Welcome to Week 2!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 2 folder**.
- Read **Chapters 3 and 4** in the latest edition of the textbook (chapters 4 and 5 in earlier editions).
- Begin **Quiz 2**. It is due Sunday at midnight.
- Begin **Assignment 1**. It is due Sunday at midnight.

Recommended (optional) activities

- Attend **chat** this Thursday night, from 8:00 pm - 9:00 pm Eastern Time. Although chat participation is optional, it is highly recommended.
- Post any questions you might have on this week's topic in the **Week 2 Discussion Forum** located in the course **Discussion Board**. Please ask as many questions as needed, and don't hesitate to answer one-another's questions.
- Try out the various **code examples** in the lecture notes. Feel free to modify them and conduct your own **"What ifs"**.

"Mr Scoot, I suspect data is not all created the same ... how does C differentiate between all those data types?"

--- Nyota Arugula
Communications Officer, USS Enterprise

Variables

In computer science, a **variable** is a data storage location whose contents can change during the course of a program's execution. A variable has a symbolic **name**, a **location** where it exists in memory, a specific **size** in that memory location (in bytes), and some **contents** (i.e., a value) that is stored at that memory location.

For example, let's say I have a variable name whose type is a simple integer called **age**, that has been set to a numerical value of **29**. Think of **integers** as whole numbers, like -5, 0, and 100.

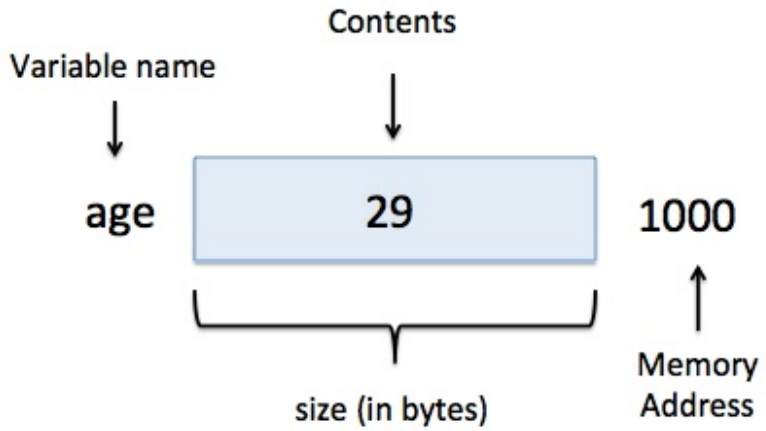
```
int age;    /* defines a variable of type integer that is called age */

age = 29  /* sets the contents of the age variable to 29 */
```

If you wish, you could also combine the two statements:

```
int age = 29;    /* Declares the variable age to be of type integer and initializes it to 29 */
```

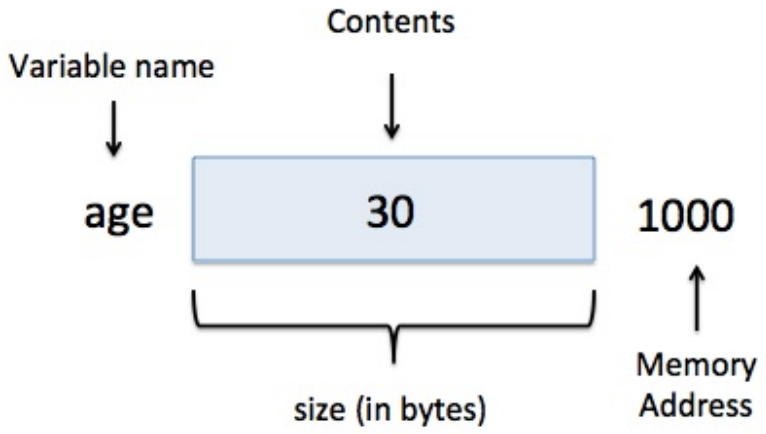
The variable has a **name** (**age**), a **type** (in this case, Integer), **contents** that contain a value (29), a **size** (let's assume an integer is 4 bytes, and it is stored at some memory **location** (let's say location 1000). To illustrate it clearer, it could be shown as:



If you then changed **age** to **30** later on in your program, the contents would then be updated.

```
age = 30;   /* changes the contents of age to a value of 30 */
```

Note that the variable **name**, **size**, and **location** remains the same ... only the **contents** are changed.



Variable Names

Most programming languages have very specific rules on what a variable name can be called, and C is no exception.

1st Character	2nd Character
A thru Z or a thru z	A thru Z, a thru z, _ (underscore), 0 thru 9

Examples of valid and invalid variables:

Valid	Invalid	Why Invalid
Int	int	int is a Reserved word in C, and while C is case sensitive, its a bad idea to also have a variable named Int, which while valid, is only confusing and prone to error.
abc	_abc	Starts with an underscore
A24Z	2A4Z	Starts with a number
Pay_Rate	&Pay_Rate	Starts with an & character

The **first character** may be upper or lower case Alphabetic. The **second and subsequent characters** may be upper or lower case Alphabetic or numbers 0 thru 9 or and underscore character.

In reality, you can start a variable in C with an underscore, but custom reserves those variable names for code used with operating systems, such as UNIX.

Some computers only examine the first 8 characters to differentiate names, but most modern computers can handle very large unique names.

C is case sensitive. **Age** is not the same as **age**, they are two different variables.

Be very careful with reserved words that are used specifically by the compiler. **Int** is not equal to **int**. Never use reserved identifiers such as **int** (see the Data Types section below for some reserved words)

Variable Naming Conventions

In C, most all functions and reserved words (if, while, do, ...) of the language are in lower case. As for naming variables in your program, its important to be consistent. Pick a particular standard, and try to stick with it. Some examples of variable naming conventions:

For example, in our homework this week, you'll need to come up with a variable name to represent the **gross pay** for an employee. Here are some suggestions that are acceptable in C. Most of these rules are good for most programming languages, but note that some languages have specific naming standards for various parts of the language, Java comes to mind.

