

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**.

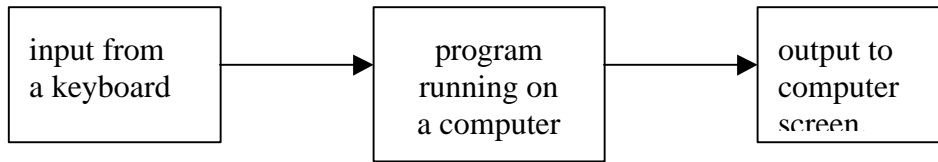
LESSONS AND ASSIGNMENTS

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.

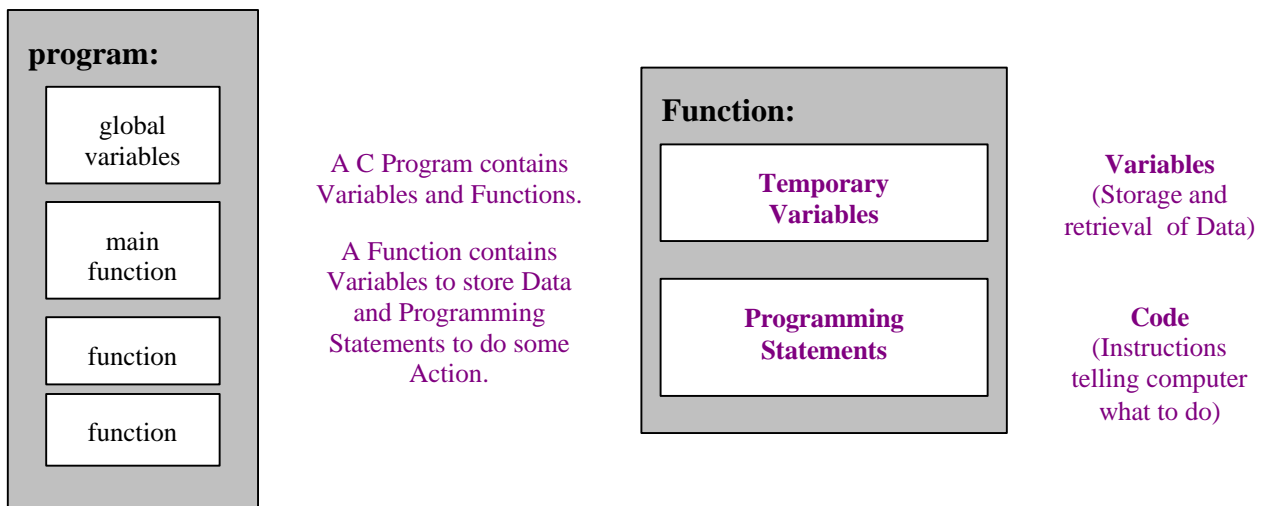
LESSON 1 C PROGRAMMING COMPONENTS AND VARIABLES

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.



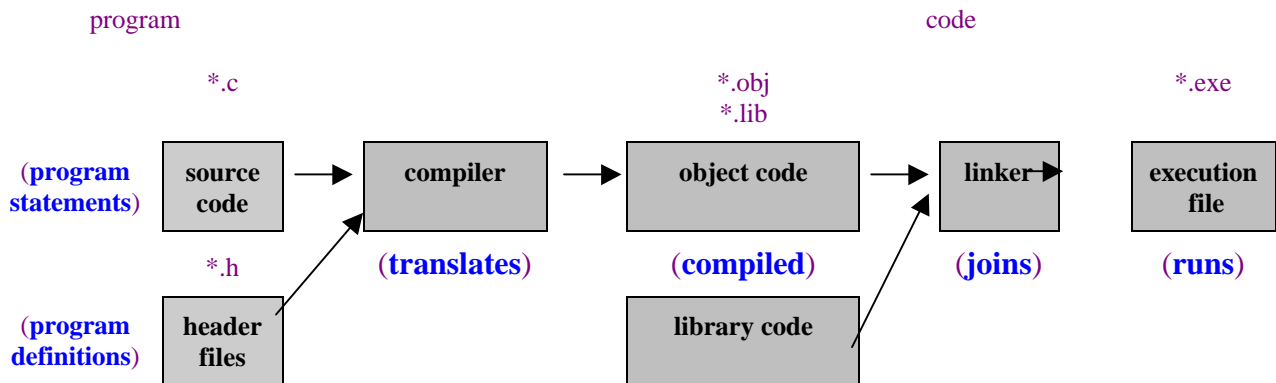
A program must be written in a computer language before it can run on the computer. A C program is written as words representing **variables** and **programming** statements grouped together in a **function**. Variables are used to store and retrieve **data** values and programming statements are **instructions** 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.



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 files 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.

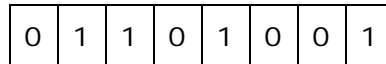
character: 'A' **numeric:** 9 **decimal:** 10.5 **string:** "hello"

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.

