

Physics 305: Ch. 1

More topics in C

You've already seen that a little C is enough to solve some useful problems. However, as the complexity of the problems grows, we will find that some more advanced features in C will allow us to simplify our approaches.

In this chapter, we introduce a number of these advanced features on C: global variables, pointers, files, and structures. Far more detail is available in books on C.

1.1 Pointers and Arrays

The time has come to learn a bit about one of C’s most powerful features: pointers. A pointer is a new kind of variable. However, rather than storing data itself, it stores the internal memory address of another variable. In other words, it tells the computer where to find another variable. The name “pointer” comes from the idea that the pointer points to another variable.

Let’s begin by understanding what normally happens when you declare a variable. When you include a line in a program such as

```
double myvar;
```

what is actually happening is that the computer is setting aside memory (eight bytes in this case) to store the value of this variable. Whenever you refer to `myvar`, the computer accesses the value in that chunk of memory.

If we define a pointer, the computer sets aside a chunk of memory (eight bytes, usually) that holds the address (i.e. location) of another chunk of memory. For example, we might define a pointer to a `double` as

```
double *mypointer;
```

The asterisk (*) indicates that we are defining a pointer rather than a normal `double` variable.

We can then the manipulate the pointer as follows.

```
1  #include <stdio.h>
2  #include <math.h>
3  int main()
4  {
5      double myvar, *mypointer;
6      myvar = 5.0;
7      mypointer = &myvar;
8      printf("The value of myvar is %f\n", myvar);
9      printf("The value of mypointer is %f\n", *mypointer);
10     return(0);
11 }
```

Line 5 defines two variables, a normal `double` variable `myvar` and a pointer `mypointer`. Line 6 assigns a value to `myvar`. Line 7 is crucial: this assigns the address of `myvar` to `mypointer`. The ampersand (&) signals that we are accessing the address of the variable. To repeat, the value of `mypointer` is now the address that specifies the location of `myvar` in the computer’s memory. Of course, we generally don’t care about the value of the address; we care about the value of the variable that is held at that address. This latter quantity is accessed by “dereferencing” the pointer. By writing `*mypointer` (as in line 9) we tell the computer not to report the address itself,

but instead to go to that address and report the value of the variable found there! `*mypointer` really is a floating point variable and can be used as any such variable could, e.g.

```
*mypointer = 5.0;
*mypointer = *mypointer*2.0;
*mypointer = sqrt(*mypointer);
```

So, to review, the key operational aspects of pointers are 1) to use the `&` modifier to find the address of variables so as to be able to assign them to pointers, and 2) to use the `*` modifier to find out the value of the variable to which the pointer points.

1.1.1 Why bother?

This sounds, well, pointless. If we already have the target variable defined, why shouldn't we just refer to it directly instead of referring to it by an indirect reference? The advantages of pointers are 1) that they are variables, meaning that where they point can change during the course of execution of the program, and 2) they allow functions to modify locations in memory that the function otherwise wouldn't be able to address.

To understand the second point, we must understand what happens when you pass a variable to a function. In particular, the computer passes a *copy* of the value of the variable, not the variable itself. So, for example, if we have

```
1  #include <stdio.h>
2  #include <math.h>
3  double cos_squared(double theta)
4  {
5      /* Given an angle in radians,
6       calculate the square of the cosine of the angle */
7      double cosx;
8      cosx = cos(theta);
9      return(cosx*cosx);
10 }
11 int main()
12 {
13     double input, output;
14     input = 4.5;
15     output = cos_squared(input);
16     printf("The square of the cosine of %f is %f.\n",
17           input, output);
18     return(0);
19 }
```

when we call `cos_squared(input)` in line 15, the computer sets aside *new* memory to hold the value of `theta`, the variable declared in the function definition of line 3. It then evaluates the value of the argument of the function, in this case `input` itself, and copies that value (4.5) into the new memory. This new memory is *only* accessed by the name `theta` and is different than the memory associated with the name `input`. This means that if we change the value of `theta` inside the function `cos_squared`, we do not change the value of the variable `input`.

At the risk of belaboring this point, let me give another (more confusing) example:

```

1  #include <stdio.h>
2  #include <math.h>
3  int assign_var(double x)
4  {
5      x=3.0;
6      return(0);
7  }
8  int main()
9  {
10     double x;
11     int status;
12     x = 4.5;
13     status = assign_var(x);
14     printf("The value of x is %f.\n", x);
15     return(0);
16 }
```

The output of this program will be

```
The value of x is 4.500000
```

You may have thought that the printed value of `x` ought to be 3. However, there are two lessons here. First, the fact that there is a variable `x` in both `main` and `assign_var` doesn't mean that they are the same variable. In fact, they are different, because variables in C are local to their functions. Second, when the function `assign_var` is called, the computer assigns a new chunk of memory to hold the variable `x` that belongs to `assign_var`. The value of `main`'s `x` is copied into that chunk of memory. When `x` has a value assigned within `assign_var`, that affects only the new chunk of memory. The memory assigned to `main`'s `x` variable is not touched. When the `assign_var` function end, the memory for its `x` variable is de-assigned, which means that any information in there is lost!

