Section Notes 7

CS50 — Fall 2008

Week of November 2, 2008

Contents

1 Enumerated Types

In programming, we constantly deal with finite sets of values for which we have conventional names: the Boolean values `true` and `false`, the days of the week, the parts of speech. We typically represent the elements of these sets as integers: 1 stands for `true`, 0 stands for `SUNDAY`, and so on. But of course, we want to use symbols in our code, not the underlying literal integers. So far we've used `#define` directives to define such symbols, but that's unsatisfying because the compiler deliberately forgets about the relationship between the symbol and its value. That means that the debugger is unable to remind you about the integer value of `SUNDAY`, for example.

An alternative is to use an *enumerated type* to associate a finite set of symbols with integer values. As a first example, here is a complete and correct C program:

```
int
main()
{
    enum { SUCCESS, FAILURE };
    return SUCCESS;
}
```

It introduces an enumerated type that has no name, but it declares no variables of that type. Even so, there are two *constants* of that enumerated type, and they can be used like integer constants. The first one, `SUCCESS`, is associated with the integer 0 and `FAILURE` is associated with 1. (If there were more constant names, they'd be given succeeding values.)

The next snippet declares a single variable, `gender` of an anonymous enumerated type:

```
enum { MALE, FEMALE } gender;
  ...
gender = (is_male(student)) ? MALE : FEMALE;
  ...
if (gender == FEMALE)
   ...
```

More commonly, we define named enumerated types that can be used to declare more than one variable. To do so we add a *tag*, an identifier between the `enum` keyword and the opening brace:

```
enum day
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};

enum day get_next_day(enum day d);
enum day get_today();

int main()
{
    enum day today, tomorrow;

    today = get_today();
    tomorrow = get_next_day(today);

      ...
}
```

Note that the name of the enumerated type is *not* just `day`, it's `enum day`. The tag `day` is not a type name by itself, and it can even be used for other purposes.

Defining your own type names. Suppose that a program were required to keep certain statistics on some basketball teams. Those numbers might be stored in a `team_stats` type. We can use an *enumeration* (an instance of an enumerated type) as an index into `team_stats`. One way to go about this would be:

```
enum bball_stats {WINS, LOSSES, POINTS_PER_GAME};
```

The above line creates a new enumerated type, `enum bball_stats`, whose values are `WINS`, `LOSSES`, and `POINTS_PER_GAME`. Alternatively, instead of using `enum bball_stats` every time we want to declare a variable of this type, we can use a `typedef` to define a synonym for an anonymous enumerated type:

```
typedef enum { WINS, LOSSES, POINTS_PER_GAME } bball_stats;
```

Now we can write `bball_stats` instead of `enum bball_stats` when declaring variables. To take another example, we could write

```
enum suit {HEARTS, CLUBS, DIAMONDS, SPADES};
enum suit mySuit1 = CLUBS;
enum suit mySuit2 = DIAMONDS;
enum suit mySuit3 = HEARTS;
```

but it's slightly more economical to write

```
typedef enum {HEARTS, CLUBS, DIAMONDS, SPADES} suit;
suit mySuit1 = CLUBS;
suit mySuit2 = DIAMONDS;
suit mySuit3 = SPADES;
```

Another example:

```
typedef enum {NOUN, VERB, PREP, CONJ, ADJ, ADV, OTHER} parts;
parts determine_part(string word_to_check);
```

And the code from which we call function `determine_part` might look like:

```
switch (determine_part(input_word))
{
    case NOUN:
        // handle noun
        break;
    case ADV:
        // handle adverb
        break;
    default:
        // handle error
}
```

2 Structures

As we've seen, an array in C is a homogeneous data stucture: every element has the same type. To create objects that combine components of different types, we use `struct` types. For example, to hold information about a student, we might define the following type:

```
struct student
{
    char name[20];
    int year;
    int age;
};
```

The name of this type is `struct student`; the tag `student` can't be used on its own. Also note that there is a semi-colon after the closing curly brace of the `struct` declaration.

A structure of type `struct student` takes up 28 bytes of memory: 20 for the `name` array, and 4 for each of the integers.

Structures are only useful if we can access the data within the structure. The fields of a structure are accessed using the "`.`" (called "dot") operator. Given any variable of some `struct` type, elements inside the structure are accessed using the following syntax:

⟨*variable name*⟩ `.` ⟨*structure field*⟩

For example:

```
struct student stud1;
stud1.name = "Joe";
stud1.year = 1999;
stud1.age = 20;
```

In C, programmers often find it useful to use pointers to structures. The syntax required to dereference the pointer and access the structure field is shown below. Note that the parentheses are mandatory here, because the dot operator has higher precedence than the dereferencing operator.

`(*`⟨*name of pointer to structure*⟩`)` `.` ⟨*structure field*⟩

However, pointers to structures are so prevalent, and the need to dereference/access the pointer so frequent, that C has a special operator for dereferencing the pointer and accessing a structure field in one step. In this case, the `->` (called "arrow") operator is used. The line below thus accomplishes the exact same task as the example directly above, but in a much cleaner manner.

