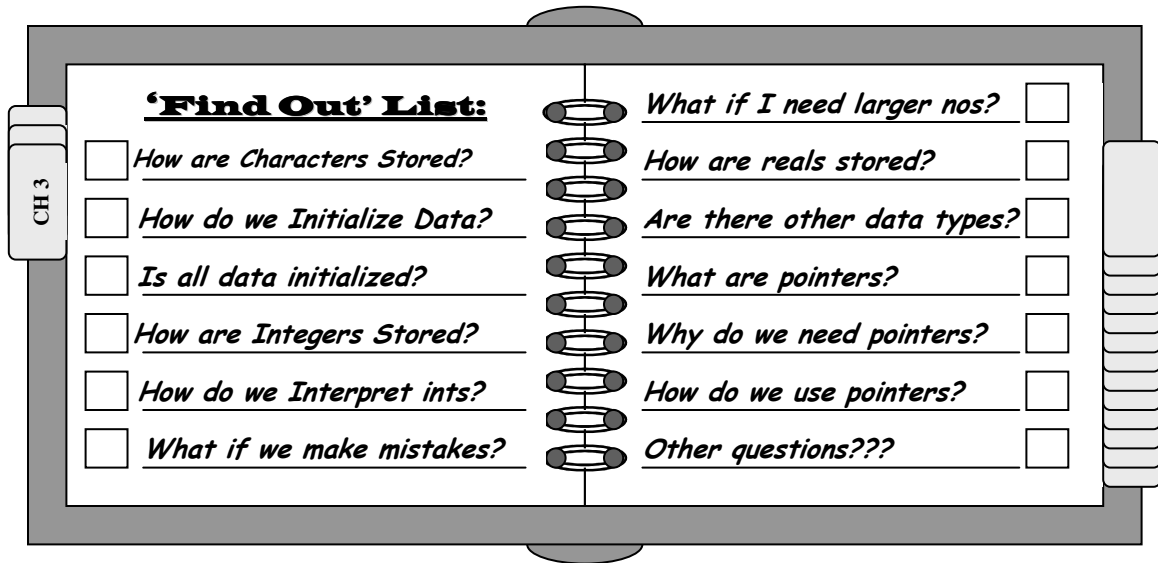


## CHAPTER 3: ALLOCATING VARIABLES TO RAM

*“All things begin in order, so shall they end”  
Thomas Browne (1605 - 1682)*



### Introduction

**D**ata structures are really nothing more than logical implementations of physically stored data. Therefore, understanding data structures assumes an awareness of how data types are stored in Random Access Memory (RAM). This chapter is intended as a (very) brief (and simplistic) description of how the basic data types are stored in RAM.

#### *How important is this?*

There is a simple axiom which we apply throughout this text:

***Give me an address and tell me what type of data is stored there, and I will tell you the value of that data type.***

This axiom applies to *ALL* data types. In fact it is the essence of programming. In the first generation of software (machine level), it was the programmer’s responsibility to assign instructions and data to specific addresses in RAM (in binary, of course). In later generations of software, through the use of mnemonics and variables, it no longer became necessary to keep track of ‘true’ memory locations. But the basic functioning of the computer has not really changed. Whenever we make reference to a variable, we still are referring to some

address in memory, even though we do not know what address that is (unless we ask the program to tell us the address).

## Storing Characters in RAM

Up to this point in time, we have described three simple data types: Characters (**char**), Integers (**int**), and real numbers (**float**).

As we have already seen, characters are generally stored as *signed* whole numbers, as are integers, BUT they only require 8-bits of storage and can therefore take on the values from  $-2^8$  to  $+2^8 - 1$  (or  $-127$  to  $+128$ ). Therefore, when we declare a character variable, we are really requesting that 8-bits (1 byte) of storage be set aside for our use.

Consider, for example, the following C/C++ declaration:

```
char a, b = 'G', c = 103;
```

C/C++ Code 3.1


We are really issuing a command to set aside 3-bytes (24-bits; 8-bits for each variable) of RAM where we can store the values of  $a^1$ ,  $b$ ,  $c$ . Additionally, we are instructing the compiler to store the value 'G' (or the numeric value 71) at location  $b$ , and the value 103 (or the ASCII character 'g') at location  $c$ .

*How are Characters Stored?*

*How do we initialize data?*

 *Where in RAM will it be stored??*

That depends on where there is space available. Don't forget, much the total space available has already been allocated for the Operating System, the c program which you are running, the environment in which it is running, and any other programs you might have previously placed into RAM. The actual location assignment will be made at run-time.

 *Will the locations assigned be contiguous (i.e., right next to each other ??)*

Possibly, but not necessarily. If contiguous space is available, then probably so. But if not, then they will be assigned wherever space is available.

For the sake of our illustration, let's assume that there is space available at addresses<sup>2,3</sup> 100, 101, and 105 (and that locations 102, 103, and 104 are presently being used for some

<sup>1</sup> When we refer to variables or c code in the body of the text, we will use *italics*. Reserved words in C will be **boldfaced**.

<sup>2</sup> These addresses are obviously made up. In fact, even a 'low-level' operating system would consume the first (lower) 64K of RAM; Windows-95/98 would consume the lower megabytes of RAM. Beyond that, the program would be stored in RAM before any variable assignments are made.

<sup>3</sup> Allocated areas are boldfaced

other purpose). The statement we made in c coded 3.1 (**char**  $a$ ,  $b = 'G'$ ,  $c = 103$ ;) might associate variable  $a$  with location 100,  $b$  with location 101, and  $c$  with location 105. If we could actually look at that section, we might see:

Figure 3.1

Address:	099	100	101	102	103	104	105
Contents:	00010101	<b>00110110</b>	<b>00110110</b>	01000111	00011000	10010000	<b>01100111</b>



*Variable  $a$ , which we associated with address 100, was not initialized. Why is there something in address 100?*

In some programming languages, if a character variable (or any other type of variable, for that matter) is assigned to an address, the address is initialized to '00000000'. Not in  $c$ . Whatever data was in that location previously remains there. It may be that location 100 previously contained a character. In that case, location 100 held the character '6' (since  $00110110_2 = 6_{10} = '6'$ ). Of course, it may be that location 100 (and 101) previously held an integer variable (**int**), which means that all we see are the 1<sup>st</sup> 8-bits of a value which was stored on 16-bits. Then again, perhaps location 100 (through 103) held a **long** data type, or a **float**. We have  *Is all data initialized?* know idea of what was previously stored there.



