Array of Structs

The following declares an array named "numbers" which holds 1000 `struct fraction`'s.

```
struct fraction numbers[1000];

numbers[0].numerator = 22;       /* set the 0th struct fraction */
numbers[0].denominator = 7;
```

Here's a general trick for unraveling C variable declarations: look at the right hand side and imagine that it is an expression. The type of that expression is the left hand side. For the above declarations, an expression which looks like the right hand side (`numbers[1000],` or really anything of the form `numbers[...]`) will be the type on the left hand side (`struct fraction`).

Pointers

A pointer is a value which represents a reference to another value sometimes known as the pointer's "pointee". Hopefully you have learned about pointers somewhere else, since the preceding sentence is probably inadequate explanation. This discussion will concentrate on the syntax of pointers in C -- for a much more complete discussion of pointers and their use see http://cslibrary.stanford.edu/102/, Pointers and Memory.

Syntax

Syntactically C uses the asterisk or "star" (*) to indicate a pointer. C defines pointer types based on the type pointee. A `char*` is type of pointer which refers to a single `char`. a `struct fraction*` is type of pointer which refers to a `struct fraction`.

```
int* intPtr;   // declare an integer pointer variable intPtr

char* charPtr; // declares a character pointer --
               // a very common type of pointer


// Declare two struct fraction pointers
// (when declaring multiple variables on one line, the *
//  should go on the right with the variable)
struct fraction *f1, *f2;
```

The Floating "*"

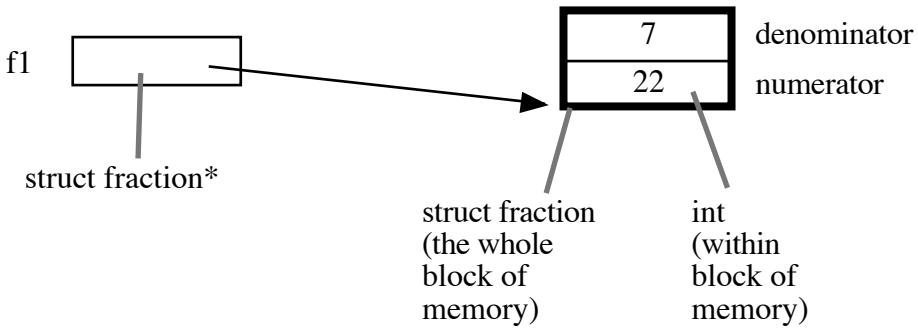
In the syntax, the star is allowed to be anywhere between the base type and the variable name. Programmer's have their own conventions-- I generally stick the * on the left with the type. So the above declaration of intPtr could be written equivalently...

```
int  *intPtr;        // these are all the same
int * intPtr;
int*  intPtr;
```

Pointer Dereferencing

We'll see shortly how a pointer is set to point to something -- for now just assume the pointer points to memory of the appropriate type. In an expression, the unary * to the left of a pointer dereferences it to retrieve the value it points to. The following drawing shows the types involved with a single pointer pointing to a struct fraction.

```
struct fraction* f1;
```



Expression	Type
f1	struct fraction*
*f1	struct fraction
(*f1).numerator	int

There's an alternate, more readable syntax available for dereferencing a pointer to a struct. A "`->`" at the right of the pointer can access any of the fields in the struct. So the reference to the numerator field could be written `f1->numerator`.

Here are some more complex declarations...

```
struct fraction** fp;      // a pointer to a pointer to a struct fraction

struct fraction fract_array[20];       // an array of 20 struct fractions

struct fraction* fract_ptr_array[20];  // an array of 20 pointers to
                                       // struct fractions
```

One nice thing about the C type syntax is that it avoids the circular definition problems which come up when a pointer structure needs to refer to itself. The following definition defines a node in a linked list. Note that no preparatory declaration of the node pointer type is necessary.

```
struct node {
   int data;
   struct node* next;
};
```

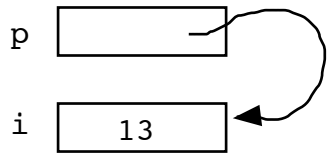
The & Operator

The & operator is one of the ways that pointers are set to point to things. The & operator computes a pointer to the argument to its right. The argument can be any variable which takes up space in the stack or heap (known as an "LValue" technically). So `&i` and `&(f1->numerator)` are ok, but `&6` is not. Use & when you have some memory, and you want a pointer to that memory.

```
void foo() {
   int* p;  // p is a pointer to an integer
   int i;   // i is an integer

   p = &i;  // Set p to point to i
   *p = 13; // Change what p points to -- in this case i -- to 13

   // At this point i is 13. So is *p. In fact *p is i.
}
```



