C Notes
The "struct" Definition: A Precursor to OOP

One of the main distinctions between C and C++ is that C++ allows user-defined classes as part of its support of OOD/OOP. C was introduced before OOD/OOP ideas were fully developed, so it does not provide true class definitions and the associated OOD/OOP concepts of encapsulation, inheritance, and polymorphism. But it does provide a simple earlier version of the "class," called a "struct." A "struct" allows the programmer to define a structured data item with multiple named fields. Structs can contain pointers and so can be used to implement simple data structures. But struct definitions don't include functions that operate on the data, and so do not support the full functionality of classes. For example, you cannot build a true abstract data type (ADT) using C structs like you can with C++ or Java classes.

Defining a Single Struct Variable
Define a structured variable "x" with three int fields a, b, and c.

```
struct {
      int a;
      int b;
      int c;
} x;

x.a = 1;
x.b = 2;
x.c = 3;
```

Note in this example, no identifier is provided immediately following the "struct" keyword, while the identifier "x" is provided between the right brace and the semicolon that ends the definition. The significance of this will be discussed below.

Defining a Datatype from a Struct

```
struct rec {
      int a;
      int b;
      int c;
}

struct rec x;     // x is an instance of "struct rec"
x.a = 1;
x.b = 2;
x.c = 3;

struct rec y;     // y is an instance of "struct rec"
y.a = 4;
y.b = 5;
y.c = 6;
```

Note in this example the identifier "rec" is placed immediately after the keyword "struct" and before the opening left brace that starts the struct definition. In the previous example, "x" became the name of a single instance of a struct that had no official or "type" name. In this case the struct is given a type name, but no variable is created. If the struct has a type name, the type name can be used in a later statement to create a variable of that type. The keyword "struct" becomes part of the type name.

Improved Datatype Using Struct
It's a little awkward to have to use the phrase "struct rec" for the datatype instead of just "rec." So we create a type name alias using the "typedef" definition.

```
struct rec {
      int a;
      int b;
      int c;
};
typedef struct rec rec; // "rec" is an alias for "struct rec"

rec x;                  // x is an instance of "rec"
x.a = 1;
x.b = 2;
x.c = 3;

rec y;                  // y is an instance of "rec"
y.a = 1;
y.b = 2;
y.c = 3;
```

Value vs. Pointer Semantics
C and C++ allow structured data to be created in two different ways: directly on the stack, or indirectly through pointers on the heap. This is unlike Java, which allows structured data to be created only indirectly through references on the heap.
C provides extra syntax for accessing structured data depending on whether it is being accessed directly on the stack or indirectly on the heap.

Example

```
struct node {
      int value;
      struct node *next;
};
typedef struct node node;

node a;
a.value = 5;
a.next = NULL;

node *b;
b = (node *) malloc(sizeof(node));
(*b).value = 5;
(*b).next = NULL;
      // Or
b->value = 5;
b->next = NULL;
```

In the case of "node a," variable a is a structured variable created directly on the stack. The variable name "a" is not a pointer or reference, it is the full structured value. In the case of "node *b," variable b is not a node but a pointer to a node on the heap.

If we want to access a field of a structured variable that is held in a value variable, the selection operator is dot ".". If we access a field of a structured variable that is accessed indirectly through a pointer, the selection operator is arrow "->". But the arrow operation is really nothing but a shortcut for using the indirection operator asterisk "*" followed by the value access operator dot ".".

By comparison, Java syntax is somewhat simpler since Java does not provide direct access to value variables, only indirect access through references. Somewhat confusingly, Java adopted the dot notation for the selection operation for indirect access, which is used by C for direct access.

