6 ※ More coding tools

If you have a good handle on Chapter 2, then you already have what you need to write some very advanced programs. But C is a world unto itself, with hundreds of utilities to facilitate better coding and many features for the programmer who wishes to delve further.

This chapter covers some additional programming topics, and some details of C and its environment. As with earlier chapters, the syntax here is C-specific, but it is the norm for programming languages to have the sort of features and structures discussed here, so much of this chapter will be useful regardless of language.

The statistician reader can likely get by with just a skim over this chapter (with a focus on Section 6.1), but readers working on simulations or agent-based models will almost certainly need to use the structures and techniques described here.

The chapter roughly divides into three parts. After Section 6.1 covers functions that operate on other functions, Section 6.2 will use such functions to build structures that can hold millions of items, as one would find in an agent-based model. Section 6.3 shows the many manners in which your programs can take in parameters from the outside world, including parameter files, enivornment variables, and the command line. Sections 6.4 and 6.5 cover additional resources that make life in front of a computer easier, including both syntactic tricks in C and additional programs useful to programmers.

6.1 FUNCTION POINTERS

A data point `d` is stored somewhere in memory, so we can refer to its address, `&d`. Similarly, a function `f` is stored somewhere in memory, so we can refer to its address as well.

What is a pointer to a function good for? It lets us write Functions that will accept any function pointer and then use the pointed-to function. [Functions calling functions is already confusing enough, so I will capitalize Function to indicate a parent function that takes a lower-case function as an input.] For example, a Function could search for the largest value of an input function over a given range, or a bootstrap Function could take in a statistic-calculating function and a data set and then return the variance of the statistic.

TYPES

Before we can start writing Functions to act on functions, we need to take the type of input function into consideration. If a function expects `int`s, then the compiler needs to know this, so it can block attempts to send the function a string or array.

The syntax for declaring a function pointer is based on the syntax for declaring a function. Say that we want to write a Function that will take in an array of `double`s plus a function, and will apply the function to every element of the array, returning an array of `int`s. Then the input function has to have a form like

```
int double_to_int (double x);                                        (A)
```

Recall that a pointer declaration is just like a declaration for the pointed-to type but with another star, like `int *i`; the same goes for declaring function pointers, but there are also extra parens. Here is a type for a function pointer that takes in a `double` and returns a pointer to `int`; you can see it is identical to line A, but for the addition of a star and parens:

```
int (*double_to_int) (double x)                                      (B)
```

By the way, if the function returned an `int*` instead of a plain `int`, the declaration would be:

```
int *(*double_to_int) (double *x)
```

The type declarations do nothing by themselves, just as the word `int` does nothing by itself. But now that you know how to define a function type, you can put the declaration of the function into your header line. A Function that applies a function to an array of `double`s would have a header like this:

```
int* apply (double *v, int (*instance_of_function) (double x));      (C)
```

Putting `typedef` *to work* Are you confused yet? Each component basically makes sense, but together it is cluttered and confusing. There is a way out: `typedef`. By putting that word before line B—

```
typedef int (*double_to_int) (double x);
```

—we have created an new type named `double_to_int` that we can use like any other type. Now, line C simplifies to

```
int* apply (double *v, double_to_int instance_of_function);
```

```
#include <apop.h>

typedef double (*dfn) (double);

double sample_function (double in){
    return log(in)+ sin(in);
}

void plot_a_fn(double min, double max, dfn plotme){
  double val;
  FILE *f = popen("gnuplot -persist", "w");
    if (!f)
        printf("Couldn't find Gnuplot.\n");
    fprintf(f, "set key off\n plot '-' with lines\n");
    for (double i=min; i<max; i+= (max-min)/100.0){
        val = plotme(i);
        fprintf(f, "%g\t%g\n", i, val);
    }
    fprintf(f, "e\n");
}

int main(){
    plot_a_fn(0, 15, sample_function);
}
```

Listing 6.1 A demonstration of a Function that takes in any function $\mathbb{R} \to \mathbb{R}$ and plots it. Online source: `plotafunction.c`.

Listing 6.1 shows a program to plot any function of the form $\mathbb{R} \to \mathbb{R}$, using the Gnuplot program described in Chapter 5. With a `typedef` in place, the syntax is easy. You don't need extra stars or ampersands in either the declaration of the Function-of-a-function or in the call to that Function, and you can call the pointed-to function like any other.

- Line 3: To make life easier, the `dfn` type is declared at the top of the file.
- Line 5: The header for the `sample_function` matches the format of the `dfn` function type (i.e., `double` in, `double` out).
- Line 9: The `plot_a_fn` Function specifies that it takes in a function of type `dfn`.
- Line 16: Using the passed-in function is as simple as using any other function: this line gives no indication that `plotme` is in any way special.
- Line 23: Finally, in `main`, you can see how `plot_a_fn` is called. The `sample_-function` is passed in with just its name.

For another example, have a look at `jackiteration.c` on page 132.

$\mathbb{Q}_{6.1}$

Turn `plotafunction.c` into a library function callable by other programs.

- Comment out the `main` function.
- Write a header `plotafunction.h` with the necesary type and function definitions.
- Write a test program that `#includes plotafunction.h` and plots the `calc_taxes` function from `taxes.c` (p 118).
- Modify the makefile to produce the final program by creating and linking both *`your_code.o`* and `plotafunction.o`.

$\mathbb{Q}_{6.2}$

Define a type `dfn` as in line three of Listing 6.1. Then write a Function with header `void apply(dfn fn, double *array, int array_len)` that takes as arguments a function, an array, and the length of the array, and changes each element `array[i]` to `fn(array[i])`. [Apophenia provides comparable functions; see page 117 for details.]
Test your Function by creating an array of the natural numbers $1, 2, 3, \ldots 20$ and transforming it to a list of squares.

$\sum$

➤ You can pass functions as function arguments, just as you would pass arrays or numbers.

➤ The syntax for declaring a function pointer is just like the syntax for declaring a function, but the name is in parens and is preceded by a star. ⋙

⋙

∑

➤ Defining a new type to describe the function helps immensely. This requires putting `typedef` in front of the function pointer declaration in the last summary point.