When using a pointer to an object created with `&`, it is important to only use the pointer so long as the object exists. A local variable exists only as long as the function where it is declared is still executing (we'll see functions shortly). In the above example, `i` exists only as long as `foo()` is executing. Therefore any pointers which were initialized with `&i` are valid only as long as `foo()` is executing. This "lifetime" constraint of local memory is standard in many languages, and is something you need to take into account when using the & operator.

NULL

A pointer can be assigned the value 0 to explicitly represent that it does not currently have a pointee. Having a standard representation for "no current pointee" turns out to be very handy when using pointers. The constant NULL is defined to be 0 and is typically used when setting a pointer to NULL. Since it is just 0, a NULL pointer will behave like a boolean false when used in a boolean context. Dereferencing a NULL pointer is an error which, if you are lucky, the computer will detect at runtime -- whether the computer detects this depends on the operating system.

Pitfall -- Uninitialized Pointers

When using pointers, there are two entities to keep track of. The pointer and the memory it is pointing to, sometimes called the "pointee". There are three things which must be done for a pointer/pointee relationship to work...

(1) The pointer must be declared and allocated

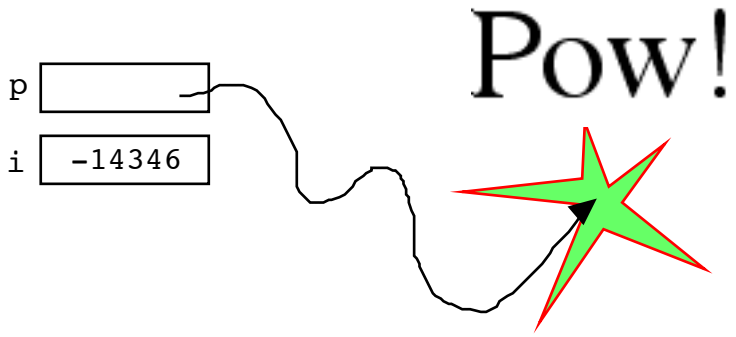
(2) The pointee must be declared and allocated

(3) The pointer (1) must be initialized so that it points to the pointee (2)

The most common pointer related error of all time is the following: Declare and allocate the pointer (step 1). Forget step 2 and/or 3. Start using the pointer as if it has been setup to point to something. Code with this error frequently compiles fine, but the runtime results are disastrous. Unfortunately the pointer does not point anywhere good unless (2) and (3) are done, so the run time dereference operations on the pointer with `*` will misuse and trample memory leading to a random crash at some point.

```
{
   int* p;

   *p = 13;     // NO NO NO p does not point to an int yet
                // this just overwrites a random area in memory
}
```



Of course your code won't be so trivial, but the bug has the same basic form: declare a pointer, but forget to set it up to point to a particular pointee.

Using Pointers

Declaring a pointer allocates space for the pointer itself, **but it does not allocate space for the pointee**. The pointer must be set to point to something before you can dereference it.

Here's some code which doesn't do anything useful, but which does demonstrate (1) (2) (3) for pointer use correctly...

```
int* p;     // (1) allocate the pointer
int i;      // (2) allocate pointee
struct fraction f1;  // (2) allocate pointee

p = &i;     // (3) setup p to point to i
*p = 42;    // ok to use p since it's setup

p = &(f1.numerator);       // (3) setup p to point to a different int
*p = 22;

p = &(f1.denominator);     // (3)
*p = 7;
```

So far we have just used the & operator to create pointers to simple variables such as `i`. Later, we'll see other ways of getting pointers with arrays and other techniques.

C Strings

C has minimal support of character strings. For the most part, strings operate as ordinary arrays of characters. Their maintenance is up to the programmer using the standard facilities available for arrays and pointers. C does include a standard library of functions which perform common string operations, but the programmer is responsible for the managing the string memory and calling the right functions. Unfortunately computations involving strings are very common, so becoming a good C programmer often requires becoming adept at writing code which manages strings which means managing pointers and arrays.