. **Input/output** means the function receives and sends a value back through the parameter list. Variables may be passed by **value**, by **pointer** or by **pointer to pointer**.

pass by value

When a variable is passed to a function **by value** the parameter in the called function holds a **copy** of the passed value. The parameter is a temporary memory location to hold the value passed to it. This means you cannot change the value of the variable in the calling function from the called function.



```
void main()
{
int x = 5;
funcv(x);
}
```

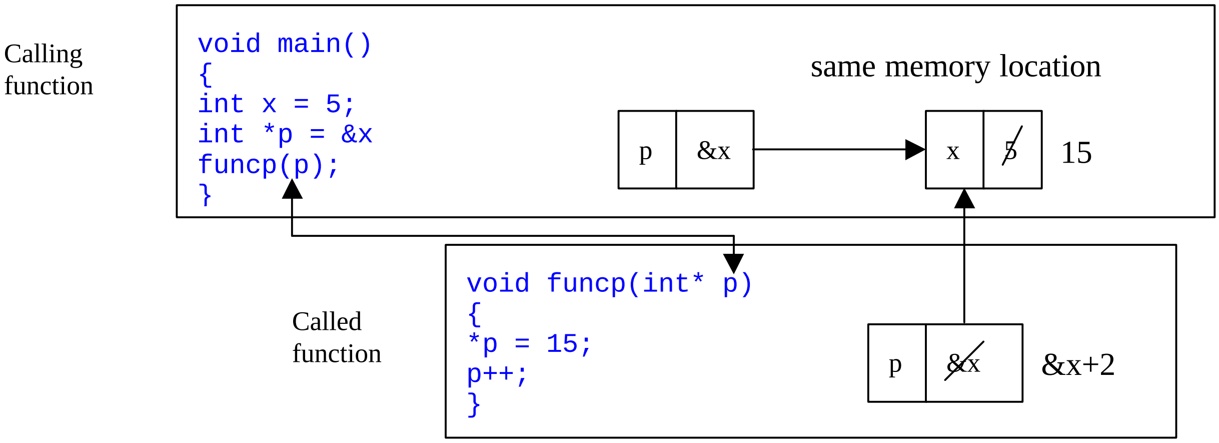
```
void funcv(int x)
{
x = 10;
}
```

Each variable has its own memory location

Each function has its own x. Each x is separate and cannot effect each other.

pass by pointer

When a variable is passed to a function by pointer, the parameter in the called function stores the contents of the pointer passed to it. This means you can change what the pointer is pointing to in the calling function. You cannot change the contents of the pointer in the calling function from the called function. Passing by pointer is known as **indirect addressing** because a memory location holds the address to be accessed.



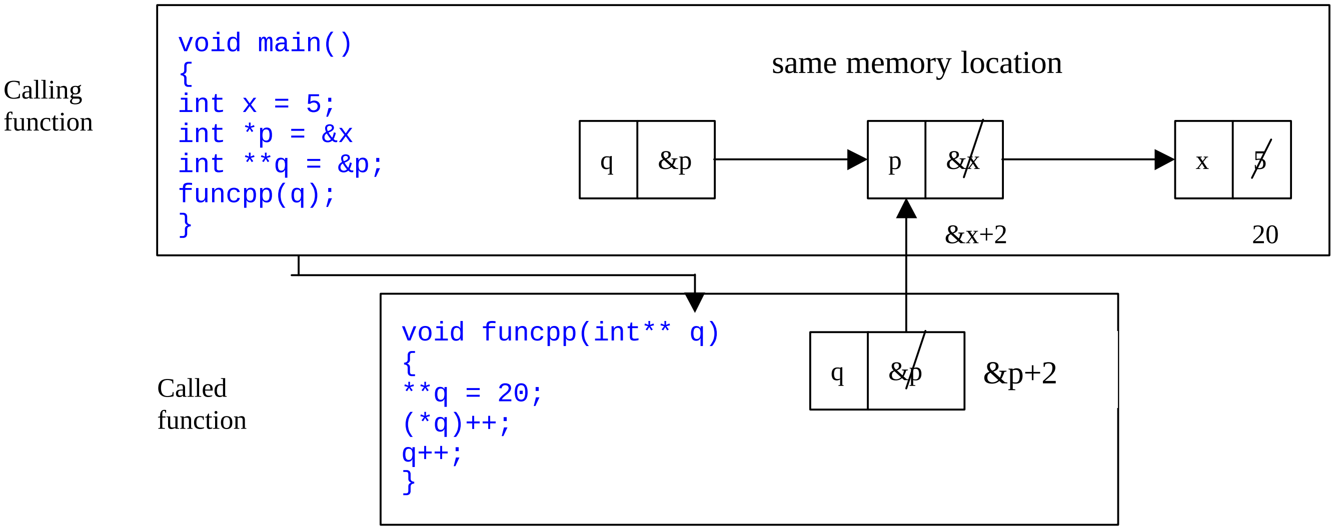
```
void main()
{
int x = 5;
int *p = &x
funcp(p);
}
```

```
void funcp(int* p)
{
*p = 15;
p++;
}
```

Each function has its own pointer p. Each pointer p is separate and cannot effect each other. Since both pointer point to the same x they both can change the value of x. You cannot change what the calling function p is pointed to inside the called function.

pass by pointer to pointer

When a variable is passed by pointer to pointer the contents of the pointer to pointer is stored in the parameter. You can change what the pointer to pointer is pointing to in the called function. You cannot change the contents of the pointer to pointer in the calling function from the called function. Passing by pointer is also known as **indirect addressing.**



```
void main()
{
int x = 5;
int *p = &x
int **q = &p;
funcpp(q);
}
```

```
void funcpp(int** q)
{
**q = 20;
(*q)++;
q++;
}
```

Each pointer to pointer points to the same pointer. In the called function we can change what's the pointer is pointing to but we cannot changer what the pointer to pointer is pointing to in the called function.

Here is a summary of what can be changed for different passed types.:

pass by	**what can be changed or not changed**
value	cannot change value outside function
pointer	can only change the value of what the pointer is pointing to outside function
pointer to pointer	can only change the value of what the pointer to pointer is pointing to and the value of what the pointers to pointer pointer is pointing to

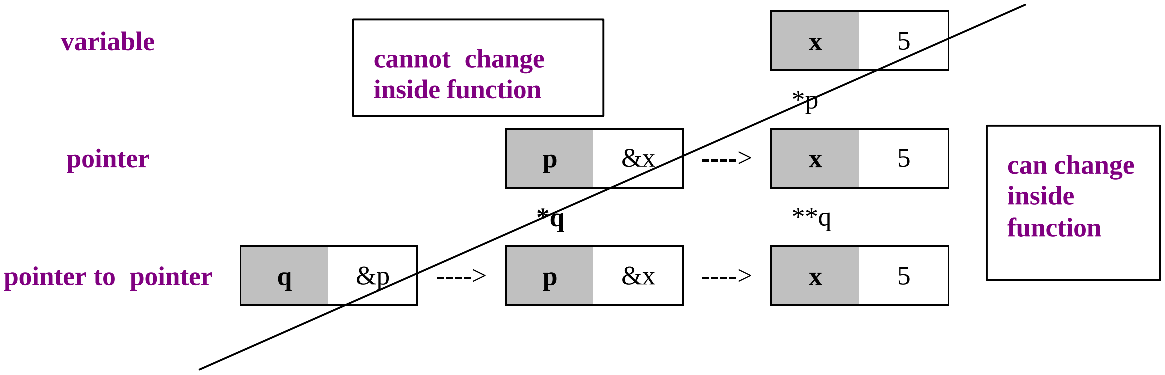
LESSON 6 EXERCISE 2

Write a program that demonstrates for pass values by value , pointer and by pointer to pointer for variable x, pointer p and pointer to pointer q. Print out the values of x, p and q before the functions are called and after the functions are called. Trace through the function line by line examining the values. Call your finished program L6ex2.c.

review pass by value, pointer and pointer to pointer

In a function you can only change what a pointer points to not the contents of a pointer. The contents of a pointer is an address location. Variables **passed by value** is simply passing the value represented by the variable name to the function. The function receives a **copy** of the value so that the original variable value cannot be changed. The variable in the calling function is different then the variable in the called function even if they have the same name.

Variables **passed by pointer** pass the contents of the pointer, which is address of the variable that the pointer is pointing to. The function receives a **copy** of the pointers contents so that the original pointers contents value cannot be changed. It is this address that lets the function change the value of the variable that the pointer is pointing to in the calling function. You cannot change the contents of the pointer only the value of what the pointer is pointing to Variables **passed by pointer to pointer** pass the contents of the pointer to pointer which is address of the second pointer. The function receives a **copy** of the address to the second pointer. The original pointers address value cannot be changed. The function can only change the contents of the second pointer is pointing to and what the second pointer is pointing to. Pass by value is input only, where pass by address is output or input/output. Since a function can only return one value, the parameter list is used a lot to pass variables as output or input/output. The following chart shows the relationships between variables, pointers and pointer to pointers. Variables on the left of the line cannot be changed inside a function.


variable
cannot change inside function
x 5
*p
pointer
p &x ----> x 5
can change inside function
*q
**q
pointer to pointer
q &p ----> p &x ----> x 5

passing values to functions

The following chart shows how to pass parameters as a variable int x; pointer int *p; and pointer to pointer
int **q;.

pass by	function call	function prototype	comment

value	fv(x);
	fv(*p);
	fv(**q);
		void fv(int i);	pass value of variable
			x, *p, **q

pointer	fp(&x);
	fp(p);
	fp(*q);
		void fp(int *i);	pass address of variable
			&x, p, * q

pointer to pointer:	fpp(&p);
	fpp(q);
		void fpp(int **i);	pass address of second pointer
			&p, q

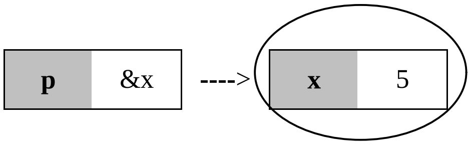
passing pointers to pointers to functions

Pointers to pointers are used if you need to change the contents of a pointer from the calling function. You must use a pointer to a pointer. If you only use a pointer then you can only change the value of the memory location that the pointer is pointing to, not the contents of the pointer itself. Only by using a pointer to a pointer you can change the what a pointer is pointing to. An easy understanding of passing pointers to pointers is this. If you want to have a variable as an output then you need a pointer *p. If you want to assign a value to what a pointer points to then you need to use **p which is a pointer to a pointer.

```
void main()

        {
        int x = 5; /* assign value 5 to x */
        int *p = &x; /* assign address of x to p */
        fp(&x); /* pass address of x */
        fp(p);  /* pass contents of p */
        }
```

```
void fp(int *p)

        {
        *p=3; /* assign 3 to x */
        }
```

pointer

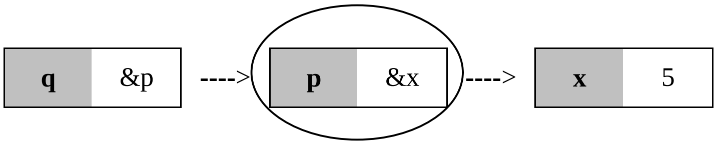


Therefore to have a pointer as an output then you need to declare the pointer in the function parameter list also with a extra star "*". Presto a pointer to a pointer "**q" To assign a new address to the contents of p you also need a star "*" on q "*q=*q+2;" The statement "*q=*q+2;" increments the contents of what q points to which happens to be p in the main function.

```
void main()

        {
        int x = 5; /* assign value 5 to x */
        int *p = &x; /* assign address of x to p */
        int **q =  &p; /* assign address of p to q */

        fpp (&p); /* pass address of p */
        fpp (q); /* pass contents of q */
        }
```

```
void fpp(int **q)

        {

        /* increment q's contents by 2 */
        *q ++;     /* p = p + 2 */
        }
```

pointer to pointer



assigning parameters to variables inside functions

Variables inside function must be declared as the function parameters. A star preceding the parameters declares as a pointer access the contents of what the pointer points to . What is the difference of the contents of a pointer and the value of what a pointer points to ?

```
void main()

        {
        int x = 5; /* assign value 5 to x */
        int *p = &x; /* assign address of x to p */
        int **q =  &p; /* assign address of p to q */

        fall (x,p,q);  /* call function */
        fall(*p,&x,&p);
        fall(**q,*q,q). ;
        }
```

```
void fall(int  x, int *p,int **q)

        {
        int y = x; /* value of x */
        int z = *p; /* contents of  what p points to
        */
        int *s = p; /* contents of p */
        int * t = * q; /* contents of what q points
        to */
        int **u = q; ./* contents of q */
        }
```

LESSON 6 QUESTION 2

In above function fall() state the contents of each variable for each argument passed to it.

variable	fall (x,p,q);	fall(*p,&x,&p);	fall(**q,*q, q).;
y			
z			
s			
t			
u			

returning values from functions

Values may be returned to the calling function by value or by address. The return statement inside a function is used to return a value from a function when it terminates.

```
return expression;

return x;   /* function returns a value */
```

The return statement without an expression can be used to terminate at any time, if the function does not return a value **void.**

```
return;

return; /* function does not return a value. terminate function execution  */
```

A return statement can be used anywhere in a function to force an immediate return. When the return statement is executed the function stops executing and execution control returns to the calling function. The return statement is not necessary at the end of a function. A function returns to the calling function when the last statement in the function is executed.

Functions that do not return a value are preceded by the keyword **void**, meaning return nothing.

```
void f ( int* list, int length ); /* function does not return any value */
```

Functions that return a pointer with no implied data type use void*

```
*void*  fp ( int* list, int length ); /* function returns a pointer with no implied data type */
```

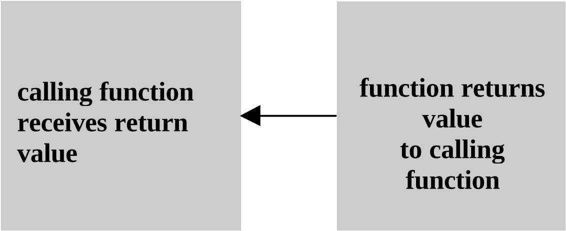
Do you know what is the difference between void and *void ?

void can also be used in the parameter list if the function accepts no arguments. This is seldom used, if so then only by programmers who are paranoid. These statements are equivalent:

```
int f ( void );  /* accept no parameters */


int f ( ); /* accept no parameters */
```

The following table demonstrates examples using return statements in functions.

return; / force return from function */*	void fv(int x);
return x; /* return the value of x */	int fvv(int x);
return p; /* return a pointer */	int* fpv(int x);
return q; /* return a pointer to pointer */	int** fppv(int x);



calling function receives return value

function returns value to calling function

examples of passing parameters and return values from a function

A function may use a combination of all passing methods to receive and return values as shown by the following examples using int x, int *p and int **q;

prototype	comment	example using
int fvv (int i);	/* pass by value, return by value */	int y = fvv(x);
int fpv (int *i);	/* pass by pointer , return by value */	int z = fpv(p);
int *fpp (int *i);	/* pass by pointer, return by pointer */	int *s = fpp(p);
int *fppp (int **i);	/* pass by pointer to pointer , return by pointer */	int *t = fppp(q);
int **fpppp (int **i);	/* pass by pointer to pointer , return by pointer to pointer */	int **r = fpppp(q);

LESSON 6 QUESTION 3

A small program is presented next to demonstrate pass by value, pass by pointer and pass by pointer to pointer and return by value. You may want to keep a chart to record the values for each function call.

function call	y	x	p	*p	q	*q	**q
start							
funca							
funcb							
funcb							
funcc							
funcd							
funcd							

```
/* this program demonstrates pass by value, pointer and pointer to pointer lesson 6 question 3 *
#include <stdio.h>

/* function prototypes */
int funca ( int i  );       /* pass by value return by value */
int funcb ( int* i );       /* pass by pointer return by value */
int funcc ( int** i );     /* pass by pointer to pointer  return by value */
int funcd ( int** i );     /* pass by pointer to pointer  return by value */

 /* main function */
void main()

          {
          int x = 5;        /* declare integer variable x and assign 5 to it */
          int y;              /* declare integer variable y */
          int* p = &x;    /* p contains the address of x */
          int **q = &p; /* q contains address  of p */

          y = funca ( x );  /* pass by value */

          /* what is the value of y ? */ /* 6 */
          /* what is the value of x ? */ /* 5 */
          /* what is the value of p ? */ /* &x */
          /* what is the value of *p ? */ /* 5 */
```

```
        y = funcb ( &x );  /* pass by pointer */

        /* what is the value of y ? */ /* 6 */
        /* what is the value of x ? */ /* 6 */
        /* what is the value of p ? */ /* &x */
        /* what is the value of *p ? */ /* 6 */

        y = funcb ( p );  /* pass by pointer */

        /* what is the value of y ? */ /* 7 */
        /* what is the value of x ? */ /* 7 */
        /* what is the value of p ? */ /* &x */
        /* what is the value of *p ? */ /* 7 */

        y = funcc ( &p );  /* pass by pointer to pointer */

        /* what is the value of y ? */ /* 8 */
        /* what is the value of x ? */ /* 8 */
        /* what is the value of p ? */ /* &x */
        /* what is the value of *p ? */ /* 8 */

        y = funcd ( &p );  /* pass by* pointer to pointer */

        /* what is the value of f y ? */ /* ??? */
        /* what is the value of x ? */ /* 8 */
        /* what is the value of p ? */ /* &x+2) */
        /* what is the value of *p ? */ /* ???*/

        y = funcd (-- q);  /* pass by pointer to pointer */

        /* what is the value of f y ? */
        /* what is the value of x ? */
        /* what is the value of q ? */
        /* what is the value of *q ? */
        /* what is the value of **q ? */
        } /* end of main */

/* functions */

/* pass by value return by value */
int funca ( int i )

        {              /* what is the value of i ? */
        i++;           /* what is the value of i ? */
        return i;     /* what does the function return ? */
        }

/* pass by pointer return by value */
int funcb ( int *i )

        {               /* what is the value of i, *i ? */
        ( *i )++;      /* what is the value of  i, *i ? */
        return *i;    /* what does the function return ?*/
        }
```

```
/* pass by pointer to pointer return by value */
int funcc ( int **i )

        {                 /* what is the value of i.*i,**i ? */
        int* q = *i;   /* what is the value of q,*q ? */
        ( *q )++;      /* what is the value of q,*q ? */
        return *q;     /* what does the function return ? */
        }

/* pass by pointer return by value */
int funcd ( int** i )

        {                /* what is the value of  i,*i,** i ? */
        int* q = *i;  /* what is the value of q, *q ? */
        *q++;          /* what is the value of q.*q ? */
        *i = q;        /* what is the value of i,*i ? */
        return *i;     /* what does the function return ? */
        }
```

LESSON 6 EXERCISE 3

Declare and define a function that receives a value, a pointer and a pointer to pointer and returns a pointer to pointer. In your main function declare and assign values to a variable, pointer and pointer to pointer. Print out the values of your variables using **printf.** Pass the variable to the function. Inside the function assign new values to the function's parameters. Print out the values of your variables using the returned pointer to pointer after the function is called. Call your program L6ex3.c.

C PROGRAMMERS GUIDE LESSON 7

File:	CGuideL7.doc
Date Started:	July 12,1998
Last Update:	Mar 1, 2002
Status:	proof

LESSON 7 USING FUNCTIONS

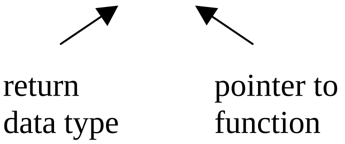
POINTERS TO FUNCTIONS

You can even have pointers to a function. Why do we need pointers to functions ? You may need a function that needs to call different functions, the functions to be called may not be known at compile time. A good example is a function that calculates the line, square and cube of a number. Instead of writing three or more calculate functions to calculate line, square and cube we can have only one function called calculate. We would have three additional functions called line, square and cube. We will then pass a function pointer to the calculate function telling which function to call, the line, square or cube. The calculate function does not know which function it receives it just gets a pointer to a function then executes the function by calling the pointer.

declaring a function pointer

It is easy to declare a function pointer. To declare a function pointer you proceed the function pointer name with a star * and include an empty parameter list () to indicate a function.

```
return data_type (* function name) ( parameter list);

int (*fp)( ); /* declare pointer to function f */
```



Brackets are needed around the star and function. Why ? Because the brackets force precedence. Without the brackets the compiler gets confused.