Example	Type	Comments
gross_pay	Lower case	Use underscores between words
Gross_Pay	Init upper case	Separate each word with Upper Case and an Underscore
GrossPay	Upper Camel Case	The first letter or each word is capitalized
grossPay	Lower Camel Case	First word is lower case, and first letters of subsequent words are in upper case

Two things I would not do (or accept) for variable names:

1) Don't use all upper case, that is normally reserved for constants (we'll cover that in an upcoming lecture note this week)

```
GROSS_PAY;     /* bad, normally all upper case reserved for constants or macros */
```

2) If multiple words, don't use all lower case letters if your variable represents multiple words

```
grosspay;   /* bad, you may or may not be able to see this is two words in the variable name meaning */
```

Summary of Variable Naming Rules in C

The **first character** must be upper or lower case Alphabetic

The **second and subsequent characters** may be upper or lower case Alphabetic or numbers 0 through 9 or and underscore character.

Some computers only examine the first 8 characters to differentiate names.

C is **case sensitive**. **Int** is not equal to **int**. Never use reserved identifiers such as **int**.

PITFALL

It is legal to use the **_** as the first character but custom reserves it for use in operating system code. Please do not create variable names that start with an underscore in my class, and generally it is not a cool thing to do in the real world as well.

Video - C Trek Variables

Lieutenant Arugula has decided to learn about computer programming and asks Chief Engineer Montgomery Scoot for help with understanding the concept of a variable and all the different types available in the C Programming Language. Mr. Scoot decides to reference a real life example of his prized Romulan Ale collection to help the Lieutenant better understand this concept.

Click the image below to watch the video.

Video - General C Program Overview

Here is a quick video that both introduces and illustrates most of the concepts covered this week, in particular: Variables, Variable Types, Operators, Constants, Input/Output and the general C Program Structure. Watch this video first and then look through the lecture notes for complete details.

The Language of Numbers

While humans can understand and work with all kinds of numbers, a computer at its lowest level only understands one language, binary code. This code is made up of a sequence of 1 and 0 values representing an "on" and "off" power state at each particular computer memory location known as a **Bit**. If you put 8 bits together, they form a memory location known as a **Byte**, which one can use to represent a particular memory location that can be referenced by an **Address.**

A computer really doesn't understand a number like 12, but it can be represented with a series of zeros and ones in a binary code. Two other numbering systems that work well as a shorthand for binary code are **Octal** (base 8) and **Hexadecimal** (base 16). They both work because they represent a power of 2, where 2 to the 3rd power (or 2 x 2 x 2) is 8, in the Octal system, can be represented in 3 binary digits, while 2 to the fourth power (2 x 2 x 2 x 2) is 16, and in the Hexadecimal system, it can represent four binary digits.

The figure illustrated below shows how the **Decimal** (base 10, the way we think) numbers 0 to 16 are represented in Binary, Octal, and Hexadecimal. Note that in the base 10 system, you can only use 10 digits or 0 to 9. For Octal numbers there are only 8 digits, 0 to 7, and finally for Hexadecimal numbers, there are 16 digits that include 0 to 9 plus A for 10, B for 11, C for 12, D for 13, E for 14, and F for 15 for a total of 16 digits.

Decimal

Place 100	Place 10	Place 1
		0
		1
		2
		3
		4
		5
		6
		7
		8
		9
	1	0
	1	1
	1	2
	1	3
	1	4
	1	5
	1	6

Binary

Place 32	Place 16	Place 8	Place 4	Place 2	Place 1
					0
					1
				1	0
				1	1
			1	0	0
			1	0	1
			1	1	0
			1	1	1
		1	0	0	0
		1	0	0	1
		1	0	1	0
		1	0	1	1
		1	1	0	0
		1	1	0	1
		1	1	1	0
		1	1	1	1
	1	0	0	0	0

Octal

Place 64	Place 8	Place 1
		0
		1
		2
		3
		4
		5
		6
		7
	1	0
	1	1
	1	2
	1	3
	1	4
	1	5
	1	6
	1	7
	2	0

Hexadecimal

Place 256	Place 16	Place 1
		0
		1
		2
		3
		4
		5
		6
		7
		8
		9
		A
		B
		C
		D
		E
		F
	1	0

Converting Binary, Decimal, and Hexadecimal

A great web site to learn about decimal, binary, and hexadecimal numbers is **Math is Fun**. If you are a bit confused about number systems, I would encourage you to first refresh your memory about the decimal number system (base 10). A nice tutorial can be found at: http://www.mathsisfun.com/decimals.html

Once you get a handle on decimal numbers, get comfortable with both binary and hexadecimal numbers:

- binary
- hexadecimal

Finally, let's look at two simple applications you can try out. The first put alls all three number systems together and shows you how to convert from one to another. It can be found at: http://www.mathsisfun.com/binary-decimal-hexadecimal.html

The second item is an actual web based conversion tool. You can enter any binary, decimal, or hexadecimal number and it will calculate the correct value for all three. I typed in "32" in the Decimal area and it told me that its 100000 in binary and 20 in hex. You can try it out at: http://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html.

Number Systems in C

A computer best handles numbers in powers of 2, such as 4, 8, 16, 32, etc. The number system that we as humans are all familiar with is decimal, which is also known as base 10. Two other number systems that work well with the C Programming Language are **Octal** (base 8) and **Hexadecimal** (base 16). At times, it can be advantageous to "think" like a computer in terms of knowing how numbers and information are stored in memory.

Number Base	Digits	Prefix	Decimal 12
Decimal - Base 10	0-9	None	12

Octal - Base 8	0-7	0	014
Hexadecimal - Base 16	0-9, A-F	0x	0xC

With hexadecimal, the letters A-F represent the 10-15 digits.

A is 10, B is 11, C is 12, D is 13, E is 14, and F is 15

In C, Hexadecimal numbers start with **0x**, and Octal numbers start with **0.** This rule allows the compiler to both identify the base of the number (decimal, octal, hexadecimal) and separate it from identifying it as a variable (which starts with an alphabetic character first) as remember that a hexadecimal number can start with the letters A through F.

It should also be noted that C does not care if you use upper or lower cases letters for hexadecimal digits A-F ... a-f will work the same (and so will 0x and 0X to start the number).

The hexadecimal number:

0x4a5

is the same as:

0x4**A**5

A value of **0x**15 would be **hexadecimal** (starts with 0x) and equate to the following in base 10: (1 * 16^{1}) + (5 * 16^{0}).

