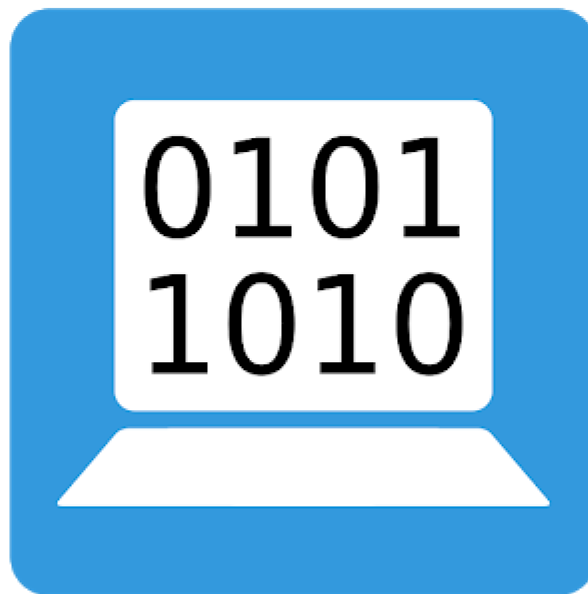
Chapter 2

Data Representation

What is in This Chapter ?

This chapter explains how data is represented in memory. It begins with an explanation of how bits are stored using various **bit models** (i.e., **magnitude only**, sign magnitude, **two's compliment**, fixed point, **floating point**, **ASCII** and **UNICODE**). It then discusses bit operators that help us understand how to manipulate memory data at the bit level which will allow us to make efficient use of storage space. Compound data types are introduced, such as **Strings** and **Arrays** (single and multi-dimensional). The chapter concludes with a discussion of custom type definitions such as **structs** and **unions**.

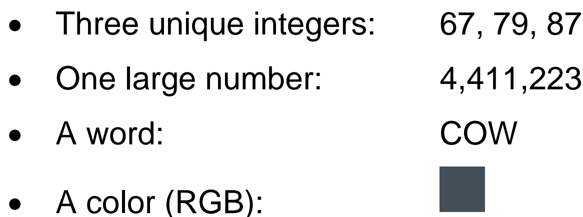


2.1 Number Representation and Bit Models

All data stored in a computer must somehow be represented numerically in some way whether it is numerical to begin with, a series of characters or an image. Ultimately, everything digitally breaks down to ones and zeros. We need to understand how data is stored (or represented) in the computer so that we interpret the 1's and 0's correctly. For example, consider this sequence of ones and zeros:

01000011 01001111 01010111

The data can be interpreted in different ways:



- Three unique integers: 67, 79, 87
- One large number: 4,411,223
- A word: COW
- A color (RGB):

The "correct" way to interpret the data depends on the context in which it is used.

Numerical values that we use normally every day are in base 10 … the decimal system. In this system, a sequence of digits such as 62389 is easily understood to be "sixty-two thousand three hundred and eighty-nine". We know this because we perform the following operation in our head:

$$(9 * 1) + (8 * 10) + (3 * 100) + (2 * 1{,}000) + (6 * 10{,}000)$$

Which is the same as doing this:

$$(9 * 10^0) + (8 * 10^1) + (3 * 10^2) + (2 * 10^3) + (6 * 10^4)$$

When dealing with computers, we often use other number systems as well such as Hexadecimal, Octal and Binary. Here is a comparison of these number systems:

Number System	Base	Digits/Characters Used	Example
Binary	2	0, 1	1001011
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	113
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	75
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	4B

Given a number in any of the non-decimal system formats, we can determine the value in the same manner as with our decimal number system, using the base as the multiplication factor.

Consider how to compute the value of the number $1001101_?$ in the various bases:

Base	Calculation	Decimal Value
2	$(1*2^0) + (0*2^1) + (1*2^2) + (1*2^3) + (0*2^4) + (0*2^5) + (1*2^6)$	77
8	$(1*8^0) + (0*8^1) + (1*8^2) + (1*8^3) + (0*8^4) + (0*8^5) + (1*8^6)$	262,721
10	$(1*10^0) + (0*10^1) + (1*10^2) + (1*10^3) + (0*10^4) + (0*10^5) + (1*10^6)$	1,001,101
16	$(1*16^0) + (0*16^1) + (1*16^2) + (1*16^3) + (0*16^4) + (0*16^5) + (1*16^6)$	16,781,569

Hopefully you can remember how to convert from one base to another base. If not, here is how to convert from **Binary** to **Decimal**, **Octal** and **Hex**:

Binary to Decimal

128 64 32 16 8 4 2 1
0 0 0 0 0 1 1 1
4 + 2 + 1 = 7_{10}

128 64 32 16 8 4 2 1
0 0 0 1 0 0 0 1
16 + 1 = 17_{10}

128 64 32 16 8 4 2 1
1 0 1 0 1 1 0 0
128 + 32 + 8 + 4 = 172_{10}

Binary to Octal

2 1 4 2 1 4 2 1
0 0 0 0 0 1 1 1
0 | 0 | 4+2+1=7 = 7_8

2 1 4 2 1 4 2 1
0 0 0 1 0 0 0 1
0 | 2 | 1 = 21_8

2 1 4 2 1 4 2 1
1 0 1 0 1 1 0 0
2 | 4+1=5 | 4 = 254_8

Binary to Hexadecimal

8 4 2 1 8 4 2 1
0 0 0 0 0 1 1 1
0 | 4+2+1 = 7 = 07_{16}

8 4 2 1 8 4 2 1
0 0 0 1 0 0 0 1
1 | 1 = 11_{16}

8 4 2 1 8 4 2 1
1 0 1 0 1 1 0 0
8 + 2 =A | 8 + 4 = C = AC_{16}

Here is how to convert from **Octal** and **Hex** to **Decimal**:

Octal to Decimal

64 8 1
1 2 5
(1*64) + (2*8) + (5*1) = 85_{10}

64 8 1
3 7 2
(3*64) + (7*8) + (2*1) = 250_{10}

64 8 1
2 0 1
(2*64) + (0*8) + (1*1) = 129_{10}

Hexadecimal to Decimal

4096 256 16 1
0 2 A F
(0*4096)+(2*256) + (10*16) + (15*1) = 687_{10}

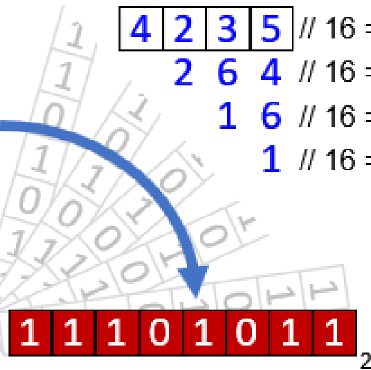
4096 256 16 1
1 7 3 C
(1*4096)+(7*256) + (3*16) + (12*1) = $5,948_{10}$

4096 256 16 1
2 B 0 5
(2*4096)+(11*256) + (0*16) + (5*1) = $11,013_{10}$

Here is how to convert from **Decimal** to **Binary** and **Hex**:

Decimal to Binary
(divide by 2 and get remainder)

235	// 2 = 117	remainder =	1
117	// 2 = 58	remainder =	1
58	// 2 = 29	remainder =	0
29	// 2 = 14	remainder =	1
14	// 2 = 7	remainder =	0
7	// 2 = 3	remainder =	1
3	// 2 = 1	remainder =	1
1	// 2 = 0	remainder =	1



11101011_2

Decimal to Hex
(divide by 16 and get remainder)

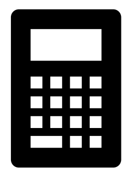
4235	// 16 = 264	rem = 11 =	B
264	// 16 = 16	rem = 8 =	8
16	// 16 = 1	rem = 0 =	0
1	// 16 = 0	rem = 1 =	1

$108B_{16}$

Here is a C example which shows how to specify literal values for decimal, octal, hex and binary numbers:

Code from **bases.c**

```
#include <stdio.h>

int main() {
  unsigned char dec  = 27;         // = (2*10) + 7 = 27
  unsigned char oct  = 027;        // = (2*8) + 7 = 23
  unsigned char hex  = 0xbf;       // = (11*16) + 15 = 191
  unsigned int  hex2 = 0xbad;      // = (11*256) + (10*16) + 13 = 2989
  unsigned char bin  = 0b00111100; // = (32 + 16 + 8 + 4) = 60

  printf("%d %d %d %d %d\n", dec, oct, hex, hex2, bin);

  return 0;
}
```



Bit Models

Consider now binary numbers. Each 1 or 0 is called a ***bit***. We can group bits together. A group of four consecutive bits is called a ***nibble*** and a group of eight consecutive bits is called a ***byte***. Two nibbles can be grouped to form a byte. We often break things down into nibbles when we do hexadecimal calculations since each hexadecimal digit is represented with a nibble. We can then group two, four or 8 bytes together for form a ***word***. We can thus have a ***16-bit word***, a ***32-bit word*** or a ***64-bit word***. When we group bits of 1's and 0's together, there are a variety of methods for interpreting them. Each method of interpreting the sequence of bits is called a ***bit model***. We will look at 6 models:

- Magnitude-only Bit Model
- Sign-Magnitude Bit Model
- Two's Compliment Bit Model
- Fixed-Point Bit Model
- Floating-Point Bit Model
- ASCII and Unicode Bit Model

In C, the model that is used will depend on the data type. So, **unsigned int**, **int**, **float** and **char** will all have their own bit model. (Note: From here-on in the notes, it is assumed that sequences of 1's and 0's are binary numbers, and so the numbers will not have a base 2 subscript indication.)

Magnitude-only Bit Model

This model is for **non-negative decimal** numbers (whole numbers). It is the simplest model since each bit represents an integer power of 2. With an 8-bit value, we can store a value in the range of **0** to **255**.

00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
…
11111101 = 253
11111110 = 254
11111111 = 255

In this representation, we consider the leftmost bit to be considered the ***most-significant bit*** and the rightmost bit as the ***least-significant bit***. (e.g., 10110101)

Interestingly, when adding numbers, we simply add the bits together starting from the least-significant bit to the most-significant bit:

Example of (10 + 7)	Example of (254 + 7)
00001010 + 00000111 -------- = 00010001 = **17**	11111110 + 00000111 -------- =**1**00000101 **OVERFLOW!** = **5**

Notice that there can be an overflow. Ultimately, there is a limit to the values that we can store with just one byte. Hence, we often use more than one byte to represent numbers in our software.

When dealing with combinations of bytes, we can have groups of 8 bits to store words to provide a larger range of values. For example, a sequence of 4 bytes can represent a much larger number:

10011001 00001110 01111001 00111001

= (10011001 * 2^{24}) + (00001110 * 2^{16}) + (01111001 * 2^{8}) + (00111001 * 2^{0})
= (153 * 16,777,216) + (14 * 65,536) + (121 * 256) + (57 * 1)
= <u>2,567,862,585</u>

Here is a table that shows the range of numbers that can be stored when using combinations of bytes together using the magnitude-only bit model:

Word Size	Bits Used	Bytes Used	Number Range
byte	8	1	0 to **255**
16-bit word	16	2	0 to **65,535**
32-bit word	32	4	0 to **4,294,967,295**
64-bit word	64	8	0 to **18,446,744,073,709,551,615**

In C, the magnitude-only bit model is used with unsigned types:

C – data type	Bits Used	Bytes Used	Number Range
`unsigned char`	8	1	**0** to **255**
`unsigned short int`	16	2	**0** to **65,535**
`unsigned int`	32	4	**0** to **4,294,967,295**
`unsigned long int`	64	8	0 to **18,446,744,073,709,551,615**

Sign-Magnitude Bit Model

This model allows **negative decimal** numbers (whole numbers). It is the simplest strategy for representing negative numbers. It has a smaller magnitude range though … allowing numbers only in the range of **-127** to **+127** for a single byte. The idea is to simply use the most-significant bit to be the ***sign bit*** (which is the reason for the reduction in magnitude using this bit model). By convention, a value of **0** in the sign bit indicates a positive value, while a **1** indicates a negative value.

00000000 = 0
00000001 = 1
00000010 = 2
…
01111110 = 126
01111111 = 127
10000000 = -0
10000001 = -1
10000010 = -2
…
11111110 = -126
11111111 = -127

This representation is not used often because of a couple of reasons. First, there are two values for zero … **+0** and **-0**. That is weird. Even more of a hassle is that the math does not work out evenly. If the signs of the two numbers are the same, we simply add the magnitudes as unsigned numbers and leave the sign bit intact.