*Exactly what is in address 101??*

Remember, we issued the command **char**  $a$ ,  $b = 'G'$ ,  $c = 103$ ; which means that not only did we request 1-byte of storage for the (character) variable  $b$ , but we also initialized the variable with the value 'G'. In reality, we ordered that the character 'G' be stored at location 101. Therefore, the numeric value 71 ( $71_{10} = 110110_2$ ), which corresponds to the ASCII character 'G', was stored.

Similarly, address 105 contains the numeric value 103 ( $103_{10} = 1100111_2$ ) which corresponds to the ASCII character 'g'.

Notice that since only 8-bits are allocated for characters, only 256 ( $= 2^8$ ) values are available. Because in  $c$ , the data type **char**, by default, is the same as the data type **signed char**, the legal values which can be assigned range from  $-2^7$  to  $+2^7 - 1$ , or -128 to +127.



*What would happen if an illegal decimal value, such as 320 or -150, were entered??*

Again, in some programming languages, the program would either not compile (if the value was entered in the source code), or would generate an error when running. Not in  $c$ . The  $c$  language will accept these values, and try to place their binary equivalents in the 8-

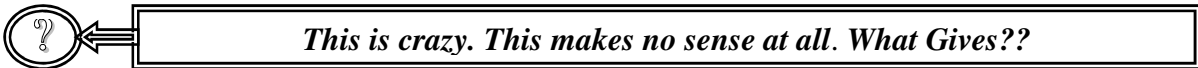
bits available, according to what we know about how values are stored, but strange things can happen.

Calculation 3.1.

$$\begin{array}{r}
 11100110 \quad \leftarrow \text{Binary Value Stored} \\
 0011001 \quad \leftarrow \text{One's Compliment} \\
 + \quad \quad \quad \underline{1} \\
 0011010 \quad \leftarrow \text{Two's Compliment}
 \end{array}$$

Take for example the integer 320. Because the largest number which can be stored is 255, entering anything larger will cause an overflow. Given that the number  $320_{10} = 101000000_2$  (using 9-bits;  $01000000_2$ , using 8-bits, and  $0000000101000000$  using 16-bits), only the last 8-bits will be stored ( $01000000_2 = 64_{10}$ ), which corresponds to the ASCII character '@'.

Assigning the value 320 actually turns out to be a simple example. A more difficult one is trying to store the number 230, for example. The number  $230_{10} = 11100110_2$  requires 8-bits of storage. If we store this number on the 8-bits available, however, it will appear to be a negative number (since the first bit is '1'). The value  $11100110_2$  will be interpreted as the binary number  $11010_2$  (see Calculation 3.1.) which equates to the decimal value  $-26 (= -2^4 + -2^3 + -2^1 = -16 + -8 + -1)$ . This, of course, is interpreted as the ASCII character 'µ' which corresponds ASCII 230.



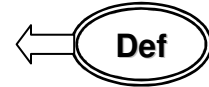
Yes, it does initially seem bizarre. In the c programming language, even though the value is stored as  $-26_{10}$ , printing the character value  $-26_{10}$  as a character would in fact print the ASCII character 'µ', which is ASCII 230. When converting to ASCII characters (using the “%c” format), the entire 8-bits are considered on an *unsigned* basis. If we were to attempt to print out the decimal value (using the “%d” format), however, we would find that the value is indeed  $-26_{10}$ . It is just one of the quirks which c programmers must get used to.

<p>Calculation 3.2.</p> $  \begin{array}{r}  0,000000010010110 \quad \leftarrow \text{Binary of } 150_{10} \\  1,11111101101001 \quad \leftarrow \text{One's complement of } 150_{10} \\  + \quad \quad \quad \underline{1} \\  1,11111101101010 \quad \leftarrow \text{Two's complement of } 150_{10}  \end{array}  $	<p>Determining how <math>-150</math> would be stored also requires the application of what we have learned about storing negative integers, although it is a little more</p>
--	--

straightforward. For example, the number  $150_{10}$  would be stored as  $0,000000010010110_2$  (using 16-bits). Using two's complement, the number would be stored as  $111111101101010$

Once again, if we consider only the lower 8-bits, the numeric value  $01101010_2 = 106_{10}$  would be stored. If we check the ASCII tables, we find that this corresponds to the character 'j'.

One more note about the data type **char** in c. The c programming language allows for an **unsigned char** data type. As with other *unsigned* data types, this means that the legal values which can be assigned are 0 to  $2^8 - 1$ , or 0 to 255<sup>4</sup>.



### *What's the advantage??*

Not much, really. However, it circumvents the confusion which we experienced, for example, when we tried to assign the value  $230_{10}$  to a variable. If we had declared the variable type **unsigned char** *ch*, and assigned the value 230 to it (*ch* = 230;), attempting to print out the decimal value (`printf("%c",ch);`) would yield the value 230; printing the ASCII character equivalent (`printf("%c",ch);`) would still yield the outcome 'μ'.

In the previous chapter's C Programming Assignments, we provide some source code which illustrates some of the issues we have been describing. Try running them. Change the values we have given. It might even be fun. It will definitely illustrate the concepts discussed above.



### *What about strings, or collections of characters??*

A string is actually an abstract data type, although some programming languages make it seem as if they are basic data types. We will discuss strings in Chapter 5.