In the C language, the * symbol represents the multiply operator.

Anything to the first power is itself and anything to the 0th power is one, so this now becomes (1 * 16) + (5 * 1) = 21

Likewise, a value of **0**15 would be **octal** (starts with zero) and be equal in base 10 to: (1 * 8^{1}) + (5 * 8^{0}) ... which equates to (1 * 8) + (5 * 1) = 13

Number	Base	Converted to Base 10
0x23	Hexadecimal	(2 * 16^{1}) + (3 * 16^{0}) which is (2 * 16) + (3 * 1) = 35
0x4F	Hexadecimal	(4 * 16^{1}) + (15 * 16^{0}) = (4 * 16) + (15 * 1) = 79 ... note that F is 15
012	Octal	(1 * 8^{1}) + (2 * 8^{0}) = (1 * 8) + (2 * 1) = 10
4A5	Decimal	Invalid as the digit 'A' is not a valid decimal digit (0 - 9). It would be valid if presented as a hexadecimal number: 0x4A5
0x4A5	Hexadecimal	(4 * 16^{2}) + (10 * 16^{1}) + (5 * 16^{0}) = (4 * 256) + (10 * 16) + (5 * 1) = 1189 ... note that this number starts with 0x to indicate it is a hexadecimal number and that A is 10
098	Octal	Invalid since it starts with a 0 meaning it is an Octal number ... both 9 and 8 are not valid Octal digits ... can only be made up of the digits 0 - 7
0	Octal, Decimal	0 is zero ... regardless if its Binary, Octal, Hexadecimal (0x0), or Base 10

Scientific Notation

Another way to represent numbers in C is through scientific notation. The table below illustrates how it is used:

```
0.00225 = 2.25e-3
22.5 = 2.25e1
2250 = 2.25e3
```

```
      2.25e-3
        |   |
  Mantissa Exponent
```

The exponent may be **e** or **E**. It represents the power of 10 to multiply the mantissa by. If an exponent is minus, move decimal left from standard position. If exponent is positive, move decimal right from standard position.

Data Types in C

Below are some standard types supported by C. Every language has various data types to sort information, each one having a unique purpose, and many of these types are also common to other programming languages.

int

An **integer** is any whole number, positive or negative and including 0. Examples include 5, -5, and 0. It may also be in number base 8 (octal), 10 (decimal), and/or 16 (hexadecimal).

float

A **floating point number** is the same as an integer but it must be in base 10 and contain a **decimal point**. Examples include 0.23456, 8, and 8.7. Notice that in the middle example, an 8.0 would be stored if a floating point variable was given an integer value. A floating point number typically has an **Exponent** section that stores the digits(s) found to the left of the decimal point, and a **Mantissa** that contains the digit(s) to the right of the decimal point.

A variable defined as a float is normally 4 bytes long and you can count on precision of floating point values up through 6 places past the decimal point. The best example of a floating point number for me would be **money** :) Fifty thousand dollars that I would hope to win on a scratch lottery ticket would be stored as: 50000.00

double

A value of **double** is the same as float but with increased precision. It can be represented as digits (such as 0.00225) or in scientific notation (such as 2.25e-3). It is best used when precise values are needed past the decimal point, to the sixth place and beyond up to 15 decimal places, as in 0.234556677772356

Note: On some computers, a type of double increases the variable memory size in terms of extra bytes usually to a size of 8 bytes. This enhances the precision by allowing a longer **mantissa**, which holds the numbers past the decimal point. The extra storage provided means that the variable can now hold more numbers past the decimal instead of just **rounding** to the nearest decimal point. Double is sometimes referred to as **long float**.

Additional Note: On some systems, a type of **long double** is supported which has a size of 10 bytes and a precision up to 19 decimal places.

char

A **character** maps to the **ASCII character set** which we will discuss in detail in Chapter 10. It represents most the characters that you can input from the terminal keyboard and each is enclosed within **single quotes**. Some examples include 'A', 'a', '#', and '\n', and '\t'. Note that the **special** characters '\n' (new line) and '\t' (which is a tab) are treated as one character by the computer. You can see all the characters at: http://www.ascii-code.com

Data Type Examples

The program below provides examples of defining variables for each of the data types previously discussed while initializing them at the same time. If you do not initialize a C Variable, it is **Undefined**. This means it could be any value and a programmer should not assume that it is zero.

The table below shows the **formats** that are used to print and read each data type.

Data Type	ANSI C	Traditional C
int	%i or %d	%d
float	%f	%f
double	%lf or %g	%f
char	%c	%c

NOTE - A common question for novice C programmers is understanding the ***difference*** between the **formats %d and %i** when used for *integers*. In terms of using either format for output, such as with a printf statement, they are identical. The only real difference is reading in numbers as input in C that are *hexadecimal* (numbers start with a 0x) and *octal* (numbers start with a 0). We will learn more about using the C library function **scanf** to read values into variables in an upcoming lecture note. For example, using the %i format, you can use the scanf function to read in hexadecimal numbers like **0x**4AD2 , and octal numbers like **0**472.

SAMPLE PROGRAM: Browse through the program below ... better yet, you can try it out at: http://ideone.com/WcRGK5. Add additional variables and see how they are printed to the screen.

```
#include <stdio.h>

main ()

{

    /* Declare and initialize each variable first  */

    int var1 = 42;
```

```
int var2 = 014;          /* any number that starts with zero is octal
*/

                          /* Octal 12 is equal to 10 decimal */

int var3 = 0x4A6;     /* any number that starts with 0x is
hexadecimal */

char var4 = '8';         /* characters can be any character from the
keyboard */

char var5 = 'a';

char var6 = '#';         /* each character should be in single quotes
*/

char var7 = 'A';

float var8 = 5.2;   /* floating pointer variables contain a decimal
point */

float var9 = 2.0;

float varA = 2;     /* will be stored as 2.0 */

float varB = 0.07;

float varC = 7.0e-2;  /* 0.07 */

double varD = 5.2;  /* double variables may contain more
decimal point numbers */

double varE = 12.33434343;

double varF = 5.667744544e3;    /* lots of digits passed the
decimal pointer */

/* Let's print some numbers to the screen */

printf ("Integers:  %i %i %i \n", var1, var2, var3);
printf ("Characters:  %c %c %c %c \n", var4, var5, var6, var7);
printf ("Floats:  %f %f %f %f %f \n", var8, var9, varA, varB, varC);
printf ("Doubles:  %g %g %g \n", varD, varE, varF);

return(0);

}
```

Output

This is what the C Program would print when executed. Note the different types of information that can be displayed.

```
Integers:  42 12 1190
Characters: 8 a # A
Floats: 5.200000 2.000000 2.000000 0.070000 0.070000
Doubles:  5.2 12.3343 5667.74
```

Working with other Numerical Systems

This program shows variables of different numerical systems being declared and initialized in a program

More than one variable can be declared and initialized on one line if separated by commas. I am only showing you this to let you know it is possible. For all your homework and exams, I want you to follow the standard of declaring and commenting each variable declaration on its own line. See the coding/homework standards for details.