Related: Union
A union definition resembles a struct definition. But a union introduces multiple fields that physically occupy the same space. There are several applications for such a definition. Imagine a variable whose value has different types depending on context. The union is one solution. Multiple fields with different types can be joined together and not waste space. The intent is that only one of the fields will be valid and in use at any given time.

```
union number {
      int integer;
      float floatingpoint;
};
typedef union number number;

number x;
x.integer = 5;
…
x.floatingpoint = 54.3;
…
```

Use the "integer" field to store an integer, use the "floatingpoint" field to store a float. But the two fields share the same storage, there are not really two separate storage slots, just two names for referring to the same slot.

Another useful application is relevant to one COMP 222 topic: floating point representation in binary. If we write the following code:

```
float v = 56.254;
printf("%f\n",v);
```

then we treat the floating point value as a floating point in both storage and output format. Output is:

```
56.254002
```

There's a reason for the odd extra digit in the far right position, we'll discuss shortly.

But what if we wanted to see the storage for the floating point number in a binary bit-by-bit format? We might try something like this:

```
unsigned w = (unsigned) v;
printf("%x\n",w);
```

The output is:

```
38
```

So all this did was cast the value 56.254 to an integer 56 then convert to base 16.

Something like this is a little closer:

```
printf("%x\n",v);
```

Here the compiler may generate a warning that the specifier %x doesn't match the datatype of the expression v. But it will compile and run, with output:

```
20000000
```

So it's still not showing us what we're looking for. The following actually works:

```
float v = 56.254;
unsigned *p = (unsigned *) &v;
printf("%x",*p);
```

The output is:

```
42610419
```

This is a 32-bit value using the IEEE 754 format for storing floating point numbers in binary. Let's take it apart (detailed rules will be covered in class):

IEEE 754 uses the following 3 fields, counting bits starting with least significant bit 0 on the right:

- bit 31: sign bit (0 for positive, 1 for negative)
- bits 30-23: 8 bit base 2 exponent in excess 128 notation
- bits 22-0: normalized mantissa or fraction, also in base 2.

We break the bits down like this:

42610419 = 0100 0010 0110 0001 0000 0100 0001 1001
Sign (bit 31): 0
Exponent(bits 30-23): 100 0010 0 = 1000 0100 = 84 = 8X16+4 = 132
132 – 127 = 5
Mantissa (bits 22-0): 110 0001 0000 0100 0001 1001
Positions represent negative powers of 2
Normalized means the 1 to the left of the decimal is implicit, we have to add it in
$1 + 2^{-1} + 2^{-2} + 2^{-7} + 2^{-13} \ldots =$
1 + 1/2 + 1/4 + 1/128 + 1/1024 + 1/2048 + … =
1 + 0.5 + 0.25 + 0.0078125 + 0.0001220703125 + … =
1.7579345703125 …
Putting the pieces together
+ 1.7579345703125 X 2^5 = + 1.7579345703125 X 32 = 56.25390625 …

Since we truncated the summation of the negative powers of two, the answer is approximate. But if you look back at the original output of the floating point value, you can see that our original value of 56.254 was not represented exactly as we wrote it either. Now you can see why. Any floating point value must be decomposed into a fraction that is a sum of negative powers of two. Not just any arbitrary value can be represented exactly in this notation, there will frequently be a small round off error. In fact, only a finite set of floating point values can be represented exactly in any finite precision notation, other values will have to be approximated. This is a well-known limitation of the IEEE 754 standard. We will discuss it in more detail shortly.

Back to our original problem of how to see the bit pattern being used to represent a floating point number. We came up with a way to see the bits, but we've resorted to a trick. A more formally correct way to get what we want is to define a union variable:

```
union { float f; unsigned u; } z;
z.f = v;
printf("%x\n",z.u);
```

This is closer to the correct way to get what we want. It also prints the right answer. In the assignment

```
z.f = v;
```

we assign a float value to the float field within the union. Then in the print statement, we print the content of

```
z.u
```

which interprets the content of the field as an unsigned int, in other words, a 32-bit bit pattern. In our union, the f field and the u field are different names for the same physical storage. The union therefore allows us to take data in one format and reinterpret it in a different format.