Declaring a function pointer is like declaring a pointer variable.

```
int *p; /* declare a integer pointer */
```

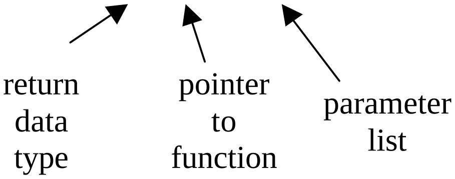
Except we use the empty round brackets to indicate we want a pointer to a function

```
int (*fp)( ); /* declare pointer to function f */
```

declaring a function pointer where the function has parameters

You need to supply a parameter list when declaring function pointers to functions having parameters.

```
return data_type (* function name) ( parameter list);

int (*fp)(int x); /* declare pointer to function f */
```



How would you declare a function pointer where the function does not return a value ?

calling a function using a function pointer

To call a function using a function pointer you need to put a * in front of the function pointer and supply arguments enclosed in round brackets. You need the * because you want to get the function pointed to by the function pointer. Again you need the additional round brackets or else the compiler gets confused.

```
(*fp)();    /* call a function pointed to by a pointer with no arguments */
```

If the function receives parameters then you have to pass arguments to the function pointed to by the function pointer.

```
(*f)(x);     /* call a function pointed to by a pointer with   arguments */
```

If the function returns a value then you use the assignment operator to assign the value from the function to a variable.

```
int y = (*f) ( x );   /* call a function pointed to by a pointer returns a value */
```

Using a function pointer is like getting the value from a pointer variable.

```
int y = *p; /* declare a integer pointer */
```

the only difference is that you are calling the function pointed to by the function pointer.

```
int y = (*f) ( x );   /* call a function pointed to by a pointer returns a value */
```

assigning functions to function pointers

Before you can use a function pointer, you need to assign the address of a function to it. Assuming we have a function called square, we assign the address of the function square to the function pointer fp.

```
fp = square;  /* assign address of function square to function pointer f */
```

Assigning a function to a function pointer is like assigning the address of a variable to a pointer variable.

```
p = &x; /* assign address of x to pointer variable p */
```

We don't need to use the & because the compiler knows to assign the address (just like when we used arrays) You could use the & if you want to.

example program using pointers to functions:

```
/* lesson 6 program 1 */

#include <stdio.h>

int square(int x);

void main()
{
int x = 5;
int (*fp)(int x);
fp = square;
x = (*fp)(x);  /* call square function using pointer to function fp */
printf("the square is: %d\n",x);
}

/* square function */
int square(int x)
{
return x*x;
}
```

```
the square is: 25
```

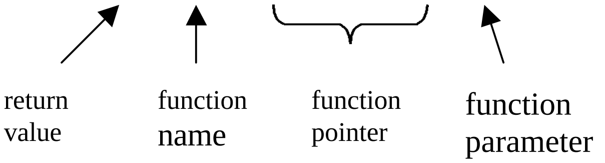
LESSON7 EXERCISE 1

Write a program that calls a function that receives a positive parameter. The function calls itself using a function pointer. Every time the function calls itself decrement the received number by one. The function should stop calling itself when the number is zero. Inside the function print out the received number. Call your program L7Ex1.c

function pointer parameters

You can use a function pointer as a parameter in a function. The arguments required by the function pointer must be supplied separately as additional parameters. They cannot be passed in the function pointer declaration. The following is a function prototype called calculate using a function pointer as a parameter.

```
return_data_type  function_name
(return_data_type (*function_pointer_name)function (pointer_paramater list), parameters_list );

int calculate (int ( *fp ) ( int x), int x ); /* declare a function that has a function pointer */
```



The function parameter is used to supply a value to the function pointer.

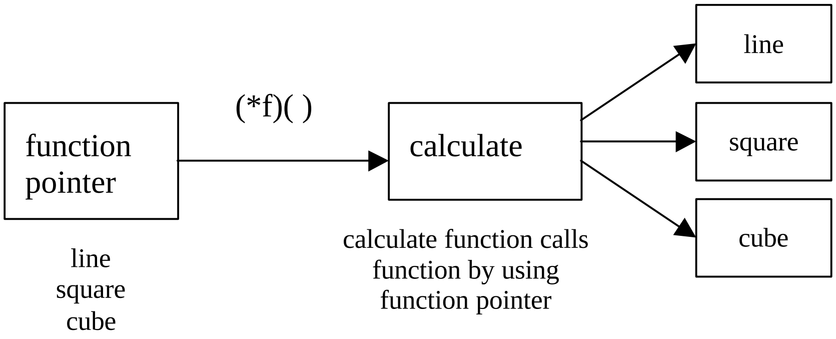
calling a function with function pointer parameter

All you need to do is supply a function name as an argument and supply any arguments needed by that function

```
function_name (  function_pointer ,  argument_list );

calculate( line, 5 );       /* call calculate passing  pointer to line function  */
```

We can use the calculate( ) function with many different functions. We just include which function we want to use as an argument.

```
x = calculate( line, 5 );      /* call calculate passing  pointer to line function  */
x = calculate( square, 5 );  /* call calculate passing  pointer to square function */
x = calculate( cube, 5);       /* call calculate passing  pointer to cube function  */
```



defining a function that receives a function pointer

A function that receives a function pointer must also receive argument to the function pointer. A function pointer cannot receive arguments. This makes sense since you cannot pass an argument with a pointer. The function then calls a function using the function pointer and passes the received parameter as arguments. The function pointer call may return or not return a value.

In our example the calculate function receives the function pointer fp and x.

```
int calculate (int (*fp)(int x ) ,int x)
```

The **calculate** function the calls a function using the function pointer fp and passes x to it in its argument list. The returned value is assigned to y.

```
int y = (*fp) ( x );        /* call function pointed to by f */
```

The **calculate** function returns the value of y when it terminates.

```
return y;
```

Here is the complete calculate function:

```
/* call calculate and pass function pointer and number */
int calculate (int (*fp)( int x) ,int x)
{
int y = (*fp) ( x );        /* call function pointed to by f */
return y;
}
```

operation

(1) pass x to function pointed to by f
(2) assign return value to y

example program using function pointers as parameters

Using pointers to functions is very professional programming, but may be very confusing to beginner programmers. Pointers to functions is just another evolution in programming that lets you do more exotic things. The following is an example program demonstrating pointers to functions as parameters. You must have the functions line, square and cube.

```
/* testing pointers to functions lesson 7 program 2 */
#include <stdio.h>

/* function prototypes */
int calculate (int (*f)( int x), int x ); /* passes pointer to function */
int line(int x);      /* return line of a number  */
int square(int x);    /* return square of a number */
int cube(int x);      /* return cube of a number  */

 /* main function */
void main()
{
int x  = calculate(line,5);          /* calculate line of 5    (x)  */
printf("the line of x is %d\n",x);
x = calculate(square,5);             /* calculate square of 5  (x * x) */
printf("the square of x is %d\n",x);
x = calculate(cube,5);               /* calculate cube of 5  (x * x * x) */
printf("the cube of x is %d\n",x);
}
```

```
/* call calculate and pass function pointer and number */
int calculate (int (*fp)( int x) ,int x)

        {
        int y = (*fp) ( x );        /* call function pointed to by f */
        return y;
        }

/*  return line of a number  */
int line(int x)

        {
        return x;          /* return line of x */
        }

/* return square of a number */
int square(int x)

        {
        return x * x;    /* return square of x */
        }

/* return cube of a number  */
int cube(int x)

        {
        return x * x * x;   /* return cube of x */
        }
```

program output:

```
the line of x is 5

the square of x is 25

the cube of x is 125
```

LESSON 7 EXERCISE 2

In your main function make an array of function pointers. Assign the function line, square and cube to the array elements. Pass one of the functions to the calculate function using the array. Call your program L7ex2.c.

LESSON 7 EXERCISE 3

In your main function make an array of function pointers. Assign the function line, square and cube to the array elements. Pass the array to the calculate function and a index to one of the functions in the array. Call the appropriate function from the array depending on the index supplied. Call your program L7ex3.c.

LESSON 7 EXERCISE 4

Make a structure that holds a value and pointers to functions line, square and cube. Call the line, square and cube functions using the function pointers stored in the structure. Call your program L7ex4.c

PASSING ARRAYS TO FUNCTIONS

passing reserved 1 dimensional arrays to functions

You may also pass arrays to functions. Can you still remember how to declare an 1 dimensional array ?

```
 int a[5]; /* declare a 1 dimensional array of 5 integer elements */
```

a	int	int	int	int	int

Memory for array a is **reserved** in compile time. Arrays the are declared in compile time are passed to functions by array.

```
        return_data_type  function_name (array_data_type array_name[optional size])

        void func1a ( int a[ 5] );    /* pass a one dimensional array */
```

The size is optional you can also say:

```
        void func1a ( int a[ ] );    /* pass a one dimensional array */
```

When you call the function you just supply the array name like a, you do not need the & because the array name is the address of the first element of the array.

```
        func1a(a); // pass array to function
```

Here is the function that receives an array

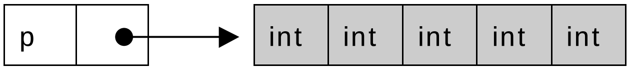
```
        /* pass a one dimensional array of unknown length
        void func1a ( int a[ ] );

        {
         printf("%d\n", a[0]);   /* print out 1 element of array */
        a[0]=a[0]+1;
        }
```

If you change the value of the array in the calling function it also is changed in the called function.

passing allocated 1 dimensional arrays to functions

To allocate memory for arrays in run time a pointer to an array is required.

```
        int *pa = (int*)malloc(sizeof(int)*5); /* declare a  pointer to a 1 dimensional array of 5 columns */
```



Alternatively you can assign a pointer an existing array. because the array name is the address of the first element of the array.

a = &a[0]

```
        int*pa = a;
```

Now you can pass an array by pointer.

```
        void func1p ( int *pa );    /* pass a one dimensional array by pointer */
```

When you call the function you just supply the pointer name like pa ,

```
    func1p(pa);
```

Here is the function that receives an array. Notice we use array indexes to access the individual elements of the array pa[1]. alternatively we can use offsets to pointers (pa + 1)

```
    /* pass a one dimensional array of unknown length
    void func1p ( int* pa );
    {
    printf("%d\n", a[0]);   /* print out 1 element of array */
    pa[0]=pa[0]+1;
    }
```

If you change the value of the array in the calling function it also is changed in the called function.

example program passing 1 dimensional arrays

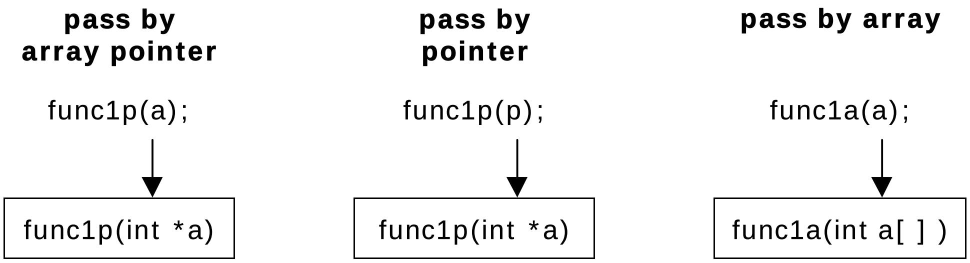
The following program uses the above function prototypes to illustrate the different methods of passing 1 dimensional arrays to functions. Notice we just pass the name of the array as a pointer because a = &a[0] the address of the first element in the array.

```
/* lesson 7 program 3 */
/*  passing 1 dimensional arrays to functions */

#include <stdio.h>
void func1a ( int a[ ] );
void func1p ( int* a );
void main()

    {
    int *pa = (int*)malloc(sizeof(int)*5); /* declare a  pointer to a 1 dimensional array of 5 columns */

    int a[5] = {1,2,3,4,5};     /* declare a 1 dimensional array of 5 columns */
```

1	2	3	4	5

```
    func1a(a); /* pass the 1 dimensional array to function pass by array */
    func1p(a); /* pass the 1 dimensional array to function pass by pointer to array */
    func1p(pa); /* pass the 1 dimensional array to function pass by pointer */
    free(pa); /*free memory for allocated array */
    }

    /* pass a one dimensional array by array */
    void func1a ( int a[ ] )

        {
         printf("%d\n", a[0]);   /* print out 1 element of array */
        }

    /* pass a one dimensional array by pointer */
    void func1p ( int* a )

        {
        printf("%d\n", a[0]);  /* print out 1 element of array */
        }
```

For a 1 dimensional array it does not make a difference pas by array or pass by pointer. The compiler teats everything the same.



pass by array pointer

func1p(a);

func1p(int *a)

pass by pointer

func1p(p);

func1p(int *a)

pass by array

func1a(a);

func1a(int a[ ])

Lesson 7 Exercise 5

Copy the above program into a program called L7ex3.c. In the func1a and func1p functions change some of the elements in the passed array. Make a function called print to print out the array elements. Print out the array values before and after the func1a and func1b functions are called.

passing reserved 2 dimensional arrays to functions

If you reserve a 2 dimensional array in compile time then you pass the 2 dimensional array by array. Can you still remember how to reserve memory for a 2 dimensional array ?

```
int b[2][3]; /* declare a 2 by 3 two dimensional array */
```

b	**col 0**	**col 1**	**col 2**
row 0	**b[0[0]]**	**b[0][1]**	**b[0][2]**
row 1	**b[1][0]**	**b[1][1]**	**b[1][2]**

For 2 dimensional arrays, when you declare an array parameter you must specify the number of columns or the compiler will report an error. The row size is optional.

```
void func2b ( int b[2 ][3 ] ); /* array of known column length , but known rows */
void func2b ( int b[ ][3] ); /* array of known column length , but unknown rows */
```

To pass a 2 dimensional array to a pointer you just use the array name. the array name represents the first elements in the array &b[0][0].

```
func2a(b);
```

b = &b[0][0]

Here is a function that receives a two dimensional array prints out an element then increments it.

```
/* pass array of known rows and columns */
void func2b ( int b[2 ][3 ] )

{
printf("%d\n", b[0][0] );  /* print out 1 element of two dimensional array */
b[0][0]++;
}
```

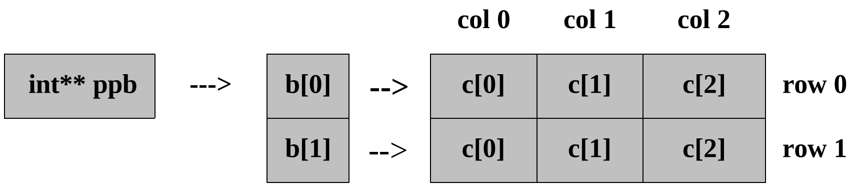
passing allocated 2 dimensional arrays to functions

Passing an array by pointer to pointer assumes the 2-dimensional array is allocated. To create a 2 row by 3 column allocated array, you first start with a array of row pointers

```
int**ppb = (int**)malloc(sizeof(int*)*2); /* declare an array of pointers for 2 rows */
```

Next you have to allocated an array of columns for each row.

```
ppb[0] = (int*)malloc(sizeof(int)*3);

ppb[1] = (int*)malloc(sizeof(int)*3);
```



To pass a 2 dimensional allocated array to a function the function expects a pointer to pointer that points to an array of pointers. (you cannot use int b[2][3])

```
void func2pp ( int** b); /* pass a two dimensional array of unknown rows and unknown columns */
```

When an 2 dimensional array is passed to a function you pas the pointer to pointer name.

```
func2pp(ppb); /* pass the 2 dimensional array to function */
```

Here is a function that receives a two dimensional array printouts an element then increments it.

```
/* pass pointer to pointer to a two dimensional allocated array */
void func2pp ( int** b)

{
printf("%d\n",b[0][0] );  /* print out 1 element of two dimensional array */
b[0][0]++;
}
```

example program passing 2 dimensional arrays

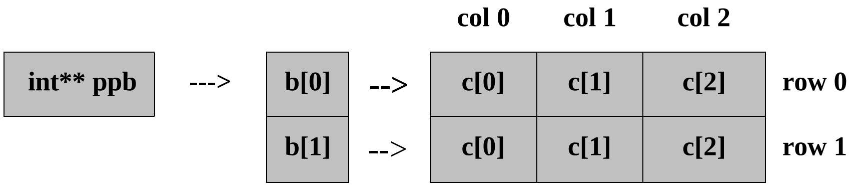
The following program uses the above function prototypes to illustrate the different methods of passing 2 dimensional arrays to functions.

```
/* lesson 7 program 34*/
#include <stdio.h>
#include <stdlib.h>
void func2bb( int b[2 ][3 ] );
void func2b ( int b[ ][3 ] );
void func2pp ( int** b);
```

```
/*  passing 2 dimensional arrays to functions */
void main()
        {
        int b[2][3];      /* declare a 2 dimensional array of 2 rows and 5 columns */
```

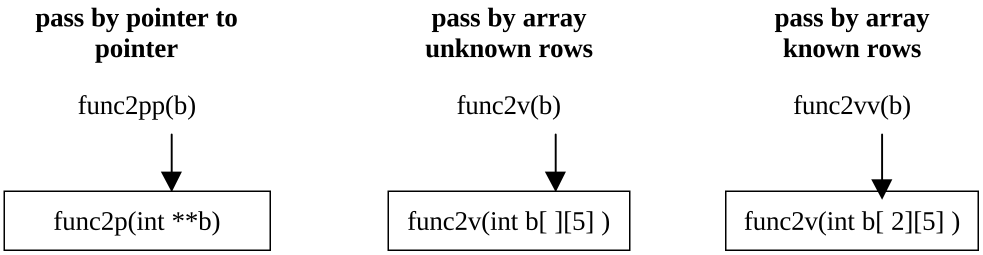
int	**int**	**int**
int	**int**	**int**

```
        int**ppb = (int**)malloc(sizeof(int*)*2); /* declare an array of pointers for 2 rows */

        ppb[0] = (int*)malloc(sizeof(int)*3);

        ppb[1] = (int*)malloc(sizeof(int)*3);
```



```
        func2b(b); /* pass the 2 dimensional reserved array to function */
        func2bb(b); /* pass the 2 dimensional reserved array to function */
        func2pp(ppb); /* pass the 2 dimensional allocated array to function */
        free(ppb[0]); /* free memory */
        free(ppb[1]);
        free(ppb);
        }

        /* array of known column length , but unknown rows */
        void func2bb( int b[2 ][3 ] )

                {
                printf("%d\n", b[0][0] );  /* print out 1 element of two dimensional array */
                }

        /* array of known column length , but known rows */
        void func2b ( int b[ ][3 ] )

                {
                printf("%d\n",b[0][0] );  /* print out 1 element of two dimensional array */
                }

        /* pass a two dimensional array of unknown rows and unknown columns */
        void func2pp ( int** b)

                {
                printf("%d\n",b[0][0] );  /* print out 1 element of two dimensional array */
                }
```

The following chart summarizes passing a 2 dimensional array to functions



pass by pointer to pointer

func2pp(b)

func2p(int **b)

pass by array unknown rows

func2v(b)

func2v(int b[ ][5])

pass by array known rows

func2vv(b)

func2v(int b[2][5])

LESSON 7 EXERCISE 6

Copy the above program into a program called L7ex6.c. In the func2b, func2bb and func2pp functions change some of the elemnts in the passed array. Make a function called print to print out the array elements. Print out the array values before and after the func1a and func1b functions are called.

PASSING A 1 DIMENSIONAL ARRAY TO A 2 DIMENSIONAL ARRAY

You can convert a 1 dimensional array to a 2 dimensional array by passing a 1 dimensional array to a function specify a 2 dimensional array as one of its parameters. The only catch is that the dimensions of the two dimensional array must be specified. This is possible because a memory block can be easily split up into a two dimensional array. a[i][j] = *(a+(i * row length)+j) (may not work in all compilers)

```
/* l7p4.c */
#include <stdio.h>
#include <stdlib.h>
void print(int b[][2]); /* declare a  2 dimensional array */

/* main function */
void main()

        {
        int *a= (int *)malloc(sizeof(int)*4)); /* allocate memory for a 1 dimensional array */
        print(a); /* print out 1 dim array as 2 dim array */
        }

/* function to print out a square matrix */
void print(int b[][2])

        {
        printf("%2d",b[0][0]);
        printf("%2d",b[0][1]);
        printf("%2d",b[1][0]);
        printf("%2d",b[1][1]);
        printf("\n");
        }
```

returning arrays from functions

You can also return arrays from functions as a pointer to an array.

```
        int* fp1 ( int a[ ] );    /* function receives and returns a pointer to an array */
```

Or allocated 2 dimensional arrays built from pointers to pointers

```
        int** fp2 ( int a[ ] );    /* function receives and returns a pointer to an array */
```

When an array is received from a function only the name is used, because an array name is already a pointer.

```
        a = fp1(a);      /* calling function receives a array from the called function */
```

LESSON 7 EXERCISE 7

Write a program that has a main function that declares an array of 5 integers pre-initialized to your favorite values. Pass the array and an index to a function called change( ). In the function add the array value and the array index at the specified index. Return the value of the array at the specified index. Print out the return value to the screen. Call your program L7ex7.c.

LESSON 7 EXERCISE 8

Write a program that has a main function that declares a two dimensional array of 2 rows and 3 columns integers pre-initialized to your favorite values. Pass the array and an row and column index to a function called change( ). Add to the array at the specified row and column index the array value and the array row and column index. Return the address of the array at the specified row and column index. Print out the return value to the screen. Call your program L7ex8.c.

LESSON 7 EXERCISE 9

Declare an array of 5 character strings each of length 81. Ask the user to enter 5 of their favorite words from the keyboard for each character string. Ask the user to type in 2 numbers from the keyboard where each number is between 0 and 4. Write a function called swap that will receive 2 character strings and swaps them. In your main function call swap to exchange the message pointers. When everything is swapped, print out all the character strings Call your program L7ex7.c.

LESSON 7 EXERCISE 8

Declare an array of 5 character string pointers. Allocate a character string of 81 characters to each pointer. Ask the uses to enter 5 of their favorite words from the keyboard for each character string. Write a function called swap that will receive 2 character string pointers. Ask the user to type in 2 numbers where each number is between 0 and 4. Call swap to exchange the character string pointers. When everything is swapped, go through the array and print out all the messages. Call your program L7ex8.c.