However, we need to be careful about overflow:

Example of (10 + 7)	Example of (-10 + -7)	Example of (126 + 7)
00001010 + 00000111 -------- = 00010001 = **17**	10001010 + 10000111 -------- = 10010001 = **-17**	01111110 + 00000111 -------- = 00000101 **OVERFLOW!** = **5**

If the signs differ, we need to subtract the smaller magnitude from the larger magnitude, and keep the sign of the larger magnitude:

Example of (10 + -7)	Example of (-10 + 7)	Example of (126 + -7)
00001010 - 10000111 -------- = 00000011 = **3**	10001010 - 00000111 -------- = 10000011 = **-3**	01111110 - 10000111 -------- = 01110111 = **119**

But this is unpleasant and a bit ugly. Because of the two-zero problem and the need to subtract instead of add … the sign-magnitude bit model is not often used. It is not used in C.

Two's Complement Bit Model

This model allows both **positive and negative decimal** numbers (whole numbers).
Regarding positive and zero numbers … things are represented the same way as a magnitude-only bit model. It has a smaller magnitude range though … allowing numbers only in the range of **-128** to **+127**.
To represent negative numbers … we take the bits that represent the number as if it were positive, then invert (or flip) all of the bits and then add 1.

00010011 = 19
11101100 flipped bits
11101101 added one

11101101 = -19

As it turns out, if the most significant bit is **1**, then the number is negative, just like the sign-magnitude bit model. To determine the magnitude of a negative number, we perform the exact same steps:

11101101 = -19
00010010 flip bits
00010011 add one

00010011 = 19 magnitude

We add numbers in the same way:

Example of (10 + 7)	Example of (-10 + -7)

`  00001010`
`+ 00000111`
`  --------`
`= 00010001`
`= 17`
	`  11110110`
	`+ 11111001`
	`  --------`
	`=111101111` extra 1 carried out, no problem
	`= -17`

It is possible, however, that there can be an overflow … resulting in the answer being invalid. It is easy to tell if an overflow has occurred. There are two cases:

- If the sum of two positive numbers results in a negative result.
- If the sum of two negative numbers results in a positive result.

Example of (126 + 7)	Example of (-80 + -100)

`  01111110`
`+ 00000111`
`  --------`
`= 10000101` … sign bit changed!
`= -123` in two's-complement
	`  10110000`
	`+ 10011100`
	`  --------`
	`=101001100` overflow … sign bit changed!
	`= 76` in two's-complement

In C, the two's complement bit model is used with signed types:

C – data type	Bits Used	Bytes Used	Number Range
`char` `signed char`	8	1	**-128** to **+127**
`short int` `signed short int`	16	2	**-32,768** to **+32,767**
`int` `signed int`	32	4	**-2,147,483,648** to **+2,147,483,647**

`long`
`signed long`
`signed long int`
	64	8	**-9,223,372,036,854,775,808** to **+9,223,372,036,854,775,807**

Here is a program that stores some positive and negative values in *signed* and *unsigned* chars (i.e., bytes). Since the range of signed chars is smaller than unsigned, you will notice some interesting results for values above **127**. You will also notice some interesting results for values that go beyond the storage range of a **char** or **byte**. The compiler actually provides a warning, but allows the code to compile. Near the end of the code, some of these overflow values are stored in short data types … and you can see that there are no problems.

It is important to understand that things may not always "appear" as you want them to when it comes to mixing signed and unsigned values. Be careful when printing out such values … especially when debugging:

<table>
<tr><th>Code from twosCompliment.c</th><th>Output</th></tr>
<tr><td><pre>
#include <stdio.h>

int main() {
  char   sc;
  unsigned char uc;

  sc = uc = 0;   //00000000
  printf("0 signed: %d\n", sc);
  printf("0 unsigned: %d\n", uc);

  sc = uc = 1;   //00000001
  printf("1 signed: %d\n", sc);
  printf("1 unsigned: %d\n", uc);

  sc = uc = 7;   //00000111
  printf("7 signed: %d\n", sc);
  printf("7 unsigned: %d\n", uc);

  sc = uc = 126;   //01111110
  printf("126 signed: %d\n", sc);
  printf("126 unsigned: %d\n", uc);

  sc = uc = 127;   //01111111
  printf("127 signed: %d\n", sc);
  printf("127 unsigned: %d\n", uc);

  sc = uc = 128;   //10000000
  printf("128 signed: %d\n", sc);
  printf("128 unsigned: %d\n", uc);

  sc = uc = 255;   //11111111
  printf("255 signed: %d\n", sc);
  printf("255 unsigned: %d\n", uc);

  sc = uc = 256;   //100000000 *overflow
  printf("256 signed: %d\n", sc);
  printf("256 unsigned: %d\n", uc);

  sc = uc = 260;   //100000100 *overflow
  printf("260 signed: %d\n", sc);
  printf("260 unsigned: %d\n", uc);

  sc = uc = -0;   //100000000 *overflow
  printf("-0 signed: %d\n", sc);
  printf("-0 unsigned: %d\n", uc);

  sc = uc = -1;   //11111111
  printf("-1 signed: %d\n", sc);
  printf("-1 unsigned: %d\n", uc);

  sc = uc = -7;   //11111001
  printf("-7 signed: %d\n", sc);
  printf("-7 unsigned: %d\n", uc);

  sc = uc = -126; //10000010
  printf("-126 signed: %d\n", sc);
  printf("-126 unsigned: %d\n", uc);
</pre></td><td><pre>

0 signed: 0
0 unsigned: 0

1 signed: 1
1 unsigned: 1

7 signed: 7
7 unsigned: 7

126 signed: 126
126 unsigned: 126

127 signed: 127
127 unsigned: 127

128 signed: -128
128 unsigned: 128

255 signed: -1
255 unsigned: 255

256 signed: 0
256 unsigned: 0

260 signed: 4
260 unsigned: 4

-0 signed: 0
-0 unsigned: 0

-1 signed: -1
-1 unsigned: 255

-7 signed: -7
-7 unsigned: 249

-126 signed: -126
-126 unsigned: 130
</pre></td></tr>
</table>

Code	Output

`sc = uc = -127; //10000001`
`printf("-127 signed:   %d\n", sc);`
`printf("-127 unsigned: %d\n", uc);`
	`-127 signed:   -127`
	`-127 unsigned: 129`

`sc = uc = -128; //10000000`
`printf("-128 signed:   %d\n", sc);`
`printf("-128 unsigned: %d\n", uc);`
	`-128 signed:   -128`
	`-128 unsigned: 128`

`sc = uc = -255; //00000001   *weird eh?`
`printf("-255 signed:   %d\n", sc);`
`printf("-255 unsigned: %d\n", uc);`
	`-255 signed:   1`
	`-255 unsigned: 1`

`sc = uc = -256; //100000000  *overflow`
`printf("-256 signed:   %d\n", sc);`
`printf("-256 unsigned: %d\n", uc);`
	`-256 signed:   0`
	`-256 unsigned: 0`

`sc = uc = -260; //011111100  *overflow`
`printf("-260 signed:   %d\n", sc);`
`printf("-260 unsigned: %d\n", uc);`
	`-260 signed:   -4`
	`-260 unsigned: 252`

`short          ss;` `unsigned short us;`	

`ss = us = 255; //0000000011111111`
`printf("255 signed short:   %d\n", ss);`
`printf("255 unsigned short: %d\n", us);`
	`255 signed short:   255`
	`255 unsigned short: 255`

`ss = us = 256; //0000000100000000`
`printf("256 signed short:   %d\n", ss);`
`printf("256 unsigned short: %d\n", us);`
	`256 signed short:   256`
	`256 unsigned short: 256`

`ss = us = 260; //0000000100000100`
`printf("260 signed short:   %d\n", ss);`
`printf("260 unsigned short: %d\n", us);`
	`260 signed short:   260`
	`260 unsigned short: 260`

`ss = us = -255; //1111111100000001`
`printf("-255 signed short:   %d\n", ss);`
`printf("-255 unsigned short: %d\n", us);`
	`-255 signed short:   -255`
	`-255 unsigned short: 65281`

`ss = us = -256; //1111111100000000`
`printf("-256 signed short:   %d\n", ss);`
`printf("-256 unsigned short: %d\n", us);`
	`-256 signed short:   -256`
	`-256 unsigned short: 65280`

`ss = us = -260; //1111111011111100`
`printf("-260 signed short:   %d\n", ss);`
`printf("-260 unsigned short: %d\n", us);`
	`-260 signed short:   -260`
	`-260 unsigned short: 65276`

`return 0;` `}`	

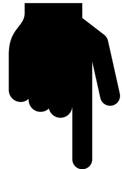
The three bit-models that we just looked at are used for representing whole numbers. But how do we represent ***real*** (i.e., non-whole) numbers ?

There are two main approaches:

1. **Fixed point**
   - Pros:
     - faster than floating-point arithmetic
       - Used in digital signal processing and game applications where performance is sometimes more important than precision
   - Cons:
     - loss of range for integer portion if more precise fraction is needed
     - works for fractional powers of 2 but not for powers of 10

2. **Floating point**
   - Pros:
     - much better precision and range
   - Cons:
     - slower than fixed-point arithmetic

Fixed-Point Bit Model



The fixed-point bit model is used to represent real numbers, such as floats and doubles. The key to the fixed-point bit model is based on the concept of a ***binary point*** … which is like the decimal point in a decimal system that separates the integer part from the fractional part. Consider the binary point number as follows:

```
11010.101
```

This can be calculated as follows:

$$(1*2^4) + (1*2^3) + (0*2^2) + (1*2^1) + (0*2^0) + (1*2^{-1}) + (0*2^{-2}) + (1*2^{-3})$$
$$= (1*16) + (1*8) + (0*4) + (1*2) + (0*1) + (1*1/2) + (0*1/4) + (1*1/8)$$
$$= 16 + 8 + 2 + 1/2 + 1/8$$
$$= \mathbf{26.625}$$

Interestingly, if we shift the binary point to the left, we end up with a number half the size:

```
1101.0101 = 13.3125
```

and if we shift the binary point to the right, we end up with a number twice the size:

```
110101.01 = 53.25
```

So … the position of the binary point is crucial in determining the result. In a fixed-point representation, therefore, we must always know how many bits are being used (a.k.a. the ***width***) … and where to position the binary point (a.k.a. the *binary point* ***position***).

Consider this number:

```
11010101
```

As we have seen it can represent various values, depending on where we place the binary point. We can use the notation **fixedPoint(w,b)** to represent a **w**-bit number with the binary point position set to have **b** digits to the right of the point:

Notation	Fixed-Point Binary	Real Number
fixedPoint(8,0)	11010101	213
fixedPoint(8,1)	1101010.1	105.5
fixedPoint(8,2)	110101.01	53.25
fixedPoint(8,3)	11010.101	26.625
fixedPoint(8,4)	1101.0101	13.3125
fixedPoint(8,5)	110.10101	6.65625
fixedPoint(8,6)	11.010101	3.328125
fixedPoint(8,7)	1.1010101	1.6640625
fixedPoint(8,8)	.11010101	0.83203125

Negative numbers are represented as either sign-magnitude or two's compliment. Assuming two's compliment, we represent the number -7.25 as follows:

000111.01 = **7.25** … positive number
111000.10 = flip bits …
111000.11 = add 1 to get negative number **-7.25**

Number addition is done the same way as two's compliment. We basically just "ignore" the decimal when adding:

Example of (2.5 + 1.75)	Example of (-2.5 + -7.25)
000010.10 + 000001.11 --------- = 000100.01 = **4.25**	111101.10 + 111000.11 --------- =1110110.01 = **-9.75**

Floating-Point Bit Model

With the floating-point representation, we can use the available bits in different ways. As a result, the total precision using 4 bytes is around 8 digits. That means, we can represent numbers like this: 0.123456, 123.456 or 123,456.0 … but we cannot represent numbers like this: 123,456.789012. With the floating-point model, the binary point position is not fixed … but may move around (i.e., float).

In the decimal number system, a number is written in scientific notation like this:

284_{10} = **2.84** x **10**^{2}

In general, a number is written in scientific notation as:

$\pm \mathbf{M} \times \mathbf{B}^{\mathbf{E}}$

where **M** = *mantissa*, **B** = *base* and **E** = *exponent*

In C, we have two types that represent real numbers:

C – data type	Bits Used	Bits used - Exponent	Bits used - Mantissa
`float`	32	8	23
`double`	64	11	52

There are many ways to represent a floating-point number. Here is one way to represent the number 284:

284 = 100011100_2 = 1.000111 x 2^8

1-bit sign	8-bit exponent	23-bit mantissa
0	00001000	100 0111 0000 0000 0000 0000

Since the leading digit in the mantissa is always 1 (for non-zero values), we can assume that this is implied in an improved representation as follows:

1-bit sign	8-bit exponent	23-bit mantissa
0	00001000	000 1110 0000 0000 0000 0000

In the IEEE 754 32-bit floating-point standard, we add a bias of **127** to the exponent as follows:

1-bit sign	8-bit biased exponent	23-bit mantissa
0	10000111	000 1110 0000 0000 0000 0000

There are some special cases:

Number	1-bit sign	8-bit biased exponent	23-bit mantissa
0		00000000	000 0000 0000 0000 0000 0000
∞	0	11111111	000 0000 0000 0000 0000 0000
-∞	1	11111111	000 0000 0000 0000 0000 0000
NaN		11111111	Non-zero

NaN is short for "Not a Number" and it is used to represent a number that does not exist.