➤ Once you have a `typedef` in place, you can declare Functions that take functions, use the passed-in functions, and call the parent Function as you would expect. Given the `typedef`, you need neither stars nor ampersands for these operations.

6.2 DATA STRUCTURES

Say that you have a few million observations to store on your computer. You want to find any given item quickly, add or delete elements easily, and not worry too much about a complicated organization system. There are several options for balancing these goals, and choosing among them is not trivial. This section will consider three: the array, the linked list, and the binary tree.

They will be implemented here via Glib, a library of general-use functions that every C programmer seems to re-implement. It includes a few features for string handling and other such conveniences, and modules to handle the data structures described here.[1] The extended example below provides documentation-by-example of initializing, adding to, removing from, and finding elements within the various structures, but your package manager will be happy to install the complete documentation, as well as Glib itself.

AN EXAMPLE This game consists of a series of meetings between pairs of birds, who compete over r *utils* of resource.[2] If two doves meet, they split the resource evenly between them. If a dove and a hawk meet, the dove backs down and the hawk gets the resource. If two hawks meet, then the hawks fight, destroying c utils in resources before finally splitting what is left. Table 6.2 shows a payoff table summarizing the outcomes. For each pairing, the row player gets the first payoff, and the column player gets the second.

[1]Glib also provides a common data structure known as a *hash table*, which is another technique for easy data retrieval. It converts a piece of data, such as a string, into a number that can then be used to jump to the string's data in a table very quickly. Binary trees tend to work better in the context of agent-based modeling, so I have omitted hashes from this chapter. See Kernighan & Pike (1999), Chapter 3, for an extended example of hash tables that produce nonsense text. It was intended as an amusement (compare with the Exquisite Corpse-type game played by Pierce (1980, p 262)), but is now commonly used to produce spam email. ℚ: Try implementing Kernighan and Pike's nonsense generator using Glib's hash tables, string hash functions, and list structures.

[2]The util is the unit of measurement for the quantity of utility an agent gets from an action.

$$\begin{array}{c|cc|}
 & \text{dove} & \text{hawk} \\
\text{dove} & (\frac{r}{2}, \frac{r}{2}) & (0, r) \\
\text{hawk} & (r, 0) & (\frac{r-c}{2}, \frac{r-c}{2})
\end{array}$$

Table 6.2 The payoff matrix for the hawk/dove game. If $c < r$, then this is a prisoner's dilemma.

With $c < r$, the game is commonly known as a *prisoner's dilemma*, due to a rather contrived story about two separated prisoners who must choose between providing evidence about the other prisoner and remaining silent. Its key feature is that being a dove (cooperating) always makes the agent worse off than being a hawk (not cooperating, which the literature calls defection). The only equilibrium to the P.D. game is when nobody cooperates, destroying resources every period, but the societal optimum is when everyone cooperates, producing r utils of utility total every time.

On top of this we can add an evolutionary twist: say that a bird that is very successful will spawn chicks. In any one interaction, a bird gets an equal or better payoff as a hawk than as a dove, so it seems that over time, the hawks would approach 100% of the population. In the simulation below, a bird's odds of reproducing are proportional to the percentage of total flock wealth the bird holds, and its odds of dying are inversely proportional to the same.

To simulate the game, we will need a flock of birds. Have a look at the header file `birds/birds.h` in the online code supplement. It begins by describing the basic structure that the rest of the functions depend upon, describing a single bird:

```
typedef struct {
    char type;
    int wealth;
    int id;
} bird;
```

The header then lists two types of function. The first are functions for each relevant action in the simulation: startup, births, deaths, and actual plays of the hawk/dove game. The second group are functions for flock management, such as counting the flock or iterating over every member of the flock.

The first set of functions are implemented in Listing 6.3. Each function, taken individually, should make sense: `play_hd_game` takes in two birds and modifies their payoff according to the game rules above; `bird_plays` takes in a single bird, finds an opponent, and then has them play against each other; et cetera.

```
#include "birds.h"
#include <time.h>

gsl_rng *r;
int periods = 400;
int initial_pop = 1000;
int id_count = 0;

void play_hd_game(bird *row, bird *col){
  double resource = 2,
            cost = 2.01;
  if (row−>type == 'd' && col−>type == 'h')
        col−>wealth += resource;
  else if (row−>type == 'h' && col−>type == 'd')
        row−>wealth += resource;
  else if (row−>type == 'd' && col−>type == 'd'){
        col−>wealth += resource/2;
        row−>wealth += resource/2;
  } else { // hawk v hawk
        col−>wealth += (resource−cost)/2;
        row−>wealth += (resource−cost)/2;
  } }

void bird_plays(void *in, void *dummy_param){
  bird *other;
    while(!(other = find_opponent(gsl_rng_uniform_int(r,id_count))) && (in != other))
        ;//do nothing.
    play_hd_game(in, other); }

bird *new_chick(bird *parent){
  bird *out = malloc(sizeof(bird));
    if (parent)
        out−>type = parent−>type;
    else{
        if (gsl_rng_uniform(r) > 0.5)
            out−>type = 'd';
        else
            out−>type = 'h';
    }
    out−>wealth = 5* gsl_rng_uniform(r);
    out−>id = id_count;
    id_count ++;
    return out; }

void birth_or_death(void *in, void *t){
  bird *b = in; //cast void to bird;
  int *total_wealth = t;
    if (b−>wealth*20./ *total_wealth >= gsl_rng_uniform(r))
        add_to_flock(new_chick(b));
    if (b−>wealth*800./ *total_wealth <= gsl_rng_uniform(r))
        free_bird(b); }

void startup(int initial_flock_size){
    flock_init();
    r = apop_rng_alloc(time(NULL));
    printf("Period\tHawks\tDoves\n");
    for(int i=0; i< initial_flock_size; i++)
        add_to_flock(new_chick(NULL)); }
```

```
int main(){
    startup(initial_pop);
    for (int i=0; i< periods; i++){
        flock_plays();
        count(i);
    } }
```

Listing 6.3 The birds. Online source: `birds/birds.c`

Now for the flock management routines, which will be implemented three times: as an array, as a list, and as a binary tree.

ARRAYS An array is as simple as data representation can get: just write each element right after the other. The matrices and vectors throughout this book keep their data in arrays of this type.