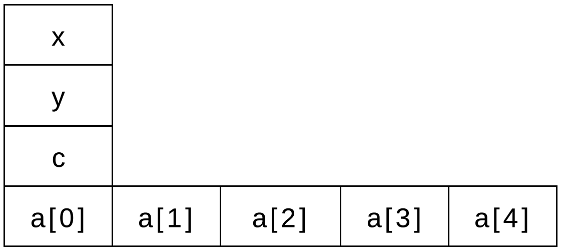
C PROGRAMMERS GUIDE LESSON 8

File:	CGuideL8.doc
Date Started:	July 12,1998
Last Update:	Mar 1, 2002
Status:	proof

LESSON 8 PASSING STRUCTURES TO FUNCTIONS

You may also pass structures to functions. Can you still remember what a structure is ? What is the difference between an array and a structure ? A structure contains many elements of different data types where an array has many elements of the same data type. The elements in a structure are declares as variables and are known as **members** of the structure.

```
typedef struct

{
int x;
long y;
char c:
int a[5];
}RECORD, *RECORD_PTR;
```

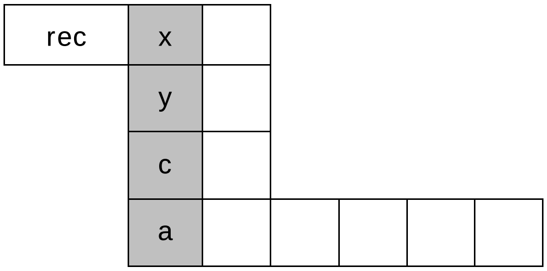


We use typedef to define our structure because we want to define our own structure data type identified by our user types RECORD and RECORD_PTR. RECORD represents the structure where RECORD_PTR represents a pointer to a structure user data type. Using typedef makes using structures much easier.

declaring structures and accessing values

You may declare a structure as a variable:

```
RECORD rec;  // reserved memory
```

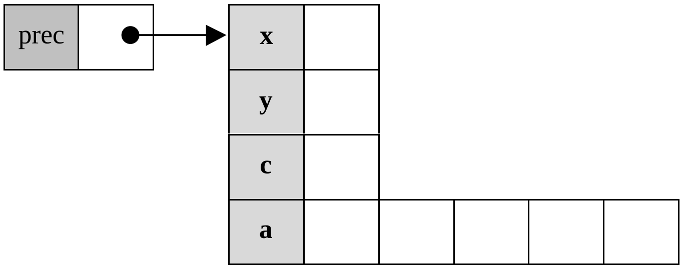


You access values of a structure by using the dot operator:

```
rec.x = 5;
int v = rec.x;
```

You may declare a structure as a pointer and allocate memory for it in run time:

```
RECORD_PTR prec = (RECORD_PTR)malloc(sizeof( RECORD)); // allocate memory
```



You access values of a structure pointed to by a pointer by using the arrow operator

```
prec->x = 5;
int v = prec->x;
```

passing structures to functions

There are 2 ways to pass Structures to functions:

(1) by value

(2) by pointer

The following function prototypes demonstrate pass structures to functions by value and by pointer

```
void fsv( RECORD rec );   /* pass a structure to function by value */
void fsp( RECORD_PTR prec );   /* pass a pointer to a structure to function f  */
```

Example of passing a RECORD by value to a function.:

```
fsv( rec);   /* structure rec is passed to function */
```

Example of passing a RECORD by pointer to a function.:

```
fsp( prec );   /* the pointer to structure  is passed to function */
fsp( &rec );   /* the address of structure rec is passes to function */
```

Here is an example program passing structures to functions:

```
/* lesson 8 program 1 */

/* this program demonstrates passing structures to functions  by value, pointer */
#include <stdio.h>
#include <stdlib.h>

/* define a structure */
typedef struct

        {
        int x;
        float f;
        char c;
        int a[5];
        }RECORD, *RECORD_PTR;

/* function prototypes */
void fsv(RECORD rec);
void fsp(RECORD_PTR prec);

/* main function */
main()

        {
        RECORD rec; /* declare RECORD as a variable */
        RECORD_PTR prec; /* declare pointer to a RECORD */

        /* assign values to RECORD */
        rec.x = 5;
        rec.f=10.5;
        rec.c ='a';
        rec.a[1]= 5;

        /* print out values stored in RECORD */
        printf("%d %f %c %d\n", rec.x,rec.f,rec.c,rec.a[1]);

        /* declare RECORD as a pointer and allocate memory for structure */
        prec = (RECORD_PTR)malloc(sizeof( RECORD));

         /* assign values to RECORD */
        prec->x = 5;
        prec->f=10.5;
        prec->c='a';
        prec->a[1]= 5;

        /* print out values stored in RECORD 1 */
        printf("%d %f %c %d\n", prec->x,prec->f,prec->c,prec->a[1]);
        } /* end main */
```

```
/************/
/* functions  */
/************/

/* pass structure by value */
void fsv(RECORD rec)
{
printf("pass by value: ");
/* print out values stored in RECORD */
printf("%d %f %c %d\n", rec.x,rec.f,rec.c,rec.a[1]);
}

/* pass by pointer */
void fsp(RECORD_PTR prec)
{
printf("pass by pointer: ");
/* print out values stored in RECORD */
printf("%d %f %c %d\n", prec->x,prec->f,prec->c,prec->a[1]);
}
```

LESSON 8 EXERCISE 1

Define a structure and write a function that receives a structure and assign different values to the received structure. In the main function declare and initialize the structure with some default values. Print out the values before and after the function is called. Write functions using pass by value, pass by pointer. You will discover something interesting. Call your program L8ex1.c

Arrays of structures

You may have an array of structures declared as a variable: (reserved memory)

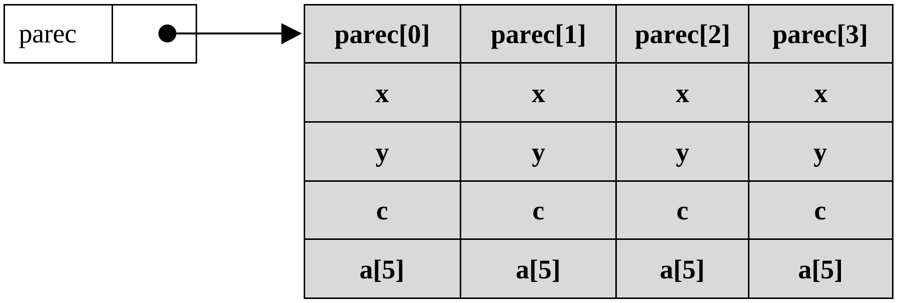
RECORD arec[4];	**arec[0]**	**arec[1]**	**arec[2]**	**arec[3]**
	x	**x**	**x**	**x**
	y	**y**	**y**	**y**
	c	**c**	**c**	**c**
	a[5]	**a[5]**	**a[5]**	**a[5]**

You access values of an array of structure by specifying the array index using the dot operator:

```
arec[1].x = 5; /* assign 5 to array structure index 1 member x */

int v = arec[1].x; /* read value at array structure index 1 member x */
```

You may allocate memory for an array of structures at run time pointed to by a structure pointer.

```
RECORD_PTR parec = (RECORD_PTR*)calloc(4,sizeof(RECORD_PTR));
```



parec[0]	parec[1]	parec[2]	parec[3]
x	x	x	x
y	y	y	y
c	c	c	c
a[5]	a[5]	a[5]	a[5]

You access values of an array structure pointed to by a pointer by specifying the array index using the dot operator

```
parec[2].x = 5;   /* assign 5 to array structure index 2 member x */

int v = parec[2].x; /* read value at array structure index 2 member x */

/* lesson 8 program 1 */

/* this program demonstrates passing structures to functions */
/* by value, pointer */
#include <stdio.h>
#include <stdlib.h>

/* define a structure */
typedef struct

        {
        int x;
        float f;
        char c;
        int a[5];
        }RECORD, *RECORD_PTR;

/* function prototypes */

/* pass array of structure by array  */
void fsaa(RECORD arec[],int i);

/* pass array of structures by pointer */
void fsap(RECORD_PTR parec, int i);
```

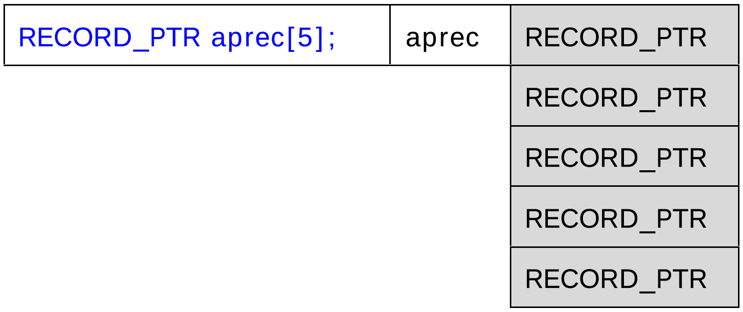
```
/* main function */
void main()

        {
        RECORD arec[4]; /* declare array of structures as a variable */
        RECORD_PTR parec;  /* pointer to an array of structures */

        /* using an array of records */
        arec[1].x = 5; /* assign values to RECORD 1 */
        arec[1].f = 10.5;
        arec[1].c = 'a';
        arec[1].a[1]= 5;

        /* print out values stored in RECORD 1 */
        printf("%d %f %c %d\n", arec[1].x,arec[1].f,arec[1].c,arec[1].a[1]);

        fsaa(arec,1); /* pass by value */
        /* print out values stored in RECORD 1 */
        printf("%d %f %c %d\n", arec[1].x,arec[1].f,arec[1].c,arec[1].a[1]);

        /* declare array of structures as a pointer */
        parec = (RECORD_PTR*)calloc(4,sizeof(RECORD));

        /* assign values to RECORD 1 */
        parec[1].x=5;
        parec[1].f=10.5;
        parec[1].c='a';
        parec[1].a[1]=5;

        /* print out values stored in RECORD 1 */
        printf("%d %f %c %d\n", parec[1].x,parec[1].f,parec[1].c,parec[1].a[1]);
        fsap(parec,1);

        /* print out values stored in RECORD 1 */
        printf("%d %f %c %d\n", parec[1].x,parec[1].f,parec[1].c,parec[1].a[1]);
        } /* end main */

        /* pass structure  by array */
        void fsa(RECORD arec[])
        {
        arec[1].x = 1;
        arec[1].f = 1.1;
        arec[1].c = 'b';
        arec[1].a[1] = 1;
        }

        /* pass structure by pointer array return structure by pointer */
        void fsap(RECORD_PTR parec,int i)
        {
        parec[i].x = 1;
        parec[i].f = 1.1;
        parec[i].c = 'b';
        parec[i].a[1] = 1;
        }
```

LESSON 8 EXERCISE 2

Declare an array of 5 structures Ask the uses to initialize 2 of the structures from the keyboard. Print out all the structure contents using the array of structures. Write a function called swap that will receive 2 structures, In main call swap to exchange the structures. When everything is swapped, print out all the structures using the array of structures. Call your program L8ex2.c

arrays of structure pointers

You may have an array of pointers to structures declared as a variable (reserved memory)



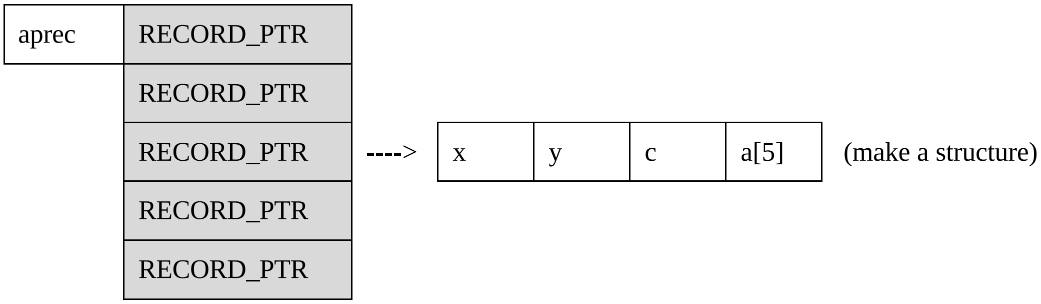
```
RECORD_PTR aprec[5];
```

aprec

RECORD_PTR
RECORD_PTR
RECORD_PTR
RECORD_PTR
RECORD_PTR

You need to allocate memory for a structure and assign it to one of the structure pointers in your array of structure pointers before you can access structures in your array of pointers to structures:

```
aprec[2]=(RECORD_PTR)malloc(sizeof(RECORD));
```



aprec

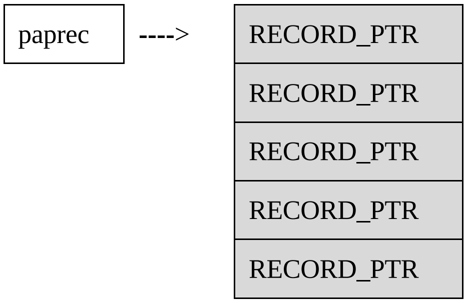
RECORD_PTR
RECORD_PTR
RECORD_PTR ----> x | y | c | a[5] (make a structure)
RECORD_PTR
RECORD_PTR

You access values of a structure pointed to by one of your pointers in your array by specifying the array index and using the arrow operator:

```
aprec[2]->x = 5; /* assign 5 to array structure index 2 member x */

int v = aprec[2]->x; /* read value at array structure index 2 member x */
```

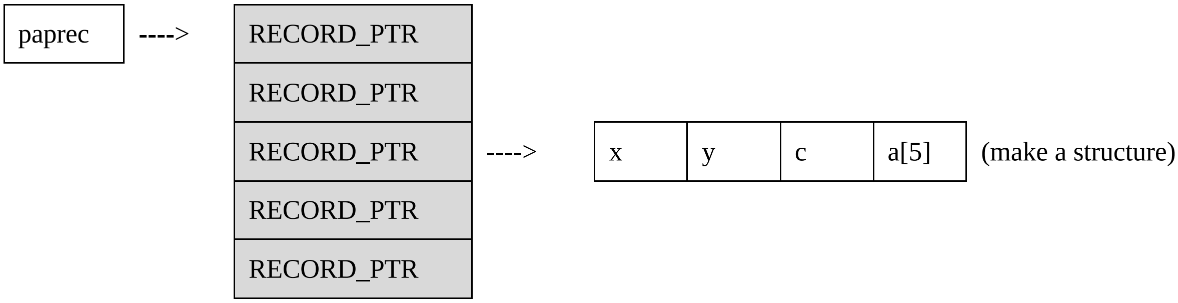
You may have an array of pointers to structures declared as a pointer to a pointer allocated in run time

```
RECORD_PTR* paprec = (RECORD_PTR*)calloc(4,sizeof(RECORD_PTR));
```



You also need to allocate memory for a structure and assign it to one of the structure pointers in your array of structure pointers before you can access structures in your array of pointers to structures:

```
paprec[1]=(RECORD_PTR)malloc(sizeof(RECORD));
```



You access values of the structure pointed by one of your pointers in your array by specifying the array index and using the arrow operator

```
paprec[2]->x = 5; /* assign 5 to array structure index 2 member x */

int v = paprec[2]->x; /* read value at array structure index 2 member x */
```

What is the difference between **arec[2].x** and **parec[2]->x** ?

arec is a array of structures so the dot operator is used to access the member of each structure. paprec is an array of structure pointers so the -> operator is needed to access members of each structure pointed to by a pointer.

Here is a sample program demonstrating passing arrays of structures by value and by pointer.

```
/* lesson 8 program 2 */

/* this program demonstrates passing arrays of structures to functions */
#include <stdio.h>
#include <stdlib.h>
```

```
    /* define a structure */
    typedef struct

         {
         int x;
         float f;
         char c;
         int a[5];
         }RECORD, *RECORD_PTR;

    /* function prototypes */
    void fsv(RECORD rec);
    void fsp(RECORD_PTR prec);

    /* main function */
    main()

         {
          RECORD_PTR aprec[10];  /* an array of pointers to structures */
         RECORD_PTR* paprec;  /* pointer to an array of pointers to structures  */

         /* allocate memory for a structure and assign to record pointer 1  */
         aprec[1]=(RECORD_PTR)malloc(sizeof(RECORD));

         /* assign values to RECORD 1 */
         aprec[1]->x = 5;
         aprec[1]->f =10.5;
         aprec[1]->c ='a';
         aprec[1]->a[1]= 5;

         /* print out values stored in RECORD 1 */
         printf("%d %f %c %d\n", aprec[1]->x,aprec[1]->f,aprec[1]->c,aprec[1]->a[1]);

         fsv(*aprec[1]); /* call pass by value */

         fsp(&(*aprec[1])); /* call pass by address */

         /* allocate memory for an array of structures as a pointer */
         paprec = (RECORD_PTR)calloc(4,sizeof(RECORD_PTR));

         /* allocate memory for a structure and assign to record pointer 1 */
         paprec[1]= (RECORD_PTR)malloc(sizeof(RECORD));

         /* assign values to RECORD 1 */
         paprec[1]->x=5;
         paprec[1]->f=10.5;
         paprec[1]->c='a';
         paprec[1]->a[1]=5;

         /* print out values stored in RECORD 1 */

printf("%d %d %d %d\n", paprec[1]->x,paprec[1]->f,paprec[1]->c,paprec[1]->a[1]);

         fsv(*paprec[1]);  /* call pass by value */
         fsp(paprec[1]); /* call pass by pointer */
```

```
/************/
/* functions  */
/************/

/* pass structure by value */
void fsv(RECORD rec)
{
printf("pass by value: ");
/* print out values stored in RECORD */
printf("%d %f %c %d\n", rec.x,rec.f,rec.c,rec.a[1]);
}

/* pass by pointer */
void fsp(RECORD_PTR prec)
{
printf("pass by pointer: ");
/* print out values stored in RECORD */
printf("%d %f %c %d\n", prec->x,prec->f,prec->c,prec->a[1]);
}
```

LESSON 8 EXERCISE 3

Declare an array of 5 structure pointers. Allocate memory for two structures and assign to two elements in the array of structure pointers. Ask the uses to initialize the structures from the keyboard. Print out all the structure contents using the array of structure pointers. Write a function called swap that will receive 2 record pointers. Each pointer will point to an element of the array. Call swap to exchange the structure pointers. When everything is swapped, print out all the structure contents using the array of structure pointers. Call your program L8ex3.c

SUMMARY PASSING STRUCTURES TO FUNCTIONS

passing structures located in arrays to functions

pass by value fsv(RECORD rec);	**pass by pointer** fsp(RECORD_PTR *prec)
fsv(arec[0]);	fsp(&arec[0]);
fsv(parec[0]);	fsp(&parec[0]);
fsv(*aprec[0]);	fsp(arec[0]);
fsv(*paprec[0]);	fsp(parec[0]);

pass individual member elements of a structure to a function

pass by value fv(int x);	**pass by pointer** fp(int *p)
fv(rec.x);	fp(&rec.x);
fv(*prec->x);	fp(prec->x);
fv(arec[0].x);	fp(&arec[0].x);
fv(*parec[0].x);	fp(parec[0].x);
fv(aprec[0]->x);	fp(&arec[0]->x);
fv(*paprec[0]->x);	fp(parec[0]->x);

RETURNING STRUCTURES FROM FUNCTIONS

A function cam also return a structure. You can return by:

(1) value

(2) by pointer

Here are examples to pass structure to functions and return structures from functions:

prototype	using
/* pass by structure value return structure by value */ RECORD fsvv(RECORD rec);	rec = fsvv(rec);
/* pass structure by pointer return structure by pointer */ RECORD_PTR fspp(RECORD_PTR prec);	prec = fspp(prec);

A function can also return an array of structures. You cam only return an array as a pointer. Here are examples to pass array of structure to functions and return array of structures from functions:

prototype	using
/* pass structure by array return structure by pointer */ RECORD_PTR fsap(RECORD arec[]);	/* use memcpy to copy array of records */ memcpy(arec,fsap(arec),sizeof(arec));
/* pass structure by pointer array return structure by pointer */ RECORD_PTR fspap(RECORD_PTR parec);	parec = fspap(parec);
/* pass structure by array pointer return structure by pointer to pointer */ RECORD_PTR* fsappp(RECORD_PTR aprec[]);	/* use memcpy to copy array of records */ memcpy(aprec,fsappp(aprec),sizeof(aprec));
/* pass structure by pointer array to array pointer /* return structure by pointer to pointer */ RECORD_PTR* fspappp(RECORD_PTR* paprec);	paprec = fspappp(paprec);