Of course, we sometimes want to write functions that can modify the variables that are passed to it. One application of this is if we want the function to return multiple pieces of information (like the value of a function *and* its derivative). To do this, we will pass the function a pointer to the variable that will be modified. As an example of this, consider an alternative version of our previous program:

```
1  #include <stdio.h>
2  #include <math.h>
3  int assign_var(double *y)
4  {
5      *y=3.0;
6      return(0);
7  }
8  int main()
9  {
10     double x;
11     int status;
12     x = 4.5;
13     status = assign_var(&x);
14     printf("The value of x is %f.\n", x);
15     return(0);
16 }
```

Now the output of this program will be

```
The value of x is 3.000000
```

Note the difference in the function arguments in line 3. The asterisk before the variable name `y` means that `y` is a pointer to a variable of type `double`. The call of the function in line 12 is also different; we must pass an address in memory, not a `double` itself. Here, we choose to pass the address of our variable `x`. This address is assigned to the pointer variable `y`. In the assignment on line 5, the memory to which `y` is pointing is assigned the value of 3. Note that we do *not* write “`y=3.0;`”; this would be nonsense because pointers such as `y` can only hold information about memory addresses. Because `y` is pointing to the same memory that is assigned to the `double x`, the assignment in line 5 rewrites the value of `x` even though the name `x` is not available within the function `assign_var`.

Now that we see that pointers allow functions to change values of variables in the parent function that calls them, we can use this to write functions that return more than one value to their calling function. For example, in a Newton-Raphson program, we need to use both the value and derivative of a function. Rather than writing separate functions to calculate these two, we’d prefer to have only one function that computes and returns both values. Aside from the extra baggage of a second function, one could imagine circumstances in which large portions of the calculation of $f(x)$ could be reused in computing $f'(x)$. We could combine the two functions into one if we use pointers. For example, if $f(x) = e^{-x^2} - 0.5$, we might write

```
1     double myfunc(double x, double *deriv)
2     {
3         double expx2;
```

```

4     expx2 = exp(-x*x);
5     *deriv = -2.0*x*expx2;
6     return(expx2-0.5);
7     }

```

The functional value $f(x)$ is returned as the value of `myfunc`, while the derivative $f'(x)$ is assigned to the variable to which `deriv` is pointing. By combining the functions, we only need to compute the `exp` function once, which gains efficiency. In the Newton-Raphson program, we might call our function by the line

```
fvalue = myfunc(x, &fprime);
```

As a second example, consider our bisection root-finding method. We want to return the value of the root (of course), but also the number of iterations required, and perhaps a variable indicating whether any problems were encountered (like the initial root not being bracketed). Prior to pointers, we could only return a single number, so we had to pick between these. Now we could write a function

```
int bisect(double xlow, double xhi, double *root, int *iter)
```

that would return the value of the root in the variable pointed to by the pointer `root` and the number of iterations in the variable pointed to by the pointer `iter`. The function itself would return an integer that we would use to indicate success or failure of the algorithm. The convention is that a return value of 0 indicates success, while non-zero return values indicate problems. For example, we might return 1 if the initial bracket didn't contain a root and 2 if we used too many iterations. The meaning of the different return values should always be documented in comments at the top of the function. The advantage of this new `bisect` function is that it can be applied in many circumstances with no modifications (save for the name of the function to be bisected, a problem we will solve below). To ensure this, it should produce no printed output and should not halt the program if it fails; all decisions about output and dealing with failure are left to the calling function.

To call our `bisect` function, we need to have some variables to store the results, e.g.,

```

1  int main()
2  {
3      double result;
4      int iterations, success;
5      success=bisect(0.0, 2.0, &result, &iterations)
6  }

```

After line 5, the root is in the variable `result`, the number of iterations is in the variable `iterations`, and the variable `success` can be tested to see if all went well.

A more advanced construction that could take the place of line 5 is

```
if (success=bisect(0.0, 2.0, &result, &iterations)) {
    /* bisect() failed; look at success to decide what to do */
}
/* If bisect() succeeded, we will proceed to here */
```

Notice that we use only a single equal sign here! The conditional test of the `if` statement includes the assignment of the result of `bisect` call to the variable `success`. The value of that assignment (i.e. the value of `success`) is then used in the conditional logical test. A zero value is false; anything non-zero is true. Hence, if `success` is non-zero, the value of the condition is true and we execute the statements in the `if` block. If `success` is zero, then we skip the block. If we had merely used

```
if (bisect(0.0, 2.0, &result, &iterations) {
```

we still would branch to the `if` block on a non-zero return of `bisect`, but we wouldn't have stored the return value to help with diagnosing the problem that `bisect` encountered.

1.1.2 Pointer Gotchas

Beware of the following common errors with pointers!

