

Chapter 3 Storing Data: Variables and Constants

3-1. Computer Memory

If you already know how a computer's memory operates, you can skip this section. If you're not sure, however, read on. This information will help you better understand certain aspects of C programming.

A computer uses random-access memory (RAM) to store information while it's operating. RAM is located in integrated circuits, or chips, inside your computer. RAM is volatile, which means that it is erased and replaced with new information as often as needed. Being volatile also means that RAM "remembers" only while the computer is turned on and loses its information when you turn the computer off.

Each computer has a certain amount of RAM installed. The amount of RAM in a system is usually specified in kilobytes (KB) or megabytes (MB), such as 512KB, 640KB, 2MB, 4MB, or 8MB. One kilobyte of memory consists of 1,024 bytes. Thus, a system with 640KB of memory actually has $640 * 1,024$, or 65,536, bytes of RAM. One megabyte is 1,024 kilobytes. A machine with 4MB of RAM would have 4,096KB or 4,194,304 bytes of RAM.

The *byte* is the fundamental unit of computer data storage. Day 20, "Working with Memory," has more information about bytes. For now, to get an idea of how many bytes it takes to store certain kinds of data, refer to Table 3.1.

Table 3.1. Memory space required to store data.

Data	Bytes Required
The letter x	1
The number 500	2
The number 241.105	4
The phrase <i>Teach Yourself C</i>	17
One typewritten page	Approximately 3,000

The RAM in your computer is organized sequentially, one byte following another. Each byte of memory has a unique address by which it is identified--an address that also distinguishes it from all other bytes in memory. Addresses are assigned to memory locations in order, starting at zero and increasing to the system limit. For now, you don't need to worry about addresses; it's all handled automatically by the C compiler.

What is your computer's RAM used for? It has several uses, but only data storage need concern you as a programmer. Data is the information with which your C program works. Whether your program is maintaining an address list, monitoring the stock market, keeping a household budget, or tracking the price of hog bellies, the information (names, stock prices, expense amounts, or hog futures) is kept in your computer's RAM while the program is running.

Now that you understand a little about the nuts and bolts of memory storage, you can get back to C programming and how C uses memory to store information.

3-2. Variables

A *variable* is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there.

3-2-1. Variable Names

To use variables in your C programs, you must know how to create variable names. In C, variable names must adhere to the following rules:

- The name can contain letters, digits, and the underscore character (`_`).
- The first character of the name must be a letter. The underscore is also a legal first character, but its use is not recommended.
- Case matters (that is, upper- and lowercase letters). Thus, the names `Count` and `count` refer to two different variables.
- C keywords can't be used as variable names. A keyword is a word that is part of the C language. (A complete list of 33 C keywords can be found in Appendix B, "Reserved Words.")

The following list contains some examples of legal and illegal C variable names:

<i>Variable Name</i>	<i>Legality</i>
<code>Percent</code>	Legal
<code>y2x5_fg7h</code>	Legal
<code>Annual_profit</code>	Legal
<code>_1990_tax</code>	Legal but not advised
<code>savings#account</code>	Illegal: Contains the illegal character <code>#</code>
<code>Double</code>	Illegal: Is a C keyword
<code>9winter</code>	Illegal: First character is a digit

Because C is case-sensitive, the names `percent`, `PERCENT`, and `Percent` would be considered three different variables. C programmers commonly use only lowercase letters in variable names, although this isn't required. Using all-uppercase letters is usually reserved for the names of constants (which are covered later in this chapter).

For many compilers, a C variable name can be up to 31 characters long. (It can actually be longer than that, but the compiler looks at only the first 31 characters of the name.) With this flexibility, you can create variable names that reflect the data being stored. For example, a program that calculates loan payments could store the value of the prime interest rate in a variable named `interest_rate`. The variable name helps make its usage clear. You could also have created a variable named `x` or even `johnny_carson`; it doesn't matter to the C compiler. The use of the variable, however, wouldn't be nearly as clear to someone else looking at the source code. Although it might take a little more time to type descriptive variable names, the improvements in program clarity make it worthwhile.