Example:

```
int var1, var2 = 014, var3 = 45, var4 = var3 + 8;
```

In the example above, the following variable would be stored as follows:

var1: UNDEFINED Undefined
var2: 12 12
var3: 45 45
var4: 53 53

Note that var2 is an *octal number,* but let us think in terms of *base 10* the rest of the semester and show it as 12 (in base 10). Octal 14 = Decimal 12 or 1 x 8 + 4

Data Type Formatting

Let's take a more detailed look at the various formatting options available when printing integer and floating point values. You will need to understand these concepts in order to properly format the output of your homework assignments. Feel free to past this into your favorite compiler and try it out. Also, consult the input/output section near the end of the Kochan book, where the first few pages go over various supported data formats. You can try out the sample program below with IDEOne at https://ideone.com/Yb4jKB.

```
#include <stdio.h>
int main ( )
{
    int   ivalue = 7;            /* simple integer variable  */
    float fvalue = 10.23;        /* simple floating point variable */

    printf ("%i \n", ivalue);    /* no special format */

    printf ("%3i \n", ivalue);   /* width of 3 minimum spaces */

    printf ("%03i \n", ivalue);  /* width of 3 spaces padded with zeros */

    printf ("%f \n", fvalue);    /* Yikes!  lots of zeros displayed */

    printf ("%7.2f \n", fvalue);  /* 7 space width, two decimal places */

    printf ("%-7.2f \n", fvalue);   /* left justify, note the minus sign  */

    return (0);
}
```

The output of the program above will be:

```
7
  7
007
10.230000
  10.23
10.23
```

When printing, output will start on column 1 of the item being printed (a file, print out, screen, ...), and go from there. Let's look at what we did above. The first printf, just uses %i to print an integer, and it has no special formatting. Since we printed a value of 7, it just starts the printing on the first available space.

Column	1	2	3	4	5	6	7

Value	7						

The second printf used %3i, this specifies a **width of a minimum of 3 spaces**. In our example, we just printed one integer digit, the 7, so it pads the other two positions to the left with spaces. If you printed an integer with 4 or more digits, it would just print it. The width is just the minimum width printed. Note that by default, output is **right justified**. I've found **right justification** is good when printing multiple numbers in rows and columns, like in a spreadsheet.

Column	1	2	3	4	5	6	7
Value			**7**				

The third printf (%03i) puts a zero value along with a **width of 3**, for which it will take up a minimum of three places and will **pad with zeros** instead of blanks. It would be a good format to know if you worked with super spy James Bond.

Column	1	2	3	4	5	6	7
Value	**0**	**0**	**7**				

The fourth printf statement (%f) will just print the number 10.23, but most systems will print floating point values up to six places past the decimal and **pad it with zeros**. In many cases, this is not a very attractive output. What you want to do is utilize **zero suppression**, in other words, don't show all those extra zeros, just display the decimal places that make sense for the data item being presented. For example, if you were showing dollars and cents, show only two places past the decimal to show the cents.

Column	1	2	3	4	5	6	7	8	9
Value	**1**	**0**	**.**	**2**	**3**	**0**	**0**	**0**	**0**

The fifth printf statement (%7.2f) will contain a **width** (7 places - includes the decimal point) and a **precision** of digits past the decimal (2 places). Since there are less digits and a decimal point here, a total of 5 items, it will pad with blanks two places to the left. Let's make sure we can visualize things to understand what is happening here.

Column	1	2	3	4	5	6	7
Value			**1**	**0**	**.**	**2**	**3**

In another example, if you want to denote **money,** you want to show only two places past the decimal to print **cents**. If you wanted to insert a **dollar sign**, you could do this as well:

```
printf ("$%7.2f\n", fvalue);
```

It would print a dollar sign in the first column, and then display the number in the next 7 spaces:

Column	1	2	3	4	5	6	7	8
Value	$			**1**	**0**	**.**	**2**	**3**

The last statement (%-7.2f) will **left justify** the output ... note that it starts at the first available space in the output and the min value is proceeded with the '-' character. **Left justification** is really useful when you want values to start in the left most column, a good example might be if you are printing names of people in a column.

It would then print:

Column	1	2	3	4	5	6	7
Value	**1**	**0**	**.**	**2**	**3**		

To conclude all this, what would happen if you printed a value larger than 6 digits and a decimal point while using a format like %7.2f, such as 12150.56? It would filter over to the next available column, but will still continue to only show two places past the decimal point. It would still print all the numbers, skewing the output to right in this case.

Column	1	2	3	4	5	6	7	8
Value	**1**	**2**	**1**	**5**	**0**	**.**	**5**	**6**

Finally, what would happen if you printed with the same format, %7.2f, BUT the number had an extra decimal place, as 12150.568? In this case, you are still only showing two places past the decimal, BUT because the last digit here is greater than or equal to 5, it would **round up** and display the following:

Column	1	2	3	4	5	6	7	8
Value	**1**	**2**	**1**	**5**	**0**	**.**	**5**	**7**

Conclusion

The **data formatting** and **zero suppression** features in C allow you to show only what is needed in an output of a variable. You don't want things cluttered up by displaying information that is not useful. Another benefit is that you can design your output so it consistently displays information in a predictable and orderly manner, especially when showing multiple variables on a given line.