## Storing Integers (the data type **int**) in RAM

The *only* difference between the way characters and integers are stored in RAM is that integers (data type **int**) require 2-bytes (16-bits) of storage. Assume that we make the c program declaration:

✓ *How are Integers Stored?*

```
int a, b = 'G', c = 1543;
```

C/C++ Code 3.2

While this statement superficially appears to be the same as our previous (character) declaration note the differences:

#### Differences Between Character and Integer Declarations

1. We have declared the variables *a*, *b*, and *c* as integers (data type **int**), meaning that they require 2-bytes (16-bits) of storage each.
2. We are requesting a total of 6-bytes (48-bits; 16-bits per variable) be set aside in RAM at locations *a*, *b*, and *c*.

<sup>4</sup> In some programming languages, such as Pascal, this data type is a numeric byte

This time, assume that locations 100, 101, 102, 105, 106, 110, and 111 are available (i.e., 104, 107-109 are being used), as shown in Figure 3.2. In this case, variable *a* will be assigned address 100 (and 101), variable *b* will be assigned address 105 (and 106), and variable *c* will be assigned address 110 (and 111), and initialized with the value 1,543.

Figure 3.2.

100	101	102	103	104	105	106
Available			In Use		Avail.	
106	107	108	109	110	111	112
In Use				Available		



**Why won't variable *b* be assigned address 102? It's available and it is the next one on the list.**

True, but integers require 2 *contiguous* bytes of memory. Because address 103 is not available, the value of *b* can not begin in address 102.

This time, if we could look inside the relevant section of RAM, after the variable locations have been assigned, we might see the 'donuts' set according to the layout in Figure 3.3.

Figure 3.3.

Address:	100	101	102	103	104	105	106
Contents:	11010101	01110100	00110110	01000111	00011000	00000000	00110110
Address:	107	108	109	110	111	112	113
Contents:	01100111	01110111	00111111	00000110	00000111	00110000	00000000



**Why is variable *a* assigned addresses 100 and 101??**

Remember, integers require 2-bytes of *contiguous* storage. We could not, for example, place part of the integer in location 100, and the other part in location 104. This holds true for all basic data types. Real numbers (data type **float**) must be placed in four contiguous bytes, data type double requires eight contiguous bytes, and so forth.

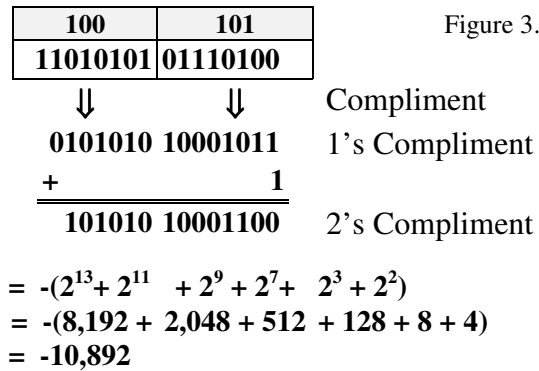


Further, when we refer to a variable's address, we will refer to its *base address*, meaning the location at which that variable starts.



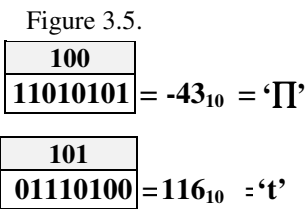
**Does variable *a* (location 100 and 101) also have a value associated with it??**

Yes, even though we have not (yet) stored anything there, whatever was previously stored there is still there. Because we declared that we expect to find the data type **int** there, when we analyze locations 100 and 101 (variable *a*), we find that it contains a negative integer. Consequently, if we were to attempt to print out the contents of the location, we would first have to compliment it, and then evaluate. When we do, we find the integer value -10,892.



Of course, it is possible that location 100 previously contained a character, as did location 101. In that case, prior to running our program, location 100 contained the numeric value  $-43_{10}$  (or the ASCII character 'Π'), and location 101 contained the numeric value  $116_{10}$  (or the ASCII character 't').

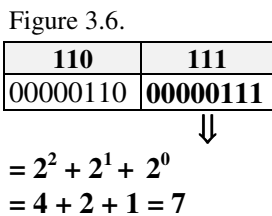
Of course, it could be that location 100 previously contained a character, and locations 101 and 102 contained an integer ( $0111010000110110_2 = 29,750_{10}$ ), or perhaps locations 100 through 103 previously contained the data type **float**. Or it could have been some other combination. We have no way of knowing.



*How do we interpret Integers?*

*How can the character 'G' be assigned to the integer variable b?*

*Why not?* Remember, the ASCII character 'G' is nothing more than the numeric value 71 ( $71_{10} = 110110_2$ ). The only difference is that integers require 16-bits, and hence the value  $110110_2$  is stored as  $0000000000110110_2$  (with  $00000000$  in address 105 and  $00110110$  stored in address 106).



We could also print out an integer as a character. For example, we know that variable *c* (stored at addresses 110 and 111) contains the value 1543. When we try to print it as a character, we in fact look at the rightmost 8-bits, determine the numeric value stored there, and then print out the corresponding ASCII character. In this case, of course, the Bell would ring (check your ASCII Tables).

C Code 3.3

```
printf("%c", c);
```

*What happens if we assign illegal values, for example the values 45,250 or -50,000, to any of the integer variables?*

Figure 3.7.

100	101
10110000	11000010

$\Downarrow$                        $\Downarrow$   
 1001111 00111101  
 +                                      1  
 -----  
 1001111 00111110  
 = -20,286<sub>10</sub>

Figure 3.7.