The following program demonstrates passing and returning structure to functions.

```
/* lesson 8 program 3 */

/* this program demonstrates passing structures to functions */
/* by value, pointer */
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>

/* define a structure */
typedef struct

        {
        int x;
        float f;
        char c;
        int a[5];
        }RECORD, *RECORD_PTR;

/* function prototypes */

void fsv(RECORD rec);
void fsp(RECORD_PTR prec);

/* pass by structure value return structure by value */
RECORD fsvv(RECORD rec);

/* pass structure by pointer return structure by pointer */
RECORD_PTR fspp(RECORD_PTR prec);

/* pass structure by array return structure by pointer */
RECORD_PTR fsap(RECORD arec[]);

/* pass structure by pointer array return structure by pointer */
RECORD_PTR fspap(RECORD_PTR parec);

/* pass structure by array pointer return structure by pointer to pointer */
RECORD_PTR* fsappp(RECORD_PTR aprec[]);

/* pass structure by pointer array to array pointer
/* return structure by pointer to pointer */
RECORD_PTR* fspappp(RECORD_PTR* paprec);

/* main function */
main()

        {
        RECORD rec; /* declare RECORD as a variable */
        RECORD_PTR prec; /* declare pointer to a RECORD */
        RECORD arec[4]; /* declare array of structures as a variable */
        RECORD_PTR parec;  /* pointer to an array of structures */
        RECORD_PTR aprec[10];  /* an array of pointers to structures */
        RECORD_PTR* paprec;  /* pointer to an array of pointers to structures  */
```

```
/* assign values to RECORD */
rec.x = 5;
rec.f=10.5;
rec.c ='a';
rec.a[1]= 5;

/* print out values stored in RECORD */
printf("%d %f %c %d\n", rec.x,rec.f,rec.c,rec.a[1]);

/* declare RECORD as a pointer and allocate memory for structure */
prec = (RECORD_PTR)malloc(sizeof( RECORD));

/* assign values to RECORD */
prec->x = 5;
prec->f=10.5;
prec->c='a';
prec->a[1]= 5;

/* print out values stored in RECORD 1 */
printf("%d %f %c %d\n", prec->x,prec->f,prec->c,prec->a[1]);

/* using an array of records */
arec[1].x = 5; /* assign values to RECORD 1 */
arec[1].f = 10.5;
arec[1].c = 'a';
arec[1].a[1]= 5;

/* print out values stored in RECORD 1 */
printf("%d %f %c %d\n", arec[1].x,arec[1].f,arec[1].c,arec[1].a[1]);
fsv(rec); /* pass by value */
fsp(&rec); /* pass by address */

/* declare array of structures as a pointer */
parec = (RECORD_PTR)calloc(4,sizeof(RECORD));

/* assign values to RECORD 1 */
parec[1].x=5;
parec[1].f=10.5;
parec[1].c='a';
parec[1].a[1]=5;

/* print out values stored in RECORD 1 */
printf("%d %f %c %d\n", parec[1].x,parec[1].f,parec[1].c,parec[1].a[1]);
fsv(*prec); /* call by value */
fsp(prec); /* call by pointer */

/* allocate memory for a structure and assign to record pointer 1  */
aprec[1]=(RECORD_PTR)malloc(sizeof(RECORD));

/* assign values to RECORD 1 */
aprec[1]->x = 5;
aprec[1]->f =10.5;
aprec[1]->c ='a';
aprec[1]->a[1]= 5;
```

```
                /* print out values stored in RECORD 1 */

printf("%d %f %c %d\n", aprec[1]->x,aprec[1]->f,aprec[1]->c,aprec[1]->a[1]);

                fsv(*aprec[1]); /* call pass by value */
                fsp(&(*aprec[1])); /* call pass by address */

                /* allocate memory for an array of structures as a pointer */
                paprec = (RECORD_PTR*)calloc(4,sizeof(RECORD_PTR));
                /* allocate memory for a structure and assign to record pointer 1 */
                paprec[1]= (RECORD_PTR)malloc(sizeof(RECORD));

                /* assign values to RECORD 1 */
                paprec[1]->x=5;
                paprec[1]->f=10.5;
                paprec[1]->c='a';
                paprec[1]->a[1]=5;

                /* print out values stored in RECORD 1 */

printf("%d %d %d %d\n", paprec[1]->x,paprec[1]->f,paprec[1]->c,paprec[1]->a[1]);

                fsv(*paprec[1]);  /* call pass by value */
                fsp(paprec[1]); /* call pass by pointer */

                /* passing individual values */

                /* pass by value */
                fv(rec.x);
                fv(prec->x);
                fv(arec[1].x);
                fv(parec[1].x);
                fv(aprec[1]->x);
                fv(paprec[1]->x);
                /* pass by pointer */
                fp(&rec.x);
                fp(&prec->x);
                fp(&arec[1].x);
                fp(&parec[1].x);
                fp(&aprec[1]->x);
                fp(&paprec[1]->x);

                /* pass array of structures to functions  */
                /* returning structures from functions */
                rec = fsvv(rec);
                prec = fspp(prec);

                /* use memcpy to copy array of records */
                memcpy(arec,fsap(arec),sizeof(arec));
                parec = fspap(parec);

                /* use memcpy to copy array of records */
                memcpy(aprec,fsappp(aprec),sizeof(aprec));
                paprec = fspappp(paprec);
                } /* end main */
```

```
/***********/
/* functions  */
/***********/

/* pass by value */
void fv(int x)
{
printf("pass by value: ");
printf("%d\n", x);/* print out values */
}

/* pass by pointer */
void fp(int *p)
{
printf("pass by value: ");
printf("%d\n", *p); /* print out values  */
}

/* pass structure by value */
void fsv(RECORD rec)
{
printf("pass by value: ");
/* print out values stored in RECORD */
printf("%d %f %c %d\n", rec.x,rec.f,rec.c,rec.a[1]);
}

/* pass by pointer */
void fsp(RECORD_PTR prec)
{
printf("pass by pointer: ");
/* print out values stored in RECORD */
printf("%d %f %c %d\n", prec->x,prec->f,prec->c,prec->a[1]);
}

/* pass by structure value return structure by value */
RECORD fsvv(RECORD rec)
{
rec.x = 1;
rec.f = 1.1;
rec.c = 'b';
rec.a[1] = 1;
return rec;
}

/* pass structure by pointer return structure by pointer */
RECORD_PTR fspp(RECORD_PTR prec)
{
prec->x = 1;
prec->f = 1.1;
prec->c = 'b';
prec->a[1] = 1;
return prec;
}
```

```
/* pass structure by array return structure by pointer */
RECORD_PTR fsap(RECORD arec[])
{
arec[1].x = 1;
arec[1].f = 1.1;
arec[1].c = 'b';
arec[1].a[1] = 1;
return arec;
}

/* pass structure by pointer array return structure by pointer */
RECORD_PTR fspap(RECORD_PTR parec)
{
parec->x = 1;
parec->f = 1.1;
parec->c = 'b';
parec->a[1] = 1;
return parec;
}

/* pass structure by array pointer */
/* return structure by pointer to pointer */
RECORD_PTR* fsappp(RECORD_PTR aprec[])
{
aprec[1]->x = 1;
aprec[1]->f = 1.1;
aprec[1]->c = 'b';
aprec[1]->a[1] = 1;
return aprec;
}

/* pass structure by pointer array to array pointer */
/* return structure by pointer to pointer */
RECORD_PTR* fspappp(RECORD_PTR* paprec)
{
paprec[1]->x = 1;
paprec[1]->f = 1.1;
paprec[1]->c = 'b';
paprec[1]->a[1] = 1;|
return paprec;
}
```

Program output:

```
5 10.500000 a 5
5 10.500000 a 5
5 10.500000 a 5
pass by value: 5 10.500000 a 5
pass by pointer: 5 10.500000 a 5
5 10.500000 a 5
pass by value: 5 10.500000 a 5
pass by pointer: 5 10.500000 a 5
5 10.500000 a 5
pass by value: 5 10.500000 a 5
pass by pointer: 5 10.500000 a 5
5 10.500000 a 5
pass by value: 5 10.500000 a 5
pass by pointer: 5 10.500000 a 5
pass by value: 5
pass by value: 5
pass by value: 5
pass by value: 5
pass by value: 5
pass by value: 5
pass by value: 5
pass by value: 5
```

LESSON 8 EXERCISE 4

Define a structure with 3 members of int, float and char data type. Define your structure using **typedef.** In your main program declare an array of 5 structures as a **variable.** Also declare a pointer variable to your structure. Pick an index of the array of structures and initialize the structure elements for that array index with your favorite values. Define a function called **choose()** that receives an array of structures and an index to one of the structures. The function should return a pointer to the structure identified by the index. Assign the return value from the function to your RECORD pointer. Using the printf statement, print out the values of the structure pointed to by your structure pointer. Call your program L8ex4.c

C PROGRAMMERS GUIDE LESSON 9

File:	CguideL9.doc
Date Started:	July 12,1999
Last Update:	Mar 1, 2002
Status:	proof

LESSON 9 USING BUILT IN C FUNCTIONS AND PROGRAMMING STATEMENTS

C has built in function to let you do common things. You have already been using them when you use printf and scanf. When you use built in functions you have to include the appropriate header files like **#include <stdio.h>** and **#include <stdlib.h>** We will present the string functions to copying and comparing strings, the time functions for printing time and data and the random number functions to generate random numbers.

STRING FUNCTIONS

C has functions for copying and joining and finding out the lengths of character strings. You must include the header file **#include <string.h>** when using these functions.

string function	prototype	example using
strcpy string copy	char * strcpy(char * dest, char * source); returns copied string from source to dest	strcpy (s1,s2);
strncpy copy string by length	char * strncpy(char * dest, char * source,int size); returns copied string from source to dest for size of string	strncpy (s1,s2,sizeof(s2));
strcat joins two strings	char* strcat(char* dest,char* source); returns joined source string to dest string	strcat (s1,s2);
strncat joins two strings by length	char* strcat(char* dest,char* source, int length); returns joined source string to dest string	strncat (s1,s2,20);
strlen string length	int strlen(char * source); returns length of string (not including terminator)	len = strlen(s1);
strcmp string compare compare 2 strings	int strcmp(char* s1, char* s2);	strcmp(s1,s2);
	less returns < 0 if (s1 < s2) equal returns 0 if (s1 = = s2) greater returns > 0 if (s1 > s2)	

string function	prototype	example using

strtok
Searches one string s1 for tokens, which are separated by delimiters defined in a second string s2.
	char *strtok(char *s1, const char *s2);
	strtok returns a pointer to a found token in string. A NULL pointer is returned when there are no more tokens. The first call to strtok returns a pointer to the first character of the first token in s1 and writes a null character into s1 immediately following the returned token. Subsequent calls with null for the first argument will work through the string s1 until no tokens remain.
		/* first call */
		p = strtok(s1, ",");
		/* A second call */
		p = strtok(NULL, ",");

strstr Scans a string s1 for the occurrence of a given substring. s2	char *strstr(const char *s1, const char *s2); strstr returns a pointer to the substring that was found. If s2 does not occur in s1, then returns null.	char *strstr(const char *s1, const char *s2);
strlwr string to lower case	char* strlwr(char * source) ; returns and converts string to lower case.	strlwr(s1);
strupr string to upper case	char* strupr(char * source); returns and converts string to upper case.	strupr(s1);
atoi ascii to integer "1234" to 1234	int atoi(char* str); returns and converts string to integer value	x = atoi(s);

itoa
integer to ascii
1234 to "1234"
	char* itoa(int num, char*, int radix);
	returns and converts integer to string at specified base (use 10 for decimal)
		char memory[81];
		itoa(x,memory,10);

String function example program. When you print out a string, you use the "%s" format specifier.

```
/* lesson 9 program 1 */

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

int square(int x);

void main()
{
char s1[]="cat";
char s2[]="dog";
char s3[81];
char s4[81];
char* p;
printf("s1: %s\n",s1);
printf("s2: %s\n",s2);
strcpy(s3,s1);
printf("s3: %s\n",s3);
strcat(s3," ");
strcat(s3,s2);
printf("s3: %s\n",s3);
printf("length s3: %d\n",strlen(s3));
printf("compare s1 to s2: %d\n", strcmp(s1,s2));
p = strtok(s3," ");
printf("first: %s\n",p);
p=strtok(NULL," ");
printf("second: %s\n",p);
printf("find: %s\n",strstr(s2,"og"));
printf("find: %s\n",strupr(s1));
printf("find: %s\n",strlwr(s1));
printf("%d\n",atoi("1234"));
itoa(1234,s4,10);
printf("%s\n",s4);
}
```

program output:

```
s1: cat
s2: dog
s3: cat
s3: cat dog
length s3: 7
compare s1 to s2: -1
first: cat
second: dog
find: og
find: CAT
find: cat
1234
1234
```

LESSON 9 EXERCISE 1

write a program that ask the user to type in five words. sort the words in alphabetically order and print to the screen.

TIME FUNCTIONS

Time functions are used to print out the current time or use time to measure program execution speed. The structure tm holds the date and time in a structure declared in **<time.h>**

.

```
struct tm
{
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

time functions:

prototype	description	example using

time_t **time**(time__t *time);	returns current calendar time of day in **seconds**, elapsed since 00:00:00 GMT, January 1, 1970.	time__t t;
		time(t)
		long t = time(NULL);

struct tm ***localtime**(time__t * time);	converts seconds to a **tm** structure returns a pointer to a **tm** structure	struct tm *local;
		local = localtime(&t);

char ***asctime**(struct tm *ptr);	converts a tm structure to a ascii string to print out data and time	printf(asctime(local));

double **difftime**(time__t t1, time__t t2);	returns the difference in seconds between time t1 and time t2	double diff;
		diff = difftime(end,start);

using the time functions

The example program gets the current time using the time() function and prints the time out using asctime(). A calculation is done next that will take a long time to do. We then get the current time and printed it out. The difference in time is calculated using the difftime() function. The time difference in seconds is printed out.

```
/* lesson 9 program 1 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* find out how long it takes to execute loop */
void main()

        {
        time_t start,end;
        tm *local;
        unsigned long i,diff;

        start = time(NULL); /* get start time */
        local = localtime(&start); /* convert to tm structure */
        printf("the starting time is: %s",asctime(local)); /* print out start time */
        for(i=0;i<100000L;i++);  /* long delay */
        end = time(NULL); /* get end time */
        local = localtime(&end); /* convert to tm structure */
        printf("the ending time is: %s",asctime(local)); /* print out end  time */
        diff = difftime(end,start); /* calculate difference in time */
        printf("The time for the calculation is: %dl seconds",diff); /* printout diff */
        }
```

program output:

```
the starting time is: Mon Jan 29 08:52:33 2001
the ending time is: Mon Jan 29 08:52:33 2001
The time for the calculation is: 0l seconds
```

LESSON 9 EXERCISE 2

Make a time structure. Ask the user to enter the time structure members from the keyboard. Print out the time using the structure,

RANDOM NUMBER GENERATOR FUNCTIONS

There are lots of times when you need random numbers. The following functions are used to generate random numbers. You must include **<stdlib.h>** when using the random number generator functions and **<time.h>** if using the srand() function. You must "**seed**" your random number generator with a starting number, if you do not then you will get the same random number every time you run your program.

prototype	description	example using

void srand(unsigned seed);	used to set the starting point for a sequence of random numbers. (**seed** with time of day)	long t = time(NULL);
		srand((unsigned) t);

void randomize();	initializes random number generator to some random number, based on the current time obtained from computer.	randomize();
int random(int num);	generates a random number between 0 and num	x = random(100);
int rand();	generates a sequence of random numbers from 0 to RAND_MAX (65536)	x = rand();

using the number random generator

This program **seeds** the random number with the current time. "**seed**" means the random number generator will generator different numbers everytime you run the program. If you do not seed the random number generator then you will get the same sequence of random numbers everytime you run the program. The first random number is between 0 and 99. Why ? The second random number generated is between 1 and 100 Why ?

```
/* lesson 9 program 2 */

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

/* generate random numbers between 0 and 99 and 1 and 100 */
void main()

     {
     int num;
     srand(time(NULL));  /* randomize(); */
     num = rand() % 100; /* random number between 0 and 99 */
     printf("%d",num);   /* print out random number */
     num = (rand() % 100) + 1; /* random number between 1 and 100 */
     printf("%d",num);   /* print out random number */
     }
```

LESSON 9 EXERCISE 3

Ask for 5 words from the keyboard and store in an array. use Generate random numbers from 0 to 4 to print out the words from the array.

FUNCTION STATEMENTS BY EXAMPLE

Function statements are now introduced by typical application. There is no specific order in which to use them. You must declare all variables at the beginning of your function before using statements. You may type in all the statements to see the program work. The programming statements are in blue and the comments are in green.

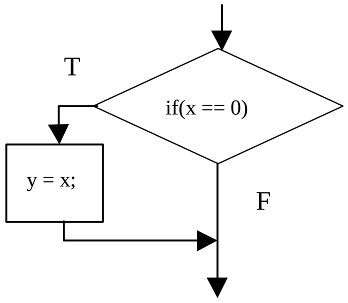
```
/* statement test program lesson 9 program 3 */

#include <stdio.h>

/* definitions */
#define NUM_ROWS  3
#define NUM_COLS  3
#define MAX  10
#define TRUE  1
#define FALSE  0

/* main function */
void main()

        {
        int i, j;
        int x =5,y=10;
```

if statement

The **if statement** is used with a condition to make a decision in a program flow. An **if** statement contains one or more conditions that are evaluated as **true** or **false**. When the **if condition** is **true** then the statements belonging to the **if** statement is executed. When the **if condition** is **false** then program flow is directed to the next program statement. If an **if statement** has more than one statement them the statements belonging to the **if** expressions must be enclosed by curly brackets. An **if** statement allows you to make a **choice** in program execution direction.

definition	one statement	more than one statement

if (condition)
true statement(s)
	if (x==0)
	y = x;
		if (x==0)
		{
		y = x;
		x=x+1;
		}

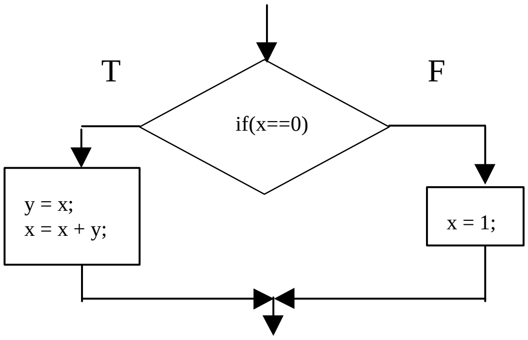


if - else statement

An **if statement** may have an optional **else** statement. The **else** statement executes the alternative **false** condition statements. Curly brackets are also needed if more than one statement belongs to the **else** or **if** statements.

```
if (condition)

     true  statement(s)

else

     false  statement(s)
```

```
if (x==0)
    {
    y = x;
    x=x+y;
    }
else
    x=1;
```

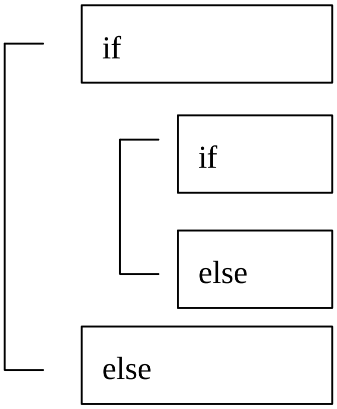


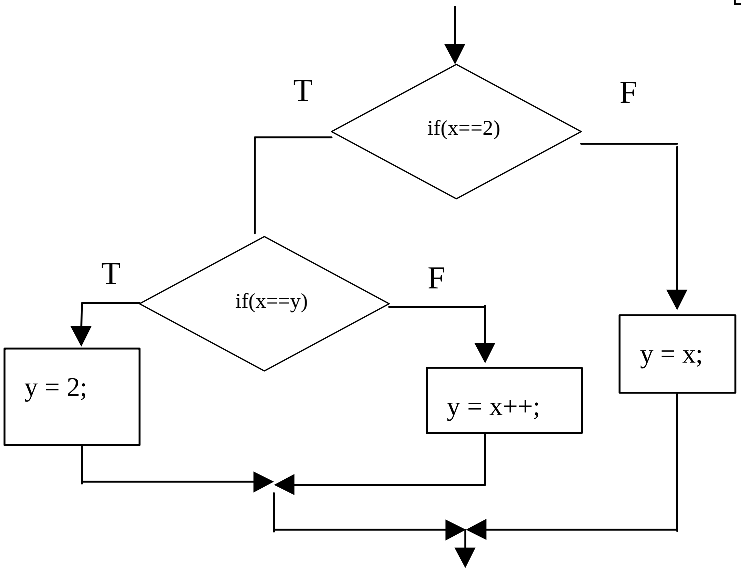
nested if-else statements

If - else statements may be nested. In this case the **else** belongs to the balanced **if**. In case of confusion to determine which **else** belongs to which **if** use curly brackets. Nested if-else statements is like taking a journey. The conditions decide to turn left or right.

```
if (condition)

     {
     statement
     if (condition)
        true  statement(s)
     else
        false  statement(s)
     }

else

          false statement(s)
```

```
if (x == 2)

     {
     if(x == y)
       y = 5;
     else
       y = x++;
     }

else

     y = x;
```

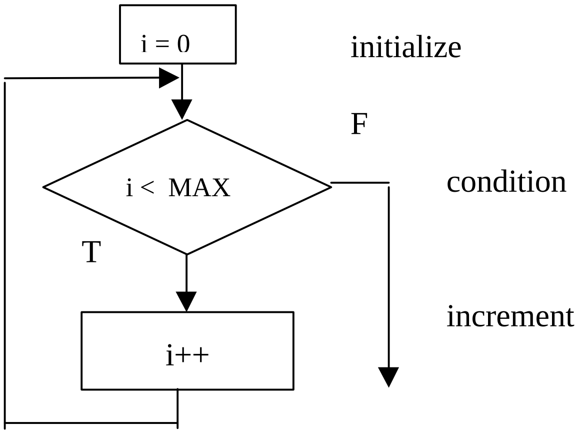




while loop

A while loop lets you repeatedly execute statements. The while loop tests a **condition** at the top of the loop. If the condition is **true** the statement inside the while loop is executed. When the while condition is **false** program execution exits the loop. You must be careful and make sure that the test condition is initialized before entering the loop. It is beneficial to initialize your test condition right in the condition. The **while statement** allows you to read in items when you do not know how many items you have. If the test condition is false before entering the loop the while loop statements will never get executed.

```
while (condition)        i = 0;

                         while(i < MAX)

   loop statement(s)           {
                               i++;
                               }

                         fprintf("I got: %d items", i);
```

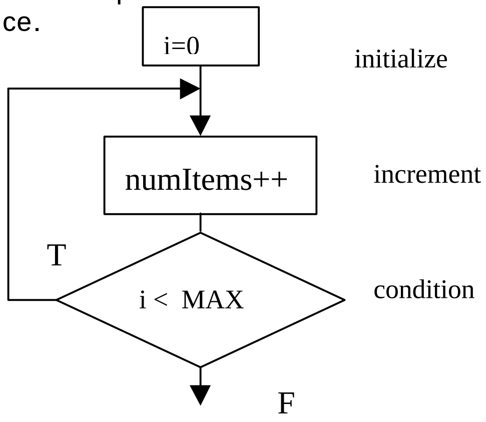


```
I got 10 items
```

do while loop

The do while loop also repeats statements. The do while loop tests the expression at the bottom of the loop. The statements inside the loop are executed at least once.

```
do                       i = 0;
   loop statement(s)     do

while ( expression );          {
                               numItems = i++;
                               } while ( i < MAX )

                         fprintf ( " I got " , %d ," items \n" , numItems );
```



```
I got 10 items
```

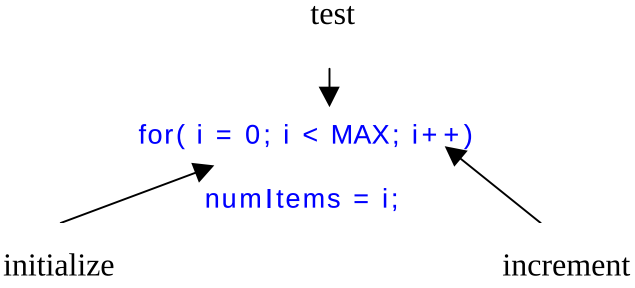
What is the difference between a **while** loop and a **do-while** loop ? A **while** loop may never execute a loop statement and a **do-while** loop will execute the loop statements at least one time.

for statement

The **for** loop is used when you know how many items you have. For example to read a known number of items from a file. The **for** loop uses an index for counting loop iterations. The index is initialized once before entering the loop, tested and increments for each loop iteration. The loop terminates when the index counter reaches its limit. A for statement is like an automatic while loop.

*for(**initializing** expression; **testing** condition ; **incrementing** expression)*

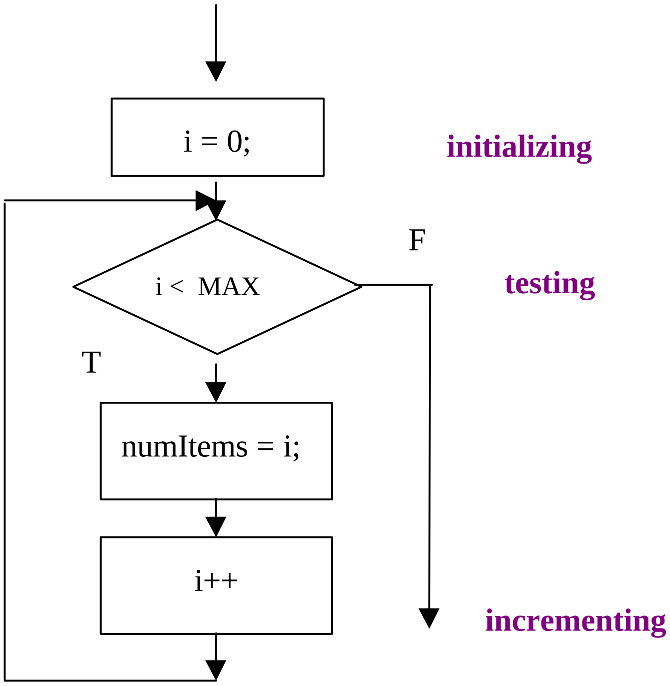
loop statements;



```
for( i = 0; i < MAX; i++)
        numItems = i;
```

If the **for** loop has more than one statement then curly brackets must be used to enclose the statements belonging to the for loop.