⟨*variable name*⟩ `->` ⟨*structure field*⟩

If the CS 50 staff were to design a type tp keep track of individual students in the course, it might look like this, an extension of the previous `struct` type:

```
struct student
{
    string  first_name;
    string  last_name;
    string  logname;
    int     midterm_exam[2];
    int     assigns[9];
    int     late_days;
    int     final_project;
    double  final_grade;
};
```

Variables for individual students could be declared and used in the following way:

```
    struct student person1, person2, person3;

    // input asst3 grades
    person1.assigns[3] = 53;
    person2.assigns[3] = 42;
    person3.assigns[3] = 49;
```

However, just as with the `enum` types in the previous section, it's often more convenient to provide `typedef` names for structure types. For example,

```
typedef struct student
{
    string  first_name;
    string  last_name;
    string  logname;
    int     midterm_exam[2];
    int     assigns[9];
    int     late_days;
    int     final_project;
    double  final_grade;
}
student;
```

The declaration of variables can then be shortened, as it was with `enum`:

```
    student person1, person2, person3;
```

In the example above, we chose a `typedef` name that's the same as the `struct` tag. While that's allowed in C, it's not required. The tag can be different from the abbreviation, and in fact, the tag can often be omitted.

Structures and pointers. Consider the following definition of the `date` structure, which the structure contains integers and a pointer to another `date` structure.

```
typedef struct date
{
    int year;
    int month;
    int day;
    struct date *tomorrowp;
}
date;
```

Assume that `datep` is a pointer to a variable of type `date`. To access the `year` field of the structure that `datep` points to:

```
datep->year
```

To access the `day` field of the structure that is "tomorrow":

```
datep->tomorrowp->day
```

Unlike arrays, structures can be assigned to each other, passed by value as arguments to functions, and returned as function results. Consider the following example:

```
typedef struct event
{
    string name;
    int year;
    int month;
    int day;
}
event;

event get_event()
{
    event game;
    game.name = GetString();
    game.year = GetInt();
    game.month = GetInt();
    game.day = GetInt();
    return(game);
}

void main()
{
    event evt, evt_copy;
    evt = get_event();
    evt_copy = evt;
}
```

3 Linked Lists

So far, our only data type for representing collections of values has been the array. The most useful feature of arrays is the ease of element lookup—all you need is the element's index. However, insertion and deletion of elements is potentially expensive with arrays! If we want to insert an element in an array anywhere except the end, we have to shift the the rest of the array to the right. With long arrays, this takes a lot of time. Luckily, we have a new data structure that can solve this problem.

Linked lists, in contrast to arrays, make element insertion and deletion very easy. A list is a good choice when the number of objects in your list is not known before you start to solve the problem, and the size of this list may grow and shrink during the task.

A linked list is a collection of structures that each contain some data, as well as a pointer to the next element in the list. These pointers create a chain of elements that we can follow from the beginning to the end, hitting each element in the list. Let's look at a very basic linked list element structure:

```
typedef struct sllist{
    VALUE val;
    struct sllist *next;
} sllist;
```

The name `sllist` stands for *singly-linked* list. We'll see later why putting more than one link pointer into each list element can be useful. As you can see, each `sllist` element has two parts - a value, and a pointer to the next element. Here, `VALUE` is just a placeholder for another type of variable (such as an `int`, `char`, `bool`, etc).

You might think it's more intuitive to use the shorter type `sllist *` when declaring the `next` component above, but that doesn't work. The codetypedef name `sllist` is not in effect when the type of `next` is being set. So instead, the full name `struct sllist` *must* be used when a structure type refers to itself in this way.

You can see why insertion/deletion is so much cleaner here than for arrays. To insert an element after a given element, we just change the `next` pointers to reflect the new ordering. With linked lists, there is no requirement that the elements next to each other in the list are next to each other in memory.

Unfortunately, for all the gains in insertion speed we gain, we lose in lookup speed. Why? In an array, you can use indexing to go directly to the element you want. In a linked list, you must traverse the entire list from the beginning until you arrive at the desired element.

Let's take a look at a simple insertion function:

```
bool
llist_insert_after(sllist *insert_after, VALUE val)
{
    // create a new element
    sllist *new_ele = malloc(sizeof(sllist));
    if(new_ele == NULL)
        return false;

    // initialize the element
    new_ele->val = val;   // no name collision here

    // insert the element
    new_ele->next = insert_after->next;
    insert_after->next = new_ele;

    return true;
}
```

As you can see, the code that does the actual insertion is only two lines. Can you think of how to implement an `llist_insert_before` function? With singly-linked lists, it's quite complicated. For this task, among others, we should consider a doubly-linked list.

Doubly-linked lists. A doubly-linked list is a linked list in which each element contains pointers to both the next *and the previous* elements. Figure 1 shows a small example. The first element of the list has a `NULL` pointer in the `prev` field to indicate that there is no previous element, and the last element has a `NULL` pointer in the `next` field to indicate that there is no next element.