Many naming conventions are used for variable names created from multiple words. You've seen one style: `interest_rate`. Using an underscore to separate words in a variable name makes it easy to interpret. The second style is called *camel notation*. Instead of using spaces, the first letter of each word is capitalized. Instead of `interest_rate`, the variable would be named `InterestRate`. Camel notation is gaining popularity, because it's easier to type a capital letter than an underscore. We use the underscore in this book because it's easier for most people to read. You should decide which style you want to adopt.

DO use variable names that are descriptive.

DO adopt and stick with a style for naming your variables.

DON'T start your variable names with an underscore unnecessarily.

DON'T name your variables with all capital letters unnecessarily.

3-3. Numeric Variable Types

C provides several different types of numeric variables. You need different types of variables because different numeric values have varying memory storage requirements and differ in the ease with which certain mathematical operations can be performed on them. Small integers (for example, 1, 199, and -8) require less memory to store, and your computer can perform mathematical operations (addition, multiplication, and so on) with such numbers very quickly. In contrast, large integers and floating-point values (123,000,000 or 0.000000871256, for example) require more storage space and more time for mathematical operations. By using the appropriate variable types, you ensure that your program runs as efficiently as possible.

C's numeric variables fall into the following two main categories:

- Integer variables hold values that have no fractional part (that is, whole numbers only). Integer variables come in two flavors: signed integer variables can hold positive or negative values, whereas unsigned integer variables can hold only positive values (and 0).
- Floating-point variables hold values that have a fractional part (that is, real numbers).

Within each of these categories are two or more specific variable types. These are summarized in Table 3.2, which also shows the amount of memory, in bytes, required to hold a single variable of each type when you use a microcomputer with 16-bit architecture.

Table 3.2. C's numeric data types.

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Integer	int	2	-32768 to 32767
Short integer	short	2	-32768 to 32767
Long integer	long	4	-2,147,483,648 to 2,147,438,647
Unsigned character	unsigned char	1	0 to 255
Unsigned integer	unsigned int	2	0 to 65535
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned long integer	unsigned long	4	0 to 4,294,967,295
Single-precision	float	4	1.2E-38 to
Floating-point			3.4E38 ¹
Double-precision	double	8	2.2E-308 to
Floating-point			1.8E308 ²
¹ Approximate range; precision = 7 digits.			
² Approximate range; precision = 19 digits.			

Approximate range means the highest and lowest values a given variable can hold. (Space limitations prohibit listing exact ranges for the values of these variables.) *Precision* means the accuracy with which the variable is stored. (For example, if you evaluate 1/3, the answer is 0.33333... with 3s going to infinity. A variable with a precision of 7 stores seven 3s.)

Looking at Table 3.2, you might notice that the variable types int and short are identical. Why are two different types necessary? The int and short variable types are indeed identical on 16-bit IBM PC-compatible systems, but they might be different on other types of hardware. On a VAX system, a short and an int aren't the same size. Instead, a short is 2 bytes, whereas an int is 4. Remember that C is a flexible, portable language, so it provides different keywords for the two types. If you're working on a PC, you can use int and short interchangeably.

No special keyword is needed to make an integer variable signed; integer variables are signed by default. You can, however, include the signed keyword if you wish. The keywords shown in Table 3.2 are used in variable declarations, which are discussed in the next section.

Listing 3.1 will help you determine the size of variables on your particular computer. Don't be surprised if your output doesn't match the output presented after the listing.

Listing 3.1. A program that displays the size of variable types.

```
1:      /* SIZEOF.C--Program to tell the size of the C variable */
2:      /*          type in bytes */
3:
4:      #include <stdio.h>
5:
6:      main()
7:      {
8:
9:          printf( "\nA char      is %d bytes", sizeof( char ) );
10:         printf( "\nAn int       is %d bytes", sizeof( int ) );
11:         printf( "\nA short      is %d bytes", sizeof( short ) );
12:         printf( "\nA long        is %d bytes", sizeof( long ) );
13:         printf( "\nAn unsigned char is %d bytes", sizeof( unsigned
char ) );
14:         printf( "\nAn unsigned int  is %d bytes", sizeof( unsigned
int ) );
15:         printf( "\nAn unsigned short is %d bytes", sizeof( unsigned
short ) );
16:         printf( "\nAn unsigned long  is %d bytes", sizeof( unsigned
long ) );
17:         printf( "\nA float        is %d bytes", sizeof( float ) );
18:         printf( "\nA double       is %d bytes\n", sizeof( double ) );
19:
20:         return 0;
21:     }
A char      is 1 bytes
An int       is 2 bytes
A short      is 2 bytes
A long        is 4 bytes
An unsigned char is 1 bytes
An unsigned int  is 2 bytes
An unsigned short is 2 bytes
An unsigned long  is 4 bytes
A float        is 4 bytes
A double       is 8 bytes
```