First, when you define a pointer variable, you allocate memory to store an address in memory. You have *not* allocated space to which that pointer points. That is, the code

```
double *pnt;
*pnt = 3.0;
```

is nonsense. The pointer `pnt` is pointing nowhere in particular, so that `*pnt` is attempting to use a random place in memory. Usually that place is not somewhere you have permission to use, so the code will crash with a “Segmentation Fault” error. To use `*pnt`, you have to assign the address of a legal memory location to `pnt`.

To repeat, always be certain that your pointers are pointing to legal memory before you try to dereference them!

A corollary of this gotcha is that pointers are not a replacement for ordinary variables. If you need to store a floating point number, define a `double`. Only use a pointer when you have to pass or otherwise manipulate a memory address.

Second, even though your pointer is pointing to legal memory, it may not be pointing where you intended, so that when you change the value of the memory referenced by the pointer, you don't accomplish what you expect. These can be difficult errors to find, so be vigilant!

Third, be careful of the type of your pointer. Pointers point to memory, but the type of pointer determines how that memory is interpreted. For example, the code fragment

```

double fvalue, *fptr;
int *iptr;
fptr = &fvalue;
iptr = &fvalue;
fvalue = 3.0;
printf("The double value is %f; the int value is %d\n",
      *fptr, *iptr);

```

will not print 3 twice. The value of `*fptr` is 3 as expected, but the value of `*iptr` will be some huge and horrible number. What has happened? The computer has gone to the memory location of `fvalue` but has interpreted the 4 bytes that were supposed to encode the value of a `double` (which, recall, is in binary exponential notation) as an `int`.

Fortunately, the compiler will usually warn you when you assign a pointer of one type to an address of a variable of another type. Pay attention to these warnings. Oddly, enough, this ability to bit-wise interpret a variable of one type as a variable of another type can actually be useful at times....

Fourth, using the value of a pointer in arithmetic is generally easy: `a**pnt`, `a-*pnt`, and `a**pnt` all work as expected, adding, subtracting, and multiplying the value of `*pnt` to the value of `a`. So of course `a/*pnt` should be `a` divided by `*pnt`, right? Not so fast! `/*` begins a comment! You are always free to use parentheses in expressions, so `a>(*pnt)` would work as expected.

Fifth, the precedence of the asterisk is lower than certain other unary operations. While `*pnt+1` adds the value of `*pnt` to 1, `*pnt++` adds 1 to the pointer address (which is legal, as we'll discover) and then tries to dereference that! Use `(*pnt)++` to get your normal increment.

Finally, don't forget to dereference! If you have a pointer `pnt` that points to a variable you're trying to use, you need to use `*pnt`, not `pnt`, to get that value. If you forget the asterisk, you will be doing math on the address, which is a very important but completely different aspect of pointers that we will deal with below.

1.1.3 Pointers to Functions

We now move to a different kind of pointer. C gives you the ability to create pointers that refer to functions, which means that you can use a function using the name of the pointer rather than the name of the function. This feature will allow us to remove the last obstacle to our writing numerical routines that are completely reusable. Until now, our bisection function (to pick an example) had to refer to the target function by a particular name (e.g. `myfunc`). This meant that if we changed the name of our function, we had to change the bisection function. Worse, if we wanted to find the roots of two different functions in the same program, we would have to have two copies of the bisection function, one for each target name. We will now be able to avoid this by passing the bisection function the address of our target function. Internal to the

bisection function, we will invoke our function by the name of the pointer. Let's see how this works.

To create a pointer to a function, we must specify both the output type of the function and the types of all of its arguments. For example, the function we intend to find roots of has the definition

```
double myfunc(double x);
```

In other words, this function takes a single argument of type `double` and returns a value of type `double`. We can define a pointer to such a function by the statement

```
double (*funcptr)(double);
```

The syntax here is important. A variable named `funcptr` has been created. The set of parentheses around the name, when combined with the asterisk, indicates that it is a pointer to a function. The information in the next set of parentheses describes the argument list. As usual, commas separate the arguments. Here, we have a single argument, which is of type `double`.

```
double (*funcptr)(double x);
```

would also be legal, but the name of the variable is ignored by the compiler. It can be helpful for the reader, however, as it hints to the purpose of that variable. Here's an example with more arguments, drawn from our Newton-Raphson case above:

```
double (*myfuncptr)(double x, double *deriv);
```

Why do we need this pair of parentheses around the pointer's name? The problem is that

```
double *funcptr(double x);
```

declares a function (not a pointer to a function) that takes a `double` as an argument and returns a pointer to a `double` as its output. This ambiguity is removed by the parentheses, but it does make for a more complicated syntax!

Once you have the pointer defined, then you can link it to a function by statements such as

```
funcptr = &myfunc;
```

Actually, the ampersand `&` is optional, because there is no other possible interpretation of

```
funcptr = myfunc;
```

You can then evaluate the function by

```
fvalue = (*funcptr)(3.0);
```

The call `(*funcptr)(3.0)` is equivalent to `myfunc(3.0)`.