Doubly-linked lists are useful when you need to insert and remove objects from the center of the list, as well as from its ends, or when you need to traverse the list forward as well as backward.

To implement a doubly-linked list, we might define the following data structure:

```
typedef struct dllist
{
    VALUE val;
    struct dllist *prev;
    struct dllist *next;
}
dllist;
```

Armed with a doubly-linked list, consider how you would go about writing the `llist_insert_before` function.

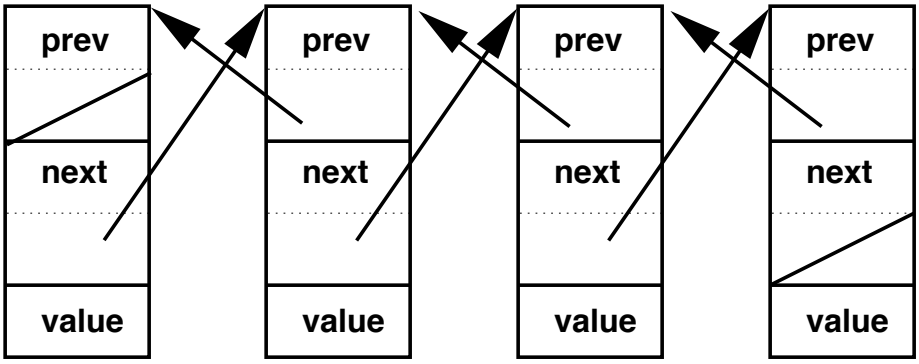

Figure 1: A doubly-linked list

4 File I/O

The ability to read data from and write data to files is the primary means of storing *persistent* data, or data which does not disappear when your program finishes running.

`FILE` pointers. The abstraction of files provided by the standard I/O library (stdio) is implemented in a structure called `FILE`. Almost all of the stdio functions take a pointer to one of these structures as one of their arguments (either explicitly or implicitly). The main exception is `fopen`, which is used to get one of these pointers in the first place.

UNIX gives your programs three default `FILE` pointers, just for showing up:

- `stdin` — standard input. For interactive programs, if you read standard input, you'll get what the user types on the keyboard. You can also put the contents of a file onto `stdin` using the redirect operator "`<`"; similarly, if you pipe ("`|`") another program's output into your program, you'll see it on `stdin`.
- `stdout` — standard output. For interactive programs, if you write to standard output, then whatever you write will appear on the users screen. This too can be piped or redirected using the shell.
- `stderr` — standard error. Like `stdout`, but can be treated separately when appropriate. For example, if you want to capture all of the error messages that your program emits, you can redirect `stdout` to a file and have only the error messages printed to the terminal screen.

Below are descriptions of some of the more common file operation functions. Bear in mind that this is by no means an exhaustive list. For a more complete reference, consult the on-line manual pages or your favorite textbook.

- `fopen` and `fclose`

  Open or close a file. If `fopen` returns `NULL`, then it could not open the file requested.

  UNIX will close any files you've left open when your program terminates, but it is considered poor coding practice to rely on this behavior.

- `fgetc` and `fputc`

  Read or write a single character.

  An important note about `fgetc` (relevant for a few other functions, too): although `fgetc` reads a *`char`*, it returns an `int`, allowing it to return a special error value. If `fgetc` just returned a `char`, then what value could it return to indicate to you that something had gone wrong? Every possible return value (any member of the set of possible `char`s) would correspond to something that might actually be in a file. If the value is 0..255 (for

8–bit `chars`) then you know that `fgetc` succeeded, but if it is outside this range, then something else happened. (This should be noted as an example of a poorly designed function interface.)

For most implementations of `fgetc` you'll see, there's only one error code, which is `EOF`. This is returned when you attempt to read past the end of the file, or some other error occurs.

- `fgets` and `fputs`

  Read or write a line of text. Assumes that the file is text. (Note: `fputs` does not add a newline, so if you want there to be one, you must add it yourself)

  There is another function named `gets` which is similar to `fgets`, but notoriously dangerous to use. (It has no way to limit the size of input, allowing the user to scribble all over memory.) Needless to say, it should never be used.

- `fread` and `fwrite`

  Read or write blocks of data of any size. Useful for reading or writing an entire array or structure.

- `fscanf` and `fprintf`

  Read or write data according to given format. `fprintf` is almost exactly like `printf`, but prints to a file instead of the screen.

  `fscanf` is sort of like `fprintf`, only backwards.

- `fseek` and `ftell`

  Move around in a file, or find out where you are in a file.

  The `rewind` function is a special case of the `fseek` that can be used to rewind to the beginning of a file. (Note that some files can't be rewound — you'd need a time machine to rewind `stdin`, for example!)

- `fflush`

  Flush changes (make them happen immediately). fwrite and other C library output functions wait until you've written a certain amount of data before actually sending it to the file, for better performance. `fflush` ensures that everything got written (as does `fclose`).

- `feof` and `ferror`

  Check whether end of file or some error has been encountered.