100	101
10110000	11000010

$= 2^{15} + 2^{13} + 2^{12} + 2^7$   
 $+ 2^6 + 2^1$   
 $= 32,768 + 8,192$   
 $+ 4,096 + 128$   
 $+ 64 + 2$   
 $= 45,250_{10}$

Basically, the same thing that happened when we tried to store illegal character values at our character locations. Assume we tried to store the value 45,250<sub>10</sub> (=1011000011000010<sub>2</sub>) at location *a* (See Figure 3.7.). Since the first bit is '1', the number would appear to be negative. To evaluate, we would first take the two's complimented number (0100111100111101 + 1 = 0100111100111110) and then determine that the value is -20,286<sub>10</sub> is stored. A similar situation would occur if we tried to store the value -50,000<sub>10</sub> at location *a* (See Calculation 3.3.) Since the value 50,000<sub>10</sub> = 1100001101010000<sub>2</sub> then to store -50,000<sub>10</sub> we first need to compliment. The 2's complimented binary equivalent would be stored as 0011110010101111<sub>2</sub> + 1<sub>2</sub> = 0011110010110000<sub>2</sub>. After evaluation, we find that the value stored is actually 15,536<sub>10</sub>.

Calc. 3.3.

11000011	01010000
$\Downarrow$	$\Downarrow$
01111100	10101111
+	1
-----	
01111100	10110000
= 15,536 <sub>10</sub>	

✓ *What Happens if I make mistakes?*



*What if I really need to store integer values such as 45,250 or -50,000, or values even greater??*

### Storing Other Integer Types in RAM

Remember, the c programming language allows for additional integer data types. Remember also that we are talking about the PC. On a mainframe, the data type **long**, for example, may use 64-bits, and therefore take on much larger values.



- Additional Integer Data Types in C**
1. **unsigned integers.** This allows the storage of all values from 0 to 65,535 ( $2^{16} - 1$ )
  2. **longs.** This allows the storage of all integers from -2,147,483,648 to 2,147,483,647 ( $-2^{31}$  to  $2^{31} - 1$ )
  3. **unsigned longs.** This allows for the storage of all values from 0 to 4,294,967,296 ( $2^{32} - 1$ )



*How are unsigned integers stored in RAM ??*

Basically, the same as integers, except that we never have to worry about complimenting.

C Code 3.4.

```
unsigned int a, b = 'G', c = 1543;
```



For example, suppose we issue the declaration given in C Code 3.4. This declaration is basically the same as the code in 3.2., except that we are declaring the variables to be of type **unsigned int**. If we further assume that the same addresses are assigned to each of the variables (see Figure 3.3.), then the only difference will occur when we examine the value associated with variable *a* (which is not initialized). As we saw, when we declared *a* to be of data type (signed) **int**, the value -10,892 was stored at this address. This time, we would find that the value 45,250<sub>10</sub> is stored at location *a*.

**How is the data type long stored in RAM ??**

The data type **long** corresponds to the same rules as the data type **int** (whether **signed** or **unsigned**), except that it requires 4-bytes (32-bits) of contiguous storage per variable. As with the data type **int** (and the data type **char**) a **long** variable is signed by default.

Consider the C Code:

```
long d = 423575; C Code 3.5.
```

This is a legal value for the data type **long** which can take on values between  $-2^{31}$  to  $+2^{31} - 1$  (or -2,147,483,648 to +2,147,483,647). In binary,  $423,575_{10} = 1100111011010010111_2$  on 19-bits or 00000000000001100111011010010111 on 32-bits. If we found that address 500 were available (which also means that addresses 501, 502, and 503 must also be available), the relevant portion of RAM might appear as:

Figure 3.8.

Address:	498	499	500	501	502	503	504
Contents:	11010101	01110100	<b>00000000</b>	<b>00000110</b>	<b>01110110</b>	<b>10010111</b>	01100111

$$\begin{aligned}
 &= 2^{18} + 2^{17} + 2^{14} + 2^{13} + 2^{12} + 2^{10} + 2^9 + 2^7 + 2^4 + 2^2 + 2^1 + 2^0 \\
 &= 262,144 + 131,072 + 16,384 + 8,192 + 4,096 + 1,024 + 512 + 128 + 16 + 4 + 2 + 1 \\
 &= 423,575
 \end{aligned}$$

All of the other considerations which we applied to characters and integers also apply:

**long variable Considerations**

If we attempt to store a negative **long** integer, or if we find that the left-most bit (in this case, the 31<sup>st</sup> bit) is '1' in a binary representation, we must compliment the other 31-bits to determine the value of the integer.

1. If we attempt to store values which require more than 32-bits to represent, the rightmost 32-bits will be stored.
2. If we attempt to print out a long variable as the data type **int**, only the rightmost 32-bits will be considered.
3. If we attempt to print out the data type **long** as a character, only the rightmost 8 bits will be considered.



**What if I need even larger values ??**

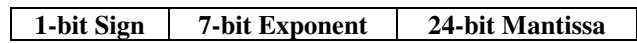
That depends. Don't forget, there is the data type unsigned long, which would allow you to store integers in the range 0 to  $2^{32} - 1$ , or 0 to 4,294,967,295. However, if larger values are needed, or negative values are needed, we must store them as floating-point (real) numbers using the data type float. This probably means, however, that we will lose some level of precision.

*What if I need larger nos?*

**Storing Floating-Point Numbers in RAM**

As we saw in Chapter 2, floating point numbers in the PC require 32-bits of contiguous RAM storage, and are generally (not universally) arranged according to the layout in Figure 3.9.

Figure 3.9.