At our stage, we will only be using pointers to functions so that we can build reusable numerical routines, i.e. routines that rely on functional values but in which we need not explicitly use the name of the function supplying those values. For example, consider our bisection method again. We of course will write a function to implement the mathematical function whose roots are being found. But now we will write the `bisect` function in such a way that the name of this target function is not used inside `bisect`. We do this by passing the address of the target function as an argument. Our definition of `bisect` is now

```
int bisect(double (*func)(double), double xlow, double xhi,
          double *root, int *iter)
```

Inside `bisect` we evaluate the value of our target function by calls to `(*func)(x)`.

If our target function is named `myfunc`, then we can invoke `bisect` by

```
int bisect(myfunc, 0.0, 1.0, &result, &iterations)
```

Again, we could use `&myfunc` to stress that we are passing an address, but with functions, the ampersand is implicit and usually omitted.

Here, then, is a fully worked example of the ideas we've been discussing:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int bisect(double (*func)(double), double xlow, double xhi,
5            double tolerance, double *root, int *iter);
6
7  double myfunc(double x) {
8      return tan(x)-3.0;
9  }
10
11 int main()
12 {
13     double result;
14     int iterations, success;
15     if (success=bisect(myfunc, 0.0, 1.5, 1e-7, &result, &iterations)) {
16         if (success==1) {
17             printf("We didn't bracket the root");
18         }
19         if (success==2) {
20             printf(
21                 "We didn't reach full convergence after %d iterations\n",
22                 iterations);
23             printf("Root is approximately %10.8f\n", result);
```

```
24     }
25     } else {
26         printf("Root is %10.8f\n", result);
27     }
28     return(0);
29 }
30
31
32 int bisection(double (*func)(double), double xlow, double xhi,
33             double tolerance, double *root, int *iter)
34 /* We search for root in func() between xlow and xhi.
35  * The fractional tolerance being sought is in tolerance
36  * If the given tolerance is negative, make it positive but treat as
37  * absolute
38  * The value of the root is returned in *root (double precision!)
39  * The number of iterations is returned in *iterations
40  * The returned value ==0 if all is well,
41  * ==1 if the initial bracket did not contain a root
42  * (in which case *root and *iter are undefined)
43  * ==2 if the maximum number of iterations was exceeded (in which case
44  * *root is the best guess and *iter is the number performed) */
45 {
46     double flow, fhi, fmid;
47
48     *iter = 0;
49     if (xlow>xhi) {
50         /* Reverse the brackets if needed */
51         fmid = xlow; xlow = xhi; xhi = fmid;
52     }
53     flow = (*func)(xlow);
54     fhi = (*func)(xhi);
55     /* If tolerance is positive, update it to the scale of
56     the function */
57     if (tolerance>0) tolerance = tolerance*(fabs(flow)+fabs(fhi))/2.0;
58     else tolerance = fabs(tolerance);
59
60     if (fabs(fhi)<tolerance) {
61         *root = xhi; return 0; /* The upper bound was a root */
62     }
63     if (fabs(flow)<tolerance) {
64         *root = xlow; return 0; /* The lower bound was a root */
65     }
66
67     if (flow*fhi>=0.0) return 1; /* We didn't bracket a root. */
68
```

```

69     /* Now we perform our bisection loop */
70     for (*iter=1;*iter<=40;(*iter)++) {
71         *root = 0.5*(xhi+xlow);
72         fmid = (*func)(*root);
73         if (fabs(fmid)<tolerance) return 0; /* Successful return */
74         if (fmid*fhi<0.0) {
75             /* The root is in the upper half */
76             xlow = *root; flow = fmid;
77         } else {
78             /* The root is in the lower half */
79             xhi = *root; fhi = fmid;
80         }
81     }
82     /* If we get here, we exceeded the number of iterations */
83     return 2;
84 }

```

The `bisect` routine is now portable: nothing need be changed in the function to use in different programs or even in different circumstances within the same program.

1.2 Arrays

We introduced arrays in the first packet, but there are some important aspects that we omitted at that time. Recall that an array is a vector of variables with a numerical index. In other words,

```
double arr[10];
```

creates 10 variables of type `double`. The variables can be used by referring to `arr[0]`, `arr[1]`, ..., `arr[9]`. You can use these 10 variables just like any other `double` variables.

We often want to pass the whole array to a function. This is straight-forward: we define the function

```
double myfunc(double myarr[], int size)
```

Inside `myfunc`, we can use the array `myarr` by indexing its elements: `myarr[0]`, `myarr[1]`, etc. Note that the size of the array is not indicated in the brackets in the argument list. You can put the size there to cue yourself, but the compiler will ignore it. Therefore, it is common to pass the size of the array into functions alongside of the array itself. The function can be invoked by, e.g.,