Data Types - Output

Let's look at the following example for printing output.

```
#include <stdio.h>
 main ()
{
    int var1, var2 = 027; /* decimal 23 */
    int var3 = 0xD;       /* decimal 13 */
    int var4;
```

var1: UNDEFINED Undefined
var2: 23 23
var3: 13 13
var4: UNDEFINED Undefined

```
   var4 = var2 + 5;
    var1 = 30;
```

var4: 28 28
var1: 30

```
    printf("%d %x %o\n", var1, var2, var3);
   return (0);
}
```

var1: 30 30
var2: 23 23
var3: 13 13
var4: 28

In **var2 + 5** a variable and a constant are added. The result of decimal 28 is assigned to var4.

variable var1 contains 30 base 10. Because the format is %d, it prints as 30.

var2 contains 23 base 10 and prints as 17 because %x causes conversion to base 16

The %o causes var3 to print as 15 (which is 13 in base 10)

The format characters are d for decimal, letter o for octal and x for hexadecimal.

The answer to what prints ? is :

30 17 15

Data Types - Input

Let us look at the following example for reading input into C variables that can be used within your program. The C library function "***scanf***" is typically used to capture input from standard input (i.e., your screen).

Below is a simple example using the standard **scanf** library function. In order to use most input and output functions, you need to remember to include the ***stdio.h*** header file that contains its proper declaration and definition. Well discuss header files in future lecture notes, but take in on faith that if you do any input or output, that you will likely need to include this file. It doesn't hurt to include it even if it is not used. Try out my example below using IDEOne at: http://ideone.com/8Ecpau

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

{
                int var1, var2 = 027; /* var2 is octal, decimal value of 23 */

                int var3 = 0xD; /* Hex number, decimal 13 */

                int var4;
```

var1: UNDEFINED Undefined
var2: 23 23
var3: 13 13
var4: UNDEFINED Undefined

```
                var4 = var2 + 5;
                var1 = 30;
```

var4: 28 28
var1: 30 30

```
                scanf ("%d %o %x", &var1, &var2, &var4);
                return (0);
}
```

If the following is entered from the keyboard:

```
30 30 30
```

What is stored?

The first 30 is read as base 10 and stored as 30

The %o treats the second 30 as base 8 and stores it as 24 base 10.

The third 30 value is treated as base 16 and stored as 48 base 10. That would be 3 * 16 + 0 * 1

Notice the use of the & in front of the variables in scanf. This will be explained later. For now, take it on faith.

var1: 30 30
var2: 24 24
var4: 48 48

A Note on Compilers from Microsoft

If you are using any of the compiler products from Microsoft, such as *Visual Studio* or *Visual C++*, you might get a warning from your compilation asking you to use a safer scanf statement called "***scanf_s***". The "safer" part more applies to reading ***character strings***, a topic we have not yet covered, so let's hold off on that discussion until we deal with them in a few weeks. Understand that this library function is *specific to Microsoft* products only (it will not work in IDEOne) and is not part of the ANSI or ISO C Standard. If you are using Microsoft products for your compiler, feel free to use scanf_s, but realize it's not *portable* if you migrate your code to a non-Microsoft compiler. I've added a few more notes and examples about some of these specific Microsoft functions in this week's homework assignment code template.

Here is what an example might look like with **scanf_s** in our sample program above ... for numeric values, it is really called the same way:

```
scanf_s ("%d %o %x", &var1, &var2, &var4);
```

You can still run your program with these warnings, but if you do not want to see them, just add the following symbolic constant to the top of the file after your include statements. It has no value, but makes a "symbol" known to the compiler to suppress warning messages in this area.