When using ***double-precision*** numbers, the bias used should be **1023** instead of **127**. Otherwise things work the same way.

Adding floating point numbers is a little more work that non-floating-point numbers.

Consider adding the following two real numbers:

```
    345.7500
+ 6,819.1875
------------
  7,164.9375
```

How do we add these numbers using floating point notation?
First, get them into binary and then scientific notation:

```
0000101011001.1100 = 1.010110011100 x 2^8
1101010100011.0011 = 1.1010101000110011 x 2^12
```

And now into floating-point representation:

0	10000111	010 1100 1110 0000 0000 0000
0	10001011	101 0101 0001 1001 1000 0000

Now to add them, there are a few steps to take:

1. Get the two mantissas, putting the "`1.`" before both fractional parts:

   N1 = `1.0101100111`
   N2 = `1.1010101000110011`

2. Compare the exponents by subtracting the 2nd exponent from the 1st:

   135 – 139 = - 4

   Since this is a negative number, we shift the binary point of N1 right by 4 bits so that the numbers both have exponent 2^{12} now:

   N1 = `0.0001010110011100`
   N2 = `1.1010101000110011`

3. Add the mantissas:

```
  0.0001010110011100
+ 1.1010101000110011
--------------------
  1.1011111111001111
```

4. Normalize the mantissa if it has more than one digit to the left of the binary point

5. Round the result if need be (but should not need to if still fits in 23 bits)

6. Assemble the exponent and mantissa back into floating-point notation

0	10001011	101 1111 1110 0111 1000 0000

7. Verify if answer is correct:

   = $1.10111111110011110000000 \times 2^{12}$
   = 1101111111100.11110000000
   = 7164 + 0.9375
   = **7164.9375**

As you can see … it can be a little tricky! It is always important to verify that your calculations are correct.

ASCII and Unicode Bit Model

This model is for representing non-numerical values. It is a way of mapping characters to numbers. **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) and **Unicode** (not a real acronym but stands for a ***Uni****versal* **code** standard) are two ways of representing characters. The **ASCII** code standard was released in 1963, and was subsequently modified in 1967 and again in 1986. It is a subset of the **Unicode** standard which was released in 1991. **Unicode** incorporates characters from different languages.

Original ASCII code mapped non-accented English text characters and punctuations to numbers. It also mapped some control characters (e.g., NULL, whitespace, tabs, newline, separators) to numbers as well. Each character is contained in one byte. Here are the standard mappings from 0 to 255:

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	**space**	64	**@**	96	**`**
33	**!**	65	**A**	97	**a**
34	**"**	66	**B**	98	**b**
35	**#**	67	**C**	99	**c**
36	**$**	68	**D**	100	**d**
37	**%**	69	**E**	101	**e**
38	**&**	70	**F**	102	**f**
39	**'**	71	**G**	103	**g**
40	**(**	72	**H**	104	**h**
41	**)**	73	**I**	105	**i**
42	*****	74	**J**	106	**j**
43	**+**	75	**K**	107	**k**
44	**,**	76	**L**	108	**l**
45	**-**	77	**M**	109	**m**
46	**.**	78	**N**	110	**n**
47	**/**	79	**O**	111	**o**
48	**0**	80	**P**	112	**p**
49	**1**	81	**Q**	113	**q**
50	**2**	82	**R**	114	**r**
51	**3**	83	**S**	115	**s**
52	**4**	84	**T**	116	**t**
53	**5**	85	**U**	117	**u**
54	**6**	86	**V**	118	**v**
55	**7**	87	**W**	119	**w**
56	**8**	88	**X**	120	**x**
57	**9**	89	**Y**	121	**y**
58	**:**	90	**Z**	122	**z**
59	**;**	91	**[**	123	**{**
60	**<**	92	****	124	**|**
61	**=**	93	**]**	125	**}**
62	**>**	94	**^**	126	**~**
63	**?**	95	**_**		

Extended ASCII characters							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	**nbsp**

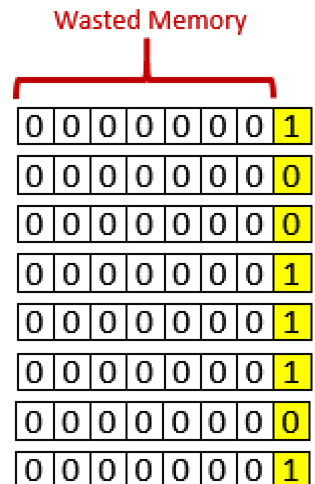
In C, there are 2 character-types that make use of ASCII:

C – data type	Bits Used	Bytes Used	Conversion
`unsigned char`	8	1	Decimal value converted to binary using **magnitude-only**
`char` `signed char`	8	1	Decimal value converted to binary using **two's compliment**

When dealing with **UNICODE** characters, there are various encoding schemes. **UTF-8** (i.e., **U**nicode **T**ransformation **F**ormat) is most commonly used to represent characters from other languages. Each character can take up to 4 bytes. But in C, we use a **short int** to hold a 2-byte Unicode value. There is a LOT to say about the Unicode text format, but we do not want to get into it too much in this course. We will focus mainly on ASCII code.

2.2 Bitwise Operations

Consider an internet company that has 1.2 million people using its services. Perhaps the company wants to keep track of some simple boolean values per customer … such as whether they are an adult, whether they are currently logged-on, or something as simple as whether their account is active. This can be done using an array:



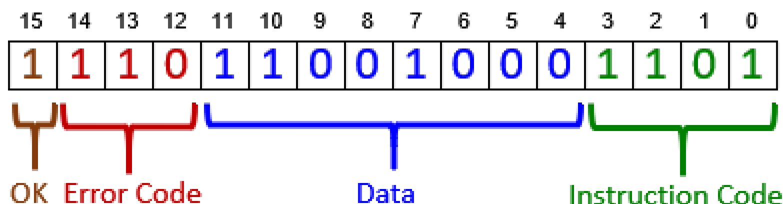

```
char        loggedOn[1200000];
```

The **char** data type is the smallest primitive type in C, which uses 1 byte … or 8 bits. We would set each value of the array to be either TRUE (i.e., 1) or FALSE (i.e., 0), depending on the log-on-status of that customer. The only values that will be stored in the array are **1** and **0**.

It is easy to see that this is a poor use of space. We would actually be making use of just **1** of the **8** bits in each byte. So we are wasting **87.5%** of the storage space required by the array!!!

A simple solution is to allow 8 people's log-on-status booleans to be grouped together and stored in a single byte.

In addition to this memory storage problem, sometimes we might need to access data coming in from (or out to) a hardware port in which a byte (or set of bytes) contains portions of bit sequences that have particular meaning, such as on/off flags, error codes, data, instruction code, etc..



At any given moment, we many need to read or set the bits that are relevant for what we are trying to do. It is for good reason then that we will need a way of manipulating bytes at the bit-level. That is what this section of the notes is all about … understanding how to manipulate the bits.

A ***bit operator*** *is an operator that takes one or two numbers and performs an operation on the bits of those numbers.*

They are symbols that are used to manipulate individual bits of whole number data types such as `signed char`, `unsigned char`, `signed int` and `unsigned int`. These operators can be used on literals or variables.

Here is a list of available bit-operators in C:

Operator	Action	Description
~	bitwise NOT	inverts every bit (works on one value)
&	bitwise AND	performs AND between every bit (requires two values)
|	bitwise OR	performs OR between every bit (requires two values)
^	bitwise XOR	performs Exclusive OR between every bit (requires two values)
>>	right- shift	moves bits into lower-order (less significant) bit positions
<<	left-shift	moves bits into higher-order (more significant) bit positions

Here is a program that shows how to use them:

Code from **bitOperators.c**

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

void printAsBinary(unsigned char);

int main() {
  unsigned char n1  = 157;    // 10011101
  unsigned char n2  = 198;    // 11000110

  unsigned char answer;

  answer = ~n1;
  printf("~157 = %d, ~10011101 = ", answer);printAsBinary(answer);
  answer = n1>>1;
  printf("\n157 >> 1 =  %d, 10011101 >> 1 = ", answer);printAsBinary(answer);
  answer = n1>>2;
  printf("157 >> 2 =  %d, 10011101 >> 2 = ", answer);printAsBinary(answer);
  answer = n1<<1;
  printf("157 << 1 =  %d, 10011101 << 1 = ", answer);printAsBinary(answer);
  answer = n1<<2;
  printf("157 << 2 = %d, 10011101 << 2 = ", answer);printAsBinary(answer);
  answer = n1&n2;
  printf("\n157 & 198 = %d, 10011101 & 11000110 = ", answer);printAsBinary(answer);
  answer = n1|n2;
  printf("157 | 198 = %d, 10011101 | 11000110 = ", answer);printAsBinary(answer);
  answer = n1^n2;
  printf("157 ^ 198 =  %d, 10011101 ^ 11000110 = ", answer);printAsBinary(answer);


  char  n3 = -100;    // 10011100
  char  answer2;

  answer2 = n3<<1;
  printf("\n-100 << 1 =  %d, 10011100 << 1 = ", answer2);printAsBinary(answer2);
  answer2 = n3<<4;
  printf("-100 << 4 = %d, 10011100 << 4 = ", answer2);printAsBinary(answer2);
  answer2 = n3>>1;
  printf("-100 >> 1 = %d, 10011100 >> 1 = ", answer2);printAsBinary(answer2);
  answer2 = n3>>4;
  printf("-100 >> 4 =  %d, 10011100 >> 4 = ", answer2);printAsBinary(answer2);
  return 0;
}
```

```
// Convert an unsigned char to an integer that looks like binary
void printAsBinary(unsigned char n) {
  for (int i=7; i>=0; i--) {
    if ((int)(n/pow(2,i)) > 0) {
      printf("1");
      n = n - pow(2,i);
    }
    else
      printf("0");
  }
  printf("\n");
}
```

The output of the program is as follows:

```
~157 = 98, ~10011101 = 01100010

157 >> 1 =  78, 10011101 >> 1 = 01001110
157 >> 2 =  39, 10011101 >> 2 = 00100111
157 << 1 =  58, 10011101 << 1 = 00111010
157 << 2 = 116, 10011101 << 2 = 01110100

157 & 198 = 132, 10011101 & 11000110 = 10000100
157 | 198 = 223, 10011101 | 11000110 = 11011111
157 ^ 198 =  91, 10011101 ^ 11000110 = 01011011

-100 << 1 =  56, 10011100 << 1 = 00111000
-100 << 4 = -64, 10011100 << 4 = 11000000
-100 >> 1 = -50, 10011100 >> 1 = 11001110
-100 >> 4 =  -7, 10011100 >> 4 = 11111001
```

Notice that during a right bit shift, zeros are added in on the left as the most significant bit. Similarly, when shifting left, zeros are added in on the right as the least significant bit. This is always the case when dealing with unsigned integers (i.e., magnitude only).

However, when right-shifting with negative numbers (i.e., two's complement), the bits coming in are 1's … the highest order sign bit.

Examining bitmasks

In order to extract portions of bits from numbers, we need to use …

> *A* ***bitmask*** *is a sequence of one or more bits that you apply to another binary number to read, set or clear the value of one or more bits.*

The bitmask number is used to indicate which bits are to be affected by an operation. The following table shows how to **read**, **set** and **clear** a particular bit (i.e., the **n**th bit) of a number:

Operation	Solution	in C code
Set n^th^ bit	x OR 2^n	`x = x | (1 << n);`
Clear n^th^ bit	x AND (NOT 2^n)	`x = x & (~(1 << n));`
Read n^th^ bit	(x AND 2^n) / 2^n	`x = (x & (1 << n)) >> n;`

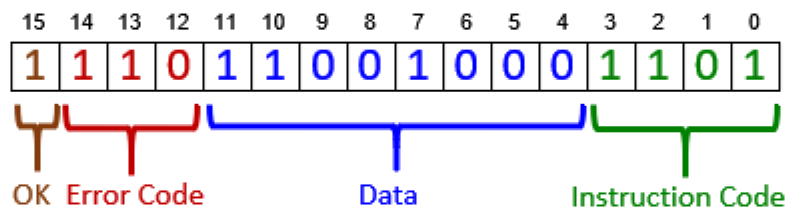
Here is an example of how we apply mask **m** to integer **x**:

Set bit 2 of x		Clear bit 4 of x		Read bit 4 of x	
`x = 19`	00010011	`x = 19`	00010011	`x = 19`	00010011
`m = 1<<2`	00000100	`m = 1<<4`	00010000	`m = 1<<4`	00010000
`x	m`	00010111	`~m`	11101111	`x & m`	00010000
		`x & ~m`	00000011	`(x & m)>>4`	00000001

Let's look at a more interesting example. Consider our earlier example of a two-byte bit-sequence that can be used to send commands to a device and to read data from that device. The device allows up to 16 instructions, some of which require 8-bits of data.

After the device performs the user's instruction, it will make use of the most significant bit (i.e., the **OK** bit) to indicate an error status. If all went well, this bit will be set to 0, otherwise it will be set to 1.

The particular kind of error will be indicated by the device which will set the **Error Code** bits to indicate one of 8 possible errors that may have occurred. Here is how the bits are organized:



We can store this command in an **unsigned short** as follows:

```
unsigned short  command;
```

Now, assume that we want to send instruction **0110** to the device with its corresponding data **10011101**. How can we ***set*** the instruction and data bits accordingly? Since the OK and Error Code bits are to be set by the device, we don't worry about what value they have when sending our instructions. So, it is easy to set the command.

We just need to shift the data left by 4 bits and add the instruction:

```
command = instruction + (data << 4);
send(command);
```

Assuming that the command is then modified by the device to contain an answer in the data bits. How can we extract that data ? We would need to provide a bitmask indicating which bits that we want to read as follows:

```
unsigned short result = receive();
printf("The result from the device is: %d",
                                    (result & (255<<4)) >> 4);
```

But we would have to first check the ok bit to make sure that the data is valid. If not, we should check the error code bits and inform the user:

```
if ((result & 0b1000000000000000) > 0)
    printf("An error has occurred with code: %d\n",
                                         (result & (0b111 << 12)) >> 12);
```

Here is a completed program that simulates the sending of 5 instructions to a device and then checks the 5 received results for errors and prints out an appropriate message:

Code from **bitmask.c**

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

void printAsBinary(unsigned short);
void send(int);
void receive(int);

unsigned char   simulatedInstructions[] = {0b0110,  // 6
                                           0b0000,  // 0
                                           0b0011,  // 3
                                           0b1111,  // 15
                                           0b0001}; // 1

unsigned char   simulatedData[] =  {0b00110011,  // 51
                                    0b11100000,  // 228
                                    0b10101010,  // 170
                                    0b00000000,  // 0
                                    0b11111111}; // 255

unsigned short  simulatedResults[] =  {0b0000101000010110,  // 161
                                       0b1010000000000000,  // error code 2
                                       0b1101001110010011,  // error code 5
                                       0b0000111111101111,  // 254
                                       0b0000000111110001}; // 31

int main() {
  unsigned short  result;

  // Send 5 commands and get 5 results
  for (int i=0; i<5; i++) {
    send(i);
    receive(i) ;
  }

  return 0;
}


// Simulate the sending of an instruction with data to the device
void send(int i) {
  unsigned short command = simulatedInstructions[i] + (simulatedData[i] << 4);
  printf("Sending Command: ");
  printAsBinary(command);
}
```

```
// Simulate the receiving of a one byte data reply from the device.  An
// error may have occurred, so this is checked for before the data is read.
void receive(int i) {
  unsigned short  result = simulatedResults[i];
  if ((result & 0b1000000000000000) > 0)
    printf("An error has occurred with code: %d\n", (result & (0b111<<12)) >> 12);
  else
    printf("The result data from the device is: %d\n", (result & (255<<4)) >> 4);
}


// Convert an unsigned short to an integer that looks like binary
void printAsBinary(unsigned short n) {
  for (int i=15; i>=0; i--) {
    if ((int)(n/pow(2,i)) > 0) {
      printf("1");
      n = n - pow(2,i);
    }
    else
      printf("0");
  }
  printf("\n");
}
```

Here is the program output:

```
Sending Command: 0000001100110110
The result data from the device is: 161
Sending Command: 0000111000000000
An error has occurred with code: 2
Sending Command: 0000101010100011
An error has occurred with code: 5
Sending Command: 0000000000001111
The result data from the device is: 254
Sending Command: 0000111111110001
The result data from the device is: 31
```

2.3 Compound Data Types

Up until now, we have examined only primitive data types such as **char**, **long int**, **short int**, **int**, **float** and **double**. We have also looked at how these are broken down into bits to be stored in the computer's memory. We will now look at working with compound data types:

> *A* ***compound data type*** *is a data type that stores multiple values under a single variable name.*

Recall that in JAVA, we have *primitive types* and we were able to combine multiple primitive types into one *compound type*. For example … in JAVA …

- **String** objects consist of an ordered-sequence of **char** primitives:
  - e.g.,

```
String s = new String("Hello ");
s = s + 'T' + 'h' + 'e' + 'r' + 'e';
```

This is JAVA code, not C code

- **Array** objects consist of an ordered-sequence of **primitives** or other *same-type* **objects**:
  - e.g.,

```
int[]  scores = {33, 56, 21, 75, 82, 44, 91};
Car[]  cars = new Car[50];
cars[0] = new Car("Red", "Porsche", 260);
cars[2] = new Car("Yellow", "Ferrari", 252);
```

This is JAVA code, not C code

- **Class** objects consist of a mixed set of attributes which may be **primitives** or **objects**:
  - e.g.,

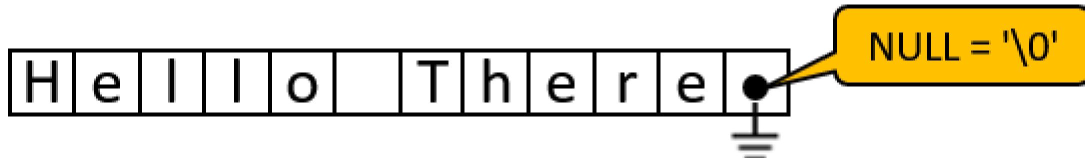
```
class Car {
     String   owner;
     int      accountNumber;
     float    balance;
}
```

This is JAVA code, not C code

In the C programming language, we can do similar things. We can create **Strings** and **Arrays** as well as **Structures** (which are similar to the notion of a collection of a class' attributes but without its methods). We will examine each of these now.

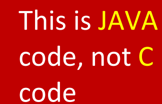
2.4 Strings and `char` Arrays

As in JAVA, strings in C are a sequence of characters. Unlike JAVA, which keeps track of a string's **length**, in C we **do not** keep track of the length. Instead, the end of a string is identified by a special *null-terminating character* … which is the **null** character **'\0'** … which has ACSII value **0**. It is a non-printable character … so it cannot be displayed on the screen:



Recall that in JAVA, we cannot modify a string once it has been created. So, for example, consider appending to a string as follows:

```
String s = new String("Hello ");
String c = s;
s = s + 'T' + 'h' + 'e' + 'r' + 'e';
System.out.println(s);  // prints "Hello There"
System.out.println(c);  // prints original string object "Hello"
```

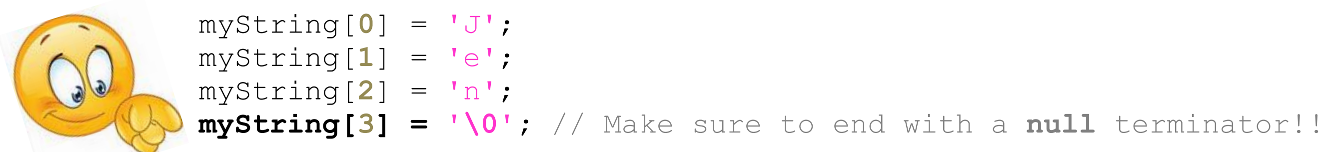


In JAVA, we get a **new string** each time that the **+** operator is used. The original string remains intact. We are actually unable to modify a created String in JAVA.

In C, however, since strings are just a sequence of **char** primitives in memory, we can modify the string at any time, given that we know the memory location of the start of the string. In fact, strings are defined with respect to a starting location in memory. In C, we declare a string type in one of two ways as follows:

```
char myString[4];      or      char *myString;
```

In the first option, we are declaring an *array of chars*. We can set the characters individually and modify the array at any time, as with any array:



```
myString[0] = 'J';
myString[1] = 'e';
myString[2] = 'n';
myString[3] = '\0'; // Make sure to end with a null terminator!!
```

We can also access the characters as a normal array, but be careful … we are allowed to access beyond the boundary:

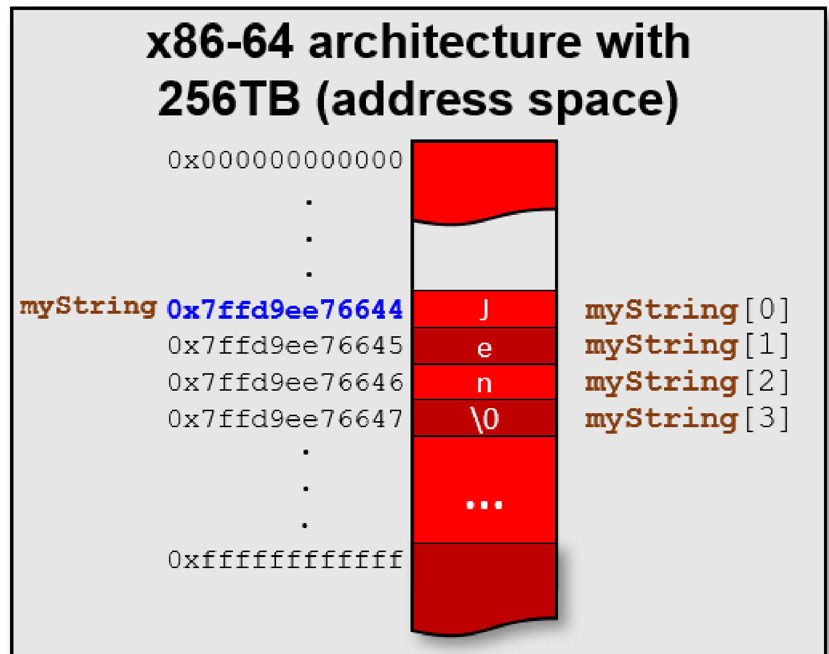
```
printf("myString    = %s\n", myString);      // Jen
printf("myString[0] = %c\n", myString[0]);   // J
printf("myString[1] = %c\n", myString[1]);   // e
printf("myString[2] = %c\n", myString[2]);   // n
printf("myString[3] = %c\n", myString[3]);   // null char
printf("myString[4] = %c\n", myString[4]);   // garbage!
printf("myString[5] = %c\n", myString[5]);   // garbage!
printf("myString[6] = %c\n", myString[6]);   // garbage!
```

In addition to setting the characters of the array one at a time, we can also hard-code the values with a constant (i.e., literal) string as follows:

```
char myString[] = "Jen";
```

Note that we do not need to specify the null-terminating character when creating a string in this manner.

Here is what it would look like as **myString** is stored in memory →



Note that it is the same as a char array in JAVA. The only difference is the use of the null-terminating character.

We can even reserve "extra" space for future characters by putting a size into the array:

```
char myString[50];
```

But in this case, the string is uninitialized … and hence filled with garbage characters. It would not display properly.

```
printf("myString = %s\n", myString);   // myString = `�XH|�
```

Of course, we could always set it to be purposely blank (and hence display nothing) like this:

```
char myString[50] = "";
```

Now, as for the 2nd way of creating a string (i.e., `char *myString`), take note of the * character. This indicates that we want the variable to ***refer to*** the location (i.e., memory address) of the first character of the string.

This ***reference*** is also known as …

*A **pointer** is a reference to a memory location.*

We could assign a value to the string as before:

```
char *myString = "Jen";
```

Again, the individual characters of the string can be accessed as if it were an array:

```
printf("myString    = %s\n", myString);      // Jen
printf("myString[0] = %c\n", myString[0]);   // J
printf("myString[1] = %c\n", myString[1]);   // e
printf("myString[2] = %c\n", myString[2]);   // n
printf("myString[3] = %c\n", myString[3]);   // null char
printf("myString[4] = %c\n", myString[4]);   // garbage!
printf("myString[5] = %c\n", myString[5]);   // garbage!
printf("myString[6] = %c\n", myString[6]);   // garbage!
```

Notice that the code is identical as when the string was declared as a `char[]`. At any time, we can determine the string's location in memory by using the **&** character (which means ***address of***) as follows:

```
printf("address of myString = %p\n", (void *) &myString);
```

This will return the memory location of the start of the string which may look something like this: **0x7ffd9ee76644** ... which will of course depend on the amount of memory installed on your computer and how much is allocated to your program etc.. We will talk much more about pointers and addresses later on.