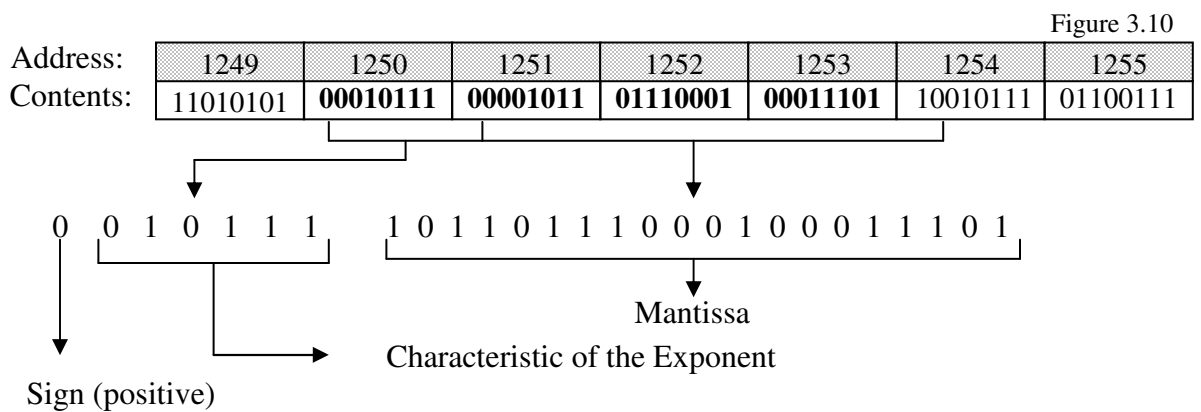
**CAVEAT**

As we noted in Chapter 2., we are taking some liberties in describing the data type float (and double, and long double) for the purpose of simplification. Interested students are advised to refer to Addendum 2.1.

Because we did not (intentionally) go over conversion of floating-point numbers into binary, true representation in RAM is difficult. However, if we were to make the statement shown in C Code 3.6, and we found that the *base address* of variable *myfloat* were actually 1250, and if we were to look at RAM (see Figure 3.10.), we would interpret the sequence of bits we found according to the layout given in Figure 3.9. (1-bit sign, 7-bit Characteristic of the Exponent, and 24-bit mantissa).

**float myfloat;** C Code 3.6.

*How are reals stored?*





### What about the other real data types??

Again, storage corresponds to the manner in which we defined the bit patterns. Since this varies from package to package, we can't be sure. The main differences, however, are:

#### Storing floating-point numbers in C

1. The data type **float** (single precision real) requires 4 contiguous bytes (32-bits) of storage.
2. The data type **double** requires 8 contiguous bytes (64-bits) of storage. ANSI C requires a minimum of 10 digits of precision, meaning that at least 34-bits must be assigned to the mantissa (leaving a maximum of 31-bits to the characteristic).
3. The data type **long double** requires 16 contiguous bytes (128-bits) of storage. ANSI C requires a minimum of 10 digits of precision, meaning that at least 34-bits must be assigned to the mantissa (leaving a maximum of 53-bits to the characteristic).

There is one other data type that we frequently store in RAM. *Are there other data types?*

### Storing Addresses in RAM: An Introduction to Pointers

It may seem like a strange concept at first, but in the c programming language, not only are variable values stored in RAM, but RAM addresses are themselves stored. Addresses



are stored in a variable type called a **pointer**. A pointer, just like every other data type, is a location in RAM.

*What is a pointer?*



### Why??

As we have seen, *everything* is stored in RAM. If we want to manipulate data, we need to be able to access it directly, which means determining its address. Once we know an address (and the type of data contained in it), we can readily access it and, if we wish, change it. This corresponds to our original adage *Give me an address, tell me what type of data is stored there, and I will give you the value of that data.*

This also brings us to another programming distinction which must be made: *Call by Reference* vs. *Call by Value*. When we pass variable values between functions in c, we can either pass the contents of some location in memory (call by *value*), or we can pass the address which contains the data (call by *reference*). If we call by value, the contents of the address can *not* be changed. If we call by reference, we *can* change whatever

*Why do we need pointers?*

is stored at that address.

Let's take a look at two simple c programs, which may look as if they perform the same tasks, but in fact, operate quite differently. In both cases, we will pass an integer variable (variable *number* in function *main*) to function *multiply*, which will place it in location *numb*, and multiply it by 5. In c Code 3.7., we will pass the contents of the variable. In c Code 3.8., we will pass the address of variable *number*. To help illustrate the differences between the two programs, we will print the value of the variable (*number*) before we pass it, after we pass it to the function, after the function manipulates it, and after the function transfers control back to the main function.

### C Code 3.7.

```
// C program 3.7: Call by value
#include <stdio.h>
void multiply(int numb);           // function prototype

void main()
{
    int number = 6;                // #1: this is location number in function main
    printf("Before calling function multiply, the value of number is %d\n", number);
    multiply(number);
    printf("After returning from function multiply, the value of number is %d\n", number);
}

void multiply (int numb)          // #2: this is location num in function multiply
{
    printf("In Function multiply, the initial value passed is %d\n", numb);
    numb = numb * 5;
    printf("In Function multiply, after multiplying by 5, the value is %d\n", numb);
}
```

The output from the C Code 3.7. would be:

### C Code 3.7. Output

```
Before calling function multiply, the value of number is 6
In Function multiply, the initial value passed is 6
In Function multiply, after multiplying by 5, the value is 30
After returning from function multiply, the value of number is 6
```

Notice that the original value (6 in function *main*) is correctly received by function *multiply* (“In Function multiply, the initial value passed is 6”). The variable *number* (in function *multiply*) is correctly received (“In Function multiply, the initial value passed is 6”), and is also changed correctly (“In Function multiply, after multiplying by 5, the value is 30”). However, when control is returned to *main*, we see that the original value has not been changed (“After returning from function multiply, the value of number is 6”).

 **What happened??**

Let's see how data was stored in RAM and what happens to it. Let's assume that addresses 34000 to 35000 are available to us. If that is true, then in function `main`, we can assign base address 34000 (meaning addresses 34000 and 34001) to variable `number`. In function `main`, when we issue the command `int number = 6`; we are placing the value  $6_{10}$  ( $110_2$ ) in location 34000, and RAM would appear as it does in Figure 3.11.

Figure 3.11.

34000	34001	34002	34003	34004	34005
00000000	00000110	01001011	11000101	0111010	11110100

Now when we issue the command `multiply(number)`; we are passing the contents of address 34000 (variable `number`) and placing them into location `numb` (let's assume integer variable `numb` will be assigned the base address 34002. RAM would appear as it does in Figure 3.12.