```
result = myfunc(arr, 10);
```

The crucial point about this is that the changes you make to elements of `myarr` inside `myfunc` do affect the values of the elements of `arr` in the calling function! This is different than the normal behavior, in which a function is only working on its local copy of a variable.

What is going on? The secret is that the name `arr` used *without* the bracketed index is actually referring to the memory address of the first element of the array, i.e. `&(arr[0])`. When we put `arr` in the argument list of the function, we are not passing copies of the 10 variables in the array, but rather passing the address of the first one. When you use `myarr[3]` inside of `myfunc`, you are actually doing a (hidden) pointer dereference! This is made possible by the fact that the 10 variables defined by the original array definition are all contiguous in memory. In other words, the array definition set aside 40 bytes in this case, but let's you refer to it as 10 4-byte numbers.

The relation between pointers and arrays is very powerful and important. If you have a pointer to a double and an array

```
double *ptr, arr[10];
```

then `ptr = arr;` sets the pointer to point to the location of `arr[0]`. `ptr = &(arr[0]);` does the same thing. `*ptr` will now give the same value as `arr[0]`. But we can also use array notation on the pointer! So `ptr[0]` will also give `arr[0]`, and `ptr[5]` will give `arr[5]`.

It gets better. Addition and subtraction on pointers is defined so that one moves up and down in memory not in units of 1 byte, but rather in units of the size of the type of variable. Hence, `*(ptr+5)` will give the value of `arr[5]`. `ptr=ptr+1;` or `ptr++;` makes `ptr` point to the location of `arr[1]`. There is no difference between the expressions `*(ptr+5)` and `ptr[5]`. Note that adding a floating point variable to a pointer doesn't make sense and will generate a compile-time error.

When you pass an array to a function, you actually have the choice as to whether to catch the address being passed as a pointer or as an array. In other words, the following two function declarations are equivalent (well, almost, but we're not going to discuss the difference):

```
double myfunc(double myarr[], int size)
```

or

```
double myfunc(double *myarr, int size)
```

Either way, you can use `myarr[0]` and so forth.

It is important to note that C does not provide any protection against referring to an index that is out of the defined range. If you define

```
double arr[10];
```

and then refer to `arr[10]`, `arr[-1]`, or some such, you will not get a reproducible result. If the memory you refer to is out of the range that your program owns, you will get a “Segmentation Fault”. Otherwise, you will overwrite some other variable!

Another important point about arrays is that C does not provide any tools for copying arrays or doing bulk arithmetic. In other words, if you define

```
double a[10], b[10];
```

the command `a=b` does *not* copy the elements of `b` into `a`. Similarly, `b=b+3` does not add 3 to each element of `b`. To do these operations, you should use `for` loops, e.g.

```
for (j=0;j<10;j++) a[j] = b[j];
for (j=0;j<10;j++) b[j] += 3.0;
```

Commands like `a=b` or `b=b+3` are actually attempting to do assignments based on the memory addresses, which not only is not what you meant to do but will also end up scrambling your data structures!

1.2.1 Varying the size of arrays with memory allocation

So far, all of the arrays we have defined have had a fixed size, set by the variable declaration in the source code, e.g.,

```
double a[10];
```

A reasonably common situation is to want to change the size of the array at run-time, e.g., in response to user input or the length of an input file. The simplest way to do that is do oversize your array and then use only part of it, e.g.,

```
#define MAXSIZE 1000
double a[MAXSIZE];
int actual_size;
```

If you enforce that `actual_size` is less than `MAXSIZE`, then you’re free to only use elements between 0 and `actual_size-1` and ignore the rest. This wastes memory (but you have a lot of that....) and risks failing in a case where `actual_size` is bigger than `MAXSIZE`, but it can be an easy and reasonable solution.

In certain circumstances, the newest versions of C will allow statements like

```
float my_function(int size, float some_other_parameter) {
    double a[size];
    // and the rest of your function
}
```

allowing you to have run-time defined array sizes in your functions.

However, the classic way to solve this problem is to declare a pointer and then allocate new memory to that pointer, so that the pointer can then be used as an array. Allocating new memory is done with the `malloc()` function. As an example:

```
int size;
double *arr;
sscanf("%d", &size); // Read in the size from the user
arr = (double *)malloc(size*sizeof(double));
```

The command `sizeof(double)` returns the size in bytes of a variable type, here `double`, which is 8 bytes. So if the user entered 10 for the variable `size`, the `malloc` statement would get 80 bytes from the operating system and assign the address of the first byte to the pointer `arr`. We have to prepend the expression with `(double *)` for arcane reasons, namely that the compiler is very paranoid about assigning one kind of pointer to another (because it changes how `arr[0]` would be interpreted) and it will issue a compile-time warning without this “type cast”.

It is good form to use the `sizeof` command because you don’t want to have to keep track of the size of the variable type and may someday be allocating space for a structure (more later) whose size you really don’t know!

After this memory is allocated, you can use `arr[0]` to `arr[size-1]` just as if it were a normal array.

An important aspect is that when you allocate memory, you should put it away when finished. This is done with