The system can retrieve an item from an array faster than from any other data structure, since the process consists of simply going to a fixed location and reading the data there. On the other hand, adding and deleting elements from an array is difficult: the simulation has to call `realloc` every time the list expands. If you are lucky, `realloc` will not move the array from its current location, but will simply find that there is more free space for the array to grow. If you are not lucky, then the array will have to be moved in its entirety to a new, more spacious home.

An array can not have a hole in the middle, so elements can not be deleted by freeing the memory. There are a few solutions, none of which are very pleasant. The last element of the list could be moved in to the space, requiring a copy, a shrinking of the array, and a loss of order in the elements. If order is important, every element could be shifted down a notch, so if item 50 is deleted, item 51 is put in slot 50, item 52 is put in slot 51, et cetera. Thus, every death could mean a call to `memmove` to execute thousands or millions of copy operations.

Listing 6.4 marks dead birds by setting their `id` to `-1`. This means that as the program runs, more and more memory is used by dead elements, and the rest of the system must check for the marker at every use.

As for finding an element, `add_to_flock` takes pains to ensure that the `id` and array index will always match one-to-one, so finding a bird given its id number is trivial. As with the code above, the code consists of a large number of short functions, meaning that it is reasonably easy to understand, read, and write each function by itself.

```
#include "birds.h"

bird *flock;
int size_of_flock, hawks, doves;

void flock_plays(){
    for (int i=0; i< size_of_flock; i++)
        if (flock[i].id >= 0)
            bird_plays(&(flock[i]), NULL); }

void add_to_flock(bird* b){
    size_of_flock = b->id;
    flock = realloc(flock, sizeof(bird)*(size_of_flock+1));
    memcpy(&(flock[b->id]), b, sizeof(bird));
    free(b); }

void free_bird(bird* b){ b->id = -1; }

bird * find_opponent(int n){
    if (flock[n].id >= 0)
        return &(flock[n]);
    else return NULL; }

int flock_size(){ return size_of_flock; }

int flock_wealth(){
  int i, total =0;
    for (i=0; i< size_of_flock; i++)
        if (flock[i].id >= 0)
            total += flock[i].wealth;
    return total; }

double count(int period){
  int i, tw = flock_wealth();
    hawks = doves = 0;
    for(i=0; i< size_of_flock; i++)
        if (flock[i].id>=0)
            birth_or_death(&(flock[i]), &tw);
    for(i=0; i< size_of_flock; i++)
        if (flock[i].id>=0){
            if (flock[i].type == 'h')
                hawks ++;
            else doves ++;
        }
    printf("%i\t%i\t%i\n", period, hawks, doves);
    return (doves+0.0)/hawks;}

void flock_init(){
  flock = NULL;
  size_of_flock = 0; }
```

Listing 6.4 The birds, array version. The fatal flaw is that birds are copied in, but never eliminated. Dead birds will eventually pile up. Online source: `birds/arrayflock.c`

Q6.3 Because the play_hd_game function sets $r == 2$ and $c == 2.01$, hawks lose 0.005 utils when they fight, so it is marginally better to be a dove when meeting a hawk, and the game is not quite a prisoner's dilemma. After running the simulation a few times to get a feel for the equilibrium number of birds, change c to 2.0 and see how the equilibrium proportion of doves changes. [For your convenience, a sample makefile is included in the birds directory of the code supplement.]
Finally, rename main to one_run and wrap it in a function that takes in a value of c and returns the proportion of doves at the end of the simulation. Send the function to the plot_a_function function from earlier in the chapter to produce a plot.

Q6.4 Rewrite arrayflock.c to delete birds instead of just mark them as dead. Use memmove to close holes in the array, then renumber birds so their id matches the array index. Keep a counter of current allocated size (which may be greater than the number of birds) so you can realloc the array only when necessary.

LINKED LISTS A *linked list* is a set of structs connected by pointers. The first struct includes a next pointer that points to the next element, whose next pointer points to the next element, et cetera; see Figure 6.5.

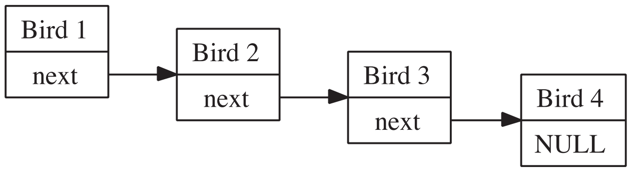

Figure 6.5 The archetypal linked list. Online source: list.dot.

The linked list is a favorite for agent-based simulation, because birth and death is easy to handle. To add an element to a linked list, just create a new node and replace the NULL pointer at the end of the list with a pointer to the new node. Deleting a node is also simple: to delete bird 2, simply reroute the next pointer from bird $1 \to 2$ so that it points from bird $1 \to 3$, and then free the memory holding bird 2.

But the real failing of the linked list is the trouble of finding an arbitrary element. In an array, finding the ten thousandth element is easy: flock[9999]. You can see in the code of Listing 6.6 that glib provides a g_list_nth_data function to return the $n\text{th}$ element of the list, which makes it look simple, but the only way that that function can find the ten thousandth member of the flock is to start at the head of the list and take 9,999 ->next steps. In fact, if you compile and run this program, you will see that it runs much more slowly than the array and tree versions.

```
#include "birds.h"
#include <glib.h>

GList *flock;
int hawks, doves;

void flock_plays(){ g_list_foreach(flock, bird_plays, NULL); }

void add_to_flock(bird* b){ flock = g_list_prepend(flock, b); }

void free_bird(bird* b){
    flock = g_list_remove(flock, b);
    free(b); }

bird * find_opponent(int n){ return g_list_nth_data(flock, n); }

void wealth_foreach(void *in, void *total){
    *((int*)total) += ((bird*)in)->wealth; }

int flock_wealth(){
  int total = 0;
    g_list_foreach(flock, wealth_foreach, &total);
    return total; }

int flock_size(){ return g_list_length(flock); }

void bird_count(void *in, void *v){
  bird *b = in;
    if (b->type == 'h')
        hawks ++;
    else doves ++;
}

double count(int period){
  int total_wealth =flock_wealth();
    hawks = doves = 0;
    g_list_foreach(flock, birth_or_death, &total_wealth);
    g_list_foreach(flock, bird_count, NULL);
    printf("%i\t%i\t%i\n", period, hawks, doves);
    return (doves+0.0)/hawks;}

void flock_init(){ flock = NULL; }
```

Listing 6.6 The birds, linked list version. The fatal flaw is that finding a given bird requires traversing the entire list every time. Online source: `birds/listflock.c`

- The `g_list_foreach` function implements exactly the sort of apply-function-to-list setup implemented in Section 6.1. It takes in a list and a function, and internally applies the function to each element.
- The folks who wrote the Glib library could not have known anything about the `bird` structure, so how could they write a linked list that would hold it? The solution is `void` pointers—that is, a pointer with no type associated, which could therefore point to a location holding data of any type whatsoever. For example,

bird_count takes in two void pointers, the first being the element held in the list, and the second being any sort of user-specified data (which in this case is just ignored).