Figure 3.12.

34000	34001	34002	34003	34004	34005
00000000	00000110	00000000	00000110	0111010	11110100

After we issue our command `numb = numb * 5`; we will change the contents of location 34002 (and 34003), or variable `numb`. We have not altered the contents of 34000 (and 34001), so that when we return control to function `main`, variable `number` will still the value 6. RAM will appear as it does in Figure 3.13. after the multiplication and after our return. The only difference, of course, is that we can no longer access addresses 34002 and 34003 once we return to function `main`, any more than we could access variable `number` when we transferred control to function `multiply`.

Figure 3.13.

34000	34001	34002	34003	34004	34005
00000000	00000110	00000000	00001111	0111010	11110100

 **Why can't we access variable `number` from function `multiply`, or variable `numb` from function `main`?? We have been working under the axiom "Tell me an address, and the type of data stored there, and I will give you the value of that data"**

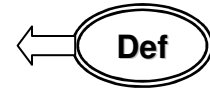
The problem is that **we do not know** the address of variable `numb` when function `main` has control of execution, nor do we know the address of variable `number` when we transfer control to function `multiply`. When we call a function, all of the variables are assigned addresses at the time of the call, not before.

*How do we use pointers?*

 **That seems like a lot of work. Isn't it??**

Yes and no. It does require a little more effort to assign function variables each time we call the function. On the other hand, it does make efficient use of RAM since when we leave a function, the RAM that was being used is freed-up for use by other functions.

By default, variables in C are *local* or *automatic* variables. It is possible to declare variables whose addresses are available to all the functions in a program (although it isn't considered good form). C allows for three different types of variable declarations:



#### Variable Classes in C

1. **Automatic Variables:**
  - Declared *INSIDE* a function
  - Exists *ONLY* for the duration of the function
  - When done, memory allocation freed
2. **External Variables:**
  - Declared *BEFORE* the function **main**
  - Known to *ALL* functions
  - Do Not Expire when a particular functions ends
  - Initialized when declared
3. **Static Variables:**
  - Like Automatic Arrays, local to the function
  - Like an External array, retains values between calls & is initialized at declaration
  - Declared with the reserved word **static** before the data type declared (e.g., **static int a;**)



***You mentioned Call by Reference before. How is that different??***

In the case of *call by reference*, we are passing the address which contains the data we wish to manipulate. Whatever changes we make to the data will be stored in the original location. This could be done in a number of ways; the (slightly modified) c program (see C Code 3.8.) illustrate one manner of passing an address. The output is given in C Code 3.8. Output.



There is one new variable type in this program. Some texts refer to it as a new data type, although actually, it is the data type integer with a slight variation (we won't go into this now). It is the variable type pointer, and it is declared as **int newnumber** (in our function prototype and the function *multiply* itself). In this statement, the asterisk (\*) in front of the variable name *newname* indicates that location *newname* will store an address, and the use of the reserved word **int** indicates that we expect to find the data type integer stored at that location.

A pointer consists of three components:

#### Pointer Components

1. A variable (which we know is actually a location in RAM) which will store an address<sup>5</sup>
2. The address to be stored (which is stored on 4-bytes of RAM), and
3. The data type which will be stored at that address

When we make the reference `int *newnumber` we are stating that variable (location) `newnumber` will hold the address of an integer (in this case, the address of variable `number`)

#### C/C++ Code 3.8.

```
// C Program 3.8: Call by reference
#include <stdio.h>
void multiply(int *newnumber);

void main()
{
    int number = 6;
    printf("Before calling function multiply, the value of number is %d\n", number);
    multiply(&number);
    printf("After returning from function multiply, the value of number is %d\n",
number);
}

void multiply (int *newnumber)
{
    printf("In Function multiply, the initial value passed is %d\n", *newnumber);
    *newnumber = *newnumber * 5;
    printf("In Function multiply, after multiplying by 5, the value is %d\n", *newnumber);
}
```

#### C Code 3.8. Output

```
Before calling function multiply, the value of number is 6
In Function multiply, the initial value passed is 6
In Function multiply, after multiplying by 5, the value is 30
After returning from function multiply, the value of number is 30
```

Notice that the MAJOR difference in the output (from the call by value program (i.e., C Code 3.7.)) is in the last line of output produced. Previously, we produced the output:

<sup>5</sup> In older DOS versions, pointers required 2-bytes of RAM (near pointers). In Windows 95/98 based environments, addresses require 4-bytes of storage. Further, addresses generally require 2 components: a data *segment*, and the data segment *offset*. However, rather than get into a lengthy discussion of segments and offsets, we will assume that pointers are stored as unsigned long data types (on 4-bytes of storage). This also corresponds to our assertion that data is usually corresponds to the high-end of memory (and allows to use larger address numbers).

***After returning from function multiply, the value of number is 6***

Whereas now, we produced the output:

***After returning from function multiply, the value of number is 30***


In the first case (call by value), we *DID NOT* change the contents of location (variable) *number* (address 3400). In the second case (call by reference), we *DID* change the contents of location *number*.

Figure 3.14.


Let's once again track the changes each of the commands make to the way the RAM would appear. Up until we call the function multiply, RAM would appear exactly as it did before (we duplicate the appearance in Figure 3.14., although it appears the same in Figure 3.11)

34000	34001	34002	34003	34004	34005
00000000	00000110	01001011	11000101	0111010	11110100

This time, when we call function multiply, we pass the address at which we will find our integer value (i.e., the base address 34000).


 ***Why don't we just pass the variable number??***

Essentially, that is exactly what we are doing. Remember, a variable is merely a location in RAM. When we pass the address we are indicating where that address is. Remember also, we can not refer to variable *number* from any function except *main*, so really we are just finding away to get around that.

 ***How do we pass an address??***

The expression *&number* (used in function *main*) means the *address* of variable *number* (in our case, address 3400). Placing an ampersand (&) in front of any variable name implies that we are referring to the address at which we storing our data, NOT the data we will find there.