```
#define _CRT_SECURE_NO_WARNINGS /* Visual Studio, turn off safe function warnings */
```

Integer Qualifiers

Integer values can be given **qualifiers** to specify addressable sizes in terms of bytes, as well as represent positive and negative numbers. The default size for integers is machine dependent. In the "old days" as a rule, many personal computers defaulted to 16 bits for an integer compared to 32 bits on a UNIX server. With the C99 and C11 programming language standards extended for the C language and the evolution of machine hardware, most computers and compilers these days can qualify integer values for 16, 32, and 64 bits. You have some control in determining the size of your integer variables in terms of how many bytes or bits are allocated.

Before we present integer qualifiers, let's take an in-depth look at how the number 9 would be represented in computer memory constrained within a variable that is two bytes long or 16 bits. Remember that each byte is 8 bits. A bit location can have either a value of zero (0) or one (1), and each represents a power of 2.

The tricky item here is what is 2^0 power?

The answer is any number to the power of zero is always 1.

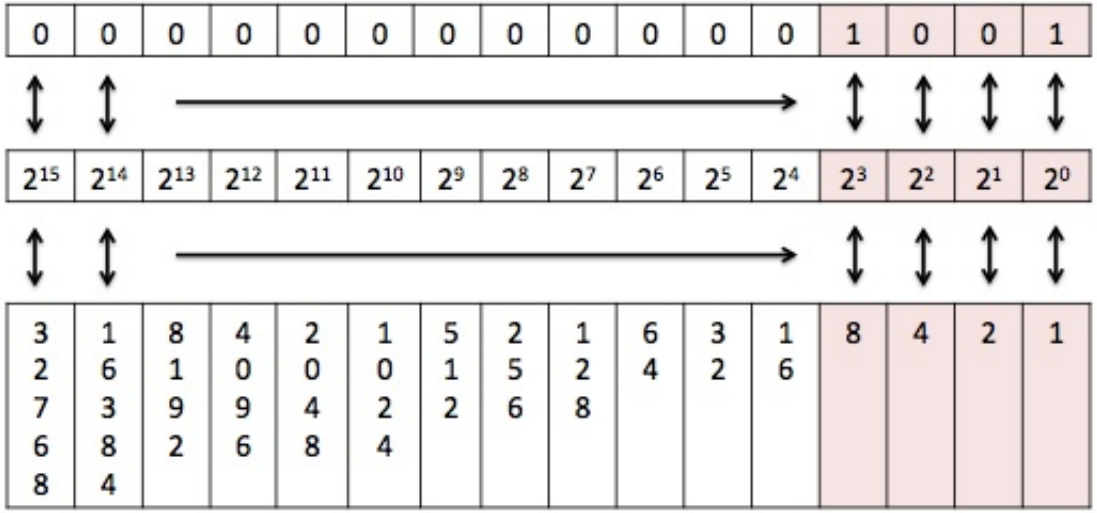
If you set all 16 bits to 1 and added up all the powers of two here, you would get a total of 65536. If you account for the value of zero, those 16 bits would allow one to store numbers in the range of 0 to 65535, or 65536 possible values.

For you math majors, that range can be easily calculated as:

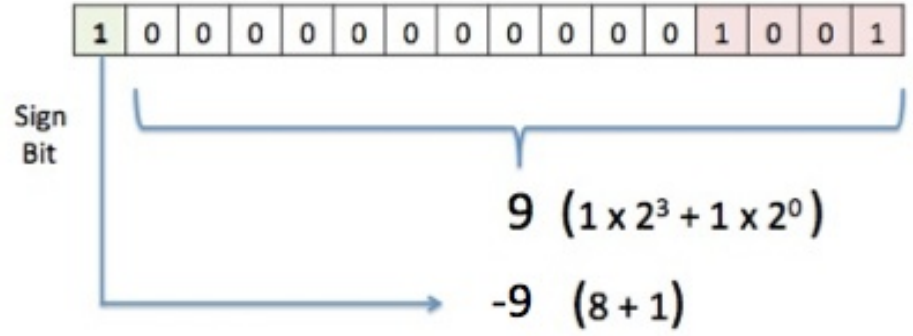
$0 \text{ to } 2^{16} - 1$

If you look at the highlighted cells below, you will see that the number being stored within these 16 bits is the number 9, which is calculated as:

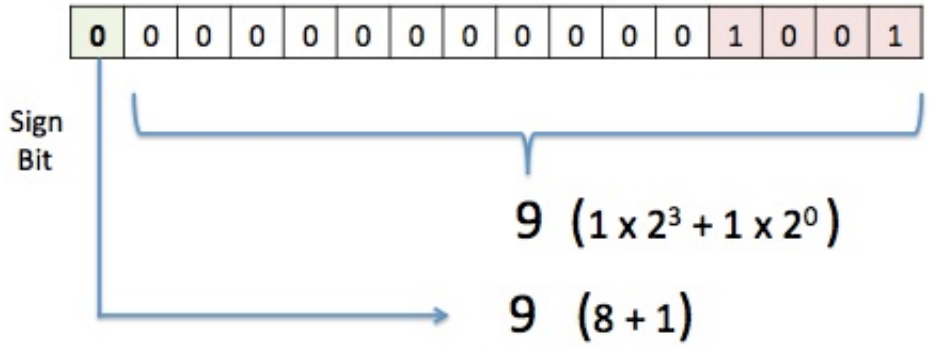
$$= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 8 + 0 + 0 + 1$$
$$= 9$$



By default, integers are **signed** and can contain positive or negative values, but that comes at a price as you will lose one bit of precision to store something called the **sign bit**, which is the **Most Significant Bit** where a value of 1 indicates the number will be **negative** as shown below:



... and a **sign bit** value of 0 indicates the number will be **positive** as shown below:



With the *sign bit* now occupying one of the bit locations, your number can now only be represented by 15 bits. If you set all 15 bits to 1, it would add up to a value of 32768. For those math inquiring minds, that's easily figured out by referring to my initial figure on this lecture note that has all the powers of 2 displayed, or on your calculator as:

2^{15} which is 32768

... and accounting for the value of zero, you can now store and represent a range of values from **-32768 to 32767** as you can utilize that *size bit* to make that value either positive or negative.

short (a.k.a. short int)

A **short** qualifier specifies that an integer size should be the minimum length. In most cases, this is 2 bytes or 16 bits.

```
short var1; /* 2 bytes or 16 bits */
short int var2; /* another way to specify a short variable */
scanf (" %hi %hi ", &var1, &var2); /* use %hi for short or short int format */
printf (" %hi %hi ", var1, var2); /* to print them */
```

long (a.k.a. long int)

A **long** qualifier generally doubles the size of an integer, extends it from 2 to 4 bytes for a total of 32 bits, which can range from –2,147,483,648 to 2,147,483,647

```
long var3;  /* 4 bytes or 32 bits */
long int var4;  /* another way to specify a long variable */
```

```
scanf (" %li %li ", &var3, &var4);  /* use %li for long or long int format */
printf (" %li %li ", var3, var4);  /* to print them */
```

long long (a.k.a. long long int)

A **long long** qualifier specifies that an integer size should be the maximum length. It will use the largest integer location available when run on specific machine. In most cases, it extends an integer to 8 bytes, or 64 bits, which can range from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

```
long long var5;   /* 8 bytes or 64 bits */
long long int var6;  /* another way to specify a long long variable */
scanf (" %lli %lli ", &var5, &var6);  /* use %lli for long long or long long int format */
printf (" %lli %lli ", var5, var6);  /* to print them */
```

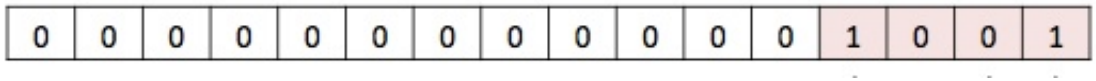
unsigned

By default, all integer values are signed (positive/negative numbers) unless the **unsigned qualifier** is specified. It can be placed in front of any integer type to indicate that the variable location will only store positive numbers, including 0. No negative numbers will be allowed. It does this by not using the ***most significant bit*** as the ***sign bit*** allowing all 16 bits to be used to represent a number.

Given that, a 16 bit **unsigned value** can range from:

0 to 65535 or 0 to 2^{16} - 1.

As previously discussed, the number 9 is shown below, but again, any number between 0 and 65535 can be represented through various combinations of zeros and ones within these 16 bits.



Taking that concept a few steps further, a 32 bit unsigned number would range from:

0 to 4,294,967,295 (or 0 to 2^{32} - 1)

... and a 64 bit unsigned number would range from:

0 to 18,446,744,073,709,551,615 (or 0 to 2^{64} - 1).

Below is an example using 16, 32, and 64 bit unsigned variables along with the proper format for reading or printing these values.