- The first step in using a void pointer is casting it to the correct type. For example, the first line in bird_count—bird *b = in;—points b to the same address as in, but since b has a type associated, it can be used as normal.
- As for adding and removing, the Glib implementation of the list takes in a pointer to a GList and a pointer to the data to be added, and returns a new pointer to a GList. The input and output pointers could be identical, but since this is not guaranteed, use the form here to reassign the list to a new value for every add/delete. For example, the flock starts in flock_init as NULL, and is given its first non-NULL value on the first call to add_to_flock.

Binary trees The *binary tree* takes the linked list a step further by giving each node two outgoing pointers instead of one. As per Figure 6.7, think of these pointers as the left pointer and the right pointer. The branching allows for a tree structure. The directions to an element are now less than trivial—to get to bird5, start at the head (bird1), then go left, then go right. But with eight data points in a linked list, you would need up to seven steps to get to any element, and on average 3.5 steps. In a tree, the longest walk is three steps, and the average is 1.625 steps. Generally, the linked list will require on the order of n steps to find an item, and a b-tree will require on the order of $\ln(n)$ steps (Knuth, 1997, pp 400–401).[3]

The tree arrangement needs some sort of order to the elements, so the system knows whether to go left or right at each step. In this case, the id for each bird provides a natural means of ordering. For text data, strcmp would provide a similar ordering. More generally, there must be a *key value* given to each element, and the tree structure must have a function for comparing keys.

Eariler, you saw an interesting implementation of a set of binary trees: a database. Since databases require fast access to every element, it is natural that they would internally structure data in a binary tree, and this is exactly how SQLite and mySQL operate internally: each new index is its own tree.

This adds some complication, because you now need to associate with each tree a function for comparing keys. In the code below, g_tree_new initializes a tree using the compare_birds function.

[3]For those who follow the reference, notice that Knuth presents the equation for the sum of path lengths, which he calls the internal path length. He finds that it is of order $n\ln(n) + \mathcal{O}(n)$ for complete binary trees; the average path length is thus $\ln(n) + \mathcal{O}(1)$.

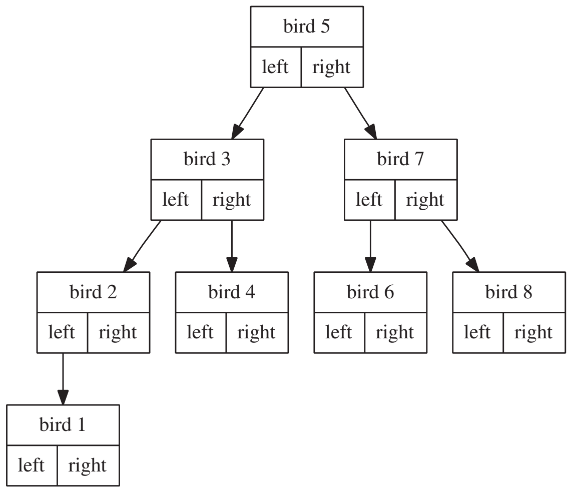

Figure 6.7 The archetypal binary tree. Online source: `btree.dot`.

What if two birds have the same `id`? Then there is no way to order them uniquely, and therefore there is no way to reliably store and retrieve them. Thus, the key for each element must be unique.[4]

The added complication of a tree solves many of the problems above. As with the list, inserting and deleting elements does not require major `realloc`ing, although there is often minor internal reshuffling to keep the branches of the tree at about even length. With the key and short chains, finding an element is much faster.

The `const` keyword

The `const` modifiers in the header for `compare_keys` indicate that the data to which the pointers point will not be changed over the course of the function. As you can see by the fact that it has not appeared until page 201, the `const` keyword is mostly optional, though it is good form and provides one more check that your functions don't do what you hadn't intended. However, when conforming with function specifications elsewhere, like GLib's function header for key-comparison functions, you may need to use it. If you then get an error like *'subfunction' discards qualifiers from pointer target type*, then you will need to rewrite the subfunction so that it too takes `const` inputs (and does not modify them).

```
#include "birds.h"
#include <glib.h>

GTree *flock = NULL;
int hawks, doves;

static gint compare_keys(const void *L, const void *R){
  const int *Lb = L;
  const int *Rb = R;
    return *Lb − *Rb;
}
```

[4]There exist tree implementations that do not require unique keys, but it is a requirement for GLib. Similarly, some databases are very strict about requiring that each table have a field representing a key, and some are not.

```
static gboolean tree_bird_plays(void *key, void *in, void *v){
    bird_plays(in, NULL);
    return 0;
}

void flock_plays(){ g_tree_foreach(flock, tree_bird_plays, NULL); }

void add_to_flock(bird* b){ g_tree_insert(flock, &(b->id), b); }

bird * find_opponent(int n){return g_tree_lookup(flock, &n);}

int flock_size(){ return g_tree_nnodes(flock); }

static gboolean wealth_foreach(void *key, void *in, void *t){
  int *total = t;
    *total += ((bird*)in)->wealth;
    return 0; }

int flock_wealth(){
  int total = 0;
    g_tree_foreach(flock, wealth_foreach, &total);
    return total; }

static gboolean tree_bird_count(void *key, void *in, void *v){
    if (((bird *)in)->type == 'h')
        hawks ++;
    else doves ++;
    return 0; }

GList *dying_birds;

void free_bird(bird* b){dying_birds = g_list_prepend(dying_birds, b);}

static gboolean tree_birth_or_death(void *key, void *in, void *t){
    birth_or_death(in, t);
    return 0; }

static void cull_foreach(void *b, void *v){
  bird* a_bird = b;
    g_tree_remove(flock, &(a_bird->id));
    free(a_bird); }

double count(int period){
  int total_wealth =flock_wealth();
    hawks = doves = 0;
    dying_birds = NULL;
    g_tree_foreach(flock, tree_birth_or_death, &total_wealth);
    g_list_foreach(dying_birds, cull_foreach, NULL);
    g_list_free(dying_birds);
    g_tree_foreach(flock, tree_bird_count, NULL);
    printf("%i\t%i\t%i\n", period, hawks, doves);
    return (doves+0.0)/hawks;}

void flock_init(){ flock = g_tree_new(compare_keys); }
```