ANALYSIS: As the preceding output shows, Listing 3.1 tells you exactly how many bytes each variable type on your computer takes. If you're using a 16-bit PC, your numbers should match those in Table 3.2.

Don't worry about trying to understand all the individual components of the program. Although some items are new, such as `sizeof()`, others should look familiar. Lines 1 and 2 are comments about the name of the program and a brief description. Line 4 includes the standard input/output header file to help print the information on-screen. This

is a simple program, in that it contains only a single function, `main()` (lines 7 through 21). Lines 9 through 18 are the bulk of the program. Each of these lines prints a textual description with the size of each of the variable types, which is done using the `sizeof` operator. Day 19, "Exploring the C Function Library," covers the `sizeof` operator in detail. Line 20 of the program returns the value 0 to the operating system before ending the program.

Although I said the size of the data types can vary depending on your computer platform, C does make some guarantees, thanks to the ANSI Standard. There are five things you can count on:

- The size of a char is one byte.
- The size of a short is less than or equal to the size of an int.
- The size of an int is less than or equal to the size of a long.
- The size of an unsigned is equal to the size of an int.
- The size of a float is less than or equal to the size of a double.

3-3-1. Variable Declarations

Before you can use a variable in a C program, it must be declared. A variable declaration tells the compiler the name and type of a variable and optionally initializes the variable to a specific value. If your program attempts to use a variable that hasn't been declared, the compiler generates an error message. A variable declaration has the following form:

```
typename varname;
```

typename specifies the variable type and must be one of the keywords listed in Table 3.2. *varname* is the variable name, which must follow the rules mentioned earlier. You can declare multiple variables of the same type on one line by separating the variable names with commas:

```
int count, number, start;    /* three integer variables */
float percent, total;       /* two float variables */
```

On Day 12, "Understanding Variable Scope," you'll learn that the location of variable declarations in the source code is important, because it affects the ways in which your program can use the variables. For now, you can place all the variable declarations together just before the start of the `main()` function.

3-3-2. The typedef Keyword

The `typedef` keyword is used to create a new name for an existing data type. In effect, `typedef` creates a synonym. For example, the statement

```
typedef int integer;
```

creates `integer` as a synonym for `int`. You then can use `integer` to define variables of type `int`, as in this example:

```
integer count;
```

Note that `typedef` doesn't create a new data type; it only lets you use a different name for a predefined data type. The most common use of `typedef` concerns aggregate data types, as explained on Day 11, "Structures." An aggregate data type consists of a combination of data types presented in this chapter.

3-3-3. Initializing Numeric Variables

When you declare a variable, you instruct the compiler to set aside storage space for the variable. However, the value stored in that space—the value of the variable—isn't defined. It might be zero, or it might be some random "garbage" value. Before using a variable, you should always initialize it to a known value. You can do this independently of the variable declaration by using an assignment statement, as in this example:

```
int count;    /* Set aside storage space for count */
count = 0;    /* Store 0 in count */
```

Note that this statement uses the equal sign (=), which is C's assignment operator and is discussed further on Day 4, "Statements, Expressions, and Operators." For now, you need to be aware that the equal sign in programming is not the same as the equal sign in algebra. If you write

```
x = 12
```

in an algebraic statement, you are stating a fact: "x equals 12." In C, however, it means something quite different: "Assign the value 12 to the variable named x."

You can also initialize a variable when it's declared. To do so, follow the variable name in the declaration statement with an equal sign and the desired initial value:

```
int count = 0;
double percent = 0.01, taxrate = 28.5;
```

Be careful not to initialize a variable with a value outside the allowed range. Here are two examples of out-of-range initializations:

```
int weight = 100000;
unsigned int value = -2500;
```

The C compiler doesn't catch such errors. Your program might compile and link, but you might get unexpected results when the program is run.