For now, here is a test program to summarize everything. Feel free to play around with the code and experiment:

<table>
<tr><th>Code from strings.c</th><th>Output</th></tr>
<tr><td>

```
#include <stdio.h>

int main() {
  char      a = 'P';
  char    b[] = "Jen";
  char c[50]  = "";
  char     *d = "Max";

  printf("a = %c\n\n",     a);


  printf("b    = %s\n",    b);
  printf("b[0] = %c\n",    b[0]);
  printf("b[1] = %c\n",    b[1]);
  printf("b[2] = %c\n",    b[2]);
  printf("b[3] = %c\n",    b[3]);
  printf("b[4] = %c\n",    b[4]);
  printf("b[5] = %c\n\n", b[5]);

  printf("c    = %s\n\n", c);

  printf("d    = %s\n",    d);
  printf("d[0] = %c\n",    d[0]);
  printf("d[1] = %c\n",    d[1]);
  printf("d[2] = %c\n",    d[2]);
  printf("d[3] = %c\n",    d[3]);
  printf("d[4] = %c\n",    d[4]);
  printf("d[5] = %c\n\n", d[5]);

  printf("address of a = %p\n",(void *)  &a);
  printf("address of b = %p\n",(void *)  &b);
  printf("address of c = %p\n",(void *)  &c);
  printf("address of d = %p\n",(void *)  &d);

  return(0);
}
```

</td><td>

```
a = P


b    = Jen
b[0] = J
b[1] = e
b[2] = n
b[3] =
b[4] =
b[5] =

c    =

d    = Max
d[0] = M
d[1] = a
d[2] = x
d[3] =
d[4] = a
d[5] =

address of a = 0x7ffc5b2b871f
address of b = 0x7ffc5b2b872c
address of c = 0x7ffc5b2b8730
address of d = 0x7ffc5b2b8720
```

</td></tr>
</table>

We may also declare a **char** or **string** as ***constant*** … or non-modifiable. We do this by using the **const** keyword:

```
const char   d = 'G';
const char  *e = "Wow";
const char f[] = "Amazing";
```

If we try to modify a variable that has been declared as constant within our code, e.g., like this:

```
d = 'H';
```

then the compiler will stop us:

```
strings.c:40:5: error: assignment of read-only variable 'd'
   d = 'H';
```

For strings, the use of **const** is like a safety measure so that we don't modify the string accidentally. It also makes it clear in your code (when someone else is reading it) that you won't be modifying this later in your program. You should use **const** whenever you have fixed Strings (e.g., error messages, text used for **printf()** statements, or labels on a GUI, etc…).

There are some useful functions that we can use on strings. They are located in the **string.h** library. To use them, we must therefore put this header file at the top of our code:

```
#include <string.h>
```

strlen()

One of the most useful functions is the **strlen()** function which returns the number of characters in a string. It actually returns the number of characters up until it finds the null-terminating character.

```
char *t1 = "Mark";
printf("strlen(t1) = %d\n", strlen(t1));    // 4
```

We have to be careful to make sure that at least one character is available in the string. For example, notice the varying results when we call the **strlen()** function on *somewhat empty* strings as shown below:

```
char *t1 = "Please enter your two names: ";
char *t2 = "";
char *t3;
char  t4[];         // won't compile, since no array size specified
char  t5[] = "";
char  t6[25];
char  t7[25] = "";
printf("strlen(t1) = %d\n", strlen(t1));    // 29
printf("strlen(t2) = %d\n", strlen(t2));    // 0
printf("strlen(t3) = %d\n", strlen(t3));    // segmentation fault!!
printf("strlen(t5) = %d\n", strlen(t5));    // 0
printf("strlen(t6) = %d\n", strlen(t6));    // 0
printf("strlen(t7) = %d\n", strlen(t7));    // 0
```

Note that **t3** does not reserve any space for characters … so it is like an *undefined* string. Finding the length of it can cause strange results as well as a segmentation fault. Also, **t4[]** will not compile because we must specify a size for all arrays. Both **t6** and **t7** reserve enough space to hold plenty of characters but there are no characters in there upon initialization … so the content of the array is unpredictable. We typically use **strlen()** in our loops. For example, if we want to count the spaces in a string we can do this:

```
char *s = "Please enter your two names: ";

int count = 0;
for (int i=0; i<strlen(s); i++) {
  if (s[i] == ' ')
    count++;
}
printf("Number of spaces = %d\n", count);
```

strcpy(dest, src) & strncpy(dest, src, n)

The **strcpy()** function copies the string pointed to by **src** (including the terminating **null** byte) to the buffer pointed to by **dest**. It is important to ensure that the destination string is large enough to receive the copy otherwise we might overwrite memory locations and crash our program. Consider this code:

```
char *s1 = "This is the original";
char *s2;
strcpy(s2, s1);
```

It generates a **segmentation fault** (i.e., program crash)! Why ? Examine the second string **s2**. It is declared as a pointer to a string, but no space has been allocated for the characters. Therefore, when we try to copy over the string, it starts writing over memory locations that may contain our program or other variables. To fix this, we need to reserve space, perhaps using a character array as follows:

```
char *s1 = "This is the original";
char s2[50];
strcpy(s2, s1);
s2[8] = ' ';
s2[9] = 'u';
s2[10] = 'n';
s2[11] = '-';
printf("s1 = %s\n", s1);   // displays "This is the original"
printf("s2 = %s\n", s2);   // displays "This is  un-original"
```

A similar function is **strncpy()** which allows us to specify that we just want to copy a few characters, not all of them. Again, we need to be careful to reserve space for at least **n+1** characters (an extra one to hold the **null** terminator).

Be aware though, if there is no **null** byte among the first **n** bytes of the **src**, then the string placed in **dest** will not be null-terminated. Also, if the length of **src** is less than **n**, then **strncpy()** will write additional **null** bytes to **dest** to ensure that a total of **n** bytes are written.

```
char *s1 = "This is the original";
char s3[20];
strncpy(s3, s1, 11);
printf("s3 = %s\n", s3);  // displays "This is the���/"
s3[11] = '\0';
printf("s3 = %s\n", s3);  // displays "This is the"
```

strcmp(s1, s2) & strncmp(s1, s2, n)

The **strcmp(s1, s2)** function compares two strings, **s1** and **s2**, for equality. If **s1** < **s2** (alphabetically), then a negative value is returned. If **s1** > **s2** (alphabetically), then a positive value is returned. If **s1** == **s2** (alphabetically), then **0** is returned.

```
char *s4 = "Chris";
char *s5 = "Jen";
char *s6 = "Will";
printf("strcmp(Chris, Jen) = %d\n",  strcmp(s4, s5));  // -1
printf("strcmp(Jen, Chris) = %d\n",  strcmp(s5, s4));  // 1
printf("strcmp(Jen, Will) = %d\n",   strcmp(s5, s6));  // -1
printf("strcmp(Will, Chris) = %d\n", strcmp(s6, s4));  // 1
printf("strcmp(Jen, Jen) = %d\n",    strcmp(s5, s5));  // 0
```

Note that when the strings are not equal, the above code returned **-1** and **+1**. However, these values are not guaranteed … only the sign is guaranteed.

The **strncmp()** function works like **strcmp()**, however it only compares the first (at most) **n** characters of the strings:

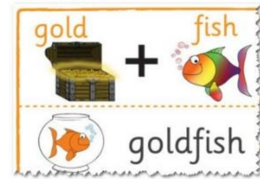
```
char *s7 = "Max";
char *s8 = "Mark";
char *s9 = "Melvin";
printf("strncmp(Max, Mark, 2) = %d\n",    strncmp(s7, s8, 2));  // 0
printf("strncmp(Max, Mark, 3) = %d\n",    strncmp(s7, s8, 3));  // 1
printf("strncmp(Melvin, Mark, 1) = %d\n", strncmp(s9, s7, 1));  // 0
printf("strncmp(Max, Melvin, 5) = %d\n",  strncmp(s7, s9, 5));  // -1
printf("strncmp(Max, Max, 8) = %d\n",     strncmp(s7, s7, 8));  // 0
```

strcat(dest, src) & strncat(dest, src, n)

The **strcat()** function appends the **src** string to the **dest** string, over-writing the terminating **null** byte at the end of **dest**, and then adds a terminating **null** byte. The **dest** string must have enough space for the result to be stored otherwise you again may overwrite valuable memory.

```
char *n1 = "Steve";
char *n2 = "Martha";
char *n3 = "Jacob";
char result[50] = "";  // 50 is big enough for this example
strcat(result, n1);
strcat(result, ", ");
strcat(result, n2);
strcat(result, ", ");
strcat(result, n3);
printf("result = %s\n", result);   // displays "Steve, Martha, Jacob"
```

Similarly, the **strncat()** appends at most **n** bytes from **src**. Therefore, the space that **dest** is stored in must be at least **strlen(dest) + n + 1** bytes.




```
char *n4 = "Steve";
char *n5 = "ariel";
char *n6 = "tup";
char result2[50] = "";
strncat(result2, n4, 2);
strncat(result2, n5, 2);
strncat(result2, n6, 3);
strncat(result2, n4, 999); // only appends 5 chars since null is there
printf("result2 = %s\n", result2);   // displays "StartupSteve"
```

sprintf()

The **sprintf()** function works like **printf()**, except that instead of printing to the console, it prints the information into a specified string. This function will return the number of bytes in the resulting string, or a negative value if an error occurs.

It is most useful for converting numbers to strings and for combining multiple values into a string.

```
char   answer[50];
char    *name = "Bob";
int       num = 1026784;
float balance = 67.32;
int     chars = sprintf(answer, "%s's account %d with $%0.2f",
                        name, num, balance);
printf("The string is \"%s\" with %d characters\n", answer, chars);
```

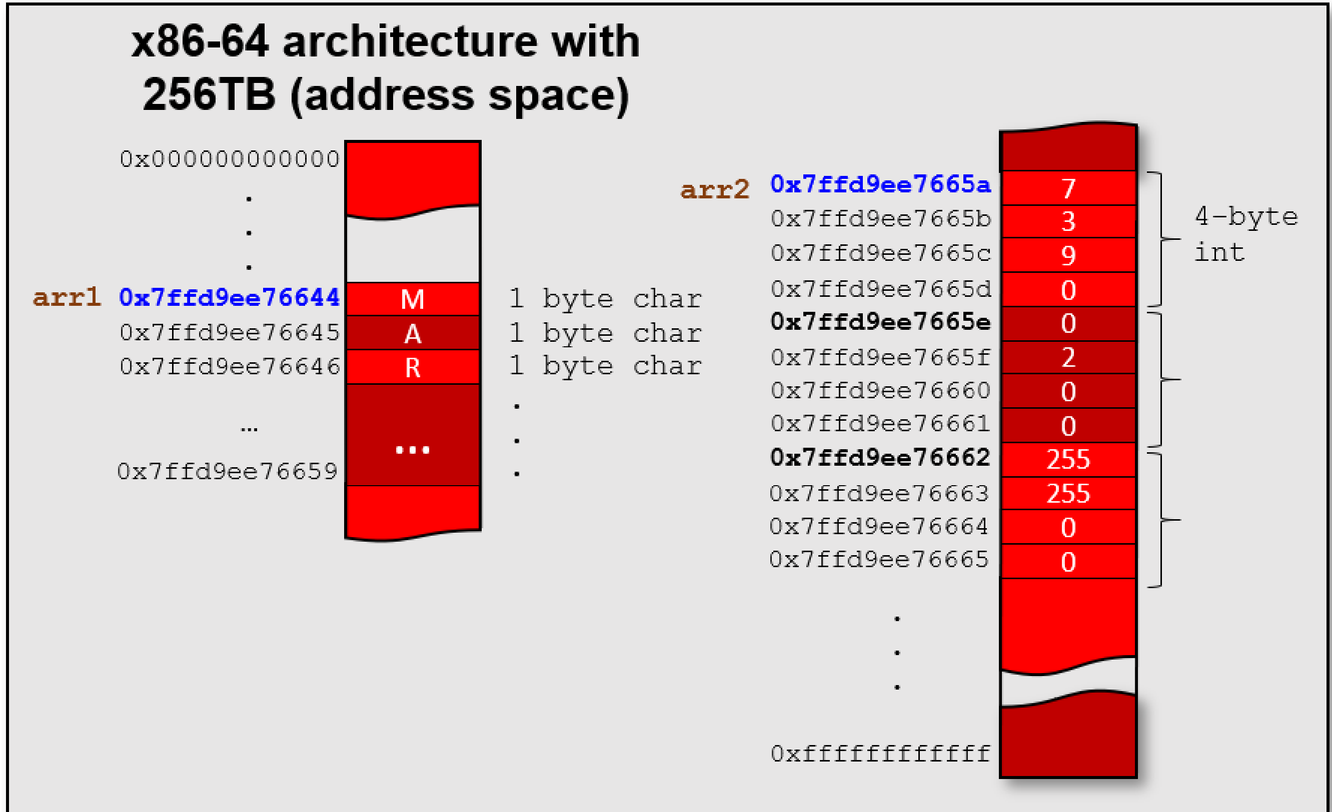
The answer is:

```
The string is "Bob's account 1026784 with $67.32" with 33 characters
```

2.5 Arrays

Arrays store multiple values of the same type. Every element takes up the same amount of memory. The size of the array in memory will therefore depend on the type of data stored. So, for example, the following two arrays would look different in memory:

```
char    arr1[20] = "MARK is having fun!";
int     arr2[20] = {590599, 512, 65535};
```



As we have seen, accessing the arrays is done the same way as in JAVA … by indices … which start at **0**. However, unlike JAVA, C does NOT do any **Array Out of Bounds** checking! Therefore, you need to be much more careful when using them.