```
for(int i = 0;i<Max;i++)

        {
        numItems = i;
        x = x + numItems;
        }
```



All expressions in the for statement are optional. No expressions would be an never ending loop.

```
for ( ; ;); /* loop for ever */
```

You can include more than 1 index expression in each **for** loop expression, by using the comma operator;

```
            for(i = 0, j = numItems-1; i < numItems; i++, j - -)

                  a [ i ] = b [ j ]; /* copy reverse of an array */
```

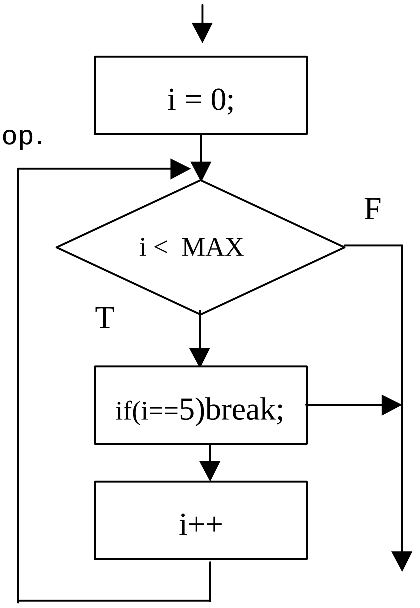
When would you use a **for** loop and when would you use a **while** loop ?

break statements

The break statement lets you get out of a **for, while** or **do** loop.

```
/* loop till break condition encountered */
i = 0;
while(i < MAX)

      {
      if(i==5)break;    /* exit loop if test condition true */
      printf("%d ",i);
      i=i+1;
      }
```

```
0 1 2 3 4
```

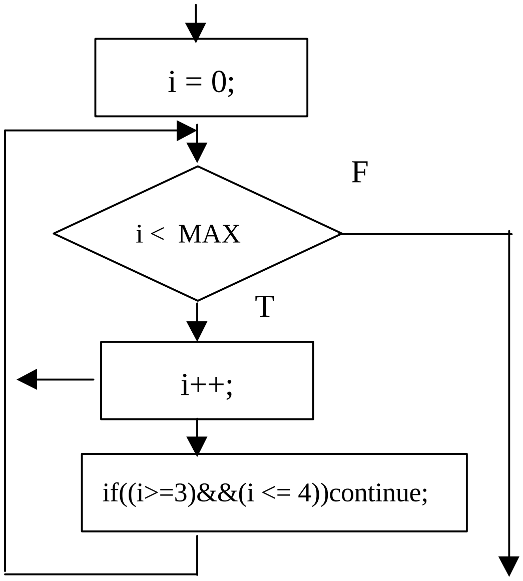


continue statement

The continue statement is used with **for, while, do** loop to continue to the next iteration of the loop to **skip** statements you do not want to evaluate for certain iterations.

```
int i= 0;

/* loop till i greater than MAX */
for(i = 0; i < MAX; i++)

{
i=i+i;   /* don't print out these numbers */
if ((i >= 3) && (i <= 4)) continue;
printf("%d ",x);
}
```

```
0 1 2 5 6 7 8 9 10
```

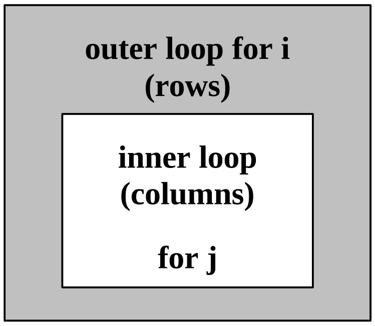


You must be careful where you put the continue statement. What is the difference between a **break** statement and a **continue** statement ? The **break** statement lets you exit the loop early and the **continue** statement lets you skip executing statements in the loop.

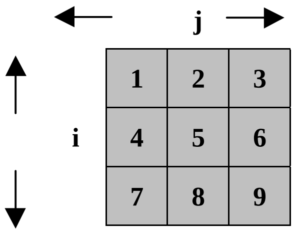
nested for statements

for statements are usually nested. This means a loop inside a loop. Nested **for** statements are handy in printing out the contents of a two-dimensional array. The inner loop prints out the columns of each row. The outer loop keeps track of which row to print.

```
int i, j,k=1 ;

/* loop for number of rows */
for (i = 0 ;i < NUM_ROWS; i++)
        {
        /* loop for number of columns */
        for(j=0; j < NUMCOLS; j++)

                    printf ( "%d", k); /* print out array element */

          printf ( "\n" ); / * start a new line for each row */
          }
```



The inner loop using the column index (j) prints out the contents of all the columns of the row pointed to by the row index (i)



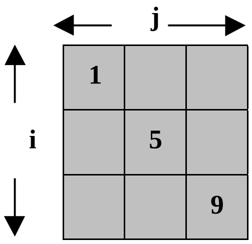
1	2	3
4	5	6
7	8	9

i	j
0	0, 1, 2
1	0, 1, 2
2	0, 1, 2

printing out diagonal of an 2 dimensional array

If you just want to print the diagonal of an array then you just print out the values of the array element when i == j, when the row index equals the column index.

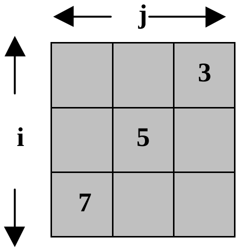
```
int i, j.k=1 ;

/* loop for number of rows */
for( i = 0; i < NUM_ROWS; i++)

        {
        /* loop for number of columns */
        for( j=0; j < NUMCOLS; j++)

                {
                /* test if column equal row */
                if( i = = j ) printf ( "%d", k++ ); /* print out rows and columns */
                }

        printf( "\n" );
        }
```



i	j
0	0, 1, 2
1	0, 1, 2
2	0, 1, 2

If the inner loop using the columns index (j) prints out the contents of the column only when the column index equals the row index. Which row to print is pointed to by the row index (i)

Can you write the code to print the out the reverse diagonal ?



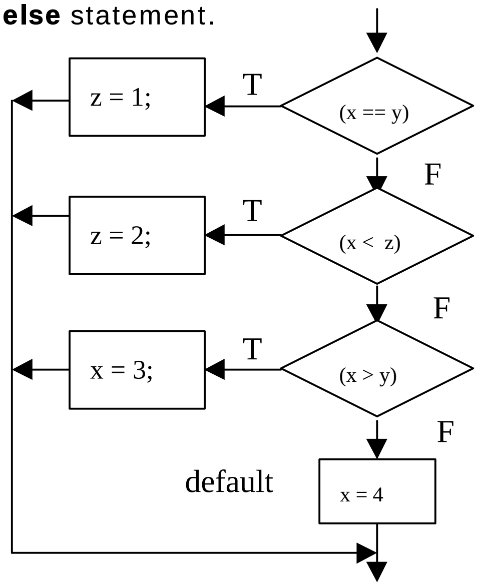
i	j
0	0, 1, 2
1	0, 1, 2
2	0, 1, 2

If the inner loop using the columns index (j) prints out the contents of the column only when the column index plus row index equals the number of columns minus 1. Which row to print is pointed to by the row index (i)

else-if

If - else statements can be sequenced to choose a certain condition. If the value you are looking for is not found then program execution is directed to the **else** statement.

```
if (condition)                  if(x == y)
        statement                       z=1;
else if (condition)             else if(x < z)
        statement                       z = 2;
else if (condition)             else if(x > y)
        statement                       x = 3;
else                            else
        default statement;              x = 4;
```



switch statement

To avoid typing in many else-if statements then the switch statement is used. A switch statement allows you to organize selections. A switch statement contains a switch expression and a case constant expression. The **switch expression** must be evaluated to a primitive data type, like char, int, float etc. The c**ase constant expression** must be a **constant value** and not a **variable name**. When the switch expression value matched a case constant program execution goes to the statements belonging to the case constant. The **break statements** stops the program execution from going to the next case. Without the **break statements**, you will get unexpected results. Sometimes it is desirable to omit the **case statements and breaks**. By using this technique you can test for multiple conditions.

```
        switch( expression)
        {
        case constant 1:
        statements
        *break;

               case constant 2:
               statements
               break;

               case constant N:
               statements
               break;

               default:
               statements
               break;
               }
```

```
switch(x)

{
case 0 :
      {
      printf ( " first item is: %d \n", items[i] ) ;
      break;
      }
/* case 1 or case 2 */
case 1:
case 2:
      {
       printf("the items are: %d %d\n", item[1], item[2]);
       break;
       }

default: break;
}
```

In the switch statement code example why does case 1 not have any statements or break statement ? Because we want to trap for two conditions when x == 1 and x == 2,

```
} /* end main */
```

Global Variables

Global variables are usually declared before the main program because it's the main program that uses them. Arrays or structures where the memory is allocated at compile time. In compile time memory is allocated before program execution begins. Functions should not access global variables and tables directly. Pointers to global variables and tables should be passed as parameters to functions. The main reason Global variable and tables are passed through parameters is to avoid data corruption. If the global variables and tables are changed directly by many functions, then global memory is easily corrupted. It is very difficult to trace which function is the culprit.

CONVENTIONS TO MAKE YOUR PROGRAMMING LIFE EASIER

If you follow certain programming style conventions then when you read your program you will instantly recognize which variables are global, which ones belong to a function etc.

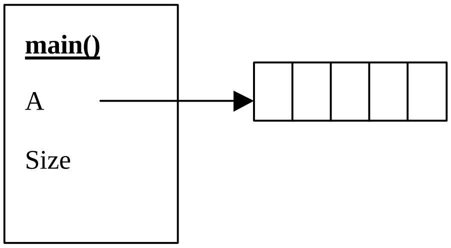
All **#defines** should be capital letters	#define MAX_DAYS 7
All **enums** should be start with a capital letter	enum Letters {A ,B, C, D};
All **typedef** data types should start with a capital	typedef node* Nodeptr;
All **function names** should be lower case	void func();
All **long function names** should split with underscore	void read_table();
All **variables** declared inside functions should be lower case	char name;
All **global variables, arrays, structures** should start with a capital	int A[ ];

**** ONLY PICK OUT 5 QUESTIONS FROM THE FOLLOWING TO DO ****

LESSON 9 EXERCISE 1

Write a program that receives numbers from the keyboard, stores them in an array, and reorders them and displays the results. The main function will get 5 single digit numbers from the keyboard from a function **get()** and send it to a function called **reorder().** The re- order function will exchange the last with the first, the second with the second last etc. Finally a **print()** function that will print out the contents of the array. Write the **get(), reorder()** and print functions. Using your print function, print out the contents of the array before and after you call the **reorder()** function. Call your program L9ex1.c. Hint: allocate memory for array.

variable or function	description
int* A;	pointer to array
int Size;	size of array
void get(int *a,int size);	get array elements from keyboard
void reorder(int *a, int size);	exchange array elements
void print(int* a, int size);	print out array elements on screen



LESSON 9 EXERCISE 2

Continue from Exercise 1. Ask the user how many numbers they want. Allocate an array to hold all the numbers. Then ask the user to enter all their numbers. Have the reorder function also keep track of the smallest number and the largest number in the array. The reorder function should pass the min and max numbers as output parameters. Print out the min and max values and the contents of the array. Call your program L9ex2.c.

LESSON 9 EXERCISE 3

Ask the user to enter how many numbers they want and array values from the keyboard. Add up all the elements in the array and print out the result. Print out the input array and the sum of all the elements of the array. Call your program L9ex3.c.

LESSON 9 EXERCISE 4

Add two arrays together into a third array. Call your program L9ex4.c. Ask the user to enter array values from the keyboard. Print out each input array and the final result.

LESSON 9 EXERCISE 5

Add two 3*3 two dimensional arrays into a third array. Call your program L9ex5.c Ask the user to enter array values from the keyboard. Print out each input array and the final result.

LESSON 9 EXERCISE 6

Multiply two 3*3 two dimensional arrays into a third array. Call your program L9ex6.c. Ask the user to enter array values from the keyboard. Print out each input and final arrays.

LESSON 9 EXERCISE 7

Write a program that takes a two dimensional array lets say 5 * 5 and rotates each element by the amount a user enters on the keyboard. If the user enters a positive number 2 all the elements in the array will shift right by 2. If they entered a negative number all the elements in the array will shift left. Print out the array before and after rotation. Call your program L9ex7.c.

LESSON 9 EXERCISE 8

Ask the user to enter in the keyboard how many characters they want on an output line. Ask the user to enter many lines from the keyboard or read text lines from a file. Produce an output on the screen where all the lines are a fixed width where the left and right margins are justified. Hint: use an output buffer to format the lines first before printing. Call your program L9ex8.c. Example the above lines would appear as follows or a line width of 28 characters:

```
*              Ask  the  user  to  enter  in  the
               keyboard how  many characters
               they  want  on  a  output  line.
               Ask  the  user  to  enter  many
               lines  from  the  keyboard  or
               read text lines from a file.
               Produce  an  output  on  the
               screen  where  all  the  lines
               are  a  fixed  width  where the
               left  and  right  margins  are
               justified.
```

LESSON 9 EXERCISE 9

Write a program that removes all the comments from a C program. file. Your program should be able to handle nested comments. "comments inside comments". Report any comments that do not end and nested comments that are unbalanced. Call your program L9ex9.c. Write the results to another file. An example input file would be:

```
/* lesson 9 program 2 */
#include <stdio.h>
/* test program /* nested comment */ first comment continued */
void main()
{
int num;
srand(time(NULL));  /* randomize(); */
num = rand() % 100; /* random number between 0 and 99 */
printf("%d",num);   /* print out random number /**/*/*/
num = (rand() % 100) + 1; /* random number between 1 and 100
printf("%d",num);   /* print out random number */
}
```

The result output file would be:

```
    #include <stdio.h>

    void main()

        {
        int num;
        srand(time(NULL));
        num = rand() % 100;
        printf("%d",num);

    error line 10: comment unbalanced
    error line 15: file ended but comment started line 12 not finished
```

LESSON 9 EXERCISE 10

Write a program that prints out the transpose of a matrix. Call your program L9ex10.c. For example

```
1  2  3              1  4
                  
4  5  6              2  5

                     3  6
```

LESSON 9 EXERCISE 11

Write a program that print out a triangle. Call your program L9ex11.c. The user of your program will specify the width of the base of the triangle.

```
    *
   * *
  * * *
 *  * * *
*  * * * *
```

LESSON 9 EXERCISE 12

Write a program that print out a Diamond. Call your program L9ex12.c. The user of your program will specify the maximum width of the diamond.

```
    *
   * *
  * * *
 * * * *
* * * * *
 * * * *
  * * *
   * *
    *
```

LESSON 9 EXERCISE 13

Write a program that calculates a magic square. In a magic square the elements of each row and column add up to the same value. No cheating all numbers cannot be 1. When your program runs, the user will enter the size of the square. Also write a function to test if your square is really a magic square. The user will enter the size of the square, the square is always odd. Call your class and file L9ex13.c. The steps to making a magic square are:

(1) set k to 1 and insert into the middle of the top upper line

(2) repeat until a multiple of n squares are in place:
    add one to k, move left one square and up one square and insert k

(3) add one to k

(4) move down one square from the last position and insert k.

(5) go back to step 2 until all squares are filled

6	1	8
7	5	3
2	9	4

A multiple of n squares will be 3 6 9 for a 3 * 2 square. If you go left and you are at the start of a column you have to wrap around to the last column. If you are the bottom of the row and you have to move down one row then you wrap around to the first row.

LESSON 9 EXERCISE 14

Write a program that take any number and outputs it into words For example the number 123 will be converted into **one hundred and twenty three**. Call your file L9ex14.c.

LESSON 9 EXERCISE 15

Writ a program that takes a numeric string like $+-12,564..56 and checks it for validity and correct it if certain elements are missing. Your program should output "INVALID" if the string is invalid and cannot be corrected, else the corrected output string. If a decimal point is missing it will insert the decimal point and two zeros. If there is only one digit after the decimal point it will insert the extra digits. There should be a comma before every three digits, if the comma is missing then your program should insert it. Call your file L9ex14.c.

LESSON 9 EXERCISE 16

Write a function that receives 3 character strings. A string message, a string to search for and the other string to replace if the string is found. Call your file L9ex16.c. Your function declaration would be

```
char*  replace(char* s, char*  f, char* r);
```

For example the input strings are:

s: "it is cold today"
f: "is cold"
r: "will be very hot tomorrow and"

returns: "it will be very hot tomorrow and today"

LESSON 9 EXERCISE 17

Write a program that asks the user to type in a 10 letter word with sequential repeating letters example: **aaaabbccdd.** Your program should scan the word and look for letters that repeat. Your program will report the number of the largest group of repeating letters. If there are no numbers that repeat then return the largest number found in the array. Call your file L9ex17.c.

LESSON 9 EXERCISE 18

Write a program that asks the user to type in a 10 letter word with repeating letters example: **acataterat**. Your program should scan the word and look for letters that repeat. Your program will report the number of the largest group of repeating letters. This question is similar to the preceding question except he repeating letters may be anywhere in the array and do not need to be sequential. If there are no numbers that repeat then return the largest number found in the array. Call your file L9ex18.c.

LESSON 9 EXERCISE 19

Write a program that calculates the **minimum** number of 25, 10, 5 and 1 cent coins needed to make change from 0 to 1 dollar. Call your file L9ex19.c.

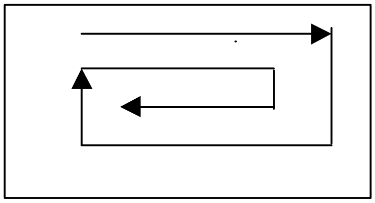
LESSON 9 EXERCISE 20

Write a program that has three arrays. One to store names, one to store addresses and one to store salaries. Each array stores 10 items. Write a program that sorts the arrays by name, address or salary on demand. This means make a menu that the uses selects for which way to sort. You can pre-initialize all arrays with values when you declare them. Hint: go through each array and print out the smallest value, use a fourth array to indicate which item you have used. Call your file L9ex20.c

LESSON 9 EXERCISE 20

Write a program that prints out any square array values as a spiral starting from any row or column. Call your file L9ex20.c.

1 2 3
4 5 6
7 8 9



1 2 3 6 9 8 7 4 5

LESSON 9 EXERCISE 21

Write a program where the user enters 3 sides of a triangle. From the entered sides determine if the triangle is **equilateral** all sides equal, **isosceles** where only two sides equal and **scalene** no sides are equal. Using can use Pythagorean theorem:

a * a = b * b + c * c.

Determine if the triangle is a **right triangle** if the largest angle is 90 degrees. An **acute triangle** if the largest angle is less than 90 degrees and an **obtuse triangle** if the largest angle is greater than 90 degrees. Finally determine if all the sides entered make a triangle where the sum all angles must add up to 180 degrees. Call your file L9ex21.c.

LESSON 9 EXERCISE 22

Write a program that can figure out what change to return for a sale. For example if someone buys an item for $4.29 and pays with a $5.00 how many quarters, dimes, nickels and pennies will be returned **?** Call your file L9ex22.c.

LESSON 8 EXERCISE 23

Make a checker board. 8 squares by 8 squares whereas each square alternates white and black.. Use the '|' char to bake a black square 8 characters wide by 3 characters high. Use the space character to make a white square. Call your file L9ex23.c.