```
free(arr);
```

If you have a program that allocates a lot of memory every time a function is called, and you don’t free the memory when the function ends, you could eventually run out of memory. This is called a memory leak. In truth, this situation is not so bad in moderation, because the memory **will** be freed when the program ends (so you’re not doing any permanent damage), but it’s a good habit to **free** what you **malloc**. Note that you have to call `free` with a pointer to the first byte of memory you allocated. If you have altered the pointer, e.g., with `arr++`, at some point after you called `malloc`, then `arr` may not point to its original value. Of course, you can always define a new pointer `original_arr` and write `original_arr=arr;` before you start modifying `arr`. Then `free(original_arr);` will put away the memory.

Remember that if you allocate memory in a function and assign it to a pointer variable of local scope, then there is a good chance that the pointer will disappear when the function ends, leaving the allocated memory (and any useful contents of it) floating without any way for you to access it. You have to guard against this in your program design.

Memory allocation is a very useful programming feature, but it also can lead to a lot of complication in big programs. More advanced languages like C++ provide mechanisms to partially tame these problems, but this is a case where you will build expertise over time.

1.2.2 Multi-dimensional Arrays

You can define multi-dimensional arrays by using multiple pairs of brackets. For example,

```
double arr[10][20];
```

defines a two-dimensional array. There are 200 floating-point variables, and you can refer to them by `arr[0][0]` to `arr[9][19]`. When you refer to `arr[j][k]`, you actually are accessing element $j \times 20 + k$ of the 200. Note that the syntax is `arr[j][k]`, not `arr[j,k]`.

One needs to be careful when passing such arrays into functions. Because converting the two or more indices into the desired element requires the size of the array, you must specify the sizes in the function definition. In other words, you could write

```
double myfunc(double myarr[10][20], int size1, int size2)
```

although actually the first index can be omitted, e.g.,

```
double myfunc(double myarr[][20], int size1, int size2)
```

If you did

```
double myfunc(double myarr[][19], int size1, int size2)
```

by accident, you will scramble the relation between the indices and the actual locations in memory, because your function will assume that `arr[j][k]` should refer to element $j \times 19 + k$, instead of $j \times 20 + k$. For this reason, if you choose to use multi-dimensional arrays, it is good to have the indices in preprocessor `#define` constants so that if you have to change the size of the array, you can do it in a single place.

The requirement that we put the size of the array in the function header is a serious flaw, because it means that we cannot build re-usable functions that import multi-dimensional arrays and use the same square bracket notation (e.g. `myarr[j][k]`). To use that notation, we would have to edit the function to include the size of the array in the function declaration (and perhaps we have arrays of different sizes in the same program). There are work-arounds, which we will discuss later in the course.

1.2.3 More information

Pointers and arrays are a complicated but important topic in C. Your C reference manual (chapter 5 in Kernighan and Ritchie, for example, will have a discussion of the details. As usual, experimentation is the key to learning, so start trying to use the concepts presented here!

1.3 Input/Output in “C”

The I/O routines listed below in the examples are only a few of almost an infinite number of routines and methods to communicate information to and from a “C” program. Your C reference manual will contain a complete description of these and other routines.

1.3.1 Input from the keyboard

To read input from the keyboard a `scanf` can be used. For example:

```
double k;  
...  
...  
scanf("%lf", &k);
```

Note, a pointer, `&k` is passed as the second argument to `scanf`. The first argument is a format string describing how `scanf` should expect input to be given. In this example, `k` is a double variable and thus the format has an “l”; `%lf`. The format specifiers are the same as those used in `printf` statements.

C was not designed as a text processing language, but it does contain facilities for handling characters and character strings. Again, we refer you to your C reference manual for a complete description. For a quick list of functions, type “`man strings`” in your UNIX shell.

It is possible to read and write character strings from the keyboard. One way is to use `scanf()` or `printf()` with the “`%c`” format for a single character or “`%s`” for a character string. You can also manipulate the characters “by hand”. For example, in the `box_simple.c` routine, a `fgetc` was used to read a single character input from the keyboard. In this case we were using a read from the keyboard to pause the program so the graphics window would not be removed until we pressed return. Specifically:

```
#include <stdio.h>  
...  
int i;  
...  
...  
i=fgetc(stdin);
```

The variable `stdin` is defined in the included file, `stdio.h`. `stdin` is defined to be the keyboard and thus we are telling `fgetc` to read a character from the keyboard. However, we could send `fgetc` a file descriptor (FILE *) and read a character from another stream such as a file or socket. `stdio.h` also defines `stdout` which is the screen. Many of the “C” routines for I/O assume reading from the keyboard and writing to the screen.

1.3.2 Writing data to file

More useful for this weeks homework is to write data directly to a file. Previously you used a pipe, `>`, to redirect data written to the screen into a file. For example:

```
% simpson_integrator > simpson.dat
```

In this way, everything you wrote to the screen is written to the file `simpson.dat` in you present working directory (`pwd`). However, it is useful to be able to put user information on the screen, and still be able to create data file. To do this, a file with a requested name must be opened, data written to it and the file closed. For example:

```
FILE *fp;
...
...
fp=fopen("test.dat", "w");
if (fp == NULL ){
    printf(" Error opening file\n");
}
for (i=0; i<=100; i++) {
    fprintf(fp, "%d %d %lf\n",i, i*i, sqrt((float) i));
}
fclose(fp);
```

The fourth line of this snippet of code opens a file named “test.dat”, the name being a string passed as the first argument. The second argument tells `fopen` what type of I/O will occur, in this case “w”=write. You can also send “r”=read. `fopen` opens the file and gives it a *file descriptor* value, which is returned to the calling program. In this case, the *file descriptor* is held in the variable `fp`. Note, the variable `fp` is declared as a type `FILE`. To write to this file, the *file descriptor*, `fp` must be passed to the routine which does the writing, in this example `fprintf`. Previously you had used `printf`. This routine assumes you want to write to the **stdout**, the screen. To write to somewhere other than the screen, the routine `fprintf` (which stands for **file printf**, should be used, with the *file descriptor* passed as the first argument. Once you are finished writing to the file, a `fclose` should be called, passing to it the *file descriptor*.

One more thing to note about the snippet of code above. A check is done on the variable `fp` to see if the file was opened properly. If there were no errors in `fopen`, the variable `fp` will have a value. If there was an error, `fp` will not have a value, and the comparison to `NULL` will be true. It is good practice to include checks such as this in your routines. In fact, the status of the `fprintf` could also be checked.

A similar bit of code can be used to read data from a file. For example:

```
FILE *fp;
int i, y;
```

```

double value;
...
...
fp=fopen("test.dat", "r");
if (fp == NULL ){
    printf(" Error opening file\n");
}
for (i=0; i<100; i++) {
    fscanf(fp, "%d %d %lf",i, j, value);
}
fclose(fp);

```

This routine reads 300 values from file. The data file could have 300 lines with one datum value per line, or all 300 input values on the same line. `fscanf` ignores carriage returns (`\n`).

A second method to read input from a file, **if** there is only one datum per line **and** the datum is the first item in the line (any comment to the right of the datum in the line of the data file will be ignored).

```

FILE *fp;
int x;
double value;
char string[80];
...
...
fp=fopen("test.dat", "r");
if (fp == NULL ){
    printf(" Error opening file\n");
}
fgets(string, 80, fp);
x = atoi(string);
fgets(string, 80, fp);
value = atof(string);
...
...
fclose(fp);

```

The file “test.dat” might look something like:

```

42      // The answer to the Universe
3.57    // The price of a gallon of gas

```

In this fragment of C code, the `fgets` function reads an entire line from the file pointed to by `fp` and places it in the character string named `string`. The second

argument to `fgets`, 80 tells `fgets` the maximum length in characters that it should read (in this case 80 because this is the length given to *string* when it was declared). The **atoi** function converts the first non-blank characters in the string *string* to an integer (hence the name a-to-i for integer). The result is that the integer variable *x* will have the value 42.

Similarly, the **atof** function converts the first non-blank characters in the string *string* to a double (hence the name a-to-f for float). The result is that the double variable *value* will have the value 3.57.

See your C reference book for more information about the **ato*** functions.

1.4 Structures

We’ve already seen how arrays can simplify data storage in situations where individual variables would be inconvenient or completely unusable. We can declare fixed-length arrays at compile-time to hold lists of related data (e.g. the x and y coordinates of the Earth’s position as it orbits the sun), and we can allocate memory for arrays at run-time (using `malloc` and pointers) to store lists of data with variable length (e.g. the locations of grid boundaries for trapezoid integration).

The primary shortcoming of arrays is that all of the elements in an array must have the same type; there’s no way to declare that some of the elements in an array should have type `double` and others should have type `int`, for example. Also, the elements in an array are stored and accessed only by index; any other relationship among the data has to be created and enforced in the mind of the programmer. Consider the example of the Earth orbiting the Sun again. The components of position are (logically) stored together in an array, and the components of velocity are stored together in a different array, but the position and velocity of the Earth are obviously related and clearly both “belong” to the Earth. How is that “belonging” enforced in C? Not at all, really; we set up separate arrays and decide that the zeroth element of each array will be the x component of the kinematic variable stored in that array, and the first element will be the y component.

Structures solve this data organization problem in C. A structure is a collection of variables—not necessarily of the same type—that are declared and stored together as a unit; individual variables inside a structure are called “members” or “components” of the structure. Members of a structure can be accessed by name (rather than by index, as in arrays), and a variable declared as a structure type can be passed from function to function in a program as a single unit, so that all of the members stay together.

Despite their different approaches to data organization, arrays and structures are not mutually exclusive; in fact, they’re complementary. We can declare and use arrays of structures in programs, and a member of a structure can be an array (or a structure, or an array of structures, or...). This ability to freely connect different variables (scalars and arrays) inside structures and also store structures inside arrays is a very powerful tool for data organization and manipulation.

1.4.1 Defining a structure

Structures are defined using the keyword `struct`. The first element of the declaration is a name for the structure (“`persondata`”, in the example below), and the second is a block (enclosed in curly braces, as usual) that encloses a list of the members of the structure.