DO understand the number of bytes that variable types take for your computer.

DO use typedef to make your programs more readable.

DO initialize variables when you declare them whenever possible.

DON'T use a variable that hasn't been initialized. Results can be unpredictable.

DON'T use a float or double variable if you're only storing integers. Although they will work, using them is inefficient.

DON'T try to put numbers into variable types that are too small to hold them.

DON'T put negative numbers into variables with an unsigned type.

3-4. Constants

Like a variable, a *constant* is a data storage location used by your program. Unlike a variable, the value stored in a constant can't be changed during program execution. C has two types of constants, each with its own specific uses.

3-4-1. Literal Constants

A *literal constant* is a value that is typed directly into the source code wherever it is needed. Here are two examples:

```
int count = 20;
float tax_rate = 0.28;
```

The 20 and the 0.28 are literal constants. The preceding statements store these values in the variables `count` and `tax_rate`. Note that one of these constants contains a decimal point, whereas the other does not. The presence or absence of the decimal point distinguishes floating-point constants from integer constants.

A literal constant written with a decimal point is a floating-point constant and is represented by the C compiler as a double-precision number. Floating-point constants can be written in standard decimal notation, as shown in these examples:

```
123.456
0.019
100.
```

Note that the third constant, `100.`, is written with a decimal point even though it's an integer (that is, it has no fractional part). The decimal point causes the C compiler to treat the constant as a double-precision value. Without the decimal point, it is treated as an integer constant.

Floating-point constants also can be written in scientific notation. You might recall from high school math that scientific notation represents a number as a decimal part multiplied by 10 to a positive or negative power. Scientific notation is particularly useful for representing extremely large and extremely small values. In C, scientific notation is written as a decimal number followed immediately by an E or e and the exponent:

1.23E2 1.23 times 10 to the 2nd power, or 123

4.08e6 4.08 times 10 to the 6th power, or 4,080,000

0.85e-4 0.85 times 10 to the -4th power, or 0.000085

A constant written without a decimal point is represented by the compiler as an integer number. Integer constants can be written in three different notations:

- A constant starting with any digit other than 0 is interpreted as a decimal integer (that is, the standard base-10 number system). Decimal constants can contain the digits 0 through 9 and a leading minus or plus sign. (Without a leading minus or plus, a constant is assumed to be positive.)
-
- A constant starting with the digit 0 is interpreted as an octal integer (the base-8 number system). Octal constants can contain the digits 0 through 7 and a leading minus or plus sign.
- A constant starting with `0x` or `0X` is interpreted as a hexadecimal constant (the base-16 number system). Hexadecimal constants can contain the digits 0 through 9, the letters A through F, and a leading minus or plus sign.

NOTE: See Appendix C, "Working with Binary and Hexadecimal Numbers," for a more complete explanation of decimal and hexadecimal notation.

3-4-2. Symbolic Constants

A *symbolic constant* is a constant that is represented by a name (symbol) in your program. Like a literal constant, a symbolic constant can't change. Whenever you need the constant's value in your program, you use its name as you would use a variable name. The actual value of the symbolic constant needs to be entered only once, when it is first defined.

Symbolic constants have two significant advantages over literal constants, as the following example shows. Suppose that you're writing a program that performs a variety of geometrical calculations. The program frequently needs the value π (3.14159) for its calculations. (You might recall from geometry class that π is the ratio of a circle's circumference to its diameter.) For example, to calculate the circumference and area of a circle with a known radius, you could write

```
circumference = 3.14159 * (2 * radius);
area = 3.14159 * (radius)*(radius);
```

The asterisk (*) is C's multiplication operator and is covered on Day 4. Thus, the first of these statements means "Multiply 2 times the value stored in the variable radius, and then multiply the result by 3.14159. Finally, assign the result to the variable named circumference."

If, however, you define a symbolic constant with the name PI and the value 3.14, you could write

```
circumference = PI * (2 * radius);
area = PI * (radius)*(radius);
```

The resulting code is clearer. Rather than puzzling over what the value 3.14 is for, you can see immediately that the constant PI is being used.