We can use the **sizeof()** function in C to determine how many bytes are required for a particular type or how many bytes a variable is using. This will help us determine how much storage space is required for our arrays. So, for example, we can determine the size of types like this:

Use **%zu** when using **sizeof()**.

```
printf("The size of int is %zu\n",    sizeof(int));    //  displays 4
printf("The size of float is %zu\n",  sizeof(float));  //  displays 4
printf("The size of double is %zu\n", sizeof(double)); //  displays 8
printf("The size of char is %zu\n\n", sizeof(char));   //  displays 1
```

And if we were to create some arrays of these types, we could determine how many bytes the entire array will require when stored in memory as follows:

```
int    array1[5];
float  array2[3];
double array3[6];
char   array4[10];

printf("The size of array1 is %zu\n",   sizeof(array1)); //  5*4 = 20
printf("The size of array2 is %zu\n",   sizeof(array2)); //  3*4 = 12
printf("The size of array3 is %zu\n",   sizeof(array3)); //  6*8 = 48
printf("The size of array4 is %zu\n\n", sizeof(array4)); // 10*1 = 10
```

In C, arrays do not have an attribute nor a function that allows us to know how many items are in the array. We could keep track of this in a separate variable, as we did in JAVA. Then we can loop through the elements using a FOR loop. However, there is a way to determine the *capacity* of an array. We can use the **sizeof()** function again. Since **sizeof()** tells us how many *total* bytes are in the array, we just need to divide that number by the *size of the type* that the array is storing … as follows:

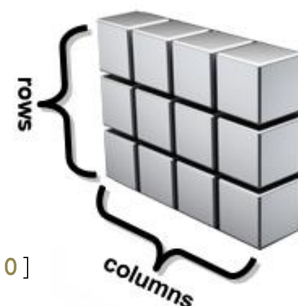
```
for (int i=0; i<sizeof(array1)/sizeof(int); i++)
  printf("%d, ", array1[i]);
printf("\n");

for (int i=0; i<sizeof(array2)/sizeof(float); i++)
  printf("%f, ", array2[i]);
printf("\n");

for (int i=0; i<sizeof(array3)/sizeof(double); i++)
  printf("%g, ", array3[i]);
printf("\n");

for (int i=0; i<sizeof(array4)/sizeof(char); i++)
  printf("%d, ", array4[i]);
printf("\n\n");
```

Here is the result … showing the "garbage" data that is in the uninitialized arrays:

```
1276585784, 32763, 0, 0, 1,
0.000000, 0.000000, 0.000000,
0, 8.40389e-315, 6.95236e-310, 6.953e-310, 6.953e-310, 6.95236e-310,
0, 0, 16, 44, 67, 78, -2, 127, 0, 0,
```

We may also create and use multi-dimensional arrays in C:

```
int array2D[3][4] = { {1,  2,  3,  4},
                      {5,  6,  7,  8},
                      {9, 10, 11, 12} };
```



Notice that we have two sets of indices. The first index indicates the *row* and the second one indicates the *column*. Therefore, `array2D[2][0]` indicates the 1st item in the 3rd row … which is **9** in the above example.

To understand fully how to access the sizes of the items, the rows and the columns, we must fully understand how to use **sizeof()**. Make sure that you understand the output from the following code:

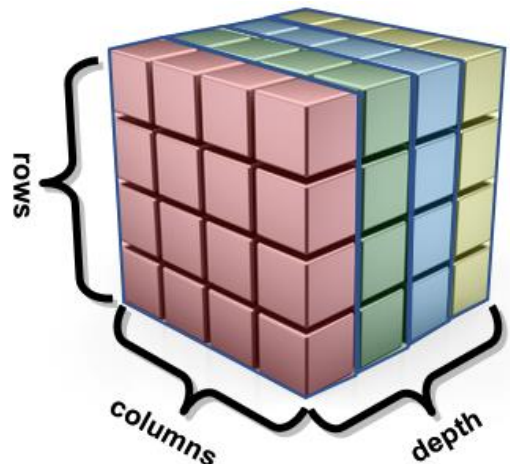
```
printf("The size of array2D is %zu \n",
                                 sizeof(array2D));
printf("The #elements in array2D is %zu \n",
                                 sizeof(array2D)/sizeof(int));
printf("The #elements in each row of array2D is %zu \n",
                                 sizeof(array2D[0])/sizeof(int));
printf("The #rows of array2D is %zu \n",
                                 sizeof(array2D)/sizeof(array2D[0]));
for (int r=0; r<3; r++) {
  for (int c=0; c<4; c++)
    printf("%02d ", array2D[r][c]);
  printf("\n");
}
printf("\n");
```

Here is the output … was it what you expected?

```
The size of array2D is 48
The #elements in array2D is 12
The #elements in each row of array2D is 4
The #rows of array2D is 3
01 02 03 04
05 06 07 08
09 10 11 12
```

Interestingly, we can create even higher dimensional arrays. For example, an array as follows may be used to represent a cube of colored blocks:

```
int cube[4][4][4] = {   {{1,  2,  3,  4},
                          {5,  6,  7,  8},
                          {9, 10, 11, 12},
                          {13, 14, 15, 16}},
                         {{17, 18, 19, 20},
                          {21, 22, 23, 24},
                          {25, 26, 27, 28},
                          {29, 30, 31, 32}},
                         {{33, 34, 35, 36},
                          {37, 38, 39, 40},
                          {41, 42, 43, 44},
                          {45, 46, 47, 48}},
                         {{49, 50, 51, 52},
                          {53, 54, 55, 56},
                          {57, 58, 59, 60},
                          {61, 62, 63, 64}}
                     };
```




```
int rows = sizeof(cube)/ sizeof(cube[0]);
int cols = sizeof(cube[0])/ sizeof(cube[0][0]);
int depth = sizeof(cube)/(rows*cols*sizeof(int));
```

```
printf("Cube has %d rows\n", rows);
printf("Cube has %d columns\n", cols);
printf("Cube has %d layers\n", depth);

for (int r=0; r<4; r++) {
  for (int c=0; c<4; c++) {
    for (int d=0; d<4; d++)
      printf("%02d ", cube[r][c][d]);
    printf("\n");
  }
  printf("\n");
}
printf("\n");
```

Here is the output:

```
Cube has 4 rows
Cube has 4 columns
Cube has 4 layers
01 02 03 04
05 06 07 08
09 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24
25 26 27 28
29 30 31 32

33 34 35 36
37 38 39 40
41 42 43 44
45 46 47 48

49 50 51 52
53 54 55 56
57 58 59 60
61 62 63 64
```

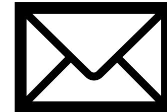
2.6 Custom Type Definitions: Structures and Unions

Recall that in JAVA we defined various classes to represent objects. These classes allowed us to group together a bunch of ***attributes*** (a.k.a. ***fields***) that are either primitive types or other objects. It keeps them all together in a bundle. You can think of a ***structure*** as a bunch of small pieces of information with an elastic around it:

*A **structure** represents multiple pieces of information that are grouped together.*

Consider a person's full address, which may be represented using multiple strings as follows:

```
char *name = "Patty O. Lantern";
char *streetNumber = "187B";
char *streetName = "Oak St.";
char *city = "Ottawa";
char *province = "ON";
char *postalCode = "K6S8P2";
```



We could display the variable values individually:

```
printf("%s\n", name);                              // Patty O. Lantern
printf("%s %s\n", streetNumber, streetName);       // 187B Oak St.
printf("%s, %s\n", city, province);                // Ottawa, ON
printf("%s\n", postalCode);                        // K6S8P2
```

Recall in JAVA that we set up a *class* to define an address. In C, we do something similar, except that instead of creating a class, we create a **struct**. There are two ways of defining the **struct** in C.

Consider the similarities and differences as shown here:

<table>
<tr><th>JAVA</th><th>C – just struct</th><th>C – using typedef</th></tr>
<tr><td><pre>class Address {
  String   name;
  String   streetNumber;
  String   streetName;
  String   city;
  String   province;
  String   postalCode;
}</pre></td><td><pre>struct AddressType {
  char   *name;
  char   *streetNumber;
  char   *streetName;
  char   *city;
  char   *province;
  char   *postalCode;
};</pre></td><td><pre>typedef struct {
  char   *name;
  char   *streetNumber;
  char   *streetName;
  char   *city;
  char   *province;
  char   *postalCode;
} AddressType;</pre></td></tr>
</table>

Note that in C, it is common to put the word "Type" at the end of the structure name. Hence, we used **AddressType** instead of simply **Address**.

Notice also, that there is a **;** character after the **struct**'s definition. That is because the **struct** keyword is just another C statement. So, we can define this **struct** anywhere in our code.

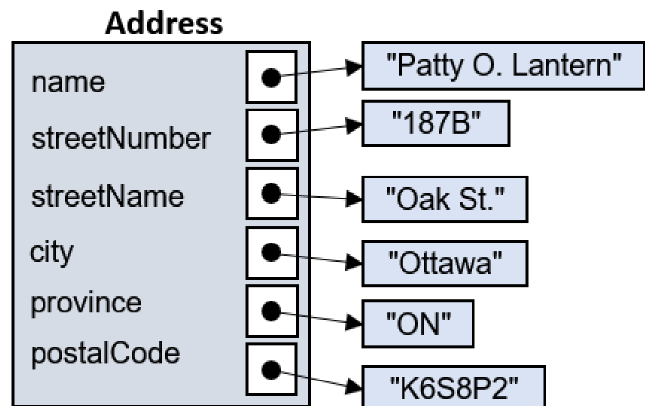
This is different than JAVA classes, which we usually define each in their own unique file. However, we usually define all of our **structs** outside of our functions … for example … at the top of our program before the main function … or in a header file (more on this later).

In the second C example, we are actually defining a new ***type***. This will make it easier when we declare variables and parameters. Notice the JAVA vs. C comparison when defining a variable called **addr**:

JAVA	C – just **struct**	C – using **typedef**
`Address addr;`	`struct AddressType addr;`	`AddressType addr;`

As you can see, unless we create the **typedef**, we would need to use the **struct** keyword when declaring the variable's type. Otherwise, the declaration is similar to that of JAVA.

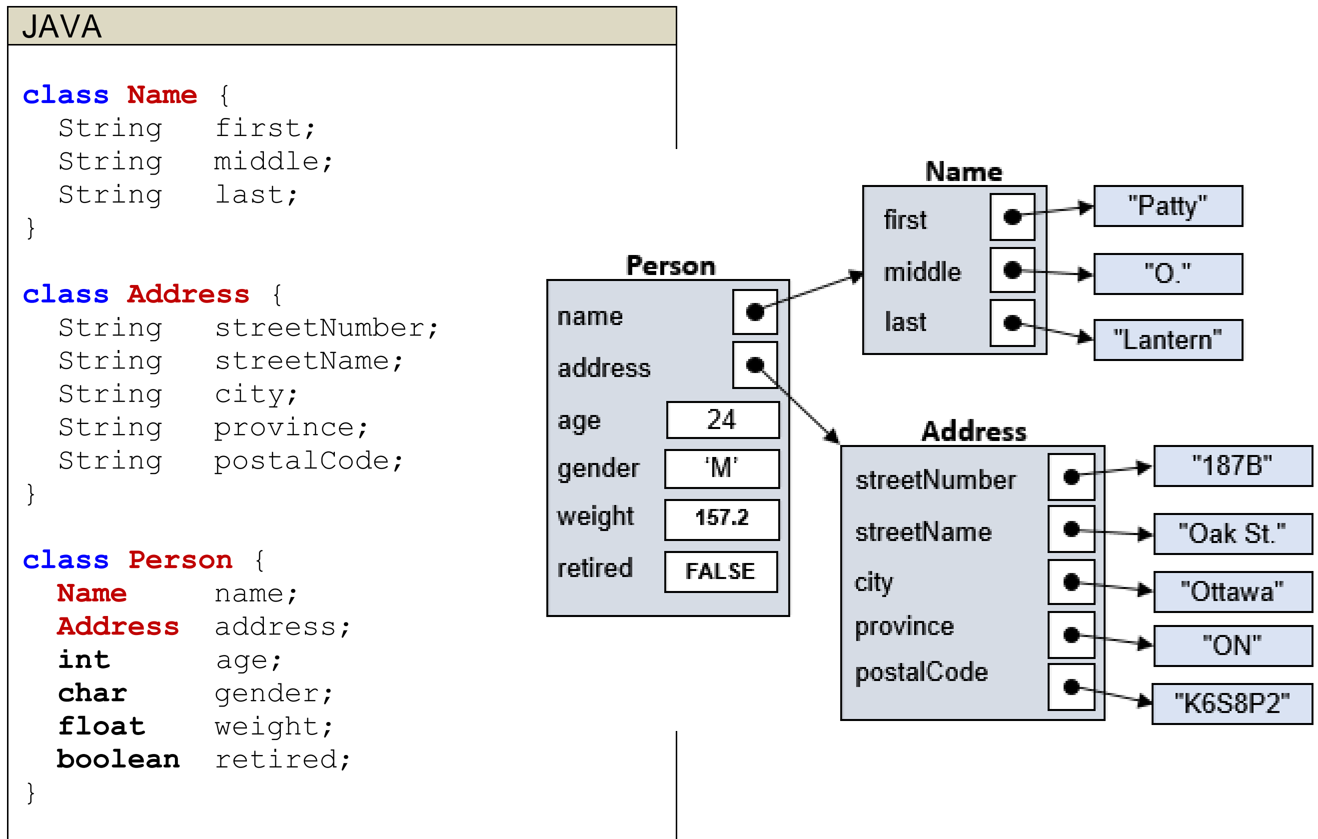
Assigning values to the structure's attributes is now easy, as it is done in a similar way to JAVA by using the dot operator:




```
addr.name = "Patty O. Lantern";
addr.streetNumber = "187B";
addr.streetName = "Oak St.";
addr.city = "Ottawa";
addr.province = "ON";
addr.postalCode = "K6S8P2";

printf("%s\n", addr.name);
printf("%s %s\n", addr.streetNumber, addr.streetName);
printf("%s, %s\n", addr.city, addr.province);
printf("%s\n", addr.postalCode);
```

We can define other **structs** as well and use them inside one another, just like using objects within objects in JAVA.