```
1 struct persondata {
2     char name[80];
```

```
3     char address[200];
4     int age;
5  };
```

The declaration above declares a structure called “personaldata” with three members; the first member is a string (array of characters) called “name”, the second is a string called “address”, and the third is an integer called “age”. Note that the three members do not have the same data type; one is an integer, and the other two are arrays of different lengths. Also note the final semicolon after the closing brace; structure definitions are C statements, so they must end with a semicolon (gcc won’t usually catch this correctly, but you will get many other errors and warnings if you forget the semicolon, so it’s good to check right away to make sure you’ve included it).

1.4.2 Structure variable declarations

Once a structure has been defined, it can serve as a type for variables. If we have already defined `struct personaldata` as above, we can declare a variable “x” of type `struct personaldata` by simply writing the declaration

```
struct personaldata x;
```

in our declaration section. Structure variables can also be passed to functions; the structure parameter declaration behaves just like the declarations of parameters with built-in C types like `double`. For example, a function “f” declared as

```
double f(int n, struct personaldata x);
```

takes one integer parameter and one parameter of type `struct personaldata`.

1.4.3 Declaring arrays of structures

We can declare arrays of structures in the same way that we declare arrays of simple types. Just as an array of ten doubles is declared as

```
double x[10];
```

we can declare an array of structures (e.g. `struct personaldata`) using

```
struct personaldata x[10];
```

The variable “x” is then an array of ten elements, each of which is a structure defined by the `struct personaldata` declaration above.

1.4.4 Dynamically allocating structures

Just as we can declare a pointer to some basic type and then use `malloc` to allocate memory for an array of that type at run-time, we can declare a pointer to a structure type and then allocate memory for an array of structures at run-time. Consider the following pointer declaration and `malloc` statement.

```
1  double *ptr;
2  ptr = malloc(100*sizeof(double));
```

The first statement declares “`ptr`” as a pointer-to-double, and the `malloc` statement allocates 100 blocks of memory, each big enough to hold a double variable, and assigns the address of the block to “`ptr`”.

Similarly, we can declare a pointer to a structure type and then allocate memory for an array of structures. Each block of memory must be large enough to hold one instance of the structure.

```
1  struct personaldata *ptr;
2  ptr = malloc(100*sizeof(struct personaldata));
```

The `malloc` statement again allocates 100 blocks of memory; here the size of each block is the size of one variable of type `struct personaldata`; the size of the structure is determined by the call to the `sizeof` function.

1.4.5 Accessing members of a structure

Just as the array subscript operator “[]” (square brackets) accesses array elements by index, the structure member operator “.” (period) accesses members of a structure by name.

If an array of doubles called “`x`” has been declared as

```
double x[10];
```

the variables “`x[0]`”, “`x[1]`”, etc., are each `double` variables. We can assign to elements of the array, perform mathematical or C operations on them, pass them to functions, and generally treat them as normal variables of type `double`.

Similarly, we can access members of structures using the “.” operator. If a variable called “`x`” of type `struct personaldata` has been declared, we can access its members as shown in the following example.

```
1  struct personaldata x;
2  x.name = ‘‘John Doe’’;
3  x.address = ‘‘1234 Elm St.’’;
4  x.age = 100;
```

The “name” and “address” members of “x” are both declared to be strings (arrays of characters) in the declaration of `struct personaldata`, so we can assign strings to them. The member “age” is declared to be an integer, so we can assign an integer to it.

We’ve already seen how to modify the value of a variable from inside a function by passing a pointer to the variable as an argument to the function. We can modify the values stored in the components of a structure in the same way—we can pass a pointer to a structure as an argument and operate on that pointer inside the function.

There are actually two different ways to access a component of a structure using a pointer to that structure. If “x” is a pointer to a `struct personaldata` structure, then one way to access, for example, the “age” member would be to first dereference the pointer (using `*`) and then access “age” using the `.` operator, as shown below:

```
(*x).age = 110;
```

Note the parentheses around `*x`. The `.` operator binds more tightly (has higher precedence) than the `*` operator, so without parentheses the statement above would be equivalent to

```
*(x.age) = 110;
```

which would generate an error, because “x”, a pointer, is not a structure itself and consequently has no member named “age”.

Because the `(*x).age` syntax is a little cumbersome, C also defines an operator that performs both the pointer dereferencing and the structure member access in one step. This operator is the `->` operator. Again, if “x” is a pointer to a structure, the following statement accesses the “age” member of that structure:

```
x->age = 110;
```

Either syntax can be used, but the `->` operator syntax is generally easier to read and less error-prone (since there’s no confusion about operator precedence).