The second advantage of symbolic constants becomes apparent when you need to change a constant. Continuing with the preceding example, you might decide that for greater accuracy your program needs to use a value of PI with more decimal places: 3.14159 rather than 3.14. If you had used literal constants for PI, you would have to go through your source code and change each occurrence of the value from 3.14 to 3.14159. With a symbolic constant, you need to make a change only in the place where the constant is defined.

C has two methods for defining a symbolic constant: the #define directive and the const keyword. The #define directive is one of C's preprocessor directives, and it is discussed fully on Day 21, "Advanced Compiler Use." The #define directive is used as follows:

```
#define CONSTNAME literal
```

This creates a constant named *CONSTNAME* with the value of *literal*. *literal* represents a literal constant, as described earlier. *CONSTNAME* follows the same rules described earlier for variable names. By convention, the names of symbolic constants are uppercase. This makes them easy to distinguish from variable names, which by convention are lowercase. For the previous example, the required #define directive would be

```
#define PI 3.14159
```

Note that #define lines don't end with a semicolon (;). #defines can be placed anywhere in your source code, but they are in effect only for the portions of the source code that follow the #define directive. Most commonly, programmers group all #defines together, near the beginning of the file and before the start of main().

How a #define Works

The precise action of the #define directive is to instruct the compiler as follows: "In the source code, replace *CONSTNAME* with *literal*." The effect is exactly the same as if you had used your editor to go through the source code and make the changes manually. Note that #define doesn't replace instances of its target that occur as parts

of longer names, within double quotes, or as part of a program comment. For example, in the following code, the instances of PI in the second and third lines would not get changed:

```
#define PI 3.14159
/* You have defined a constant for PI. */
#define PIPETTE 100
```

Defining Constants with the const Keyword

The second way to define a symbolic constant is with the const keyword. const is a modifier that can be applied to any variable declaration. A variable declared to be const can't be modified during program execution--only initialized at the time of declaration. Here are some examples:

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000, float tax_rate = 0.21;
```

const affects all variables on the declaration line. In the last line, debt and tax_rate are symbolic constants. If your program tries to modify a const variable, the compiler generates an error message, as shown here:

```
const int count = 100;
count = 200;          /* Does not compile! Cannot reassign or alter */
                    /* the value of a constant. */
```

What are the practical differences between symbolic constants created with the #define directive and those created with the const keyword? The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C programming, and they are covered on Day 9, "Understanding Pointers," and Day 12.

Now let's look at a program that demonstrates variable declarations and the use of literal and symbolic constants. Listing 3.2 prompts the user to input his or her weight and year of birth. It then calculates and displays a user's weight in grams and his or her age in the year 2000. You can enter, compile, and run this program using the procedures explained on Day 1, "Getting Started with C."

Listing 3.2. A program that demonstrates the use of variables and constants.

```
1:  /* Demonstrates variables and constants */
2:  #include <stdio.h>
3:
4:  /* Define a constant to convert from pounds to grams */
5:  #define GRAMS_PER_POUND 454
6:
7:  /* Define a constant for the start of the next century */
8:  const int NEXT_CENTURY = 2000;
9:
10: /* Declare the needed variables */
11: long weight_in_grams, weight_in_pounds;
12: int year_of_birth, age_in_2000;
13:
14: main()
15: {
16:     /* Input data from user */
17:
```