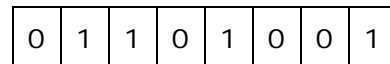
1 byte is 8 bits



bits are 1 and 0's

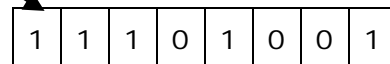
Numbers may use from 1 to many bytes. Groups of memory bytes are used to represent numbers. The smallest to largest number a data type can represent is known as the **range**. Small range numbers use few bytes of memory where large range numbers use many bytes of memory. Numbers may be **unsigned** or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

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



sign bit

Signed **Negative** numbers start with a 1



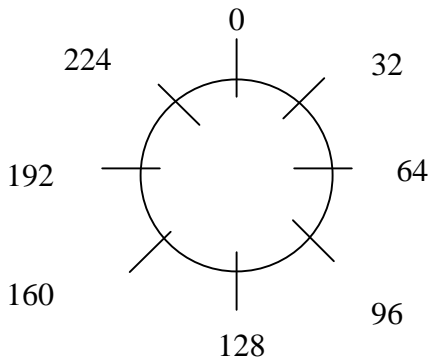
For unsigned numbers, it does not matter since the number will always be positive grater or equal to zero. It is important now to understand how signed and unsigned numbers are represented in computer memory. The same binary numbers are used to represent negative or positive numbers. It is just how they 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

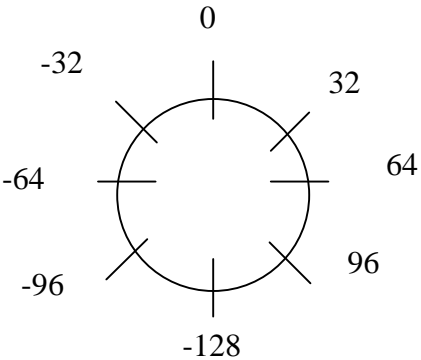
The **range** of a number tells us the smallest number and the largest number a memory can represent. Two types of numbers:

(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 **unsigned** 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 $(2^N \text{ bits}) - 1$	
To calculate the range of a unsigned number for $N = 8$:	
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	1.2e-38 to 3.4e38	
double	8	64	3456.76545 2.3456e156	2.2e308 to 1.8e308	

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)	$(-1)^{\text{sign}} * 2^{\text{exponent}-127} * (\text{mantissa})$
-	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)	$(-1)^{\text{sign}} * 2^{(\text{exponent}-127)} * (\text{mantissa})$
-	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 bytes used by the variable. Some variables take 2 bytes, 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.

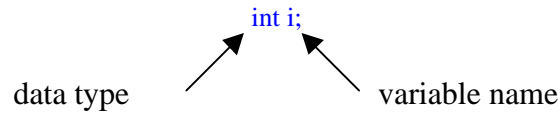
The variable x represents the memory location where the value 5 is stored in the computer memory

	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

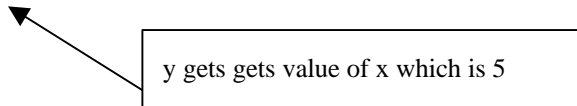
Variables may be initialized when they are declared. This would be like opening a bank account with an initial deposit. A variable is initialized when it is declared by using an **expression**. An expression represents a calculated value. An expression may be a constant or an another variable. If the 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 */*



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

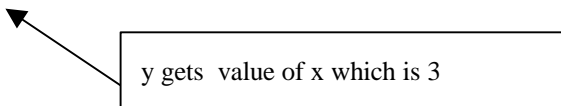
An assignment statements allow you to assign data values to a variable name by using the assignment operator "=". An assignment statement is like putting and taking money out of a bank account that was previously opened. Before an assignment statement can be used, the variable must be declared. You do 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)



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 */
```

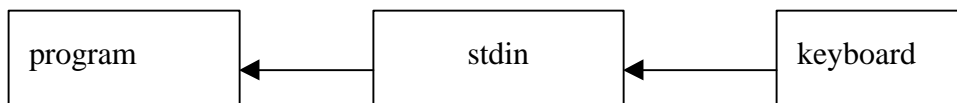
address	name	value
1013	a	10
1015	b	10
1017	c	10

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**.

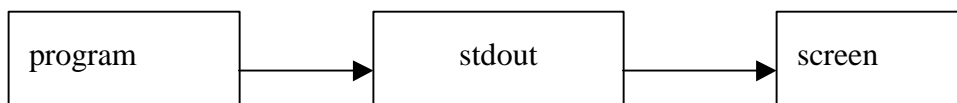
input/output streams

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

Format Specifier

variable value
to print out

In the above example the value of **x** is inserted where the **%d** format specifier 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 */
```