Listing 6.8 The birds, binary tree version. The fatal flaw is the complication in maintaining the key for every bird. Online source: `birds/treeflock.c`

- Culling the flock is especially difficult because a tree can internally re-sort when an element is added/deleted, so it is impossible to delete elements while traversing a tree. In the implementation of Listing 6.8, the `free_bird` function actually freed the bird; here it just adds dying birds to a `GList`, and then another post-traversal step goest through the `GList` and cull marked birds from the tree.

∑

➤ There are various means of organizing large data sets, such as collections of agents in an agent-based model.

➤ Arrays are simply sequential blocks of structs. Pros: easy to implement; you can get to the $10,000\mathrm{th}$ element in one step. Cons: no easy way to add, delete, or reorganize elements.

➤ A linked list is a sequence of structs, where each includes a pointer to the next element in the list. Pro: adding/deleting/resorting elements is trivial. Con: Getting to the $10,000\mathrm{th}$ element takes 9,999 steps.

➤ A binary tree is like a linked list, but each struct has a left and right successor. Pros: adding and deleting is only marginally more difficult than with a linked list; getting to the $10,000\mathrm{th}$ element takes at most 13 steps. Con: Each element must be accessed via a unique key, adding complication.

6.3 PARAMETERS

Your simulations and analyses will require tweaking. You will want to try more agents, or you may want your program to load a data set from a text file to a database for one run and then use the data in the database for later runs.

This section will cover a cavalcade of means of setting parameters and specifications, in increasing order of ease of use and difficulty in implementation.

The first option—a default of sorts—is to set variables at the top of your `.c` file or a header file. This is trivial to implement, but you will need to recompile every time you change parameters.

Interactive The second option is to interactively get parameters from the user, via `scanf` and `fgets`. Listing 6.9 shows a program that asks data of the user and then returns manipulated data. The `scanf` function basically works like `printf` in reverse, reading text with the given format into pointers to variables. Unfortunately, the system tends to be rather fragile in the real world, as a stray comma or period can entirely throw off the format string. The `fgets` function will read an entire line into a string, but has its own quirks. In short, the interactive input features are good for some quick interrogations or a bit of fun, but are not to be heavily relied upon.

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

int main(){
  float indata;
  char s[100];
    printf("Give me a number: ");
    scanf("%g", &indata);
    printf("Your number squared: %g\n", indata*indata);
    printf("OK, now give me a string (max length, 100):\n");
    fgets(s, 99, stdin); //eat a newline.
    fgets(s, 99, stdin);
    printf("Here it is backward:\n");
    for (int i=strlen(s)−2; i>=0; i−−)
        printf("%c", s[i]);
    printf("\n");
}
```

Listing 6.9 Reading inputs from the command line. Online source: `getstring.c`.

Environment variables These are variables passed from the shell (aka the command prompt) to the program. They are relatively easy to set, but are generally used for variables that are infrequently changing, like the username. Environment variables are discussed at length in Appendix A.

Parameter files There are many libraries that read parameter files; consistent with the rest of this chapter, Listing 6.10 shows a file in Glib's *key file* format, which will be read by the program in Listing 6.11. The configuration file can be in a human language like English, you can modify it as much as you want without recompiling the code itself, it provides a permanent record of parameters for each run, and you can quickly switch among sets of variables.

- The payoff for Listing 6.11 is on line 22: printing the name of a distribution, a parameter, and the mean of that distribution given that parameter. The program to that point finds these three items.
- Line seven indicates which section of Listing 6.10 the following code will read. By commenting out line seven and uncommenting line eight, the code would read the Exponential section. Below, you will see that setting the `config` variable on the command line is not difficult.
- Line 10 reads the entire `glib.config` file into the `keys` structure. If something

```
#gkeys.c reads this file

[chi squared configuration]
distribution name = Chi squared
parameter = 3

[exponential configuration]
distribution name = Exponential
parameter = 2.2
```

Listing 6.10 A configuration in the style of Glib's key files. Online source: `glib.config`.