```

18:     printf("Enter your weight in pounds: ");
19:     scanf("%d", &weight_in_pounds);
20:     printf("Enter your year of birth: ");
21:     scanf("%d", &year_of_birth);
22:
23:     /* Perform conversions */
24:
25:     weight_in_grams = weight_in_pounds * GRAMS_PER_POUND;
26:     age_in_2000 = NEXT_CENTURY - year_of_birth;
27:
28:     /* Display results on the screen */
29:
30:     printf("\nYour weight in grams = %ld", weight_in_grams);
31:     printf("\nIn 2000 you will be %d years old\n", age_in_2000);
32:
33:     return 0;
34: }
Enter your weight in pounds: 175
Enter your year of birth: 1960
Your weight in grams = 79450
In 2000 you will be 40 years old

```

ANALYSIS: This program declares the two types of symbolic constants in lines 5 and 8. In line 5, a constant is used to make the value 454 more understandable. Because it uses `GRAMS_PER_POUND`, line 25 is easy to understand. Lines 11 and 12 declare the variables used in the program. Notice the use of descriptive names such as `weight_in_grams`. You can tell what this variable is used for. Lines 18 and 20 print prompts on-screen. The `printf()` function is covered in greater detail later. To allow the user to respond to the prompts, lines 19 and 21 use another library function, `scanf()`, which is covered later. `scanf()` gets information from the screen. For now, accept that this works as shown in the listing. Later, you will learn exactly how it works. Lines 25 and 26 calculate the user's weight in grams and his or her age in the year 2000. These statements and others are covered in detail in the next chapter. To finish the program, lines 30 and 31 display the results for the user.

DO use constants to make your programs easier to read.

DON'T try to assign a value to a constant after it has already been initialized.

3-5. Summary

This chapter explored numeric variables, which are used by a C program to store data during program execution. You've seen that there are two broad classes of numeric variables, integer and floating-point. Within each class are specific variable types. Which variable type--`int`, `long`, `float`, or `double`--you use for a specific application depends on the nature of the data to be stored in the variable. You've also seen that in a C program, you must declare a variable before it can be used. A variable declaration informs the compiler of the name and type of a variable.

This chapter also covered C's two constant types, literal and symbolic. Unlike variables, the value of a constant can't change during program execution. You type literal constants into your source code whenever the value is needed. Symbolic constants are assigned a name that is used wherever the constant value is needed. Symbolic constants can be created with the `#define` directive or with the `const` keyword.

Q&A

Q long int variables hold bigger numbers, so why not always use them instead of int variables?

A A long int variable takes up more RAM than the smaller int. In smaller programs, this doesn't pose a problem. As programs get bigger, however, you should try to be efficient with the memory you use.

Q What happens if I assign a number with a decimal to an integer?

A You can assign a number with a decimal to an int variable. If you're using a constant variable, your compiler probably will give you a warning. The value assigned will have the decimal portion truncated. For example, if you assign 3.14 to an integer variable called pi, pi will only contain 3. The .14 will be chopped off and thrown away.

Q What happens if I put a number into a type that isn't big enough to hold it?

A Many compilers will allow this without signaling any errors. The number is wrapped to fit, however, and it isn't correct. For example, if you assign 32768 to a two-byte signed integer, the integer really contains the value -32768. If you assign the value 65535 to this integer, it really contains the value -1. Subtracting the maximum value that the field will hold generally gives you the value that will be stored.

Q What happens if I put a negative number into an unsigned variable?

A As the preceding answer indicated, your compiler might not signal any errors if you do this. The compiler does the same wrapping as if you assigned a number that was too big. For instance, if you assign -1 to an unsigned int variable that is two bytes long, the compiler will put the highest number possible in the variable (65535).

Q What are the practical differences between symbolic constants created with the #define directive and those created with the const keyword?

A The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C programming and are covered on Days 9 and 12. For now, know that by using #define to create constants, you can make your programs much easier to read.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What's the difference between an integer variable and a floating-point variable?
2. Give two reasons for using a double-precision floating-point variable (type double) instead of a single-precision floating-point variable (type float).
3. What are five rules that the ANSI Standard states are always true when allocating size for variables?
4. What are the two advantages of using a symbolic constant instead of a literal constant?
5. Show two methods for defining a symbolic constant named MAXIMUM that has a value of 100.
6. What characters are allowed in C variable names?
7. What guidelines should you follow in creating names for variables and constants?
8. What's the difference between a symbolic and a literal constant?
9. What's the minimum value that a type int variable can hold?

Exercises

1. In what variable type would you best store the following values?
 - a. A person's age to the nearest year.
 - b. A person's weight in pounds.
 - c. The radius of a circle.
 - d. Your annual salary.
 - e. The cost of an item.
 - f. The highest grade on a test (assume it is always 100).
 - g. The temperature.
 - h. A person's net worth.
 - i. The distance to a star in miles.
2. Determine appropriate variable names for the values in exercise 1.
3. Write declarations for the variables in exercise 2.
4. Which of the following variable names are valid?
 - a. 123variable
 - b. x

- c. total_score
- d. Weight_in_#s
- e. one.0
- f. gross-cost
- g. RADIUS
- h. Radius
- i. radius
- j. this_is_a_variable_to_hold_the_width_of_a_box