Consider this JAVA code:



JAVA

```java
class Name {
   String   first;
   String   middle;
   String   last;
}

class Address {
   String   streetNumber;
   String   streetName;
   String   city;
   String   province;
   String   postalCode;
}

class Person {
   Name     name;
   Address  address;
   int      age;
   char     gender;
   float    weight;
   boolean  retired;
}
```


Here is a C program that constructs and uses this set of objects as **structs** within **structs**:

Code from **structsInStructs.c**

```c
#include <stdio.h>

#define TRUE   1;
#define FALSE  0;


// Structure that represents a person's full name
typedef struct {
  char    *first;
  char    *middle;
  char    *last;
} NameType;


// Structure that represents a person's full address
typedef struct {
  char    *streetNumber;
  char    *streetName;
  char    *city;
  char    *province;
  char    *postalCode;
} AddressType;
```

```
// Structure that represents a person
typedef struct {
  NameType        name;
  AddressType     address;
  int             age;
  char            gender;
  float           weight;
  char            retired;
} PersonType;

int main() {
  PersonType   bob;

  bob.name.first = "Patty";
  bob.name.middle = "O.";
  bob.name.last = "Lantern";
  bob.address.streetNumber = "187B";
  bob.address.streetName = "Oak St.";
  bob.address.city = "Ottawa";
  bob.address.province = "ON";
  bob.address.postalCode = "K6S8P2";
  bob.age = 24;
  bob.gender = 'M';
  bob.weight = 157.2;
  bob.retired = FALSE;

  printf("%s %s %s\n", bob.name.first, bob.name.middle, bob.name.last);
  printf("%d year old %s %s weighing %f pounds\n",
           bob.age,
           bob.retired ?  "retired" : "non-retired",
           (bob.gender == 'M') ? "male": "female",
           bob.weight);


  printf("Living at: %s %s, ",
           bob.address.streetNumber,
           bob.address.streetName);

  printf("%s, %s  ", bob.address.city, bob.address.province);
  printf("%s\n", bob.address.postalCode);


  return(0);
}
```

Notice that structs are being used within other structs.

Attributes are accessed the same way as in JAVA by using the dot operator.

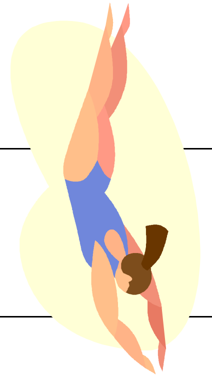
Here is the output:

```
Patty O. Lantern
24 year old non-retired male weighing 157.199997 pounds
Living at: 187B Oak St., Ottawa, ON  K6S8P2
```

Just a few points about **scope** … notice that the **structs** are all defined before the **main** function. This makes them **global**, which means that they can be accessed from anywhere. By contrast, if we define a **struct** within a function, then we may ONLY use that **struct** within that function. Typically, we want our **structs** to be used throughout our entire program, so we usually put their definitions before the **main** function.

As with JAVA objects, we may also make arrays of **structs** and we can use them as parameters to functions. Consider an example in JAVA where we have objects that represent a **Dive**, a **Performance** and an **Athlete** in a system where athletes perform 3 dives each and each performance is judged by exactly 8 judges who provide a score each time.

<table>
<tr><th>JAVA code</th><th>C code</th></tr>
<tr><td><pre>class Dive {
  String   name;
  int   difficulty;
}</pre></td><td><pre>typedef struct {
  char   *name;
  int   difficulty;
} DiveType;</pre></td></tr>
<tr><td><pre>class Performance {
  Dive   dive;
  float[]   scores;
}</pre></td><td><pre>typedef struct {
  DiveType dive;
  float   scores[8];
} PerformanceType;</pre></td></tr>
<tr><td><pre>class Athlete {
  String   name;
  String   country;
  Performance[]   performances;
}</pre></td><td><pre>typedef struct {
  char   *name;
  char   *country;
  PerformanceType   performances[3];
} AthleteType;</pre></td></tr>
</table>



Remember … when creating arrays in C, even in a **struct** definition, we need to supply the size of the array. This is different from JAVA attribute definitions, where we just ***defined*** the array and then created the array using a constructor at a later time.

We can create arrays to hold 5 dives and 3 athletes as follows:

```
DiveType     dives[5];
AthleteType  athletes[3];
```

Given that a *pointer* takes **8** bytes in a 64-bit system, do you understand why the following *should* produce the output shown?

```
printf("%zu\n", sizeof(DiveType));          //should logically print 12
printf("%zu\n", sizeof(PerformanceType));   //should logically print 44
printf("%zu\n", sizeof(AthleteType));       //should logically print 148
printf("%zu\n", sizeof(dives));             //should logically print 60
printf("%zu\n", sizeof(athletes));          //should logically print 444
```

Data Structure Byte Alignment:

Although the above size of **DiveType** is logical, it actually is **16**!! The compiler will attempt to "align" data values in similar-sized byte chunks. It sometimes will add more bytes within a **struct** (called ***padding***) to make it a multiple of 2, 4, or 8. Here, for example, is what you will get as an output:

```
printf("%zu\n", sizeof(DiveType));          // prints 16
printf("%zu\n", sizeof(PerformanceType));   // prints 48
printf("%zu\n", sizeof(AthleteType));       // prints 160
printf("%zu\n", sizeof(dives));             // prints 80
printf("%zu\n", sizeof(athletes));          // prints 480
```

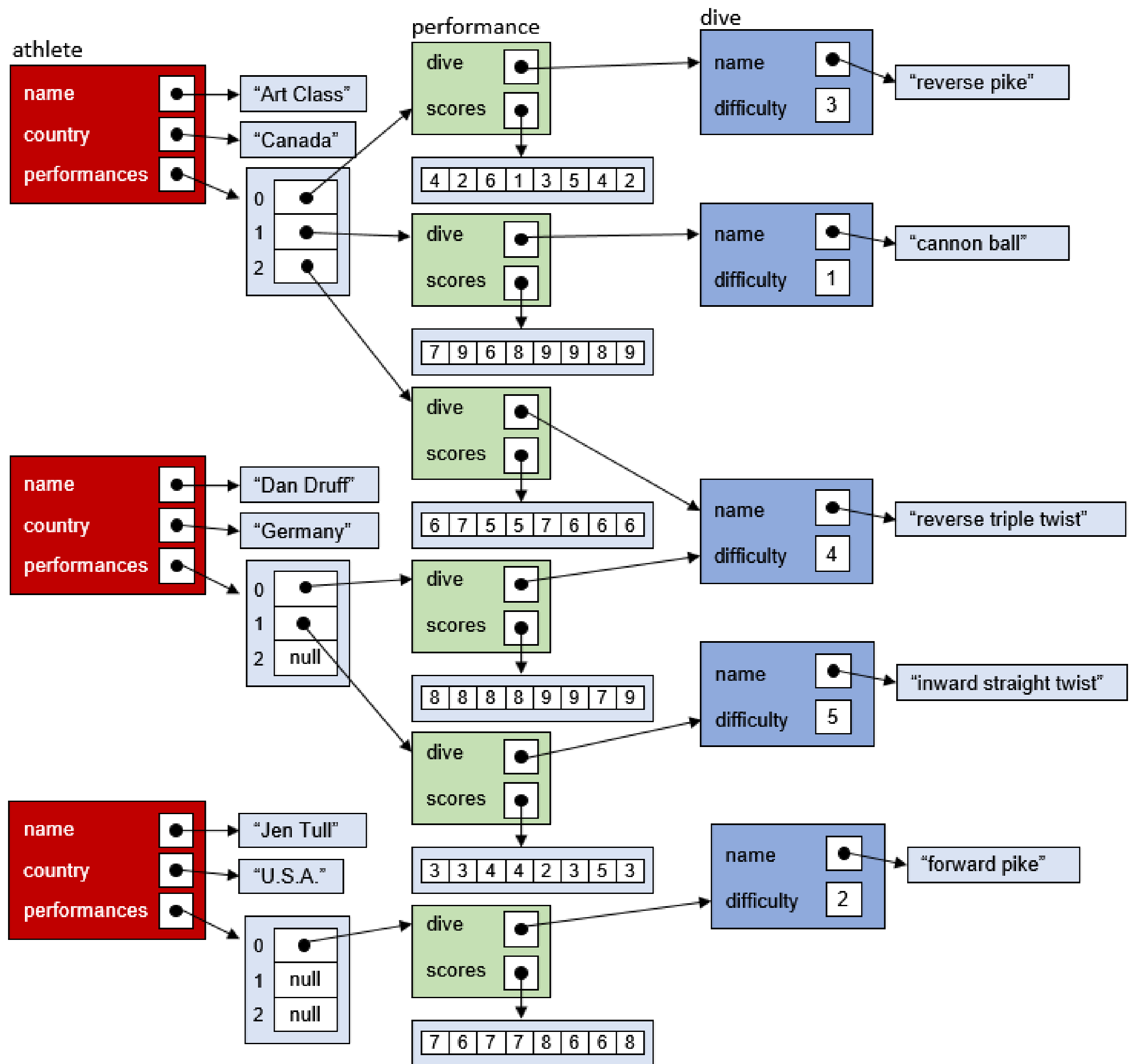
Although the manner in which the bytes are aligned depends on the compiler implementation, the system may tend to look at the largest of the types in the **struct** and decide to add extra padding to make it a multiple of that largest type. Some example are as follows:

Definition	`struct X {`
	`char a;`
	`}`
		`struct X {`
		`char a;`
		`char b;`
		`char c;`
		`}`
			`struct X {`
			`char a;`
			`short b;`
			`}`
				`struct X {`
				`char a;`
				`short b;`
				`char c;`
				`char d;`
				`}`
					`struct X {`
					`char a;`
					`int b;`
					`}`

Max type size	char = **1** byte	char = **1** byte	short = **2** bytes	short = **2** bytes	int = **4** bytes
Padding	none	none	1 extra byte	1 extra byte	3 extra bytes
sizeof(struct X)	1	3	4	6	8

The next page has a diagram showing how the **structs** are contained within one another. The example shows a snapshot in time after **Art** performed three dives, **Dan** performed two and **Jen** one.

Take note of the fact that the **Dives** are created just once, but then may be shared between performances. That is logical, since all athletes, in theory, could perform the exact same dive … but their performances will likely differ in terms of their individual scores. Hence, each athlete has unique performances.



Here is some code that creates an array of **5** fixed dives and an array of **3** athletes. It then generates three performances for each athlete, choosing a random dive each time. The performance is then judged via the production of 8 simulated scores. Finally, the athlete's data is displayed through use of a function that takes an **AthleteType** struct as a parameter.