Note also that an address is a constant term, meaning we can not change its value (anymore than we can change the address of the house we live in; we can change our home address, but only if we move. The address of the house remains the same. We pass the address of variable *number* (using the expression *&number*) to function *multiply*, which receives it into an integer *pointer newnumber* (expressed as *\*newnumber*).

 ***Integer Pointer newnumber ??? What does that mean ???***



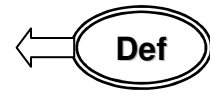
Remember, it is insufficient to merely pass an address without indicating what type of data we

Figure 3.15.

34000	34001	34002	34003	34004	34005
00000000	00000110	00000000	00000000	10000100	11010000

expect to find there. When we pass the address 34000 to function *multiply* (placing it at address *newnumber*, which we assume will be assigned the base address 34002 (through 34005, since it requires 4-bytes of storage)), we are also indicating that address 34000 will contain an integer (which it does, the variable *number*). After we pass the address, RAM might appear as it does in Figure 3.15. If we analyze the contents of location *newnumber*, we find that we have stored the integer value  $34000_{10}$  ( $= 1000010011010000_2$ ) there<sup>6</sup>.

In function *multiply*, we wish to manipulate the integer 6 (which we know is stored at address 34000), and NOT the address 34000. Therefore we need to apply some new notation. We need to go to the address contained at location *newnumber* to do our calculations, or we need to **redirect** to that address. In C, we redirect by placing the symbol `*` in front of the pointer variable. In this case, the command `*newnumber` implies that we wish to manipulate the integer which we expect to find at location 34000.



When we issue the command:

Figure 3.16.

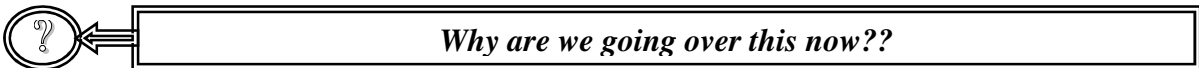
`*newnumber = *newnumber * 5;`  
we are really ordering that the

34000	34001	34002	34003	34004	34005
00000000	00011110	00000000	00000000	10000100	11010000

contents of the address contained in variable *newnumber* be multiplied by 5, and the result stored at the address stored in location *newnumber*. In other words, go to address 34000 multiply the integer we will find there by 5 (yielding 30), and place the result back into address 34000. At the end of the operation, and after we return control to function **main**, RAM would appear as it does in Figure 3.16.

Notice that this time, we **did** actually alter the contents of variable *number* in function **main**.

We know that pointers are (at first) very confusing. In point of fact, we do not expect you to fully understand them right away.



This was an *introduction* to pointers. You will be seeing them more and more from now on. By the time we get to Sections 3 and 4, essentially *ALL* of the data structures introduced will rely on pointers. *Get used to it.*

<sup>6</sup> We are ‘cheating’ a little here. See the previous footnote.

## Determining Pointer Contents

The C programming language allows us to print out the addresses at which we store our variables the use of the **%p** print specifier (see C Code 3.7.) The **p** specifier prints the input argument as a pointer; the format depends on which memory model was used. It will be either XXXX:YYYY (base segment: offset) or YYYY (offset only). Consider the code given in C Code 3.9.



C Code 3.9.

```
#include <stdio.h>
int main()
{
    char ch1, ch2;
    int int1, int2;
    long long1, long2;
    float float1, float2;
    double double1, double2;
    printf("character addresses:  %p  %p\n", &ch1, &ch2);
    printf("integer addresses:    %p  %p\n", &int1, &int2);
    printf("long addresses:       %p  %p\n", &long1, &long2);
    printf("float addresses:       %p  %p\n", &float1, &float2);
    printf("double addresses:      %p  %p\n", &double1, &double2);
}
```

The output from this program might appear as<sup>7</sup>:

C Code 3.9. Output

```
character addresses:  1C57:0FFF  1C57:0FFE
integer addresses:   1C57:0FFC  1C57:0FFA
long addresses:      1C57:0FF6  1C57:0FF2
float addresses:     1C57:0FEE  1C57:0FEA
```

The output actually consists of two components: the base segment address and the offset are given in hex (we are not about to go into too much detail at this time). The base segment address refers to that portion of RAM where the data is stored; the offset refers to how far into that portion of that segment a particular piece of data can be found. Notice, that the complete address does consist of 4-bytes (32-bits) since each component requires 16-bits or 2-bytes (the largest value  $FFFF_{16} = 65535_{10} = 1111111111111111_2$ ). For the sake of clarity, if we converting the hex addresses to decimal, we would find:

<sup>7</sup> This is the actual output from a run; the addresses obviously vary each time the program is run.

**Transformed C Code 3.9. Output**

character addresses:	7255:4095	7255:4094	(1-byte per variable)
integer addresses:	7255:4092	7255:4090	(2-bytes per variable)
long addresses:	7255:4086	7255:4082	(4-bytes per variable)
float addresses:	7255:4078	7255:4074	(4-bytes per variable)
double addresses:	7255:4066	7255:4058	(8-bytes per variable)

Notice that (this time), the variables were allocated to RAM in a contiguous fashion (although in reverse order - from highest to lowest). If we were to look in memory, we would find that the variables were allocated as (substituting offset addresses as actual addresses):

Figure 3.17.

4058	4059	4060	4061
Variable <i>double2</i> (through address 4065)			
4062	4063	4064	4065
4066	4067	4068	4069
Variable <i>double1</i> (through address 4073)			
4070	4071	4072	4073
4074	4075	4076	4077
Variable <i>float2</i> (through address 4077)			
4078	4079	4080	4081
Variable <i>float1</i> (through address 4081)			
4082	4083	4084	4085
Variable <i>long2</i> (through address 4085)			
4086	4087	4088	4089
Variable <i>long1</i> (through address 4089)			
4090	4091	4092	4093
Variable <i>int2</i>		Variable <i>int1</i>	
4094	4095		
Variable <i>ch1</i>	Variable <i>ch2</i>		

## Summary

In this chapter, we have tried lay down the foundations for understanding data structures. All data structures are simply logical implementations of physical data stored in RAM.