```
||||||||
||||||||
||||||||
```

Use nested for loops. No cheating

C PROGRAMMERS GUIDE LESSON 10

File:	CGuideL10.doc
Date Started:	July 12,1998
Last Update:	Feb 29, 2002
Status:	proof

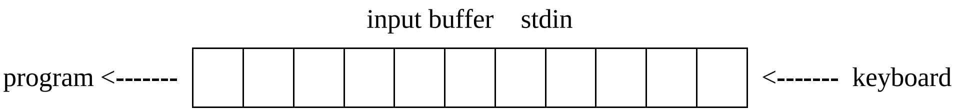
LESSON 10 C INPUT and OUTPUT STREAMS

STREAM I/O

C programming makes good use of the standard I/O system functions included in **<stdio.h>** to read input from the keyboard and files and send output to the screen or files. You should always include the appropriate system library header file at the top of your program. The input stream is called **stdin** and the output stream is called **stdout**. Streams are buffered, meaning you can type keys faster then your program can read them or send output to the screen faster than the screen can update. A buffer is a memory block that temporary holds data.

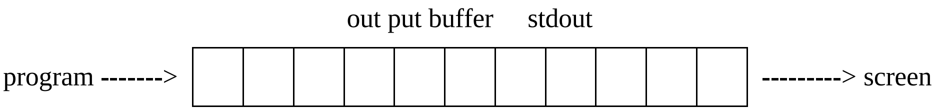
input buffer

An input buffer receives data from the keyboard or input file. The output buffer and temporary holds the data until the program reads it. The input buffer must be sufficient length to hold the incoming data.



output buffer

An output buffer receives data from a program and temporary holds it until the buffer is filled and then sends it out as one big chunk of information to an output device like a computer screen or output file.



reading and writing characters

These functions read and write single characters.

prototype	header	stream	buffer	description	example using
int getch();	<conio.h>	direct	no	returns a character from keyboard	char c = getch();
int getchar();	<stdio.h>	stdin	yes	returns character from stdin	char c = getchar();
int putch(int c);	<conio.h>	direct	no	outputs character to the screen return character written or EOF	putch(c);
int putchar(int c);	<stdio.h>	stdout	yes	outputs character on stdout return character written or EOF	putchar(c);

You can read characters in a while lop and exit if EOF. (END OF FILE)

```
while((ch=getchar()) != EOF)
      putchar(ch);
```

reading and writing strings

These functions write characters strings to the screen and read characters strings from the keyboard. You use the function **gets( )** when you want to get a complete message from the keyboard including all white space. White space is the spaces between words. **gets( )** will read all the characters until a '\n' character is found. A line on the terminal has room for 80 characters. Lines are separated by a '\n' and character strings are terminated by a '\0' EOS. **puts( )** writes a string to the output device.

prototype	header	description	returns	example using

char* gets(char *s)	<stdio.h>	get a whole string from stdin stream (keyboard)	returns the string argument or NULL if end of file or error	char s[81];
				gets(s);

int puts(char *s)	<stdio.h>	output a string to stdout stream (screen)	returns a non negative value if success or EOF if not success	char s[ ] = "hello";
				puts(s);

reading formatted strings

The **scanf** and **sscanf** functions let you read formatted strings from the keyboard or from a memory block. **scanf** is used for reading from the keyboard where **sscanf** is used for reading formatted strings stored at a memory location. **Formatted strings** allow use to specify the type of data you want to scan. A **control string** is used with **format specifiers** to specify the data type. A format specifier starts with a '%' and ends with the format specifier. Common format specifiers are "**%c**" for character, "**%d**" for integer, "**%ld**" for long, "**%f** " for float, "**%lf**" for double and "**%s**" for string. A complete list of format specifiers are listed in this section. All formatted string functions are defined in <stdio.h> Each format specifier must have a matching address argument. You may include any number of arguments. There must be matching address for each format specifier. An & must be used in front of every variable address to specify the address location of the variable. The "&" is not needed with character strings because character strings already represent an address. The format specifiers tell you want kind of data to read. Don't use width specifies with scanf like scanf("%6.2f",&d); else the user would have to type this is exactly. use scanf("%f",&d); instead.

Read a formatted string from the keyboard.

scanf	read formatted string from keyboard
prototype	int scanf(const char *format, address list);
returns	number of items scanned
example using	scanf("%d",&x); scanf("%s",str); scanf("%f",&d);

Read a formatted string from a memory location

sscanf	read formatted string from memory block
prototype	int sscanf(const char *memory, const char *format[, address, ...]);
returns	number of items scanned
example using	char message[81]; /* memory block */ sscanf(message,"%d",&x); sscanf(message,"%s",str);

Format specifiers for scanf

The format specifiers specify the data type you want to read from the input device.

format specifier	description	example	input to scan
%c	single character	scanf("%c",&ch);	'a'
%h	signed short	scanf("%h",&snums)	5
%d	signed integer	scanf("%d",&nums)	5
%i	integer	scanf("%i",&x);	5
%u	unsigned integer	scanf("%u",&numu);	56
%ld	long	scanf("%ld",&numl);	1234
%f	float	scanf(%f",&numf)	10.56
%lf	double	scanf(%ld,&davg);	2345.6734
%e	scientific notation	scanf("%e",&nume);	32e+04
%g	decimal or scientific	scanf("%g",&numg);	34.67
%s	string of characters	scanf("%s",name);	hello
%nd	number of digits to scan	scanf("%5d"&,num);	12345
%ns	number of chars to scan	scanf("%10s",name);	hellotoday
%[A-z]	string of characters between 'A' and 'z'	scanf("%[A-x]",name);	hello,today
%*c	suppress assignment read but do not assign	scanf("%c%*c",&ch);	'a' 'b'

using getchar and scanf

Using getchar() together with scanf() can be dangerous. You need to include a slash "\n" with scanf if you are using getchar() after scanf(). If you do not include '\n' in the control string then the "\n" will be left in the input buffer.

```
scanf (%d\n",&x); /* get number from keyboard */

c = getchar(); /* get character from keyboard */
```

If you do not include the "\n" then getchar will read the '\n' character instead of the character you type in and your variable will have nothing. in it.

using scanf with " %c" format specifier

You need to include a slash "\n" before the %c control format specifier when using **scanf**

```
 scanf("\n%c",&ch); /* get character from keyboard using scanf */
```

If you do not read the '\n' character first then scanf will only return the enter key and you will bot get a character. You do not need the '\n' if this is the program is just startig and this is the first character you want to read. You can also use a leading space to eat up the enter key.

```
scanf(" %c",&ch); /* get character from keyboard using scanf */
```

using fflush

Another alternative is to flush the input buffer before reading characters. Flushing the input buffer means you clear it of all characters. Some people use flush to clear the input stream before reading characters.

```
fflush(stdin); /* flush input stream */
scanf("%d",&x); /* get input from keyboard using scanf */
```

enter wrong data format using scanf

To prevent scanf() from getting stuck always use the return number of items to validate your data. You use scanf to tell you how many items you scanned. Scanf() returns the number of items successfully scanned. You use the **while** statement to test if improper data has been entered. You should use **fflush()** to clear the input stream.

```
while(scanf(%d",&x)!=1)  /* validate input */
{
   fflush(stdin); /* flush input stream */
   printf("bad input data entered/n");
}
```

writing formatted strings

The printf and sprintf functions allow you to send formatted strings to the screen or to a memory block. The formatted strings work as previously described except now the data direction is output. Here is the function prototype for the printf function that sends a formatted output to the screen.

```
int printf(const char *format, argument_list);
```

You will notice the argument_list in the function prototype declaration. This means a varying amount of argument values. You may include any number of arguments. Since a control string may have many format specifiers there must be matching argument for each format specifier. You must have a variable for every format specifier. The **printf( )** function is used to write formatted string to screen:

printf	print formatted character string to screen
prototype	int printf(const char *format, argument_list);
returns	returns number of characters printed
example using	printf("message: %s value: %d\n", str,x);

```
hello value: 5
```

The s**printf( )** function is used to write formatted strings to a memory location. This allows you to convert integers into string representation: 1234 into "1234"

sprintf	send formatted character string to a memory block
prototype	int sprintf(char *memory, const char *format, argument_list);
returns	returns number of characters printed

example using	char mem[81];
	sprintf(mem,"message: %d", x);

Format control string specifiers for printf

You use the format specifiers to specify the data type you want to read or write.

format specifier	description	example	output to screen
%c	single character	printf("%c",ch);	a
%h	signed short	printf("%h",snums);	5
%d	signed integer	printf("%d",nums);	5
%i	integer	printf("%i",x);	5
%u	unsigned integer	printf("%u",numu);	56
%ld	long	printf("%ld",numl);	1234
%f	float	printf(%f",numf);	10.56343

format specifier	description	example	output to screen
%w.nf	specify width and decimal places	printf("%3.2f",numf);	10.56
%.n	specify decimal points	printf("%.1f",numf);	10.5
%lf	double	printff(%ld,davg);	2345.6734
%e	scientific notation	printf("%e",nume);	32e+04
%g	decimal or scientific	printf("%g",numg);	34.67
%s	string of characters	printf("%s",name);	hello
%-s	left justified string char	printf("%-s",name);	hello
%nd	number of digits to scan	printf("%5d",num);	12345
%ns	number of chars to scan	printf("%10s",name);	hellotoday
%p	display a pointer address	printf("%p",num);	**3453:9098** **0x565ef43**
%0d	suppress zero	printf("%0d",num);	**0005**
% -	**left justify**	printf("%-d",num);	**55**
%x	print hexadecimal lower case	printf("%x",num);	0x34ed8f
%X	print hexadecimal upper case	printf("%X",num);	0x34ED8F

backslash character codes

The backslash codes allow you to include special characters like new line or tab in character strings.

code	description	code	description
\b	backspace	\0	end of string terminator
\n	new line	\\	backslash
\r	carriage return	\v	vertical tab
\t	horizontal tab	\a	bell
\"	double quotation mark	\o	octal constant
\'	single quotation mark	\x	hexadecimal constant

LESSON 10 EXERCISE 1

Read a line of text with many different words. Reverse all the words in the line and print to the screen.

FILE I/O

File Streams connect files to your program. File streams may be **text** or **binary**. A text stream is a sequence of characters terminated by a newline character '\n'. A text stream has all printable characters. Character translations occur in text streams. The newline \n' may be converted into a carriage return line and a newline **\r\n** line feed pair. The control 'Z' character is used to indicate the end of a text file. A **binary stream** is a sequence of bytes that represent any binary number. A binary file is a file that contains printable and non printable characters having values 0 to 255 (0 to 0xff hex). EOF is used to indicate the end of binary file. What is the difference between a binary file and a text file ?

opening files

Before you can use a file it has to be opened for **reading, writing** or **append** mode and specify if it is a **text** file or **binary** file. Append means open at the end of the file rather than at the start. You must declare a pointer to a **FILE** structure before you open a file. The FILE structure contains all the information about the opened file. When you open a file for reading or writing you must specify the **name** of file to open and the file **status mode**. The file status mode is used to specify if the file is to be opened for text, binary, reading, writing or append.

```
FILE * fp = fopen (char* filename,  char* mode);
```

Examples using **fopen()** to open files, for reading, writing and writing append:

```
FILE* fp = fopen("input.dat","r"); /* open text file for read */
FILE* fp = fopen("output.dat","w"); /* open text file for write */
FILE* fp = fopen("output.dat","a"); /* open text file for append  (at end of file )*/
```

file opening modes

File opening modes specify if the file is to be opened for text, binary read/write or append. Append means open at the end of the file rather than at the start When writing to a file using append prevents the original contents of the file from being destroyed. The **+** sign after **r+**, **w+** and **a+** means open the file for both reading and writing. **r** and **rt** are the same **t** just stands for **text**, the b in rb stands for binary.

text file open status modes			binary file open status modes	
"r"	**"rt"**	opens a text file for reading	**"rb"**	opens a binary file for reading
"w"	**"wt"**	opens a text file for writing at start of file	**"wb"**	opens a binary file for writing at start of file
"a"	**"at"**	append, open to write at end of text file	**"ab"**	append, open to write at end of binary file
r+	**r+t**	open text file for both read/write	**r+b**	open a binary file for read/write
w+	**w+t**	create a text file for read/write (destroys previous contents)	**w+b**	create a binary file for read/write (destroys previous contents)
a+	**a+t**	open or create a text file for read/write append (open at end of file)	**a+b**	open or create a binary file for read/write append (open at end of file)

Before you can read or write to a file you must test if the file has been successfully opened. If the file pointer is NULL then the file has not been opened successfully. A message must be reported to the user of your program if the file cannot be opened and the program aborted. The library function **exit(1)** found in <stdlib.h> is used to close all open streams before terminating the program.

```
FILE* fp = fopen("input.dat","r"); /* open text file for read */

/* testing if a file was successfully opened */
if(fp == NULL)

        {
        printf("the file %s cannot be opened",file_name);
        exit(1); /* close all file streams and terminate */
        }
```

checking for end of file

The **feof()** function can be used to detect when the end of file is encountered.

feof	Detects end-of-file on a stream.
prototype	int feof(FILE *stream);
returns	feof returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream and 0 if end-of-file has not been reached.
example using	if (feof(fp))break;

flush buffer

The **fflush()** function can be used to detect when the end of file is encountered.

fflush	flushes a stream (empties all characters in buffer)
prototype	**int fflush(FILE *_stream_);**
returns	returns 0 if the buffer was successfully flushed. The value 0 is also returned in cases in which the specified stream has no buffer or is open for reading only. A return value of **EOF** indicates an error.
example using	fflush(stdin);

checking for file stream error's

The **ferror()** function can be used to detect if an error has occurred in the file stream

ferror	Detects errors on stream.
prototype	int ferror(FILE *stream);
returns	ferror returns nonzero if an error was detected on the named stream.
example using	if (ferror(fp))break;

closing files

To retain all data written to a open file it must be closed. To let other programs to read a open file it must be closed. If you do not close your files data my not be written to the file properly. Always close your files.

prototype	description	example using
int **fclose**(FILE *fp);	close a file stream	fclose(fp);

sequential file access

Sequential file access is used when we want to read the file sequentially from the start to the end of the file.

prototype and example using	description	returns

int f**getc** (FILE* fp);
char c = getc(fp);
	get a character from file stream.	EOF at end of file

int f**putc** (int ch,FILE *fp);
putc(c,fp);
	write a character to file stream.	EOF at end of file

char ***fgets** (char *str,int length,FILE *fp);
fgets(msg,81,fp);
	get a string from file stream until a new line, length of characters is reached or end of file encountered	NULL if at end of file

char ***fputs** (char* str, FILE *fp);
fputs(msg,fp);
	writes a string to a file stream.	returns 0 on success

int **fscanf**
(FILE *stream, const char *format, address_list);
fscanf(fp,"%d",&x);
	read a formatted string from file stream. Convert characters strings into numbers, Skips white space.	returns number of arguments written., or **EOF** if at end-of-file,

int **fprintf**
(FILE *stream, const char *format, value_list);
fprintf(fp,"%d",&x);
	write a formatted string to file stream. Convert numbers into character strings.	returns the number of bytes output. In the event of error it returns EOF.

When using scanf you can read the and test for EOF at the same rime

```
while(fscanf("%s",buffer)!=EOF)
```

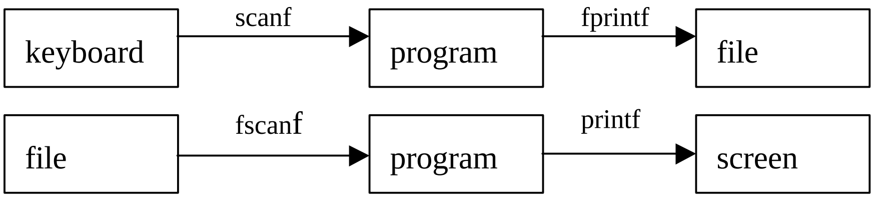
Lesson 10 Program 1

Here is an example program that open's a file for reading and another one for writing. We use **gets()** to get a message from the keyboard. We open a file for writing. We use **fputs()** for writing a line to a file. We close the file and reopen in read mode. We use **fgets()** for reading a line of the file.

```
/* this program writes a line to a file and then reads it back */.
#include <stdio.h>
#include <stdlib.h>

void main()

    {
    char line[81]; /* used to store a line read from file */
    FILE *fp; /* write and read file pointer */
    fp = fopen("test.dat","w"); /* open file for writing */

    /* check if both files open */
    if(fp == NULL)

        {
        printf("the file's cannot be opened for writing");
        exit(1); /* close all file streams  and terminate */
        }

    /* user enters message from keyboard */
    gets(line,sizeof(line);

    /* write line to output file */
    fputs(line,fp);

    fclose(fp); / *close write file */

    /* reopen in read mode */
    fp = fopen("test.dat","r"); /* open file for writing */

    /* check if both files open */
    if(fp == NULL)

        {
        printf("the file's cannot be opened for reading");
        exit(1); /* close all file streams  and terminate */
        }

    /* read line from file and print line to screen */
    if(fgets(line,sizeof(line),fp1) != NULL)

          puts(line); /* print out line contents */

    fclose(fp); /* close read file */
    }
```

LESSON 10 EXERCISE 1

Write a program that asks the user to enter 5 words from the keyboard. Open up a text file for write mode and write the words to the file. Close the file and then reopen the file for read mode. Print out the contents of the file. Use **scanf** to get information from the keyboard. Use **fscanf** to read the words from the file. You do not need to use loops. Call your data file L8ex1.dat. Use **printf** to print the words to the screen. Call your program file L10ex1.c.



main using arc and argcv

When you need to pass information to the main function them you can the argv and argc command line arguments on the command line when you execute your program. The information usually passed to a program is usually a file name that you want to read or write to. In the following command line example a program sortfiles is called to sort the file input.dat and write the sorted output to outpiut.dat.

sortfile input.dat output.dat

The main function has the **argc** and **argv** parameters to receive command line arguments. The **argc** parameter holds the count of the number of arguments in the command line and the **argv** parameter is a pointer to an array of characters strings. Each element in the array of character strings holds a command line argument. The first string character array element argv[0] holds the program name, the remaining index hold the sequential entries.

argc	3

(number of arguments)

index	value	use
argv[0]	sortfiles	execution file run
argv[1]	input.dat	input file name
argv[2]	output.dat	output file name

The standard thing to do when using command line arguments is to check if the correct number of arguments have been entered. the argc parameter is always checked first to make sure The count is equal to all the arguments entered including the program name.

```
/* program using argc and argv  */
#include <stdio.h>

/* function prototypes */
*sort_file(char *in_name, char *out_name);

/* main function to receive command line arguments */
void main(int argc, char *argv[])
{
if(argc != 3) /* check for correct number of command arguments entered */
    {
     printf(" please re-enter  as: sort  input file name   output file name");
     return;
     }
sort_file(argv[1],argv[2]); /* call sort_file  routine to sort files */
}
```

READING AND WRITING TO BINARY FILES

Data is stored as **records** in binary files. The **fread** and **fwrite** functions are used to read and write records to and from a binary file. You must specify the number of bytes in your buffer to read the record and how many times you want to read/write from each record. A record can described by a structure specifying the different **fields** in the record. A file pointer points to individual bytes in the file. If you read and write your file record by record the file pointer will point to the start of each record read or written to. The file pointer points to individual bytes in the file. As you read data from the file the file pointer increments.

fileptr --->	byte 0	start of file
	byte 1	
	byte 2	
	....	
	byte n	end of file

To read contents of a binary file:

fread	reads number of bytes and number of records from binary file into a memory block
prototype	int fread(void* memory, int num_bytes, int count, FILE* fp)
returns	On success fread returns the number of items actually read. On end-of-file or error it returns a value less than count
example using	fread (&record,sizeof(record),1,fp) ; /* read 1 block from binary file */

To write data to a binary file:

fwrite	writes number of bytes and number and records to binary file from a memory block
prototype	int fwrite(void *memory, int num_bytes, int count, FILE *FP)
returns	On successful completion fwrite returns the number of items actually written. On error it returns a value less than count.
example using	fwrite (&record,sizeof(record),1,fp); /* write 1 block to binary file */

LESSON 10 EXERCISE 2

Define a structure of your favorite data types. declare and initialize to some values. Write to a binary file then close. Reopen the binary file and read the file into the structure. print out the contents of the structure.

BINARY FILE RANDOM FILE ACCESS

Random file access allows you to read any byte of a file by seeking to its position. When a file is opened a file pointer points to the start or end of the file depending on what mode it was opened in. Seek allows you to specify where you want to read or write to the file by changing the location where the file pointer points to.

fseek	move file pointer to indicated position from specified origin
prototype:	int fseek(FILE*fp, long int num_bytes, int origin);
paramaters:	num_bytes: the number of bytes from the origin
	origin: may be at the beginning, end of or , current position of the file.
returns	returns 0 if successful
example using	fseek(fp, position, SEEK_SET); /* seek from start of record */

seek file origins

The file origin lets you specify if you want the file pointer relocation offset from the start of the file, the current location the file pointer is at or from the end of the file.

origin	name	value	explanation
beginning of file	SEEK_SET	0	absolute from start of file
current position	SEEK_CUR	1	relative to current position
end of file	SEEK_END	2	absolute from end of file

The **ftell()** function tells you the position of the file pointer.

ftell	returns the current value of the file position indicator for the opened file stream the number of bytes from the beginning of the file
prototype	long ftell(File *fp);
example using	position = ftell(fp); /* get file position */

The **rewind()** function repositions a file pointer to the beginning of the file.

rewind	reposition file pointer to start of file
prototype	void rewind(File *fp);
example using	rewind(fp); /* set file position to start of file */

The following program reads a byte from a binary file, and writes the byte back to the same file location. Notice we need **fseek** before the write and after the write. Why ?

Lesson 10 Program 2

```
/* read and write from the same location of a binary file */
#include <stdio.h>
#include <stdlib.h>

void main()

        {
        char rec[1]; /* record with 1 character */
        long pos; /* file pointer position */
        FILE *fp = fopen("input2.dat","r+b"); /* open file for read and write binary */
        if(fp == NULL) /* check if file open */
            {
            printf("the file cannot be opened");
            exit(1);
            }