```
unsigned int var7;   /* 16 bits, Positive values only: 0 to 65535 ... you could also say unsigned short int */

unsigned long var8;  /* 32 bits, Positive values only: 0 to 4,294,967,295 */

unsigned long long var9;  /* 64 bits, Positive values only: 0 to 18,446,744,073,709,551,615 */

scanf (" %u %lu %llu ", &var7, &var8, &var9);   /* various formats to read in unsigned variables */

printf (" %u %lu %llu ", var7, var8, var9);   /* to print them */
```

Important Note: One last thought to consider here is what would happen if you tried to put a number into a variable greater than the max value? Take the case of a unsigned short integer type that would be two bytes or 16 bits. The numeric range here is 0 to 65535. If you put in a number like 65538, it would start overlapping where 65536 would be 0, 65537 would 1, 65538 would be 2 ... and so on. You can see my example which proves this at: http://ideone.com/VcuwDd

```
unsigned short var1;       /* 2 bytes or 16 bits */
unsigned short int var2;   /* another way to specify a short variable */

scanf (" %u %u ", &var1, &var2);  /* read and print the short integers */
printf (" %u %u ", var1, var2);
```

Entering 65538 and 4 for var1 and var2 results in var1 being 2 and var2 being 4. The important thing to note here is that the number 65538 could not and would not be represented in any short integer variable whose size is two bytes or 16 bits. The same would be true for values that exceed the upper or lower limits of 32 and 64 bit variables, regardless of whether they are signed or unsigned.

Summary

On some computers, **long** will increase storage space allowing the capacity to hold a longer number; **short** may be used to save space; **unsigned**, by permitting only positive numbers, *doubles* the size of the number that can be held. Note in Chapter 16/17 depending on your book version, Kochan notes that there is an "l" modifier for printf and scanf, a

short modifier "h" is for scanf only, and the "u" is not a modifier, it replaces i, d, o, or x

Binary Operators

Programming languages typically support a set of **operators** that provide a wide range of functions. They will normally work on two items (a **binary** operator) or a single item (a **unary** operator). Operators are very similar between programming languages, although they may different in syntax. Some of the most common operators can be categorized as Arithmetic, Relational and Comparison, Logical, and Compound Assignment. Let's save Logical operators for later and discuss the others below, as they are fairly common from one language to the next.

As we cover many of the operators, you'll note that there is an **order of precedence**. The Kochan book has a nice precedence table in the back, it is something you should become familiar with, as it is important to understand. The best way to deal with precedence issues is to follow the coding standard at the end of this lecture note. If you need access to a C Precedence table, just search for it on line as there are many good ones. Here is a link to one I found that contains the basic items we are talking about this week:

http://www.swansontec.com/sopc.html

C provides basic mathematical binary operators that work on a pair (i.e., binary) of values. Below are the common operators:

Evaluate First: * (multiply), / (divide), and % (modulus or mod)

Evaluate Second: + (add) and - (subtract)

Evaluate Third: = (assignment)

The order of evaluation is **left to right**.

*, / and % have **equal precedence**. They are evaluated in left to right order in an equation. Intermediate values are created to hold the result. Then +, - are evaluated from left to right.

The order of evaluation may be forced by using the innermost parentheses evaluated first.

Example	Means
X = 5 + 3 * I;	X = 5 + (3 * I);
X = 5 * 3 + I;	X = (5 * 3) + I;
X = 5 * 3 + I / 5;	X = (5 * 3) + (I / 5);

A word on the mod function: Mod takes the first number and divides it by the second number returning the **remainder** only. For example:

5 % 2 = 2R1, so it returns 1

6 % 2 = 3R0, so it returns 0

4 % 6 = 0R4, so it returns 4

TIP

A very common error with C is not understanding the precedence rules that can affect the person that created the code, is reading the code, and the one that has to modify and maintain that section of code. Consider the following statement:

```
value = 3 + 12 * 5;
```

In C, multiplication has higher precedence than addition, so it would in reality be executed by the program as:

```
value = 3 + ( 12 * 5 );
```

You can of course, change the precedence yourself with parentheses, to make the add go first, as in:

```
value = ( 3 + 12 ) * 5;
```

You should never put yourself or anyone else in this situation, please conform to the following coding standard:

Standard *- Use vertical and horizontal white space (indention) and redundant parentheses to make the code more readable. Extra parentheses and indentation will never generate additional executable code.*

The latter point in this coding standard is that your code is transformed and executed as machine code when you run you program. Adding extra parentheses and indentation may look like you are adding more code, but it is not adding any additional executable code that will be run. Readable and maintainable code should be your main focus, making it as efficient whenever possible, but not so efficient that it is difficult to read and understand, hence, the potential for generating more errors in the future.

PITFALL:

One thing to always watch out for is a **Division by Zero** error. If this happens during the execution of your program, it will typically halt the processing and die a quick death, often dumping core errors and indicating that a zero divide condition has occurred. It can be an easy thing to do when you divide one variable by another variable (and remember that the Modulus operator always does a divide). A statement like:

```
5/0
```

... would be obvious, don't do this ... your compiler might also catch it during the compilation process. but a statement like:

```
current_temp / temperature
```

... would not be obvious, as temperature could be zero at some point when running your program.

C++ provides ***exception handling*** to check for this, but not so with C. We can use ***if statements*** (covered next week) to check for these conditions.

Unary Operators

Unary operators work on only **one** value.

```
#include <stdio.h>
main ()
{
        int a = 10, b = 4, c;
        c = a *-b;      /* unary minus, negates the value of b */
        printf (" %i \n", c);
        return (0);
}
```

What would print?

Unary operators use ONE operand. The unary minus operator (-) deals with one item, in this case, the variable "b". Note that a minus operator (-) can also be used as a binary operator to subtract values. The binary operator * deals with the variables a and b. Review relative precedence of unary, binary, and assignment operators in the operator precedence chart which can be found in the Input and Output chapter of your textbook. The print value for this program would be -40

Assignment Operators

Assignment operators work on only one value. It is a good idea to get familiar with them and use them ... I'll be on the look out during the semester making sure you use them in your homework and exams. Read my notes at the end of this page, and you'll know why.

Operator	Example	Means
=	a = b + c	a = b + c;
+=	a += b;	a = a + b;
-=	a -= c;	a = a - c;
*=	a *= c + d;	a = a * (c + d);
/=	a /= b;	a = a / b;
%=	a %= b * c;	a = a % (b * c);

The combined operators are a type of shorthand notation not found in most other languages.

Notice the **precedence** in a statement like a *= b + c (addition happens first). Use parentheses if you want to change how the statement would work. Adding parentheses does NOT add any extra executable code and helps both you and any future programmers easily understand the exact precedence. For example, the statements.