message to
print out

Format Specifier

variable value
to print out

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 */
```

Format Specifier

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

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

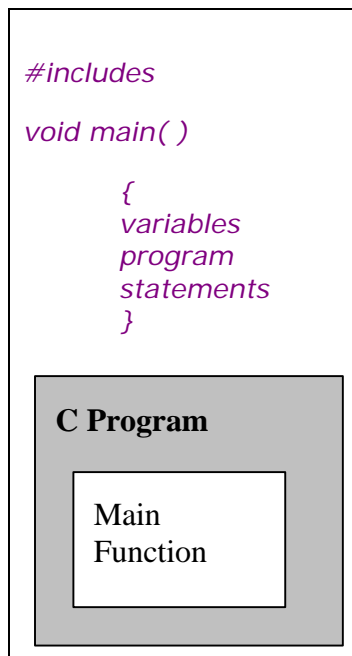
```
scanf(" %c",&ch); /* last character to read */
```

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.



```

/* 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 than the variable name.

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

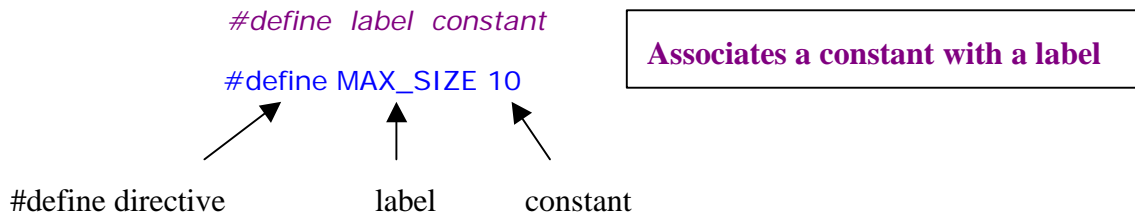
Print out the size in bytes of each variable in the above exercise.

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**.



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
```

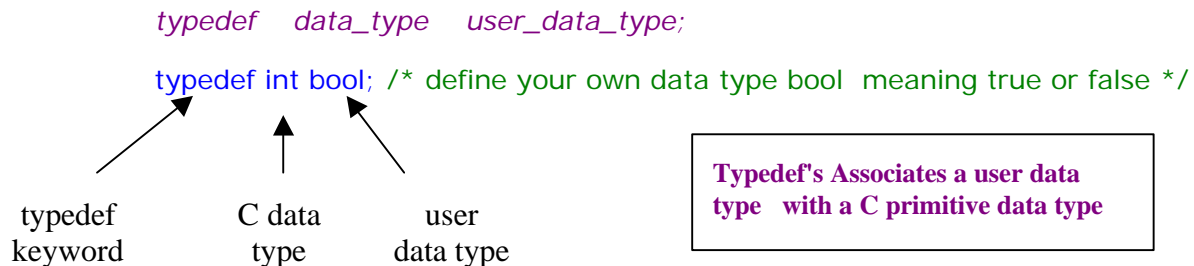
Typedef's

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.



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 than 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 represented 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.

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

Associates a sequence of labels with a common name

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.

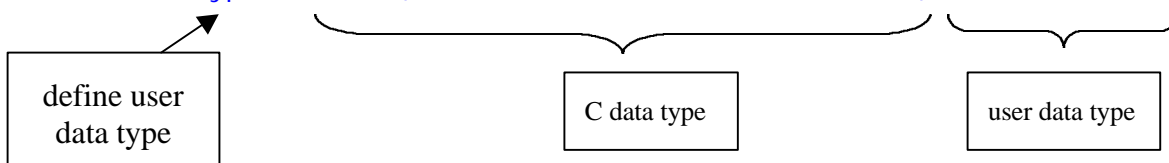
```
int weekday = 0; /* compiler representation */
```

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 */

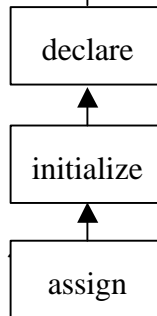
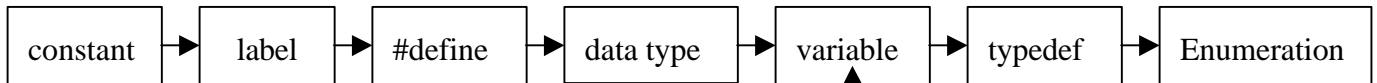
    day = TUE; /* assign Tuesday to day */
    printf("Today is: %d\n",day); /* print out day */
}
```

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 people's 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 preceding the variable or value you want to force.

`new_data_type = (typecast) original_data_type.`

`int x = (int) d;`

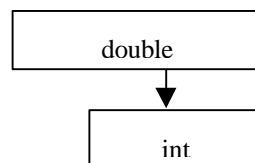
↑ ↑ ↑

new (int) typecast original (double)

(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 lose some of the data. In this case we lose 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.

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 exercises to:

students@cstutoring.com

This assignment is copyright (C) 1998-2002 by The Computer Science Tutoring Center "cstutoring"
This document is not to be copied or reproduced in any form. For use of student only