```
 1  #include <glib.h>
 2  #include <apop.h>
 3
 4  int main(){
 5     GKeyFile *keys = g_key_file_new();
 6     GError *e = NULL;
 7     char *config = "chi squared configuration";
 8  // char *config = "exponential configuration";
 9     double (*distribution)(double, double);
10        if (!g_key_file_load_from_file(keys, "glib.config", 0, &e))
11              fprintf(stderr, e->message);
12        double param = g_key_file_get_double(keys, config, "parameter", &e);
13        if (e) fprintf(stderr, e->message);
14        char* name = g_key_file_get_string(keys, config, "distribution name", &e);
15        if (e) fprintf(stderr, e->message);
16
17        if (!strcmp(name, "Chi squared"))
18              distribution = gsl_cdf_chisq_Pinv;
19        else if (!strcmp(name, "Exponential"))
20              distribution = gsl_cdf_exponential_Pinv;
21
22        printf("Mean of a %s distribution with parameter %g: %g\n", name,
23                    param, distribution(0.5, param));
24  }
```

Listing 6.11 A program that reads Listing 6.10. Online source: `gkeys.c`.

goes wrong, then line 11 prints the error message stored in `e`. Properly, the program should exit at this point; for the sake of brevity the `return 0` lines have been omitted.

- Now that `keys` holds all the values in the config file, lines 12 and 14 can get individual values. The two `g_key_file_get...` functions take in a filled key structure, a section name, a variable name, and a place to put errors. They return the requested value (or an error).

- Unfortunately, there is no way to specify functions in a text file but by name, so lines 17–20 set the function pointer `distribution` according to the name from the config file.

Q6.5 Rewrite the code in Listing 6.11 to set parameters via database, rather than via the command line.

- Write a text file with three columns: configuration, parameters, and data.
- Read in the file using `apop_text_to_db` at the beginning of `main`.
- Write a function with header `double get_param(char *config, char *p)` that queries the database for the parameter named `p` in the configuration group `config` and returns its value. Then modify the program to get the distribution and its parameter using the `get_param` function.

Command line Reading parameters from the command line can take the most effort to implement among the parameter-setting options here, but it is the most dynamic, allowing you to change parameters every time you run the program. You can even write batch files in Perl, Python, or a shell-type language to run the program with different parameter variants (and thus keep a record of those variants).

The `main` function takes inputs and produces an output like any other. The output is an integer `return`ed at the end of `main`, which is typically zero for success or a positive integer indicating a type of failure. The inputs are always an integer, giving the number of command-line elements, and a `char**`—an array of strings—listing the command-line elements themselves. Like any function specification, the types are non-negotiable but the internal name you choose to give these arguments is arbitrary. However, the universal custom is to name them `argc` (argument count) and `argv` (argument values).[5] This is an ingrained custom, and you can expect to see those names everywhere.[6]

Listing 6.12 shows the rudimentary use of `argc` and `argv`. Here is a sample usage from my command line:

[5]They are in alphabetical order in the parameters to `main`, which provides an easy way to remember that the count comes first.

[6]Perl uses `argv`, Python uses `sys.argv`, and Ruby uses `ARGV`. All three structures automatically track array lengths, so none of these languages uses an `argc` variable.

```
#include <stdio.h>

int main(int argc, char **argv){
    for (int i=0; i< argc; i++)
        printf("Command line argument %i: %s\n", i, argv[i]);
}
```

Listing 6.12 This program will simply print out the command line arguments given to it. Online source: `argv.c`.

```
>>> /home/klemens/argv one 2 --three fo\ ur "cinco -- \"five\""
command line argument 0: /home/klemens/argv
command line argument 1: one
command line argument 2: 2
command line argument 3: --three
command line argument 4: fo ur
command line argument 5: cinco -- "five"
```

- Argument zero (`argv[0]`) is always the name of the command itself. Some creative programs run differently if they are referred to by different names, but the norm is to just skip over `argv[0]`.
- After that, the elements of `argv` are the command line broken at the spaces, and could be dashes, numbers, or any other sort of text.
- As you can see from the parsing of `fo\ ur`, a space preceded by a backslash is taken to be a character like any other, rather than an argument separator.
- Argument five shows that everything between a pair of quotation marks is a single argument, and a backslash once again turns the quotation mark into plain text.

For some purposes, this is all you will need to set program options from your command line. For example you could have one program to run three different actions with a `main` like the following:

```
int main(int argc, int **argv){
    if (argc == 1){
        printf("I need a command line argument.\n")
        return 1;
    }
    if (!strcmp(argv[1], "read"))
        read_data();
    else if (!strcmp(argv[1], "analysis_1"))
        run_analysis_1();
    else if (!strcmp(argv[1], "analysis_2"))
        run_analysis_2();
}
```

Q6.6 The `callbyval.c` program (Listing 2.5, page 38) calculated the factorial of a number which was hard-coded into `main`. Rewrite it to take the number from the command line, so `factorial 15` will find 15!. (*Hint*: the `atoi` function converts a text string to the integer it represents; for example, `atoi("15") == 15`.)

getopt For more complex situations, use `getopt`, which parses command lines for *switches* of the form `-x...`. It is part of the standard C library, so it is fully portable.[7]

Listing 6.13 shows a program that will display a series of exponents. As explained by the message on lines 9–14, you can set the minimum of the series via `-m`, the maximum via `-M`, and the increment via `-i`. Specify the base of the exponents after the switches. Sample usage (which also demonstrates that spaces between the switch and the data are optional):

```
>>> ./getopt −m 3 −M4 −i 0.3 2
2^3: 8
2^3.3: 9.84916
2^3.6: 12.1257
2^3.9: 14.9285
```

There are three steps to the process:

- `#include <unistd.h>`.
- Specify a set of letters indicating valid single-letter switches in a crunched-together string like line 15 of Listing 6.13. If the switch takes in additional info (here, every switch but `-h`), indicate this with a colon after the letter.
- Write a `while` loop to call `getopt` (line 27), and then act based upon the value of the `char` that `getopt` returned.

- `argv` is text, but you will often want to specify numbers. The functions `atoi`, `atol`, and `atof` convert ASCII text to an `int`, `long int`, or `double`, respectively.[8]

[7]The GNU version of the standard C library provides a sublibrary named `argp` that provides many more functions and does more automatically, but is correspondingly more complex and less portable. Glib also has a subsection devoted to command-line arguments, which also provides many more features so long as the library is installed.

[8]Getopt readily handles negative numbers that are arguments to a flag (`-m -3`), but a negative number after the options will look like just another flag, e.g., `./getopt -m -3 -M 4 -2` looks as if there is a flag named `2`. The special flag `--` indicates that `getopt` should stop parsing flags, so `./getopt -m -3 -M 4 -- -2` will work. This is also useful elsewhere, such as handling files that begin with a dash; e.g., given a file named `-a_mistake`, you can delete it with `rm -- -a_mistake`.