        /* read and write to end of file */
        fread(rec,sizeof(rec),1,fp);
         putch(rec[0]); /* print byte read */
        pos = ftell(fp); /* get file pointer pos */
        fseek(fp,pos-sizeof(rec),SEEK_SET); / go back */
        write(rec,sizeof(rec),1,fp); /* write byte */
        fseek(fp,0,SEEK_CUR); /* seek to read pos */
        fclose(fp); /* close file */
        }
```

LESSON 10 EXERCISE 3

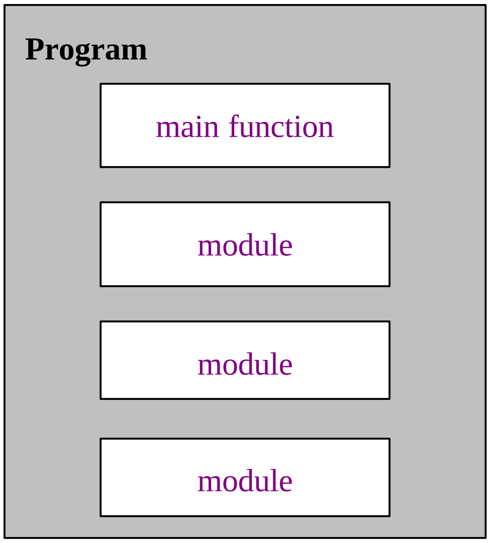
Define a structure of your favorite data types. Ask the user of your program to enter data for a structure and write to a binary file. Seek to the start of the file and print out the file contents. Ask the user to change the information in one of the structures located on the data file by asking for a structure index. Calculate the file location from which structure chosen and seek to this position. Read the structure from the file and print out. to the screen. Ask the user to enter new data for the chosen structure. Write the structure data back to the file. Verify that the new data has been written correctly by printing out contents of the file. Call your program file L10ex2.c.

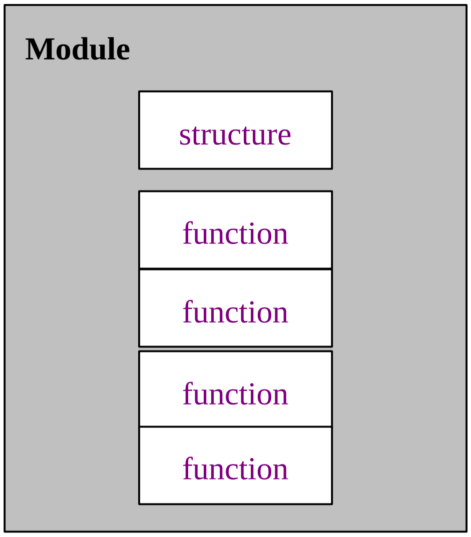
C PROGRAMMERS GUIDE LESSON 11

File:	CGuideL11.doc
Date Started:	July 12,1998
Last Update:	Feb 29, 2002
Status:	proof

LESSON 11 MODULAR PROGRAMMING I

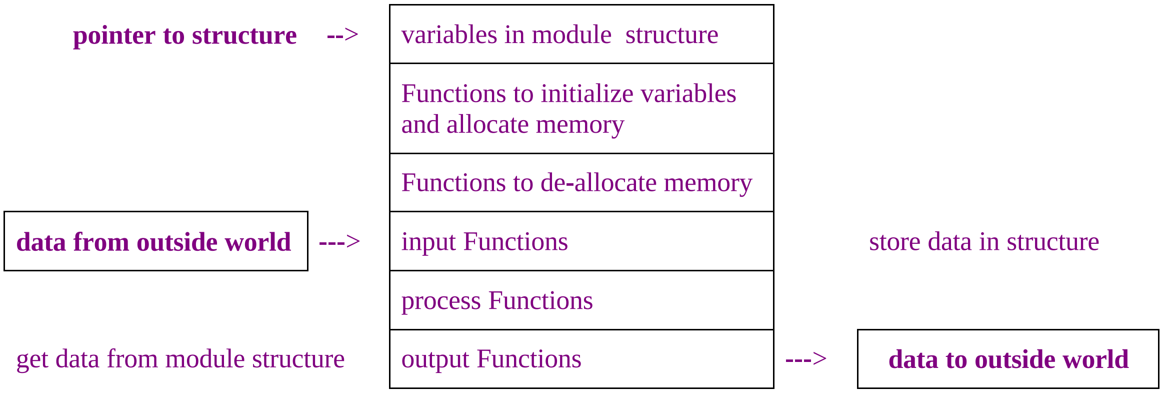
If you have a large program with many functions, then you should group functions having a specific operations together in the same location of your program or in a separate program file. These functions are said to belong to a **module**. All variables shared by the functions of the module should be declared in a structure. The structure should have the same name as the module name. The file that the module is in should also be same as the module name. A program is made up of modules and modules are made up of structures and functions. Each module has its own structure. All functions of the structure share variables belonging to the structure.





A module is a group of functions that performs a common task. An example of a module can be a group of functions to read a file. You would need a function to open a file, read the file contents, check for the end of a file, and to close a file. A module must initialize variables in the structure, allocate memory and gets data from the keyboard or a file. The variables of the module structure are used to exchange data between all functions in the module. A module must allocate memory for its variables, process the input data by doing a calculation, and produce the output on a screen or store results in a file for future use. A module should deallocate all allocated memory when it is done. Modules should have separate functions to initialize variables, allocate memory , de-allocate memory, get input data, do calculations and output data. Each function will get a pointer to the module.

module system model



pointer to structure -->

variables in module structure

Functions to initialize variables and allocate memory

Functions to de-allocate memory

data from outside world --->

input Functions

store data in structure

process Functions

get data from module structure

output Functions

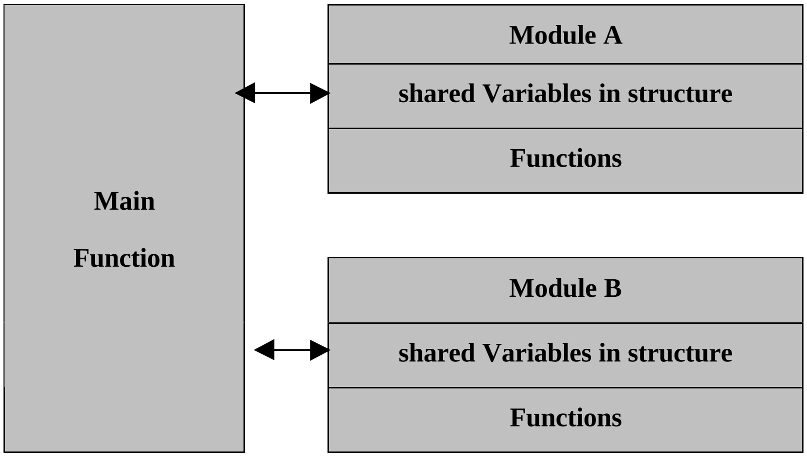
--->

data to outside world

Module variables stored in structures make good sense rather than using global variables. This way we know whom the variables belong to. If the module variables are not in a dedicated structures then other functions could access them and maybe corrupt the data. In this case the program will crash and the computer will shut down. By putting module variables in structures allow you to duplicate names, which you could not do with global variables. The **initializing functions** of the module will initialize module variables and allocate memory. The **de-allocating functions** of the module will de-allocate memory when the module is no longer needed. This is usually done when the program is ending or the module is no longer needed. The **input functions** get data from the outside world put into the module variables. The **process functions** use the module variables for calculations. The **output functions** get the data from the module variables and give to the outside world. The advantage of **input/output** functions is that the users retrieving the data do not need to know how the data is stored internally, what database they are using, what data structure they are using. They just want the data!!! The input/output functions will handle all of the overhead in managing the data. This takes the burden away from the programmer.

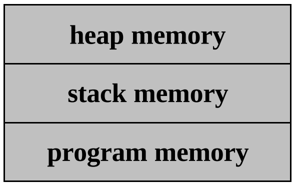
Think that your program is like a factory. Think the raw materials (as data) being delivered by delivery trucks. The roads are the communication between functions. The delivery tucks bring the raw material to the delivery department (the input functions), the factory manufacturers the products (process functions) and the delivery trucks pick up the finished product from the shipping dept (the output functions). Always think of a module as a factory with specific operation to do. Since there are many factories that do different things, then there also may be many modules that do different things. If you have to do many different things to do then you would need many different modules to do them. Think that a module is a group of functions each performing a dedicated task. By grouping together common functions together into a module, each having a specific purpose your program will be highly organized and efficient. Most programmers will have a separate file for each module. This is called modular programming. When it comes to program maintenance the programmers only have to debug the module file containing the suspect functions, not to read a 10,000 line of program code, just to find where the bug is !!!

Program Organization:

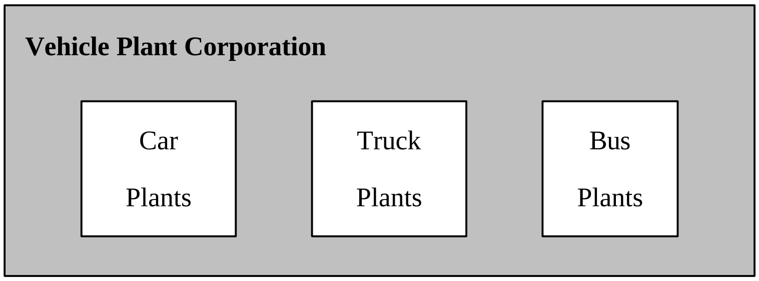


Every program has a main function. The main function is the first function to execute when a program runs. The purpose of the main function is to allocate memory for the module structures and call the various functions in the modules to perform the required program purpose. The variables in the main function are used for module structures, to transfer data between module functions and by use of the main function itself. Memory may be allocated at compile time or run time. Allocating memory at compile time is good because then you know that you will have sufficient memory when your program runs. You allocate memory at run time when you need lots of memory. When you allocate memory at compile time. the compiler will allocate memory on the program **stack**. The stack is a little bit of memory reserved for your program when it runs. The stack was designed for function calls not huge amounts of memory. Usually stack memory is insufficient. By allocating memory at run time memory is allocated from the **heap**. The heap has memory available from the whole computer. There is a lot more memory available on the heap then on the stack. When your program is loaded to run it is put into program memory.

Computer Memory:

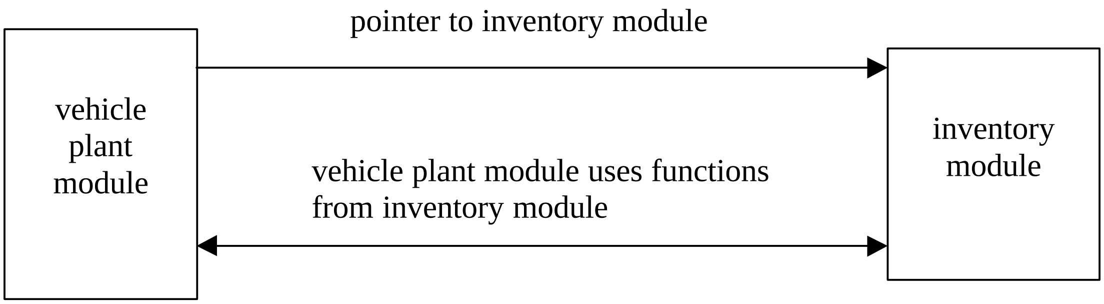


An example of modular programming is a corporation that manufactures vehicles. The corporation will manufacture cars, trucks and buses. Since this is a large corporation it has manufacturing plants worldwide. It would need many car plants, truck plants and bus plants all identified by a name. To solve this huge programming task we break all program tasks into modules. Each module would have a dedicated task. Each module will have module variables, initializing functions input functions, calculating functions and output functions. The module required would be a Vehicle Plant module and a Inventory module.



VEHICLE PLANT MODULE

The Vehicle Plant module purpose is to manufacture a specific vehicle, receive inventory and inform the shipping department that the vehicles are ready for delivery. The Vehicle Plant module will be used to manufacture cars, trucks and buses. The information about each Vehicle Plant will be kept in an structure. The Vehicle Plant Module data structure must store the plant name, plant type, plant status, number of vehicles ready and a pointer to the Inventory module data. The vehicle plant module will also call functions from the inventory module to check and get inventory.


pointer to inventory module
vehicle plant module
inventory module
vehicle plant module uses functions from inventory module

The member variables of the vehicle plant structure are as follows. The plant **name** is the name of the plant like "speedy", the plant **type** is "car", "truck" or "bus". The plant status is "operating" or "closed". numVehicles tells how many vehicles are manufactured. The Inventory module pointer inventory tells the vehicle plant module where the inventory data is. The vehicle plant structure name is called VehPlant and the pointer to the structure is called VehPlantPtr. The vehicle plant data structure is as follows:

```
typedef struct   {

                 char name[32];            /* plant name */                 /* "speedy" */

                 char type[32];            /* plant type */                 /* "car" */

                 char status[32];          /* plant status */               /* "operating" */

                 int numVehicles;          /* vehicles ready for delivery */ /* 20 */

                 InventoryPtr inventory;   /* pointer to inventory data */  /* &inventory */

                 }VehPlant,*VehPlantPtr;
```

A module to manufacture vehicles is defined as follows. The module header file is presented first before the code implementation file . The module header file contains the vehicle module structure and function prototype declarations, The code implementation file contains all the function definitions belonging to the vehicle plant module. You will notice in the header file, compiler directives that prevent a header file from being read twice.

```
#ifndef __VEHPLANT    which means if not defined 'file_name' and
#define __VEHPLANT    which means define 'file_name'
```

To make sure the vehicle plant module definition header file is included only once we use the **#ifndef** and **#define** compiler directives. #ifndef means if the label is not define and #define means define the label. If __VEHPLANT is not defined then define it and include the header file. If it is defined do not include the header file. If the header file is included more than once then multiple definition errors will occur. The header files ends with #endif to signify the end of the #ifndef block.

module definition file:	vehplant.h
module implementation file:	vehplant.c

The Vehicle Plant module header file:

```
/* vehplant.h */
/* vehicle plant module definitions */

#include "invent.h" /* include inventory module definitions */

#ifndef __VEHPLANT /* prevent multiple includes */
#define __VEHPLANT

/* vehicle plant info */
typedef struct

        {
        char name[32]; /* plant name */
        char type[32]; /* plant type */
        char status[32]; /* plant status */
        int numVehicles; /* number of vehicles ready for delivery */
        InventoryPtr inventory; /* pointer to inventory data */ }VehPlant,*VehPlantPtr;

 /* initialize vehicle plant module */
void initVehPlant
(char* name, char* type,int num, InventoryPtr invent, VehPlantPtr plant);

/* close plant */
void closeVehPlant(VehPlantPtr plant);

/* get number of engines in inventory */
int getEngines(int engines,VehPlantPtr plant);

/* get number of wheels in inventory */
int getWheels(int wheels,VehPlantPtr plant);

/* check number of engines in inventory */
int checkEngines(VehPlantPtr plant);

/* check number of wheels in inventory */
int checkWheels(VehPlantPtr plant);

/* make specified number of vehicles */
int makeVehicles(int num,VehPlantPtr plant);

/* deliver finished vehicles */
int deliverVehicles(int vehicles,VehPlantPtr plant);
```

```
/* get plant name */
char* plantName(VehPlantPtr plant);

/* get plant type */
char* plantType(VehPlantPtr plant);

/* get plant status */
char* plantStatus(VehPlantPtr plant);

/* print vehicle plants */
void printVehPlant(VehPlantPtr plant);

#endif
```

The Vehicle Plant module implementation file:

```
/* vehplant.c */

/* vehicle plant module */
#include <string.h>
#include <stdio.h>
#include "vehplant.h"
#include "invent.h"

/* initialize vehicle plant module */
void initVehPlant
(char* name, char* type,int num, InventoryPtr invent, VehPlantPtr plant)

        {
        strcpy((char *)plant->name,name);
        strcpy((char *)plant->type,type);
        strcpy((char *)plant->status,"operating");
        plant->numVehicles = num;
        plant->inventory = invent;
        }

/* close plant */
void closeVehPlant(VehPlantPtr plant)

        {
        strcpy((char *)plant->status,"closed");
        plant->numVehicles = 0;
        }

/* get number of engines in inventory */
int getEngines(int engines,VehPlantPtr plant)

        {
        return subEngines(engines,plant->inventory);
        }
```

```
/* get number of wheels in inventory */
int getWheels(int wheels,VehPlantPtr plant)

        {
        return subWheels(wheels,plant->inventory);
        }

/* check number of engines in inventory */
int checkEngines(VehPlantPtr plant)

        {
        return Engines(plant->inventory);
        }

/* check number of wheels in inventory */
int checkWheels(VehPlantPtr plant)

        {
        return Wheels(plant->inventory);
        }

/* make specified numbers of vehicles */
int makeVehicles(int num,VehPlantPtr plant)

        {
        /* get number of each inventory item */
        int numWheels = checkWheels(plant);
        int numEngines = checkEngines(plant);

        /* check if there is enough inventory to make vehicle */
        if((numWheels < num) || (numEngines < num))

                /* choose lowest number of inventory item */
                {
                if((numWheels > numEngines) && (numWheels < num))
                num = numWheels;
                else num = numEngines;
                }

        /* take out items from inventory */
        getWheels(num,plant);
        getEngines(num,plant);

        /* add the number of vehicles to the plant */
        plant->numVehicles += num;
        return num;
        }
```

```
/* deliver finished cars */
int deliverVehicles(int vehicles,VehPlantPtr plant)

        {
        /* check if there is enough cars */
        if(vehicles <= plant->numVehicles)

                plant->numVehicles -= vehicles; /* deliver vehicles */

        /* just deliver how many vehicles you have */
        else

                {
                vehicles = plant->numVehicles;
                plant->numVehicles = 0; /* all vehicles delivered */
                }

        return vehicles; /* return number of vehicles left */
        }

/* return plant name */
char* plantName(VehPlantPtr plant)

        {
        return (char *)plant->name;
        }

/* return plant type */
char* plantType(VehPlantPtr plant)

        {
        return (char *)plant->type;
        }

/* return plant status */
char* plantStatus(VehPlantPtr plant)

        {
        return (char *)plant->status;
        }

/*print out vehicle plant info */
void printVehPlant(VehPlantPtr plant)

        {
        printf("\nplant name: %s\n",plant->name);
        printf("plant type: %s\n",plant->type);
        printf("plant status: %s\n",plant->status);
        printf("number of vehicles ready: %d \n",plant->numVehicles);
        }
```

INVENTORY MODULE

The inventory module is used to keep track of inventory. There are functions to add, remove and check the levels of inventory. The inventory module data structure keeps track of how many engines and wheels the inventory holds. The structure name is called Inventory and the pointer to the Inventory structure is called InventoryPtr. The inventory data structure with data examples is as follows:

```
typedef struct    {

                  int numEngines;        /* number of engines in inventory */     /* 20 */

                  int numWheels;         /* number of engines in inventory */     /* 80 */

                  }Inventory,*InventoryPtr;
```

A module to manage Inventory is defined as follows. The module header file is presented first before the code implementation file . The module header file contains the vehicle module structure and function prototype declarations, The code implementation file contains all the function definitions belonging to the vehicle plant module. You will notice in the header file, compiler directives that prevent a header file from being read twice.

#ifndef __INVENT which means if not defined 'file_name' and
#define __INVENT which means define 'file_name'

To make sure the vehicle plant module definition header file is included only once we use the **#ifndef** and **#define** compiler directives. #ifndef means if the label is not define and #define means define the label. If __INVENT is not defined then define it and include the header file. If it is defined do not include the header file. If the header file is included more than once then multiple definition errors will occur. The header files ends with #endif to signify the end of the #ifndef block.

module definition file:	invent.h
module implementation file:	invent.c

The Inventory module header file:

```
/* invent.h */
/* Inventory module definitions: */

#ifndef __INVENT /* prevent multiple includes */
#define __INVENT
/* inventory item data structure */
typedef struct

        {
        int numEngines; /* number of engines in inventory */
        int numWheels; /* number of wheels in inventory */
        }Inventory,*InventoryPtr;
```

```
/* initialize inventory module */
void initInventory(int engines,int wheels,InventoryPtr invent);

/* add engines */
void addEngines(int engines,InventoryPtr invent);

/* add wheels */
void addWheels(int wheels,InventoryPtr invent);

/* get number of engines */
int Engines(InventoryPtr invent);

/* get number of wheels */
int Wheels(InventoryPtr invent);

/* remove engines */
int subEngines(int engines,InventoryPtr invent);

/* remove wheels */
int subWheels(int wheels,InventoryPtr invent);

/* print out inventory */
void printInventory(InventoryPtr invent);

#endif
```

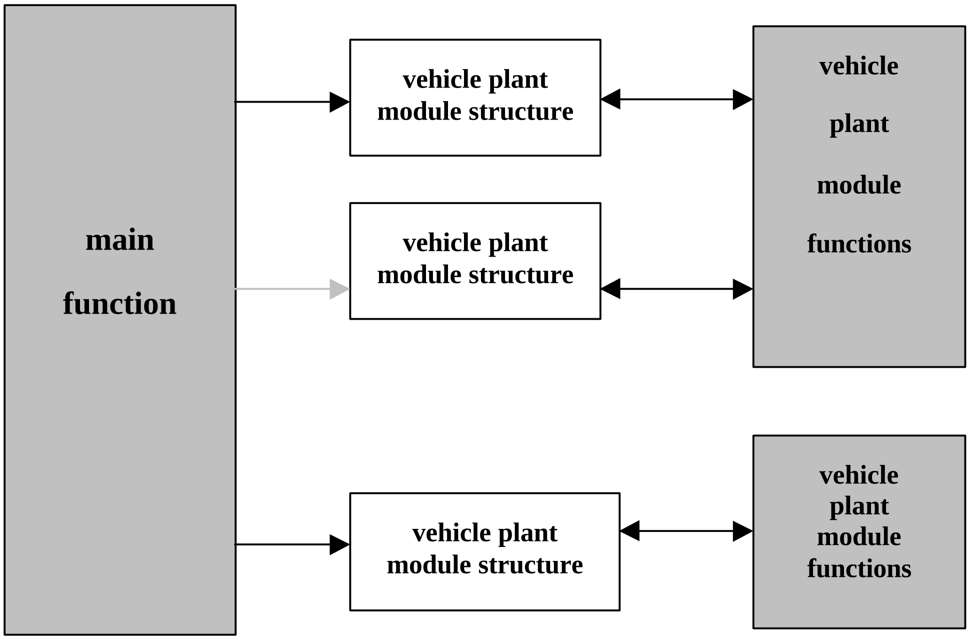
The Inventory module code implementation file:

```
/* invent.c */

/* Inventory module code implementation */

#include <stdio.h>
#include "invent.h" /* include inventory header */

/* initialize inventory module */
void initInventory(int engines,int wheels,InventoryPtr invent)

        {
        invent->numEngines = engines;
        invent->numWheels = wheels;
        }

/* add engines to inventory */
void addEngines(int engines,InventoryPtr invent)

        {
        invent->numEngines += engines;
        }

/* add wheels to inventory */
void addWheels(int wheels,InventoryPtr invent)

        {
        invent->numWheels += wheels;
        }
```

```
/* remove engines from inventory */
int subEngines(int engines,InventoryPtr invent)

        {
        /* check if thee is enough engines in inventory */
        if(engines <= invent->numEngines)
           invent->numEngines -= engines;

        /* just return number of vehicles there is */
        else invent->numEngines = 0;

        return invent->numEngines;
        }

/* remove engines from inventory */
int subWheels(int wheels,InventoryPtr invent)

        /* check if thee is enough wheels in inventory */
        {
        if(wheels <= invent->numWheels)
        invent->numWheels -= wheels;
        else
        invent->numWheels = 0;
        return invent->numWheels;
        }