The mechanics of implementing data structures are performed by the c compiler, but unless we have some idea of the data it is working with, the probability of misusing a data structure increases.

We have seen how the various data types are stored in RAM. While the computer groups data together on 8-bits, it makes no attempts to try and make any meaning of what it stores; that is also the function of the c compiler. It is for that reason that we can make mistakes which will produce results, sometimes obvious, and sometimes subtle. In order to detect these mistakes, we need to be aware of the types of mistakes which can be made, and how we can go about preventing them.

This concludes our section on fundamental concepts. The material covered here, if understood, should be more than enough to carry the student through the subsequent sections of the text. Confusion about these basic concepts, however, may lead to difficulties in understanding the following sections. For this reason, we strongly suggest that the student contact their instructor for any clarifications which might be needed.

In the following chapters, we will introduce a number of new abstract data structures. However, we will still make reference to the physical storage of data in RAM in order to show how the structures are applied, and how their misuse can be avoided.

## Chapter Terminology: Be able to fully describe these terms

Address	Data segment offset	Integer storage
Automatic variables	Double addresses	Location contents
Call by reference	External variables	Long addresses
Call by value	Float addresses	Long double addresses
Character addresses	Floating-point storage	Pointers
Character storage	'Garbage'	Random access memory
Contiguous space	Initialization	Redirection
Data segment	Integer addresses	Static variables

## Review Questions

1. How many bytes of contiguous storage are required for each of the following c data types:
  - a. **char**
  - d. **int**
  - b. **double**
  - e. **long**
  - c. **float**
  - f. **long double**

**For the following question, assume that RAM appears as given in figure 3.18:**

2. Suppose that we find that the following variables can be associated with the given locations in RAM (note: this is a 2's compliment machine):

<u>Variable</u>	<u>data type</u>	<u>RAM location</u>
vara	char	4060
varb	char	4063
varc	int	4065
vard	signed int	4068
vare	unsigned int	4072
varf	long	4077
varg	unsigned long	4084

- a. What is the decimal value of **vara**?
- g. What is the decimal value of **vard**?
- b. What is the ASCII character for **vara**?
- h. What is the ASCII character for **vard**?
- c. What is the decimal value of **varb**?
- i. What is the decimal value of **vare**?
- d. What is the ASCII character for **varb**?
- j. What is the ASCII character for **vare**?
- e. What is the decimal value of **varc**?
- k. What is the decimal value of **varf**?
- f. What is the ASCII character for **varc**?
- l. What is the decimal value of **varg**?

Figure 3.18

4058	4059	4060	4061
00010010	11001011	01011001	00010010
4062	4063	4064	4065
11010001	10011100	00100001	00100100
4066	4067	4068	4069
00001111	00000001	11110101	00011101
4070	4071	4072	4073
00110011	00100100	10001001	11011001
4074	4075	4076	4077
11101101	00111011	10001001	11111110
4078	4079	4080	4081
10110111	11001001	01101001	11000100
4082	4083	4084	4085
00100100	11111100	11000001	00000000
4086	4087	4088	4089
01001001	00000001	00011010	10001001

*Any Other Questions?*



**Review Question Answers (NOTE: CHECKING THE ANSWERS BEFORE YOU HAVE TRIED TO ANSWER THE QUESTIONS DOESN'T HELP YOU AT ALL)**

1. Number of bytes of contiguous storage for data type:

- |                          |                            |
|--------------------------|----------------------------|
| a. <b>char</b> : 1 (one) | d. <b>int</b> : 2          |
| b. <b>double</b> : 8     | e. <b>long</b> : 4         |
| c. <b>float</b> : 4      | f. <b>long double</b> : 16 |

2.a. The decimal value of *vara* given *vara* is of datatype **char** and has the address **4060**:

$$\text{Contents of } \mathbf{4060}: \mathbf{01011001} = 2^6 + 2^4 + 2^3 + 2^0 = 64 + 16 + 8 + 1 = \mathbf{89}$$

b. The ASCII character associated with *vara* is: 'X'

c. The decimal value of *varb* given *varb* is of datatype **char** and has the address **4063**:

Contents of **4063**: **10011100** BUT since *varb* is of data type (signed) char, we must compliment:

$$\begin{array}{r} 10011100 \Rightarrow 01100011 \\ + \quad \quad \quad 1 \\ \hline 01100100 = -(2^6 + 2^5 + 2^2) = -(64 + 32 + 4) = \mathbf{-100} \end{array}$$

d. Since the ASCII character is determined according to all 8 bits:

$$10011100 = 2^7 + 2^4 + 2^3 + 2^2 = 128 + 16 + 8 + 4 = 156$$

which is associated with the ASCII character **f**

e. The decimal value of *varc* given *varc* is of datatype (signed) **int** at the address **4065** (& 4066):

$$\begin{aligned} 0010010000001111 &= 2^{13} + 2^{10} + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 8,192 + 1,024 + 8 + 4 + 2 + 1 = \mathbf{9,231} \end{aligned}$$

f. The ASCII character is determined using only the right-most 8 bits of an integer:

$$00001111 = 2^3 + 2^2 + 2^1 + 2^0 = 8 + 4 + 2 + 1 = \mathbf{15}$$

or the char NAK (Neg, Acknowl).

g. The decimal value of *vard* given *vard* is of datatype **signed int** at the address **4068** (& 4069):

1111010100011101 => The number is negative, so we must compliment:

$$\begin{array}{r} 1111010100011101 \Rightarrow 0000101011100010 \\ + \quad \quad \quad \quad \quad \quad \quad 1 \\ \hline 0000101011100011 \end{array}$$