```
a *= c + d;
```

is equivalent to

```
a = a * (c + d);
```

However, if you wanted **a** to be multiplied by **c** and then add **d** to **a**, you could do it in two statements:

```
a *= c;
a += d;
```

which is equivalent to:

```
a = (a * c) + d;
```

Binary operators like add, multiple, subtract, divide, and modulus take precedence over assignment operators

Important Insider TIP

It is important to note that if you had a statement like **a = a * b;** verses **a *= b;** ... that the latter statement with the ***=** would be better because with the right compiler, it will often (but not always) translate into less assembly statements (i.e., equivalent machine instructions).

Why is that important? **C is built for speed**, it is one of the fast running languages, which is why it is still popular today. Each C instruction can be seen as an equivalent set of one or more assembly language instructions that are actually executed on your target machine. The less statements executed, the better potential for a faster running program.

As a software engineer, code optimization is quite of bit of what we do, as coding is probably only 10 to 15 percent of the total effort. By utilizing these tips ***consistently*** throughout the semester, you'll become a more efficient C programmer ... and with other programming languages too!

Conversion Rules

Given that int_value is an integer, and float_val is a floating point value, let's look at 4 conversion rules with an example for each. You will find it is important to understand whether you are dealing with integer verses floating point values, as it makes a big difference in the results produced.

```
int   int_value;
float float_value;
```

1. If a float value is assigned to an integer value, the decimal part is truncated.

```
int_value = 4.567;
```

int_value: 4

2. If all operands are integer, fraction result is truncated. No rounding.

```
int_value = 14/5; /* 2.8 */
```

int_value: 2

3. If integer value is assigned to float value, .0 is added.

```
float_value = 5;
```

float_value: 5.0

4. If mixed arithmetic (some integer, some float), it is performed as float.

```
float_value = 4.56 + 8;
```

float_value: 12.56

Constants

Unlike a variable in computer programming, a **constant** is an identifier whose associated value is not modified by the program during its execution. The idea is to specify a value for a constant in one place, and then reference it as many times as needed in your program. If that constant value needs to be changed in the future, you need only change it in one place and then recompile your program.

In larger programs, you will often have a value that appears more than once. For example, let's say you do lots of equations with PI, which has a value of 3.14

In various places in your code, you might have statements like:

```
circleArea = 3.14 * radius * radius;   /* area of a circle  */

value = 3.14 * value2;   /* use it in multiple places */
```

... and its probably in MANY places in your program or a set of programs you and your organization may have to maintain. Instead, you could use a symbolic constant that would be set up once and could be used anywhere in your program. It would have the following format ... I normally put them right after my include statements at the top of the C file.

```
#define CONSTANT VALUE_FOR_CONSTANT
```

Given that, you could define a constant called PI as:

```
#define PI 3.14
```

... in your code, you would change any reference of 3.14 to PI, thus:

```
circleArea = PI * radius * radius;

value = PI * value2;
```

One thing to note here is with some compilers, putting a value like 3.14 in your code (as the symbolic constant would do) may require you to append the letter 'f' after it to indicate its a literal floating point value. Use can use upper or lower case for the letter, each will work:

```
#define PI 3.14f
```

or

```
#define PI 3.14F
```

... Finally, let's say PI changes (not likely) but you might want to give it more precision in your program:

```
#define  PI   3.14159265f
#define  BIG_PI   3.14159265f    /* ... or just create another constant */
```

... You now need only to change it in ONE place and recompile your program.

The only places where a constant value would get ignored is if it happens to be in a literal string (like in a printf) or in a comment, as comments are ignored by the compiler.

```
/* Print out PI */
printf ("PI is %f \n", PI);
```

... would print a default six digits (rounded) past the decimal point since the format is %f

```
PI is  3.131593
```

C actually has a pre-compiler that would actually change the above two statements and the compiler would see it as:

```
/* Print out PI */
printf ("PI is %f \n", 3.1412345);
```

Alternative: const

Another way to use constants in C is to use the **const** feature which allocates space for a variable, but does not allow you to change its value later in the program. It essentially makes that variable READ-ONLY throughout its scope. The syntax is shown below. Note that you have to pick a *valid C type*, it is assigned an *initial value*, and it ends with a *semicolon*.

```
const <type> <variable_name> = <value>;
```

We will learn more about scope of variables later on in this class, but if you want to use *const*, just put it with all the

other variables you declare in a given function (which for now, is the main function).

```
const float PI = 3.14;    /* ... or just create another constant */
float my_pi;              /* just a simple float variable */

my_pi = PI:               /* OK, my_pi value can change, it is a variable */

PI = 3.15;                /* will cause a compiler error, PI is read-only, its value can't change */
```

Constant DO's and DON'Ts

- DO ***uppercase*** each symbolic constant name as it helps a programmer to quickly identify if an item is a variable or constant.

```
#define PI 3.14                 /* yes */
#define big_pi 3.1412325        /* no, should be upper case ... BIG_PI */
#define BIG_PI 3.14159265       /* yes */
```

- DON'T ***mix variables and constants*** in your constants ... you can't always guarantee that a variable by that exact name would exist in a future program.

```
#define AREA PI * radius * radius   /* no, don't mix variables in a constant definition */
#define TWO_PI (PI * PI)            /* yes, OK to use various constants together */
```

- DON'T ***add a semicolon*** at the end of constant declaration as it will likely cause a compiler error when the constant is used in a statement

```
#define STD_HOURS 40.0    /* yes */
#define STD_HOURS 40.0;   /* no, lose the semicolon at the end */
```

- DO use the **const** feature if you wish as well.

```
const double PI = 3.14159265;
```

Benefits of Constants

In summary, the benefit of constants is:

- **Improve readability**... PI is better than 3.1412345
- **Improve maintainability** ... change it in one place and its changed everywhere it is referenced in your code
- **Reduce Errors** - Consistent values used throughout your program