/* return number of engines in inventory */
int Engines(InventoryPtr invent)

        {
        return invent->numEngines;
        }

/* return number of wheels in inventory */
int Wheels(InventoryPtr invent)

        {
        return invent->numWheels;
        }

/* print out inventory */
void printInventory(InventoryPtr invent)

        {
        printf("\nInventory:\n");
        printf(" number of engines: %d\n",invent->numEngines);
        printf(" number of wheels: %d\n",invent->numWheels);
        }
```

MAIN FUNCTION

The main program coordinates the activity of customer requests, inventory delivery and finished product shipping. The vehicle plant module does not do this, it only knows how to manufacture vehicles ! The inventory is not stored in the vehicle plant module but is managed by the Inventory module. The reason is that there can be only one inventory and it would be too much overhead for the Vehicle Plant module to manage the inventory. It is better in modular programming to have many modules all doing specific tasks than one module that does all. The vehicle plant module gets a pointer to the inventory module. Each module will receive a pointer where the structure representing module data is. Every time we need a new vehicle plant we declare a vehicle pant structure. There may be many vehicle plant structure but only one set of vehicle plant functions. Since we will only have one inventory there will only be one inventory module structure and 1 set of inventory module functions. The programmers system model is as follows.



LESSON 11 EXERCISE 1

Type in the vehicle plant header file vehplant.h .and implementation file vehplant.c. Type in the Inventory module header file invent.h and inventory module implementation file invent.c. Declare a inventory structure and initialize a inventory module with wheels and engines. Declare a vehicle plant structure and initialize a vehicle plant module. Test out all the inventory and vehicle plant module functions. add make and deliver vehicle's, print out inventory and vehicle plant module information. Put all your work in separate files as indicated in the following table. Call your main program file L11ex1.c.

module/function	module name	definition	implementation	execution
Vehicle Plant	VehPlant	vehplant.h	vehplant.c	
Inventory	Inventory	invent.h	invent.c	
Main		defs.h	L11ex1.c	L11ex1.exe

DEFINITON FILE

You can define common constants in a definition header file called defs.h

```
 /* defs.h */

#define TRUE 1
#define FALSE 0
```

C PROGRAMMERS GUIDE LESSON 12

File:	CguideL12.doc
Date Started:	July 12,1998
Last Update:	Feb 29, 2002
Status:	proof

LESSON 12 MODULAR PROGRAMMING II

ARRAY OF VEHICLE PLANT STRUCTURES

All vehicle plants information is kept in a structure. Since we will have many vehicle plants we would need an **array** of Vehicle Plant structures. In your main function you make an array of vehicle plant structures. Since this is a large corporation you will have many car, truck and bus plants. You can keep track of each plant by its plant name. Each structure will have a unique name, since we have an array of structures the index of the array will point to the data for a particular vehicle plant.

plants ----->

0	data structure for car plant "speedy"
1	data structure for truck plant "orion"
2	data structure for bud plant "glider"
3	data structure for car plant "economy"
4	data structure for truck plant "runner"

array of data structures holding information about each vehicle plant

declaring an array of structures

You may declare an array of structure as a **variable** or as a **pointer**. The pointer method is better then you will be able to allocate memory for your array of structures in run time.

To declare an array of structures as a **variable** :

```
#define MAX_PLANTS 50

VehPlant plants[MAX_PLANTS];
```

You access the array of structures directly by index to put data into each structure:

```
strcpy(plants[numVehiclePlants].name, name);
strcpy(plants[numVehiclePlants].type, type);
strcpy(plants[numVehiclePlants].status, status);
plants[numVehiclePlants].numVehicles = numVehicles ;
plants[numVehiclePlants].inventory = invent;
```

You can also declare a **pointer** to an array of structures by using malloc:

```
        VehplantPtr plants = (VehPlantPtr) malloc ( sizeof (VehPlant) * MAX_PLANTS);
```

To access each individual item of the array of structure pointed to by pointer plants:

```
        strcpy(plants[numVehiclePlants]->name, name);
        strcpy(plants[numVehiclePlants]->type, type);
        strcpy(plants[numVehiclePlants]->status, status);
        plants[numVehiclePlants]->numVehicles = numVehicles ;
        plants[numVehiclePlants]->inventory = invent;
```

To access the vehicle plant structure data you get a pointer to the address of the structure.

```
        VehPlantPtr plant = &plants[index];
```

The "&" gives you the address memory location where the structure is in the array of structures.

searching for a plant in the array of structures

Each vehicle plant will have a **unique name** to identify it like "speedy". You will need a **search() function** to search each vehicle plant structure for its name. For any vehicle plant operation you will need to look up the name for the plant, get the index of the array of structure where the plant is and get a pointer to the structure. Once you got a pointer to the vehicle plant structure you will pass the pointer to the vehicle plant functions. The array of vehicle plant structures, gives you the flexibility to add more car, truck plants and bus plants. Here is the search function.

```
/* search for vehicle plant by name */
VehPantPtr search(char* name)
      {
      VehPantPtr plant = NULL; /* declare pointer to plant structure */

        /* loop through all plants */
        for(i = 0; i < numPlants; i++)

                {
                /* check if the plant name in the array is equal to the name you want */
                if(strcmp(plants[i].name,name)==0)

                        {
                        plant = &plants[i]; /* get pointer to plant data */
                        break;
                        }

                return plant; /* return pointer to plant */
                }

        }
```

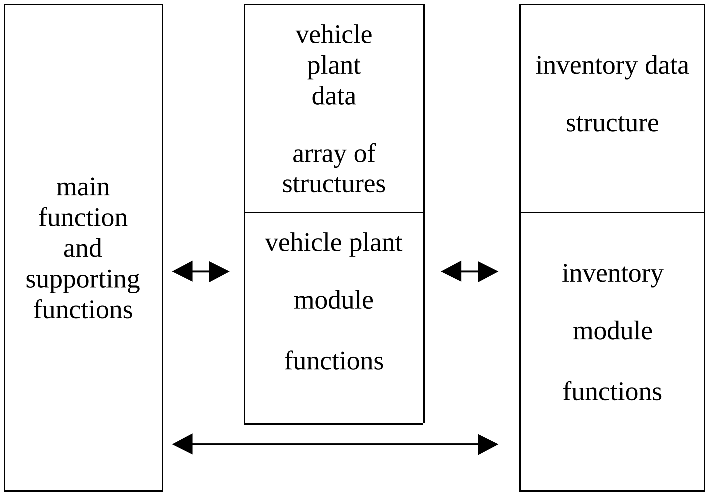
delivering vehicles

You will also need a function to deliver vehicles when its time to deliver vehicles to the showrooms. All we have to do is the following, is to go through the table and calling functions to do all the work for them You may deliver vehicles from individual plants by searching for the plant name or deliver vehicles by type by going through the array looking for matching plant types.

```
/* deliver by type */
void deliver(char* type)

        {
        /* go through plants */
        for (int i = 0; i > numVehiclePlants; i++)

                {
                /* only deliver cars */
                if(strcmp(plants[i].type,type)==0)

                        {
                        plant = &plants[i]; /* get pointer to plant data */
                        deliverPlant(plant); /* deliver plant */
                        }

                }

        }
```

Our programming system model Organization:

We now present our programming system model. Programming in modules and making block diagrams makes our programming task easier to accomplish. The block diagram gives us an overview of all the dedicated modules. By using separate programming modules, this make our programming project easier to understand and organized. By breaking up each section into separate modules our programming task is much easier. Each programmer can work on a dedicated module independently, and then combine all modules together into a finished product.

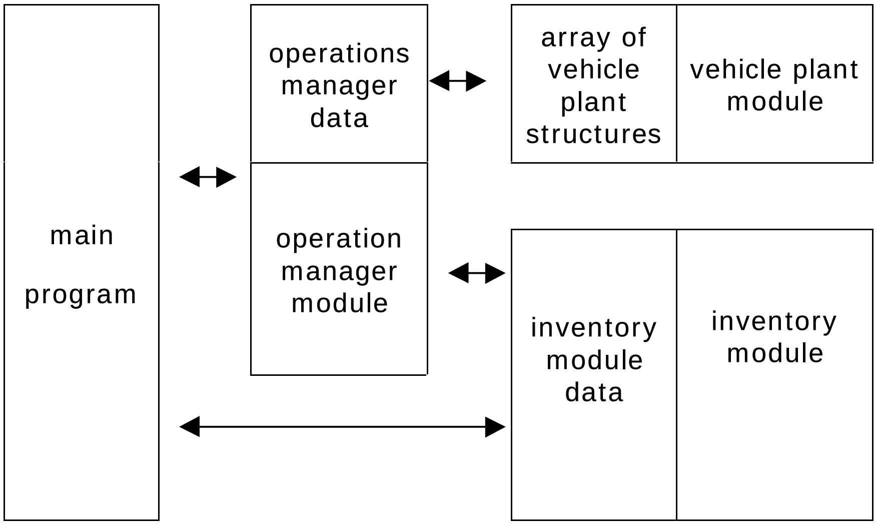


OPERATIONAL MANAGER MODULE

Some one has to keep track of which vehicle plant gets what inventory, gets orders, delivers vehicles, keep track of inventory etc. Why not have the Operations Manager module to do this ??? The Operations Manager module could hold the Vehicle Plant table, keep track of how many Car plants, Truck plants and Bus plants it has and communicate to the outside world through functions to add Vehicle Plants, remove Vehicle Plants (non profitable ones) etc. The competition will be kept out of he dark. The Inventory module will be responsible for managing the inventory. The Vehicle Plant module is responsible to manufacture vehicles!!! The job of the Operations Manager module is to manage Vehicle Plant and Inventory modules. The Operation Module is the interface to the main program and the Vehicle Plant Module. The added benefit of the Operational Manager module is that all the supporting functions of the main function will now be contained in a separate module. This will make your program very organized. The Operational Manager structure will hold the array of vehicle plant structures, the maximum number of plants and how many plants are in the structure.

```
typedef struct

{
VehplantPtr * plants;  /* pointer to an array of structures */
int maxPlants; /* max number of vehicle plants */
int numPlants; /* number of vehicle plants in structure */
}OpsMgr,OpsMgrPtr;
```

Our new programming system model with the added Operational Manager module:



LESSON 12 EXERCISE 1

Write the Operations Manager Module to add/remove vehicle Plants. The Operational Manager module structure will hold the array of vehicle plant structures. The Operational manager module needs methods to add/remove vehicle plants, order vehicles, deliver vehicles and to print vehicle plants. You will need the search function by plant name to find out which structure holds the plant information. The main function should only access functions of the Operations Manager module. It cannot access the Vehicle Plant and Inventory modules directly. You will need the vehicle plant module and inventory module. Write the main program to declare and initialize Inventory module data and Operation Manager module. The main program will also tell the Operational Manager module to add plants, delete plants, add inventory, make some cars, trucks buses and deliver finished products etc. You will need pant status of "operating" and "closed". Try to test for all conditions, especially to order vehicles from a non existing plant !!! Put all your work in separate files as indicated in the following table. Call your main program file L12ex1.c.

module/function	module name	definition	implementation	execution
Vehicle Plant	VehPlant	vehplant.h	vehplant.c	
Inventory	Inventory	invent.h	invent.c	
Operations Manager	OpsMgr	opsmgr.h	opsmgr.c	
Main		defs.h	L12ex1.c	L12ex1.exe

building an interactive interface

Now tat you got your system working you need to add an interactive interface. There are two approaches to take. You can have a selection menu where people select what operation they want to do.

```
Super Corporation Menu
===================

(1) initialize inventory
(2) initialize operational manager
(3) add wheels
(4) add engines
(5) report inventory
(6) add plant
(7) report plants
(8) report plant
(9) order vehicles
(8) deliver vehicles by plant name
(9) deliver vehicles by plant type
(10) close plant

Make your selection now dude:
```

You can have a file with commands written on it to add plants, deliver vehicles etc, or you can have both. You probably want both. The menu can be used testing and demonstration and the file can be used to run the program automatically once all the bugs have been worked out. Using a file relieves the user of your program to type in the same things all the time.

LESSON 12 EXERCISE 2

Add an interactive interface to your code of Exercise 1. The main program will have a menu to receive commands from the keyboard, and to read commands from a data file. The data file method is better suited when debugging your program. The keyboard method is better for testing your program. Either method is good to run your program. You can make a menu so that the user can select commands. For each command selected the user will be prompted to supply additional information. Call your main program file L12ex2.c. Possible command format for your program could be:

"init" "inventory" <initial number of engines> <initial number of wheels>

"init" "opsmgr" <max plants>

"add" "inventory" <engines> <number of engines>

"add" "inventory" <wheels> <number of wheels>

"report" "inventory"

"add" "plant" >plant name> <plant type><initial number of vehicles>

"order" <plant name> <number of vehicles>

"deliver" "plant" <plant name> <number of vehicles>

"deliver" "plants" <plant type>

"report" "plants"

"report" "plant" <plant name>

"close" "plant"

"exit"

An example file would be:

```
init inventory 20 10
init opsmgr 50
add inventory engines 20
add inventory wheels 40
report inventory
add plant speedy car 15
add plant glider truck 25
deliver plant speedy 20
deliver plants car 100
report plants
report plant speedy
close speedy
exit
```

You can put all your commands in a data file and use input, output or input/output indirection:

```
L12ex2 < test.dat        L12ex2 > out.dat        L12ex2 < test.dat > out.dat
```

C PROGRAMMERS GUIDE LESSON 13

File:	Cguide13.doc
Date Started:	July 24,1998
Last Update:	Feb 29, 2002
Status:	proof

C COURSE PROJECT PROJECTS = WORK = LEARNING = FUN

Continue from last lesson Exercise 1 and 2. Expand the inventory class. Design a structure for inventory items called **InventoryItem**. Make an array of InventoryItem structure objects in your inventory module to hold information about individual stock items. Your inventory module should receive the max number of inventory items from the user. You will need to modify the add search, retrieve, and fill stock items functions in your inventory module. Have one routine to handle all items, rather then separate routines for each separate items. You should store your inventory stock on a file. When the program starts up all inventory items should be read from the file. When the program finishes all inventory items should be written to the file. Call your inventory data file **"invent.dat"**. You should search on the part name or part number to find inventory items.

The InventoryItem structure will be used to store part name, number, quantity and back orders. When the program starts up all inventory items should be read from a file and stored in the InventoryItem structures. An example inventory item would be:

part name	"engine"
part number	1000765
quantity	5
back order	**0**

You will also need a Vehicle module so that the vehicle plant class knows how to make the vehicles. Call your vehicle module Vehicle. The Vehicle module store's all the information about the vehicle. The Operational Manager module will store each vehicle module in an array. When the vehicle needs to be manufactured, you need to give the VehPlant class a pointer to the corresponding Vehicle module stored in the array. You will also need functions to read and write vehicle info to I/O and file streams. When the program starts up all vehicle information should be read from a file and stored in the Operational Manager class. Call your vehicle data file **"vehicles.dat".** Each Vehicle will have a 2 dimensional array of parts indexed to the inventory by part name or part number and the quantity need to build the vehicle.

An example vehicle data would be:

vehicle name	"speedy"		
vehicle type	"car"		
number of parts	15		
2 dimensional array of parts required to build a vehicle	---->	**part name or part number**	**quantity required**
		engines	4
		doors	1
		wheels	4
		etc	etc

The main program will still create the Inventory and Operational Manager modules, receive commands from the keyboard, file or both, and call the functions from the Operational Manager module. The Operations Manager module will still be responsible for adding vehicle plants, removing vehicle plants, receive orders, and cancel orders. The added task for the Operational Manager module is to keep an array of vehicle's that need to be manufactured. Store all your Vehicle plants on a file. When the program finishes all Vehicle Plant data should be written to a file. When the program starts up all Vehicle Plants should be read from the file. Call the Vehicle Plant data file **"vehplants.dat".** As each transaction is processed, a day to day operation output log file should be produced The output log file would show how much inventory was received and how many cars, trucks and buses were produced. Call your log file **"log.dat"**

You must test for and show all possible conditions. For example: an order was received to produce 500 cars and there were only 200 engines and the car plant was closed down. Reports should tell how many car plants there are, how many orders that are, how many cars completed or in the process of being completed. When adding parts to a vehicle make sure that the part is an inventory item.

Use the interactive interface of the last Lesson of Exercises 1 and 2 . The main program will have a menu or receive commands from the keyboard and from a data file. By using both method your command decoder will accept menu commands from the user and file commands. Using a data file is good when debugging your program. Using a keyboard menu is good for testing and demonstrating your program. You will need to change the command language slightly to accommodate the inventory items. You need to search on inventory items by part name or by part number.

Possible command format for the input command file read by your program could be:

```
"init" "inventory" <max inventory items>
"init" "opsmgr" <max plants>
"add" "inventory" <part name> <part number><quanity>
"report" "inventory"
"report" "inventoryItem" <part name>
"report" "inventoryItem" <part number>
"add" "plant" <plant name> <plant type> <initial number of vehicles>
"add" "vehicle" <vehicle name> <vehicle type>
"add" "vehiclePart" <vehicle name> <part name> <quantity required>
"report" "vehicles"
"report" "vehiclePart" <vehicle name> <part name>
"report" "vehicleParts" <vehicle name>
"order" "plant" <number of vehicles>
"deliver" "plant" <number of vehicles>
"report" "plants"
"report" "plant" <plant name>
"close" <plant name>
"exit"
```

An example file would be:

```
init inventory 20 10
init opsmgr 50
add inventoryItem engines 234543 20
add inventoryItem wheels 234324 40
add inventoryItem seats 3454233 23
report inventoryItem 234543
report inventoryItem seats
add plant speedy car 15
add plant glider truck 25
add vehicle speedy car
report vehicle speedy
add vehiclePart speedy seat 4
report vehiclePart speedy seat
report vehicle speedy
order speedy 5
deliver plant speedy 20
report plants
report plant speedy
close speedy
exit
```

You can put all your commands in a data file and use input indirection:

```
assign1 < test.dat
```

You can also use output indirection to store the output screen results on a file.

```
assign1 < test.dat > out.dat
```

Modules and Functions:

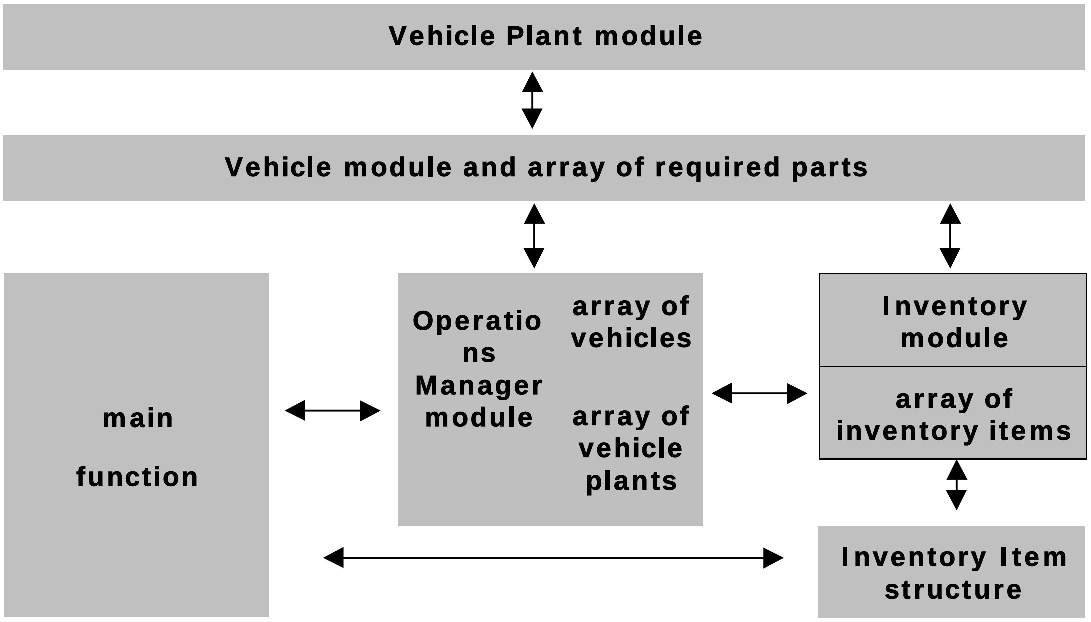
Put all your work in separate files as indicated in the following table. Call your project assign1.

module or function	definition	implementation	execution
VehiclePlant	vehplant.h	vehplant.c	
Vehicle	vehicle.h	vehicle.c	
Inventory	invent.h	invent.c	
InventItem	item.h	item.c	
OpsManager	opsmgr.h	opsmgr.c	
main	defs.h	assign1.c	assign1.exe

Files:

file purpose	file name	direction
transaction	trans.dat	read
vehicle plants	vehplants.dat	read / write
inventory	invent.dat	read / write
vehicles	vehicles.dat	read
day to day operation	log.dat	write

Program System Model:



Hints on completing the assignment:

1. Use **fscanf** and **fprintf** for reading and writing files.
   You may use **fwrite** and **fread** for vehicle and inventory data files.

2. Keep organized as possible. get and test 1 module before working on the next

3. Use the main function as a test driver to test your modules

4. Use **malloc** to allocate memory for your data structure arrays

Marking Scheme

CONDITION	RESULT	GRADE
Program crashes	retry	**R**
Program works	pass	**P**
Program is impressive	good	**G**
program is ingenious	excellent	**E**

If your program crashes then you must fix it and resubmit your assignment.

YOU MUST HAVE A WORKING PROGRAM TO COMPLETE THIS PROJECT

IMPORTANT

You should use all the material in all the lessons to do this assignment. If you do not know how to do something or have to use additional books or references to do the questions or exercises, please let us know immediately. We want to have all the required information in our lessons. By letting us know we can add the required information to the lessons. The lessons are updated on a daily bases. We call our lessons the "living lessons". Please let us keep our lessons alive.

E-Mail all typos, unclear test, and additional information required to:

courses@cstutoring.com

E-Mail all attached files of your completed project to: ($35 Charge)

students@cstutoring.com