```
 1  #include <stdio.h> //printf
 2  #include <unistd.h> //getopt
 3  #include <stdlib.h> //atof
 4  #include <math.h> //powf
 5
 6  double min = 0., max = 10.;
 7  double incr = 1., base = 2.;
 8
 9  void show_powers(){
10      for (double i=min; i<=max; i+= incr)
11          printf("%g^%g: %g\n", base, i, powf(base, i));
12  }
13
14  int main(int argc, char ** argv){
15      char c, opts[]= "M:m:i:h";
16      char help[]= "A program to take powers of a function. Usage:\n"
17                  "\t\tgetopt [options] [a number]\n"
18                  "-h\t This help\n"
19                  "-m\t The minimum exponent at which to start.\n"
20                  "-M\t The maximum exponent at which to finish.\n"
21                  "-i\t Increment by this.\n";
22
23      if (argc==1) {
24          printf(help);
25          return 1;
26      }
27      while ( (c=getopt(argc, argv, opts)) != -1)
28          if (c=='h'){
29              printf(help);
30              return 0;
31          } else if (c=='m'){
32              min = atof(optarg);
33          } else if (c=='M'){
34              max = atof(optarg);
35          } else if (c=='i'){
36              incr = atof(optarg);
37          }
38      if (optind < argc)
39          base = atof(argv[optind]);
40      show_powers();
41  }
```

Listing 6.13 Command-line parsing with `getopt`. Online source: `getopt.c`.

- `optarg` also sets the variable `optind` to indicate the position in `argv` that it last visited. Thus, line 38 was able to check whether there are any non-switch arguments remaining, and line 39 could parse the remaining argument (if any) without `getopt`'s help.

- The program provides human assistance. If the user gives the `-h` switch or leaves off all switches entirely, then the program prints a help message and exits. Every variable that the user could forget to set via the command line has a default value.

Q6.7 Listing 6.1 (page 191) is hard-coded to plot a range from $x = 0$ to $x = 15$. Modify it to use `getopt` to get a minimum and maximum from the user, with zero and fifteen as defaults. Provide help if the user uses the `-h` flag.

Σ

➤ There are many ways to change a program's settings without having to recompile the program.

➤ Environment variables, covered in Appendix A, are a lightweight means of setting variables in the shell that the program can use.

➤ There are many libraries for parsing parameter files, or you could use SQL to pull settings from a database.

➤ The `main` function takes in arguments listed on the command line, and some C functions (like `getopt`) will help you parse those arguments into program settings.

6.4 ※ Syntactic sugar

Returning to C syntax, there are several ways to do almost everything in Chapter 2. For example, you could rewrite the three lines

```
b = (i > j);
a += b;
i++;
```

as the single expression `a+=b=i++>j;`. The seventh element of the array `k` can be called `k[6]`, `*(k+6)`, or—for the truly perverse—`6[k]`. That is, this book overlooks a great deal of C syntax, which is sometimes useful and even graceful, but confuses as easily as it clarifies.

This section goes over a few more details of C syntax which are also not strictly necessary, but have decent odds of being occasionally useful. In fact, they are demonstrated a handful of times in the remainder of the book. If your interest is piqued and you would like to learn more about how C works and about the many alternatives that not mentioned here, see the authoritative and surprisingly readable reference for the language, Kernighan & Ritchie (1988).

The obfuscatory if There is another way to write an `if` statement:

```
if (a < b)
        first_val;
else
        second_val;

/* is equivalent to */

a < b ? first_val : second_val;
```

Both have all three components: first the condition, then the 'what to do if the condition is true' part, and then the 'what to do if the condition is false' part. However, the first is more-or-less legible to anybody who knows basic English, and the second takes the reader a second to parse every time he or she sees it. On the other hand, the second version is much more compact.

The condensed form is primarily useful because you can put it on the right side of an assignment. For example, in the `new_chick` function of Listing 6.3 (p 195), you saw the following snippet:

```
if (gsl_rng_uniform(r) > 0.5)
      out−>type = 'd';
else
      out−>type = 'h';
```

Using the obfuscatory if, these four lines can be reduced to one:

```
out−>type = gsl_rng_uniform(r) > 0.5 ? 'd' : 'h';
```

Macros As well as `#include`-ing files, the preprocessor can also do text substitution, where it simply replaces one set of symbols with another.

Text substitution can do a few things C can't do entirely by itself. For example, you may have encountered the detail of C that all global variables must have constant size (due to how the compiler sets them up).[9] Thus, if you attempt to compile the following program:

[9]Global and static variables are initialized before the program calls `main`, meaning that they have to be allocated without evaluating any non-constant expressions elsewhere in the code. Local variables are allocated as needed during runtime, so they can be based on evaluated expressions as usual.

```
int array_size = 10;
int a[array_size];

int main(){ }
```

you will get an error like `variable-size type declared outside of any function`.

The easy alternative is to simply leave the declaration at the top of the file but move the initialization into `main`, but you can also fix the problem with `#define`. The following program will compile properly, because the preprocessor will substitute the number 10 for `ARRAY_SIZE` before the compiler touches the code:

```
#define ARRAY_SIZE 10
int a[ARRAY_SIZE];

int main(){ }
```

Do not use an equals sign or a semicolon with `#defines`.

You can also `#define` function-type text substitutions. For example, here is the code for the `GSL_MIN` macro from the `<gsl/gsl_math.h>` header file:

```
#define GSL_MIN(a,b) ((a) < (b) ? (a) : (b))
```

It would expand every instance in the code of `GSL_MIN(a,b)` to the if-then-else expression in parens. `GSL_MAX` is similarly defined.

This is a *macro*, which is evaluated differently from a function. A function evaluates its arguments and then calls the function, so `f(2+3)` is guaranteed to evaluate exactly as `f(5)` does. A macro works by substituting one block of text for another, without regard to what that text means. If the macro is

```
#define twice(x) 2*x
```

then `twice(2+3)` expands to `2*2+3`, which is not equal to `twice(5) = 2*5`.

We thus arrive at the first rule of macro writing, which is that everything in the macro definition should be in parentheses, to prevent unforseen interactions between the text to be inserted and the rest of the macro.

Repeated evaluation is another common problem to look out for. For example, `GSL_MIN(a++, b)` expands to `((a++) < (b) ? (a++) : (b))`, meaning that `a` may be incremented twice, not once as it would with a function. Again, the first solution is to not use macros except as a last resort, and the second is to make sure calls to macros are as simple as possible.

The one thing that a macro can do better than a function is take a type as an argument, because the preprocessor just shunts text around without regard to whether that text represents a type, a variable, or whatever else. For example, recall the form for reallocating a pointer to an array of `doubles`:

```
var_array = realloc(var_array, new_length * sizeof(double))
```

This can be rewritten with a macro to create a simpler form:

```
#define REALLOC(ptr, length, type) ptr = realloc((ptr), (length) * sizeof(type))
//which is used like this:
REALLOC(var_array, new_length, double);
```

It gives you one more moving part that could break (and which now needs to be `#included` with every file), but may make the code more readable. This macro also gives yet another demonstration of the importance of parens: without parens, a call like `REALLOC(ptr, 8 + 1, double)` would allocate $8 + \texttt{sizeof}(\texttt{double})$ bytes of memory instead of $9 \cdot \texttt{sizeof}(\texttt{double})$ bytes.

- If you need to debug a macro, the `-E` flag to `gcc` will run only the preprocessor, so you can see what expands to what. You probably want to run `gcc -E` *`onefile.c`* `| less`.
- The custom is to put macro names in capitals. You can rely on this in code you see from others, and are encouraged to stick to this standard when writing your own, as a reminder that macros are relatively fragile and tricky. Apophenia's macros can be written using either all-caps or, if that looks too much like yelling to you, using only an initial capital.

Σ

➤ Short `if` statements can be summarized to one line via the `condition ? true_value : false_value` form.

➤ You can use the preprocessor to `#define` constants and short functions.

6.5 More tools

Since C is so widely used, there is an ecosystem of tools built around helping you easily write good code. Beyond the debugger, here are a few more programs that will make your life as a programmer easier.

Memory debugger

The setup is this: you make a mistake in memory handling early in the program, but it is not fatal, so the program continues along using bad data. Later on in the program, you do something innocuous with your bad data and get a segfault. This is a pain to trace using `gdb`, so there are packages designed to handle just this problem.

If you are using the GNU standard library (which you probably are if you are using `gcc`), then you can use the shell command

```
export MALLOC_CHECK_=2
```

to set the `MALLOC_CHECK_` enviornment variable; see Appendix A for more on environment variables. When it is not set or is set to zero, the library uses the usual `malloc`. When it is set to one, the library uses a version of `malloc` that checks for common errors like double-freeing and off-by-one errors, and reports them on `stderr`. When the variable is set to two, the system halts on the first error, which is exactly what you want when running via `gdb`.

Another common (and entirely portable) alternative is Electric Fence, a library available via your package manager. It also provides a different version of `malloc` that crashes on any mis-allocations and mis-reads. To use it, you would simply recompile the program using the efence library, by either adding `-lefence` to the compilation command or the `LINKFLAGS` line in your makefile (see Appendix A).

Revision control

The idea behind the revision control system (RCS) is that your project lives in a sort of database known as a repository. When you want to work, you check out a copy of the project, and when you are done making changes, you check the project back in to the repository and can delete the copy. The repository makes a note of every change you made, so you can check out a copy of your program as it looked three weeks ago as easily as you could check out a current copy.

This has pleasant psychological benefits. Don't worry about experimenting with your code: it is just a copy, and if you break it you can always check out a fresh copy from the repository. Also, nothing matches the confidence you get from making major changes to the code and finding that the results precisely match the results from last month.

Finally, revision control packages facilitate collaboration with coauthors. If your changes are sufficiently far apart (e.g., you are working on one function and your coauthor on another), then the RCS will merge all changes to a single working copy. If it is unable to work out how to do so, then it will give you a clearly demarcated list of changes for you to accept or reject.

This method also works for any other text files you have in your life, such as papers written in LaTeX, HTML, or any other text-based format. For example, this book is under revision control.

There is no universal standard revision control software, but the Subversion package is readily available via your package manager. For usage, see Subversion's own detailed manual describing set-up and operation from the command line, or ask your search engine for the various GUIs written around Subversion.

THE PROFILER If you feel that your program is running too slowly, then the first step in fixing it is measurement. The *profiler* times how long every function takes to execute, so you know upon which functions to focus your clean-up efforts.

First, you need to add a flag to the compilation to include profiler symbols, `-pg`. Then, execute your program, which will produce a file named `gmon.out` in the directory, with the machine-readable timings that the profiler will use.[10] Unlike the debugger's `-g` option, the `-pg` option may slow down the program significantly as it writes to `gmon.out`, so use `-g` always and `-pg` only when necessary.

Finally, call `gprof ./my_executable` to produce a human-readable table from `gmon.out`.[11] See the manual (`man gprof`) for further details about reading the output.

As with the debugger, once the profiler points out where the most time is being taken by your program, what you need to do to alleviate the bottleneck often becomes very obvious.

If you are just trying to get your programs to run, optimizing for speed may seem far from your mind. But it can nonetheless be an interesting exercise to run a modestly complex program through the profiler because, like the debugger's backtrace, its output provides another useful view of how functions call each other.

[10]If the program is too fast for the profiler, then rename `main` to `internal_main` and write a new `main` function with a `for` loop to call `internal_main` ten thousand times.

[11]`gprof` outputs to `stdout`; use the usual shell tricks to manipulate the output, such as piping output through a pager—`gprof ./my_executable | less`—or dumping it to a text file—`gprof ./my_executable > outfile`—that you can view in your text editor.

Optimization The `gcc` compiler can do a number of things to your code to make it run faster. For example, it may change the order in which lines of code are executed, or if you assign `x = y + z`, it may replace every instance of `x` with `y + z`. To turn on optimization, use the `-O3` flag when compiling with `gcc`. [That's an 'O' as in optimization, not a zero. There are also `-O1` and `-O2`, but as long as you are optimizing, why not go all out?]

The problem with optimization, however, is that it makes debugging difficult. The program jumps around, making stepping through an odd trip, and if every instance of `x` has been replaced with something else, then you can not check its value. It also sometimes happens that you did not do your memory allocation duties quite right, and things went OK without optimization, but suddenly the program crashes when you have optimization on. A memory debugger may provide some clues, but you may just have to re-scour your code to find the problem. Thus, the `-O3` flag is a final step, to be used only after you are reasonably confident that your code is debugged.

$\mathbb{Q}_{6.8}$ Add the `-pg` switch to the makefile in the birds directory and check the timing of the three different versions. It may help to comment out the `printf` function and run the simulation for more periods. How does the `-O3` flag change the timings?