Code from **structArrays.c**

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

#define NUMBER_OF_SCORES         8
#define NUMBER_OF_PERFORMANCES   3


typedef struct {      // Structure that represents Dive data
  char  *name;
  int    difficulty;
} DiveType;


typedef struct {      // Structure that represents Performance data
  DiveType   dive;
  float      scores[NUMBER_OF_SCORES];
} PerformanceType;


typedef struct {      // Structure that represents Athlete data
  char             *name;
  char             *country;
  PerformanceType  performances[NUMBER_OF_PERFORMANCES];
} AthleteType;


DiveType     dives[5];     // An array to hold the 5 types of dives
AthleteType  athletes[3];  // An array to hold the 3 athletes



// Procedure that displays an athlete, his/her performances as well as
// the 8 individual scores for each performance.
void displayAthlete(AthleteType  athlete) {
  printf("Athlete: %s from %s:\n", athlete.name, athlete.country);
  for (int p=0; p<NUMBER_OF_PERFORMANCES; p++) {
      printf("  Performed %s (diff: %d): [",
           athlete.performances[p].dive.name,
           athlete.performances[p].dive.difficulty);

      for (int s=0; s<NUMBER_OF_SCORES; s++) {
           printf("%0.2f",athlete.performances[p].scores[s]);
          if (s != NUMBER_OF_SCORES-1)
             printf(", ");
      }
      printf("]\n");
    }
  printf("\n");
}
```

```
int main() {
  // Fill in the athletes array with 3 athletes
  athletes[0].name = "Art Class";
  athletes[0].country = "Canada";
  athletes[1].name = "Dan Druff";
  athletes[1].country = "Germany";
  athletes[2].name = "Jen Tull";
  athletes[2].country = "U.S.A.";

  // Fill in the dives array with 5 dives
  dives[0].name = "reverse pike";
  dives[0].difficulty = 3;
  dives[1].name = "cannon ball";
  dives[1].difficulty = 1;
  dives[2].name = "reverse triple twist";
  dives[2].difficulty = 4;
  dives[3].name = "forward pike";
  dives[3].difficulty = 2;
  dives[4].name = "inward straight twist";
  dives[4].difficulty = 5;

  // Assign random performances to athletes and generate random scores
  srand(time(NULL));
  for (int a=0; a<3; a++) {
    for (int p=0; p<NUMBER_OF_PERFORMANCES; p++) {
      // choose a random dive for this performance
      athletes[a].performances[p].dive =
                                          dives[(int)(rand()/(double)RAND_MAX*5)];
      for (int s=0; s<NUMBER_OF_SCORES; s++) {
        // choose a random judge's score
        athletes[a].performances[p].scores[s] =
                                                   rand()/(double)RAND_MAX*10;
      }
    }
  }
  // Display the results
  for (int a=0; a<3; a++)
    displayAthlete(athletes[a]);
}
```

Here is the output (although the scores will vary randomly) ... does it make sense to you?

```
Athlete: Art Class from Canada:
  Performed inward straight twist (diff: 5): [0.1, 6.7, 6.7, 2.1, 1.4, 3.2, 3.2, 3.4]
  Performed reverse pike (diff: 3): [5.4, 6.1, 0.6, 7.7, 4.7, 5.6, 9.0, 9.5]
  Performed reverse pike (diff: 3): [0.6, 7.8, 1.4, 2.9, 4.7, 2.6, 0.5, 3.3]

Athlete: Dan Druff from Germany:
  Performed inward straight twist (diff: 5): [4.8, 1.0, 6.2, 4.5, 1.1, 2.9, 1.2, 3.2]
  Performed reverse triple twist (diff: 4): [4.4, 6.4, 7.7, 5.5, 1.8, 3.8, 6.1, 9.4]
  Performed inward straight twist (diff: 5): [1.7, 8.5, 8.0, 2.4, 9.1, 5.8, 3.9, 2.0]

Athlete: Jen Tull from U.S.A.:
  Performed reverse pike (diff: 3): [6.4, 2.5, 3.7, 4.7, 7.4, 4.7, 0.9, 1.9]
  Performed reverse triple twist (diff: 4): [3.8, 3.1, 9.0, 8.1, 7.5, 5.4, 5.8, 3.0]
  Performed forward pike (diff: 2): [9.6, 9.1, 6.6, 8.1, 0.8, 5.0, 6.1, 3.2]
```

The **typedef** keyword actually allows you to have multiple names (i.e., aliases) for a type. So, for example, you can do this:

```
typedef AthleteType   Athlete;
typedef int           StudentNumber;
typedef char          Letter;
```

This would allow you to use the types **AthleteType** and **Athlete** interchangeably.

```
Athlete   athletes[20];  // instead of   AthleteType athletes[20];
Athlete   jen;           // instead of   AthleteType jen;
```

And you could use **char** and **Letter** interchangeably as well as **int** and **StudentNumber**:

```
typedef struct {
   char              *name;
   StudentNumber      studentID;    // instead of int studentID;
   Letter             gender;       // instead of char gender;
} StudentType;
```

But there is a need to be cautious with these aliases. The more you make, the harder it is for someone else to know what type a variable is actually storing. Imagine that we have these variables being used in our code:

```
Name     n;
Age      a;
Status   s;
```

Just looking at this code, we cannot be sure what each of these types are. **Name**, for example, may be a simple **char** * string or it could be a **struct** with multiple attributes. **Age** could be an **int** or perhaps a **float** where we want more accurate ages. And **Status** … well … we have no clue what that is. We would need to go looking through all our C code (which may be spread out over many files) in order to hunt down the particular **typedef** definition. So, we should try to minimize the number of aliases that we create so as to make life easier for our fellow programmers who will read and maintain our code at some time in the future.

When creating objects in JAVA, there is a tendency to be wasteful because we don't usually worry about memory usage and memory allocation/deallocation. The same could be true in C. However, we are normally more space-usage-conscious when programming in the lower-level C language. This can be important in embedded systems where memory is scarce.

One way to be space-efficient, is to allow variables of different types to share the same chunk of memory.

> *A* ***union*** *is a value that may have any of several representations or formats within the same position in memory.*

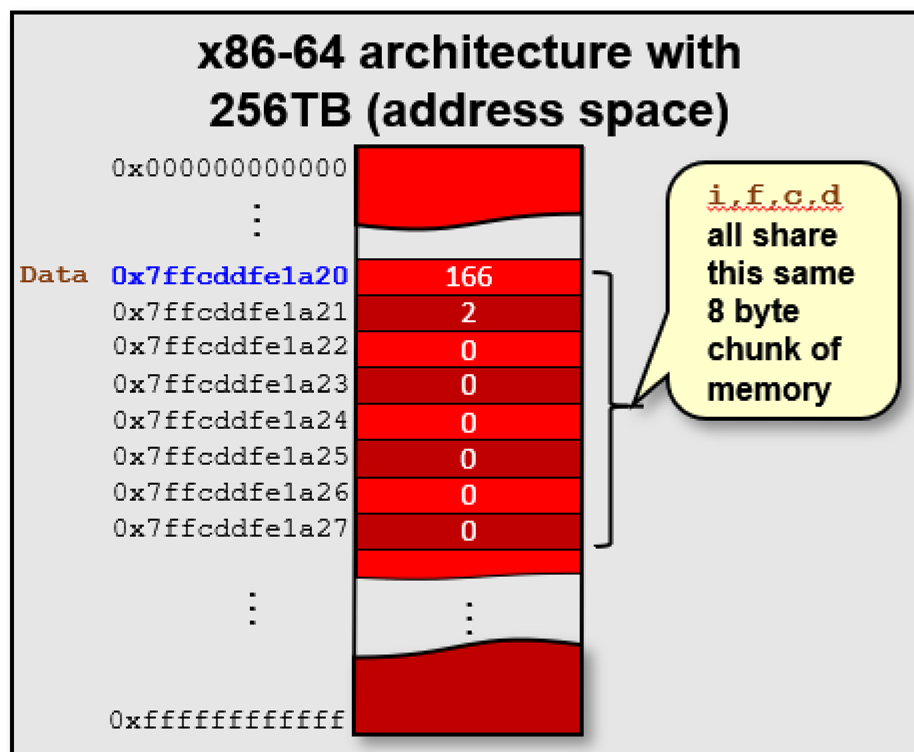
The **union** itself does not have a specific *type* to it. Rather, it kind of "takes on the role" of one of its member types dynamically. It is similar to the idea of having a chunk of memory changing its type at various times throughout a program's execution. The main advantage of using **unions** is to be able to access pieces (or components) of a type value. In C, defining a **union** is as easy as replacing the word **struct** by **union**.

Consider this **union**:

```
typedef union {
    int     i;
    float   f;
    char    c[5];
    double  d;
} Data;
```

How much memory space does this **union** take up? If it were a **struct** instead of a **union**, it would require (4 + 4 + 5 + 8 + 3 bytes padding) = **24** bytes. However, when a **union** is used, since all the members of the **union** overlap one another in memory, it only takes up MAX(4, 4, 5, 8) = **8** bytes.

```
printf("sizeof(Data): %zu\n", sizeof(Data));        // returns 8
```



The downside, is that we only get to use one member at a time:

```
Data   data;

data.i = 678;
printf("data.i is:    %d\n", data.i);  // 678
data.f = 3.1415;
printf("data.f is:    %f\n", data.f);  // 3.141500
strcpy(data.c, "Hi!");
printf("data.c is:    %s\n", data.c);  // Hi!

printf("data.i has been overwritten: %d\n", data.i);   // 2189640
printf("data.f has been overwritten: %f\n\n", data.f); // 0.000000
```

Notice that once we give a value to any member of the **union**, all other **union** member's data is overwritten, and will therefore be invalid.

There are, however, some uses of **unions** in which we don't care about the memory being overwritten. Consider this example which is useful for determining the bytes that make up an integer:

```
typedef union {
    unsigned int  value;
    unsigned char bytes[4];
} DecomposableInteger;


DecomposableInteger   number;

number.value = 584340;
printf("%d, %d, %d, %d\n", number.bytes[0], number.bytes[1],
                           number.bytes[2], number.bytes[3]);
```

The code allows you to treat the number either as an **unsigned int**, or as an array of **4 bytes**. So, in this example, we put a large number (i.e., 584340) into the **union**. Then, we extract it as the 4 bytes that are used to store the number. The resulting output is:

```
148, 234, 8, 0
```

Which is correct, since **148**$*2^0$ + **234**$*2^8$ + **8**$*2^{16}$ + **0**$*2^{32}$ = 148 + 59,904 + 524,288 = 584340

In this example, the least significant byte is first in the sequence of bytes. This ordering is called ***Little Endian Byte Order***. The least significant byte (the "**little** end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory. A different ordering is the ***Big Endian Byte Order***. In that case, the most significant byte (the "**big** end") of the data is placed at the byte with the lowest address.

Endianness is a property of the CPU, not of the operating system. If you want to determine the ordering of the bytes on your machine, you can type `lscpu` into the terminal window:

```
student@COMPBase:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              1
On-line CPU(s) list: 0
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               30
Model name:          Intel(R) Core(TM) i7 CPU         870  @ 2.93GHz
Stepping:            5
CPU MHz:             2926.002
BogoMIPS:            5852.00
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            8192K
NUMA node0 CPU(s):   0
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm
constant_tsc rep_good nopl xtopology nonstop_tsc cpuid pni monitor ssse3
sse4_1 sse4_2 x2apic hypervisor lahf_lm pti
student@COMPBase:~$
```

Here is the info.

Here is another example that lets us store a phone number string in format (xxx)xxx-xxxx and then allows us to extract the individual pieces:

Code from **unions.c**

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

typedef union {
  char whole[14];  // enough to store (613)220-2600\0
  struct {
    char openPar;
    char area[3];
    char closePar;
    char prefix[3];
    char dash;
    char lineNum[4];
    char null;
  } parts;
} PhoneNumber;
```

```c
int main() {
  PhoneNumber myNumber;

  strcpy(myNumber.whole, "(613)520-2600");
  printf("\noriginal num: %s\n", myNumber.whole);

  strcpy(myNumber.parts.area, "416");
  strcpy(myNumber.parts.prefix, "555");
  strcpy(myNumber.parts.lineNum, "8888");
  myNumber.parts.null = 0;

  printf("whole num:     %s\n", myNumber.whole);       // corrupted now
  printf("areaCode:      %s\n", myNumber.parts.area);
  printf("prefix:        %s\n", myNumber.parts.prefix);
  printf("lineNumber:    %s\n", myNumber.parts.lineNum);

  return 0;
}
```

The output is as follows:

```
original num:  (613)520-2600
whole num:     (416
areaCode:      416
prefix:        555
lineNumber:    8888
```

Notice that once we alter the original number, we cannot display it using the **whole** field. That is because when we wrote the individual **parts.area**, **parts.prefix** and **parts.lineNum**, each time it added a **'\0'** null terminating character:

after writing **whole**:	(	6	1	3	)	5	2	0	-	2	6	0	0	\0
after writing **area**:	(	4	1	6	\0	5	2	0	-	2	6	0	0	\0
after writing **prefix**:	(	4	1	6	\0	5	5	5	\0	2	6	0	0	\0
after writing **lineNum**:	(	4	1	6	\0	5	5	5	\0	8	8	8	8	\0

In general, **unions** should be used sparingly, as it complicates what is happening in your program and could make things genuinely confusing for the typical programmer when many **unions** are used since at any instant of time, we may not be sure which part of the **union** was written last. Therefore, we may make wrong assumptions and end up with wrong data.